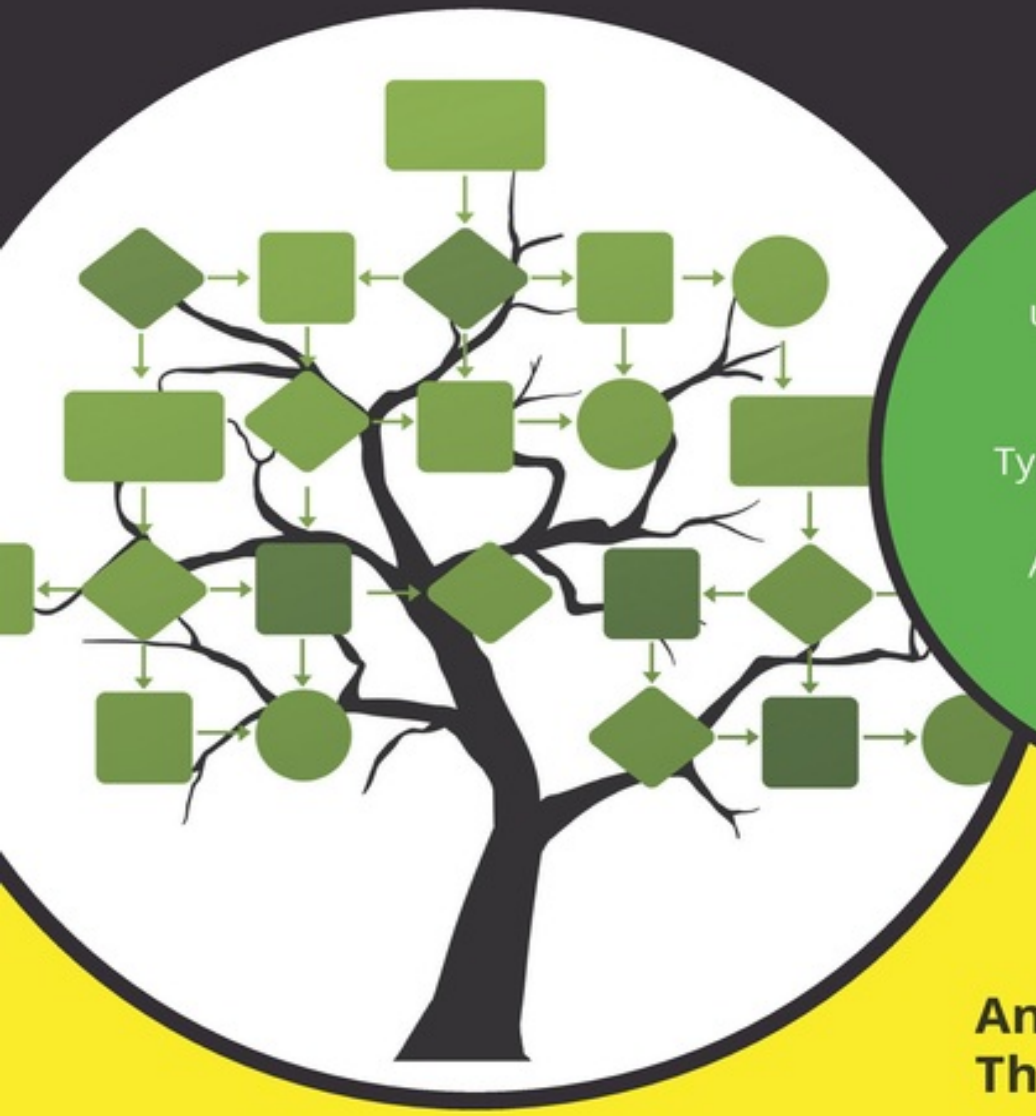
LERNEN EINFACH GEMACHT

Algorithmen und Datenstrukturen

für dummies®



Mit Algorithmen und Datenstrukturen programmieren

Typische Probleme lösen

Algorithmische Ideen als Pseudocode dokumentieren

Andreas Gogol-Döring
Thomas Letschert

Algorithmen und Datenstrukturen für Dummies

Schummelseite

FUNKTIONEN

Name	*Notation*	*Erklärung*
Logarithmus	$\log_b n$	das y, für das $b^y = x$ gilt
natürlicher Logarithmus	$\ln n$	das y, für das $e^y = x$ gilt
binärer Logarithmus	$\lg n$	das y, für das $2^y = x$ gilt
Fakultät	$n!$	$1 \cdot 2 \cdot \ldots \cdot n$
Binomialkoeffizient	$\binom{n}{k}$	$\frac{n!}{k! \cdot (n-k)!}$

LAUFZEITKLASSEN

Name	*Notation*	*Beispiel*
konstant	$\mathcal{O}(1)$	arithmetische Operation, Zugriff auf ein Array
logarithmisch	$\mathcal{O}(\log n)$	binäre Suche, Suche in Rot-Schwarz-Baum
linear-logarithmisch	$\mathcal{O}(n \log n)$	MergeSort, HeapSort
quadratisch	$\mathcal{O}(n^2)$	InsertionSort, alle Paare aufzählen, zwei geschachtelte Schleifen
kubisch	$\mathcal{O}(n^3)$	Tripel aufzählen, drei geschachtelte Schleifen
exponentiell	$\mathcal{O}(2^n)$	Teilmengen, Permutationen aufzählen

LAUFZEIT REKURSIVER ALGORITHMEN

Laufzeit rekursiv	$T(n) \sim$	*Beispiel*
$T(n) = T(n/2) + 1$	$\lg n$	binäre Suche
$T(n) = 2 \cdot T(n/2) + n$	$n \lg n$	MergeSort
$T(n) = T(n-1) + n$	n^2	InsertionSort
$T(n) = 2 \cdot T(n/2) + 1$	n	Baumtraversierung

MASTERTHEOREM

Vereinfachte Variante (»Bachelortheorem«): Bei einem Teilen-und-Herrschen-Algorithmus, bei dem die Eingabe immer möglichst gleichmäßig in zwei Teile aufgeteilt wird, ergibt sich bei polynomiellem Zusatzaufwand der Form $f(n) = c \cdot n^k$ die Gesamtlaufzeit $T(n)$ für den Algorithmus als:

$$T(n) = \begin{cases} \mathcal{O}(n^k) & \text{falls } k > 1 \quad \text{(Fall 1)} \\ \mathcal{O}(n \cdot \log n) & \text{falls } k = 1 \quad \text{(Fall 2)} \\ \mathcal{O}(n) & \text{falls } k < 1 \quad \text{(Fall 3)} \end{cases}$$

KOMBINATORIK

Anzahl der Elemente kombinatorischer Objekte:

Kollektion	*Anzahl der Elemente*
alle Permutationen von n Objekten	$n!$
alle Teilmengen einer n-elementigen Menge	2^n
alle Partitionen einer n-elementigen Menge	B_n

mit $B_{n+1} = \sum_{K=0}^{n} \binom{n}{k} \cdot B_k$

Anzahl der Möglichkeiten, k Elemente aus einer Kollektion von n Elementen auszuwählen:

	Anordnung relevant	*Anordnung nicht relevant*
Wiederholung erlaubt	n^k	$C_{n-1}^{k+n-1} = \binom{k+n-1}{n-1}$
Wiederholung nicht erlaubt	$P_k^n = \frac{n!}{(n-k)!}$	$C_k^n = \binom{n}{k}$

$$P_k^n = \frac{n!}{(n-k)!} = n \cdot (n-1) \cdot \ldots \cdot (n-k+1)$$

$$C_k^n = \binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} = \frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}{k \cdot (k-1) \cdot \ldots \cdot 1}$$



Andreas Gogol-Döring
Thomas Letschert

Algorithmen und Datenstrukturen

für dummies®

Fachkorrektur von Christopher Schölzel

WILEY

WILEY-VCH Verlag GmbH & Co. KGaA

Algorithmen und Datenstrukturen für Dummies

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

© 2020 WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim

Wiley, the Wiley logo, Für Dummies, the Dummies Man logo, and related trademarks and trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries. Used by permission.

Wiley, die Bezeichnung »Für Dummies«, das Dummies-Mann-Logo und darauf bezogene Gestaltungen sind Marken oder eingetragene Marken von John Wiley & Sons, Inc., USA, Deutschland und in anderen Ländern.

Das vorliegende Werk wurde sorgfältig erarbeitet. Dennoch übernehmen Autoren und Verlag für die Richtigkeit von Angaben, Hinweisen und Ratschlägen sowie eventuelle Druckfehler keine Haftung.

Coverfoto: © Mark Stay – stock.adobe.com
Korrektur: Matthias Delbrück, Dossenheim/Bergstraße

Print ISBN: 978-3-527-71432-2
ePub ISBN: 978-3-527-81202-8

Inhaltsverzeichnis

- Cover
- Einleitung
  - Über dieses Buch
  - Törichte Annahmen über den Leser
  - Wie dieses Buch aufgebaut ist
  - Symbole, die in diesem Buch verwendet werden
  - Wie es weitergeht
- Teil I: Grundbegriffe
  - Kapitel 1: Algorithmen
    - Das sind Algorithmen
    - Algorithmen klar beschreiben
    - Algorithmen sollten korrekt sein
  - Kapitel 2: Qualität von Algorithmen
    - So gut sind Algorithmen
    - Laufzeitanalysen
    - Laufzeiten konkret bestimmen
    - Randomisierte Laufzeitanalyse
    - Amortisierte Laufzeitanalyse
  - Kapitel 3: Daten und ihre Struktur
    - Aus Daten Strukturen bauen
    - Funktion folgt Struktur
- Teil II: Algorithmen in den Gärten der Strukturen
  - Kapitel 4: Listen: Immer einer nach dem anderen
    - Listen: Datenmodell und Implementierung
    - Alles ist ewig und Rekursion ist gut
    - Nicht alles ist ewig und Form ist nicht Inhalt
  - Kapitel 5: Bäume: Immer einer über dem anderen
    - Wo ist die Kokosnuss?
    - Die Kleinen nach links, die Großen nach rechts
    - Bäume verkleiden sich gerne mal
    - Bäume als Datenmodell
  - Kapitel 6: Graphen: Jeder mit jedem

Im Irrgarten der sozialen Netzwerke
Datenstrukturen für Graphen
Erst die Schuhe, dann das Hemd – oder wie?
Teil III: Probleme und ihre Lösungen
Kapitel 7: Sortieren
Alles in Ordnung
MergeSort: Geteiltes Leid ist halb sortiert
QuickSort: Quick and Easy
HeapSort: Ein Haufen Arbeit
Die maximale Sortiergeschwindigkeit
Sortieren in Linearzeit
Weitere Sortieralgorithmen
Kapitel 8: Rucksack packen
Wie man einen Koffer packt
Lineare Optimierung
Kapitel 9: Mengen und ihre Speicherung
Ich bin eine Menge
Gut gehackt ist schnell gefunden
Viel ist oft eine Menge
Kapitel 10: Verbindungen finden
Kürzeste Pfade
Verbundenes aufspüren
Spann mir einen Graphen auf
Teil IV: Algorithmische Techniken
Kapitel 11: Probleme totschlagen
Erschöpfende Suche
Iterative und rekursive Erzeugung des Suchraums
Backtracking
Verzweigen und Begrenzen
Komplexitätsklassen: Schwere Probleme führen zu anstrengender Arbeit
Wie man mit schweren Problemen umgeht
Kapitel 12: Teilen und Herrschen
Aufgaben auf Mitarbeiter abwälzen
Laufzeiten bei Teilen und Herrschen
Binärsuche
Weitere Beispiele für Teilen und Herrschen

Kapitel 13: Dynamisches Programmieren

Ein profitabler Bauauftrag

Probleme wachsen lassen

Vergleich von Texten

Der Weg zum Optimum

Kapitel 14: Näherungslösungen

Heuristiken

Approximation

Gier

Teil V: Der Top-Ten-Teil

Kapitel 15: Zehn Datenabstraktionen und Datenstrukturen

Stapel

Warteschlange

Prioritätswarteschlange

Liste

Array

Menge

Verzeichnis

Relation

Graph

Baum

Kapitel 16: Zehn Ratschläge, wenn (bevor) der kleine Frust kommt

Rekursion ist deine Freundin

Mathematik ist einfach

Pseudocode ist verstehbar

Abstraktion ist gut

Sei auch mal funktional

Ein Bild sagt mehr als 1000 Worte

Vieles ist solides Handwerk

Es geht auch um Kreativität

Unterscheide Datenmodell und Datenstruktur

Was schwierig aussieht, ist oft auch schwierig

Stichwortverzeichnis

End User License Agreement

Tabellenverzeichnis

Kapitel 2

Tabelle 2.1: Landau-Symbole

Kapitel 7

Tabelle 7.1: Verschiedene Sortierverfahren und ihre Eigenschaften

Kapitel 8

Tabelle 8.1: Einige Anwendungen des Rucksackproblems

Kapitel 11

Tabelle 11.1: NP-schwere Probleme in diesem Buch

Kapitel 13

Tabelle 13.1: Verschiedene Optimierungsprobleme, die mit dynamischem Programmieren gelöst wer...

Illustrationsverzeichnis

Kapitel 1

Abbildung 1.1: Dieses Backrezept erfüllt nicht die strengen Anforderungen...
Abbildung 1.2: Die gaußsche Summenformel
Abbildung 1.3: Turingmaschine und RAM sind zwei Maschinenmodelle.
Abbildung 1.4: Algorithmen und Programme in Modellwelten
Abbildung 1.5: Ein Algorithmus und drei ausführbare Programme, die ihn im...
Abbildung 1.6: Der Startpunkt n = 25 für den Pfad durch das Collatz-Gebir...
Abbildung 1.7: Die zwei Fälle der ägyptischen Multiplikation

Kapitel 2

Abbildung 2.1: Algorithmen können unterschiedlich schnell sein.
Abbildung 2.2: Bauarbeiter vergleichen ihr Gewicht mit einer Waage.
Abbildung 2.3: Laufzeiten der beiden Algorithmen zur Bestimmung des leichtesten Bau...
Abbildung 2.4: langsamer (oben) und schneller (unten) als linear wachsende Laufzeit...
Abbildung 2.5: Gängige Laufzeitklassen
Abbildung 2.6: Eine, zwei und drei geschachtelte Schleifen
Abbildung 2.7: $\mathcal{O}(n^3)$ dominiert die Summe.
Abbildung 2.8: Je nach Quartil ist die Wahl von x unterschiedlich günstig...
Abbildung 2.9: Die erste Woche des Binärzählers
Abbildung 2.10: Durchschnittlich werden $\leq$ 2 Ziffern übermalt.
Abbildung 2.11: Stapel und Warteschlangen
Abbildung 2.12: Wachsen und Schrumpfen der Speicherkapazität eines sparsamen Stapel...
Abbildung 2.13: Die amortisierten Kosten A_i sind in diesem Beispiel ≤ 2.

Abbildung 2.14: Dreieckige Potenzialfunktion beim sparsamen Stapel

Kapitel 3

Abbildung 3.1: Die Basis aller Datenstrukturen
Abbildung 3.2: Datenstrukturen bestehen aus Nutz- und Verwaltungsinformat...
Abbildung 3.3: (Teilweise) zusammenhängende Speicherung
Abbildung 3.4: Völlig verkettete Speicherung
Abbildung 3.5: Case-Konstrukt: eine Musterkennung in Strukturen
Abbildung 3.6: Zeiger, Variablen und Werte
Abbildung 3.7: Ablauf von Algorithmus FList(5)
Abbildung 3.8: Liste als verkettete Zellen
Abbildung 3.9: Cons-Implementierung: Erzeuge eine Speicherzelle mit Wert ...
Abbildung 3.10: Das Wesentliche: die hierarchische Struktur einer Liste
Abbildung 3.11: GGT: iterative und rekursive Version
Abbildung 3.12: Endrekursion und lineare Rekursion
Abbildung 3.13: Strukturelle Rekursion: Funktion folgt Form.
Abbildung 3.14: Iterator

Kapitel 4

Abbildung 4.1: Ordnung, Anordnung und Listen
Abbildung 4.2: Kasperle (Algorithmus) und Datenmodell (Liste)
Abbildung 4.3: Imperative Datenstruktur für eine Warteschlange
Abbildung 4.4: Neues Element zur Warteschlange
Abbildung 4.5: Einfügen in eine verkettete Liste: imperativ
Abbildung 4.6: Einfügen in eine verkettete Liste mit Referenzen
Abbildung 4.7: Alle Permutationen rekursiv erzeugen: Beispiel
Abbildung 4.8: Map: Eine Funktion auf alle Elemente einer Liste anwenden
Abbildung 4.9: Einfügehelfer InsH: Ein Element auf alle Arten in eine Lis...
Abbildung 4.10: Persistenz: Was ein Wert ist, das vergeht nicht.
Abbildung 4.11: Imperatives und funktionales Denken
Abbildung 4.12: Zwei Listen als eine Liste mit zwei Eingängen
Abbildung 4.13: Funktionales Anfügen erfordert Kopieren
Abbildung 4.14: Warteschlangen funktional
Abbildung 4.15: Statt Zeiger auf Anfang und Ende: Eingangs- und Ausgangs...
Abbildung 4.16: (Zeiger-)Variablen innerhalb von Datenstrukturen: Alles ...
Abbildung 4.17: Append

Kapitel 5

Abbildung 5.1: Ein Binärbaum vom Typ BinTree
Abbildung 5.2: Ein Binärbaum in vereinfachter Darstellung
Abbildung 5.3: Traversierungen, globale Strategie

Abbildung 5.4: Traversierungen: lokale Strategie
Abbildung 5.5: Tiefentraversierungen eines Binärbaums
Abbildung 5.6: Inorder-Traversierungen erzeugen eine Projektion des Baumes.
Abbildung 5.7: Inorder-Traversierung mit Stapel aufgeschobener Arbeit
Abbildung 5.8: Interne Traversierung
Abbildung 5.9: Externe Traversierung mit einem Iterator
Abbildung 5.10: Breitentraversierung: Traversierung gegen die Natur des Baumes
Abbildung 5.11: Breitentraversierung mit Warteschlange
Abbildung 5.12: Zwerge anordnen: Die Großen nach rechts, die kleinen nach links
Abbildung 5.13: Suchbaum: Kompromiss zwischen Ordnung und Chaos
Abbildung 5.14: Suchbaum: Einfügen von Schlüssel 5 in einen Baum mit Wurzel 7
Abbildung 5.15: Ein Trie
Abbildung 5.16: Trie als persistente Datenstruktur
Abbildung 5.17: Trie als Datenstruktur für persistente Listen
Abbildung 5.18: Eine Prioritätswarteschlange als Heap-Baum gespeichert
Abbildung 5.19: Aufbau eins Heaps
Abbildung 5.20: Binärbaum in einem Array
Abbildung 5.21: Die Welt ist voller Bäume.
Abbildung 5.22: Ein Rechenbaum: ein arithmetischer Ausdruck als Baum

Kapitel 6

Abbildung 6.1: Drei verschiedene Arten von Graphen
Abbildung 6.2: Hildegard die Durstige mit Blutorangen-Karotten-Schorle
Abbildung 6.3: Einige Vampire und ihre *Follower*
Abbildung 6.4: Ein gerichteter Graph
Abbildung 6.5: Ein ungerichteter Graph
Abbildung 6.6: Ein gerichteter Graph mit Markierungen
Abbildung 6.7: Wege im Labyrinth als Graph
Abbildung 6.8: Tiefentraversierung in einem Graphen
Abbildung 6.9: Ein Graph als nachgebaute Datenstruktur (meistens Unsinn)
Abbildung 6.10: Ein Graph als Adjazenzen
Abbildung 6.11: Ein Graph und seine Adjazenzlisten-Darstellung
Abbildung 6.12: Ein Graph in einer Darstellung als Adjazenzmatrix
Abbildung 6.13: Vieles kann als Graph modelliert werden.
Abbildung 6.14: Abhängigkeitsgraph: Das Datenmodell der topologischen So...
Abbildung 6.15: Der gleiche Graph, wenn die Knoten in der Reihenfolge ih...
Abbildung 6.16: Datenmodell, Datenstruktur, abstrakter und konkreter Alg...

Kapitel 7

Abbildung 7.1: Bauarbeiter, nach Gewicht sortiert

Abbildung 7.2: In-place-Sortieren des Arrays a zwischen den Positionen l ...
Abbildung 7.3: So funktioniert SelectionSort.
Abbildung 7.4: Ein Beispiel für SelectionSort
Abbildung 7.5: Die Laufzeit von SelectionSort ist $\mathcal{O}(n^2)$.
Abbildung 7.6: Sortieren mit Teilen und Herrschen
Abbildung 7.7: Merge verwendet zwei »Finger« i_l und i_r.
Abbildung 7.8: Die Laufzeit von MergeSort
Abbildung 7.9: Beispiel für einen Durchlauf von »Partition«
Abbildung 7.10: Der Heap ist ein Baum, der in einem Array gespeichert wi...
Abbildung 7.11: Der Heap von Abbildung 7.10 nach Einfügen einer 5 und an...
Abbildung 7.12: Entscheidungsbäume eines Sor...
Abbildung 7.13: Maximale Blätterzahl bei Bäumen unterschiedlicher Höhe k
Abbildung 7.14: SortByDigit
Abbildung 7.15: RadixSort

Kapitel 8

Abbildung 8.1: Wenn man die Hantel einpackt, ist für das Bier kein Platz ...
Abbildung 8.2: Sobald das Goldkörnchen eingepackt ist, passt das Silber n...
Abbildung 8.3: Eine Abzweigung im Entscheidungsbaum
Abbildung 8.4: Ein Entscheidungsbaum
Abbildung 8.5: Tiefensuche durch den Entscheidungsbaum mit KnapsackRek
Abbildung 8.6: Tiefensuche durch den Entscheidungsbaum mit KnapsackBnB
Abbildung 8.7: Die Tabelle **Tab** (links) und wie sie berechnet wird (rechts...
Abbildung 8.8: Preissenkung führt zur Laufzeitsenkung
Abbildung 8.9: Zwei Holzspielzeuge und wie viel Gewinn man jeweils damit ...
Abbildung 8.10: Grenzen der Produktionszahlen in Abhängigkeit von den be...
Abbildung 8.11: Das Optimum liegt am Rand des Lösungsraums.

Kapitel 9

Abbildung 9.1: Mengenlehre – die Wissenschaft von geometrischen Figuren i...
Abbildung 9.2: Wenn derselbe nicht mehr der Gleiche ist, bleibt er in der...
Abbildung 9.3: Das Konzept der Datenabstraktion
Abbildung 9.4: Menge als Datenabstraktion
Abbildung 9.5: Menge als imperative Datenabstraktion mit Array-Repräsenta...
Abbildung 9.6: Zwei Datenstrukturen als die gleiche Menge maskiert
Abbildung 9.7: Eine Menge als charakteristischer Vektor
Abbildung 9.8: Suchen wird mit aufwendigem Prüfen langsam.
Abbildung 9.9: Speichern unter einem (hier optischen) Hashwert
Abbildung 9.10: Speichern unter einem Hashwert
Abbildung 9.11: Hashfunktion

Abbildung 9.12: Kollision: Zwei in einen Eimer
Abbildung 9.13: Massenkollision: Zu viele im gleichen Eimer
Abbildung 9.14: Universelle Hashfunktion mit zufällig eingestellten Ausw...
Abbildung 9.15: Egon: Derselbe, aber nicht mehr gleich
Abbildung 9.16: Optimale Datenstruktur für schnelle Suche (links) und sc...
Abbildung 9.17: Amortisiert es sich, wenn ich hier einmal im Jahr richti...
Abbildung 9.18: Die Baumhöhe ist die maximale Knotentiefe
Abbildung 9.19: Höhenbalanciertheit von Bäumen
Abbildung 9.20: Rotation in Suchbäumen
Abbildung 9.21: Linksrotation eines binären Suchbaums
Abbildung 9.22: Ein Rot-Schwarz-Baum
Abbildung 9.23: Oben: Ein Rot-Schwarz-Baum u...
Abbildung 9.24: Rot und Schwarz, aber nicht Rot-Schwarz
Abbildung 9.25: Ein Rot-Schwarz-Baum gerät aus dem Gleichgewicht.
Abbildung 9.26: Zwei Rotationen …
Abbildung 9.27: … und einmal Umfärben bringen ihn wieder ins Gleichgewic...
Abbildung 9.28: Transformationen von Rot-Rot-Konstellationen (!! unten F...

Kapitel 10

Abbildung 10.1: Ein ungerichteter, gewichteter Graph
Abbildung 10.2: Kürzeste Wege: Extremfälle
Abbildung 10.3: Kürzeste Wege: Erweiterung einer Teillösung
Abbildung 10.4: Kürzeste Weg: Alternativen bei der Erweiterung des Bekannten
Abbildung 10.5: Kürzeste Weg: Von innen nach außen
Abbildung 10.6: Lokale Plausibilität des Algorithmus
Abbildung 10.7: Globale Plausibilität des Algorithmus
Abbildung 10.8: Gibt es eine geheime Abkürzung zur nächstgelegenen Grenzstation?
Abbildung 10.9: Nicht alle Pfadlängen müssen neu berechnet werden.
Abbildung 10.10: T mit allen Grenzstationen
Abbildung 10.11: Datenstrukturen für Dijkstras Algorithmus
Abbildung 10.12: Verbundene und nicht verbundene Graphen
Abbildung 10.13: Aus der Ameisenperspektive verbundene Komponenten finden
Abbildung 10.14: Ameisen bilden (mit etwas globaler Hilfe) verbundene Komponenten.
Abbildung 10.15: Manche Kanten führen nicht in andere Regionen.
Abbildung 10.16: Partitionen: Aufteilungen einer Menge
Abbildung 10.17: Zwei Regionen vereinigen
Abbildung 10.18: Nach der Vereinigung einiger Regionen
Abbildung 10.19: Kleinere Bäume unter größere hängen
Abbildung 10.20: Die verbundenen Komponenten sind als Bäume identifiziert.

Abbildung 10.21: Bäume invers in einem Array
Abbildung 10.22: Menge von Mengen als Menge von Bäumen als Tabelle
Abbildung 10.23: Spannbaum: Zykelfreier Teilgraph mit allen Knoten
Abbildung 10.24: Ein Graph mit eindeutigem minimalem Spannbaum
Abbildung 10.25: Einem wurzel- und richtungslosen Baum Orientierung geben
Abbildung 10.26: Spannbaum als Beifang beim Fischen nach verbundenen Komponenten

Kapitel 11

Abbildung 11.1: Suchraum: so klein und einfach zu beschreiben wie möglic...
Abbildung 11.2: Acht friedliche Damen, die sich nicht gegenseitig bedroh...
Abbildung 11.3: Suchraum für eine Auswahl von 2 von 16?möglichen Positio...
Abbildung 11.4: Suchraum der Größe $\binom{n}{k}$ beim 2-Dame-Problem
Abbildung 11.5: Suchraum der Größe n^n beim 2-Dame-Problem
Abbildung 11.6: Suchraum als Pfade in einem Baum
Abbildung 11.7: Suchraum als Pfade in einem nicht perfekten Baum
Abbildung 11.8: Suchraum als Tupelmenge
Abbildung 11.9: Suchraum als Blätter des Zustandsbaums
Abbildung 11.10: Graphenmodell eines Irrgartens
Abbildung 11.11: Baumartiger Aufbau des Suchraums der Wege (Anfang)
Abbildung 11.12: Backtracking mit Verzweigen und Begrenzen
Abbildung 11.13: Bewertung von Problemen
Abbildung 11.14: Status der Bewertung von Problemen
Abbildung 11.15: Man braucht mindestens vier Kameras, um diesen Irrgart...
Abbildung 11.16: Ein Orakel wählt die richtigen Standorte für die Kamer...
Abbildung 11.17: Wenn P $=$ NP gilt, sind viel mehr Probleme gutartig.
Abbildung 11.18: Lösung des Knotenüberdecku...

Kapitel 12

Abbildung 12.1: Hierarchie im Einwohnermeldeamt von Bürokrazien
Abbildung 12.2: Die drei Fälle des »Bachelortheorems« für quadratischen, linearen u...
Abbildung 12.3: Suche nach dem Wert 5 in einem Suchbaum und in einem sortierten Arr...
Abbildung 12.4: Linker und rechter Rand aller Vorkommen des Wertes 5 im Array
Abbildung 12.5: Die Suche nach dem linken Rand des Trefferbereichs
Abbildung 12.6: Matrixmultiplikation mit Teilen und Herrschen
Abbildung 12.7: Minimaler Punktabstand mit Teilen und Herrschen

Kapitel 13

Abbildung 13.1: Verlauf der Bahnstrecke und der Profit pro Bauabschnitt ...
Abbildung 13.2: Maximale Teilsumme und maximale Endsumme
Abbildung 13.3: MaxEndSum wird $\mathcal{O}(n^2)$-mal aufgerufen.

Abbildung 13.4: Memoisation verhindert doppelte Arbeit.
Abbildung 13.5: Beispielablauf für Kadanes Algorithmus
Abbildung 13.6: Minimum mit dynamischem Programmieren
Abbildung 13.7: Fibonaccis Kaninchen vermehren sich wie die Karnickel.
Abbildung 13.8: Berechnung der Fibonacci-Zahlen mit dynamischem Programm...
Abbildung 13.9: Die Editierdistanz entspricht nicht unbedingt den histor...
Abbildung 13.10: Drei unterschiedliche Alignments von »LIEBEN« und »FLI...
Abbildung 13.11: Drei Möglichkeiten, wie die letzte Spalte eines Alignm...
Abbildung 13.12: Das Schema (rechts), nachdem die Tabelle **Tab** (links) a...
Abbildung 13.13: Die Tabelle als Graph dargestellt, durch den der Backt...
Abbildung 13.14: »Vertikal« ist die beste Option

Kapitel 14

Abbildung 14.1: Interpolationssuche: Der Wert x wird als Nächstes an Pos...
Abbildung 14.2: Kürzester Weg aus einem Labyrinth: Die Länge eines berei...
Abbildung 14.3: Beim A*-Algorithmus fließt das Wasser eher in Richtung Z...
Abbildung 14.4: Dreiecksungleichung für die ...
Abbildung 14.5: Zwei unterschiedliche lange Rundreisen zu 12 deutschen S...
Abbildung 14.6: Näherungsweise Lösung des TSP mit einem Greedy-Algorithm...
Abbildung 14.7: Schrittweise Verbesserung einer Rundreise durch lokale S...
Abbildung 14.8: Approximation der Kreiszahl π mithilfe von Polygonen
Abbildung 14.9: Approximation eines TSP mit einem minimalen Spannbaum
Abbildung 14.10: Wenn Egon gierig in Richtung Ziel läuft, landet er in ...
Abbildung 14.11: Das Wechselgeldproblem für $m = 146$
Abbildung 14.12: Mindestens fünf Münzen für das Wechselgeld von 1,46 €
Abbildung 14.13: Gierig Münzen sammeln, bis 1,46 € erreicht ist
Abbildung 14.14: Huffman-Codierung
Abbildung 14.15: Starte mit einem Wald voll winziger Bäume.
Abbildung 14.16: Vereinigung von B und E
Abbildung 14.17: Vereinigung der nächsten Bäume mit der geringsten Wahr...

Einleitung

IT-Technologien bestimmen immer weitere Bereiche unseres Lebens. Die Entwicklung schreitet immer schneller voran. Was gestern aktuell war, ist heute oft schon völlig veraltet. Gibt es da nichts von Substanz und Dauer? Was dem Außenstehenden als amorphe Lawine von immer neuen Tricks und technischem Schnickschnack vorkommt, hat aber tatsächlich schon einen soliden wissenschaftlichen Kern: die Informatik.

Algorithmen und Datenstrukturen sind das Herz der Informatik, der ewig junge Klassiker mit seinen unvergänglichen Wahrheiten und Strategien. Sie helfen immer wieder, aus einer Geschäftsidee ein funktionierendes Produkt zu machen. Hier gibt es nichts, das in ein paar Jahren schon veraltet oder irrelevant ist. Hier können Sie Wissen, Können und Kreativität kombinieren und die Basis für eine erfolgreiche IT-Karriere legen, eine Karriere, die oft genug und nicht ohne Grund mit einem Vorstellungsgespräch beginnt, das Algorithmen und Datenstrukturen zum Thema hat.

Algorithmen und Datenstrukturen sind nicht nur nützlich. Je länger Sie sich mit ihnen beschäftigen, umso mehr können Sie entdecken und umso mehr Spaß haben Sie mit ihnen. Egal ob spezialisierter Wissenschaftler, mit beiden Beinen in der Praxis stehender Software-Entwickler oder Hobby-Programmierer, jeder kann hier eine spannende Ecke finden und dort Algorithmen und Datenstrukturen auf seine Art kultivieren. Was will man mehr. Das Nützliche ist auch noch schön.

Schönheit ist allerdings manchmal etwas spröde und nicht immer sieht man sie auf den ersten Blick. Das gilt auch für Algorithmen und Datenstrukturen. Sie haben ihre spezielle Notation, den Pseudocode, ihre spezielle Sprache und Terminologie, die man kennen und akzeptieren muss.

Der Einstieg gleicht oft einem Sprung in sehr tiefes Wasser. Zur Vorbereitung könnten Sie Schwimmübungen an Land machen. Das ist nicht das Ziel in diesem Buch. Es soll schon ins Wasser gehen, abstrakte Themen und spröder Pseudocode werden nicht vermieden. Im Gegenteil, wir machen auf Dinge aufmerksam und diskutieren ausführlich Themen, die in der akademischen Literatur gerne mal ganz knapp abgehandelt werden und welche die explizite Einsteigerliteratur eher vermeidet. Sie werden sehen, so spröde ist das alles gar nicht.

Über dieses Buch

Fünf plus drei ergibt acht. Das nennt man Rechnen. Es ist sehr allgemein: Fünf plus drei Sofas, Elefanten, Kaninchen ergibt acht Sofas, Elefanten, Kaninchen. Das macht das Rechnen so nützlich. Viele Aufgaben können in Rechenaufgaben umgewandelt werden und als Rechenaufgaben gelöst werden. Man modelliert das Problem in Form von Zahlen

und löst es dann in der Zahlenwelt. Das ist so weit verbreitet und allgemein üblich, dass eine stilisierte Kurznotation für solche Aufgaben entwickelt wurde und schon in der Grundschule gelehrt wird: $5 + 3$.

Das gleiche Prinzip des Modellierens des Realen in abstrakten Datenwelten, um es dort mit bewertbaren Verfahren lösen zu können, ist das Grundthema von Algorithmen und Datenstrukturen. Es gibt eine Flut von Erkenntnissen, die in einer beeindruckenden akademischen Tradition aufbereitet wurden und dargelegt werden. Nichts davon ist veraltet, aber wir wollen es gelegentlich mal von einem anderen Blickwinkel aus betrachten und laden Sie ein, uns bei diesem Wechsel der Perspektive zu begleiten.

Informatik war einmal ein recht esoterisches Gebiet. Wer sich vor etlichen Jahren akademisch mit Algorithmen und Datenstrukturen beschäftigte, hatte in der Regel einen soliden mathematischen Hintergrund und war gewohnt, Dinge in der Sprache der Mathematik zu modellieren. Kompetenzen in der Software-Entwicklung waren dagegen weniger weit verbreitet. Einfache Programme, in denen die Daten in Arrays hin- und hergeschoben wurden, das war das Übliche.

Heute ist die Situation eher umgekehrt. Statt mathematischer gehören heute eher softwaretechnische Abstraktionen, wie die Objektorientierung, zum weitverbreiteten Wissens- und Kenntnisstand. Die Programmiersprachen haben sich weiterentwickelt und erlauben es, Programme in einer Notation zu schreiben, die vor Jahren als reiner Pseudocode angesehen worden wäre. Konzepte der funktionalen Programmierung sind von fortgeschrittenen Spezialveranstaltungen ins Grundstudium gewandert.

Mit diesem Buch wollen wir Ihnen eine Tür zu der klassischen Thematik der Algorithmen und Datenstrukturen öffnen. Dinge, die nicht so selbstverständlich sind, wie manche Spezialisten denken, werden ausführlich behandelt: Was ist und warum verwendet man Pseudocode, was ist der Unterschied zwischen einem Datenmodell und einer Datenstruktur und so weiter.

Persistente Datenstrukturen und die Beschreibung von Datenstrukturen als algebraische Datentypen sind nicht wirklich neu, aber heute sind sie keine Spezialthemen mehr und möglicherweise sind sie Ihnen, lieber Leser, auch schon begegnet. Glücklicherweise steckt ein einfaches und klares Konzept dahinter und Sie werden hier sehen, dass es sich gut zur Beschreibung von Datenstrukturen einsetzen lässt.

Und dann sind da noch die Kaninchen, Bauarbeiter, Vampire und so weiter. Speziell die kleinen Hobbler haben unser Verständnis der Algorithmen und Datenstrukturen ganz wesentlich gefördert und geprägt. Wir möchten sie Ihnen nicht vorenthalten.

Selbstverständlich wurden alle Bestimmungen des Tierschutzgesetzes beachtet. Bei der Arbeit an diesem Buch ist kein Tier zu Schaden gekommen und alle Mitwirkenden haben ihren Beitrag freiwillig und in artgerechter Weise geleistet.



Törichte Annahmen über den Leser

Wir nehmen an, dass Sie, lieber Leser, jemand sind, der sich für Algorithmen und Datenstrukturen interessiert und tiefer in diese Materie eindringen möchte. Die Behandlung der Thematik in der gängigen Spezialliteratur ist Ihnen aber gelegentlich zu akademisch, zu trocken und zu weit entfernt von ihren sonstigen Erfahrungen.

Sie haben erste Erfahrungen in der Software-Entwicklung und brauchen keine weitere Einführung in die Programmierung oder jemanden, der Ihnen eine bestimmte Programmiersprache als das Maß aller Dinge erklärt. Sie möchten sich die akademische Behandlung von Algorithmen und Datenstrukturen erschließen und suchen einen Einstieg

in diese spannende Materie, der Ihre Erfahrungen respektiert und sie mit der akademischen Behandlung des Stoffs verbindet.

Vielleicht haben Sie auch als Studierende das Fach *Algorithmen und Datenstrukturen* belegt. Die Klausur rückt näher, der Stoff ist widerspenstig. Die üblichen Erklärungen scheinen sich in erster Linie an die bereits Eingeweihten zu richten. Sie suchen nach einem Zugang zu diesem schwierigen Thema, der wirklich von außen nach innen führt.

Wie dieses Buch aufgebaut ist

Algorithmen und Datenstrukturen für Dummies ist in vier Teile gegliedert und jeder Teil hat seinen Schwerpunkt. Es ist nicht notwendig, das Buch systematisch von vorne nach hinten durchzulesen. Wir ermuntern dazu, nach der Lektüre von Teil I eigene Schwerpunkte zu setzen und auch mal hin- und herzuspringen. Wie bei allen etwas komplexeren Wissensgebieten muss man auch hier die einzelnen Puzzleteile mal von dieser mal von jener Seite betrachten, bis dann schließlich alles an seinen Platz fällt und das große Ganze sichtbar wird.

Der Stoff kann und wird üblicherweise insgesamt entweder nach Problembereichen (Sortieren, Traversieren etc.) oder nach algorithmischen Techniken (Teilen und Herrschen, Gier etc.) gegliedert. Zwischen beiden Themenbereichen gibt es viele Querbezüge, sodass sich das Gesamtbild erst nach und nach erschließen lässt. Wir lösen dies, indem wir die Thematik in Teil III von der problemorientierten Seite und in Teil IV von der Seite der algorithmischen Techniken her betrachten. In Teil I werden dazu die Grundlagen gelegt. Teil II vertieft und erweitert diese Grundlagen mit den ersten Beispielen.

Neben den beiden üblichen Themenbereichen *algorithmische Techniken* und *Problemstellungen* haben wir zwei weitere Querschnittthemen: *Datenmodell und Datenstruktur* und *imperative und funktionale Algorithmen*, die sich durch viele Kapitel ziehen. Sie werden in der Literatur meist knapper behandelt, als es ihrer Bedeutung entspricht.

Teil I Grundbegriffe

In Teil I geht es um die Grundlagen: Algorithmen und Datenstrukturen. Was ist das und wie beschreibt und bewertet man es? Was ist ein Algorithmus, wie beschreibt man ihn in Pseudocode und wie wird seine Effizienz bewertet? Was ist eine Datenstruktur, welche Arten gibt es und wie beschreibt man sie?

Teil II Algorithmen in den Gärten der Strukturen

In diesem Teil geht es um das Zusammenspiel von Datenmodellen, Datenstrukturen und Algorithmen. Listen, Bäume und Graphen sind Datenmodelle. Sie liefern ihre jeweils spezifischen Problemstellungen, die mit Algorithmen und Datenstrukturen gelöst werden. Datenstrukturen und Algorithmen arbeiten dabei Hand in Hand.

Teil III Probleme und ihre Lösungen

Hier nehmen wir einen problemorientierten Standpunkt ein. Es beginnt mit Sortieren und dem Packen eines Rucksacks. Hier werden schon algorithmische Techniken angesprochen. Dann geht es um die Speicherung von Mengen. Datenabstraktionen als Hülle um Algorithmen und ihre Datenstrukturen werden besprochen und Hashtabellen und Bäume als geeignete Datenstrukturen behandelt. Mit zwei beispielhaften Problemen in Graphen endet dieser Teil.

Teil IV Algorithmische Techniken

Im vierten Teil gehen wir systematisch auf algorithmische Techniken ein. Das ist die Königsdisziplin der Algorithmen und Datenstrukturen, in der es darum geht, wie man auf Lösungsideen kommen kann und sie zu Lösungen organisiert.

Teil V Der Top-Ten-Teil

Im fünften und letzten Teil zählen wir noch einmal die zehn wichtigsten Datenstrukturen auf und geben zehn kleine Tipps für den Umgang mit Algorithmen.

Symbole, die in diesem Buch verwendet werden


Die Definition eines wichtigen technischen Begriffs wird mit diesem Symbol markiert. Diesen Begriff sollten Sie sich merken, denn er wird in dieser Bedeutung von allen genutzt, die in die Kunst der Algorithmen und Datenstrukturen eingeweiht sind.


Neben diesem Symbol finden Sie einen Hinweis auf die Behandlung des aktuellen Themas an anderer Stelle. Dort wird das Thema vielleicht vertiefend, vielleicht auch von einem anderen Gesichtspunkt aus betrachtet.


Wichtige Dinge sind oft einfach, werden aber trotzdem (oder gerade deswegen) schnell mal übersehen. Dieses Symbol weist auf so etwas hin. Ein praktischer und einfacher Tipp, der das Leben leichter macht.


Eine Anekdote oder ein vertiefender Hinweis, der interessante Aspekte des aktuellen Themas zeigt, die aber für das technische Verständnis nicht zwingend notwendig sind, werden mit diesem Symbol gekennzeichnet.


Gelegentlich ist es wichtig, sich Dinge noch einmal ins Gedächtnis zurückzurufen.

Mit diesem Symbol weisen wir darauf hin.



Die Thematik dieses Buchs ist reichhaltig genug, um ihr problemlos sein ganzes Leben widmen zu können. Mit diesem Symbol machen wir Sie gelegentlich auf vertiefenden Stoff aufmerksam. Solche Vertiefungen werden aber an anderer Stelle nicht vorausgesetzt und können darum problemlos übersprungen werden.

Wie es weitergeht

Am besten beginnen Sie mit Teil I. Von da aus können sie die anderen Teile nach Interesse und Bedarf erkunden.

Implementierungen zu den Algorithmen in diesem Buch finden Sie im Internet unter

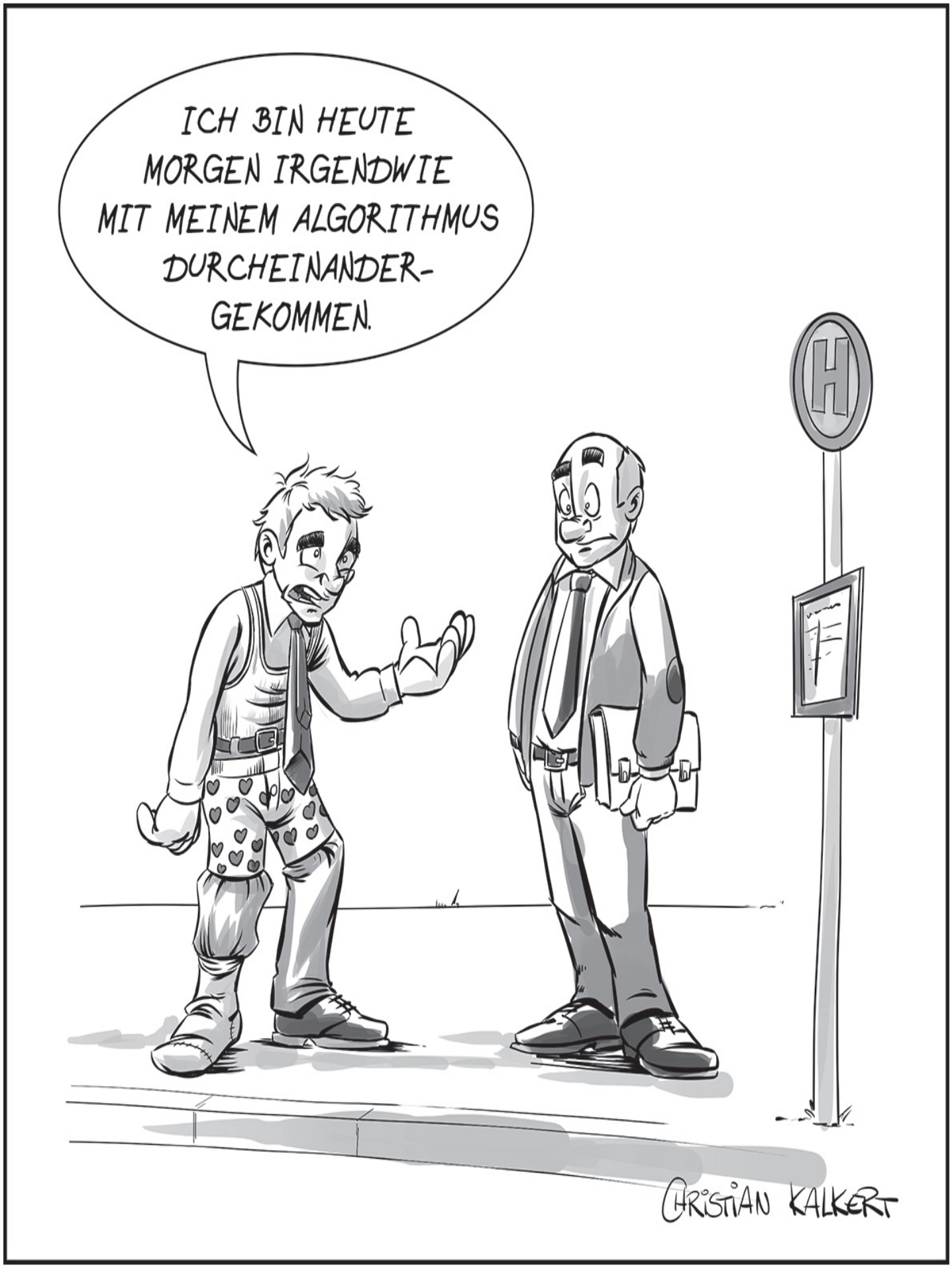
`www.algorithmendummies.de`

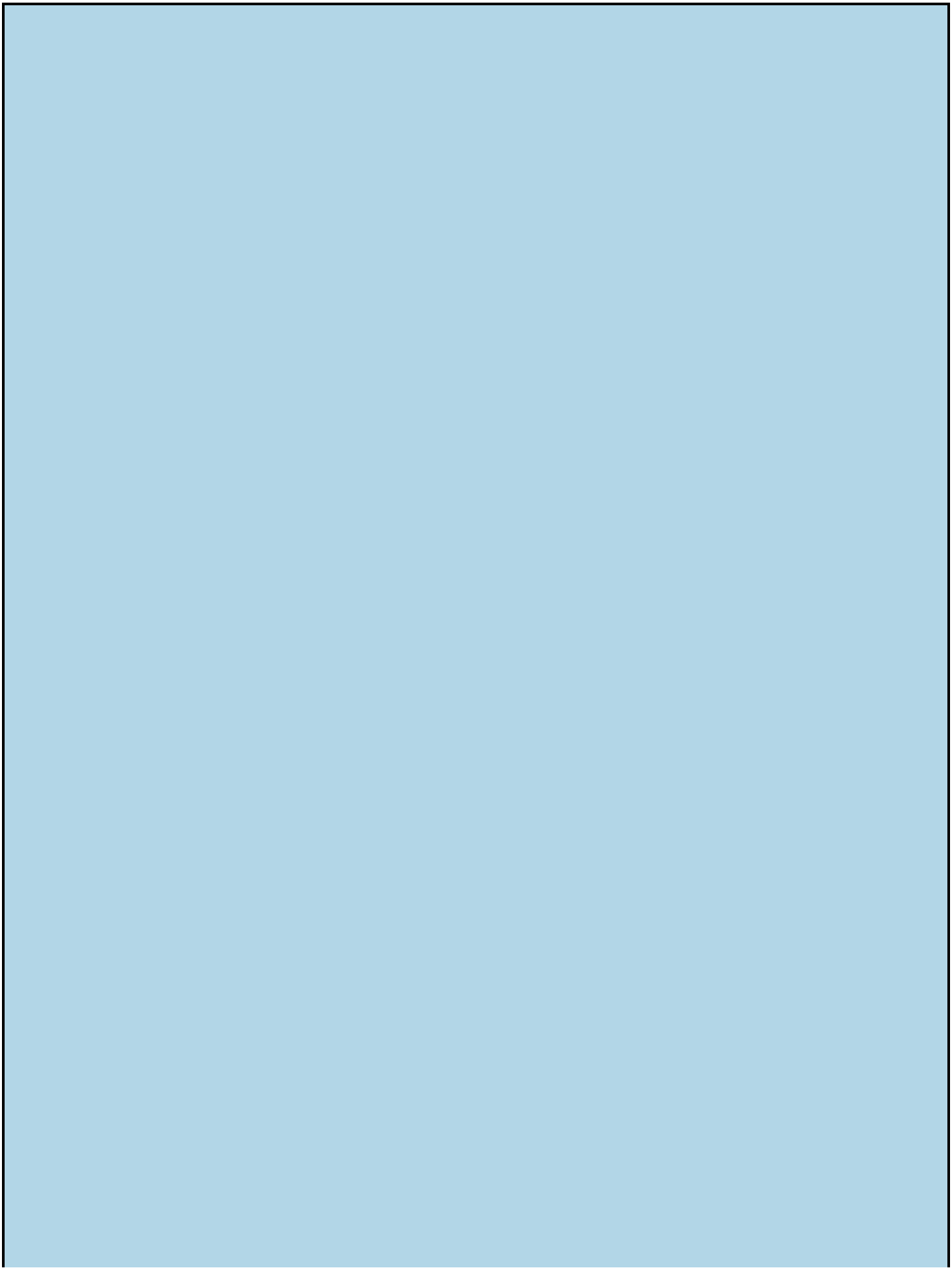
oder auf

`www.wiley-vch.de/ISBN9783527714322`

Teil I

Grundbegriffe


ICH BIN HEUTE MORGEN IRGENDWIE MIT MEINEM ALGORITHMUS DURCHEINANDER-GEKOMMEN.
H
CHRISTIAN KALKERT



IN DIESEM TEIL …

Was Algorithmen sind

Wie man sie beschreibt und bewertet

Was sind Daten?

Wie werden sie in Strukturen organisiert?

Kapitel 1

Algorithmen

IN DIESEM KAPITEL

Was sind Algorithmen?

Wie man Algorithmen klar formuliert

Der Pseudocode

Die Korrektheit von Algorithmen

Die »bedrohliche Macht der Algorithmen« und die »finsteren Datenkraken«, die sie nutzen, um unser Leben in ihren Griff zu bekommen, sind in aller Munde. Gemeint sind natürlich Algorithmen zur Analyse der Daten, die jeder bei seinem Gang durchs Leben produziert. Man mag davon halten, was man will, ein Gutes hat die Sache: Jeder hat den Begriff *Algorithmus* schon einmal gehört. Doch was genau ist ein Algorithmus?

Das sind Algorithmen

Algorithmen sind Verfahren, mit denen ein Problem gelöst oder eine Aufgabe erfüllt wird. Ein Verfahren braucht jemanden, der es ausführt. Dieser Jemand kann vielleicht nur ein einziges Verfahren, dann ist es eine spezialisierte Maschine. Papas alte Quarzuhr aus den Achtzigern kann zum Beispiel nichts weiter als die Zeit anzeigen und lässt sich auch nicht umprogrammieren. Die geniale Idee der Informatik sind aber die allgemeinen Maschinen, die *Computer*, die Beschreibungen beliebiger Verfahren akzeptieren und ausführen können.

Die Verfahren für Computer werden als *Programme* verfasst. Warum reden wir also noch von Algorithmen und nicht gleich von Programmen? Nun, weil Programme und Algorithmen nicht das Gleiche sind. Algorithmen sind der wesentliche Kern, der Gedanke, der durch völlig verschiedene Programme ausgedrückt werden kann. Programme werden in Programmiersprachen aufgeschrieben, Algorithmen meist in *Pseudocode*. Pseudocode ist nicht noch eine Programmiersprache. Es ist eine Notation für Algorithmen, also eine spezielle Schreibweise, mit der sich Menschen besser über den Ablauf eines Verfahrens austauschen können. Pseudocode ist eine einfache und doch klare Methode, Algorithmen aufzuschreiben, die unabhängig von den Moden der Programmiersprachen auch noch in 20 Jahren aktuell und lesbar sein wird wie heute.

In diesem Abschnitt beschäftigen wir uns darum mit folgenden Themen etwas genauer:

- ✔ Was genau versteht man unter einem Algorithmus?
- ✔ In welchem Verhältnis stehen
  - Programme und Algorithmen sowie
  - Funktionen und Algorithmen?
- ✔ Was ist Pseudocode und warum wird er verwendet?
- ✔ Warum spielt die mathematische Modellierung eine wichtige Rolle bei der Formulierung von Algorithmen?

Algorithmen lösen Probleme

Backrezepte sind Algorithmen. Natürlich werden dabei keine Daten verarbeitet, sondern Backzutaten. Auch wenn man bei einem Algorithmus heute an Daten denkt, so ist doch der Begriff *Algorithmus* etwas allgemeiner definiert.

 Ein *Algorithmus* ist ein

- ✔ wohldefiniertes,
- ✔ endlich beschreibbares,
- ✔ schrittweises
- ✔ Verfahren zum Lösen eines Problems oder der Erfüllung einer Aufgabe.

Bei einem Algorithmus geht es darum, dass nur Dinge formuliert werden, die wirklich machbar sind und keinerlei »magische Aktionen« erfordern. Beispielsweise ist folgendes Verfahren *kein* Algorithmus:

1. Hole einen Lottoschein.
2. Mache eine Zeitreise in die nächste Woche und beobachte, welche Zahlen gezogen werden.
3. Trage diese in den Schein ein und
4. gib ihn im Lottobüro ab.

Die notwendige Zeitreise ist nämlich etwas, das nur von ganz besonders befähigten Personen oder Maschinen ausgeführt werden kann. Wer einen Algorithmus liest und versteht, soll anschließend wissen, was genau er zu tun hat und wie er es tun soll. Auch das Pizzarezept in Abbildung 1.1 ist kein guter Algorithmus, denn auch wenn damit jeder zu einer Pizza kommt, lässt es doch zu viele Fragen offen. Und selbst wenn man in Einzelfällen vielleicht darüber streiten kann, was genau »wohldefiniert« ist und was nicht – das Prinzip ist sicher klar.



Abbildung 1.1: Dieses Backrezept erfüllt nicht die strengen Anforderungen an gute Algorithmen.

Bei den Algorithmen, die uns in diesem Buch am meisten interessieren, werden allerdings keine Backzutaten, sondern Daten verarbeitet. Freiheit von Magie bedeutet, dass alle Aktionen des Algorithmus auch von einem einfachen Computer ausgeführt werden können. Computer mit adressierbaren Speicherzellen, in denen Werte liegen, die durch

Zuweisungen und Rechenoperationen modifiziert werden können, sind die Basis für alle solchen Algorithmen.

Jeder Algorithmus hat ein Ziel. Er soll eine Aufgabe erfüllen oder ein Problem lösen. Beginnen wir mal mit einer einfachen Aufgabe, die wir großspurig *Summationsproblem* nennen.

> **Problem:** Summation
>
> **Eingabe:** eine natürliche Zahl n
>
> **Berechne:** die Summe s aller natürlichen Zahlen, die kleiner oder gleich n sind:
>
> $$s = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n$$

Wollen wir die Summe bis zu einem bestimmten Wert berechnen, sagen wir bis $n = 100$, dann haben wir es mit einer *Instanz* des Problems zu tun. Die allgemeine Fassung der Aufgabenstellung ist also das Problem, eine ganz spezielle Aufgabenstellung mit festgelegten Eingabedaten ist eine Probleminstanz.

Probleme kann man auf verschiedene Arten lösen. Am einfachsten ist es, wenn man die Lösung in einem Buch nachschlagen oder im Internet suchen kann. Aber das zählt hier nicht. Unsere Methoden sollen das Problem nämlich selbst lösen.

Für das Summationsproblem gibt es zwei Verfahren, von denen zumindest das erste jeder kennt. Entweder können wir die Zahlen, eine nach der anderen, aufaddieren. Alternativ können wir auch die sogenannte gaußsche Summenformel anwenden (Abbildung 1.2). Ein Problem, zwei Algorithmen, die auf unterschiedlichen Ideen zur Lösung des Problems beruhen. Hier zunächst der naive Aufaddier-Algorithmus: »Nimm als Eingabe eine natürliche Zahl n . Nun initialisiere die Summe s mit 0. Zähle anschließend alle Zahlen von 1 bis n auf und addiere sie jeweils zu s hinzu. Gib am Ende s zurück.«

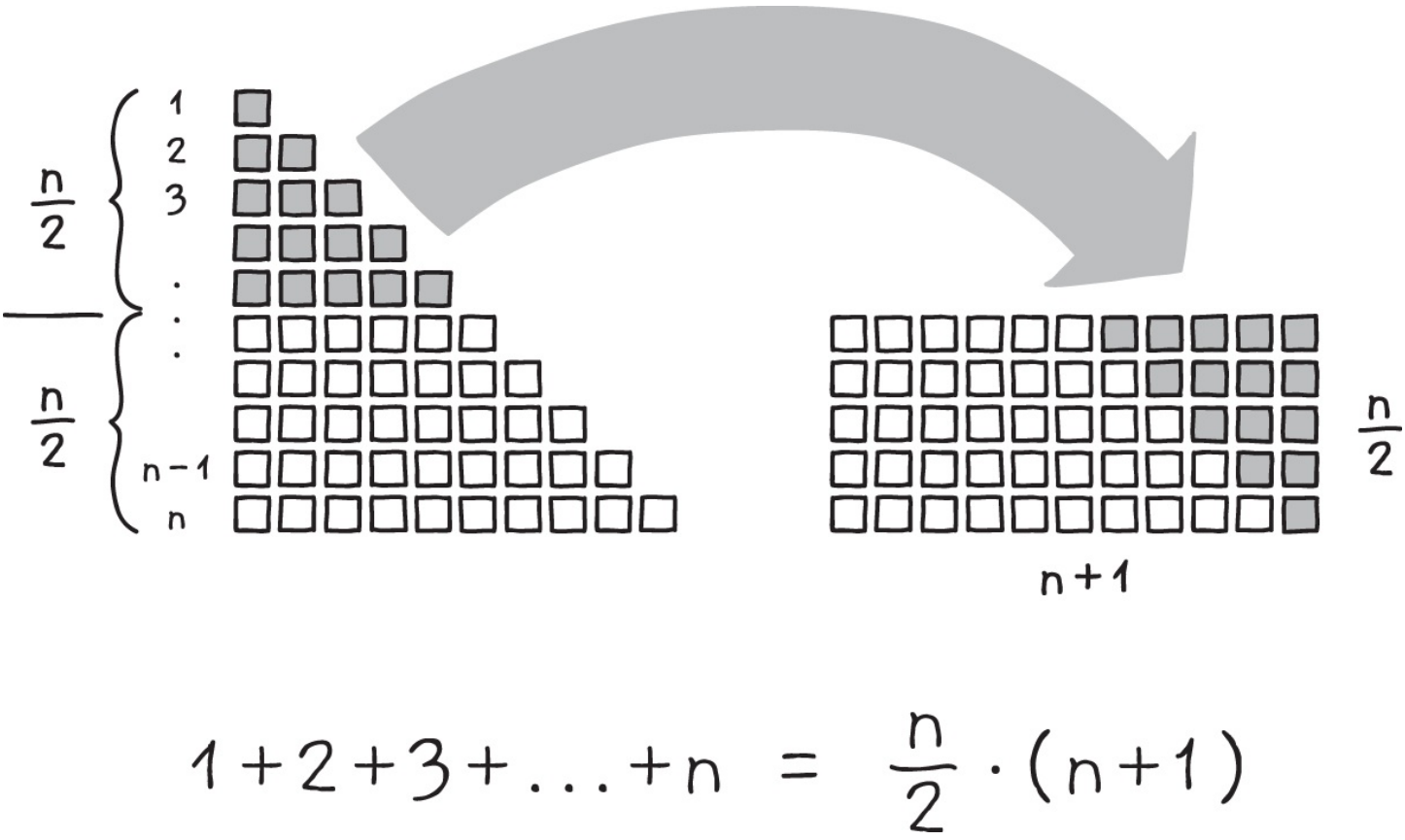


Abbildung 1.2: Die gaußsche Summenformel

Ein einfacher Algorithmus wie dieser lässt sich noch ganz gut auf Deutsch hinschreiben. Sobald die Algorithmen aber etwas komplizierter werden – und es werden in diesem Buch noch um einiges kompliziertere Algorithmen vorkommen –, wird es mit Pseudocode übersichtlicher. Hier also der gleiche Algorithmus in Pseudocode:

```
Summation1(n)                    n: eine natürliche Zahl
    s ← 0                        beginne mit 0 als bisher berechneter Summe
    s ← s + 1                    addiere 1 zur Summe s
    if n = 1                     wenn n = 1 ist,
        return s                 fertig
    s ← s + 2                    addiere 2 zur Summe s
    if n = 2                     wenn n = 2 ist,
        return s                 fertig
    s ← s + 3                    addiere 3 zur Summe s
    if n = 3                     wenn n = 3 ist,
        return s                 fertig
    s ← s + 4                    addiere 4 zur Summe s
    if n = 4                     wenn n = 4 ist,
        return s                 fertig
    ...                          und so weiter, und so weiter
```

Die drei Pünktchen »…« sollen andeuten, dass der Pseudocode immer weiter und weiter

und weiter geht. Schließlich kann das n beliebig groß sein. Aber Moment mal, das ist ja blöd, denn ein Algorithmus soll schließlich *endlich beschreibbar* sein! Wenn man einen unendlich langen Text braucht, um den Algorithmus aufzuschreiben – dann ist das überhaupt kein Algorithmus. Zugegeben: Wir haben auch keinen unendlich langen Text hingeschrieben (sonst wäre das Buch ja viel zu dick geworden), sondern ihn mit »…« abgekürzt. Diese drei Punkte sind allerdings ein wenig ungenau, denn es wird ja nicht direkt gesagt, was wiederholt werden soll. Wenn man den Algorithmus verstehen will, muss man im Abschnitt vor den drei Punkten nach einem wiederkehrenden Muster suchen. In diesem Fall wäre das zum Beispiel: »Die Zahl, die zu s addiert wird, wird jedes mal um 1 größer, und anschließend vergleicht man n auch immer mit dieser um 1 größeren Zahl«. Ein solcher Interpretationsbedarf ist aber umständlich, und darum wollen wir hier lieber eine Schreibweise verwenden, die *explizit*, also klipp und klar sagt, was genau zu tun ist:

```
Summation1(n)                 n: eine natürliche Zahl
    s ← 0                     beginne mit 0 als bisher berechneter Summe
    for i ← 1, ..., n         nimm jede Zahl i von 1 bis n
        s ← s + i             und addiere sie zur bisher berechneten Summe
    return s                  fertig
```

Das ist immer noch der gleiche Algorithmus Summation1 wie oben, nur deutlicher hingeschrieben. Was die ganzen Zeichen wie der Pfeil ← oder Wörter wie **for** und **return** bedeuten, werden wir etwas später noch genauer sagen. Nun erst einmal zur alternativen Methode mit gaußscher Summenformel:

```
Summation2(n)
    s ← n·(n+1)/2             berechne die Summe mit der gaußschen Formel
    return s                  fertig
```

Bei der Summation handelt es sich um ein *Berechnungsproblem*, bei dem aus Eingabedaten Ausgabedaten berechnet werden. Wenn man nur eine Ja-/Nein-Antwort haben will, dann handelt es sich um ein *Entscheidungsproblem*, und wenn man die beste unter vielen möglichen Lösungen benötigt, dann hat man es mit einem *Optimierungsproblem* zu tun.

Die *Lösung*, die ein Algorithmus zu einer Instanz des Problems liefert, muss natürlich korrekt sein. Beim einfachen Aufaddieren ist die Korrektheit offensichtlich. Die Formel liefert ebenfalls das korrekte Ergebnis. Das ist nicht ganz so offensichtlich, aber mit etwas Nachdenken oder der Hilfe von Wikipedia kann man sich davon überzeugen.

Fassen wir zusammen:



Ein *Algorithmus* ist ein Verfahren zur Lösung eines *Problems*. Es gibt verschiedene Arten von Problemen:

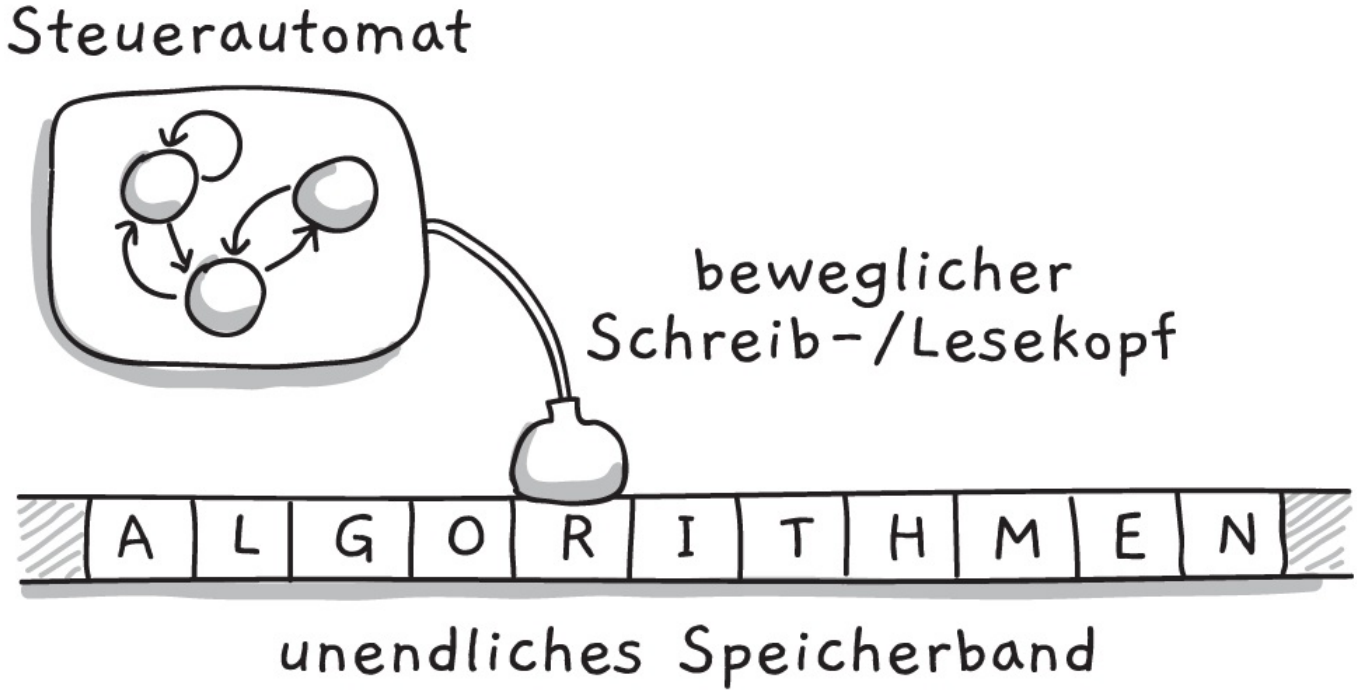
- ✔ Berechnungsprobleme,
- ✔ Entscheidungsprobleme und
- ✔ Optimierungsprobleme.

Eine *Probleminstanz* ist ein Problem für ganz bestimmte Eingabedaten.

Algorithmen basieren auf einem einfachen Maschinenmodell

Am Beginn der Informatik standen folgende Fragen aus dem Bereich der Grundlagenforschung in Mathematik und Informatik: Was darf man denn überhaupt hinschreiben, wenn man eine Berechnung definiert? Welche Rechnungen können tatsächlich ausgeführt werden? Was also ist überhaupt »rechenbar«? Daraus ist die Theorie der *Berechenbarkeit* entstanden. Am Schluss war man sich einig, dass etwas »rechenbar« ist, wenn es von einer wohldefinierten einfachen Maschine ausgeführt werden kann. Von allen Maschinenmodellen ist die *Turingmaschine* (Abbildung 1.3) wohl die bekannteste. Sie besteht aus einem einfachen Steuerautomaten, der sozusagen das Programm der Maschine beinhaltet, und hat darüber hinaus ein unbegrenzt langes Speicherband, in dem Buchstaben aus einem endlichen Alphabet gespeichert werden können. Mit einem Schreib-/Lesekopf kann die Turingmaschine in jedem Schritt eine Zelle des Speicherbands auslesen und verändern, und anschließend darf sie den Schreib-/Lesekopf um ein Feld nach links oder rechts bewegen. Das ist also eine denkbar einfache Maschine, die jedoch erstaunlich vielseitig ist.

Turingmaschine



Random Access Machine

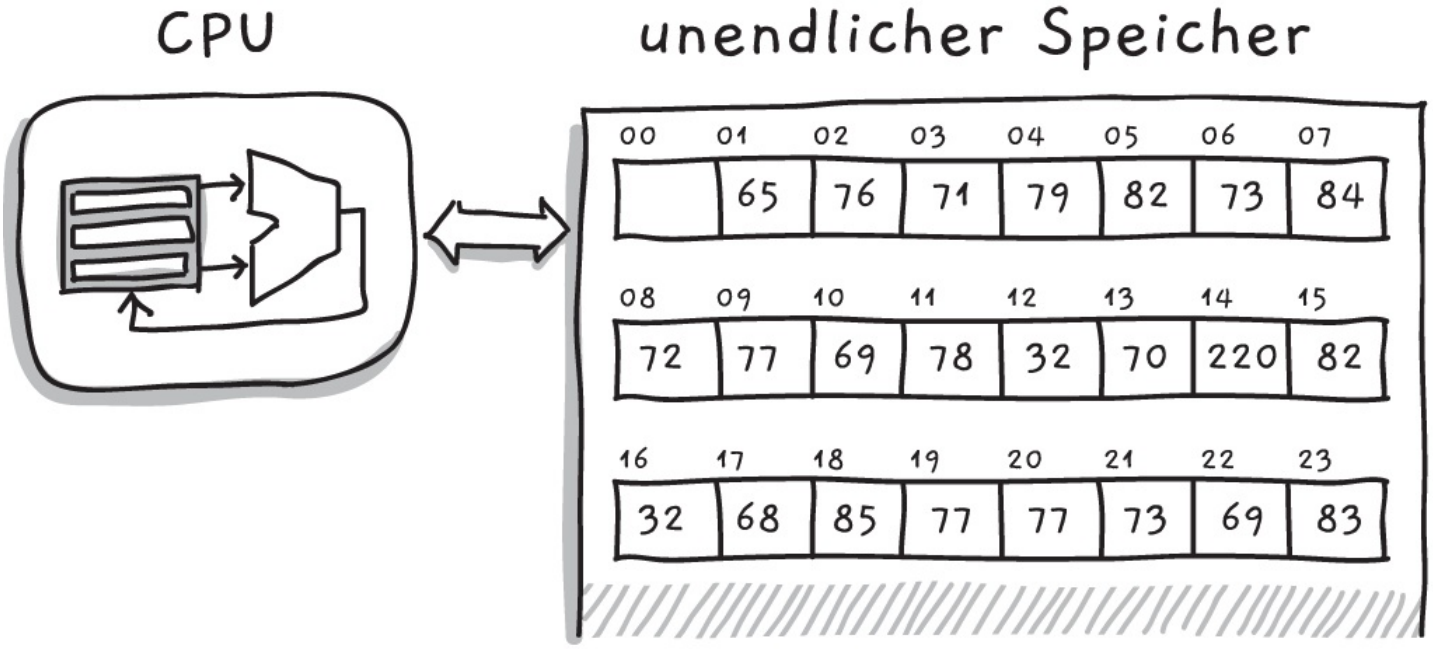


Abbildung 1.3: Turingmaschine und RAM sind zwei Maschinenmodelle.

Manch andere Maschinen sind eingeschränkter und man kann weniger mit ihnen berechnen als mit einer Turingmaschine. Nimmt man der Turingmaschine beispielsweise ihr Speicherband weg, sodass sie nur noch ein endlicher Automat ist, verliert sie deutlich an Möglichkeiten. Andere Maschinenmodelle hingegen können genauso viel wie die Turingmaschine, sie sind *Turing-vollständig*. Es gibt die unterschiedlichsten Turing-vollständigen Maschinenmodelle, die nach sehr verschiedenen Prinzipien funktionieren, aber wenn die eine Maschine ans Ziel kommt, dann schaffen das die anderen auch immer irgendwie.

Für die Theorie der Algorithmen verwendet man meist die sogenannte *Random Access Machine* (RAM), was auf Deutsch mit »Maschine mit wahlfreiem Speicherzugriff« übersetzt werden könnte. Das ist ein einfacher hypothetischer Computer mit einer unbegrenzten Zahl an adressierbaren Speicherstellen, auf dem Programme mit Zuweisungen und den üblichen Kontrollanweisungen ablaufen können. Das Gute an der RAM: Sie ist den heute üblichen Computern hinreichend ähnlich, sodass jeder mit ein wenig Programmiererfahrung sofort ein Gefühl dafür hat, was für die RAM eine »elementare« Operation sein könnte und was nicht. Wie reale Computer ist sie außerdem Turing-vollständig und eignet sich damit ausgezeichnet als Grundlage für Algorithmen. In Pseudocode soll alles erlaubt sein, was in Aktionen einer RAM übersetzt werden kann.

Die RAM hat folgende Fähigkeiten:

- Sie hat Speicherstellen für beliebige endliche Daten.
- Diese können mit Zuweisungen und den üblichen arithmetischen Operationen manipuliert werden.

Daneben kann die RAM auch Kontrollanweisungen ausführen, nämlich

- bedingte Anweisungen,
- Schleifen sowie
- (rekursive) Funktions- und Prozedurdefinitionen.

Im Zweifelsfall müsste noch spezifiziert werden, welche Operationen im Detail zur Verfügung stehen. Reelle Zahlen sind beispielsweise nicht mit endlichem Speicher exakt darstellbar, sie gehören darum nicht zu den elementaren Daten der RAM. Wenn das von Relevanz ist, dann muss zum Algorithmus angegeben werden, wie genau die Zahlen angenähert dargestellt werden und welche Operationen mit wie vielen Schritten diese Zahlen verarbeiten. Ist es irrelevant, dann wird das Thema ignoriert.

Algorithmen sind bewertbar

Wenn man nicht gerade ein Lehrer ist, der mal seine Ruhe haben will und seine Schüler dazu mit Rechenarbeiten beschäftigt hält, dann ist die zweite Methode der Summation, die mit der gaußschen Formel, natürlich die bessere. Statt wie Summation1 $n-1$ Additionen zu benötigen, wird von Summation2 die Summe von 1 bis n mit nur einer Addition, einer Multiplikation und einer Division berechnet.

Wir haben gerade die *Effizienz* der beiden Algorithmen bewertet. Genauer gesagt die *Laufzeit-Effizienz*. Dabei geht es darum, wie viele elementare Rechenschritte ein Algorithmus ausführen muss, um das Problem zu lösen. Die Zahl der Rechenschritte hängt in der Regel von der Eingabe ab. Für ein großes n werden bei der ersten Summationsmethode mehr Rechenschritte benötigt als bei einem kleinen n. Die Zahl der Rechenschritte bei der Anwendung der gaußschen Formel ist dagegen unabhängig von der Eingabe. Allerdings ist für sehr kleines n wie zum Beispiel $n=2$ die Berechnung der Summe mit der gaußschen Formel

$$s = \frac{2 \cdot (2+1)}{2} = 3$$

sicher weniger effizient als das einfache Aufaddieren

$$s = 1 + 2 = 3$$

Um solche Ausreißer zu vermeiden, lässt man kleine Werte darum bei der Bewertung der Effizienz eines Algorithmus außer Acht.



Die Laufzeit-Effizienz eines Algorithmus sagt etwas darüber aus, wie die Zahl der Rechenschritte, die der Algorithmus für eine bestimmte Eingabe ausführt, von deren Größe abhängt. Mehr darüber erfahren Sie in Kapitel 2.

Fassen wir zusammen:



Ein Problem kann oft von mehreren unterschiedlichen Algorithmen gelöst werden, die sich in ihrer Effizienz unterscheiden. Die *Laufzeiteffizienz* ist das wichtigste Effizienzkriterium. Sie gibt an, wie viele Rechenschritte er bei der Ausführung benötigt.

Algorithmen agieren in Modellwelten

Im Prinzip kann ein Algorithmus mit beliebigen Dingen umgehen. Ein Kochrezept ist ein Algorithmus, der mit Kochzutaten arbeitet. Wir betrachten aber nur Algorithmen, die Daten verarbeiten. Natürlich stammen die Daten meist aus der realen Welt und die vom Algorithmus berechneten Daten wirken auch wieder auf die reale Welt ein.

Ein Programm, das den kürzesten Weg zwischen zwei Orten sucht, läuft nicht in der Gegend herum, sondern arbeitet mit Daten, welche die Landschaft repräsentieren. Die digitale Version einer Landkarte enthält viele Informationen, von denen nur manche für

das aktuelle Problem relevant sind. Bei dem Problem des kürzesten Weges sind das beispielsweise die Orte und Wege, die sie verbinden, und deren jeweilige Länge.

Ein Algorithmus zur Wegfindung wird so formuliert, dass er nur die auf das Wesentliche konzentrierten Informationen als Eingabe hat. Bei der Wegfindung ist das ein Graph: Objekte und ihre mit einer Bewertung versehen Verbindungen. In der Welt der Algorithmen und Datenstrukturen verwendet man gerne Konzepte aus der diskreten Mathematik: Mengen, Relationen, Graphen. Damit können die wesentlichen Informationen einer Problemstellung und deren Lösung leicht und übersichtlich zum Ausdruck gebracht werden.

Ein Programm, das mit einer bestimmte Form einer digitalen Landkarte arbeitet, muss natürlich mit den Daten in der konkreten Form arbeiten, in der sie zur Verfügung stehen. Das können Tabellen einer Datenbank sein, Klassen und Objekte einer objektorientierten Programmiersprache und vieles mehr. Ein Algorithmus löst das Problem in der mathematischen Modellwelt. Das ist sozusagen die »Zeichnung« der Lösung. Wird der Algorithmus in einer bestimmten Programmiersprache implementiert, dann transferiert man die Aktionen des Algorithmus auf »mathematischen Objekten« in Aktionen des Programms auf konkreten Daten (Abbildung 1.4).

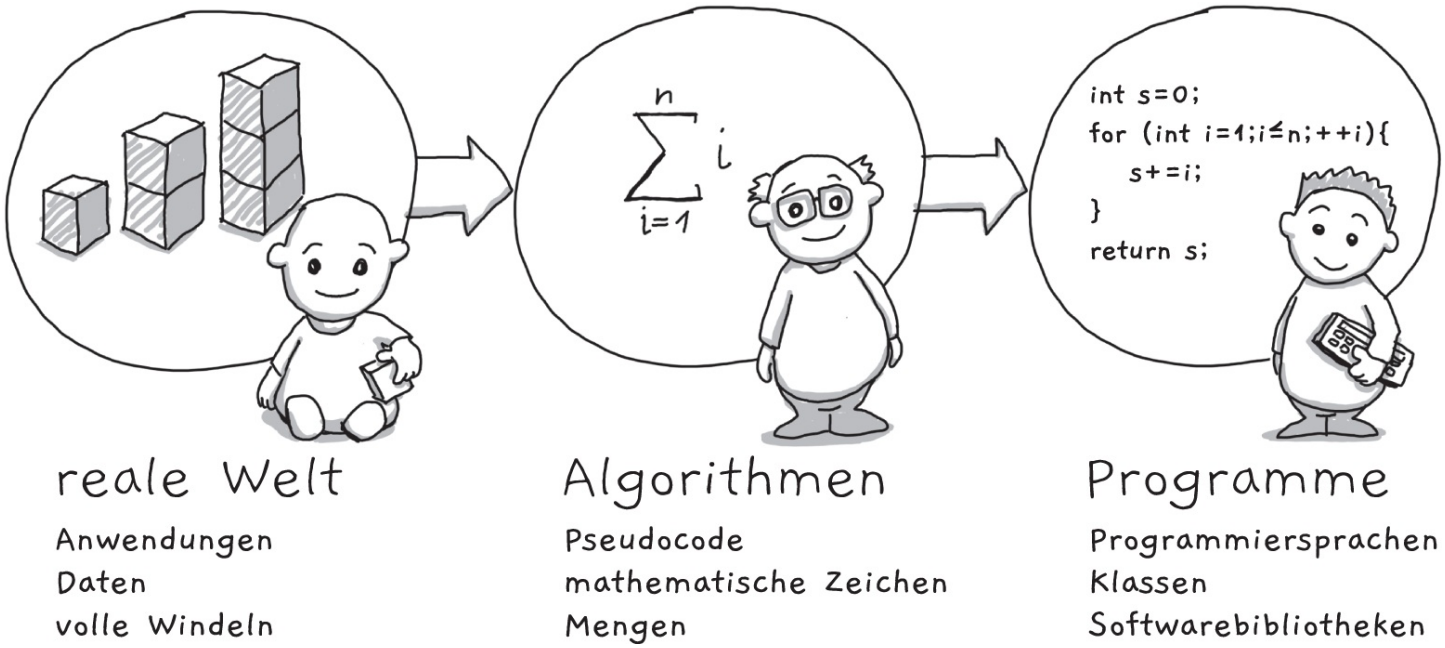


Abbildung 1.4: Algorithmen und Programme in Modellwelten

Algorithmen sind keine Programme

Ein Algorithmus ist ein Verfahren zur Lösung eines Problems, bei dem man sich auf das Wesentliche konzentriert: Welche relevanten Informationen werden mit welchen elementaren Schritten in eine Lösung transformiert? Das ist wie eine Melodie, die man singen oder auf diversen Musikinstrumenten spielen kann. Unterschiedlich, aber irgendwie doch immer das erkennbar Gleiche.

Pseudocode entspricht der Notenschrift, in der das Wesentliche aufgeschrieben wird: Knapp, klar und unabhängig von bestimmten Instrumenten. Ein Programm ist etwas, das ein Prozessor ausführen kann. Zum Beispiel eine bestimmte synaptische Struktur im Gehirn eines Sängers, eine Rille in einer Schallplatte, Nullen und Einsen in einer MP3-Datei und so weiter. Bei einer Ausführung beziehungsweise Aufführung wird das Programm vom Prozessor abgearbeitet (Abbildung 1.5). Die Musik ertönt entsprechend der Melodie, und wenn die Noten nicht verloren gehen, wird sie noch genauso zum Klingen gebracht werden können, wenn jede Erinnerung an Schallplatten und MP3-Dateien vergangen ist.

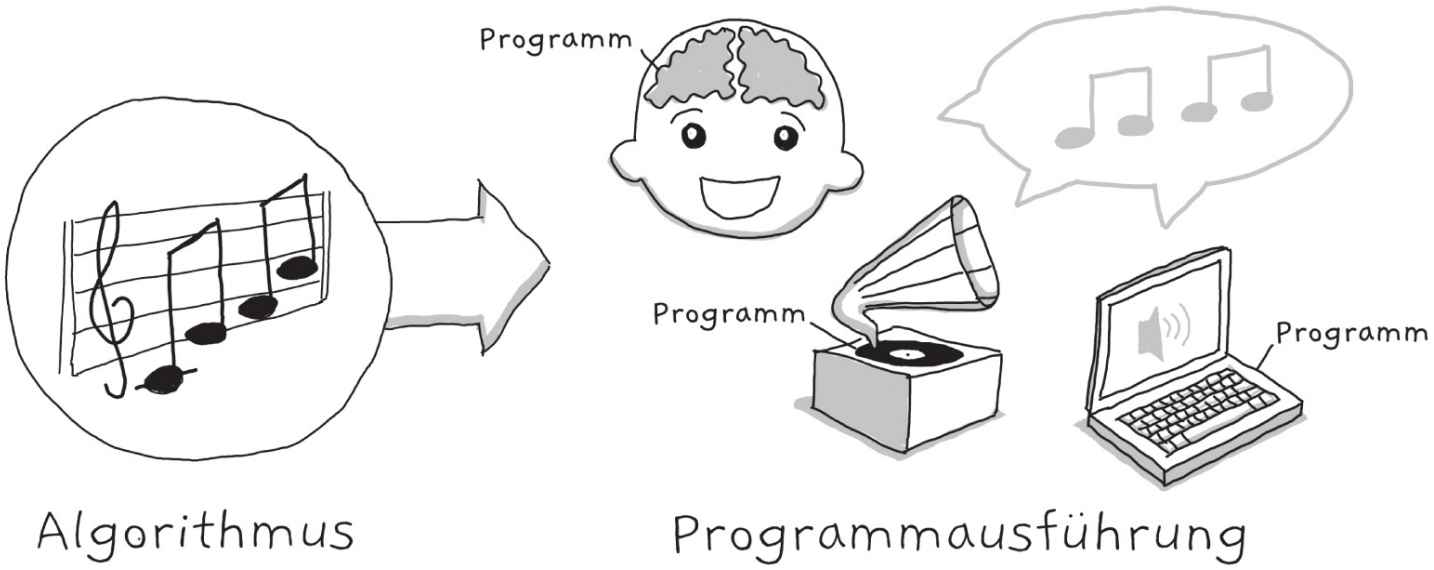


Abbildung 1.5: Ein Algorithmus und drei ausführbare Programme, die ihn implementieren

Programme und Algorithmen sind also unterschiedliche Dinge. Eine *Programmiersprache* hat auch viel mehr zu leisten, als nur eine Notation für die Ausführung von Algorithmen zu sein. Sie muss

- ✔ in effizienten Maschinencode übersetzbar oder von einem Interpreter ausführbar sein,
- ✔ von ihren Anwendern effektiv zu nutzen sein und
- ✔ bestimmten softwaretechnischen Anforderungen genügen.

Programmiersprachen haben immer ein exakt definiertes beschränktes und festes Repertoire an Ausdrucksmitteln. Dazu gehört vieles, das nicht direkt mit Algorithmen zu tun hat. Manche legen Wert darauf, dass die Sprache mit wenig Mühe schnell erlernbar ist, andere dass sie in einem bestimmten Anwendungsgebiet besonders effektives Arbeiten ermöglicht oder dass sie die Konstruktion von Anwendungen mit hunderttausenden von Zeilen Quellcode effektiv unterstützt etc.

Diese Ziele widersprechen sich teilweise. Um sie zu erreichen, gibt es diverse Ebenen, auf denen sich die Programmierer bewegen können, und eine mehr oder weniger umfangreiche Sammlung von Konzepten zur Modularisierung und Abstraktion: Klassen,

Funktionen, Pakete, generische Klassen, Module, was auch immer. All das unterliegt nicht nur dem aktuellen Zeitgeist, der mal hierhin und mal dahin weht. Es unterscheidet sich auch drastisch von Sprache zu Sprache je nach deren Einsatzgebiet, vermuteter Kompetenz der Entwickler und so weiter.

All dies ist sehr wichtig, lenkt aber hier nur vom eigentlichen Kern der Dinge, dem Algorithmus ab.

Algorithmen klar beschreiben

Algorithmen werden oft in Pseudocode beschrieben. Pseudocode ist für Menschen gemacht. Er darf darum deren Kreativität ausschöpfen. Alles Magische ist verboten, alles andere prinzipiell erlaubt. Magisch ist das, was sich nicht in Aktionen der RAM ausdrücken lässt. Im Sinne einer guten Lesbarkeit und Präzision versucht man in der Regel alles möglichst formal hinzuschreiben. Unter guten Freunden darf man aber auch manchmal etwas vage bleiben und einzelne Schritte zum Beispiel auf Deutsch beschreiben. Es muss aber immer klar sein, wie diese Notation in zählbare Aktionen auf der RAM umgesetzt werden kann.

Sprechen Sie Pseudocode?

Der Sprachumfang des Pseudocodes muss nicht bis ins Detail formal definiert werden, denn es handelt sich ja nicht um eine Programmiersprache, sondern um ein Kommunikationsmedium, mit dem sich Menschen über Algorithmen austauschen. Wir wollen trotzdem einmal die wichtigsten »Sprachelemente« durchgehen und uns darüber verständigen, was wir damit meinen.

In diesem Buch wird ein Pseudocode benutzt, der weithin übliche Sprachelemente benutzt und vielleicht ein wenig an die Programmiersprache Python erinnert. Anweisungsblöcke in einer Schleife oder solche, die in einer **if** -Anweisung ausgeführt werden, werden durch eine gemeinsame Einrückung nach rechts gekennzeichnet, enden also da, wo die Anweisungen nicht mehr rechts eingerückt sind.

Algorithmen

Algorithmen im Pseudocode entsprechen grob den Funktionen, Prozeduren oder Methoden in den Programmiersprachen. Jeder Algorithmus beginnt zunächst mit seinem Namen gefolgt von einer Liste an Eingabeparametern in runden Klammern:

Die gleiche Notation verwenden wir beim Aufruf eines Algorithmus.

Algo(n) *Algo ist ein Algorithmus, der n als Eingabe bekommt*

Variablen und Zuweisungen

Variablen dienen zum Speichern von Daten aller Art. Sie ändern ihren Wert durch eine Zuweisung. Wir verwenden für Zuweisungen nicht wie oft üblich ein Gleichheitszeichen,

sondern einen Pfeil:

$i \leftarrow 17$	*die Variable i wird auf den Wert 17 gesetzt*
$j \leftarrow i + 1$	*j ist nun 18, i bleibt 17*

Das Gleichheitszeichen »=« reservieren wir für Vergleiche.

Variablen sind in der Regel lokal, das heißt, sie gelten nur innerhalb des aktuellen Aufrufs des Algorithmus. Das ist insbesondere bei rekursiven Algorithmen nützlich.

Schleifen

Bei einer Schleife wird ein Anweisungsblock mehrfach ausgeführt. Wir verwenden zwei unterschiedliche Arten von Schleifen, die **for** - und die **while** -Schleife.

Bei der **for** -Schleife wird eine Variable nacheinander auf eine Reihe von Werten gesetzt und dann jeweils ein Anweisungsblock ausgeführt:

for $i \leftarrow 1, \ldots, 17$	*i wird nacheinander auf 1, 2, ..., 17 gesetzt ...*
Algo(i)	*... und für jedes i der Algorithmus Algo aufgerufen*
Algo(100)	*diese Zeile wird erst nach der Schleife ausgeführt*

Bei der **while** -Schleife wird zunächst eine Bedingung geprüft und im Fall, dass diese erfüllt ist, ein Anweisungsblock ausgeführt. Dies wird so lange wiederholt, bis die Bedingung zum ersten Mal nicht erfüllt wurde:

$i \leftarrow 1$	*die Variable i wird auf den Wert 1 gesetzt*
while $i < 17$	*solange $i < 17$ ist ...*
Algo(i)	*... rufe Algo mit i als Parameter auf ...*
$i \leftarrow i + 1$	*... und erhöhe i um 1*
Algo(100)	*diese Zeile wird erst ausgeführt, wenn $i \geq 17$ ist.*

Bedingte Anweisungen

Mit **if** wird eine Bedingung geprüft, und wenn sie erfüllt ist, wird ein Anweisungsblock ausgeführt:

if $i \geq 17$	*wenn $i \geq 17$ ist, ...*
Algo(i)	*... rufe den Algorithmus Algo mit i als Parameter auf*

Folgt nach einem **if** ein **else** , so wird der nachfolgende Anweisungsblock ausgeführt, wenn die Bedingung der vorangegangenen **if** -Anweisung *nicht*- erfüllt war:

if $i \geq 17$	*wenn $i \geq 17$ ist, ...*
Algo(i)	*... rufe den Algorithmus Algo mit i als Parameter auf*
else	*ansonsten, also wenn $i < 17$ ist, ...*
Algo(100)	*... rufe den Algorithmus Algo mit Parameter 100 auf*

Sonstiges

Mit break verlässt man die aktuelle Schleife.

Mit return beendet man den aktuellen Aufruf eines Algorithmus. Dabei kann man optional auch noch einen Rückgabewert für den aufrufenden Algorithmus zurückgeben.


Weitere Sprachelemente werden bei der Diskussion der Datenstrukturen in den Kapiteln 3 bis 6 eingeführt.

Mathematische Ausdrücke sind erlaubt

Im Pseudocode verwenden wir viele Elemente der üblichen mathematischen Notation. Wir nutzen zum Beispiel Mengendefinitionen, bei denen man nicht jedes einzelne Element aufzählt, sondern die Elemente der Menge über eine Eigenschaft bestimmt. Wenn M zum Beispiel eine Menge ist, können wir Folgendes schreiben:

$a \leftarrow \{x \in M \mid x \text{ ist gerade}\}$ *a ist die Menge aller geraden Zahlen in M*

Entsprechendes kann man auch für geordnete Folgen L definieren, nur dann mit eckigen statt geschweiften Klammern:

$b \leftarrow [x \in L \mid x \text{ ist gerade}]$ *b ist eine Liste, die alle geraden Zahlen aus L enthält*

Derartige Konstrukte erleichtern die Formulierung von Algorithmen enorm. Sie werden von vielen modernen Programmiersprachen unterstützt (leider nicht von Java) und dort zum Beispiel *For-Comprehension* genannt.

Wenn man mag, kann man es auch noch weitertreiben und zum Beispiel noch ein Summenzeichen davor schreiben:

$s \leftarrow \sum\{x \in M \mid x \text{ ist gerade}\}$ *s ist die Summe aller geraden Zahlen in M*

Das ist deshalb kein Problem, weil jeder Leser des Pseudocodes sofort weiß, wie er das in »normalen« Pseudocode umsetzen kann, nämlich zum Beispiel so:

```
s ← 0                          s auf 0 initialisieren
for x ← M                      alle Elemente in M durchlaufen
    if x ist gerade            sobald eine gerade Zahl x gefunden wurde, ...
        s ← s + x              ... wird sie zu s addiert
```

Die Regel lautet immer: Alles ist erlaubt, solange jeder versteht, was gemeint ist und wie man es in elementare Operationen des Maschinenmodells (also zum Beispiel der RAM) umsetzen kann.

Algorithmen sprechen sogar Deutsch

Wenn man möchte, darf man im Pseudocode sogar einfach die deutsche Sprache verwenden, wie wir das zum Beispiel gerade mit dem Ausdruck »x **ist gerade** « getan haben. Das ist vollkommen in Ordnung, solange jedem im Prinzip klar ist, wie man auf der RAM prüfen kann, ob eine Zahl gerade oder ungerade ist. Bei Bedarf kann man das natürlich weiter ausformulieren, zum Beispiel mit »x **modulo 2 = 0** «. Dabei gehen wir

davon aus, dass für die RAM wie für alle gängigen Computer die Operation **modulo** zu den Basisoperationen gehört, dass sie also den Rest einer Division zweier ganzen Zahlen berechnen kann.

Wenn man es eilig hat und in einem Algorithmus die Summe aller geraden Zahlen in einer Menge M benötigt, so könnte man das im Pseudocode auch einfach so schreiben:

$s \leftarrow$ die Summe aller geraden Zahlen in M

Aber Vorsicht: Im Zweifelsfall müssen Sie immer in der Lage sein, das genauer zu erklären, es also bis auf elementare Operationen einer RAM herunterzubrechen. Nachdem wir im vorangegangenen Abschnitt ausführlich darüber geredet haben, sollte das nun ja auch kein Problem mehr sein.

Algorithmen sind Lösungen, keine Probleme

Wenn wir mathematische Definitionen oder sogar deutsche Sätze in unsere Algorithmen einbauen, müssen wir allerdings aufpassen, dass am Ende auch wirklich ein Algorithmus dabei herauskommt. Ein Algorithmus ist ein Verfahren zum Lösen eines Problems, nicht das Problem selbst. Mathematiker haben zum Beispiel keine Hemmungen, Mengen zu definieren, von denen niemand weiß, wie man sie tatsächlich berechnen könnte. Bei manchen mathematischen Mengen kann man sogar beweisen, dass man sie gar nicht berechnen kann. Im Abschnitt *Das Halteproblem ist unlösbar* werden Sie eine solche Menge kennenlernen. Ein Problem bloß zu definieren, ist also etwas ganz anderes, als es auch zu lösen.

Lassen Sie uns dazu eine Aufgabe betrachten, die schon die Landvermesser im alten Ägypten zu lösen hatten, nämlich der Suche nach dem größten gemeinsamen Teiler zweier natürlicher Zahlen, also zweier Zahlen aus der Menge $\mathbb{N} = \{1, 2, 3, \ldots\}$:

> **Problem:** Größter Gemeinsamer Teiler (GGT)
>
> **Eingabe:** zwei natürliche Zahlen n und m
>
> **Berechne:** die größte natürliche Zahl t, die sowohl ein Teiler von n als auch von m ist

Man könnte versuchen, das Problem auf folgende Weise zu lösen:

```
GGT1(n, m)                                   n und m seien natürliche Zahlen
    t ← max{x ∈ ℕ | x teilt n und x teilt m}     berechne den GGT
    return t
```

Lassen Sie uns der Klarheit halber die Menge noch einmal in Form einer Schleife ausformulieren:

```
GGT1(n, m)                          n und m seien natürliche Zahlen
    t ← 1                           1 ist ein gemeinsamer Teiler von n und m
    for x ← ℕ                       durchlaufe alle Zahlen aus ℕ in aufsteigender Reihenfolge
        if x teilt n und x teilt m  wenn x gemeinsamer Teiler von n und m ist, ...
            t ← x                   ... merke dir x
    return t
```

Das ist der gleiche Algorithmus, nur anders hingeschrieben. Allerdings gibt es ein Problem: Es gibt unendlich viele natürliche Zahlen, also wird die **for** -Schleife niemals damit fertig, sie aufzuzählen. Der Algorithmus ist in einer unendlichen Schleife gefangen, umgangssprachlich ausgedrückt: *Er hängt sich auf.*

Dass sich Algorithmen »aufhängen« können, wird uns gleich noch ausführlich beschäftigen. Zum Glück kann man den Algorithmus leicht retten: Alle Zahlen größer als n sind keine Teiler von n , und alle Zahlen größer als m keine Teiler von m . Der GGT muss also sowohl $\leq n$ also auch $\leq m$ bleiben, das heißt, es reicht aus, die Schleife bis zum Minimum von m und n laufen zu lassen:

```
GGT2(n, m)                          n und m seien natürliche Zahlen
    t ← 1                           1 ist ein gemeinsamer Teiler von n und m
    for x ← 2, ..., min(n, m)       durchlaufe alle Zahlen von 2 bis min(n, m)
        if x teilt n und x teilt m  wenn x gemeinsamer Teiler von n und m ist, ...
            t ← x                   ... merke dir x
    return t
```

Das können wir dann auch wieder verkürzt als mathematische Menge formulieren, und wie folgt hinschreiben:

```
GGT2(m, n)                                          n und m seien natürliche Zahlen
    g ← min(n, m)                                   g sei das Minimum von n und m
    t ← max{x ∈ {1, ..., g} | x teilt n und x teilt m}   berechne den GGT
  return t
```

Das ist zwar nicht der schnellste Weg zur Berechnung des GGT, aber immerhin eine Möglichkeit.

Algorithmen haben zählbare Schritte

Wenn wir später Algorithmen miteinander vergleichen wollen, so sollte exakt bestimmbar sein, wie viele elementaren Schritte ein Algorithmus benötigt, wenn man ihn auf einer Eingabe einer gegebenen Größe ausführt. Die Laufzeit des Algorithmus GGT2 zur Berechnung des größten gemeinsamen Teilers wird zum Beispiel entscheidend vom Minimum der beiden Eingabewerte n und m beeinflusst: Bis zu diesem Wert läuft die Schleife.

In jedem Schleifendurchlauf des Algorithmus wird eine Zahl x daraufhin geprüft, ob sie ein Teiler von n und m ist. Diese Prüfung kann natürlich auch auf unterschiedliche

Arten erfolgen. Man könnte beispielsweise x so lange von m beziehungsweise n subtrahieren, bis das Ergebnis 0 oder eine Zahl kleiner 0 ist. Endet man bei 0, dann ist x ein Teiler, ansonsten nicht. Damit hat unser Algorithmus dann die folgende Definition:

```
GGT3(n, m)                          n und m seien natürliche Zahlen
    t ← 1                           1 ist ein gemeinsamer Teiler von n und m
    for x ← 2, ..., min(n, m)       durchlaufe alle Zahlen von 2 bis min(n, m)
        if isDivisor(x, m) und isDivisor(x, n)   wenn x Teiler von n und m ist, ...
            t ← x                   ... merke dir x
    return t
isDivisor(x, z)                     prüft, ob x ein Teiler von z ist
    while z > 0                     solange noch z größer als 0 ist, ...
        z ← z − x                   ... ziehe x von z ab
    return (z = 0)                  wenn am Ende z = 0 ist, war z am Anfang
                                    durch x teilbar, sonst nicht
```

Natürlich braucht ein Algorithmus, der die Teilbarkeit in einem Schritt bestimmen kann, weniger Schritte als einer, der für jeden Test eine Schleife laufen lassen muss. Die meisten modernen Prozessoren unterstützen die Modulo-Operation »n **modulo** x«, die den Rest einer ganzzahligen Division von n geteilt durch x berechnet. In vielen Programmiersprachen wird diese Operation mit einem %-Zeichen geschrieben, andere schreiben dafür mod. Ist dieser Rest gleich 0, so ist n offenbar durch x teilbar; ist der Rest hingegen ungleich 0, so ist n nicht durch x teilbar. Dieser Test geht schnell, in einem einzigen Schritt. Es ist darum legitim anzunehmen, dass der Test auf Teilbarkeit unabhängig von der Größe der involvierten Werte ist. Wir unterstellen unserer RAM einfach die entsprechende Fähigkeit.

Wenn der Algorithmus andererseits auf einer Maschine läuft, bei der dies nicht gegeben ist, dann sieht die Sache völlig anders aus. Die Berechnung kann dann unter Umständen nicht mehr in einem Schritt erfolgen, sondern benötigt zum Beispiel eine Extraschleife, deren Laufzeit von x und n abhängt. In diesem Fall sollte man das bei der Beschreibung des Algorithmus auch dringend so hinschreiben, damit niemand auf die irrige Idee kommt, dass die Teilbarkeit eine elementare Operation des hinter dem Algorithmus stehenden Maschinenmodells ist.



Mehr über die Abschätzung und den Vergleich der Laufzeiten von Algorithmen finden Sie in Kapitel 2.

Algorithmen sollten korrekt sein

Wenn Sie sich einen Algorithmus ausdenken und hinschreiben, dann möchten Sie natürlich auch, dass er korrekt ist. Er soll genau das tun, was Sie von ihm wollen, und

zwar bei jeder zulässigen Eingabe. Wie aber kann man dafür sorgen, dass ein Algorithmus tatsächlich so funktioniert, wie man sich das wünscht?

Die kurze Antwort auf diese Frage: Dafür gibt es kein Patentrezept. Algorithmen können im Allgemeinen nämlich ziemlich schwer zu durchschauen sein.

Betrachten Sie zum Beispiel einmal den folgenden unbekannte Algorithmus und überlegen Sie, was er berechnet, wenn man ihm zwei natürliche Zahlen x und y übergibt:

```
Riddle(n, m)                          n und m seien zwei natürliche Zahlen
    x ← n
    y ← m
    p ← 0
    while x ≥ 1
        if x ist gerade
            x ← x/2
        else
            p ← p + y
            x ← (x−1)/2
        y ← 2 · y
    return p
```

Am besten setzen Sie einfach ein paar Zahlen für n und m ein, finden heraus, was Riddle bei diesen Eingaben berechnet, und stellen anschließend eine Hypothese darüber auf, was der Algorithmus wohl berechnen könnte. Die Auflösung des Rätsels finden Sie im Abschnitt *Die Lösung des Rätsels*.

Algorithmen können sich aufhängen

Ein Algorithmus sollte sich möglichst nicht »aufhängen«, also keine unendliche Laufzeit haben, weil sie sich zum Beispiel in einer unendlichen Schleife verfangen haben. Wenn sich ein Algorithmus einfach aufhängt, brauchen wir über seine Korrektheit gar nicht mehr nachzudenken. Ein Ausdruck wie

$$t \leftarrow \max\{x \in \mathbb{N} \mid x \text{ teilt } n \text{ und } x \text{ teilt } m\}$$

ist darum nicht erlaubt. Deswegen hängt sich der Algorithmus GGT1 im Abschnitt *Algorithmen sind Lösungen, keine Probleme* ja auch auf.

Im Fall einer Mengennotation, die ja letztlich nur eine verkürzte Schreibweise für eine **for** -Schleife ist, kann man das auch sehr gut erkennen und vermeiden, denn bei einer **for** -Schleife sieht man ja von Anfang an, wie oft die Schleife durchlaufen wird. Schwieriger wird das bei **while** -Schleifen oder auch bei rekursiven Algorithmen, also Algorithmen, die sich wiederholt selbst aufrufen. Da ist es oft gar nicht so leicht zu erkennen, was sie

tun und ob sie überhaupt irgendwann einmal fertig werden. Betrachten Sie zum Beispiel folgenden einfachen Algorithmus:

```
Collatz(n)                          n: eine natürliche Zahl
    while n > 1                     solange n noch größer als 1 ist, tue Folgendes:
        if n ist gerade
            n ← n/2                 wenn n gerade ist, teile n durch 2
        else
            n ← 3 · n + 1           ansonsten multipliziere n mit 3 und addiere 1 dazu
```

Kommt dieser Algorithmus immer irgendwann zu einem Ende oder gibt es Eingaben n, bei der er immer weiterläuft und niemals endet? Der Mathematiker Lothar Collatz hat diese Frage erstmals vor über 80 Jahren gestellt, und sie ist bis heute unbeantwortet geblieben. Zwar wurde noch keine Zahl n gefunden, bei der sich der Algorithmus aufhängen würde, umgekehrt hat aber auch noch niemand bewiesen, dass es keine solche Zahl gibt. Schaut man sich an, wie sich n im Laufe des Algorithmus verändert, so erkennt man, dass n immer wieder wild rauf- und runterspringt. In Abbildung 1.6 sehen Sie das am Beispiel $n = 25$: Für diese Eingabe braucht der Algorithmus 23 Durchläufe der **while** -Schleife, und n erreicht dabei eine maximale Höhe von 88. Wären wir stattdessen bei $n = 27$ gestartet, hätten wir 111 Schleifendurchläufe benötigt, und n wäre dabei bis in eine Höhe von 9323 aufgestiegen. Größere Startwerte bedeuten aber nicht zwangsläufig längere Laufzeiten; bei $n = 32$ geht es zum Beispiel immer bergab und die »Talstation« ist nach nur fünf Schritten erreicht.

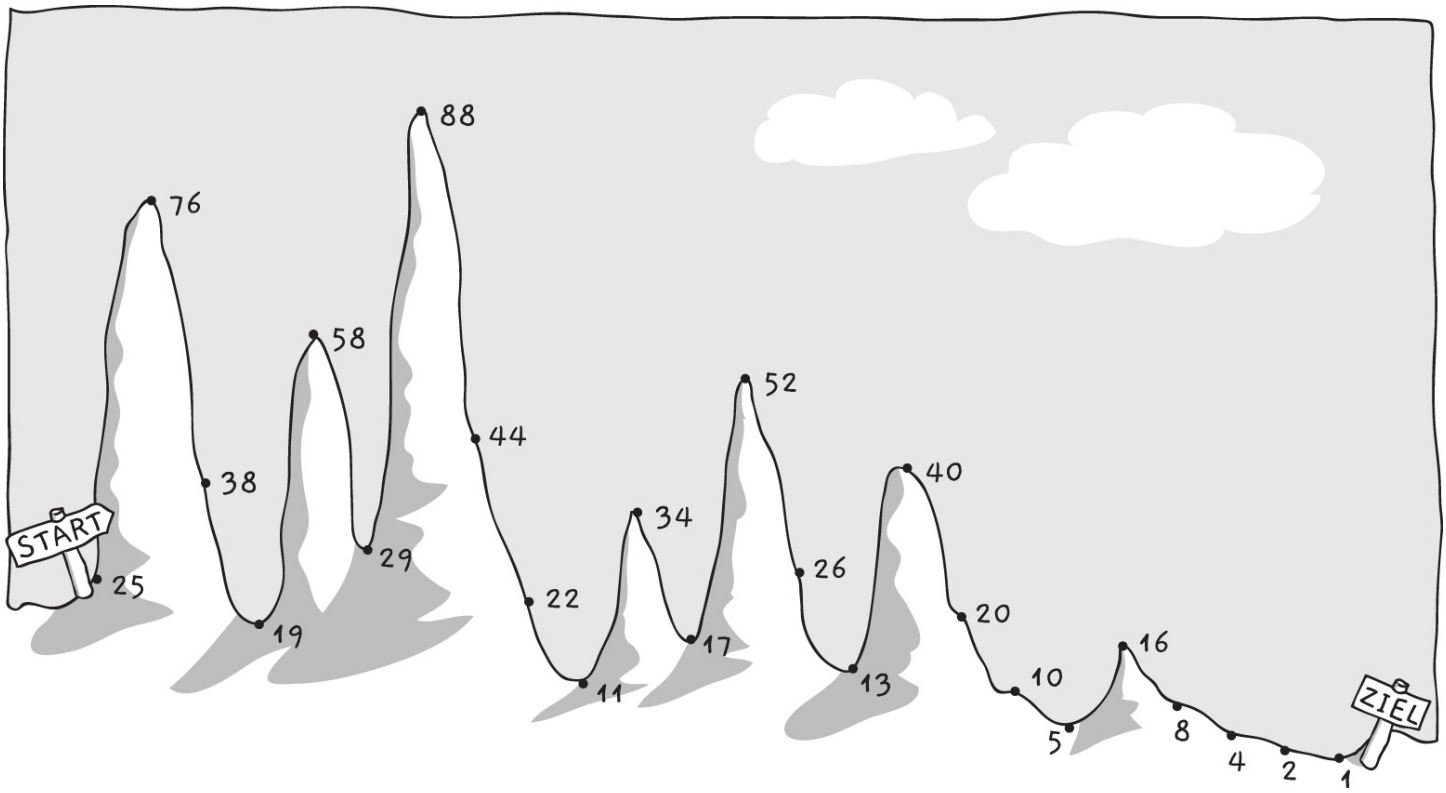


Abbildung 1.6: Der Startpunkt n = 25 für den Pfad durch das Collatz-Gebirge

Das Halteproblem ist unlösbar

Bei der Wanderung durch das »Collatz-Gebirge« sind Abstürze also nicht ausgeschlossen. Aber warum ist es bloß so schwer herauszufinden, ob ein Algorithmus irgendwann endet oder nicht? Könnte man nicht eine generelle Methode finden, um genau das zu entscheiden? Lassen Sie es uns als algorithmisches Problem formulieren:

Problem: das Halteproblem

Eingabe: ein Algorithmus A und eine Eingabe x

Berechne: entscheide, ob A gestartet auf x irgendwann anhält oder ob sich der Algorithmus aufhängt

Leider gibt es keinen Algorithmus, der das Halteproblem für beliebige Eingaben A und x löst. Das kann man sogar beweisen. Das Halteproblem ist ein klassisches Beispiel für ein nicht berechenbares Problem. Zwar könnte man einfach A mit der Eingabe x starten und sehen, was passiert, wenn aber A endlos weiterläuft, dann findet man das auf diese Weise auch erst nach unendlich langer Zeit – also niemals – heraus. Der Ausdruck

$$\{(A,x)|A \text{ ist ein Algorithmus, } x \text{ eine Eingabe, und } A(x) \text{hängt sich nicht auf}\}$$

ist also ein Beispiel für eine Menge, die man zwar durchaus mathematisch korrekt definieren kann, für die es aber keine Berechnungsmethode gibt.

Zwar kann man bei vielen Algorithmen tatsächlich beweisen, dass sie sich nicht aufhängen, es gibt nur kein Verfahren, mit dem sich das für beliebige Algorithmen zeigen ließe.

Wenn wir es also schwer haben herauszufinden, ob sich ein Algorithmus aufhängt oder nicht, warum verändern wir dann nicht einfach unser Maschinenmodell auf eine Weise, dass sich Algorithmen grundsätzlich nicht mehr aufhängen können? Lassen Sie uns doch die ganzen bösen **while** -Schleifen und Rekursionen verbieten, die uns so große Probleme bereiten. Stattdessen könnten wir uns zum Beispiel auf die sicheren **for** -Schleifen beschränken. Ja, das würde gehen, hätte allerdings einen Nachteil: Unser Maschinenmodell wäre anschließend nicht mehr Turing-vollständig. Gäbe es nämlich ein Turing-vollständiges Maschinenmodell, das sich nicht aufhängen kann, so könnte man damit das Halteproblem lösen, und das ist wie gesagt gar nicht möglich. So gesehen ist es also eigentlich ganz gut, dass Algorithmen grundsätzlich die Möglichkeit haben, sich aufzuhängen, denn nur dadurch können Sie ihr volles Potenzial auszuschöpfen.



Algorithmen können sich »aufhängen«, das heißt, sie laufen immer weiter und enden nie. Ob sich ein Algorithmus aufhängt oder nicht, lässt sich leider oft nur schwer abschätzen, und es gibt es keine formale Methode, mit der sich das immer sicher verhindern lässt.

Algorithmen richtig verstehen

Wenn es schon schwer ist festzustellen, ob ein Algorithmus irgendwann anhält oder nicht, dann kann es sicherlich auch nicht leichter sein herauszufinden, was der Algorithmus überhaupt tut, und ob er auch das tut, was man sich von ihm erwünscht.

Die Lösung des Rätsels

Um das zu veranschaulichen, lassen Sie uns über den geheimnisvollen Algorithmus Riddle weiter vorne in diesem Kapitel reden.

Man kann ziemlich leicht erkennen, dass sich der Algorithmus immerhin nicht aufhängt. Bei jedem Durchlauf der while-Schleife verringert sich nämlich der Wert von x, und das tut er so lange, bis $x < 1$ ist und die Schleife abbricht.

Was genau berechnet der Algorithmus denn nun? Haben Sie es herausgefunden?

Die Antwort ist einfach: Riddle berechnet das Produkt aus den beiden Eingaben n und m, also

$$\textbf{Riddle}(n, m) = n \cdot m$$

Übrigens nennt man den Algorithmus auch »ägyptische Multiplikation«. Die Methode scheint auch tatsächlich zu funktionieren. Wenn man eine Stichprobe macht und irgendwelche Werte für x und y einsetzt, kommt jedenfalls das Richtige dabei heraus.

Aber können wir uns sicher sein, dass Riddle tatsächlich *immer* das Produkt von n und m berechnet? Nur weil es bei ein paar Beispielen für n und m gestimmt hat, muss es ja nicht für alle Eingaben funktionieren. Wenn wir ganz sicher sein wollen, müssten wir das Ergebnis also eigentlich für alle möglichen natürlichen Zahlen n und m überprüfen. Aber das geht nicht, denn dann hätten wir ja unendlich viel zu tun.

Nun hätte man das Gleiche auch für praktisch alle anderen Algorithmen sagen können, die wir bisher in diesem Buch besprochen haben. Wie können wir uns beispielsweise sicher sein, dass GGT2 im Abschnitt *Algorithmen sind Lösungen, keine Probleme* auch tatsächlich den größten gemeinsamen Teiler ausrechnet, und zwar für *alle* möglichen Eingaben? Auch hierfür gibt es ja wieder unendlich viele Möglichkeiten, und die können wir unmöglich alle ausprobiert haben. Der Unterschied zwischen den beiden Algorithmen ist, dass GGT2 sehr einfach zu verstehen ist, während man bei Riddle nicht sofort einsieht, warum der Algorithmus überhaupt funktioniert.

Wenn wir uns sicher sein wollen, dass ein Algorithmus korrekt ist, müssen wir ihn also *wirklich* verstehen. Ihn ganz und gar durchdringen. Dafür reicht es nicht aus, dass man den Algorithmus für Beispieleingaben Schritt für Schritt nachvollziehen kann. Jede einzelne Zeile in Riddle ist klar und verständlich, aber in ihrem Zusammenspiel ergeben sich Konsequenzen, die man nicht direkt überblickt.

Am besten wäre es, wenn uns jemand den Algorithmus erklären könnte. Wenn wir einen

Text hätten, der mit klaren, nachvollziehbaren Argumenten schlüssig begründet, warum der Algorithmus das Produkt seiner Eingaben n und m berechnet. Was wir also brauchen, ist ein *Beweis*.



Wenn bei einem Algorithmus nicht offensichtlich ist, dass er das tut, was er soll, dann benötigt man einen Beweis, um ihn richtig zu verstehen und seine Korrektheit einzusehen.

Korrektheit beweisen

Wenn ein Algorithmus nur eine feste Anzahl von Anweisungen nacheinander ausführt, dann ist es noch vergleichsweise einfach, seine Korrektheit zu überprüfen. Schwieriger wird es, wenn die Anzahl der Anweisungen, die während eines Algorithmus ausgeführten werden, nicht von Anfang an feststeht, sondern von der Eingabe abhängt. In Riddle gibt es eine **while** -Schleife, die so lange durchlaufen wird, bis der Wert $x < 1$ wird. Am Anfang wird x auf den Eingabewert n gesetzt und anschließend bei jedem Schleifendurchlauf halbiert und dabei abgerundet, bis x schließlich den Wert 0 erreicht hat und die Schleife abbricht. Je größer n ist, desto häufiger wird also die Schleife durchlaufen. Da n beliebig groß sein kann, kann die Schleife beliebig oft durchlaufen werden. Unser Beweis soll aber nicht aus unendlich viel Text bestehen, also brauchen wir eine Methode, die Korrektheit des Algorithmus zu begründen, ohne für jeden einzelnen Schleifendurchlauf extra zu argumentieren.

Der Trick, den man in diesem Fall gewöhnlich verwendet, heißt *Induktionsbeweis*. Die Idee dabei ist, dass man wie folgt argumentiert:

> »Wenn bis zum i -ten Schleifendurchlauf alles korrekt funktioniert hat, dann funktioniert auch im nachfolgenden $i + 1$ -ten Schleifendurchlauf alles korrekt.«

Dabei muss man sich natürlich erst einmal im Klaren darüber sein, was es genau heißen soll, dass bei einem Schleifendurchlauf »alles korrekt funktioniert«. Hierzu verwendet man gewöhnlich eine sogenannte *Schleifeninvariante*. Das ist zum Beispiel eine Größe, die sowohl vor als auch nach jedem Schleifendurchlauf gleich bleibt, oder allgemein eine Aussage, die sowohl vor als auch nach jedem Schleifendurchlauf gültig ist. Im Fall der ägyptischen Multiplikation (also des Algorithmus Riddle weiter vorne in diesem Kapitel) könnten wir zum Beispiel folgende Gleichung als Schleifeninvariante (SI) verwenden:

$$n \cdot m = x \cdot y + p$$

Der Korrektheitsbeweis läuft dann so:

1. Ganz am Anfang, also bevor die **while** -Schleife das erste Mal durchlaufen wird, gilt die SI, denn es wird $x \leftarrow n$, $y \leftarrow m$ und $p \leftarrow 0$ gesetzt, und das bedeutet, dass $n \cdot m = x \cdot y + p$ ist (*Induktionsanfang*)

2. Angenommen, wir stehen am Anfang eines Schleifendurchlaufs, und bisher gilt die SI. Wir müssen zeigen, dass die SI auch am Ende des Schleifendurchlaufs gilt (*Induktionsschritt*). Zwei Fälle sind zu unterscheiden (siehe Abbildung 1.7):

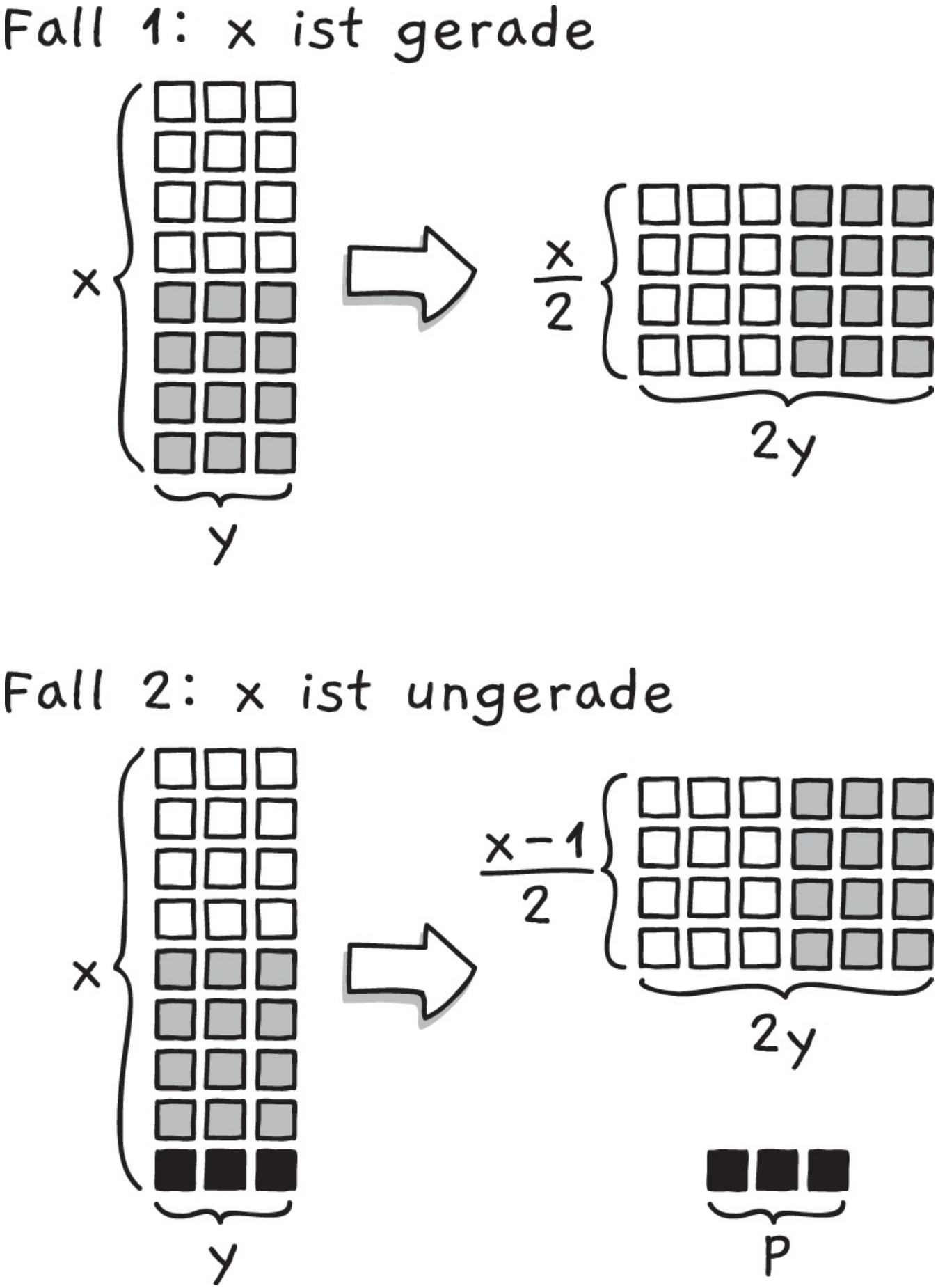


Abbildung 1.7: Die zwei Fälle der ägyptischen Multiplikation

a. Fall 1: x ist am Anfang des Schleifendurchlaufs *gerade*. Dann wird x halbiert und y verdoppelt, das Produkt $x \cdot y$ bleibt also gleich. Da sich p ebenfalls nicht ändert, verändert sich der Ausdruck $x \cdot y + p$ nicht, und die SI behält auch nach dem Schleifendurchlauf ihre Gültigkeit.

b. Fall 2: x ist am Anfang des Schleifendurchlaufs *ungerade*. Nun wird beim Halbieren von x abgerundet, und wie man in Abbildung 1.7 sehen kann, wird dadurch das Produkt $x \cdot y$ um die schwarzen Kästchen kleiner. Diese gehen zum Glück nicht verloren, sondern werden zu p hinzugezählt. Am Ende der Schleife hat sich $x \cdot y + p$ also nicht geändert, die SI gilt somit immer noch.

3. Die **while** -Schleife endet, wenn x nicht mehr ≥ 1 ist. Das passiert, sobald $x = 0$ geworden ist; und wenn $x = 0$ ist, dann ist auch $x \cdot y = 0$. Da die SI über alle Schleifendurchläufe hinweg bestanden hat, gilt sie auch jetzt noch, und somit ist $n \cdot m = x \cdot y + p = p$. Der Algorithmus liefert p als Ergebnis zurück und berechnet somit das Produkt aus n und m .



Korrektheitsbeweise von Algorithmen sind häufig Induktionsbeweise, die über die Durchläufe einer Schleife hinweg die Gültigkeit einer Schleifeninvariante belegen. Dabei ist eine Schleifeninvariante etwas, das über die Schleifendurchläufe hinweg »invariant« ist, also gleich bleibt.

Auf der Suche nach einem Korrektheitsbeweis können Schleifeninvarianten also nützlich sein. Dabei reicht es allerdings nicht aus, *irgendeine* Schleifeninvariante zu finden, denn letztlich soll sie auch dazu geeignet sein, die Korrektheit des Algorithmus zu beweisen. Aber wie findet man denn eine geeignete Schleifeninvariante? Sie ahnen es sicherlich: Auch dafür gibt es kein Patentrezept. Beim Führen von Beweisen sind Sie leider auf ihre eigene Kreativität angewiesen.

Kapitel 2

Qualität von Algorithmen

IN DIESEM KAPITEL

- Die Effizienz von Algorithmen
- Asymptotische Laufzeitanalyse
- Groß-$\mathcal{O}$-Notation
- Faustregeln zur Laufzeitbestimmung

Wenn man ein Problem mit einem Algorithmus lösen will, dann ist es natürlich erst einmal am wichtigsten, es *überhaupt* zu lösen. Das ist oft schwierig genug und mitunter kann man froh sein, wenn man irgendwie eine Lösung hinbekommt. Andererseits reicht dieses »irgendwie« manchmal nicht aus. Wenn Sie zum Beispiel einen Algorithmus programmieren, und er würde auf den schnellsten Computern der Welt hunderttausend Jahre brauchen, um zu einem Ergebnis zu kommen, oder Millionen von Petabytes an Speicherplatz benötigen, dann nützt Ihnen dieser Algorithmus vermutlich gar nichts. In der Praxis muss ein Algorithmus nicht nur *im Prinzip* seine Aufgabe erfüllen können, er darf dabei auch nur eine vernünftige Menge an Ressourcen wie Zeit oder Speicherplatz verbrauchen.

So gut sind Algorithmen

Dieses Buch beschäftigt sich ausführlich damit, wie man *gute* Algorithmen findet, also solche, die möglichst wenige Ressourcen verbrauchen. Je weniger solcher Ressourcen ein Algorithmus benötigt, desto besser. Wenn ein Algorithmus zum Beispiel zu viel Zeit verbraucht, seine Laufzeit also zu lang ist, dann ist er langsam und das können Algorithmiker gar nicht leiden. Nur schnelle Algorithmen sind gute Algorithmen. Bevor wir aber gute Algorithmen finden können, müssen wir erst einmal überlegen, ob und wie man den Ressourcenverbrauch von Algorithmen überhaupt bestimmen kann.

Wenn Algorithmen Programme wären, dann wäre das leicht. Um die Laufzeit eines Programms zu messen, muss man es nur laufen lassen und die Zeit messen, die das Programm braucht, bis es fertig ist. Fast ebenso leicht kann man bestimmen, wie viel Speicherbedarf ein Programm verschlingt; dazu besitzen alle Betriebssysteme mehr oder weniger präzise Messmethoden.

Ein Algorithmus ist aber kein Programm, sondern eine Methode, die durch ein Programm

umgesetzt werden kann. Algorithmen abstrahieren von den konkreten Umständen ihrer Implementierung, von den verschiedenen Programmiersprachen, Betriebssystemen und Rechnern, auf denen man sie verwirklichen kann. Darum ist es zum Beispiel schwierig zu sagen, welche Rechenzeit ein Algorithmus tatsächlich benötigt, denn diese Rechenzeit hängt ja nicht nur vom Algorithmus selbst ab, sondern auch ganz entscheidend davon, auf welchem Computer das Programm läuft. Opas alter Rechner aus den Achtzigern wird es in Sachen Geschwindigkeit kaum mit Enkelchens brandneuem Gaming-PC aufnehmen können, selbst wenn es gelänge, dasselbe Programm auf beiden Computern zum Laufen zu bringen.

Darum ist es schwierig, von »der« Laufzeit eines Algorithmus zu sprechen, und wir sollten stattdessen darüber nachdenken, wie man verschiedene Algorithmen miteinander vergleichen kann (Abbildung 2.1).

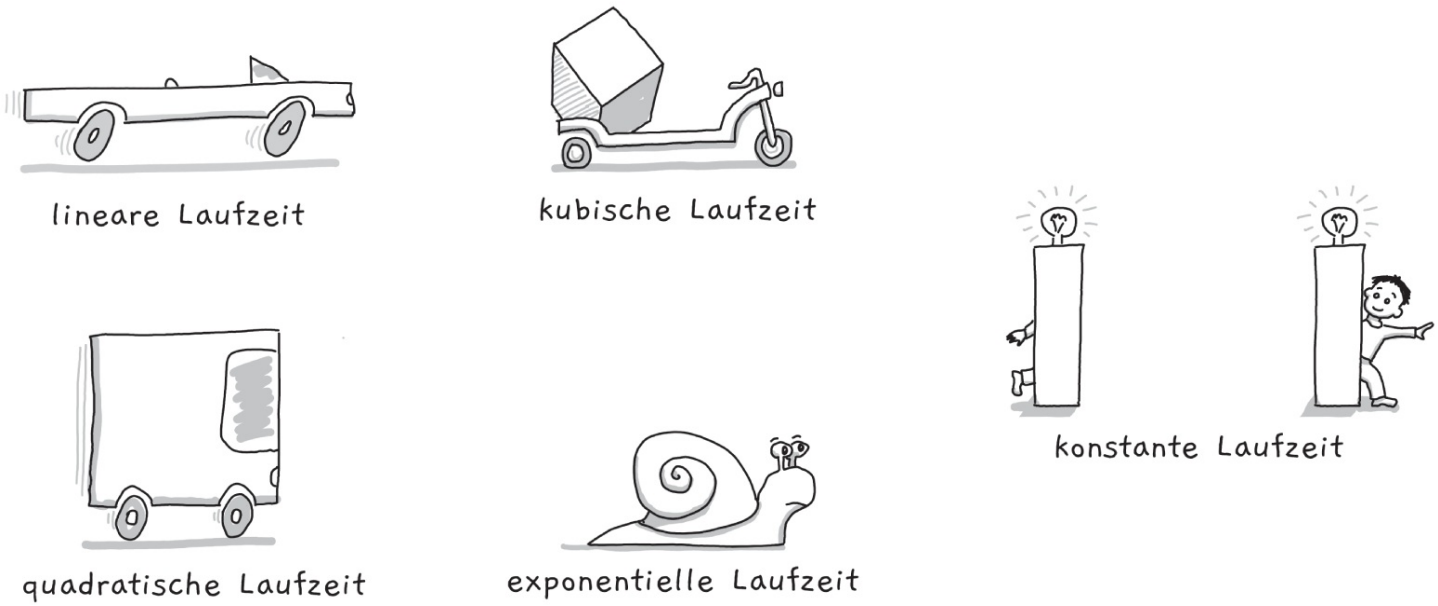


Abbildung 2.1: Algorithmen können unterschiedlich schnell sein.

Wir werden uns im Folgenden sehr eingehend mit der Analyse von Laufzeiten beschäftigen; alle vorgestellten Methoden können entsprechend auch für den Verbrauch anderer Ressourcen verwenden werden. Zuerst kommt jetzt aber ein Beispiel für zwei unterschiedlich schnelle Algorithmen, die das gleiche Problem lösen sollen.

Wer ist der Leichteste?

Eine Gruppe von Bauarbeitern – rein zufällig alles Männer – will bestimmen, wer von ihnen das Loch auf dem Dach eines Neubaus reparieren soll. Da sie beim Bau des Hauses nicht sehr gewissenhaft vorgegangen sind und keiner von ihnen viel Vertrauen in die Stabilität der Dachkonstruktion hat, soll diese gefährliche Aufgabe derjenige von ihnen übernehmen, der am wenigsten wiegt.

Leider weiß keiner der Männer sein genaues Gewicht, doch zum Glück gelingt es ihnen, aus einem Brett und einem dreieckigen Ziegelstein eine Waage zu bauen (Abbildung 2.2).

Stellen sich zwei der Arbeiter auf die beiden Enden des Brettes, so zeigt sich sofort, wer von ihnen der leichtere ist. Manchmal kann es sein, dass zwei Arbeiter genau das gleiche Gewicht haben, und auch das kann man zuverlässig mit der improvisierten Waage herausfinden. Mehr als zwei Männer sollten sich allerdings nicht gleichzeitig auf die Waage stellen, sonst könnte sie noch kaputtgehen und das verbietet der Vorarbeiter.

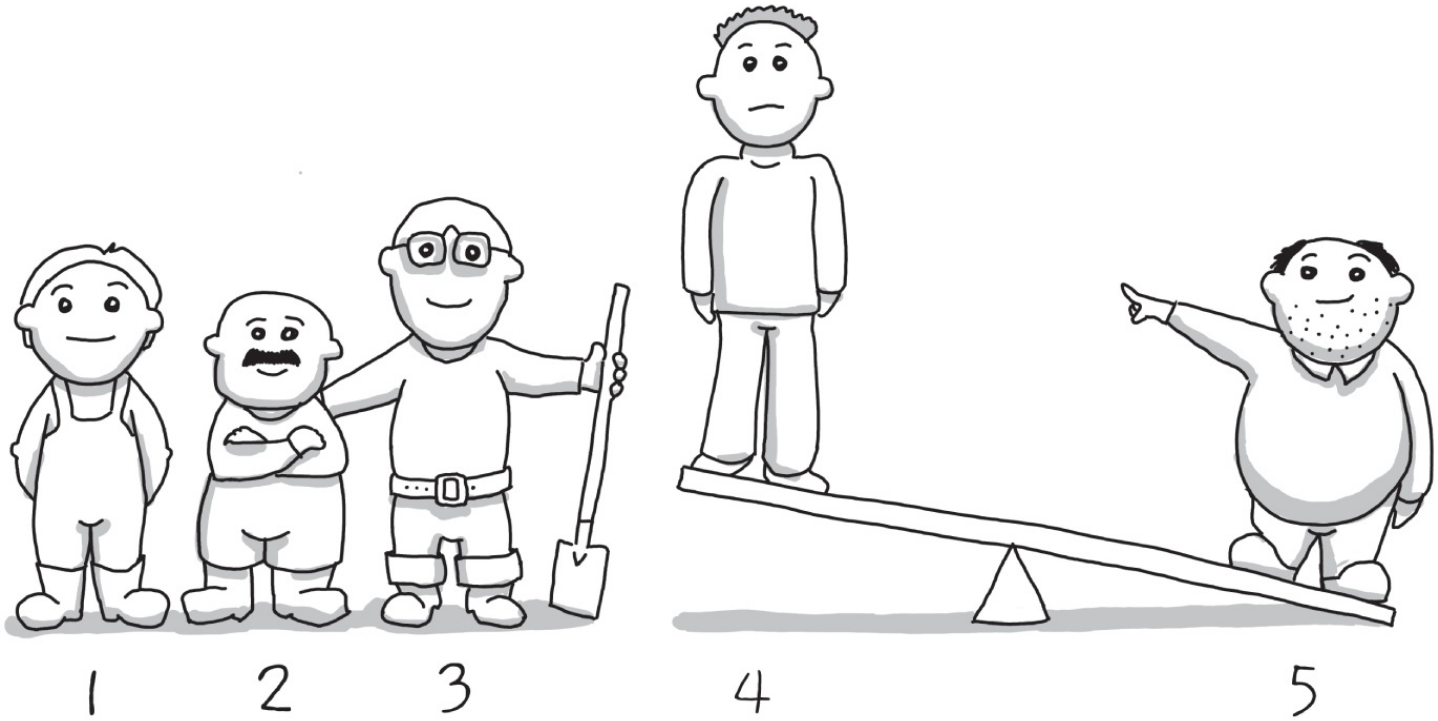


Abbildung 2.2: Bauarbeiter vergleichen ihr Gewicht mit einer Waage.

Problem: den Leichtesten finden

Eingabe: eine Reihe von n Männern

Berechne: die Nummer eines Mannes, der höchstens so viel wiegt wie jeder der anderen Männer in der Reihe

Wie genau sollen die Bauarbeiter nun vorgehen, um den leichtesten – oder wenn es mehre davon gibt, einen der leichtesten – von ihnen herauszufinden? Sie werden nun zwei alternative Lösungsmöglichkeiten für dieses Problem kennenlernen.

Methode 1: Alle nacheinander prüfen

Angenommen, es gibt insgesamt n Bauarbeiter. Damit wir niemanden vergessen, wollen wir die Männer nummerieren: Mann 1, Mann 2, Mann 3 und so weiter. Ob zum Beispiel Mann i der leichteste ist, lässt sich ganz leicht prüfen: Man muss ihn nur auf der Waage mit jedem anderen Mann vergleichen. Sobald sich ein anderer findet, der leichter als Mann i ist, so ist Mann i nachweislich nicht der leichteste und scheidet als Kandidat aus. Findet sich hingegen niemand, der leichter wäre, so gehört Mann i offenbar zu den leichtesten Männern auf der Baustelle. Da wir nicht wissen, welcher der Männer der

leichteste ist, prüfen wir einfach nacheinander alle durch. Der Algorithmus sieht dann insgesamt so aus:

```
MinWeight1(Liste von n Männern)
    for i ← 1, ..., n                    prüfe alle Männer der Reihe nach
        if isLightest(i) then            wenn Mann i der leichteste ist ...
            return i                     ... dann ist man fertig
isLightest(i)                            prüft, ob Mann i der leichteste ist
    for j ← 1, ..., n                    vergleiche Mann i mit allen anderen
        if Mann i ist schwerer als Mann j then
            return false                 Mann i ist nicht der leichteste
    return true                          Mann i ist (einer) der leichteste(n)
```

Dieser Algorithmus führt eine erschöpfenden Suche durch, ein Verfahren, das man auch als »rohe Gewalt« oder *brute force* bezeichnet.



Mehr zum algorithmischen Prinzip der rohen Gewalt können Sie in Kapitel 11 nachlesen.

Methode 2: Der Schwerere scheidet aus

Vermutlich kommt Ihnen die eben beschriebene Methode etwas umständlich vor – und sie haben natürlich recht, es geht auch einfacher: Am Anfang stellt man Mann 1 und Mann 2 auf die Waage, der schwerere von beiden scheidet aus. Der leichtere wird mit Mann 3 verglichen, wieder scheidet der schwerere aus. Der leichtere wird mit Mann 4 verglichen und so weiter. Man geht die Männer also der Reihe nach durch und merkt sich immer den leichtesten, den man bislang gefunden hat. Wenn alle Männer mindestens einmal auf der Waage gestanden haben, bleibt am Ende nur einer übrig und der wiegt dann weniger als alle anderen Männer.

```
MinWeight2(Liste von n Männern)
    m ← 1                                Mann m ist der leichteste bisher bekannte Mann
    for i ← 2, ..., n                    prüfe alle Männer der Reihe nach
        if Mann i ist leichter als Mann m then
            m ← i                        i ist der leichteste Mann, den wir bisher gehen haben
    return m
```

Zu jedem Zeitpunkt speichert man in der Variablen m die Lösung für ein Teilproblem, nämlich die Nummer des Mannes, der unter den ersten i Männern am wenigsten wiegt. Bei jedem Schleifendurchlauf wird i um eins größer, das gelöste Teilproblem also Schritt für Schritt ausgeweitet, bis man am Ende die Lösung für das Gesamtproblem hat, also die Nummer des leichtesten von allen n Bauarbeitern kennt. Diese Nummer – sie steht in m – gibt der Algorithmus am Ende zurück.


Dieser Algorithmus verwendet das Prinzip des »dynamischen Programmierens«. Was man genau man darunter versteht, wird eingehend in Kapitel 13 beschrieben.

Laufzeiten vergleichen

Wenn Sie sich beide Lösungsvorschläge zur Suche nach dem leichtesten Bauarbeiter anschauen, so haben Sie vermutlich den Eindruck, dass die Methode 2 besser als Methode 1 ist. Doch wie genau kann man die beiden Methoden vergleichen? Dafür gibt es verschiedene Möglichkeiten.

Wettlauf der Programme

Man könnte die Algorithmen programmieren und dann die Laufzeiten der verschiedenen Programme auf dem *gleichen* Computer miteinander vergleichen. Das ist der Ansatz der »experimentellen Algorithmik« und es ist die einzige Methode, mit der man unmittelbar praxisnahe Ergebnisse produzieren kann. Allerdings spielen bei modernen Rechnern so viele Faktoren bei dem Ablauf eines Programms eine Rolle, dass selbst bei genausten Messungen jeder Programmdurchlauf eine etwas andere Zeit benötigt. Aus diesem Grund müsste man die Laufzeitmessungen mehrfach wiederholen und mitteln.

Das Ergebnis dieses Vergleichs kann außerdem davon abhängen, welchen Computer man verwendet. Ein Algorithmus mit großem Speicherbedarf könnte zum Beispiel auf einem Rechner mit viel Speicherplatz einen anderen Algorithmus mit geringerem Speicherbedarf überholen, gegen den er auf einem Rechner mit wenig Speicher verloren hätte.

Ein anderes Problem ist, dass die Laufzeit eines Programms natürlich auch von der jeweiligen Implementierung abhängt. Programmiert man die Algorithmen mit unterschiedlicher Geschicklichkeit oder gibt man sich bei dem einen einfach mehr Mühe als bei dem anderen, so kommt man vielleicht zu falschen Schlussfolgerungen über die Qualität der verschiedenen Algorithmen. Letztlich vergleicht man bei dieser Methode nicht die Algorithmen, sondern eben nur Programme.

Laufzeiten ausrechnen

Am besten wäre es natürlich, wenn man direkt ausrechnen könnte, welcher Algorithmus schneller ist. Dazu muss man allerdings von der Komplexität realer Computer absehen und stattdessen die Laufzeit für eine möglichst einfache, abstrakte Modellmaschine berechnen. Ein Beispiel für ein solches Modell wäre die *Random Access Machine* (RAM), über die wir schon in Kapitel 1 gesprochen haben. Für die Laufzeiten könnte man einfache Annahmen machen, wie zum Beispiel: »Jeder Speicherzugriff und jede Rechenoperation benötigt genau eine Zeiteinheit«. Dann müsste man nur die genaue Zahl der Speicherzugriffe und Rechenoperationen ausrechnen – das wäre dann ein Maß für die Rechenzeit.

Allerdings wird selbst bei so simplen Laufzeitmodellen die Analyse oft ziemlich mühselig, da es bei komplexen Algorithmen gar nicht einfach ist, die genaue Anzahl von

Rechenoperationen und Speicherzugriffen zu ermitteln. Der Aufwand für eine exakte Laufzeitberechnung lohnt sich oft nicht, da sich die Ergebnisse ohnehin nicht direkt auf reale Maschinen übertragen lassen. Es hat sich darum unter den meisten Algorithmikern eine andere Methode für den Laufzeitvergleich durchgesetzt, nämlich die »asymptotische Laufzeitanalyse«. Diese Methode werden Sie jetzt kennenlernen.

Asymptotische Laufzeitanalyse

Die Laufzeit eines Algorithmus hängt natürlich auch von der Menge der Daten ab, die er zu verarbeiten hat. Es geht bestimmt schneller, den leichtesten von fünf Bauarbeitern herauszufinden, als wenn es 100 Bauarbeiter wären. Die Menge an Daten, die ein Algorithmus zu verarbeiten hat, nennt man »Eingabelänge«. In unserem Beispiel wäre das zum Beispiel die Zahl n der Bauarbeiter. Bei einer *asymptotischen Laufzeitanalyse* betrachtet man, wie sich die Laufzeit entwickelt, wenn man die Eingabelänge immer größer und größer wählt.

Beim Algorithmus MinWeight2 zum Beispiel wird die **for** -Schleife für jeden Bauarbeiter einmal durchlaufen. Die Zeit, die jeder einzelne Schleifendurchlauf benötigt, hängt ihrerseits nicht von der Anzahl n der Männer ab. Wenn sich die Zahl der Bauarbeiter verdoppelt, so verdoppelt sich also auch ungefähr die Laufzeit des Algorithmus, bei dreifachem n verdreifacht sich die Laufzeit und so weiter. Einen solchen Zusammenhang zwischen der Eingabelänge n und der Laufzeit nennt man *linear*.

Bei MinWeight1 hingegen wird für jeden einzelnen Durchlauf der **for** -Schleife einmal der Algorithmus isLightest aufgerufen. Dort gibt es eine weitere **for** -Schleife, die durch alle n Männer läuft. Die Laufzeit von isLightest hängt also linear von der Eingabelänge n ab. Wenn man Glück hat und Mann 1 der leichteste ist, so ist man nach dem ersten Durchlauf der **for** -Schleife in MinWeight1 fertig und die Laufzeit von MinWeight1 ist somit insgesamt linear. Hat man hingegen Pech und der letzte Mann n in der Reihe ist der leichteste von allen, so wird die **for** -Schleife in MinWeight1 insgesamt n -mal durchlaufen.

Das ungünstigste Beispiel für diesen Algorithmus geht so: Alle Männer wiegen gleich viel, nur der letzte in der Reihe ist ein wenig leichter. isLightest muss in diesem Fall also die gesamte Reihe der Männer durchlaufen, um zu einem Ergebnis zu kommen. Wenn sich nun die Zahl n der Männer verdoppelt, so verdoppelt sich nicht nur der Aufwand für jeden isLightest-Aufruf, sondern auch die *Anzahl* der isLightest-Aufrufe verdoppelt sich; insgesamt hat man also viermal so viel zu tun. Verdreifacht sich n , so steigt der Aufwand sogar auf das Neunfache. Im schlimmsten Fall wächst bei MinWeight1 die Laufzeit also mit der Eingabelänge n um den Faktor n^2. Dies nennt man eine »quadratische« Laufzeit.

Zeichnet man die Laufzeiten der beiden Algorithmen in Abhängigkeit von n auf, so ergibt sich bei MinWeight2 ungefähr eine Gerade, während die Laufzeitkurve bei

MinWeight1 einer Parabel entspricht, die sich nach oben hin krümmt (Abbildung 2.3). Nun wissen wir vielleicht nicht genau, welcher der beiden Algorithmen auf einem bestimmten Computer und bei einer gegebenen Eingabe tatsächlich wie schnell läuft, wo diese Kurven also genau im Koordinatensystem liegen. Sie könnten irgendwo nach oben oder unten gestreckt, gestaucht oder verschoben sein. Doch eins ist klar: Wenn wir in der Grafik weit genug nach rechts gehen, die Eingabelänge n also groß genug wird, so liegt irgendwann die Parabel über der Geraden. Bei genügend großem n steigt die Laufzeit von MinWeight1 stärker an als die Laufzeit von MinWeight2 und schließlich braucht MinWeight1 mehr Zeit als MinWeight2, was dann auch so für alle Eingabelängen $> n$ bleibt.

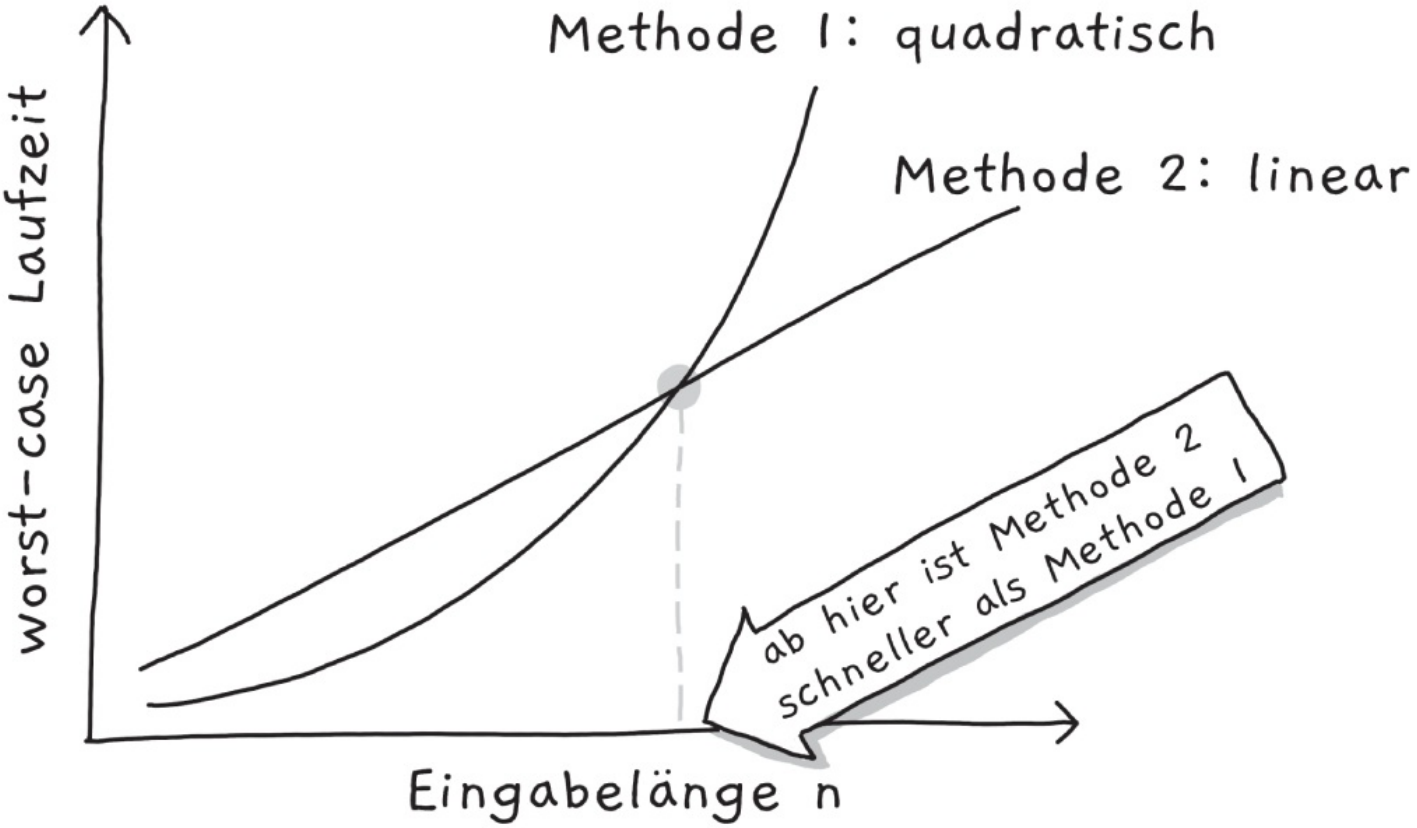


Abbildung 2.3: Laufzeiten der beiden Algorithmen zur Bestimmung des leichtesten Bauarbeiters

Das ist der Trick der asymptotischen Laufzeitanalyse: Da man sich nur auf generelle Entwicklung der Laufzeit bei immer weiter wachsender Eingabelänge n konzentriert, spielen alle technischen Einflüsse realer Computer und Programme keine Rolle mehr und man kann sich ganz auf die theoretische Betrachtung von Algorithmen konzentrieren. In der Praxis ist Methode 2 übrigens immer mindestens so schnell wie Methode 1, aber selbst wenn das anders wäre, müsste man n nur groß genug wählen, damit Methode 2 schneller als Methode 1 wird.

Asymptotische Laufzeitanalysen sind also das Mittel der Wahl und darum werden wir uns in diesem Buch ab sofort vor allem damit beschäftigen.

Laufzeitanalysen

Wenn ein Mathematiker oder eine Mathematikerin von »Abschätzen« spricht, dann meinen sie damit, dass man einen Wert mithilfe von Ungleichungen einzugrenzen versucht. Eine (Worst-Case-)Laufzeit $T(n)$ »nach oben hin abzuschätzen« bedeutet, dass man irgendeinen Wert g (die »Grenze«) findet, für den die Ungleichung $T(n) \leq g$ gilt. Wenn die Laufzeit $T(n)$ von der Eingabelänge n abhängt (darum schreiben wir ja »(n)« hinter das T) und mit steigendem n immer weiter wächst, so wird g zwangsläufig ebenfalls von n abhängen und darum schreiben wir $g(n)$.

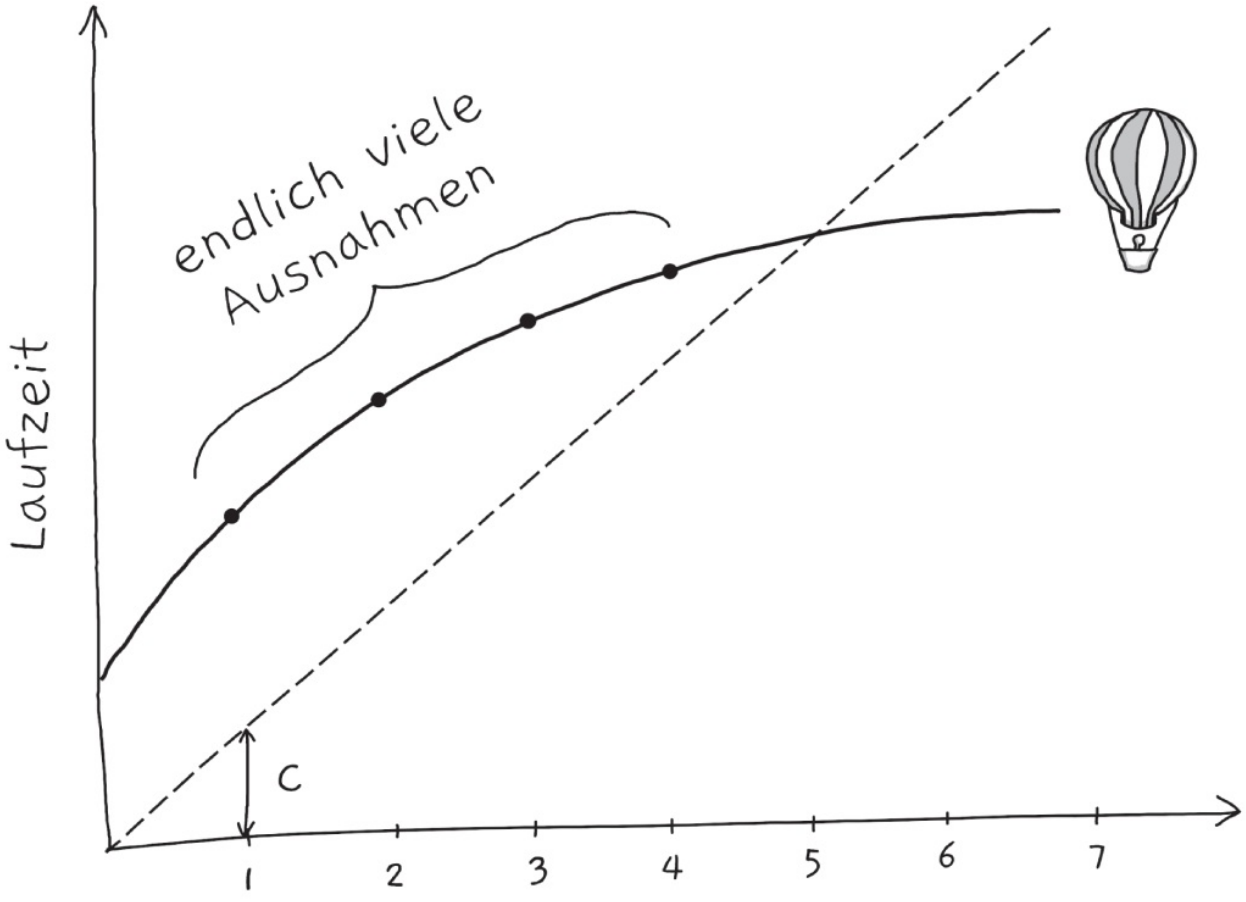
Da es uns bei einer asymptotischen Laufzeitanalyse darum geht, wie sich die Laufzeit $T(n)$ für immer weiter steigende n verhält, ist es allerdings nicht so wichtig, dass die Ungleichung $T(n) \leq g(n)$ buchstäblich für alle Eingabelängen n gilt. Es reicht völlig aus, wenn die Abschätzung für genügend große n funktioniert. Wir gönnen uns sozusagen den Luxus, dass für endlich viele Ausnahmen $T(n) > g(n)$ sein darf, solange es nur eine Eingabelänge n_0 gibt, sodass für alle $n > n_0$ die Abschätzung $T(n) \leq g(n)$ gilt. Da es unendlich viele Zahlen $n > n_0$, aber nur endlich viele Zahlen $n < n_0$ gibt, sagt man übrigens auch, dass $T(n) \leq g(n)$ für *fast alle* Zahlen n gilt.

Lineare Laufzeiten

Eine Laufzeit $T(n)$ steigt *linear*, wenn ihre Kurve ganz gerade verläuft, sich also weder nach oben noch nach unten krümmt. Mathematisch gesehen bedeutet das, dass es zwei Zahlen a und b gibt, für die $T(n) = a \cdot n + b$ ist. Für eine asymptotische Analyse sind die genauen Werte für a und b natürlich wieder egal, tatsächlich würde uns eine Abschätzung völlig ausreichen. Wir sagen darum:

> » $T(n)$ wächst (höchstens) linear, wenn es eine Konstante c gibt mit $T(n) \leq c \cdot n$ für genügend große n .«

Dafür gibt es eine spezielle Schreibweise, nämlich: » $T(n)$ ist $\mathcal{O}(n)$«. In Abbildung 2.4 ist c als die Steigung der gestrichelten Geraden eingezeichnet. Wenn Sie entscheiden wollen, ob eine Laufzeit $T(n)$ nun $\mathcal{O}(n)$ ist oder nicht, dann müssen Sie folgende Fragen beantworten: Können wir c so wählen, dass die Kurve von $T(n)$ irgendwann einmal unter der gestrichelten Gerade $c \cdot n$ liegt, wenn man nur weit genug in der Grafik nach rechts geht?


endlich viele
Ausnahmen
Laufzeit
c
1
2
3
4
5
6
7

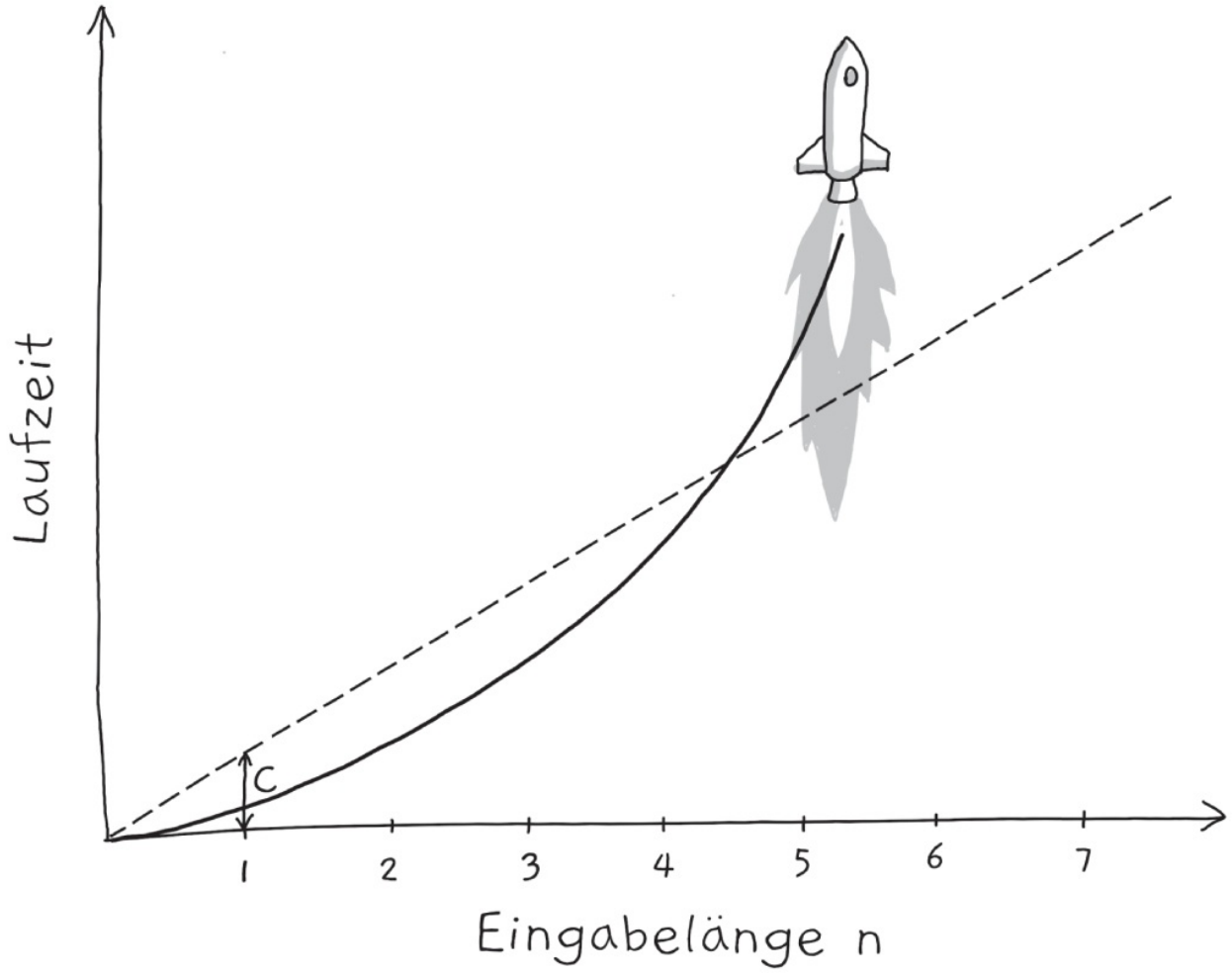

Laufzeit
c
1
2
3
4
5
6
7
Eingabelänge n

Abbildung 2.4: langsamer (oben) und schneller (unten) als linear wachsende Laufzeiten

Für die Bahn des Luftballons zum Beispiel (Abbildung 2.4 rechts) ist das der Fall: Sie ist nach unten gekrümmt, liegt damit für genügend große n unter der Geraden und ist somit $\mathcal{O}(n)$.

Das gleiche würde auch für jede Gerade gelten, denn wir dürfen c ja frei wählen, das heißt, die gestrichelte Gerade kann so steil sein, wie wir wollen. Wenn also $T(n) = a \cdot n + b$ ist, dann könnten wir $c = a + b$ wählen und schon ist $T(n) \leq c \cdot n$, das heißt, $T(n)$ ist $\mathcal{O}(n)$.

Bei der Rakete andererseits (Abbildung 2.4 links) biegt sich die Bahn nach oben. Es spielt keine Rolle, wie groß wir das c wählen, wie steil die gestrichelte Gerade also nach oben zeigt (ganz senkrecht kann sie nicht stehen), es gibt immer einen Schnittpunkt zwischen der Geraden und der Bahn der Rakete und von da an fliegt die Rakete über die Geraden weg. Die Bahn der Rakete ist also nicht $\mathcal{O}(n)$.

Oh du großes $\mathcal{O}$!

Das große $\mathcal{O}$ ist das gebräuchlichste der praktischen *Landau-Symbole*, mit denen Algorithmiker gerne Laufzeitabschätzungen angeben. (Weitere Landau-Symbole finden Sie in Tabelle 2.1). Das $\mathcal{O}$ hat ein paar spezielle Eigenschaften, die Sie immer im Hinterkopf behalten sollten:

Symbol	*Relation*	*Bedeutung*
$\mathcal{O}(f(n))$	$\leq$	»Die Laufzeit wächst höchstens mit $f(n)$«: Es gibt ein c mit $T(n) \leq c \cdot f(n)$ für genügend große n.
$\Omega(f(n))$	$\geq$	»Die Laufzeit wächst mindestens mit $f(n)$«: Es gibt ein c mit $T(n) \geq c \cdot f(n)$ für genügend große n. (Ω = »groß Omega«)
$\Theta(f(n))$	$=$	»Die Laufzeit wächst mit $f(n)$«: $T(n)$ ist $\mathcal{O}(f(n))$ und $T(n)$ ist $\Omega(f(n))$. (Θ = »groß Theta«)
$o(f(n))$	$<$	»Die Laufzeit wächst schwächer als $f(n)$«: Für alle $c > 0$ ist $T(n) < c \cdot f(n)$ für genügend große n.
$\omega(f(n))$	$>$	»Die Laufzeit wächst stärker als $f(n)$«: Für alle $c > 0$ ist $T(n) > c \cdot f(n)$ für genügend große n. (ω = »klein Omega«)

Tabelle 2.1: Landau-Symbole

- ✔ Das $\mathcal{O}$ steht für eine $\mathcal{O}$bere Schranke: Es gibt an, wie sich das Wachstum der Laufzeit $T(n)$ mit steigender Eingabelänge n *höchstens* entwickelt. Selbst wenn die Laufzeit konstant wäre und gar nicht von n abhinge, wäre sie dennoch $\mathcal{O}(n)$.

- Im $\mathcal{O}$ ist eine Konstante c versteckt, auf die man sich nicht weiter festlegt, wenn man die $\mathcal{O}$-Schreibweise verwendet. Wie groß c ist, spielt eigentlich keine Rolle, es reicht aus, dass es irgendein c gibt, für das die Abschätzung $T(n) \leq c \cdot n$ funktioniert. Wenn man einen Algorithmus mit Laufzeit $\mathcal{O}(n)$ hat, dann bedeutet das also nicht, dass er in der Praxis auch schnell läuft, denn dieses c kann beliebig groß sein. Im c steckt auch die Zeiteinheit, mit der wir rechnen müssen. c könnte also zum Beispiel »ein Tag« sein und dann wäre die Laufzeit $T(7)$ für $n = 7$ mit »einer Woche« nach oben abgeschätzt.
- Das c heißt »c«, weil es $c$onstant – eine Konstante – ist, also insbesondere nicht von n abhängt. Damit eine Laufzeit $\mathcal{O}(n)$ ist, muss es *ein* c geben, bei dem die Laufzeit $T(n) \leq c \cdot n$ ist für (fast) alle n. Wenn man c in Abhängigkeit von n wählen dürfte, dann wäre es immer möglich, diese Ungleichung zu erfüllen, alle Funktionen $T(n)$ wären $\mathcal{O}(n)$ und die ganze Definition von $\mathcal{O}$ wäre komplett sinnlos.

Laufzeitklassen

Bisher haben wir uns nur damit beschäftigt, ob eine Laufzeit $T(n)$ in $\mathcal{O}(n)$ liegt, also linear wächst. Das können wir auch verallgemeinern und nach dem gleichen Prinzip andere Arten der Laufzeitentwicklung abschätzen:



Sei $f(n)$ eine Funktion. Wir sagen »$T(n)$ ist $\mathcal{O}(f(n))$«, wenn es eine Konstante c gibt, sodass $T(n) \leq c \cdot f(n)$ für genügend große n.

Für $f(n)$ können wir irgendeine mathematische Funktion einsetzen, die von n abhängt. Zum Beispiel kann man $f(n) = n^2$ wählen; die Worst-Case-Laufzeit von MinWeight1 ist zum Beispiel $\mathcal{O}(n^2)$, nicht aber $\mathcal{O}(n)$.

Je nachdem, welches $f(n)$ man verwendet, kann man eine andere *Laufzeitklasse* $\mathcal{O}(f(n))$ bekommen. Abbildung 2.5 zeigt einige der gängigsten Laufzeitklassen, sortiert nach ihrer Wachstumsstärke.

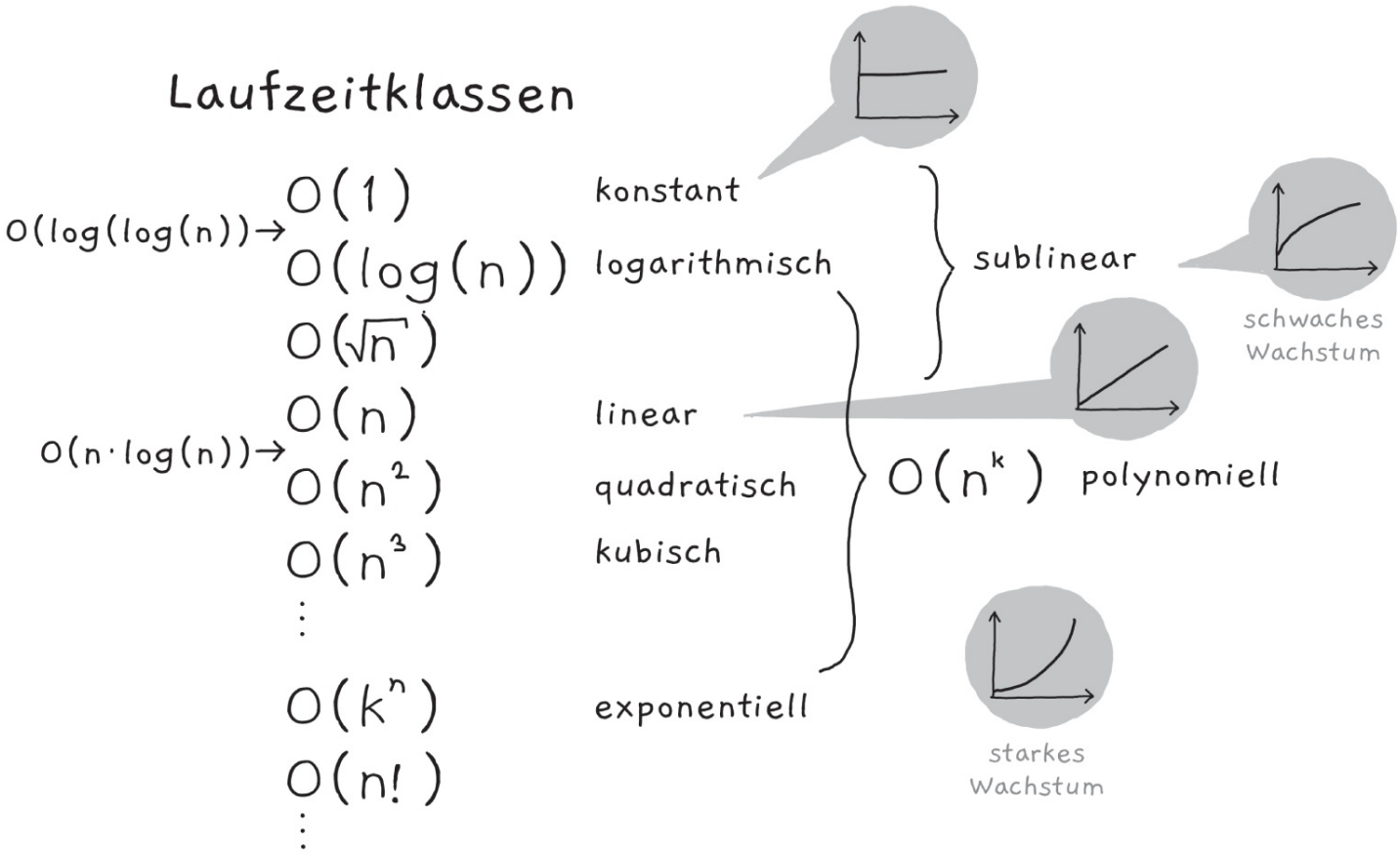


Abbildung 2.5: Gängige Laufzeitklassen



Das Universum der Laufzeitklassen

Ganz oben in Abbildung 2.5 steht $\mathcal{O}(1)$, also die Laufzeitklasse mit der Funktion $f(n) = 1$. Eine Laufzeit $\mathcal{O}(1)$ hängt entweder überhaupt nicht von n ab (»konstante Laufzeit«) oder es gibt zumindest eine feste maximale Laufzeit, die bei keiner Eingabe überschritten wird, ganz egal wie lang sie ist.

Je weiter wir auf der Liste nach unten gehen, desto stärker können die Laufzeiten wachsen. Dabei umfasst jede Laufzeitklasse alle über ihr stehenden Laufzeitklassen. Beispielsweise ist jede Laufzeit in $\mathcal{O}(n)$ auch $\mathcal{O}(n^2)$. Das liegt daran, dass $\mathcal{O}$ eben eine *obere* Schranke ist: Wenn $T(n)$ unter die engere Schranke $\mathcal{O}(n)$ passt, dann erst recht unter die weitere Schranke $\mathcal{O}(n^2)$.

Man könnte die Liste immer weiter nach unten verlängern, denn für jede Laufzeitklasse $\mathcal{O}(f(n))$ gibt es Laufzeiten $T(n)$, die stärker wachsen als $f(n)$ und darum nicht in $\mathcal{O}(f(n))$ liegen. Es gibt also keine »am stärksten wachsende« Laufzeit. Auch bei der Laufzeit von Algorithmen gilt also die alte Regel: »Schlimmer geht immer.«

Zwischen den in Abbildung 2.5 dargestellten Laufzeitklassen kann man außerdem überall noch Zwischenklassen einfügen. Für jedes $k > 0$ gibt es eine eigene Laufzeitklasse $\mathcal{O}(n^k)$, das heißt, zwischen $\mathcal{O}(n^2)$ und $\mathcal{O}(n^3)$ liegen alle $\mathcal{O}(n^k)$ mit $2 < k < 3$, also zum Beispiel $\mathcal{O}(n^{2,5})$. Außerdem liegt zwischen $\mathcal{O}(\log(n))$ und $\mathcal{O}(1)$ zum Beispiel $\mathcal{O}(\log(\log(n)))$, zwischen $\mathcal{O}(\log(\log(n)))$ und $\mathcal{O}(1)$ liegt $\mathcal{O}(\log(\log(\log(n))))$ und so weiter.

Auch zwischen der konstanten Laufzeit $\mathcal{O}(1)$ und der linearen Laufzeit $\mathcal{O}(n)$ gibt es eine Vielzahl unterschiedlicher Laufzeitklassen, die man unter dem Begriff »sublinear« zusammenfasst. Algorithmen, die in diese Laufzeitklassen fallen, haben nicht einmal genug Zeit, um die Eingabe der Länge n zu lesen.

Übrigens kann man bei Logarithmen hinter dem $\mathcal{O}$ die Basis weglassen. Statt $\mathcal{O}(\log_2 n)$ oder $\mathcal{O}(\log_{10} n)$ schreibt man einfach $\mathcal{O}(\log n)$, denn Logarithmen verschiedener Basis unterscheiden sich ja nur durch konstante Faktoren voneinander und konstante Faktoren spielen hinter dem $\mathcal{O}$ bekanntlich keine Rolle.

Arten der Laufzeitanalyse

Laufzeiten von Algorithmen hängen oft nicht nur von der *Länge* der Eingabe ab, sondern gewöhnlich auch von der Eingabe selbst. Steht zum Beispiel der leichteste Bauarbeiter am Anfang der Liste, so findet MinWeight1 ihn sofort, kommt er hingegen als letzter in der Reihe vor, so dauert die Suche deutlich länger. Je nach Gemüt und Zielsetzung kann man sich dann bei der Laufzeitanalyse auf verschiedene Aspekte konzentrieren.

worst case

Die häufigste Art der Laufzeitanalyse ist die Betrachtung des *worst case*, also des schlimmsten Falls. Genauer gesagt: Für jede mögliche Eingabelänge n betrachtet man die Laufzeit $T(n)$, die der Algorithmus bei Eingaben dieser Länge *maximal* benötigt. Beim Algorithmus MinWeight1 im Abschnitt *Methode 1: Alle nacheinander prüfen* wäre das zum Beispiel eine Eingabe, bei der alle Männer genau das Gleiche wiegen, bis auf Mann n, den letzten in der Reihe, der ein wenig leichter ist als die anderen. In diesem Fall läuft die Schleife in MinWeight1 n-mal durch. Und auch die Schleife in isLightest wird bei jedem der n Aufrufe dieses Algorithmus volle n-mal durchlaufen. Laufzeittechnisch ist das das Schlimmste, was passieren kann.



Warum interessieren sich die Algorithmiker eigentlich immer so sehr für den *worst case*? Kann man nicht auch einmal etwas optimistischer sein und sich zum Beispiel den besten Fall, den *best case*, anschauen? Sicher kann man das, allerdings ist das oft wenig interessant. Oft ist es nämlich einfach, seinem Algorithmus eine ganz hervorragende Best-Case-Laufzeit zu verpassen. Beispielsweise würde jeder Sortieralgorithmus aus Kapitel 7 eine optimale Best-Case-Laufzeit bekommen, wenn man ihn vor dem Sortieren einmal kurz über die Eingabe laufen lässt, um zu prüfen, ob sie nicht zufällig bereits sortiert ist. Dass ein Algorithmus in Ausnahmefällen besonders schnell ist, sagt aber noch lange nichts darüber aus, wie er generell mit der Lösung seiner Aufgabe zurechtkommt. Die Worst-Case-Laufzeit ist dafür gewöhnlich ein viel besseres und faireres Maß.

average case

Manche Algorithmen wie der Sortieralgorithmus QuickSort (Kapitel 7) laufen meistens sehr schnell und nur in seltenen Ausnahmefällen vergleichsweise langsam. Bei diesen Algorithmen bekäme man bei einer reinen Worst-Case-Analyse einen falschen Eindruck von der Qualität des Algorithmus. Für solche Algorithmen bietet es sich an, die Laufzeit in einem durchschnittlichen Fall (*average case*) zu betrachten. Dazu muss man allerdings klären, was ein »durchschnittlicher Fall« genau sein soll. Das ist nämlich in der Praxis oft

gar nicht so einfach, insbesondere dann, wenn die Laufzeit von der Eingabe abhängt. Welche Eingaben »durchschnittlich« oder »typisch« sein sollen, lässt sich aus algorithmischer Sicht nämlich nur schwer zu sagen. Relevant ist diese Art der Laufzeitanalyse darum insbesondere bei randomisierten Algorithmen, also bei Algorithmen, die in ihrem Ablauf den Zufall mit einbeziehen. Mit ein wenig Wahrscheinlichkeitsrechnung ist es hier nämlich oft möglich, den *Erwartungswert* der Laufzeit abzuschätzen. Im Fall von QuickSort sieht dieser Erwartungswert mit $\mathcal{O}(n \log n)$ deutlich günstiger aus als die engste Worst-Case-Laufzeitabschätzung $\mathcal{O}(n^2)$.



Ein Beispiel für eine randomisierte Laufzeitanalyse finden Sie weiter unten in diesem Kapitel.

QuickSort wird mit einer Reihe anderer Sortieralgorithmen in Kapitel 7 erklärt.

amortisiert

Eine weitere Form der durchschnittlichen Laufzeitbestimmung ist die amortisierte Analyse. Ähnlich wie die Average-Case-Analyse wird auch sie vor allem relevant, wenn der *worst case* nur sehr selten eintritt und der Algorithmus in den meisten Fällen mit einer deutlich kürzeren Laufzeit auskommt. Eine amortisierte Laufzeitanalyse wendet man an, wenn sich bei einem Algorithmus, der wiederholt aufgerufen wird, die teureren Aufrufe so mit den billigeren Aufrufen verrechnen lassen, dass auf lange Sicht insgesamt eine niedrige Laufzeit herauskommt. Wenn ein Aufruf einmal länger dauert, so kann man dies sozusagen als »Investition« betrachten, mit der man dafür sorgt, dass die nächsten Aufrufe wieder schneller gehen werden. Die Average-Case-Analyse eines randomisierten Algorithmus berechnet ja lediglich einen Erwartungswert für die Laufzeit, es ist also möglich, dass man Pech hat und jeder einzelne Aufruf eine Worst-Case-Laufzeit hat. Im Gegensatz dazu garantiert eine amortisierte Laufzeitanalyse, dass die durchschnittliche Laufzeit pro Aufruf unterhalb der angegebenen Schranke bleibt, und ist damit sozusagen krisensicher.

Im Laufe des Kapitels werden wir noch einmal auf die amortisierte Analyse zurückkommen.

Laufzeiten konkret bestimmen

Nachdem Sie nun so viel Generelles über Laufzeitanalysen erfahren haben, bleibt vor allem eine Frage zu klären: Wie genau kann man bei einem Algorithmus erkennen, zu welcher Laufzeitklasse er gehört? Leider gibt es dafür kein Patentrezept, denn manchmal können Algorithmen mit – dem Anschein nach – minimalen Unterschieden völlig verschiedene Laufzeiten haben.



Der erste und wichtigste Schritt, um die Laufzeit eines Algorithmus zu bestimmen, besteht darin, ihn zu verstehen. Erst wenn man genau erkannt hat, was ein Algorithmus tut, vermag man seine Laufzeit abzuschätzen und auch zu begründen, warum der Algorithmus diese Laufzeit hat.

In dieser Hinsicht verhält es sich mit der Bestimmung der Laufzeit ähnlich wie mit der Untersuchung seiner Korrektheit: Manchmal ist dafür ein komplizierter Beweis nötig, und einen Beweis kann natürlich nur jemand führen, der auch tatsächlich weiß, wovon er redet.

Wenn Sie nun also Ihren Algorithmus verstanden haben, bleibt immer noch die Frage: Wie die Laufzeit abschätzen? Dafür lernen Sie nun ein paar Faustregeln kennen.

Faustregel 1: Bei Schleifen muss man multiplizieren

Angenommen, die Eingabelänge ist n, und in unserem Algorithmus durchläuft eine Schleife einmal alle Zahlen von 1 bis n. Dann wächst die Anzahl der Schleifendurchläufe linear mit der Eingabelänge. Wenn nun bei jedem Schleifendurchlauf unabhängig von n immer die gleiche Zahl an Operationen durchgeführt wird, jeder einzelne Schleifendurchlauf also eine Laufzeit von $\mathcal{O}(1)$ hat, so hat die gesamte Schleife – also alle Schleifendurchläufe zusammen – eine Laufzeit von $\mathcal{O}(n)$. Wenn andererseits die Laufzeit jedes einzelnen Schleifendurchlaufs von n abhängt, also zum Beispiel die Laufzeit $\mathcal{O}(f(n))$ hat, dann entsteht durch die n-fache Wiederholung also eine Laufzeit von insgesamt $\mathcal{O}(n \cdot f(n))$:

```
for i ← 1, ..., n                    Schleife wird n-mal durchlaufen: Laufzeit O(n · f(n))
    tue etwas in O(f(n))
```

Die Gesamtlaufzeit einer Schleifen erhält man also, wenn man die Anzahl der Schleifendurchläufe mit der Laufzeit pro Durchlauf multipliziert.

Manchmal brauchen die Durchläufe von Schleifen unterschiedlich lange, das heißt, die Laufzeit pro Schleifendurchlauf hängt nicht nur von n ab, sondern auch zum Beispiel von der Laufvariablen i. In diesem Fall ist $\mathcal{O}(n \cdot f(n))$ zwar immer noch eine gültige obere Abschätzung für die Gesamtlaufzeit der Schleife; es kann allerdings sein, dass die Laufzeit auch in einer weniger stark wachsenden Laufzeitklasse liegt. Für eine genauere Laufzeitanalyse hilft in diesem Fall wieder nur eigenes Nachgrübeln weiter.

Die Faustregel lässt sich sehr gut auf geschachtelte Schleifen anwenden. Beim Algorithmus MinWeight1 weiter vorn in diesem Kapitel wird in jedem Schleifendurchlauf der Algorithmus isLightest aufgerufen, der die Laufzeit $\mathcal{O}(n)$ hat, und darum hat MinWeight1 eine Laufzeit von $\mathcal{O}(n \cdot n) = \mathcal{O}(n^2)$. Die Schleife von isLightest ist in die Schleife von MinWeight1 »geschachtelt« und bei geschachtelten Schleifen multiplizieren sich die Laufzeiten. Das geht auch bei tieferen »Schachtelungen«,

hier ein Beispiel:

```
for i ← 1, ..., n                    äußere Schleife
    for j ← 1, ..., n                mittlere Schleife
        for k ← 1, ..., n            innere Schleife
            tue etwas in O(1)
```

Die äußere Schleife von DreifachGeschachtelt wird n -mal durchlaufen, die mittlere Schleife also insgesamt $n \cdot n = n^2$-mal und die innerste Schleife $n^2 \cdot n = n^3$-mal (siehe Abbildung 2.6). Da jeder Durchlauf der innersten Schleife die Zeit $\mathcal{O}(1)$ braucht, ist die Gesamtlaufzeit des Algorithmus $\mathcal{O}(n^3)$.

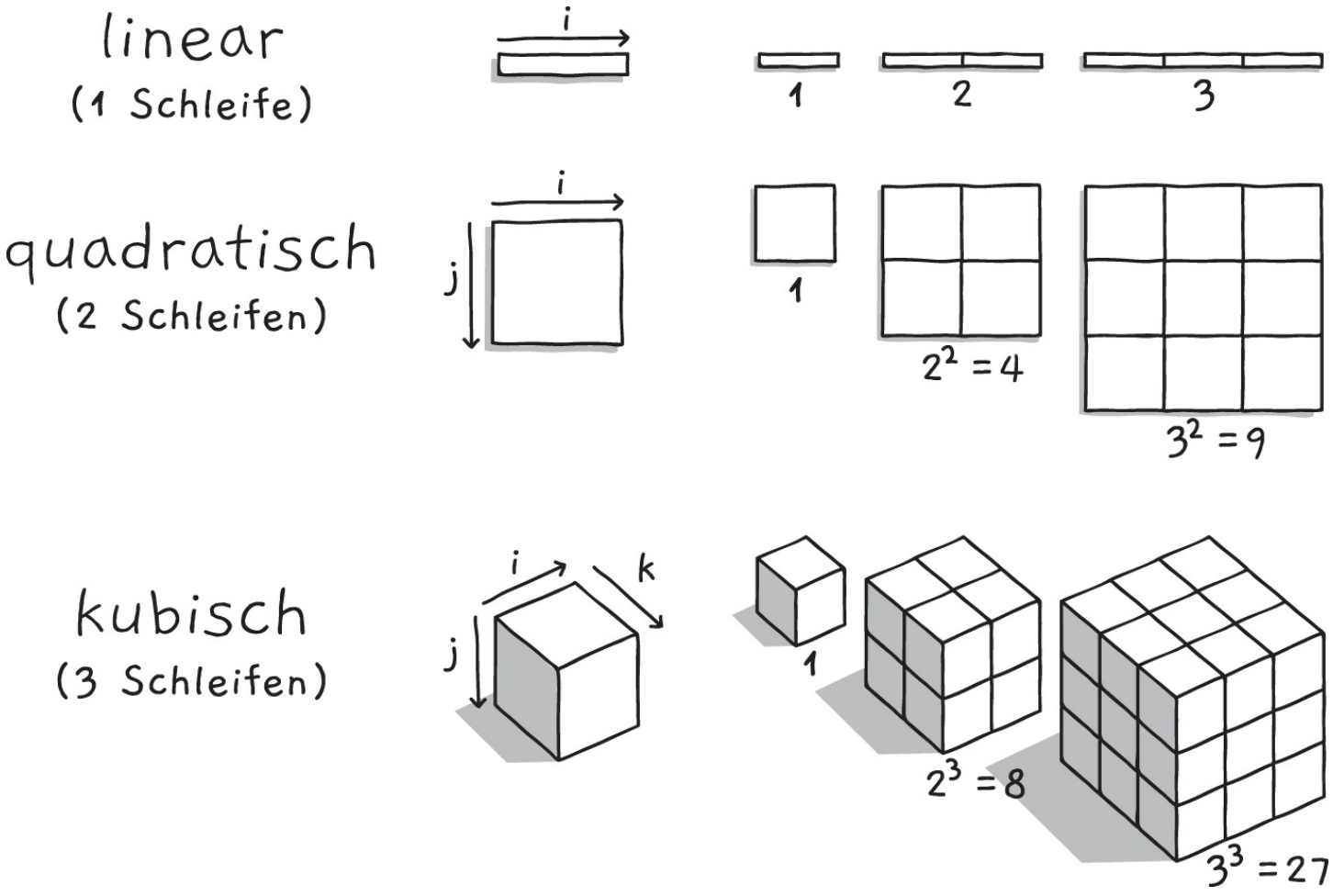


Abbildung 2.6: Eine, zwei und drei geschachtelte Schleifen

Anders sieht es aus, wenn die zwei Schleifen nicht ineinander geschachtelt sind, sondern hintereinander stehen. In diesem Fall muss man die Laufzeiten nicht multiplizieren, sondern addieren. So ist das immer bei Algorithmenteilen, die nacheinander ausgeführt werden: Ihre Laufzeiten werden zur Gesamtlaufzeit zusammengezählt. Am Ende hat man eine Summe von unterschiedlichen Laufzeiten, die man dann oft noch vereinfachen kann, und zwar nach der folgenden Faustregel.

Faustregel 2: Der stärkste Summand dominiert

Angenommen, Ihr Algorithmus besteht aus drei Teilen: zuerst speichert der Algorithmus die Eingabedaten in einer Datenstruktur, dann durchläuft er die Datenstruktur und sammelt alle relevante Daten zusammen und am Ende berechnet er daraus ein Ergebnis und gibt es zurück:

```
IhrAlgorithmus(Daten)
    lade Daten in Datenstruktur      Laufzeit sei O(n · log n)
    durchsuche Datenstruktur         Laufzeit sei O(n²)
    berechne Ergebnis                Laufzeit sei O(n)
```

Wenn für Teil 1 eine Laufzeit von $\mathcal{O}(n \cdot \log n)$, für Teil 2 eine Laufzeit von $\mathcal{O}(n^2)$ und für Teil 3 eine Laufzeit von $\mathcal{O}(n)$ benötigt wird, wie kann man dann die Gesamtlaufzeit abschätzen? Da Ihr Algorithmus alle drei Dinge nacheinander tut, wird er insgesamt eine Laufzeit von $\mathcal{O}(n \cdot \log n + n^2 + n)$ haben. Diese Laufzeitklasse sieht komplizierter aus, als sie ist. Wie ein Blick auf Abbildung 2.5 zeigt, wächst die Laufzeitklasse $\mathcal{O}(n^2)$ für Teil 2 von allen drei Teilen des Algorithmus am stärksten. Für alle $n \geq 1$ ist sowohl $n \cdot \log n \leq n^2$ als auch $n \log n \leq n^2$ und darum kann man die Gesamtlaufzeit genauso gut auch mit $\mathcal{O}(n^2)$ abschätzen. Tatsächlich sind beide Laufzeitklassen identisch: Jede Laufzeit, die in $\mathcal{O}(n \cdot \log n + n^2 + n)$ liegt, ist auch in $\mathcal{O}(n^2)$ und umgekehrt ist jede Laufzeit in $\mathcal{O}(n^2)$ auch in $\mathcal{O}(n \cdot \log n + n^2 + n)$. Wenn wir in unserer Laufzeitabschätzung also eine (endliche) Summe stehen haben, so können wir getrost alle Summanden wegstreichen bis auf den, der am stärksten wächst (Abbildung 2.7). Für genügend große n wird dieser Summand nämlich größer als alle anderen sein und reicht darum als Abschätzung vollkommen aus.

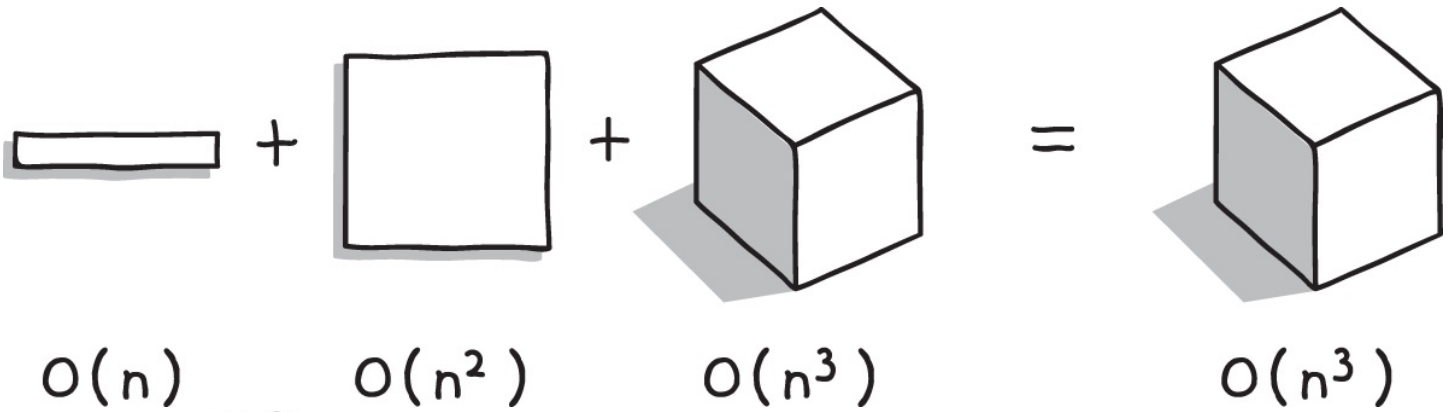


Abbildung 2.7: $\mathcal{O}(n^3)$ dominiert die Summe.

Vorsicht vor versteckten Kosten

Wer Pseudocode verwendet, muss dabei immer im Hinterkopf behalten, für welches Maschinenmodell er seinen Algorithmus formuliert, denn letztlich muss sich ja jeder Schritt des Pseudocodes irgendwie in Basisoperationen dieses Maschinenmodells übersetzen lassen. Das ist auch bei der Bestimmung der Laufzeit wichtig, denn natürlich

hängt diese ebenfalls vom Maschinenmodell ab. Wir verwenden hier die *Random Access Machine*, die auf jede ihrer Speicherzellen in konstanter Zeit zugreifen kann. Eine Turingmaschine kann das zum Beispiel nicht, da sie ja nur über einen einzelnen Schreibe-/Lesekopf verfügt, den sie immer erst umständlich über das Band bewegen muss, bevor sie an eine entfernte Speicherzelle herankommt. Was im Pseudocode nach einem einzelnen Schritt aussieht, könnte je nach Maschinenmodell eine Laufzeit erfordern, die nicht mehr einfach mit einer Konstanten abgeschätzt werden kann. Darum Vorsicht vor versteckten Kosten!

Betrachten Sie zur Illustration folgenden Algorithmus, mit dem aus einer Menge T von Wörtern dasjenige gesucht werden soll, das im Lexikon am weitesten vorne stehen würde:

```
LexicalFirst(T)                    T sei eine nicht leere Menge von n Wörtern
    f ← irgendein Element aus T    f: lexikalisch erstes Wort in T
    for s ← T                      durchlaufe alle Wörter in T
        if s steht im Lexikon vor f    wenn s vor f steht ...
            f ← s                      ... dann merke dir s
    return f
```

Der Algorithmus sieht nach einer schlichten Variante von MinWeight2 aus und die Laufzeit scheint auf den ersten Blick klar zu sein: Es gibt nur eine Schleife, die n -mal durchlaufen wird, also ist die Laufzeit $\mathcal{O}(n)$.

Aber Moment mal! In der Zeile

```
if s steht im Lexikon vor f
```

gibt es einen lexikalischen Vergleich zwischen zwei Wörtern und der läuft auf einer RAM keinesfalls in konstanter Zeit ab, sondern hängt von der Länge der beteiligten Wörter s und f ab. Ein Wort kann ja beliebig lang sein und passt darum nicht in eine einzelne Speicherzelle der RAM hinein, sondern muss – zum Beispiel Buchstabe für Buchstabe – auf mehrere Speicherzellen verteilt werden. Ein lexikalischer Vergleich würde dann zum Beispiel eine Schleife enthalten, die im schlimmsten Fall komplett über alle Buchstaben der beiden Wörter laufen muss, bevor er feststellen kann, welches Wort tatsächlich lexikalisch vor dem anderen kommt.

Wenn also T aus n Wörtern besteht, die alle n lang sind, dann wäre die eigentliche Laufzeit des Algorithmus auf einer RAM also $\mathcal{O}(n^2)$. Bezogen auf die Eingabelänge wäre das übrigens dennoch eine Linearzeit, denn die Eingabe ist ja insgesamt $n \cdot n = n^2$ Zeichen lang.

Randomisierte Laufzeitanalyse

Die meisten Algorithmen, die in diesem Buch vorkommen, laufen für jede Eingabe nach ganz festen Regeln ab. Manchmal kann es aber auch nützlich sein, in Algorithmen den

Zufall mit einzubauen. Algorithmen, die so etwas tun, werden *randomisiert* genannt. Sie werden gleich ein Beispiel für einen randomisierten Algorithmus kennenlernen und anschließend sehen, wie man die Laufzeit eines solchen Algorithmus ermittelt.



In der Informatik arbeitet man meistens nicht mit »echtem« Zufall, sondern mit sogenannten »Pseudozufallszahlen«. Das sind Zahlenfolgen, die den Anschein von Zufälligkeit erwecken, obwohl sie nach festen Regeln berechnet werden. Je nach Anwendung braucht man Pseudozufallszahlen unterschiedlicher Qualität. Eine sichere Verschlüsselung beispielsweise stellt hohe Anforderungen an die verwendeten Zufallszahlengeneratoren. Die meisten randomisierten Algorithmen sind hingegen weit weniger anspruchsvoll und laufen in der Praxis hinreichend gut mit ganz simpel berechneten Zufallszahlen.

Das Auswahlproblem

Wie man am schnellsten aus einer Menge mit sagen wir mal n Elementen das Minimum ermittelt, haben wir bereits am Beispiel der Bauarbeiter gesehen: Man muss nur einmal die Menge durchlaufen und sich immer den kleinsten Wert (beziehungsweise den leichtesten Bauarbeiter) merken, den man bisher gesehen hat. Diesmal wollen wir aber nicht das Minimum suchen, sondern zum Beispiel das zweitkleinste oder drittkleinste – allgemein das k -kleinste Elemente aus der Menge für irgendein k zwischen 1 und n . Um uns das Problem ein wenig zu erleichtern, legen wir fest, dass alle n Element in M unterschiedlich sein sollen.

Problem: das Auswahlproblem

Eingabe: eine Menge M mit n unterschiedlichen Elementen und eine Zahl $k \in \{1, \ldots, n\}$

Berechne: das k -kleinste Element in M

Ein einfache Lösung des Auswahlproblems ginge so:

»Speichere die Elemente in einer Liste. Sortiere die Liste aufsteigend. Gib das k -te Element aus der sortierten Liste zurück.«

Das Thema »Sortieren« wird ausführlich in Kapitel 7 behandelt. Dort wird auch gezeigt, dass die schnellsten Methoden zum (vergleichsbasierten) Sortieren von n Werten eine Laufzeit von $\mathcal{O}(n \cdot \log n)$ haben. Das ist dann auch die Laufzeit des eben skizzierten Algorithmus zur Lösung des Auswahlproblems. Wie das schneller gehen kann, sehen Sie jetzt.

QuickSelect: Ein randomisierter Algorithmus

Zur Lösung des Auswahlproblems gibt es einen einfachen randomisierten Algorithmus namens QuickSelect. Der wählt zunächst zufällig ein Element x aus M aus. Genau hier kommt also der Zufall ins Spiel: Jedes Element hat die gleiche Chance, ausgewählt zu werden, und welches Element tatsächlich ausgewählt wird, entscheidet der Zufall. Die restliche Menge wird dann in zwei Teile geteilt: Zum einen in die Menge $M_<$, die alle Elemente kleiner als x enthalten soll, und zum anderen die Menge $M_>$, in die wir alle Elemente größer als x hineinpacken. Dann ermittelt man, wie viele Elemente in M kleiner oder gleich x sind. Wenn das zufällig gerade k sind, ist x das gesuchte Element und wir sind mit der Suche fertig. Ansonsten müssen wir entweder in $M_<$ oder in $M_>$ weitersuchen, je nachdem, ob sich das gesuchte Element in der einen oder in der anderen Menge befindet.

```
QuickSelect(M, k)                    M: Menge mit n unterschiedlichen Elementen, k: Zahl
    x ← ein zufällig ausgewähltes Element aus M
    M_< ← {y ∈ M | y < x}            M_<: Menge der Elemente < x
    M_> ← {y ∈ M | y > x}            M_>: Menge der Elemente > x
    count_≤ ← |M_<| + 1              count_≤: Anzahl der Elemente ≤ x
    if k = count_≤                   wenn das k-kleinste Element gefunden wurde:
        return x                     fertig!
    if k < count_≤                   wenn das k-kleinste Element < x ist, ...
        return QuickSelect(M_<, k)   ... rekursiv in M_< suchen
    else                             ansonsten ist das k-kleinste Element > x, ...
        return QuickSelect(M_>, k − count_≤)   ... rekursiv in M_> suchen
```

Wir haben den Algorithmus hier rekursiv formuliert. Genauer gesagt: Er ist endrekursiv und ließe sich darum auch leicht in einen iterativen Algorithmus umwandeln.



Mehr zu (end-)rekursiven Algorithmen finden Sie zum Beispiel in den Kapitel 3 und Kapitel 12.

QuickSelect ist nah mit dem Sortieralgorithmus QuickSort verwandt, über den Sie in Kapitel 7 mehr erfahren können. Der Algorithmus arbeitet intensiv auf Mengen herum; wie man Mengen effizient abspeichern kann, wird in Kapitel 9 erklärt.

Worst-Case-Laufzeit

Sieht man einmal von den rekursiven Aufrufen am Ende des Algorithmus ab, benötigt ein einzelner Durchlauf von QuickSelect eine Laufzeit von $\mathcal{O}(n)$. Diese Laufzeit wollen wir $T_{first}(n)$ nennen. Sie ist linear in n, da alle Elemente von M durchlaufen werden müssen, um M in die beiden Teilmengen $M_\leq$ und $M_>$ aufgespaltet werden können.

Wenn sich der Algorithmus am Ende rekursiv selbst aufruft, kann das zu einer höheren Laufzeit führen. Im ungünstigsten Fall sucht man zum Beispiel mit $k = 1$ das Minimum

in M, wählt aber zufällig gerade das größte Element in der Menge aus. Im nächsten rekursiven Aufruf müssen wir dann in einer Menge weitersuchen, die nur um ein einzelnes Element kleiner ist als die Menge zuvor. Erwischt man dann wieder das Maximum, so wird die Menge ebenfalls nur um eins kleiner, und wenn das immer so weitergeht, hat man QuickSelect am Ende n -mal aufgerufen. Daraus ergibt sich eine Worst-Case-Laufzeit von $\mathcal{O}(n^2)$.

Average-Case-Laufzeit

Der *worst case* ist aber nur ein möglicher Fall von vielen und er ist noch nicht einmal sehr wahrscheinlich. Niemand hat immer nur Pech. Die Worst-Case-Analyse gibt uns in diesem Fall also kein faires Bild vom typischen Laufzeitverhalten des Algorithmus. Darum ist es sinnvoll, stattdessen die *durchschnittliche* Laufzeit zu betrachten, oder wie man in der Wahrscheinlichkeitsrechnung sagt: Den *Erwartungswert* der Laufzeit. Im Fall von QuickSelect stellt sich heraus, dass die erwartete Laufzeit $\mathcal{O}(n)$ ist. Details dazu finden Sie im Kasten.



Erwartete Laufzeit ausgerechnet

Die erwartete Laufzeit von QuickSelect bei Eingabe einer Menge M mit n Elementen wollen wir $E(n)$ nennen. $E(n)$ besteht zum einen aus $T_{first}(n)$ und zum anderen aus der erwarteten Laufzeit für die restliche Suche, die wir E_{rest} nennen wollen.

E_{rest} hängt davon ab, welches x wir erwischt haben. Lassen Sie uns die Elemente der Menge je nach ihrer Größe in vier Viertel einteilen: Das Viertel mit den kleinsten Elementen wird »erstes Quartil« (Q1) genannt, das Viertel mit den größten Elementen »viertes Quantil« (Q4), die beiden restlichen Viertel heißen »zweites Quartil« (Q2) und »drittes Quartil« (Q3), siehe Abbildung 2.8. Nun gibt es zwei Fälle:

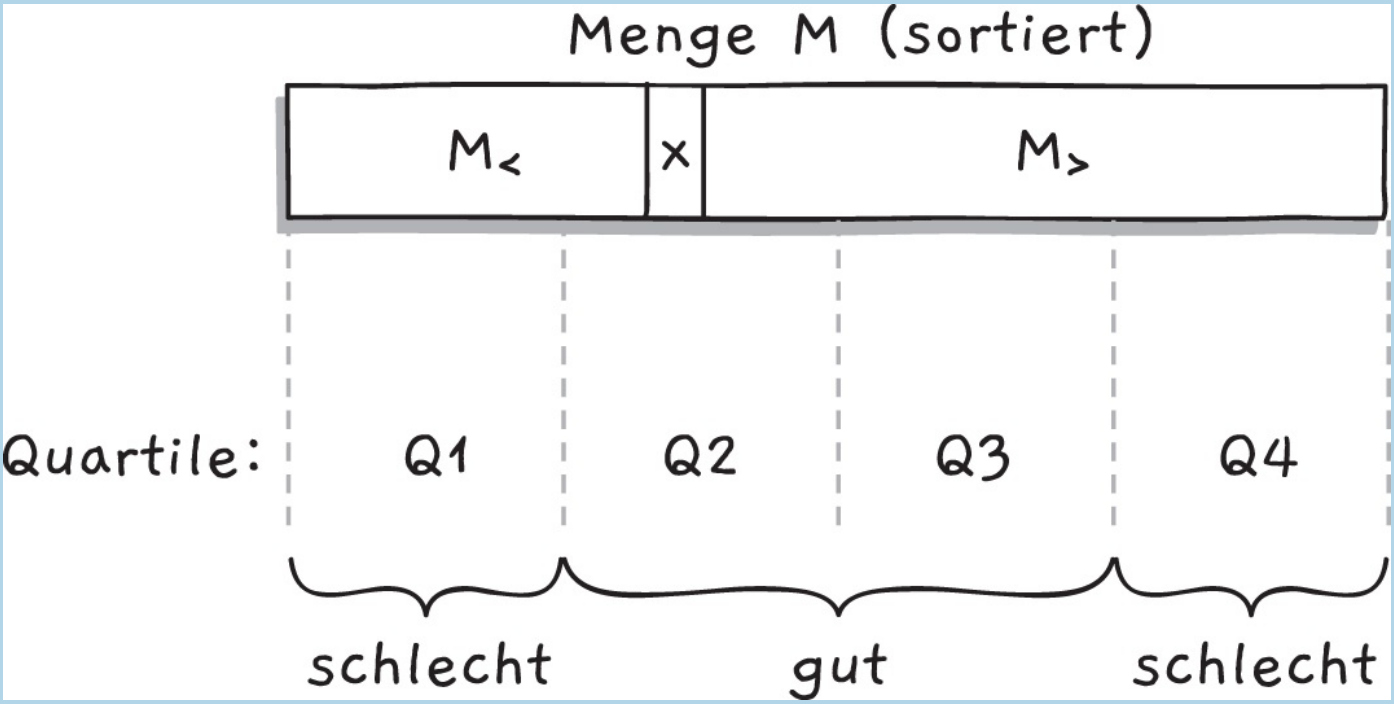


Abbildung 2.8: Je nach Quartil ist die Wahl von x unterschiedlich günstig.

1. Wenn unser x in Q2 oder Q3 liegt, dann haben wir Glück gehabt, denn dann bestehen die Mengen $M_<$ und $M_>$ jeweils aus mindestens einem Viertel der Elemente aus M, und egal in welcher der beiden Mengen wir weitersuchen müssen, die Elemente der jeweils anderen Menge fallen bei der weiteren Suche schon einmal weg. E_{rest} ist in diesem Fall also $\leq E(\frac{3}{4}n)$.
2. Wenn hingegen x in Q1 oder in Q4 liegt, dann kann es sein, dass wir Pech haben und bei der weiteren Suche weniger als ein Viertel der Elemente wegfallen. Immerhin wird das Problem dadurch nicht größer, also ist $E_{rest} \leq E(n)$.

Beide Fälle sind gleich wahrscheinlich, darum können wir E_{rest} mit dem Durchschnitt von $E(\frac{3}{4}n)$ und $E(n)$ abschätzen:

$$E_{rest} \leq \frac{E(\frac{3}{4}n) + E(n)}{2}$$

Für $E(n)$ folgt daraus:

$$\begin{aligned} E(n) &= T_{first}(n) + E_{rest} \\ &\leq T_{first}(n) + \frac{E(\frac{3}{4}n) + E(n)}{2} \end{aligned}$$

Löst man dies nach $E(n)$ auf, so erhält man:

$$E(n) \leq E(\frac{3}{4}n) + 2 \cdot T_{first}$$

Es gibt verschiedene Methoden, derartige Rekursions-(un-)gleichungen aufzulösen. Wir können es uns hier einfach machen und das Mastertheorem aus Kapitel 12 anwenden. Im vorliegenden Fall 1 des Mastertheorems (»der Chef arbeitet mehr als seine Mitarbeiter zusammen«) hat $E(n)$ die Laufzeitklasse des »Zusatzaufwands« $2 \cdot T_{first}$ und die ist $\mathcal{O}(n)$.

Amortisierte Laufzeitanalyse

Manchmal ist der *worst case* auch bei nicht-randomisierten Algorithmen selten und sein Auftreten geht in der Menge der guten Fälle unter und wird von ihnen systematisch ausgeglichen. Über solche Algorithmen und wie man ihre Laufzeit analysiert, soll es im Folgenden gehen.

Ein Binärzähler an der Fassade

Am Gebäude des Instituts für Mathematik prangt in großen Lettern die Kreiszahl π (sprich: »pi«), jedenfalls die ersten paar hundert Ziffern davon. Da möchten die Informatiker natürlich nicht zurückstehen: Ab sofort soll am Informatikgebäude die Zahl der Tage seit Gründung des Instituts stehen, und zwar stilecht als Binärzahl, also nur Einsen und Nullen. Natürlich ändert sich dieses Zahl jeden Tag, also bekommt der

Hausmeister einen breiten Pinsel und etliche große Eimer mit Farbe überreicht. Ab sofort ist er dafür verantwortlich, jeden Morgen die Zahl an der Fassade zu korrigieren. Da der Hausmeister ohnehin fest angestellt ist, veranschlagt die Institutsleitung als Kosten für diese Werbemaßnahme lediglich die reinen Sachausgaben für die verbrauchte Farbe, nämlich einen Euro für jede überpinselte Ziffer.

Wie hoch sind die durchschnittlichen Betriebskosten der neuen Fassadengestaltung am Tag? Wir wollen diese Frage klären, ohne genau zu wissen, wie lange es das Institut schon gibt. Denn eins ist klar: Natürlich muss der Hausmeister immer nur diejenigen Ziffern übermalen, die sich geändert haben. Das sind aber zum Glück meistens nur sehr wenige. Wenn das Institut zum Beispiel gerade seinen ersten Geburtstag feiert, also 365 Tage (binär geschrieben 101101101) alt wird, sind es am Vortag noch 364 Tage (101101100) gewesen und das heißt, der Hausmeister muss lediglich an der letzten Stelle die 0 mit einer 1 übermalen (siehe Abbildung 2.9). Schon am nachfolgenden Tag Nummer 366 (101101110) müssen zwei Ziffern geändert werden, am Tag 367 (101101111) dafür wieder nur eine Ziffer. Am Tag 368 (101110000) muss der arme Hausmeister dann allerdings Überstunden machen, um insgesamt fünf Ziffern zu überpinseln. Die Zahl der veränderten Ziffern ist also jeden Tag eine andere.

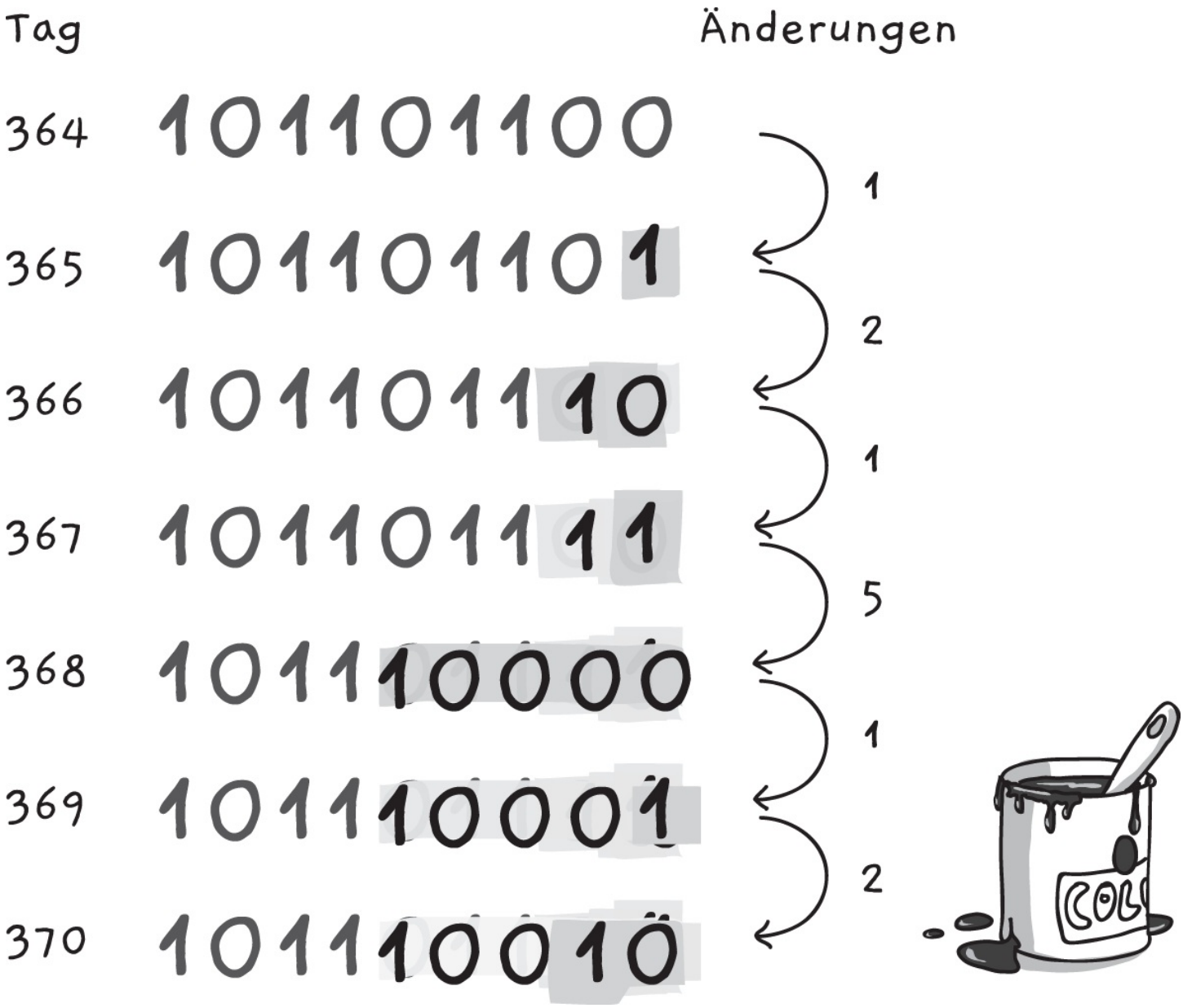


Abbildung 2.9: Die erste Woche des Binärzählers

Der schlimmste Fall ist selten

Im schlimmsten Fall sind alle Ziffern falsch und müssen übermalt werden. Bei einer Zählerlänge von k liegen die Worst-Case-Kosten für den Binärzähler also bei k Euro am Tag.

Diese Schätzung gibt uns allerdings einen völlig falschen Eindruck von den tatsächlichen Kosten des Fassadenzählers. Nehmen wir zum Beispiel einmal an, die Binärzahl wäre 20 Stellen lang. Damit könnte der Zähler auf über eine Million anwachsen, ohne dass man neue Stellen hinzunehmen müsste, und das reicht für die nächsten 2800 Jahre. Nach der Hälfte dieser Zeit, also am Tag 2^{19}, ist es dann schließlich so weit: Das höchstwertige Bit klappt von 0 auf 1 um und alle restlichen Stellen gehen zurück von 0 auf 1. An einem bestimmten Tag im 35. Jahrhundert hat der Hausmeister also einmal eine ganze Menge zu tun. In der Hälfte der restlichen Tage muss er dagegen nur eine einzige Ziffer übermalen.

Der *worst case* ist also extrem selten. Das Wort »selten« drückt dabei aber keine Wahrscheinlichkeit aus, denn der Zähler ist ja in seiner Form vollständig festgelegt, nirgendwo spielt bei ihm der Zufall eine Rolle, also kann man auch nicht von »Wahrscheinlichkeit« reden. Vielmehr ist es so, dass der *worst case* garantiert nur etwa alle 1400 Jahre eintritt. Wenn der Hausmeister also einmal viel gearbeitet hat, dann hat er für die nächste Zeit erst einmal Ruhe.

Der durchschnittliche Aufwand

Wenn wir etwas wiederholt tun (zum Beispiel einen Binärzähler erhöhen) und wenn der Aufwand dafür (zum Beispiel die Anzahl der Ziffern, die man übermalen muss) stark schwankt, oft gering und manchmal hoch ist, dann lohnt sich die Frage nach dem *durchschnittlichen* Aufwand. Angenommen, wir wollen über m Wiederholungen mitteln und T_i ist der Aufwand, den wir bei der i-ten Wiederholung haben. Dann können wir den durchschnittlichen Aufwand T_{mittel} berechnen mit:

$$T_{mittel} = \frac{T_{ges}}{m} = \frac{1}{m} \cdot \sum_{i=1}^{m} T_i$$

der Durchschnittsaufwand ist der Gesamtaufwand geteilt durch die Anzahl der Wiederholungen

Der schwierige Teil dieser Rechnung ist, alle T_i zusammenzuzählen. Wenn der Aufwand stark schwankt, ist so eine Summation manchmal nicht einfach. Beim Binärzähler kommen wir noch vergleichsweise glimpflich davon, denn er arbeitet ja nach festgelegten Regeln und die lassen sich recht leicht durchschauen.



Die mittleren Kosten des Binärzählers

Jede zweite Binärzahl endet auf 0, und wenn eine solche Zahl an der Fassade steht, muss der Hausmeister am nachfolgenden Tag nur diese 0 mit einer 1 überpinseln (Aufwand $T_i = 1$). Von den restlichen Binärzahlen endet die Hälfte – also insgesamt ein Viertel der Zahlen – auf 01, bei diesen Zahlen muss der Hausmeister die beiden letzten Ziffern mit 10 übermalen (Aufwand $T_i = 2$). Ein Achtel der Zahlen endet auf 011 (Aufwand $T_i = 3$), ein Sechzehntel auf 0111 (Aufwand $T_i = 4$). Und so weiter, und so weiter.

Was das für den durchschnittlichen Aufwand bedeutet, kann man sich wie folgt klar machen: An jedem Tag muss der Hausmeister mindestens eine Ziffer übermalen. An jedem zweiten Tag (die Zahl endet auf 1) mindestens noch eine weitere. An jedem vierten Tag (die Zahl endet auf 11) wiederum mindestens noch eine weitere Ziffer. An jedem achten Tag (die Zahl endet auf 111) wiederum mindestens noch eine weitere Ziffer und so weiter. Der durchschnittliche Aufwand beträgt also $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots$ und diese Reihe nähert sich nach und nach immer weiter der 2 an (Abbildung 2.10).

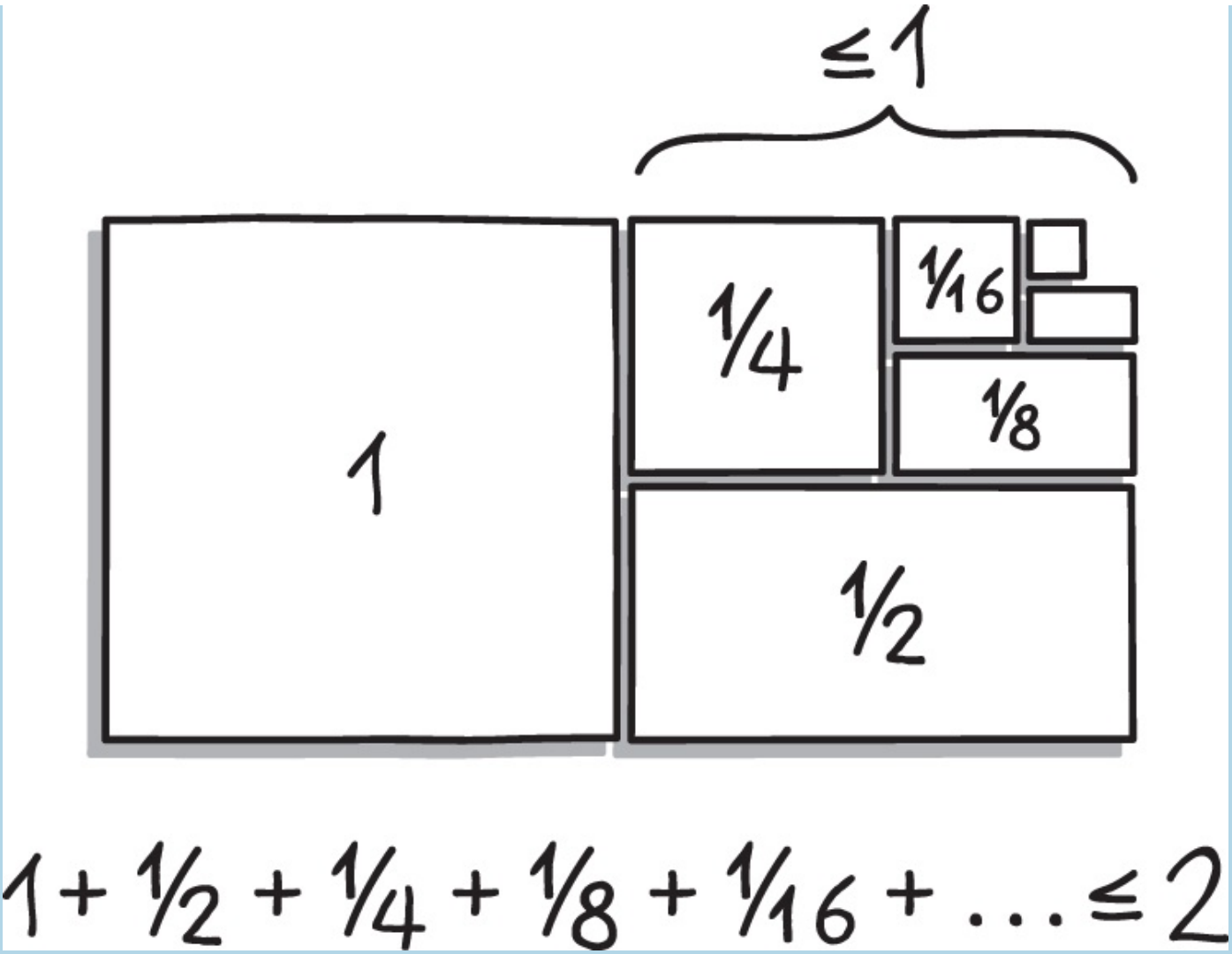


Abbildung 2.10: Durchschnittlich werden $\leq$ 2 Ziffern übermalt.

Viel schwieriger wird es aber, wenn der Aufwand T_i nicht vollkommen vorbestimmt ist, wie bei einem Binärzähler, sondern zum Beispiel von der Eingabe abhängt. So wie bei folgendem Beispiel.

Ein sparsamer Stapel

Programme müssen oft viele Daten speichern. Ein häufig dabei verwendeter Datentyp ist der *Stapel* (Englisch: *Stack*, siehe Abbildung 2.11). In ihm kann man eine beliebig große Menge an Werten speichern und wieder herauslöschen. Einen neuen Wert hinzufügen kann man mit der Methode **push**, einen Wert entfernen mit **pop**. Dabei entfernt man immer denjenigen Wert, der zuletzt auf den Stapel gelegt worden ist, also am kürzesten im Stapel gespeichert war (»LIFO« = »Last In, First Out«). Beide Operationen gehen normalerweise in konstanter Zeit $\mathcal{O}(1)$, vorausgesetzt der Stapel hat noch genügend Platz zur Speicherung neuer Werte. So ein Stapel muss sich ja den Speicherplatz immer mit jeder Menge anderer Objekte teilen. Damit er nicht zu viel Speicher verbraucht, reserviert er sich am Anfang erst einmal nur ein gewisse Menge davon. Wächst der Stapel

mit der Zeit immer weiter an, ist irgendwann seine Kapazität erschöpft und der Platz reicht nicht mehr aus. Nun ist es leider oft nicht möglich, den Speicherplatz einfach auszuweiten, insbesondere wenn die angrenzenden Speicherbereiche bereits mit anderen Daten belegt sind. Statt vor Ort zu wachsen, muss der Stapel also umziehen: Er reserviert sich einen größeren Speicherbereich und kopiert alle bisher gespeicherten Werte in den neuen Bereich um. Seinen bisherigen Speicherbereich braucht er anschließend nicht mehr und kann ihn für andere Objekte freigeben.

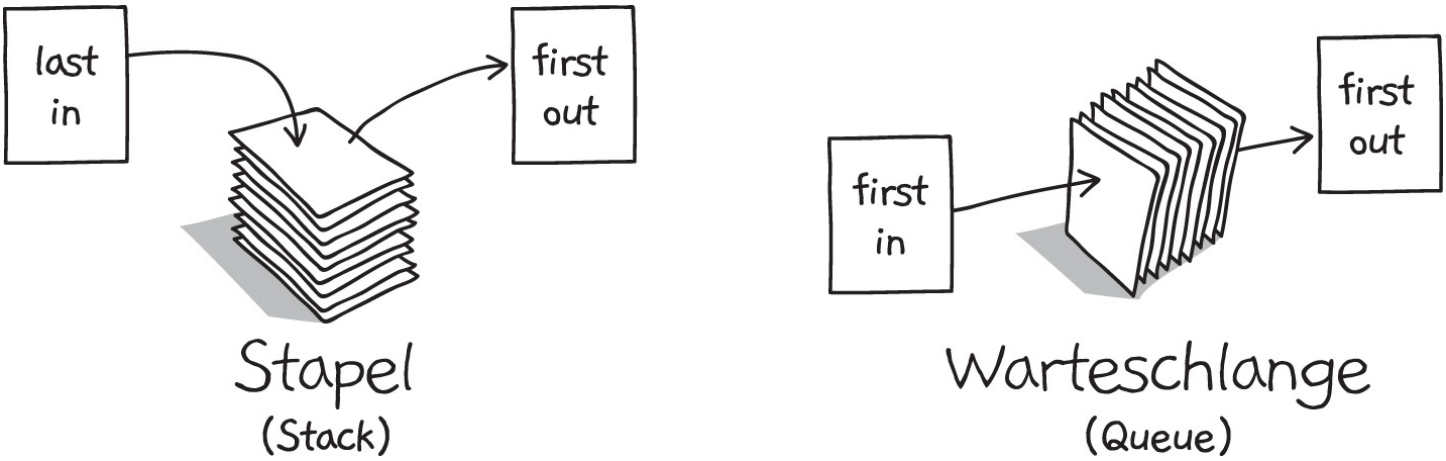


Abbildung 2.11: Stapel und Warteschlangen

Um Speicherplatz zu sparen, kann der Stapel auf die gleiche Weise auch wieder schrumpfen. Ist nur ein kleiner Teil seiner Kapazität mit Werten gefüllt, so zieht er einfach in einen neuen, kleineren Speicherbereich um und gibt den alten, zu groß gewordenen Speicherbereich wieder frei.

Allerdings ist natürlich jeder Umzug mit etlichen Mühen verbunden. Hat ein Stapel bereits n Werte gespeichert, so kostet jeder Umzug eine Laufzeit von $\mathcal{O}(n)$ für das Kopieren der Werte. Die Worst-Case-Laufzeit für diese Operationen ist also doch nicht $\mathcal{O}(1)$, sondern $\mathcal{O}(n)$. Zum Glück ist dieser *worst case* aber selten, zumindest wenn man die Regeln, nach denen der Stapel wächst und schrumpft, einigermaßen geschickt einrichtet. Zum Beispiel so wie in Abbildung 2.12 dargestellt: Sobald der Speicher voll ist, wird er um die Hälfte vergrößert, wird der Speicher hingegen nur zu einem Drittel verwendet, lässt man ihn auf die halbe Größe schrumpfen. Jeder Umzug kostet $\mathcal{O}(n)$ viel Zeit, aber nach jedem Umzug – egal ob der Speicherbereich ausgeweitet oder verkleinert worden ist – sind weitere Umzüge die nächste Zeit nicht mehr nötig. Denn man bräuchte mindestens $\frac{n}{2}$ **push** -Operationen, um den Speicherbereich komplett zu füllen, oder aber mindestens $\frac{n}{2}$ **pop** -Operationen, um die Belegung des Stacks auf ein Drittel der Kapazität zu reduzieren. Welche Kombination von **push** - oder **pop** -Operationen auch immer kommen mag, die nächsten $\frac{n}{2}$ Operationen ist jedenfalls kein erneuter Umzug fällig.

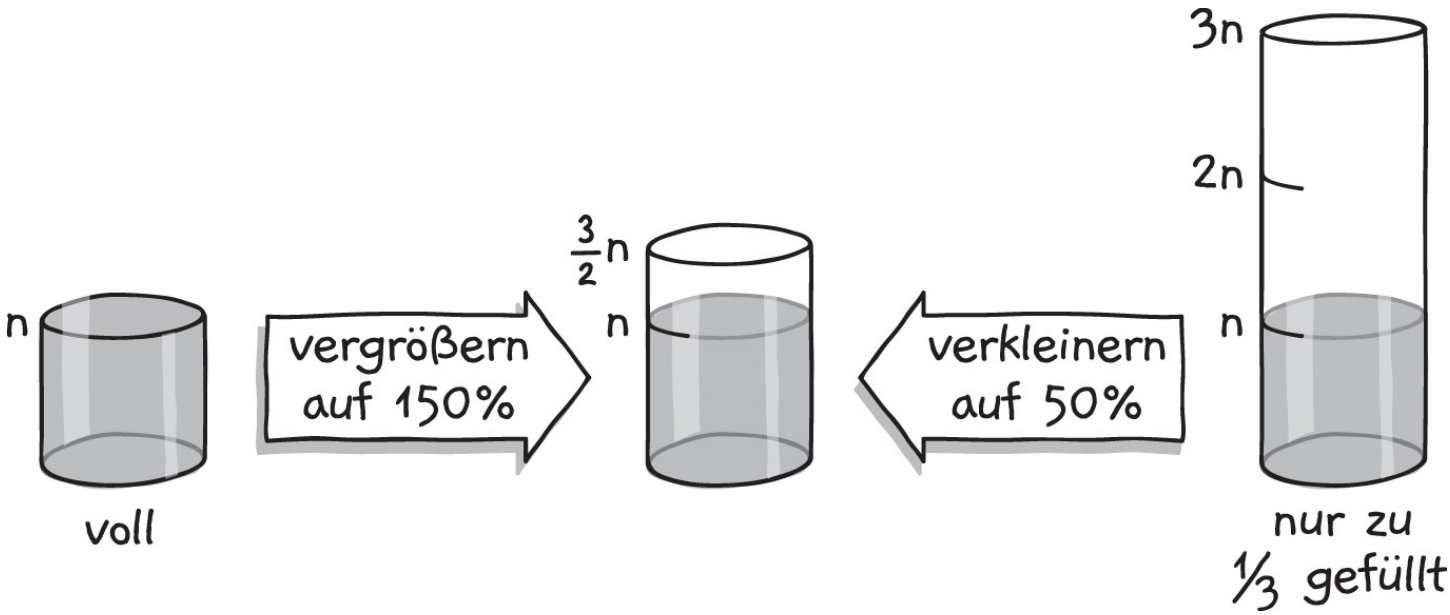


Abbildung 2.12: Wachsen und Schrumpfen der Speicherkapazität eines sparsamen Stapels

Einer Worst-Case-Operation, die $\mathcal{O}(n)$ dauert, stehen also $\mathcal{O}(n)$ Operationen gegenüber, die nur konstante Zeit benötigen. Die »teure« Operation ist gleichsam eine Investition, mit der wir dafür sorgen, dass die nachfolgenden Operationen »billig« werden. Dadurch *amortisiert* sich die Investition mit der Zeit, wie der Kaufmann sagen würde.

Mit dem Begriff der *Amortisation* übernehmen wir auch den folgenden Trick aus der Buchhaltung: Anstatt bei jeder teuren Operation gleich die vollen Kosten T_i als Verlust zu verbuchen, schreiben wir sie über die nachfolgenden Operationen hinweg ab, das heißt, wir verteilen die Mehrkosten der teuren Operationen auf die billigen Operationen. Dadurch werden insgesamt natürlich keine Kosten eingespart, sie werden nur gleichmäßiger verteilt. Statt alle T_i zu addieren, können wir dann die amortisierten Kosten A_i zusammenzählen. Beim sparsamen Stapel klappt das, weil wir dort die Regeln für das Wachsen und Schrumpfen so gewählt haben, dass nach jedem »Umzug« ausreichend Operationen stattfinden, auf die man die Kosten des Umzugs verteilen kann (siehe Abbildung 2.13).

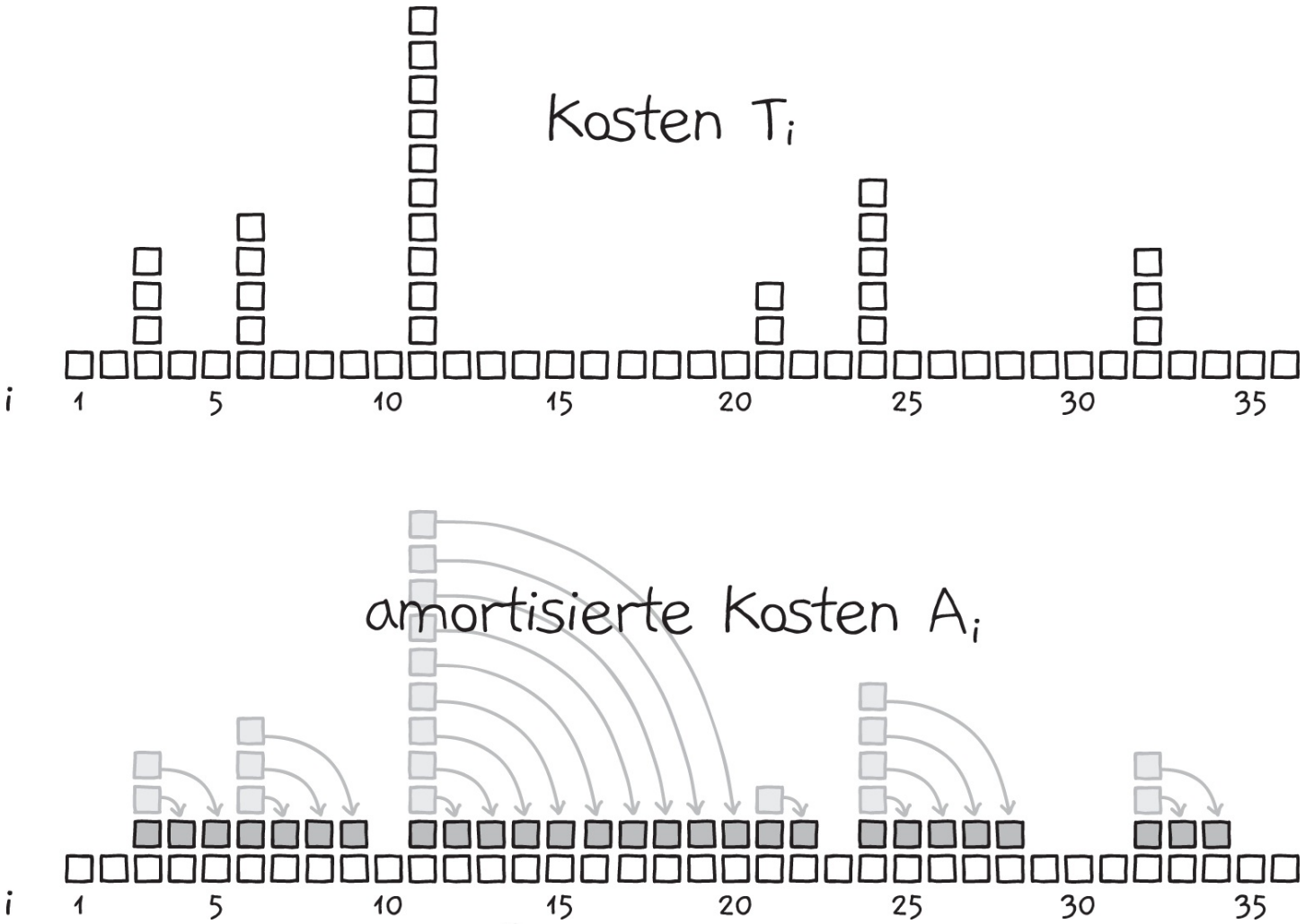


Abbildung 2.13: Die amortisierten Kosten A_i sind in diesem Beispiel ≤ 2.

Die Potenzialmethode

Den Trick des Buchhalters, Investitionen über eine längere Zeit abzuschreiben, kann man auch so beschreiben: Wenn man zum Beispiel einen Computer für 1000 € kauft, dann verliert man zwar 1000 €, bekommt aber gleichzeitig auch einen neuen Computer im Wert von 1000 €, das heißt, insgesamt sind die Kosten eigentlich gleich null. Allerdings verliert der Computer anschließend kontinuierlich an Wert, sagen wir mal 50 € im Monat. Wir haben mit dem Computer gleichsam eine Art Guthaben erworben, das wir anschließend nach und nach aufbrauchen.

Auch beim sparsamen Stapel schlägt sich ein teurer Umzug in einer Art Guthaben nieder (siehe Abbildung 2.14). Nach einem Umzug befindet er sich nämlich in einem optimalen Zustand: Zwei Drittel seines Speicherplatzes ist mit Werten belegt, das heißt, er ist so weit wie möglich von den beiden Grenzen »vollständig gefüllt« und »nur zu einem Drittel gefüllt« entfernt, an denen der nächste Umzug drohen würde. Mit jeder darauf folgenden **push** - oder **pop** -Operation kann der Stapel dieses Optimum immer weiter verlassen und sich einer der beiden Grenzen annähern. Damit verbraucht er sozusagen sein Guthaben, bis es am Ende aufgebraucht ist und ein neuer Umzug ansteht.

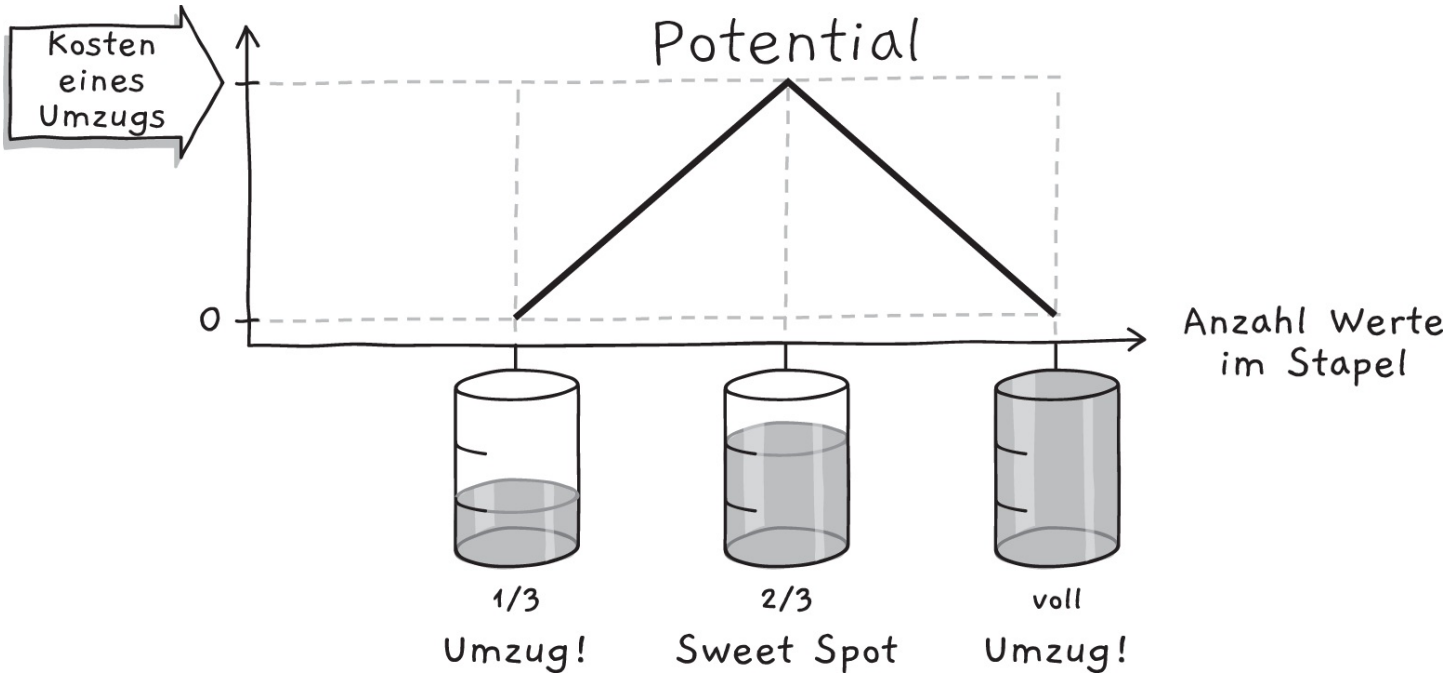


Abbildung 2.14: Dreieckige Potenzialfunktion beim sparsamen Stapel

Für eine genaue Berechnung der amortisierten Kosten muss man natürlich definieren, was dieses Guthaben – in der Fachliteratur wird gewöhnlich von »Potenzial« gesprochen – genau sein soll. Mit dem richtigen Potenzial kann man dann sehr leicht die amortisierten Kosten ausrechnen. Im Kasten »Amortisierte Kosten nachgerechnet« finden Sie ein Beispiel für eine solche Rechnung.



Amortisierte Kosten nachgerechnet

Mit P_i wollen wir das Guthaben – auch »Potenzial« genannt – nach der i-ten Operation bezeichnen. Das durch die i-te Operation verbrauchte Guthaben ist somit $P_{i-1} - P_i$. Das sind die zusätzlichen Kosten, die wir zu T_i addieren müssen, um die amortisierten Kosten A_i zu erhalten:

$$A_i = T_i + (P_{i-1} - P_i)$$

die amortisierten Kosten A_i sind die Kosten T_i plus die Potenzialänderung

Bei billigen Operationen mit kleinem T_i kommt mit $P_{i-1} - P_i$ eventuell noch ein weiterer konstanter Betrag zu den amortisierten Kosten A_i hinzu. Bei einer teuren Operation, also einem Umzug, steigt das Potenzial an, $P_{i-1} - P_i$ ist also negativ und hebt bei geschickter Wahl der Konstanten die hohen Kosten T_i gerade auf – so wie sich die Kosten für den neuen Computer mit dem Wert des Computers beim Kauf gerade aufheben. In beiden Fällen ist A_i nach oben durch eine Konstante beschränkt, A_i über m Operationen ist also $\mathcal{O}(m)$.

Was uns aber eigentlich interessiert, ist die Summe der Kosten T_i und die können wir so berechnen:

$$T_{ges} = \sum_{i=1}^{m} T_i = \sum_{i=1}^{m} (A_i - (P_{i-1} - P_i))$$
$$= P_m - P_1 + \sum_{i=1}^{m} A_i$$

Bis auf P_m und P_1 heben sich alle Potenziale P_i in der Summe gegenseitig weg. Man sagt, die Summe »kontrahiert«. P_1 ist das Potenzial eines leeren Stapels, also gleich 0. P_m ist das Potenzial eines höchstens mit m Werten gefüllten Stapels, also $\mathcal{O}(m)$. Die Gesamtkosten T_{ges} sind also $\mathcal{O}(m)$ und die Durchschnittskosten pro Operation $T_{mittel} = \frac{T_{ges}}{m}$ somit $\mathcal{O}(1)$.

Potenzial beim Binärzähler

Die Rechenmethode mit Potenzialen lässt sich häufig in Beweisen zu amortisierten Kosten einsetzen. Dafür braucht man allerdings eine zündende Idee, wie genau man das Potenzial definiert. Beim Binärzählerproblem funktioniert es mit folgendem Potenzial: » P_i sei die Anzahl der Nullen im Binärzähler am Tag i «. Für den Binärzähler auf der Institutsfassade sind Nullen nämlich ein Guthaben: Sobald der Hausmeister die erste Null von rechts mit einer Eins überpinselt hat, ist er ja mit der Arbeit fertig. Teuer hingegen sind die rechts neben dieser Null stehenden Einsen, die er allesamt in Nullen korrigieren muss. An jedem Tag i zerstört der Hausmeister also eine Null, produziert dafür aber k neue Nullen, übermalt also insgesamt $T_i = k + 1$ Ziffern. Das Potenzial steigt dabei um $k - 1$ an (oder fällt um eins, wenn $k = 0$ ist), also sind die amortisierten Kosten $A_i = T_i + (P_{i-1} - P_i) = k + 1 - (k - 1) = 2$. Das gleiche Ergebnis hatten wir bereits weiter oben erwähnt: Im Schnitt übermalt der Hausmeister zwei Ziffern des Binärzählers. Nur mit der Potenzialmethode konnten wir das ganz ohne komplizierte Summenformeln herausfinden.

Kapitel 3

Daten und ihre Struktur

IN DIESEM KAPITEL

Datenstrukturen: atomar oder strukturiert

Zusammenhängend oder verkettet

Zusammenhängend: Arrays, Records, Tupel

Verkettete Datenstrukturen mit Zeigern

Hierarchische Strukturen als algebraische Datentypen

Rekursive Daten und rekursive Algorithmen

Iteratoren

Aus Daten Strukturen bauen

Strukturen entstehen, wenn Komponenten zusammenkommen und in eine Beziehung treten. Die einfachste Struktur ist die Atomarität: ein einzelnes einsames Element bildet eine *atomare Struktur*. Interessanter wird es, wenn mehrere sich zusammenfinden und in eine Beziehung treten. Als Frau und Mann in einer Ehe oder vorne und hinten auf einem Tandem. Es gibt auch unterschiedliche Arten von Beziehungen: räumliche Beziehungen – vor, neben, übereinander –; organisatorische Beziehungen – die eine die Chefin der anderen – und so weiter.

Strukturen sind die Gestalten oder Muster von Beziehungen. Völlig unterschiedliche Dinge können in Beziehungen stehen, die dem gleichen Muster entsprechen. Die Warteschlange vor der Essensausgabe in der Mensa hat die gleiche Struktur wie die Warteschleife, in der wir uns wiederfinden, wenn wir ein Anliegen an unseren Internet-Provider haben und sein Callcenter anrufen: Das gemeinsame Muster, die Struktur ist *Einer nach dem anderen*.

Wie lässt sich die gemeinsame Struktur der Warteschlange in der Mensa und der Warteschleife in einem Callcenter beschreiben? Klar, es gibt eine Reihenfolge. Einer ist der Erste, dann kommt der Zweite und so weiter. Das Ganze kann beliebig lang werden und auch mal leer sein. Diese Struktur wird in der Mensa ganz real physisch als räumliche Beziehung von Studierenden erzeugt. Wie genau die Warteschleifen in einem Callcenter organisiert sind, das wissen wir nicht, aber jeder weiß, worum es geht, wenn jemand von

einer Warteschlange spricht, auch wenn dort sicher nichts in einem Raum hinter etwas anderem steht.

Datenstrukturen: die Essenz

Datenstrukturen sind Strukturen, deren Komponenten Daten sind. Die Komponenten stehen dann in einer bestimmten Beziehung zueinander. Diese Beziehungen machen die Struktur aus. Glücklicherweise sind die Beziehungen, die Daten miteinander eingehen können, außerordentlich übersichtlich, zumindest im Vergleich zu den Beziehungen zwischen realen Dingen oder gar Personen. Die Komponenten liegen hintereinander und haben einen Index oder einen Namen oder aber sie liegen im Speicher verstreut und werden über ihre Adressen zusammengehalten.


Daten sind entweder

- *atomar*: das Datum hat keine Komponenten oder
- *strukturiert*: das Datum hat Komponenten.

Zahlen, Buchstaben sind atomar, sie sind nicht aus Teilen zusammengesetzt, die für sich allein auch schon sinnvolle Informationen darstellen. Zusammengesetzte Daten bestehen dagegen aus eigenständigen Komponenten, die zusammen eine neue Einheit bilden. Dabei gibt es zwei grundsätzliche Arten der Zusammensetzung (siehe Abbildung 3.1).

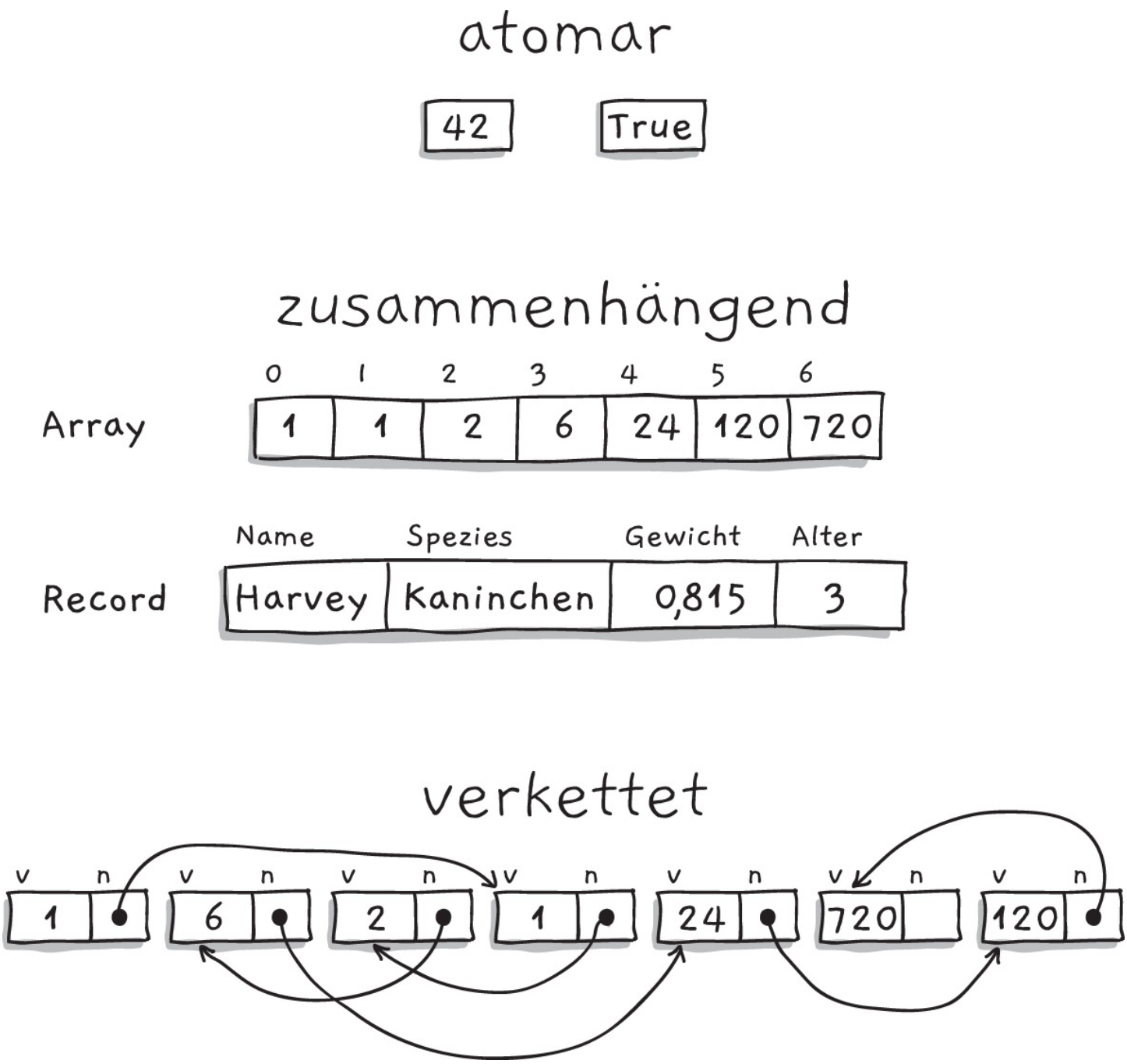


Abbildung 3.1: Die Basis aller Datenstrukturen

Strukturierte Daten bestehen aus Komponenten, die auf zwei Arten in Beziehung stehen können:

- ✔ *Zusammenhängend*: Die Komponenten liegen direkt nebeneinander im Speicher.
- ✔ *Verkettet*: Die Komponenten liegen in Zellen verstreut im Speicher. Die Zellen werden dabei durch Zeiger verbunden.

Eine Datenstruktur enthält neben der *Nutzinformation*, um die es eigentlich geht, oft auch noch *Verwaltungsinformation*. Beispiele für solche zusätzlichen Daten, die nur zu

Organisationszwecken benötigt werden, sind

- ✔ Längenangaben und Stopp-Markierungen: Sie geben Auskunft darüber, inwieweit ein Speicherbereich mit relevanter Information gefüllt ist.
- ✔ Zeiger: Das sind die Adressen von Speicherbereichen, die auch noch »dazugehören«.

Wollen wir beispielsweise ein Wort speichern, dann muss für jeden Buchstaben des Wortes ein Speicherplatz reserviert werden. Wenn dieser Speicherplatz reserviert wird, bevor das Wort bekannt ist, dann können wir einen Speicherbereich fester Größe reservieren und dort alle Buchstaben ablegen, soweit sie in den Speicherbereich passen. Bei Wörtern, die kürzer als die maximale Länge sind, bleibt unbenutzter Speicherplatz übrig. Da aber belegte nicht von unbelegten Speicherzellen zu unterscheiden sind, muss man das Ende mit einer Stopp-Markierung kennzeichnen oder in einer weiteren Zelle die Länge des Wortes speichern. Die Länge des Wortes und die Stopp-Markierung sind dann alternative zusätzliche Verwaltungsinformationen.

Will man Flexibilität in der Länge des Wortes ohne irgendeine Längenbegrenzung, dann wäre eine verkette Liste von Zellen die geeignete Speicherung. Das alles kann dann noch beliebig verschachtelt werden und man erhält am Ende beliebig komplexe Strukturen mit einem Mix an Nutz- und Verwaltungsinformation. In Abbildung 3.2 sehen wir eine der vielen Möglichkeiten, das Kaninchenpaar *Harvey* und *Bruno* in einer Datenstruktur zu speichern.

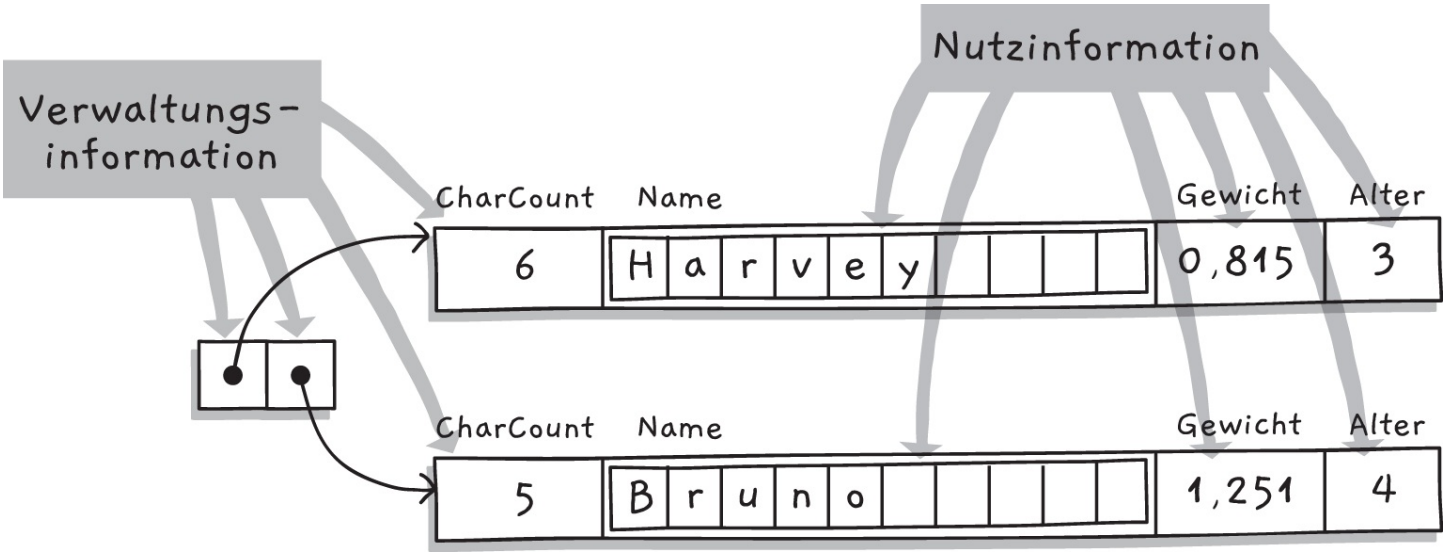


Abbildung 3.2: Datenstrukturen bestehen aus Nutz- und Verwaltungsinformation.

Datenstrukturen im Pseudocode

Datenstrukturen enthalten Anwendungsdaten: die Nutzinformationen, die Sachverhalte der realen Welt modellieren. Manche Sprachen wollen die Entwickler in dieser Modellbildung möglichst gut unterstützen. Sie sollen beispielsweise »in Objekten denken« oder auch »in Funktionen«. Die Sprachimplementierung macht dann daraus die

passenden elementaren Strukturen. Solche Sprachen legen mehr Wert auf die effiziente Nutzung der Entwicklerkapazität als auf die effiziente Nutzung der Maschinen. Verwaltungsinformationen, insbesondere die Zeiger, sind dem direkten Zugriff entzogen. Sie gehören nicht mehr zu den Werten, die ein Programm manipulieren kann. Entwickler sollen sich mit den Nutzinformationen beschäftigen und Verwaltungsinformationen gefälligst dem System überlassen.

Andere Sprachen sind bodenständiger. Entwickler haben den vollen Zugriff auf alle Speicherbereiche und können dabei denken, in was sie wollen. In C kann man beispielsweise sehr nah an der Maschine arbeiten. Entwickler haben es in der Hand, beliebige Dinge sehr effizient zu realisieren. Als Nutzer einer solchen Sprache hat man den vollen Zugriff auf alle Informationen, auch auf die Verwaltungsinformationen einer Datenstruktur. Zeiger sind in solchen Sprachen Werte des Programms, die wie Zahlen frei manipuliert werden können.

Genau das ist der Unterschied zwischen *High-level-Sprachen*, in denen man mit anwendungsnahen Strukturen arbeitet, und den *Low-level-Sprachen*, in denen man systemnah Strukturen manipuliert. Wie ordnet sich hier der Pseudocode ein, ist der anwendungsnah oder systemnah? Die vielleicht etwas verwirrende Antwort ist: Gar nicht. Es geht ihm um etwas ganz anderes.

Wie bei den Algorithmen wollen wir uns im Pseudocode auch bei den Datenstrukturen auf das Wesentliche beschränken und dabei beliebige Abstraktionsebenen zulassen, solange klar ist, was damit auf der Ebene des einfachen Maschinenmodells gemeint ist. Genau wie bei Algorithmen gilt auch für Datenstrukturen die goldene Regel des Pseudocodes:


In *Pseudocode* wird über Details nur dann geredet, wenn sie

- ✔ *relevant* und
- ✔ *nicht selbstverständlich* sind.

Oder umgekehrt: Wenn etwas irrelevant oder selbstverständlich ist, dann wird es weggelassen. Eigentlich genau wie bei einer Kommunikation in natürlicher Sprache und natürlich kann und muss man sich fragen: »relevant für was?« und »selbstverständlich für wen?« Darauf gibt es die immer passende Antwort: »Es kommt drauf an.« Schauen wir also mal an einen Beispiel an, worauf es ankommen kann.

Datenstrukturen können auch mal offen bleiben

Im Pseudocode verwenden wir Strukturen, die

- ✔ bekannt und üblich sind,
- ✔ eine klare Interpretation auf Maschinenebene haben und

- ✔ eine übersichtliche Formulierung von Algorithmen zulassen.

Manchmal erlauben wir es uns aber auch, in guter Pseudocode-Art die Dinge offen zu lassen.

Nehmen wir als Beispiel den Algorithmus LexicalFirst:

```
LexicalFirst(T)                          T sei eine nicht-leere Menge von Wörtern
    f ← irgendein Element von T
    for s ← T
        if s steht im Lexikon vor f
            f ← s
    return f
```

Hier wird das lexikalisch erste Wort in einer Menge T von Wörtern gesucht. Eine Menge ist *keine* konkrete Datenstruktur. Bei T handelt es sich um eine Menge von Wörtern, die als Nutzinformationen in *irgendeiner Datenstruktur* gespeichert sind. Die genaue Datenstruktur, also die Verwaltungsinformationen, etwa ob verkettet oder zusammenhängend, das interessiert hier gerade mal gar nicht.

Wir nehmen uns die Freiheit, bei Datenstrukturen derart unspezifisch zu sein. Sie ergibt sich aus dem Prinzip des Pseudocodes, nach dem Selbstverständliches und Unwesentliches einfach weggelassen werden kann. Die zu verwendende Datenstruktur ist sicher nicht selbstverständlich. Dafür gibt es zu viele unterschiedliche Möglichkeiten. Also ist sie nicht relevant? Nicht relevant für was?

Bei Algorithmen geht es nicht immer, aber oft um die Qualität und die wird in Laufzeit gemessen. Oft ist die asymptotischen Laufzeit von Interesse und für die ist hier die genaue Struktur, in der die Daten von T gespeichert sind, nicht relevant. Relevanz bedeutet also »Relevanz für das große $\mathcal{O}$«. Warum ist nun die Wahl einer Datenstruktur ohne Bedeutung für die Laufzeitabschätzung?

Der Algorithmus LexicalFirst kann auf jeder Datenstruktur agieren, die Wörter enthält und hat eine Laufzeit in $\mathcal{O}(n)$. Für das $\mathcal{O}(n)$ werden folgende »Selbstverständlichkeiten« angenommen:

- ✔ Die Eingabelänge n repräsentiert die Zahl der Wörter.
- ✔ die lexikalische Ordnung von zwei Wörtern kann mit einer Laufzeit in $\mathcal{O}(1)$ verglichen werden.
- ✔ die Elemente von T stecken in einer Datenstruktur, die ihre einzelnen Elemente mit einer Laufzeit in $\mathcal{O}(1)$ ausliefern kann.

In Kapitel 2 wird bei der Diskussion von LexicalFirst darauf hingewiesen, dass die erste und die zweite Annahme unter Umständen nicht zutreffen. Das ist dann der Fall, wenn es

keine feste Beschränkung der Wortlänge gibt. In dem Fall wird die Eingabelänge n auch von der Länge der Wörter bestimmt: Das n in $\mathcal{O}(n)$ ist dann nicht mehr die Zahl der Wörter in T, sondern die aufsummierte Länge der Wörter.

Genauso wie die erste und zweite kann auch die dritte Annahme unter Umständen falsch sein. Hier wird implizit angenommen, dass der Zugriff auf ein einzelnes Element von T *innerhalb einer For-Schleife* in konstanter Zeit erfolgt, also völlig unabhängig von n, egal ob die Eingabegröße n die Zahl der Wörter ist oder ob n auch die Länge der Wörter enthält. Nächstes Wort: ein Zugriff fester Dauer, egal ob T 10 oder 10 Millionen Wörter enthält.

```
Alg(M)                          M sei eine Menge von Wörtern
    for x ← M
        ... x ...               Zugriff mit O(1)
```

Diese Annahme ist vernünftig, aber es kann sein, dass sie nicht zutrifft. Vernünftig, weil man von einer Datenstruktur, die eine Kollektion von Elementen verwaltet, erwarten kann, dass sie in einer For-Schleife jedes ihre Elemente in konstanter Zeit ausliefern kann. Beispielsweise mit einem Iterator. Iteratoren wandern durch eine Datenstruktur und liefern ein Element nach dem anderen aus. Weiter vorne im Abschnitt *Strukturen durchlaufen: Iteratoren und Traversierungen* gehen wir näher auf Iteratoren ein. Oft wird einfach vorausgesetzt, dass ein solcher Iterator existiert, ohne sich näher damit zu beschäftigen, wie er realisiert ist.



Die Verwendung einer Menge im Pseudocode bedeutet meist, dass die konkrete Datenstruktur für den Algorithmus im Rahmen von offensichtlichen Standardannahmen unwesentlich ist. An anderer Stelle kann sie eventuell noch präzisiert werden.

Mengen in Pseudocode bedeuten also nicht, dass es sich um eine Datenstruktur vom Typ »Menge« handelt, sondern dass wir es mit einer Menge an Nutzinformationen zu tun haben, die in irgendeiner Datenstruktur gespeichert sind. Ähnliches gilt für »Listen«. Gelegentlich beginnt ein Algorithmus damit, dass gewisse Informationen sortiert werden. Beispielsweise könnten wir das lexikalisch erste Wort auch mit folgendem Algorithmus finden:

```
LexicalFirstS(T)                T sei eine nicht-leere Menge von Wörtern
    S ← T lexikalisch sortiert
    return S(0)                 das erste nach der Sortierung
```

oder auch noch einfacher:

```
LexicalFirstS(T)                T sei eine nicht-leere Menge von Wörtern
    Sortiere T lexikalisch
    return T(0)
```

Die Wörter werden mit irgendeinem Verfahren sortiert. Nach der Sortierung haben sie eine Reihenfolge: es gibt einen Ersten, einen Zweiten und so weiter. In dieser Reihenfolge kann auf sie zugegriffen werden. Kollektionen, in denen die enthaltenen Elemente eine Reihenfolge haben, nennen wir Listen, Folgen oder Sequenzen. Es bleibt dabei offen, in welcher Datenstruktur diese Elemente stecken. Eine For-Schleife liefert sie in ihrer Reihenfolge aus:

```
...
Sortiere T lexikalisch
for t ← T                    die Elemente kommen sortiert, jeweils mit O(1)
    ... t ...
```

Der Zugriff auf das i -te Element ist möglich:

```
...
Sortiere T lexikalisch
T(i)                         das i-te Element kommt mit O(?)
...
```

aber die Laufzeit des Zugriffs ist unbekannt. Mengen stehen für Datenstrukturen, über die wir nichts wissen oder fordern, außer dass auf ihre Elemente in einer For-Schleife mit $\mathcal{O}(1)$ zugegriffen werden kann. Von einer Datenstruktur kann man natürlich mehr fordern, wie etwa den Zugriff über einen Index:

```
Alg(L)                       L sei eine Liste von Wörtern
    while i < ...            allgemeine Schleife
        x ← L(i) ...         Zugriff mit unbekanntem O(?)
        i ← ...

    for x ← L                For-Schleife über Struktur
        ... x ...            Zugriff mit O(1)
    ...
```

L ist jetzt streng geordnet. Es hat eine Länge, die Elemente in L können mehrfach vorkommen und für jedes $i <$ Länge von L ist $L(i)$ definiert. So etwas nennt man *Liste* oder auch Sequenz. Wir fordern von L , dass es mehr kann als die Menge M oben. Damit ist aber die konkrete Datenstruktur noch nicht festgelegt und es ist noch nichts über die Laufzeit des Zugriffs auf $L(i)$ gesagt.



Ein Algorithmus fordert von einer Datenstruktur meist nur bestimmte Zugriffsoperationen. Für die *Funktionalität* des Algorithmus ist es unerheblich, wie die Datenstruktur konkret aussieht und wie die Zugriffsoperationen realisiert werden. Für die *Laufzeit* des Algorithmus ist dies aber ganz und gar nicht unwesentlich.

Möglicherweise ist der Aufwand, der in Alg getrieben werden muss, um auf das i -te Element von L zuzugreifen, von der Eingabegröße abhängig. Etwa wenn es sich bei

(einer Implementierung von) L um eine mit Zeigern verkettete Struktur handelt, die bei jedem Zugriff vom Anfang bis zum i -ten Element durchlaufen werden muss. Natürlich ist es denkbar, dass ein Iterator in dieser ineffizienten Art realisiert wird. Von einem Iterator erwartet man aber einen effizienteren Zugriff. Darum nehmen wir $\mathcal{O}(1)$ bei Zugriffen in For-Schleifen an. Bei einer While-Schleife gilt diese Annahme aber nicht.



Notation für Datenmodelle als nicht näher spezifizierte Datenstrukturen

- *Menge*: Werte sind drin oder nicht drin; keine oder unbekannte innere Organisation.
  - Wert: $\{Element\ 1, ..., Element\ n\}$ oder: $\{x : Bereich\ von\ x \mid Eigenschaft\ von\ x\}$
  - Zugriff: nur in einer For-Schleife mit $\mathcal{O}(1)$
- *Liste*: Positionen sind Werte zugeordnet. Eine innere Organisation der Daten macht diese Zuordnung möglich.
  - Wert: $[Element\ 1, ..., Element\ n]$ oder: $[x : Bereich\ von\ x \mid Eigenschaft\ von\ x]$ Die Elemente müssen dabei eine natürliche Ordnung haben oder die Ordnung muss angegeben werden.
  - Zugriff: mit einem Index $L[i]$ mit unbestimmter Laufzeit oder in einer For-Schleife mit $\mathcal{O}(1)$

Datenstrukturen und das große $\mathcal{O}$

Manches kann offen bleiben, anderes muss explizit gesagt werden. Der Begriff der »Eingabelänge« spielt bei der Unterscheidung eine wichtige Rolle. Das gilt auch für Zeiger.

In vielen Programmiersprachen gibt es eine spezielle »Referenzlogik«: Manche Dinge können nicht direkt in einer Struktur gespeichert werden, sondern müssen immer über einen Zeiger angesprochen werden. In Java können Strings beispielsweise nicht direkt in einer Variablen, einem Array oder einem Record (Objekt) gespeichert werden. Sie müssen darum in solch einem Fall immer über Zeiger angesprochen werden. Das liegt daran, dass sie keine Längenbeschränkungen haben und jederzeit gegen einen längeren ausgetauscht werden können. Das ist ungünstig für eine zusammenhängende Speicherung. In C ist das wiederum nicht der Fall. Dort ist eine Zeichenkette ein Array von Zeichen. Ein solches Array kann in einem separaten Speicherbereich stecken, das muss aber nicht zwingend so sein. In einer zusammenhängenden Struktur kann man einen Speicherbereich fester Größe für eine Zeichenkette reservieren und diese dann direkt in eine zusammenhängende Struktur stecken, ohne dass ein Zeiger im Spiel ist.

Im Pseudocode ignorieren wir oft den Unterschied zwischen einer Speicherung direkt in einer zusammenhängenden Struktur und einer Speicherung, bei der Zeiger im Spiel sind. Das sind Implementierungsdetails, die von einer bestimmten Programmiersprache in bestimmter Art gefordert werden oder auch nicht. Für Effizienzbetrachtungen, die sich an asymptotischen Laufzeitbeschränkungen orientieren, ist nur das wesentlich, was von der Eingabelänge beeinflusst wird. Ob eine Datenstruktur oder ihre Bestandteile verkettet oder zusammenhängend gespeichert sind, ist unerheblich.

Schauen wir noch einmal auf LexicalFirst und betrachten die beiden Extremfälle einer zusammenhängenden und einer verketteten Speicherung. Bei der zusammenhängenden Speicherung (siehe Abbildung 3.3) sind alle Daten im Speicher in einem Array von Arrays aneinander gereiht.

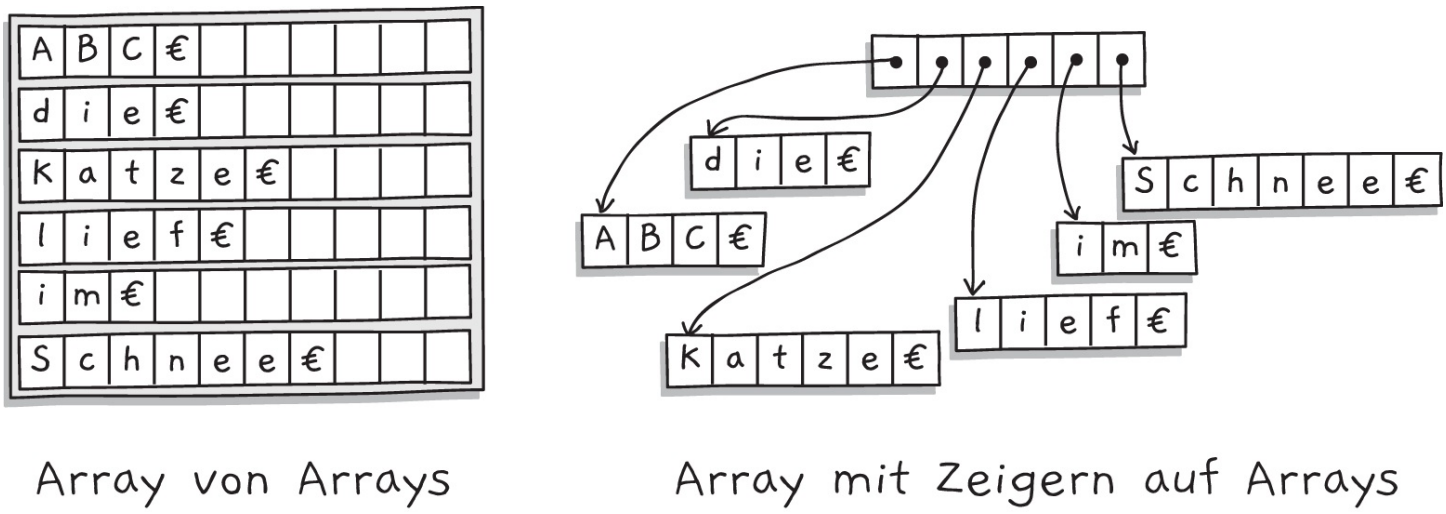


Abbildung 3.3: (Teilweise) zusammenhängende Speicherung

Für jedes Wort steht in der Struktur ein Speicherplatz fester Größe zur Verfügung. Das Ende eines Wortes wird dabei durch ein spezielles Sonderzeichen (hier €) markiert. Einem Mix aus zusammenhängender und verketteter Speicherung hätten wir bei einer Speicherung in einem Array mit Zeigern auf Arrays mit jeweils passender Länge (siehe Abbildung 3.3).

Bei einer völlig verketteten Speicherung werden alle Informationen über Zeiger verkettet abgespeichert (siehe Abbildung 3.4).

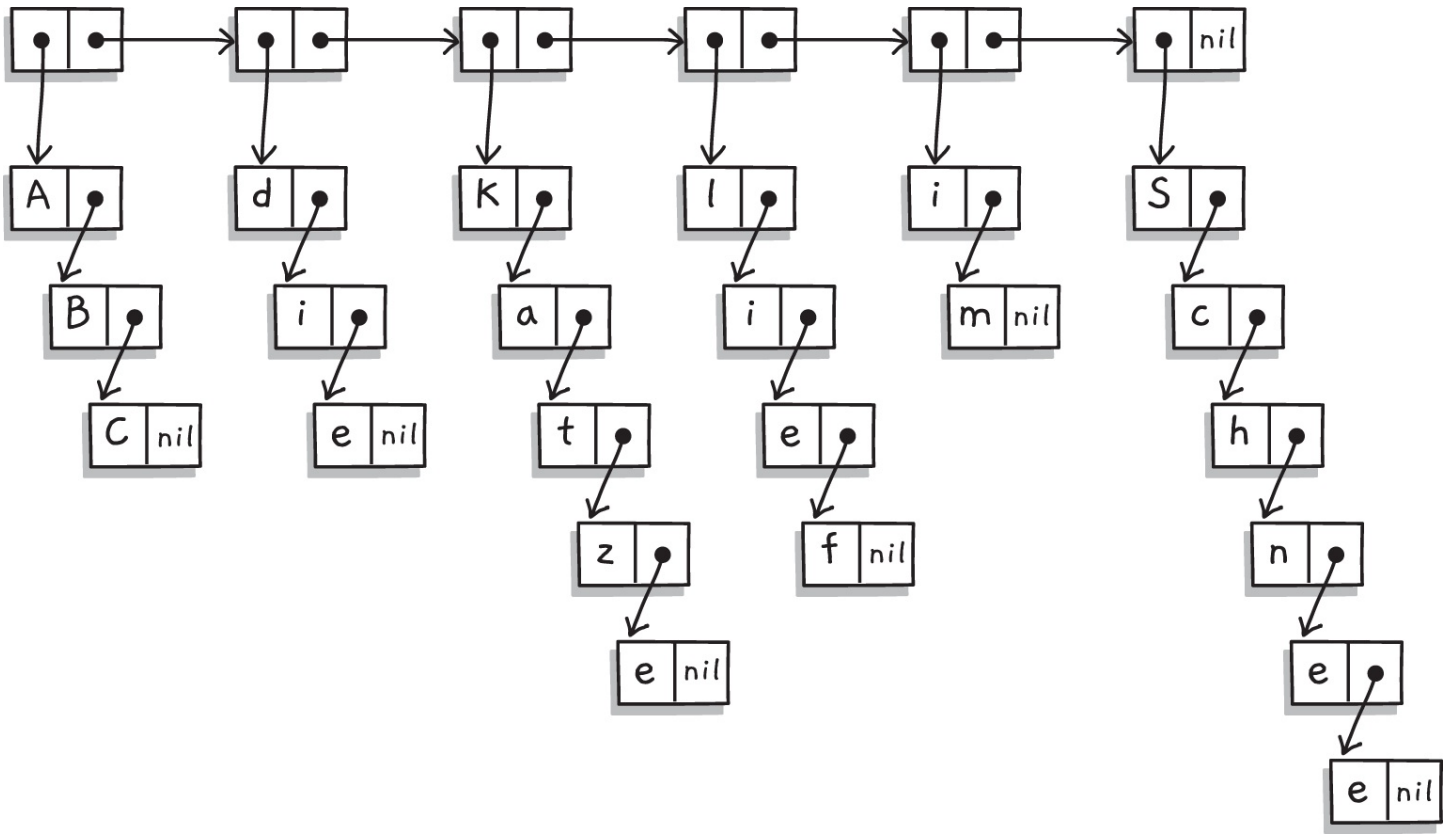


Abbildung 3.4: Völlig verkettete Speicherung

Jede Struktur hat ihre Vor- und Nachteile. Manche werden von manchen Sprachen zugelassen und unterstützt, andere können in bestimmten Sprachen gar nicht realisiert werden. Bei jeder dieser Strukturen, egal ob sie zusammenhängend, verkettet oder in einem Mix von beiden gespeichert sind, sind alle Zugriffe auf Wörter und Buchstaben in $\mathcal{O}(1)$, falls es eine Obergrenze für die Wortlänge gibt – und nur dann.

Es ist klar, dass bei einer völlig verketteten Speicherung mehr Aktionen notwendig sind, um auf das letzte »e« von »Schnee« zuzugreifen, als bei der kompakten völlig zusammenhängenden Speicherung. Da die Wortlänge begrenzt ist, gibt es aber eine feste Obergrenze für den Zusatzaufwand. Der Zugriff ist zwar langsamer, aber konstant langsamer, er ist immer noch in $\mathcal{O}(1)$.

Die Sache sieht etwas anders aus, wenn es keine Obergrenze für die Länge der Wörter gibt. In dem Fall ist die völlig oder teilweise zusammenhängende Speicherung zwar auch möglich, sie fordert aber einen Zusatzaufwand, der nicht zu vernachlässigen ist: Wenn sich herausstellt, dass der Platz nicht ausreicht, müssen eine größere Struktur allokiert und die Daten umkopiert werden. Das kann beliebig oft vorkommen. Allerdings sind das Aktionen während des Einlesens der Daten, also bevor der Algorithmus LexicalFirst startet. Es ist darum nicht irrelevant, hat aber mit LexicalFirst nichts zu tun.

Die unbeschränkte Wortlänge hat trotzdem einen Einfluss auf die Laufzeit von LexicalFirst. Die Wortlänge geht nämlich in die Eingabelänge n ein. Das sieht man deutlicher, wenn wir den Algorithmus etwas umschreiben:

```
LexicalFirst(T)                          T sei eine Menge von n Wörtern
    f ← Leerstring ""
    for s ← T                            die Zahl n der Durchläufe hängt von der Größe von T ab
        if Before(s, f)
            f ← s
    return f
Before(wort1, wort2)                     Eingabelänge n ist die Zahl der Buchstaben
    i ← 0
    while i < length(wort1) and i < length(wort2)      Durchlauf-Zahl hängt ab von n
        if Buchstabe i von wort1 >       Laufzeit des Zugriffs auf Buchstabe i
                Buchstabe i von wort2    hängt von der Datenstruktur ab
            return false
        return true
```

Die Laufzeit der Schleife in LexicalFirst ist, bis auf einen konstanten Faktor, unabhängig davon, ob die Wörter zusammenhängend in einem Array oder in einer verketteten Struktur gespeichert werden. Ein Zugriff über einen Array-Index oder Iterator ist konstant.

Die Schleife in Before ist keine For-Schleife über alle Elemente einer Struktur. Die $\mathcal{O}(1)$-Selbstverständlichkeit des Zugriffs gilt nicht mehr. Jetzt und hier müssen wir genauer werden. Falls, für den Zugriff auf den i-ten Buchstaben eines Wortes, eine Liste von Buchstaben durchlaufen werden muss, deren Länge von der *Eingabelänge* beeinflusst wird, ist der Zugriff in $\mathcal{O}(n)$. In jedem Schleifendurchlauf oben haben wir sogar zwei solcher Zugriffe. Jetzt kommt es also darauf an, wie die Datenstruktur organisiert ist und wie der Zugriff realisiert wird.



Datenstrukturen und die Laufzeitabschätzung von Zugriffen

- ✔ Die Größe einer Struktur ist von der Eingabe *unabhängig*: Jeder Zugriff ist in $\mathcal{O}(1)$, egal ob die Struktur zusammenhängend, verkettet oder mit einem Mix von beiden gespeichert wird.
- ✔ Die Größe einer Struktur ist von der Eingabe *abhängig*:
  - Innerhalb von For-Schleifen, die über die Komponenten einer Struktur laufen: Der Zugriff auf die *aktuelle* Komponente wird mit $\mathcal{O}(1)$ angenommen, egal ob die Struktur zusammenhängend, verkettet oder mit einem Mix von beiden gespeichert wird.
  - Ansonsten: Der Zugriff auf eine Komponente und dessen Laufzeit werden von der Eingabelänge n beeinflusst ($\mathcal{O}(?n?)$). Die Art des Einflusses ist abhängig von der Art der Speicherung und der Organisation des Zugriffs.

Wir müssen uns also gelegentlich unter Deck bewegen und die Maschinerie der Datenstrukturen genauer ansehen. Selbst dann, wenn wir nur an einer asymptotischen Laufzeitabschätzung interessiert sind. Wenn uns die Zahl der Rechenschritte und nicht nur ihre asymptotische Abschätzung interessiert, dann gilt dies natürlich umso mehr. Ein Rechenschritt und 100 000 Rechenschritte sind in der gleichen Laufzeitklasse $\mathcal{O}(1)$, aber ein kleiner Unterschied besteht dann doch noch zwischen den beiden Situationen.

Eine genauere Betrachtung der Datenstrukturen ist nicht zu vermeiden. Wir müssen nur noch entscheiden, mit welchen Begriffen wir über ganz konkrete Datenstrukturen reden wollen.

Zusammenhängende Strukturen: Arrays, Records, Tupel

Das Angebot an zusammenhängenden Strukturen unterscheidet sich von Programmiersprache zu Programmiersprache. Üblich sind:

- ✔ *Arrays*: Gleichartige Komponenten, die über einen Index angesprochen werden
- ✔ *Records*, *Tupel*, Objekte: Nicht unbedingt gleichartige Komponenten, die über einen Namen (Record, Objekt) oder einen Index (Tupel) angesprochen werden

Wir verwenden Arrays, Records und Tupel. Tupel unterscheiden sich nicht wesentlich von Records, sie vereinfachen aber oft die Formulierung von Algorithmen. Objekte sind Records »mit Mechanismen der Objektorientierung«, an denen wir hier nicht interessiert sind. Records dürfen allerdings auch Funktionen enthalten.

Bei einem Array werden die Komponenten über einen Index angesprochen. Arrays enthalten immer Komponenten gleicher Art. In Records und Tupeln können beliebige Mischungen von Komponenten auftreten. Bei Records werden die Komponenten über Namen angesprochen, bei Tupeln über einen Index. Wir erläutern dies an Beispiel-Algorithmen.

Der erste Beispiel-Algorithmus erzeugt ein Array, das die Fakultäten bis n enthält, also $1, 1, 2, 6, 24, 120, \ldots$ Das Array wird angelegt und dann in einer Schleife mit Werten gefüllt:

```
FArray1(n)
    a ← array[0..n] of Int        leeres Array a von ganzen Zahlen anlegen
    a[0] ← 1
    for i ← 1 ... n               a mit Fakultäten füllen
        a[i] ← a[i − 1] · i
    return a
```

Mit dem folgenden Algorithmus wird ein Array mit Wertepaaren erzeugt. Ein Paar ist ein *Tupel* mit zwei Elementen. $a[i]$ enthält das Tupel (Paar) $(i, i!)$, also eine Zahl und ihre Fakultät. Insgesamt wird das Array also mit den Paaren

$(0, 1), (1, 1), (2, 2), (3, 6), (4, 24), (5,120), \ldots$ befüllt:

```
FArray2(n)
    a ← array[0..n] of (Int, Int)        leeres Array a von Zahlenpaaren anlegen
    a[0] ← (0, 1)                        das Paar (0, 1) in a[0] speichern
    for i ← 1 ... n
        a[i] ← (i, a[i − 1].2 · i)       das Paar (i, i!) in a[i] speichern
    return a
```

Ein Tupel t besteht aus mehreren durchnummerierten Elementen. Auf das i -te Element von t wird mit t .i zugegriffen. Dabei ist *i* ein beliebiger Ausdruck mit ganzzahligem Wert. Mit

$$a[i] \leftarrow (\, i \,,\, a[i-1].2 \cdot i \,)$$

wird im Beispiel ein Paar in $a[i]$ gespeichert. Dessen erste Komponente ist i und die zweite ist das Produkt von i und $a[i-1].2$. Es wird also aus i und der zweiten Komponente des vorherigen Array-Elements berechnet.

Statt der Tupel kann man auch *Records* verwenden. Die Komponenten eines Records haben Namen statt Nummern. Im folgenden Beispiel sind die Namen der Komponenten v und f:

```
FArray3(n)
    a ← array[0..n] of record[v: Int, f: Int]
    a[0] ← record[v:0, f:1]              einen Record mit v=0 und f=1 in a[0] speichern
    for i ← 1 ... n
        a[i] ← record[v: i, f: t[i − 1].f · i]        i-ten Record aus vorherigem berechnen
    return a
```

Auf die Komponenten wird über ihre Namen zugegriffen. **record** [v:3, f:6] ist ein Record, der eine v-Komponente mit dem Wert 3 und eine f-Komponente mit dem Wert 6 enthält. Mit der Punktnotation kann auf die Komponenten zugegriffen werden:

```
record[v:3, f:6].v = 2
record[v:3, f:6].f = 6
```

Bei der Definition von Records lassen wir die Namen weg, wenn der Name der Komponente aus deren Reihenfolge hervorgeht. Also beispielsweise:

```
record[3, 6].v = 2
record[3, 6].f = 6
```

Mit *Typdefinitionen* können die Dinge gelegentlich übersichtlicher gestaltet werden. Wir können beispielsweise einem Record-Typ einen Namen geben:

```
type Cell = record[v:Int, f:Int]
```

Diese Möglichkeit ist hilfreich, wenn verkettete Strukturen definiert werden. Dazu sehen

wir im nächsten Abschnitt ein Beispiel.

Gelegentlich stecken wir auch Algorithmen in einen Record. Diese haben dann Zugriff auf alle Komponenten des Records, ohne dass diese explizit übergeben werden müssen. Ein Beispiel ist ein Record, der einen Zählerwert und zwei Operationen enthält, eine, die den Zählerwert erhöht, und eine, die ihn ausliefert:

```
counter ← record[
    v ← 0
    Inc()
        v ← v + 1
    GetV()
        return v
]
```

Mit $counter$.Inc() wird der Zählerwert erhöht und $counter$.GetV() liefert seinen aktuellen Stand.



Notation für Datenstrukturen mit zusammenhängender Speicherung

- ✔ **Array:**
  - Definition: **array** [*Indexbereich*] **of** *Wertetyp*
  - Wert: [*Komponente 1*, ..., *Komponente n*]
  - Zugriff: $a[i]$
- ✔ **Tupel:**
  - Definition: (*Wertetyp 1*, ..., *Wertetyp n*)
  - Wert: (*Komponente 1*, ..., *Komponente n*)
  - Zugriff: $t.i$
- ✔ **Record:**
  - Definition: **record** [*Name 1*: *Wertetyp 1*, ..., *Name n*: *Wertetyp n*]
  - Wert: **record** [*Komponente 1*, ..., *Komponente n*]
  - Zugriff: $t.name_i$

Case-Notation

Zusammenhängende Strukturen wie Records oder Tupel werden in Algorithmen üblicherweise in ihre Komponenten zerlegt. Das kann mit der **case** -Notation übersichtlich beschrieben werden:

```
FArray3(n)
    a ← array[0..n] of record[v: Int, f: Int]
    a[0] ← record[0, 1]                 einen Record mit v=0 und f=1 in a[0] speichern
    for i ← 1 ... n
        case a[i − 1] = record[x, v]    wenn der vorherige Record so aussieht, ...
            a[i] ← record[i, v · i]     ... dann speichere an Index i diesen Record
    return a
```

Case-Ausdrücke sind als *Mustererkennung* (*Pattern Match*) zu lesen (siehe Abbildung 3.5). In

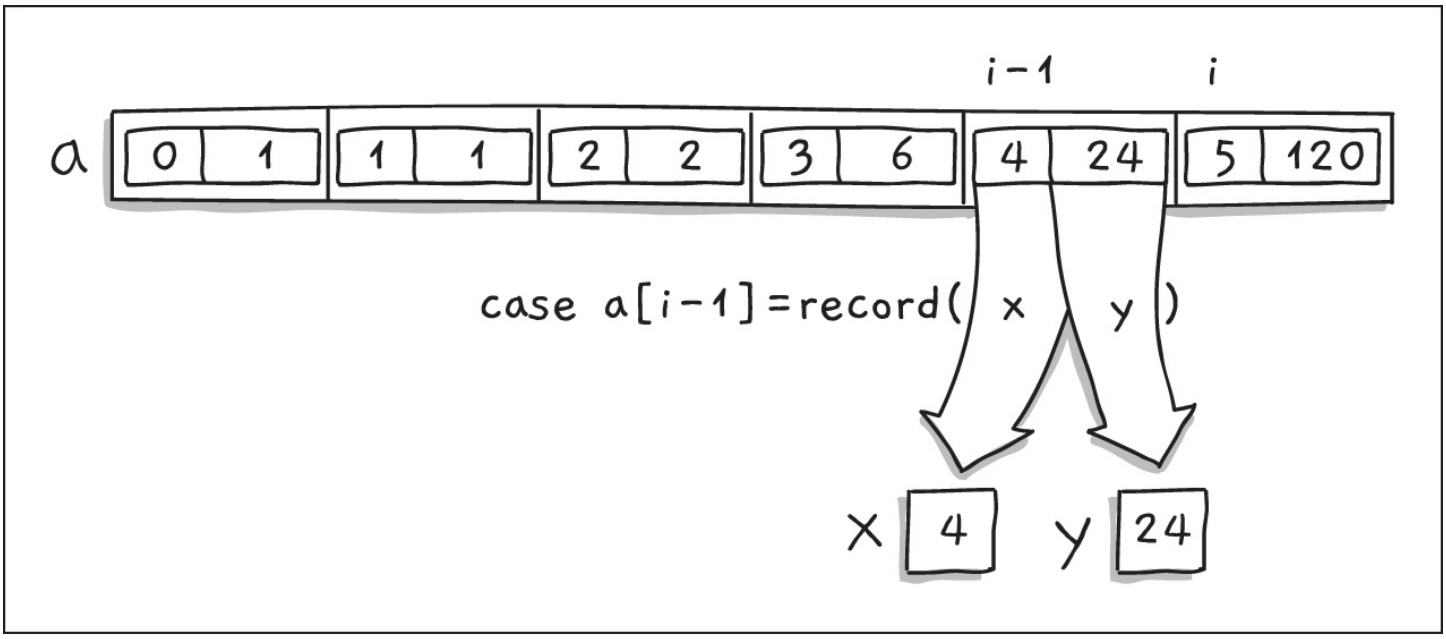


Abbildung 3.5: Case-Konstrukt: eine Musterkennung in Strukturen

```
case a[i − 1] = record[x, v]        seien x und v die Komponenten im Wert von a[i-1]
```

ist i der Wert, den die Variable i gerade hat. x und v sind dagegen die Namen für die beiden Werte, die in dem Record stecken, der sich gerade in der $(i-1)$-ten Zelle des Arrays a befindet. Am besten liest man dies als:

> *Wenn $a[i-1]$ einen Record enthält mit einem ersten Wert, den wir x nennen und einem zweiten Wert, den wir v nennen, dann …*

Wie bereits erwähnt lassen wir die Namen der Komponenten weg, wenn sie klar sind.

Die **case** -Notation passt noch besser, wenn sie in Zusammenhang mit Tupeln verwendet wird, zum Beispiel in folgender Umformulierung von FArray2 (Tupel statt Records):

```
FArray2(n)
  a ← array[0..n] of (Int, Int)      leeres Array a von Zahlenpaaren anlegen
  a[0] ← (0, 1)                      das Paar (0, 1) in a[0] speichern
  for i ← 1 ... n
    case a[i − 1] = (_, x)           der erste Wert des Tupels in a[i] interessiert nicht
      a[i] ← (i, x · i)              das Paar (i, x · i) in a[i] speichern
  return a
```

Der Unterstrich bedeutet, dass uns die erste Komponente des Tupels nicht interessiert und wir darum darauf verzichten, ihr einen Namen zu geben. Der Test von Kaninchenpaaren auf Namensgleichheit wird mit Tupeln und der **case** -Notation zu

```
type Tier = (Wort, Real, Int)        ein Tier ist ein Tupel mit drei Komponenten
type Paar = (Tier, Tier)             ein Paar hat zwei Komponenten
GleichnamigesPaar(paar: Paar)
    case paar = ((n1, _, _), (n2, _, _))   Paar ist gleichnamig, wenn die zwei
        return n1 = n2                     Elemente gleiche erste Komponenten haben
```

Die **case** -Notation macht die Formulierung etlicher Algorithmen klarer und wird auch von funktionalen Programmiersprachen wie etwa Haskell oder Scala unterstützt.

Verkettete Strukturen und Zeiger

Richtige Handwerker schrauben den Sicherungskasten auf und klemmen Drähte in Verteilerkästen an. Richtige Programmierer arbeiten mit Zeigern! Algorithmiker manchmal auch.

Verkettete Strukturen bestehen aus Speicherbereichen, die über Zeiger miteinander verbunden sind. Ein Zeiger ist die Adresse eines Speicherplatzes. Speicheradressen gehören nicht zu Daten der Anwendung. Sie sind Verwaltungsinformationen, mit denen die Komponenten einer Struktur zu einer Einheit »zusammengewebt« werden. Wenn die Zusammengehörigkeit nicht durch einfaches Nebeneinanderlegen im Speicher ausgedrückt werden kann oder soll, dann werden Zeiger eingesetzt.

Programmiersprachen unterscheiden sich in der Behandlung von Zeigern. Für die einen sind Zeiger etwas, auf das man den Entwicklern keinen direkten Zugriff geben sollte, weil die damit bloß Schaden anrichten. Besser ist es, wenn diese *Pink-Latte*-Trinker sagen, welche Struktur sie haben möchten, und dann das System mit den Zeigern schon die richtigen Verbindungen knüpft. Das ist die Haltung der meisten anwendungsorientierten Sprachen und für die meisten Anwendungsentwickler ist sie völlig angemessen.

Für Leute, die nicht wissen, was *Pink Latte* ist, geschweige denn so etwas trinken, kommen richtige Programmiersprachen in Frage. Sprachen wie C, in denen man frei mit Zeigern hantieren und gegebenenfalls auch rechnen kann. Zeiger sind dort Speicheradressen als (fast) normale atomare Werte, die unter der direkten Kontrolle des Programms in eine Struktur eingebaut und manipuliert werden können.

Bei den Programmiersprachen gibt es also solche, in denen Zeiger richtige Werte sind, und solche, bei denen sie eine Sonderexistenz als rein interne Verwaltungsdaten im Hintergrund führen. Welche Haltung zu Zeigern ist die richtige im Pseudocode? Nun wie immer: Wenn es selbstverständlich oder irrelevant ist, dann wird nicht darüber geredet, wenn es nicht selbstverständlich und auch noch relevant ist, dann macht man ein Thema daraus.

Genau wie bei Programmiersprachen machen Zeiger die Diskussion auch in Pseudocode komplexer. Ohne zu tief in die Thematik gehen zu wollen, für das Verständnis von Zeigern ist der Unterschied zwischen Variablen und Werten wichtig.


Unterscheide *Werte* und *Variablen*!

- ✔ *Variablen* sind Speicherbereiche mit eventuell wechselnden Werten.
- ✔ Speicherbereiche haben eine *Adresse*.
- ✔ Zeiger sind Adressen als Werte: Wenn Adressen als Werte von einem Algorithmus manipuliert werden können, dann spricht man von *Zeigern*.

Arbeitet ein Algorithmus mit Zeigern, dann ist die ansonsten strikte Trennung der Welten von Nutz- und Verwaltungsinformationen und von Werten und Variablen aufgehoben. Man kann die Adresse einer Variablen zu einem Wert machen. Das schreiben wir als

$$ptr \leftarrow \textbf{address}\,(v)$$

Hier wird der Variablen ptr die Adresse von v als Wert zugewiesen. Zeiger sind also nichts anderes als Adressen von Speicherplätzen, mit denen der Algorithmus »rechnen« kann – Adressen als Werte. Sie werden oft grafisch als Pfeile dargestellt, die von der Speicherstelle des Zeigers auf die Variable mit der Adresse zeigen. In Abbildung 3.6 enthält eine Variable die Adresse 2. Dies wird als Pfeil auf die Speicherstelle mit der Adresse 2 dargestellt.

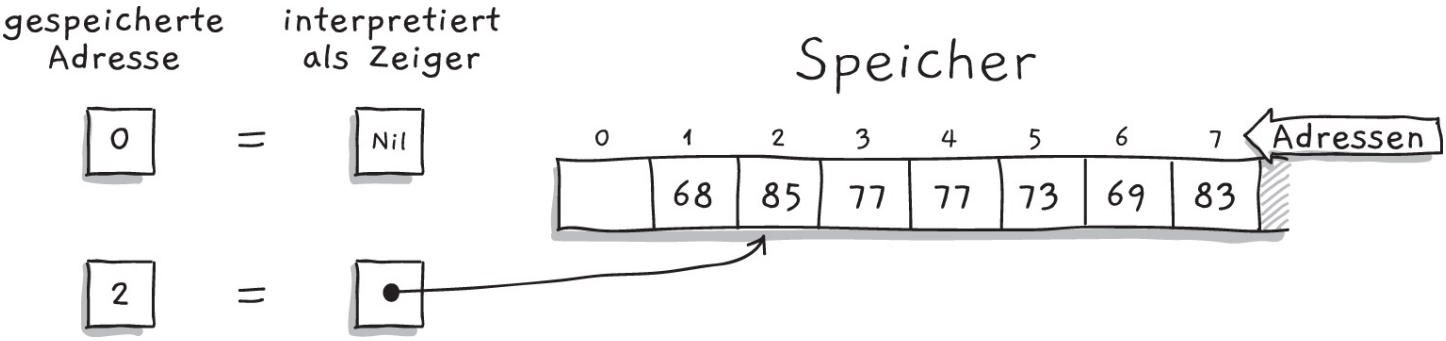


Abbildung 3.6: Zeiger, Variablen und Werte

Typen im Allgemeinen und die Typen von Zeigern ergeben sich in der Regel aus dem Kontext und werden darum meist nicht angegeben. In Zweifelsfällen kann der Typ aber

auch explizit definiert werden.

Die komplementäre Operation zu **address** ist **object**. Das hat hier nichts mit Objektorientierung zu tun. Wir meinen damit das Objekt, den Speicherbereich, die Variable, auf die ein Zeiger zeigt, die also eine bestimmte Adresse hat. In

object(ptr) ← 42

wird der Variablen, deren Adresse als Zeigerwert in ptr gespeichert ist, der Wert 42 zugewiesen. In

v ← 42

ptr > ← **address**(v)

object(ptr) ← **object**(ptr) + 1

wird der Variablen v im Endeffekt der Wert 43 zugewiesen.

Zeiger können mit den Variablen, auf die sie zeigen, neu erzeugt werden. Wir verwenden das Wort **new**. Beispiel:

ptr ← **new** Int

Die Variable ptr wird hier mit einen Zeiger auf einen neu erzeugten Speicherplatz belegt.

Mit Zeigern können beliebig komplexe Strukturen aufgebaut werden, deren elementare Bestandteile oft »Zellen« genannt werden. Zellen können als Records, Tupel oder Arrays realisiert werden. Nehmen wir Records, um eine verkettete Liste von Fakultäten zu erzeugen:

```
FList(n)
    type Ptr = Zeiger auf Cell
    type Cell = record[v:Int, n: Ptr]
    l: FL ← null
    f: Int ← 1                          1 = 0! in f speichern
    for i ← 1 ... n
        l ← new Cell(f, l)              neuen Record mit v=f und n=l anlegen und in l speichern
        f ← f · i                       nächste Fakultät berechnen
    return l
```

Ptr ist der Typ von Zeigern auf eine Speicherstelle vom Typ **Cell**. Der Null-Zeiger **null** ist ein besonderer Zeigerwert, der auf nichts zeigt und das Ende der Liste markiert.

Die Liste wird in unserem Beispiel rückwärts erzeugt: Sie beginnt mit der größten und zuletzt hinzugefügten Zahl (siehe Abbildung 3.7). Natürlich kann auch eine verkettete Liste erzeugt werden, die die Werte in aufsteigender Reihenfolge enthält:

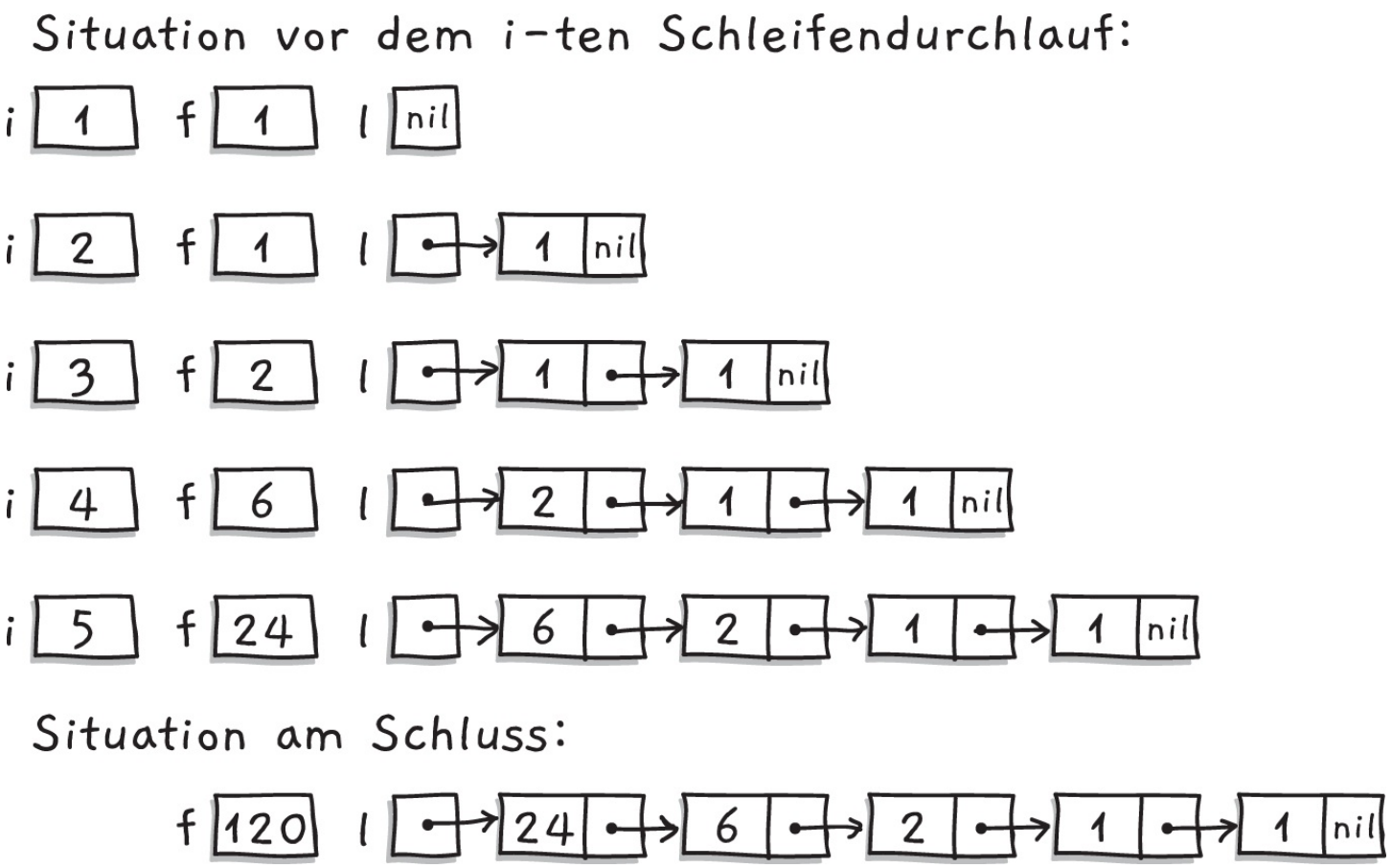


Abbildung 3.7: Ablauf von Algorithmus FList(5)

```
FListAE(n)
    a ← null                          Zeiger auf die erste Zelle
    e ← null                          Zeiger auf die letzte Zelle
    for i ← 0 ... n
        if i = 0
            a ← new Cell(1, null)
            e ← a
        else
            t ← new Cell(e.v · i, null)      neue Zelle erzeugen ...
            e.n ← t                          ... und hinten anhängen
            e ← t
    return a
```

Hier wird in den Variablen a und e ein Zeiger auf die erste beziehungsweise die letzte Zelle gespeichert. In der Schleife wird, außer beim ersten Durchlauf, der Wert der letzten Zelle mit i multipliziert, das Ergebnis in eine neue Zelle gesteckt und diese dann hinten angehängt. Beim ersten Durchlauf wird die erste Zelle erzeugt.

Der Umgang mit Zeigern ist etwas anspruchsvoll, aber dafür ist die Notation einfach und kompakt.



Zeiger sind Speicheradressen als Werte, mit denen man rechnen kann. Die Notation ist:

- ✔ Definition (*in natürlicher Sprache*): Zeiger auf …
- ✔ Zeigerwert erzeugen : **address**(*Speicherbereich/Variable*) oder : **new** *Typ einer Zelle*
- ✔ Speicherbereich zu Zeigerwert: **object**(*Zeiger-Wert*)

Der mühsame explizite Umgang mit Zeigern ist oft nur Routine. In einer Programmiersprache kann die Routine automatisiert werden. In Pseudocode muss nichts automatisiert werden, aber wir können die Notation so vereinfachen, dass nicht ständig alle Details ausgeführt werden müssen.

Algebraische Datentypen

Das Programmieren mit Zeigern ist etwas kniffelig und erfordert eine hohe Konzentration, auch schon bei der Konstruktion einfacher Strukturen. Einmal nicht aufgepasst und schon wird man mit einer *Null-Pointer-Exception* bestraft! Oder noch schlimmer, man führt ungewollt ein bösartiges Programm aus, das Daten in Stellen im Speicher schreibt, die es eigentlich nichts angehen. Mit Zeigern kann man vieles machen, auch viele Fehler. Um diese Situation zu verbessern, wurden in modernen Programmiersprachen die Verwendungsmöglichkeiten von Zeigern eingeschränkt und durch Automatismen vereinfacht.

Denken wir an das Beispiel der Liste von Fakultäten, die wir oben als Verkettung von Zellen realisiert haben:

type FList = Zeiger auf Cell
type Cell = **record**[v:Int, n: FList]
l: FList ← **null**

Der **null** -Zeiger repräsentiert die leere Liste. Jede Zelle enthält zwei Werte: eine Zahl und einen Zeiger auf die nächste Zelle oder **null** (siehe Abbildung 3.8).

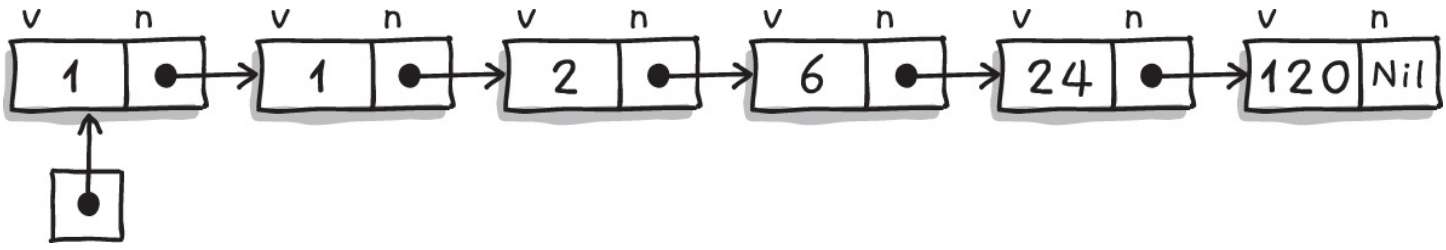


Abbildung 3.8: Liste als verkettete Zellen

Hierarchische Strukturen einfach definieren

Eigentlich ist das nichts anderes als eine mühselige Art, etwas Einfaches auszudrücken: Eine Liste von Fakultäten vom Typ FList ist

- ✔ *entweder* leer
- ✔ *oder* besteht aus zwei Werten:
  - einer Zahl *und*
  - einer weiteren FListe.

Eine Fakultäten-Liste wird aufgebaut, indem wir eine Zelle bilden, also eine Komponente *und* eine andere Komponente zusammenfügen oder eine Auswahl treffen: **null** -Zeiger *oder* ein Zeiger auf eine weitere Zelle.

Diese *Und-Oder-Struktur* im Typ einer Fakultätenliste kann man klarer und kürzer darstellen als:

```
type FList = Cons(Int, FList) | Nil
```

Ein Objekt vom Typ FList ist *entweder* ein Cons *oder* ein Nil. Der Strich (|) steht dabei für »oder«. Ein Cons hat zwei Bestandteile: Einen Int-Wert *und* ein weiteres FList-Objekt. Ein Nil hat keine Bestandteile.

Cons und Nil sind Konstruktor-Funktionen mit beliebig gewählten Namen. Statt Cons hätten wir auch makeCell oder makeNode oder irgendwas anderes sagen können. Das Gleiche gilt für Nil. **Cons** und **Nil** sind traditionelle Bezeichnungen, die von der Programmiersprache Lisp eingeführt wurden und seit Langem für derartige Konstrukte verwendet werden. Man kann sich Cons als eine Funktion vorstellen, die eine Zelle erzeugt, und Nil als eine Funktion, die ein beliebiges Objekt ohne Bestandteile erzeugt, beispielsweise in Form eines **null**-Zeigers (siehe Abbildung 3.9).

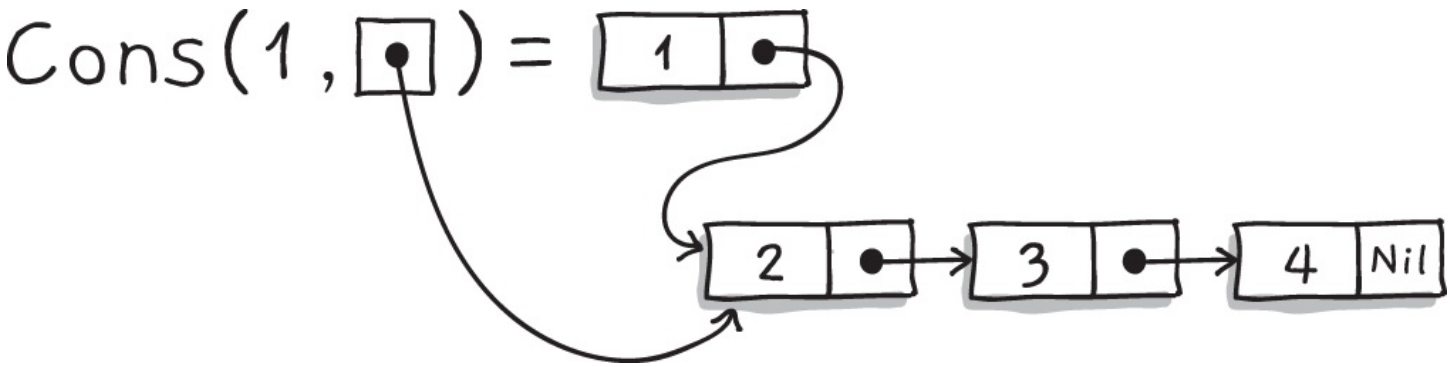


Abbildung 3.9: Cons-Implementierung: Erzeuge eine Speicherzelle mit Wert und Zeiger.

Statt »Cons« werden häufig auch ein oder zwei Doppelunkte (::) als Infix-Operator verwendet. Also beispielsweise

$$lst \leftarrow x :: y :: \text{Nil}$$

statt

$$lst \leftarrow \text{Cons}(x, \text{Cons}(y, \text{Nil})).$$

Ein etwas anderes Beispiel für eine solche Definition im Und-Oder-Stil sind Eierkartons. Ein Eierkarton enthält 6 Plätze, an denen vielleicht ein Ei zu finden ist oder eben auch mal keins.

```
type Eierkarton = Karton(VE, VE, VE, VE, VE, VE)      6-mal vielleicht ein Ei
type VE = Ei | Nil                                    vielleicht ein Ei: ein Ei oder nichts
type Ei = ...
```

Hier wird nichts über ein Ding der Wirklichkeit gesagt, sondern es wird die Struktur einer bestimmten Art von Datenstrukturen definiert. Wie **Cons** oben ist **Karton** wieder der beliebig gewählte Name einer Konstruktor-Funktion mit sechs Argumenten. Jedes Argument muss ein Ei sein oder ein Nil.

Die Implementierung einer so definierten Struktur könnte aus einem Tupel mit sechs Zeigern bestehen, die auf ein Ei-Objekt zeigen oder auf ein leeres Tupel. Leere Tupel repräsentiert dann Nil, das inhaltslose Objekt. Statt eines Zeigers auf ein leeres Tupel (Nil) könnten man auch den Null-Zeiger verwenden.



Nil und der Null-Zeiger

Nil ist ein üblicher, aber im Prinzip willkürlich gewählter Name für etwas, das da ist, aber keinerlei weitere Informationen enthält. So wie beispielsweise das Ende einer Liste – es ist da, aber mehr gibt es nicht darüber zu wissen – oder der leere Platz in einem Eierkarton. Da ist nichts, mehr gibt es über den Platz gerade nicht zu wissen.

Der **null** -Zeiger ist eine mögliche Art, dies effizient zu implementieren. Wenn der Typ einer Datenstruktur definiert wird, dann braucht man die Möglichkeit zu sagen, dass es Stellen gibt, an denen Komponenten einer bestimmten Art zu finden sind, die Stelle aber auch leer sein kann. Bei einer Implementierung einer solchen Struktur mit Zeigern wird dann ein Zeiger auf eine völlig leere Zelle zeigen. Stattdessen kann man die leere Zelle weglassen und den Zeiger auf sie durch den **null** -Zeiger ersetzen.

Diese feinen Unterschiede werden im Allgemeinen aber ignoriert. **null** und Nil werden identifiziert, obwohl sie sich auf unterschiedlichen Abstraktionsniveaus befinden. Nil ist das Konzept des Behälters, der nichts enthält, **null** ist der Zeiger, der auf nichts zeigt.

Die Definition von FList legt fest, dass es sich um eine hierarchische Struktur handelt. Sie kann und wird typischerweise in Form »verzeigerter« Zellen realisiert. Das ist aber nicht zwingend. Andere Speicherformen sind ebenfalls denkbar: als verschachtelte Zellen, wie auch immer. Hauptsache man erkennt, um was es sich handelt, und kann die Komponenten identifizieren (siehe Abbildung 3.10).

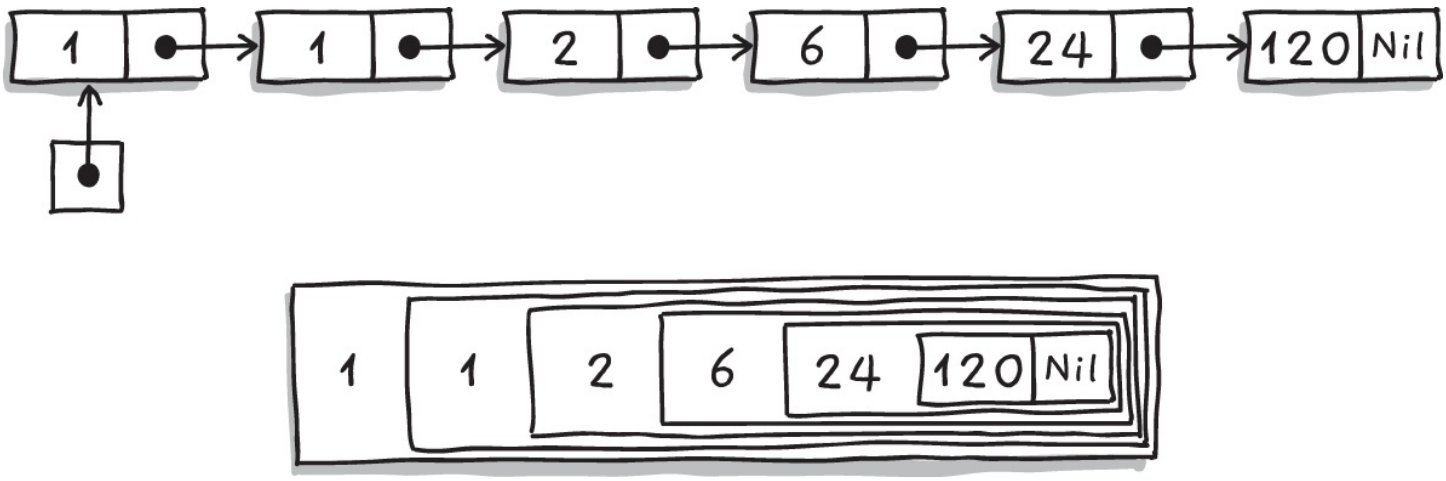


Abbildung 3.10: Das Wesentliche: die hierarchische Struktur einer Liste

Wie genau Konstruktor-Funktionen wie **Cons** arbeiten und was genau sie erzeugen, wird also offen gelassen. Es ist irrelevant. Wichtig ist nur, dass für jedes FList-Objekt klar ist,

- ✔ mit welcher der Konstruktor-Funktionen, **Cons** oder **Nil** , es erzeugt wurde und
- ✔ falls es mit **Cons** erzeugt wurde, welche Argumente an **Cons** übergeben wurden.

Die *Konstruktion* muss also eine eindeutige *Dekonstruktion* zulassen. Konkret kann man sich immer eine Konstruktion mit Zellen und Zeigern vorstellen.

Der Algorithmus zur Erzeugung der Liste der Fakultäten bis n kann damit wie folgt definiert werden:

```
type FL = Cons(Int, FL) | Nil

FList(n)
    l ← Nil
    for i ← 0 ... n
        l ← Cons(F(i), l)
    return l
F(x)
    if x = 0
        return 1
    else
        return x·F(x − 1)
```

Ohne dabei genauer darauf eingehen zu müssen, was genau Cons und Nil sind. Wir wissen, wie es geht, und das mag reichen. Die übliche Implementierung arbeitet mit Zeigern. Die Definition wird meist als verkettete Struktur realisiert. In einer Programmiersprache dient es der Bequemlichkeit, wenn man nicht über Zeiger reden muss. Im Pseudocode steckt noch mehr dahinter.



Weiter oben haben wir argumentiert, dass es für asymptotische Laufzeitabschätzungen oft unerheblich ist, ob eine Struktur verkettet oder zusammenhängend gespeichert wird.

Das bedeutet, dass wir nicht nur von Details verschont werden wollen, wenn wir die verzeigerte und die verschachtelte Struktur identifizieren, sondern dass diese Details auch unter Umständen und zu einem gewissen Grad tatsächlich unerheblich sind.

Hierarchische Strukturen einfach zerlegen

Ein Algorithmus, der eine solche Und-Oder-Struktur nicht aufbaut, sondern analysiert, muss die Möglichkeit haben festzustellen, wie sie aufgebaut wurde, also welche Alternative gewählt wurde und aus welchen Komponenten eine Struktur zusammengesetzt wurde. Die entsprechenden Ausdrucksmittel zeigen wir am Beispiel eines Algorithmus, der die Summe der Elemente einer verketteten Liste von ganzen Zahlen bestimmt. Die Liste definieren wir nicht mit Zeigern, sondern als **Cons** -oder- **Nil** -Struktur.

```
type IntList = Cons(Int, IntList) | Nil

ListSum(lst)                          lst ist eine IntList
    s ← 0
    while lst ≠ Nil
        case lst = Cons(v, rest)
            s ← s + v                 addiere erstes Element zu s
            lst ← rest                gehe weiter zum Rest
    return s
```

Hier sieht man den Nutzen der **case** -Notation: Wenn lst mithilfe von **Cons** aus einen Wert v und einer IntList $rest$ erzeugt wurde, dann addiere v zu s und weise $rest$ der Variablen lst zu.

Man beachte, dass es völlig offen bleiben kann, was **Cons** tatsächlich treibt. Es kann einen Record oder ein Tupel mit Zeigern erzeugt haben oder eine verschachtelte Struktur oder einen Mix aus verschachtelt und verkettet. Wie aus immer. Egal. Was immer da wie auch immer zusammengebaut wurde, es kann wieder zerlegt werden. Mit der **case** -Notation sehen wir das Ergebnis der Zerlegung.

Statt der Schleife kann man zum Durchlaufen einer rekursiv definierten Struktur einen rekursiv definierten Algorithmus verwenden:

```
ListSumR(lst)                         lst ist eine IntList
    case lst = Nil                    die Summe der Werte einer leeren Liste ist 0
        return 0
    case lst = Cons(v, rest)          erstes Element + Summe des Rests der Liste
        return v + ListSumR(rest)
```

Strukturen, die ausschließlich mithilfe von *und* (Zusammenfügung) und *oder* (Auswahl) aus Elementen von vorgegebenen Typen erzeugt werden, nennt man *algebraische Datentypen*, kurz *alg. DT* oder auch gelegentlich *ADT*. Jede Datenstruktur, die als alg. DT definiert wurde,

- ✔ hat eine hierarchische, baumartige Struktur und
- ✔ kann mit Zeigern als zykelfreie verkettete Struktur implementiert werden.

Umgekehrt ist aber nicht jede verzeigerte Struktur ein algebraischer Datentyp: Zyklische Strukturen und Strukturen, deren Elemente veränderlich sind, sind nicht algebraisch.



Algebraische Datentypen (alg. DT) sind die Typen von Datenstrukturen, die aus gegebenen Typen ausschließlich mit Zusammenfügung und Auswahl gebildet werden. Sie sind hierarchische Strukturen, die mit unveränderlichen und zykelfreien verketteten Strukturen implementiert werden können.

Verkettete Strukturen, deren Komponenten *veränderlich* sich, und solche, die *Zyklen* enthalten, können *nicht* durch algebraische Datentypen beschrieben werden. Das sind die komplizierteren Fälle. Da kommen wir nicht darum herum, explizit mit Zeigern zu arbeiten. Viele Algorithmen arbeiten aber mit verketteten Strukturen, die nur aufgebaut, aber dann nicht mehr in ihrem Inneren verändert werden und die keine Zyklen enthalten. In dem Fall verwenden wir die Notation der algebraischen Datentypen. In der funktionalen Programmierung sind Veränderungen verboten. Algebraische Datentypen sind darum dort besonders beliebt.

Funktion folgt Struktur

Ein bekannter Grundsatz des Designs lautet *Form follows function*. Damit ist gemeint, dass die Dinge so gestaltet werden sollen, dass sie ihrer Funktion entsprechen. Wenn man kein Designer ist und die Dinge in der Form benutzen muss, die sie haben, dann bleibt oft keine Wahl: das Motto muss umgekehrt werden, die Funktion folgt der Form. Hat eine Straße noch so viele völlig überflüssige Kurven, wir können nicht einfach geradeaus fahren, wir müssen ihr folgen. Genauso geht es den Algorithmen, die auf Datenstrukturen operieren. Wenn möglich, folgen sie deren Struktur und haben es leicht.

Algorithmen können Strukturen aufbauen und sie nutzen. Beim Aufbau kann und sollte man an die Nutzung denken. Bei der Nutzung muss man sich an die Struktur anpassen.

Strukturen ohne Größenbeschränkung, potenziell unendlich große Strukturen, sind immer rekursiv aufgebaut. Und so wie ich beim Fahren dem Kurvenverlauf der Straße folge, so folgt ein Algorithmus auf einer rekursiven Struktur sinnvollerweise deren rekursiver Form. Zunächst aber werfen wir einen genaueren Blick auf die Rekursion. Es gibt

Rekursion bei Algorithmen und bei Daten. Bei den Algorithmen kann man noch verschiedene Arten unterscheiden:

- ✔ Endrekursion, das ist eine Rekursion, die eigentlich eine Schleife ist, die sich als Rekursion herausgeputzt hat.
- ✔ Lineare Rekursion, das ist die Rekursion der Fakultätsfunktion: Ein rekursiver Aufruf im rekursiven Fall.
- ✔ Allgemeine Rekursion, da gibt es keinerlei Beschränkungen.

Endrekursive und linear-rekursive Funktionen

Algorithmen können rekursiv definiert werden. Ein Algorithmus ist rekursiv (definiert), wenn er mithilfe von sich selbst definiert wird. Man gibt den Wert für einfache Fälle direkt an und berechnet den Wert für komplexe Fälle aus dem Wert für einfachere Fälle. Nehmen wir die Berechnung des größten gemeinsamen Teilers (ggt). Der ggt von zwei natürlichen Zahlen x und y ist x, wenn die beiden gleich sind, ansonsten ist es der ggt von der größeren minus der kleineren und der kleineren.

```
GGTRec(m, n)
    if m = n
        return n
    else if m > n
        return GGTRec(m − n, n)
    else
        return GGTRec(m, n − m)
```

Oder noch einfacher:

```
GGTRec(m, n)
    if m = n
        return n
    else
        return GGTRec(max(m, n) − min(m, n), min(m, n))
```

Die elementare Hardware beherrscht keine Rekursion. Rekursion ist aber eine außerordentlich wichtige Operation bei der Definition von Algorithmen und sie kann in eindeutiger Art in elementare Konstrukte umgewandelt werden. Die rekursive Definition von Algorithmen wird darum genau wie die rekursive Definition von Datenstrukturen im Pseudocode zugelassen.

Rekursion in Algorithmen wird mithilfe von Sprunganweisungen implementiert, bei denen von einer Anweisung zu einer anderen Anweisung mit einer bestimmte Adresse gesprungen wird. Sprunganweisungen sind bei den Algorithmen das, was die Zeiger bei den Datenstrukturen sind. Rekursion in Algorithmen basiert auf dem gezähmten, vom

Compiler erzeugten Herumspringen im Code, genauso wie rekursiv definierte alg. DTen auf gezähmten, vom Compiler erzeugten Zeigern basieren.

Jeder rekursive Algorithmus kann ohne Rekursion implementiert werden. Manchem rekursiven Algorithmus entspricht unmittelbar eine Schleife. Andere können mit etwas mehr Aufwand in eine oder zwei Schleifen umgewandelt werden. Mithilfe eines Stapels und Sprunganweisungen kann letztlich jeder rekursive Algorithmus in einen äquivalenten nicht-rekursiven umgewandelt werden. Alle Implementierungen von Programmiersprachen mit Rekursion beherrschen diese Technik.

Der Algorithmus GGTRec oben ist einer von den harmlosen Fällen. Er ist völlig äquivalent zu folgender iterativen Variante:

```
GGTIter(x, y)
    while x ≠ y
        if x > y
            x ← x − y
        else
            y ← y − x
    return x
```

Mit einer Zuweisung von Tupeln sieht man die Ähnlichkeit mit der endrekursiven Variante noch deutlicher:

```
GGTIter(x, y)
    while x ≠ y
        (x, y) ← (max(x, y) − min(x, y), min(x, y))
    return x
```

Beide Varianten des Algorithmus führen auch die gleiche Anzahl an elementaren Schritten aus, also gleich viele Vergleiche und Subtraktionen. Statt der Zuweisungen haben wir jetzt Parameterübergaben, was sich aber nicht wesentlich unterscheidet (siehe Abbildung 3.11).

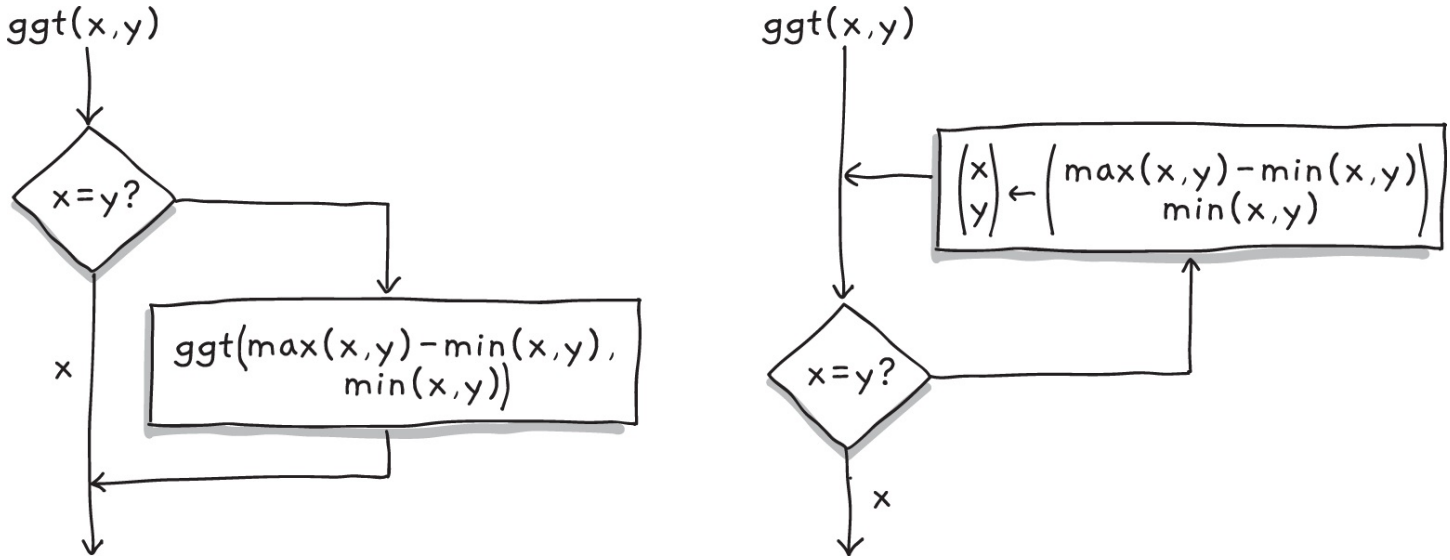


Abbildung 3.11: GGT: iterative und rekursive Version

Rekursive Algorithmen, die in dieser Art direkt in Schleifen umgewandelt werden können, bezeichnet man als *endrekursiv*. Sie sind dadurch gekennzeichnet, dass der rekursive Aufruf immer die letzte Aktion ist. Ein Beispiel für einen *nicht* endrekursiven Algorithmus ist die Berechnung der Fakultät:

```
FRec(n)                                  NICHT endrekursiver Algorithmus
    if n = 0
        return 1
    else
        return FRec(n−1) ·n              Multiplikation nach dem rekursiven Aufruf!
```

Nach dem rekursiven Aufruf wird dessen Ergebnis noch mit n multipliziert.

Bei einem *linear-rekursiven* Algorithmus darf der rekursive Zweig höchstens *einen* rekursiven Aufruf enthalten. Dieser darf aber, anders als bei der Endrekursion, in eine andere Operation eingebettet sein. FRec oben ist linear-rekursiv.

Bei endrekursiven Funktionen geht es immer tiefer (oder höher) in die Rekursion hinein. Irgendwann ist die Sache dann in den tiefsten Tiefen oder höchsten Höhen der Rekursion mit einem Schlag zu Ende. Bei der linearen Rekursion geht es mit einer »nachhängenden Operation« wieder Schritt für Schritt aus der Rekursion hinaus (siehe Abbildung 3.12).

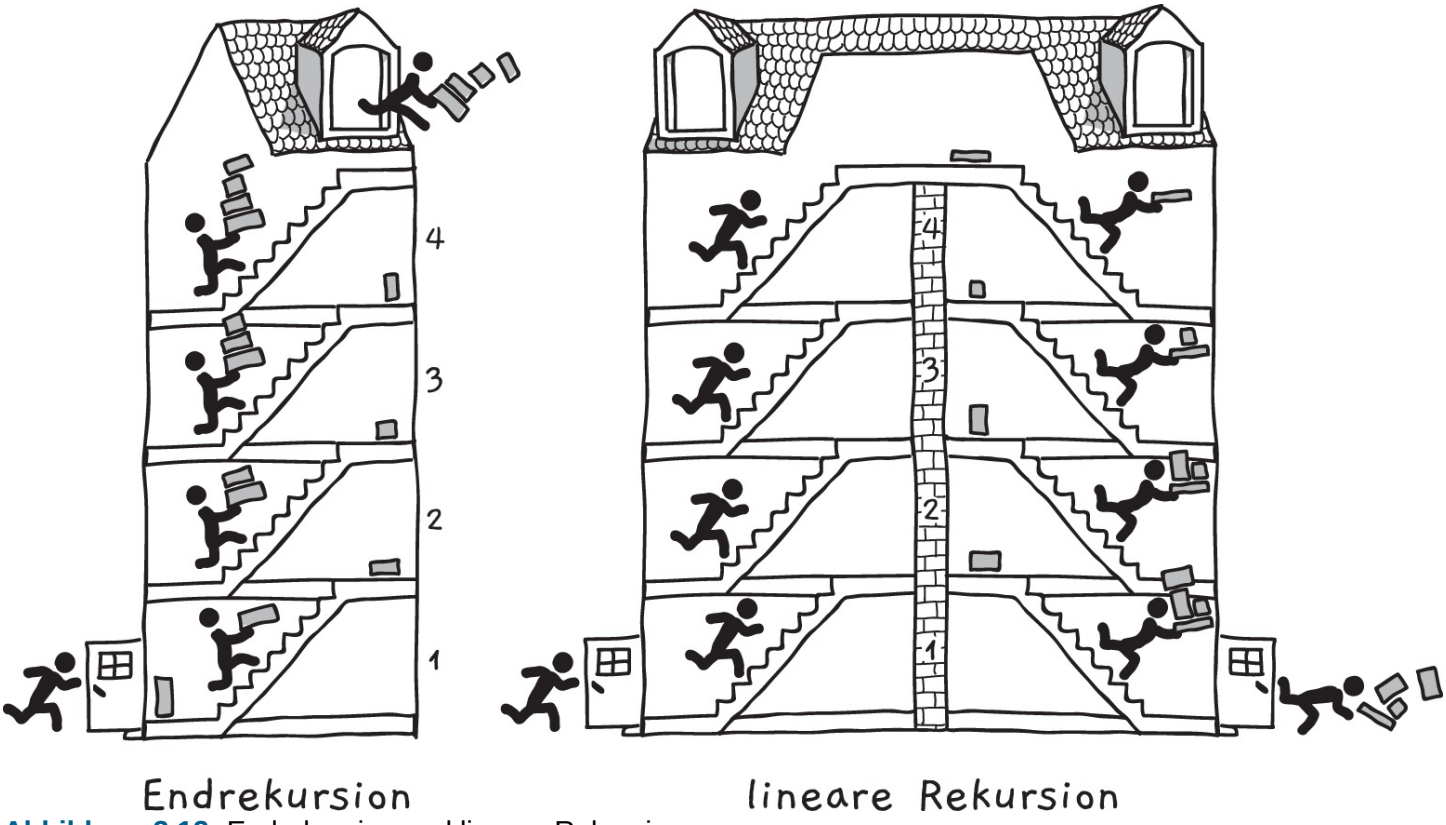


Abbildung 3.12: Endrekursion und lineare Rekursion



- Schema einer *endrekursiven* Funktion:

$$f(x) = \begin{cases} g(x) & \text{falls } p(x) \\ f(h(x)) & \text{sonst} \end{cases}$$

- Schema einer *linear-rekursiven* Funktion:

$$f(x) = \begin{cases} g(x) & \text{falls } p(x) \\ h_2(f(h_1(x)), x) & \text{sonst} \end{cases}$$

Die Fakultätsfunktion ist eine linear-rekursive Funktion nach diesem Schema mit

$$p(x) = x = 0 \quad g(x) = 1 \quad h_1(x) = x - 1 \quad h_2(x, y) = x \cdot y$$

h_2 ist dabei die »nachhängende« Operation.

Linear-rekursive Funktionen und die Akkumulatortechnik

Bei linear-rekursiven Funktionen wird die Arbeit in zwei Phasen geleistet: Beim Einstieg in die Rekursion wird Phase eins abgewickelt. Der Ausstieg geht Schritt für Schritt und dabei werden die Berechnungen der zweiten Phase ausgeführt.

Linear-rekursive Berechnungen können auch oft recht einfach in eine äquivalente Schleife umgewandelt werden. Nehmen wir als Beispiel die linear-rekursive Fakultätsfunktion. Zunächst wird die lineare in eine Endrekursion umgewandelt:

```
FTR(n)
    return f(n, 1)

f(n, acc)
    if n = 0
        return acc
    else
        return f(n − 1, acc · n)
```

Hier wird mit einem Akkumulator-Parameter acc gearbeitet. Im *Akkumulator* werden die Zwischenergebnisse aufgesammelt. Beim Herausgehen aus der Rekursion ist darum keine weitere Aktion mehr erforderlich. Diese endrekursive Variante kann jetzt quasi mechanisch in eine Schleife umgewandelt werden:

```
FIter(n)
    f ← 1
    while n ≠ 0
        (n, f) ← (n − 1, n · f)
    return f
```

Auch hier kann man sich davon überzeugen, dass die Zahl der ausgeführten elementaren Operationen in jeder Variante die gleiche ist. Man muss dazu Zuweisungen und Parameterübergaben bei einem Funktionsaufruf als gleichwertig betrachten und den reinen Aufruf einer Funktion genauso behandeln wie die Rückkehr an den Anfang einer Schleife.

Bei einer linearen Rekursion werden Aktionen beim Auf- und beim Abstieg ausgeführt. Die Akkumulatortechnik vereint beide Aktionen zu einer. Das geht aber nur, wenn es unerheblich ist, in welcher Reihenfolge die Aktionen ausgeführt werden. Bei Multiplikationen ist das der Fall. Berechnet man beispielsweise $4!$, dann wird beim Einsatz von FRec

$$((((1 \cdot 1) \cdot 2) \cdot 3) \cdot 4)$$

berechnet, beim Einsatz von FIter dagegen

$$(1 \cdot (2 \cdot (3 \cdot (4 \cdot 1)))).$$

Das Ergebnis ist das Gleiche, aber nur weil die Multiplikation assoziativ ist. Hätten wir es mit einer nicht-assoziativen Operation zu tun, beispielsweise mit einer Subtraktion, dann würde das nicht gelten.

Im Allgemeinen benötigt man zwei Schleifen, um eine lineare Rekursion in Schleifenform zu bringen. Eine, die die Berechnungen ausführt, die beim Einstieg in die Rekursion ausgeführt werden, und eine für die, die zum Ausstieg aus der Rekursion gehören. Auch das lässt sich leicht an der Fakultätsfunktion demonstrieren. Spalten wir die Berechnung in zwei Schritte auf: Zuerst werden die zu multiplizierenden Werte aufgesammelt, das entspricht den Aktionen beim Einstieg in die Rekursion. Im zweiten Schritt werden die gesammelten Werte aufmultipliziert. Das entspricht den Aktionen beim Ausstieg aus einem rekursiven Aufruf.

```
type NList = Cons(Int, NList) | Nil      Liste der zu multiplizierenden Werte

FTR2(n)
    return f2(f1(n, Nil), 1)
```

```
f1(n, acc1: NList)                          erst Werte aufsammeln ...
    if (n = 0)
        return acc1
    else
        return f1(n − 1, Cons(n, acc1))
f2(rev: NList, acc2: Int)                   ... dann Multiplikationen ausführen
    case rev = Nil
        return acc2
    case rev = Cons(first, rest)
        return f2(rest, acc2 · first)
```

Da f1 und f2 endrekursiv sind, können sie unmittelbar in Schleifen umgewandelt werden und wir erhalten eine iterative Version des Algorithmus:

```
type NList = Cons(Int, NList) | Nil

FIter2(n)
    acc1 ← Nil                              f1 als Schleife, in der alle Faktoren gesammelt werden
    while n ≠ 0
        acc1 ← Cons(n, acc1)
        n ← n − 1
    acc2 ← 1                                f2 als Schleife, in der alle Faktoren aufmultipliziert werden
    rev ← acc1
    while rev ≠ Nil
        case rev = Cons(head, rest)
            acc2 ← acc2 · head
            rev ← rest
    return acc2
```

Algorithmen, die nicht endrekursiv oder wenigstens linear-rekursiv sind, können im Allgemeinen nur mit Stapeln iterativ abgewickelt werden. Wie dies im Einzelnen geht, ist Thema des Compilerbaus. Wir wollen uns hier nicht weiter damit beschäftigen. Es reicht zu wissen, dass rekursive Algorithmen in nicht-rekursive Programme der RAM-Maschine umgewandelt werden können. Damit ist die algorithmische Rekursion ebenso ein legitimes Ausdrucksmittel des Pseudocodes wie algebraische Datentypen, die rekursiv und ohne explizite Erwähnung von Zeigern definierten Datenstrukturen.



Zeiger sind die »Sprunganweisungen « der Strukturen, Sprunganweisungen sind die »Zeiger der Algorithmen«. Algorithmen und Datenstrukturen können sich elegant in Rekursionen kleiden und so diese plumpen Rohheiten verbergen.

Strukturelle Rekursion

Nicht nur Algorithmen können rekursiv definiert werden, auch bei Daten wird Rekursion sehr häufig als Definitionsmittel eingesetzt. Die beiden, die algorithmische Rekursion und die Datenrekursion, harmonieren oft aufs Beste. Das ist nicht verwunderlich: Viele Algorithmen, die hierarchische Strukturen verarbeiten, folgen letztlich nur deren Struktur.

Betrachten wir eine einfache rekursiv definierte Struktur, die Liste als alg. DT, und einen einfachen Algorithmus, der auf dieser Struktur arbeitet. Die Summe der Elemente einer Liste, die definiert wurde als

type IntList = Cons(Int, IntList) | Nil

kann mit folgendem Algorithmus berechnet werden:

```
ListSum(lst)
    case lst = Nil
        return 0
    case lst = Cons(first, rest)
        return first + ListSum(rest)
```

Man sieht, wie sich die Struktur der Daten in der Struktur des Algorithmus wiederfindet. Ein etwas komplexeres Beispiel für eine rekursiv definierte Datenstruktur sind Binärbäume. Ein Binärbaum ist ein Baum, dessen Blätter Werte enthalten und dessen Knoten je zwei Unterbäume enthalten. Binärbäume können als alg. DT definiert werden:

type IntTree = Node(IntTree, IntTree) | Leaf(Int)

Ein Binärbaum vom Typ IntTree ist hier also entweder ein Node oder ein Leaf. Wenn er ein Node ist, dann hat er zwei Komponenten vom Typ IntTree, ist er ein Leaf, dann hat er einen Int-Wert als Komponente.

Die Summe aller Blätter in einem IntTree kann leicht mit struktureller Rekursion berechnet werden (siehe Abbildung 3.13):

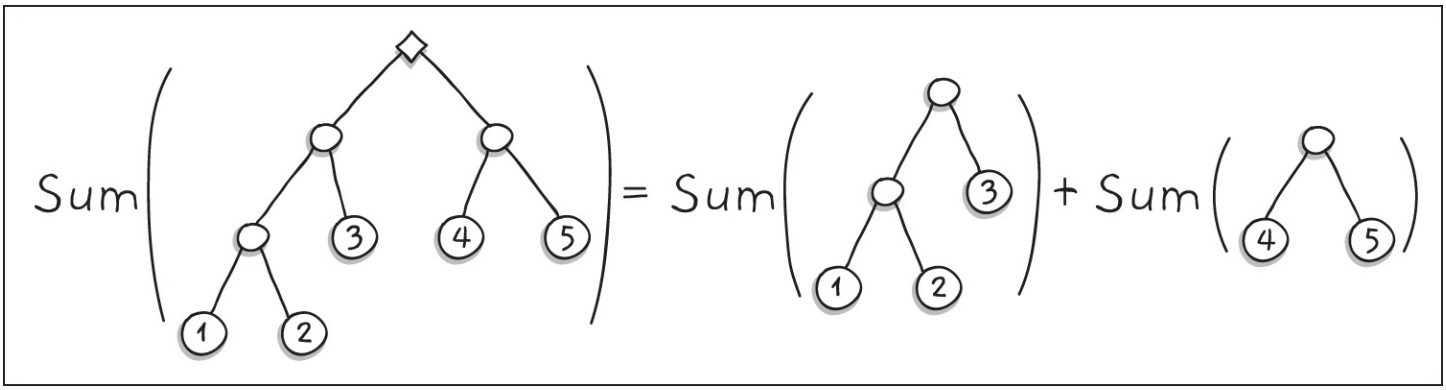


Abbildung 3.13: Strukturelle Rekursion: Funktion folgt Form.

```
TreeSum(tree)
    case tree = Leaf(v)
        return v
    case tree = Node(left, right)
        return TreeSum(left) + TreeSum(right)
```

Viele Algorithmen auf Datenstrukturen folgen dem Schema der strukturellen Rekursion, zum Beispiel *InsertionSort*, das Sortieren durch Einfügen:

```
InsSort(lst)                                  InsertionSort
    case lst = Nil
        return Nil
    case lst = Cons(first, rest)
        return insert(first, InsSort(rest))   füge erstes Element im sortierten Rest ein

insert(x, lst)                                füge x am richtigen Platz in lst ein
    case lst = Nil
        Cons(x, Nil)
    case lst = Cons (first, rest)
        if x < first
            return Cons(x, lst)
        else
            return Cons(first, insert(x, rest))
```

Während eine Datenstruktur durchlaufen wird, kann gleichzeitig eine andere aufgebaut werden. Beispielsweise kann ein Baum durchlaufen und dabei in eine Liste transformiert werden:

```
treeToList(tree)
    case tree = Leaf(v)
        return Cons(v, Nil)
    case tree = Node(left, right)
        return append(treeToList(left), treeToList(right)

append(lst1, lst2)
    case lst1 = Nil
        return lst2
    case lst1 = Cons(first1, rest1)
        return Cons(first1, append(rest1, lst2)
```

Teilen und Herrschen

Viele, aber nicht alle Algorithmen auf hierarchischen Datenstrukturen folgen dem Schema der strukturellen Rekursion. Die Rekursion muss immer vom Komplexeren oder Größeren zum Einfacheren oder Kleineren gehen und schließlich in einem Basisfall enden. Dabei

geht es oft, aber nicht zwingend immer, von einer Struktur zu ihren Teilstrukturen.

Der QuickSort-Algorithmus ist ein interessantes Beispiel für eine Rekursion, die *nicht* strukturell ist. Bei diesem Algorithmus wird nicht wie bei InsertionSort der Rest der Liste sortiert und der sortierte Rest dann genutzt, um eine sortierte Gesamtliste zu konstruieren. Stattdessen werden zwei kürzere Listen sortiert und diese dann genutzt, um die Gesamtlösung zu konstruieren. Die beiden waren aber nicht (!) Bestandteil der Ausgangsliste. Die eine Liste enthält die Elemente, die kleiner oder gleich einem ausgewählten *Pivot*-Element sind, die zweite enthält diejenigen, die größer sind als das Pivot-Element. Das Pivot-Element ist irgendein beliebiges Element der Liste, der Einfachheit halber oft das erste Element der Liste. (Man kann übrigens »Pivot« sowohl französisch (»piwoh«) also auch englisch (»piwit«) aussprechen, es bedeutet auf Deutsch so viel wie »Dreh- und Angelpunkt«.)

```
Quicksort(lst)
    case lst = Nil
        return Nil
    case lst = Cons(first, Nil)
        return lst
    case lst = Cons(first, rest)
        pivot ← first
        l1 ← lessEq(pivot, rest)        Elemente von lst ≤ pivot
        l2 ← larger(pivot, rest)        Elemente von lst > pivot
        return append(Quicksort(l1), Quicksort(l2))

lessEq(x, l)                            Elemente von l, die kleiner oder gleich x sind
    case l = Nil
        return Nil
    case l = Cons(first, rest)
        if first ≤ x
            return Cons(first, lessEq(x, rest))
        else
            return lessEq(x, rest)

larger(x, l)                            Elemente von l, die größer als x sind
    case l = Nil
        return Nil
    case l = Cons(first, rest)
        if first > x
            return Cons(first, larger(x, rest))
        else
            return larger(x, rest)
```

```
append(lst1, lst2)                    verknüpfe lst1 und lst2 zu einer Liste
    case lst1 = Nil
        return lst2
    case lst1 = Cons(first1, rest1)
        return Cons(first1, append(rest1, lst2)
```

Achtung, diese Variante von QuickSort funktioniert nicht, wenn das erste Element einer Teil-Liste größer gleich allen anderen ist. In dem Fall ist $l2$ leer und $l1$, die Liste der Elemente, die kleiner oder gleich dem ersten sind, umfasst alle Elemente. Die Rekursion macht dann keine Fortschritte. Das Problem lässt sich aber leicht beheben. Man teilt die Liste in drei Teile: Die Elemente, die kleiner sind als das Pivot-Element, die Liste, die nur aus dem Pivot-Element besteht, und die Liste der Elemente, die größer sind als das Pivot-Element.


QuickSort wird ausführlich im Abschnitt *QuickSort: Quick and Easy* in Kapitel 7 behandelt.

Strukturen durchlaufen: Iteratoren und Traversierungen

In sehr vielen Algorithmen werden Datenstrukturen durchlaufen und dabei deren Elemente der Reihe nach verarbeitet. Ein einfaches Beispiel ist das Aufsummieren aller Elemente einer Liste oder eines Arrays. Bei einem Array können wir einfach eine Schleife über alle Indizes einsetzen:

```
ArraySum(a)
    s ← 0
    case a = array[0 .. n − 1]
        for s ← 0 .. n − 1
            s ← s + a[i]
    return s
```

Noch einfacher können wir sagen, dass alle Elemente verarbeitet werden sollen, wenn wir eine Notation verwenden, die auf die explizite Angabe der Indizes verzichtet:

```
ArraySum(a)
    s ← 0
    for x ← a                         für jeden Wert x in a, klar: natürlich
        s ← s + x                     wird auf x mit einem Index zugegriffen
    return s
```

Es ist klar, dass hier mit x Elemente des Arrays gemeint sind, auch wenn tatsächlich mit Indizes gearbeitet werden muss: Ein Index i muss über das Array laufen und die x-Werte müssen über den Index ($x \leftarrow a[i]$) aus dem Array geholt werden. Solche Details interessieren aber oft nicht: Irgendein Mechanismus läuft über das Array und holt dabei ein Element nach dem anderen. Das ist einfach und klar. Es kann mit einem Laufindex

implementiert werden und das muss man nicht ständig neu erklären.

Ein derartiges Durchlaufen einer Struktur nennt man *Traversieren*. Mit dem Index wird ein Array traversiert. Es handelt sich dabei um eine externe Traversierung. Die Kontrolle liegt nicht in der Struktur, sondern außerhalb. Die Traversierung könnte ja jederzeit mit einem **return** abgebrochen werden.

Mechanismen zur externen Traversierung einer Struktur nennt man Iteratoren. Ein *Iterator* ist eine Verallgemeinerung eines Index: Ein Mechanismus, der sich in einer Struktur auskennt und den man losschicken kann (*next!*), um ein Element nach dem anderen herbeizuschaffen (siehe Abbildung 3.14). In der zweiten Variante von ArraySum setzen wir also einfach voraus, dass jedes Array einen Iterator hat. Das ist legitim, denn es ist allgemein bekannt, dass auf Array-Elemente mit einem Index als Iterator mit Komplexität $\mathcal{O}(1)$ zugegriffen werden kann.

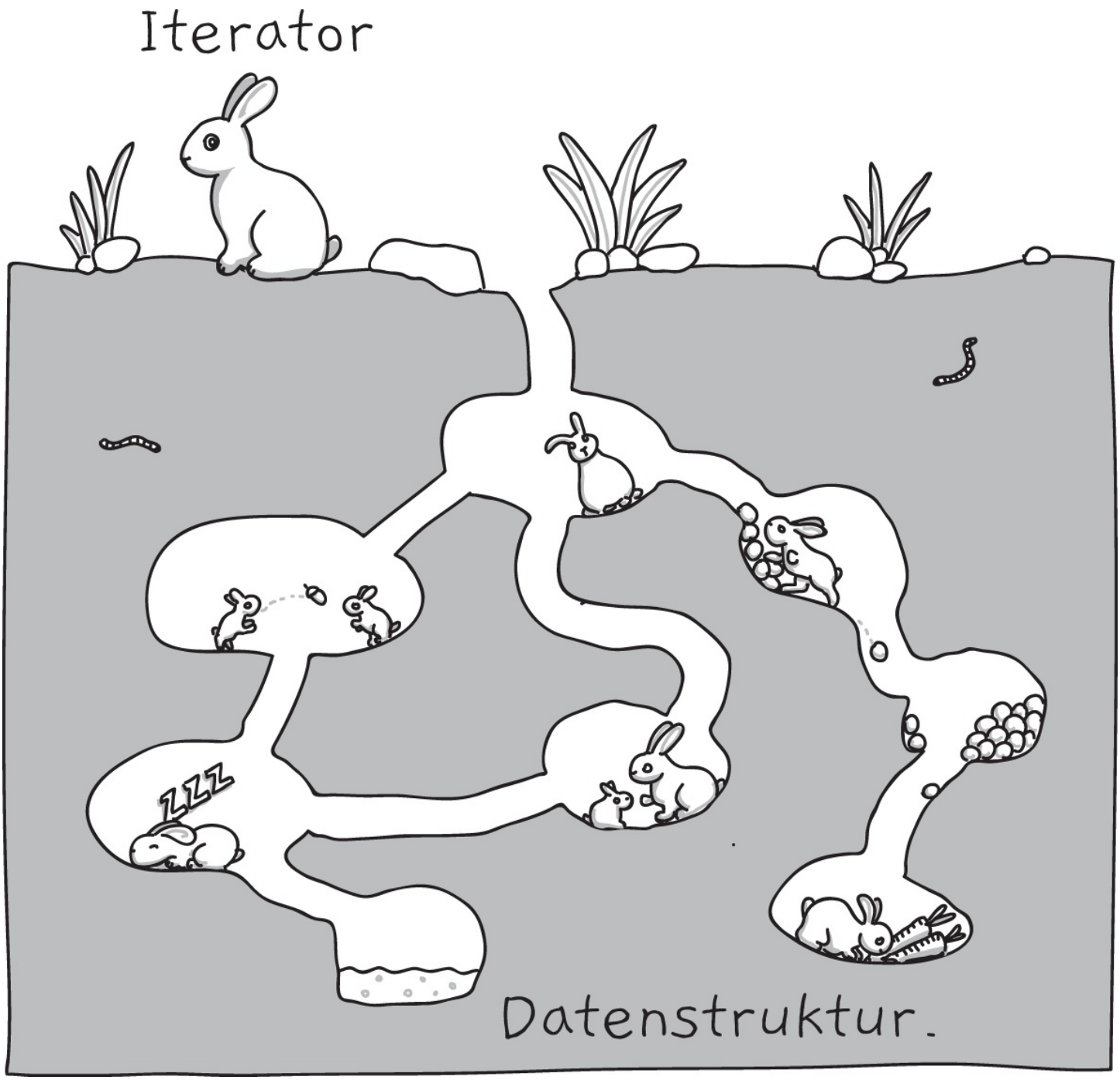


Abbildung 3.14: Iterator

Nehmen wir als Beispiel einen Iterator, der über ein Array läuft. In unserer Notation wäre dies:

```
Iterator(a) = record [                    Iterator über ein a = [Element 0 ... Element a.length-1]
    pos ← 0
    HasNext()
        return pos < a.length
    Next()
        oldPos ← pos
        pos ← pos + 1
        return a[oldPos]
]
```

Generell nehmen wir an, dass der Zugriff auf die Elemente einer Struktur mit einem Iterator die Laufzeitkomplexität $\mathcal{O}(1)$ hat. Die Annahme ist legitim, denn der Iterator hat einen Zustand: Er »weiß, wo gerade ist« und kann ohne jedes Suchen direkt zugreifen.

Die Elemente einer Liste wurden oben mit folgendem Algorithmus aufsummiert:

```
ListSum(lst)                              Struktur durchlaufen und dabei ihre Daten verarbeiten
    case lst = Nil
        return 0
    case lst = Cons(head, rest)
        return head + ListSum(rest)
```

Das ist eine *interne Traversierung*. Intern, weil die Traversierung der Struktur und die Verarbeitung ihrer Daten in einem erfolgen. Man kann sich vorstellen, dass jemand, der die Feinheiten der Struktur kennt, in ihr herumläuft und dabei die in ihrem Innern herumliegenden Daten verarbeitet. Der Auftraggeber einer internen Traversierung kann sie nicht abbrechen. Er übergibt die Verarbeitungsfunktion an die Struktur und muss dann warten, bis alles fertig ist. Dieses »Geh und verarbeite alles mit der Funktion, die ich dir gebe« sieht man deutlich, wenn die Verarbeitungsfunktion ein Parameter der Traversierung ist:

```
Add(x, y)                                 eine Aktion
    return x + y

ListSum(lst) ← TraverseWith(lst, Add, 0)  nimm und mach!

TraverseWith(lst, f, start)               Struktur durchlaufen ...
    case lst = Nil                        ... dabei ihre Daten mit f verarbeiten
        return start                      los geht's mit start
    case lst = Cons(head, rest)
        return f(head, ListSum(rest, f))
```

Bei einer externen Traversierung sind das Durchlaufen der Struktur und die Verarbeitung der in ihr abgelegten Daten getrennt. Die Verarbeitung wird extern, außerhalb der Struktur

von jemandem ausgeführt, der keinerlei Kenntnis ihrer inneren Organisation hat. Der Iterator kennt sich aus und holt die Daten herbei, die der Auftraggeber dann verarbeitet. Alles unter der Kontrolle des Auftraggebers.

Auch bei einer Liste ist es legitim, eine externe Traversierung zu nutzen. Wenn die Datenstruktur (noch) nicht genau angegeben ist, dann wird einfach angenommen, dass es einen Iterator oder etwas Äquivalentes gibt:

```
ListSum(lst)                    summiere alle Elemente einer Liste (mit einem Iterator)
    s ← 0
    for x ← lst                 sicher kann für lst ein Iterator definiert werden
        s ← s + x
    return s
```

Mit diesem Algorithmus wird implizit vorausgesetzt, dass es eine Iterator-Implementierung für die Liste gibt, die Zugriffsoperationen mit der Komplexität $\mathcal{O}(1)$ zulässt. Das geht nicht ganz so einfach wie der Zugriff auf die Elemente eines Arrays mit einem Index. Schwierig ist es aber nicht: Der Iterator muss unter Umständen mit Zeigern arbeiten. Klar, das geht auch mit unserem algebraischen Datentyp.

Wenn es also konkreter wird mit der Liste, dann kann der Iterator genau definiert werden. Beispielsweise als

```
Iterator(lst) = record [        Iterator über eine Liste mit alg. DT IntList
    act ← lst
    HasNext()
        return act ≠ Nil
    Next()
        case act = Cons(first, rest)
            act ← rest
            return first
]
```

Ein Iterator auf einer Baumstruktur ist dagegen schon nicht mehr ganz so trivial. Wenn wir schreiben

```
ListSum(tree)                   summiere alle Elemente eines Baums (mit einem Iterator)
    s ← 0
    for x ← tree                für jeden Wert x in tree; klar, da ist ein Zeiger-basierter
        s ← s + x               Iterator im Spiel, aber wie funktioniert der?
    return s
```

dann muss schon noch erklärt werden beziehungsweise klar sein, wie die externe Traversierung des Baums organisiert wird.

Zum Schluss erinnern wir noch einmal an die Diskussion am Anfang dieses Kapitels: In

vielen interessanten Algorithmen kommen Iterationen über abstrakte Kollektionen vor, bei denen (noch) nicht festgelegt wurde, welche Datenstruktur sie realisieren wird. Sehr oft sind das Mengen oder Listen. Mengen haben keine Struktur und sind darum auch keine Datenstruktur. Trotzdem ist es sehr üblich, dass im Pseudocode Konstrukte vorkommen wie

AbstractSetOp(*aSet*)
    ...
    $s \leftarrow \emptyset$
    for $e \leftarrow aSet$
        $s \leftarrow s \cup \{f(e)\}$
    ...

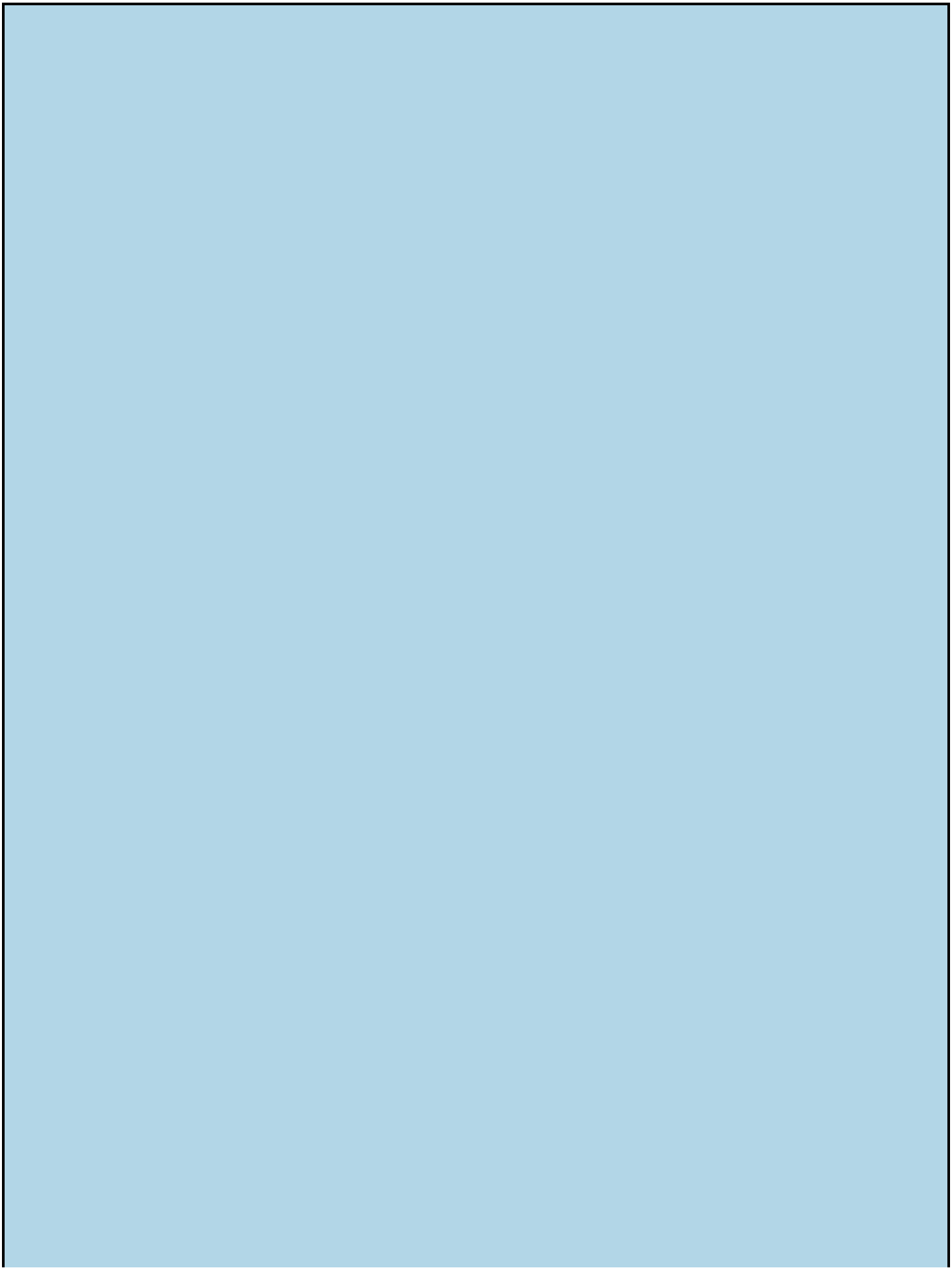
Hier ist noch offen, mit welcher Datenstruktur die Menge implementiert wird. Es wird aber angenommen, dass es möglich ist, diese Datenstruktur mit einen Iterator zu traversieren, welche Datenstruktur auch immer genutzt wird, um die Menge zu realisieren. Das ist realistisch: Für jede Datenstruktur kann ein Iterator definiert werden, mit dem sie durchlaufen werden kann, mit $\mathcal{O}(1)$ für jeden Zugriff. Alles im Speicher kann über Indizes und/oder Zeiger angesprochen werden.

Bei Listen ist die Sache nicht immer klar. Listen sind eine Forderung an Funktionalität einer Datenstruktur: »Irgendeine Datenstruktur, deren Elemente mit einem Index verknüpft sind.« Es gibt aber auch Datenstrukturen, die listenartig aufgebaut sind: Mit Zeigern verkettete Listen oder algebraische Datentypen wie IntList. Hier hängt es vom Kontext ab, was gemeint ist.

Teil II

Algorithmen in den Gärten der Strukturen


HIER IST
DAS BÜRO VON UNSEREM
IT-MANAGER.
Start
Anklopfen
Sagt jemand "Herein"?
Nein
Geh weg!
Ja
Eintreten
Ende
CHRISTIAN KALKERT



IN DIESEM TEIL …

Datenmodelle und Datenstrukturen

Listen, Bäume und Graphen

Von Algorithmen auf Modellen zu solchen auf Datenstrukturen

Kapitel 4

Listen: Immer einer nach dem anderen

IN DIESEM KAPITEL

Listen: Datenmodell, Datenabstraktion, Datenstruktur

Stapel, Warteschlange, Liste mit wahlfreiem Zugriff

Funktionale und imperative Algorithmen

Persistente Datenstrukturen

Was ist eine Liste? Alle in einer Reihe, einer neben dem anderen, die einfachste Art, Dinge anzuordnen. Drei Zahlen können eine Liste bilden: $[1, 2, 3]$, oder auch beliebige andere Dinge: [*der Hase Harvey*, *Roger Rabbit*, *Schneewittchen*]. Alle Primzahlen zwischen zwei und zehn. Die Liste der Backzutaten einer Erdbeertorte. Listen sind endliche Kollektionen von Dingen mit einer festgelegten Reihenfolge. Jedes Element einer Liste hat eine Position, zu jeder Position gibt es ein Element, es sei denn, die Position liegt außerhalb der Liste. Das ist das *Datenmodell* einer Liste.

Hmm, was ist die Reihenfolge der Backzutaten? Stehen die nicht einfach nur in einer Liste, weil wir so schlecht etwas ohne irgendeine Anordnung aufschreiben können und es darum meist gleich in die Form einer Liste bringen? Listen sind also auch eine *Datenstruktur*: Eine bestimmte Art der *Anordnung*. Dinge, die eine natürliche Ordnung haben, groß/klein etwa, kann man in dieser Ordnung anordnen: Das größere vor dem kleineren beispielsweise. Bei der Ordnung in einer Datenstruktur geht es um *wer steht wo*, bei einer Ordnung allgemein geht es um *Größe*, *Wichtigkeit*, *Alter* und so weiter.

Man kann aber auch Dinge, die keine Ordnung haben, so wie die Backzutaten, in einer Liste anordnen. Das was »eigentlich« nur eine Menge ist, also eine ungeordnete Anhäufung von Dingen, wird als Liste angeordnet. Gelegentlich kann man Dinge nach verschiedenen Ordnungskriterien oder einfach nur zufällig anordnen (siehe Abbildung 4.1).

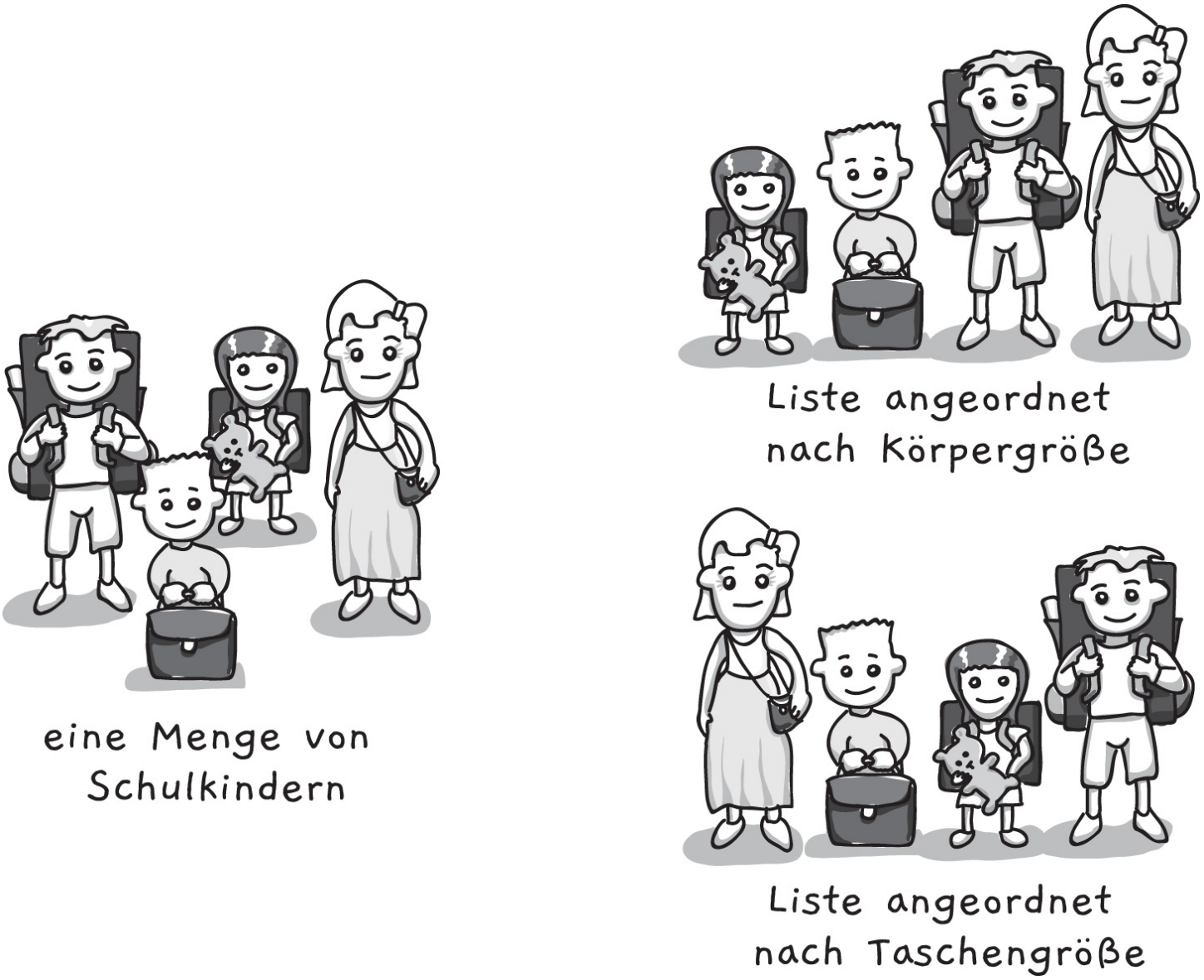


Abbildung 4.1: Ordnung, Anordnung und Listen

Listen kann man speichern. Die Backzutaten einer Erdbeertorte fein säuberlich zeilenweise untereinander auf einem Zettel. Das wäre eine zusammenhängende Struktur. Die Ordnung der Dinge kann ihre Anordnung im Speichermedium bestimmen. Verkettet geht auch: Ein Listenelement zeigt auf das nächste. Man muss die Listenordnung aber nicht zwingend in eine Speicherstruktur übersetzen. Im Wartezimmer sitzen die Patienten ja auch nicht genauso in einer Reihe, wie sie dran kommen werden. Sie bilden trotzdem zumindest eine Art Warteschlange, also eine Art von Liste.

Mit Listen kann man etwas machen. Jede nicht-leere Liste hat ein erstes Element. Das hat eine Menge nicht. Eine Liste kann auch im Gegensatz zu einer Menge das Gleiche mehrfach enthalten. Natürlich nicht an der gleichen Position. Wo es bei Mengen nur ein Drin oder Nicht-drin gibt, kann man bei Listen diverse Operationen definieren, die mit der Reihenfolge in der Liste zusammenhängen: Erstes, letztes, i -tes Element, Kopf und Rest der Liste, ein Element vorne anfügen, entnehmen oder austauschen und so weiter. Das sind Funktionalitäten, die mit dem Konzept einer Liste möglich werden und gerne als

Datenabstraktion (auch *abstrakter Datentyp, ADT*) beschrieben werden.



Listen, die Datenstruktur, in der ihre Elemente gespeichert werden, und das, was man damit machen kann, sind verschiedene Dinge. Leider wird für all das derselbe Begriff »Liste« benutzt.

Mit »Liste« kann also vieles gemeint sein. Die Bezeichnungen gehen da gerne auch mal etwas durcheinander und manche verwenden auch andere Begriffe wie etwa »Sequenz« als Obergriff, zur klareren Unterscheidung oder zur vollständigen Verwirrung. Wir verzichten darauf, eine eigene klare Terminologie zu definieren. Sie würde die Begriffsverwirrung ja nicht beenden, sondern nur bereichern und es ist auch wichtig, mit der Mehrdeutigkeit klarkommen zu lernen. Was genau gemeint ist, geht ja in der Regel aus dem Kontext hervor. Konzentrieren wir uns also auf die Konzepte.

Im Kern geht es um Algorithmen. Sie agieren auf Datenmodellen, hinter denen Datenstrukturen stehen, die wir aber nicht immer so direkt sehen wollen (siehe Abbildung 4.2).



Abbildung 4.2: Kasperle (Algorithmus) und Datenmodell (Liste)


Der Begriff »Liste« bezieht sich regelmäßig auf unterschiedliche Konzepte:

- **Datenmodell:** Eine endliche Menge von Positionen $1, 2, \ldots, n$, denen jeweils ein Element zugeordnet ist. (»Kopfkino«: Dinge mit Nummer)
- **Datenstruktur:** Eine Kollektion von Elementen, die linear im Speicher angeordnet sind. (Speicherstruktur: Dinge irgendwie nebeneinander)
- **Datenabstraktion:** Das Konzept einer Kollektion von Dingen, die bestimmte Operationen erlauben, die einen Bezug zu einer Position haben, etwa: »Gib mir den Ersten«, »Lösche den Dritten« etc. (Konzept der Softwaretechnik: Dinge, mit denen man etwas Bestimmtes machen kann)

Listen: Datenmodell und Implementierung

Eine Liste ist eine endliche Folge von null oder mehr Elementen. Listen sind ein wichtiges und häufig genutztes Datenmodell des täglichen Lebens: Die Monate eines Jahres, die Primzahlen unter 10, die Patienten in einem Wartezimmer. All das kann in Listen angeordnet werden. Zu einer Liste gehört immer eine Reihenfolge. Es gibt ein erstes, zweites, … und letztes Element. Auch wenn die Reihenfolge nicht explizit genannt wird, so muss sie doch immer existieren und klar definiert sein. Meist hat man eine *Ordnung der Dinge*, die sich als *Anordnung* in der Liste wiederfindet. Die Liste der Monate hat eine zeitliche Ordnung und die Monate können in einer Liste entsprechend angeordnet werden:

[Januar, Februar, März, April, ..., Dezember]

Die Primzahlen kleiner 10 können entsprechend ihrer Größe angeordnet werden:

$$[2, 3, 5, 7]$$

Die Patienten im Wartezimmer haben viele vergleichbare Eigenschaften, beispielsweise die Zeit ihrer Ankunft. Sie können, müssen aber nicht in dieser Ordnung angeordnet sein, man erwartet aber, dass diese Ordnung die Reihenfolge der Bedienung bestimmt. (Beim Zahnarzt hofft man gelegentlich auf etwas anderes.)

Mathematiker können Anordnungen als Abbildungen beschreiben, die Werten aus einem Bereich der natürlichen Zahlen jeweils einen Wert zuweisen:

$$l = [2, 3, 5, 7] = \{1 \to 2,\ 2 \to 3,\ 3 \to 5,\ 4 \to 7\}$$

Einem Index $i \in \{1, 2, 3, 4\}$ ist eine Listenelement l_i der Liste l zugeordnet:

$$l_1 = 2,\quad l_2 = 3,\quad l_3 = 5,\quad l_4 = 7$$

Sie beginnen meist mit 1 als dem kleinsten Index, Informatiker tendieren eher zu 0, denn dann kann der Index auch schon gleich als *Offset* (Distanz zum Anfang) in einer

Datenstruktur mit Startadresse l interpretiert werden.

Jede Liste hat eine *Länge*. Das ist die Zahl ihrer Elemente. In Algorithmen schreiben wir *length* als Attribut einer Liste, also beispielsweise:

$[2, 3, 5, 7]$.length = 4

Eine Liste kann *leer* sein. Sie hat dann die Länge null:

$[\ ]$.length = 0

Die (An-)Ordnung der Dinge in der Liste erlaubt die Definition weiterer *Operationen*. Eine nicht-leere Liste hat ein *erstes* Element »head« und einen Rest »tail«:

$[2, 3, 5, 7]$.head = 2

$[2, 3, 5, 7]$.tail = $[3, 5, 7]$

Zu jedem Index im Indexbereich einer Liste gibt es ein Element:

$[2, 3, 5, 7][1] = 2$

$[2, 3, 5, 7][4] = 7$

Zwei Listen kann man zu einer verknüpfen:

$$[2, 3, 5, 7] + [11, 13, 17] = [2, 3, 5, 7, 11, 13, 17]$$

Eine Liste kann man permutieren. Eine *Permutation* ist eine Anordnung der Elemente der Liste in einer anderen Reihenfolge. Beispielsweise ist

$$[7, 5, 3, 2] = \{1 \to 7,\ 2 \to 5,\ 3 \to 3,\ 4 \to 2\}$$

eine Permutation von

$$[2, 3, 5, 7] = \{1 \to 2,\ 2 \to 3,\ 3 \to 5,\ 4 \to 7\}$$

Das ist das Datenmodell der Listen. Es ist noch nichts darüber gesagt, wie die Listen gespeichert sind. Es geht nur um das, was konzeptionell möglich ist, wenn man mit einer Kollektion von Dingen arbeitet, die jeweils einem Index zugeordnet sind. Das reicht aus, um Algorithmen definieren und auch um (in Grenzen) über deren Laufzeitklasse sprechen zu können. Betrachten wir den Algorithmus zur Suche des leichtesten Mannes aus Kapitel 2:

```
MinWeight2(Liste von n Männern)
    m ← 1                          Mann m ist der leichteste bisher bekannte Mann
    for i ← 2, ..., n              prüfe alle Männer der Reihe nach
        if Mann i ist leichter als Mann m
            m ← i                  i ist der leichteste Mann, den wir bisher gehen haben
    return m
```

In diesem Algorithmus wird angenommen, dass die Liste nicht leer ist. Über die Reihenfolge der Männer in der Liste ist nichts bekannt. Sie sind jedenfalls nicht nach

Gewicht sortiert. Vielleicht stehen sie in einer rein zufälligen Reihenfolge. Mit unserer Listennotation können wir den Algorithmus umformulieren in

```
MinWeight2a(lst)                    lst ist eine nicht leere Liste von Männern
    m ← 1
    for i ← 2, ..., n
        if lst(i) ist leichter als lst(m)    ob lst(i) in O(1) oder O(n) ist, ist hier unklar
            m ← i
    return m
```

Diese Definition des Algorithmus lässt offen, um welche Datenstruktur es sich bei lst genau handelt. Es ist irgendeine Datenstruktur, die einen indizierten Zugriff erlaubt. Möglicherweise ist die Laufzeit von $lst(i)$ von i und damit von der Eingabelänge n abhängig. Für eine klare Aussage über die Laufzeitkomplexität müssten wir etwas präziser werden. Etwa mit einem Iterator-Lauf über die Struktur:

```
MinWeight2Iter(lst)                 lst ist eine nicht leere Liste von Männern
    m ← lst(1)
    for man ← lst
        if man ist leichter als m   Iterator-Zugriffe sind immer in O(1)
            m ← man
    return m                        gibt den leichtesten Mann zurück, nicht dessen Index
```

Bei Zugriffen in einer For-Schleife über eine Struktur nehmen wir ja an, dass der Zugriff in konstanter Zeit erfolgt. Natürlich können wir auch verlangen oder voraussetzten, dass die Liste mit einer Datenstruktur implementiert wird, die Zugriffe mit einer Laufzeit in $\mathcal{O}(1)$ zulässt. Wir können auch eine konkrete Datenstruktur verwenden, die die gewünschten Eigenschaften hat. Ein Array ist eine geeignete Datenstruktur für Listen und hat Zugriffsoperationen mit einer Laufzeit in $\mathcal{O}(1)$:

```
MinWeight2Array(a)                  a ist ein Array der Länge n von Männern
    m ← 1
    for i ← 2, ..., n
        if a[i] ist leichter als a[m]    Array-Zugriffe sind in O(1)
            m ← i
    return m
```

Arrays sind eine naheliegende Datenstruktur für Listen. Listen kann man auch in anderen Strukturen speichern, sogar in solchen, die gar keine Listenform haben.

Was wir uns als Erstes vorstellen, ist das Datenmodell. Das *Datenmodell Liste* ist einfach die Vorstellung einer linearen Anordnung: einer nach dem anderen. Das ist so etwas wie unser »Kopfkino« der Daten. Für die Auswahl einer geeigneten Datenstruktur ist es aber nicht so wichtig, wie wir uns etwas vorstellen, sondern wie unsere Algorithmen auf die Daten zugreifen. Das ist die Frage nach der Funktionalität, die von einer Liste erwartet

wird.

Datenabstraktion: Was Listen so können sollen

Mit einer Liste kann man unterschiedliche Sachen machen. Die wichtigste Operation ist der Zugriff auf die Elemente. Manchmal will man nur auf das erste Element zugreifen. In anderen Fällen auf jedes Element. Man kann lesenden und schreibenden Zugriff unterscheiden. Manchmal will man die Liste am Anfang oder am Ende verlängern, verkürzen oder sogar etwas mitten hineinschieben oder auch mittendrin ein Element austauschen oder löschen.

Betrachtet man nur die Operationen, die man auf einer Liste ausführen kann, dann kommt man zum Begriff der Datenabstraktion. Einige übliche *Datenabstraktionen* (auch *abstrakter Datentyp*, *ADT*) für das Datenmodell »Liste« sind:

Bezeichnung	***Funktionalität***
Stapel	Zugriff nur an einem Ende
Warteschlange	Zugriff an beiden Enden
Array	Zugriff überall, keine Änderung der Größe
Tupel	lesender Zugriff überall, keine Änderung der Größe
dynamisches Array	Zugriff überall, Änderung der Größe möglich
Liste mit wahlfreiem Zugriff	lesender und schreibender Zugriff überall

Hier geht es um das Können, die Funktionalität, die von einer Kollektion geboten werden kann, deren Komponenten nummeriert sind. Soll nur an einem Ende zugegriffen werden, dann reicht ein Stapel. Ein Stapel ist ein Datenmodell: Wir stellen uns Dinge vor, die aufeinander gestapelt werden. Stapel sind eine Variante der Liste: alles ist in einer Reihe, aber die Reihe steht senkrecht, die Elemente liegen direkt aufeinander und darum kommt man immer nur an das heran, was ganz oben liegt.

Stapel

In der physischen Welt kann man Stapel-Strukturen erzeugen: Ich stapele Teller oder Stapelstühle aufeinander und dann gibt wirklich nur einen Zugriff auf den obersten Teller oder Stuhl. Als Datenstruktur ist so etwas natürlich nicht möglich: Im Speicher eines Rechners kann man Daten nicht »aufeinanderlegen«. Man kann sie verkettet oder zusammenhängend hintereinanderspeichern und kommt dann natürlich immer an jede Komponente.

Es geht auch gar nicht darum, wer was kann, es geht darum, was ein Algorithmus benötigt. Es kann ja sein, dass der Algorithmus immer nur an einen Ende zugreift. In dem Fall reicht die Funktionalität eines Stapels. Zur klaren Trennung zwischen der Funktionalität und ihrer Implementierung gibt es das softwaretechnische Konzept der Datenabstraktion. Eine Datenstruktur und einige Zugriffsoperationen werden

zusammengepackt und nur die Zugriffsoperationen sind von außen nutzbar. Wir sind hier nicht an softwaretechnischen Konzepten interessiert und wollen das Thema nicht weiter vertiefen. Im Kern kann eine Datenabstraktion als Record modelliert werden, der neben Daten auch Operationen enthält:

```
Stack = record [                          Datenabstraktion Stapel als Record realisiert
    type Cell = (Elementtyp, Zeiger auf Cell)     Paar aus Wert und einem Zeiger

    head ← null                           Zeiger auf den Anfang
    Push(x)                               ein Element vorne anfügen
        head ← new Cell(x, head)

    Pop()                                 erstes Element entfernen und ausliefern
        case object(head) = Cell(v, next)
            head ← next
            return head
]
```

Die Operation Pop ist nur definiert, wenn $head$ auf eine Zelle zeigt. Wir treiben hier keine Softwaretechnik und ignorieren die Probleme, die entstehen können, wenn Pop für einen leeren Stapel aufgerufen wird. Ebenso wenig interessieren wir uns hier für *public* und *private* oder irgendwelche Vererbungsrelationen.

Hier ist $head$ eine lokale (und »private«) Variable, die einen Zeiger auf eine verkettete Liste enthält. Die verkettete Liste repräsentiert den Stapel in seinem aktuellen Zustand. Push und Pop können auf $head$ als globale Variable zugreifen.

Eine Datenabstraktion ist ein Datenmodell, dem eine konkrete softwaretechnische und sprachliche Gestalt gegeben wurde. Ein Stapel mit seinem veränderlichen Zustand passt perfekt zur imperativen/objektorientierten Sicht der Welt. Ein Objekt ist nichts anderes als ein Record mit Funktionen, die auf die Komponenten zugreifen können. Dies ist somit eine ganz spezielle Art von Variable. Man greift nicht wie bei einer »normalen« Variablen lesend und schreibend zu, sondern mit besonderen Zugriffsmethoden, hier: Push und Pop.


Imperative Datenabstraktionen verallgemeinern Variablen. Der Stapel ist hier eine »Variable« mit den Zugriffsmethoden Push und Pop.

Ein Stapel, eine in ihren Fähigkeiten etwas eingeschränkte Liste, wird oben mit der *Datenstruktur* »einfach verkettete Liste« implementiert. Das ist nur eine von vielen Möglichkeiten, einen Stapel, etwas mit Push und Pop zu realisieren. Ein in seiner Größe beschränkter Stapel kann mit einem Array realisiert werden:

```
Stack(n) = record [                    Stapel auf Basis eines Arrays
    a ← new array[0 .. n−1] of ...
    top ← 0                            Zahl der Elemente, Index erster freier Platz
Push(x)                                ein Element vorne anfügen
    if top < n
        a[top] ← x                     Array-Zugriff
        top ← top + 1
    Pop()                              erstes Element entfernen und ausliefern
        if top > 0
            top ← top − 1
            return a[top]
]
```

Die Zugriffsoperationen sind unabhängig von der Länge (Höhe) des Stapels und so in $\mathcal{O}(1)$. Variablen- und Array-Zugriffe erfolgen in konstanter Zeit. Den Stapel können wir natürlich auch mit einem algebraischen Datentyp realisieren:

```
Stack = record [                       Stapel auf Basis eines alg. DT
    type TList = Cons(T, TList) | Nil  T ist irgendein Typ
    stackRep ← Nil

    Push(x)                            ein Element vorne anfügen
        stackRep ← Cons(x, stackRep)

    Pop()                              erstes Element entfernen und ausliefern
        case stackRep = Cons(head, tail)
            stackRep ← tail
            return head
]
```

Der algebraische Datentyp TList ist eine leichte Abstraktion der verketteten Liste. Wir sind dabei etwas weniger genau darin, inwieweit die Daten verkettet oder zusammenhängend im Speicher angeordnet sind, können aber immer noch von Zugriffsoperationen in $\mathcal{O}(1)$ ausgehen, da die Zugriffe direkt am Anfang stattfinden und nichts durchsucht werden muss.

Mit einer Datenabstraktion wird eine Datenstruktur aufgehübscht. Ohne das geht es natürlich auch:

```
PushA(x, stackRep)
    return Cons(x, stackRep)           neuer größerer Stapel
```

```
PopA(stackRep)
    case stackRep = Cons(head, tail)
        return (head, tail)          Wert und neuer kleinerer Stapel
```

Push und Pop haben hier einen zusätzlichen Parameter. Die Datenstruktur, auf der sie arbeiten, ergibt sich jetzt ja nicht mehr implizit aus dem Kontext der Datenabstraktion, sie muss explizit übergeben werden. Das Gleiche gilt für das Ergebnis. Beide Operationen erzeugen eine neue Liste, die anschließend als Stack genutzt werden soll. In der Datenabstraktion oben sind die Dinge klar. Die neue Liste ist der neue Repräsentant *stackRep* des Stapels. Push und Pop haben in der Record-Version einen Seiteneffekt auf *stackRep*. Hier haben Push und Pop den Repräsentanten des Stapels als expliziten Parameter. Der von ihnen erzeugte neue Repräsentant muss irgendwo abgelegt werden. Er geht zurück an den Aufrufer, damit dieser sich darum kümmern kann.

Der Stapel wird in der letzten Version von Push und Pop funktional verwendet: Es sind reine Funktionen, deren Wirkung auf die Erzeugung eines Ergebnisses beschränkt ist. Sie ändern nichts irgendwo nebenbei. Damit sind wir in der funktionalen Welt, in der Datenabstraktionen auch eher unüblich sind. Davon später mehr. Bleiben wir erst einmal in der imperativen Welt und sehen uns interessantere Exemplare aus dem Listenzoo an.

Warteschlangen

Bei einer *Warteschlange* wird vorne weggenommen und hinten angefügt. Das ist eine dynamische Struktur: Es ist nicht vorhersehbar, wie groß die Schlange wird. Eine veränderliche, imperative Implementierung ist ebenfalls naheliegend: Der Neuankömmling wird an den bisherigen Letzten angehängt. Das ist eine Aktion, die von einer Operation, nennen wir sie Enqueue, im Hintergrund ausführt wird. Die Warteschlange wird verändert, aber der Aufrufer von Enqueue sieht diese Veränderung nicht direkt, denn Enqueue gibt ja nichts zurück, es verändert.

```
Enqueue(5)                                  imperativ: irgendwo ändert sich etwas
newQversion ← Enqueue(oldQversion, 5)       funktional: nichts passiert im Hintergrund
```

Etwas im Hintergrund »heimlich« ändern, das ist nicht funktional, das ist imperativ.

Mit einer verketteten Liste kann die Warteschlange realisiert werden. Dazu nehmen wir noch zwei Variablen, deren Inhalt Zeiger sind, die auf den Anfang beziehungsweise auf das Ende zeigen.

Die Zeiger in der verketteten Liste müssen veränderlich sein: Es kann ja jederzeit ein neues Element hinzukommen oder weggenommen werden (siehe Abbildung 4.3).

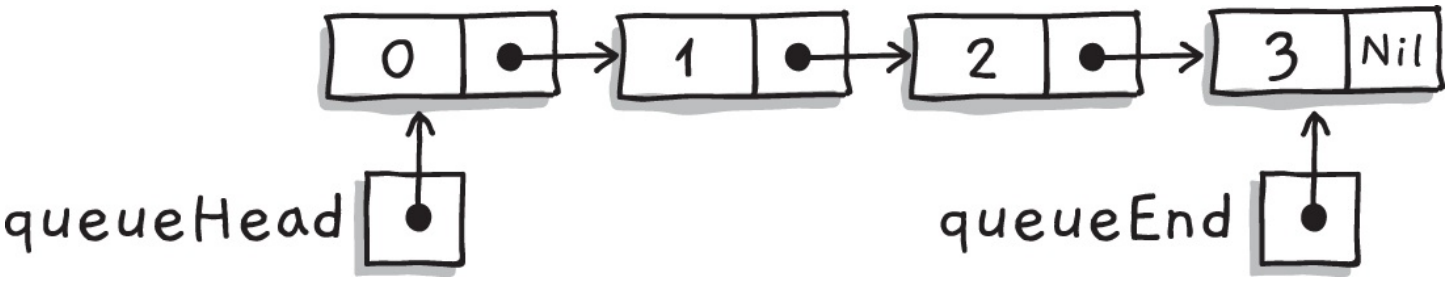


Abbildung 4.3: Imperative Datenstruktur für eine Warteschlange

Die Zeiger auf den Anfang beziehungsweise das Ende werden mit dem Null-Zeiger initialisiert. Die Warteschlange ist am Anfang ja leer:

queueHead ← **null**
queueEnd ← **null**

Wenn ein Element angehängt werden soll, dann benötigen wir eine neue Zelle, in der es gespeichert wird. Zellen enthalten einen Wert und einen Zeiger auf die nächste Zelle:

type Cell = **record**(content: Int, next: CellPointer)
type CellPointer = Zeiger auf Cell

Der Wert ist eine ganze Zahl, der Zeiger auf den Nachfolger muss veränderlich sein. So etwas kann nicht mit einem algebraischen Datentyp ausgedrückt werden. Strukturen, die mit einem algebraischen Datentyp beschrieben werden, können zerlegt werden und aus den Teilen können neue Exemplare zusammengebaut werden, aber Änderungen im Inneren einer Struktur gibt es dort nicht. Wir müssen die Algorithmen auf dieser Datenstruktur mit »nackten Zeigern« definieren.



Algebraische Datentypen werden mit Zeigern implementiert, bei ihnen wird aber ein Zeiger niemals verändert. Mit veränderlichen Zeigern erschließen sich neue Möglichkeiten – leider auch für Fehler.

Beim Einfügen wird zuerst eine neue Zelle angelegt und mit dem Wert und einem Null-Zeiger gefüllt. Ist die Warteschlange leer, dann enthalten *queueHead* und *queueEnd* den Null-Zeiger, der anschließend durch den Zeiger auf die neue Zelle ersetzt wird (siehe Abbildung 4.4).

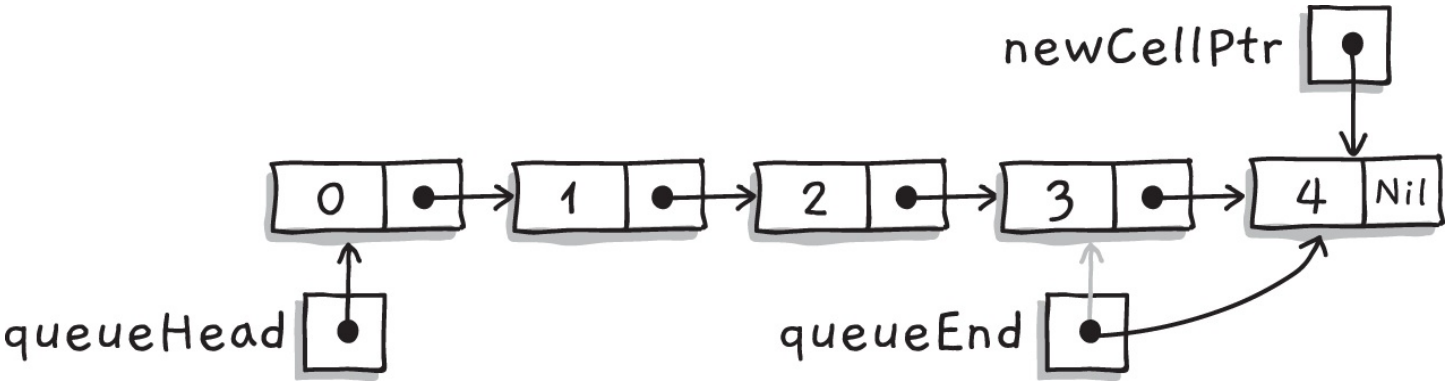


Abbildung 4.4: Neues Element zur Warteschlange

Ein neues Element wird Enqueue hinzugefügt. Dabei wird es einfach an das bisherige letzte angehängt:

```
Enqueue(x)
    newCellPtr = new Cell(x, null)
    if queueHead = null
        queueHead ← newCellPtr
        queueEnd ← newCellPtr
    else
        object(queueEnd).next ← newCellPtr
        queueEnd ← newCellPtr
```

Beim Entnehmen mit Dequeue werden der Wert der ersten Zelle ausgeliefert und die Zelle aus der Verkettung entfernt. Die Warteschlange muss dazu natürlich mindestens ein Element enthalten. Wenn nur ein einziges Element vorhanden ist und dieses gelöscht wird, dann muss auch *queueEnd* auf den Null-Zeiger gesetzt werden.

```
Dequeue()
    firstCell ← object(queueHead)
    if firstCell.next = null          die erste ist auch die letzte Zelle
        queueHead ← null              die WS ist jetzt leer
        queueEnd ← null
    else
        queueHead ← object(queueHead).next      der Zweite wird zum Ersten
    return firstCell.content
```

Ein Element darf natürlich nur entnommen werden, wenn die Warteschlange nicht leer ist. Einen entsprechenden Test haben wir hier weggelassen.

Zu einer Warteschlange gehören also

- ✔ eine Datenstruktur bestehend aus zwei Variablen *queueHead* , *queueEnd* und einer verketteten Liste, die aus einer variablen Anzahl von Zellen besteht, sowie
- ✔ zwei Algorithmen, einer zum Einfügen und einer zum Entnehmen.

Die Warteschlange als (verborgene) Datenstruktur plus Operationen können wir wieder zu einer Einheit zusammenfassen, nämlich zu einer imperativen *Datenabstraktion* »Warteschlange«.

```
Queue = record [                  Warteschlange als imperative Datenabstraktion
    queueHead ← null
    queueEnd ← null
```

```
Enqueue(x)                          kein Lauf durch die Datenstruktur: in O(1)
    newCellPtr = new Cell(x, null)
    if queueHead = null
        queueHead ← newCellPtr
        queueEnd ← newCellPtr
    else
        object(queueEnd).next ← newCellPtr
        queueEnd ← newCellPtr
    Dequeue()                       kein Lauf durch die Datenstruktur: in O(1)
        firstCell ← object(queueHead)
        if firstCell.next = null
            queueHead ← null
            queueEnd ← null
        else
            queueHead ← object(queueHead).next
        return firstCell.content
]
```

Eine Warteschlange mit ihrem veränderlichen Zustand passt perfekt zu einer imperativen/objektorientierten Sicht der Welt. Ein Objekt ist nichts anders, als ein Record mit Funktionen, die auf die Komponenten zugreifen können, also einer speziellen Art von Variable. Man greift nicht wie bei einer »normalen« Variablen lesend und schreibend zu, sondern mit besonderen Zugriffsmethoden, hier: Enqueue und Dequeue. Beide Operationen haben offensichtlich konstante Laufzeiten.

Verkettete Listen mit wahlfreiem Zugriff

Listen mit wahlfreiem Zugriff sind eine Verallgemeinerung der Warteschlange: Elemente können nicht nur an den beiden Enden, sondern an einer beliebigen Position eingefügt und entnommen werden. Wir nehmen die gleiche Datenstruktur wie bei der imperativen Warteschlange oben. Mit einem zusätzlichen Rückwärtszeiger in jedem Knoten könnten die Algorithmen auch noch etwas vereinfacht werden. Wir lassen ihn hier mal weg.

Beim Einfügen geben wir die Position an, an der das neue Element eingefügt werden soll. Dann wird die Verkettung so lange durchlaufen, bis wir vor dieser Position sind. Das neue Element wird dann in einer neuen Zelle dort eingeschoben. Natürlich muss die Liste lang genug sein, um eine Zelle an der gewünschten Stelle einfügen zu können, in eine Liste der Länge 3 darf man nichts an der 10-ten Position einfügen.

Der Zeiger auf den Anfang der Liste liegt in einer Variablen $listHead$. Dabei handelt es sich um eine Variable. Sie wird mit dem **null** -Zeiger initialisiert.

Normalerweise geht aus dem Kontext klar hervor, ob ein Wert (Konstante) oder eine Variable gemeint ist. Bei imperativen Algorithmen, die mit Zeigern arbeiten, muss man

etwas sorgfältiger zwischen beiden unterscheiden. Im Zweifelsfall muss explizit klargestellt werden, ob es sich um einen Wert oder eine Variable handelt.

Angenommen, es soll ein neues Element x an einer bestimmten Stelle $index$ eingefügt werden. Als erstes wird eine neue Zelle erzeugt, die x enthält.

$newCellPtr \leftarrow$ **new** Cell(x, **null**)

Der Nachfolger der Zelle ist noch unbekannt, also wird der Zeiger auf die nächste Zelle erst einmal auf **null** gesetzt. Die Adresse dieser Zelle wird später in der Vorgängerzelle gespeichert und der Null-Zeiger der neuen Zelle wird durch den Zeiger auf die Nachfolgerzelle ersetzt werden. Dazu muss natürlich die Einfügestelle, also die Position von Vorgänger- und Nachfolgerzelle, bekannt sein.

Wir müssen mithin die Stelle suchen, an der die neue Zelle in die Verkettung eingefügt werden soll. Dazu wandern wir mit zwei Zeigern so lange durch die Verkettung, bis die richtige Stelle gefunden wurde (siehe Abbildung 4.5).

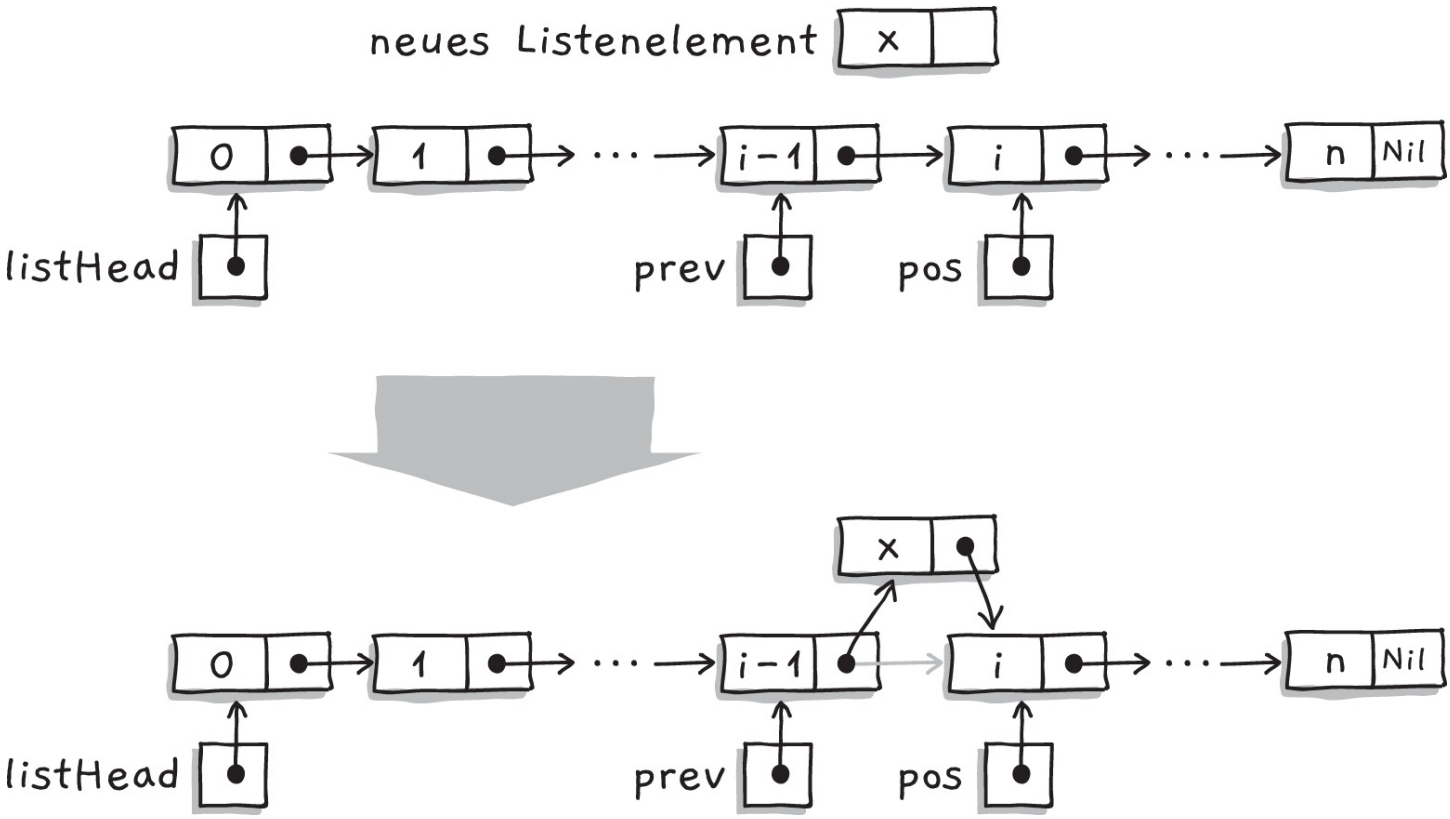


Abbildung 4.5: Einfügen in eine verkettete Liste: imperativ

pos startet mit $listHead$ und $prev$ läuft ihm hinterher.

Die richtige Stelle wird durch $index$ bestimmt, den Parameter der Einfügeoperation. Alle Zellen haben eine Nummer. Diese wird nicht in den Zellen gespeichert, man kann ihn nur durch Abzählen bestimmen. Dazu nehmen wir eine Laufvariable i. Wie in der Informatik üblich, beginnen wir das Zählen mit 0.

$i \leftarrow 0$

Die Suche nach der richtigen Stelle zum Einfügen wird also insgesamt von drei Variablem kontrolliert, deren gegenseitige Beziehungen durch folgende *Schleifeninvariante* beschrieben werden:

- i zählt die Elemente durch.
- pos zeigt auf die Zelle, in der das i -te Element gespeichert ist.
- $prev$ zeigt die Zelle, in der das $(i-1)$-te Element gespeichert ist, also auf den Vorgänger der i -ten Zelle.

Wenn an der Position $index$ ein neues Element eingefügt werden soll, dann muss das neue Element das alte Element verdrängen und mitsamt all seiner Nachfolger um eins nach rechts verschieben. In Abbildung 4.5 wird i mit all seinen Nachfolgern nach hinten verdrängt. Dabei muss sich aber niemand wirklich bewegen. Das neue Element wird in die Kette eingefädelt.

Wenn die richtige Stelle gefunden wurde, dann gilt (siehe Abbildung 4.5):

- $i = index$
- pos zeigt auf die Zelle, die jetzt mit all ihren Nachfolgern weiter nach hinten rückt.
- $prev$ zeigt auf die Zelle, hinter der die neue Zelle eingefügt wird.

Mit einer Schleife kann die richtige Position, also die richtigen Werte für pos und $prev$, gesucht werden:

```
i ← 0
while i < index                    es gilt die Schleifeninvariante
    prev ← pos
    pos ← object(pos).next
    i ← i + 1
```

Nach dem Ende der Schleife wurde die richtige Stelle gefunden: pos und $prev$ haben die gewünschten Werte. Es kann eingefügt werden:

```
object(newCellPtr).next ← pos      neue Zelle zwängt sich vor alte an Position i und
object(prev).next ← newCellPtr     wird Nachfolger des Vorgängers der Verdrängten
```

Ein Sonderfall muss noch beachtet werden: Am Anfang gibt es keinen Vorgänger. In dem Fall ist $prev$ der Null-Zeiger und statt im Vorgänger muss das neue Element in $listHead$ verankert werden:

```
object(newCellPtr).next ← pos    neues i-tes Element vor altes i-tes Element
if prev = null                   kein Vorgänger, also ganz an den Anfang
    listHead ← newCellPtr
else                             hinter den Vorgänger
    object(prev).next ← newCellPtr
```

Insgesamt haben wir folgenden Algorithmus zum Einfügen eines Wertes x an Position $index$ in einer verketteten Liste:

```
InsertAtPos(x, index)            füge x an Position index in die Verkettung ein
    newCellPtr ← new Cell(x, null)
    prev ← null
    pos ← listHead
    i ← 0
while i < index
    prev ← pos
    pos ← object(pos).next
    i ← i + 1

object(newCellPtr).next ← pos
if prev = null
    listHead ← newCellPtr
else
    object(prev).next ← newCellPtr
```

Es ist klar, dass die Operationen in schlechtesten Fall die gesamte Liste durchlaufen müssen. Einfügen, Suchen und Löschen sind damit in der Laufzeitklasse $\mathcal{O}(n)$, wobei n die Länge der Liste als Eingabelänge repräsentiert.

Zeiger auf Zeigervariablen

Will man sich ernsthaft auf der Ebene der Zeiger bewegen, dann wird man eine Programmiersprache nutzen, in der Zeiger »richtige« Werte (*first class values*) sind und die es erlaubt, veränderliche Variablen an Funktionen zu übergeben. Das kann elegant in Form von Referenzparametern geschehen (beispielsweise in C++) oder es muss mühsam als Zeiger auf Zeiger nachgebaut werden (beispielsweise in C). In Sprachen, die dieses Konzept nicht unterstützen (beispielsweise Java), kann man Referenzparameter emulieren, für ernsthafte systemnahe Entwicklungen ist das aber nur ein Notbehelf. Sprachen wie Java sind darum für die systemnahe Programmierung eher ungeeignet.

Da Adressen und Zeiger für viele Algorithmen und Datenstrukturen von Bedeutung sind, sollten dem Algorithmiker das Konzept Zeiger auf Zeiger und der Referenzparameter zumindest prinzipiell bekannt sein. Wir erläutern es kurz am Beispiel der verketteten Liste von oben, in der einfachen Variante ohne spezielle sprachliche Hilfsmittel.

Beim Einfügen kommt der Zeiger auf die neue Zelle entweder in die Vorgängerzelle oder in die Verankerung der Liste:

```
if prev = null
  listHead ← newCellPtr
else
  object(prev).next ← newCellPtr
```

Diese Unterscheidung ist nicht notwendig, wenn *prev* nicht auf eine Zelle zeigt, sondern direkt auf die Stelle, an der der Zeiger zu platzieren ist (siehe Abbildung 4.6).

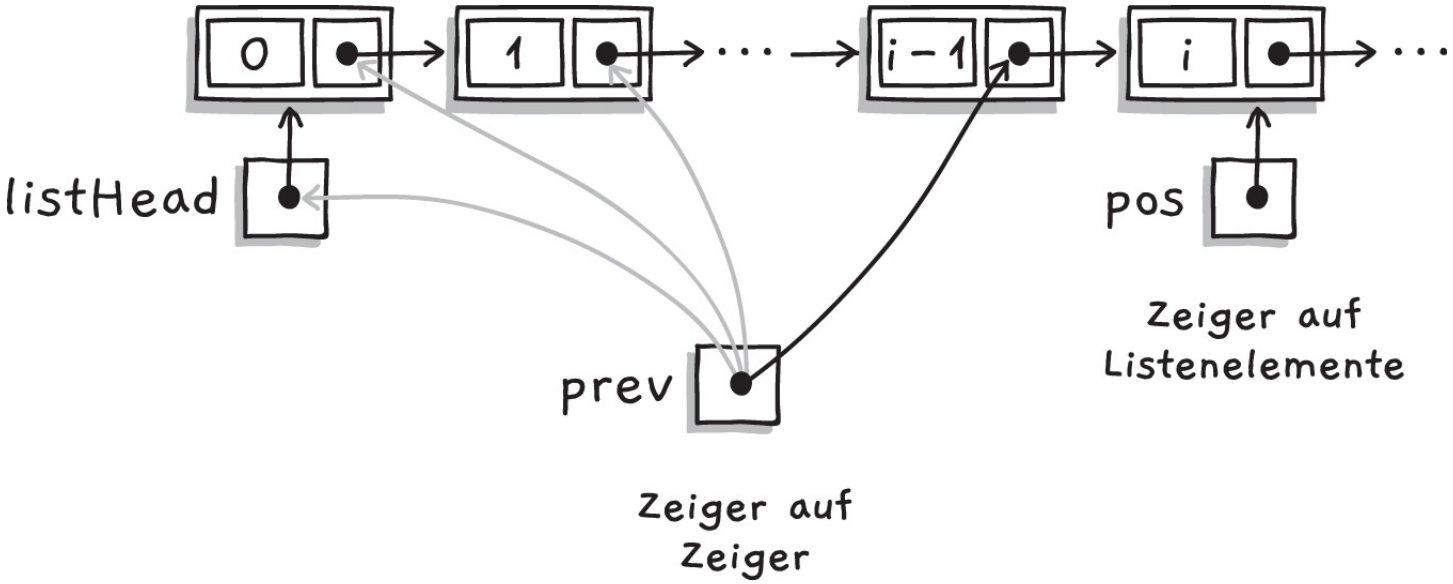


Abbildung 4.6: Einfügen in eine verkettete Liste mit Referenzen

Der Zeiger *prev* zeigt also nicht mehr auf eine ganze Zelle, sondern direkt auf die Variable, deren Wert verändert werden soll. Das ist entweder *listhead* oder der next-Zeiger innerhalb einer der Zellen:

```
object(prev) ← newCellPtr
object(newCellPtr).next ← next
```

Der gesamte Algorithmus in diesem Stil ist

```
InsertAtPos(x, index)
    newCellPtr ← new Cell(x, null)
    prev ← address(listHead)        Zeiger auf die Variable mit Zeiger auf den Anfang

    if object(prev) = null
        next ← null                 es gibt kein erstes Element
    else
        next ← object(prev)         das erste ist das erste nächste Element
        i ← 0
```

```
while i < index
    prev ← address(object(prev).next)      prev auf das next in der Zelle setzen
    next ← object(prev)                    next zeigt auf gleiche wir der Inhalt von prev
    i ← i + 1
object(prev) ← newCellPtr
object(newCellPtr).next ← next
```

Bisher haben wir es uns mit den Veränderungen ein wenig einfach gemacht. Veränderlich waren beim Sortieren nur einzelne Variablen und Arrays. Bei komplexeren Strukturen sind wir davon ausgegangen, dass sie unveränderlich sind. Das hat den großen Vorteil, dass wir uns nicht weiter um Zeiger kümmern müssen.



In Kapitel 3 wird beschrieben, wie algebraische Datentypen, also unveränderlich hierarchische Strukturen, systematisch mithilfe von Zeigern realisiert werden können und man sie dann nach dieser Klärung vergessen kann – solange nichts in ihrem Inneren und ihrer Struktur verändert wird.

So, genug der Zeiger. Tauchen wir auf aus den tiefsten Tiefen des schlammigen Gewässers voller Adressen, Zeiger, Zeiger auf Zeiger und ähnlichem Getier und begeben uns ins helle, manchmal gleißende Licht der funktionalen Abstraktionen. So runden wir unseren Eindruck von der Bandbreite der algorithmischen Themen von »ganz unten« nach »ganz oben« ab.

Alles ist ewig und Rekursion ist gut

Algorithmen und die Datenstrukturen, auf denen sie operieren, sind aufs Engste miteinander verknüpft. Ein guter Algorithmus kann sich in einen schlechten verwandeln, wenn die Daten, auf denen er arbeitet, ungeschickt oder unpassend organisiert sind. Wenn sich alle an der Kasse ordentlich anstellen, geht's schneller als dann, wenn die Kunden in einem großen Haufen herumstehen und die Reihenfolge der Bedienung ständig neu festgestellt werden muss.

Das ist zunächst einmal selbstverständlich. Dahinter steckt aber auch etwas Systematisches: Es gibt mit den funktionalen und den imperativen Algorithmen zwei Arten von Algorithmen. Sie basieren nicht nur auf unterschiedlichen Arten, algorithmisch zu denken, sie benötigen auch jeweils ihre eigene angemessene Art, die Daten zu organisieren. Wir beschäftigen uns in diesem Kapitel darum mit imperativen und funktionalen Algorithmen: Zwei Denkarten von Algorithmen, die auch ihre eigenen Datenstrukturen mit sich bringen. Funktionale und imperative Algorithmen haben also sozusagen ihre jeweils eigenen passenden Datenstrukturen.

Funktional ist mathematisch. Man definiert die Berechnung von Werten, oft mithilfe von rekursiven Funktionen. Imperativ bedeutet »gebieterisch« oder etwas weniger

hochtrabend »mit Anweisungen«. In einem imperativen Algorithmus werden vor allem Variablen angewiesen, ihren Wert zu ändern.



Bei der *funktionalen Beschreibung* eines Algorithmus wird eine konstruktive *Definition* der Problemlösung angegeben. Konstruktiv bedeutet dabei, dass es klar sein muss, dass und wie das Definierte berechnet werden kann.

Bei der *imperativen Beschreibung* eines Algorithmus wird beschrieben, wie die Problemlösung durch die schrittweise *Modifikation von Variablen* erreicht wird. Dabei ist ein beliebiger Abstraktionsgrad erlaubt, solange klar ist, dass und wie dies erreicht werden kann.

Zwar läuft alles letztlich auf einer Registermaschine ab, aber der Unterschied zwischen einem funktionalen und einem imperativen Algorithmus ist doch beträchtlich und möglicherweise größer, als man zunächst einmal vermutet. Funktionale und imperative Algorithmen benötigen jeweils ihre eigenen angepassten Datenstrukturen. Arrays sind beispielsweise der Prototyp einer imperativen Datenstruktur, so wie verkettete Listen die Mutter aller funktionalen Datenstrukturen sind.

Funktionale Datenstrukturen waren lange Zeit ein fortgeschrittenes Spezialthema. Mit der zunehmenden Verbreitung funktionaler Sprachen und funktionaler Konzepte auch in gängigen Programmiersprachen wie beispielsweise Java, Python oder Javascript ist das Thema aber inzwischen für immer mehr Entwickler praktisch relevant. Viele Algorithmen lassen sich funktional eleganter und weniger fehlerträchtig formulieren und die heute fast allgegenwärtige verteilte und Parallelverarbeitung ist mit funktionalen Konzepten einfacher zu beherrschen.

Der Pseudocode, der in der Literatur zu Datenstrukturen und Algorithmen verwendet wird, enthält schon immer Konstrukte, die wie funktionale Programmierung aussehen. Beispielsweise wird in

$$s \leftarrow \{f(x) \mid x \in S \text{ und } p(x)\}$$

beschrieben, dass s mit den durch f transformierten Werten einer Menge S belegt werden soll, wenn diese Werte das Prädikat p erfüllen. Eine solche Zuweisung ist traditionell als Bestandteil eines abstrakten Algorithmus zu lesen, bei dem die Festlegung der Datenstrukturen (noch) offen ist. Das heißt, es ist offen, durch welche Datenstruktur die vorkommenden Mengen repräsentiert werden.

In vielen Programmiersprachen mit funktionalen Konzepten kann dies aber als Anweisung fast genauso hingeschrieben werden. Das ist dann ganz konkret und hat nichts Abstraktes. Damit spart man sich natürlich viel Arbeit. Vor allem das Nachdenken über eine geeignete Datenstruktur für s. Bei dem Einsatz funktionaler Konzepte erreicht man so eine hohe *Entwickler-Effizienz*, die aber, vor allem bei einer ungeschickten Wahl der Datenstruktur, eventuell mit schlechter *Laufzeit-Effizienz* bezahlt wird. Ein kurzer Blick

hinter die Kulissen schadet also nicht.

Wir haben also zwei Gründe, uns zumindest ein wenig mit funktionalen Konzepten zu beschäftigen:

- ✔ Pseudocode hat viele funktionale Ausdruckselemente. Diese kann man heute oft wörtlich nehmen und sollte deshalb mit ihnen wenigstens ansatzweise vertraut sein.
- ✔ Algorithmen im funktionalen Stil und solche im imperativen Stil stellen unterschiedliche Anforderungen an die Datenstrukturen, auf denen sie agieren. Man sollte diese kennen und beachten.

Die Entscheidung über die richtige Datenstruktur wird in funktionalen Programmiersprachen oft an die Sprachimplementierung delegiert. Pseudocode ist aber keine funktionale Programmiersprache. Nicht die Implementierung irgendeiner Sprache soll etwas für uns tun. Wir sagen, was passieren soll, und sagen auch, eventuell an anderer Stelle, wie das passieren soll. – Oder aber: das *genaue Wie* ist selbstverständlich, beziehungsweise (vielleicht für die Laufzeitabschätzung) uninteressant und wird darum weggelassen. Sie wissen ja:


In Pseudocode wird nur über das geredet, was relevant und nicht selbstverständlich ist.

Listen in Funktionalistan

In Kapitel 3 haben wir strukturell rekursive Algorithmen betrachtet, also Algorithmen, die über die Struktur ihrer Argumente definiert sind. Solche Algorithmen lassen sich besonders schön in funktionaler Notation beschreiben. Im nächsten Abschnitt betrachten wir funktionale Algorithmen, die passenden Datenstrukturen und etwaige damit verbundene Probleme etwas genauer. Zunächst wollen wir aber ein wenig in die Thematik einstimmen.

Die bevorzugte Datenstruktur der funktionalen Algorithmiker ist die »Liste«. Damit ist etwas gemeint, das in einem nicht-funktionalen Kontext wohl eher als »Stapel« bezeichnet würde: Eine Liste, die einen unmittelbaren Zugriff nur auf ihr erstes Element und den Rest der Liste zulässt. Das lässt sich als algebraischer Datentyp definieren:

```
type List = ElementType :: List | Nil
```

Die beiden Doppelpunkte (::) sind eine übliche Infix-Notation für die Verbindung eines Elements und einer Liste zu einer neues List. Also für die Operation, die sonst oft *Cons* genannt wird.

Auf einer solchen Datenstruktur lassen sich strukturell-rekursive Algorithmen definieren, beispielsweise die Länge einer Liste:

```
Length(lst)
    case lst = Nil
        return 0
    case lst = head :: tail
        return 1 + Length(tail)
```

Das ist die Formulierung eines Algorithmus als Definition der Lösung. Man sieht das besser in einer mathematischen Umformulierung:

$$length(lst) = \begin{cases} 0 & \text{falls } lst = Nil \\ 1 + length(tail) & \text{falls } lst = head :: tail \end{cases}$$

oder auch

$$length(Nil) = 0$$

$$length(head :: tail) = 1 + length(tail)$$

Wir sehen in der mathematischen Notation zunächst einmal Gleichungen. Gleichungen sagen, dass die Dinge links und rechts vom Gleichheitszeichen gleich sind. Das sind Aussagen, keine Algorithmen. In einem Algorithmus wird etwas berechnet und keine Aussage über die Gleichheit von Dingen getroffen. Eine Gleichung kann man aber trotzdem als Handlungsanweisung verstehen: Gleiches kann durch Gleiches ersetzt werden. Beispielsweise kann $length(3 :: 2 :: 1 :: Nil)$ ersetzt werden durch $1 + length(2 :: 1 :: Nil)$. Natürlich auch umgekehrt.

$$length(3 :: 2 :: 1 :: Nil) \Leftrightarrow 1 + length(2 :: 1 :: Nil)$$

Wir bevorzugen aber die Richtung, in der $length$ irgendwann verschwunden ist:

$$\begin{aligned} & length(3 :: 2 :: 1 :: Nil) \\ & = 1 + length(2 :: 1 :: Nil) \\ & = 1 + 1 + length(1 :: Nil) \\ & = 1 + 1 + 1 + length(Nil) \\ & = 1 + 1 + 1 + 1 + 0 = 3 \end{aligned}$$

Eine Definition ist ja nur dann sinnvoll, wenn sie sich nicht im Kreis dreht und der Wert $length(3 :: 2 :: 1 :: Nil)$ auch angegeben werden kann, ohne dabei $length$ zu verwenden. Wenn eine solche *Reduktion* in einem endlichen Prozess möglich ist, dann ist die Funktionsdefinition durch Gleichungen »konstruktiv« und kann als Algorithmus verwendet werden.

Die Summe aller Elemente einer Liste berechnet folgender Algorithmus:

```
Sum(lst)
    case lst = Nil
        return 0
    case lst = head :: tail
        return head + Sum(tail)
```

Das Gleiche in mathematischer Notation:

$$sum(lst) = \begin{cases} 0 & \text{falls } lst = Nil \\ head + sum(tail) & \text{falls } lst = head :: tail \end{cases}$$

Die Verknüpfung von zwei Listen zu einer kann rekursiv über die Struktur der ersten Liste definiert werden:

```
Append(l1, l2)
    case l1 = Nil
        return l2
    case l1 = head :: tail
        return head :: Append(tail, l2)
```

In mathematischer Notation:

$$append(l_1, l_2) = \begin{cases} l_2 & \text{falls } l_1 = Nil \\ head :: append(tail, l_2) & \text{falls } l_1 = head :: tail \end{cases}$$

Quicksort funktional

In Kapitel 7 wird der Algorithmus QuickSort in einer imperativen Version vorgestellt (siehe Abschnitt *QuickSort: Quick and Easy*). Die Idee des Verfahrens sieht man eventuell einfacher in der funktionalen Formulierung. Quicksort ist ein Beispiel für eine rekursive Funktion, die *nicht* als strukturelle Rekursion definiert ist.



In Kapitel 3 wird *strukturelle Rekursion* erläutert. Es handelt sich dabei um die Gattung der rekursiven Algorithmen, die auf einer Fallunterscheidung auf ihrem Argument basieren, bei der *nur* dessen *Struktur* beachtet wird. Der Inhalt spielt keine Rolle. QuickSort gehört nicht dazu, denn es teilt nicht einfach sein Argument in dessen Komponenten (z.B. *Erster* und *Rest*) auf, sondern konstruiert für seine rekursiven Unteraufrufe neue Argumente. Argumente, die zwar kleiner sind als das übergebene Argument, aber nicht einfach aus diesem »herausgebrochen« wurden.

Rekursion, die nicht strukturell ist, das kann oft etwas knifflig werden. QuickSort in funktionaler Notation ist aber noch recht übersichtlich:

```
QuickSort(lst)                          sortiere lst mit dem alg. DT List
    case lst = Nil                      die leere Liste ist sortiert
        return Nil
    case lst = first :: rest
        pivot ← first                   wähle das erste Element als Pivot (Trenner)
        small ← Filter(lst, x ⇒ x < pivot)      alle, die kleiner als pivot sind
        equal ← Filter(lst, x ⇒ x = pivot)      alle, die gleich pivot sind sind
        large ← Filter(lst, x ⇒ x > pivot)      alle, die größer pivot sind sind
        return QuickSort(small) ::: equal ::: QuickSort(large)
```

Die drei Doppelpunkte (:::) haben wir als Operator für Append verwendet:

$lst1 ::: lst2 = \textbf{Append}(lst1, lst2)$

Der Algorithmus Filter filtert eine Liste mit einem Prädikat, also einer Funktion, die einen Wahrheitswert liefert:

```
Filter(lst, p)                          filtere die Liste lst mit einer Funktion p
    case lst = Nil
        return Nil
    case lst = head :: tail
        if p(head)
            return head :: Filter(tail, p)
        else
            return Filter(tail, p)
```

Diese Definitionen sind sehr konkret in der verwendeten Datenstruktur. Im Pseudocode muss man nicht zwingend so konkret sein. Wir können QuickSort auf dem *Datenmodell* »Liste« definieren, also auf einer beliebigen listenartigen Datenstruktur:

```
QuickSort(lst)                          sortiert eine Liste lst
    if lst = [ ]                        die leere Liste ist sortiert
        return [ ]
    else
        pivot ← lst[0]                  wähle das erste Element als Pivot (Trenner)
        small ← [x ∈ lst | x < pivot]       alle, die kleiner als pivot sind
        equal ← [x ∈ lst | x = pivot]       alle, die gleich pivot sind
        large ← [x ∈ lst | x > pivot]       alle, die größer pivot sind
        return QuickSort(small) + equal + QuickSort(large)
```

Hier wird eine Liste von Werten sortiert, die in irgendeiner listenartigen Datenstruktur stecken. Von dieser wird erwartet, dass ein indizierter Zugriff möglich ist. Die Laufzeit des Zugriffs ist offen. Sie kann beispielsweise einfach mit $\mathcal{O}(1)$ angenommen werden. Das ist dann eine Bedingung, welche die (später) gewählte Datenstruktur und ihre

Zugriffsoperationen erfüllen müssen. Oder es wird, wie oben mit dem algebraischen Datentyp List, eine konkrete Datenstruktur angegeben, bei der die Laufzeit klar bestimmt werden kann ($\mathcal{O}(n)$ bei List). Das Gleiche gilt für die Listenverknüpfung, die hier abstrakt mit $+$ bezeichnet wird. Die Laufzeit ist unbekannt. Bei einer konkreten Datenstruktur kann sie angegeben werden, beim alg. DT List mit Append als Implementierung von $+$ beispielsweise mit $\mathcal{O}(n)$, wobei n die Länge der ersten Liste ist.

Listenausdrücke

Ein Konstrukt wie $[x \in lst \mid x = pivot]$ oben in QuickSort wird *Listenausdruck* genannt. Es ist eine knappe Notation für eine Berechnung, die als Ergebnis (und wie hier oft auch als Argument) eine Liste hat. Eine solche Berechnung kann als Schleife interpretiert werden:

```
equal ← [x ∈ lst | x = pivot]      alle, die gleich pivot sind
```

ist nichts anderes als

```
equal ← [ ]                        initialisiert mit der leeren Liste
for x ← lst                        für jedes x in lst
    if x = pivot                   wenn es gleich pivot ist, ...
        equal ← equal + [x]            ... dann kommt es zu equal hinzu
```

Die Verwendung von Listenausdrücken ermöglicht die knappe und übersichtliche Definition von Algorithmen. Manchmal wirkt sie vielleicht auch ein wenig kryptisch. In dem Fall kann die Schleife explizit hingeschrieben werden.



Listen-Ausdrücke können als Schleifen beziehungsweise als endrekursive Funktionen gelesen werden. Mit etwas Übung ist das gar nicht so schwierig. Dabei helfen die folgenden Schemata:

```
                          result ← [ ]
[ f(x) | x ∈ L ]   ~      for x ← L
                              result ← result + [f(x)]

                          result ← [ ]
[ x ∈ L | p(x) ]   ~      for x ← L
                              if p(x)
                                  result ← result + [x]
```

Die Schleifen können als endrekursive Funktion formuliert werden:

$T(l) = [\, f(x) \mid x \in L \,] \;\sim$

```
T(L)
    case L = [ ]
        return [ ]
    case L = [x, y, ..., z]
        retutn [f(x)] + T([y, ..., z])
```

$F(L) = [\, x \in L \mid p(x) \,] \;\sim$

```
F(L)
    case l = [ ]
        return [ ]
    case L = [x, y, ..., z]
        if p(x)
            return [x] + F([y, ..., z])
        else
            return F([y, ..., z])
```

Hier haben wir es mit einer allgemeinen Formulierung zu tun, die für beliebige Datenstrukturen gültig ist, welche das Datenmodell »Liste« realisieren. Wenn die Listen auf konkrete Datentypen spezialisiert werden, dann können die allgemeinen Operationen durch konkrete Operationen auf diesem Datentyp ersetzt werden. Für den algebraischen Datentyp

```
type List = ElementType :: List | Nil
```

werden die Schemata konkret zu

$T(L) = [\, f(x) \mid x \in L \,] \;\sim$

```
T(L)
    case L = Nil
        return Nil
    case L = x :: rest
        return f(x) :: T(rest)
```

$F(L) = [\, x \in L \mid p(x) \,] \;\sim$

```
F(L)
    case l = Nil
        return Nil
    case L = x :: rest
        if p(x)
            return x :: F(rest)
        else
            return F(rest)
```

Das Transformieren und Filtern von Listen kann mit einer *Funktion höherer Ordnung* verallgemeinert werden. Das sind Funktionen, die eine Funktion als Ergebnis liefern oder wie hier Funktionen, die eine andere Funktion als Parameter haben:

$[\, f(x) \mid x \in L \,] \sim \mathsf{Map}(l, f)$

```
Map(l, f)
    case l = Nil
        return Nil
    case l = x :: rest
        return f(x) :: Map(rest, f)
```

$[\, x \in L \mid p(x) \,] \sim \mathsf{Filter}(l, p)$

```
Filter(l, p)
    case l = Nil
        return Nil
    case l = x :: rest
        if p(x)
            return x :: Filter(rest, p)
        else
            return Filter(rest)
```

For-Schleife auf dem algebraischen Datentyp Liste

Die For-Schleife gehört zu den »universellen Operationen«, die auf jeder Liste definiert sind und von denen wir annehmen, dass in ihnen Zugriffe auf Listenelemente in $\mathcal{O}(1)$ mit einem irgendwie gearteten Iterator erfolgen. Der algebraische Datentyp List hat zunächst einmal keinen Iterator, aber er kann leicht definiert werden:

```
Iterator(l) = record [
    pos ← l
    HasNext()
        return pos ≠ Nil
    Next()
        case pos = head :: tail
            pos ← tail
            return head
]
```

Damit kann *in einer For-Schleife*, die wie allgemein angenommen mit dem Iterator arbeitet, auf jedes Element der Liste von Typ List mit $\mathcal{O}(1)$ zugegriffen werden.

Um den Umgang mit funktionalen Algorithmen, Rekursion, Listen und Listenausdrücken noch etwas zu vertiefen, betrachten wir die Erzeugung aller Permutationen einer Liste. Ein Problem, das erstmal rein akademisch erscheint, aber doch gelegentlich Bestandteil von

realen Algorithmen ist.



In Kapitel 11 geht es um die Suche in einen Suchraum möglicher Lösungen. Solche Suchräume können oft als Permutationen definiert werden. Eine Sortierung ist beispielsweise nichts anderes als eine bestimmte Permutation. Generieren wir also alle Permutation, wird eine davon sortiert sein.

Permutationen funktional

Eine Permutation ist eine Umsortierung einer Liste. Der Algorithmus Perm soll alle Permutationen einer Liste erzeugen. Zunächst ein Beispiel:

$$Perm([a,b,c]) = [[a,b,c],[a,c,b],[b,a,c],[c,a,b],[b,c,a],[c,b,a]]$$

Permutationen können mit einem Verfahren nach dem Prinzip *Reduzieren und Herrschen* bestimmt werden.



Reduzieren und Herrschen ist ein Spezialfall von Teilen und Herrschen. In Kapitel 12 wird Teilen und Herrschen ausführlich besprochen.

Die Berechnung der Permutationen von $[a,b,c]$ wird dabei zunächst auf die einfachere Berechnung der Permutationen der Ausgangsliste ohne das erste Element reduziert (also auf $[b,c]$). Die reduzierte Problemstellung wird gelöst:

$$Perm([b,c]) = [[b,c],[c,b]]$$

Die Lösung des reduzierten Problems wird dann zur gesuchten Lösung erweitert.

$$Perm([b,c]) = [[b,c],[c,b]] \Rightarrow Perm([a,b,c])$$

Hier besteht die Erweiterung darin, dass das erste Element a in jedem Element von $Perm([b,c])$ (das sind Listen) an jede mögliche Stelle gesetzt wird.

Die Erweiterung $Perm([b,c]) \Rightarrow Perm([a,b,c])$ ist:

$$\begin{aligned} Perm([\mathbf{a},b,c]) &= Ins(\mathbf{a}, Perm([b,c]) \\ &= Ins(\mathbf{a}, [[b,c],[c,b]]) \\ &= InsH(\mathbf{a}, [b,c]) + InsH(\mathbf{a}, [c,b]]) \\ &= [[\mathbf{a},b,c],[b,\mathbf{a},c],[b,c,\mathbf{a}]] + [[\mathbf{a},c,b],[c,\mathbf{a},b],[c,b,\mathbf{a}]] \\ &= [[\mathbf{a},b,c],[b,\mathbf{a},c],[b,c,\mathbf{a}],[\mathbf{a},c,b],[c,\mathbf{a},b],[c,b,\mathbf{a}]] \end{aligned}$$

$Perm$ erzeugt eine Liste von Listen. Die Permutationen einer leeren Liste sind die Liste, die nur die leere Liste enthält. Zur Erzeugung der Permutationen einer nicht-leeren Liste wird $Perm$ auf die um eins verkürzte Liste aufgerufen und das Ergebnis mit Ins zum

Gesamtergebnis erweitert. Ins nutzt dabei seinen Helfer $InsH$, um einzelne Listen zu bearbeiten (siehe unten und Abbildung 4.7).

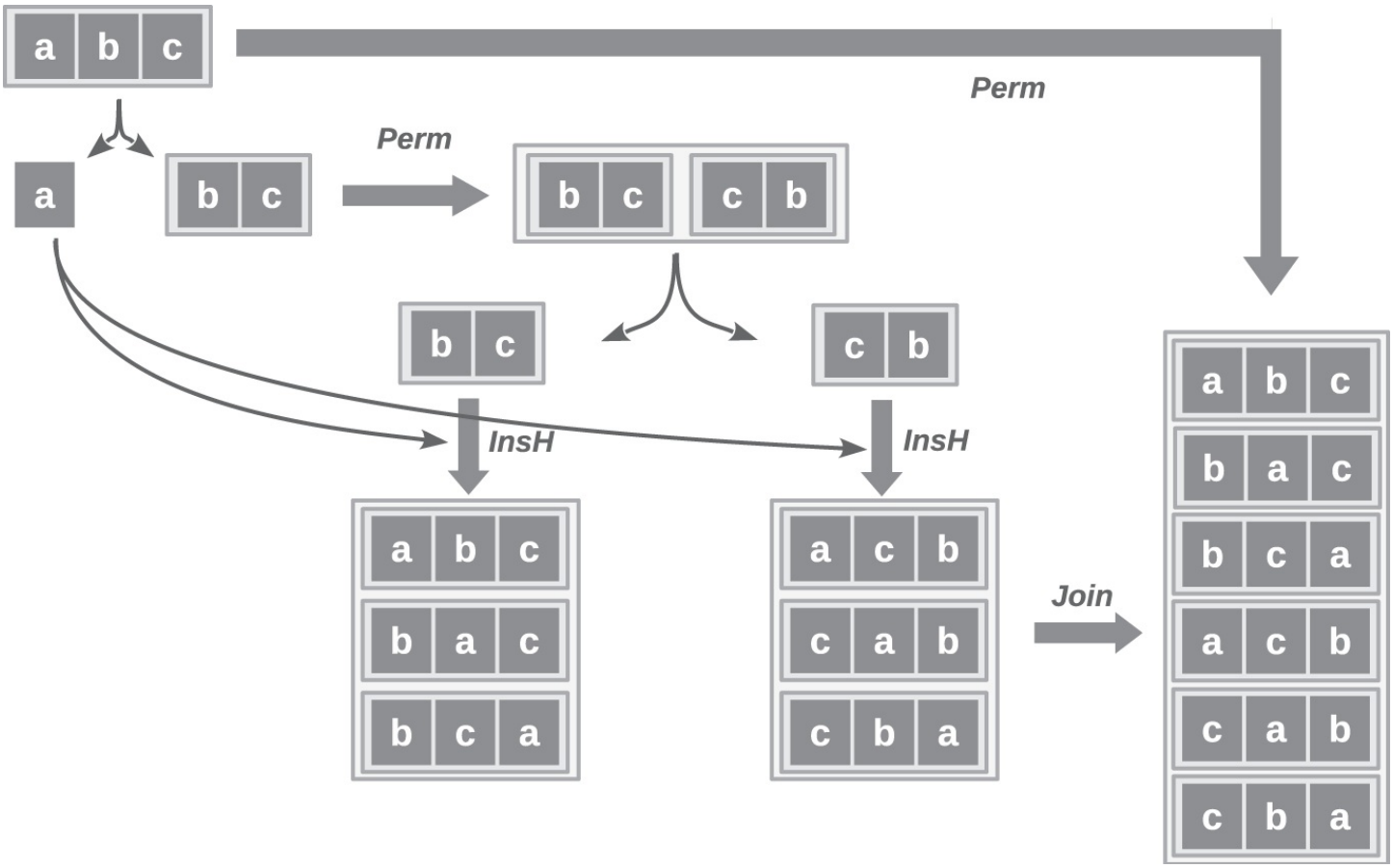


Abbildung 4.7: Alle Permutationen rekursiv erzeugen: Beispiel

Insgesamt:

$$Perm(l) = \begin{cases} [\,[\,]\,] & \text{falls } l = [\,], \\ Ins(x_1, Perm([x_2, \ldots, x_n])) & \text{falls } l = [x_1, x_2, \ldots, x_n] \end{cases}$$

$Perm$ liefert eine Liste von Listen. Ins muss also ein Element auf alle möglichen Arten in jede Liste einer Liste einfügen. Dazu geben wir ihm einen kleinen Helfer $InsH$, der die Aufgabe für eine Liste erledigt.

Die Suche nach den diversen Möglichkeiten, ein Element in *eine* Liste einzufügen, kann wieder mit Reduzieren und Herrschen gelöst werden: Will man a auf alle möglichen Arten in $[b, c]$ einfügen, dann hat man die Möglichkeiten

- ✔ **Reduktion:** a auf alle möglichen Arten in $[c]$ einfügen, man erhält $[[a, c], [c, a]]$
- ✔ **Erweiterung:**
  - Es wird b vor jedes Element von $[[a, c], [c, a]]$ gesetzt, das ergibt $[[b, a, c], [b, c, a]]$.

- Dazu nimmt man noch $[b, c]$ mit vorgesetztem a, also $[a, b, c]$ und hat das Gesamtergebnis: $[[a, b, c], [b, a, c], [b, c, a]]$

Das *auf jede Art Einfügen* ist die Aufgabe von Ins. Ihm hilft der Einfügehelfer *InsH*. Wir betrachten wieder ein Beispiel für den Helfer *Füge* a *auf jede mögliche Art in* $[b, c]$ *ein* (siehe Abbildung 4.7):

$$\begin{aligned} InsH([a,[b,c]) &= [\ [a,b,c]\ ] + [\ [b] + [\ l \mid l \in InsH(a,[c])\ ] \\ &= [[a,b,c]] + [\ [b] + [\ l \mid l \in [[a,c],[c,a]]\ ] \\ &= [[a,b,c]] + [[b,a,c],[b,c,a]] \\ &= [[a,b,c],[b,a,c],[b,c,a]] \end{aligned}$$



Entwickle Algorithmen nach dem Prinzip *Teilen und Herrschen* an *Beispielen* und nach dem Motto: *Angenommen, ich habe das Problem für eine kleinere Problemstellung gelöst. Würde das bei der Lösung des Gesamtproblems helfen und wenn ja, wie?*

Das Beispiel führt uns zu der allgemeinen Definition

$$InsH(x, [\ ]) = [[x]]$$

$$InsH(x, [l_1, l_2, \ldots, l_n]) = [x, l_1, \ldots, l_n] + [\ [l_1] + l' \mid l' \in InsH(x, [l_2, \ldots, l_n])\ ]$$

Das sieht auf den ersten Blick vielleicht etwas einschüchternd aus, darum zerpflücken wir es etwas. Der Basisfall

$$InsH(x, [\ ]) = [[x]]$$

bedeutet, dass es nur eine einzige Möglichkeit gibt, x in die leere Liste einzufügen, und zwar die Liste $[x]$. Das Ergebnis, eine Liste von Listen, hat ein Element, die Liste $[x]$.

Etwas beeindruckender ist der rekursive Fall.



Wir haben es mit einem Listenausdruck zu tun. Im letzten Abschnitt wurde beschrieben, wie man mit diesen schematisch umgehen kann.

Mit

$$InsH(x, [l_1, l_2, \ldots, l_n]) = [x, l_1, \ldots, l_n] + [\ [l_1] + l' \mid l' \in InsH(x, [l_2, \ldots, l_n])\ ]$$

wird beschrieben, wie x in eine nicht-leere Liste $[l_1, l_2, \ldots, l_n]$ eingefügt wird. Mit

$$InsH(x, [l_2, \ldots, l_n])$$

wird x zunächst auf alle möglichen Arten in $[l_2, \ldots, l_n]$ eingefügt. Das erste

Listenelement l_1 kommt dann vor jede Liste l' im Ergebnis von $InsH(x, [l_2, \ldots, l_n])$:

$$[\ [l_1] + l' \mid l' \in InsH(x, [l_2, \ldots, l_n])\ ]$$

und schließlich wird x auch noch vor die gesamte Liste gesetzt:

$$[x, l_1, \ldots, l_n]$$



Listen-Ausdrücke wie $[\ [l_1] + l' \mid l' \in InsH(x, [l_2, \ldots, l_n])\ ]$ können stets als Schleifen gelesen werden:

```
result ← [ ]
for l' ← InsH(x, [l2, ..., ln])
    result ← result + [[l1] + l']
```

$Perm$ muss den Einfügehelfer $InsH$ auf jedes Element der Permutation der Teilliste anwenden und die Ergebnisse zusammensetzen. Der gesamte Algorithmus in mathematischer Notation ist damit kurz und bündig:

$$Perm(l) = \begin{cases} [\ [\ ]\ ] & \text{falls } l = [\ ], \\ Ins(x_1, Perm([x_2, \ldots, x_n])) & \text{falls } l = [x_1, x_2, \ldots, x_n] \end{cases}$$

$$Ins(x, L) = Join([\ InsH(x, l) \mid l \in L])$$

$$Join([l_1, \ldots, l_n]) = l_1 + \ldots + l_n$$

Join nimmt eine Liste von Listen und vereinigt diese zu einer Gesamtliste.

Permutationen eines algebraischen Datentyps Liste

Die Erzeugung der Permutationen wurde als Funktion definiert, die ganz allgemein auf irgendwie gearteten Listen operiert. Für die Auswahl einer konkreten Datenstruktur sieht man sich an, welche Operationen der Algorithmus auf den Listen ausführt. Bei unserem Beispiel der Permutationen ist der simple algebraische Datentyp

```
type List = ElementType :: List | Nil
```

als Datenstruktur völlig ausreichend. Formulieren wir also die Funktion so um, dass sie mit Listen von diesem Typ arbeitet:

```
Perm(l)
    case l = Nil
        return l = Nil :: Nil          Liste von Listen mit der leeren Liste als einzigem Element
    case l = x :: rest
        return Ins(x, Perm(rest))      x auf alle Arten in Perm(rest) einfügen
Ins(x, L)                              füge x in jeder Art in jede Liste von L ein
    return Join(Map(L, l ⇒ InsH(x, l)))
```

Map wendet eine Funktion auf jedes Element einer Liste an. Wir benötigen das, um $[\ InsH(x, l)\ |\ l \in L]$ auf unserer Datenstruktur zu realisieren (siehe Abbildung 4.8):

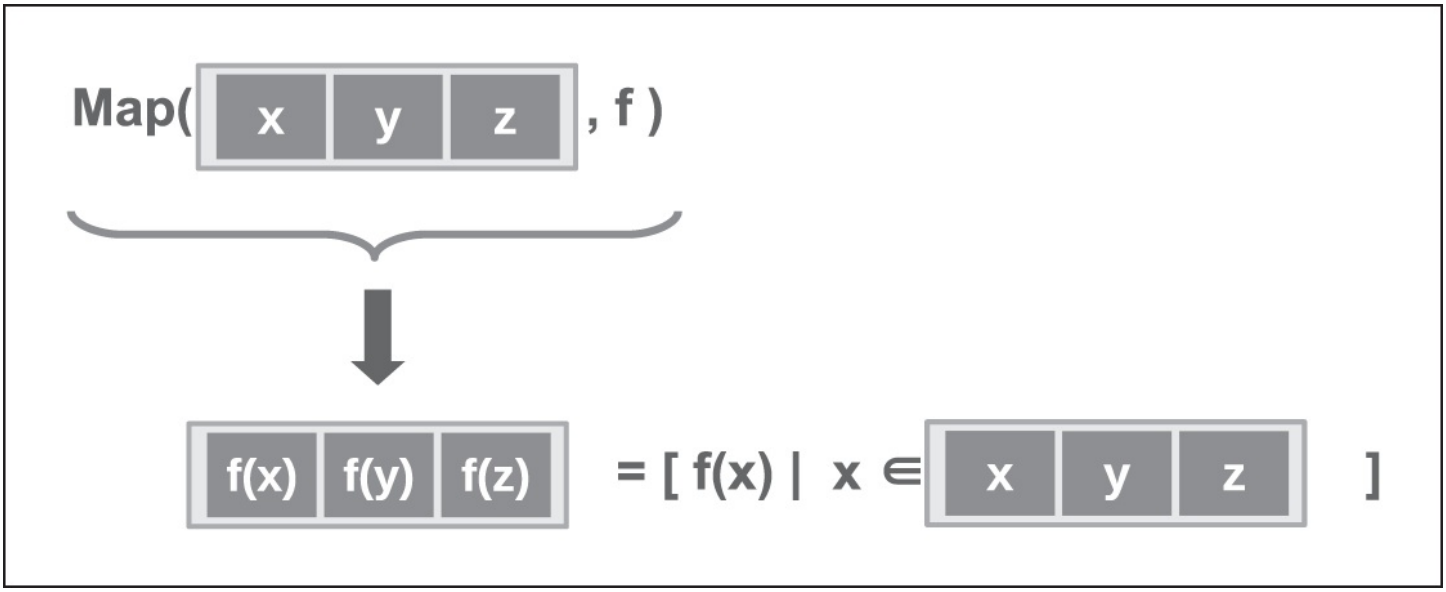


Abbildung 4.8: Map: Eine Funktion auf alle Elemente einer Liste anwenden

```
Map(l, f)
    case l = Nil
        return Nil
    case l = h :: t
        return f(h) :: Map(t, f)
```

Join ist die Vereinigung einer Liste von Listen zu einer einzigen Liste:

```
Join(L)
    case L = Nil
        return Nil
    case L = l :: T
        return Append(l, Join(T))
```

Append ist unsere Funktion zum Zusammenfügen von zwei Listen von oben. Bleibt nur noch der Einfügehelfer *InsH*. Der Basisfall ist einfach:

$$InsH(x, []) = [[x]]$$

```
InsH(x, l)
    case l = Nil
        return (x :: Nil) :: Nil
```

Der rekursive Fall sieht etwas anspruchsvoller aus:

$$InsH(x, [l_1, l_2, \ldots, l_n]) = [x, l_1, \ldots, l_n] + [\ [l_1] + l'\ |\ l' \in InsH(x, [l_2, \ldots, l_n])\ ]$$

Auch hier kann die Aktion *setzte* l_1 *vor jede Liste in* $InsH(x, t)$, das heißt

$$[\ [l_1] + l' \mid l' \in InsH(x, [l_2, \ldots, l_n])\ ]$$

wieder durch Map erledigt werden (siehe Abbildung 4.9).

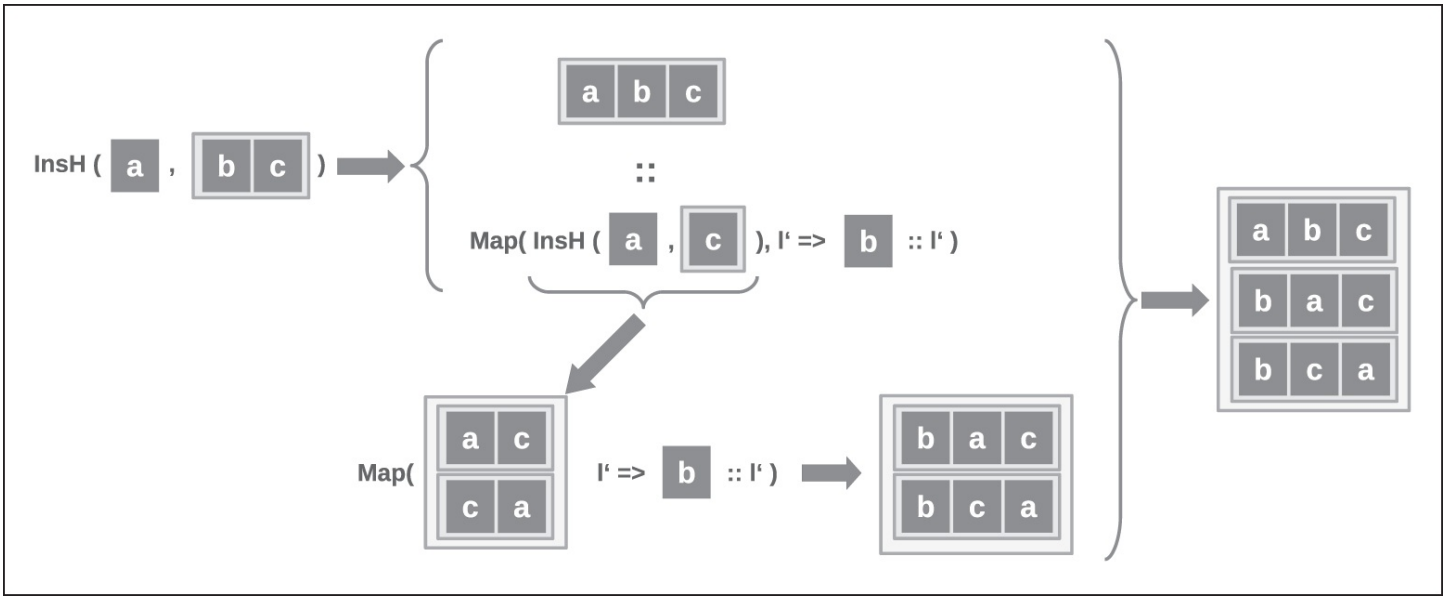


Abbildung 4.9: Einfügehelfer InsH: Ein Element auf alle Arten in eine Liste einfügen

```
InsH(x, l)                          füge x in jeder Art in die Liste l
    case l = Nil
        return (x :: Nil) :: Nil
    case l = h :: t
        return (x :: l) :: Map(InsH(x, t), l' ⇒ h :: l')
```

Alles zusammen in einer Übersicht:

```
Perm(l)                             alle Permutationen einer Liste l
    case l = Nil
        return l = Nil :: Nil
    case l = x :: rest
        return Ins(x, Perm(rest))
Ins(x, L)                           füge x in jeder Art in jede Liste von L ein
    return Join(Map(L, l ⇒ InsH(x, l)))
InsH(x, l)                          füge x in jeder Art in die Liste l ein
    case l = Nil
        return (x :: Nil) :: Nil
    case l = h :: t
        return (x :: l) :: Map(InsH(x, t), l' ⇒ h :: l')
```

mit den allgemein verwendbaren Hilfsfunktionen Join und Map.

So, nach dieser Trainingseinheit in funktional formulierten Algorithmen auf Listen allgemein und speziell auf einem algebraischen Datentyp List, können wir etwas tiefer in

das Reich der funktionalen Algorithmen und Datenstrukturen einsteigen. Der Begriff der *Persistenz* spielt dabei eine zentrale Rolle. Dinge sind *persistent*, wenn sie dauerhaft, vielleicht sogar völlig unvergänglich und ewig sind.

Persistente Datenstrukturen

Der eine kann gut erklären, der andere kann es gut machen. Der eine sagt, was eine Fibonacci-Zahl ist, der andere rechnet sie aus oder experimentiert gar mit Kaninchen. Damit sind wir schon bei dem wichtigsten Unterschied zwischen funktional und imperativ. Die funktionale Definition eines Algorithmus ist mathematisch. Sie definiert den gesuchten Wert. Typischerweise mit Rekursion. In der Mathematik gibt es ja keine Schleifen. In Schleifen werden Dinge verändert. Das ist Machen, Tun, Herumhantieren. Die Mathematik kennt so etwas nicht. Es gibt dort keine Veränderung, alles ist alles ewig und wahr.

Eine Zuweisung in einem Algorithmus verändert den Wert einer Variablen. In

$$x \leftarrow 5$$
$$x \leftarrow x + 1$$

wird der Wert von x von 5 auf 6 erhöht. Die 5 wird dabei nicht in eine 6 verwandelt. Sie ist immer noch da (existiert noch), wenn auch nicht mehr in x. Etwas anders sieht es aus bei

$$a \leftarrow \textbf{array}[7, 6, 9, 8]$$
$$\textbf{Sort}(a)$$

wobei Sort irgendein Sortieralgorithmus ist. Nach der Ausführung von Sort hat a einen neuen Wert. Statt

array[7, 6, 9, 8]

enthält a jetzt

array[6, 7, 8, 9]

Der alte Wert, 7, 6, 9, 8 in dieser Reihenfolge, ist aber nicht, wie die 5 oben, noch irgendwo »da«. Er ist verschwunden. Nicht die einzelnen Zahlen, aber die Zahlen in der Reihenfolge. Vielleicht auch nicht in unserem Kopf, im Speicher ist er aber weg. Jetzt mag man vielleicht sagen, dass es nie schlimm ist, wenn Chaos durch Aufräumen beseitigt wird. Wem jemals der Schreibtisch von einer wohlmeinenden anderen Person aufgeräumt wurde, weiß, dass es durchaus gravierende Unterschiede in der Einschätzung von Chaos und Ordnung geben kann und dass Ordnung-Schaffen im funktionalen Stil, also mit *persistenten* Datenstrukturen, seine Vorteile gehabt hätte:

> »Ich habe mal deinen Schreibtisch aufgeräumt. Gefällt dir das? Wenn nicht, kein Problem: Im Keller steht er noch in seiner alten unordentlichen Version.«

Ob Schreibtisch oder Array, der alte Wert oder Zustand von a mag ja an anderer Stelle oder für andere Algorithmiker durchaus von Wert sein. Wenn also mehr als ein Akteur im Spiel ist, dann wird es kritisch: Vorsicht bei Veränderungen. Bitte nur mit Absprachen und gegenseitiger Synchronisation. Wenn man einfach gar nichts verändert, dann gibt es auch ohne lästige und für Missverständnisse anfällige Absprachen keine Probleme. Das ist Persistenz, die funktionale Art, Konflikte beim Umgang mit Daten zu vermeiden.

Aus diesem Grund sind Änderungen in Algorithmen im funktionalen Stil verboten. Genauer gesagt sind sie nicht verboten, es gibt sie gar nicht. So wie 5 sich durch keine Addition in eine 6 verwandeln lässt, so lässt sich in einem funktionalen Algorithmus ein **array** [7, 6, 9, 8] durch nichts und niemand in ein **array** [6, 7, 8, 9] verwandeln. Es kann nur eine neue Version erzeugt werden.

Das bedeutet zunächst einmal, dass Sort im funktionalen Stil ein Ergebnis produzieren muss:

$$a \leftarrow \textbf{array}[7, 6, 9, 8]$$
$$a \leftarrow \textbf{Sort}(a)$$

Damit erhält a einen neuen Wert.



Rein funktionale Sprachen und Zuweisungen: In rein funktionalen Sprachen, wie beispielsweise Haskell, sind Zuweisungen an Variablen nicht erlaubt. Genauer gesagt gibt es dort gar keine Variablen und Zuweisungen. Es gibt nur Werte, denen man Namen geben kann. In diesen Sprachen ist es nicht erlaubt, den gleichen Namen für zwei unterschiedliche Werte zu verwenden.

Zwei Definitionen (unsinnig):	Zwei Zuweisungen (Ok):
$a = \textbf{array}[7, 6, 9, 8]$	$a \leftarrow \textbf{array}[7, 6, 9, 8]$
$a = \textbf{Sort}(a)$	$a \leftarrow \textbf{Sort}(a)$

Persistente (funktionale) Datenstrukturen können aber auch in Sprachen verwendet werden, in denen Variablen und Veränderungen nicht grundsätzlich verboten sind. Pseudocode ist keine rein funktionale Sprache. Das Konzept der Variablen ist erlaubt und es wird unterschieden zwischen Gleichheit ($=$) und Zuweisungen ($\leftarrow$).

Der Wert von a wird nicht verändert, er wird ausgetauscht, durch einen neuen ersetzt. Der alte Wert darf dabei nicht verloren gehen, sondern bleibt weiter bestehen. Genau wie die 5, die nicht zur 6, wird und weiter verfügbar bleibt. Er muss dazu *persistent* (haltbar) gemacht werden. Die Sortierung muss etwas Neues schaffen und das Letzte (Alte) muss erhalten bleiben (siehe Abbildung 4.10).

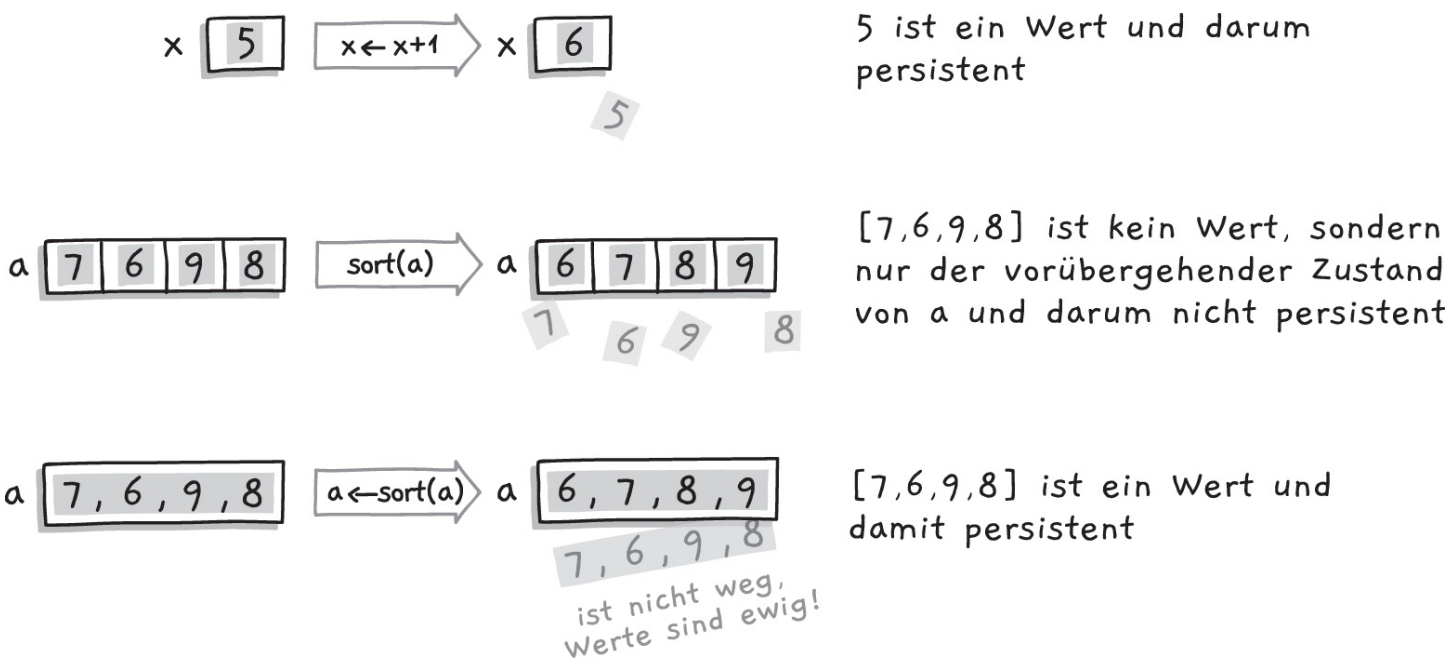


Abbildung 4.10: Persistenz: Was ein Wert ist, das vergeht nicht.

Das Verfügbarhalten hat gewichtige Konsequenzen. Eine 5 wird durch wenige Bits repräsentiert, die jederzeit reproduziert werden können. Bei einer längeren Sequenz von Werten sieht das schon anders aus. Jede Version der Datenstruktur muss verfügbar bleiben. Bei jeder Aktion aber eine komplette Kopie neu anzulegen, das ist nicht besonders effizient. Man versucht darum, das Ausmaß des Kopierens so weit wie möglich zu reduzieren. Ein wichtiges Thema funktionaler Algorithmen ist darum die Minimierung von Kopieraktionen ohne Beeinträchtigung der Persistenz.

Bei manchen Datenstrukturen geht das gut, bei verketteten Listen beispielsweise geht es recht gut. Andere Datenstrukturen, Arrays vor allem, sind für den funktionalen Stil eher ungeeignet. Im folgenden Beispiel eines einfachen Sortieralgorithmus, InsertionSort, wollen wir den Einfluss des funktionalen Stils auf die Datenstrukturen etwas genauer betrachten.



Im Abschnitt *InsertionSort: Spielkarten dazwischen schieben* in Kapitel 7 wird dieses Sortierverfahren im imperativen Stil besprochen.

Ordnung herstellen: imperativ und funktional

InsertionSort ist das Sortierverfahren, mit dem man Spielkarten in der Hand sortiert. Man teilt die Karten in zwei Bereiche, einen sortierten und einen unsortierten, nimmt dann eine Karte nach der anderen aus dem unsortierten Bereich und fügt sie im sortierten Bereich an der richtigen Stelle ein. Am Anfang sind alle Karten im unsortierten und am Ende im sortierten Bereich.

```
InsertionSortI(a: array[0..n − 1])      InsertionSort, imperative Variante
    for j ← 1 .. n − 1                  links von j ist a sortiert
        key ← a[j]                      der nächste Wert, key, soll einsortiert werden
        i ← j − 1
        while i > −1 and a[i] > key     richtigen Platz für key suchen und frei machen
            a[i + 1] ← a[i]
            i ← i − 1
        a[i + 1] ← key                  key einsortieren
```

Diese imperative Version des Sortierens *verändert* in einer Schleife das übergebene Array. Dabei wird die Grenze zwischen *sortiert* und *unsortiert* als Wert der Variablen j in der For-Schleife von links nach rechts vorangeschoben. Wenn j rechts ankommt, dann ist a sortiert.

Die Veränderung von a ist der Kern des imperativen Algorithmus. Veränderung gibt es in der funktionalen Welt nicht. Hier werden Funktionen definiert und die liefern Werte. Sortieren ist darum eine Funktion, die einen Wert liefert. Also lassen wir den Algorithmus einen Wert liefern, warum nicht das sortierte Array:

```
InsertionSortF(a: array[0..n − 1])      InsertionSort, Versuch einer funktionalen Variante
  for j ← 1 .. n − 1
    key ← a[j]
    i ← j − 1
    while i > −1 and a[i] > key
      a[i + 1] ← a[i]
      i ← i − 1
    a[i + 1] ← key
  return a
```

So nicht! Das ist Trickserei. Veränderungen sind verboten. Auch die Veränderung von a innerhalb des Algorithmus ist verboten. Warum? Nun, es fehlt die Persistenz. In einem funktionalen Algorithmus müssen die Datenstrukturen unveränderlich sein. Es könnte sein, dass a noch an anderer Stelle benutzt wird, vielleicht sogar gleichzeitig mit unserer Sortiererei, oder einem anderen gefällt es aus irgendeinem Grund nicht, dass wir Ordnung geschaffen haben. Der Sinn der funktionalen Programmierung liegt darin, ein derartiges Sich-in-die-Quere-Kommen grundsätzlich zu vermeiden, auch ohne dass dazu besondere Überlegungen der Softwareentwickler notwendig sind.

Für die funktionale Version des Sortierens müssen wir funktional denken. Das heißt, wir müssen das Sortieren als eine Aktion sehen, bei der eine unsortierte Datenkollektion angenommen wird und eine völlig neue konstruiert wird, die die gleichen Elemente in sortierter Reihenfolge enthält.

Die Lösung ist damit schon formuliert: Das Ergebnis enthält die gleichen Elemente wie

die Eingabe, nur in sortierter Reihenfolge. Die gleichen Elemente, nur in anderer Reihenfolge, das nennt man *Permutation*. Das Sortieren als Funktion kann darum als das Finden einer sortierten Permutation definiert werden. Das ergibt eine erste Version der Sortierung im Geist der funktionalen Algorithmen: Gehe alle Permutationen der Elemente durch, bis du eine gefunden hat, in der die Elemente auf- beziehungsweise absteigend angeordnet sind.

```
SortP(a)                                  Sortieren durch Permutieren
  for p ← alle Permutationen von a
    if p ist sortiert
      return p
```

Das ist ein Algorithmus, er ist funktional formuliert und liefert das gewünschte Ergebnis. Es ist aber ein sehr schlechter, weil langsamer Algorithmus.



In Kapitel 11 werden Algorithmen besprochen, die darauf beruhen, dass alle Kandidaten so lange durchsucht werden, bis sich einer als Lösung herausstellt. Eine derartige *erschöpfende Suche* ist sehr aufwendig und sollte nur dann zum Einsatz kommen, wenn man sonst keine Lösungsidee hat.

Ist funktional also immer schlecht und langsam, weil nach dem Gewünschten einfach nur gesucht wird? Nein, eine schlechte Idee ist eine schlechte Idee und keine Idee ist keine Idee, egal ob funktional oder imperativ. Kann man die Idee, die in InsertionSort steckt, in die funktionale Welt übertragen? Was ist überhaupt die Idee?

Reduzieren und Herrschen ohne und mit Rekursion

Sortieren durch Permutieren ist schlecht. Nicht weil es funktional ist, sondern weil es so uninspiriert ist. Hat man keine Idee, wie man zur Lösung kommen könnte, dann bleibt nur die Suche. Die entscheidende Idee von InsertionSort ist:

> Eine sortierte Folge bleibt eine sortierte Folge, wenn man ein weiteres Element an der richtigen Stelle einfügt.

Kann diese Idee auch mathematisch, funktional ausgedrückt werden? Klar, das geht. Man muss sich dazu nur kurz mal auf den Kopf stellen. Ein Kopfstand ist nicht nur eine gute Yoga-Übung, er hilft auch bei der Entwicklung von Algorithmen. Man denkt im imperativen Algorithmus aufbauend, Bottom-up: Wie kann der sortierte Bereich vergrößert werden? Das ist die Technik *Reduzieren und Herrschen* im Kopfstand.



In Kapitel 12 wird die algorithmische Technik *Teilen und Herrschen* besprochen. Bei dieser Technik geht es darum, wie man ein Problem löst, indem man die Lösungen von Teilproblemen zu einer Gesamtlösung kombiniert. Wenn es nur um ein einziges Teilproblem geht, dann spricht man oft von *Reduzieren* statt von Teilen.

Auf unser Sortierproblem angewendet, würde Reduzieren und Herrschen zum Beispiel bedeuten: Sortiere ein kleines Array und schau dann, wie die Lösung zu einer Sortierung des gesamten Arrays ausgebaut werden kann.

Rekursion, Top-down: Reduziere das Array auf ein kleines, dessen Sortierung zu einer Sortierung des gesamten Arrays ausgebaut werden kann.

Schleife, Bottom-up: Baue ein kleines sortiertes Array zu einem sortierten etwas größeren Array aus.

Das sind zwei Sichten der gleichen Strategie. In unserem konkreten Fall:

Auf den rekursiven Kopf gestellt:

- ✔ Nimm das erste Element aus dem zu sortierenden Teil,
- ✔ Sortiere den kleineren Rest und
- ✔ füge das Element in den sortierten Rest ein.

Imperativ, Bottom-up:

- ✔ Nimm das nächste Element aus dem zu sortierenden Teil und
- ✔ füge es in den sortierten Teil ein.

Warum ist die eine Sicht der Dinge imperativ und die andere funktional? Nun, eine Schleife arbeitet mit der Veränderung von Variablen. Das sind Aktionen, Veränderungen, nichts Funktionales. Die andere, rekursive Version sieht nicht viel anders aus. Ebenfalls Aktionen: Nimm heraus, sortiere, füge ein. Da klingt schon sehr nach Befehl, Anweisung, imperativ.

Der entscheidende Unterschied ist, dass die rekursive Formulierung des Algorithmus auch als Gleichung gelesen werden kann.

$$\text{Sortieren von a} = \begin{array}{l}\text{Einfügen des ersten Elements in eine} \\ \text{Sortierung von a ohne das erste Element}\end{array}$$

Eine Sortierung von $5, 1, 3, 2$ ist das Gleiche wie das Einfügen von 5 in eine Sortierung von $1, 3, 2$ und weiter so: Eine Sortierung von $1, 3, 2$ ist das Gleiche wie ein Einfügen von 1 in eine Sortierung von $3, 2$. Und … *ist das Gleiche wie* … ist, ja genau, Mathematik: eine Gleichung!

$$
\begin{aligned}
sort([5,1,3,2]) &= insert(sort([1,3,2]),5) \\
&= insert(insert(sort([3,2]),1),5) \\
&= insert(insert(insert(sort([2]),3),1),5) \\
&= insert(insert(insert(insert([],2),3),1),5) \\
&= insert(insert(insert([2],3),1),5) \\
&= insert(insert([2,3],1),5) \\
&= insert([1,2,3],5) \\
&= [1,2,3,5]
\end{aligned}
$$

Das sind auch Gleichungen, keine Aktionen (siehe Abbildung 4.11). Natürlich kann man die Gleichungen auch als Aktionen lesen. Das ist richtig und wird Leuten mit einer ausgeprägten Rekursionsphobie auch gerne so erklärt. Aber:

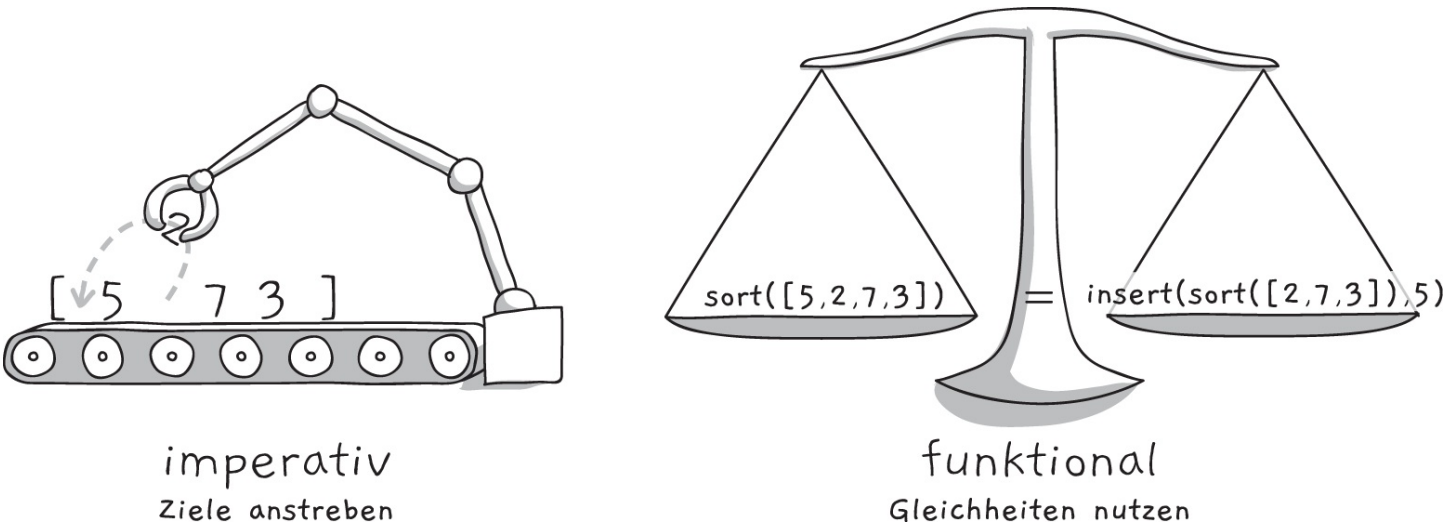


Abbildung 4.11: Imperatives und funktionales Denken



Der Schlüssel zum funktionalen Denken liegt darin, sich vom Denken in Aktionen zu lösen und die Gleichungen mal als Gleichungen zu akzeptieren. Man muss nicht alle Aktionen in jeder Rekursionstiefe nachvollziehen. Es reicht, wenn man die Gleichung als korrekte Aussage akzeptieren kann. Zum Auseinanderdröseln gibt es Computer.

Wenn es mir gelingt, $[1,3,2]$ zu sortieren, dann brauche ich nur noch 5 einzufügen und habe eine Sortierung von $[5,1,3,2]$. Der Kerngedanke ist eine Problemreduktion, also Reduzieren und Herrschen: Das große Sortierproblem wird auf ein etwas kleineres Sortierproblem reduziert. Diese Reduktion ist möglich, weil ich korrekt einfügen kann.

InsertionSort ist darum ein *Reduzieren-und-Herrschen*-Algorithmus oder allgemeiner ein *Teilen-und-Herrschen*-Algorithmus: Das große Problem wird auf ein gleichartiges, aber kleineres reduziert und damit beherrscht. Das sieht man schön an der mathematischen Definition von InsertionSort:

$$sort(a) = \begin{cases} a & \text{falls } a = [\,] \\ insert(sort(a'), x) & \text{falls } a = x :: a' \end{cases}$$

Das Einfügen kann genauso behandelt werden. Etwas Yoga und Kopfstand und wir sehen, dass

$$insert(a, x) = \begin{cases} [x] & \text{falls } a = [] \\ x :: a & \text{falls } a = y :: a' \text{und } x < y \\ y :: insert(a', x) & \text{falls } a = y :: a' \text{und } x >= y \end{cases}$$

Die beiden Doppelpunkte bedeuten hier das Davorsetzen eines Elements vor eine Folge. Beispiel:

$$1 :: [2, 3] = [1, 2, 3]$$



Mit $x :: lst$ bezeichnet man oft die Konstruktion einer neuen Liste aus einem Element x, das vor eine existierende Liste lst gesetzt wird. Es handelt sich um eine Infix-Notation für die Cons-Operation im algebraischen Datentyp der Liste. Also

```
type List = Element :: List | Nil
```

Die geklammerten Listen $[a, b, c]$ stehen für eine entsprechende Liste:

$$[a, b, c] = a :: b :: c :: Nil$$

Arrays haben es nicht gerne funktional

InsertionSort beruht also in der imperativen und der funktionalen Variante auf derselben Idee: Sortieren kann durch systematisches Einfügen in eine kürzere Folge erledigt werden. Das lässt sich aktionistisch in einer Schleife oder mathematisch in rekursiven Gleichungen formulieren.

Definieren wir jetzt die funktionale Variante des Algorithmus etwas präziser:

```
InsertionSortFA(a: array[0..n − 1])    Insertion-Sort, funktionale Variante mit Array
    if n > 0
        r ← InsertionSortFA(a ohne das erste Element)
        return insert(r, a[0])
    else
        return array[ ]                gib ein neues leeres Array zurück
insert(a: array[0..n − 1], x)          Einfügen als Funktion
    if n = 0                           Einfügen in leeres Array
        result ← new array[0]
        result[0] ← x
        return result
    else                               Einfügen nicht leeres Array
        result ← new array[0...n]
        if x < a[0]                    x ist kleiner als das erste Element
            result[0] ← x              x wird erstes Element
            for i ← 0 .. n
                result[i + 1] ← a[i]
        else                           x kommt weiter hinten rein
            a1 ← insert(a[1 .. n − 1], v)    Zwischenergebnis bilden
            result[0] ← a[0]
            for i ← 0 .. n             Zwischenergebnis in Ergebnis kopieren
                result[i + 1] ← a1[i]
    return result
```

In diesem Algorithmus ist das Ergebnis ein komplett neu erzeugtes Array. Da es sich um einen rekursiven Algorithmus handelt, werden dabei sehr viele neue Arrays erzeugt. Ein kürzeres wird nach unten weitergegeben und das von unten entgegengenommene wird kopiert und um ein Element erweitert. Ein Alptraum an Ineffizienz und in vielen Programmiersprachen nur sehr umständlich und fehleranfällig zu codieren. Kein Vergleich zu der einfachen und eleganten mathematischen Funktionsdefinition. Das ist aber auch kein Wunder. Arrays sind die ideale Datenstruktur, wenn es um die Veränderungen ihrer Elemente geht. Sie eignen sich aber nicht, wenn es darum geht, regelmäßig eine neue modifizierte Version einer Datenstruktur zu erzeugen.

Einfach verkettete Listen sind die Leibspeise funktionaler Algorithmen

Strukturen in funktionalen Sprachen sind *persistent*, das heißt, was einmal geschaffen wurde, wird nicht mehr verändert. Im Prinzip muss darum viel kopiert werden. Tatsächlich können Kopien aber oft vermieden werden. Wird beispielsweise eine Liste am Anfang erweitert, dann muss der Rest nicht kopiert werden. Man baut eine »Liste mit zwei Eingängen«, die sich so verhält wie zwei völlig getrennte Listen (siehe Abbildung 4.12).

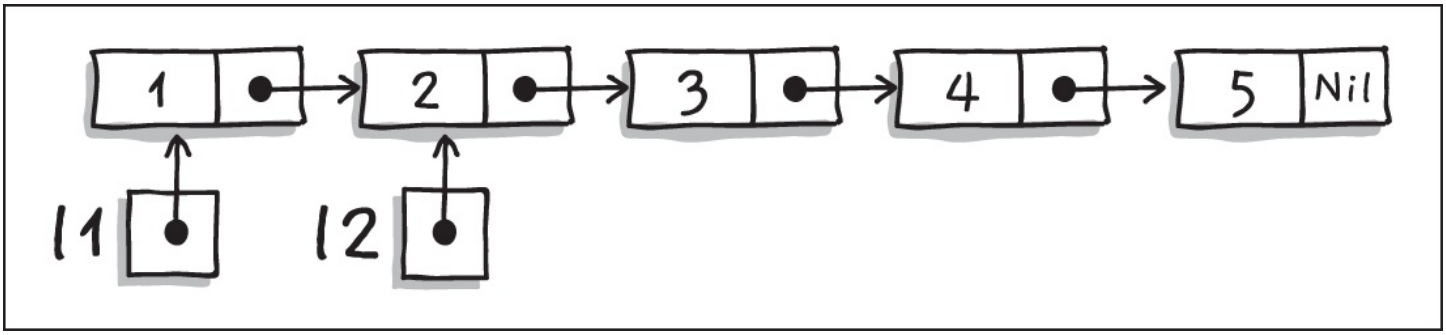


Abbildung 4.12: Zwei Listen als eine Liste mit zwei Eingängen

$l2 \leftarrow \text{List}(2, 3, 4, 5)$
$l1 \leftarrow 1 :: l2$ *vorne anfügen: gut!*

Eine Liste *vorn* erweitern ist darum die bevorzugte Operation bei Listen in funktionalen Sprachen. Ein Element *hinten* anhängen ist weniger erwünscht, denn es muss mit dem Kopieren der gesamten Liste bezahlt werden (siehe Abbildung 4.13).

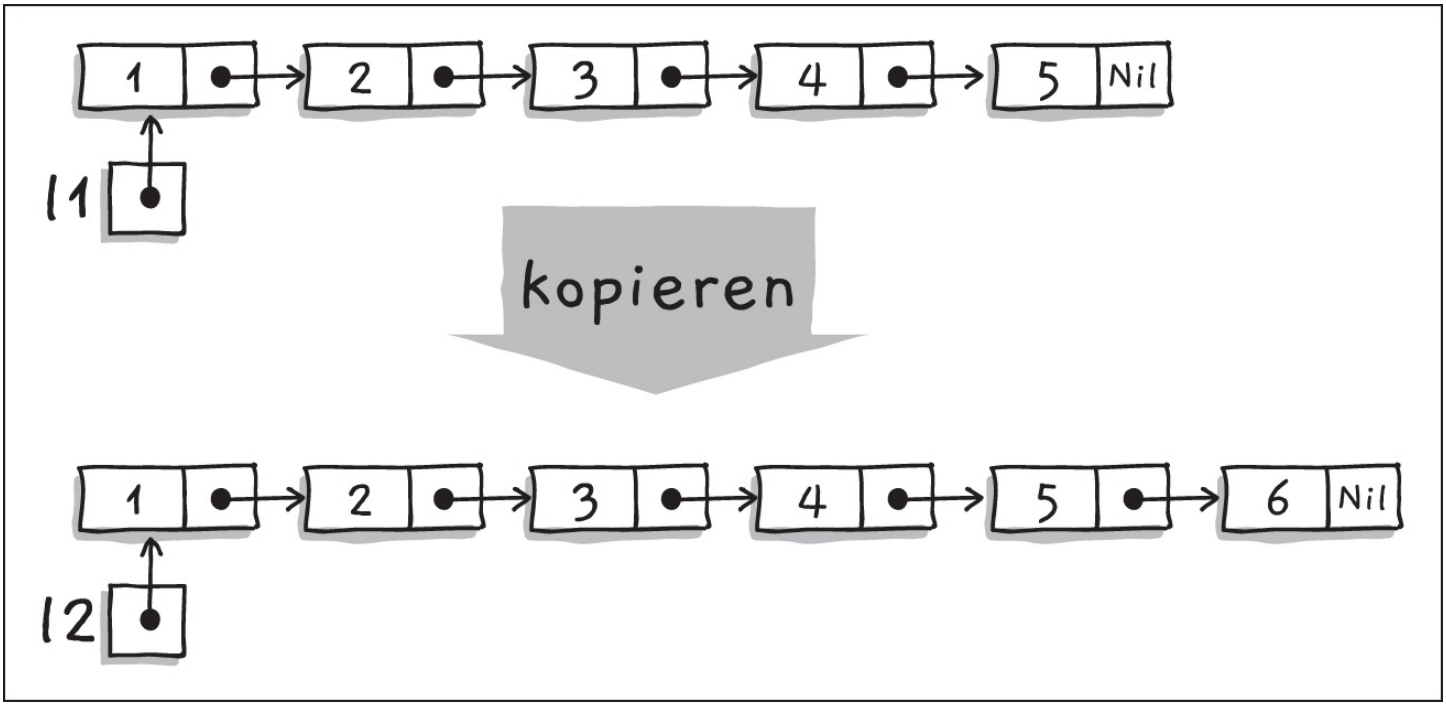


Abbildung 4.13: Funktionales Anfügen erfordert Kopieren

$l1 \leftarrow \text{List}(1, 2, 3, 4, 5)$
$l2 \leftarrow l5 + \text{List}(6)$ *hinten anfügen: nicht gut!*

Mit Listen wird unser funktionaler InsertionSort-Algorithmus deutlich effizienter und gleichzeitig so klar wie die Definition der Funktion. An die Effizienz der imperativen Array-Variante kommt er zwar nicht ganz heran, aber es ist in der gleichen Komplexitätsklasse $\mathcal{O}(n^2)$ und die Ineffizienz der funktionalen Version mit Arrays lässt er weit hinter sich. Dem entsprechenden Algorithmus sieht man jetzt auch wieder die Eleganz der Lösungsidee an, die bei den Arrays ja völlig in Kopieren und Verschieben untergegangen war.

```
InsertionSortFL(lst)                    Insertion-Sort, funktionale Variante mit Listen
    case lst = Nil
        return Nil
    case lst = x :: rest
        return insert(InsertionSortFL(rest), x)
insert(lst, x)                          füge x an der richtigen Stelle in lst ein
    case lst = Nil
        return x :: Nil
    case lst = v :: rest and x < v
        return x :: a
    case lst = v :: rest and x ≥ v
        return v :: insert(rest, x)
```

Insgesamt zeigt schon der Blick auf einen einfachen Sortieralgorithmus, dass funktionale und imperative Algorithmen von unterschiedlichen Lösungskonzepten ausgehen:

- ✔ funktional: Definition von Werten
- ✔ imperativ: Veränderung von Variablen

Die Unterschiede führen zu deutlich unterschiedlichen Algorithmen, selbst dann, wenn sie auf der gleichen Grundidee basieren. Funktionale und imperative Programmierung erfordern zudem

- ✔ angepasste Datenstrukturen (veränderliche Arrays/unveränderliche Listen) und
- ✔ angepasste Kontrollstrukturen: Schleife oder Rekursion, Funktionen mit mehreren Ergebnissen, einfache Analyse und Zerlegung von Datenstrukturen mit einer Case-/Pattern-Match-Notation.

Nicht alles ist ewig und Form ist nicht Inhalt

Sequenzen, die nur an einem Ende (vorne) verändert werden, lassen sich gut und effizient funktional verarbeiten. Eine unveränderliche dynamische Datenstruktur, die verkettete Liste ist bestens geeignet. Wenn aber Veränderungen an beiden Enden möglich sein sollen, dann führt das Unveränderliche unweigerlich zu Ineffizienz, die mit Tricks gemildert werden kann und sollte. Als Beispiel betrachten wir im Folgenden Warteschlangen. Die allgemein wohl eher vertraute imperative Variante haben wir im Abschnitt *Warteschlangen* schon betrachtet.

Warteschlange als funktionale Datenabstraktion

Warteschlangen gibt es natürlich auch in »Funktionalistan«. Da sehen sie allerdings etwas anders aus als in unserer profanen irdisch-imperativen Welt. Datenstrukturen im

Allgemeinen, und damit auch Warteschlangen, sind dort Werte: ewig, unveränderlich. Realisiert wird das dadurch, dass eine persistente Datenstruktur zum Einsatz kommt.

Das hat praktische Konsequenzen: Stellt man sich hinten an, dann verlängert sich die Schlange nicht. Sie bleibt wie sie ist. Es entsteht eine neue, die genauso ist wie die alte, nur dass wir hinten anstehen. Warum? Nun, es könnte ja sein, dass wir uns in einem Paralleluniversum dazu entschließen, nicht beim veganen Tofu, sondern beim Wiener Schnitzel anzustehen. Dann muss es die Tofu-Schlange ohne uns ja auch noch geben.

Eine funktionale Variante der Warteschlange darf also keine veränderliche Liste nutzen. Sie muss auf einer persistenten Datenstruktur beruhen. Es ist naheliegend, auch wieder mit einer Liste zu arbeiten. Diesmal jedoch mit einer, deren Zeiger und Werte nicht modifiziert werden. Das kann als algebraischer Datentyp modelliert werden, ohne schmutzige Wort wie »Zeiger« oder »Adresse« in den Mund zu nehmen (die natürlich trotzdem hinter den Kulissen zum Einsatz kommen):

```
type QueueRep = Empty | Entries(Int, QueueRep)
```

In dieser Version wird die Warteschlange nicht verlängert. Es muss vielmehr eine etwas längere Kopie erzeugt werden.

```
Enqueue(x, queueRep)
    return AddToTail(x, queueRep)

AddToTail(x, queueRep)
    case queueReq = Empty
        return Entries(x, Empty)
    case queueReq = Entries(y, rest)
        return Entries(y, AddToTail(x, rest)
```

Man sieht, wie das neue Element x rekursiv bis ans Ende weitergereicht wird. Das funktionale Enqueue wird ebenfalls anders benutzt als seine imperatives Pendant:

```
queueRep ← Empty
queueRep ← Enqueue(x, queueRep)
```

Hier wird der Variablen $queueRep$ zwar eine neuer Wert zugewiesen, wegen der Persistenz von QueueRep verschwindet damit aber nicht ihr alter Wert, er ist noch da. Mit einer Zuweisung an eine andere Variable

```
queueRepOther ← Enqueue(y, queueRep)
```

wird darum die erste Variable $queuerRep$ ihren Wert behalten: Eine Warteschlange ohne y. Es gibt kein Herumfummeln an meinen Daten durch andere wie bei der imperativen Variante, bei der jede Veränderung sich auf alle auswirkt.

In der funktionalen Variante holt Dequeue ebenfalls nichts aus der Warteschlange heraus, es konstruiert eine neue Variante ohne den ersten Wert:

```
Dequeue(queueRep)
    case queueReq = Entries(y, rest)
        return (x, rest)
```

Hier sehen wir wieder einen Algorithmus, der zwei Werte als Ergebnis liefert. Tupel sind eben ein notwendiges Ausdrucksmittel für funktionale Algorithmen.

Kann man die Datenstruktur und die beiden Zugriffsfunktionen in eine Datenabstraktion stecken? Es ist klar, dass eine *funktionale Datenabstraktion* anders aussehen muss als eine imperative. Imperative Datenabstraktionen sind eine Verallgemeinerung von Variablen. Enqueue und Dequeue statt Lesen und Schreiben wie bei »normalen« Variablen.

Eigentlich ist klar, wie eine funktionale Datenabstraktion aussehen muss. In der funktionalen Sicht der Welt ist alles ein Wert, unveränderlich, ewig, persistent. In der imperativen Welt gibt es nur eine Warteschlange in verschiedenen Zuständen. Die funktionale Welt hat dagegen viele Warteschlangen. Jede einzelne Warteschlange hat ihren unveränderlichen Zustand und dieser Zustand ist ein QueueRep-Wert. Funktionale Datenabstraktionen verallgemeinern/simulieren Werte. Ein Queue-Wert ist darum ein QueueRep-Wert, der um spezielle Operationen erweitert wurde:

```
Queue(queueRep: QueueRep) = record [      Warteschlange mit Repräsentation queueRep
    type QueueRep = Empty | Entries(Int, QueueRep)
    Queue()                               leere Warteschlange erzeugen
        return Queue(Empty)

Enqueue(x)                                WS mit x hinten angestellt
    return Queue(AddToTail(x, queueRep))
AddToTail(x, qR)
    case qR = Empty
        return Entries(x, Empty)
    case qR = Entries(y, rest)
        return Entries(y, AddToTail(x, rest))

    Dequeue()                             Paar: Erster und die WS ohne den Ersten
        case queueReq = Entries(y, rest)
            return(x, Queue(rest))
]
```

Wir nutzen hier die Record-Notation, um Datenabstraktionen zu beschreiben.

```
Queue(queueRep) = record [ ... ]
```

Queue($queueRep$) ist ein aus einer Datenstruktur $queueRep$ erzeugter Record mit diversen Komponenten. Diese Datenabstraktion wird dann im funktionalen Stil verwendet (siehe Abbildung 4.14):

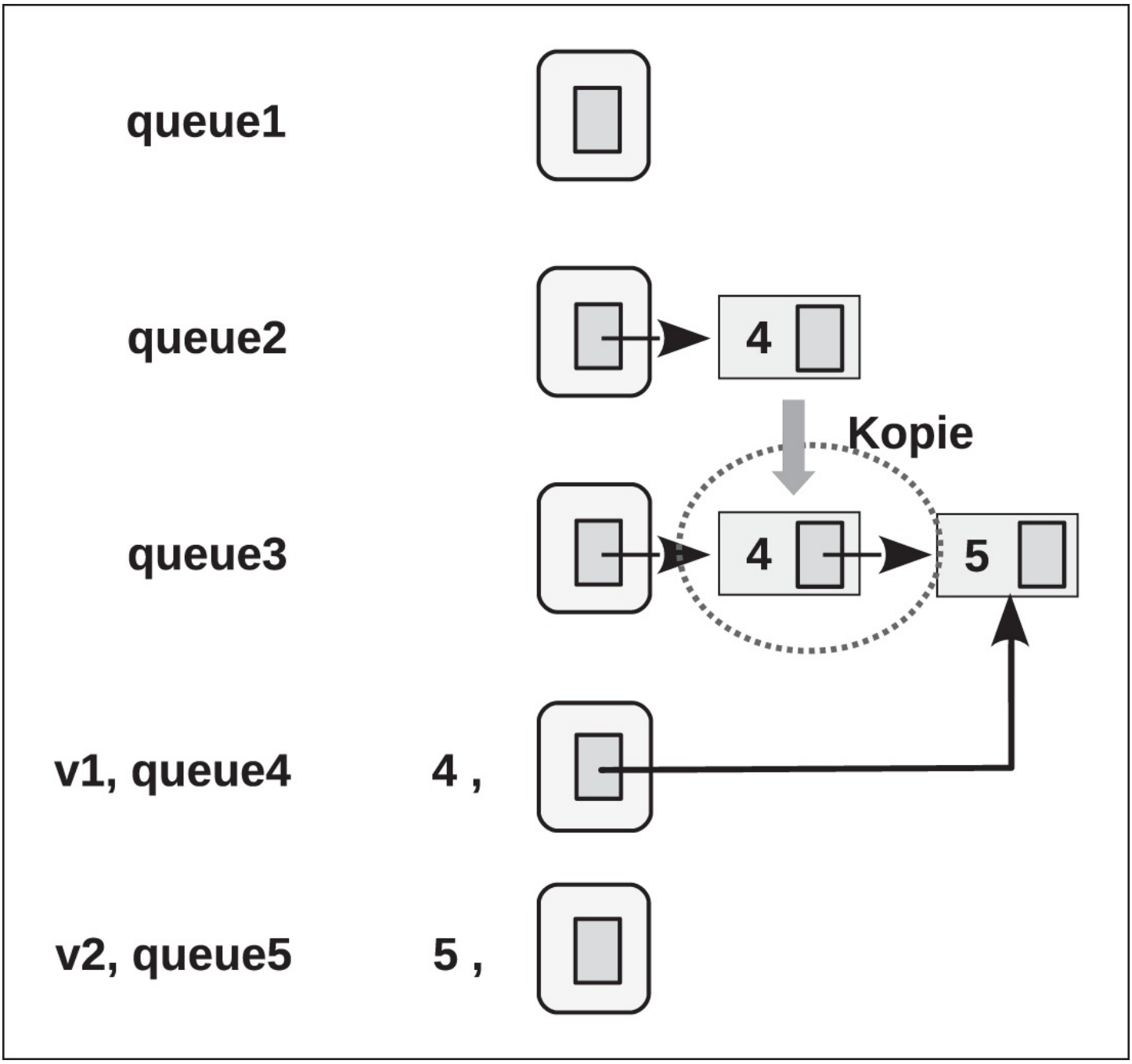


Abbildung 4.14: Warteschlangen funktional

$queue1 \leftarrow$ **Queue**()
$queue2 \leftarrow queue1$.**Enqueue**(4)
$queue3 \leftarrow queue3$.**Enqueue**(5)

$queue3$ zeigt auf eine Kopie der Liste, auf die $queue2$ zeigt. Die zu kopierende Liste besteht hier nur aus einer Zelle mit dem Wert 4. Die Zelle mit 5 einfach hinten an $queue2$ anzuhängen, ist nicht erlaubt, damit würde ja *nebenbei* auch $queue2$ verlängert werden. Solche *Nebenbei*-Aktionen sind nicht erlaubt.

Nach diesen Zuweisungen existieren also drei Warteschlangen-*Werte*: eine leere, eine mit 4 und eine mit 4 und 5. Beim Entnehmen wird die Sache dann noch etwas unhandlicher:

$(v1, queue4) \leftarrow queue3.\textbf{Dequeue}()$
$(v2, queue5) \leftarrow queue4.\textbf{Dequeue}()$

Mit $v1$ = 4 und $v2$ = 5.

$queue4$ kann ohne Kopieren erzeugt werden. Der Zeiger zeigt auf den Rest der Liste, die hier nur aus der Zelle mit 5 besteht. Es gibt keinerlei *Nebenbei*-Auswirkungen auf $queue3$.

Funktionale Datenstrukturen sind Datenstrukturen als Werte. Werte sind unveränderlich. Dazu muss aber nicht immer alles kopiert oder neu erzeugt werden. Es reicht, wenn die Datenstrukturen von außen als unveränderliche reine Werte erscheinen. Das sind dann persistente Datenstrukturen.



Imperative Datenabstraktion verallgemeinert Variablen. Funktionale Datenabstraktion verallgemeinert Werte.

Es ist klar, dass diese funktionale Version der Warteschlange sehr ineffizient ist im Vergleich zur imperativen Version. Genauer gesagt ist Enqueue der Problemfall. In der imperativen Version greifen wir direkt auf das letzte Element zu. Das gibt eine Laufzeitkomplexität in der Klasse $O(1)$. In der funktionalen Variante muss die Repräsentation komplett kopiert werden. Damit sind wir in der Klasse $O(n)$. Kann man da vielleicht noch etwas machen?

Warteschlangen mit Amortisation

Die funktionale Warteschlange ist ein wenig aus dem Gleichgewicht. Einfügen mit $O(1)$, Entnehmen mit $O(n)$. Lässt sich da noch etwas drehen? Vielleicht gibt es ja eine persistente Datenstruktur für funktionale Warteschlangen, mit der eine bessere Laufzeit erreicht werden kann?

Operationen an einem Ende einer listenartigen Struktur sind unproblematisch. Referenzen (Zeiger) auf den alten Zustand bleiben gültig, wenn ich vorn etwas hinzufüge oder wegnehme, wobei das Wegnehmen ja nicht wirklich etwas wegnimmt. Es wird nur eine neue (!) Referenz auf das zweite Element gesetzt. Das geht aber nur an einem Ende. Operationen am anderen Ende müssen mit Kopieraktionen realisiert werden.

Wir können jedoch die Warteschlange durch zwei verkettete Strukturen realisieren. Bei der einen, der Eingangsliste, wird nur (vorne) eingefügt, bei der anderen, der Ausgangsliste, nur (vorne) entnommen. Wenn die Ausgangsliste leer ist, dann werden einfach die Daten von der Eingangs- in die Ausgangsliste herübergeschaufelt (siehe Abbildung 4.15).

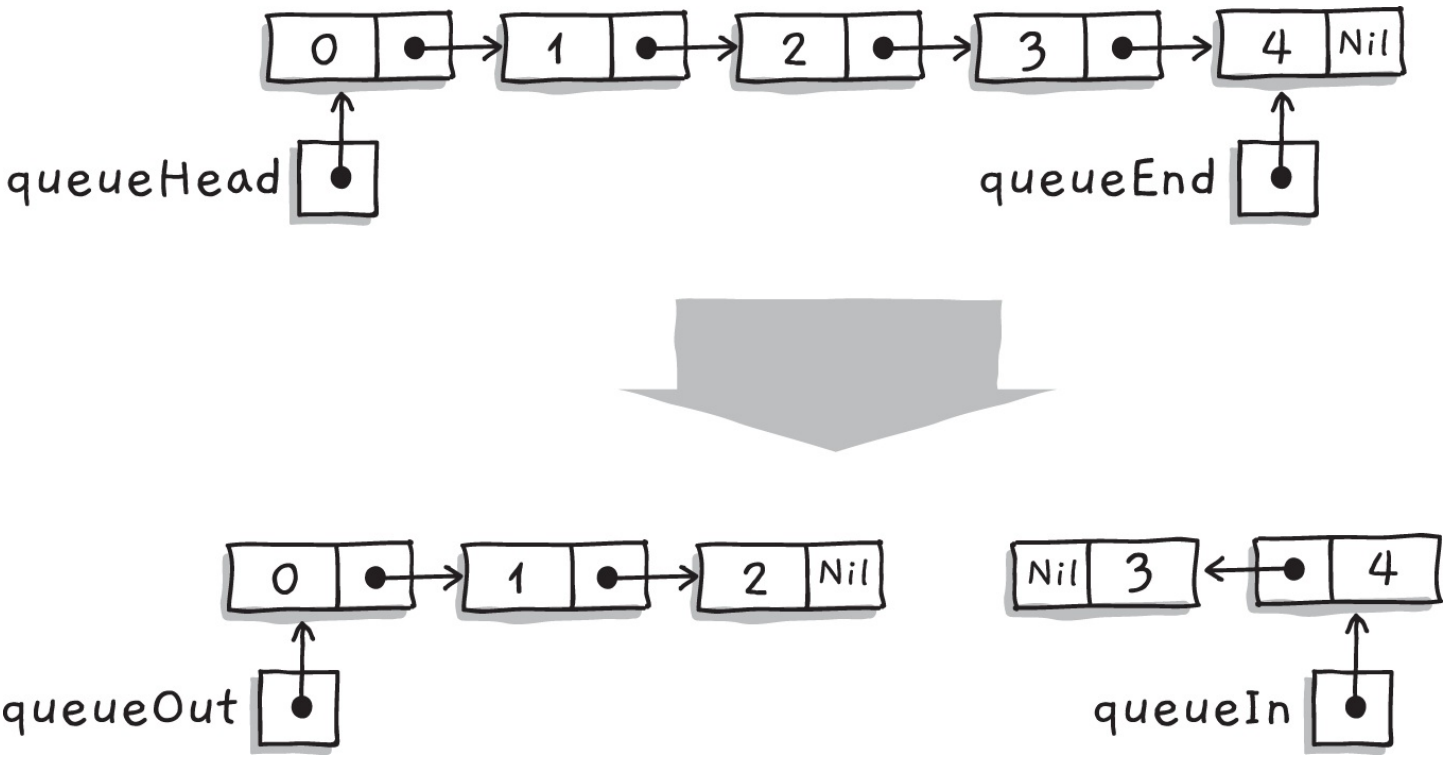


Abbildung 4.15: Statt Zeiger auf Anfang und Ende: Eingangs- und Ausgangsliste

Man beachte, dass in der Eingangsliste die Werte in umgekehrter Reihenfolge liegen: der zuletzt hinzugefügte ist vorn. Einfügen wird damit zum einfachen Davorsetzen und Entnehmen wird zum einfachen Vorne-Wegnehmen. Die beiden Operationen können jetzt offensichtlich mit konstanter Laufzeit ausgeführt werden.

Eine Warteschlange wird jetzt somit durch zwei listenartige Datenstrukturen repräsentiert. Einfügen und Entnehmen modifizieren (vorne, also in einer für die Persistenz wohlgefälligen Art) jeweils eine der beiden. Entnehmen hat natürlich wieder die kleine funktionale Umständlichkeit eines Ergebnis-Tupels.

```
QueueA(queueIn: QueueRep, queueOut: QueueRep) = record [
    type QueueRep = Empty | Entries(Int, QueueRep)

QueueA()                                leere Warteschlange erzeugen
    return QueueA(Empty, Empty)

Enqueue(x)                              Eingänge in die Eingangsliste stecken
    return QueueA(Entries(x, queueIn), queueOut)

    Dequeue()                           Ausgänge aus der Ausgangsliste entnehmen
        case queueOut = Entries(x, rest)    funktioniert nicht, wenn die Liste leer ist
            return (x, QueueA(queueIn, rest))
]
```

Da fehlt natürlich noch etwas. Wenn die Ausgangsliste leer ist, dann können ja noch Werte in der Eingangsliste stecken. Diese werden dann aus der Eingangsliste in die Ausgangsliste übertragen:

```
QueueA(queueIn: QueueRep, queueOut: QueueRep) = record [

    ...                                          der Rest ist unverändert
Dequeue()                                        Ausgänge aus der Ausgangsliste
    case queueOut = Entries(x, rest)
        return (x, QueueA(queueIn, rest))
    case queueOut = Empty                        Ausgangsliste ist leer
        newQueueOut ← CopyReverse(queueIn, Empty)    Eingang => Ausgang
        case newQueueOut = Entries(x, rest)
            return (x, QueueA(Empty, rest))      neue WS mit leerer Eingangsliste

    CopyReverse(queueRep, acc)                   Eingang (in umgekehrter Reihenfolge) in den Ausgang
        case queueRep = Empty
            return acc
        case queueRep = Entries(x, rest)
            return Reverse(rest, Entries(x, acc))
]
```

Wenn die Ausgangsliste leer ist, dann werden mit CopyReverse die Daten von der Eingangs- in die Ausgangsliste geschaufelt. Die Eingangsliste wird dabei in die umgekehrte Reihenfolge der Ausgangsliste gebracht. Das kostet keine besondere Mühe: Die eine Liste wird von vorne abgebaut, die andere von hinten aufgebaut. Die Eingangsliste wird anschließend geleert. Genauer gesagt wird eine neue Warteschlange mit leerer Eingangsliste und befüllter Ausgangsliste erzeugt.

Die Funktion CopyReverse arbeitet mit der Akkumulatortechnik und ist endrekursiv. Sie ist also äquivalent zu einer Schleife, in der das Ergebnis in einer »Akkumulator«-Variablen aufgesammelt wird.

Einfügen und Entnehmen sind immer noch $O(1)$, bis auf die Fälle, in denen die Eingangsliste in umgekehrter Reihenfolge zur Ausgangsliste wird. Das ist eine sporadische $O(n)$-Aktion. Jedes Einfügen und die meisten Entnahmen sind $O(1)$. Gelegentlich müssen wir das mit einer Entnahme mit $O(n)$ bezahlen. Das lohnt (»amortisiert«) sich aber.



In Kapitel 2 wird der Begriff der *amortisierten Analyse* eingeführt. Dabei geht es darum zu analysieren, ob sich die Investition in gelegentlich anfallende, aufwendige Aktionen lohnt, wenn es dafür dann ansonsten schneller geht.

Rückblick: Imperative und funktionale Algorithmen

Arrays sind *statisch*, das heißt von bekannter fester Größe und *veränderlich*. Sie verändern ihre Form nicht, aber ihr Inhalt kann sich ändern.

$a \leftarrow \textbf{array}[1,2,3]$
$a[0] \leftarrow 7$ *Änderungen im Inneren*

Das liegt daran, dass Arrays keine Folge von Werten, sondern eine Folge von Variablen sind. Die Form ist statisch, der Inhalt dynamisch.

Algebraische Datentypen können als Komponenten in eine größere Struktur eingebaut werden. Beispielsweise so:

$a \leftarrow 2 :: 3 :: \mathsf{Nil}$
$b \leftarrow 1 :: a$ *a ist Bestandteil des größeren b*

Eine Liste wird hier zum Rest einer größeren Liste. Es wird dabei nichts verändert, das schon existiert. Das gilt auch, etwas eingeschränkt, bei

$a \leftarrow 2 :: 3 :: \mathsf{Nil}$
$a \leftarrow 1 :: a$ *der alte Wert von a ist Bestandteil seines neuen Wertes*

Die *Variable* a wird zwar verändert, aber a ist nicht Bestandteil einer Datenstruktur. Auch hier wird keine Datenstruktur verändert. Alle Komponenten einer Struktur sind Werte und keine Variablen, vor allem die Zeiger. Die äußere Form ist dynamisch, aber der Inhalt unveränderlich.

Wenn *Datenstrukturen* nicht nur Werte, sondern auch *Variablen* enthalten können, die dazu eventuell auch noch Zeigervariablen sind, dann kann die Struktur nicht nur wachsen, sie kann auch in ihrem Inneren modifiziert werden. Werte können ausgetauscht und Strukturen beliebiger Form erzeugt werden: Zyklische Strukturen, Komponenten mit gemeinsamen Unterkomponenten und so weiter (siehe Abbildung 4.16).

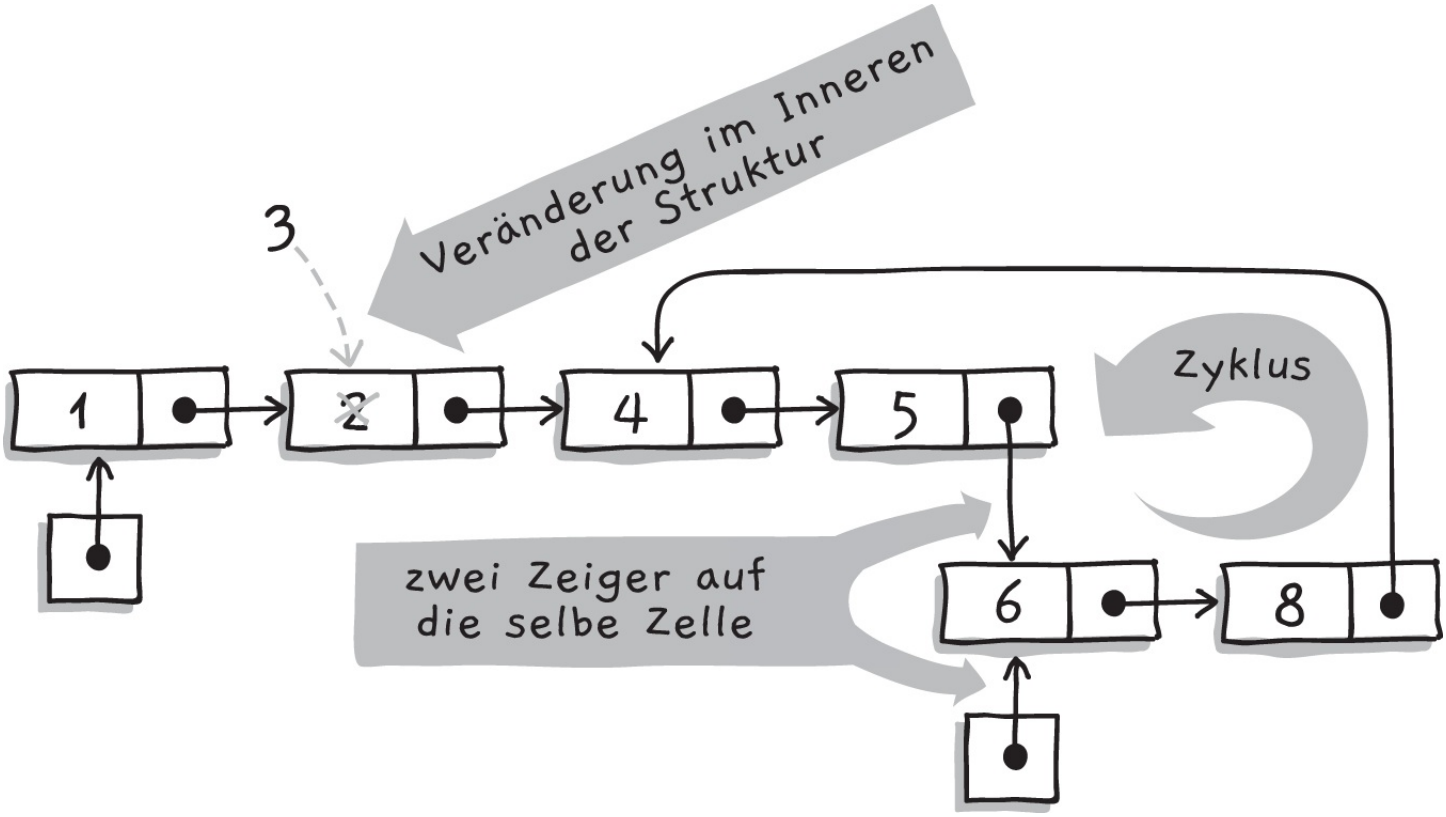


Abbildung 4.16: (Zeiger-)Variablen innerhalb von Datenstrukturen: Alles ist möglich

Die Algorithmen, die mit solchen Strukturen umgehen, dürfen Variablen innerhalb von Strukturen modifizieren, auch solche, die Zeiger enthalten. Und zwar nicht in der langweiligen Art der algebraischen Datentypen, die vorhersehbar ist und vom Unterbewusstsein oder dem Compiler übernommen werden kann. Diese erweiterten Möglichkeiten sind aber mühsam und oft nicht einfach zu beherrschen.

Zum Glück muss man weder als Softwareentwickler noch beim Entwerfen von Algorithmen zwingend auf die mühselige Arbeit mit Zeigern eingehen. Auf der abstrakten Ebene der algebraischen Datentypen kann alles gesagt werden und es gibt keinerlei Entscheidungsspielraum für algorithmische Überlegungen bei der Umsetzung in elementare Speicherstrukturen. Das spricht für die Arbeit mit persistenten Datenstrukturen und funktional formulierte Algorithmen.

Die Eleganz und Einfachheit vieler funktionaler Algorithmen ist beeindruckend. Gelegentlich verleitet sie aber zu sorgloser Naivität, die mit enormen Effizienzverlusten bezahlt werden muss. Ein einfaches Beispiel ist die Verknüpfung von zwei Listen zu einer:

```
append(lst1, lst2)                    verknüpfe zwei Listen zu einer
    case list1 = Nil
        return lst2
    case lst1 = head :: tail
        head :: append(tail, lst2)
```

Eine persistente Liste kann problemlos am Anfang erweitert werden. Zeiger auf den Rest bleiben korrekt, denn in einer »funktionalen« Liste kann man nicht rückwärtsgehen. Wenn aber hinten etwas angehängt wird, dann muss alles davor kopiert werden (siehe Abbildung 4.17).

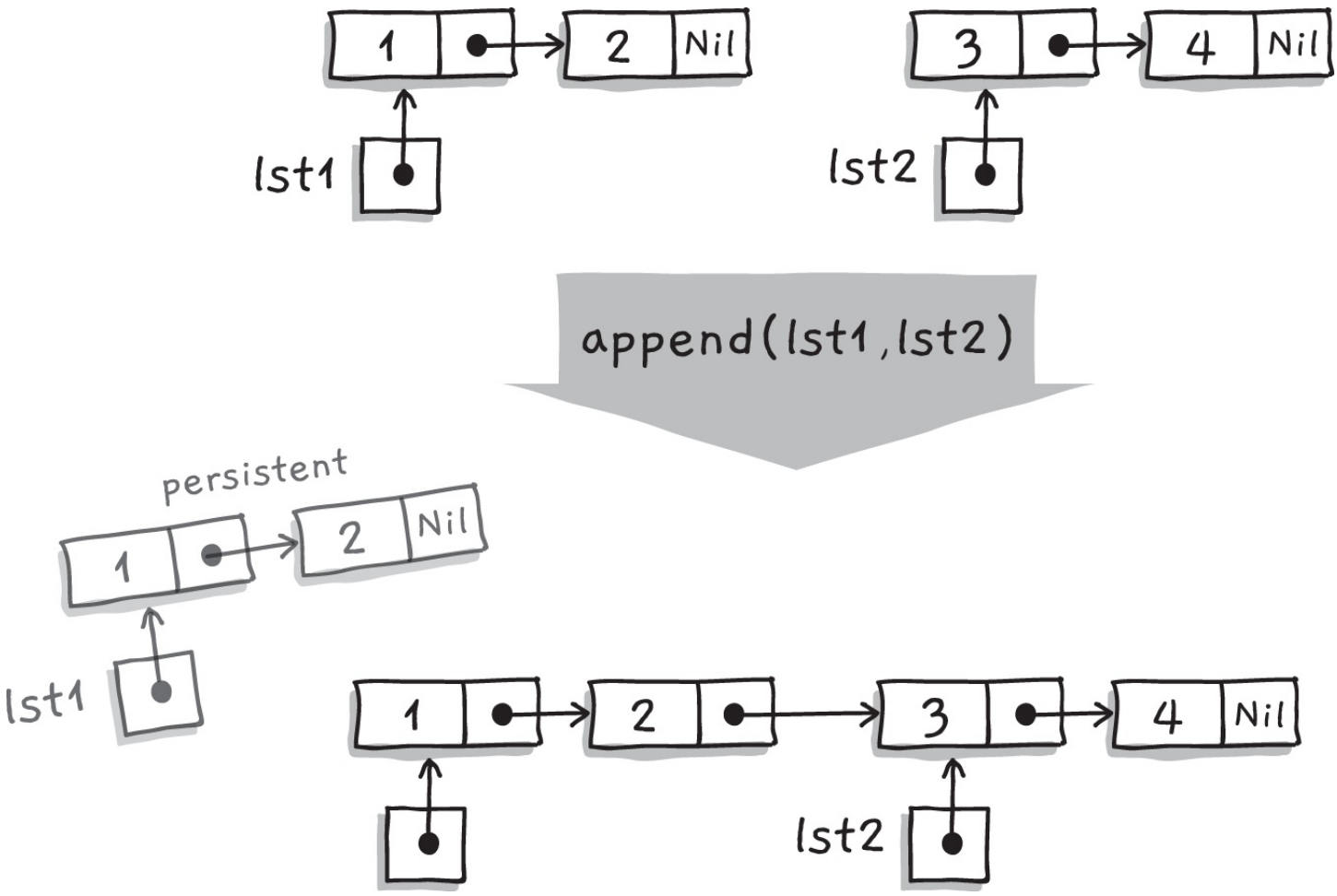


Abbildung 4.17: Append

Kopieren ist notwendig, weil funktionale Algorithmen *Persistenz* erfordern: Einmal erzeugte Datenstrukturen müssen in ihrem Zustand »für immer« erhalten bleiben.

Listenverknüpfungen sind also nach Möglichkeit zu vermeiden. Oder anders herum: wenn zwei Listen regelmäßig zu einer verknüpft werden müssen, dann ist die *Datenstruktur persistente verkettete Liste* vielleicht nicht die richtige Implementierung des *Datenmodells Liste*. Man kann auf die Persistenz verzichten oder eine besser geeignete persistente Datenstruktur verwenden. Genauso wie wir das für die Warteschlange gemacht haben. Eine besser geeignete persistente Datenstruktur für verkettbare Listen wären Bäume, das Thema des nächsten Kapitels.



Bei der Verwendung funktionaler Algorithmen darf man die zur Verfügung stehenden Datenstrukturen nicht zu naiv verwenden. Die Persistenz der Datenstrukturen kann erhebliche Auswirkungen auf die Effizienz der Algorithmen

haben. Durch eine Anpassung und Modifikation der Algorithmen und Datenstrukturen kann das aber oft kompensiert werden. Dazu ist es wichtig, sich der (Zeiger-)Implementierung persistenter Datenstrukturen bewusst zu sein und Datenstruktur und Datenmodell unterscheiden zu können.

Kapitel 5

Bäume: Immer einer über dem anderen

IN DIESEM KAPITEL

- Baumtraversierungen und Iteratoren
- Bäume als flexible Informationsspeicher
- Bäume als Modell hierarchischer Strukturen

Algorithmen sollen Daten verarbeiten. Daten sind Informationen über bestimmte Sachverhalte in der realen Welt. Bevor man über den Algorithmus nachdenken kann, muss man sich mit den Daten beschäftigen. Wenn es viele sind, dann haben sie eine Struktur. Erstmal in unserer Vorstellung. Das ist das Datenmodell. Es gibt viele mögliche Datenmodelle für einen Sachverhalt. Manche mehr, andere weniger geeignet für die auszuführende Verarbeitung. Die Wahl eines geeigneten Datenmodells ist der erste Schritt auf dem Weg zu einer Problemlösung. Als Nächstes werden die Algorithmen entworfen. Dabei stellt sich heraus, welche Operationen auf den Daten auszuführen sind. Passend zu den Operationen kann dann schließlich die konkrete Datenstruktur konzipiert werden, auf der die benötigten Operationen möglichst effizient ausgeführt werden können.



In Kapitel 4 haben wir dieses Vorgehen am Beispiel von Listen besprochen und dabei Listen betrachtet: als »«Kopfkino»« der Anordnung von Daten (Datenmodell), als Funktionalität auf dieser gedachten Anordnung (Datenabstraktion) und als konkrete Organisation der Daten im Speicher.

Mit Listen verbindet man die Vorstellung einer linearen Anordnung, eins nach dem anderen. Bäume sind in der Informatik das Modell hierarchischer Beziehungen: eins über den anderen, oder auch: das eine besteht aus den anderen beziehungsweise das eine enthält die anderen. Das ist das *Datenmodell Baum*. Außerordentlich viele Dinge der Realität sind hierarchisch geordnet. Bäume sind darum ein sehr wichtiges Datenmodell.

Bäume sind auch eine konkrete Organisationsform von Daten im Speicher. Man kann die Daten ineinander verschachteln. Ein Record kann einen anderen Record enthalten und dieser wiederum selbst weitere Komponenten. Das Verschachteln wird irgendwann sehr unhandlich und man speichert die hierarchische Struktur besser in Form verketteter Zellen. Das Äußere, Übergeordnete enthält dann nicht das Innere/Untergeordnete, sondern nur einen Zeiger darauf.

Es gibt viele Anwendungen mit Daten, die wie Listen, Graphen oder Mengen aussehen, im Inneren aber tatsächlich Bäume sind. Umgekehrt gibt es wenige Situationen, in denen sich etwas wie ein Baum verhalten soll, tatsächlich aber etwas völlig anderes ist. Aber auch das gibt es. Arrays zum Beispiel können »«Baum spielen»«. In dieser Rolle nennt man sie *Heap*. Beispiele für Datenabstraktionen auf Basis von Bäumen gibt es auch: Der Zugriff auf grafische Benutzeroberflächen wird oft in Abstraktionen verpackt, deren Schnittstelle die Daten als hierarchische (Baum-)Struktur präsentiert. Prominentestes Beispiel ist das *Document Object Model* (DOM). Bäume sind aber in erster Linie Datenmodell und Datenstruktur, als Datenabstraktion sind sie weniger interessant.

Bäume sind also wichtig als Modell vieler Gegebenheiten und als ganz konkrete Struktur im Speicher. Oft auch beides: Datenmodell und konkrete Struktur, in der das Modell abgespeichert wird. Beginnen wir mit Bäumen als konkrete Datenorganisationsform im Speicher und den allgemeinen, für viele Anwendungen nützlichen Algorithmen, mit denen sie bearbeitet werden können. Der Schwerpunkt liegt dabei auf Bäumen als persistente Datenstruktur.

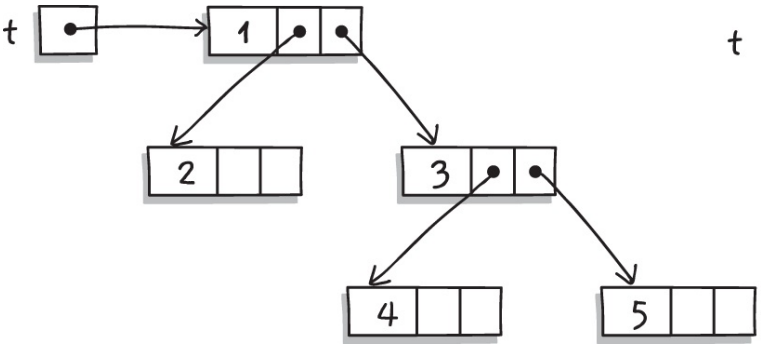
Wo ist die Kokosnuss?

Mit Bäumen kann man allerlei machen. Zum Beispiel kann man zur Zeit der Ernte die reifen Früchte pflücken und einsammeln. Wenn man Glück hat, dann ist man ein Affe oder zumindest so begabt beim Klettern wie ein Affe. Dann braucht man keine Leiter, sondern kann den Baum einfach so auf der Suche nach Früchten durchlaufen. Oder traversieren, um es vornehmer auszudrücken. Beginnen wir das Traversieren mal bei einem außerordentlich mageren Exemplar der Gattung Baum.

Eine der einfachsten Formen eines Baumes ist der Binärbaum. Ein Binärbaum ist entweder leer oder er hat einen Wert und zwei Nachfolger, die ebenfalls Binärbäume sind. Ein Binärbaum über ganzen Zahlen kann man wie folgt als algebraischen Datentyp (alg. DT) definieren:

type BinTree = Node(Int, BinTree, BinTree) | Empty

In Abbildung 5.1 sehen wir einen Binärbaum mit diesem algebraischen Datentyp in einer Zeiger-Implementierung. Im Folgenden reduzieren wir in der Regel die Darstellung auf das Wesentliche, so wie in Abbildung 5.2 als übersichtliche Darstellung des Binärbaums von Abbildung 5.1.




```
t = Node(1,
         Node(2,
              Empty,
              Empty),
         Node(3,
              Node(4,Empty,Empty),
              Node(5,Empty,Empty)))
```

Abbildung 5.1: Ein Binärbaum vom Typ BinTree

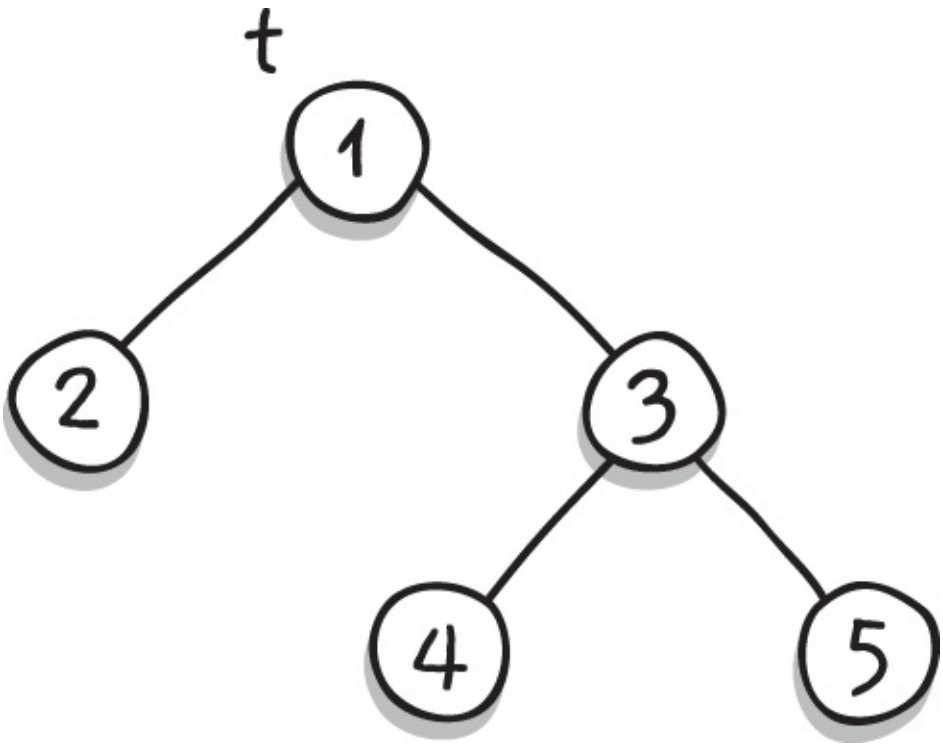


Abbildung 5.2: Ein Binärbaum in vereinfachter Darstellung

Ein algebraischer Datentyp beschreibt immer eine persistente Datenstruktur. Der Inhalt eines solchen Baumes kann nicht modifiziert werden und Bäume, die einmal erschaffen wurden, sind unvergänglich. So und jetzt hinauf (hinab?) in den Baum.

Baumtraversierungen

Ein Binärbaum nach der Definition oben ist entweder leer oder er besteht aus einem Wert und zwei Unterbäumen. Der Wert ist die Kokosnuss. Die sammeln wir auf. Bei einem leeren Baum haben wir Pech, da gibt es nichts zu sammeln. Bei einem nicht-leeren Baum werden natürlich auch noch die beiden Unterbäume durchlaufen und alles eingesammelt, was dort zu finden ist.

Das ist strukturelle Rekursion. Funktion folgt Form. Als Affe in einem Binärbaum hat man dabei die Wahl: Soll erst die Kokosnuss hier aufgelesen werden und dann in die Unterbäume abgeerntet werden oder umgekehrt? Beides hat seine Vor- und Nachteile. Erst aufzusammeln und dann weiterzuklettern sichert die Beute frühzeitig, aber man hat den

Ballast mitzuschleppen. Umgekehrt klettert es sich einfacher, aber vielleicht ist die Beute schon weg, wenn man zurückkommt.

Auf Bäumen können viele Algorithmen in Form einer strukturellen Rekursion definiert werden: Wenn der Baum kein Blatt ist, dann verarbeitet man die Nachfolger mit rekursiven Aufrufen und kombiniert deren Ergebnisse mit den Informationen aus der Wurzel. In den meisten Fällen folgt man dabei dem Prinzip des Kletterns ohne Ballast. Ein Algorithmus zur Berechnung der Summe aller Werte in einem Baum in diesem Stil ist:

```
Sum(binTree)
    case binTree = Empty
        return 0
    case binTree = Node(v, l, r)
        return Sum(l) + Sum(r) + v
```

Oder wir nehmen diesen Algorithmus zur Berechnung der Tiefe eines Baumes, also des längsten Wegs von der Wurzel zu einem Blatt:

```
Depth(binTree)
    case binTree = Empty
        return 0
    case binTree = Node(v, l, r)
        return max(Depth(l), Depth(r))+1
```

In beiden Fällen geht es erst in die Rekursion zu den Unterbäumen. Von dort kommt man mit Zwischenergebnissen zurück, die zusammen mit dem Wert des Knotens genutzt werden, um das Gesamtergebnis zu berechnen.

Wir können auch mit einem einfachen Algorithmus eine Liste zu einen Binärbaum auffalten:

```
ToBTree(lst)
    case lst = []
        return Nil
    case lst = [x_0]
        return Node(x_0, Empty, Empty)
     case lst = [x_0, x_1]
        return Node(x_0, Node(x_1, Empty, Empty), Empty)
    case lst = [x_0, x_1, ...x_n]
        return Node(x_0, ToBTree([x_1..n_{n/2}]), ToBTree([x_{n/2+1}..x_n]))
```

Das wäre dann eine Rekursion über die Liste. Ein Teilen-und-Herrschen-Algorithmus, aber keine strukturelle Rekursion. Bleiben wir aber bei den Bäumen und beobachten die Klettereien, mit denen ein Baum durchlaufen werden kann, etwas genauer.

Traversieren (Durchlaufen) ist eine sehr häufige Aktion auf allen Datenstrukturen, nicht nur auf Bäumen. Beim Durchlaufen eines Baumes hat man, wie bereits gesagt, verschiedene Möglichkeiten: zuerst in die Tiefe oder zuerst in die Breite. Bei einem Tiefendurchlauf (siehe Abbildung 5.3) werden die Unterbäume vor den Nachbarn betrachtet. Das haben wir oben gemacht. Eine Breitentraversierung geht in Ebenen: erst zu den Nachbarn, bevor es zu den Nachkommen geht.

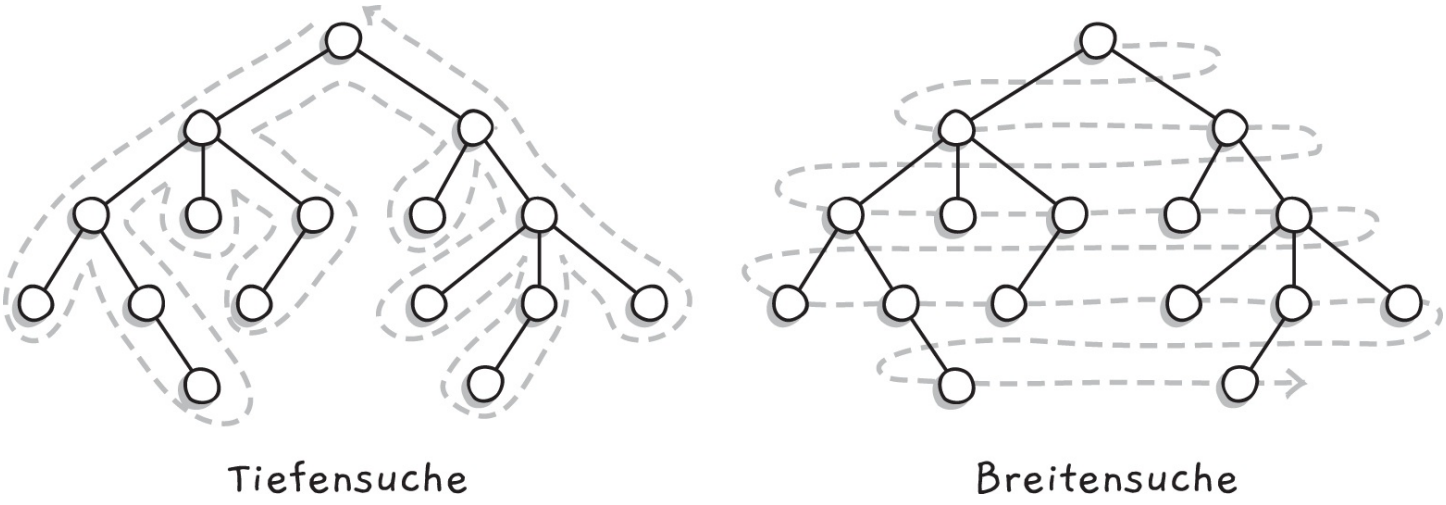


Abbildung 5.3: Traversierungen, globale Strategie



Bei der Verarbeitung einer hierarchischen Struktur (eines Baumes) unterscheidet man folgende *globale Traversierungsstrategien* (siehe Abbildung 5.3):

- ✔ **Tiefentraversierung** (auch *Tiefendurchlauf* oder *Tiefensuche*): Die untergeordneten Strukturen werden vor denen auf gleicher Ebene durchlaufen (durchsucht).
- ✔ **Breitentraversierung** (auch *Breitendurchlauf* oder *Breitensuche*): Die Strukturen auf gleicher Ebene werden vor den Nachfolgern/Unterstrukturen durchlaufen (durchsucht).

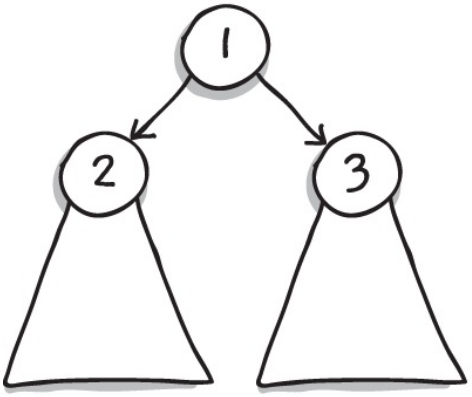
Die üblichere und einfacher zu organisierende Strategie ist der Tiefendurchlauf: Folge allen Ästen, Unterästen und Zweigen, bis es nicht mehr weitergeht. Kehre dann um und folge auf dem Rückweg dem ersten Ast, Unterast oder Zweig, der auftaucht und noch nicht besucht wurde. Dieses Verfahren lässt sich sehr einfach rekursiv organisieren.

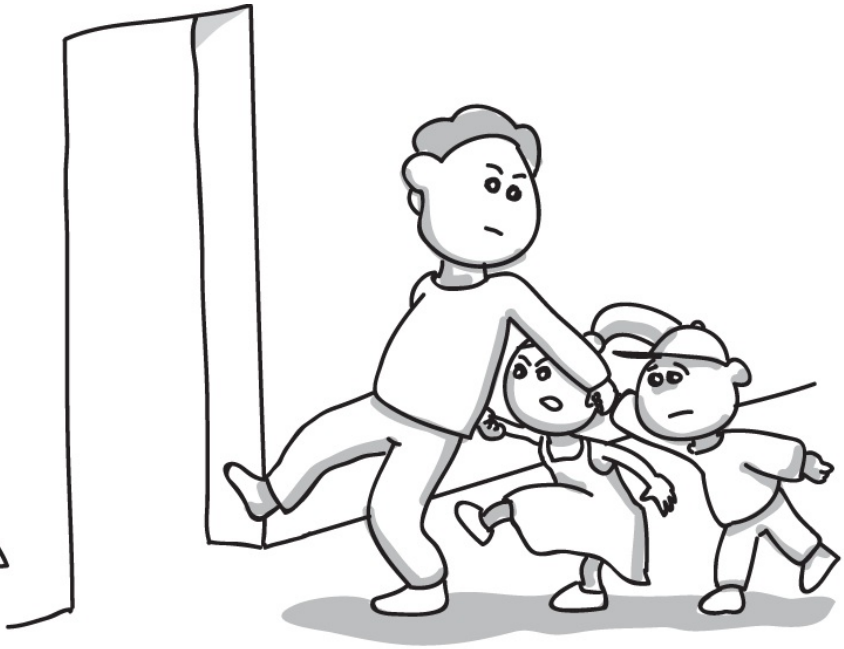
Bei einer Tiefentraversierung geht es also zuerst in die Tiefe. Neben der globalen Strategie des Baumdurchlaufs kann man bei der Tiefensuche auch noch unterschiedliche lokale Strategien anwenden – lokal, weil es darum geht, wie man sich an einem bestimmten Knoten verhält.



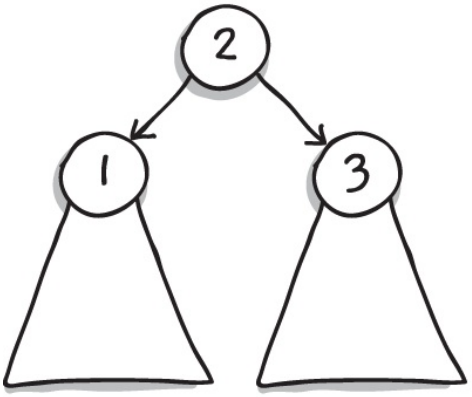
Lokale Strategie der *Tiefentraversierung* (siehe Abbildung 5.4):

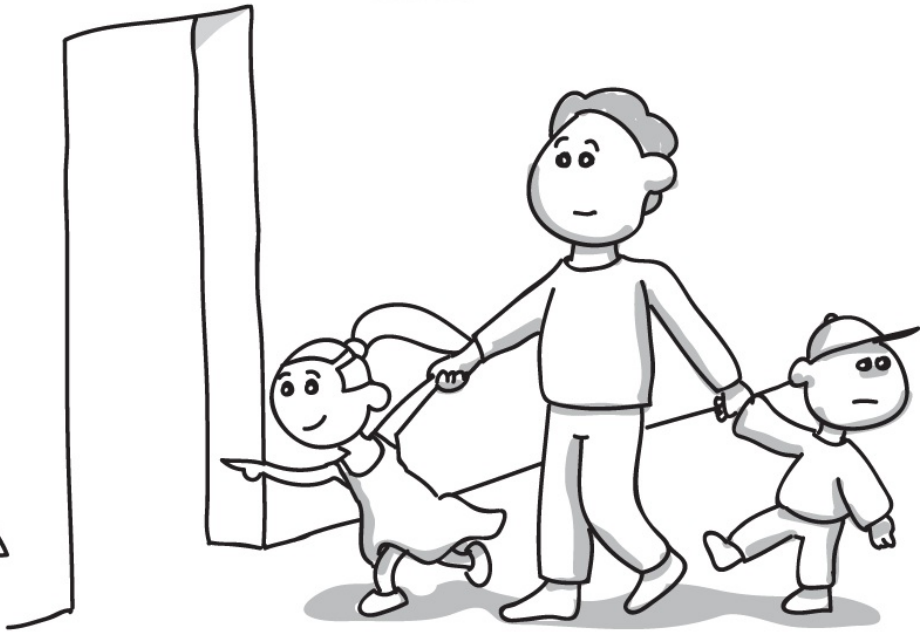
pre-order



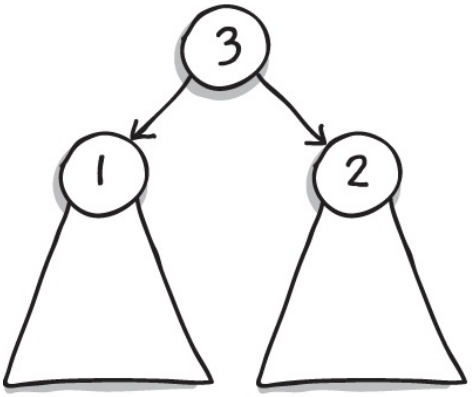


in-order





post-order



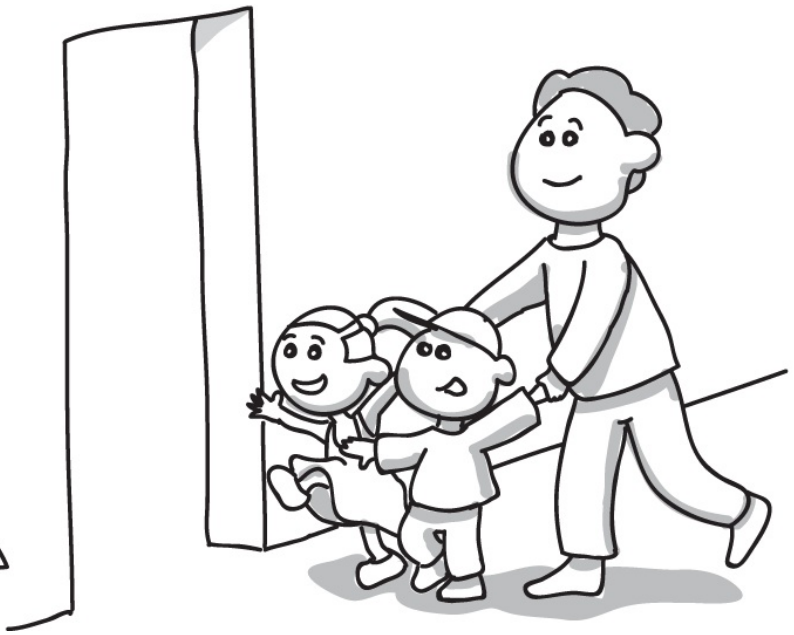


Abbildung 5.4: Traversierungen: lokale Strategie

- ✔ Preorder: erst der Knoten, dann die linke, dann die rechte Unterstruktur,
- ✔ Inorder: erst die linke Unterstruktur, dann der Knoten selbst, dann die rechte Unterstruktur,
- ✔ Postorder: erst die linke, dann die rechte Unterstruktur, dann der Knoten selbst.

Angenommen, wir wollten die Werte in einem Baum aufsammeln und in einer Liste speichern. Das geht am einfachsten mit einer Tiefentraversierung. Beginnen wir mit der *Preorder*-Variante:

```
Preorder(binTree)
    case binTree = Empty
        return Nil
    case binTree = Node(v, l, r)
        return v :: append(Preorder(l), Preorder(r))
```

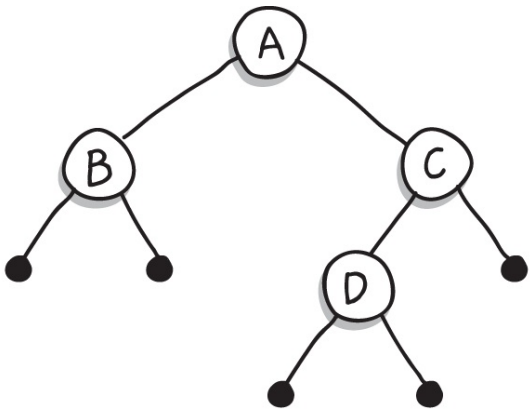
Die Operation append ist dabei die Verknüpfung von zwei Listen zu einer.

Bei der Konstruktion des Gesamtergebnisses wird zuerst der Knotenwert selbst und dann das Ergebnis der Traversierung der Unterstrukturen verwendet. v ist der Knotenwert. Mit ihm als Anfang wird eine Liste erzeugt. Dann folgen die Werte der Unterbäume.

Für den Baum

$tree$ = Node(A , Node(B , Empty, Empty), Node(C , Node(D , Empty, Empty), Empty))

erzeugt Preorder($tree$) die Liste $[A, B, C, D]$ (siehe Abbildung 5.5).




```
Preorder(tree)    = [A,B,C,D]
Inorder(tree)     = [B,A,D,C]
Postorder(tree)   = [B,D,C,A]
```

```
tree=Node(A,Node(B,Empty,Empty),Node(C,Node(D,Empty,Empty),Empty))
```

Abbildung 5.5: Tiefentraversierungen eines Binärbaums

In der *Inorder*-Variante wird die Arbeit bei den Unterkomponenten mit der Arbeit mit dem

Knotenwert gemischt:

```
Inorder(binTree)
    case binTree = Empty
        return Nil
    case binTree = Node(v, l, r)
        return append(Inorder(l), (v :: Inorder(r))
```

Zuerst wird der linke Unterbaum l besucht, dann der Knotenwert v und schließlich der rechte Unterbaum r. Bei einer *Postorder*-Traversierung kommt der Knotenwert nach den beiden Unterbäumen als letzter dran:

```
Postorder(binTree)
    case binTree = Empty
        return Nil
    case binTree = Node(v, l, r)
        return append(Postorder(l), append(Postorder(r), v :: Nil))
```

Alle diese Traversierungsfunktionen sind Beispiele für eine strukturelle Rekursion in funktionalen Algorithmen. Natürlich kann auch imperativer Code mit struktureller Rekursion arbeiten. Bei den Traversierungen trifft dieser vielleicht noch eher unsere Intuition als die funktionale Variante: Gehe überall hin und mache dort etwas mit den Daten. Eine imperative Variante der Inorder-Traversierung wäre beispielsweise

```
Inorder(binTree)
    case binTree = Empty
        return
    case binTree = Node(v, l, r)
        Inorder(l)
        Process(v)
        Inorder(r)
```

Dabei ist Process irgendeine Aktion, die auf den Werten ausgeführt wird. Wenn Process(v) die Ausgabe von v ist, dann wird in dieser Variante die Projektion des übergebenen Baumes ausgegeben (siehe Abbildung 5.6).

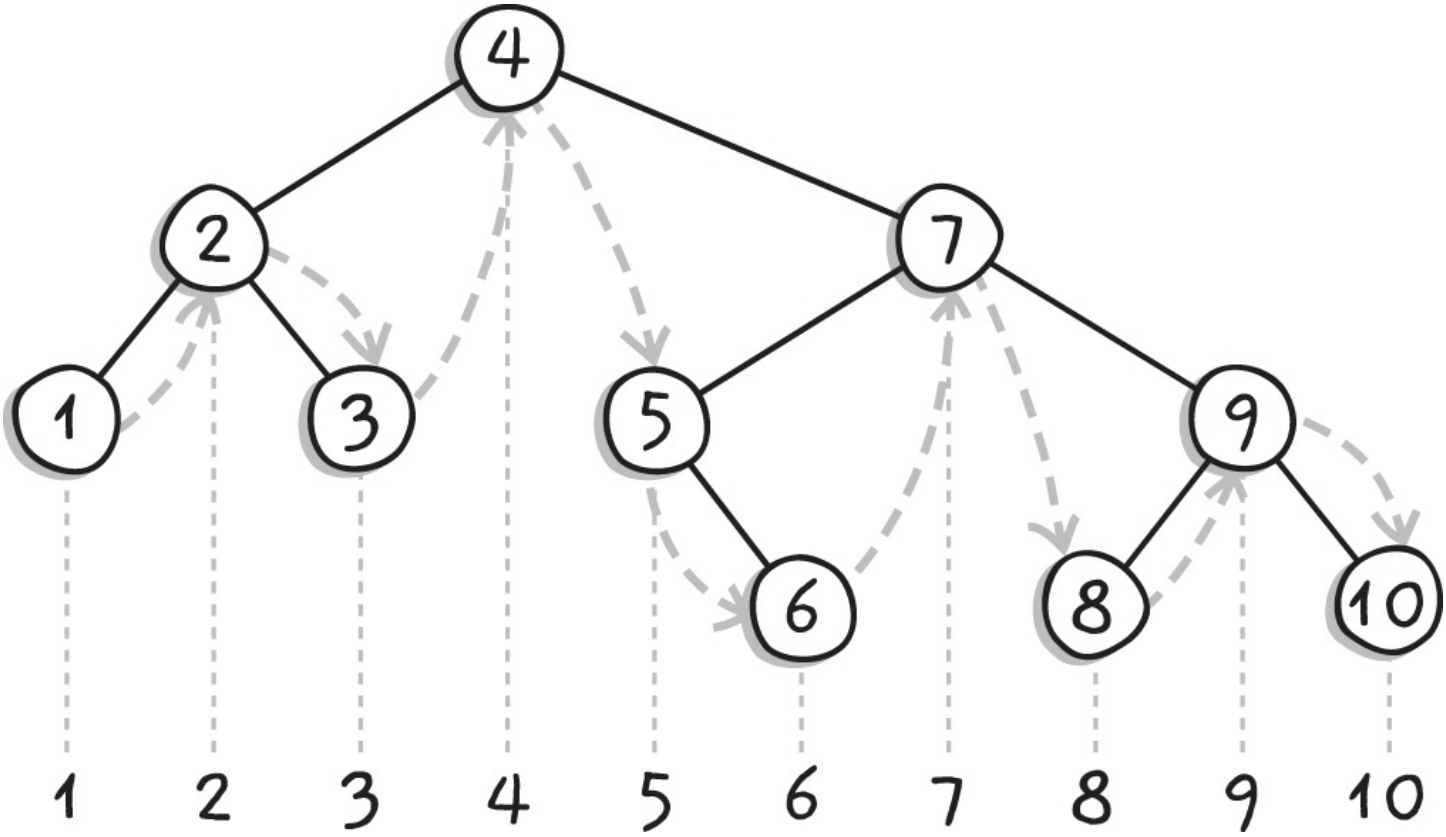


Abbildung 5.6: Inorder-Traversierungen erzeugen eine Projektion des Baumes.

Wird ein Baum durchlaufen, dann kann man sich vorstellen, dass der Fokus zu jedem Zeitpunkt auf einem der Knoten liegt. Einige Knoten wurden vor dem aktuellen besucht, andere werden später besucht werden. So zieht sich ein Faden der Aktivität durch den gesamten Baum. Dieser Faden entspricht genau der Liste, die unsere Traversierungsbeispiele oben (siehe Abbildung 5.5) erzeugt haben, oder der Reihenfolge, in der Process in der imperativen Variante aufgerufen wird.

Um das Beispiel etwas interessanter zu gestalten, betrachten wir einen Baum, bei dem auch die Blätter Werte tragen können (siehe Abbildung 5.6). Das kann man sich als »«normalen»« Binärbaum vorstellen, bei dem einfach alle leeren Blätter weggelassen wurden.

Der Fokus wandert dabei zunächst immer weiter nach links unten – so lange, bis kein linkes Kind mehr anzutreffen ist. Der Knoten selbst wird bearbeitet und dann ist der rechte Unterbaum dran. Anschließend geht es zurück zur Wurzel. Dort kommt dann der Knotenwert selbst und dann sein rechter Unterbaum dran.

Mit Stapeln in die Tiefe tauchen

Die Rekursion macht das alles recht einfach und elegant. Ein Glück, das wie sie haben. Was wäre, wenn nicht? Nicht auszudenken, daran wollen wir gar nicht denken. Oder vielleicht doch? Probieren wir es mal.

Iterierend Traversieren

Betrachten wir das Beispiel in Abbildung 5.7. Zuerst werden die Knoten 6, 4, 2 und 1 auf dem Weg nach links unten passiert. Wir können sie nicht vergessen, sie kommen später dran. Das merken wir uns in einer Liste der aufgeschobenen Arbeiten. Wir kommen auf jeden Knoten in dieser Liste zurück, sobald sein linker Unterbaum abgearbeitet ist.

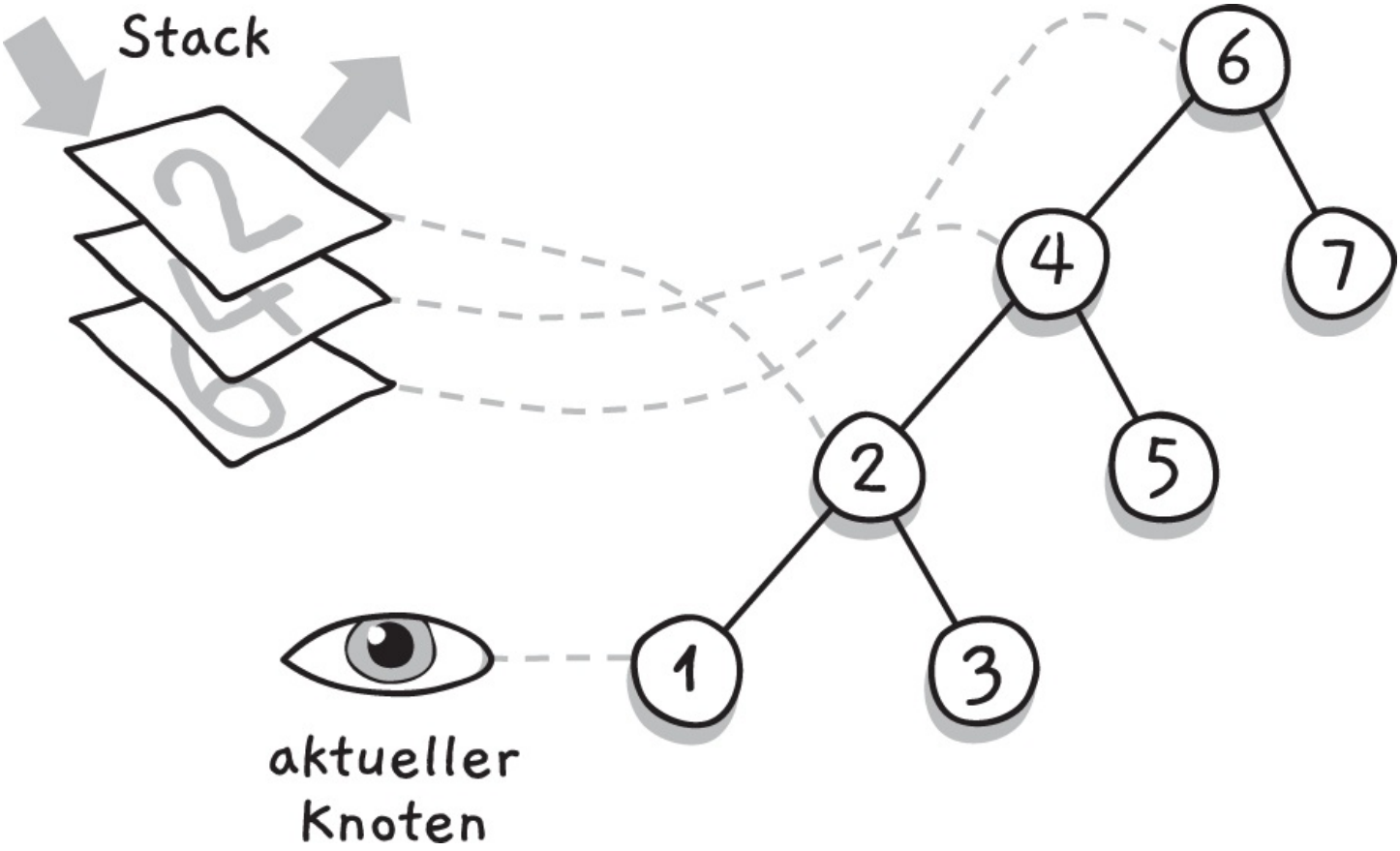


Abbildung 5.7: Inorder-Traversierung mit Stapel aufgeschobener Arbeit

Bei 1 wird festgestellt, dass es keinen linken Nachfolger gibt. Die 1 ist die erste Zahl, die verarbeitet oder ausgeliefert werden kann. Die 1 hat auch keinen rechten Nachfolger. Der Zweig ist fertig abgehandelt. Aus der Liste der aufgeschobenen Arbeiten entnehmen wir den zuletzt hinzugefügten Knoten, den mit der 2. Dessen linker Nachfolger ist erledigt. Jetzt (Inorder-Traversierung) ist der Wert 2 dran. Anschließend geht es nach rechts unten zum Knoten mit der 3. Er hat keinen Nachfolger. Die 3 wird ausgeliefert. Der Unterbaum mit Wurzel 2 ist jetzt komplett erledigt. Aus der Liste der aufgeschobenen Arbeiten nehmen wir den Knoten mit der 4. Dessen linker Unterbaum ist erledigt, die 4 wird ausgeliefert und es geht weiter mit dem rechten Unterbaum von 4. Das ist der Knoten mit der 5. Die 5 wird ausgeliefert und damit ist auch der rechte Unterbaum der 4 erledigt. Aus der Liste wird der Knoten mit der 6 geholt. Dessen linker Unterbaum ist abgearbeitet. Die 6 wird ausgeliefert. Dann schließlich ist der rechte Unterbaum dran. Der besteht nur aus dem Knoten mit der 7.

Eine zentrale Rolle spielt hier die Liste der aufgeschobenen Arbeiten. Formulieren wir in einem ersten Schritt den Algorithmus so um, dass er explizit mit dieser Liste der

aufgeschobenen Arbeiten hantiert. Wir nennen sie kurz Stapel (*Stack*), weil sich dort die aufgeschobenen Arbeiten stapeln.

```
stack ← Nil                                  aufgeschobene Arbeit

Inorder1(tree)
    InorderStack(tree)
InorderStack(tree)
    case tree = Empty                        Baum ist leer
        ProcessStack()                       erledige aufgeschobene Arbeit
    case tree = Node(v, l, r)
        stack ← tree :: stack                aktuellen Knoten (Position im Baum) merken
        InorderStack(l)                      weiter nach links unten
ProcessStack()
    case stack = Nil                         nichts zu tun: fertig
    case stack = Node(v, l, r) :: tail           Restarbeit: Wert, rechter Unterbaum
        stack ← tail
        Process(v)                           verarbeite Knotenwert
        InorderStack(r)                      verarbeite rechten Unterbaum
```

Das ist, dank Rekursion, immer noch recht übersichtlich. Auf dem Weg nach links unten werden die passierten Knoten im Stapel gespeichert.

Unten angekommen werden die Restarbeiten im Stapel erledigt: Für jeden Knoten im Stapel werden der Wert und der rechte Unterbaum verarbeitet (siehe Abbildung 5.7).

Wenn wir etwas genauer auf InorderStack schauen, dann sehen wir, dass es sich um eine *endrekursive* Funktion handelt. Der rekursive Aufruf ist die letzte Aktion. InorderStack ist also »«eigentlich»« eine Schleife. Schreiben wir es auch als Schleife hin:

```
InorderStack(tree)
    while tree ≠ Empty
        stack ← tree :: stack                aktuellen Knoten merken
        case tree = Node(v, l, r)
     tree ← l                                weiter nach links unten
 ProcessStack()                              erledige aufgeschobene Arbeit
```

ProcessStack ist nicht rekursiv. Seine Aktionen können direkt in InorderStack eingesetzt werden:

```
stack ← Nil                          aufgeschobene Arbeit

Inorder2(tree)
    InorderStack(tree)

InorderStack(tree)
    while tree ≠ Empty or stack ≠ Nil      es gibt noch Arbeit
        while tree ≠ Empty                 links runter so weit wie möglich
            stack ← tree :: stack          aktuellen Knoten merken
            case tree = Node(v, l, r)
                tree ← l                   weiter nach links unten
                                           links unten angekommen, erledige aufgeschobene Arbeit:
        case stack = Nil                   nichts zu tun: fertig
        case stack = Node(v, l, r) :: tail Restarbeit: Wert, rechter Unterbaum
            stack ← tail
            Process(v)                     verarbeite Knotenwert
            tree ← r                       weiter mit rechtem Unterbaum
```

Mit der äußeren Schleife wird geschaut, ob überhaupt noch etwas zu tun ist. Wenn wir ganz unten im Baum sind, gibt es ja noch die aufgeschobenen Arbeiten im Stapel. Wenn es etwas zu tun gibt, wird zuerst geschaut, ob wir im Baum weiter nach links unten laufen können. Wenn ja, dann wird das gemacht.

Sicht wir ganz unten, dann wird der Stapel abgearbeitet. Dabei kommen wieder neue Unterbäume in Bearbeitung, bei denen wieder ganz nach links unten gelaufen wird.

Ein Baumiterator

So, schön. Und wozu das alles? Rekursion ist schön. Wozu sollte sie eliminiert werden? Nun, leider muss das gelegentlich sein. Angenommen, der Baum soll mit einem Iterator durchlaufen werden. Natürlich könnte man zuerst die Liste erzeugen und dann über diese iterieren. Allerdings würde dabei mit der Liste eine Kopie der Daten erzeugt werden. Das ist nicht gerade effizient. Eine andere, relativ einfache Methode wäre die sogenannte *Fädelung*. Dabei wird der Baum (rekursiv) durchlaufen und die Knoten werden mit zusätzlichen Zeigern verknüpft, die zusammen einen Faden bilden, an Hand dessen sich ein Iterator durch den Baum hangeln kann.

Es geht aber auch anders. Mit Inorder2 haben wir einen iterativen/imperativen Algorithmus, mit dem der Baum durchlaufen werden kann. Jeder Aufruf von Process verarbeitet einen Wert. Das ist eine *interne Traversierung*. Der Traversierungsalgorithmus hat die Kontrolle über das Geschehen. Man übergibt ihm mit Process die auszuführende Aktion und wartet dann ab, was passiert (siehe Abbildung 5.8).

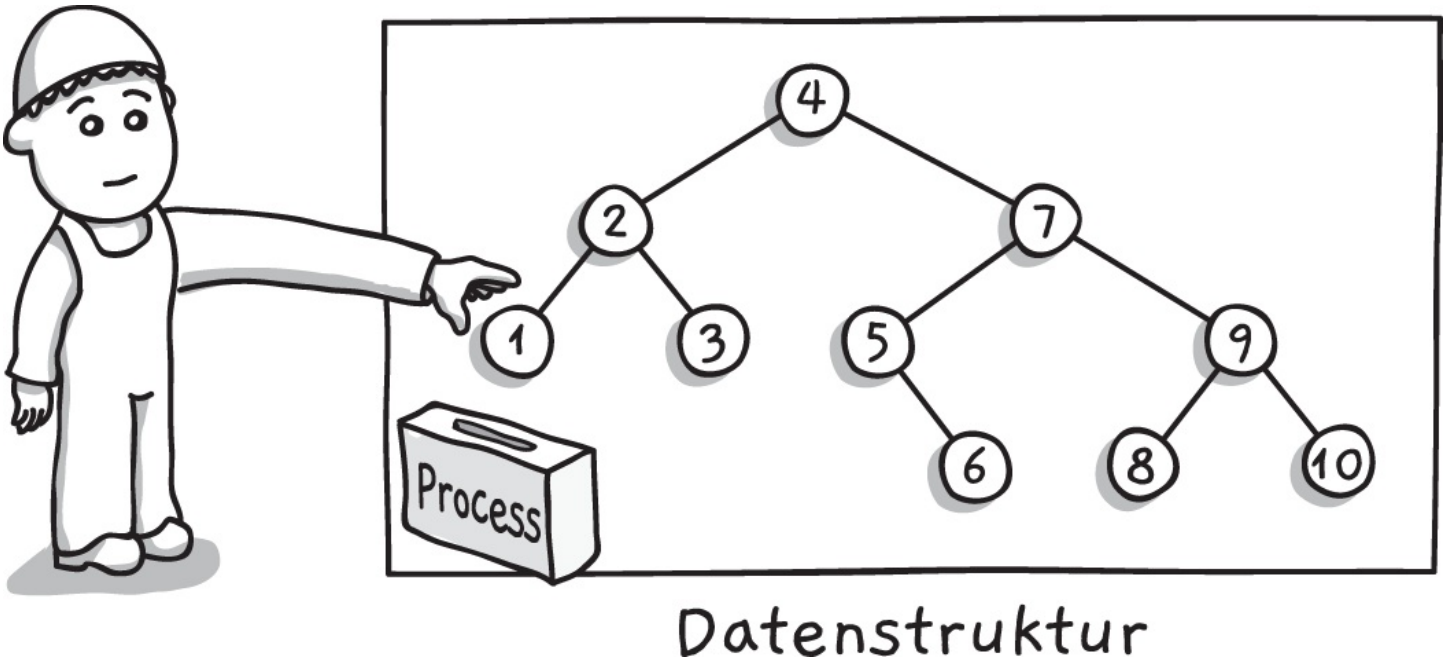


Abbildung 5.8: Interne Traversierung

Ein Iterator realisiert das Konzept der *externen Traversierung*. Die Kontrolle über den Verarbeitungsprozess bleibt bei dem Benutzer der Struktur. Der Iterator liefert einen Wert nach dem andern, der dann außerhalb der Kontrolle der Struktur verarbeitet wird (siehe Abbildung 5.9).

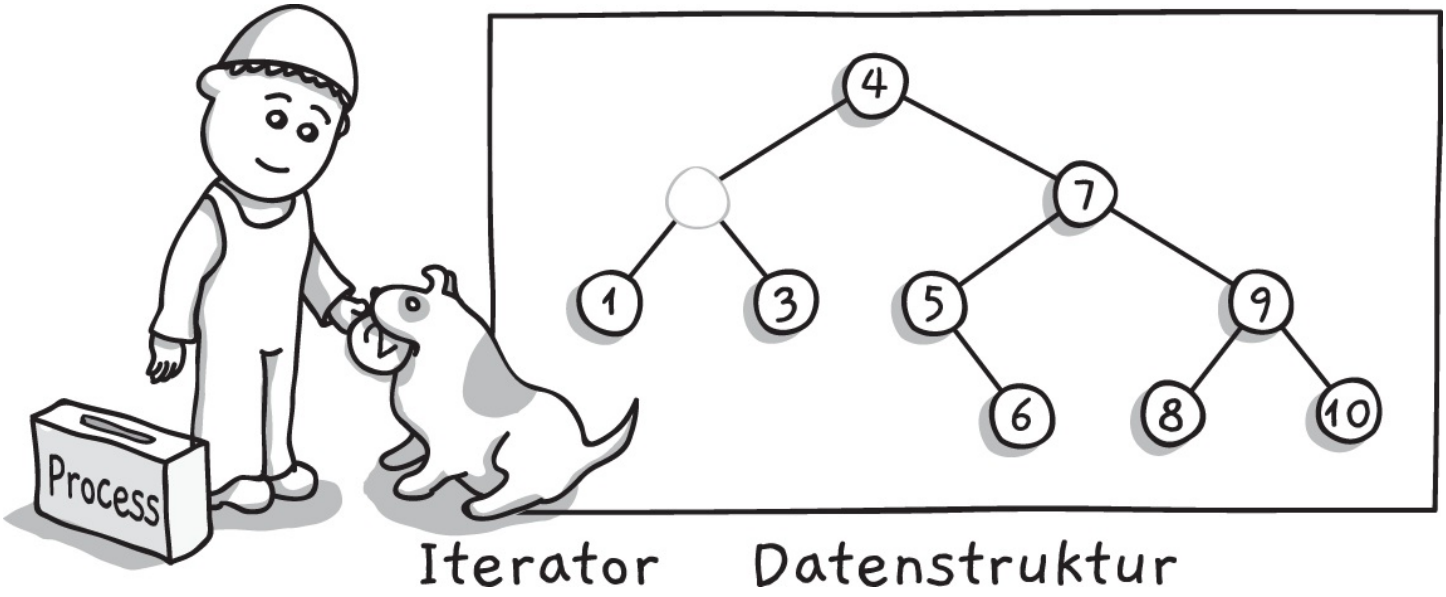


Abbildung 5.9: Externe Traversierung mit einem Iterator

Ein Iterator kann nicht mit Rekursion arbeiten. Die Funktionalität eines Iterators steckt in next. Diese Funktion muss abhängig davon, wo sie beim vorherigen Aufruf aktiv war, den nächsten Wert holen und sich für den nächsten Aufruf merken, wo sie war. Mit hasNext wird geprüft, ob überhaupt noch Elemente zur Verfügung stehen.

Betrachten wir InorderStack, die rekursionsfreie Inorder-Traversierung, dann erkennen wir, dass die notwendigen Informationen für eine externe Traversierung in den Variablen

Stack und *tree* stecken. Wenn *tree* = Empty und *stack* = Nil, dann sind wir am Ende und hasNext muss *false* liefern.

```
Iterator(tree: BinTree) = record [
    stack ← Nil

    hasNext()
        return not (tree = Empty and stack = Nil)

    next()
        ???
]
```

Das war einfach. Das Nächste ist aber auch nicht so schwer. Soll next einen Wert liefern, dann muss die Schleife in InorderStack so lange abgewickelt werden, bis wir auf einen Aufruf von Process stoßen. Statt Process auszuführen, liefert next dann den Wert, den Process verarbeitet hätte. Die Schleife, die zu wiederholten Aufrufen von next werden soll, hat die folgende Struktur:

```
InorderStack(tree)
    while tree ≠ Empty or stack ≠ Nil
        while tree ≠ Empty
            weiter links runter

        Links unten auf Empty angekommen
        Node(v, l, r) vom Stack holen
        Process(v)
        tree ← r
```

Der Nutzer des Iterators übernimmt die Rolle der äußeren Schleife. Ihm steht dazu hasNext zur Verfügung. Die Bedingung der äußeren Schleife wird von hasNext geprüft und die äußere Schleife fällt weg. Wir können also davon ausgehen, dass sie beim Aufruf von next gültig ist. Die innere Schleife muss ausgeführt werden. Statt mit Process den Wert zu verarbeiten, wird er einfach zurückgegeben. Damit haben wir die Grundstruktur von next:

```
next()
    while tree ≠ Empty
        weiter links runter
    Links unten auf Empty angekommen
    v, r vom Stack holen
    result ← v
    tree ← r
    return result
```

Das konkretisiert ergibt den kompletten Baumiterator:

```
Iterator(tree: BinTree) = record [
    t ← tree
    stack ← Nil
next()
    while t ≠ Empty                    links runter
        stack ← tree :: stack
        case t = Node(v, l, r)
            t ← l
    case stack = Nil                   kein unbesuchter Knoten vorhanden
        return Fehlermeldung: alles abgelaufen
    case stack = Node(v, l, r) :: tail
        nextValue ← v                  nächsten Wert vom Stack holen
        stack ← tail                   Stack kürzen
        t ← r                          später geht's rechts weiter
        return nextValue
    hasNext()
        return t = not(tree = Empty and stack = Nil)
]
```



Rekursion kann mit einem Stapel abgewickelt werden. Iteratoren für eine Tiefentraversierung sind externe nicht-rekursive Varianten der strukturell-rekursiven Traversierungsalgorithmen.

Mit Warteschlangen in die Breite gehen

Bei einer Breitentraversierung werden die Geschwister eines Knotens vor dessen Kindern besucht. Es ist klar, dass dies nicht mit einer einfachen strukturellen Rekursion möglich ist. Die Verarbeitung des Baumes entspricht nicht seiner natürlichen Struktur. Er wird ebenenweise traversiert, er ist aber in Unterbäumen organisiert und nicht in Ebenen (siehe Abbildung 5.10).

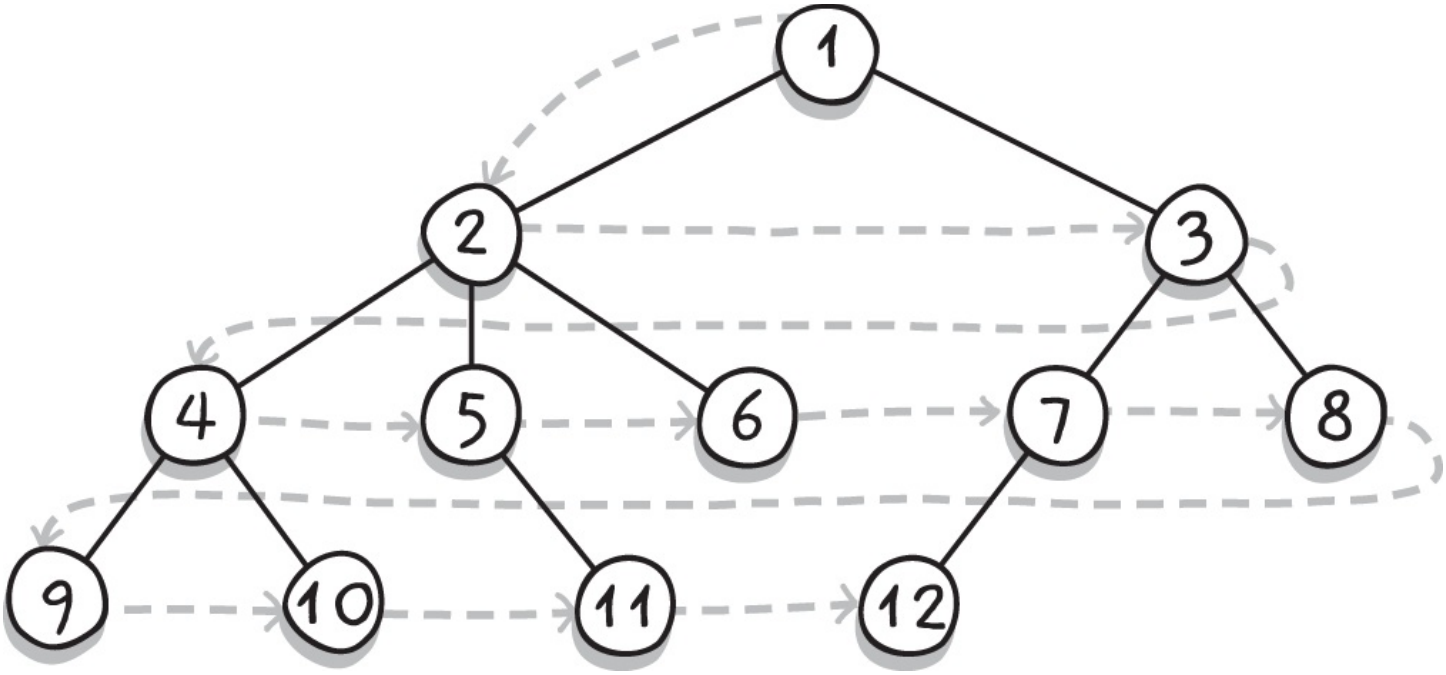


Abbildung 5.10: Breitentraversierung: Traversierung gegen die Natur des Baumes

Bei der Breitentraversierung müssen die Ebenen berücksichtigt werden. Alle direkten Nachfolger eines Knotens liegen auf der gleichen Ebene. Wir durchwandern also alle Nachfolger. Im Beispiel von Abbildung 5.10 sind 4, 5 und 6 Nachfolger von 2. Sie werden somit nacheinander durchlaufen. Dann kommen 7 und 8 als die Nachfolger von 3 dran. Wie kommen wir an die 7, nachdem wir bei der 6 waren? Ganz einfach, mit dem gleichen Prinzip wie bei der Elimination der Rekursion bei Tiefentraversierung: Wenn wir an einem Knoten vorbeikommen, der nicht fertig behandelt werden kann, dann merken wir ihn uns. Knoten 7 und 8 haben wir uns gemerkt. als wir bei der 3 waren.

Bei der Tiefentraversierung sind wir auf dem Weg nach unten bei Knoten vorbei gekommen, für die noch Arbeiten zu erledigen sind. Hier kommen wir auf dem Weg in der Ebene an Knoten vorbei, für die noch etwas zu tun ist. Dann gibt es noch einen kleinen Unterschied: Nach der 3 kommt die 4 dran. Das ist der *erste* Knoten mit unerledigten Nachfolgern, den wir beim Gang von links nach rechts passiert haben. Bei der Tiefentraversierung war es dagegen der nächste, der *letzte* Knoten mit unerledigten Nachfolgern, der beim Gang von oben nach unten passiert wurde und noch unerledigte Nachfolger hat.

Insgesamt ist die Breitentraversierung eine Variante der iterativen Tiefentraversierung, bei der

- ✔ unerledigte Arbeit in einer Warteschlange statt einem Stapel gespeichert wird,
- ✔ es, bevor unerledigte Arbeit in Angriff genommen wird, zu allen direkten Nachbarn geht statt zu allen linken Kindern und Kindeskindern.

Der Algorithmus kann damit direkt als angepasste Modifikation der iterativen

Tiefentraversierung formuliert werden:

```
queue ← Queue()

BreadthFirst(tree)
    TraverseQueue(tree)

TraverseQueue(tree)
    while tree ≠ Empty
        case tree = Node(v, l, r)
            if l ≠ Empty                    linker Nachfolger in die WS
                queue.enqueue(l)
            if r ≠ Empty                    rechter Nachfolger in die WS
                queue.enqueue(r)
            Process(v)                      Knoten selbst verarbeiten
        if queue.empty() = false            Unerledigtes erledigen
            tree ← queue.dequeue()
        else                                fertig
            tree ← Empty
```

Der Algorithmus ist bemerkenswert einfach. In Abbildung 5.11 sind wir beispielsweise gerade bei Knoten 7. Vorher wurden die Knoten 1, 2, 3, 4, 5 und 6 passiert. Deren Nachfolger sind teilweise abgearbeitet, teilweise stecken sie noch in der Warteschlange. 9, 10, 11 wurden beim Gang durch 4, 5 und 6 in die Warteschlange gesteckt und sind noch drin. Ebenso ist Knoten 8, der zweite Nachfolger von 3, noch nicht verarbeitet. Er ist der älteste in der Warteschlange, darum ganz vorn und kommt so direkt nach der 7 dran.

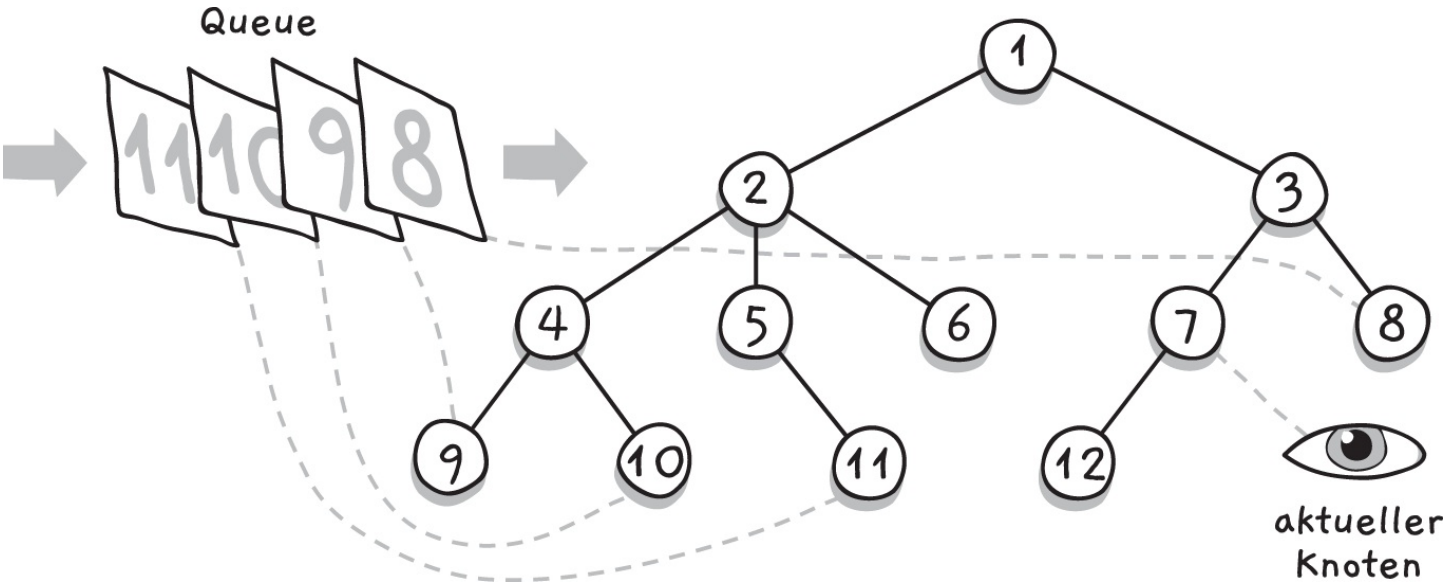


Abbildung 5.11: Breitentraversierung mit Warteschlange

Die Traversierung ist iterativ. Ein ausgezeichneter Startpunkt, um daraus einen Iterator zu

machen. Als erstes beobachten wir, dass Process an das Ende von TraverseQueue verschoben werden kann:

```
TraverseQueue(tree)                    traversiere tree, verarbeite alle Werte mit Process
    while tree ≠ Empty
        case tree = Node(v, l, r)
            if l ≠ Empty
                queue.enqueue(l)
            if r ≠ Empty
                queue.enqueue(r)
            if not queue.empty()
                tree ← queue.dequeue()
            else
                tree ← Empty
            Process(v)                 interne Traversierung: Verarbeitung von v hier
```

Aus dieser Formulierung des Algorithmus kann wieder ein Iterator abgeleitet werden. Die Schleifenkontrolle wird in negierter Form zu hasNext. Der Schleifenkörper wird zu next. Damit haben wir dann den Iterator für eine Breitentraversierung des Baumes:

```
IteratorBF(tree: BinTree) = record [
    queue ← Queue()
    t ← tree
next()                                 liefere den nächsten Wert im Baum
    case t = Node(v, l, r)
        if l ≠ Empty
            queue.enqueue(l)
        if r ≠ Empty
            queue.enqueue(r)

        if not queue.empty()
            t ← queue.dequeue()
        else
            t ← Empty
        return v                       externe Traversierung: Verarbeitung von v außerhalb
    hasNext()                          gibt es noch unausgelieferte Werte im Baum?
        return t ≠ Empty
]
```

Gelegentlich kann ein Algorithmus auf einer Datenstruktur nicht deren natürlicher Struktur folgen und so auch nicht als strukturelle Rekursion formuliert werden. Mit der Breitentraversierung haben wir ein Beispiel dafür. In einem solchen Fall kann der Versuch

einer funktionalen Lösung, auf den wir uns hier gar nicht erst eingelassen haben, zu komplexen Algorithmen führen – bei einer erstaunlich einfachen iterativen Lösung.

Die Kleinen nach links, die Großen nach rechts

Bäume können eine geeignete Datenstruktur sein, auch wenn es eigentlich um etwas völlig anderes geht. Die Datenstruktur *Baum* ist das Arbeitspferd unter den Datenstrukturen. Nützlich in vielen Situationen. Beispielsweise, wenn Ordnung zu schaffen ist.

Angenommen, die WG von Schneewittchen ist über die Jahre enorm angewachsen und unsere Aufgabe ist es, die vielen Wichte zu beaufsichtigen. Wenn Schneewittchen den Wunsch äußert, mit einem Zwerg der Größe 1,12 m das Becherchen zu teilen, haben wir diesen schnell herbeizuschaffen. Schneewittchen stellt sich vor, dass die Zwerge dazu schön nach Größe geordnet in ihren Bettchen liegen. Aber da ständig neue hinzukommen und andere die Lust verlieren und ihr Glück woanders suchen, muss ständig neu sortiert werden. Wirklich Lust, sich mit den Gesellen mühsam auseinanderzusetzen, hat Schneewittchen aber nicht. Das müssen die irgendwie selbst hinbekommen.

Eines schönen Tages, als mal kein Zwerg mehr da war, hat sie sich ein neues System ausgedacht. Wenn nur ein Zwerg da ist, dann gibt es nicht viel zu sortieren. Kommt noch einer hinzu, dann vergleicht er sich mit dem ersten. Ist er kleiner, dann kommt sein Bettchen links hinter das Bettchen des ersten, ist er größer, dann kommt es rechts hinter das erste Bettchen. Ist der neue so groß wie der erste, dann muss er wieder gehen. Schneewittchen mag keine Zwerge mit der gleichen Größe.

Die Zwerge sind nicht die hellsten, aber diese Art der Sortierung bekommen sie hin. Wenn ein neuer kommt, geht er zum ersten, vergleicht sich mit ihm und wird weggeschickt (gleiche Größe) oder zum nächsten rechts oder links dahinter geschickt (größer oder kleiner) oder er darf sein Bettchen aufschlagen (der Platz ist frei; siehe Abbildung 5.12).

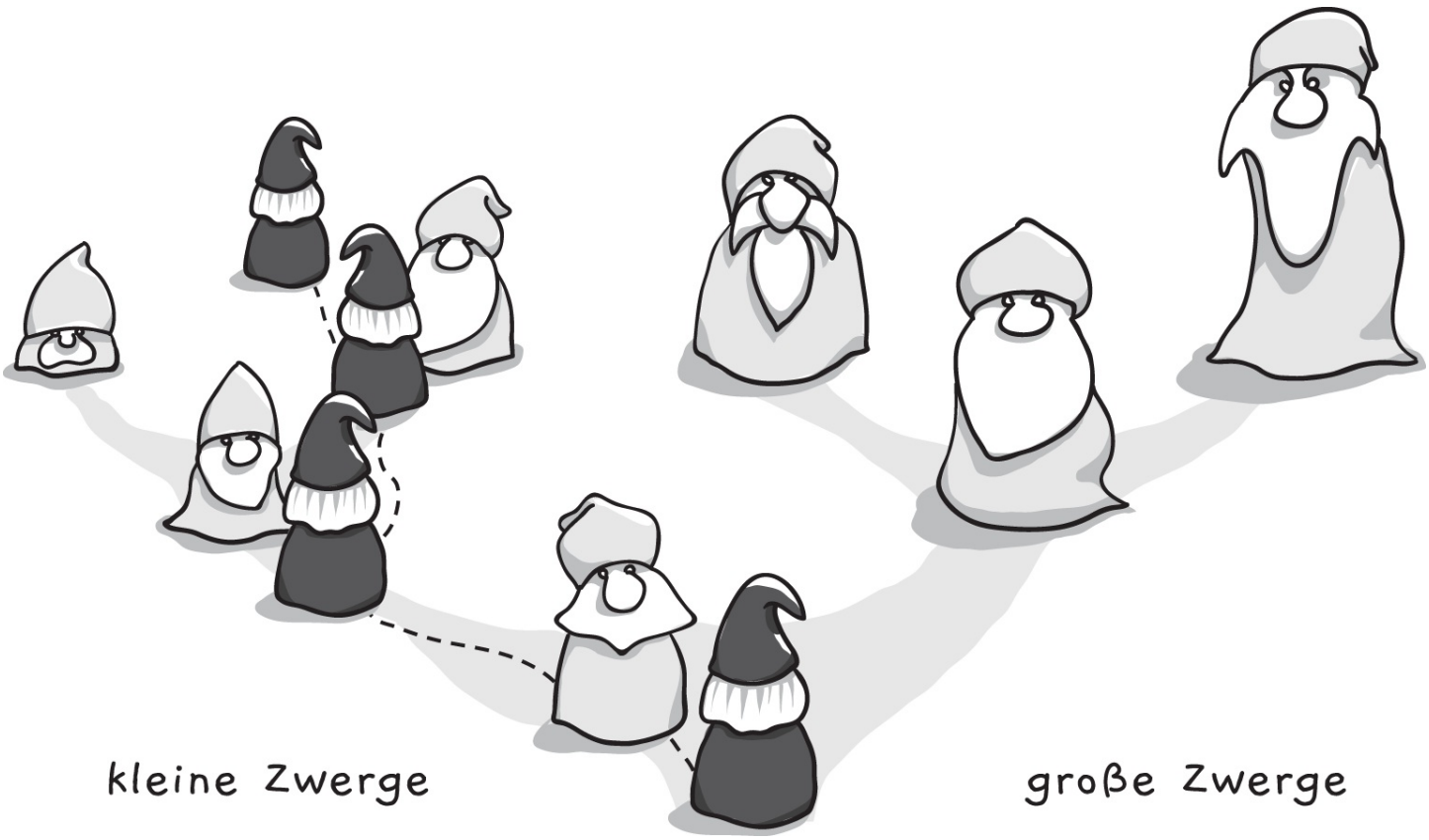


Abbildung 5.12: Zwerge anordnen: Die Großen nach rechts, die kleinen nach links

Wenn Schneewittchen dann einen Zwerg mit einer bestimmten Größe sucht, dann läuft sie einfach so lange von einem zum anderen und vergleicht, bis sie den Passenden gefunden hat, oder weiß, dass ihre Sammlung nicht das Gewünschte enthält. Prima, Einfügen und Suchen mit $O(\log n)$ und die machen es auch noch selbst. Was will man mehr? Ok, was passiert, wenn einer keine Lust mehr hat oder bei Schneewittchen in Ungnade gefallen ist und raus muss? Naja, das überlegen wir uns später, wenn es so weit ist.

Binäre Suchbäume

Ein *Suchbaum* ist eine Datenstruktur zur Speicherung von Informationen, die ein Ordnungskriterium erfüllen. Die Ordnung muss total sein. Das heißt, zwei beliebige Informationen sind vergleichbar. Beispiele für totale Ordnungen sind Zwerge und Zahlen, die nach Größe, Bauarbeiter, die nach Gewicht oder Wörter, die lexikografisch verglichen werden können. Manche Dinge können nach unterschiedlichen Kriterien verglichen und geordnet werden. Die Zwerge beispielsweise nach Größe, Gewicht oder Namen. Es darf aber keine zwei Dinge geben, die nach dem gewählten Kriterium nicht vergleichbar sind. Zwerge kann man nach Größe ihrer Zipfelmütze vergleichen, Bauarbeiter aber nicht. Es gibt ja Bauarbeiter, die ohne Zipfelmütze zur Arbeit gehen.

Es ist naheliegend, die Dinge nach ihrer Ordnung *an*zuordnen. Die Zwerge beispielsweise nach Größe.

Vergleich $\Rightarrow$ Ordnung $\Rightarrow$ Anordnung

Die Idee der Suchbäume besteht darin, von der naheliegenden Anordnung entsprechend der Ordnung abzuweichen und die Dinge so anzuordnen, dass ihre *Anordnung* nur *halbwegs* ihrer inneren/natürlichen *Ordnung* entspricht.

Vergleich ⇒ Ordnung ⇝ Anordnung

Die Zwerge stehen dann nicht nach ihrer Größe in einer Reihe, sondern platzieren sich in einem Suchbaum, dessen Struktur von der Ordnung, der Größe, beeinflusst, aber nicht komplett bestimmt wird (siehe Abbildung 5.12).

Das *Halbwegs-(an)geordnet-Sein* ist dabei ein Kompromiss zwischen

- ✔ völlig ungeordneter Speicherung mit schnellem Abspeichern und aufwendiger Suche und
- ✔ völlig geordneter Speicherung mit aufwendigem Abspeichern und schneller Suche.

Der Suchbaum ist also ein Kompromiss zwischen Chaos und pedantisch-penibelster (An-)Ordnung (siehe Abbildung 5.13).

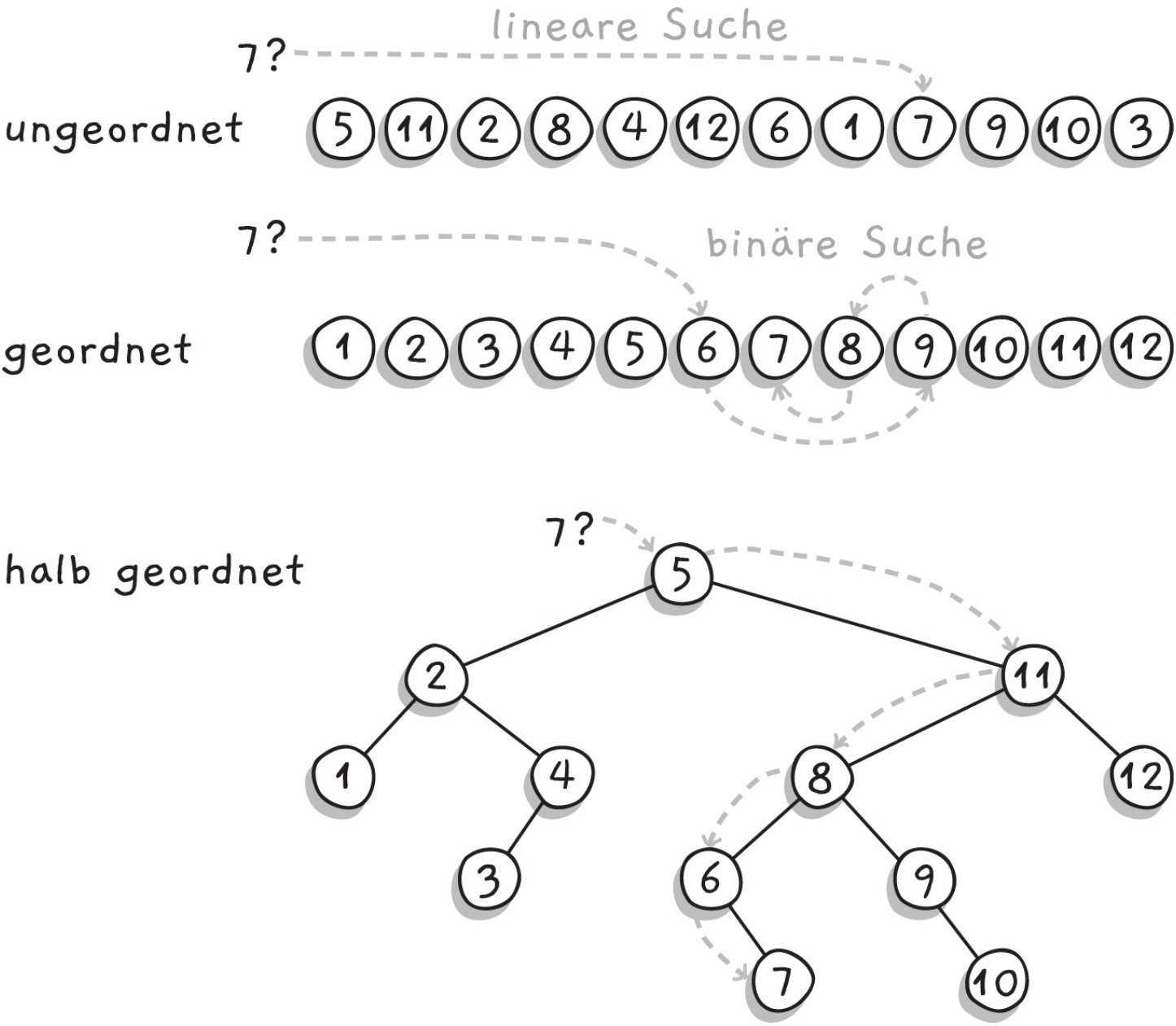


Abbildung 5.13: Suchbaum: Kompromiss zwischen Ordnung und Chaos

Die einfachste Form eines Suchbaums ist der binäre Suchbaum:

Ein *binärer Suchbaum* ist ein binärer Baum mit folgenden Eigenschaften:

- ✔ Jeder Knoten N in einem solchen Baum hat einen Wert N_v.
- ✔ Die Werte sind total geordnet (das heißt, sie sind vergleichbar mittels der Relationen $>, =, <$).
- ✔ Die Werte im Baum sind eindeutig: Zwei unterschiedliche Knoten haben stets unterschiedliche Werte.
- ✔ Für jeden Knoten N mit Nachfolger L und R gilt: $L_v < N_v < R_v$.

In einem Suchbaum können nur Daten gespeichert werden, die zu einer total geordneten Menge gehören. Mit Suchbäumen können diese Daten so verwaltet werden, dass Suchen, Eintragen und Entfernen einen ausgeglichenen Aufwand erfordern.

Der zu erzielende Gewinn ist nicht fix, er hängt von der Form des Baumes ab. Im schlechtesten Fall ist der Baum zu einer Liste degeneriert. Jede Operation ist dann in $\mathcal{O}(n)$. Generell sind wir in $\mathcal{O}(h)$, wobei h die Baumhöhe ist. Bei schön gewachsenen kompakten, *balancierten* Bäumen, ohne herausstakende überlange Äste, ist $h = \lg n$.

Verzeichnisse als Suchbäume

In einem Verzeichnis sind Daten unter einem Schlüssel abgespeichert. In einem Telefonverzeichnis findet man zu einem Namen beispielsweise die zugehörige Telefonnummer. Die typischen Operationen bei einem Verzeichnis sind Einfügen und Suchen.

Ein solches Verzeichnis stellt man sich in der Regel als Liste vor. Das intuitive Datenmodell ist darum *Liste*. Verzeichnisse werden in der Regel auch als Liste präsentiert. Seine Kontakte sieht man auf dem Smartphone beispielsweise als Liste. Das bedeutet nicht, dass sie auch in einer Datenstruktur *Liste* gespeichert werden. Ok, nur bei einer außerordentlich gut vernetzten Person wird es sich wohl großartig lohnen, die Kontaktliste in einer besonderen Datenstruktur zu speichern. Das Handy-Betriebssystem verwaltet die Kontakte all seiner Nutzer aber ganz sicher nicht in einer Liste.

Wegen ihres ausgeglichenen Verhaltens beim Suchen und Einfügen sind Suchbäume eine gute Wahl, wenn es um eine Datenstruktur für Verzeichnisse geht. Die Knoten enthalten dann die zu speichernde Information. Ein Teil dieser Information wird als *Schlüssel* bezeichnet. Der Schlüssel ist das, nach dem gesucht werden kann. In einem Telefonverzeichnis ist der Schlüssel beispielsweise der Name, die Telefonnummer ist dann der dem Schlüssel zugeordnete Wert. Natürlich ist der Schlüssel die relevante Information bei den Operationen Suchen und Eintragen. Wie stellen uns also vor, dass eine Information ein Paar aus einem Schlüssel und einem zugeordneten Wert ist.

Damit können wir schon einen binären Suchbaum als Datenstruktur für Verzeichnisse definieren:

type BSTree = Node((*Key*, *Value*), BSTree, BSTree) | Empty

Ein BSTree ist also leer oder er enthält ein Schlüssel-Wert-Paar und einen linken und einen rechten Unterbaum. Schlüssel und Wert gehören zu einem Typ Key beziehungsweise $Value$, die hier nicht weiter relevant sind. Key könnte beispielsweise die Zwergengröße in cm sein und $Value$ der zugehörige Zwerg.

Am Anfang ist der Baum leer. Alle Zwerge haben die WG verlassen.

$t : BSTree \leftarrow$ Nil *der Suchbaum als Datenspeicher, am Anfang leer*

```
insertI(key, value)
 ...                                   füge key und value in den Baum t ein, verändere ihn damit
lookup(key)
 ...                                   suche value zu key in t
 return value
```

In der Objektorientierung fasst man die Operationen und die Datenstruktur in einem Objekt beziehungsweise einer Klasse zusammen. Verzeichnisse sind in der Regel imperativ gemeint: Einfügen verändert die Datenstruktur.

Bei einer funktionalen Version des Verzeichnisses liefert insert dagegen einen neuen Baum, der dann statt t weitergenutzt wird:

```
insertF(key, value, t)
 ...                                   konstruiere mit key, value und t einen neuen Baum t'
 return t'
```

In einer milden Form eines imperativen Algorithmus wird insertF in insertI genutzt:

```
insertI(key, value)
 t ← insertF(key, value, t)
```

Mild imperativ, weil intern der Baum durch eine modifizierte neue Version ersetzt wird. In einer aggressiveren Form eines imperativen Algorithmus wird t nicht persistent sein: insertI wird die internen Zeiger des Baumes modifizieren.

Bleiben wir erst einmal bei insertF, dem Einfügen »«in»« eine persistente Datenstruktur, mit seiner recht einfachen rekursiven Definition:

```
insertF(key, value, t)
    case t = Empty                          der Baum ist leer: erzeuge neuen Knoten
        return Node((key, value), Empty, Empty)
    case t = Node((k, _), _, _) and k = key     key ist schon enthalten, wird ignoriert
        return t
    case t = Node((k, v), l, r) and k > key     key ist kleiner als der Wert der Wurzel
        return Node((k, v),
                    insertF(key, value, l),     gehe rekursiv nach links
                    r)
    case Node((k, v), l, r) and k < key         key ist größer als der Wert der Wurzel
        return Node((k, v),
                    l,
                    insertF(key, value, r))     gehe rekursiv nach rechts
```

Was beim Einfügen passiert, ist Folgendes (siehe Abbildung 5.14):

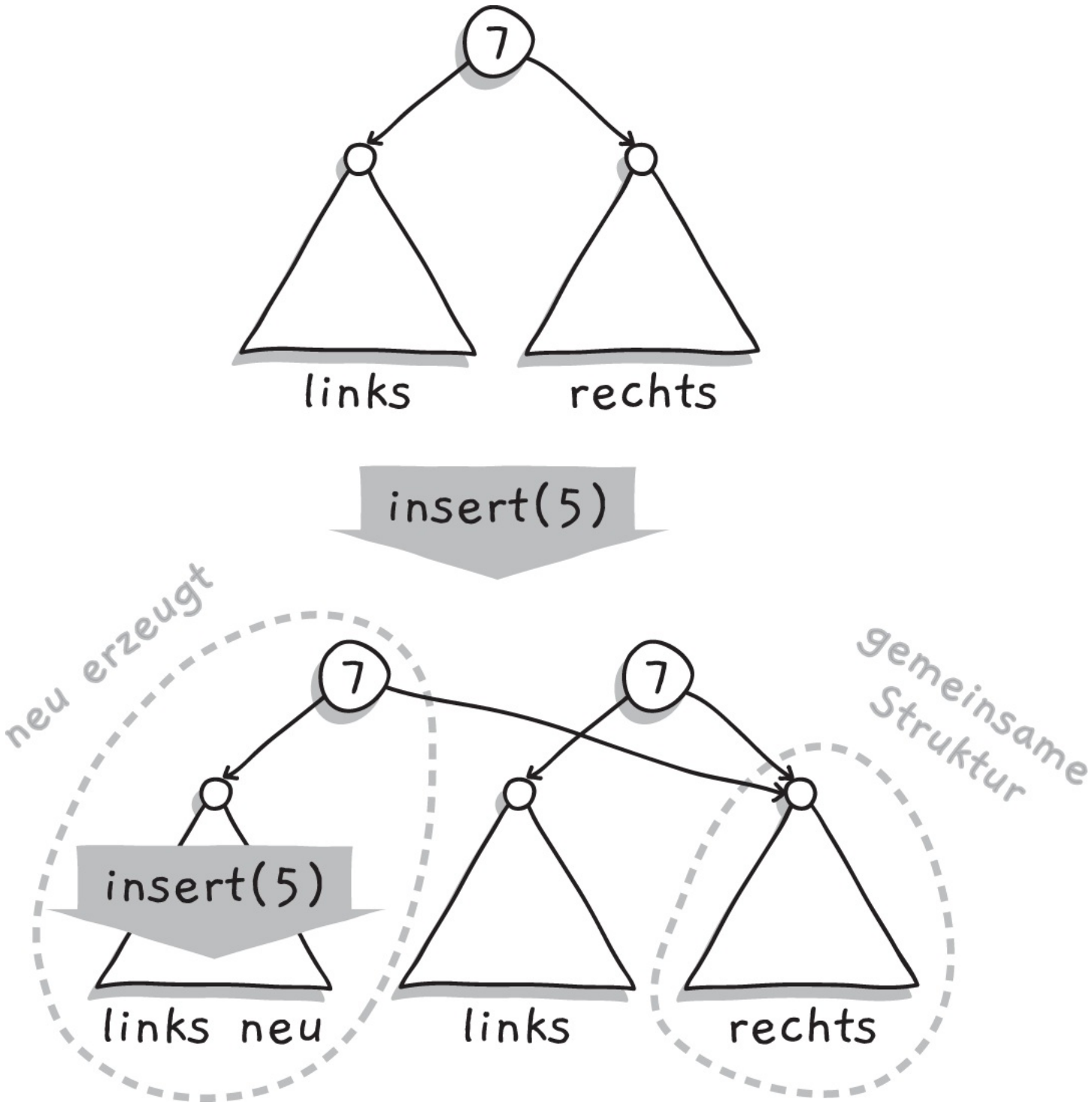


Abbildung 5.14: Suchbaum: Einfügen von Schlüssel 5 in einen Baum mit Wurzel 7

- ✔ Ist der Baum leer, dann wird er neuer Knoten erzeugt.
- ✔ Ist key kleiner als der Schlüssel k in der Wurzel, dann wird der linke Unterbaum mit der neuen Information neu erzeugt. Der rechte Unterbaum wird von der neuen und der alten Version gemeinsam genutzt.
- ✔ Ist key größer als der Schlüssel k in der Wurzel, dann wird der rechte Unterbaum mit der neuen Information neu erzeugt. Der linke Unterbaum wird von der neuen und der alten Version gemeinsam genutzt.

- ✔ Ist der Schlüssel in der Wurzel gleich key , dann ist der Schlüssel schon enthalten und die Aktion wird ignoriert.

Das soll hier erst einmal genügen, um einen Eindruck davon zu bekommen, wie Bäume als Informationsspeicher genutzt werden können, die wir uns intuitiv keineswegs als Baum vorstellen.

Bäume verkleiden sich gerne mal

Bäume sind die Mutter aller Datenstrukturen. Im letzten Abschnitt haben wir binäre Suchbäume betrachtet, die verwendet werden, um Informationsspeicher zu realisieren, die zunächst einmal nichts mit Bäumen zu tun haben. Daneben gibt es noch viele Anwendungen, in deren Kern Bäume stecken, die aber als etwas anderes verkleidet wurden. Beispielsweise als Listen, Warteschlangen oder als Wörterverzeichnisse.

Tries

Die Datenstruktur *Trie* ist eine Variante der Suchbäume. Der etwas seltsame Name ist ein Wortspiel von Tree mit dem Begriff *information reTrieval*. Es geht, wie bei binären Suchbäumen, um eine Datenstruktur zur Speicherung von Informationen, die effizienten Lese- und Schreibzugriff möglich macht.

Ein Trie ist eine Datenstruktur für Verzeichnisse, also Schlüssel-Wert Beziehungen. Die Schlüssel müssen dazu in Folgen vergleichbarer Objekte zerlegt werden können. Worte (Strings) sind da der Standardfall. Sie bestehen aus Zeichen die miteinander verglichen werden können. Zur Vereinfachung ignorieren wir die Werte und nehmen an, dass nur Schlüssel gespeichert werden.



Ein *Trie* ist eine baumförmige Datenstruktur zur Speicherung einer Menge von Worten, die aus vergleichbaren Buchstaben bestehen:

- ✔ Verbindungen werden mit Buchstaben markiert.
- ✔ Ein Wort wird durch einen Pfad von der Wurzel aus repräsentiert.
- ✔ Worte müssen nicht in einem Blatt enden.
- ✔ Um zu unterscheiden, ob an einem Knoten ein Wort endet, werden die Knoten markiert:
    - Markierte Knoten repräsentieren ein Wort-/Schlüssel-Ende.
    - Keine Markierung: kein Wortende, der Pfad bis hier ist kein Schlüssel.
- ✔ Markierte Knoten enthalten den Wert zum Schlüssel, dessen Pfad in diesem Knoten endet.

Beispielsweise kann ein Verzeichnis mit den Schlüsseln *der, dampf, dauer, da, daumen* kompakt in einem *Trie* gespeichert werden (siehe Abbildung 5.15). Ein solches Wörterverzeichnis in Trie-Form ist ideal für einen Suchalgorithmus mit Autovervollständigung. Zu jeder Buchstabenfolge kann das Verzeichnis ja leicht alle möglichen Vervollständigungen finden und anzeigen.

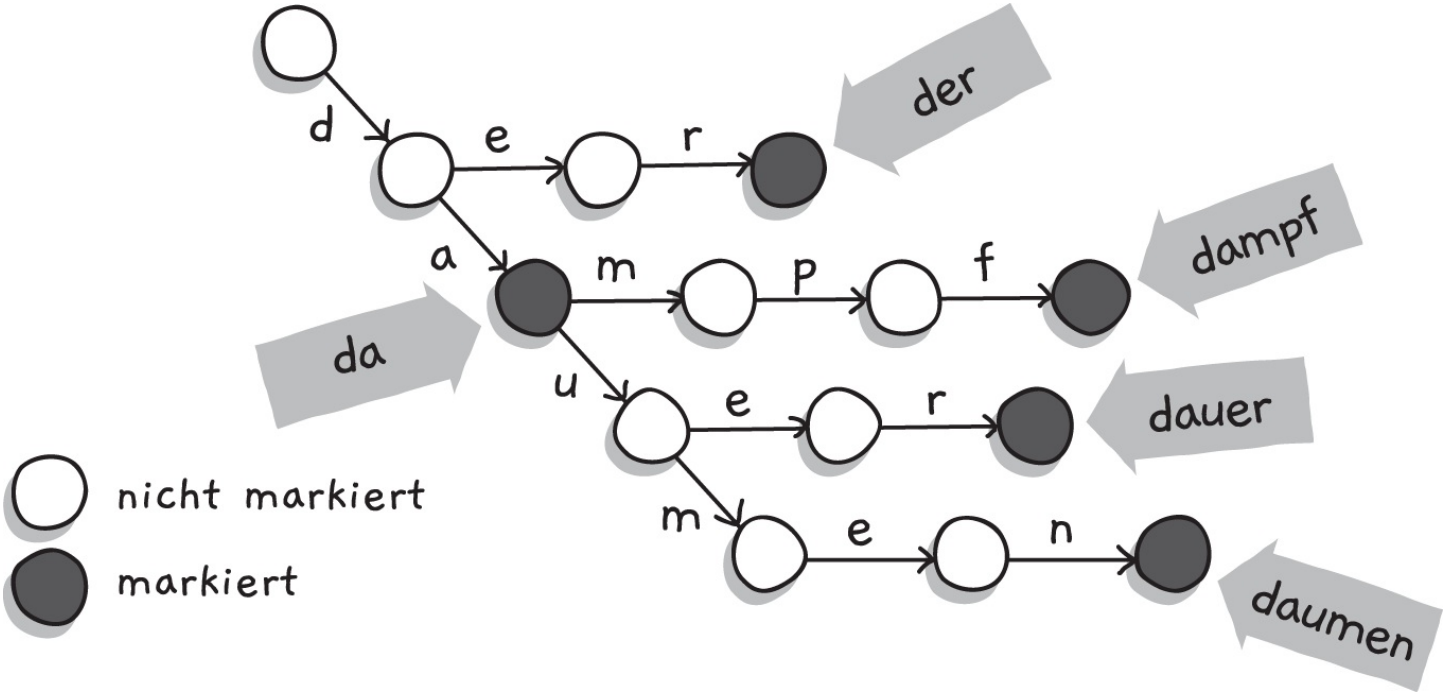


Abbildung 5.15: Ein Trie

Tries können funktional oder imperativ realisiert werden. In der funktionalen Variante ist die Datenstruktur persistent: Alles, was einmal erschaffen wurde, wird nicht mehr verändert. In der imperativen Variante werden Daten im Inneren der Struktur verändert.

Als persistente Datenstrukturen haben Tries ein gewisses Potenzial, denn sie bieten mehr »«Einstiegsstellen»« als Listen. Gemeinsame Teilstrukturen von alter und neuer Version einer Datenstruktur können damit geteilt werden (siehe Abbildung 5.16).

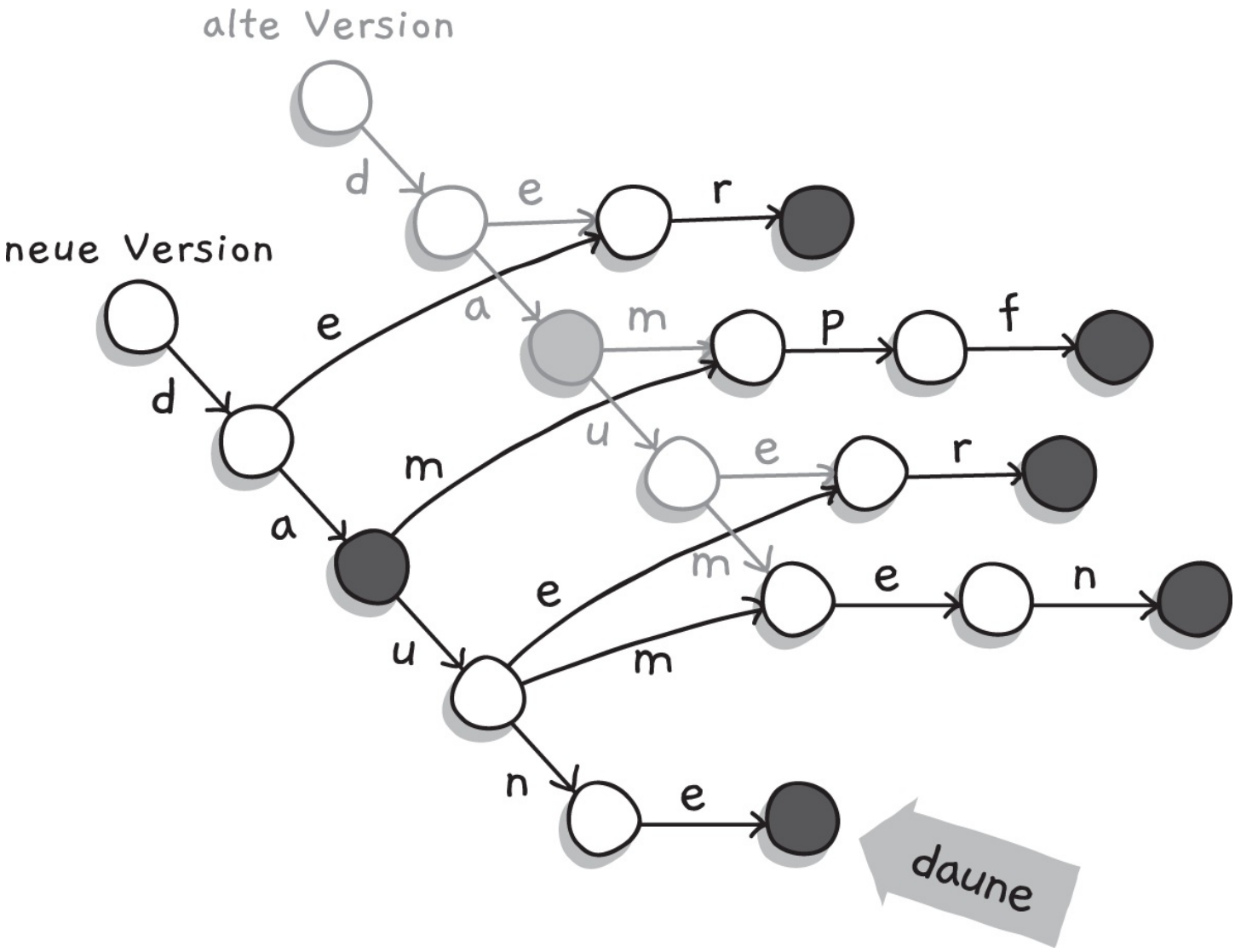


Abbildung 5.16: Trie als persistente Datenstruktur

Wegen dieser günstigen »«sharing»«-Eigenschaft werden Tries gelegentlich als Datenstruktur bei der Implementierung von persistenten Listen verwendet – speziell dann, wenn die Liste in ihrem Inneren verändert, oder besser »«weiterentwickelt»« werden soll, führt Persistenz zu erheblichen Effizienzeinbußen. Die neue Version muss als komplette Kopie der alten erstellt werden. Eine Liste als Trie bietet erhebliche Effizienzvorteile (siehe Abbildung 5.17).

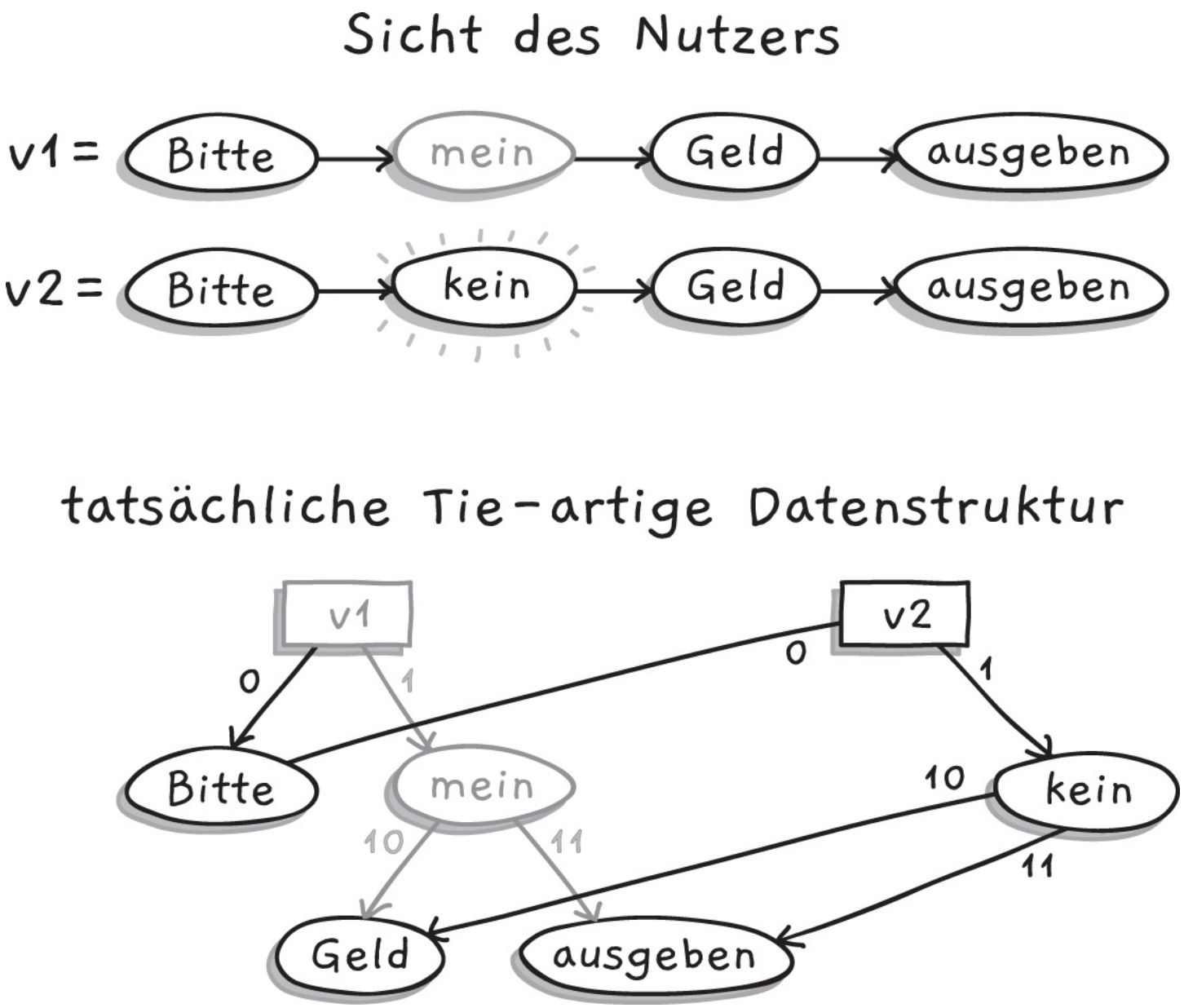


Abbildung 5.17: Trie als Datenstruktur für persistente Listen

Insgesamt hat sich also gezeigt, dass Bäume sich als Datenstruktur für Dinge eignen, die eigentlich gar keine Bäume, sondern Verzeichnisse, Mengen, Listen sind.



In Kapitel 9 wird das Thema Bäume als Datenstruktur für Verzeichnisse und andere Informationsspeicher vertieft.

Prioritätswarteschlangen

Gerecht ist es, wenn in einer Warteschlange alle in der Reihenfolge stehen, in der sie ankommen. Gerechtigkeit ist wichtig. Aber manchmal ist man selbst wichtiger als die Gerechtigkeit. In dem Fall möchte man natürlich in einer Warteschlange ganz vorn stehen, egal wann man gekommen ist. So etwas wird von *Prioritätswarteschlangen* unterstützt. Bei einer solchen Warteschlange kommt nicht der als nächster dran, der schon am längsten wartet, sondern der Wichtigste unter den Wartenden.

Prioritätswarteschlangen haben viele Anwendungen. Wahllisten sind üblicherweise Prioritätswarteschlangen der Mitglieder einer Partei, bei einer Notfallversorgung erwarten wir, dass es nach Priorität geht. Nicht nur in der wirklichen Welt, auch in vielen Softwaresystemen werden Prioritätswarteschlangen verwendet. Das gilt speziell für die Systemprogrammierung. Das Betriebssystem nimmt beispielsweise den Prozess, dem als nächstem Rechenzeit zugestanden wird, aus einer Prioritätswarteschlange. Grund genug, um über eine geeignete Datenstruktur nachzudenken.

Warteschlangen mit Priorität und Heaps

Der Einfachheit halber nehmen wir an, dass sich ganze Zahlen anstellen, deren Wichtigkeit ihrer Größe entspricht. Die Prioritätswarteschlange hat dann zwei Operationen Insert(x) und Remove. Remove liefert den größten bisher eingefügten Wert und entfernt ihn aus der Warteschlange. Insert(x) fügt x in die Warteschlange ein.

Als Datenstruktur zur Unterstützung dieser beiden Operationen bieten sich wieder Bäume an. Wenn jeder Knoten nur Nachkommen mit Werten hat, die kleiner oder gleich seinem eigenen Wert sind, dann ist der nächste schnell gefunden. Es ist die Wurzel. Beim Einfügen muss man sich auch nicht besonders anstrengen. Der neue Wert muss so platziert werden, dass er nicht über einem steht, der größer (wichtiger) ist als er selbst. Weitere Einschränkungen gibt es nicht. Ein schwach sortierter Baum modelliert die Anwendung perfekt.



Im letzten Abschnitt haben wir binäre Suchbäume betrachtet. Dabei handelt es sich auch um eine schwach sortierte Datenstruktur. Die schwache Sortierung hier ist eine andere, noch etwas schwächere.

Eine Baumstruktur mit dieser schwachen Sortierung nennt man *Heap*. In Abbildung 5.18 haben wir einen Heap mit 60 als wichtigstem Wert. Er ist in der Wurzel zu finden. Die beiden Unterbäume sind ebenfalls Heaps.

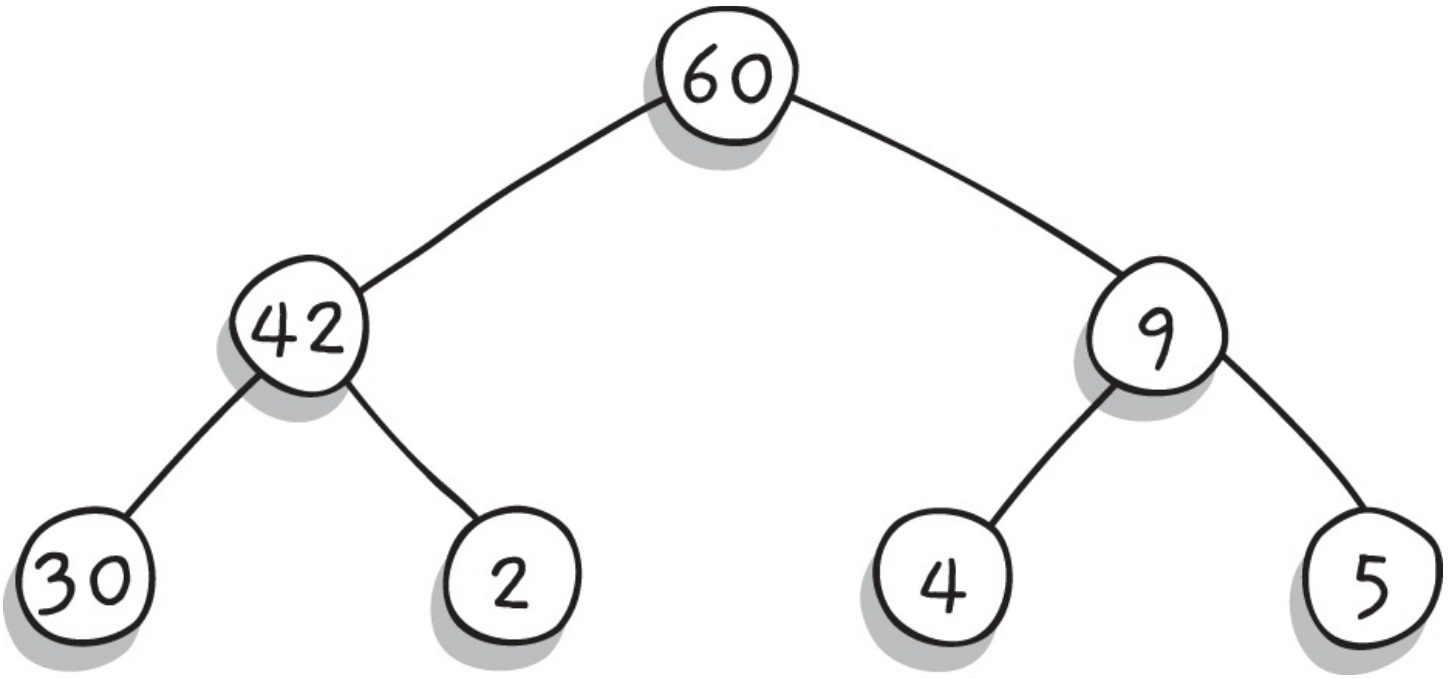


Abbildung 5.18: Eine Prioritätswarteschlange als Heap-Baum gespeichert

Ein Warteschlange ist etwas Veränderliches und sie soll auch mit einer veränderlichen Datenstruktur realisiert werden. Eine Implementierung mit einer persistenten Datenstruktur wäre natürlich auch möglich. Der Algorithmus soll hier aber aus Effizienzgründen nicht mit einem algebraischen Datentyp arbeiten. Wir bevorzugen einen *veränderlichen* Baum, dessen Knotenwerte sich ändern können. Die Wurzel des Baumes ist jetzt eine Variable, die Zeiger enthält, die auf einen Knoten oder auf nichts zeigen (**null** -Pointer). Knoten enthalten einen variablen Wert und zwei variable Zeiger auf die Unterbäume beziehungsweise auf Nichts.

```
type Node = record[value: Int, left: Heap, right: Heap]
newNode(x, l, r)
    return new record[value: x, left: l, right: r]
```

Insert($heap$, x) fügt x in einem Baum ein, auf dessen Wurzel der Wert der Variablen $heap$ zeigt. Dabei wird die Struktur des Baumes so wenig wie möglich verändert.

Es geht los mit einer Prüfung, ob der Baum leer ist. Wenn ja, werden ein neuer Knoten angelegt und ein Zeiger auf ihn abgespeichert: Wir nehmen hier an, dass dem Algorithmus die Variable übergeben wird, in der der Zeiger auf die Wurzel gespeichert ist, nicht der Zeiger selbst.

```
Insert(var root, x)                    füge x ein in Baum mit Wurzel in der Variablen heap
    if root = null
        root ← newNode(x, null, null)      ändere die übergebene Variable
    else
        ...
```

Hier wird geschaut, ob der Zeiger in $root$ überhaupt auf einen Knoten zeigt. Wenn nicht,

dann wird ein Knoten angelegt, mit x gefüllt und $root$ mit einem Zeiger auf den neuen Knoten belegt.

Wenn $root$ auf einen Knoten zeigt, dessen Wert kleiner als x ist, dann wird der Wert der Wurzel durch x ersetzt. Der alte Wurzelwert $rootValue$ muss jetzt weiter unten im Baum gespeichert werden. Andernfalls bleibt die Wurzel, wie sie ist, und x muss weiter nach unten wandern. Wenn irgendwo ein Nachfolger fehlt, dann wird x an die Leerstelle gesetzt. Im Detail:

```
else                                    root zeigt auf einen Record
    case object(rootP) = record[var rootValue, left, right]
        if x ≥ rootValue
            valueDown ← rootValue       rootValue muss nach unten wandern
            rootValue ← x               x ersetzt rootValue
        else
            valueDown ← x               x muss nach unten wandern
```

Wenn $root$ auf einen Knoten zeigt, dann kann der neue Wert x den Wert im Knoten ersetzen und der alte Knotenwert muss nach unten wandern oder der Wert der Wurzel bleibt und der neue Wert muss nach unten wandern. Was getan werden muss, hängt vom Vergleich des neuen Wertes mit dem Wert der Wurzel ab. Wenn der neue Wert x größer ist als der Wert in der Wurzel, dann ersetzt x den Wert in der Wurzel und dieser muss dann weiter nach unten wandern.

$rootValue$ wird hier (so wie $root$) als Variable genutzt: Links vom Zuweisungszeichen $\rightarrow$ ist wie üblich der Speicherplatz gemeint, rechts der Wert des Speicherplatzes.

Wenn der neue Wert x kleiner ist als der in der Wurzel, dann muss der neue Wert nach unten wandern. Einer muss also in jedem Fall nach unten und der steckt jetzt in $valueDown$. Es muss noch entschieden werden, ob er links oder rechts nach unten wandern muss. Das kann einigermaßen willkürlich entschieden werden, er muss ja nur weiter nach unten. Wenn einer der Unterbäume leer ist, dann kommt der neue Wert an den leeren Platz. Ansonsten kommt er in den Unterbaum, der die größere Wurzel hat. Damit erreichen wir, dass der Baum nicht so schnell außer Form gerät.

```
case left = null                    der linke Unterbaum ist leer, weiter nach rechts
    Insert(left, valueDown)

case right = null                   der rechte Unterbaum ist leer, weiter nach links
    Insert(right, valueDown)

                                    beide nicht leer: weiter zum größeren
case object(left) = record[lv, ll, lr],
    object(right) = record[rv, rl, rr]
    if lv > rv
        Insert(left, valueDown)
    else
        Insert(right), valueDown)
```

Fügt man beispielsweise $2, 9, 7, 6, 5, 8$ in den leeren Baum ein, dann wird sukzessive der Baum in Abbildung 5.19 aufgebaut. Im letzten Schritt verdrängt die 8 die 7 aus ihrem Knoten nach links unten. Dort verdrängt die 7 dann die 6, für die schließlich ein neuer Knoten angelegt wird.

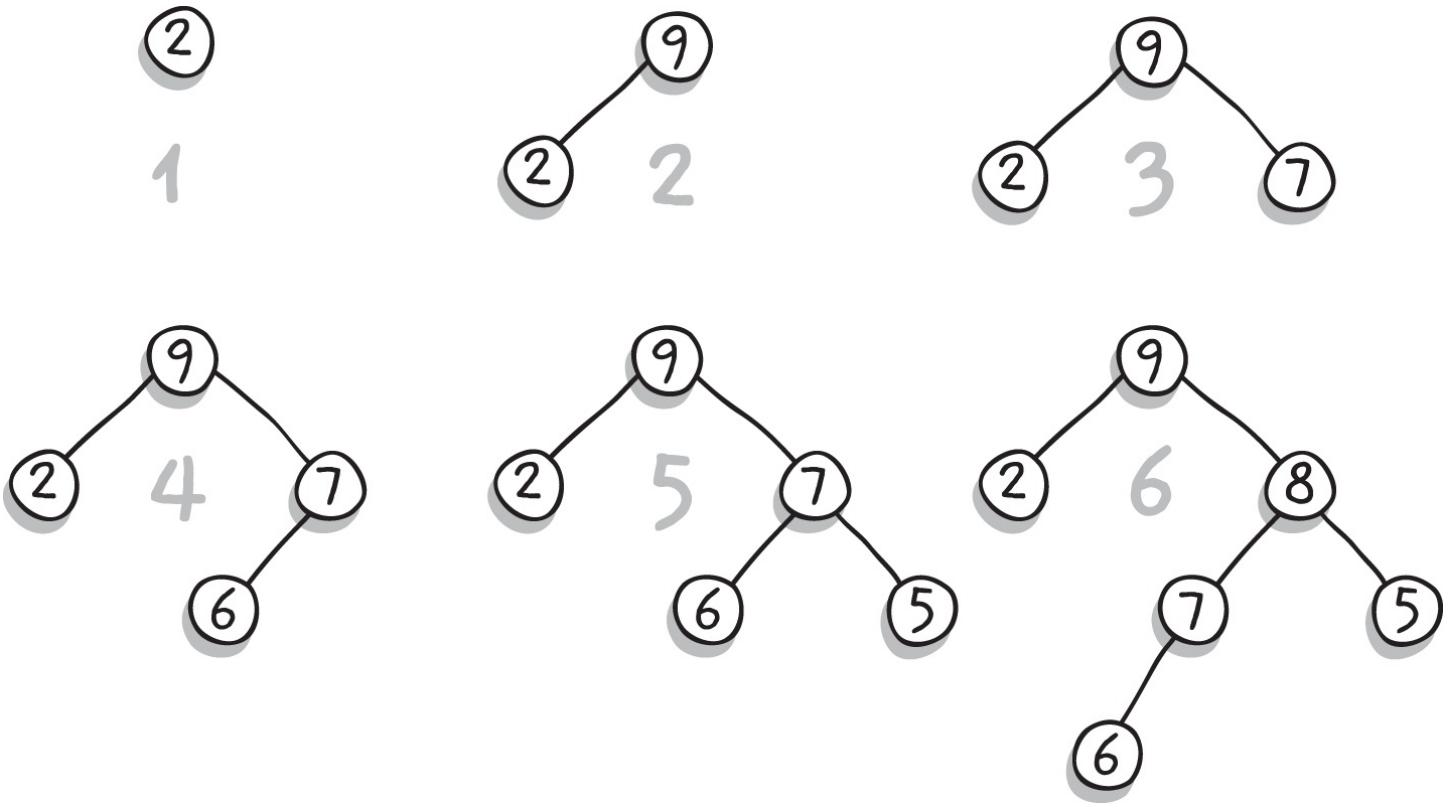


Abbildung 5.19: Aufbau eins Heaps



In diesem Algorithmus »«auf Zeigerebene»« bewegen wir uns auf der Ebene der systemnahen Programmierung. Es werden sowohl Variablen als auch Werte an Funktionen übergeben. Dazu verwendet man üblicherweise eine systemnahe

Programmiersprache, welche die entsprechenden Aktionen auf einfache und elegante Art unterstützt.

Beim Entnehmen wird der wichtigste, hier als der größte definierte Wert aus dem Baum entfernt und ausgeliefert. Wenn der Wert aus der Wurzel entnommen wird, dann ist diese frei. Wir nehmen einfach den Wert eines Nachfolgers und setzten ihn in die Wurzel. Wenn es keinen Nachfolger gibt, dann kann der jetzt leere Knoten entfernt werden. Gibt es nur einen Nachfolger, dann wird der Wert aus diesem nach oben geschoben. Gibt es zwei Nachfolger, dann nehmen wir den größeren. Dabei wird natürlich weiter unten etwas frei. Das Verfahren muss dann noch rekursiv nach unten fortgesetzt werden.

Bäume als Arrays

Bäume verkleiden sich nicht nur oft als etwas anderes. Etwas anderes kann sich auch als Baum verkleiden. Ein Binärbaum kann beispielsweise gut in einem Array abgespeichert werden, wenn er gleichmäßig gefüllt ist. Wenn also die Elemente sich Ebene für Ebene in ein Array einfügen lassen (siehe Abbildung 5.20). Das Array kann dann die Rolle eines Baumes übernehmen.

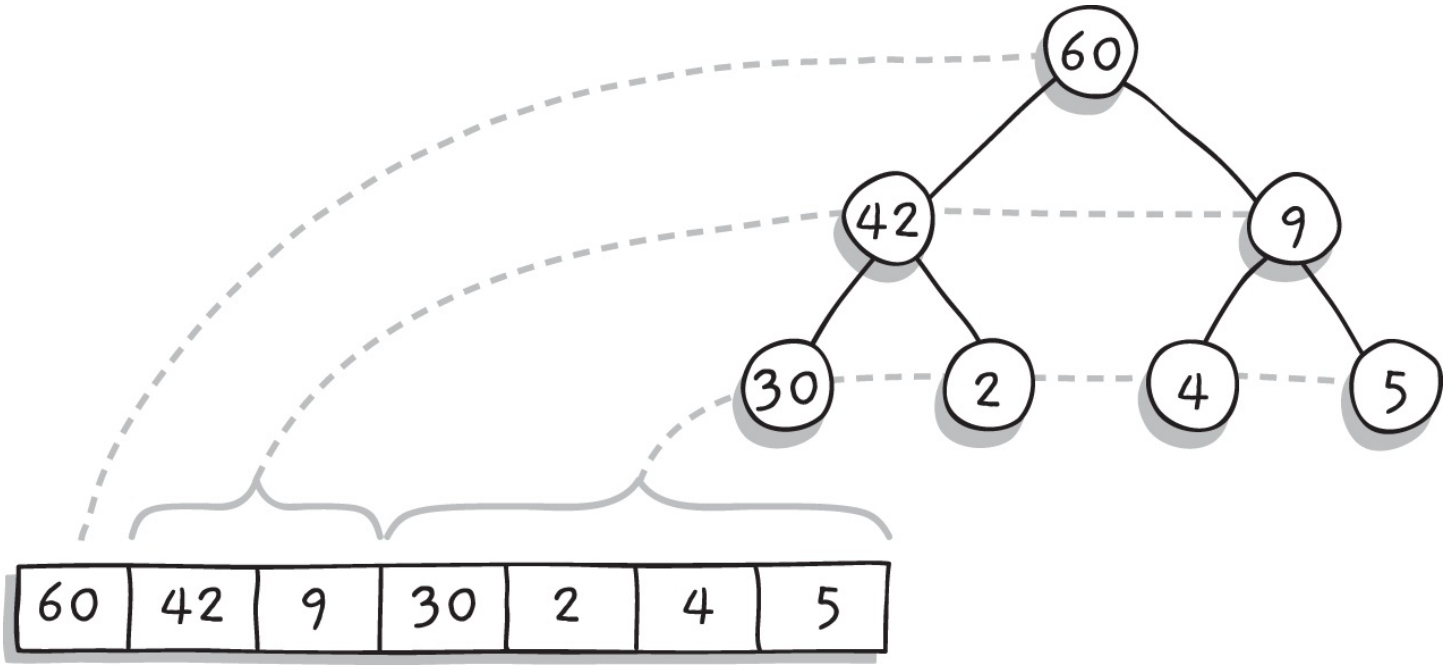


Abbildung 5.20: Binärbaum in einem Array

Bei dieser Speicherung in Ebenen können die Zeiger leicht durch eine Indexrechnung ersetzt werden. Jede Ebene hat doppelt so viele Elemente wie die Ebene über ihr. Verdoppelt man den Index eines Elements, dann kommt man zum Index seines Nachfolgers, halbiert man ihn, dann landet man beim Vorgänger. Das gilt, wenn der erste Index die Nummer 1 hat. Beginnen die Indizes mit 0, dann führt die Verdopplung vor den ersten Nachfolger.

Die Operationen auf Zeigern können so durch Indexrechnungen ersetzt werden. Der Baum muss allerdings völlig gefüllt sein oder man benötigt eine spezielle Markierung für

die Lücken. Wenn zumindest alle Ebenen bis auf die letzte befüllt sind und diese auch nur rechts eine Lücke haben darf, dann reicht es zu wissen, wie viele Elemente der Baum hat.

Alle Operationen auf diesem, als Baum verkleideten Array sind im Prinzip die gleichen wie auf einem »«richtigen»« Baum. Statt mit Zeigern wird mit Indizes hantiert. Dadurch wird die Sache etwas effizienter: Man spart sich die Speicherverwaltung (Anfordern von neuem Speicherplatz vom Betriebssystem, Freigabe von nicht benutztem Speicherplatz) und die kompakte Speicherung in einem zusammenhängenden Bereich hat auch gelegentlich Vorteile beim Zugriff. Dafür ist der Baum aber auf eine fixe Maximalgröße beschränkt.

Insgesamt sehen wir, dass nicht nur Dinge, die wie Listen aussehen, tatsächlich Bäume sind und Dinge, die wie Bäume aussehen, tatsächlich Arrays sind. Datenmodelle und Datenstrukturen müssen halt auseinander gehalten werden.



Im Abschnitt *Besser sortieren mit dem Heap* in Kapitel 7 wird die Datenstruktur Heap in Gestalt eines Arrays genauer betrachtet. Dabei wird auch erläutert, wie sie zum Sortieren genutzt werden kann.

Selbstverständlich sind Heaps in Arrayform auch eine geeignete Datenstruktur für Prioritätswarteschlangen.

Kommen wir jetzt von den Dingen, die eigentlich keine Bäume sind, aber aus Effizienzgründen eine Baumgestalt angenommen haben, zu den Dingen, die nicht nur tatsächlich, sondern auch eigentlich Bäume sind.

Bäume als Datenmodell

Hinter vielen komplexen Beziehungen der Wirklichkeit steckt ein gemeinsames Muster. Vieles ist beispielsweise hierarchisch aufgebaut. Eine Organisation ist hierarchisch aufgebaut: Geschäftsführung, Abteilung, Unterabteilung und so weiter. Ein Satz in einer natürlichen Sprache ist zwar etwas völlig anderes, teilt sich aber mit der Organisation den hierarchischen Aufbau: Satz, Hauptsatz, Nebensatz, Nominal- und Verbalphrase, Worte, Silben, Laute.

Für Informatiker ist alles Hierarchische ein Baum. Etwas mit einer Wurzel, ab der sich die Dinge nach unten verzweigen. (Es ist schwer begreiflich, warum die hölzernen Dinge, die in der Gegend herumstehen und ebenfalls Baum genannt werden, immer auf dem Kopf stehen.)

Manchmal und für manches ist ein Baum wie der andere. Viele Algorithmen müssen nicht beachten, was der Baum in der Wirklichkeit darstellt. Datenstrukturen sind darum auch oft Modelle der Wirklichkeit, die zu vielen Dingen und Situationen passen. Probleme, die auf den ersten Blick kaum etwas miteinander zu tun haben, sind dann irgendwie doch die

gleichen, die mit den gleichen, hoffentlich bekannten und bewährten Verfahren behandelt werden können.

Ein Baum kann also dazu dienen, eine beliebige hierarchische Struktur abzubilden, zu modellieren (siehe Abbildung 5.21). Er kann aber auch eine Speicherorganisation für etwas sein, das mit Hierarchien nichts zu tun hat (siehe Abbildung 5.17). Ein Baum ist Datenmodell oder Datenstruktur oder beides.

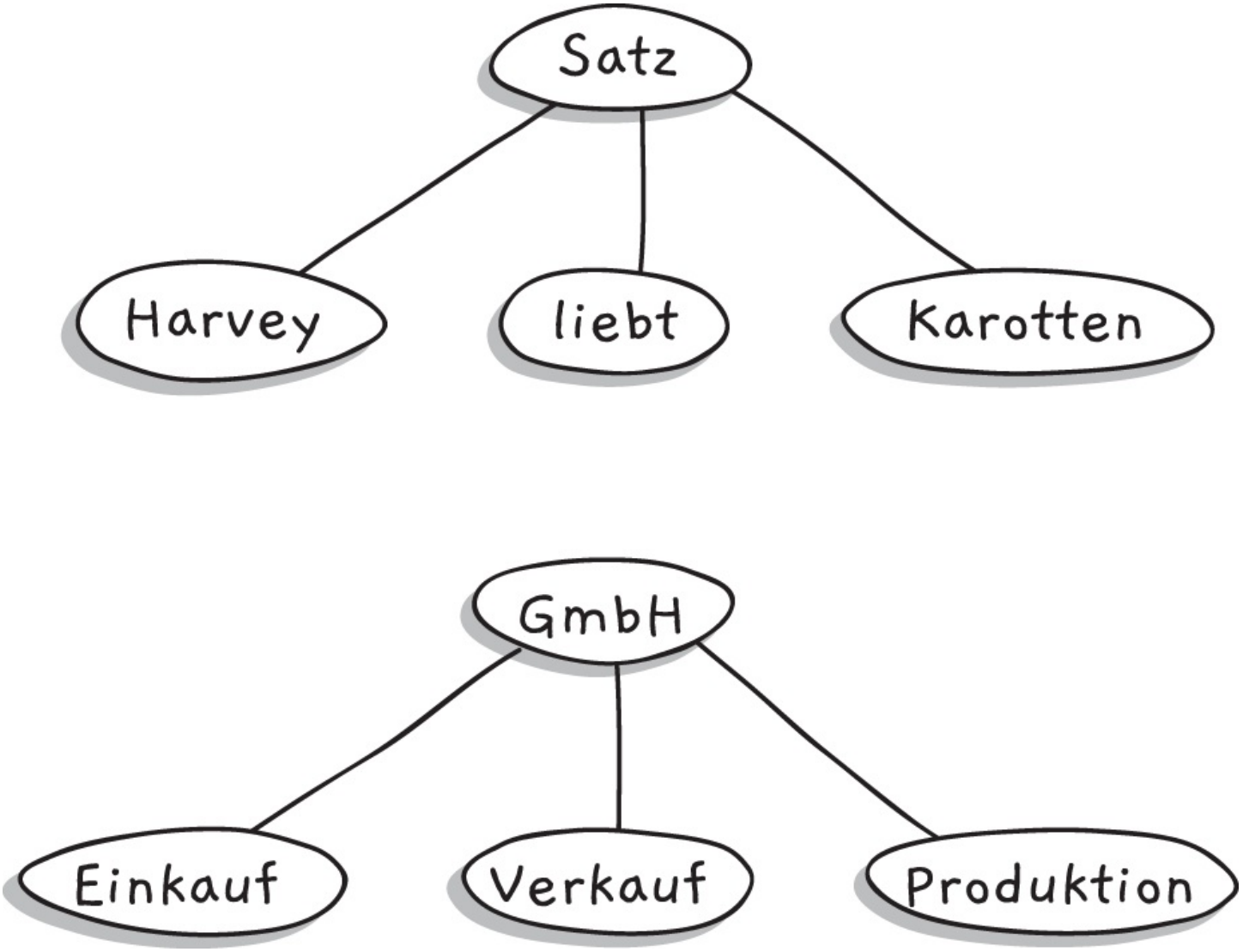


Abbildung 5.21: Die Welt ist voller Bäume.



Bäume sind ein wichtiges Datenmodell und gleichzeitig eine wichtige Datenstruktur. Bäume als Datenstruktur und Traversierungen als allgemeine Algorithmen auf dieser Datenstruktur haben wir am Anfang von diesem Kapitel betrachtet.

In einem Baum herrscht eine strenge Ordnung, alles geht von der Wurzel aus und verzweigt in klaren Stufen. Nicht überall geht es so streng zu. Manchmal sind die Dinge irgendwie miteinander verbunden. Das bringt ein *Graph* zum Ausdruck. So wie ein Baum das Modell aller Dinge ist, die hierarchisch organisiert sind, so ist der Graph das Modell

von allem, das irgendwie organisiert ist.



Graphen sind in erster Linie ein Datenmodell. In Kapitel 6 beschäftigen wir uns genauer mit Graphen und dem, was mit Graphen modelliert werden kann.

Hierarchische Strukturen lassen sich grafisch gut in Baumform darstellen. Es gibt übergeordnete Elemente, denen andere Elemente untergeordnet sind, die selbst eventuell wieder anderen übergeordnet sind. Es ist nicht zwingend notwendig, derartige Verhältnisse in einer baumartig verzweigten Datenstruktur zu speichern, aber naheliegend und in der Regel auch sinnvoll ist es schon. In der Form von *algebraischen Datentypen* können hierarchische Strukturen in manchen Programmiersprachen sehr abstrakt und weit weg von Zeigern definiert werden. In anderen muss man mit weniger Hilfe vom Compiler auskommen und die Knoten händisch anlegen und sie auch explizit mit Zeigern verknüpfen.

Ausdrucksbäume

Dinge, die hierarchisch strukturiert sind, lassen sich gut als Bäume modellieren. Und wenn wir Glück haben, dann können unsere Algorithmen sich an dieser Struktur entlanghangeln: Das Problem kann mit struktureller Rekursion gelöst werden.

Nehmen wir als Beispiel einfache arithmetischen Ausdrücke wie $2 + 3 \cdot 4$. Zu 2 soll $3 \cdot 4$ addiert werden. 2 und $3 \cdot 4$ sind der Addition damit untergeordnet: Sie müssen zuerst berechnet werden. $3 \cdot 4$ ist in gleicher Art als Multiplikation mit untergeordneter 3 und 4 strukturiert. Das Ganze ist dann ein *Rechenbaum* oder allgemeiner ein *Ausdrucksbaum* (siehe Abbildung 5.22).

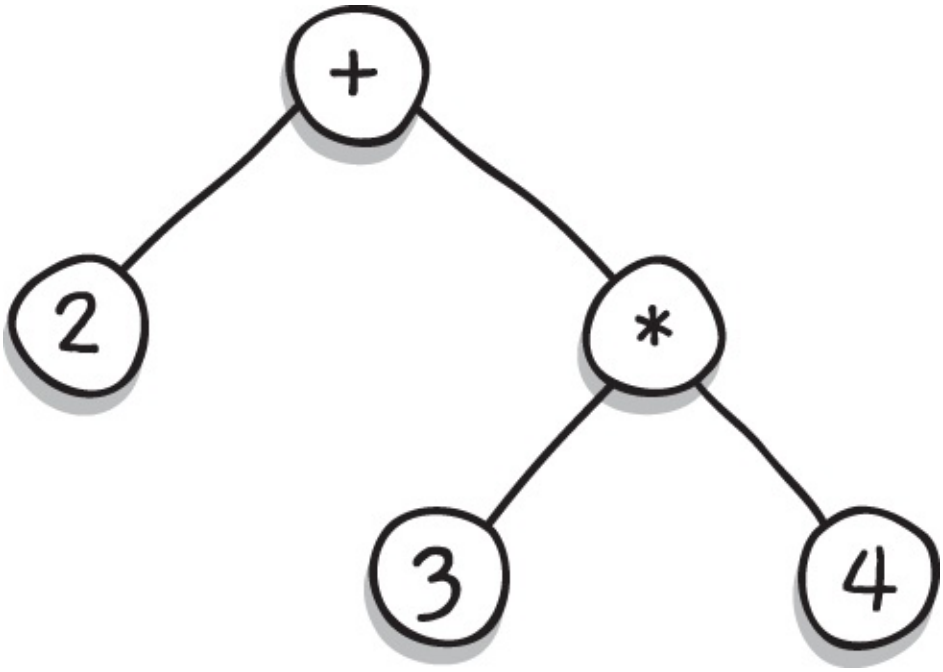


Abbildung 5.22: Ein Rechenbaum: ein arithmetischer Ausdruck als Baum

Das können wir genauso in einer Datenstruktur darstellen. Zur Definition bietet sich ein algebraischer Datentyp an:

type Exp = Value(Int) | Add(Exp, Exp) | Sub(Exp, Exp) | Mult(Exp, Exp)

Ein solcher Baum kann mit einem strukturell rekursiven Algorithmus ausgewertet werden:

```
Evaluate(exp)
    case exp = Value(v)
        return v
    case exp = Add(e1, e2)
        return Evaluate(e1) + Evaluate(e2)
    case exp = Sub(e1, e2)
        return Evaluate(e1) − Evaluate(e2)
    case exp = Mult(e1, e2)
        return Evaluate(e1) ∗ Evaluate(e2)
```

Dabei wird der Baum in Postorder durchlaufen: Erst werden die Kinder betrachtet, dann der Knoten selbst. Klar, bevor die Operation ausgeführt werden kann, müssen die Operanden berechnet werden.

Mit Stapeln übersetzen und auswerten

Statt die Operationen gleich auszuführen, können wir beim Durchlaufen der Struktur auch alles aufsammeln. Dabei entsteht eine Liste von Werten und Operationen:

type Node = ValueNode(Int) | AddNode | MultNode

```
EvalToNodes(exp)
    case exp = Value(v)
        return [ ValueNode(v) ]
    case exp = Add(e1, e2)
        return evalToNodes(e1) + evalToNodes(e2) + [ AddNode ]
    case exp = Sub(e1, e2)
        return evalToNodes(e1) + evalToNodes(e2) + [ SubNode ]
    case exp = Mult(e1, e2)
        return evalToNodes(e1) + evalToNodes(e2) + [ SubNode ]
```

Der Kürze halber verwenden wir hier wieder »«+»« als Zeichen für die Verknüpfung von Listen und »«[ ]»« für Listen, die durch die Angabe ihrer Elemente definiert werden.

Für den Ausdruck in Abbildung 5.22 liefert der Algorithmus beispielsweise

[ValueNode(2), ValueNode(3), ValueNode(4), MultNode, AddNode]

mit der Bedeutung: nimm 2, nimm 3, nimm 4, addiere, multipliziere.

Bei der Abarbeitung dieser Aktionsliste müssen die Zahlen und Zwischenergebnisse so lange gespeichert werden, bis sie verarbeitet werden können. Die 2, das erste Element der Liste, kann erst in der letzten Operation verarbeitet werden. Ein Algorithmus, der Knoten in ihrer Postorder-Reihenfolge verarbeitet, muss darum ein Gedächtnis haben, in dem die unverarbeiteten Werte gespeichert werden. Das Gedächtnis ist eine Liste unverarbeiteter Zahlen und wir nennenes Stapel (*Stack*), denn dort werden die unverarbeiteten Zahlen aufeinandergestapelt und von oben nach unten abgearbeitet:

```
EvalNodes(nodes, stack)
    case nodes = Nil
        return stack(0)

    case ValueNode(v) :: rest
        return EvalNodes(rest, v :: stack)

    case nodes = AddNode :: rest
        case stack = x :: y :: stackRest
            return evalNodes(rest, y + x :: stackRest)

    case nodes = SubNode :: rest
        case stack = x :: y :: stackRest
            return EvalNodes(rest, y − x :: stackRest)

    case nodes = MultNode :: rest
        case stack = x :: y :: stackRest
            return EvalNodes(rest, y · x :: stackRest)
```

Die Verarbeitung der Liste $nodes$, die EvalToNodes aus unserem Ausdruck von oben (Abbildung 5.22) erzeugt hat, erfolgt beispielsweise in den folgenden Schritten:

nodes	*stack*
[ValueNode(2),ValueNode(3),ValueNode(4),MultNode,AddNode]	[]
[ValueNode(3),ValueNode(4),MultNode,AddNode]	[2]
[ValueNode(4),MultNode,AddNode]	[3,2]
[MultNode,AddNode]	[4,3,2]
[AddNode]	[12,2]
[]	[14]

Der Algorithmus EvalNodes ist *endrekursiv*: Der rekursive Aufruf ist die letzte Aktion in einem Zweig. Wir können den Algorithmus darum recht leicht in eine Schleife umformen. Die Parameter werden zu Variablen, die rekursiven Aufrufe zu Schleifendurchläufen, in

denen sie verändert werden.

```
EvalNodesIter(nodes)
    stack ← Nil

    while nodesV ≠ Nil
        case nodes = ValueNode(v) :: rest
            stack ← v :: stack
            nodes ← rest

        case nodes = opNode :: rest
            x ← stack.head
            stack ← stack.tail
            y ← stack.head
            stack ← stack.tail
            case opNode = AddNode
                stack ← y + x :: stack
            case opNode = SubNode =>
                stack ← y − x :: stack
            case opNode = MultNode =>
                stack ← y ∗ x :: stack
        nodes ← rest

        return stack.head
```

In dieser Variante durchlaufen die Variablen $nodes$ und $stack$ die gleichen Werte wie die Argumente der rekursiven Aufrufe oben.

Die Auswertung von Ausdrucksbäumen kann also in zwei Phasen aufgeteilt werden:In eine Postorder-Traversierung, in der Aktionen gesammelt und linear geordnet werden, und in eine Auswertungsphase, in der diese dann iterativ in einer Schleife ausgewertet werden. Im Allgemeinen ist diese Aufteilung der Auswertung von Ausdrücken in zwei Phasen nicht sinnvoll. Für einen Compiler aber schon. In einem Programm wird derselbe Ausdruck ja mehr als einmal auszuwerten sein: Mindestens bei jedem Lauf des Programms, und taucht er in einer Schleife oder Funktion auf, dann noch dazu in jedem Schleifendurchlauf oder Funktionsaufruf. Es ist dann sinnvoll, ihn auf diese Weise in eine Folge von Maschinencode-Aktionen aufzubereiten, die sofort rekursionsfrei ausgeführt werden können.

Kapitel 6

Graphen: Jeder mit jedem

IN DIESEM KAPITEL

Der Graph: ein allgemeines Datenmodell

Graphen traversieren

Erreichbarkeit

Graphen speichern

Topologisches Sortieren

Geordnete Verhältnisse, über- und untergeordnet, Schachtel und Inhalt, das sind die Verhältnisse, die man mit Bäumen modelliert. So klar geregelt geht es natürlich nicht immer zu. Oft hat man es Strukturen zu tun, in denen es keinerlei Art von Hierarchie gibt. Wenn alles mit jedem in irgendeiner Beziehung stehen kann, dann hat man es mit einem *Graphen* zu tun, siehe Abbildung 6.1.

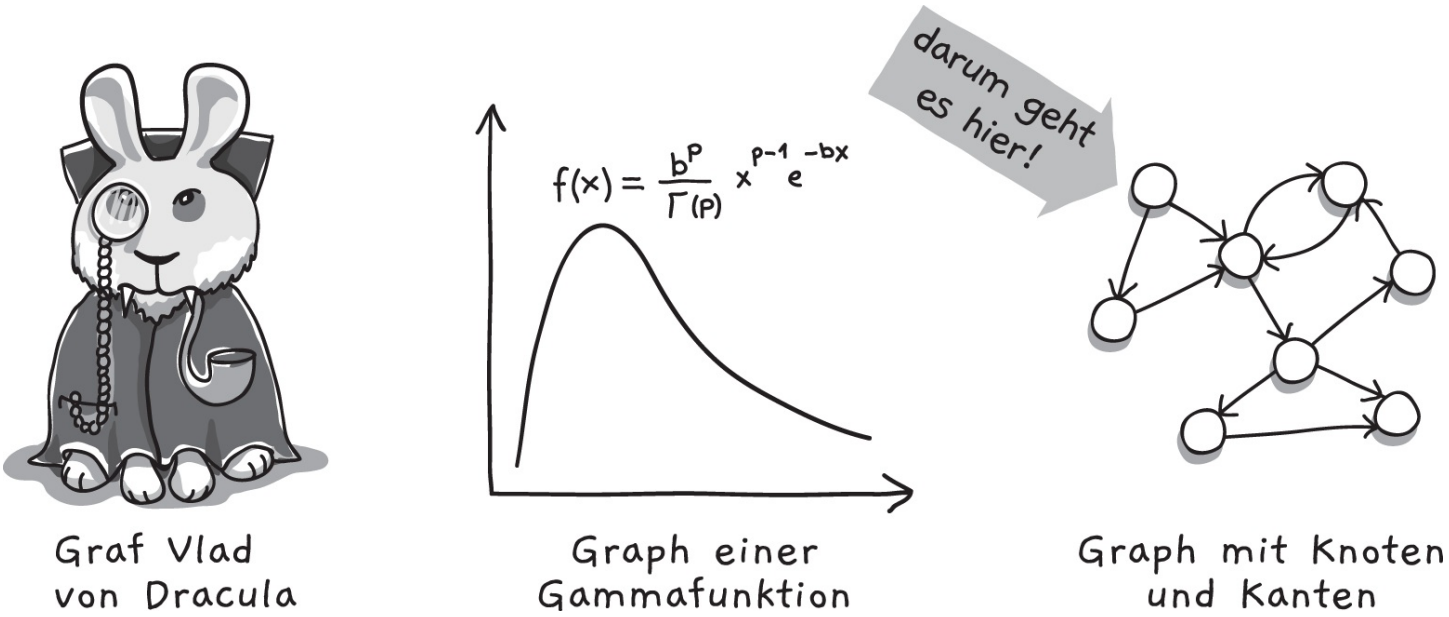


Abbildung 6.1: Drei verschiedene Arten von Graphen

Ein Graph ist etwas, das aus Dingen besteht, die irgendwie miteinander in Beziehung stehen. In gewisser Weise besteht natürlich so gut wie alles aus Dingen, die irgendwie miteinander in Beziehung stehen. Graphen sind darum ein sehr universelles, unspezifisches Datenmodell. Der ultimative große Hammer der Datenmodellierung, den man dann herausholt, wenn ein Sachverhalt nicht mit Listen oder Bäumen modelliert

werden kann.

Ganz so allgemein ist aber auch ein Graph nicht. Datenmodelle haben ihren Sinn darin, der Wirklichkeit eine Struktur zu geben, beziehungsweise die für eine bestimmte Anwendung relevante Struktur der realen Verhältnisse offenzulegen. Dazu zwängen sie die realen Verhältnisse in ihr Schema. Das passt natürlich mal gut und auch mal weniger gut.

In einem Graphen gibt es zwei Arten von Dingen: *Knoten* (engl. *node* oder häufiger *vertex*) und *Kanten* (engl. *edge*). In einer grafischen Darstellung eines Graphen zeigt man die Knoten als Punkte oder Kreise, die durch Striche (ungerichteter Graph) oder Pfeile (gerichteter Graph) – die Kanten – verbunden sind. Die Knoten und Kanten modellieren irgendwelche Dinge sowie die Beziehungen zwischen ihnen. Dabei stehen immer genau zwei Dinge in einer Beziehung. Will man zum Ausdruck bringen, dass die durch eine Kante verbundenen Dinge eine unterschiedliche Rolle in der Beziehung spielen, dann nimmt man gerichtete Kanten. Sind die Dinge in so einer Zweierbeziehung gleichwertig, dann stellt man sie durch eine ungerichtete Kante dar. Eine Kante kann niemals mehr als zwei Knoten verbinden.

Städte, die jeweils durch eine Straße verbunden sind, können so direkt als Graph modelliert werden. Eine Autobahn als Verbindung aller Städte mit einer Autobahnausfahrt ist *keine* Kante und das Ganze dann auch *kein* Graph. So ganz allgemein ist der Graph also auch wieder nicht. Die Beschränkungen haben ihren Sinn: Man hat klare Rahmenbedingungen, innerhalb derer Probleme modelliert und dann mit Standardalgorithmen gelöst werden können. Für Graphen gibt es viele Algorithmen, die darum in vielen Situationen anwendbar sind.



Wenn das Wesentliche der für ein Problem relevanten realen Gegebenheiten in Form von Objekten (Knoten) und Zweierbeziehungen (Kanten) dargestellt werden kann, dann verwende eine *Graph*-Modellierung:

- ✔ Modelliere die Verhältnisse als Graph.
- ✔ Formuliere das Problem als Graphenproblem und löse es mit einem Graphenagorithmus.
- ✔ Übersetze die Lösung zurück in die realen Gegebenheiten.

Zeit für ein Beispiel.

Im Irrgarten der sozialen Netzwerke

Die jüngsten noch lebenden Vampire (siehe Abbildung 6.2) sind zwar schon über 100 Jahre alt, aber der Zeitgeist ist nicht völlig an ihnen vorbeigegangen. Unbemerkt von

gewöhnlichen Menschen betreiben sie ihre eigenen sozialen Netzwerke wie *VampDiaries*, in denen sie bestens miteinander vernetzt sind. Für etliche Unternehmen ist es nun von Interesse zu analysieren, was in diesen Netzwerken so vorgeht. Um eine Werbekampagne zu lancieren, ist es beispielsweise interessant herauszufinden, wer die wichtigsten *Influencer* sind, welche die Meinung über ein bestimmtes Produkt, einen angesagten Stehkragen etwa oder den neusten Lifestyle-Drink, maßgeblich beeinflussen.



Abbildung 6.2: Hildegard die Durstige mit Blutorangen-Karotten-Schorle

Die Firma *Vampbridge Analytica* hilft hier. Schauen wir ihr mal bei der Arbeit zu. Zuerst werden die Daten beschafft. Befreundete Firmen helfen dabei und schnell liegen die entscheidenden Informationen in Tabellenform vor. Jede Zeile enthält einen Vampir und seine *Follower*. Ein kleiner Ausschnitt ist

Vampir	***wird gefolgt von***
Brunhilde von Fledderzahn	Rüdiger
Vlad, Graf von Dracula	Olga; Hildegard

Olga, Fräulein von Seifenschwein	Anna; Hildegard
Hildegard die Durstige	
Rüdiger von Schlotterstein	
Anna von Schlotterstein	Rüdiger; Hildegard

Es ist schnell klar, dass ein Graph das geeignete Datenmodell ist. Kurz stand die Idee im Raum, dass ein Baum geeignet sei: Vampire und ihre *Follower*, das ist auf den ersten Blick eine hierarchische Beziehung. Die grafische Darstellung in Abbildung 6.3 zeigt aber sofort, dass dem nicht so ist. Ein Graph ist das geeignete Datenmodell für die Gegebenheit. Es passt auch zur realen Problemstellung, wenn diese als Graphenproblem formuliert werden kann.

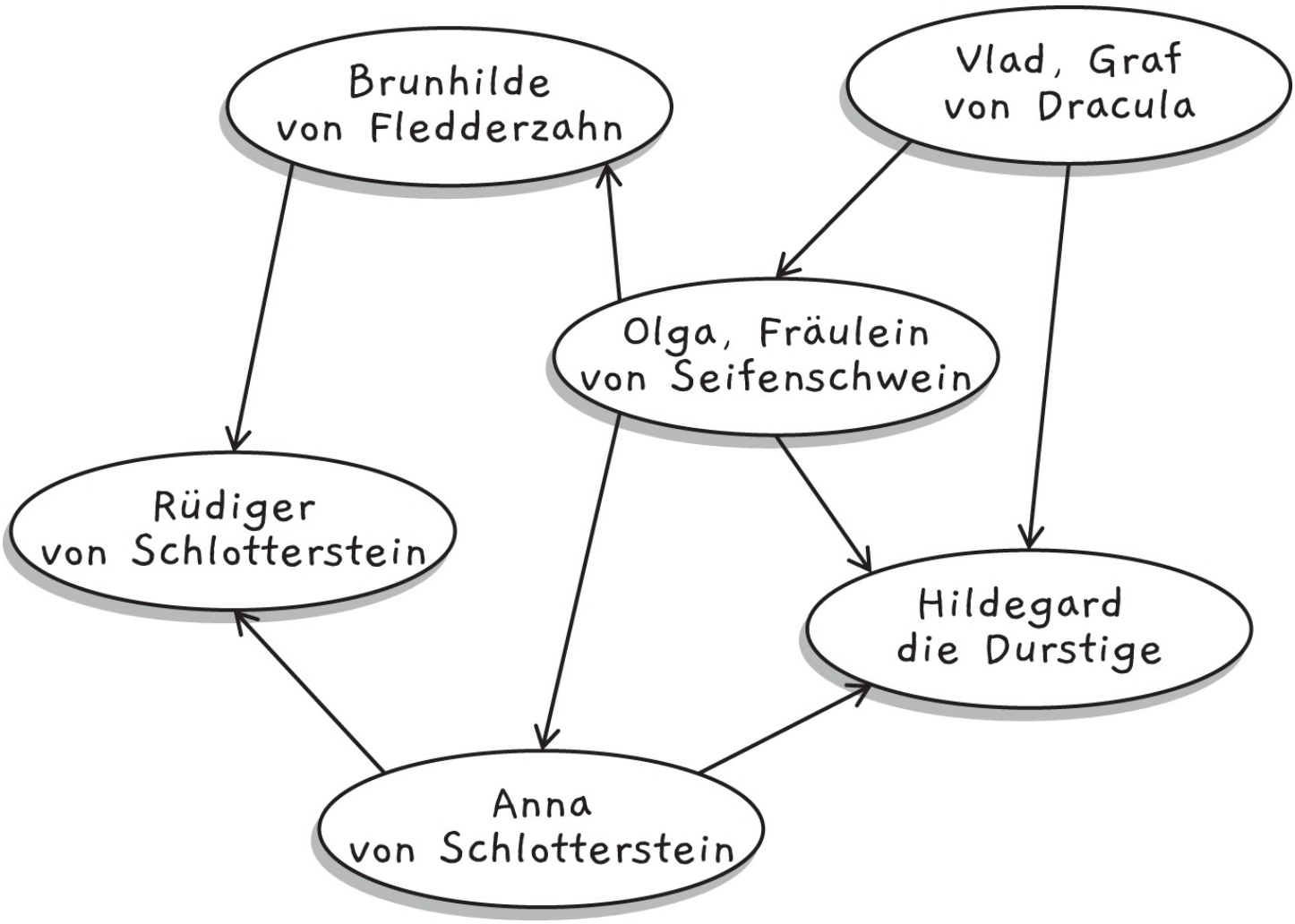


Abbildung 6.3: Einige Vampire und ihre *Follower*

Das originale Problem ist:

Problem: wichtigster Influencer

Eingabe: eine Liste mit Vampiren und ihren Followern

Berechne: den Vampir mit den meisten direkten und indirekten Followern

Bevor das Ganze in die Welt der Graphen transferiert werden kann, sollten wir uns zunächst ein wenig mit dieser Welt vertraut machen.

Ein kurzer Blick in die Welt der Graphen

Man unterscheidet *gerichtete* und *ungerichtete* Graphen. Je nachdem, ob die Kanten eine Richtung haben oder nicht. Beide werden als mathematisches Konzept beschrieben.

Gerichtete und ungerichtete Graphen



Ein *gerichteter Graph* $G(V, E)$ besteht aus

- einer Menge V (von engl. *Vertex*) von Knoten und
- einer Menge E (von engl. *Edge*) von Kanten; E ist dabei immer eine *zweistellige Relation* auf den Knoten.

Eine zweistellige Relation ist eine Menge von Paaren. E ist also eine Menge, deren Elemente Paare von Knoten sind, und eine Kante ist ein Paar (v_1, v_2). v_1 ist Startknoten der Kante und v_2 deren Zielknoten.

Ein Graph kann gut in einem Diagramm dargestellt werden. In Abbildung 6.4 sehen wir die Darstellung von $G(V, E)$ mit

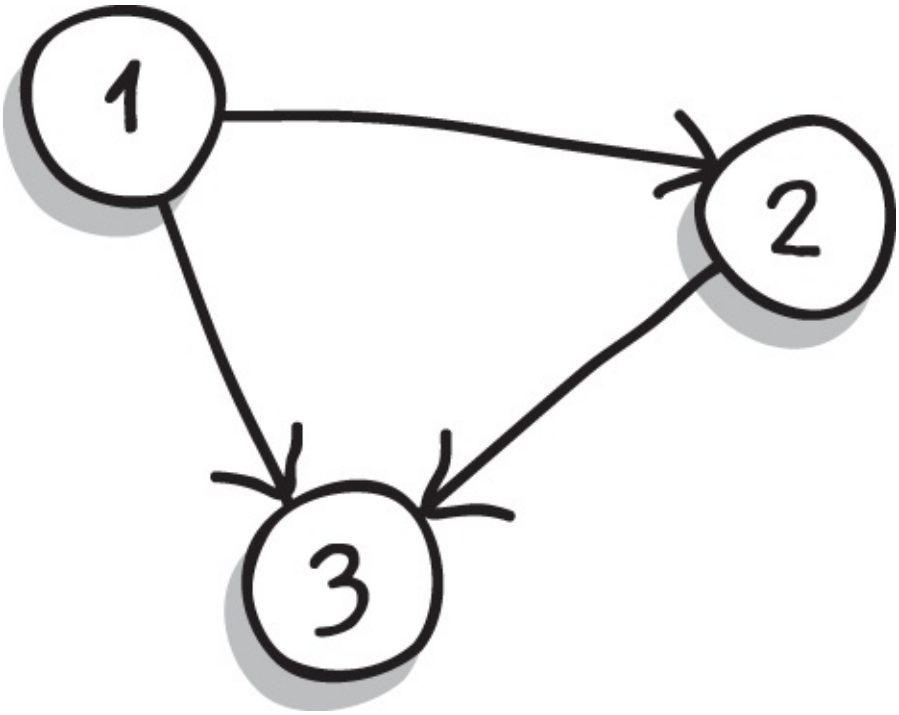


Abbildung 6.4: Ein gerichteter Graph

$$V = \{1, 2, 3\}, \quad E = \{(1, 2), (1, 3), (2, 3)\})$$

Die Knoten sind eine beliebige Menge und die Kanten können beliebige Beziehungen zwischen den Knoten darstellen. Bei gerichteten Graphen wird eine Kante als Paar (s, e)

von zwei Knoten beschrieben. Der erste, s, ist der Startknoten der Kante, der zweite, e, ist deren Endknoten. Eine gerichtete Kante zwischen zwei Knoten sagt also nichts anderes, als dass es eine Beziehung gibt und dass die beiden beteiligten Knoten eine unterschiedliche Rolle in dieser Beziehung spielen.

Manchmal modelliert eine Kante in einem Graphen eine Beziehung, bei der es keinen Sinn macht, die beteiligten Knoten zu unterscheiden. Straßen kann man beispielsweise in der Regel in zwei Richtungen befahren. Will ich einen Weg in einem Gebiet suchen, in dem es keine Einbahnstraßen gibt, dann kann die Wegfindung als Problem in einem Graphen modelliert werden, in dem Kanten keine Richtung haben.

Natürlich kann man solche ungerichteten Kanten immer auch als zwei gegensätzlich gerichtete Kanten darstellen. Viele Algorithmen sind aber einfacher, wenn grundsätzlich jede Kante ungerichtet ist und dies nicht noch extra modelliert werden muss.

Ungerichtete Graphen werden mathematisch durch zwei Mengen beschrieben: die Menge der Knoten plus die Menge der Kanten. Mathematisch gesehen ist also auch ein ungerichteter Graph ein Paar von Mengen.


Ein *ungerichteter Graph* $G(V, E)$ besteht aus

- einer Menge V (von engl. *Vertex*) von Knoten und
- einer Menge E (von engl. *Edge*) von Kanten; E ist dabei eine Menge von *zweielementigen Mengen* von Knoten.

Der ungerichtete Graph von Abbildung 6.5 kann mathematisch dargestellt werden als $G(V, E)$ mit

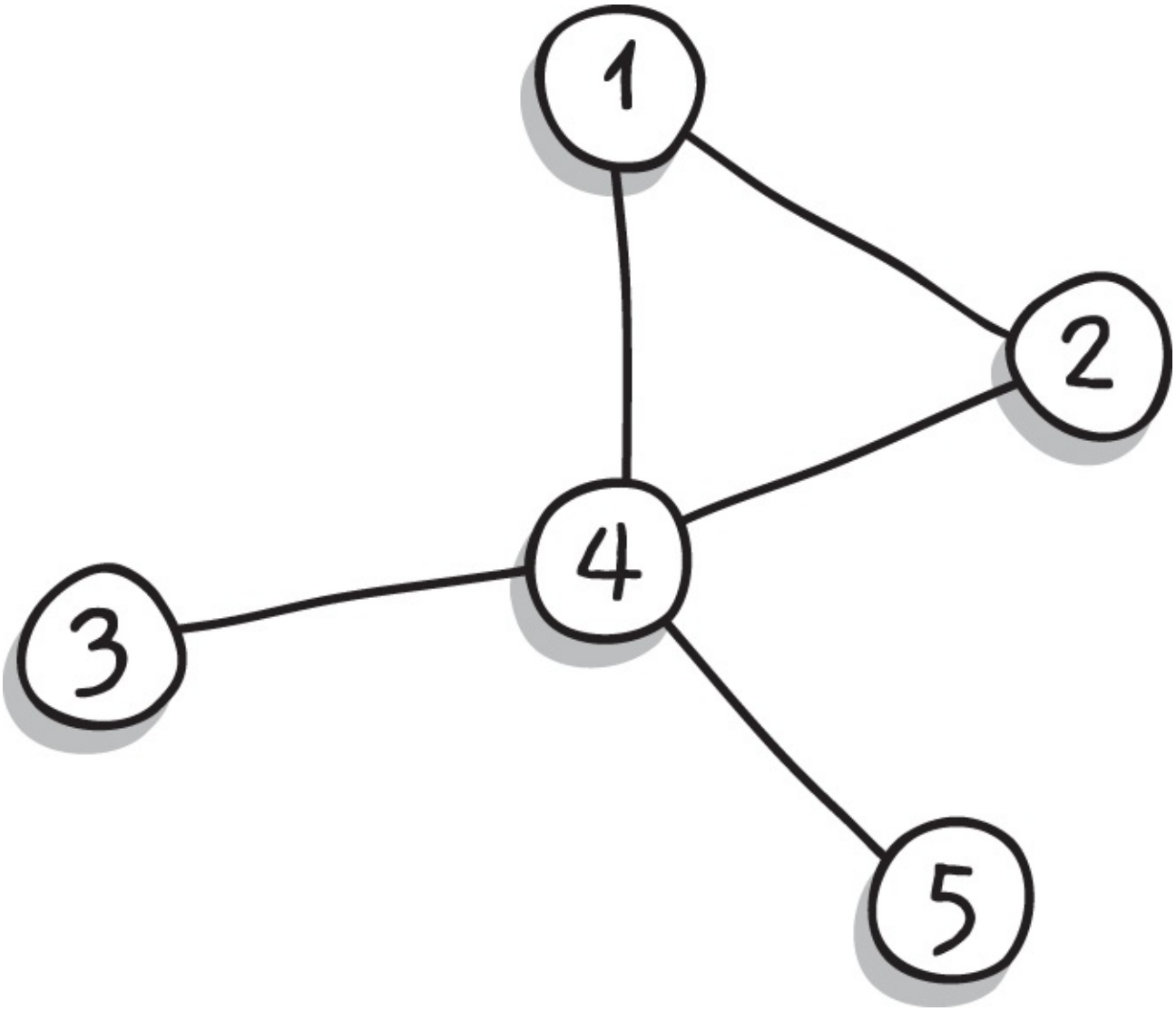


Abbildung 6.5: Ein ungerichteter Graph

$$V = \{1, 2, 3, 4, 5\}, \quad E = \{\{1, 2\}, \{1, 4\}, \{2, 4\}, \{3, 4\}, \{4, 5\}\}$$

Die vielleicht etwas verwirrende Beschreibung von Kanten als zweielementige Mengen hat ihren Grund darin, dass einen Menge wie $\{1, 2\}$ zum Ausdruck bringt, dass 1 und 2 dabei sind und sonst nichts. Also vor allem ist 1 nicht das erste Element der Menge und 2 nicht das zweite. In einer Menge gibt es kein Erstes oder Zweites, alle sind gleich:

$$\{1, 2\} = \{2, 1\}$$

Bei gerichteten Graphen sind Kanten Paare. Bei diesen gibt es ein erstes und ein zweites Element und es ist

$$(1, 2) \neq (2, 1)$$

daher kann man mit ihnen eine Richtung ausdrücken.

Manchmal ist die Darstellung der Kanten als Paare etwas lästig. Man kann die Menge der Kanten dann als symmetrische Relation darstellen: Wo es eine Vorwärtsrichtung gibt, muss es auch eine Rückrichtung geben. Die Darstellung *Kanten sind Mengen*

(beziehungsweise E ist eine Menge von Mengen), also

$$E = \{\{1,2\},\{1,4\},\{2,4\},\{3,4\},\{4,5\}\}$$

entspricht dann

$$E = \{(1,2),(2,1),(1,4),(4,1),(2,4),(4,2),(3,4),(4,3),(4,5),(5,4)\}$$

Das wäre die Modellierung der Menge der Kanten als symmetrische Relation.

Markierungen

Knoten und Kanten werden oft weitere Informationen zugeordnet. Sowohl Knoten als auch Kanten können eine *Markierung* (ein *Label*) tragen. Markierungen können von beliebigem Typ sein. In der Darstellung schreibt man sie neben den Knoten oder die Kante. Das geht natürlich nur, wenn die Markierung eine textuelle Darstellung hat.

Markierungen dürfen nicht mit den Knoten oder Kanten verwechselt werden. In Abbildung 6.6 hat die Kante $(2,3)$ die Markierung liebt. Knoten 2 ist mit Klaus markiert. 2 ist nicht die Markierung, 2 ist ein Knoten. Natürlich könnte Knoten 2 auch mit der Markierung »2« markiert sein. Markierte Knoten sind weniger üblich als markierte Kanten.

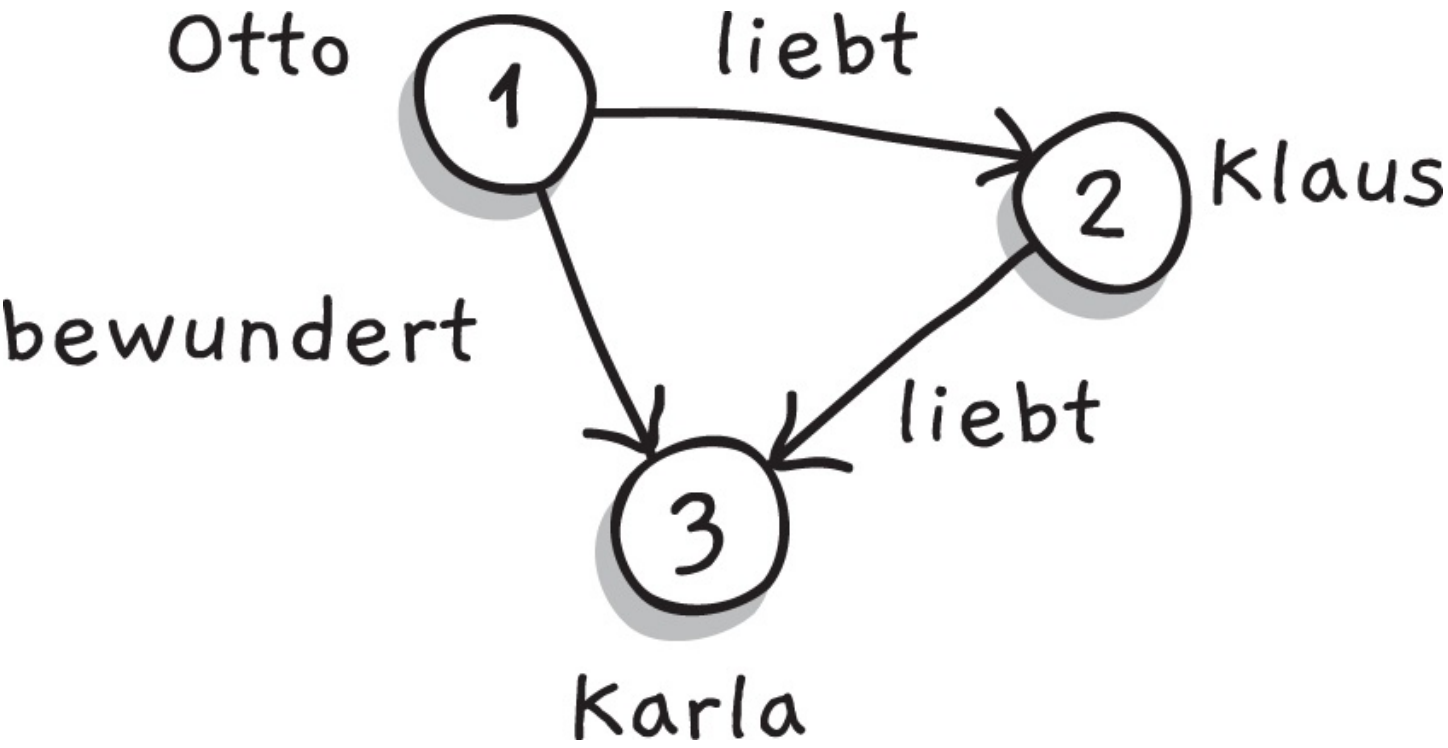


Abbildung 6.6: Ein gerichteter Graph mit Markierungen

Markierungen repräsentieren wichtige Informationen aus der realen Welt, nämlich die Daten, die mit dem Graphen repräsentiert werden. Um was es sich da genau handelt, ist in den allgemeinen Graphenalgorithmen nicht wichtig. Der allgemeine Algorithmus zur Wegfindung sucht beispielsweise den kürzesten Weg zwischen zwei Knoten. Dabei ist es völlig egal, ob die Knoten Städte oder Personen darstellen und die Kanten Straßen oder

persönliche Beziehungen.

Auch wenn es für die Algorithmen egal ist, was eine Markierung bedeutet, so muss sie doch oft vorhanden sein. Messbare Größen als Kantenmarkierungen sind oft wesentlich für die Algorithmen. Wenn man den kürzesten Weg berechnen will, dann spielen Entfernungen zwischen verbundenen Knoten eine wichtige Rolle. Für eine Partnervermittlung ist vielleicht Intensität von Gefühlen eine relevante Größe. Man definiert so etwas als Markierung der Kanten und nennt es dann Kanten-*Gewicht* oder -*Kosten*.

Pfade

Ein *Pfad* ist ein Weg von einem Knoten zu einem anderen Knoten, der über die Kanten führt. Natürlich in der richtigen Richtung. Ein Pfad ist eine Folge von Kanten. Da jede Kante in einem Pfad dort beginnt, wo ihr Vorgänger endet, kann man Pfade als Knotenliste darstellen. Der Pfad von 1 nach 3 im Graph von Abbildung 6.6 geht über die Kanten $(1, 2)$ und $(2, 3)$ und kann als Liste $[1, 2, 3]$ dargestellt werden.

Die Länge eines Pfades ist die Anzahl der Kanten, über die er geht. Der Pfad $[1, 2, 3]$ hat also die Länge 2: Er geht über zwei Kanten.

Eine Kante darf von einem Knoten zum gleichen Knoten führen. Man nennt solche Kanten *Schleifen*. Ein *Zykel* ist ein Pfad, der mit dem gleichen Knoten beginnt und endet. Schleifen sind also Zykel der Länge 1. Ein Pfad ist *zyklisch*, wenn er ein Zykel ist. Ansonsten ist er *azyklisch*.

Ein Zykel ist *einfach*, wenn kein Knoten außer dem Startknoten doppelt vorkommt und der Startknoten genau zweimal: am Anfang und am Ende. Ein einfacher Zykel ist also ein Zykel, der nicht zweimal über den gleichen Knoten geht. Nicht jeder Graph enthält Zykel. Im Graph von Abbildung 6.6 gibt es beispielsweise keine Zykel.

Einen Graphen ohne Zykel nennt man *azyklisch*. Wenn ein Graph Zykel enthält, dann enthält er auch mindesten einen einfachen Zykel. Man kann jeden Zykel in einen einfachen Zykel mit dem gleichen Start-/Endknoten transformieren. Ein azyklischer Graph enthält naturgemäß keine zyklischen Pfade.

Einflussnahme als Graphenproblem

Die Bestimmung des wichtigsten *Influencers* kann jetzt als ein Graphenproblem formuliert werden: Finde den Knoten mit der größten Erreichbarkeitsmenge.

> **Problem:** Erreichbarkeit
>
> **Eingabe:** ein gerichteter Graph $G(V, E)$ und ein Element $x \in V$
>
> **Berechne:** die Menge $R(x)$ aller von s aus erreichbaren Knoten; ein Knoten y ist von x aus erreichbar, wenn es in G einen Pfad p gibt mit x als erstem und y als letztem Knoten

Das Problem der einflussreichsten Vampire kann als Optimierungsproblem der Erreichbarkeit formuliert werden:

Problem: Maximum der Erreichbarkeit

Eingabe: ein gerichteter Graph $G(V, E)$

Berechne: das Element $x \in V$ mit der größten Menge $R(x)$ an Knoten, die von x aus erreichbar sind

Damit hat man auch schon eine erste Idee für den Algorithmus: Berechne $R(x)$ für jeden Knoten x und wähle das x mit dem größten $R(x)$. Es fehlt nur noch ein Algorithmus für die Berechnung von $R(x)$.

Glücklicherweise lebten wichtige Mitarbeiter von *Vampbridge Analytica* schon in früheren Zeiten und erinnern sich noch an die damals grassierende Mode der Labyrinthe und Irrgärten. Schon damals konnten sie mit wissenschaftlicher Beratung Erfolge einfahren. Ein Labyrinth kann als Graph dargestellt werden. Man modelliert dabei jede Stelle, an der man sich für eine bestimmte Richtung entscheiden muss, als Knoten und die Wege zwischen diesen Entscheidungsstellen als Kanten (siehe Abbildung 6.7). Mithilfe dieser Modellierung konnten sie systematische Verfahren entwickeln, mit denen sich ein Ausgang aus einem Irrgarten finden lässt.

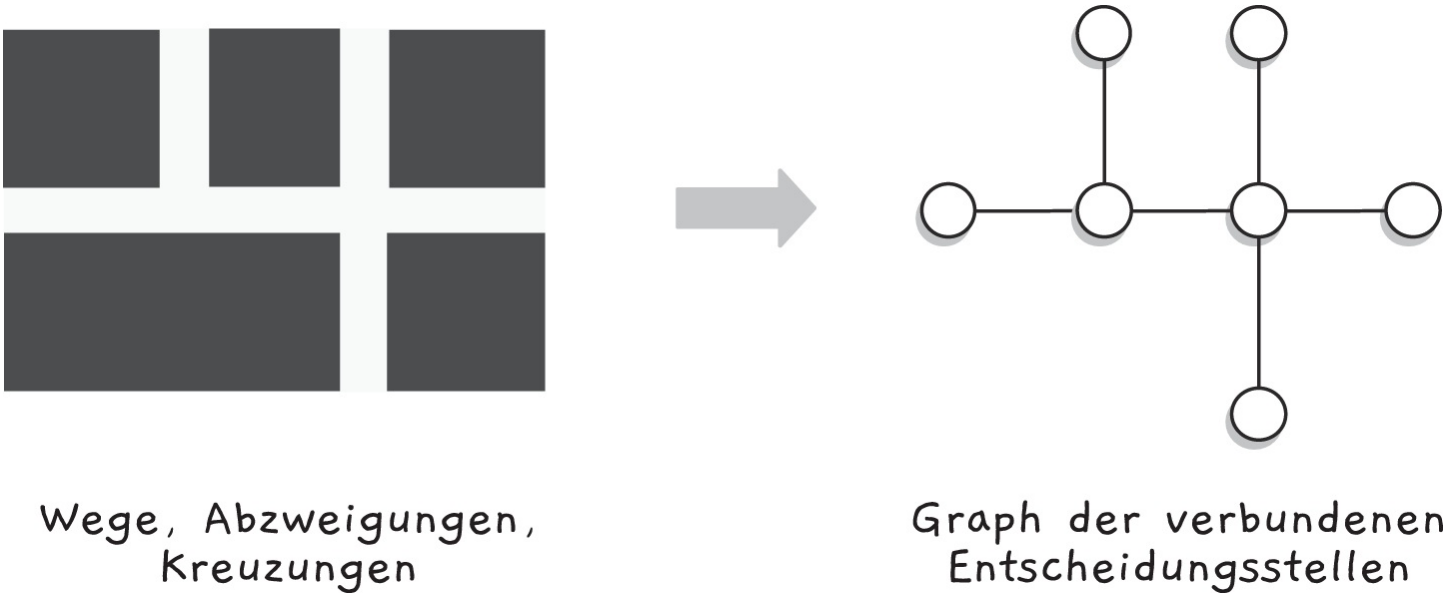


Abbildung 6.7: Wege im Labyrinth als Graph

Die entscheidende Idee ist recht einfach. Wenn man überallhin geht und dabei vermeidet, im Kreis zu laufen, dann wird man zum Ausgang kommen. Das »gehe überallhin« haben wir im letzten Kapitel vornehm »Traversierung« genannt. Die Lösung des Problems »Traversierung« findet ein Traversierungsalgorithmus. Mit einem Traversierungsalgorithmus, der die Menge der von einem Knoten aus erreichbaren anderen Knoten ausspuckt, kann das Problem der maximalen Erreichbarkeit und damit

das des maximalen Einflusses berechnet werden.

MaxReachable$(G(V,E))$
$R \leftarrow \{(x, size_x) \mid x \in V,\ size_x = |\ \textbf{Reachable}\ (G,x)\ |\ \}$ *Erreichbarkeitmengen*
$(r, size_r) \leftarrow max(R)$ mit der zweiten Komponente der Paare in R
als Vergleichskriterium *vergleiche die Größen der Mengen*
return r *Knoten mit der größten Erreichbarkeitsmenge*

Die Menge R besteht aus Paaren $(x, size_x)$ wobei x ein Knoten ist und $size_x$ die Zahl der Knoten, die von diesem Knoten aus erreichbar sind. Da nimmt man dann den Startknoten r des Paares $r, size_r$ mit der größten $size$. ($|$ Reachable (G,x) $|$ ist die Mächtigkeit dieser Menge, das heißt die Zahl der Elemente in Reachable (G,x).)

Es bleibt nur noch die Berechnung der Erreichbarkeitsmengen. Wir können sie mit einer Traversierung des Graphen finden, die bei einem Knoten x startet:

Reachable$(G(V,E), x \in V)$
return eine Traversierung von G mit Start bei x

Das ist natürlich erst einmal nur die Idee für einen Algorithmus. Seine wichtigsten Elemente sind:

- ✔ Traversiere den Graphen mit jedem Knoten als Startpunkt und sammle dabei die erreichten Knoten auf.
- ✔ Bestimme die Größe der Menge der jeweils erreichbaren Knoten.
- ✔ Gib den Knoten zurück, dessen Menge die größte ist.

Graphen traversieren

Bei Bäumen hatten wir eine äquivalente Problemstellung betrachtet: Starte an der Wurzel und gehe zu jeden Knoten. Entweder mit einer Tiefen- oder mit einer Breitentraversierung. Oft auch Tiefen- oder Breitensuche genannt, auch wenn gar nichts gesucht wird. Eine Tradition aus der Zeit, als der Ausgang aus einem Irrgarten gesucht wurde. Jetzt starten wir bei x und laufen zu allen Knoten, die von x aus erreichbar sind.

Die einfachste Traversierungsmethode ist die Tiefentraversierung. Bei einem Baum wird die Traversierungsfunktion einfach für jeden Nachfolger aufgerufen: strukturelle Rekursion, Funktion folgt Form. Bei einem Graphen gibt es da zwei kleine Probleme:

- ✔ Man kann im Kreis herumlaufen und immer wieder zu gleichen Stelle kommen, ohne es zu merken.
- ✔ Es gibt nicht so etwas wie eine Wurzel, also einen Knoten, von dem aus alle anderen garantiert erreicht werden können.

Das Problem der Zykel kann mit einem Stück Kreide gelöst werden: Jeder besuchte Knoten wird markiert und ignoriert, wenn wir wieder auf ihn treffen. Gegen das Problem der Nichterreichbarkeit hilft nur, dass die Traversierung systematisch von allen Knoten aus gestartet wird.

Im Graphen von Abbildung 6.8 können beispielsweise von a aus alle Knoten erreicht werden. Von jedem anderen Knoten gibt es aber keinen Pfad zu a. Im Allgemeinen muss die Traversierung darum von jedem Knoten aus gestartet werden. Der entsprechende Algorithmus lautet

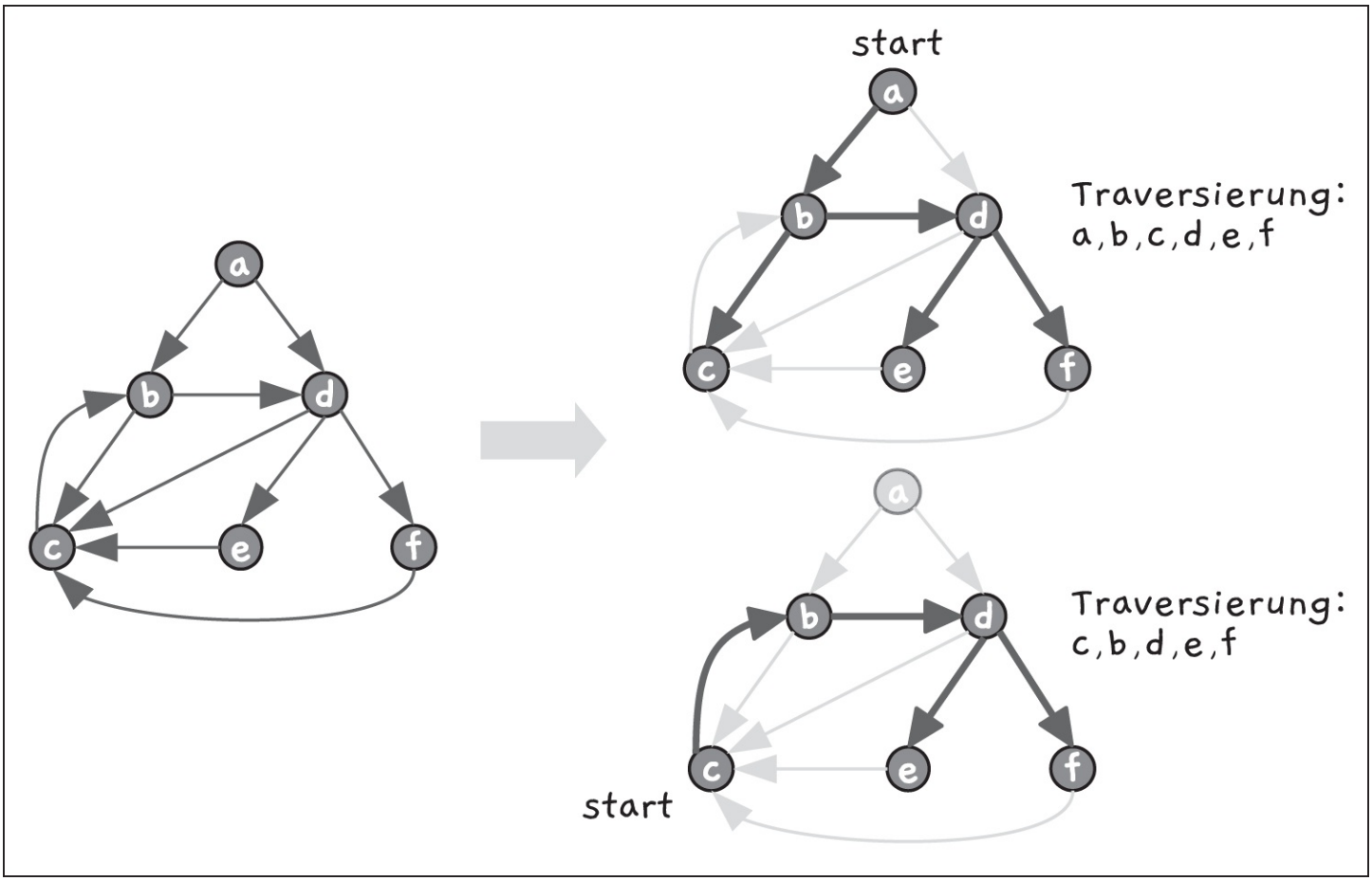


Abbildung 6.8: Tiefentraversierung in einem Graphen

```
Trav(G(V, E))                    traversiere den Graphen G
    for v ∈ V                    markiere erstmal alle als unbesucht
        v.mark ← unvisited
    for v ∈ V                    traversiere die noch nicht besuchten – Achtung, imperativ:
        if v.mark = unvisited    die Markierungen ändern sich während eines Laufes
            Reach(G, v)
```

```
Reach(G(V, E), v)                    besuche alle von v aus erreichbaren Knoten in G
    v.mark ← visited
    for w ← {w | (v, w) ∈ E}         w: Ziel einer von v ausgehenden Kante
        if w.mark = unvisited
            Reach(G, w)
```

Die Traversierung besteht in der Reihenfolge, in der die Knoten mit »visited« markiert werden.

Nicht so schwer. Vor allem dank der Rekursion, die heute möglich ist. Mit Schrecken denken die Spezialisten zurück an die Zeiten, als die Kollegen sich im 19. Jahrhundert mit Algorithmen für Irrgärten abmühen mussten, ohne auf rekursive Funktionen zurückgreifen zu können.

Die Erreichbarkeit, die für das Problem des einflussreichsten Vampirs gebraucht wird, ist also lediglich eine Hilfsfunktion der Traversierung. Damit kann man sich dem gesuchten Algorithmus schon etwas weiter annähern. In Trav und Reach werden die Knoten einfach nur besucht. Es passiert sonst weiter nichts. Das ist nur das Grundgerüst eines Algorithmus. Reach wird gebraucht und muss etwas liefern: die besuchten Knoten. Der Algorithmus kann um diese Funktionalität erweitert werden:

```
Reach(G(V, E), v)                    berechne (!) alle von v aus erreichbaren Knoten in G
    v.mark ← visited
    result ← {v}                     jeder Knoten ist von sich selbst aus erreichbar
    for w ← {w | (v, w) ∈ E}         w: ein direkter Nachbar
        if w.mark = unvisited
            result ← result ∪ Reach(G, w)      gehe zu den unmarkierten Nachbarn
    return result
```

Bei der Traversierung und der Erreichbarkeit werden markierte und unmarkierte Knoten unterschieden. Am Anfang setzt Trav dazu alle Knoten auf unmarkiert:

```
for v ∈ V
    v.mark ← unvisited
```

Das kann man so wörtlich nehmen, muss man aber nicht. Die Knoten sind zunächst einmal schon alle unmarkiert und statt eine Markierung explizit zu setzen, können die markierten Knoten auch in einer Menge verwaltet werden. Diese Mengen können in einer globalen Variablen M verwaltet werden. Die Traversierung insgesamt wird dann zu Folgendem:

```
M ← ∅                                   markierte Knoten

Trav(G(V, E))                           traversiere den Graphen G
    for v ∈ V
        if v ∉ M
            Reach(G, v)

Reach(G(V, E), v)                       berechne alle von v aus erreichbaren Knoten in G
    M ← M ∪ {v}
    result ← {v}                        jeder Knoten ist von sich selbst aus erreichbar
    for w ← {w | (v, w) ∈ E}
        if w ∉ M                        w ist ein unmarkierter direkter Nachbar von v
            result ← result ∪ Reach(G, w)   gehe zu den unmarkierten Nachbarn
    return result
```

Damit ist der Algorithmus Reach, an dem *Vampbridge Analytica* interessiert ist, etwas konkreter geworden, er bewegt sich aber immer noch auf einer recht abstrakten mathematischen Ebene. Das erkennt man sofort an den geschweiften Klammern und den Mengensymbolen. Der Algorithmus operiert auf Mengen. Klar, viele moderne Sprachen bieten »Mengen« an. Das ist dann eine *Datenabstraktion*, hinter der eine echte Datenstruktur steht, wahrscheinlich ein Baum oder eine Hashtabelle. Mithilfe der Programmiersprachen wird dann diese Datenstruktur verwendet. Für Testzwecke mag das ganz Ok sein, aber so sind die Mengen in einem solchen »abstrakten« Algorithmus *nicht* (unbedingt) gemeint!



$\{,\},\emptyset$: Huch – Mathe! Algorithmen werden oft in Form von Aktionen auf Mengen definiert. Die Verwendung einer *Menge* bedeutet aber in der Regel *nicht*, dass die Algorithmen tatsächlich auf einer Datenstruktur »Menge« agieren sollen. Die Mengen stehen für bestimmte Informationen, bei denen (noch) *offengelassen* wurde, in welcher *Datenstruktur* sie gespeichert werden.

Es ist oft sinnvoll, die Wahl der Datenstrukturen aus einem Algorithmus herauszuhalten. Der Algorithmus soll zum einen möglichst allgemein verwendbar sein und Datenstrukturen haben schon etwas mit konkreten Daten zu tun. Zum anderen wird die Wahl der Datenstruktur sinnvollerweise auch stark von dem beeinflusst, was die Implementierungssprache zu bieten hat.

Gelegentlich wird die Wahl der Datenstruktur auch in die Kategorie der trivialen Implementierungsdetails eingestuft. So sieht man das auch bei *Vampbridge Analytica*. Dort wird der Algorithmus

```
MaxReachable(G(V, E))
    R ← {(x, size_x) | x ∈ V, size_x = |Reach(G, x)| }      Erreichbarkeitsmengen
    (r, size_r) ← max(R) mit der zweiten Komponente der Paare in R
                    als Vergleichskriterium      vergleiche die Größen der Mengen
    return r                                     Knoten mit der größten Erreichbarkeitsmenge
```

zusammen mit Reach an den jüngsten Mitarbeiter, Hans-Heinrich Geiermeier, übergeben. Geiermeier ist ein ehemaliger Friedhofsgärtner, der sich erst vor etwa 60 Jahren beim Versuch, einen Vampir ans Tageslicht zu holen, seinen Biss einfing (da war er gerade einmal 58 Jahre alt). Als jüngster ist Geiermeir für alle niederen Arbeiten zuständig. Arbeiten wie beispielsweise die Auswahl von konkreten Datenstrukturen zu Algorithmen fallen in sein Ressort. In höheren Gehaltsstufen beschäftigt man sich ausschließlich mit mathematischen Objekten. Schauen wir also Herrn Geiermeier bei der Arbeit zu.

Als erstes lassen wir uns von ihm über das Notwendigste zur Implementierung von Graphen in Form von Datenstrukturen aufklären.

Datenstrukturen für Graphen

Graphen könnte man in einer Graph-artigen Datenstruktur speichern, die aus Knoten in Form von Speicherzellen besteht. Die Zellen sind über Zeiger verknüpft, die die Rolle der Kanten übernehmen. Wenn die Kanten Markierungen tragen, dann müssten diese in den Knoten gespeichert werden. Einer Kante als Zeiger kann man ja keine Informationen anhängen. Der Graph von Abbildung 6.6 beispielsweise könnte so gespeichert werden wie in Abbildung 6.9 dargestellt. Das ist eine Idee zur Speicherung eines Graphen, das heißt für die Wahl einer Datenstruktur zur Speicherung von Graphen. Allerdings ist so etwas nicht immer die am besten geeignete Datenstruktur.

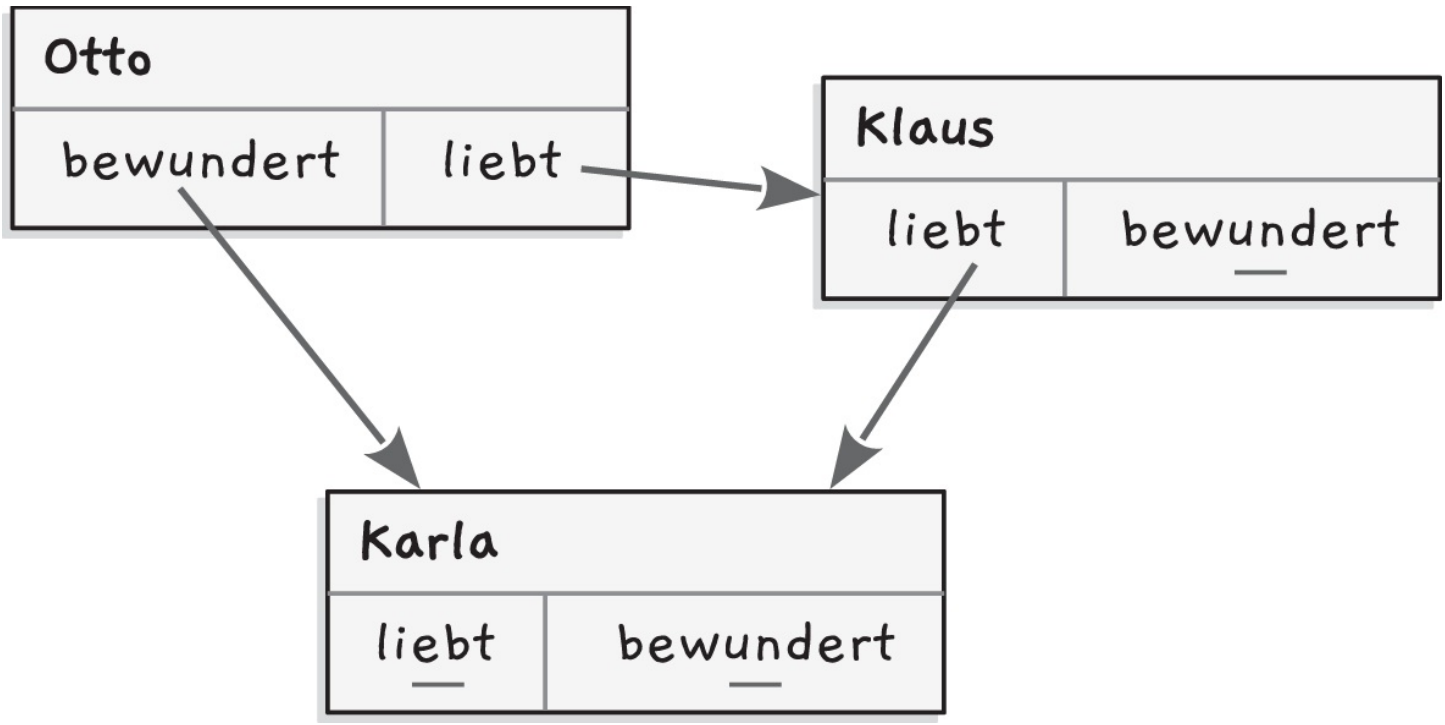


Abbildung 6.9: Ein Graph als nachgebaute Datenstruktur (meistens Unsinn)

Zunächst einmal können wir nur die Daten speichern, die wir auch haben. Vielleicht haben wir in unseren Daten eine Liste aller Knoten, oder genauer gesagt, eine Liste all der Objekte, die die Rolle der Knoten übernehmen sollen. Vielleicht gibt es eine solche Liste aber auch nicht und die Knoten können nur mühsam über die Verbindungsdaten rekonstruiert werden. Eine Aktion, die eventuell bei der geplanten Anwendung völlig unnötig ist.

Bei der Suche nach dem wichtigsten *Influencer* in der Scene der Vampire stehen Geiermeier die Daten von *VampDiaries* zur Verfügung, und zwar in Form einer Liste mit Vampiren und ihren *Followern*. Im Graph-Modell sind das alle Kanten in Form einer Nachbarschaftsbeziehung. Es ist oft so, dass die Information über die Knoten und Kanten in der Form »Knoten und seine Nachfolger« vorliegen. Es liegt dann nahe, den Graphen in einer Datenstruktur zu speichern, die diese Nachbarschaftsbeziehung widerspiegelt. Vielleicht muss das dann noch geändert werden, weil diese Datenstruktur sich für die Aktionen des Algorithmus als ungeeignet herausstellt.

 Die *Datenstruktur* für einen *Graphen* richtet sich nach den

- ✔ Daten, die als Graph modelliert werden sollen, und den
- ✔ Aktionen, die die Graph-Algorithmen auf diesen Daten ausführen.

Nachbarschaften dokumentieren

Die Darstellung von Graphen im Speicher mit Zeigern nachzubauen, ist hier also keine gute Idee. Geiermeier lässt die Daten in der Form, in der sie zu ihm kommen, das heißt als

Nachbarschaftsbeziehung. Jeder Knoten hat Nachbarn und die werden in einer Liste zusammengefasst und dem Knoten zugeordnet. Das ist im Falle der Vampire eine Abbildung

$$V \to \{F \mid F \subseteq V\}$$

V ist die Menge der Vampire (oder Knoten). Jedem Vampir ist die Menge seiner *Follower* zugeordnet, diese ist bekanntlich ein Element der Potenzmenge der Vampire (siehe Abbildung 6.10).

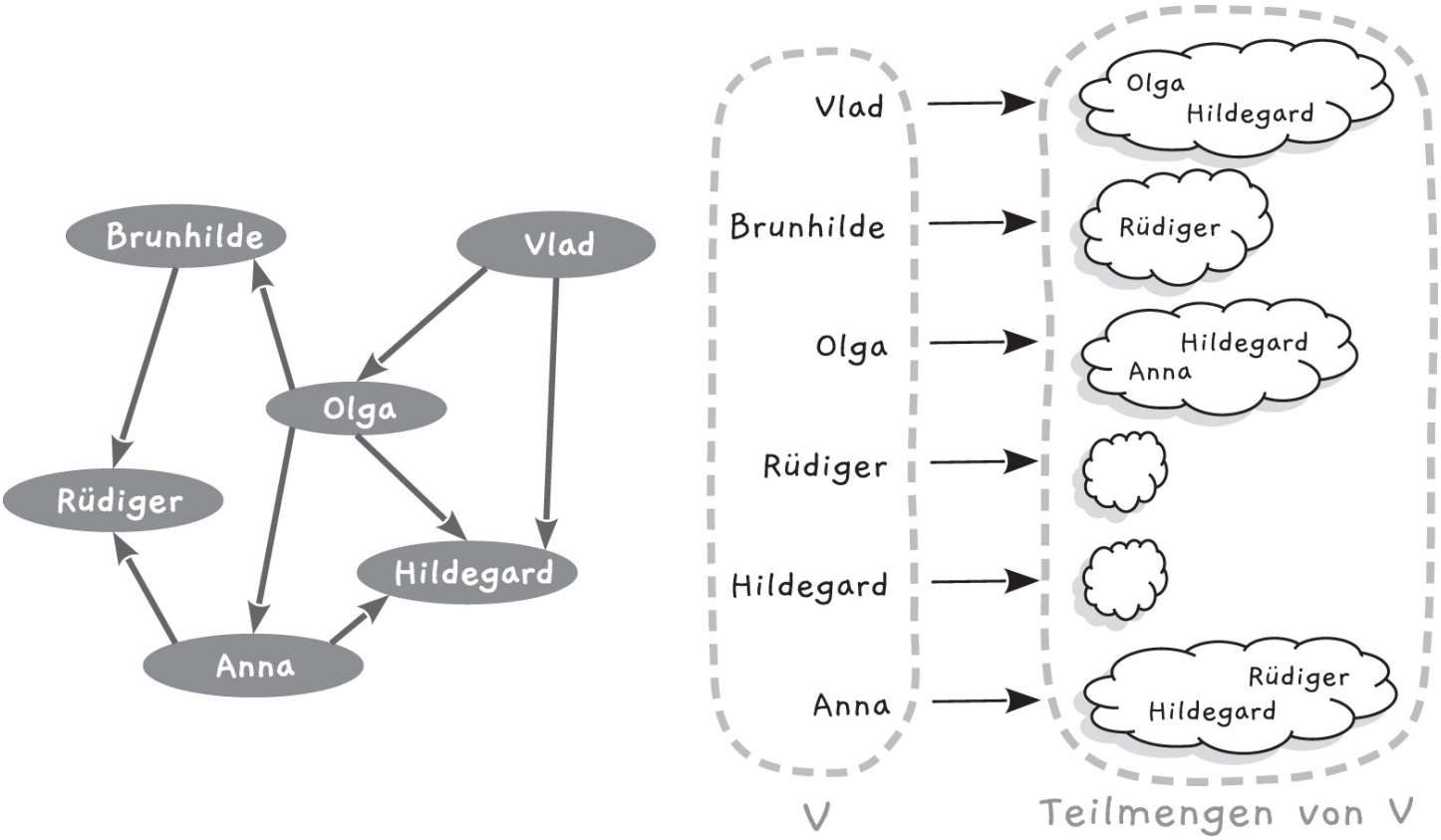


Abbildung 6.10: Ein Graph als Adjazenzen

Igitt, schon wieder Mengen. Mengen sind keine Datenstruktur, sondern ein Indiz dafür, dass es noch keine Idee für eine Datenstruktur gibt. (Es wurde noch keine Entscheidung getroffen, wie die Informationen in einer Datenstruktur zu speichern sind, die mit der mathematischen Notation beschriebenen werden.)

Die wesentliche Information über den Graphen wird also durch eine Abbildung beschrieben. Eine Abbildung, die die Nachbarschaftsbeziehungen beschreibt. Viele moderne Programmiersprachen bieten unter verschiedenen Namen (zum Beispiel *Map*, *assoziatives Array*) Abbildungen an. So etwas könnte man in einer Implementierung verwenden.

Das vornehmere Wort für Nachbarschaft ist *Adjazenz*. Übliche Speicherformen für Graphen sind die Adjazenzmatrix und Adjazenzlisten. Beide sind aber »eigentlich« nur zwei der üblichen Datenstrukturen für Abbildungen.



Speichere (die Kanten eines) Graphen als Abbildung! Graphen werden meist als *Adjazenzen* gespeichert, also als *Abbildung*, die jedem Knoten die Menge seiner Nachbarn zuordnet. Prinzipiell kann jede Datenstruktur verwendet werden, die für Abbildungen geeignet ist. Viele Programmiersprachen bieten solche Datenstrukturen als gebrauchsfertige Abbildungen an. Sie sind die erste Option bei der Implementierung von Graphen.

Trotz seines jugendlichen Alters ist Geiermeier aber von altem Schrot und Korn und entwickelt seine eigene Datenstruktur für diese Nachbarschaftsbeziehung. Vielleicht erinnert er sich aber auch nur an seine A&D-Vorlesung.



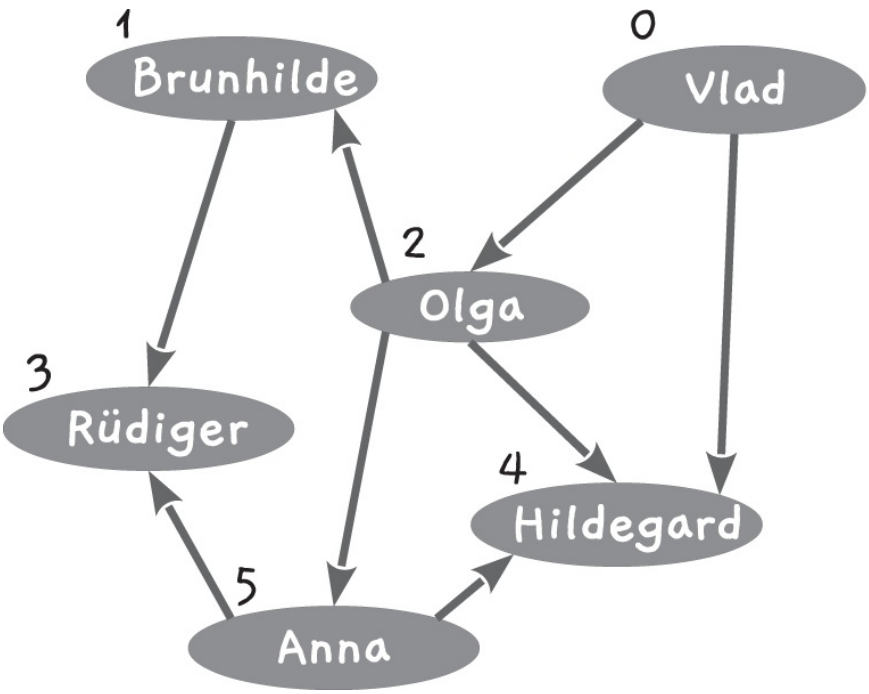
In Kapitel 5 haben wir gesehen, dass sich Bäume gut zur Speicherung von Mengen und Abbildungen eignen. In Kapitel 9 werden Hashtabellen besprochen, eine weitere oft verwendete Datenstruktur zur Speicherung von Mengen und Abbildungen.

Adjazenzlisten

Eine *Adjazenzliste* ist eine Liste von Nachbarn. Dies ist die naheliegendste Art, Adjazenzen (Nachbarschaften) zu speichern: Man speichert in einem Array, auf das mit Indizes vom Typ V zugegriffen werden kann, die Listen der Nachfolger.

a[v] = Liste der direkten Nachbarn von v

Wenn auf die Elemente eines Arrays nicht mit Indizes von beliebigem Typ zugegriffen werden kann, was heute die Regel ist, dann muss man indirekt über eine Nummerierung der Knoten gehen. Statt des Arrays kann natürlich auch eine Liste verwendet werden (siehe Abbildung 6.11).



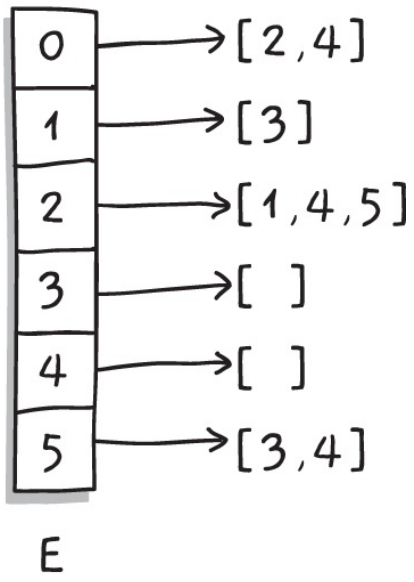


Abbildung 6.11: Ein Graph und seine Adjazenzlisten-Darstellung

Wenn ein ungerichteter Graph in dieser Art gespeichert werden soll, dann muss eine Kante zweimal eingetragen werden. Einmal in die eine und einmal in die andere Richtung. Manchmal will man Graphen definieren, bei denen zwei Knoten mit zwei Kanten in der gleichen Richtung oder zwei ungerichtete Kanten verbunden sind. Das nennt man parallele Kanten. Bei parallelen Kanten taucht ein Nachbar dann zweimal in der Liste der Nachbarn auf. Parallele Kanten werden selten verwendet. Wenn sie gebraucht werden, dann sollte man sich fragen, ob ein Graph wirklich das geeignete Datenmodell für die Problemstellung ist.

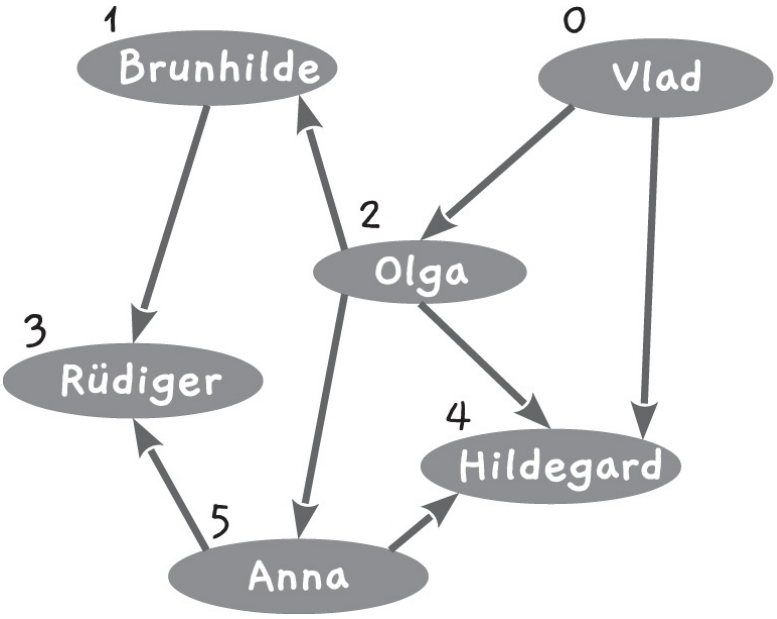
Bei gewichteten Graphen kann das Kantengewicht in den Listenelementen gespeichert werden. Sie enthalten das Ziel der Kante und das Gewicht dieser Kante. Das gilt natürlich auch ganz allgemein für Markierungen, auch wenn diese keine Gewichte sind.

Adjazenzmatrix

Bei einer Speicherung als Adjazenzliste wird jedem Knoten die Liste seiner Nachfolger zugeordnet. Bei einer Speicherung als Adjazenzmatrix wird zu jedem Paar von Knoten die Information gespeichert, ob die beiden Knoten verbunden sind. Beispielsweise als 0 (= unverbunden) oder 1 (= verbunden). Dies ist die Abbildung

$$V \times V \to \{0, 1\}$$

die als Matrix gespeichert werden kann. Die Menge V muss dazu üblicherweise wieder auf natürliche Zahlen abgebildet werden (siehe Abbildung 6.12).



u \ v	0	1	2	3	4	5
0	0	0	1	0	1	0
1	0	0	0	1	0	0
2	1	0	0	0	1	1
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	1	1	0

Abbildung 6.12: Ein Graph in einer Darstellung als Adjazenzmatrix

Bei einem ungerichteten Graphen werden Kanten wieder in beide Richtungen angegeben. Die Matrix für einen ungerichteten Graphen ist darum immer symmetrisch. Das heißt, für alle Knoten u, v gilt:

$$a[u,v] = a[v,u]$$

Bei gewichteten Graphen und Graphen, deren Kanten sonst irgendwie markiert sind, wird das Kantengewicht in der Matrix gespeichert:

$$a[u,v] = \text{Gewicht oder Markierung der Kante von } u \text{ nach } v$$

Daten und Probleme machen Graphen

So, das war jetzt genug vom Allgemeinen. Zurück zum *Influencer*-Problem. Daten und Algorithmus sind auf allgemeiner Ebene geklärt. Geiermeier kann sich daran machen, den »abstrakten« Algorithmus zu konkretisieren. Der Startpunkt ist

```
M ← ∅                                              markierte Knoten
MaxReachable(G(V,E))
    R ← {(x, size_x) | x ∈ V, size_x = |Reach(G,x)| }     Erreichbarkeitsmengen
    M ← ∅                                          Markierungen zurücksetzen
    (r, size_r) ← max(R) mit der zweiten Komponente der Paare in R
                       als Vergleichskriterium     vergleiche die Größen der Mengen
    return r                                       Knoten mit der größten Erreichbarkeitsmenge

Reach(G(V,E), v)                                   berechne alle von v aus erreichbaren Knoten in G
    M ← M ∪ {v}
    result ← {v}                                   jeder Knoten ist von sich selbst aus erreichbar
    for w ← {w | (v,w) ∈ E}
        if w ∉ M
            result ← result ∪ Reach(G, w)          gehe zu den unmarkierten Nachbarn
    return result
```

Reach war der Traversierungsalgorithmus, der die von x aus erreichbaren Knoten berechnet. Geiermeier formuliert ihn so um, dass er auf einem Graphen agiert, der in Adjazenzlisten-Darstellung kommt. Als Anhänger der funktionalen Programmierung entscheidet er sich für eine Liste von Paaren. Deren erste Komponente ist ein Element aus V, also ein Vampir, die zweite ist die Liste seiner *Follower*. Das ist insgesamt E, die Datenstruktur der Kanten. V wird als Liste verwaltet. Mengen im mathematischen Sinn sind ja keine Datenstruktur und Listen sind die einfachste Datenstruktur zur Speicherung von Mengen. Da hätte man auch eine komplizierte Datenstruktur nehmen können, einen Suchbaum beispielsweise, aber das ist erst einmal nicht notwendig. Die *Follower* von Vlad würden von Reach, das mit Graphen in dieser Datenstruktur arbeitet, folgendermaßen aufgerufen:

```
Reach(
    [Vlad, Brunhilde, Olga, Rüdiger, Hildegard, Anna],      die Knoten
    [ (Vlad, [Olga, Hildegard]),                            die Kanten
        (Brunhilde, [Rüdiger]),
        (Olga, [Hildegard, Anna]),
        (Rüdiger, Nil),
        (Hildegard, Nil),
        (Anna, [Rüdiger, Hildegard])],
    Vlad)                                                   der Startknoten
```

Als erstes fällt auf, dass die Liste der Knoten nicht explizit übergeben werden muss. V ist Bestandteil des Graphen $G(V, E)$. Der Algorithmus nutzt direkt aber nur E. Der Aufruf vereinfacht sich zu:

```
Reach(
    [ (Vlad, [Olga, Hildegard]), (Brunhilde, [Rüdiger]),(Olga, [Hildegard, Anna]),
        (Rüdiger, Nil), (Hildegard, Nil), (Anna, [Rüdiger, Hildegard])],
    Vlad)                                                   der Startknoten
```

Die Auswahl der Datenstruktur für den Graphen erfolgte insgesamt in den folgenden Schritten:

- ✔ Der Graph wird im Kern durch eine Abbildung realisiert, die Knoten auf ihre Nachfolger abbildet. Die Knotenmenge ist der Definitionsbereich dieser Abbildung.
- ✔ Die Abbildung wird als Menge von Argument-Wert-Paaren dargestellt.
- ✔ Die Menge der Argument-Wert-Paare wird als Liste implementiert.

Der Algorithmus Reach konkretisiert sich damit zu einem Algorithmus, der den Graphen in Form von einer Liste verarbeitet – der Liste E der Paare (v, l), wobei v ein Knoten ist und l seine Nachfolger.

```
M ← Nil                                          Liste (!) der markierten Knoten
Reach(E, v)
    M ← v :: M                                   markiere v
    result ← v :: Nil                            das Ergebnis wird als Liste berechnet
    for (x, W) ← E                               suche Nachfolger von v
        if x = v                                 W sind die Nachfolger von x
            for w ← W                            w ist ein Nachfolger von v
                if w ∉ M                         w ist nicht markiert
                    result ← append(result, Reach(E, w))
            return result
```

Hmm, und wer ist denn nun der einflussreichste Vampir? Natürlich Vlad, Graf von

Dracula, mit seinen direkten und indirekten *Followern* Olga, Hildegard, Anna und Rüdiger.

Was nicht passt, wird passend gemacht

Graphen sind zwar ziemlich universell und man kann sehr vieles als Graph modellieren. Aber nicht jede Modellierung ist ein Graph. Nehmen wir als Beispiel Städte und Straßen. Eine Kante in einem Graphen verbindet zwei Knoten. Eine Straße verbindet aber in der Regel nicht genau zwei Städte. Autobahnen führen beispielsweise an vielen Städten vorbei, an manchen von ihnen gibt es eine Ausfahrt, bei anderen nicht. Eine Autobahn ist darum zweifelsfrei eine Verbindung vieler Städte. Sie wird darum viel treffender durch eine Kante dargestellt, die viele Knoten verbindet.

Das wäre eventuell ein zulässiges Modell des Straßensystems, aber keine (!) Modellierung als Graph. Kanten, die mehr als zwei Knoten verbinden? So etwas gibt es in einem Graphen nicht. Dort haben Kanten genau zwei Verbindungsstellen zu den Knoten (siehe Abbildung 6.13). Natürlich ist es erlaubt, Kanten zu definieren, die viele Knoten verbinden. Klar, auf dem Papier ist alles erlaubt. Aber das ist dann eben kein Graph mehr.

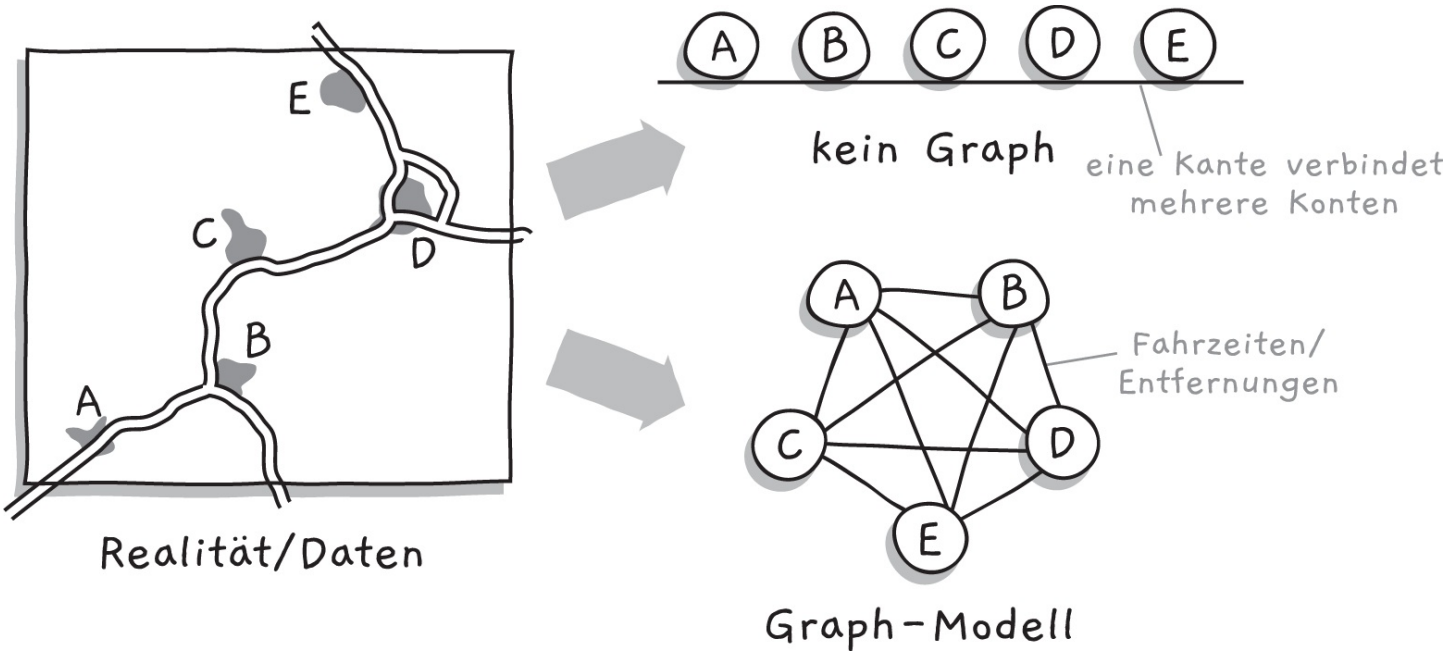


Abbildung 6.13: Vieles kann als Graph modelliert werden.

Ja und? Wenn's doch besser zur Realität passt? Warum nicht? – Nun, ganz einfach: Kein Graph, keine Graphentheorie, keine Graphenalgorithmen. Und: Keine Hilfe, keine Tradition, nur einsames Herumirren im dunklen Wald der Probleme.

Bei einer Modellierung als Graph muss alles in die Kategorien *Knoten* und *Kante* einsortiert werden. Sollen Autobahnen und die Städte, die sie verbinden, als Graph modelliert werden, dann ist es zunächst nicht offensichtlich, was Knoten und was Kanten sind. Autobahnen sind mit vielen Städten verbunden und Städte sind mit Autobahnen verbunden. Wir wissen also nicht, wie wir unsere Problemstellung als Graph modellieren

sollen. Genau genommen wissen wir nicht einmal, warum wir sie überhaupt als Graph modellieren sollten.

Ob und gegebenenfalls wie eine bestimmte Gegebenheit als Graph modelliert werden soll, hängt nicht allein von den Gegebenheiten ab. Es geht auch darum, welches Problem gelöst werden soll. Die wichtigste Eigenschaft der Graphen ist die Tatsache, dass für sehr viele *allgemeine Problemstellungen* wohlbekannte Lösungen von hoher Qualität zur Verfügung stehen. Typischerweise geht es um

- ✔ die Berechnung des kürzesten Weges zwischen zwei Knoten,
- ✔ die Bestimmung eines Weges, der über alle Knoten oder alle Kanten führt,
- ✔ die Erreichbarkeit oder
- ✔ Traversierungen als Basis oder Teil speziellerer Algorithmen
- ✔ und so weiter.

Es lohnt sich oft, Problemstellungen und Daten in das mehr oder weniger passende Korsett der Knoten und Kanten zu zwängen. Immer wenn es gelingt, ein Problem als ein solches Graphenproblem zu beschreiben, dann ist die Modellierung als Graph sinnvoll. Die spezielle Lösung für das echte Problem kann aus der allgemeinen Lösung in der Kunstwelt der Graphen entwickelt werden, die bestens untersucht ist und viele Lösungen zu bieten hat.

Nur aus bestimmten Gegebenheiten, ohne etwas über eine Problemstellung zu wissen, ist es kaum möglich, ein passendes Graphenmodell zu konstruieren. Eine naheliegende Problemstellung ist die Berechnung der kürzesten Verbindung zwischen zwei Städten. Wobei mit *kurz* sowohl die Entfernung in Kilometern als auch die Fahrzeit in Stunden gemeint sein kann. Die uns zur Verfügung stehenden Daten könnten die Fahrzeit oder die Entfernung zwischen jeweils zwei Städten beschreiben. Als Graph ausgedrückt haben wir die Städte als Knoten und die Fahrzeit beziehungsweise die Wegstrecke als Kanten. Im entsprechenden Graphen ist dann jede Stadt mit jeder Stadt direkt verbunden.

In der Welt der Graphen löst dann ein »abstrakter Algorithmus« das Problem. »Abstrakt«, weil in ihm mathematische Objekte vorkommen: Mengen, Abbildungen und so weiter. Diese »sphärischen« Dinge müssen dann in konkrete Datenstrukturen übersetzt werden. Dabei kann es dann schon auch von der aktuellen Situation, den (unter Umständen sehr zahlreichen) Daten, der verwendeten Programmiersprache und dem Programmierstil abhängen, welche Datenstrukturen genau zu wählen sind.

Im nächsten Abschnitt sehen wir ein weiteres Beispiel für die Fähigkeit der Graphen, sich an unterschiedlichste Problemstellungen anzupassen, auch an solche, die auf den ersten Blick nichts mit Knoten und Kanten zu tun haben.

Erst die Schuhe, dann das Hemd – oder wie?

Graphen sind gelegentlich auch eine sinnvolle Modellierung für Gegebenheiten und Problemstellungen, die nichts mit Verbindungen, Entfernungen oder Ähnlichem zu tun haben. Das Problem der topologischen Sortierung ist ein schönes Beispiel. Es geht um die Planung von Aktivitäten, die einander teilweise voraussetzen.

Beim morgendlichen Ankleiden ist beispielsweise darauf zu achten, dass die Socken vor den Schuhen angezogen werden und die Hose nicht vor der Unterhose. Es geht darum, Dinge in eine Reihenfolge zu bringen und dabei gewisse Beschränkungen zu beachten. Das ist eine übliche Planungsaufgabe. Außer Kleinkindern und zerstreuten Professoren bekommt das morgendliche Anziehen sicher jeder problemlos hin. Bei komplexeren Problemstellungen sind aber schon mal Hunderte oder Tausende von Abhängigkeiten zu beachten. Ohne algorithmische Unterstützung ist das kaum zu beherrschen.

Topologische Sortierung, ein guter Start in den Tag

Bleiben wir aber bei unserem Beispiel und beobachten uns selbst, wie wir morgens vorgehen. Viele greifen das erstbeste Teil, das nicht über etwas anderes angezogen wird, und ziehen es an. Dann nimmt man das nächste, das nicht unter etwas kommt, das man noch nicht angezogen hat. Und so weiter, bis alles angezogen ist.

So, damit haben wir unseren morgendlichen Graphenalgorithmus ausgeführt. Die Daten wurden dabei zunächst in die Form eines Abhängigkeitsgraphen gebracht. Aktivitäten sind Knoten, die Abhängigkeiten (*vorher–nachher*-Beziehungen) sind die Kanten (siehe Abbildung 6.14). Dann geht es los mit irgendetwas, das ganz ohne Voraussetzungen geht. Beispielsweise mit dem Unterhemd oder den Socken. Beide können als erstes angezogen werden. Nehmen wir das Unterhemd. Das ist jetzt weg von unserem Kleiderhaufen und kann in ihm gelöscht werden. Alle Abhängigkeiten vom Unterhemd sind nun auch gelöst und die entsprechenden Kanten können ebenfalls gelöscht werden. Dann kommt die Unterhose. Sie kann vom Haufen genommen und voraussetzungslos angezogen werden. Dann die Brille, die Uhr und so weiter. So geht es weiter, bis alle Abhängigkeiten aufgelöst und alle Knoten und Kanten gelöscht sind.

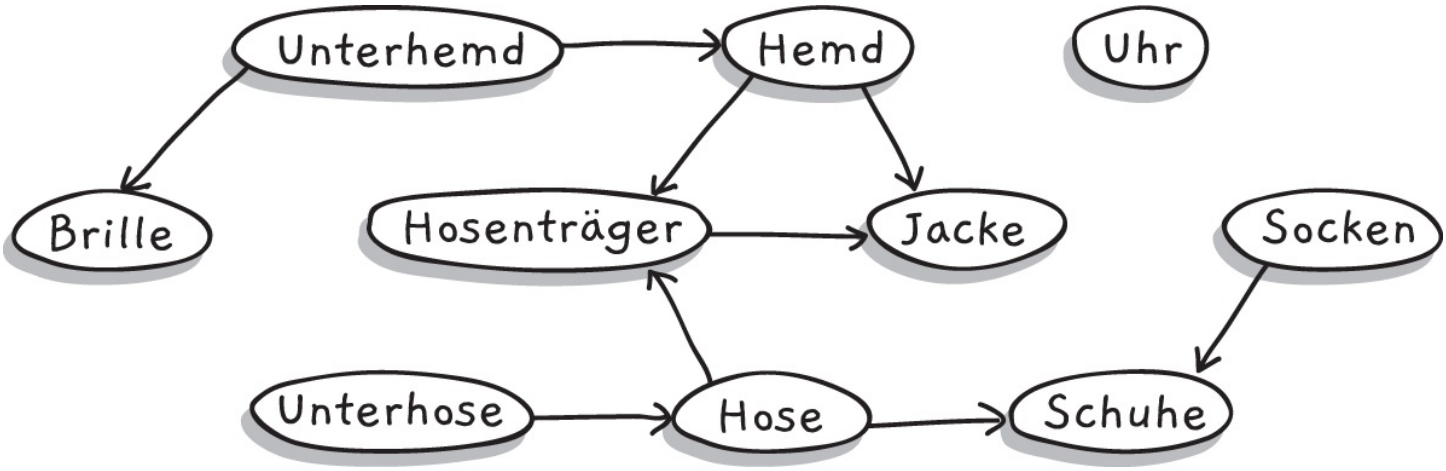


Abbildung 6.14: Abhängigkeitsgraph: Das Datenmodell der topologischen Sortierung

Das Ergebnis dieser Sortierung ist eine Liste, in der die Aktivitäten so aufgelistet sind, dass keine Abhängigkeiten verletzt werden (siehe Abbildung 6.15). Das Ganze funktioniert natürlich nur, wenn es keine zyklischen Abhängigkeiten gibt. Klar, das Henne-Ei-Problem ist unlösbar. Der Algorithmus ist insgesamt:

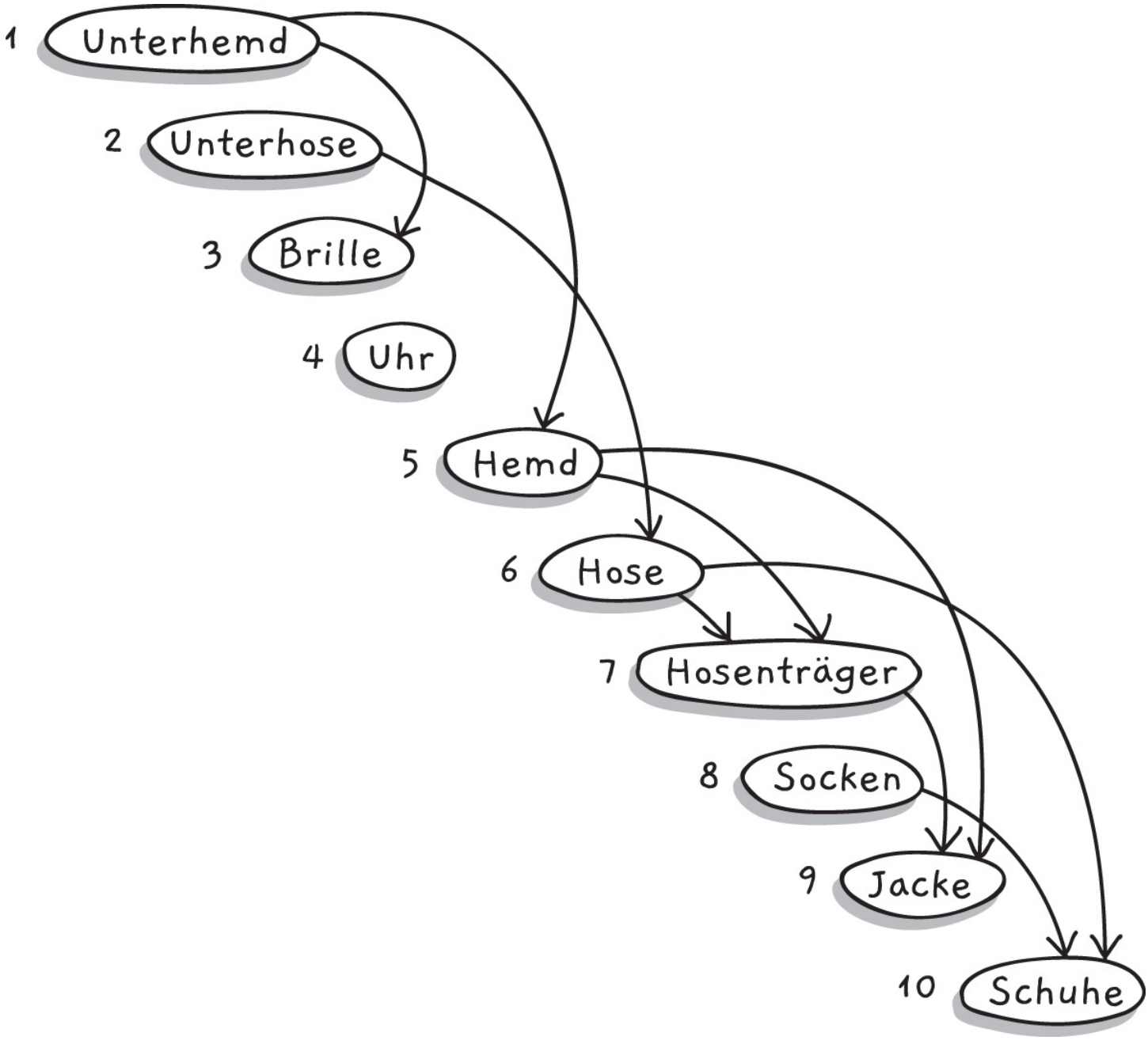


Abbildung 6.15: Der gleiche Graph, wenn die Knoten in der Reihenfolge ihrer Löschung angeordnet werden

TopoSort$(G(V,E))$
case $V = \emptyset$ *keine Knoten: nichts zu tun, fertig*
return [ ]
case $V \neq \emptyset$ *nimm eine der Aktionen ohne Voraussetzung als erste ...*
$n \leftarrow$ ein Knoten in $G(V,E)$ ohne eingehende Kante *... dann weiter mit dem Rest*
return n :: **TopoSort**(G ohne n und alle von n ausgehenden Kanten)

Mit den mathematischen Objekten und Operationen (Mengen, Mengenschnitt etc.) bewegen wir uns wieder auf einem recht abstrakten Niveau. Das Ganze ist trotzdem als Algorithmus akzeptabel. Es ist ja zweifelsfrei klar, dass alle Operationen mit einfachen Aktionen auf elementaren Datenstrukturen ausgeführt werden können. Die Frage ist, welche das sein sollen. Die Form sollte immer der Funktion folgen. Schauen wir uns also

die Aktionen auf dem Graphen an. Das ist in diesem Fall recht übersichtlich. Der Algorithmus enthält nur zwei Operationen:

- ✔ Wähle einen Knoten ohne eingehende Kanten aus.
- ✔ Entferne diesen Knoten mit all seinen ausgehenden Kanten aus dem Graph.

Zu einem Knoten müssen bei der ersten Operation alle bei ihm eingehenden Kanten betrachtet werden und bei der zweiten Operation alle von ihm ausgehenden Kanten.

In einer imperativen Implementierung wird die Datenstruktur dann so verändert, dass ein Knoten und alle von ihm ausgehenden Kanten entfernt werden. In einer funktionalen Variante konstruiert man eine neue Version der Datenstruktur ohne den einen Knoten und seine ausgehenden Kanten.

Liste folgt Funktional

Für den funktionalen Programmierer ist die Liste die Mutter aller Datenstrukturen. Nehmen wir also zwei Listen für die funktionale Variante des Algorithmus: eine Liste, die alle Knoten enthält, und eine Liste, die alle Kanten enthält. Die Kanten sind diesmal ganz einfach Paare von Knoten.

```
TopoSortFunc(V, E)
 case V = Nil
   return Nil
 case V ≠ Nil
   n ← findSource(V, E)
   return n :: TopoSortFunc(withoutSource(V, E, n))
```

Damit werden die Algorithmen konkreter. Sie operieren jetzt nicht mehr abstrakt auf Mengen, sondern auf ganz konkreten Datenstrukturen, in denen die Informationen zu finden sind, die in den Mengen konzeptionell gespeichert sind. Einen Knoten ohne eingehende Kante können wir in dieser Datenstruktur finden, indem wir die Knoten so lange durchgehen, bis wir einen finden, der nicht als zweite Komponente in der Liste der Kanten auftaucht:

```
findSource(V, E)                          finde Knoten ohne eingehende Kante
 for v ← V                                betrachte alle Knoten v
   if {(x, v) | x ∈ V} = ∅                gibt es keine Kante, die in v endet?
     return v                             dann gib v zurück
```

Es fehlt noch das Entfernen eines Knotens und der von ihm ausgehenden Kanten. In der funktionalen Version wird natürlich nichts entfernt, wir konstruieren stattdessen eine neue Version des Graphen, bei der Knoten und Kanten jetzt in einer Liste, also in konkreten Datenstrukturen zu finden sind:

```
withoutSource(V, E, n)                 entferne n aus dem Graphen
  return ([v ∈ V | v ≠ n],             Liste aller Knoten außer n
          [(x, y) ∈ E | x ≠ n])        Liste alle Kanten, die nicht in n starten
```

Eine aktuelle Programmiersprache, die den funktionalen Stil unterstützt, sollte die in diese Algorithmen verwendeten Operationen mehr oder weniger direkt anbieten.

Array folgt Imperativ

Die Mutter der imperativen Datenstrukturen ist das Array. Die *Adjazenzmatrix* als zweidimensionales Array $adjMatrix$ ist eine angemessene imperative Datenstruktur zur Speicherung der Abhängigkeiten. Wir erstellen eine Adjazenzmatrix $adjMatrix$ und belegen sie an Position i,j mit $true$, falls Knoten i und j in einem Abhängigkeitsverhältnis stehen:

```
for v ← V,
  for (x, y) ← E
    if x = v
      adjMatrix[V.indexOf(x), V.indexOf(y)] ← true
```

Der Algorithmus kann dann im Prinzip so wie in der funktionalen Variante laufen. Statt V und E in neuen Versionen zu erstellen, verändern wir jetzt Arrays. Die Adjazenzmatrix speichert die Kante, also ein veränderliches E. Wir benötigen dann noch eine veränderliche Variante von V. Wir nehmen ein Array $V\ remaining$, in dem die Knoten vermerkt sind, die noch nicht entfernt wurden. Am Anfang sind noch alle dabei:

```
V remaining ← array[0 .. V.length −1](true, ..., true)
```

In einer Schleife werden dann diejenigen gesucht, auf die keine Kante zeigt:

```
while V remaining.contains(true)
  for v ← 0 .. V.length − 1            solange es einen Knoten v gibt, ...
    if V remaining[v]                  ... der nicht entfernt wurde ...
      if ∀x . adjMatrix[x, v] = false  ... und ohne eingehende Kante ist
        ... entferne v und seine abgehenden Kanten
```

Knoten und Kanten werden entfernt indem, sie in $V\ remaining$ beziehungsweise in $adjMatrix$ auf $false$ gesetzt werden. Die imperative Variante des topologischen Suchens ist dann insgesamt:

```
TopoSortImp(V, E)
  adjMatrix ← array[0 .. V.length −1, 0 .. V.length −1](false, ..., false)
  for v ← V,
    for (x, y) ← E
      if x = v
        adjMatrix[V.indexOf(x), V.indexOf(y)] ← true
  V remaining ← array[0 .. V.length −1](true, ..., true)
  result ← Nil

  while V remaining.contains(true)
    for v ← 0 .. V.length − 1
      if V remaining[v]
        if adjMatrix[x, v] = false forall x
          V remaining[v] ← false
          result ← V[v] :: result
          for j ← 0 .. V.length − 1
            adjMatrix[v,j] ← false
  return result.reverse
```

Das Ergebnis wird in umgekehrter Reihenfolge in *result* aufgesammelt (es ist besser, Listen an ihrem Anfang zu erweitern) und dann invertiert ausgeliefert. Es hängt natürlich vom persönlichen Geschmack ab, ob man des Morgens seine Kleidung mithilfe der funktionalen oder der imperativen Variante des Algorithmus ordnet. Wer mit beidem Probleme hat, kann die Kleidung natürlich auch abends in der Reihenfolge des Ausziehens auf einen Stapel legen. Das ist eine umgekehrte topologische Sortierung, bei der es aber schwieriger ist, Fehler zu machen: Knoten mit eingehenden Kanten sind abends schwieriger vom Körper zu entfernen als morgens vom Haufen der Kleidungsstücke.

Blicken wir noch einmal zurück auf diesen Teil, in dem Datenmodelle, Datenstrukturen, abstrakte und konkrete Algorithmen behandelt wurden:

-  Listen, Bäume und Graphen können jeweils als Datenmodell und als Datenstruktur gemeint sein – es ist wichtig, das zu unterscheiden.
  - Ein Graph ist fast immer ein Datenmodell, fast nie eine Datenstruktur.
  - Algorithmen werden oft auf mathematischen Objekten definiert. Das macht sie allgemeiner verwendbar und ihre wesentlichen Aktionen klarer.
  - Die Auswahl konkreter Datenstrukturen ist aber auch ein wichtiger Teil einer Problemlösung.

Abbildung 6.16 zeigt noch einmal ganz grob den Weg zu einer Problemlösung mit Algorithmen und Datenstrukturen.

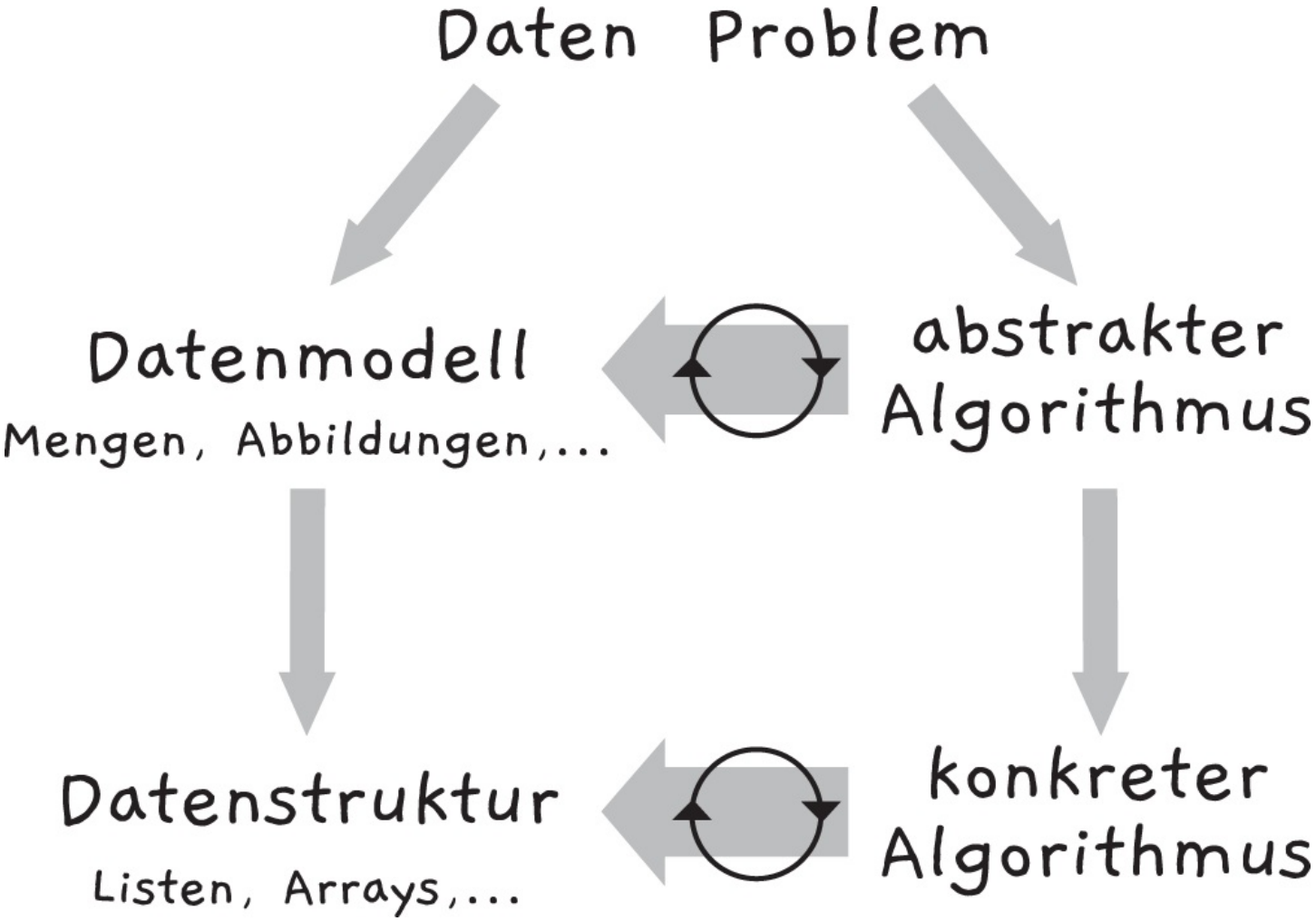
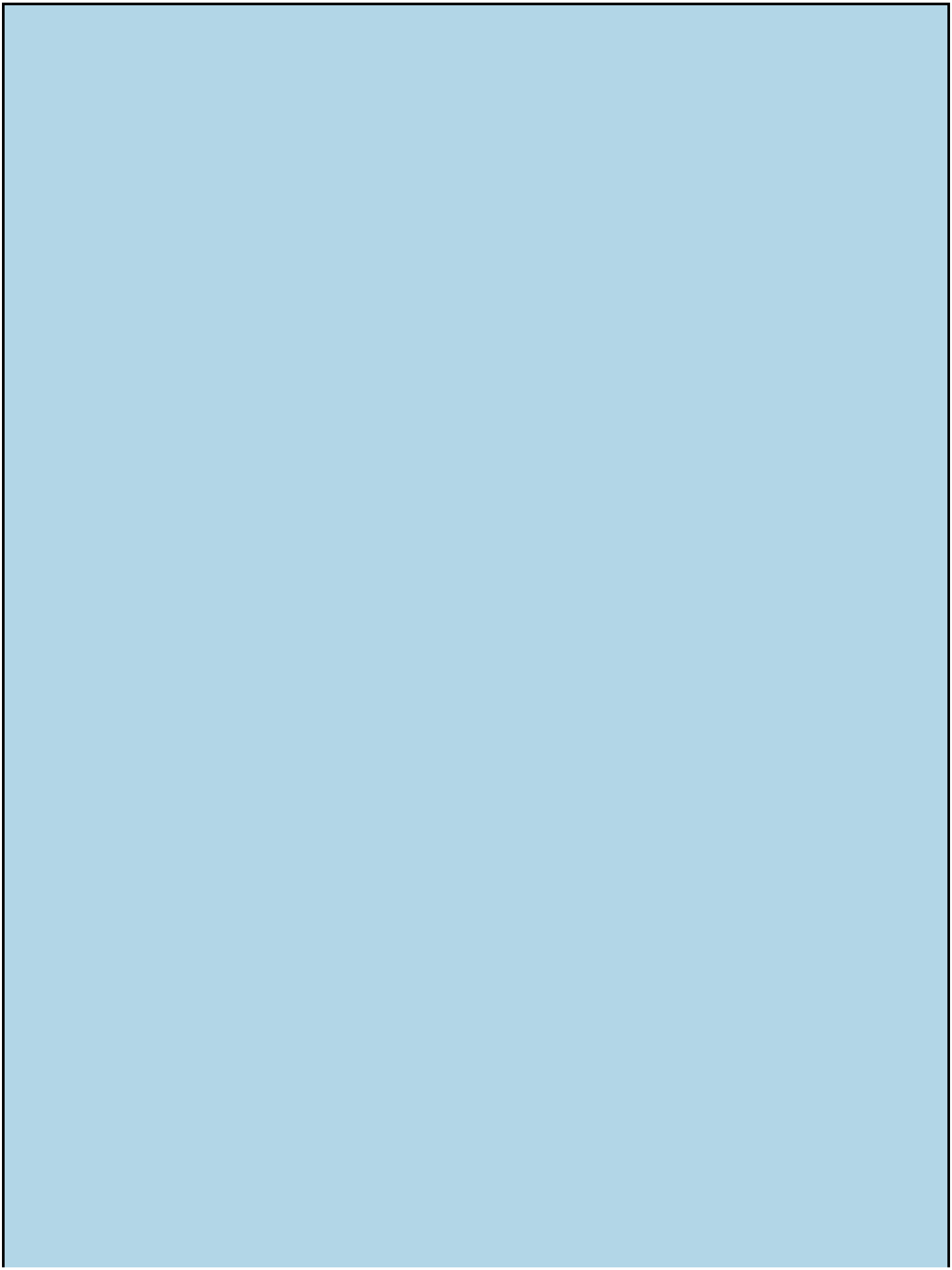


Abbildung 6.16: Datenmodell, Datenstruktur, abstrakter und konkreter Algorithmus

Teil III

Probleme und ihre Lösungen


GLAUBST DU AUCH, ES WAR SCHICKSAL, DASS WIR UNS KENNEN GELERNT HABEN?
NÖ, EIN ALGORITHMUS.
CHRISTIAN KALKERT



IN DIESEM TEIL …

Sortieren, der zeitlose Klassiker der algorithmischen Problemstellungen

Optimale Packungen finden: ein wichtiges Optimierungsproblem

Speicherung von Datenmassen: Hashtabellen und Suchbäume

Analyse von Strukturen: Kürzeste Pfade und Spannbaum

Kapitel 7

Sortieren

IN DIESEM KAPITEL

- Verschiedene Sortierverfahren
- Warum Sortieren nicht beliebig schnell gehen kann
- Algorithmische Techniken in Aktion: Teilen und Herrschen, Randomisierung, effiziente Datenstrukturen

Alles in Ordnung

Aus Kapitel 2 kennen sie schon die Bauarbeiter, die mithilfe einer Waage versucht haben, den leichtesten von ihnen herauszufinden. Solange nur ein einzelner Arbeiter ausreicht, alle notwendigen Arbeiten auf baufälligen Dachkonstruktionen auszuführen, sind die dort aufgeführten Methoden zur Suche nach dem Minimum einer Reihe von Werten dem Problem vollkommen angemessen. Sobald aber mehrere möglichst leichtgewichtige Männer für einen Montageauftrag gebraucht werden, wird die Sache schon etwas schwieriger. Außerdem wäre es schön, wenn man nicht bei jedem neuen Auftrag wieder seine Zeit mit einer erneuten aufwendigen Wiegeaktion verschwenden müsste. Der Vorarbeiter hat darum beschlossen, dass die Männer in ihrer Pause eine Liste erstellen sollen, in der die Namen aller Männer in aufsteigender Reihenfolge ihres Körpergewichts stehen (siehe Abbildung 7.1). Sollte dann im Laufe des Tages ein Job anfallen, bei dem das Gewicht der Beteiligten eine Rolle spielt, so kann der Vorarbeiter die Liste einfach von oben durchgehen und eine entsprechende Anzahl an Männern für die Aufgabe abkommandieren.

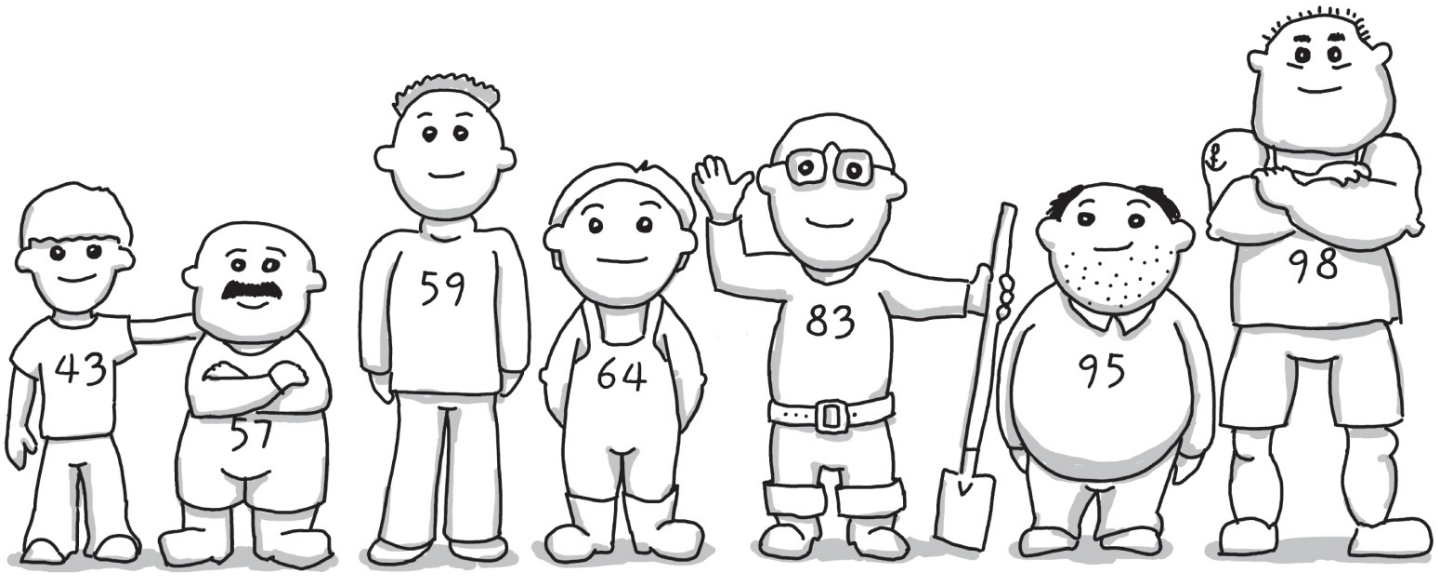


Abbildung 7.1: Bauarbeiter, nach Gewicht sortiert

In der Informatik und ihren Anwendungen gibt es viele Situationen, in denen sortierte Listen gebraucht werden. Da sollen zum Beispiel E-Mails nach Datum oder Absender sortiert werden, Suchergebnisse nach ihrer Relevanz, Waren oder Verkäufer nach Angebotspreisen oder Lieferfristen, Videos nach ihren »Likes«, Fußballspieler nach ihren Scorerpunkten oder Pizzalieferanten nach ihren positiven Bewertungen. In einem sortierten Array kann man viel schneller suchen als in einer unsortierten Liste; das können Sie in Kapitel 12 im Abschnitt zur Binärsuche nachlesen.

Methoden für das Sortieren von Listen zu finden, ist ein klassisches Thema der Algorithmik, und obwohl das Gebiet schon seit Jahrzehnten beackert wird und die meisten relevanten Algorithmen schon uralt sind, gibt es immer mal wieder neue Ideen, wie man in bestimmten Situationen vielleicht noch etwas schlauer sortieren kann. Interessant ist das Problem auch, weil man an ihm sehr anschaulich wichtige algorithmische Methoden wie zum Beispiel das schon ziemlich oft erwähnte »Teilen und Herrschen« demonstrieren kann.

Das Sortierproblem

Keiner der Bauarbeiter weiß sein genaues Gewicht, also besteht die einzige Möglichkeit, die Gewichte der Männer zu unterscheiden, darin, dass sich jeweils zwei von ihnen auf eine Waage stellen. Die Männer sollen sich also allein durch eine Abfolge von paarweisen Vergleichen sortieren, das heißt, sie müssen herausbekommen, wer von ihnen der leichteste, welcher der zweitleichteste, welcher der drittleichteste und so weiter ist. Algorithmen, die diese Aufgabe erfüllen, nennt man »vergleichsbasierte Sortierverfahren«. Es gibt auch Algorithmen, die anders arbeiten als mit paarweisen Vergleichen; damit werden wir uns in den Abschnitten *Sortieren in Linearzeit* und *Weitere Sortieralgorithmen* beschäftigen.

Lassen Sie uns zunächst das allgemeine Sortierproblem etwas abstrakter formulieren.

Abbildung 7.2 zeigt die Ausgangssituation und das Ziel: Eine Reihe von Werten steht hintereinander in einem Array a, der erste an Position l, der letzte an Position r mit $r \geq l$. Die Werte in diesem Bereich (also an den Positionen $l, l+1, l+2, \ldots, r$) sollen nun in eine andere Reihenfolge gebracht werden, und zwar so, dass sie aufsteigend sortiert sind. Am Ende sollen also genau die gleichen Werte wieder im Array a stehen, wieder an den Position $l, l+1, l+2, \ldots, r$, diesmal aber so, dass die Werte von links nach rechts immer größer werden. Je nach Wahl der Grenzen l und r können wir also entweder einen Teil des Arrays a sortieren oder auch das ganze Array. Bitte beachten Sie, dass wir die Werte *innerhalb* des Arrays umordnen wollen, also »*in place*« sortieren. Beim Sortieren wird das Array a also verändert und die ursprüngliche unsortierte Reihenfolge der Werte in a geht verloren.

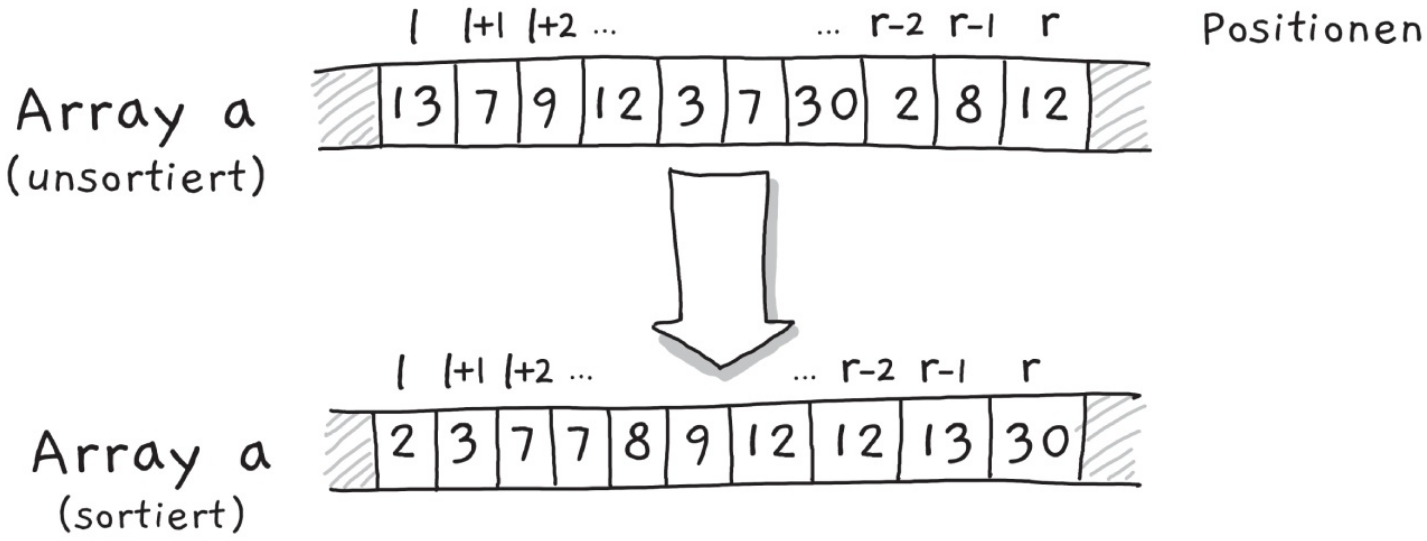


Abbildung 7.2: In-place-Sortieren des Arrays a zwischen den Positionen l und r

Im Fall der Bauarbeiter wären die im Array a gespeicherten Werte beispielsweise die »Gewichte« der Bauarbeiter, gemessen in Kilogramm. Statt die Bauarbeiter auf die Waage zu schicken, können wir nun einfach die Zahlen miteinander vergleichen. Statt Zahlen könnte man im Array a aber auch ganz andere Datentypen speichern. Dort könnten zum Beispiel die Namen von Studierenden einer Algorithmik-Übungsgruppe stehen, die wir lexikografisch aufsteigend sortieren wollen. Oder offene Rechnungen, die nach Fälligkeitsdatum oder Dringlichkeit sortiert werden müssen. Wenn man komplexe Objekte sortiert, kann man sich zum Beispiel ein Attribut auswählen und dieses als *Sortierschlüssel* verwenden; nach dem Sortieren stehen dann die Objekte in aufsteigender Reihenfolge dieses Sortierschlüssels im Array. Alternativ kann man auch für seine Objekte eine geeignete *Ordnungsrelation* definieren, nach der die Objekte sortiert werden sollen, die dann auch verschiedene Attribute der Objekte für die Vergleiche heranziehen könnte.



Ordnungsrelationen

Für das Sortieren ist vor allem eins wichtig, nämlich dass immer klar ist, welcher von zwei beliebigen Werten (also zum Beispiel zwei Zahlen oder zwei Namen oder zwei Rechnungen) in der Sortierung vor dem anderen stehen soll. Anders ausgedrückt: Es muss ein geeignetes $<$-Zeichen geben, das für alle möglichen Werte $x \neq y$ sagt, ob $x < y$ oder $y < x$ ist. Wenn wir zum Beispiel Namen sortieren wollen, so würde das $<$-Zeichen bestimmen, in welcher Reihenfolge die Namen im Lexikon stehen.

Für *alle* Werte x und y muss also immer *genau* eine der folgenden drei Möglichkeiten zutreffen: Entweder ist $x = y$ oder $x < y$ oder aber $y < x$. Diese wünschenswerte Eigenschaft eines $<$-Zeichens nennt man »*Trichotomie*«.

Das reicht aber noch nicht aus. Sortieren bedeutet, dass man Werte in eine *Reihenfolge* bringt, das heißt, man ordnet sie so an, dass wenn ein Wert x kleiner als ein anderer Wert y ist, x innerhalb dieser Reihenfolge weiter links als y steht. Nun nehmen Sie einmal an, Sie hätten drei Werte x, y und z mit $x < y$, $y < z$ und $z < x$. Dass ist klar, dass sich diese drei Werte nicht in eine aufsteigend sortierte Reihenfolge bringen lassen. Der Wert x kann nämlich nicht ganz links stehen, da nämlich z wegen $z < x$ links von x stehen muss; links von z müsste wegen $y < z$ der Wert y stehen und die Position von x müsste dann wegen $x < y$ noch weiter links sein. Das ist aber ein Widerspruch, denn x kann nicht gleichzeitig links *und* rechts von z stehen. »Kreise« dieser Art müssen wir also verbieten und darum fordern wir von unserem $<$-Zeichen, dass für alle Werte x, y und z mit $x < y$ und $y < z$ auch bitteschön $x < z$ zu gelten habe. Das nennt man »*Transitivität*«.

Ein $<$-Zeichen, das diese beiden Eigenschaften der Trichotomie und der Transitivität erfüllt, nennt man eine *strenge totale Ordnungsrelation*. Wenn Ihnen diese Eigenschaften irgendwie schwer verständlich vorkommen, dann seien Sie beruhigt: Wir fordern damit im Grunde nichts anderes, als Sie ohnehin von einem $<$-Zeichen erwarten würden, wenn es »kleiner als« bedeuten soll.

Wenn Sie Ihren eigenen $<$-Operator für ihre Daten definieren, so achten Sie darauf, dass $<$ tatsächlich eine strenge totale Ordnungsrelation wird. Fehlt hingegen einem $<$-Operator eine der beiden dafür notwendigen Eigenschaften, so kann man mit ihm nichts sortieren. Vielleicht würde ein Sortieralgorithmus, der diesen $<$-Operator verwendet, irgendetwas Nützliches tun, aber »sortieren« kann man das dann jedenfalls nicht nennen. Vergessen Sie dabei auch nicht, dass der Vergleichsoperator $=$ ebenfalls zum $<$-Operator passen muss, damit die Bedingung der Trichotomie erfüllt bleibt.

Sobald man sich auf einen geeigneten $<$-Operator festgelegt hat, kann man übrigens auch gleich die dazu passenden Operatoren $>$, $\leq$ und $\geq$ definieren, und zwar so:

- ✔ »$x > y$« bedeutet das gleiche wie »$y < x$«
- ✔ »$x \leq y$« bedeutet das gleiche wie »$x < y$ oder $x = y$«
- ✔ »$x \geq y$« bedeutet das gleiche wie »$x > y$ oder $x = y$«

Sie werden in diesem Kapitel eine ganze Reihe von unterschiedlichen Sortierverfahren kennenlernen. Eine Übersicht finden Sie in Tabelle 7.1.

Algorithmus	*Laufzeit*	*stabil*	*Kommentar*
SelectionSort	$\mathcal{O}(n^2)$	nein	einfach
MergeSort	$\mathcal{O}(n \log n)$	ja	Teilen und Herrschen
QuickSort	$\mathcal{O}(n^2)$	nein	randomisiert, erwartete Zeit $\mathcal{O}(n \log n)$

HeapSort	$\mathcal{O}(n \log n)$	nein	Datenstruktur Heap
CountingSort	$\mathcal{O}(n)$	ja	nur Integers; hoher Platzverbrauch
RadixSort	$\mathcal{O}(n)$	ja	nur Integers
BubbleSort	$\mathcal{O}(n^2)$	ja	einfach, aber langsam
GnomeSort	$\mathcal{O}(n^2)$	ja	langsam, braucht nur eine Schleife
InsertionSort	$\mathcal{O}(n^2)$	ja	einfach

Tabelle 7.1: Verschiedene Sortierverfahren und ihre Eigenschaften

SelectionSort: So lange wählen, bis es passt

Eine ganz einfache Sortiermethode geht so: Zunächst sucht man das Minimum in a – also einen Wert in a, der $\leq$ alle anderen Werte in a ist – und vertauscht es mit dem Wert an Position l. Dort, an den Anfang, gehört das Minimum im sortierten Array ja hin. Anschließend sucht man im restlichen Array wiederum nach dem Minimum – im gesamten Bereich von l nach r also der *zweit*kleinste Wert – und tauscht diesen Wert mit dem Wert an Position $l + 1$. Danach sind bereits die ersten beiden Stellen in a richtig sortiert; im restlichen Array sucht man dann wiederum das Minimum und tauscht es an die Position $l + 2$ und so weiter.

Der Algorithmus heißt SelectionSort. Bei ihm gibt es im Array immer zwei Bereiche: Im vorderen Teil (von l bis $i - 1$, in Abbildung 7.3 weiß dargestellt) ist das Array bereits fertig sortiert, wohingegen es im hinteren Teil (von i bis r, grau dargestellt) noch unsortiert ist. In jedem Schritt i wird das Minimum im hinteren Teil gesucht und an die Position i getauscht. Damit ist der vordere, sortierte Teil um eine Position gewachsen, während der hintere, unsortierte Teil um eine Position kleiner geworden ist. Das wiederholt man so lange, bis das Array fertig sortiert ist.

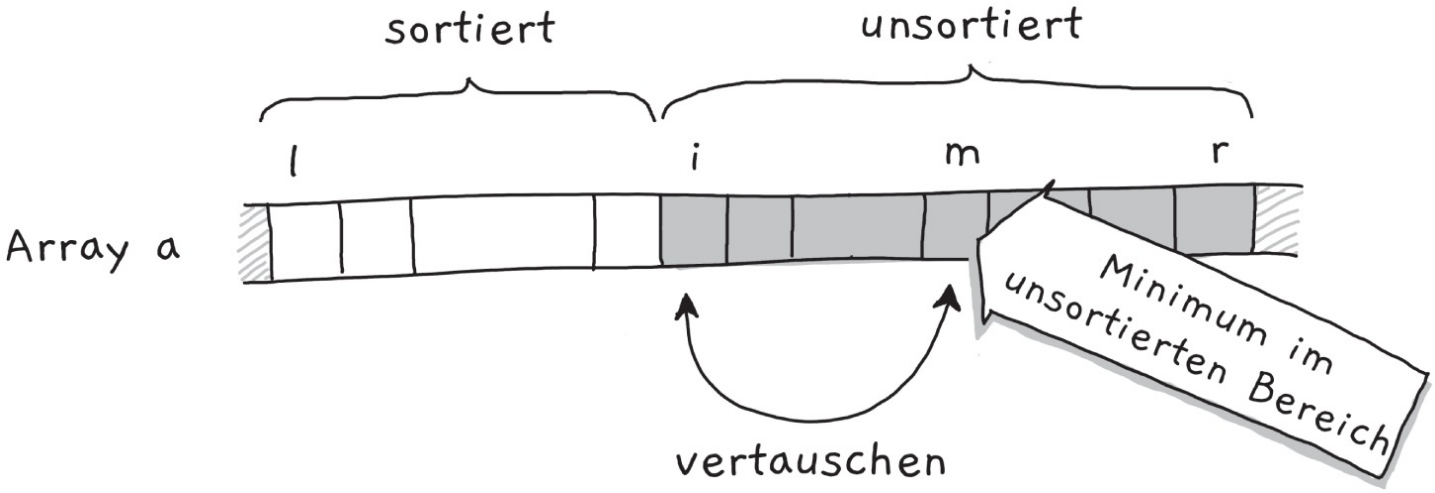


Abbildung 7.3: So funktioniert SelectionSort.

```
SelectionSort(a, l, r)                  a: Array von Werten; l, r: Positionen in a, l ≤ r
    for i ← l, ..., r − 1               durchlaufe alle Position i in {l, ..., r − 1}
        m ← argmin(a, i, r)             bestimme Minimum im bislang unsortierten Teil von a ...
        vertausche a[i] mit a[m]        ... und sortiere es an der richtigen Position ein

argmin(a, i, r)                         Position des Minimums in a zwischen Positionen i und r
    m ← i                               m: die Position des kleinsten bisher gesehenen Werts
    if r > i                            wenn es noch weitere Positionen zwischen i und r gibt, ...
        for j ← i + 1, ..., r           ... gehe alle Positionen der Reihe nach durch
            if a[j] < a[m] then         sobald ein kleinerer Wert in a gefunden ist, ...
                m ← j                   ... merke dir seine Position
    return m
```

Sobald der hintere Teil nur noch aus einem Element besteht, haben wir es geschafft; die Schleife muss also nur bis $i = r - 1$ gehen. Das Minimum im Array a zwischen den Positionen i und r wird mit der Hilfsfunktion argmin gefunden. argmin gibt dabei nicht das Minimum selbst zurück (wie es min aus dem Abschnitt *Beispiel 1: Minimum* in Kapitel 13 tut), sondern dessen Position m im Array a. SelectionSort vertauscht dann die Werte an den Positionen m und i miteinander. Da der Algorithmus immer nur Werte im Array miteinander vertauscht, geht niemals einer verloren.

Abbildung 7.4 zeigt SelectionSort in Aktion. Der unsortierte Bereich ist jeweils in grau eingezeichnet; diesen Bereich muss argmin bei jedem Durchlauf durchsuchen, um das jeweilige Minimum zu finden.

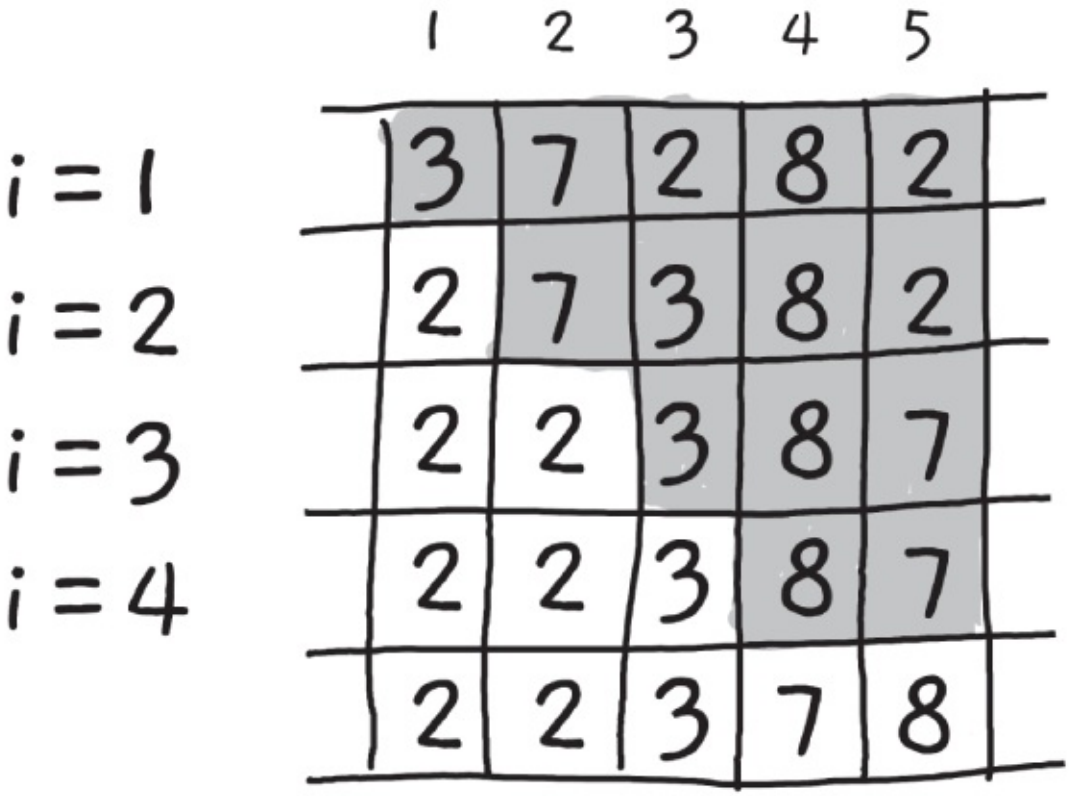


Abbildung 7.4: Ein Beispiel für SelectionSort

Laufzeit von SelectionSort

Alle grauen Bereiche zusammen, also der Gesamtaufwand aller argmin-Aufrufe, bestimmen die asymptotische Laufzeit von SelectionSort. Ganz am Anfang, wenn also $i = l$ ist, muss argmin das ganze Array durchsuchen. Mit steigendem i wird die Liste der unsortierten Werte immer kleiner und argmin hat immer weniger zu tun. Wie Sie in Abbildung 7.4 sehen können, summiert sich zum Beispiel der Aufwand für alle Aufrufe von argmin beim Sortieren von $n = 5$ Werten auf insgesamt $5 + 4 + 3 + 2 = 14$ graue Felder. Das ist in etwa das obere rechte Dreieck der in der Abbildung dargestellten 5×5-Matrix. In Abbildung 7.5 ist das noch einmal schematisch für beliebige n dargestellt. Insgesamt durchläuft argmin bei der Sortierung von n Werten ungefähr $\frac{1}{2} \cdot n^2$ viele Felder. (Genau genommen sind es $\frac{1}{2} \cdot (n^2 + n - 2)$, aber dieser Unterschied ist für die Abschätzung der asymptotischen Laufzeit unerheblich.) Was SelectionSort abgesehen von argmin sonst noch macht – wie die Abwicklung der i-Schleife oder das Vertauschen von $a[i]$ mit $a[m]$ – braucht insgesamt nur $\mathcal{O}(n)$ viel Zeit und fällt gegenüber dem Gesamtaufwand für die Aufrufe von argmin ebenfalls nicht ins Gewicht. Die Laufzeit von SelectionSort ist somit $\mathcal{O}(n^2)$: Verdoppelt man n, so hat argmin das Vierfache zu tun; wenn man n verdreifacht, so verneunfacht sich der Aufwand.

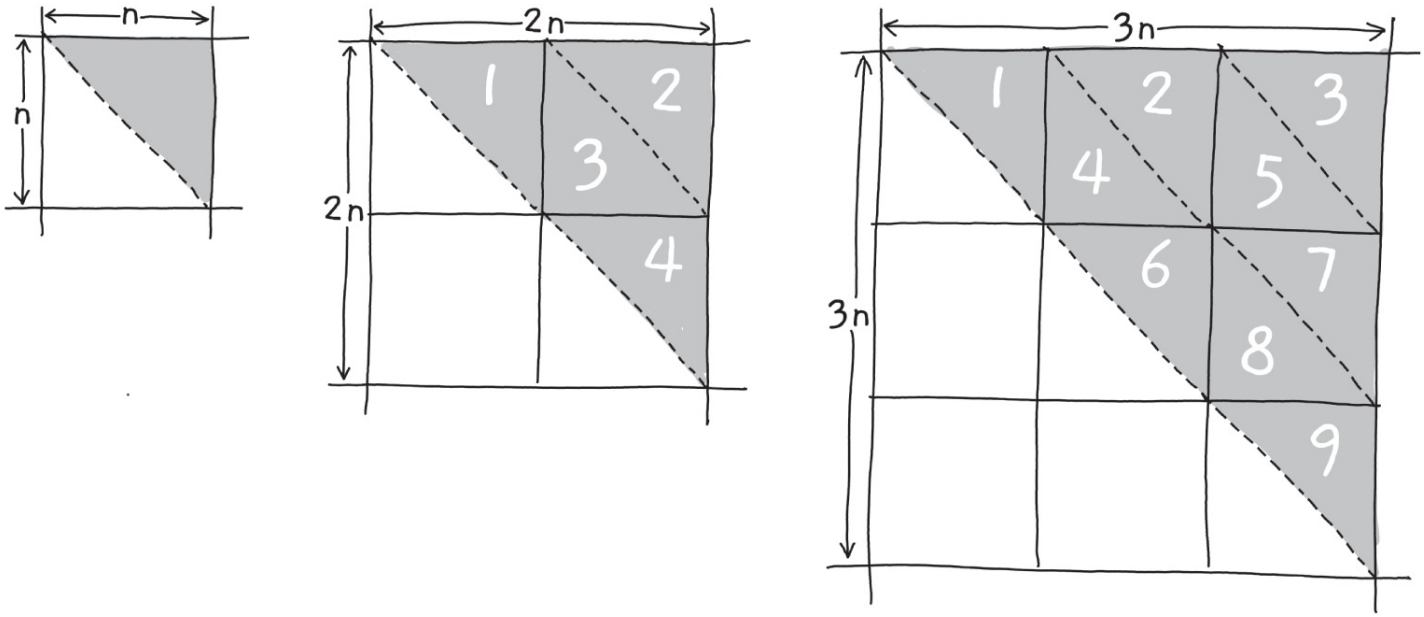


Abbildung 7.5: Die Laufzeit von SelectionSort ist $\mathcal{O}(n^2)$.



Die Groß-$\mathcal{O}$-Notation für die asymptotische Abschätzung der Worst-Case-Laufzeit wird ausführlich in Kapitel 2 behandelt.

MergeSort: Geteiltes Leid ist halb sortiert

An dieser Stelle werden Sie als Algorithmikerin oder Algorithmiker natürlich fragen: »$\mathcal{O}(n^2)$ ist ja schön und gut, aber geht das nicht auch schneller?« Wenn wir jeden der n Werte mit jedem anderen Wert vergleichen müssten, so wären das $\mathcal{O}(n^2)$ viele Vergleiche, und damit wäre die Laufzeit für das Sortieren tatsächlich mindestens quadratisch. Zum Glück kommt uns hier eine Eigenschaft der Ordnungsrelation $<$ zupass, die wir weiter oben bereits mit dem Begriff »Transitivität« bezeichnet haben. Wenn wir zum Beispiel schon wissen, dass $x < y$ und außerdem $y < z$ ist, dann müssen wir x und z gar nicht mehr miteinander vergleichen, denn aus der Transitivität folgt ja bereits, dass $x < z$ sein muss.

Wenn man geschickt auswählt, welche Werte man mit welchen vergleicht, kann man die Zahl der benötigten Vergleiche tatsächlich so weit reduzieren, dass eine Laufzeit herauskommt, die besser als $\mathcal{O}(n^2)$ ist. Die Antwort auf die Frage lautet also: »Ja, das geht schneller!«. Sie werden gleich einen Algorithmus kennenlernen, der das schafft. Dafür brauchen wir einen Trick, der »Teilen und Herrschen« genannt wird, und den wir ausführlich in Kapitel 12 untersuchen werden.

Das ist die Idee des Algorithmus: Wenn Sie zum Beispiel n Werte im Array a sortieren wollen, dann teilen Sie das Array a in zwei möglichst gleich große Hälften auf und sortieren diese Hälften erst einmal getrennt voneinander (siehe Abbildung 7.6).

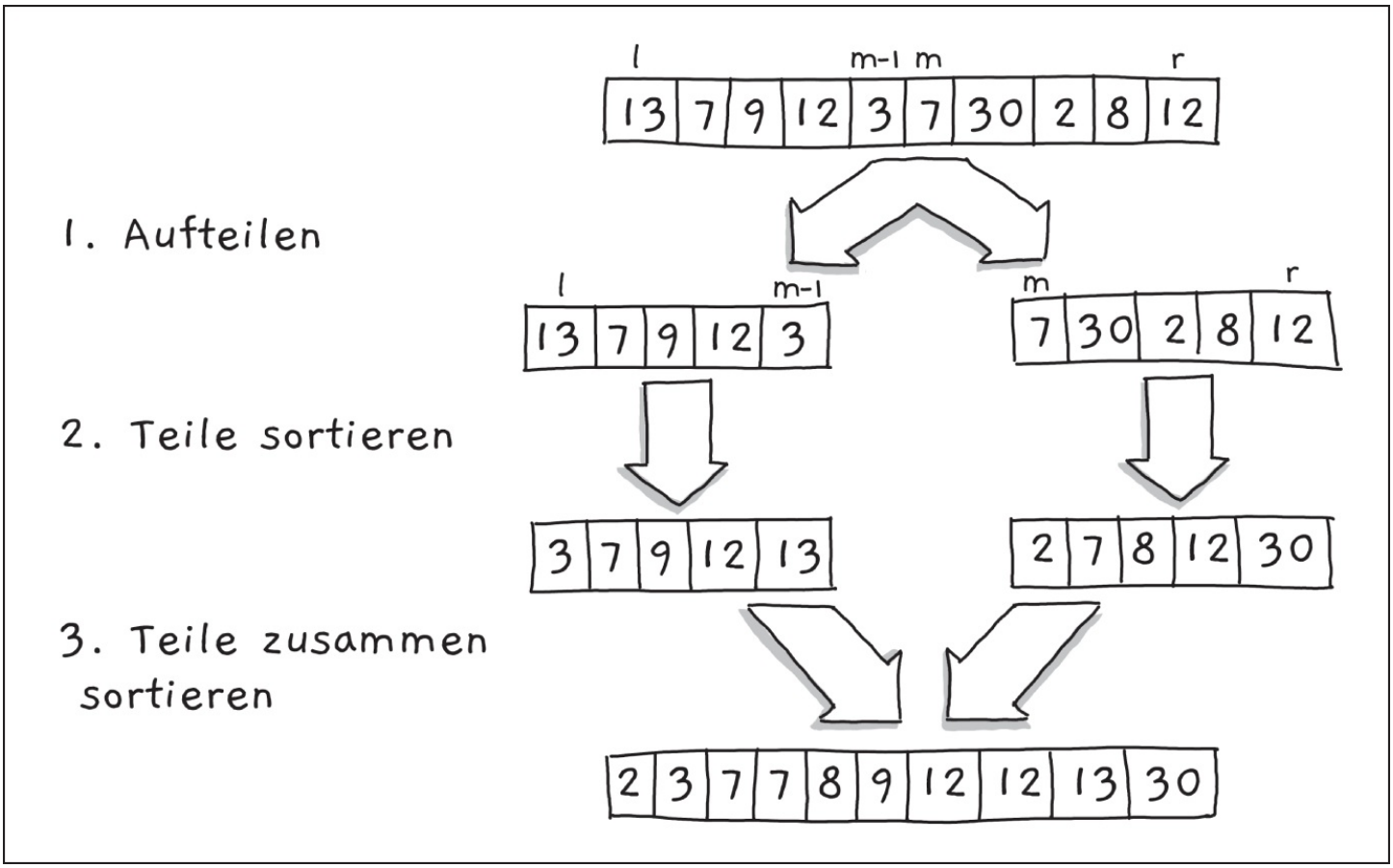


Abbildung 7.6: Sortieren mit Teilen und Herrschen

MergeSort(a, l, r) *sortiert die Werte im Array a zwischen Position l und r*
  if $l < r$ **then** *wenn es mehrere Werte zu sortieren gibt, tue Folgendes:*
    $m \leftarrow (l + r)/2$, aufgerundet *m ist die Mitte zwischen den Positionen l und r*
    sortiere a von l bis $m - 1$ (die eine Hälfte von a)
    sortiere a von m bis r (die andere Hälfte von a)
    Merge(a, l, m, r) *sortiere die beiden vorsortierten Hälften zusammen*

Sobald beide Teilarrays sortiert sind, müssen Sie diese nur noch zusammensortieren. Dazu dient der Algorithmus Merge, der die beiden Teilarrays miteinander mischt wie ein Kartenspieler zwei Stöße Spielkarten.

Sortierte Teilarrays verschmelzen mit Merge

Merge durchläuft das linke und das rechte Teilarray gleichzeitig. Dazu verwendet der Algorithmus zwei »Finger«, die jeweils auf die nächsten bislang noch nicht einsortierten Werte in den Teilarrays zeigen. Der eine Finger i_l durchläuft das linke Teilarray und der andere Finger i_r durchläuft das rechte Teilarray. In jeder Runde werden die Werte, auf die die Finger zeigen, miteinander verglichen und der kleinere von beiden wird an das Ende eines Hilfsarrays b geschrieben. Anschließend wird derjenige Finger, dessen Wert gerade nach b geschrieben worden ist, um eins weitergeschoben. Sobald beide Teilarrays komplett durchlaufen worden sind, stehen in b alle Werte in aufsteigend sortierter

Reihenfolge und müssen von dort nur noch zurück nach a kopiert werden.

```
Merge(a, l, m, r)
    b sei ein Hilfsarray
    i_l ← l                                  linker Finger startet auf Position l
    i_r ← m                                  rechter Finger startet auf Position m
    j ← l                                    j ist die aktuelle Endposition in b
    while (i_l < m) or (i_r ≤ r)             wenn noch nicht alle Werte sortiert sind:
        if (i_r > r) or ((i_l < m) and (a[i_l] ≤ a[i_r])) then    vergleiche die Finger
            b[j] ← a[i_l]                    nimm den Wert aus dem linken Teilarray
            i_l ← i_l + 1                    linken Finger um eins weiterrücken
        else
            b[j] ← a[i_r]                    nimm den Wert aus dem rechten Teilarray
            i_r ← i_r + 1                    rechten Finger um eins weiterrücken
        j ← j + 1                            Endposition in b um eins weiterrücken
    for i ← l, ..., r                        fertig sortiert; kopiere b zurück nach a
        a[i] ← b[i]
```

Mit ein paar zusätzlichen Bedingungen in der **if** -Abfrage wird in Merge dafür gesorgt, dass der Algorithmus auch dann korrekt funktioniert, wenn einer der beiden Finger das Ende seiner Liste erreicht hat (siehe Abbildung 7.7).

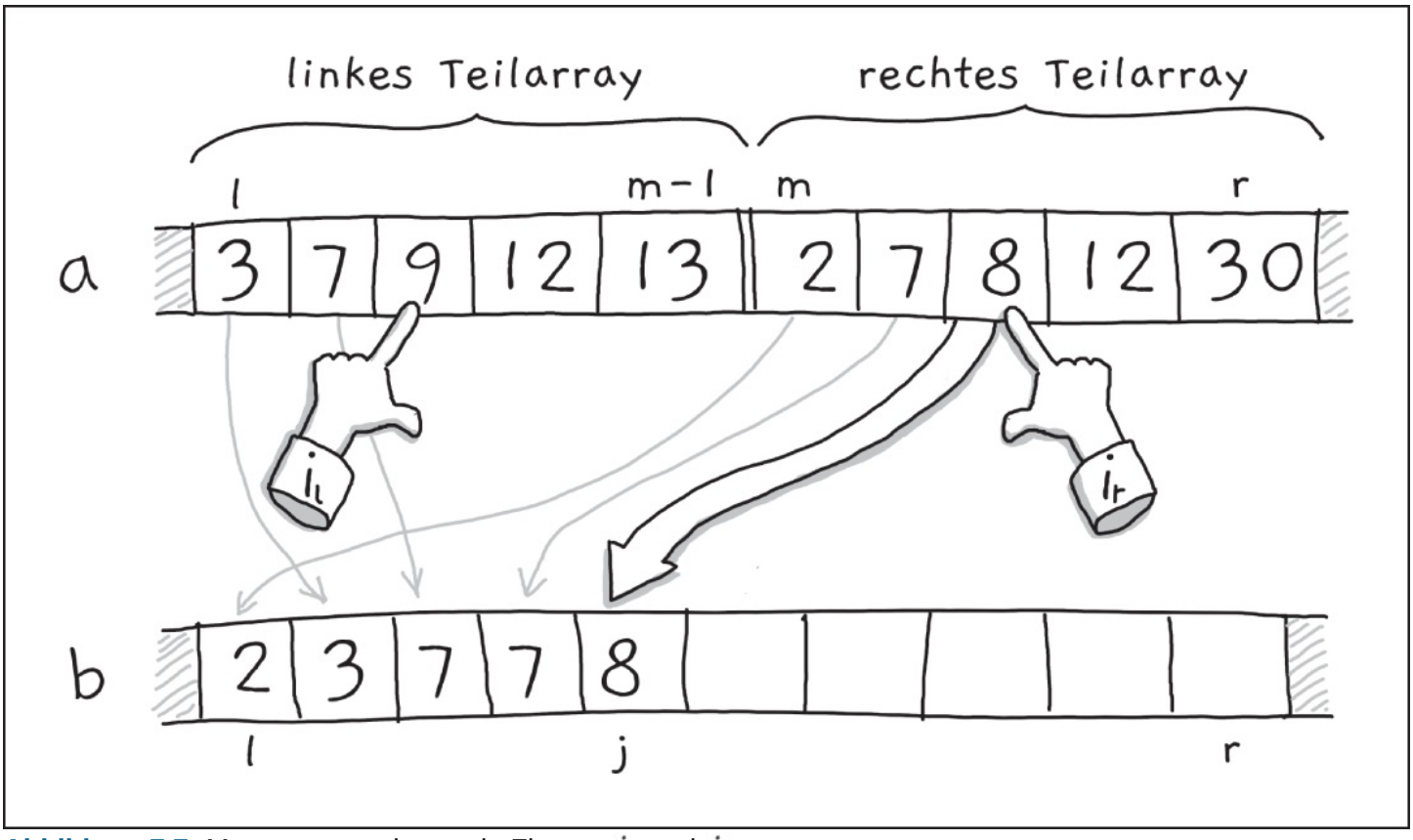


Abbildung 7.7: Merge verwendet zwei »Finger« i_l und i_r.

Wenn Sie diesen Algorithmus programmieren, sollten Sie beim Hilfsarray b übrigens ein wenig aufpassen. Anstatt dass jeder Funktionsaufruf von Merge sein eigenes Hilfsarray verwendet, besorgt man sich am besten ganz am Anfang ein einzelnes Hilfsarray b, das so lang wie das Array a ist, und benutzt es bei allen rekursiven Aufrufen von MergeSort und Merge wieder, immer auf den entsprechenden Abschnitten zwischen l und r.

Teilen und Herrschen

Welche Laufzeit hat MergeSort? Nun, das hängt davon ab, mit welcher Methode wir die beiden Teilarrays sortieren. Versuchen wir es erst einmal mit SelectionSort. Beim Sortieren von n Werten teilt MergeSort das Eingabearray a in zwei Teilarrays jeweils der Länge $\frac{n}{2}$. SelectionSort ist ein Sortieralgorithmus mit quadratischer Laufzeit, braucht zum Sortieren jedes Teilarrays also $\mathcal{O}\left(\left(\frac{n}{2}\right)^2\right) = \mathcal{O}\left(\frac{n^2}{4}\right) = \mathcal{O}(n^2)$ viel Zeit. Merge benötigt danach noch einmal $\mathcal{O}(n)$, um die beiden Teilarrays wieder zusammenzusortieren, aber das fällt gegenüber der Laufzeit von SelectionSort schon nicht mehr ins Gewicht. Insgesamt hat MergeSort damit also eine Laufzeit von $\mathcal{O}(n^2)$. Von der Laufzeitklasse her gesehen hat uns das Teilen des Arrays rein gar nichts gebracht.

Wir müssen also eine andere Methode finden, um die beiden Teilarrays zu sortieren. Abgesehen von SelectionSort haben wir in diesem Kapitel bislang nur noch eine einzige weitere Sortiermethode kennengelernt, und zwar MergeSort. Also lassen Sie uns doch einmal MergeSort selbst dafür verwenden:

```
MergeSort(a, l, r)                    sortiert die Werte im Array a zwischen Position l und r
    if l < r then                     wenn es mehrere Werte zu sortieren gibt
        m ← (l+r)/2, aufgerundet      m: Mitte zwischen den Positionen l und r
        MergeSort(a, l, m − 1)        sortiere rekursiv die eine Hälfte von a
        MergeSort(a, m, r)            sortiere rekursiv die andere Hälfte von a
        Merge(a, l, m, r)             sortiere die beiden vorsortierten Hälften zusammen
```

Auch wenn diese Idee auf den ersten Blick völlig verrückt zu sein scheint – sie funktioniert tatsächlich. MergeSort teilt das Array so lange in zwei Hälften und ruft sich rekursiv wieder selbst auf, bis die Teilarrays die Länge 1 haben. Arrays, die nur einen Wert enthalten, müssen aber gar nicht mehr sortiert werden. Dies erkennt MergeSort daran, dass die Bedingung » **if l < r** « nicht mehr gilt, und kommt daraufhin umgehend zurück (»Abbruchbedingung«). Solange die beiden Teilarrays durch rekursive Aufrufe von MergeSort korrekt sortiert werden, sorgt Merge im Anschluss dafür, dass auch das Gesamtarray korrekt sortiert ist. Wenn am Schluss alle Rekursionen beendet sind und jeder Aufruf von MergeSort zurückgekommen ist, ist a also tatsächlich korrekt sortiert.

Laufzeit von MergeSort

Und nun aufgepasst, denn jetzt geschieht ein Wunder! Dadurch, dass wir MergeSort

rekursiv gemacht haben, ist die Laufzeit des Algorithmus nun nicht mehr nur quadratisch, sondern sogar $\mathcal{O}(n \log n)$.

Diese Laufzeit folgt auch aus dem *Mastertheorem*, welches Sie in Kapitel 12 kennenlernen werden.



Dass die Laufzeit von MergeSort $\mathcal{O}(n \log n)$ ist, kann man in Abbildung 7.8 erkennen. Die Laufzeit von MergeSort wird wesentlich von der Laufzeit von Merge bestimmt, und die Laufzeit von Merge hängt linear von der Anzahl der Elemente ab, die zusammensortiert werden sollen. In der Abbildung sind diese Elemente grau gekennzeichnet. Wenn wir n Werte sortieren wollen, so wird der erste Aufruf von MergeSort am Ende Merge dazu benutzen, zwei Teilarrays der Gesamtlänge n zusammenzusortieren. Eine Rekursionsebene tiefer stehen zwei Aufrufe von Merge mit jeweils $\frac{n}{2}$ vielen Werten. Noch eine Rekursionsebene tiefer haben wir es dann mit insgesamt vier Aufrufen von Merge mit jeweils $\frac{n}{4}$ vielen Werten zu tun. Auf jeder Rekursionsebene verarbeiten die Merge-Aufrufe insgesamt n Werte, das heißt, jede Rekursionsebene verursacht also einen Rechenaufwand von $\mathcal{O}(n)$.

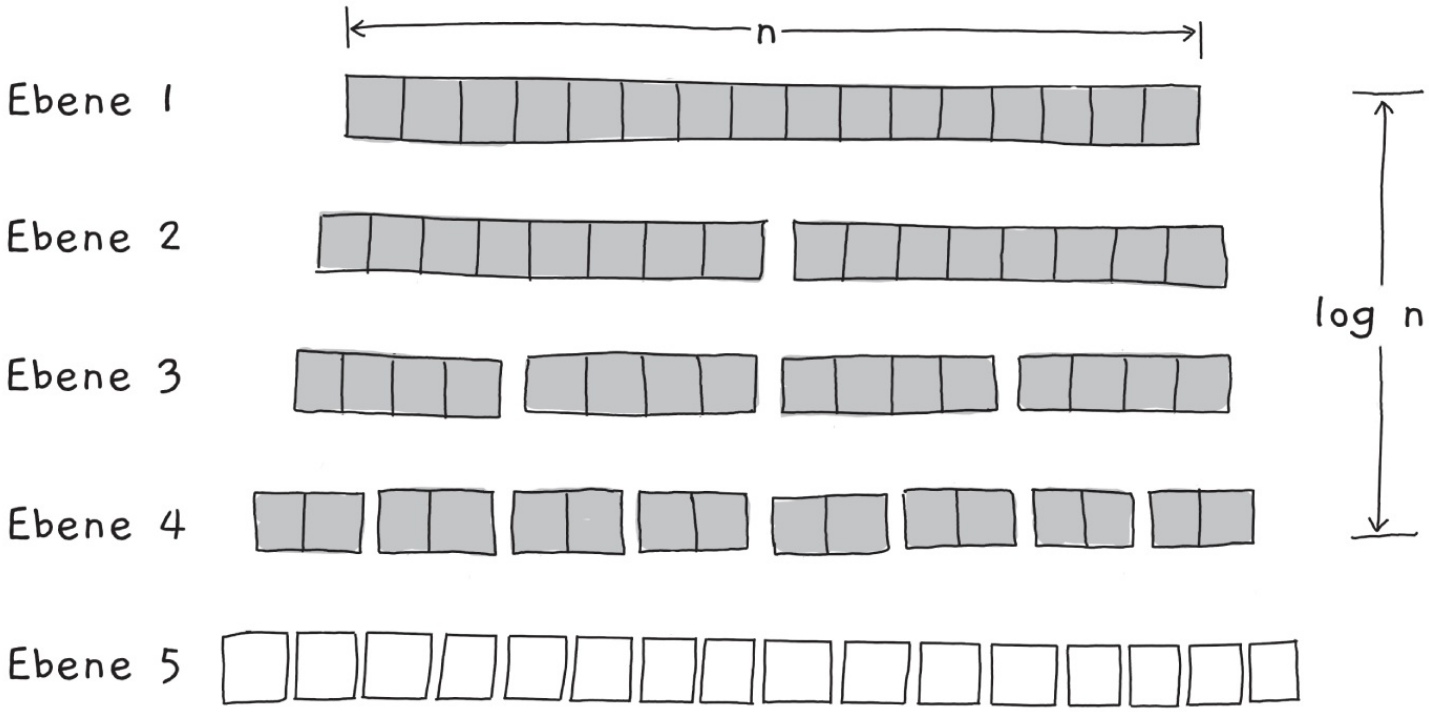


Abbildung 7.8: Die Laufzeit von MergeSort

Um die Laufzeit von MergeSort zu bestimmen, müssen wir nun den Aufwand aller Rekursionsebenen zusammenzählen. Wie viele Rekursionsebenen gibt es eigentlich? MergeSort teilt die Arrays so lange in zwei Hälften, bis die Teilarrays die Länge 1 haben. Wenn wir zum Beispiel $n = 2$ Werte sortieren wollen, dann müssen wir nur einmal teilen, bevor die Rekursion abbricht, bei $n = 4$ Werten zweimal, bei $n = 8$

Werten dreimal, bei $n = 16$ Werten viermal (siehe Abbildung 7.8), bei $n = 32$ Werten fünfmal und so weiter. Allgemein gesagt: Bei $n = 2^k$ Werten brauchen wir k Rekursionsebenen. Die Funktion, mit der man zu einem gegebenen $n > 0$ dasjenige k ausrechnen kann, für das $n = 2^k$ ist, heißt $\log_2$, der Logarithmus zur Basis 2. Die Anzahl der Rekursionsebenen bei der Sortierung von n Elementen mit MergeSort ist somit (aufgerundet) $\log_2 n$. Jede dieser Rekursionsebenen braucht $\mathcal{O}(n)$ viel Zeit, also ist die Laufzeit von MergeSort $\mathcal{O}(n \cdot \log_2 n)$.



Der Clou beim Prinzip des Teilens und Herrschens ist, dass man die Teilprobleme *rekursiv* wiederum mit Teilen und Herrschen löst. Durch diesen Trick schafft MergeSort eine Laufzeit von sagenhaften $\mathcal{O}(n \log n)$.

Mit MergeSort haben wir einen Sortieralgorithmus kennengelernt, der in der Praxis vielfach angewendet wird. Insbesondere, wenn man das Sortieren parallelisieren möchte oder wenn man riesige Datenmengen auf einem Sekundärspeicher sortieren muss, ist MergeSort das Mittel der Wahl. Auch in Standardbibliotheken findet dieser Algorithmus Verwendung und wird dabei gerne mit verschiedenen Tricks optimiert und beschleunigt.

QuickSort: Quick and Easy

Der Algorithmus MergeSort verbringt seine Zeit hauptsächlich mit Merge, also dem Zusammensortieren von bereits sortierten Teilarrays. Nun stellen Sie sich vor, wir hätten vor dem rekursiven Aufruf die Werte bereits so auf die zwei Teilarrays aufgeteilt, dass in dem linken Teilarray nur Werte stehen, die kleiner als die Werte im rechten Teilarray sind. Wenn Sie nun die beiden Teilarrays getrennt sortieren, ist damit schon automatisch das gesamte Array sortiert und Sie müssen gar nichts mehr tun. Durch geschickte Aufteilung der Werte auf die beiden Teilarrays kann man sich also das Merge sparen.

Partition teilt das Array auf

Allerdings müssen wir dann für das Aufteilen des Arrays ein wenig mehr arbeiten. Wir brauchen einen Algorithmus, den wir Partition nennen und der Folgendes leistet: Als Eingabe bekommt er ein Array a sowie die beiden Grenzen l und r des Bereichs auf dem Array, der sortiert werden soll. Als Ausgabe soll er eine Position m zwischen l und r zurückgeben. Man nennt den Wert in a an Position *Pivot* (siehe Kapitel 3). Außerdem soll der Algorithmus dafür sorgen, dass Folgendes gilt:

- ✔ Alle Werte in a, die links von Position m stehen, sollen kleiner oder gleich dem Pivot sein, also: für alle i mit $l \leq i < m$ soll $a[i] \leq a[m]$, und
- ✔ alle Werte in a, die rechts von Position m stehen, sollen größer oder gleich dem Pivot sein, also: für alle j mit $m < j \leq r$ soll $a[j] \geq a[m]$ sein.

Um das zu erreichen, darf der Algorithmus die Werte in a neu anordnen. Er könnte sie zum Beispiel aufsteigend sortieren – das wäre aber ziemlich aufwendig. Es gibt nämlich noch einfachere Methoden, wie man die genannten Bedingungen ebenfalls erreichen kann, ohne gleich das ganze Array zu sortieren. In der Literatur gibt es unterschiedliche Ansätze für Partition, Sie werden nun einen besonders einfachen kennenlernen.

Der Algorithmus wählt den Wert $a[r]$ als Pivot aus. Anschließend durchläuft er alle übrigen Werte in a von Position l bis $r-1$. Wann immer ein Wert gefunden wird, der kleiner als der Pivot ist, wird dieser Wert an das Ende des linken Teils des Array getauscht. Die Position dieses Endes merkt sich der Algorithmus in der Variablen m. Am Ende wird dann der Pivot von $a[r]$ nach $a[m]$ getauscht und die Position m des Pivot zurückgegeben. Abbildung 7.9 zeigt ein Beispiel, wie Partition die Werte in a nach und nach umordnet.

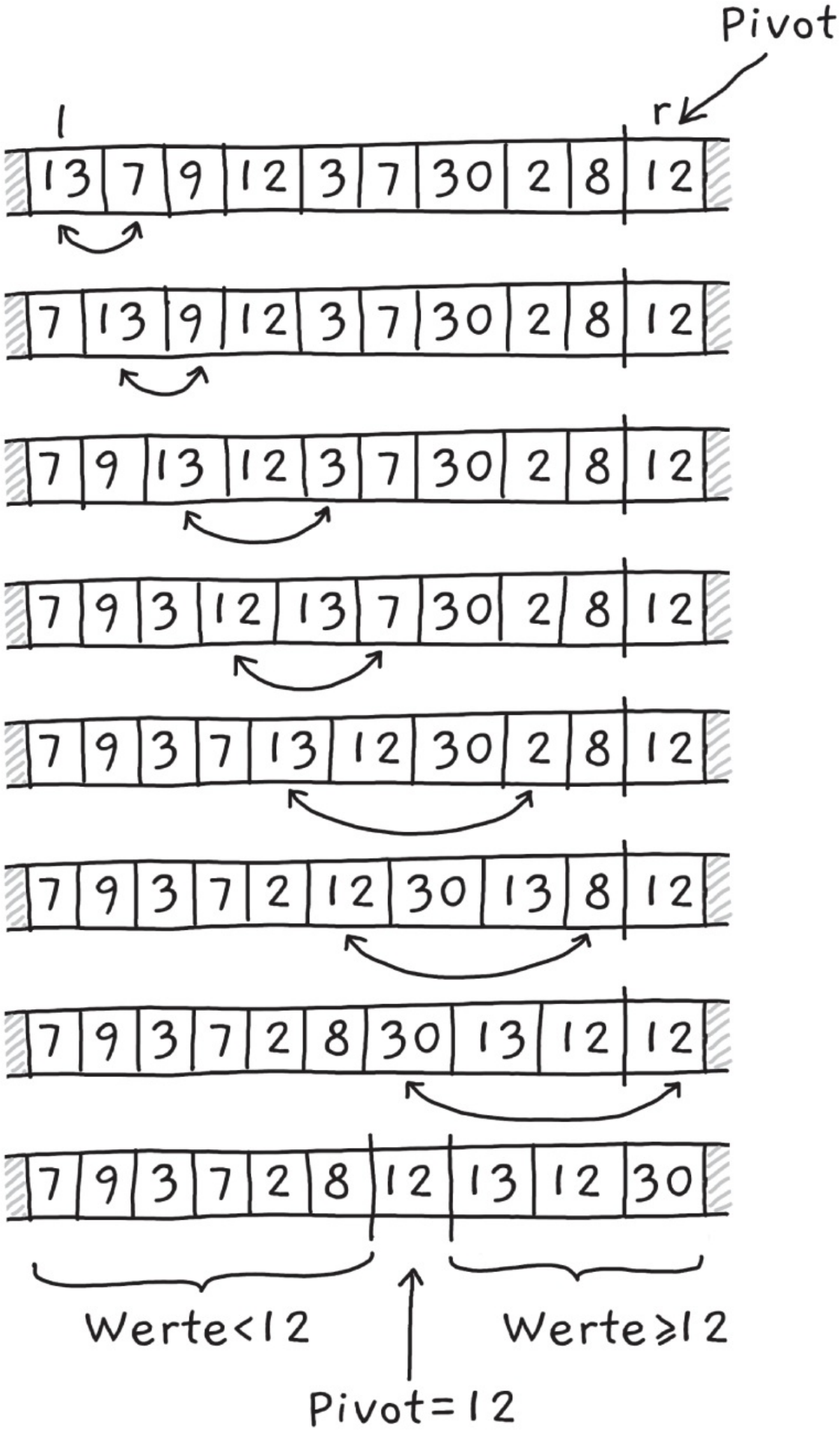

Pivot
l
r
13 7 9 12 3 7 30 2 8 12
7 13 9 12 3 7 30 2 8 12
7 9 13 12 3 7 30 2 8 12
7 9 3 12 13 7 30 2 8 12
7 9 3 7 13 12 30 2 8 12
7 9 3 7 2 12 30 13 8 12
7 9 3 7 2 8 30 13 12 12
7 9 3 7 2 8 12 13 12 30
Werte<12
Werte≥12
Pivot=12

Abbildung 7.9: Beispiel für einen Durchlauf von »Partition«

```
Partition(a, l, r)                    teile die Werte im Array a in zwei Teilarrays auf
    Pivot ← a[r]                      wähle den letzten Wert als Pivot
    m ← l                             m ist die Position, an der am Ende der Pivot stehen wird
    for i ← l, ..., r − 1             durchlaufe alle restlichen Werte im Array
        if a[i] < Pivot then          alle Werte, die kleiner als der Pivot sind, ...
            vertausche a[i] mit a[m]      ... werden nach links vertauscht und ...
            m ← m + 1                 ... die Pivotposition um 1 nach rechts verschoben
    vertausche a[m] mit a[r]          vertausche am Ende den Pivot an die Position m
    return m                          gib die Position des Pivot zurück
```

Sortieren mit QuickSort

Nun bauen wir Partition in einen Teilen-und-Herrschen-Algorithmus nach dem Muster von MergeSort ein:

```
QuickSort(a, l, r)                    a: Array von Werten; l, r: Positionen in a, l ≤ r
    if l < r then                     (Abbruchbedingung)
        m ← Partition(a, l, r)        teile a in zwei Teilarrays auf
        QuickSort(a, l, m − 1)        sortiere das linke Teilarray
        QuickSort(a, m + 1, r)        sortiere das rechte Teilarray
```

Beachten Sie, dass nach dem Aufruf von Partition der Pivot bereits an der Stelle m in a steht, in der er im sortierten Array am Ende stehen soll, und dass er darum nicht mehr zu den beiden Teilarrays gehört, die durch rekursive Aufrufe von QuickSort sortiert werden müssen. Nachdem die beiden Teilarrays sortiert sind, ist auch das gesamte Array im Bereich zwischen den Positionen l und r sortiert und der Algorithmus kann somit seine Arbeit ohne Umschweife beenden. Die Rekursion geht so lange weiter, bis das Teilarray zwischen den Positionen l und r nicht mehr sortiert werden muss, weil es nur noch einen Wert enthält oder (wenn $r < l$) sogar ganz leer ist (»Abbruchbedingung«).

Worst-Case-Laufzeit von QuickSort

QuickSort gehört in der Praxis zu den schnellsten Sortieralgorithmen und darum mag es Sie erstaunen, dass er nur eine enttäuschende Worst-Case-Laufzeit besitzt. Startet man QuickSort(a, 1, n) nämlich zum Beispiel auf einem Array a, das bereits aufsteigend sortiert ist, so wählt Partition den maximalen Wert $a[n]$ als Pivot und teilt in der Folge das Array in zwei sehr ungleiche Teile auf: Das linke Teilarray enthält dann $n-1$ Werte an den Positionen 1 bis $n-1$, während das rechte Teilarray leer ist. Auch beim nächsten rekursiven Aufruf QuickSort(a, 1, $n-1$) zum Sortieren des linken Teilarrays sieht es nicht besser aus: Partition wählt wiederum das Maximum $a[n-1]$ als Pivot und teilt das Array wiederum sehr ungleichmäßig auf in ein linkes Teilarray der Länge $n-2$ und ein rechtes Teilarray der Länge 0. So setzt es sich fort, bis das linke Teilarray nur noch den

einen Wert $a[1]$ enthält und die Abbruchbedingung zieht. Insgesamt wird QuickSort also mit Arrays aller Längen n, $n-1$, $n-2$, $\cdots$, 2 und 1 aufgerufen. Jedes Mal wird der Algorithmus Partition aufgerufen, der einmal komplett über den aktuellen Teil des Arrays läuft und dafür eine Zeit $\mathcal{O}(n)$ benötigt. Insgesamt braucht QuickSort also im *worst case* eine Zeit von $\mathcal{O}(n^2)$. Das ist die gleiche Laufzeit wie bei SelectionSort und viel schlechter als die Laufzeit $\mathcal{O}(n \cdot \log n)$ von MergeSort.

Anders als bei MergeSort, der das Array immer möglichst in der Mitte teilt, kann es bei Partition passieren, dass das Array sehr ungleichmäßig aufgeteilt wird, und wenn das bei jedem rekursiven Aufruf wieder passiert, wirkt es sich fatal auf die Worst-Case-Laufzeit des Algorithmus aus.



Stellen Sie sich vor, Sie wären der Erfinder von QuickSort und präsentierten auf einer Konferenz Ihren Sortieralgorithmus voller Stolz den Fachkollegen. Leider sitzt unter den Zuhörern auch der Erfinder von MergeSort, der sich nach dem Vortrag meldet und fragt, wie sich QuickSort denn verhält, wenn er ein bereits sortiertes Array als Eingabe bekommt. Nun müssen Sie zugeben, dass Ihr Algorithmus peinlicherweise nur eine quadratische Laufzeit hat. Sie könnten argumentieren, dass von allen möglichen Arrays nur eine sehr kleine Minderheit für ihren Algorithmus so ungünstig ist. In den meisten Fällen ist QuickSort sehr schnell und Sie könnten versucht sein zu behaupten, dass er »fast in $\mathcal{O}(n \cdot \log n)$« läuft. Unerbittlich würde ihr Kollege Sie dann darauf hinweisen, dass $\mathcal{O}$ eine obere Laufzeitschranke angibt, die ab einer gewissen Eingabelänge n für *alle* Eingaben stimmen muss. So etwas wie »fast $\mathcal{O}(n \cdot \log n)$« gibt es nicht.

Sie könnten auch versuchen, einen alternativen Partition-Algorithmus vorzustellen, der mit sortierten Arrays besser zurechtkommt, doch die Gefahr ist groß, dass ihr Kollege mit einem anderen Worst-Case-Beispiel kontern kann, bei dem QuickSort mit dem neuen Partition-Algorithmus wiederum langsam läuft. Das ist ähnlich wie Knobeln: Egal, ob Sie »Schere«, »Stein« oder »Papier« wählen, es gibt für den anderen Spieler immer einen Gegenzug, mit dem er gegen Sie gewinnt. Beim Knobeln ist es darum das Beste, seine Wahl rein zufällig zu treffen, denn so hat der Gegner keine Möglichkeit, den Zug vorauszuahnen und seine Wahl entsprechend anzupassen. In der Spieltheorie nennt man das eine »gemischte Strategie« – und wir werden nun die gleiche Idee bei QuickSort anwenden.

Randomisierung

Um zu verhindern, dass Partition bei bestimmten Eingaben systematisch einen ungünstigen Pivot wählt, lassen wir den Algorithmus seinen Pivot einfach zufällig auswählen. Ein Algorithmus, der den Zufall verwendet, wird *randomisiert* genannt. So funktioniert der randomisierte Partition-Algorithmus:

Partition(a, l, r) *randomisierte Variante*

$z \leftarrow$ eine zufällige Zahl aus $\{l, l+1, ..., r\}$

vertausche $a[z]$ mit $a[r]$ *bringe den Pivot ans Ende des Arrays*

... *der Rest geht so wie bisher*

Verwendet man QuickSort mit dem Partition-Algorithmus, so verbessert sich dadurch nicht die Worst-Case-Laufzeit. Die bleibt weiterhin $\mathcal{O}(n^2)$, denn man könnte großes Pech haben und Partition wählt zufällig immer den ungünstigsten Pivot. Allerdings ist das ausgesprochen unwahrscheinlich und tatsächlich kann man Beweisen, dass der *Erwartungswert* der Laufzeit von QuickSort in $\mathcal{O}(n \cdot \log n)$ liegt.

Mit diesem Ergebnis kann QuickSort nun auch offiziell mit MergeSort mithalten, und in manchen Fällen ist QuickSort sogar schneller als MergeSort und findet sich dementsprechend auch in manchen Standardbibliotheken als grundlegender Sortieralgorithmus wieder.

Neben Randomisierung gibt es auch andere Strategien, wie man mit Worst-Case-Fällen bei QuickSort umgehen kann. Man könnte den Algorithmus zum Beispiel so umgestalten, dass er auf alternative Sortierverfahren ausweicht, sobald QuickSort im gegebenen Fall ineffizient wird. Beispielsweise könnte man dann MergeSort einsetzen oder auch den Algorithmus, den Sie als nächstes kennenlernen werden.

HeapSort: Ein Haufen Arbeit

Wenn man nach einer schnellen Lösung für ein algorithmisches Problem sucht, ist es oft eine gute Idee, einmal über Datenstrukturen nachzudenken. Für das Sortieren wäre zum Beispiel ein Datentyp nützlich, in den man zunächst alle Werte aus dem Array eintragen und sie dann in sortierter Reihenfolge wieder herausholen kann. Eine solcher Datentyp ist die *Priority Queue* (»Prioritätswarteschlange«). Sie unterstützt die zwei Methoden **push** zum Hinzufügen eines neuen Elements in die Menge der gespeicherten Werte und **extractMin** , welche das kleinste Element – das mit der höchsten »Priorität« – aus der Menge entfernt und zurückgibt.



Die Priority Queue ähnelt einer Queue (»Warteschlange«). Anders als die Queue, bei der man mit der Methode **pop** das *älteste* Element aus der gespeicherten Menge entfernt, entfernt **extractMin** das *kleinste* Element.

Mithilfe der Priority Queue lässt es sich leicht sortieren:

```
PriorityQueueSort(a, l, r)          a: Array von Werten; l, r: Positionen in a, l ≤ r
    Q sei eine Priority Queue
    for i ← l, ..., r               durchlaufe das Array ...
        Q.push(a[i])                ... und füge alle Werte in die Priority Queue ein
    for i ← l, ..., r               hole die Werte sortiert wieder aus der ...
        a[i] ← Q.extractMin()       ... Priority Queue und schreibe sie in das Array
```

Die Laufzeit dieses Algorithmus hängt natürlich davon ab, wie schnell die Methoden **push** und **extractMin** sind, und das ist wiederum von der Datenstruktur abhängig, die wir zur Implementierung der Priority Queue verwenden. Sie werden nun eine einfache, aber recht nützliche Datenstruktur kennenlernen, die man zu diesem Zweck verwenden kann.

Die Datenstruktur Heap

Ein Heap ist ein binärer Baum, das heißt, jeder Knoten im Baum hat maximal zwei Kinder. An jedem Knoten ist ein Wert gespeichert. Diese Werte sind so auf den Baum verteilt, dass Kindknoten mindestens so große Werte speichern wie ihre Eltern (»Heap-Eigenschaft«). Ein in einem Knoten gespeicherter Wert ist somit $\leq$ als alle in den Unterbäumen des Knotens gespeicherten Werte. Damit steht das Minimum aller im Heap gespeicherten Werte an der Wurzel des Baumes.



Verwechseln Sie den Heap nicht mit einem Suchbaum. Beim Suchbaum ist an jedem Knoten ein Wert gespeichert, der *größer* als alle Werte im linken Unterbaum und *kleiner* als alle Werte im rechten Unterbaum ist. Bei einem Heap hingegen ist der im Knoten gespeicherte Wert höchstens so groß wie die Werte in *beiden* Unterbäumen. Suchbäume lernen Sie in den Kapiteln 5 und 9 kennen.



Manchmal braucht man eine Datenstruktur, mit der man schnell das Maximum statt des Minimums bestimmen kann. Dafür eignet sich ebenfalls ein Heap – nur muss man ihn dann mit »umgedrehter« Heap-Eigenschaft aufbauen, also so, dass der Elternknoten immer einen *größeren* Wert speichert als seine Kinder. Mehr über diese Art des Heaps finden Sie in Kapitel 5.

Heaps kann man sehr effizient in Arrays abspeichern, so wie es in Abbildung 7.10 zu sehen ist. Dazu speichert man die Knoten des Heaps, angefangen mit der Wurzel, nacheinander ebenenweise im Array ab. Die Knoten werden in der Reihenfolge im Array gespeichert, wie man sie bei einer Breitensuche erhalten würde. Das hat den großen Vorteil, dass man die Kanten des Baumes gar nicht zusätzlich abspeichern muss, denn aus der Position eines Knotens im Array lässt sich ganz leicht die Position der damit verbundenen Knoten errechnen. Die Kinder eines Knotens an Position i sind nämlich an den Positionen $2i$ und $2i+1$ im Array gespeichert und dementsprechend steht der

Elternkonten an Position $\frac{i}{2}$ (abgerundet).

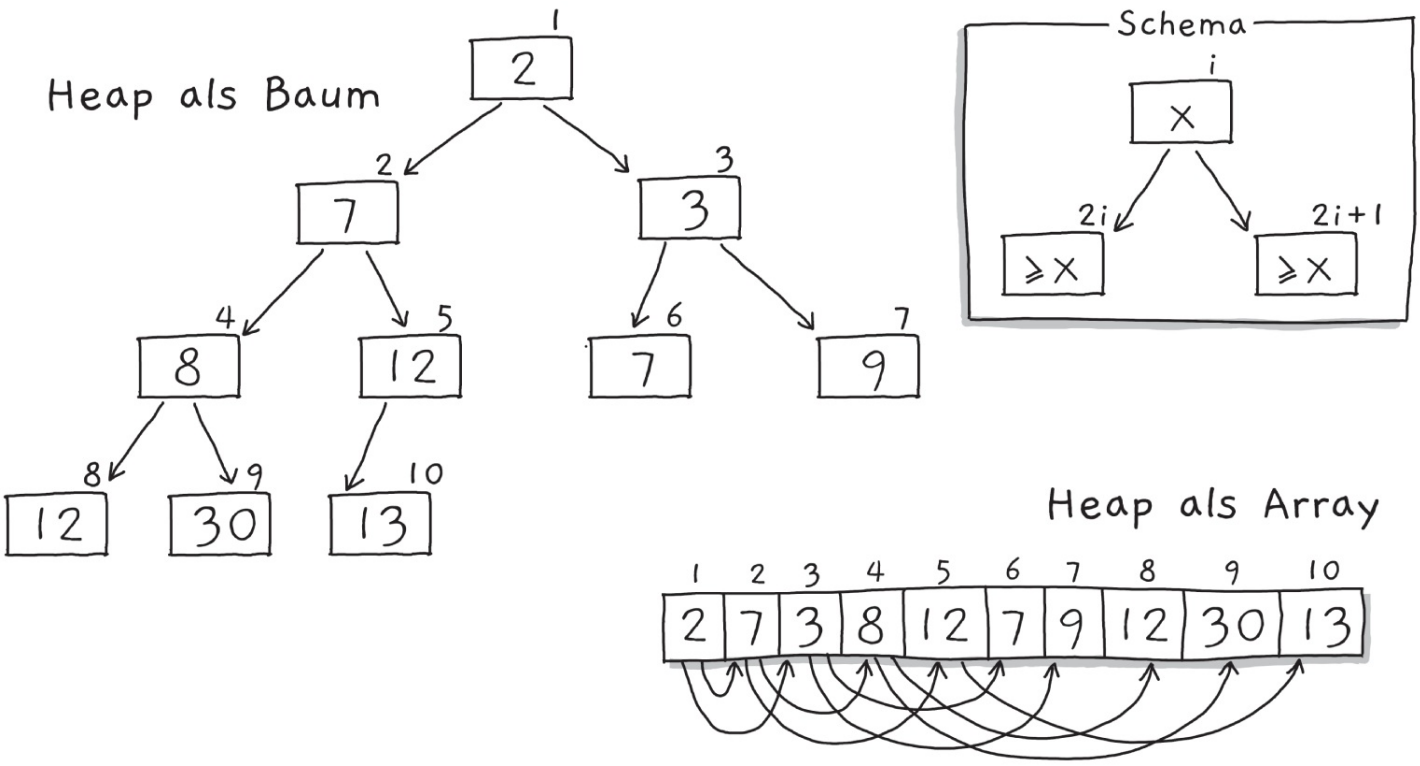


Abbildung 7.10: Der Heap ist ein Baum, der in einem Array gespeichert wird.

Der Heap als Priority Queue

Schauen wir uns an, wie man mit einem Heap eine Priority Queue realisieren kann. Zunächst brauchen wir die Methode **push** , die ein neues Element in die Priority Queue einfügt. Dafür muss das Array, in dem der Heap gespeichert wird, um einen Eintrag verlängert werden. Wir hängen das neue Element also zunächst hinten an das Array an. Diese letzte Position im Array entspricht einem Blatt im Baum; von diesem Blatt ausgehend läuft man im Baum so lange Richtung Wurzel und vertauscht dabei das neue Element mit seinem jeweiligen Elternknoten, bis die Heap-Eigenschaft wieder erfüllt ist.

```
push(H, x)                          H: Heap, speichert n Elemente; x: ein neues Element
    H[n+1] ← x                      hänge x hinten an den Heap an
    i ← n+1                         i: Position des aktuellen Knotens
    while i > 1                     solange noch nicht die Wurzel erreicht wurde:
        p ← i/2 (abgerundet)        p: Elternknoten von Knoten i
        if H[i] ≥ H[p] then break       fertig, sobald Heap-Eigenschaft hergestellt ist
        vertausche H[i] mit H[p]    ansonsten Heap-Eigenschaft herstellen ...
        i ← p                       ... und mit Elternknoten weitermachen
```

Außerdem benötigen wir die Methode extractMin, die das minimale Element in der Priority Queue (beziehungsweise *ein* minimales Element, wenn es mehrere kleinste Elemente gibt) aus der Priority Queue entfernt und zurückgibt (siehe Abbildung 7.11):

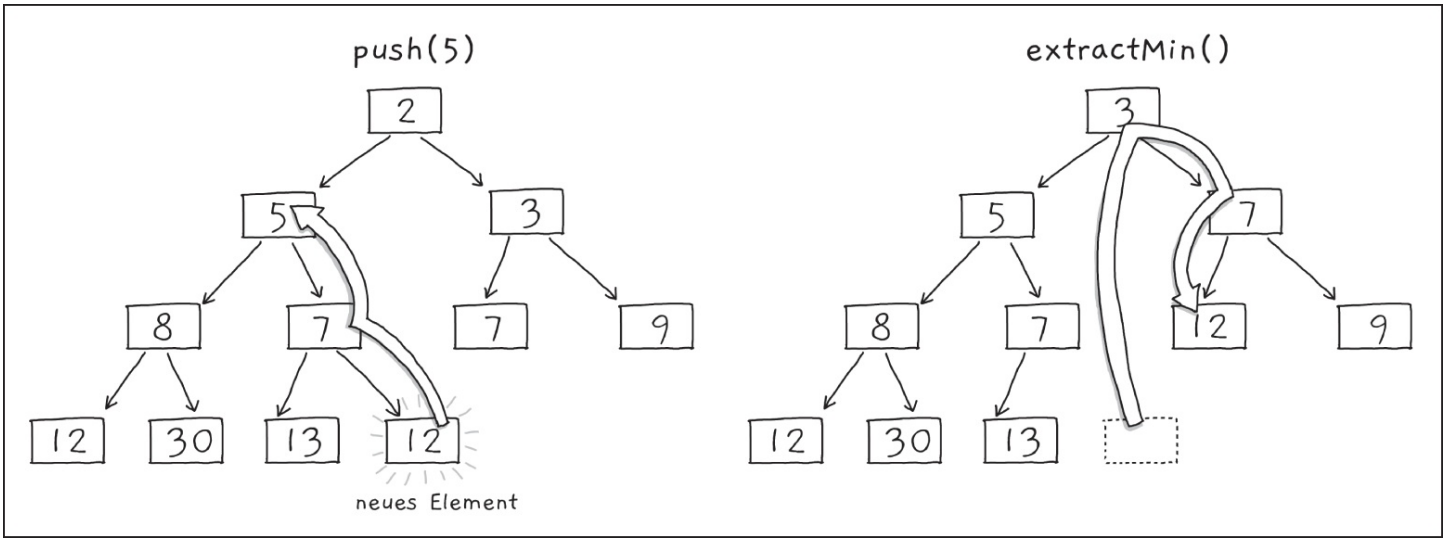


Abbildung 7.11: Der Heap von Abbildung 7.10 nach Einfügen einer 5 und anschließendem Extrahieren des Minimums

```
extractMin(H)                                   H: Heap, speichert n Elemente
    x ← H[1]                                    das gesuchte Minimum steht an der Wurzel H[1]
    H[1] ← H[n]                                 verschiebe das letzte Element in die Wurzel
    i ← 1                                       i: Position des aktuellen Knotens
    while i ≤ n                                 solange man im Baum noch absteigen kann:
        m ← i                                   m: Index des Minimums von Knoten i und seinen Kindern
        if 2i < n and H[2i] < H[m] then         wenn das linke Kind kleiner ist, ...
            m ← 2i                              ... merke dir das linke Kind
        if 2i + 1 < n and H[2i + 1] < H[m] then wenn das rechte Kind kleiner ist, ...
            m ← 2i + 1                          ... merke dir das rechte Kind
        if m = i then break                     Heap-Eigenschaft hergestellt, Schleife beenden
        vertausche H[m] mit H[i]                ansonsten Heap-Eigenschaft herstellen
        i ← m                                   mit Kindknoten weitermachen
    return x                                    Minimum zurückgeben
```

Das kleinste Element im Heap ist leicht zu finden: Es ist an der Wurzel gespeichert, also an Position eins im Array. Da sich die Anzahl der im Heap gespeicherten Elemente um eins reduziert, muss das letzte Element im Array seinen Platz räumen. Wir verschieben es zunächst an die gerade frei gewordene Position $A[1]$, danach vertauschen wir so lange Elemente zwischen Kind- und Elternknoten, bis die Heap-Eigenschaft wieder hergestellt ist.

Besser sortieren mit dem Heap

Heaps sind vergleichsbasierte Datenstrukturen, das heißt, die Reihenfolge der in ihnen gespeicherten Werte wird anhand einer Ordnungsrelation $<$ ermittelt. Verwendet man in PriorityQueueSort einen Heap zur Implementierung der Priority Queue Q, so erhält man als Algorithmus also ein vergleichsbasiertes Sortierverfahren.

Die beiden Methoden push und extractMin haben je eine Laufzeit von $\mathcal{O}(\log n)$, denn im schlimmsten Fall muss man jeweils einmal den Baum rauf oder runter und ein Binärbaum mit n Knoten hat eine Höhe von höchstens $\log_2(n)$. Verwendet man also einen Heap zur Implementierung einer Priority Queue, so läuft PriorityQueueSort genau wie MergeSort in $\mathcal{O}(n \cdot \log n)$.

Damit die Laufzeit des Algorithmus auch in der Praxis konkurrenzfähig wird, optimiert man ihn noch ein wenig. Für das Sortierungsproblem ist es nämlich ein Glücksfall, dass man Heaps in Arrays speichern kann, denn man kann zur Speicherung des Heaps einfach das Array verwenden, das man sortieren möchte. Man hat dann nämlich für die Priority Queue keinen zusätzlichen Speicheraufwand und außerdem spart man sich umständliche Kopieroperationen. Anstatt wie in PriorityQueueSort die Werte im Array eines nach dem anderen mit push in den Heap einzufügen, kann man auch alle Werte im Array in einem Durchgang so umordnen, dass die Heap-Eigenschaft erfüllt wird. Dazu kann man entweder das Array von vorne bis hinten durchgehen, jedes Element mit dem Element seines Elternknotens vergleichen und gegebenenfalls austauschen. Alternativ könnte man das Array rückwärts durchgehen und wie in extractMin das Element an jedem inneren Knoten mit dem Minimum aus den Elementen an diesem Knoten und seinen zwei Kindknoten vertauschen. Insgesamt ergibt sich in beiden Fällen eine Laufzeit von $\mathcal{O}(n)$ für die Verwandlung des Arrays in einen Heap.

Beim zweiten Teil des Algorithmus, in dem die Elemente in sortierter Reihenfolge wieder aus dem Heap entfernt werden, ergibt sich allerdings das Problem, dass man die extrahierten Elemente nicht unkontrolliert wieder ins Array zurückschreiben kann, solange dort noch der Heap gespeichert ist. Der vom Heap verwendete Bereich des Arrays schrumpft beim Entfernen von Elementen vom Ende her, also bietet es sich an, die entfernten Elemente dort zu speichern. Allerdings wird dann das Array nicht *aufsteigend* (also vom kleinsten zum größten Wert) sortiert, sondern *absteigend* (vom größten zum kleinsten Wert).

```
HeapSort(a, l, r)                    a: Array von Werten; l, r: Positionen in a, l ≤ r
    verwandle a in einen Heap
    for i ← r, r − 1,..., l          hole die Werte sortiert aus dem Heap ...
        a[i] ← Q.extractMin()        ... und schreibe sie von hinten nach vorne in das Array
```

Wenn man den Heap so umkonstruiert, dass an den Knoten immer größere (anstatt kleinere) Elemente als an den Kindern gespeichert werden, sich an der Wurzel also immer das größte Element im Heap befindet, kann man HeapSort dazu bringen, die Werte in *aufsteigender* Reihenfolge zu sortieren. Die Laufzeit des Algorithmus ist in jedem Fall $\mathcal{O}(n \cdot \log n)$.

Die maximale Sortiergeschwindigkeit

Bisher haben wir mit MergeSort und HeapSort vergleichsbasierte Sortierverfahren mit einer Worst-Case-Laufzeit von $\mathcal{O}(n \cdot \log n)$ kennengelernt. Auch QuickSort hatte im *average case* diese Laufzeit. Eine bessere Laufzeitschranke lässt sich für diese Algorithmen aber leider nicht geben, oder um es mit einem anderen Landau-Symbol auszudrücken, das eine untere Grenze für die Laufzeit angibt (zu dieser Notation siehe Kapitel 2): Diese Algorithmen haben eine Laufzeit von $\Omega(n \cdot \log n)$.

Die Frage ist nun: Geht es nicht doch noch etwas schneller? Vielleicht mit irgendeinem besonders ausgeklügelten Algorithmus? Um es kurz zu machen: Die Antwort lautet »nein«, jedenfalls nicht mit einem rein vergleichsbasierten Verfahren. »Vergleichsbasiert« heißt, dass der Algorithmus einzig durch Vergleiche von jeweils zwei Werten herausfinden muss, in welche Reihenfolge die Werte zu sortieren sind. Die Bauarbeiter, die sich nach dem Gewicht sortieren sollen (siehe Abbildung 7.1), wissen zum Beispiel ihr genaues Gewicht nicht, sondern müssen sich immer paarweise auf eine Waage stellen, um herauszufinden, wer von ihnen schwerer ist als der andere. Auch wenn wir Zahlen sortieren, so können wir das rein vergleichsbasiert tun, indem wir keine weitere Eigenschaften der Zahlen verwenden, als dass man sie durch eine Ordnungsrelation »ist kleiner als« paarweise miteinander vergleichen kann. Die Behauptung ist also:

> Jeder vergleichsbasierte Sortieralgorithmus benötigt zum Sortieren von n Werten mindestens eine Laufzeit von $\Omega(n \cdot \log n)$. Oder anders ausgedrückt: Die beste Worst-Case-Laufzeitklasse, in der ein vergleichsbasierter Sortieralgorithmus sein kann, ist $\mathcal{O}(n \cdot \log n)$.

Der Grund dafür ist, kurz gesagt, dass es so viele Möglichkeiten dafür gibt, wie n unsortierte Werte in einem Array stehen können, dass man mindestens $\frac{n}{2} \cdot \log n$ paarweise Vergleiche braucht, um alle diese Möglichkeiten auseinanderzuhalten.



Nochmal genauer nachgerechnet

Lassen Sie uns zunächst Arrays sortieren, in denen n *verschiedene* Werte in beliebiger Reihenfolge gespeichert sind. Diese n verschiedene Werte können in $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1 = n!$ (sprich: »n Fakultät«) vielen unterschiedlichen Reihenfolgen (der Mathematiker nennt sie »Permutationen«) angeordnet sein. Jeder Sortieralgorithmus muss irgendwie herausfinden, welche dieser Permutationen tatsächlich vorliegt, um die Werte so umordnen zu können, dass sie anschließend sortiert im Array stehen. Dazu wendet der Algorithmus paarweise Vergleiche zwischen verschiedenen Werten an. Je nachdem, was bei einem solchen Vergleich herauskommt, kann der Algorithmus entscheiden, welchen Vergleich er danach durchführen will oder ob er bereits mit dem Sortieren fertig ist.

Jeder vergleichsbasierte Sortieralgorithmus arbeitet sozusagen einen Entscheidungsbaum ab, wie er exemplarisch in Abbildung 7.12 skizziert ist. Die inneren Knoten des Entscheidungsbaums entsprechen den paarweisen Vergleichen: Kommt beispielsweise bei einem Vergleich zwischen den Werten x und y heraus, dass $x < y$, dann folgt der Algorithmus dem linken Ast des Entscheidungsbaumes, ansonsten steigt er den rechten Ast herab. Erreicht der Algorithmus ein Blatt des Baumes, so ist das Array fertig sortiert und der

Algorithmus stoppt. Die Zahl der Vergleiche entspricht also der Anzahl der Knoten auf dem Pfad von der Wurzel des Entscheidungsbaums zu einem Blatt. Die Länge des längsten Pfades von der Wurzel bis zu einem Blatt – man nennt das auch die *Höhe* des Baumes – entspricht also der maximalen Anzahl an Vergleichen, die der Algorithmus benötigt.

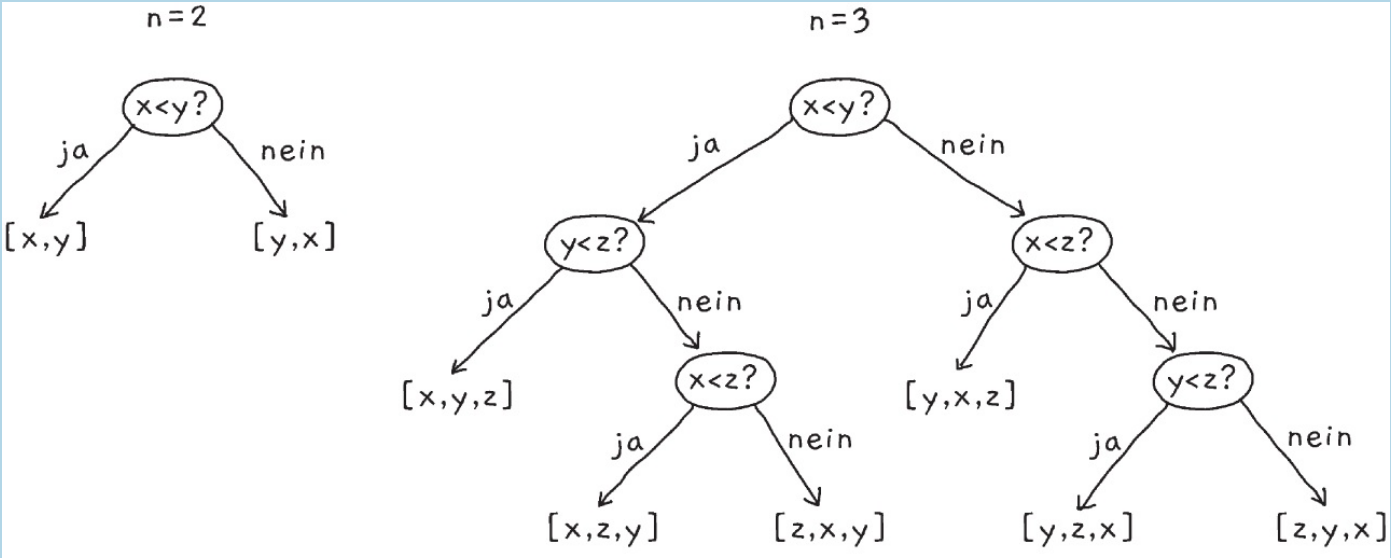


Abbildung 7.12: Entscheidungsbäume eines Sortieralgorithmus

Natürlich macht der Algorithmus außer den paarweisen Vergleichen noch andere Dinge, zum Beispiel verändert oder vertauscht er die Werte im Array. Das muss er für jede der $n!$ verschiedenen Permutationen anders machen, damit jedes Mal am Ende das Array korrekt sortiert ist. Das bedeutet, dass es mindestens $n!$ unterschiedliche Pfade von der Wurzel zu einem Blatt im Entscheidungsbaum geben muss, und da jeder Pfad an einem anderen Blatt endet, muss der Entscheidungsbaum mindestens $n!$ viele Blätter haben.

Wie hoch muss ein Entscheidungsbaum mindestens sein, wenn er wenigstens $n!$ viele Blätter haben muss? Lassen Sie uns die Frage zunächst umdrehen: Wie viele Blätter kann ein Entscheidungsbaum maximal haben, wenn er eine maximale Höhe von zum Beispiel k haben darf? Für ein paar kleine Werte von k sehen wir die Antwort in Abbildung 7.13. Bei $k = 1$ ist nach dem ersten Vergleich bereits Schluss und der Entscheidungsbaum hat nur zwei Blätter. Bei $k = 2$ können wir maximal zwei Vergleiche pro Pfad machen und das führt zu maximal vier Blättern, bei $k = 3$ wären es acht und so weiter. Entscheidungsbäume sind binäre Bäume, das heißt, alle Knoten haben höchstens zwei Kinder und ein binärer Baum der Höhe k hat höchstens 2^k Blätter. Wenn wir nun mindestens $n!$ Blätter haben wollen, müssen wir k so groß wählen, dass $2^k \geq n!$. Nach k umgeformt bekommen wir dann $k \geq \log_2(n!)$. Nun kann man leicht zeigen, dass $n! \geq n^{\frac{n}{2}}$ ist. Wir können also rechnen:

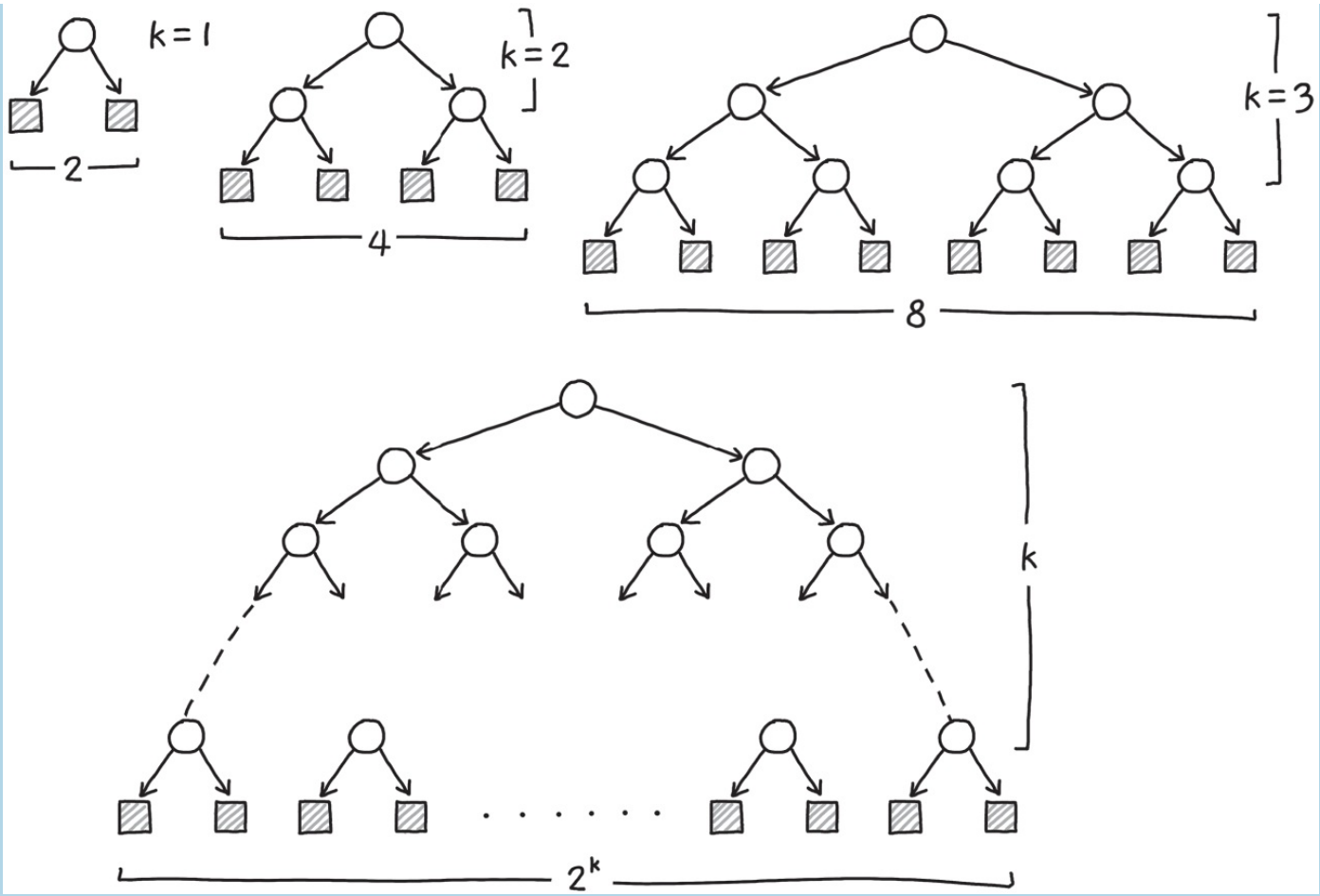


Abbildung 7.13: Maximale Blätterzahl bei Bäumen unterschiedlicher Höhe k

$$k \geq \log_2(n!) \geq \log_2(n^{\frac{n}{2}}) = \frac{n}{2} \cdot \log_2 n$$

Da die Anzahl k der Vergleiche beim Sortieren also mindestens mit $\Omega(n \cdot \log n)$ wächst und die Laufzeit des Algorithmus mindestens mit k, ist die Laufzeit des Sortieralgorithmus somit $\Omega(n \cdot \log n)$.

Wenn man sich also nur für asymptotische Laufzeiten interessiert, dann ist mit MergeSort bereits das Ende der Fahnenstange erreicht. Besser geht es nicht.

Sortieren in Linearzeit

Nachdem Sie jetzt gerade gelernt haben, dass man bestenfalls in $\mathcal{O}(n \cdot \log n)$ sortieren kann, wollen wir uns nun Sortieralgorithmen zuwenden, die in $\mathcal{O}(n)$ laufen.

CountingSort: Sortieren durch Zählen

Betrachten Sie bitte den folgenden Sortieralgorithmus und überlegen Sie kurz, in welcher Laufzeitklasse er liegt:

```
CountingSort(a, l, r) a: Array von natürlichen Zahlen; l, r: Positionen in a, l ≤ r
    if r − l ≤ 1 then return          weniger als zwei Zahlen muss man nicht sortieren
    M ← a[l]                          in M wird das Maximum der Zahlen in a berechnet
    for i ← l + 1, ..., r             durchlaufe alle restliche Zahlen in a ...
        if a[i] > M then              ... und wenn du eine größere Zahl findest, ...
            M ← a[i]                  ... dann merke dir diese Zahl
    count sei ein Array von M Zählern
    for j ← 1, ..., M
        count[j] ← 0                  initialisiere alle Zähler mit 0
    for i ← l, ..., r                 durchlaufe alle Zahlen in a ...
        count[a[i]] ← count[a[i]] + 1 ... und erhöhe den entsprechenden Zähler
    i ← l                             i = Position der nächsten nach a geschriebenen Zahl
    for j ← 1, ..., M                 durchlaufe alle Zähler
        if count[j] > 0 then          sobald ein Zähler j größer als 0 ist, ...
            for k ← 1, ..., count[j]  ... schreibe entsprechend viele Zahlen j nach a
                a[i] ← j
                i ← i + 1
```

Der Algorithmus bestimmt zunächst das Maximum M der Zahlen im Array a im Bereich zwischen den Positionen l und r. Wenn wir n Zahlen sortieren wollen, dauert das $\mathcal{O}(n)$. Anschließend wird ein Array **count** von M Zählern angelegt und mit dem Wert 0 initialisiert, ein Zähler für jede Zahl in der Menge $1, 2, 3, \ldots, M$ (Laufzeit $\mathcal{O}(M)$. Nun wird das Array a durchlaufen und für jede Zahl in a wird der entsprechende Zähler in **count** erhöht (Laufzeit $\mathcal{O}(n)$). Nun steht in **count** für jede Zahl, wie oft sie in a vorkommt. Am Schluss muss man nur noch durch das **count** -Array laufen und das Array a passend auffüllen. Die Laufzeit der beiden letzten geschachtelten Schleifen könnte man grob mit $\mathcal{O}(M \cdot n)$ abschätzen; da aber die innere Schleife nur genau n -mal durchlaufen wird, ist die Laufzeit der beiden Schleifen auch $\mathcal{O}(M + n)$. Insgesamt hat CountingSort damit eine Laufzeit von $\mathcal{O}(M + n)$, ist also linear in n .

Moment mal, wie ist das möglich? Haben wir nicht gerade gelernt, dass Sortieralgorithmen nicht in Linearzeit laufen können, sondern bestenfalls eine Laufzeit von $\mathcal{O}(n \cdot \log n)$ haben? Ja, aber das haben wir nur für *vergleichsbasierte* Sortierverfahren gezeigt. CountingSort ist aber gar kein vergleichsbasiertes Sortierverfahren; nirgends im Algorithmus gibt es einen Vergleich zwischen zwei Werten aus a .



CountingSort verwendet eine sehr einfache Form des »Hashings«: Das Array **count** ist eine Art Hashtabelle, nur dass dort nicht die Werte selbst gespeichert werden, sondern nur ihre Anzahl, und als Hashfunktion wird die »Identität«

hashCode $(x) = x$ verwendet. Mehr über Hashing erfahren Sie im Kapitel 9.

Von der asymptotischen Laufzeitklasse her ist CountingSort also formal allen bislang hier vorgestellten Sortieralgorithmen überlegen. Es hat allerdings mehrere große Nachteile. Zum einen kann es nur natürliche Zahlen sortieren. Würde man es zum Beispiel zum Sortieren von reellen Zahlen verwenden wollen, so hat man die schwierige Aufgabe zu lösen, die Menge von eventuell benötigten Zählern zu verwalten und immer den richtigen zu finden. Das Hauptproblem ist aber, dass sowohl Platzverbrauch als auch Laufzeit von der größten in a vorkommenden Zahl M abhängen. Das sorgt dafür, dass es nur dann sinnvoll ist, CountingSort einzusetzen, wenn M klein ist.

Ein anderer Algorithmus namens RadixSort löst das letzte Problem eleganter. Wie ihm das gelingt, werden Sie auf den nächsten Seiten erfahren.

Sortieren nach Ziffern

Zahlen schreiben man ja gewöhnlich als Reihen von Ziffern. Im Dezimalsystem sind das zehn verschiedene Ziffern zwischen 0 und 9. Beim Hexadezimalsystem nimmt man noch die Buchstaben A bis F dazu und hat dann 16 Ziffern und im Binärsystem verwendet man nur die Ziffern 0 und 1. Ob man nun für die »Basis« R (im Englischen sagt man dazu *radix*) 10, 16, 2 oder was auch immer wählt, spannend an Stellenwertsystemen ist in jedem Fall, dass man beliebig große Zahlen mit einer begrenzten Anzahl unterschiedlicher Ziffern darstellen kann. Das werden wir uns nun beim Sortieren zunutze machen. Das Problem von CountingSort ist ja, dass bei großen Zahlen der Platzbedarf für die Speicherung der Zähler so groß wie die größte Zahl M wird. Wenn man aber die Zahlen nur nach *einer einzelnen Ziffer* sortiert, so hängt der Platzbedarf nur noch von der Anzahl R der unterschiedlichen Ziffern ab und die Basis R kann man gerne so groß oder klein wählen, wie einem das gerade passend erscheint.

Lassen Sie uns die Ziffern durchzählen: Am weitesten rechts steht die erste Ziffer (»Einerstelle«), links daneben die zweite Ziffer (im Dezimalsystem die »Zehnerstelle«), noch eins weiter links die dritte Ziffer (im Dezimalsystem die »Hunderterstelle«) und so weiter. Wir wollen nun CountingSort so abwandeln, dass der Algorithmus als Sortierschlüssel die k-te Ziffer der Zahlen verwendet. Zunächst legen wir uns in einem Array **count** R Zähler an, mit denen wir zählen, wie viele Zahlen im Array a welche k-te Ziffer haben. Im Beispiel in Abbildung 7.14 ist **count** $[3] = 2$, weil es in a gerade die zwei Zahlen 331 und 030 gibt, die als zweite Ziffer eine 3 haben. Aus **count** kann man dann berechnen, an welchen Positionen die Zahlen stehen werden, sobald man sie nach der k-ten Ziffer sortiert hat. Im Beispiel ist **count** $[0] = 3$ ist, also werden die ersten drei Zahlen als zweite Ziffer eine 0 haben. Keine Zahl hat als zweite Ziffer eine 1 (**count** $[1] = 0$), dafür haben aber 4 Zahlen als zweite Ziffer eine 2 (**count** $[2] = 4$), das heißt, diese vier Zahlen kommen im sortierten Array als nächstes. Wir berechnen aus **count** also ein Array **pos** von R Positionen, die uns für jede der R möglichen Ziffern

angibt, wo die Zahl hingehört. Nun müssen wir nur noch durch das Array a laufen und die Zahlen entsprechend ihrer k -ten Ziffer einsortieren. Damit wir uns dabei keine Zahlen in a überschreiben, die wir noch nicht bearbeitet haben, tragen wir – ähnlich wie beim Algorithmus Merge – die Zahlen zunächst in ein Hilfsarray b ein. Sobald alle Werte von a nach b übertragen worden sind, kopieren wir b zurück nach a und sind fertig.

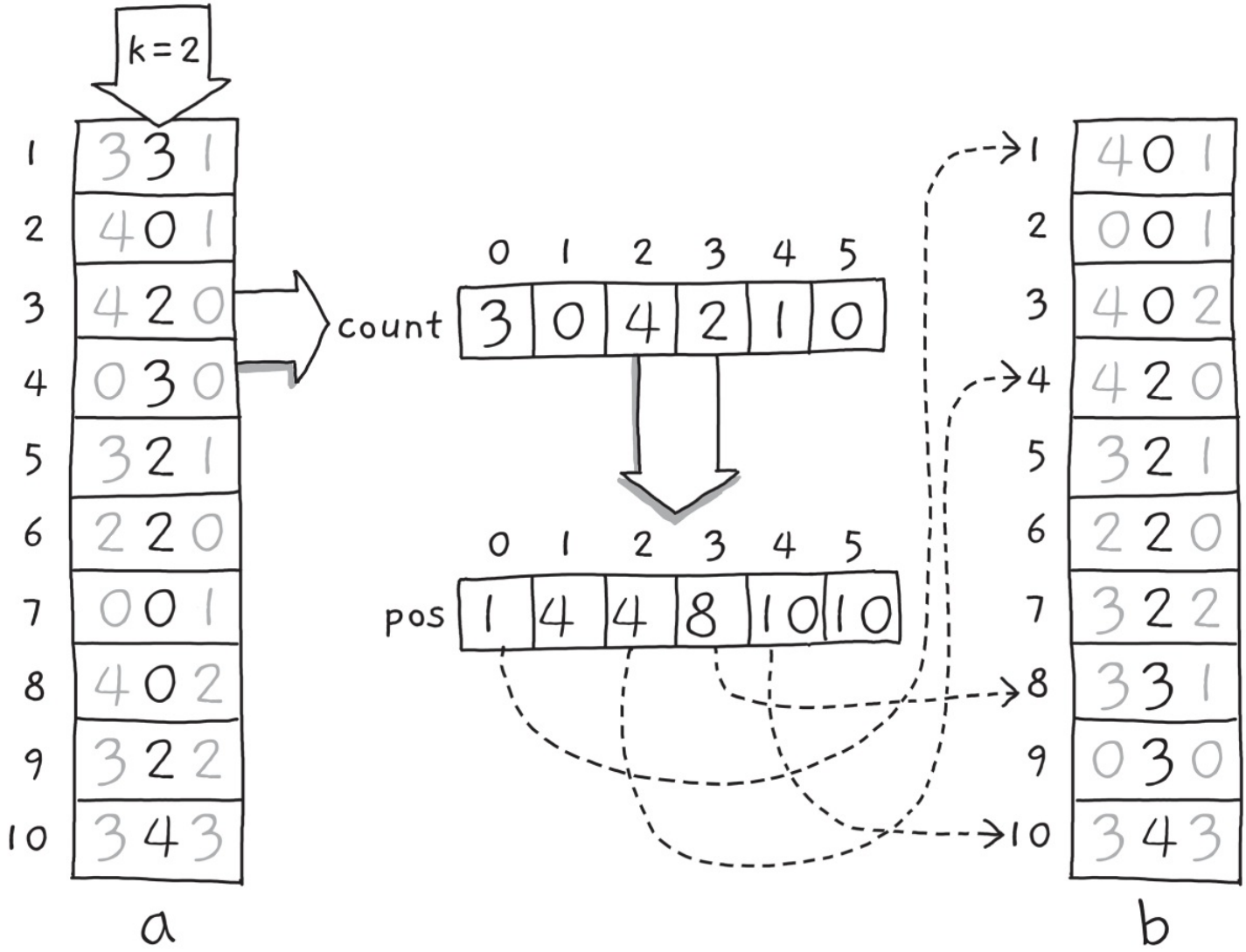


Abbildung 7.14: SortByDigit

```
SortByDigit(a, l, r, k)
    a: Array von n natürlichen Zahlen; l, r: Positionen in a, l ≤ r;
    k: Nummer der Ziffer, nach der sortiert werden soll
    for j ← 0, ..., R − 1
        count[j] ← 0                    initialisiere alle Zähler mit 0
    for i ← l, ..., r                   durchlaufe alle Zahlen im Array
        j ← die k−te Ziffer von a[i]
        count[j] ← count[j] + 1         zähle, wie oft welche Ziffer vorkommt
    i ← l
    for j ← 0, ..., R − 1
        pos[j] ← i                      pos[j]: Position für Zahlen mit Ziffer j
        i ← i + count[j]                Zähler aufsummieren
    b sei ein Hilfsarray
    for i ← l, ..., r
        j ← die k−te Ziffer von a[i]
        b[pos[j]] ← a[i]                Zahl an die richtige Position in b kopieren
        pos[j] ← pos[j] + 1             Position um eins weiterschieben
    for i ← l, ..., r
        a[i] ← b[i]                     Array b nach Array a kopieren
```

Der Algorithmus enthält 5 Schleifen, die aber nicht geschachtelt sind, sondern nacheinander ausgeführt werden. Die Laufzeit ist somit $\mathcal{O}(n + R)$, also linear in der Anzahl n der Zahlen in a.

Stabile Sortierverfahren

SortByDigit ist ein sogenanntes *stabiles Sortierverfahren*.



Ein Sortierverfahren heißt *stabil*, wenn es die Reihenfolge von Werten mit gleichem Sortierschlüssel bewahrt. Wenn man zum Beispiel eine Reihe von Studenten nach ihrem Nachnamen sortiert und »Hans Müller« im unsortierten Array irgendwo vor »Paula Müller« steht, dann garantiert ein stabiles Sortierverfahren, dass nach dem Sortieren Hans immer noch vor Paula im Array vorkommt, dass die beiden also ihre relative Reihenfolge im Array beibehalten.

Wenn ein Sortierverfahren *nicht* stabil ist, so darf es Werte mit gleichem Sortierschlüssel in eine beliebige Reihenfolge bringen. Im Beispiel könnte dann also im fertig sortierten Array Hans immer noch vor Paula im Array stehen oder auch umgekehrt Paula vor Hans. Das sind ja beides korrekte Sortierreihenfolgen, denn Hans und Paula haben den gleichen Nachnamen. Beispiele für stabile Sortierverfahren sind neben SortByDigit auch SelectionSort und MergeSort, wohingegen QuickSort und HeapSort nicht stabil sind.

Bei SortByDigit bedeutet Stabilität, dass SortByDigit die Reihenfolge von Zahlen mit der

gleichen k -ten Ziffer nicht verändert. Das können Sie sehr schön am Beispiel in Abbildung 7.14 erkennen: Im unsortierten Array **a** steht 401 an Position 2 vor 001 an Position 7 und danach kommt 402 an Position 8. Im sortierten Array **b** stehen dieselben Zahlen an den Positionen 1, 2 und 3 und sie haben dort die gleiche Reihenfolge wie zuvor.

Der nächste Algorithmus funktioniert nur deshalb, weil SortByDigit stabil ist.

RadixSort: Mehrfach ziffernweise sortieren

Wie Sie am Beispiel in Abbildung 7.14 leicht erkennen können, sind die Werte in b natürlich nicht sortiert. SortByDigit ordnet die Zahlen eben nur nach den k -ten Ziffern und nicht nach ihren Werten. Trotzdem ist der Algorithmus durchaus nützlich, wenn man natürliche Zahlen nach ihren Werten sortieren will. Dafür muss man SortByDigit nur mehrfach aufrufen, und zwar für jedes k einmal, angefangen mit der »niedrigstwertigen« Ziffer 1 bis hin zur »höchstwertigen« Ziffer k_{max} :

RadixSort(a, l, r) *a: Array von n natürlichen Zahlen; l, r: Positionen in a, $l \leq r$;*
    $k_{max} \leftarrow$ Anzahl der Stellen, die Zahlen in a haben
    for $k \leftarrow 1, \ldots, k_{max}$
        SortByDigit(a, l, r, k)

In Abbildung 7.15 sehen Sie an einem Beispiel, wie der Algorithmus funktioniert. Für das Verfahren ist es wichtig, dass SortByDigit ein stabiles Sortierverfahren ist. Greifen Sie einmal zufällig zwei unterschiedliche Zahlen in a heraus. Da die beiden Zahlen unterschiedlich sind, müssen sie sich in irgendeiner ihrer Ziffern unterscheiden, sagen wir einmal, das wäre die k -te Ziffer. Wir wählen das k möglichst groß, was bedeutet, dass die beiden Zahlen ab der $(k+1)$-ten Ziffer miteinander übereinstimmen. Da SortByDigit stabil ist, wird sich die relative Reihenfolge zwischen den Zahlen also ab $k+1$ nicht mehr ändern. Das ist gut so, denn SortByDigit hat bereits beim Sortieren nach der k -ten Stelle die beiden Zahlen in die richtige Reihenfolge gebracht.

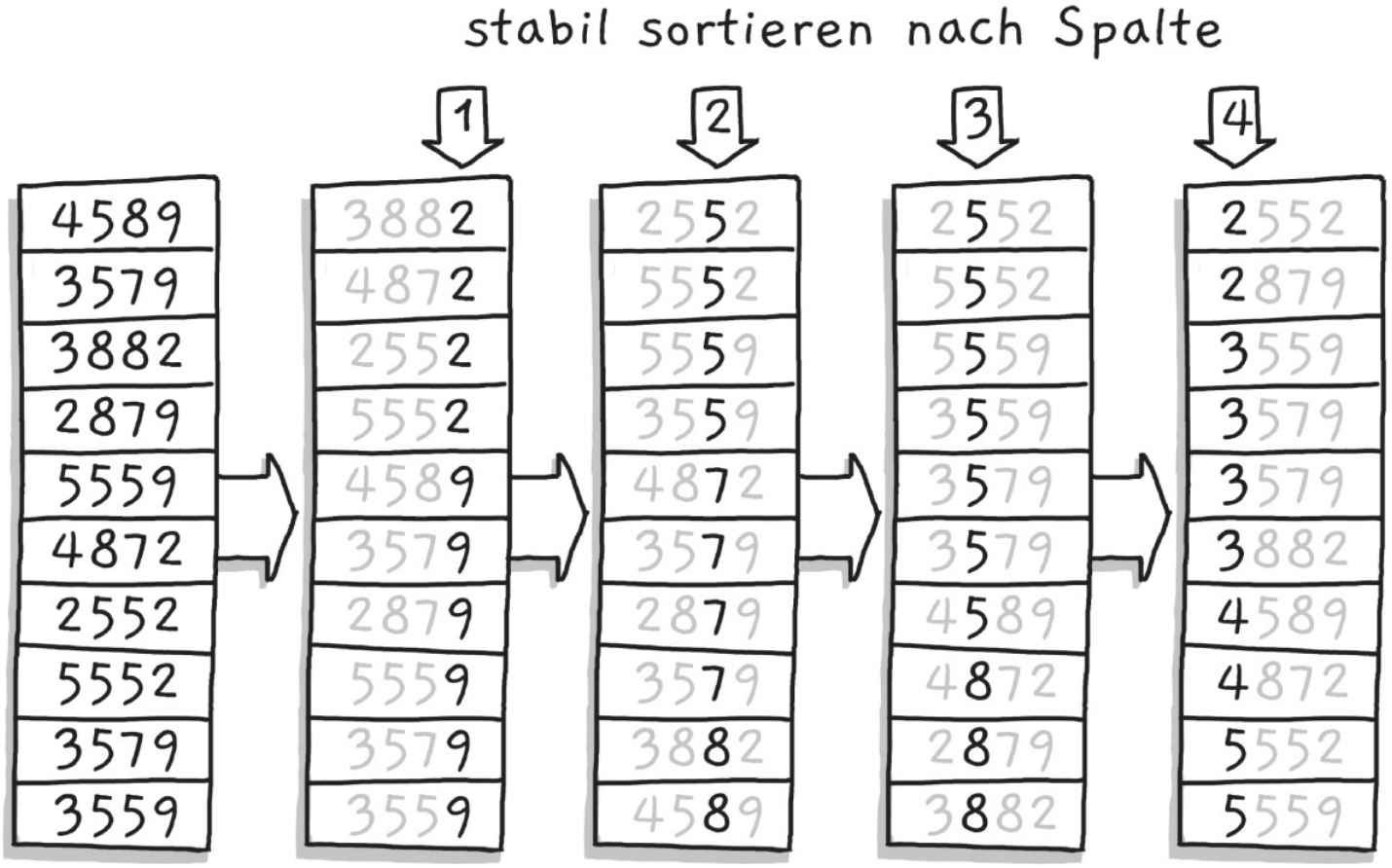


Abbildung 7.15: RadixSort

RadixSort ist ein Linearzeit-Sortieralgorithmus. Seine Laufzeit ist $\mathcal{O}(k_{max} \cdot (R+n))$ und der Platzbedarf $\mathcal{O}(R+n)$. Beides hängt also nicht wie BubbleSort (auf das wir gleich zu sprechen kommen) von der maximalen Zahl M im Array ab, die ja sehr groß sein kann. Wie BubbleSort ist RadixSort kein vergleichsbasiertes Sortierverfahren und ausschließlich auf das Sortieren ganzer Zahlen spezialisiert.

In der Praxis bietet es sich übrigens oft an, die Basis $R = 256$ zu wählen, dann lassen sich nämlich bei üblichen Rechnerarchitekturen die Ziffern von ganzen Zahlen ohne weitere Umstände als Bytes direkt im Speicher adressieren. Die üblichen 32-Bit-Integerzahlen wären dann ganz einfach vierstellige Zahlen zur Basis 256.

Weitere Sortieralgorithmen

Lassen Sie uns zum Schluss dieses Kapitels noch ein paar Sortieralgorithmen betrachten, die zwar nicht in der Oberliga der schnellsten Sortierverfahren mitspielen, also vor allem von akademischen Interesse sind, aber dennoch gerne in Vorlesungen oder Klausuren (!) drankommen.

BubbleSort: Nachbarn vertauschen

Bei BubbleSort handelt es sich um ein besonders einfaches Sortierverfahren, das

allerdings geradezu legendär langsam ist.



Als einmal der spätere amerikanische Präsident Barack Obama in einer Fernsehsendung spaßeshalber gefragt wurde, mit welcher Methode man am besten eine Millionen 32-Bit-Integerzahlen sortieren könne, zog er sich mit der süffisanten Antwort »I think the BubbleSort would be the wrong way to go« geschickt aus der Affäre. Sicherlich hat er mit seiner Vermutung recht, auch wenn völlig unklar ist, welche algorithmischen Details dieses Sortierverfahrens Herrn Obama tatsächlich geläufig sind. Sollten Sie jemals amerikanischer Präsident werden wollen, empfehlen wir Ihnen jedenfalls die Antwort »I think the RadixSort would be a good way to go«.

```
BubbleSort(a, l, r)                    a: Array von Werten; l, r: Positionen in a, l ≤ r
    do
        sorted ← true                  zeigt an, ob a fertig sortiert ist
        for i ← l, ..., r − 1          laufe durch das Array a
            if a[i] > a[i + 1] then        wenn zwei Nachbarn in falscher Reihenfolge stehen, ...
                vertausche a[i] mit a[i + 1]    ... bringe sie in die richtigen Reihenfolge
                sorted ← false         a war noch nicht fertig sortiert
    while not sorted                   wiederhole das so lange, bis a fertig sortiert ist
```

Der Algorithmus durchläuft das Array und sucht nach benachbarten Werten, die »falsch herum« im Array stehen, bei denen der vordere Wert also größer ist als der Wert dahinter. Wird ein solches Nachbarpaar gefunden, so werden die beiden Werte vertauscht. Der Algorithmus läuft wiederholt über das ganze Array, bis er keine Nachbarpaare mehr findet, die er vertauschen müsste.

Ein Worst-Case-Beispiel für BubbleSort wäre ein Array, in dem sich das Minimum an der letzten Position $a[r]$ befindet. Bei jedem Durchlauf der äußeren Schleife rückt dieser Wert nämlich nur eine Position nach vorne. Vermutlich hat der Algorithmus daher seinen Namen: Wie Blasen (*bubbles*) in einem Sektglas langsam aufsteigen, wandern die kleinen Werte Runde für Runde immer weiter im Array nach vorne. Das Minimum muss an die erste Position im Array; wenn es am Anfang ganz hinten war, so muss die äußere Schleife also $(n-1)$-mal durchlaufen werden, bevor das Array sortiert ist. Nach einem weiteren, abschließenden Durchlauf der äußeren Schleife endet der Algorithmus. Da die innere Schleife eine Laufzeit von $\mathcal{O}(n)$ hat, ergibt sich somit für BubbleSort insgesamt eine Laufzeit von $\mathcal{O}(n^2)$.

Natürlich kann man noch schlechtere Sortieralgorithmus konstruieren, doch BubbleSort ist in der Praxis schon ziemlich langsam und man sollte diesen Algorithmus darum nur verwenden, wenn man absolut zu faul ist, irgendetwas einigermaßen Performantes zu programmieren.

Gnomesort: Immer hin und her

Der folgende Algorithmus ist ein naher Verwandter von BubbleSort:

```
GnomeSort(a, l, r)                          a: Array von Werten; l, r: Positionen in a, l ≤ r
    i ← l                                   i: aktuelle Position in a
    while i ≤ r                             solange i noch im Array liegt:
        if (i = l) or (a[i − 1] ≤ a[i]) then    solange die Werte korrekt sortiert sind, ...
            i ← i + 1                           ... gehe eine Position weiter
        else
            vertausche a[i − 1] mit a[i]        ansonsten: korrigiere Reihenfolge der Werte ...
            i ← i − 1                           ... und gehe eine Position zurück
```

Anstatt wie BubbleSort immer wieder ganz über das Array zu laufen, läuft GnomeSort immer hin und her. Zunächst startet der Algorithmus an Position 1 und läuft vorwärts über das Array. Sobald er auf einen Wert stößt, der kleiner ist als sein Vorgänger, vertauscht er die beiden und geht eine Position zurück. In der Folge wird das kleinere Element so lange nach vorne verschoben, bis es nicht mehr kleiner als sein Vorgänger ist. Danach läuft GnomeSort wieder vorwärts und sucht das nächste falsch sortierte Element. Sobald der Algorithmus am Ende des Array herausläuft, ist er fertig.

GnomeSort ist vor allen Dingen deswegen interessant, weil seine Laufzeit auf den ersten Blick schlecht einzuschätzen ist. Im Algorithmus gibt es nur eine einzige Schleife und die Laufvariable i nimmt Werte zwischen 1 und n an. Dennoch ist die Worst-Case-Laufzeit von GnomeSort nicht, wie man vielleicht denken könnte, $\mathcal{O}(n)$, sondern $\mathcal{O}(n^2)$. Die Schleife kann nämlich für das gleiche i mehrfach durchlaufen werden. Worst-Case-Beispiel wäre eine Reihe von Werten, die absteigend sortiert sind, also zum Beispiel $a = [n, n-1, \ldots, 2, 1]$.

InsertionSort: Spielkarten dazwischen schieben

Den Algorithmus GnomeSort kann man mit ein paar Tricks deutlich optimieren. Statt einen Wert x mehrfach hintereinander mit seinem jeweiligen Vorgänger zu tauschen, ist es viel sinnvoller, zunächst alle diese Vorgänger um eine Position nach hinten zu verschieben und erst danach das x in die entstandene Lücke zu füllen. Auch ist es unnötig, anschließen wie GnomeSort noch einmal im Array vorwärts durch alle diese Vorgängerwerte zu laufen; stattdessen kann man gleich dort weitermachen, wo man zuvor den Wert x gefunden hat. Der auf diese Art angepasste Algorithmus heißt InsertionSort und er tut ungefähr das, was viele Leute machen, wenn Sie beim Skat oder Doppelkopf ihre Karten sortieren: Man nimmt die Karten nacheinander auf und steckt sie jeweils an die richtige Stelle zwischen die bereits sortierten Karten.

```
InsertionSort(a, l, r)                  a: Array von Werten; l, r: Positionen in a, l ≤ r
    for i ← l + 1, ..., r               durchlaufe das Array
        x ← a[i]                        bringe den Wert x an die richtige Position
        j ← i                           j: Position, an die x gehört
        while j > l and a[j − 1] > x    solange noch nicht angekommen, ...
            a[j] ← a[j − 1]             ... verschiebe Vorgängerwert nach rechts
            j ← j − 1                   suche links weiter
        a[j] ← x                        Position i gefunden: x eintragen
```

Vom Standpunkt der asymptotisch Laufzeit gesehen sind InsertionSort und GnomeSort gleich: Beide haben eine Laufzeit von $\mathcal{O}(n^2)$. Der Worst-Case-Fall wäre zum Beispiel ein absteigend sortiertes Eingabearray a.

 Eine funktionale Variante des Algorithmus InsertionSort finden Sie in Kapitel 3.

Kapitel 8

Rucksack packen

IN DIESEM KAPITEL

Rucksackproblem

Lineare Optimierung

Algorithmische Techniken bei der Arbeit: Gier, erschöpfende Suche, Backtracking und Branch & Bound, dynamisches Programmieren, Approximation

Wie man einen Koffer packt

Koffer sind eigentlich immer zu klein. Das weiß jeder, der schon einmal in den Urlaub gefahren ist. Hat man schließlich alles hineingequetscht, wartet spätestens bei der Gepäckaufgabe am Flughafen die böse Überraschung. Bei den üblichen Preisen für Übergewicht könnte man schon darüber nachdenken, ob am Ballermann wirklich eine zweite Garnitur Kleidung benötigt wird. Vielleicht reicht ja auch die Badehose? Dann wäre sicherlich noch Platz für ein paar zusätzliche Flaschen Sonnenöl.

Wenn Sie Ihre Freizeit lieber mit einem Computerspiel auf dem Sofa verbringen, kann Ihnen dort das gleiche Problem begegnen. Oft ist nämlich das Einsammeln (Fachbegriff: das »Looten«) diverser Schätze ein wichtiges Spielelement. Wenn die Gegner im Sekundentakt Gegenstände »droppen« (noch ein Fachbegriff), steht der Spieler schnell vor der Frage: Lohnt es sich, diese schwere Orkrüstung mitzuschleppen, für die der Händler sowieso kaum was bezahlt, oder wirft man den Müll lieber gleich weg und hält sich den Platz im »Inventar« für wertvollere Dinge frei?

In diesem Kapitel geht es also darum, aus einer Menge von Objekten, die man nicht alle haben kann, eine möglichst gute Auswahl zu treffen. Dieses Problem wird in der Literatur üblicherweise das »Rucksackproblem« (auf Englisch: »knapsack problem«) genannt.

Das Rucksackproblem

Der erste Schritt zur Lösung eines Problems besteht darin, es ganz genau – also formal – zu beschreiben: Angenommen, Sie können aus einer Menge M von Objekten auswählen. Jedes Objekt x aus der Menge M hat einen Wert (Nutzen) von $x.wert$ und ein Gewicht (Kosten) von $x.gewicht$. Aus diesen Objekten sollen Sie eine Teilmenge $S \subseteq M$ von Objekte bestimmen, deren Gesamtwert

$$\sum_{x \in S} x.wert \qquad \text{(die Summe über die Werte aller Objekte in } S\text{)}$$

maximal ist. Dabei dürfen Sie aber nur so viele Objekte auswählen, dass das Gesamtgewicht unter der maximalen Tragkraft T liegt:

$$\sum_{x \in S} x.gewicht \leq T \qquad \text{(Summe der Gewichte aller Objekte in } S \text{ darf höchstens } T \text{ sein)}$$

Um uns das Leben etwas einfacher zu machen, gehen wir davon aus, dass die Werte $x.wert$ und Gewichte $x.gewicht$ einschließlich der Tragkraft T des Rucksacks allesamt (positive) ganze Zahlen sind. Wir betrachten also das *ganzzahlige* Rucksackproblem (engl: *integer knapsack problem*). Sollten wir in der Praxis einmal mit einer Aufgabe konfrontiert sein, bei der der das nicht so ist, so können wir dieses Aufgabe in der Regel problemlos entsprechend umwandeln. Sind zum Beispiel die Werte keine ganzen Zahlen, wie 2,34 €, dann rechnet man eben in ganzen Cent, also 234 Cent. Allgemein gesagt: Man multipliziert einfach alle Werte beziehungsweise Gewichte einheitlich mit einer großen Zahl (dem Skalierungsfaktor, zum Beispiel 100). Sollten die Zahl dann noch immer nicht ganz sein, so rundet man das Ergebnis eben auf die nächste ganze Zahl; je größer der Skalierungsfaktor, desto genauer ist am Ende die Annäherung.



Beim Rucksackproblem hängt der Wert eines Objekts nicht davon ab, welche Objekte sonst noch eingepackt werden. Das kann im echten Leben auch anders sein. Lässt man zum Beispiel die Kamera zu Hause, so braucht man auch das dazugehörende Ladegerät nicht mitzunehmen. Umgekehrt könnte das Ladegerät durchaus nützlich sein, wenn man die Kamera in den Urlaub mitnimmt. Der Nutzen, den das Ladegerät für Sie hat, hängt also stark davon ab, ob Sie die Kamera mitnehmen oder nicht.

Mit einem kleinen Trick kann man auch in einen solchen Fall bei der bisherigen Formulierung des Rucksackproblems bleiben: Anstatt Kamera und Ladegerät als voneinander getrennte Objekte zu behandeln, fassen Sie die beiden einfach zu einem einzelnen Objekt »Kamera mit Ladegerät« zusammen.

Das Rucksackproblem ist ein echter Klassiker unter den algorithmischen Fragestellungen und es gibt ganze Bücher, die sich nur mit diesem Problem beschäftigen. Außerdem hat es eine ganze Reihe von Anwendungen, ein paar davon sind in Tabelle 8.1 aufgeführt.

Aufgabe	*Objekte*	*Kosten*	*Nutzen*
Rucksack/Koffer packen	Gegenstände	Gewicht	Gebrauchswert
im Computerspiel »looten«	Schätze	Platz im Inventar	Verkaufswert
Investitionen aussuchen	Investitionsobjekt	Investitionskosten	Gewinn
für eine Prüfung lernen	Prüfungsthema	Lernaufwand	Klausurpunkte

Warensortiment festlegen	Waren	Lagerplatz	Gewinn
Freizeit planen	Freizeitaktivitäten	Freizeit	Lustgewinn
Umzugswagen beladen	Möbel	Volumen	Wert

Tabelle 8.1: Einige Anwendungen des Rucksackproblems



Optimierungsprobleme sind Aufgaben, bei denen man nicht nur irgendeine, sondern *die beste* Lösung finden soll. Ein Beispiel dafür ist das Rucksackproblem.

Das Wichtigste zuerst einpacken

Ein naheliegender Lösungsansatz für das Rucksackproblem ist KnapsackGreedy: Man sucht aus allen Objekten zunächst das beste aus und prüft, ob es in den Rucksack passt, das heißt, ob das Gewicht des Rucksacks mit diesem Objekt kleiner oder gleich der Tragkraft T ist. Ist das der Fall, so packt man das Objekt in den Rucksack. Danach macht man das Gleiche mit dem zweitbesten Objekt, dann mit dem drittbesten und so weiter.

```
KnapsackGreedy(M, T)          M: Liste von n Objekten, T: maximale Tragkraft
    sortiere die Objekte in M absteigend nach ihrer Attraktivität
    S ← {}                    Menge der ausgewählten Objekte, am Anfang leer
    g ← 0                     Gesamtgewicht der ausgewählten Objekte, am Anfang 0
    for x ← M[1], ..., M[n]   gehe alle Objekte nacheinander durch
        if g + x.gewicht ≤ T  wenn das Gesamtgewicht nicht zu groß wird, ...
            S ← S ∪ {x}       ... dann füge x der Menge S hinzu ...
            g ← g + x.gewicht ... und addiere das Gewicht von x zum Gesamtgewicht
    return S                  S: eine Auswahl von Objekten aus M
```

Sobald man ein Objekt einmal eingepackt hat, gibt man es nicht mehr her. Darum nennt man diese Methode *greedy* – also »gierig«.

Bevor man die Objekte in M der Reihe nach durchgehen kann, muss man sie zunächst in eine Reihenfolge bringen, und zwar so, dass das erste Objekt $M[1]$ das beste oder attraktivste Objekt ist und das letzte Objekt $M[n]$ das schlechteste oder am wenigsten attraktive. Als Maß für die Attraktivität eines Objekts könnten Sie einfach seinen Wert verwenden, also:

$$\text{Attraktivität eines Objekts } x = x.wert$$

Solange alle Objekte gleich viel wiegen, klappt das auch prima. Bei unterschiedlich schweren Objekten kann diese Strategie allerdings furchtbar schiefgehen; ein Beispiel dafür sehen Sie in Abbildung 8.1.

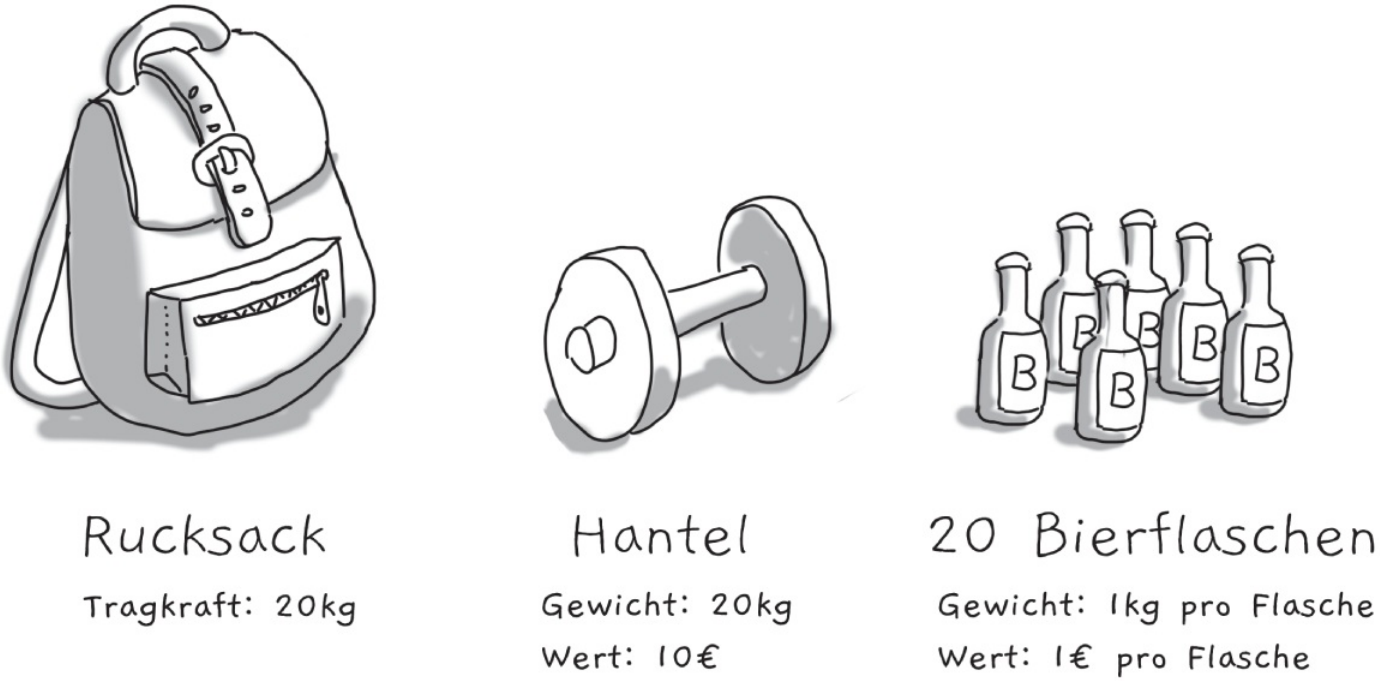


Abbildung 8.1: Wenn man die Hantel einpackt, ist für das Bier kein Platz mehr.

Da eine Hantel mehr wert ist als eine Bierflasche, packt KnapsackGreedy zuerst die Hantel ein (10 €). Danach ist der Rucksack voll, sodass keine der zwanzig Bierflaschen (jede gilt als einzelnes Objekt, Gesamtwert 20 €) mitgenommen werden kann. Das mag zwar gut für die Figur sein, ist aber finanziell wie partytechnisch gesehen ein herber Verlust.

Offenbar scheint es manchmal unklug zu sein, bei der Wahl nur auf den Wert und nicht auch auf das Gewicht zu achten. Stattdessen könnte man Objekte bevorzugen, die ein gutes Kosten-Nutzen-Verhältnis haben, also:

$$\text{Attraktivität eines Objekts } x = \frac{x.wert}{x.gewicht}$$

Leider funktioniert KnapsackGreedy auch mit dieser Auswahlmethode nicht immer perfekt, wie man am Beispiel in Abbildung 8.2 sehen kann: Da Gold einen höheren Wert pro Gewicht hat als Silber, packt KnapsackGreedy das Goldkörnchen (10 €) ein. Danach passt der Silberbarren (500 €) gerade nicht mehr in den Rucksack hinein.



Abbildung 8.2: Sobald das Goldkörnchen eingepackt ist, passt das Silber nicht mehr in den Rucksack.

Sie können sich gerne weitere Strategien ausdenken, wie man die Objekte in M in die »richtige« Reihenfolge bringen könnte, aber es wird ihnen vermutlich nicht gelingen, dass KnapsackGreedy in jedem Fall das optimale Ergebnis erzielt. KnapsackGreedy ist leider nur eine *Heuristik* für das Rucksackproblem, das heißt, es handelt sich um ein Verfahren, das oft eine gute, aber nicht unbedingt die beste Lösung liefert. Wie die Beispiele oben beweisen, liefert der Algorithmus manchmal sogar ausgesprochen schlechte Lösungen.

Das bedeutet jedoch nicht, dass KnapsackGreedy immer schlecht wäre. In bestimmten Spezialfällen liefert der Algorithmus stets das richtige Ergebnis, zum Beispiel wenn alle Objekte das gleiche Gewicht haben.

 Mehr zu Heuristiken lesen Sie im Kapitel 16.

Der Algorithmus KnapsackGreedy wendet das Prinzip der »Gier« an, das heißt, eine einmal von ihm getroffene Entscheidung nimmt er später nicht wieder zurück; mehr über gierige Algorithmen erfahren Sie ebenfalls in Kapitel 16.

In Kapitel 7 haben Sie erfahren, dass man n Gegenstände in einer Zeit $\mathcal{O}(n \log n)$ sortieren kann, zum Beispiel mit MergeSort. Insgesamt hat KnapsackGreedy damit ebenfalls eine Laufzeit von $\mathcal{O}(n \log n)$. Immerhin ist der Algorithmus also ziemlich schnell.

Alles ausprobieren

Wenn Sie sich wirklich sicher sein wollen, dass Sie die beste aller möglichen Lösungen für ein Problem gefunden haben, dann gibt es dafür eine ganz einfache Methode: Schauen Sie sich einfach *alle* möglichen Lösungen an, bewerten Sie jede davon und merken Sie sich immer die beste Lösung, die Sie bisher gefunden haben. Sobald Sie eine bessere finden, merken Sie sich also ab sofort diese. Die Lösung, die Sie sich am Ende gemerkt haben, ist die optimale Lösung. Weil sie alle Möglichkeiten ausschöpft, wird diese Methode »erschöpfende Suche« genannt. Sie wird in Kapitel 12 ausführlich behandelt. Da diese Strategie mit viel Aufwand, aber wenig Finesse vorgeht, hört sie auch auf den Namen »rohe Gewalt« (auf Englisch: *brute force*); darum soll der Algorithmus KnapsackBruteForce heißen:

```
KnapsackBruteForce(M, T)          M: Menge von Objekten, T: maximale Tragkraft
    S_opt ← {}                    die beste bislang gefundene Auswahl von Objekten
    w_opt ← 0                     der Gesamtwert der Objekte in S_opt
    for each S ⊆ M                untersuche nacheinander alle Teilmengen von M
        w ← Σ_{x∈S} x.wert        berechne Gesamtwert und -gewicht der Objekte in S
        g ← Σ_{x∈S} x.gewicht
        if (g ≤ T) and (w > w_opt)        wenn S in den Rucksack passt ...
                                              ... und wertvoller als S_opt ist, ...
            S_opt ← S             ... dann ist S ab sofort das neue S_opt
            w_opt ← w
    return S_opt
```

Alle Möglichkeiten durchzuprobieren, kann allerdings sehr lange dauern. KnapsackBruteForce muss sich schließlich alle Teilmenge S von M anschauen, und das sind leider ziemlich viele, sobald in der Menge M mehr als nur ein paar Objekte enthalten sind.



Bei nur einem Objekt $M = \{x_1\}$ kann $S = \{x_1\}$ oder $S = \{\}$ (die leere Menge) sein. Bei zwei Objekten $M = \{x_1, x_2\}$ gibt es schon vier Möglichkeiten: $S = \{x_1, x_2\}$, $S = \{x_1\}$, $S = \{x_2\}$ oder $S = \{\}$. Bei drei Objekten sind es acht, bei vier Objekten sechzehn Möglichkeiten und so weiter. Jedes weitere Objekt verdoppelt die Anzahl der verschiedenen Teilmengen S, die KnapsackBruteForce durchprobieren muss. Bei n Elementen, also $M = \{x_1, x_2, \ldots, x_n\}$ sind es 2^n Möglichkeiten. Für jede dieser Möglichkeiten berechnet der Algorithmus den Gesamtwert w und das Gesamtgewicht g; das dauert jeweils $\mathcal{O}(n)$. Insgesamt ist die Laufzeit von KnapsackBruteForce also $\mathcal{O}(n \cdot 2^n)$; der Algorithmus hat damit eine überexponentielle Laufzeit.

Stehen zum Beispiel $n = 100$ Objekte zur Auswahl, so müsste der Algorithmus über eine Quintillion Möglichkeiten durchspielen (das ist eine 1 mit 30 Nullen). Damit könnte man selbst einen Supercomputer schon mal ein paar Millionen Jahre beschäftigen. Trotzdem ist der Fall vielleicht doch nicht komplett hoffnungslos, denn wie Sie gleich sehen werden, lässt sich der Ablauf mit ein paar Tricks – und etwas Glück – noch erheblich beschleunigen.

Suchen im Entscheidungsbaum

Wie zählt man bei KnapsackBruteForce eigentlich alle möglichen Teilmengen S von M auf, wobei M die Gesamtmenge an verfügbaren Objekten ist? Am einfachsten geht das so: Zuerst entscheiden Sie, ob Sie das erste Objekt $M[1]$ der Menge M in den Rucksack packen wollen. Dann entscheiden Sie das für das zweite Objekt $M[2]$, dann für das dritte Objekt $M[3]$ und so weiter. Jede Entscheidungsmöglichkeit kann man sich als eine Art Abzweigung vorstellen, so wie es in Abbildung 8.3 dargestellt ist.

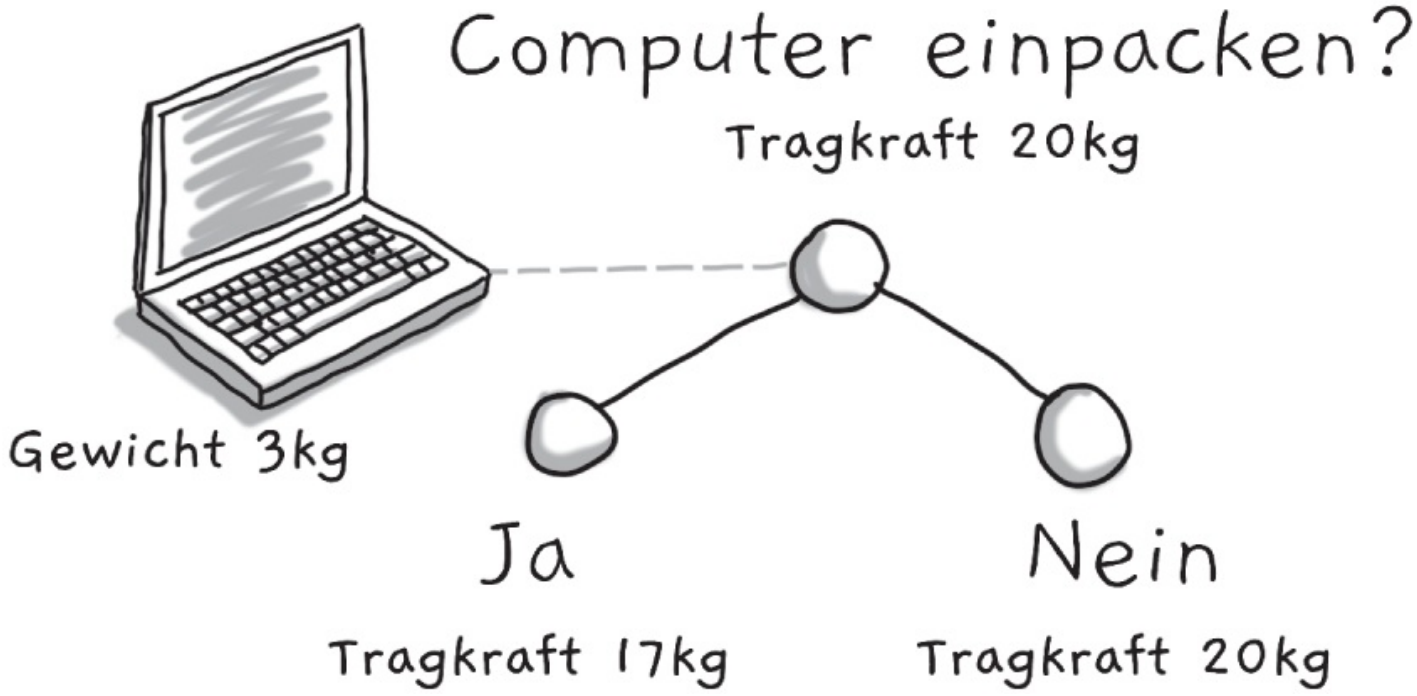


Abbildung 8.3: Eine Abzweigung im Entscheidungsbaum

Stellen Sie sich vor, Sie stehen bei Ihrer Auswahl am oberen Knoten und der Rucksack kann noch mit 20 kg befüllt werden. Nun müssen Sie entscheiden, ob Sie den Computer mitnehmen möchten. Wenn Sie ihn in den Rucksack packen wollen, so nehmen Sie den linken Abzweig (»Ja«). Dadurch sinkt die restliche Tragkraft des Rucksacks um das Gewicht des Computers, von 20 kg auf 17 kg. Andernfalls gehen Sie nach rechts (»Nein«). Sobald man einen der Wege genommen hat, steht man schon vor der nächsten Entscheidung für das nächste Objekt in der Menge. Zeichnet man das Ganze auf, so entsteht ein *Entscheidungsbaum* (Abbildung 8.4).

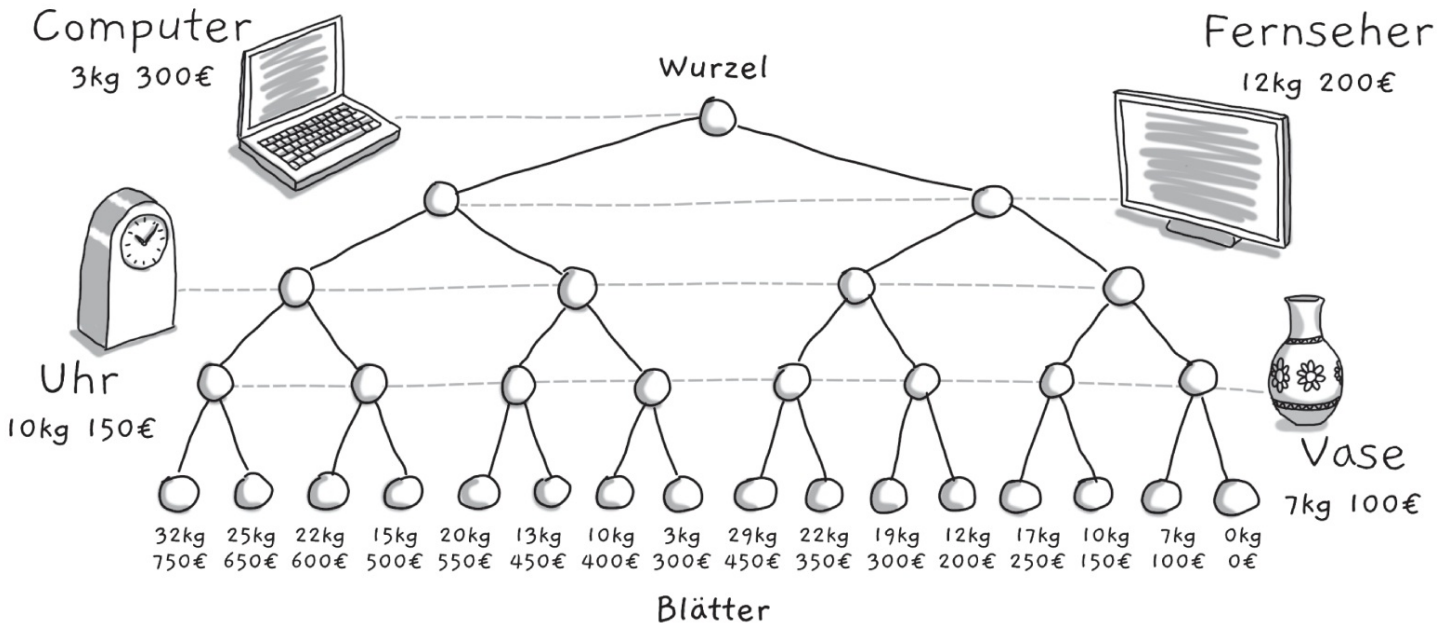


Abbildung 8.4: Ein Entscheidungsbaum



In der Informatik ist es üblich, die Bäume von oben nach unten »wachsen« zu lassen. Der Knoten ganz oben ist die Wurzel; nach unten hin verzweigt sich der Baum immer weiter, bis ganz unten die Blätter kommen. Diese etwas ungewöhnliche Orientierung ist allerdings eher nicht dadurch zu erklären, dass die Kollegen Informatiker einfach zu selten vor die Tür gehen, um sich dort über die korrekte Wuchsrichtung von Bäumen zu informieren. Die Wurzel eines Baumes ist sozusagen sein »Ursprung« oder »Anfang« und der Anfang gehört im Text nun mal ganz nach oben.

Um seine Auswahl zu treffen, beginnt man an der Wurzel, ganz oben im Baum, steigt nach und nach den Baum herunter und trifft auf jeder Ebene die Entscheidung über einen der Gegenstände, bis man schließlich ganz unten an einem Blatt des Baumes ankommt. Je nachdem, welche Entscheidungen man auf dem Weg von der Wurzel bis zum Blatt getroffen hat, ergeben sich am Blatt das Gesamtgewicht und der Gesamtwert der eingepackten Objekte.

Nun wissen Sie, wie man *eine* Auswahl aus den Objekten in M treffen kann: Man läuft einmal von der Wurzel bis zu einem Blatt. Aber wie zählt man *alle* Auswahlmöglichkeiten auf?

Die Antwort ist ziemlich einfach, wenn man das Prinzip der Rekursion beherrscht (ansonsten können Sie mehr zu diesem Thema in Kapitel 3 nachlesen). Wenn man an einer Abzweigung steht, zählt man zunächst alle Blätter auf, die man über den linken Abzweig erreichen kann, und dann alle Blätter, die man über den rechten Abzweig bekommen kann. Im linken Unterbaum liegen alle Kombinationen, bei denen man das aktuelle Objekt in den Rucksack gepackt hat, und im rechten Unterbaum alle Kombinationen, in denen dieses Objekt fehlt. Für die Aufzählung der Kombinationen in

den Unterbäumen sind jeweils rekursive Aufrufe zuständig. Der Algorithmus sieht dann so aus:

```
S_opt ← {}                          beste bislang gefundene Auswahl, globale Variable
w_opt ← 0                           Gesamtwert der Objekte aus S_opt, globale Variable
KnapsackRek(M, T) M: Menge von Objekten, T: maximale Tragkraft
    SearchTreeRek(M, T, {}, 0)      traversiere den Suchbaum rekursiv
    return S_opt

SearchTreeRek(M, T, S, w)
    M: restliche Menge von wählbaren Objekten, T: restliche Tragkraft
    S: bislang ausgewählte Objekte, w: Gesamtwert der Objekte in S
    if M = {}                       eine neue Auswahl S wurde fertiggestellt
        if w > w_opt                ist S besser als S_opt?
            S_opt ← S               ab sofort ist S das neue S_opt
            w_opt ← w
    else
        Wähle ein Objekt x ∈ M aus
        M´ ← M \ {x}                x aus M entfernen
        if x.gewicht ≤ T            Option 1: x einpacken
            T´ ← T − x.gewicht      Gewicht des Objekts x von der Tragkraft abziehen
            S´ ← S ∪ {x}            x der Menge S hinzufügen
            w´ ← w + x.wert
            SearchTreeRek(M´, T´, S´, w´)
        SearchTreeRek(M´, T, S, w)  Option 2: x nicht einpacken
```

Zu Anfang sind M die Gesamtmenge der verfügbaren Objekte und T die vollständige Tragkraft des Rucksacks. KnapsackRek wählt ein Objekt $x \in M$ aus und packt es einmal ein (Option 1) und einmal nicht (Option 2). Die weitere Suche im Suchbaum besteht aus rekursiven Aufrufen, bei denen jeweils das Objekt x aus M entfernt wird. Die Rekursion endet, wenn der Algorithmus zu allen Objekten eine Entscheidung getroffen hat, das heißt, wenn die Variable M keine weiteren Objekte mehr enthält. Jedes Mal wenn das passiert, hat der Algorithmus ein neues Blatt des Suchbaums erreicht. Ist die Auswahl S von Objekten, die diesem Blatt entspricht, besser als die wertvollste bisher gefundene Auswahl S_{opt}, so merkt sich der Algorithmus das, indem er die globalen Variablen S_{opt} und w_{opt} entsprechend anpasst. Wenn der Algorithmus endet, steht in der globalen Variablen S_{opt} die optimale Auswahl von Objekten aus M.

Welche Laufzeit hat KnapsackRek? Im schlimmsten Fall wird der Algorithmus für jeden Knoten im Suchbaum einmal aufgerufen. Das geschieht zum Beispiel, wenn der Rucksack so groß ist, dass in ihn alle Objekte gleichzeitig hinein passen. Wenn man alles mitnehmen

kann, dann ist das natürlich die lukrativste Option. KnapsackRek weiß das aber nicht und so probiert der Algorithmus alle Fälle durch. Insgesamt hat der Suchbaum bei n Objekten 2^n Blätter und $2^n - 1$ innere Knoten, das macht also $2^{n+1} - 1$ Aufrufe von KnapsackRek. Abgesehen von den rekursiven Aufrufen braucht jeder Aufruf nur konstant viel Zeit. Damit ergibt sich also eine Gesamtlaufzeit von $\mathcal{O}(2^{n+1} - 1) = \mathcal{O}(2^n)$. Schaut man sich in Abbildung 8.5 an, in welcher Reihenfolge die Knoten des Suchbaums bearbeitet werden, so wird klar, dass der Algorithmus den Suchbaum mit einer »Tiefensuche« traversiert (siehe auch Kapitel 10).

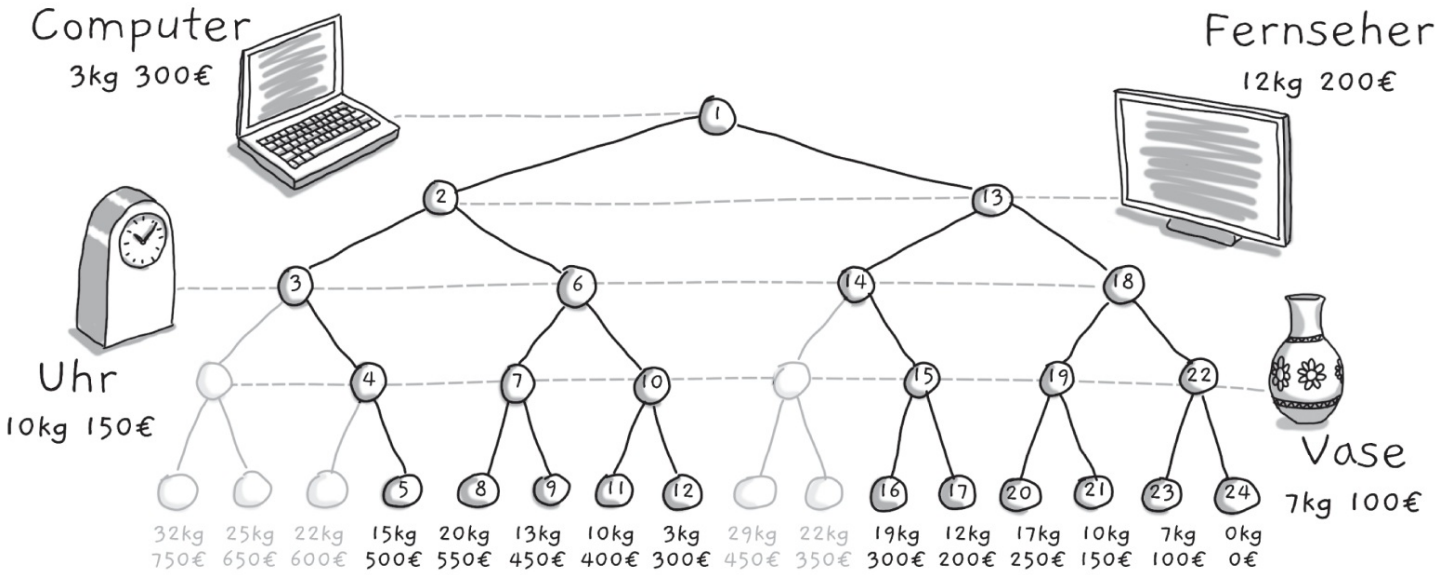


Abbildung 8.5: Tiefensuche durch den Entscheidungsbaum mit KnapsackRek

An dieser Grafik können Sie noch etwas anderes erkennen: Der Algorithmus spart nämlich systematisch alle Blätter aus, an denen das Gesamtgewicht größer als die maximale Tragkraft T ist. Hat er zum Beispiel sowohl den Computer (3 kg) als auch den Fernseher (12 kg) eingepackt, so passt die Uhr (10 kg) nicht mehr in den Rucksack. Der Algorithmus muss diesen Fall also gar nicht mehr berücksichtigen und darf alle Blätter ignorieren, die unter dem entsprechenden Knoten im Suchbaum liegen. KnapsackRek erreicht dies dadurch, dass das ausgewählte Objekt x nur dann in den Rucksack gepackt wird (Option 1), wenn sein Gewicht $x.gewicht$ kleiner als die restliche noch verfügbare Tragkraft T des Rucksacks ist. Dies führt dazu, dass der Algorithmus alle in der Grafik grau gedruckten Knoten komplett von der Suche ausspart.

Wenn man Glück hat, lässt sich durch diesen Trick der Suchraum stark einschränken und der Algorithmus entsprechend beschleunigen. Wie sich der Suchbaum noch weiter beschneiden lässt, werden Sie gleich erfahren.

Den Suchraum begrenzen

Schauen Sie sich noch einmal das Beispiel für KnapsackRek in Abbildung 8.5 an. Irgendwann einmal (am Knoten 8) sind Computer, Uhr und Vase in den Rucksack gepackt

und damit Objekte im Gesamtwert von 550 €. Das ist der höchstmögliche Betrag, besser geht es nicht. Eigentlich könnte man also den Algorithmus nun abbrechen. Das Problem ist nur, dass der Algorithmus zu diesem Zeitpunkt noch gar nicht weiß, dass er das Optimum bereits gefunden hat. Was er aber weiß, ist, dass das Optimum mindestens bei 550 € liegt.

Später kommen Sie dann bei Knoten 13 an, bei dem Sie den Fernseher nicht eingepackt haben. Nun ist der Wert aller restlichen Objekte zusammen (also Fernseher, Uhr und Vase) gerade einmal 300 €. Selbst wenn Sie alle diese Objekte in den Rucksack packen würden (was nicht geht, weil sie zusammen 29 kg wiegen, der Rucksack aber für maximal 20 kg ausgelegt ist), könnten Sie den bereits zuvor gefundenen Wert von 550 € nicht erreichen. Alle Blätter, die unterhalb von Knoten 13 liegen, müssen Sie sich also gar nicht mehr anschauen.

Aus dieser Beobachtung lässt sich folgende Strategie formulieren, die den Namen »Verzweigen und Begrenzen« (engl. *branch and bound*) hat: Während Sie den Suchbaum durchlaufen, merken Sie sich immer den Wert w_{opt} der besten bisher gefundenen Auswahl S_{opt}. Wenn Sie dann an einem Knoten ankommen, prüfen Sie zunächst, ob unterhalb dieses Knotens überhaupt Blätter liegen können, die mehr wert sind als w_{opt}. Ist das nicht der Fall, müssen Sie dort gar nicht herumsuchen. Der folgende Algorithmus zeigt, wie man diese Idee umsetzen könnte:

```
S_opt ← {}                          beste bislang gefundene Auswahl, globale Variable
w_opt ← 0                           Gesamtwert der Objekte aus S_opt, globale Variable
KnapsackBnB(M, T) M: Menge von Objekten, T: maximale Tragkraft
    SearchTreeBnB(M, T, {}, 0)      traversiere den Suchbaum rekursiv
    return S_opt

SearchTreeBnB(M, T, S, w)
    M: restliche Menge von wählbaren Objekten, T: restliche Tragkraft
    S: bislang ausgewählte Objekte, w: Gesamtwert der Objekte in S
    if M = {}                       eine neue Auswahl S wurde fertiggestellt
        if w > w_opt                ist S besser als S_opt?
            S_opt ← S               ab sofort ist S die neue S_opt
            w_opt ← w
    else
        w_rest ← Σ_{x∈M} x.wert     Gesamtwert aller restlichen Objekte
        if w + w_rest ≤ w_opt
            return                  S kann niemals besser als S_opt werden

        Wähle ein Objekt x ∈ M aus
        M´ ← M \ {x}                x aus M entfernen
        if x.gewicht <= T           Option 1: x einpacken
            T´ ← T − x.gewicht      Gewicht des Objekts x von der Tragkraft abziehen
            S´ ← S ∪ {x}            x der Menge S hinzufügen
            w´ ← w + x.wert
            SearchTreeBnB(M´, T´, S´, w´)
        SearchTreeBnB(M´, T, S, w)  Option 2: x nicht einpacken
```

Im Vergleich zu KnapsackRek sind nur zwei Zeilen dazugekommen: Der Algorithmus berechnet nun zunächst die Summe w_{rest} der Werte aller restlichen Objekte in M. Die Summe $w + w_{rest}$ aus den Gesamtwerten der aktuell eingepackten Objekte in S und im fraglichen Unterbaum ist eine obere Schranke (Englisch: *bound*) für die Werte aller Blätter unterhalb des aktuellen Knotens. Wenn diese Schranke nicht größer ist als der Wert w_{opt} der bislang besten gefundenen Auswahl S_{opt}, dann lohnt sich die Weitersuche im aktuellen Teilbaum nicht mehr und der aktuelle Rekursionsaufruf wird abgebrochen.

Die mögliche Zeitersparnis kann man im Beispiel bewundern (Abbildungen 8.4 bis 8.6): Während der komplette Suchbaum noch aus 31 Knoten (davon 16 Blättern) besteht, werden von KnapsackRek nur noch 24 Knoten (davon 11 Blätter) besucht und KnapsackBnB muss sich nur nocsh 9 Knoten (davon 3 Blätter) anschauen. Allerdings

muss KnapsackBnB in jedem Schritt eine Schranke w_{rest} berechnen, was natürlich zusätzliche Zeit kostet. Trotzdem lohnt sich *branch and bound* in der Praxis oft ungemein und man kann damit Probleme lösen, die mit einer einfachen erschöpfenden Suche kaum zu bewältigen wären.

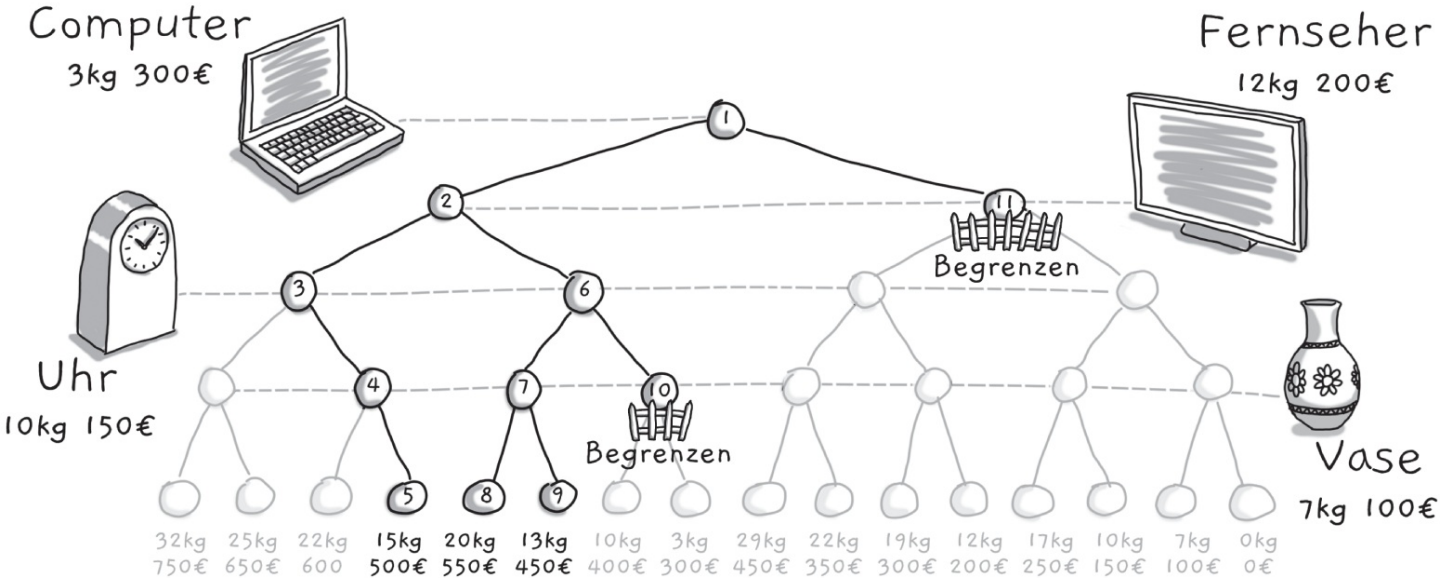


Abbildung 8.6: Tiefensuche durch den Entscheidungsbaum mit KnapsackBnB

Wenn man allerdings Pech hat, dann bringen alle Strategien zur Einschränkung des Suchraums nichts. *branch and bound* zum Beispiel ist darauf angewiesen, dass man schon möglichst bald eine möglichst gute Lösungen findet. Sonst bleibt w_{opt} nämlich zu klein, um größer als die Schranke $w + w_{rest}$ zu werden, und KnapsackBnB muss sich am Ende doch alle Knoten anschauen, die auch KnapsackRek durchlaufen hätte. Es wäre zum Beispiel ziemlich blöd, wenn man im Baum ganz oben (also an der Wurzel) über ein Objekt entscheiden müsste, das viel wiegt, aber nur wenig wert ist. Bei einem solchen Objekt lohnt sich vermutlich das Einpacken nicht; dennoch durchsucht KnapsackBnB zunächst den ganzen linken Teilbaum, in dem nur Kombinationen vorkommen, bei dem dieses Objekt den Rucksack vollmüllt.

Damit man bei der Tiefensuche durch den Entscheidungsbaum möglichst bald bei Blättern mit hohem Gesamtwert ankommt, wäre es also ratsam, wenn die Objekte, die vermutlich am Ende bei der besten Auswahl dabei sind, möglichst weit oben im Suchbaum stehen. Da man allerdings am Anfang nicht weiß, welche das sind, bietet sich eine Heuristik an, die Sie schon bei KnapsackGreedy kennengelernt haben. Man könnte zum Beispiel zunächst die Objekte nach ihrem Wert pro Gewicht sortieren und in dieser Reihenfolge KnapsackBnB die Objekte auswählen lassen. Dieser Trick steigert die Chance, dass KnapsackBnB schnell zu einem Ergebnis kommt, gibt darauf allerdings keine Garantie.

Die Worst-Case-Laufzeit aller bisher vorgestellten Suchalgorithmen KnapsackBruteForce, KnapsackRek und KnapsackBnB wächst (mindestens) exponentiell mit der Anzahl n verfügbarer Objekte. Stehen viele Objekte zur Auswahl,

so kann der Aufwand also ganz enorm ansteigen.



Tatsächlich ist es bislang noch niemandem gelungen, einen Algorithmus zur Lösung des Rucksackproblems zu finden, der in polynomieller Zeit läuft. Das Rucksackproblem gehört nämlich zu den NP-schweren Problemen. Was das genau bedeutet, steht in Kapitel 11.

Probleme langsam wachsen lassen

In einigen Fällen ist es trivial, die optimale Lösung für das Rucksackproblem zu finden, zum Beispiel wenn es gar keine Objekte gibt, also wenn $n = 0$ ist oder wenn die maximale Tragkraft $T = 0$ ist. In beiden Fällen kann man nämlich überhaupt nichts in den Rucksack packen – und ist darum auch sofort mit dem Packen fertig. Als grobe Faustregel könnte man sagen: »Je größer die Anzahl n der auswählbaren Objekte, desto aufwendiger wird die Lösung für das Rucksackproblem.«

Auch ein Weltmeister im Gewichtheben hat früher einmal mit leichten Hanteln trainiert, bevor er sich an die ganz schweren Gewichte gewagt hat. Wenn Ihnen also die großen Probleme (mit vielen Objekten) zu schwierig sind, dann probieren Sie es doch mal so: Fangen Sie erst einmal mit etwas Leichtem an und lösen Sie ein kleineres Teilproblem. Wenn Sie das geschafft haben, vergrößern Sie das Problem nach und nach, bis Sie schließlich das vollständige Probleme lösen können. Der Trick dabei: Zur Lösung des nächstgrößeren Problems benutzen Sie immer die Erkenntnisse aus den Lösungen der etwas kleineren Probleme.



Diese Methode wird »dynamisches Programmieren« genannt und ausführlich in Kapitel 13 erklärt.

Angenommen, Sie können schon das Rucksackproblem für die Menge $M = \{\text{Computer, Fernseher, Uhr}\}$ lösen und nun wollen Sie auch das Rucksackproblem für die um ein Objekt größere Menge Computer, Fernseher, Uhr, Vase lösen. Für die Vase gibt es wie immer zwei Möglichkeiten:

1. Sie packen die Vase ein. Wenn anschließend im Rucksack noch Platz ist, können Sie weitere Objekte aus M mit dem Gesamtgewicht von $T - Vase.gewicht = T - 7$ kg einpacken.
2. Sie packen die Vase *nicht* ein. Dann bleibt der komplette Rucksack frei für Objekte aus M.

In beiden Fällen können Sie ausrechnen, welchen Gesamtwert man bestenfalls in den Rucksack packen kann, und zwar indem Sie jeweils ein (kleineres) Rucksackproblem für die Objektmenge M lösen, einmal für die komplette Tragkraft T und einmal für die um 7 kg reduzierte Tragkraft.

Bevor wir das jetzt als Algorithmus hinschreiben können, brauchen wir noch ein wenig Notation. Unser Rucksack soll die maximale Tragkraft T haben und wir wählen aus einer Menge M mit n Objekten aus. Um das Problem zu verkleinern, beschränken wir uns auf eine Tragkraft t mit $0 \leq t \leq T$ und benutzen nur die ersten i Objekte aus M, wobei i irgendwo zwischen 0 und n liegt ($0 \leq i \leq n$). Lassen Sie uns die Ergebnisse für jedes t und jedes i in einer Tabelle speichern:

Tab [t , i] = »Der maximale Gesamtwert von Objekten, die sich in einen Rucksack mit der maximalen Tragkraft t packen lassen, wenn man nur die ersten i Objekte auswählen darf.«

Hinter »**Tab**« schreiben wir also in eckigen Klammern zwei Werte: Als erstes die Tragkraft des Rucksacks t – sie entspricht einer Zeile in der Tabelle – und als zweites die Anzahl i der Objekte aus M, aus der wir auswählen dürfen – das entspricht einer Spalte in **Tab**. Abbildung 8.7 zeigt **Tab** für unser Beispiel. Am Ende interessiert uns natürlich **Tab**[T , n], aber ehe wir diesen Wert berechnen können, müssen wir erst einmal die komplette Tabelle ausgefüllt haben. Angenommen, das i -te Objekt in M (wir schreiben kurz: $M[i]$) ist die Vase mit dem Gewicht $M[i].gewicht = 7$ kg und dem Wert M [i] $.wert = 100$ €. Betrachten Sie wieder die beiden Möglichkeiten:

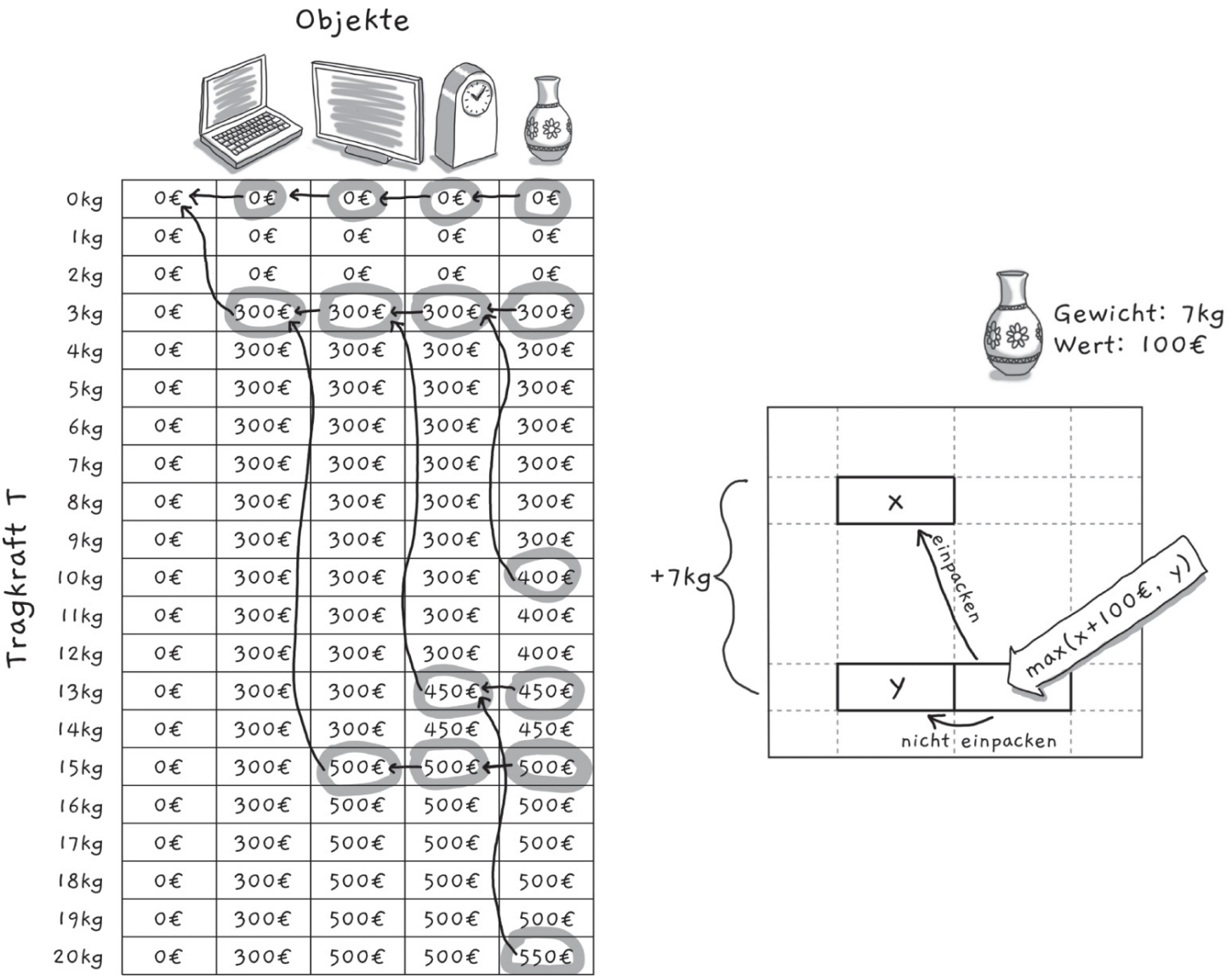


Abbildung 8.7: Die Tabelle **Tab** (links) und wie sie berechnet wird (rechts)

1. Sie packen die Vase ein. Das geht natürlich nur, wenn auch genug Platz im Rucksack ist, also wenn $t \geq 7$ kg. Den restlichen Platz von $t - 7$ kg im Rucksack können Sie dann mit einer Auswahl aus den ersten $i - 1$ Objekten aus M füllen. Das bedeutet: Sie können zusätzlich zu den 100 € für die Vase noch Objekte im Gesamtwert von **Tab** $[t - 7, i - 1]$ in den Rucksack packen.
2. Sie packen die Vase *nicht* ein. Stattdessen füllen Sie den Rucksack mit einer Auswahl aus den ersten $i - 1$ Objekten aus M und können also maximal Objekte im Gesamtwert von **Tab** $[t, i - 1]$ einpacken.

Ob wir nun das i -te Objekt $M[i]$ in den Rucksack packen oder nicht, hängt davon ab, welcher der beiden Fälle lukrativer ist. Das heißt: Wir packen $M[i]$ ein, wenn das Objekt erstens in den Rucksack passt und zweitens der Wert in Fall 1 höher ist als in Fall 2. In Pseudocode ausgedrückt, liest sich das so:

```
TakeObject(i, t)                    prüft, ob das Objekt M[i] eingepackt werden sollte
                                    i: Objektnummer, t: Tragkraft
  if M[i].gewicht ≤ t               M[i] würde in den Rucksack passen
    rein ← Tab[t − M[i].gewicht, i − 1] + M[i].wert    bester Wert mit M[i]
    raus ← Tab[t, i − 1]            bester Wert ohne M[i]
    if rein > raus
      return true                   M[i] einzupacken lohnt sich
  return false                      M[i] einzupacken geht nicht oder lohnt sich nicht
```

In der Tabelle auf Abbildung 8.7 sehen Sie für ein paar interessante Felder (die grau eingekringelten), aus welchen anderen Feldern sich der Wert jeweils ergeben hat. Wenn der Pfeil auf den linken Nachbarn zeigt, dann ist »nicht einpacken« die bessere Wahl; zeigt der Pfeil hingegen auf ein Feld ein paar Zeilen weiter oben, so lohnt es sich, das Objekt einzupacken.

Die Lösung für das Gesamtproblem steht im Feld $M[T, n]$ = 550 €, ganz unten rechts in der Tabelle. Wenn man nun wissen möchte, welche Gegenstände zu diesem Gesamtwert beigetragen haben, so muss man von dort aus nur den Pfeilen bis ganz nach links folgen. Diesen Pfad rückwärts durch **Tab** nennt man »Backtrace«. In unserem Beispiel sind die gewählten Optionen: Die Vase »einpacken«, die Uhr ebenfalls »einpacken«, den Fernseher »nicht einpacken« und den Computer »einpacken«. Tatsächlich müssen wir die Pfeile gar nicht abspeichern, denn wir können ja an jeder Position in der Tabelle mit TakeObject prüfen, ob wir das Objekt einpacken sollen oder nicht.

```
KnapsackDP(M, T)          M: Menge von n Objekten, T: maximale Tragkraft
  – PHASE 1: Tab ausfüllen –
  for t ← 0, ..., T
      Tab[t, 0] ← 0                Initialisierung Spalte ganz links
  for i ← 1, ..., n                fülle Tab spaltenweise von links nach rechts
      Tab[0, i] ← 0                Initialisierung das Feld ganz oben
      for t ← 1, ..., T            berechne die Felder der Spalte i von oben bis unten
          if TakeObject(i, t)
              Tab[t, i] ← Tab[t − M[i].gewicht, i − 1] + M[i].wert
          else
              Tab[t, i] ← Tab[t, i − 1]
  – PHASE 2: Backtrace –
  S ← {}                           Auswahl an Objekten
  t ← T                            restliche Tragkraft
  for i ← n, ..., 1                folge dem »Pfad« von rechts nach links
      if TakeObject(i, t)
          S ← S ∪ {M[i]}           M[i] in den Rucksack packen
          t ← t − M[i].gewicht     restliche Tragkraft berechnen
  return S
```

Die Laufzeit von KnapsackDP ergibt sich aus der Größe von **Tab** und ist $\mathcal{O}(n \cdot T)$. Wir haben hier also einen Algorithmus, der – anders als KnapsackGreedy – garantiert die optimale Lösung ermittelt und dessen Laufzeit – anders als die Suchalgorithmen KnapsackBruteForce, KnapsackRek und KnapsackBnB – nicht mehr exponentiell von der Anzahl n der Objekte abhängt. Hurra!



Pseudopolynomielle Algorithmen

Leider zählt KnapsackDP dennoch nicht offiziell zu den Algorithmen mit polynomieller Laufzeit, und zwar, weil seine Laufzeit von T abhängt. Denn es ist in der Informatik üblich, die Laufzeit von Algorithmen in Abhängigkeit von der Eingabelänge zu betrachten. Bei KnapsackDP ist aber nun weniger die Eingabe*länge* entscheidend, als vielmehr der Eingabe*wert* T. Nun kann man ziemlich große Zahlen ziemlich kurz hinschreiben. Wenn zum Beispiel der Eingabe*wert* $T = 1\ 000\ 000$ ist, dann ist die Eingabe*länge* nur 7 Dezimalziffern lang oder auch 20 Bits, wenn man T als Binärzahl 11110100001001000000 kodiert. So kommt es, dass die Laufzeit von KnapsackDP zwar linear bezüglich des Eingabe*werts* T, aber exponentiell bezüglich der Eingabe*länge* ist.

Wenn ein Algorithmus wie KnapsackDP eine Laufzeit hat, die zwar polynomiell mit einem Eingabe*wert*, aber nicht polynomiell mit der Eingabe*länge* wächst, dann ist sie »pseudopolynomiell«.

Die Abhängigkeit von T ist der große Nachteil von KnapsackDP. Stellen Sie sich vor, wir würden das Gewicht der Objekte nicht mehr in Kilogramm, sondern in Gramm messen, das heißt, die Zahlenwerte sowohl aller Objektgewichte als auch der maximalen

Tragkraft des Rucksacks würden um den Faktor 1000 anwachsen. Das Problem würde sich dadurch letztlich nicht ändern, für KnapsackDP wäre das allerdings ungünstig, denn sowohl die Laufzeit als auch der Speicherbedarf des Algorithmus würde ebenfalls auf das 1000-Fache anwachsen. **Tab** wäre dann allerdings extrem redundant, denn fast jede Zeile in der Tabelle sähe genauso aus wie die vorangegangene Zeile.

Wachsende Probleme klug speichern

Um den Algorithmus KnapsackDP zu verbessern, sollten wir uns also eine andere Speichermethode überlegen; vielleicht lässt sich dadurch auch die Laufzeit verbessern.

Schauen Sie sich noch einmal genau die letzte Spalte von **Tab** in Abbildung 8.7 an. Die mit der Vase. Da kommen gar nicht so viele unterschiedliche Werte vor. Alle Objekte wiegen mindestens 3 kg, wenn der Rucksack also weniger tragen kann, bleibt er leer und in der Tabelle stehen 0 €. Liegt die Tragkraft des Rucksacks T zwischen 3 kg und 9 kg, packt man am besten den Computer ein (300 €). Ab 10 kg Tragkraft lohnt es sich, zusätzlich die Vase einzupacken (Gesamtwert 400 €), und ab 13 kg packt man statt der Vase die Uhr ein (Gesamtwert 450 €). Sobald man $T = 15$ kg einpacken kann, wären Computer und Fernseher (Gesamtwert 500 €) die beste Wahl und bei T = 20 kg kann man schließlich Computer, Vase und Uhr einpacken (Gesamtwert 550 €). Von den prinzipiell 16 unterschiedlichen Auswahlmöglichkeiten aus vier Objekten sind also letztlich nur fünf Kombinationen überhaupt sinnvoll, je nach Gewichtslimit. Es wäre zum Beispiel unvernünftig, den Fernseher und die Vase (Gesamtgewicht 19 kg, Gesamtwert 300 €) einzupacken, weil man ja stattdessen auch den Computer und die Uhr (Gesamtgewicht 13 kg, Gesamtwert 450 €) einpacken könnte und damit einen höheren Wert bei geringerem Gewicht erzielen würde.

Allgemein kann man das so definieren: Eine Kombination A von Objekten *dominiert* eine andere Kombination B, wenn A höchstens so viel wiegt wie B, aber mehr wert ist als B, also wenn

$$A.gewicht \leq B.gewicht \quad \text{und} \quad A.wert > B.wert$$

In diesem Fall ist die Auswahl B niemals optimal, denn immer wenn man B in den Rucksack packen kann, könnte man stattdessen die wertvollere Kombination A einpacken.

Wenn Sie im Beispiel von den 16 möglichen Auswahlkombinationen alle weglassen, die von irgendeiner anderen Kombination dominiert werden, dann bleiben gerade die fünf Kombination übrig, die wir oben aufgezählt haben: {}, {Computer}, {Computer, Vase}, {Computer, Uhr} und {Computer, Vase, Uhr}. Wir wollen sie »dominante« Kombinationen nennen. Im Beispiel können Sie sehen, dass diese fünf dominanten Kombinationen innerhalb der Tabellenspalte sortiert sind, und zwar sortiert sowohl nach dem Gewicht als auch nach dem Wert. Das muss so sein, denn wenn eine dominante Kombination B weniger wert ist als eine andere dominante Kombination A, so muss

sie auch weniger wiegen, denn sonst wäre B von A dominiert und dann eben keine dominante Kombination.

Wir können uns diese Beobachtung zunutze machen, um einen Algorithmus zu entwickeln, der zu einer gegebenen Liste L von Auswahlkombinationen alle dominanten Kombinationen heraussucht. Dazu durchlaufen wir die Kombinationen in aufsteigender Reihenfolge ihres Gewichts und schmeißen jede Kombination weg, die nicht mehr wiegt als die Kombination davor.

```
GetDominants(L)                 gibt alle dominanten Kombinationen in L zurück
                                L: Liste von n Objektkombinationen
    sortiere L absteigend nach dem Wert
    sortiere L stabil aufsteigend nach dem Gewichten
    combi ← L[1]                erste dominante Kombination
    D ← [combi]                 Liste der dominanten Kombinationen aus L
    for i ← 2, ..., n           durchlaufe nacheinander die Kombinationen aus L
        if L[i].wert > combi.wert
            combi ← L[i]        neue dominante Kombination gefunden
            D.append(combi)     füge sie hinten an die Liste D an
    return D
```

Durch die beiden Sortierschritte zu Beginn von GetDominants wird dafür gesorgt, dass wenn es zwei Kombinationen A und B mit dem gleichen Gewicht gibt, diejenige mit dem größten Wert zuerst kommt und als dominante Kombination ausgewählt wird. Der Algorithmus beachtet auch einen Sonderfall, denn wir bislang noch gar nicht berücksichtigt haben: Es kann nämlich sein, dass es in der Liste L mehrere dominante Kombinationen mit gleichem Gewicht und gleichem Wert gibt. GetDominants übernimmt in diesem Fall nur eine der Kombinationen in die Ergebnisliste D, die restlichen gleichwertigen Kombinationen werden nicht benötigt.

Wenn sowohl die Gewichte als auch die Werte aller Objekte ganze Zahlen sind, so kann man zum Sortieren der Liste L einen Linearzeit-Sortieralgorithmus verwenden, wie er in Kapitel 7 beschrieben ist. Nach dem Sortieren läuft der Algorithmus nur noch einmal durch alle Kombinationen in der Liste, sodass GetDominants bei m Kombinationen in L insgesamt eine Laufzeit von $\mathcal{O}(m)$ hat.

Mit GetDominants und der Idee des dynamischen Programmierens lässt sich ein neuer Algorithmus zur Lösung des Rucksackproblems stricken. Nehmen Sie wieder an, Sie hätten für die ersten $i-1$ Objekte in der Menge M bereits die vollständige Liste D aller dominanten Kombinationen bestimmt und wollen das nun auf die Liste aller dominanten Kombinationen für die ersten i Objekte erweitern. Dann gibt es für das i-te Objekt $M[i]$ wieder nur zwei Möglichkeiten:

1. Man packt $M[i]$ in den Rucksack. Den übrigen Platz füllt man mit einer

Kombination aus den ersten $i-1$ Objekten auf, und da man den Gesamtwert maximieren will, kommt dafür nur eine *dominante* Kombination aus den ersten $i-1$ Objekten infrage.

2. Man packt $M[i]$ *nicht* in den Rucksack. Dann füllt man den Rucksack eben mit einer dominanten Kombination aus den ersten $i-1$ Objekten.

Das bedeutet: »Alle dominanten Kombinationen aus den ersten i Objekten bestehen aus einer dominanten Kombination aus den ersten $i-1$ Objekten, zu der man entweder das Objekt $M[i]$ hinzugefügt hat oder auch nicht.«

```
KnapsackDomi(M, T)          M: Menge von n Objekten, T: maximale Tragkraft
    D ← [ {} ]              Liste der dominanten Kombinationen, enthält anfangs {}
    for i ← 1, ..., n       durchlaufe nacheinander alle Objekte in M
        L ← [ ]             Liste von Kombinationen aus den ersten i Objekten
        foreach combi in D  durchlaufe alle dominanten Kombinationen aus den ersten
                            i − 1 Objekten und füge sie zu L hinzu, und zwar ...
            L.append(combi)                          ... einmal ohne M[i] ...
            if combi.gewicht + M[i].gewicht < T      ... und wenn es passt, ...
                L.append(combi ∪ {M[i]})             ... einmal mit M[i]
        D ← GetDominants(L)     ermittle die dominanten Kombinationen
    return letztes Element in D die dominante Kombination mit dem größten Wert
```

Am Ende ist die dominante Kombination mit dem größten Wert die optimale Auswahl an Objekten; KnapsackDomi kann also auf ein Backtrace verzichten.

Die Laufzeit von KnapsackDomi hängt zum einen von der Anzahl n der Objekte ab und zum anderen von der Länge der Listen D und L. Wie viele dominante Kombinationen kann es maximal geben? Die dominanten Kombinationen haben alle ein unterschiedliches Gewicht. Da die Gewichte ganze Zahlen sind, kommt als Gesamtgewicht einer Kombination nur eine Zahl aus $\{0, 1, 2, \ldots, T\}$ infrage, das heißt, es gibt maximal $T+1$ Kombinationen in D. Die Liste L andererseits enthält höchstens doppelt so viele Kombinationen wie D. Insgesamt hat KnapsackDomi also eine Laufzeit von $\mathcal{O}(n \cdot T)$.

Gut, das gleiche Ergebnis hatten wir ja schon bei KnapsackDP erreicht. Immerhin hat KnapsackDomi eine Chance, dass es deutlich weniger als T Kombinationen in D geben könnte, was sich dann positiv auf die Laufzeit auswirken würde.

Außerdem können wir die Laufzeit von KnapsackDomi statt mit der Tragkraft T auch mit dem maximalen Wert w_{opt} einer Auswahl von Objekten, die in den Rucksack passen, abschätzen. Da dominante Kombinationen alle unterschiedlich viel Wert sein müssen (je schwerer, desto wertvoller), gibt es von ihnen maximal $w_{opt}+1$. KnapsackDomi hat also auch eine Laufzeit von $\mathcal{O}(n \cdot w_{opt})$. Dieses Ergebnis wird uns

gleich noch sehr nützlich sein.

Sich dem Optimum annähern

Es sollte inzwischen klar sein, dass das Rucksackproblem als ein NP-schweres Problem nicht einfach so auf die Schnelle aufzudröseln ist. Alle bisher vorgestellten Algorithmen lösen das Problem entweder gar nicht (wie KnapsackGreedy) oder haben im *worst case* eine (pseudo-)exponentielle Laufzeit und es ist bislang auch noch niemandem gelungen, das besser hinzubekommen. Zum Glück ist man nicht immer darauf angewiesen, das Rucksackproblem exakt zu lösen, denn oft reicht eine möglichst gute Näherung an das Optimum aus. Das Rucksackproblem gehört zu den Problemen, die sich sehr gut näherungsweise lösen lassen. Es gibt nämlich eine Methode, mit der man sich dem Optimum beliebig annähern kann, ohne exponentielle Laufzeit aufwenden zu müssen. Um genauer zu sagen, was »beliebig annähern« bedeutet, brauchen wir folgende Definition:



Sei ε eine Zahl zwischen 0 und 1. Eine Auswahl an Objekten mit einem Gesamtwert $\geq (1-\varepsilon) \cdot w_{opt}$ nennen wir eine ε *-Approximation* und einen Algorithmus, der ε-Approximationen berechnen kann, ist ein ε *-Approximationsalgorithmus*.

»Approximation« bedeutet »Annäherung«. Die Zahl ε ist sozusagen ein Maß für den erlaubten Fehler. Je kleiner ε ist, desto näher kommt der ε-Approximationsalgorithmus dem Optimum w_{opt} und bei $\varepsilon = 0$ würde der Algorithmus keine Annäherung mehr berechnen, sondern das Optimum selbst. Wie Sie gleich sehen werden, gibt es im Falle des Rucksackproblems für jedes $\varepsilon > 0$ einen ε-Approximationsalgorithmus, der in polynomieller Zeit läuft.



Mehr zum Thema Approximation lesen Sie in Kapitel 14.

Billigere Objekte erleichtern die Auswahl

Der Algorithmus KnapsackDomi hat eine Laufzeit von $\mathcal{O}(n \cdot w_{opt})$, das heißt, die Laufzeitschranke hängt nicht nur von der Anzahl n der Objekte ab, sondern auch vom Gesamtwert w_{opt} einer optimalen Auswahl von Objekten. Nehmen Sie nun an, es wäre Sommerschlussverkauf und die Preise (beziehungsweise Werte) aller Objekte würde um den gleichen Faktor sinken (Abbildung 8.8). Durch diese »Rabattaktion« würde sich die optimale Auswahl an Objekten nicht verändern, nur der Gesamtwert w_{opt} würde sinken, und zwar um den gleichen Faktor wie die Preise der Objekte. Wenn die Laufzeitschranke also von w_{opt} abhängt und wir w_{opt} beliebig senken können, ohne die optimale Auswahl an Objekten zu ändern, so haben wir anscheinend eine hübsche Methode gefunden, KnapsackDomi so schnell zu machen, wie wir wollen. Leider hat die Idee einen Haken: Die hübsche Laufzeitschranke funktioniert nur bei ganzzahligen Werten. Wenn wir also alle Werte $M[i].wert$ durch die gleiche Zahl r teilen, sind die Werte

anschließend vermutlich nicht mehr ganzzahlig. Bevor wir KnapsackDomi aufrufen können, müssen wir also die neuen Werte auf ganze Zahlen runden. Leider können sich durch das Runden die Werteverhältnisse der unterschiedlichen Objekte verschieben, und darum ist es danach nicht mehr garantiert, dass KnapsackDomi bei reduzierten Werten noch die gleiche Objektauswahl berechnet wie vor der »Rabattaktion«. Man kann aber die Auswahl der Objekte bei reduzierten Werten als Approximation der Auswahl bei den ursprünglichen Werten benutzen. Dazu setzen wir r wie folgt:

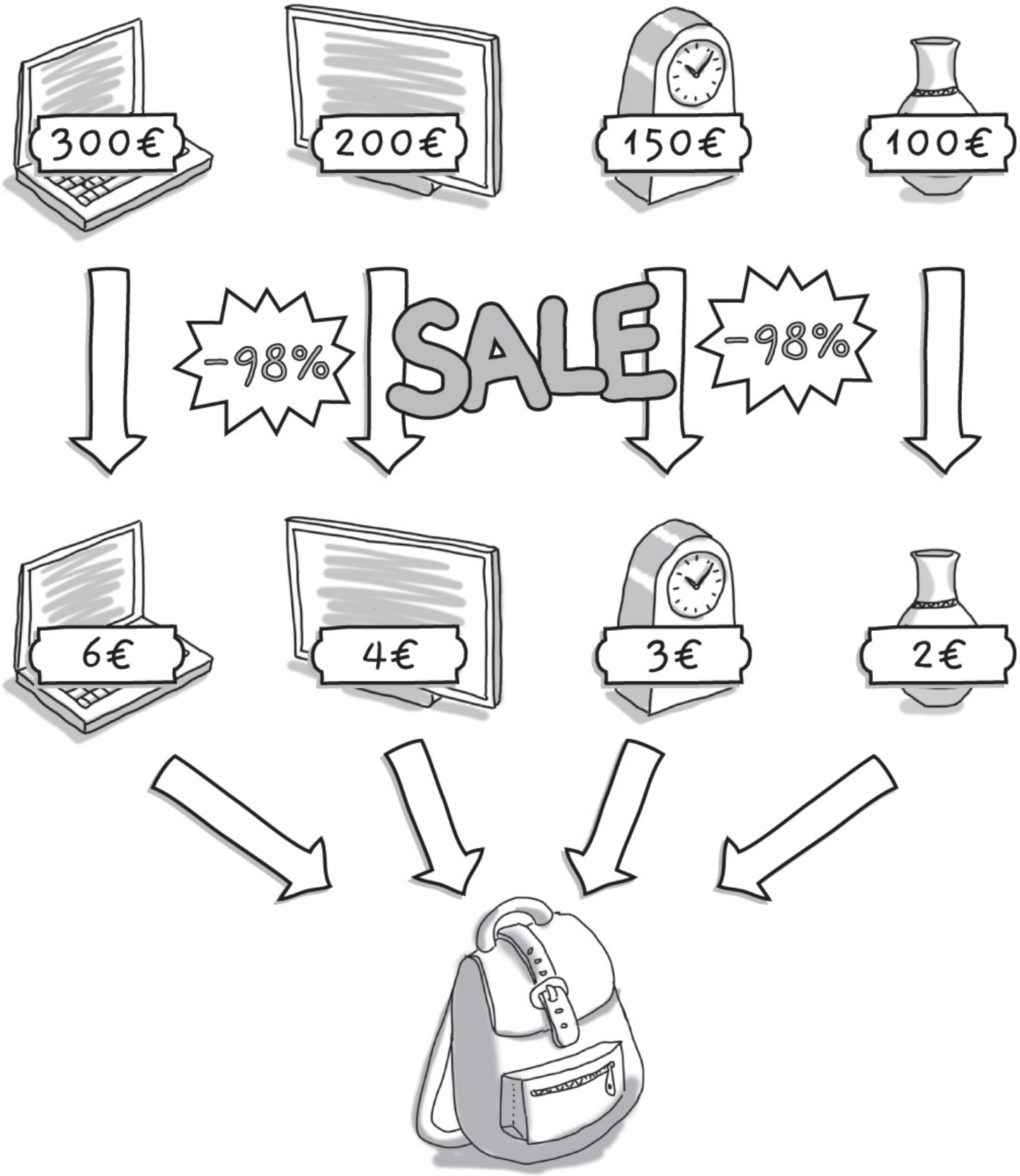


Abbildung 8.8: Preissenkung führt zur Laufzeitsenkung

$$r = \frac{\varepsilon \cdot w_{max}}{n},$$

wobei w_{max} der Wert des wertvollsten Objekts in der Menge M sein soll. Dabei gehen

wir davon aus, dass jedes Objekt in M einzeln in den Rucksack passen würde, also dass $M[i].gewicht \leq T$, denn ein schwereres Objekt lässt sich ohnehin nicht einpacken und man kann es darum auch gleich aus M entfernen.

KnapsackApprox(M, T, ε) — *M: n Objekte, T: max. Tragkraft, ε: Approximationsgüte*

$w_{max} \leftarrow$ maximaler Wert in M

$r \leftarrow \frac{\varepsilon \cdot w_{max}}{n}$ — *berechne r*

for $i \leftarrow 1, \ldots, n$ — *für alle Objekte – tue Folgendes*

$M[i].urwert \leftarrow M[i].wert$ — *sichere ursprünglichen Wert in Attribut urwert*

$M[i].wert \leftarrow \frac{M[i].wert}{r}$ abgerundet — *reduziere Wert um den Faktor $1/r$*

$S \leftarrow$ **KnapsackDomi**(M, T) — *berechne optimale Auswahl bei reduzierten Werten*

foreach $x \in S$ — *für alle Objekte der Auswahl – tue Folgendes:*

$x.wert \leftarrow x.urwert$ — *stelle ursprünglichen Wert wieder her*

return S

Der Grund, warum wir r so und nicht anders setzen, ist einfach: Damit wird unser KnapsackApprox zu einem ε-Approximationsalgorithmus. Wenn Sie das nachrechnen möchten, so werfen Sie einen Blick in den Kasten.



KnapsackApprox ist ein ε-Approximationsalgorithmus

Der (ursprüngliche) Wert eines Objekts x soll $x.urwert$ heißen, der reduzierte und gerundete Wert $x.redwert$. Durch das eventuelle Abrunden ist $x.redwert + 1 \geq \frac{x.urwert}{r} \geq x.redwert$. Die optimale Auswahl von Objekten bei normalen Werten sei S_{opt}, die bei reduzierten und gerundeten Werten nennen wir S.

KnapsackDomi gibt also S zurück. Betrachten wir nun den Gesamtwert der Auswahl:

$$
\begin{aligned}
\sum_{x \in S} x.urwert &= r \cdot \sum_{x \in S} \frac{x.urwert}{r} \\
&\geq r \cdot \sum_{x \in S} x.redwert, && \text{da } \frac{x.urwert}{r} \geq x.redwert \\
&\geq r \cdot \sum_{x \in S_{opt}} x.redwert, && \text{da } S \text{ optimal ist bei reduzierten Werten} \\
&\geq \sum_{x \in S_{opt}} (x.urwert - r), && \text{da } x.redwert + 1 \geq \frac{x.urwert}{r} \\
&= \sum_{x \in S_{opt}} x.urwert - \sum_{x \in S_{opt}} r \\
&\geq w_{opt} - n \cdot r, && \text{da } S \text{ höchstens } n \text{ Elemente enthält} \\
&= w_{opt} - \varepsilon \cdot w_{max}, && \text{da } r = \frac{\varepsilon \cdot w_{max}}{n} \text{ gesetzt wurde} \\
&\geq w_{opt} - \varepsilon \cdot w_{opt}, && \text{da jedes Objekt allein in den Rucksack passt} \\
&= (1 - \varepsilon) \cdot w_{opt}
\end{aligned}
$$

Wenn wir also alle Zwischenschritte weglassen, kommt insgesamt

$$\sum_{x \in S} x.urwert \geq (1 - \varepsilon) \cdot w_{opt}$$

heraus, und das ist genau die Definition für einen ε-Approximationsalgorithmus.

Gleichzeitig kann man leicht ausrechnen, dass KnapsackApprox eine Laufzeit von $\mathcal{O}\left(\frac{n^3}{\varepsilon}\right)$ hat. Details können Sie im nächsten Kasten nachlesen. Wenn man die Approximationsgüte ε konstant wählt, hat KnapsackApprox also eine polynomielle Laufzeit. Das ist prima, denn alle bekannten Algorithmen, die das Rucksackproblem exakt und nicht nur näherungsweise lösen, benötigen eine exponentielle Laufzeit.



KnapsackApprox läuft in polynomieller Zeit

Die Laufzeit von KnapsackApprox hängt davon ab, wie lang KnapsackDomi braucht. Am Ende des Abschnitts *Wachsende Probleme klug speichern* haben Sie gesehen, dass KnapsackDomi eine Laufzeit von $\mathcal{O}(n \cdot w_{opt})$ hat. Mit w_{opt} ist in unserem Fall die Summe der *reduzierten* Werte der ausgewählten Objekte gemeint. Jetzt noch mal ein wenig rechnen:

$$
\begin{aligned}
w_{opt} &= \sum_{x \in S} x.redwert \\
&\leq \sum_{x \in S} \frac{x.urwert}{r}, && \text{da } x.redwert \leq \frac{x.urwert}{r} \\
&= \sum_{x \in S} \frac{x.urwert \cdot n}{\varepsilon \cdot w_{max}}, && \text{da } r = \frac{\varepsilon \cdot w_{max}}{n} \\
&= \frac{n}{\varepsilon} \cdot \sum_{x \in S} \frac{x.urwert}{w_{max}} \\
&\leq \frac{n}{\varepsilon} \cdot \sum_{x \in S} 1, && \text{da } w_{max} \text{ der maximale Objektwert ist} \\
&\leq \frac{n}{\varepsilon} \cdot n = \frac{n^2}{\varepsilon}, && \text{da } S \text{ höchstens alle } n \text{ Objekte enthalten kann.}
\end{aligned}
$$

Es ist also $w_{opt} \leq \frac{n^2}{\varepsilon}$, also läuft KnapsackDomi in $\mathcal{O}\left(\frac{n^3}{\varepsilon}\right)$. Da KnapsackApprox abgesehen von KnapsackDomi nur noch Aufgaben zu tun hat, die man in $\mathcal{O}(n)$ erledigen kann, ist seine Gesamtlaufzeit ebenfalls $\mathcal{O}\left(\frac{n^3}{\varepsilon}\right)$.



Das Rucksackproblem ist NP-schwer und alle bekannten Methoden, es zu lösen, benötigen eine exponentielle Laufzeit. Näherungsweise kann man das Problem hingegen in polynomieller Zeit lösen. Dabei kann man die Fehlertoleranz ε beliebig klein wählen und doch bleibt die Laufzeit für alle $\varepsilon > 0$ polynomiell. Erst wenn man $\varepsilon = 0$ setzen würde, also gar keine Approximation mehr zuließe, würde die Methode nicht mehr funktionieren und es blieben nur Algorithmen mit exponentieller Laufzeit übrig.

Lineare Optimierung

Zum Schluss des Kapitels sollen Sie nun die vielleicht einfachste und schnellste Lösungsmethode für das Rucksackproblem kennenlernen. Anstatt das Rucksackproblem als solches zu lösen, verwandeln wir es einfach in ein anderes Problem, für das es schon probate Lösungsverfahren gibt. Der Informatiker nennt das »ein Problem auf ein anderes *reduzieren*«. Der Ausdruck ist allerdings etwas verwirrend, da das Problem durch die

»Reduktion« gewöhnlich nicht gerade kleiner wird. Im Gegenteil: Die Komplexität des Problems steigt dadurch eher an. Trotzdem kann sich dieser Schritt lohnen, nämlich dann, wenn man das neue Problem schon gelöst hat oder weiß, wen man dazu fragen kann.

Das Problem, auf das wir das Rucksackproblem reduzieren wollen, nennt sich *lineare Optimierung* oder auch *lineare Programmierung*, auf Englisch *linear programming* (LP), und es ist so berühmt, dass es dazu wirklich viele Veröffentlichungen, Lösungsansätze und sogar fertige Programme gibt. Manche Software für lineare Optimierung ist so ausgereift, dass man viele Probleme oft schneller dadurch lösen kann, dass man sie in ein lineares Programm verwandelt und durch diese Software jagt, als wenn man sich selbst um einen auf das eigene Problem spezialisierten Algorithmus und seine Implementierung bemüht. Sie werden jetzt an einem einfachen Beispiel sehen, was lineare Optimierung ist und wozu man sie brauchen kann.

Optimierung von Produktionsmengen

Nehmen Sie an, Sie wären ein Spielzeugmacher. Sie haben sich auf genau zwei verschiedene Holzspielzeuge spezialisiert (siehe Abbildung 8.9): Zum einen können Sie aus einem Holzbrettchen, zwei Federn und zwei Glasaugen eine kleine Eule herstellen, zum anderen stellen Sie auch putzige Tiger her, und zwar jeweils aus einem Holzbrettchen, einem Stück Kordel und einem Glasauge. Die Eulen können Sie mit 3 € Gewinn pro Stück verkaufen; pro Tiger machen Sie nur einen Gewinn von 2 €.

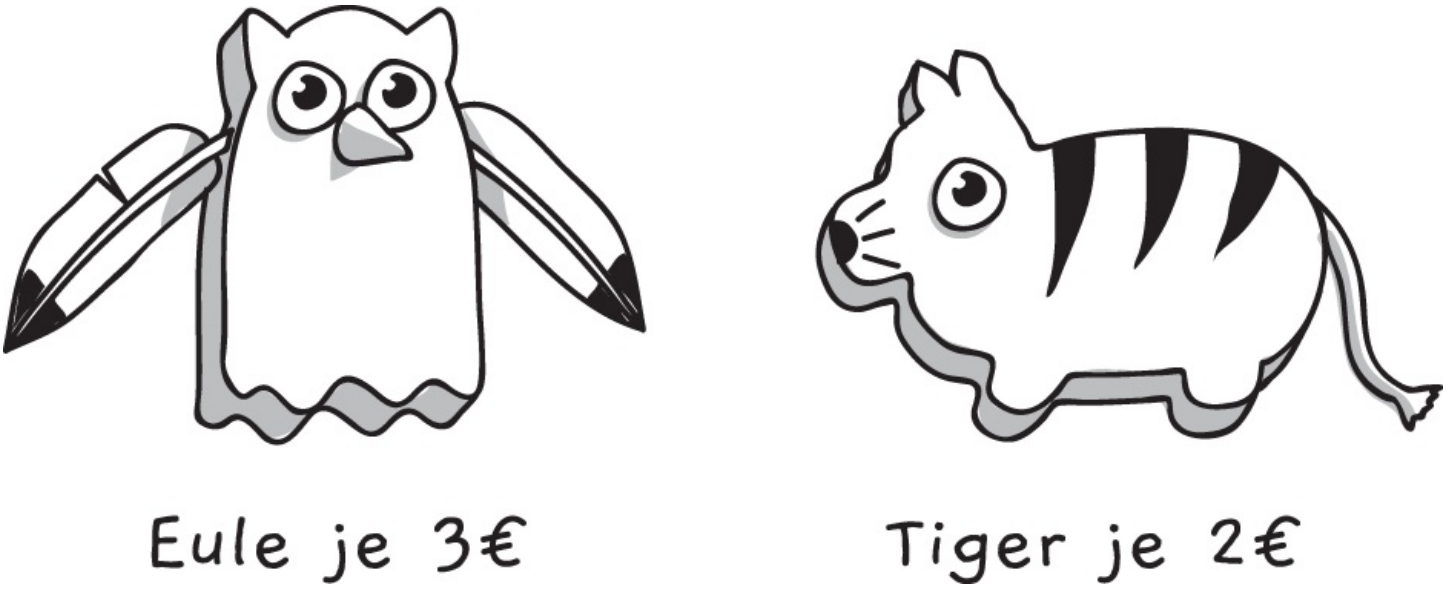


Abbildung 8.9: Zwei Holzspielzeuge und wie viel Gewinn man jeweils damit erzielen könnte

Leider laufen die Geschäfte nicht sonderlich gut. Im Moment haben Sie leider kein Geld, um neues Material für Ihre Produktion einzukaufen, sondern müssen mit den kläglichen Resten vorliebnehmen, die sich noch in Ihrem Lager befinden. Insgesamt haben Sie nur noch acht Kordeln, zwölf Federn, zehn Brettchen und vierzehn Glasaugen (siehe Abbildung 8.10). Damit wollen Sie nun so viel Gewinn wie möglich erwirtschaften. Wie viele Exemplare von jeder Ware wollen Sie produzieren?

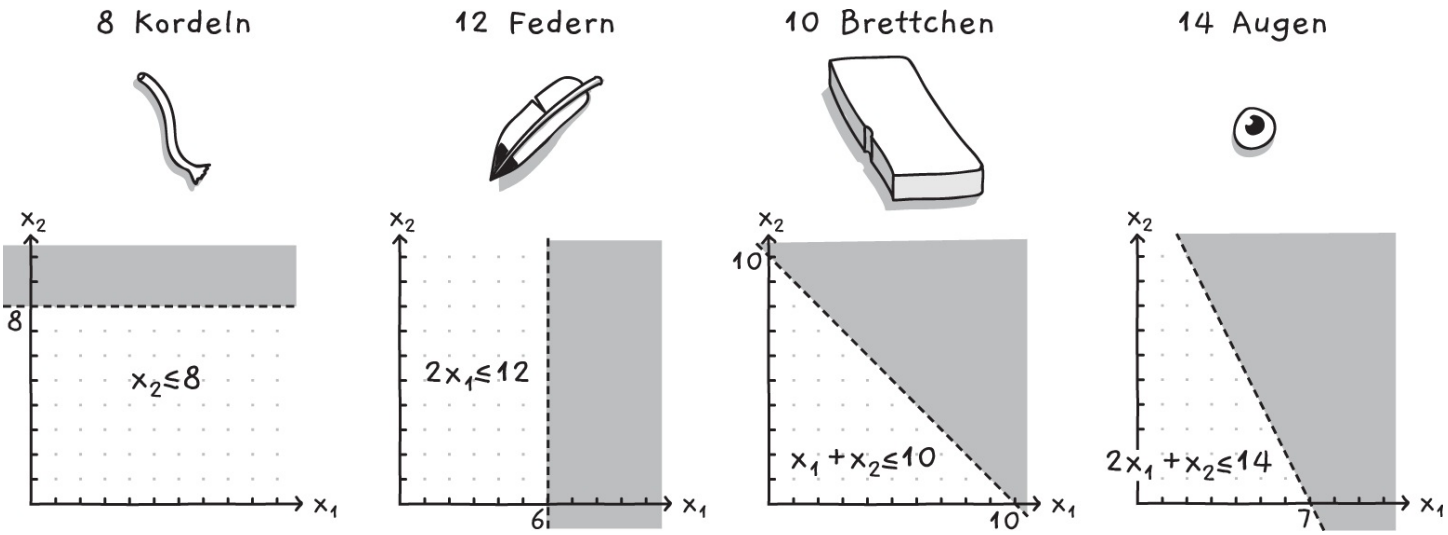


Abbildung 8.10: Grenzen der Produktionszahlen in Abhängigkeit von den benötigten Bauteilen

Ein System von Ungleichungen

Wir haben es also mit einem Optimierungsproblem zu tun. Die Anzahl der Eulen, die wir produzieren wollen, soll x_1 heißen und die Anzahl der Tiger x_2. Jede mögliche Kombination von Produktionszahlen können wir in ein Koordinatensystem eintragen: Nach rechts tragen wir x_1 auf, nach oben x_2. Jeder Punkt auf dem Raster entspricht einer möglichen Antwort auf die Frage, wie viele Eulen und wie viele Tiger Sie produzieren wollen.

Aber nicht alle beliebigen Zahlenkombinationen für x_1 und x_2 sind möglich. Da Sie für jeden Tiger eine Kordel als Schwanz benötigen und sich im Lager nur acht Kordeln finden, können Sie auch nicht mehr als acht Tiger herstellen. Aus den begrenzten Anzahlen der übrigen Bauteile ergeben sich ebenfalls Grenzen für die Produktionszahlen. Da Sie zum Beispiel für jede Eule zwei Augen und für jeden Tiger ein Auge benötigen, insgesamt aber nur vierzehn Augen haben, folgt: $2 \cdot x_1 + x_2 \leq 14$. Wir können also den Raum der möglichen Lösungen des Optimierungsproblems mit einer Reihe von Ungleichungen beschreiben:

$x_2 \leq 8$	*für jeden Tiger wird eine von 8 Kordeln gebraucht*
$2 \cdot x_1 \leq 12$	*jede Eule braucht zwei von 12 Federn*
$x_1 + x_2 \leq 10$	*jede Eule und jeder Tiger braucht ein von 10 Brettchen*
$2 \cdot x_1 + x_2 \leq 14$	*Eulen haben zwei Augen, Tiger eins, insgesamt gibt es 14*

Außerdem müssen natürlich $x_1 \geq 0$ und $x_2 \geq 0$ sein.

Ziel: Gewinnmaximierung

Jede dieser Ungleichungen setzt eine Grenze in Form einer Geraden in das Koordinatensystem und trennt damit die möglichen von den unmöglichen Lösungskombinationen. Das Gebiet, das auf der erlaubten Seite von *allen* Grenzgeraden liegt, ist der *Lösungsraum* (Abbildung 8.11). In diesem Lösungsraum müssen wir die

Kombination mit dem größten Gewinn suchen.

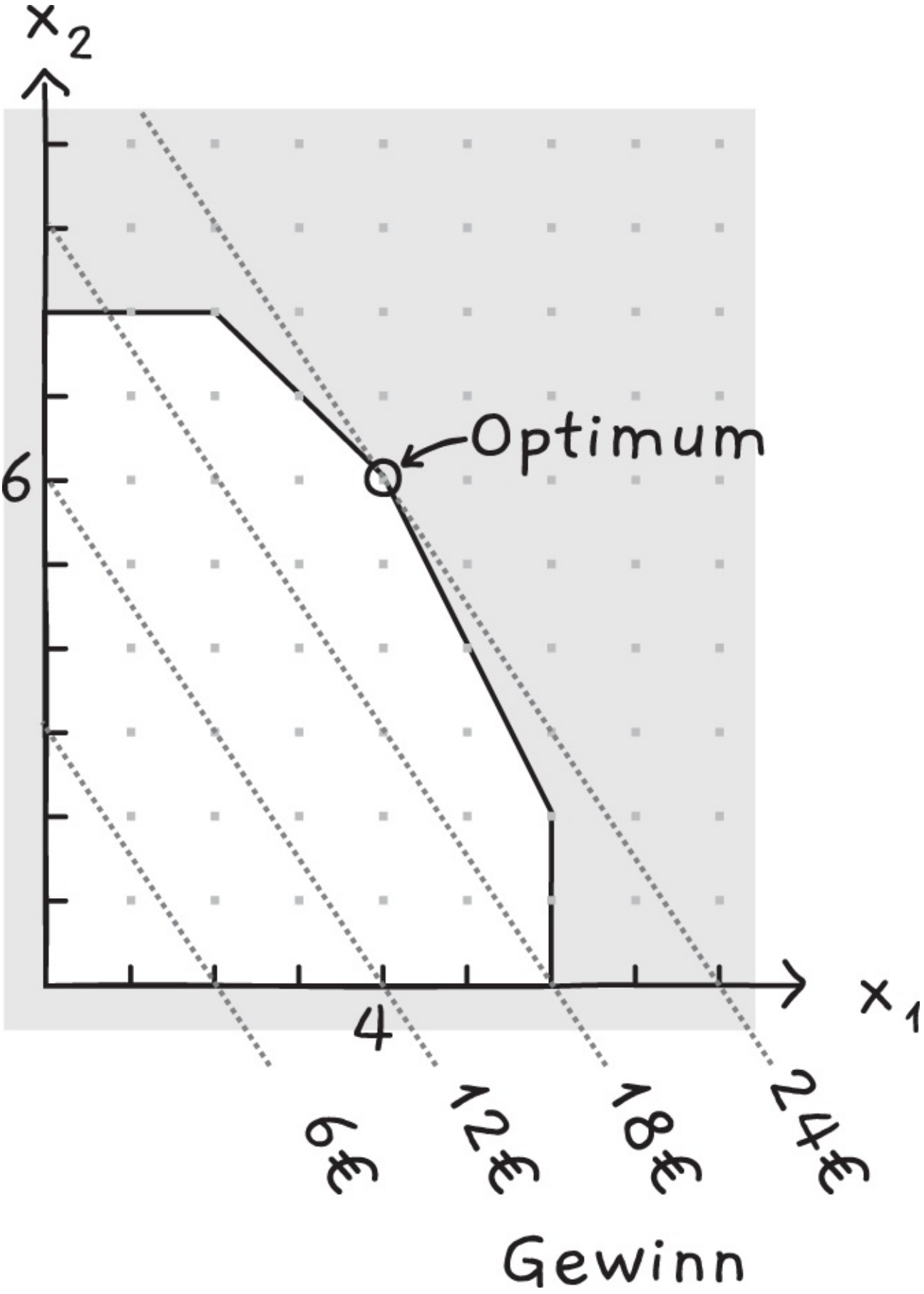


Abbildung 8.11: Das Optimum liegt am Rand des Lösungsraums.

Der Gewinn ist einfach zu berechnen: Für jede Eule bekommen wir $3\ €$ und für jeden Tiger $2\ €$, also:

$$\text{Gewinn} = 3 \cdot x_1 + 2 \cdot x_2$$

Nun haben wir das Problem fertig beschrieben: Durch ein System von Ungleichungen und eine Zielfunktion, die maximiert werden soll. Alle Ungleichungen und auch die Zielfunktion sind ähnlich aufgebaut: Die Variablen x_1 und x_2 werden jeweils mit einem konstanten Faktor multipliziert und anschließend aufaddiert. Bei Ungleichungen wie $x_2 \leq 8$ ist der konstante Faktor für x_1 eben null, sodass x_1 aus der Ungleichung komplett herausfällt. Funktionen dieser Art nennt man *linear* und genau darum heißt das Problem auch *lineare* Optimierung.

Wenn Sie nun mehr als zwei verschiedene Spielzeuge herstellen könnten, also abgesehen von Eulen und Tigern auch noch Pinguine, Elefanten, Hasen und so weiter, dann steigt damit einfach die Zahl der Variablen des Problems. Dadurch wird es zwar deutlich schwieriger, den Lösungsraum zu zeichnen, denn bei k Variablen ist er ein k-dimensionaler Raum, aber die Mathematik ändert sich nicht wesentlich.

Ganzzahlige lineare Optimierung

Eine weitere Bedingung an die Lösung haben wir bislang noch nicht erwähnt: Niemand kauft eine halbe Eule oder ein Drittel eines Tigers. Die Variablen x_1 und x_2 müssen also ganze Zahlen sein. Wir haben es hier mithin mit der ganzzahligen Variante der linearen Optimierung, dem sogenannten *integer linear programming* (ILP), zu tun.

Im Fall des Holzspielzeugproblems kann man die optimale Lösung noch recht leicht aus der Abbildung 8.11 ablesen: Am besten stellen Sie vier Eulen und sechs Tiger her, damit machen Sie einen Gesamtgewinn von $4 \cdot 3\ € + 6 \cdot 2\ € = 24\ €$. Das Optimum liegt so weit wie möglich oben rechts, an einer Ecke, also an einem Schnittpunkt von zwei Geraden. Auch beim allgemeinen LP ist es immer so, dass man sich bei der Suche nach dem Optimum auf die »Ecken« des Lösungsraums beschränken kann. Beim ILP reicht das leider nicht, denn die Ecken müssen ja nicht unbedingt auf Punkten mit ganzzahligen Koordinaten liegen. Bei unserem Beispiel haben wir Glück gehabt, da die optimale Ecke am Ort $x_1 = 4$ und $x_2 = 6$ eine gültige ganzzahlige Lösung ist.

Das Rucksackproblem als ILP

Um das Rucksackproblem in ein ganzzahliges lineares Optimierungsproblem zu verwandeln, müssen wir zunächst festlegen, welche Variablen es beim linearen Optimieren geben soll. Uns reicht für jedes Objekt a in der Menge eine Variable, die wir x_a nennen wollen. Die Variablen sollen entweder gleich 0 oder gleich 1 sein. Ist die Variable 1, dann packen wir das entsprechende Objekt ein, ist sie hingegen 0, wird das Objekt nicht eingepackt. Die Zielfunktion, nämlich die Maximierung des Gesamtwerts der

Objekte im Rucksack, kann man dann ganz einfach als lineare Funktion formulieren:

$$\text{maximiere} \sum_{a \in M} a.wert \cdot x_a$$

Wenn ein Objekt a im Rucksack ist ($x_a = 1$), dann geht sein Wert $a.wert$ mit in die Zielfunktion ein, hat man es hingegen nicht eingepackt, wird sein Wert mit $x_a = 0$ multipliziert und fällt aus der Summe raus. Allerdings müssen wir auch noch die maximale Tragkraft T des Rucksacks berücksichtigen. Dafür können wir mit einer einzelnen Ungleichung sorgen, nämlich

$$\sum_{a \in M} a.gewicht \cdot x_a \leq T$$

Nun müssen wir noch den Variablenbereich auf 0 und 1 einschränken. Dazu fordern wir für jedes Objekt a :

$$x_a \leq 1$$

$$x_a \geq 0$$

Bei n Objekten bekommen wir damit $2n$ weitere Ungleichungen hinzu, mit denen wir dafür sorgen, dass die Variablen x_a zwischen 0 und 1 liegen beziehungsweise – wenn sie ganzzahlig sind – entweder 0 oder 1 sind. Damit haben wir es geschafft: Wir haben das Rucksackproblem als ILP beschrieben. Eine Lösung des ILP besteht aus einer Belegung der Variablen mit den Werten 0 oder 1. Anhand dieser Werte können wir dann unseren Rucksack packen.

Wir haben gerade gezeigt, dass man das Rucksackproblem auf eine ILP *reduzieren* kann. Da das Rucksackproblem NP-schwer ist, bedeutet das, dass ILP ebenfalls NP-schwer ist. Die allgemeine LP, bei der die Variablen also nicht unbedingt ganze Zahlen sein müssen, ist hingegen nicht NP-schwer und es gibt verschiedene Algorithmen, die das in polynomieller Zeit lösen.

Kapitel 9

Mengen und ihre Speicherung

IN DIESEM KAPITEL

Speicherung von großen Datenmengen …

… in Baumstrukturen und Hashtabellen

Balancierte Bäume und Rot-Schwarz-Bäume

Mengenlehre ist die Lehre von Kreisen voller geometrischer Figuren, die sich überschneiden oder auch nicht und die manchmal sektorweise schraffiert werden. In dem Augenblick, in dem man sie verstanden hat, weiß man, dass man nichts gelernt hat, was man nicht schon vorher wusste (siehe Abbildung 9.1).

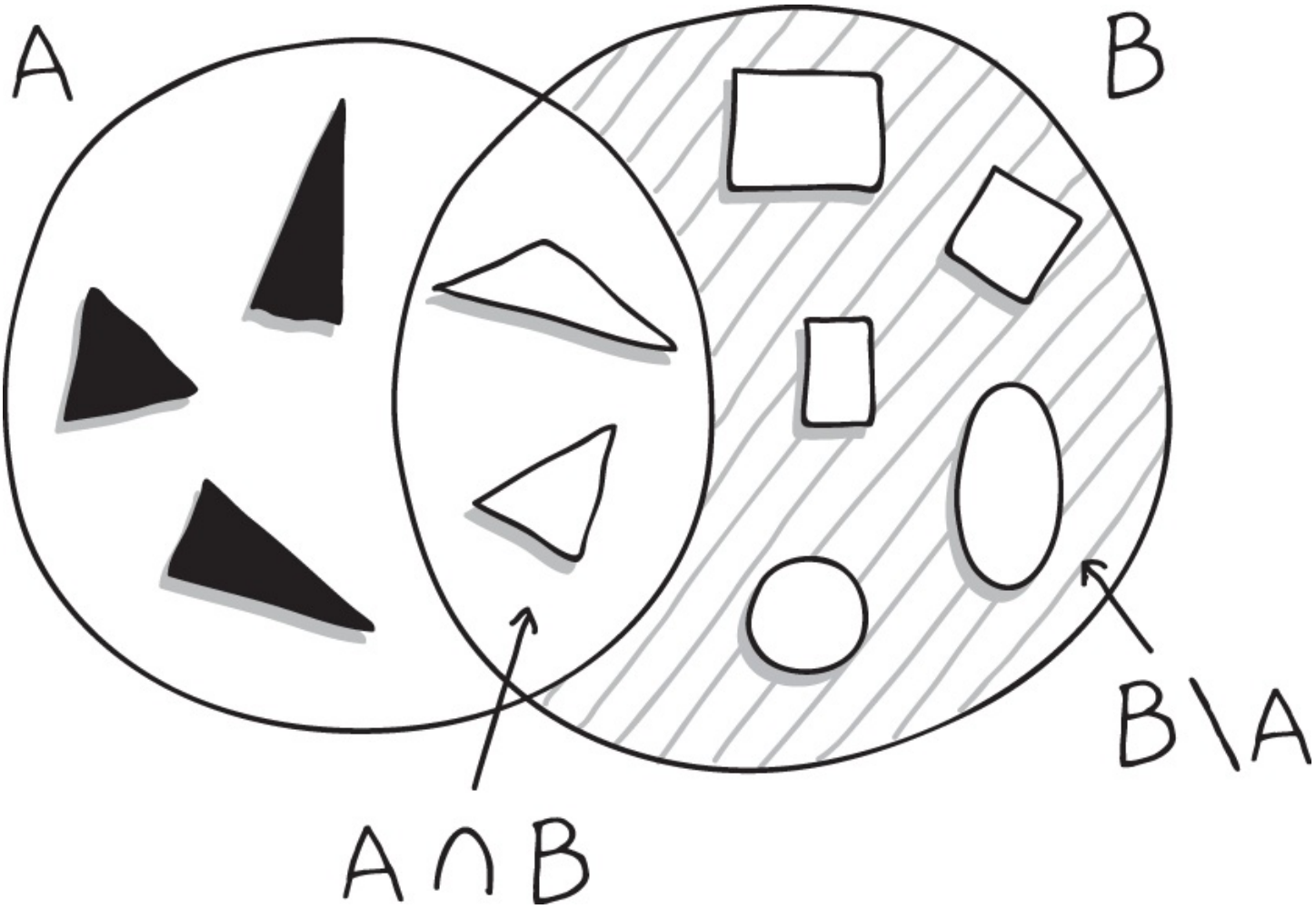


Abbildung 9.1: Mengenlehre – die Wissenschaft von geometrischen Figuren in Kreisen

In diesem Stil wurde die Mengenlehre als eine der ersten Verirrungen der modernen Pädagogik in der Grundschule eingeführt und kurze Zeit später wieder abgeschafft. Heute erinnert sich dort zum Glück kaum einer mehr daran. Man vertritt (wenn überhaupt) das modernere Konzept, nachdem eine Menge einfach ziemlich viel ist, es sei denn, es handelt sich um eine geringe Menge.

Wir sind nicht mehr in der Grundschule und es ist uns natürlich klar, dass Mengen in der Mathematik das zentrale Modellierungskonzept darstellen. So gut wie alles wird in irgendeiner Form als Menge beschrieben. Wie jede ernsthafte Wissenschaft arbeitet auch die Informatik mit mathematischen Begriffen, also auch mit Mengen. Probleme und Algorithmen, die sie lösen, werden mit Begriffen der einfachen Mengenlehre beschrieben.

Die Vorstellung, dass es dabei meist um viele Dinge, große Mengen geht, ist auch nicht völlig abwegig. Wenn eine Menge in einer Datenstruktur gespeichert wird, dann wird die Sache erst interessant, wenn es sich um eine große Menge handelt. Erst wenn wirklich viele Elemente abgelegt, durchsucht oder sonst irgendwie verarbeitet werden, spielt die Art der Speicherung eine Rolle. Es geht hier also um die Verwaltung von großen Datenkollektionen.

Mathematiker, in ihrer manchmal schwer verständlichen Art, bestehen darauf, dass in einer Menge ein Element nur maximal einmal vorkommen kann. In einem Array oder einer Liste kann ein Wert, beispielsweise 2, natürlich mehr als einmal vorkommen. Damit ist so etwas wie

$$[0, 1, 2, 0, 1, 2]$$

keine Menge. Tatsächlich steckt in dieser Liste auch mehr Information als in der Menge $\{0, 1, 2\}$, nämlich die Reihenfolge. $[0, 1, 2, 0, 1, 2]$ ist tatsächlich eine Abbildung von Indexpositionen auf Werte:

$$\{(0 \to 0), (1 \to 1), (2 \to 2), (3 \to 0), (4 \to 1), (5 \to 2)\}$$

Klar, das ist bekannt, aber gibt es in der wirklichen Wert Kollektionen von Dingen, ohne dass sie in irgendeiner Art angeordnet sind? Steckt nicht alles in Listen, Bäumen, Graphen in irgendeiner Struktur? Muss man sich in der Informatik also wirklich mit Mengen beschäftigen?

Mengen sind nach G. Cantor »Zusammenfassungen von bestimmten wohl unterschiedenen Objekten unserer Anschauung oder unseres Denkens zu einem Ganzen.« Das ist durchaus sinnvoll. Wenn ich die Rezepte meiner Lieblingsgerichte in einer Datei oder einem Aktenordner speichere, dann geht das natürlich nur in einer bestimmten Ordnung und ich kann natürlich das gleiche Rezept mehrfach abspeichern. Als prinzipielles Konzept sind meine Lieblingsrezepte aber eine Menge. Keins davon kann zwei- oder dreimal zu meinen Lieblingsrezepten gehören, und ihnen eine Ordnung zu verpassen, geht über das Konzept erstmal hinaus.

Es geht hier um Zusammenfassungen von bestimmten wohl unterschiedenen Objekten.

Bei der Wohlunterschiedenheit gibt es dabei noch ein gelegentlich überraschendes Maß an Flexibilität. Wenn zweimal das Gleiche in einer Menge sein soll, dann erkläre ich die beiden einfach als ungleich. Zwei Zweien als ungleich zu definieren ist sicher wenig überzeugend, aber wann sind zwei Rezepte wirklich gleich? Oder mein Kumpel, bleibt er in der Menge meiner Kumpel, auch wenn er heute kaum wiederzuerkennen ist (siehe Abbildung 9.2)?

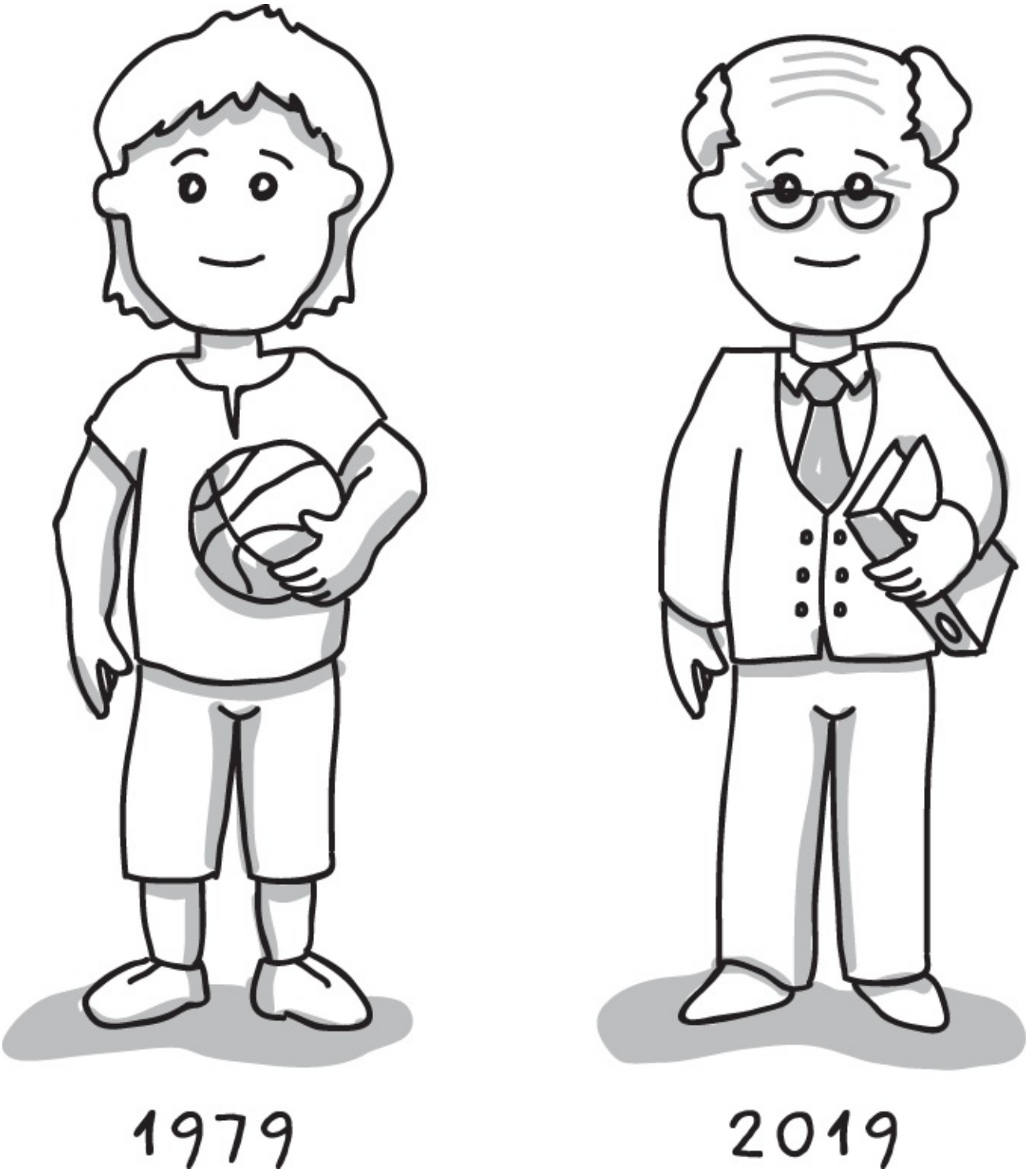


Abbildung 9.2: Wenn derselbe nicht mehr der Gleiche ist, bleibt er in der Menge meiner Kumpel oder ist er sogar zweimal in dieser Menge?

Damit haben wir schon einige wichtige Themen für dieses Kapitel angesprochen:

- ✔ Mengen sind ein vielseitig verwendetes Datenmodell.
- ✔ Dinge, die als Menge modelliert werden, können nicht als Menge gespeichert werden. »Menge« ist keine Datenstruktur, nicht im Keller und nicht im Speicher. Sie sind eine Datenabstraktion und es geht darum, was die »richtige Datenstruktur« ist, die eine Menge repräsentiert – vor allem dann, wenn es um eine große Menge von Daten geht.
- ✔ Mengen enthalten nie zweimal das Gleiche. Aber wir können und müssen definieren, was »gleich« bedeutet. Das ist – vielleicht überraschenderweise – ein interessantes und wichtiges Thema.

Ich bin eine Menge

Eine Menge ist etwas sehr Einfaches. Es gibt Dinge, die drin sind, und alle anderen Dinge sind nicht drin. Also muss man sagen können, wer drin ist. Dazu gibt es zwei Methoden. Man kann die Elemente aufzählen oder man kann sie beschreiben. Zum Beispiel so:

$$M = \{2, 4, 6, 8, 10\}, \quad M = \{x \in \mathbb{N} \mid x \leq 10, x \text{ gerade}\}$$

Die wichtigste Operation auf einer Menge ist der Test, ob etwas in der Menge ist.

$$5 \in M = \mathit{false}$$

Dann gibt es die Mengenoperationen Vereinigung, Schnitt und Differenz (Restmenge). Beispielsweise:

$$\{2, 4, 6\} \cup \{1, 2, 3\} = \{1, 2, 3, 4, 6\}$$

$$\{2, 4, 6\} \cap \{1, 2, 3\} = \{2\}$$

$$\{2, 4, 6\} \setminus \{1, 2, 3\} = \{4, 6\}$$

Es gibt keine Datenstruktur »Menge«. Mengen haben keine Struktur, also kann es keine Struktur »Menge« geben. Wir können nur eine existierende Datenstruktur nehmen und diese dann mit den Mengenoperationen ausstatten, sodass sie so tut, als sei sie eine Menge. So etwas nennt man *abstrakter Datentyp* oder auch *Datenabstraktion*.

Jede Programmiersprache hat Datenabstraktionen in ihrer Grundausstattung. Die Hardware kann außer Bits nichts speichern oder verarbeiten. Alle Zahlen und Zeichen sind schon Datenabstraktionen, die von der Programmiersprache geliefert werden (siehe Abbildung 9.3).

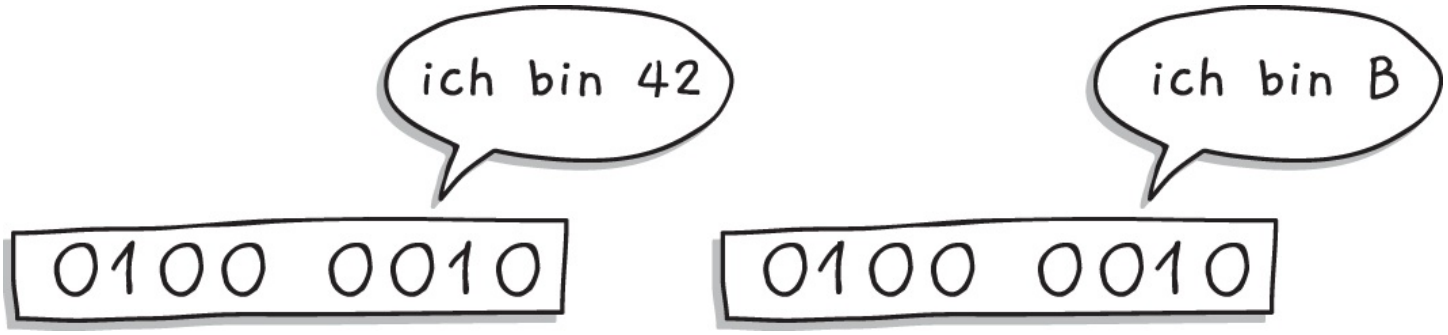


Abbildung 9.3: Das Konzept der Datenabstraktion

Es gibt Programmiersprachen, bei denen Mengen zur Grundausstattung gehören, die sind aber eher selten. Mengen gibt es in der Regel nur entweder in der mitgelieferten Bibliothek (der »API«) der Sprache oder gar nicht. Sie müssen dann selbst definiert werden und dazu gehört die Auswahl einer geeigneten Datenstruktur, welche die Menge repräsentiert (siehe Abbildung 9.4).

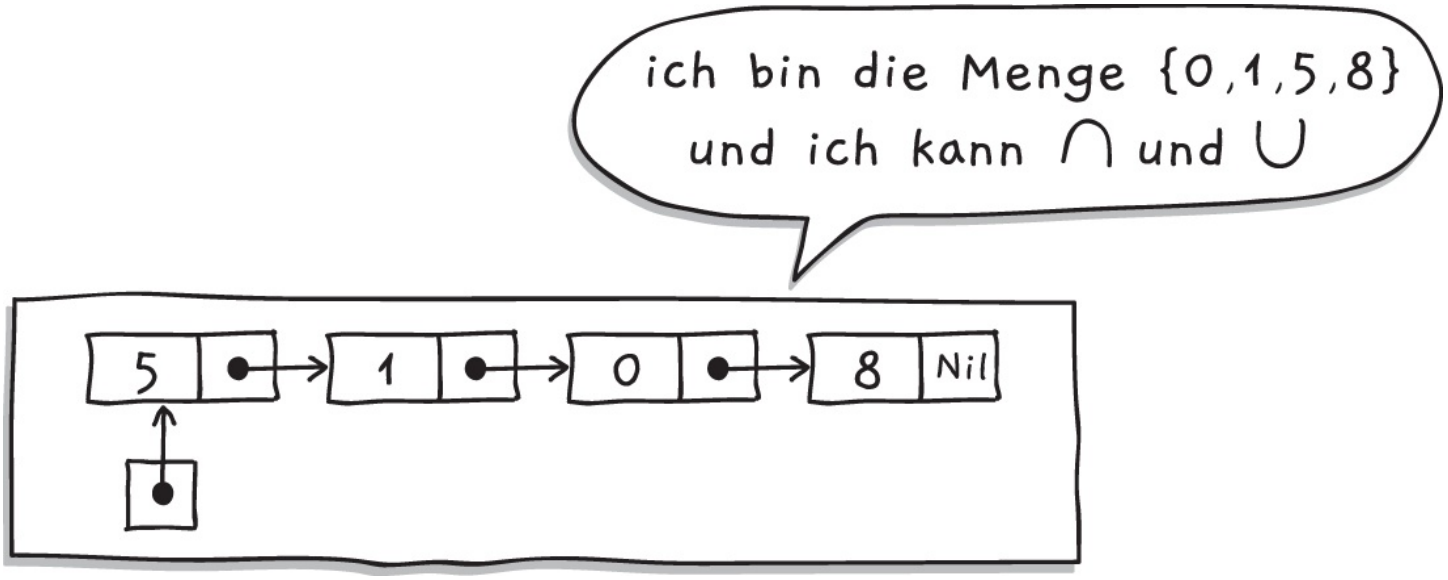


Abbildung 9.4: Menge als Datenabstraktion



Ein *abstrakter Datentyp*, auch *Datenabstraktion* genannt, ist eine Datenstruktur, die zusammen mit den passenden Operationen einem anderen – Mensch oder Codestück – vorgaukelt, etwas anderes zu sein.

Arrays und Listen sind naheliegende Datenstrukturen für Mengen. Arrays sind eher geeignet, wenn die Zahl der Mengenelemente nicht zu stark variiert. Listen sind flexibler und natürlich auch eher geeignet, wenn die Menge funktional realisiert werden soll.

In der Softwaretechnik legt man Wert darauf, dass das Innere, die Repräsentation, also die Datenstruktur, auf der alles basiert, und das Äußere, die Zugriffsoperationen, streng getrennt werden. Das soll hier aber nicht weiter von Bedeutung sein.

Imperative Datenabstraktion für Mengen

Im imperativen Konzept ist eine Menge, so wie prinzipiell alles andere, etwas, das sich ändern kann. Die Abstraktion besteht darin, dass der Zugriff auf die gespeicherten Daten ausschließlich über Operationen erfolgt, die in unserem Fall den Eindruck einer Menge erwecken sollen. Man beachte, dass man mit dem imperativen, nicht-funktionalen Ansatz »Mengen« nur sehr grob mit dem mathematischen Konzept einer Menge in Verbindung bringen kann.

Definieren wir mal eine *imperative* Datenabstraktion für Mengen. Wir geben ihr die Syntax eines Records. Klassen, Vererbung und anderes aus der Objektorientierung spielt hier keine Rolle und ohne Vererbung sind Objekte nichts anderes als Records, die auch Funktionen enthalten können (siehe Abbildung 9.5).

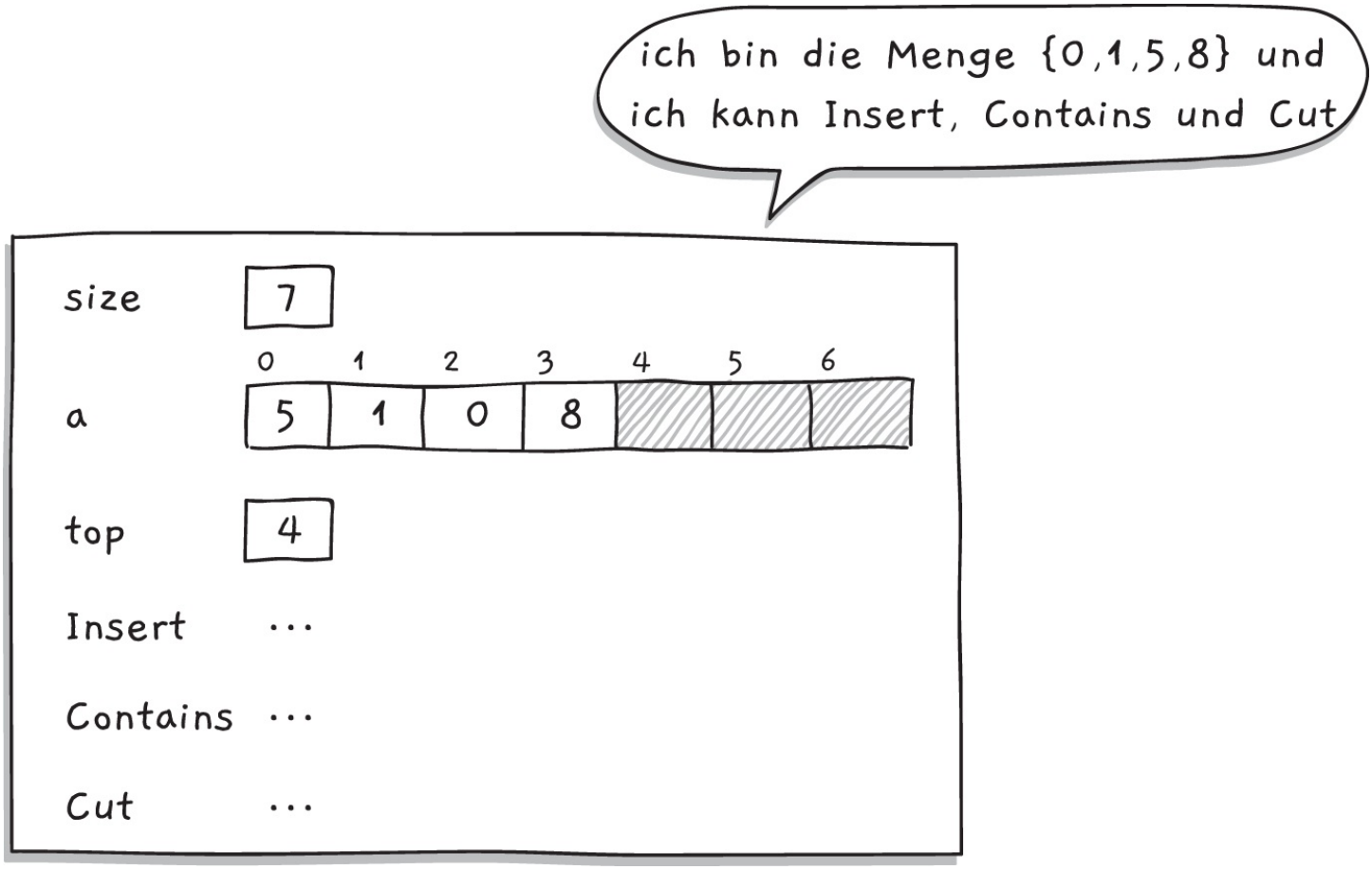


Abbildung 9.5: Menge als imperative Datenabstraktion mit Array-Repräsentation

Eine veränderliche »Menge« wird als Objekt modelliert. Objekte sind Erweiterungen von Records um Operationen. Wir definieren Set als eine Funktion, die eine Datenstruktur zusammen mit ihren Zugriffsoperationen erzeugt. Set hat ein Argument $size$, das die maximale Zahl der Elemente angibt, die in dieser Menge gespeichert werden können. Die Datenstruktur besteht aus einem Array a für die Elemente und einer Variablen top, die anzeigt, wie weit a aktuell gefüllt ist.

```
Set(size) = record [
    a: array[0 .. size] of Int ← [0, ... 0],   Daten
    top: Int ← 0,                              Grenze der Belegung
Insert(x)                                      Element einfügen
    if Contains(x)                             schon drin, wird ignoriert
        return
    if top < size
        a[top] ← x
        top ← top + 1
Contains(x)                                    Test, ob x enthalten ist
    for i ← 0 .. top − 1
        if a[i] = x
            return true
    return false
Intersection(otherSet)                         Schnitt mit einer anderen Menge
    i ← 0
    while i < top
        if not otherSet.Contains(a[i])
            a[i] ← a[top − 1]
            top ← top − 1
        i ← i + 1
=(otherSet)                                    Gleichheit mit einer anderen Menge als Operator =
    if top ≠ otherSet.top                      die andere Menge ist gleich groß
        return false
    else                                       und diese ist eine Teilmenge der anderen
        i ← 0
        while i < top
            if not otherSet.Contains(a[i])
                return false
    return true
]
```

Bei Einfügen wird getestet, ob das Element bereits vorhanden ist, wenn nicht, wird es hinzugefügt. Wenn doch oder wenn a voll ist, dann wird die Aufforderung zum Einfügen ignoriert. Bei der Schnittmengenbildung *Intersection* wird diese Menge mit einer anderen geschnitten. Es werden alle Elemente entfernt, die nicht auch Element der anderen Menge sind.

Mit $=$ wird getestet, ob zwei Repräsentanten die gleiche Menge repräsentieren, auch wenn sie (die Repräsentanten) eventuell nicht gleich sind. Wir verwenden einfach das

Gleichheitszeichen (=) als Name, obwohl nicht jede Programmiersprache diesen freien Umgang mit $=$ zulässt. Zwei Records repräsentieren die gleiche Menge, wenn bei beiden im Array a an den relevanten Positionen die gleichen Elemente zu finden sind, ohne dabei auf die Reihenfolge zu achten. Wiederholungen sind ausgeschlossen, darum reicht es zu prüfen, ob beide die gleiche Anzahl an relevanten Elementen haben und jedes Element der einen in der anderen Menge zu finden ist. Kurz gesagt prüft $=$ die Prämisse von

$$A \subseteq B \ \wedge \ |A| = |B| \ \Rightarrow \ A = B$$

Funktionale Datenabstraktion für Mengen

Eine echte Mathematikerin wird sicher an die Grenze ihrer Leidensfähigkeit kommen, wenn wir ihr unsere imperative Datenabstraktion von oben als »Menge« verkaufen wollen. Eine Menge, die, wenn man sie mit einer anderen Menge schneidet, plötzlich weniger Elemente enthält, ist wie eine Fünf, die bei einer Subtraktion zu einer Vier wird. Igitt!

Bei einer funktionalen Sicht der Menge ist schon der Begriff »Einfügen« suspekt. Er erinnert zu sehr an das Unwort »Veränderung«. Wenn wir beim Einfügen bleiben wollen, dann handelt es sich um eine Funktion, die aus einer Menge und einem Element eine neue Menge erzeugt. Beim Mengenschnitt wird natürlich nichts aus einer Menge herausgeschnitten, sondern zwei Mengen eine neue Menge konstruiert.

Für »Funktionale« ist es naheliegend, neue Strukturen als *algebraische Datentypen* zu definieren. In unserm Fall der Mengen käme man damit zu folgender Definition:

type IntSet = **Add**(Int, **IntSet**) | **Empty**

Eine Menge ist leer oder sie ist durch Add aus einem Wert und einer Menge entstanden.

$\{1, 2\} \sim$ **Add**(2, **Add**(1, **Empty**)) *zur leeren Menge erst 1, dann 2 hinzufügen*

Diese Konstruktion ist etwas problematisch. Man kann »Mengen« konstruieren, die eigentlich gleich sein sollten, tatsächlich aber nicht gleich sind. In einer Menge gibt es keine Reihenfolge und es ist darum unerheblich, ob man eine Menge so

$$\{2\} \cup (\{1\} \cup \emptyset)$$

oder so

$$\{1\} \cup (\{2\} \cup \emptyset)$$

konstruiert. Das Ergebnis ist in beiden Fällen das Gleiche: $\{1, 2\}$. Aber

Add(2, **Add**(1, **Empty**)) *zur leeren Menge erst 1, dann 2 hinzufügen*

und

Add(1, **Add**(2, **Empty**)) *zur leeren Menge erst 2, dann 1 hinzufügen*

sind nicht gleich. Das ist keine Kleinigkeit. Will man Mengen implementieren, dann sollen sie sich auch wie Mengen verhalten. Was eine Ente sein will, muss schnattern und watscheln wie eine Ente, was eine Menge sein will, darf von Reihenfolgen nichts wissen.

$$\{2, 1\} = \{1, 2\}$$

Folglich muss für unsere Mengen-Schauspieler IntSet gelten:

Add(1, **Add**(2, **Empty**)) = **Add**(2, **Add**(1, **Empty**)) *so SOLLTE es sein*

Bei einem algebraischen Datentyp erzeugen unterschiedliche Konstruktionen jedoch immer unterschiedliche Exemplare.

$$\{1, 2\} \sim \textbf{Add}(1, \textbf{Add}(2, \textbf{Empty})) \neq \textbf{Add}(2, \textbf{Add}(1, \textbf{Empty})) \sim \{2, 1\}$$

Wir hätten, wenn auch unter einem irreführenden Namen, tatsächlich wieder Listen definiert.

Die Lösung für das Problem der fehlenden Gleichheit ist einfach. Zwei sehen nicht gleich aus, sollten aber gleich aussehen. Ein Topf Schminke oder besser noch eine Maske herbei und die Unterschiede werden verdeckt. Die Maske macht aus dem algebraischen Datentyp IntSet eine »Menge« – scheinbar. Nutzer der Menge können dann nicht mehr direkt auf die Datenstruktur (hier ein alg. DT) zugreifen, sondern nur noch auf eine Umhüllung (Maske), die diese Eigenschaften garantiert.

Mengen können nicht als algebraische Datentypen (alg. DT) definiert werden, denn mit einem alg. DT kann nicht ausgedrückt werden, dass zwei Dinge, die auf unterschiedliche Art konstruiert wurden, trotzdem unter Umständen das Gleiche sind. Sie werden als Datenabstraktionen definiert: Algebraische Datentypen hinter der Maske einer Datenabstraktion (siehe Abbildung 9.6).



Abbildung 9.6: Zwei Datenstrukturen als die gleiche Menge maskiert



Die beiden völlig unterschiedlichen Konzepte *algebraischer Datentyp* und *Datenabstraktion/abstrakter Datentyp* werden oft mit dem gleichen Akronym »ADT« angesprochen. Das fördert die allgemeine Verwirrung. Die Datenabstraktion ist die Maske, das vorgespiegelte Sein. Ein algebraischer Datentyp ist Fleisch, eine mögliche Form des realen Seins einer Datenstruktur.

Als funktionale Datenabstraktion kann die Maskierung leicht erreicht werden:

```
Set(rep: IntSet) = record [          Menge mit interner Repräsentation rep
    type IntSet = Add(Int, IntSet) | Empty

    Set()                            leere Menge erzeugen
        return Set(Empty)
Insert(x)                            neue Menge mit einem zusätzlichen Element
    if not Contains(x, rep)
        return Set(Add(x, rep))
    else
        return Set(rep)
Contains(x)                          Test, ob Element enthalten ist
    return containsR(x, rep)

Intersection(otherSet)               Schnittmenge erzeugen
    return Set(IntersectionR(rep, otherSet))
=(otherSet)                          Gleichheit mit einer anderen Menge
    case otherSet = Set(otherRep)
        if Length(rep) ≠ Length(otherRep)
            return false
        else
            return EqualsR(rep, otherRep)
containsR(x, r)                      Hilfsfunktionen für Contains auf Repräsentanten
    case r = Empty
        return false
    case r Add(h, t)
        if h = x
            return true
        else
            return containsR(x, t)
```

```
IntersectionR(rep, otherSet)        Hilfsfunktionen für Schnitt auf Repräsentanten
    case rep = Empty
        return Empty
    case rep Add(h, t)
        if otherSet.Contains(h)
            return Add(h, IntersectionR(t, otherSet))
        else
            return IntersectionR(t, otherSet)
    EqualsR(rep1, rep2)                 Hilfsfunktion für Gleichheit
            case rep1 = Empty
                return true
            case rep1 = Add(x, rest)
                return containsR(x, rep2) and EqualsR(rest, rep2)
]
```

Das Ganze hätten wir natürlich auch etwas billiger bekommen können, denn wie bereits erwähnt, der Typ der Repräsentanten *IntSet* ist eigentlich die bekannte Liste. Es ist nicht notwendig so etwas noch einmal zu definieren. Bei den Listen können wir auch die bekannten Listenfunktionen verwenden und das Ganze vereinfacht sich zu:

```
Set(rep: List[Int]) = record [      Menge mit interner Repräsentation rep
    Set()                           leere Menge erzeugen
        return Set(Nil)
Insert(x)                           Einfügen = neu mit einem zusätzlichen Element
    if not Contains(x, rep)
        return Set(x :: rep)
    else
        return Set(rep)
Contains(x)                         Test, ob Element enthalten ist
    return rep.contains(x)

Intersection(otherSet)              Schnittmenge erzeugen
    r ← Nil
    for e ← rep
        if otherSet.contains(e)
            r ← e :: r
    return Set(r)
```

```
    =(otherSet)                          Gleichheit der Mengenabstraktion
        case otherSet = Set(otherRep)
            if rep.length ≠ otherRep.length    die andere Menge ist gleich groß
                return false
            else                         und diese ist eine Teilmenge der anderen
                for e ← rep
                    if not otherSet.Contains(e)
                        return false
            return true
]
```

Damit haben wir eine Menge als funktionale Datenabstraktion mit Listenimplementierung definiert.



Wenn ein algebraischer Datentyp »als etwas anderes auftreten« soll, also als Datenstruktur in einer Datenabstraktion verwendet wird, dann müssen einige seiner Werte als gleich erklärt werden. Diese »erweiterte Gleichheit« ist der wesentliche Kern der Datenabstraktion.

Wir können Datenabstraktionen realisieren, indem wir Werte eines algebraischen Datentyps, also letztlich Bäume, konstruieren, denen wir dann eine bestimmte andere Bedeutung geben. Das läuft darauf hinaus, dass Bäume als gleich betrachtet werden, die es eigentlich gar nicht sind. Ungleiche Dinge als gleich zu betrachten, ist leicht möglich, man muss ja nur bestimmte Unterschiede ignorieren. Gleiche Dinge als ungleich betrachten: Das geht dagegen gar nicht.

Mengen als charakteristische Vektoren

Menge können also in funktionaler und in imperativer Art behandelt werden. Listen und Arrays sind dabei jeweils die naheliegenden Datenstrukturen, aber sie sind nicht die einzigen Möglichkeiten. Im Einzelfall sollte man immer auf die zu speichernden Daten und die Anforderungen des sie verarbeitenden Algorithmus achten.

Manchmal hat man es beispielsweise mit Mengen zu tun, deren Elemente aus einer überschaubaren Grundkollektion kommen, also mit Teilmengen einer kleinen Grundmenge. In diesem Fall ist die Speicherung in einem charakteristischen Array oft eine gute Wahl. Manche Kartenspiele verwenden beispielsweise unterschiedliche 32 Karten, die zu Spielbeginn an die Spieler ausgeteilt werden. Jeder Spieler hat dann eine Teilmenge der 32 Karten in der Hand.

Ordnet man jeder Karte eine Nummer zu, dann kann jedes Blatt, also die Karten, die ein Spieler in der Hand hat, durch einen Vektor v mit 32 binären Werten beschrieben werden: $v[i] = 0$, die i -te Karte gehört nicht dazu, $v[i] = i$, die i -te Karte gehört dazu (siehe Abbildung 9.7).

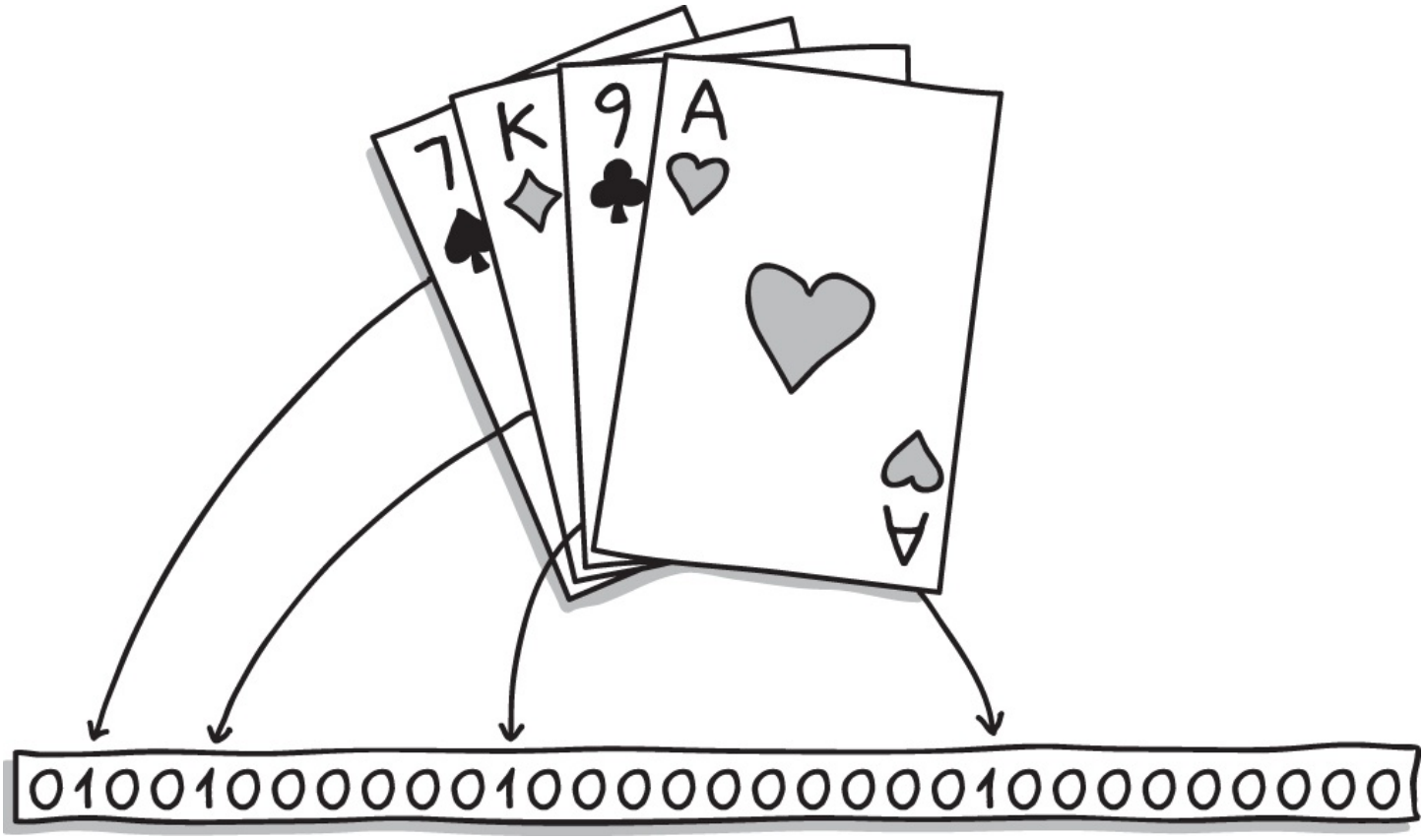


Abbildung 9.7: Eine Menge als charakteristischer Vektor

Operationen auf charakteristischen Vektoren sind natürlich unschlagbar schnell. Die Schnittmenge kann beispielsweise mit einer einfachen Schleife berechnet werden, so etwa $R = S \cap T$, wobei die beiden Mengen R und T jeweils durch einen charakteristischen Vektor $R.v$ und $T.v$ repräsentiert werden:

for $i \leftarrow 0 \,..\, length - 1$
    $R.v[i] \leftarrow S.v[i]$ **and** $T.v[i]$

Noch schneller geht es, wenn der charakteristische Vektor in ein Datenwort und damit ein Register passt und die Operation in der CPU in einem Maschinenzyklus ausgeführt werden kann.



Für Teilmengen $V \subset X$ einer kleinen Basismenge X sind charakteristische Vektoren die ideale Speichermethode. Sie benötigen wenig Speicherplatz und haben konstante $\mathcal{O}(1)$-Zugriffsmethoden.

Gut gehackt ist schnell gefunden

Die morgendliche Suche in Kleiderschrank nach einem passenden Outfit für den Tag ist schwierig, hat aber ein gewisses kreatives Potenzial. Richtig nervig wird die Suche aber, wenn man im kalten Werkzeugkeller nach einem Ersatz sucht für die verrostete Schraube, die Abflussrohr und Spüle zusammenhält. Jede Menge Zeug ist da. Alles schön in

Kästchen, aber leider ohne Beschriftung. Man muss jedes einzelne Kästchen aufziehen und den Inhalt inspizieren. Der Vergleich des Inhalts eines Kästchens mit unserem Suchkriterium dauert: Schublade aufziehen, durchwühlen, wieder zurückschieben und weiter (siehe Abbildung 9.8).



Abbildung 9.8: Suchen wird mit aufwendigem Prüfen langsam.

Die im Baumarkt machen das besser. Da hängen kleine Bildchen an den Regalen und man erkennt sehr viel schneller, was wo zu finden ist. Also fahren wir, statt im Keller zu suchen, lieber gleich in den Baumarkt und später machen wir es wie die Leute dort.

Es hilft also, wenn die Daten auf einen schnell und einfach vergleichbaren *Hashwert* reduziert werden. Für Menschen, mit ihrem auf das Erkennen optischer Muster optimierten Hirn, sind einfache Grafiken ideale Hashwerte (siehe Abbildung 9.9). Es kommt natürlich vor, dass mehrere Dinge den gleichen Hashwert haben. In der Schublade mit dem Bildchen »*Kleine Metallschraube*« sind wahrscheinlich neben der gesuchten M6-Schraube auch ein paar andere kleine Schräubchen zu finden, aber eine kleine Schublade zu durchsuchen, das können wir noch ertragen.

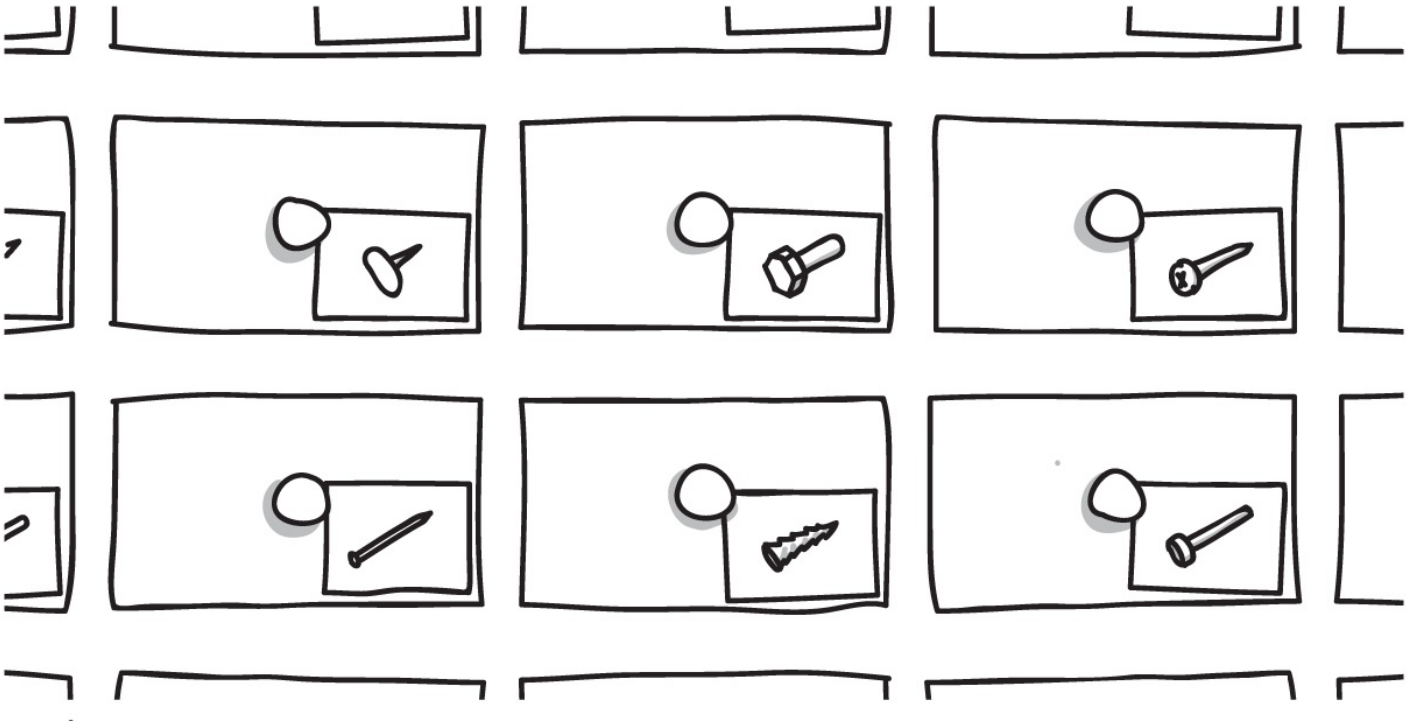


Abbildung 9.9: Speichern unter einem (hier optischen) Hashwert

Rechner haben es eher mit Zahlen, also verwenden sie besser Zahlen als Hashwerte. Zahlen als Hashwerte haben einen weiteren großen Vorteil: Legt man die Daten in einem Array ab, dann können die Hashwerte als Index verwendet werden. Die Suche nach der richtigen Schublade ist dann besonders schnell (siehe Abbildung 9.9).

Hashfunktionen

Bei einem charakteristischen Vektor gibt es eine feste und in beide Richtungen eindeutige Zuordnung von Mengenelementen zu Indexwerten. Ein potenzielles Element ist das erste, das nächste ist das zweite und so weiter. Jedes Element x der Menge der möglichen Elemente X hat einen eindeutigen Index i_x, der durch Durchnummerieren der Elemente von X festgestellt wird. Ein charakteristischer Vektor v repräsentiert eine Teilmenge von X. In v steht an Position i_x, ob x Element der Menge ist oder nicht.

$$x \overset{num}{\rightarrow} i_x \overset{v}{\rightarrow} v[i_x] \in \{0,1\}$$

Hashing erweitert diese Idee so, dass sie auch für Teilmengen von großen Mengen geeignet ist. Wollten wir beispielsweise eine Menge von Strings in einem charakteristischen Vektor speichern, dann müssten erst einmal alle Strings durchnummeriert werden und dann müsste auch noch für jede dieser Nummern ein Platz im charakteristischen Vektor reserviert werden.

Hashing modifiziert die Idee des charakteristischen Vektors in einem entscheidenden Punkt: Es gibt keine eindeutige Nummerierung der potenziellen Mengenelemente. Stattdessen bestimmt eine *Hashfunktion* h einen Index $h(x)$. An $v[h(x)]$ findet man dann die Information, ob x zur Menge gehört, die durch v repräsentiert wird.

$$x \xrightarrow{h} h(x) \xrightarrow{v} v[h(x)] \subset X$$

Die Hashfunktion h zerhackt also den Wert x zu seinem eigenen Index $h(x)$ (siehe Abbildung 9.10).

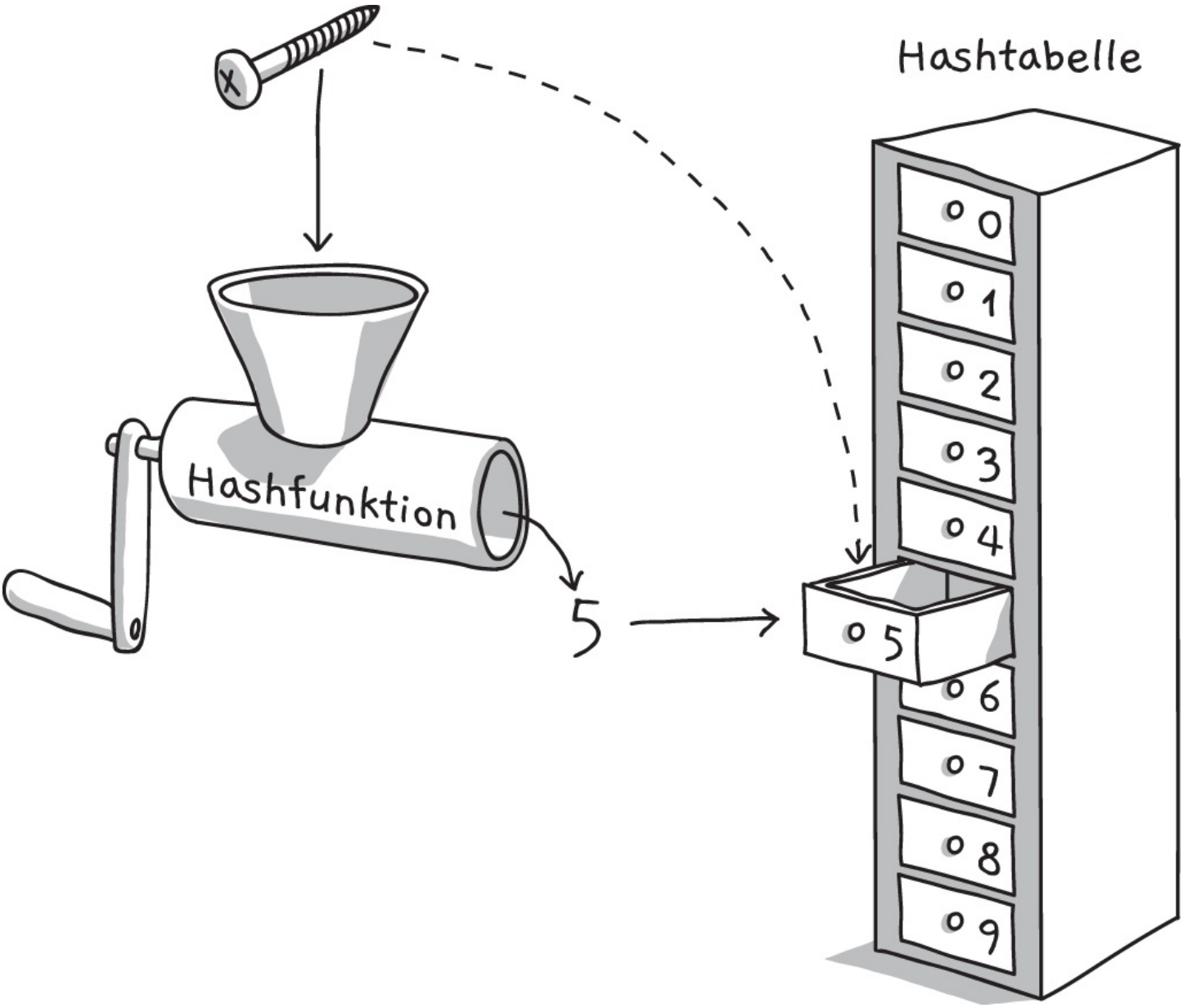


Abbildung 9.10: Speichern unter einem Hashwert

Jetzt können allerdings viele Elemente von h auf den gleichen Hashwert abgebildet werden. Wir können die Werte darum nicht einfach als Bitwert – da oder nicht da – in einem Array speichern. Haben x und y den gleichen Hashwert

$$h(x) = h(y) = i$$

Dann kann $v[i]$

- ✔ einen Wert, x oder y, oder

- ✔ zwei Werte, x und y oder
- ✔ keinen Wert

enthalten, je nachdem, ob x oder y oder x und y oder keiner von beiden zur der Menge gehört, die von v repräsentiert wird. $v[h(x)]$ ist darum einen Teilmenge von X : alle Elemente $x \in X$, deren Hashwert $h(x)$ ist und die in der von v repräsentierten Menge $V \subset X$ sind.

$$v[i] = \{x \mid h(x) = i \text{ und } x \in V\}$$

Das ist die Idee der Hashfunktion. Das Großartige dabei ist, dass das Nachschauen, ob ein Wert in einer Menge vorhanden ist, genauso schnell wir bei einem charakteristischen Vektor mit sensationellem $\mathcal{O}(1)$ entschieden kann.



Für Teilmengen $V \subset X$ einer großen Basismenge X sind Vektoren, die über eine Hashwert indiziert werden, die Alternative zu charakteristischen Vektoren. Sie bieten ebenfalls konstante $\mathcal{O}(1)$-Zugriffsmethoden.

Jetzt müssen wir uns noch etwas mit der Größe der Vektoren v beschäftigen. Selbst wenn h viele Werte auf den gleichen Index abbildet, ist es nicht sinnvoll, v exakt so groß zu machen, dass es für jeden möglichen Wert von h ein Plätzchen hat.

Hashtabellen

Eine *Hashtabelle* ist eine Datenstruktur zur Speicherung von Teilmengen $V \subset X$ einer Basismenge X. Sie besteht aus einem Array a der Größe N und einer Hashfunktion hashCode, die Elemente von X auf natürliche Zahlen abbildet. Nehmen wir als Beispiel Strings als Basismenge X und die Tabelle soll die Namen der Preisträger des Kaninchenzuchtvereins von Groß-Hobbelhausen enthalten.

Die Hashfunktion h (siehe Abbildung 9.11) nimmt Namen von Kaninchen und bildet sie ab auf einen Wert im Bereich $0, \ldots, N-1$. N und die Zahl der potenziellen Elemente stehen dabei zunächst einmal in keiner Beziehung. Wir beginnen mal mit einer übersichtlichen Tabelle mit $N = 10$, auch wenn es wesentlich mehr als 10 unterschiedliche Namen gibt. »Egon« und »Karla« haben Preise gewonnen und werden in die Tabelle eingetragen. Als erstes werden die Namen von der Hashfunktion in einen Zahlenwert verwandelt. Die Hashfunktion ist erst einmal sehr einfach definiert:

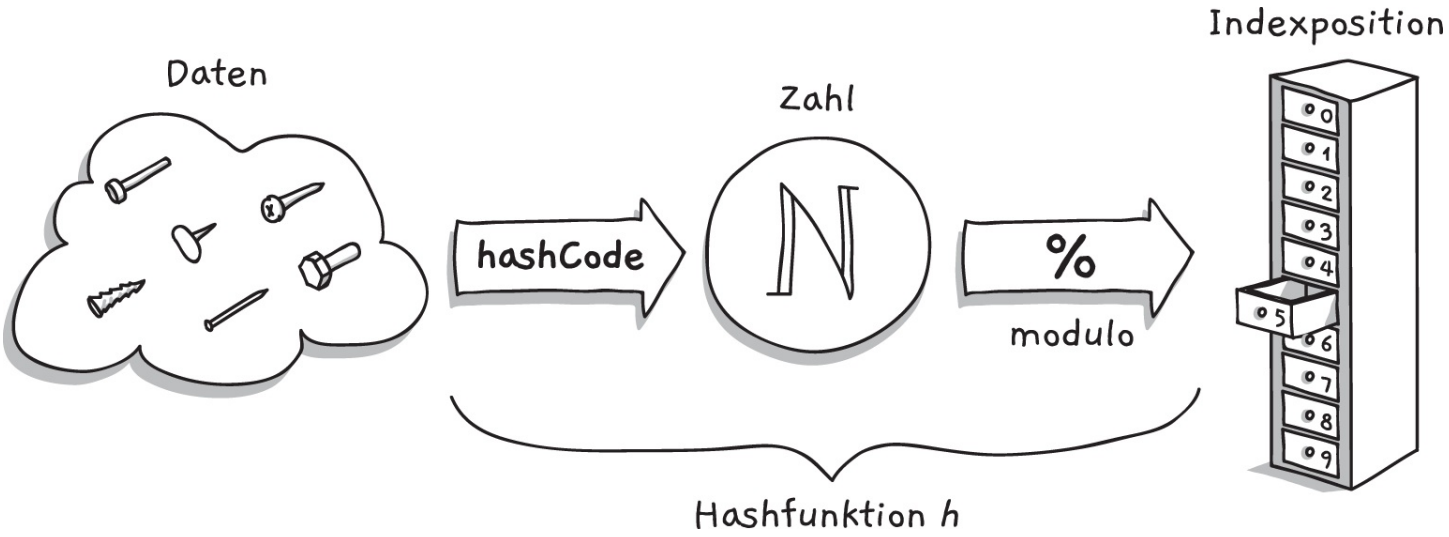


Abbildung 9.11: Hashfunktion

hashCode(S)
  case S = [$s[0], s[1], ..., s[n-1]$]
    return ord($s[0]$) + **ord**($s[1]$) 0 ... + **ord**($s[n-1]$)

Der String S wird in seine Zeichen zerlegt, jedes Zeichen wird von ord in eine Zahl abgebildet und alles addiert. ord ist eine beliebige Nummerierung der Zeichen, zum Beispiel der ASCII-Code. So könnten beispielsweise die folgende Hashwerte berechnet werden:

hashCode(Egon) = **ord**(E)+**ord**(g)+**ord**(o)+**ord**(n) = $69 + 103 + 111 + 110 = 393$
hashCode(Karla) = 491

Nun sind natürlich 393 und 491 keine zulässigen Indizes in einem Array der Größe 10. Wir könnten das Array an die Hashwerte anpassen und statt der Größe $N = 10$ beispielsweise $N = 500$ nehmen. Oder alternativ dazu die Hashfunktion an die Größe des Arrays anpassen.

Es gibt aber noch eine bessere Idee. Die Hashfunktion passt sich an die Tabellengröße an. Die Tabellengröße kann dann an den Grad der Befüllung angepasst werden, ohne dass etwas geändert werden muss. Als Index nehmen wir einfach den Hashwert modulo der Tabellengröße:

```
HashTable = record [
    N ← 10
    a ← array[0 .. N−1]

    hashCode(S)
        case S = [ s[0], s[1], ..., s[n − 1]] ]
            return ord(s[0]) + ord(s[1]) +...+ ord(s[n − 1])
insert(s)
    hc ← hashCode(s)
    i ← hc mod N                    hier wird die Größe der Tabelle verwendet
    a[i] ← s

contains(s)
    hc ← hashCode(s)
    i ← hc mod N
    return a[i] = s
]
```

Kollisionen

Das funktioniert prima. Mit jedem prämierten Kaninchen wird die Tabelle voller. So kommen der Reihe nach hinzu:

Name	*Eintrag an Index*
Egon	3
Karla	1
Hobbler	2
Sissi	8
Rammler	0
Löffler	9
Bunny	4
Blacky	8

Die Tabelle füllt sich sogar erstaunlich gut. Die einfache Hashfunktion verteilt die Namen recht gleichmäßig auf die 10 Indexpositionen. Ein Problem kann aber nicht ausgeschlossen werden: Früher oder später muss es passieren, dass zwei Namen auf die gleiche Indexposition abgebildet werden. In unserem Fall kommen sowohl Blacky als auch Sissi an Position 8. Das nennt man *Kollision*. Da Blacky später hinzugefügt wird, wird Sissi verdrängt, das heißt aus der Tabelle gelöscht. Nun mag man sagen, dass Sissi sowieso kein anständiger Name für ein preisgekröntes Kaninchen ist, aber das wäre doch etwas unfair.

Mit einer Kollision kann man auf unterschiedliche Arten fairer umgehen. Eine naheliegende Methode ist, alle kollidierende Werte in einen *Eimer* (engl. *bucket*) zu stecken. Jeder Eimer enthält dann jeweils alle Elemente mit einem bestimmten Index in der Tabelle (Hashwert modulo Tabellengröße; siehe Abbildung 9.12). Später, beim Nachschauen, muss dann natürlich der Eimer durchsucht werden. Je größer die Tabelle ist, umso geringer ist die Wahrscheinlichkeit einer Kollision. Die Zahl der Leerstände nimmt dann aber natürlich auch zu.

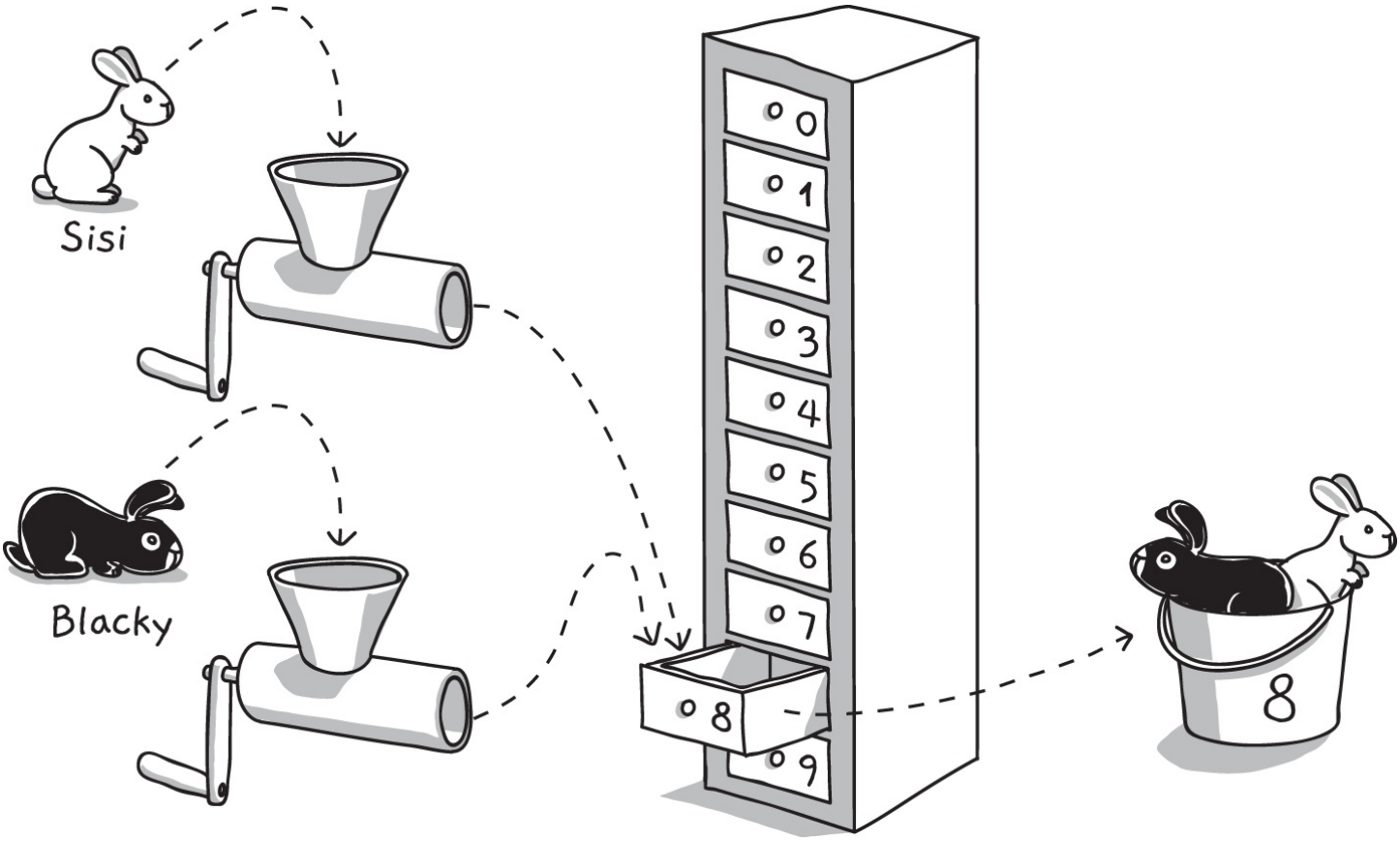


Abbildung 9.12: Kollision: Zwei in einen Eimer

Ein Eimer enthält alle Elemente e mit $h(e) \bmod N = x$ und wird in der Tabelle unter $a[x]$ abgespeichert. Die Eimer werden üblicherweise als verkettete Listen implementiert.

Es gibt auch Verfahren, bei denen Kollisionen dadurch behandelt werden, dass man die Hashfunktion so abändert, dass keine Kollisionen mehr auftreten. Die Anpassung der Hashfunktion muss natürlich bei der nächsten Kollision wiederholt werden. Dieses Verfahren ist heute eher unüblich.

Treten zu viele Kollisionen auf, dann ist die Hashfunktion schlecht oder die Tabelle ist zu voll. Eine schlechte Hashfunktion streut die Werte nicht breit genug, sondern konzentriert sie auf bestimmte Werte. Je voller die Tabelle ist, umso wahrscheinlicher sind Kollisionen, egal wie gut die Hashfunktion ist. Die Reaktion darauf ist einfach: Man nimmt eine größere Tabelle und kopiert alle Werte in die neue Tabelle. Die Tabelle und damit die Zahl der Eimer wird so erhöht. Der Index der Eimer wird modulo Tabellengröße berechnet und

passt sich damit automatisch an die neue Größe an.

Hashfunktionen

Beim Hashing wird ein meist sehr großer Bereich an Daten durch die Hashfunktion auf einen Bereich natürlicher Zahlen abgebildet und dieser Bereich dann mit der Modulo-Operation an die Tabellengröße angepasst (siehe Abbildung 9.11).

Der englische Begriff *to hash* bedeutet wörtlich *zerhacken, schreddern*. Es ist gut, wenn die Hashfunktion ihrem Namen alle Ehre macht. Sie sollte ihr Argument wirklich gut schreddern und die Brocken dann ordentlich durchmischen. Das sorgt dafür, dass sich die »gehashten«Werte nicht in wenigen Stellen zusammenklumpen, sondern über den Bereich der Hashwerte gut verteilen. Damit wird die Wahrscheinlichkeit von Kollisionen verringert.

Dabei sollte man bedenken, dass die aus den Werten berechneten Zahlen noch an die Tabelle angepasst werden. Dazu werden sie mit der Modulo-Operation bearbeitet. Die höherwertigen Bits werden dabei entfernt. Die Hashfunktion sollte darum für Werte sorgen, die sich möglichst gut in den niederwertigen Bits unterscheiden.

Die Qualität einer Hashfunktion hängt nicht nur von der Funktion selbst, sondern auch von der Wahrscheinlichkeitsverteilung der Werte ab. In wirklich kritischen Fällen sollte diese analysiert und bei der Konzeption der Hashfunktion berücksichtigt werden. Ansonsten reicht es normalerweise aus, wenn man beispielsweise die sogenannte Divisionsrest-Methode verwendet, um einen Hashwert zu berechnen. Dabei wird einfach der Rest $s \bmod p$ bei Division von s durch eine Primzahl p verwendet.

$$h(s) = s \bmod p$$

Als p kann man die Tabellengröße nehmen, falls diese eine Primzahl ist. Ansonsten nimmt man eine andere Primzahl, die größer ist als die Tabellengröße N:

$$h(s) = (s \bmod p) \bmod N$$

Das setzt natürlich voraus, dass s eine natürliche Zahl ist. Wenn andere Daten »gehasht« werden sollen, dann müssen diese zuerst in natürliche Zahlen transformiert werden. Dabei muss man darauf anchten, dass möglichst alle Bits eines Datums bei der Umwandlung in eine Zahl mitspielen. Ein Beispiel für die Berechnung des Hashwerts für Zeichenketten ist beispielsweise:

hashCode(S)
  case $S = [\, s[0], s[1], ..., s[n-1]] \,]$
    return $(s[0] * 31^{n-1} + s[1] * 31^{n-2} + ... + s[n-1]) \bmod p$

wobei p eine Primzahl ist, die größer als die Tabellengröße ist.

Laufzeiten

Beim Zugriff auf ein Element in einer Hashtabelle muss zuerst die Hashfunktion

berechnet werden, dann wird in der Tabelle der Eimer, also die Liste der Elemente mit gleichem Hashcode selektiert und diese Liste durchsucht. Die Berechnung des Hashcodes ist unabhängig von der Größe der Tabelle und der Eimer und kann mit $\mathcal{O}(1)$ abgeschätzt werden.

Wenn in einer Tabelle der Größe N insgesamt n Elemente abgespeichert sind, dann haben wir n Elemente auf N Eimer verteilt und ein Eimer enthält im Schnitt n/N Elemente. Die Suche in einem als verkettete Liste gespeicherten Eimer ist damit $\mathcal{O}(n/N)$. Das ist dann auch der Aufwand bei der Suche und dem Eintrag eines Elements. Vor dem Eintrag muss ja geprüft werden, ob es nicht schon enthalten ist.

Die Tabellengröße N sollte möglichst in der Nähe der Zahl ihrer Einträge sein, denn dann ist $n/N \sim 1$. Meist wachsen Tabellen während ihrer Nutzung. Es werden mehr Einträge hinzugenommen als gelöscht. Es ist darum üblich, dass die Tabellen regelmäßig an die Zahl ihrer Elemente angepasst werden. Bei Hashtabellen, die Bestandteil einer Programmiersprache oder von deren Standardbibliothek sind, passiert das in der Regel automatisch, ohne dass die Nutzer etwas davon bemerken.

Die Qualität der Hashfunktion ist keine rein theoretische Angelegenheit. Sie hat Einfluss auf die Effizienz der Zugriffe auf die Tabelle. Der Zugriff besteht ja aus zwei Schritten:

- ✔ Zugriff auf den Eimer: $\mathcal{O}(1)$
- ✔ Suche im Eimer : $\mathcal{O}(m)$

Dabei ist m die Größe des zu durchsuchenden Eimers ist. Im *schlechtesten Fall* tummeln sich alle Elemente in einem großen Eimer. m ist dann gleich n, der Zahl der Element in der Tabelle, und die Hashtabelle ist zu einer Liste degeneriert.

Die *durchschnittliche* Größe der Eimer ist die Zahl der Elemente n in der Tabelle dividiert durch die Zahl der Eimer (Tabellengröße N)

$$m = n/N$$

Wenn die Tabelle so groß ist, wie sie Elemente enthält, dann sind $n = N$ und $m = 1$. Ein Zugriff hat dann konstante Laufzeitkomplexität:

$$\mathcal{O}(1) + \mathcal{O}(m) = \mathcal{O}(1) + \mathcal{O}(n/N) = \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(1)$$

Kommen mehr Elemente hinzu, dann verschlechtert sich das. Eine größere Tabelle verbessert es aber nicht: Besser als $\mathcal{O}(1)$ kann es nicht werden. Im Idealfall entspricht die Tabellengröße genau der Zahl der gespeicherten Elemente und in jedem Tabellenplatz (Eimer) ist exakt ein Element gespeichert.

Garantiert gut gehackt

Leider hat die Laufzeitrechnung einen Haken: Groß-$\mathcal{O}$ steht ja für »$\mathcal{O}$bere Schranke«

und sie ist eine Worst-Case-Abschätzung für die Laufzeit (siehe Kapitel 2). Bei der Wahl der Hashfunktion kann man allerdings auch großes Pech haben, wenn man beispielsweise ganz viele Kaninchen abspeichern will, bei denen hashCode zufällig den gleichen Wert ausspuckt (siehe Abbildung 9.13).



Abbildung 9.13: Massenkollision: Zu viele im gleichen Eimer

In einem solchen Fall nützt auch die Vergrößerung der Hashtabelle nichts. Alles landet immer im selben Eimer und das ganze Gehashe hat überhaupt keinen Sinn. Da hilft nur eins: Die unbrauchbare Hashfunktion beim Händler reklamieren und gegen eine andere umtauschen! Mit der neuen Hashfunktion klappt es dann vielleicht besser.



Wenn man gegen alle Eventualitäten gewappnet sein will, reicht eine einzelne Hashfunktion nicht aus, sondern man braucht eine ganze Familie von Hashfunktionen, aus denen man auswählen kann. Das nennt man *universelles Hashen*.

Wo bekommt man nun die ganzen Hashfunktionen her? Am einfachsten ist es, eine simple Hashfunktion zu nehmen und sie mit ein paar zusätzlichen Parametern zu versehen, mit denen man sie variieren kann. Diese Parameter sind gleichsam Einstellknöpfe, an denen man drehen kann; verschiedene Kombinationen von Einstellungen führen immer zu neuen Arten des Hashings (siehe Abbildung 9.14).

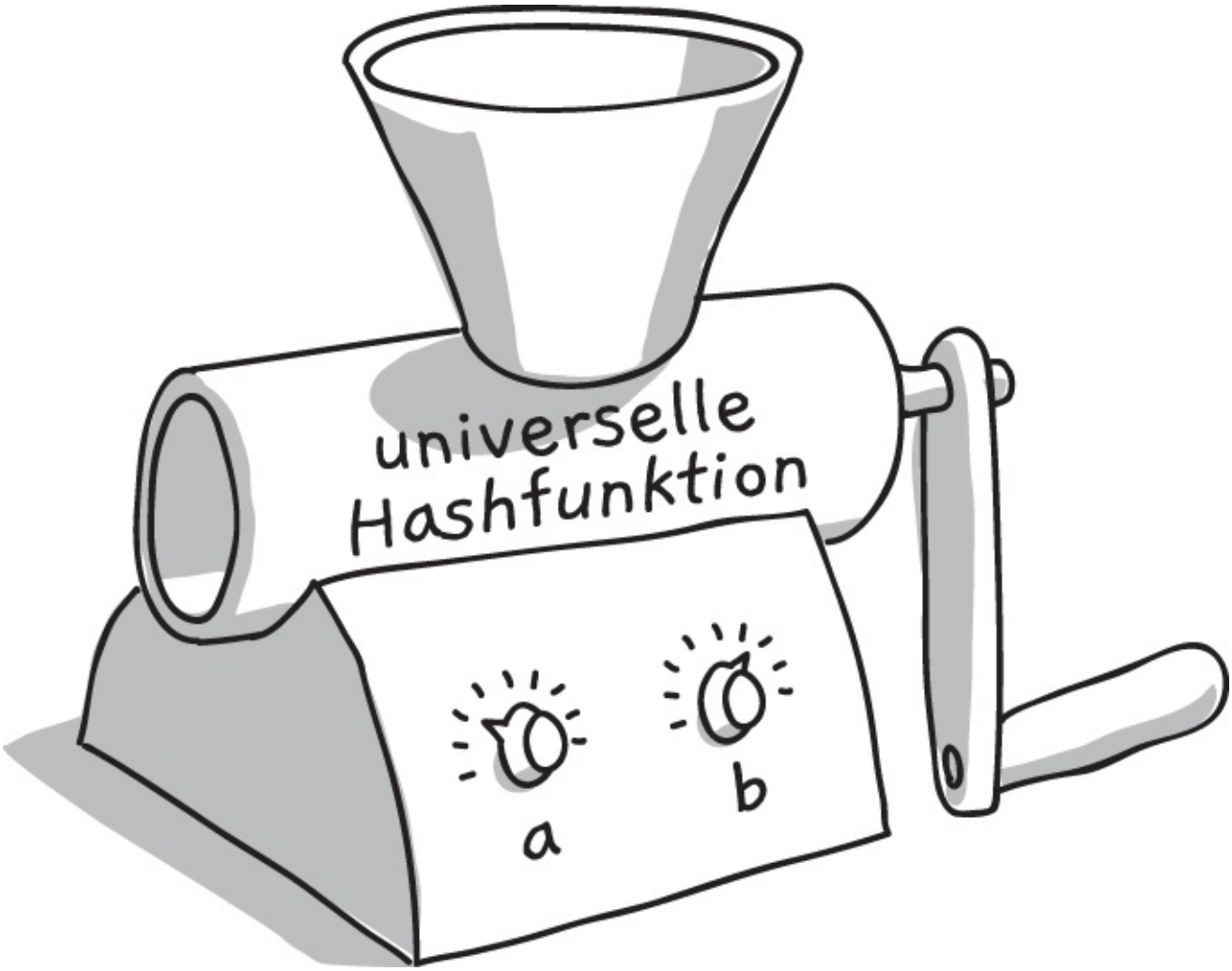


Abbildung 9.14: Universelle Hashfunktion mit zufällig eingestellten Auswahlknöpfen

Die einfachste und bei Weitem gebräuchlichste Methode, diese Parameter einzustellen, besteht darin, sie einfach irgendwie und völlig zufällig zu wählen. Wenn man das einmal gemacht hat, sollte man allerdings anschließend die Finger von den Knöpfen lassen. Sobald sich die Parameter auch nur ein wenig verstellen, ändert sich nämlich sofort die Hashfunktion und man findet in der Hashtabelle nichts mehr wieder. Die Parameter werden also *einmal* zufällig gesetzt und damit werden dann *alle* Elemente der Menge in die Hashtabelle gehasht.

Sollten dann irgendwann zu viele Elemente der Menge im selben Eimer landen, so muss man natürlich aktiv werden und die Parameter neu einstellen – sie also wieder zufällig auf irgendwelche Werte setzen. Anschließend muss man allerdings die Hashtabelle komplett neu aufbauen, das heißt, alle Elemente daraus entfernen und erneut in die Hashtabelle eintragen, diesmal mit der neu eingestellten Hashfunktion. Das nennt man »Rehashing« und es kann ein sehr aufwendiges Vergnügen sein, insbesondere bei großen Hashtabellen mit vielen Einträgen. Darum darf das besser nicht zu oft passieren, das heißt, man sollte die Familie von Hashfunktionen so wählen, dass bei einem Großteil der möglichen Parametereinstellungen die Daten, die man speichern möchte, auch tatsächlich gut auf die Hashtabelle verteilt werden. Ganz egal, welche Daten auch immer man abspeichern

möchte.

Wenn man großes Pech hat, stellt sich am Ende des Rehashings heraus, dass auch die neue Parametereinstellung nichts taugt. Dann wiederholt man den ganzen Prozess eben noch einmal, und noch mal, bis man endlich eine gute Hashfunktion gefunden hat. Theoretisch kann das ewig dauern, doch da man bei jedem Versuch eine sehr gute Chance hat, eine brauchbare Einstellung der Parameter zu erwischen, ist die Gefahr verschwindend gering.

Ein Beispiel für eine gute universelle Hashfunktion für ganze Zahlen s bekommt man, wenn man die einfache Hashfunktion aus dem Abschnitt *Hashfunktionen* erweitert: Zunächst legen wir eine feste große Primzahl p fest, so wie wir das oben bereits gemacht hatten. p sollte mindestens so groß sein wie die Tabellengröße N. Als Parameter wählen wir zwei beliebige Werte, a aus dem Bereich $\{1, 2, \ldots, p\}$ und b aus $\{0, 1, 2, \ldots, p\}$. Die Hashfunktion ist dann:

$$h_{(a,b)}(s) = ((a \cdot s + b) \bmod p) \bmod N$$

Die Hashfunktion hat damit zwei Einstellknöpfe a und b, die wir frei in den angegebenen Bereichen wählen können, um die Verteilung der Zahlen s auf die Hashtabelle zu verändern.



Übrigens ist es jetzt wichtig, dass man für p tatsächlich eine Primzahl benutzt. Wenn nämlich a und p einen gemeinsamen Teiler haben, verteilt die Hashfunktion die Zahlen nicht mehr gleichmäßig auf alle Felder der Hashtabelle. Nehmen wir als Beispiel $p = 30$, $a = 6$ und $b = 0$, dann kann $h_{(a,b)}$ nur Zahlen aus dem Bereich $\{0, 6, 12, 18, 24\}$ produzieren. Ist p hingegen eine Primzahl, so haben a und p keinen gemeinsamen Teiler größer als 1 und das Problem verschwindet.

Derselbe ist nicht immer der Gleiche

In Groß-Hobbelhausen hat der Kaninchenzuchtverein zwischenzeitlich mit seinen Prachtexemplaren immer neue Erfolge gefeiert. Das ist in einschlägigen Kreisen nicht unbemerkt geblieben. Der international bekannte Großinvestor Dagobert Gruhgel hat dem Vereinsvorsitzenden vorgeschlagen, dass die Preisträger dem Publikum in größerem Detail vorgestellt werden sollen. Weltweit soll jeder die Möglichkeit haben, Fan preisgekrönter Kaninchen zu werden und sie mit virtuellem Löwenzahn zu füttern, der zu günstigen Konditionen online erworben werden kann.

Als erster Schritt wird die Kaninchentabelle erweitert. Die Neugier des Publikums kann nicht mit einem simplen Namen befriedigt werden. Zu einem Kaninchen müssen mehr Daten als nur ein String gespeichert werden. Zum Namen kommen nun das Gewicht, das Alter und die Zahl der gewonnenen Preise hinzu. Ein Tabelleneintrag ist jetzt nicht mehr nur ein Name, sondern ein Record mit Name und Gewicht. Der diesjährige Preisträger, das Jungkaninchen Egon, erscheint als Record

```
type Kaninchen = record[name: String, age: Int, weight: Int, prices: Int]
egon ← Kaninchen(Egon, 2, 6, 1)
```

Die Tabelle wird entsprechend angepasst:

```
HashTable = record [
    N ← 10
    a ← array[0 .. N−1] of List()        Array von Listen als Eimer

hashCode(k)                              Hashwert von Kaninchen k berechnen
    ??

insert(k)                                Kaninchen k eintragen
    hc ← hashCode(k)
    i ← hc mod N
    if not a[i].contains(k)              noch nicht im Eimer a[i]?
        a[i] ← k :: a[i]                 dann hinein mit ihm

    contains(k)                          Kaninchen k suchen
        hc ← hashCode(k)
        i ← hc mod N
        return a[i].contains(k)          ist es im Eimer a[i]?
]
```

Die Berechnung des Hashwerts fehlt noch. Hashwerte sind so zu berechnen, dass jedes Bit des Ausgangswerts eine Rolle spielt. Wir lassen dazu einfach das Alter, das Gewicht und die Zahl der Preise so in die Berechnung einfließen wie die Buchstaben des Namens:

```
hashCode(k)                              Hashwert von Kaninchen k berechnen
case k = Kaninchen(n, a, w, p)
    return (a · 31^(n+2) + w · 31^(n+1) p · 31^n +
        s[0] · 31^(n−1) + s[1] · 31^(n−2) + ... + s[n − 1]) mod 17
```



31 ist keine magische Zahl, sondern eine Primzahl und die Multiplikation mit 31 kann effizient mit einer Shift-Operation und einer Subtraktion berechnet werden:

$$x \cdot 31 = x \cdot (2^5 - 1) = x \cdot 2^5 - x = (x << 5) - x$$

Bei den Multiplikationen kommt es regelmäßig zu einem arithmetischen Überlauf, der wie eine Modulo-Operation wirkt. Multiplikationen modulo 2^{32} mit 31 (teilerfremd zu 2^{32}) führen zu einer ordentlichen »Verwirbelung« der Bits.

Nach 6 Monaten hat Egon ordentlich zugelegt, wiegt 12 statt 6 Kilo und gewinnt die Goldmedaille des Verbands der deutschen Riesenzüchter. Leider kann die Goldmedaille nicht an die Sektion Groß-Hobbelhausen des Verbands ausgehändigt werden. In deren Kaninchentabelle ist Egon nicht zu finden (siehe Abbildung 9.15). Ein Fall von Betrug und Korruption?

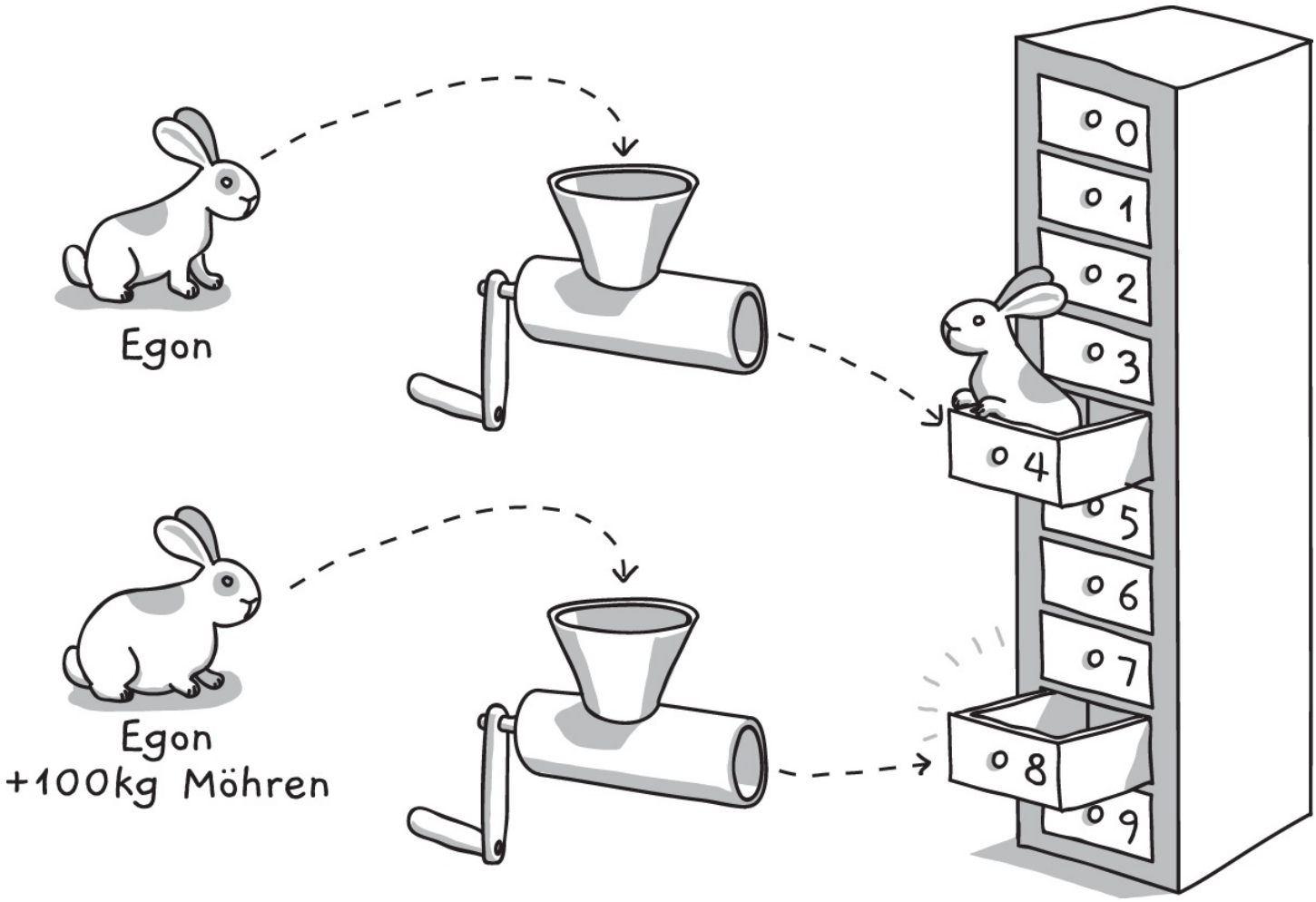


Abbildung 9.15: Egon: Derselbe, aber nicht mehr gleich

Nein kein Betrug, der Hashcode der Kaninchen wurde falsch berechnet. Das Gewicht geht dummerweise in die Berechnung ein, und da die Tabelle natürlich davon ausgeht, dass Egon stets den gleichen Hashwert hat, wird er bei der zweiten Suche nicht gefunden. Egon in seinem alten Zustand

Kaninchen(*Egon*, 2, 6, 1)

ist in der Tabelle, aber nicht Egon in seinem neuen Zustand

Kaninchen(*Egon*, 2, 12, 2)

mit jetzt zwölf Kilo und zwei statt einem gewonnen Preis.



In die Berechnung des Hashwerts dürfen nur die Attribute eines zusammengesetzten Wertes einfließen, die Bestandteil seiner Identität sind. Das sollten unveränderliche Attribute sein. Wenn nicht, dann hat man es mit einem Identitätsdiebstahl oder einem schweren Entwurfsfehler zu tun.

Egon wird durch seinen Namen identifiziert, also darf auch nur der Name in die Berechnung des Hashcodes einfließen.

```
hashCode(k)                          Hashwert von Kaninchen k berechnen
    case k = Kaninchen(n, a, w, p)
        return hashCode(n)
```

Es reicht aber nicht, nur die Berechnung des Hashcodes anzupassen. Versäumt man es, die Gleichheit in passender Weise zu definieren, dann kann es ebenfalls passieren, dass Dinge mehrfach in der Tabelle vorhanden sind.

Nach einer Korrektur der Hashfunktion wird Egon in jedem seiner Zustände auf den gleichen Hashwert abgebildet. Anschließend kommt aber noch die Suche im Eimer, also in der Liste der Objekte mit dem gleichen Hashwert wie Egon. Für diese Suche ist Gleichheit relevant, aber wohlgemerkt nicht die Gleichheit der Kaninchen, sondern die Gleichheit der Namen. Andernfalls haben wir in der Liste das gleiche (oder dasselbe?) Problem wir mit der Tabelle insgesamt: Der junge, schlanke Egon wird von dem alten, dicken Egon unterschieden.

```
HashTable = record [
    N ← 10
    a ← array[0 .. N−1] of List()        Array von Listen als Eimer
    hashCode(k)                          Hashwert ist der Hashwert des Namens
        case k = Kaninchen(name, age, weight, prices)
        return hashCode(name)
    ...
    isinBucket(k, lst)                   ist k im Eimer lst?
        case lst = Nil
            return false
        case lst = head :: tail
            if head.name = k.name        Gleichheit von Kaninchen ist Gleichheit der Namen
                return true
            else
                return isinBucket(k, tail)
]
```

Die Berechnung des Hashcodes ist hier ein Bestandteil der Hashtabelle. Das ist nur sinnvoll, weil es sich um eine auf Kaninchen spezialisierte Tabelle handelt. Die Definition der Hashfunktion muss zu den »gehashten« Daten passen, deshalb gehört sie eher zu den Daten als zu der Tabelle. Von einem softwaretechnischen Standpunkt aus sollten darum hashCode und die Gleichheit equals Attribute der Tabellenelemente und nicht der Tabelle sein:

```
HashTable = record [
    ...
    hashCode(k)                    Tabellenelemente definieren ihre Hashfunktion selbst
        return k.hashCode()
    ...
    isinBucket(k, lst)             ist k im Eimer lst?
        case lst = Nil
            return false
        case lst = head :: tail
            if head.equals(k)      Tabellenelemente definieren ihre Gleichheit selbst
                return true
            else
                return isinBucket(k, tail)
]
```

Die Hashfunktion und die Gleichheit müssen harmonieren. Dinge die konzeptionell gleich sind, müssen den gleichen Hashwert haben. Dinge, die den gleichen Hashwert haben, müssen aber nicht unbedingt gleich sein. Unterschiedliche Dinge sollten zwar auch möglichst unterschiedliche Hashwerte haben. Es liegt aber in der Natur der Sache und ist unvermeidlich, dass Unterschiedliches gelegentlich auch mal den gleichen Hashwert hat.



- ✔ Bedenke zuerst die (konzeptionelle) Gleichheit der Tabellenelemente,
- ✔ dann definiere h , sodass $x = y \Rightarrow h(x) = h(y)$.
- ✔ Oder kurz: *Wer equals sagt, der muss auch hashCode sagen!*

Hashtabellen sind ein ganz hervorragender Datenspeicher. Es ist aber nicht völlig trivial, die Dinge so zu gestalten, dass sie dort ordentlich abgespeichert werden.

- ✔ Für jeden potenziellen Wert muss die Funktion $hashCode$ definiert werden.
- ✔ Die Definition sollte möglichst breit über einen großen Bereich der ganzen Zahlen streuen.
- ✔ Vor der Definition von $hashCode$ muss bedacht werden, was es bedeutet, gleich zu sein, und die Gleichheit der gespeicherten Objekte muss mit ihrem $hashCode$ harmonieren:
    - Unterschiedliches darf, aber sollte möglichst selten den gleichen Hashcode haben.
    - Gleiches muss den gleichen Hashcode haben.

Viel ist oft eine Menge

Eine Menge ist oft ganz einfach ziemlich viel. Und sie wird auch gerne als Datenspeicher für ziemlich viel Information genutzt. Allerdings sind Mengen in aller Regel nicht Kollektionen von einfachen unstrukturierten Objekten. Man speichert strukturierte Informationen. Das sind die Kaninchen des Zuchtvereins, die Mieter eines Wohnblocks oder die Steuerpflichtigen eines Finanzamts. Das Kaninchen *Egon* hat beispielsweise einen Namen, eine Rasse, ein Alter und vielleicht hat es auch noch einige Preise gewonnen. Wie wir schon im letzten Abschnitt gesehen haben, ist die natürliche Art der Speicherung dieser Informationen ein Record:

```
egon ← record [
    name: "Egon Hobbel",
    age: 2,
    weight: 12,
    prices: 1
]
```

In dieser Form kann Egon in einer Tabelle gespeichert werden. Die Tupelform wäre auch Ok, man müsste nur festlegen, dass alle Tupel im gleichen Format sind: Name, Alter, Gewicht, Anzahl der Preise.

```
egon ← ("Egon Hobbel", 2, 12, 1)
```

Wir könnten Egon in die Tabelle stecken. Beispielsweise in die Tabelle der Kaninchen der Sektion Groß-Hobbelhausen des Verbands der Kaninchenzuchtvereine:

Name	*Alter*	*Gewicht*	*Preise*
…	…	…	…
Sissi Hase	1	2	0
Egon Hobbel	2	12	1
Gotthold von Grube	8	7	12
…	…	…	…

Eine Menge von Tupeln nennt man in der Mathematik *Relation*. Eine Relation ist eine Teilmenge des Kreuzprodukts. Ein einfaches Beispiel für eine Relation ist die Menge der Paare von natürlichen Zahlen, von denen die zweite doppelt so groß ist wie die erste. Diese Menge ist eine Teilmenge des Kreuzprodukts $\mathbb{N} \times \mathbb{N}$:

$$\{(1,1),(2,4),\ldots\} = \{(x,y) \in \mathbb{N} \times \mathbb{N} \mid 2 \cdot x = y\} \subset \mathbb{N} \times \mathbb{N} = \{(1,1),(1,2),\ldots,(2,1),\ldots\}$$

Unsere Tabelle der Kaninchen kann man als Relation sehen, also als Teilmenge des Kreuzprodukts

$$R = S^* \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} = \{(n, a, w, p) \mid n \in S^*, a \in \mathbb{N}, w \in \mathbb{N}, p \in \mathbb{N}\}$$

Ein bestimmtes Tupel r ist ein Element von R und beschreibt bekannte Fakten über ein Kaninchen. Die Kaninchen von Groß-Hobbelhausen bilden eine Relation, also eine Menge von Tupeln. Diese ist eine Teilmenge des Kreuzprodukts als Menge aller Informationen über alle denkbaren Kaninchen.

Schreibt man die Tupel einer Relation in beliebiger Reihenfolge untereinander, dann hat man eine Tabelle. Tabellen sind Relationen, deren Tupel man in eine Reihenfolge gebracht hat. Relationen sind Tabellen, von deren Reihenfolge man abgesehen hat.

Relationen bilden die Basis der (relationalen) Datenbanktechnologie. Eine Kollektion von Relationen wird *Datenbank* genannt. Die Grundideen der Datenbanken sind:

- ✔ Informationen werden als Relationen gespeichert, Relationen sind abstrakte Tabellen ohne definierte Reihenfolge.
- ✔ Informationen (Tupel) können zu einer Relation hinzugefügt, gesucht und gelöscht werden.
- ✔ Es gibt mächtige Operationen, mit denen ganze Tabellen verarbeitet werden können und dabei neue Tabellen mit neu arrangierten Informationen erstellen.

In den Tupeln einer Tabelle/Relation spielt eine Komponente oft eine besondere Rolle. Bei den Kaninchen ist es der Name. Er identifiziert ein ganz bestimmtes Kaninchen und muss darum eindeutig sein. In der Relation/Tabelle dürfen darum keine zwei Tupel mit dem gleichen Namen vorkommen. Der Name ist der *Schlüssel* (engl. *key*) der Relation.

Relationen, bei denen eine Komponente als eindeutiger Schlüssel herausgehoben ist, können als Funktionen betrachtet werden. Die Kaninchentabelle ist damit eine Funktion, die Namen auf die restliche Information des Tupels abbildet:

Name	⇒	***Info***		
		(Alter,	Gewicht,	Preise)
…	…	…	…	…
Sissi Hase	⇒	(1	2	0)
Egon Hobbel	⇒	(2	12	1)
Gotthold von Grube	⇒	(8	7	12)
…	…	…	…	…

Eine Relation mit einem Schlüssel ist damit eine Funktion, die endlich, gespeichert und nicht berechnet ist. So etwas wird auch *Verzeichnis* (engl. *dictionary*) oder Abbildung (engl. *map*) genannt. Manchmal findet man auch den Begriff *assoziatives Array*. Damit soll angedeutet werden, dass es sich um ein Array handelt, dessen Indizes keine natürlichen Zahlen, sondern beliebige Werte sind, die mit dem Inhalt einer Array-Position

verbunden (assoziiert) sind.

Verzeichnisse sind eine frühe Form der Datenbanktechnologie. Sie wurden zur Speicherung großer Mengen an Daten auf externen Speichermedien wie Festplatten und Bänder eingesetzt. Inzwischen beherrscht die wesentliche mächtigere Technologie der relationalen Datenbanken die Szene der externen Speicherung großer Datenmengen. Durch die Möglichkeit, Operationen auf ganzen Tabellen zu formulieren, ist ihr Datenmodell wesentlich mächtiger und flexibler.

Für einfachere Anwendungen, vor allem bei Daten, die im Hauptspeicher gehalten werden können, spielen Verzeichnisse aber immer noch eine herausragende Rolle. Jede anständige Programmiersprache für die Anwendungsentwicklung bietet sie in ihrem Sprachkern oder als Bibliothekskomponente an.

Als Datenstruktur werden Hashtabellen und Bäume verwendet. Oft hat man auch die Auswahl, denn beide haben ihre jeweils eigenen Vor- und Nachteile. Bis zur Erfindung der Hashtabellen waren Suchbäume die einzig bekannte Speicherform für Verzeichnisse oder allgemein größere Datenmengen. Sie wurden darum intensiv untersucht und es entstand daraus eine Vielzahl von Baumvarianten für Verzeichnisse.

Wer Ordnung hält, ist nur zu faul zum Suchen

Datenstrukturen werden so gestaltet, dass die Operationen auf ihnen möglichst effizient ausgeführt werden können. Unterschiedliche Operationen stellen unterschiedliche Anforderungen, und was gut für die eine Aktion ist, kann schlecht für die andere sein. Sollen Daten gespeichert werden, dann geht es am schnellsten, wenn man sie einfach auf einen Haufen wirft (siehe Abbildung 9.16). Die Geschwindigkeit, die man bei der Ablage gewinnt, bezahlt man dann natürlich später bei der Suche.

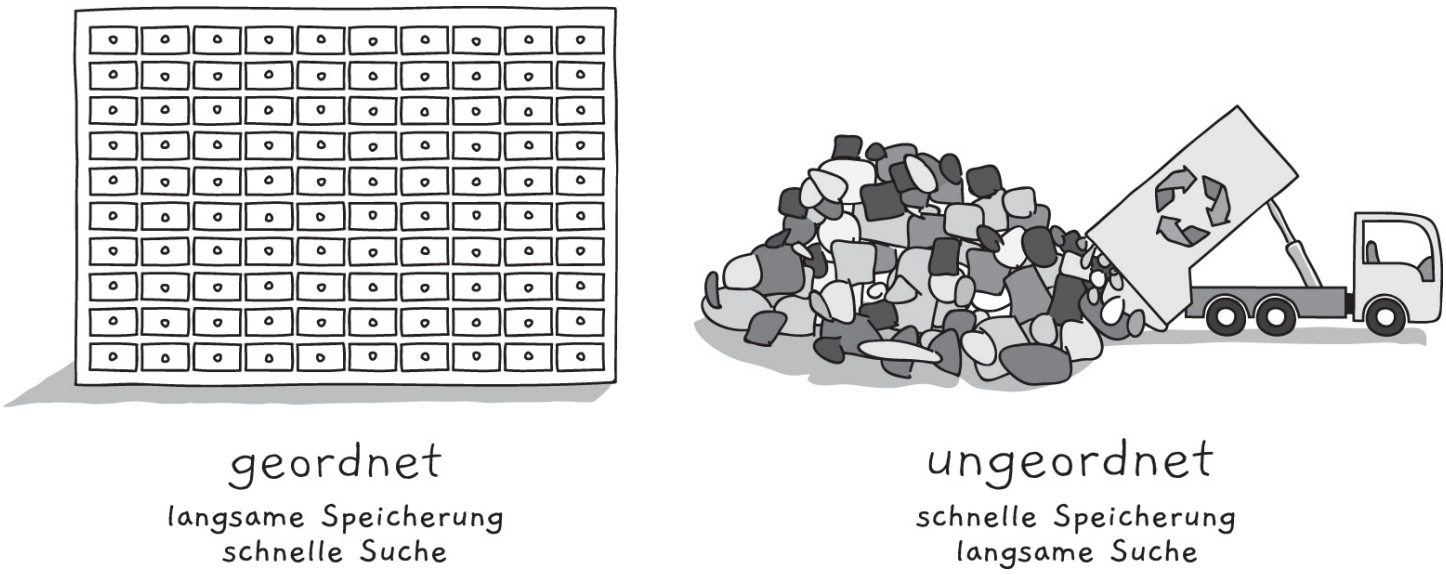


Abbildung 9.16: Optimale Datenstruktur für schnelle Suche (links) und schnelles Speichern (rechts

Es gilt hier, ein Gleichgewicht zu finden. Ganz wie im richtigen Leben, perfekte Ordnung

ist so aufwendig, dass die Mühe, die sie kostet, sich meist nicht lohnt. Komplette Unordnung ist auch nicht gut, die Geschwindigkeit bei der Ablage wird meist zu teuer bei der Suche bezahlt. Es ist darum sinnvoll, eine Organisationsform zu wählen, bei der die durchschnittliche Laufzeit aller auf dieser Struktur operierenden Algorithmen möglichst gering ist.

Die *mittlere Laufzeit* eines Algorithmus ist seine durchschnittliche Laufzeit über viele Ausführungen. Bei einer Datenstruktur mit zugeordneten Zugriffsoperationen ist es sinnvoll, diesen Begriff auf die mittlere Laufzeit aller Operationen zu erweitern. Dabei können durchaus auch mal besonders teure langsame Operationen in Kauf genommen werden, wenn damit andere Operationen beschleunigt werden. Das führt zu dem in Kapitel 2 bereits angesprochenen Begriff der *amortisierten Laufzeit*.

So etwas kennt manch einer auch aus eigener Anschauung: Einmal im Jahr wird das Gartenhaus richtig aufgeräumt. Das dauert zwar einen ganzen Tag, aber im restlichen Jahr braucht man es bei der Ablage nicht so zu genau zu nehmen und findet trotzdem alles einigermaßen schnell.



Bei der *Amortisierung* geht es darum, die durchschnittlichen Kosten von Operationen auf Datenstrukturen zu minimieren. Dabei können

- ✔ mittlere Laufzeiten unterschiedlicher Operationen optimiert und
- ✔ seltene kostspielige Operationen zu Beschleunigung häufiger Operationen eingesetzt werden.

Im Gartenhaus halten wir es also mit der Ordnung nicht so ganz genau. Damit geht Suchen und Ablegen in etwa gleich aufwendig. Damit die Unordnung nicht aus dem Ruder läuft, wird einmal im Jahr gründlich aufgeräumt. Natürlich muss man sich dabei Gedanken machen, ob die teure Operation sich tatsächlich lohnt und wie oft sie ausgeführt werden muss, um eine möglichst positive Wirkung zu haben (siehe Abbildung 9.17).



Abbildung 9.17: Amortisiert es sich, wenn ich hier einmal im Jahr richtig aufräume?

Bäume balancieren

Im Kapitel über Bäume haben wir Suchbäume als eine Alternative zur Speicherung von Datenkollektionen betrachtet. Der Suchbaum agiert dabei als halb-geordnete Struktur: Einfügen und Suchen erfolgen beide mit passablem und ausgewogenem Aufwand. Dabei ist auch nicht unerwähnt geblieben, dass ein Suchbaum eventuell zu einer Liste entarten kann. Das wollen wir nicht: Suchen und Einfügen in einem ordentlichen Baum ist in der Laufzeitklasse $\mathcal{O}(\log n)$. Dabei entspricht $\log n$ der Höhe eines Baums mit n Elementen. Ist der Baum zur Liste entartet, dann ist die Höhe n und wir sind wieder bei einer Komplexität von $\mathcal{O}(n)$.

Es könnte sich also lohnen (amortisieren!), den Baum von Zeit zu Zeit – oder vielleicht auch nach jeder Modifikation – wieder in Form zu bringen. Bevor wir tiefer in diese Thematik einsteigen, wollen wir ein paar Begriffe in Zusammenhang mit Bäumen

präzisieren (siehe Abbildung 9.18).

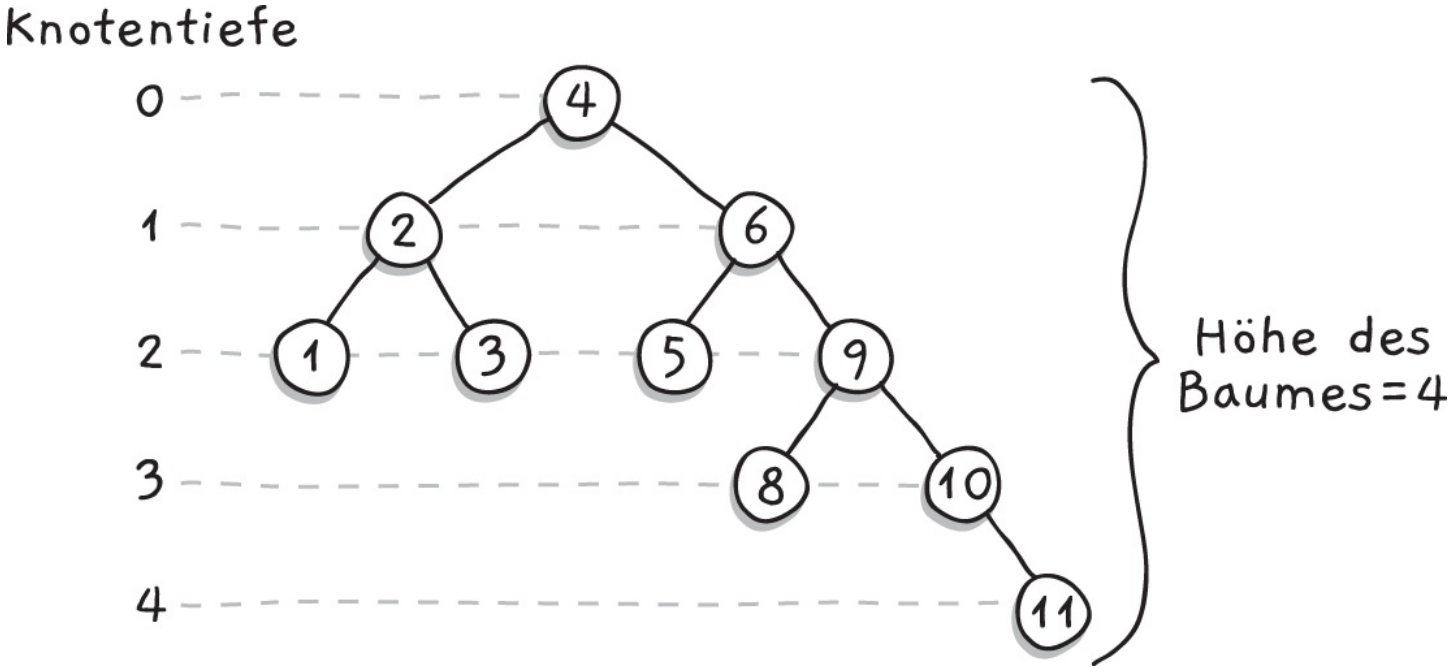


Abbildung 9.18: Die Baumhöhe ist die maximale Knotentiefe

- Die *Tiefe* eines Knotens ist die Länge des Pfades (Anzahl der Knoten) von der Wurzel zu diesem Knoten.
- Die *Höhe* $h(T)$ eines Baums T ist das Maximum der Tiefen all seiner Knoten. Der leere Baum hat die Höhe -1. Ein Baum mit einem Knoten hat die Höhe 0.
- Das *Gewicht* $g(T)$ eines Baums T ist die Anzahl seiner Blätter.
- Ein Baum ist *entartet*, wenn seine Höhe gleich der Zahl seiner Knoten ist.

Ein *balancierter Baum* ist ein ordentlich geformter Baum: ein Baum, der auch wie ein Baum aussieht. Statt balanciert sagt man auch *ausgeglichen*. Bei der Balancierung kann man auch noch etwas genauer sein. Zwei gängige Varianten der Balanciertheit (Ausgeglichenheit) von Bäumen sind höhen- beziehungsweise gewichtsbalanciert.

- Bei einem *höhenbalancierten* Baum ist die Differenz der Höhe der Unterbäume beschränkt.
- Bei einem *gewichtsbalancierten* Baum ist die Differenz des Gewichts der Unterbäume beschränkt.

Man kann auch den Grad der Balanciertheit messen. Wir beschränken uns hier auf höhenbalancierte Bäume. Die Definition für gewichtsbalancierte Bäume ist äquivalent. Bei einem k-balancierten Baum ist der Unterschied in der Höhe (dem Gewicht) der

Unterbäume eines Knotens beschränkt auf maximal k . In Abbildung 9.19 sehen wir einen 2-balancierten Baum: die Unterbäume mit dem größten Höhenunterschied sind T_L und T_R mit den Höhen 1 und 3. Damit ist die Anforderung der k -Balanciertheit für $k \leq 2$ erfüllt.

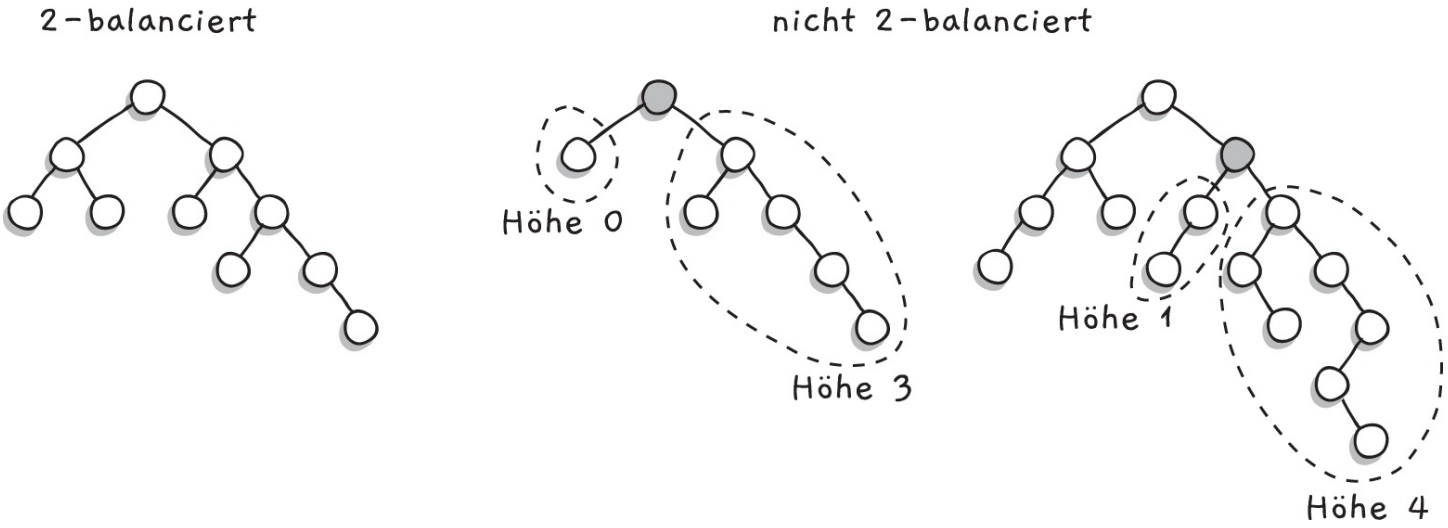


Abbildung 9.19: Höhenbalanciertheit von Bäumen

Ein k *-balancierter* Baum ist ein Baum, bei dem die Differenz der Höhe der Unterbäume der Knoten maximal k ist.

Ein k -balancierter Binärbaum T ist entweder leer oder er hat zwei Unterbäume T_L und T_R mit $|h(T_L) - h(T_R)| \leq k$. Mit k , also dem Grad der Balanciertheit, kann das Kosten-Nutzen-Verhältnis einer Balancierung variiert werden, bei der ein Baum durch Balancierung wieder in Form gebracht wird. Kleines k : sehr schön ausgewogene Bäume, aber es sind eventuell viele Reorganisationen notwendig, um diesen Zustand zu erhalten. Großes k : Schlechtere Baumform mit verlangsamten Zugriffsoperationen. Dafür sind dann aber natürlich viel weniger Reorganisationen notwendig.

So wie beim Gartenhaus ein gelegentliches Aufräumen sich amortisiert, so kann sich bei Suchbäumen ein gelegentliches Ausbalancieren lohnen. Um zu sehen, was da auf uns zukommt, schauen wir uns die unangenehmen Arbeiten mal direkt an. Bei den Bäumen ist die sogenannte *Rotation* die zentrale Aufräumaktion, mit der Bäume wieder in Balance gebracht werden.

Rotationen in Bäumen

Rotationen sind Aufräumarbeiten in Bäumen. Bei einer *Rotation* wird ein Baum umgebaut (Abbildung 9.20). Es gibt Rechts- und Linksrotationen; man kann sie auch als Rotationen im und gegen den Uhrzeigersinn bezeichnen. Beide kehren sich gegenseitig um, das heißt, wenn man einen Baum zum Beispiel einmal links rotiert und anschließend einmal rechts, so hat man am Ende wieder den ursprünglichen Baum. Rotationen sollen die Struktur verbessern, ohne dabei die allgemeinen Anforderungen an den Baum zu zerstören. Ein

binärer Suchbaum soll also auch nach einer Rotation ein korrekter binärer Suchbaum bleiben, nur halt besser in Form. Man kann nicht nur den ganzen Suchbaum rotieren, sondern natürlich auch jeden einzelnen seiner Teilbäume für sich und hat auf diese Weise viele Möglichkeiten, den Baum umzustrukturieren.

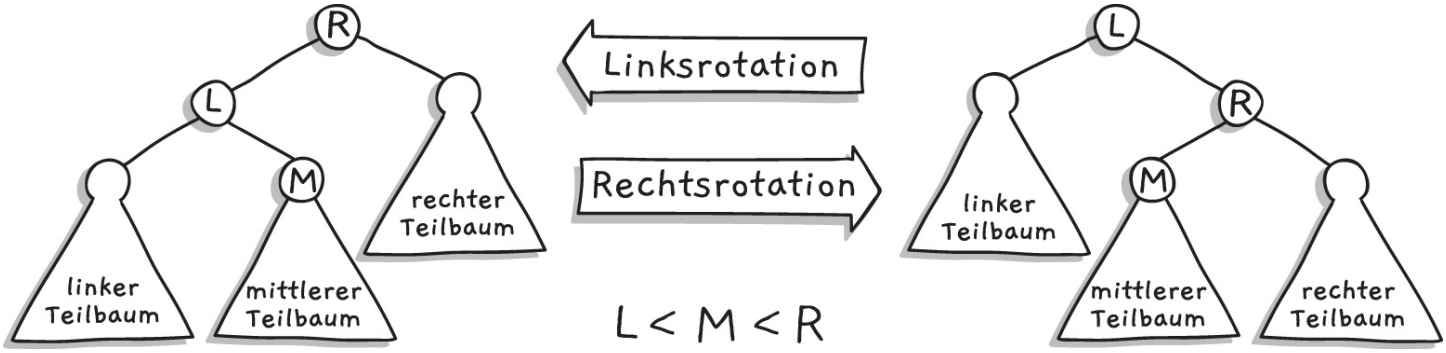


Abbildung 9.20: Rotation in Suchbäumen

Wir erinnern uns, dass binäre Suchbäume Bäume sind, deren Knoten maximal zwei Unterbäume haben und bei denen alle Werte im linken Unterbaum kleiner und alle Werte im rechten Unterbaum größer sind als der Wert der Wurzel. Der 2-balancierte Baum von Abbildung 9.18 ist ein binärer Suchbaum. Das ist auch schon gleich ein Baum, der einer Rotationsbehandlung unterzogen werden könnte, denn 2-balanciert ist schon ein wenig aus dem Gleichgewicht.

Bei einer *Linksrotation*, also einer Rotation gegen den Uhrzeigersinn, lässt sich ein Baum balancieren, wenn sein rechter Teilbaum deutlich tiefer ist als sein linker Teilbaum. Nehmen wir das Beispiel von Abbildung 9.21: Bislang ist $L = 4$ die Wurzel des Baumes, nun rückt $R = 6$ an diese Stelle und die alte Wurzel 4 wird zum linken Unterbaum der neuen Wurzel. Dafür muss 6 seinen bisherigen linken Unterbaum $M = 5$ loswerden, und zwar an 4, wo ja ohnehin ein neuer rechter Unterbaum benötigt wird.

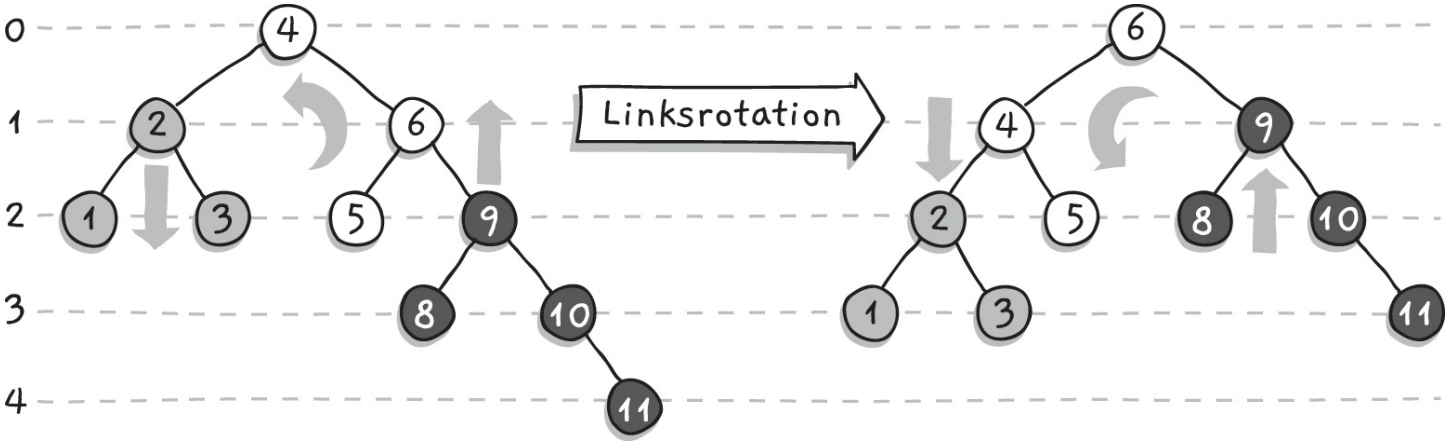


Abbildung 9.21: Linksrotation eines binären Suchbaums

Durch die Rotation bleibt der Suchbaum korrekt sortiert, das heißt, dass in linken Teilbäumen weiterhin nur Werte kleiner als der des Knotens stehen und in rechten Teilbäumen nur Werte größer als der Knotenwert. Geändert hat sich jedoch die Balance

des Baumes: Der rechte Teilbaum von Knoten 6 (in der Abbildung mit schwarzen Knoten) ist nach der Rotation eine Stufe weiter nach oben gewandert, während der linke Teilbaum von Knoten 4 (in der Abbildung mit grauen Knoten) nun eine Stufe weiter unten liegt.

Eine *Rechtsrotation*, also eine Rotation im Uhrzeigersinn, ist die komplementäre Aktion. Mit einer Rechtsrotation könnten wir die eben ausgeführte Linksrotation somit wieder rückgängig machen. In unserem Beispiel wird damit der Baum natürlich wieder in die alte unbalancierte Form gebracht. Eine Rotation muss also nicht unbedingt einen positiven Effekte haben – wir setzen Sie aber natürlich nur dann ein, wenn sie die Baumstruktur verbessert.

Rotationen sind zwar etwas haarige Operationen auf Zeigern, aber wir bemerken, dass nur wenige Zeiger betroffen sind und dass der Aufwand einer Rotation völlig unabhängig von der Größe des Baums ist. Rotationen können darum stets in konstanter Zeit, das heißt in $\mathcal{O}(1)$ durchgeführt werden.

Rot-Schwarz-Bäume

Balancierte Bäume sind eine sehr wichtige Klasse von Datenstrukturen für die interne (im Hauptspeicher) und externe Speicherung von großen Datenmengen. Es gibt sie in diversen Varianten. Die älteste wurde in den 1960ern von den russischen Wissenschaftlern G. M. Adelson-Velsky und E. M. Landis definiert und wird nach den Namenskürzeln der beiden *AVL-Baum* genannt. 1970 entwickelte der amerikanische Informatiker *2-3-Bäume*, bei denen ein Knoten zwei oder drei Kinder hat. AVL-Bäume haben heute nur noch aus historischem Interesse Bedeutung und auch 2-3-Bäume werden nicht mehr sehr oft eingesetzt. Den häufigsten Einsatz finden die sogenannten *Rot-Schwarz-Bäume*, die auf den Münchner Informatiker R. Bayer zurückgehen.

Ein Rot-Schwarz-Baum ist ein binärer Suchbaum, dessen Knoten entweder rot oder schwarz gefärbt sind (siehe Abbildung 9.22). Ein solcher Baum kann mit einem algebraischen Datentyp beschrieben werden:

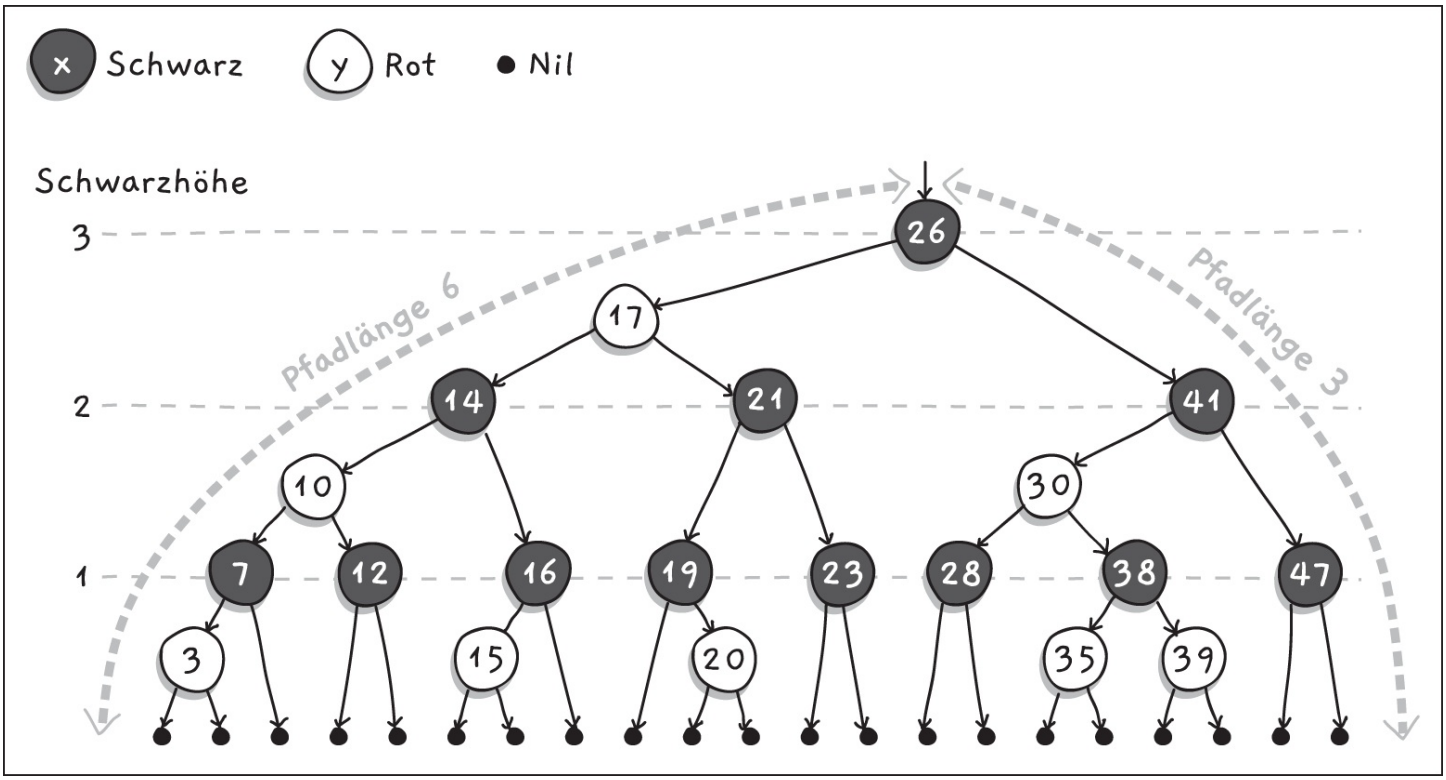


Abbildung 9.22: Ein Rot-Schwarz-Baum

type *Value* = ... irgendein Typ mit vergleichbaren Werten ...
type *Color* = *Red* | *Black*
type RBTree = Nil | Node (*Color*, **RBTree**, *Value*, **RBTree**)

Jeder Rot-Schwarz-Baum endet also in leeren *Nil* -Blättern. Innere Knoten haben eine Farbe, einen Wert und zwei Unterbäume. Der Wert kann zu einem beliebigen Typ gehören. Die Werte müssen natürlich vergleichbar sein, ansonsten kann ja kein Suchbaum erstellt werden.



Ein *Rot-Schwarz-Baum* ist ein binärer Suchbaum mit roten und schwarzen Knoten: Die Werte im linken Unterbaum eines Knotens sind kleiner und die im rechten Unterbaum sind größer als der Wert im Knoten. Weiterhin gilt:

- ✔ Ein roter Knoten darf keinen roten Knoten als Vorgänger haben. Das heißt, rote Knoten haben stets zwei schwarze Kinder. Schwarze Knoten dürfen rote und schwarze Nachfolger haben.
- ✔ Jeder Pfad von einem Knoten zu einem Blatt enthält die gleiche Anzahl an schwarzen Knoten.
- ✔ Die Wurzel ist schwarz.
- ✔ Die Blätter sind leer und gelten als schwarze Knoten.

Ein Rot-Schwarz-Baum hat eine eingeschränkte Forderung an Ausgeglichenheit: Nur die

sogenannte *Schwarzhöhe* muss ausbalanciert werden: Die Schwarzhöhe ist die Zahl der schwarzen Knoten, die auf dem Weg zu einem Blatt passiert werden. Etwas genauer:



Die *Schwarzhöhe* eines Knotens v ist die Anzahl der schwarzen Knoten auf jedem möglichen Pfad von v zu einem Blatt. (Jeder dieser Pfade muss in einem Rot-Schwarz-Baum die gleiche Anzahl an schwarzen Knoten enthalten.) Das Blatt wird dabei mitgezählt, v selbst jedoch nicht.

Was sollen diese Forderungen? Nun, sie formulieren zwar etwas seltsame Forderungen, aber deren Einhaltung ist vergleichsweise wenig aufwendig und garantiert eine gewisse Balanciertheit. Eine tieferen Sinn in Rot und Schwarz gibt es nicht und sollte auch nicht gesucht werden.

Die Rot-Schwarz-Bedingungen garantieren, dass die Höhe $h(T)$ eines Baums T beschränkt ist auf $2 \cdot \log_2(n+1)$. Wobei n die Zahl der Knoten ist. Das bedeutet, dass ein Rot-Schwarz-Baum schlimmstenfalls doppelt so hoch ist wie ein ideal gefüllter binärer Suchbaum.

Der Knoten mit der größten Schwarzhöhe ist die Wurzel. Alle Pfade von der Wurzel zu irgendeinem Blatt enthalten die gleiche Anzahl s an schwarzen Knoten. s ist die Schwarzhöhe der Wurzel. Der kürzeste mögliche Pfad nach unten passiert s schwarze Knoten. Auf dem längsten möglichen Pfad von der Wurzel nach unten wechseln sich s schwarze Knoten mit roten Knoten ab. Der längste Pfad komplett durch den Baum ist also maximal doppelt so lang wie der kürzeste Pfad quer durch den Baum.



Mit den Rot-Schwarz-Bedingungen werden algorithmisch effizient umsetzbare Kriterien definiert, deren Einhaltung dafür sorgen, dass der längste und der kürzeste Pfad quer durch einen Baum sich nicht um mehr als den Faktor 2 unterscheiden. Damit kann der Baum mit vertretbarem Aufwand einigermaßen in Form gehalten werden.

Rot und Schwarz sind nichts anderes als willkürlich gewählte Namen für zwei Arten von Markierungen. Rot-Schwarz-Bäume werden häufig für die interne Speicherung großer Datenmengen verwendet und sind die Basis der Mengen-/Abbildungstypen etlicher Programmiersprachen. Sie sind eine Weiterentwicklung der sogenannten (2,4)-Bäume, die bei der Implementierung von Datenbanken zum Einsatz kommen.



(2,4)-Bäume

Rot-Schwarz-Bäume sind eng mit den sogenannten *(2,4)-Bäumen* verwandt. (2,4)-Bäume (auch 2-3-4-Bäume genannt) sind immer perfekt balanciert, das heißt, alle Pfade von der Wurzel zu einem Blatt sind gleich lang. Dafür können innere Knoten aber zwei, drei oder vier Nachfolger haben. Knoten mit *k* Nachfolgern enthalten *k – 1* Werte, die jeweils die Grenzen für die Werte in den entsprechenden

Unterbäumen angeben (siehe Abbildung 9.23 oben). Die Unterbäume eines Knotens enthalten stets nur Werte, die zwischen zwei Werten im Knoten liegen oder kleiner als der erste oder größter als der letzte Wert sind. (2,4)-Bäume und Rot-Schwarz-Bäume können leicht ineinander verwandelt werden. Fasst man bei einem Rot-Schwarz-Baum jeden schwarzen Knoten mit den ihm eventuell direkt nachfolgenden roten Knoten zusammen, so erhält man einen Knoten im entsprechenden (2,4)-Baum. Abbildung 9.23 unten zeigt beispielsweise die (2,4)-Baum-Variante des Rot-Schwarz-Baumes aus Abbildung 9.22.

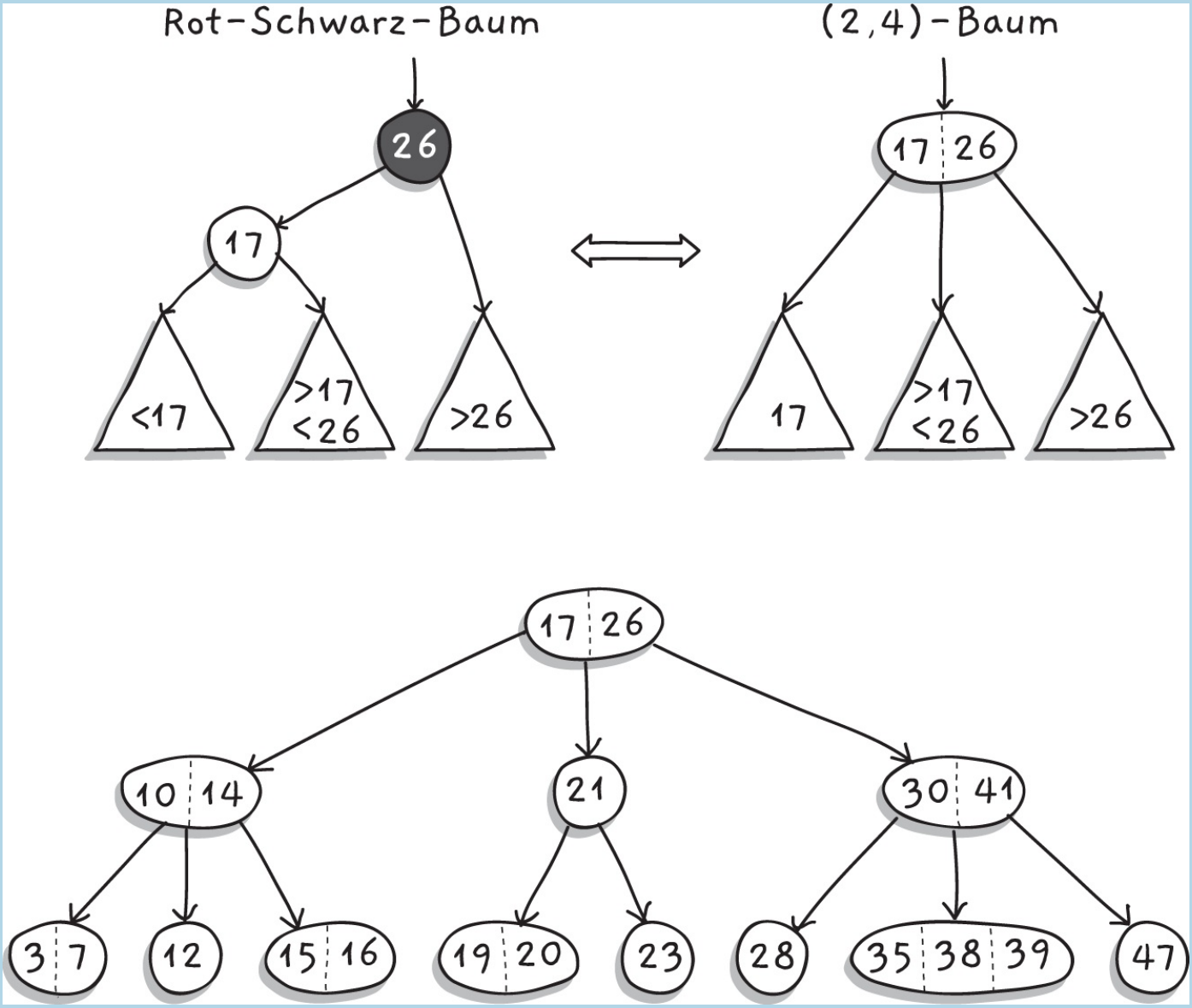


Abbildung 9.23: Oben: Ein Rot-Schwarz-Baum und der entsprechende (2,4)-Baum; unten: Der (2,4)-Baum zum Rot-Schwarz-Baum aus Abbildung 9.22

Suchen und Einfügen

Eine Suche in einem Rot-Schwarz-Baum verändert diesen nicht. Er kommt nicht aus der Balance und nach einer Suchoperation gibt es darum nichts zu balancieren. Ein Suchalgorithmus, der für jeden binären Suchbaum geeignet ist, kann in einem Rot-Schwarz-Baum verwendet werden. Beispielsweise:

```
Member(x, rbTree)
    case rbTree = Nil
        return false
    case rbTree = Node (c, left, v, right) and x = v
        return true
    case rbTree = Node (c, left, v, right) and x < v
        return Member(x, left)
    case rbTree = Node (c, left, v, right) and x > v
        return Member(x, right)
```

Das Einfügen eines neuen Wertes ist schon etwas kniffeliger. Wir müssen darauf achten, dass die Rot-Schwarz-Eigenschaften erhalten bleiben.

Sehen wir uns die Rot-Schwarz-Bäume einmal etwas genauer an. Bei einer Balancierung wird ein Baum in Form gebracht. In Abbildung 9.24 haben wir einen Baum, der nicht in Rot-Schwarz-Form ist: Der rote Knoten 25 hat einen roten Knoten als Nachfolger und die Schwarzhöhe der Pfade von einem Knoten aus ist auch nicht gleich: Von der Wurzel über 25 und 12 ist die schwarze Pfadlänge 2 und über 75 und 60 ist sie 3.

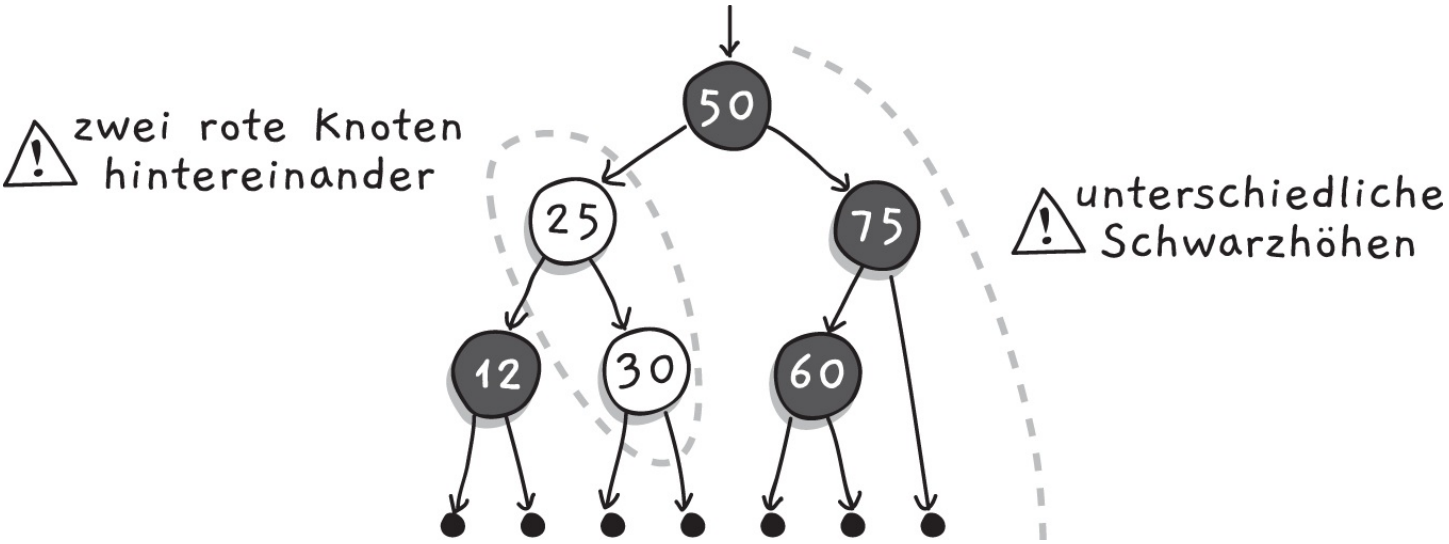


Abbildung 9.24: Rot und Schwarz, aber nicht Rot-Schwarz

Ein solcher Baum könnte der Ausgangspunkt einer Balancierung sein, die nach einer Einfüge-Operation notwendig wird. Bevor wir die Einfüge-Operation in Rot-Schwarz-Bäumen angehen, rufen wir uns zunächst noch einmal die Einfüge-Operation für binäre Suchbäume ohne Balancierung in Erinnerung:

```
Insert(x, t)
    case t = Nil
        return Node(Nil, x, Nil)
    case t = Node(_, v, _) and v = x      x ist schon enthalten, wird ignoriert
        return t
    case Node(l, v, r) and x < v          x ist kleiner als die Wurzel
        return Node(Insert(x, l),         füge x in den linken Unterbaum ein
                    v,
                    r)
    case Node(l, v, r) and x > v          key ist größer als die Wurzel
        return Node(l,
                    v,
                    Insert(x, r))         füge x in den rechten Unterbaum ein
```

Bei Rot-Schwarz-Bäumen müssen die Farben und die Nil-Blätter berücksichtigt gezogen werden. Insbesondere muss aber nach dem Einfügen eines neuen Wertes die Balance geprüft und eventuell wiederhergestellt werden.

Delegieren wir das Balancieren erst einmal an eine Hilfsfunktion, dann kommen wir zu folgendem Algorithmus:

```
InsertRB(x, t)
    return paintBlack(InsR(x, t))

paintBlack(t)                             Schwärze die Wurzel
    case t = Node(c, left, v, right)
        return Node(Black, left, v, right)

InsR(x, s)                                füge rekursiv ein, dann balanciere
    case s = Nil                          erzeuge neuen roten Knoten
        return Node(Red, Nil, x, Nil)

    case s = Node(c, left, v, right) and x < v      füge links ein, dann balanciere
        return Balance(c, InsR(x, left), v, right)

    case s = Node(c, left, v, right) and x = v      schon drin: nichts zu tun
        return s

    case s = Node(c, left, v, right) and x > v      füge rechts ein, dann balanciere
        return Balance(c, left, v, InsR(x, right))
```

Das ist im Wesentlichen das gleiche wie bei unbalancierten binären Suchbäumen. Die Unterschiede sind:

- ✔ Neue Knoten werden rot gefärbt.
- ✔ Die Wurzel wird am Schluss schwarz gefärbt.
- ✔ Mit *Balance* wird der Baum ausbalanciert.

Neue Knoten werden als rote Knoten angelegt. Sie ändern die Schwarzhöhe darum erst einmal nicht. Allerdings kann es passieren, dass jetzt, unerlaubterweise, zwei rote Knoten untereinander hängen. In Abbildung 9.25 hängt beispielsweise eine rote 30 unter einer roten 25.

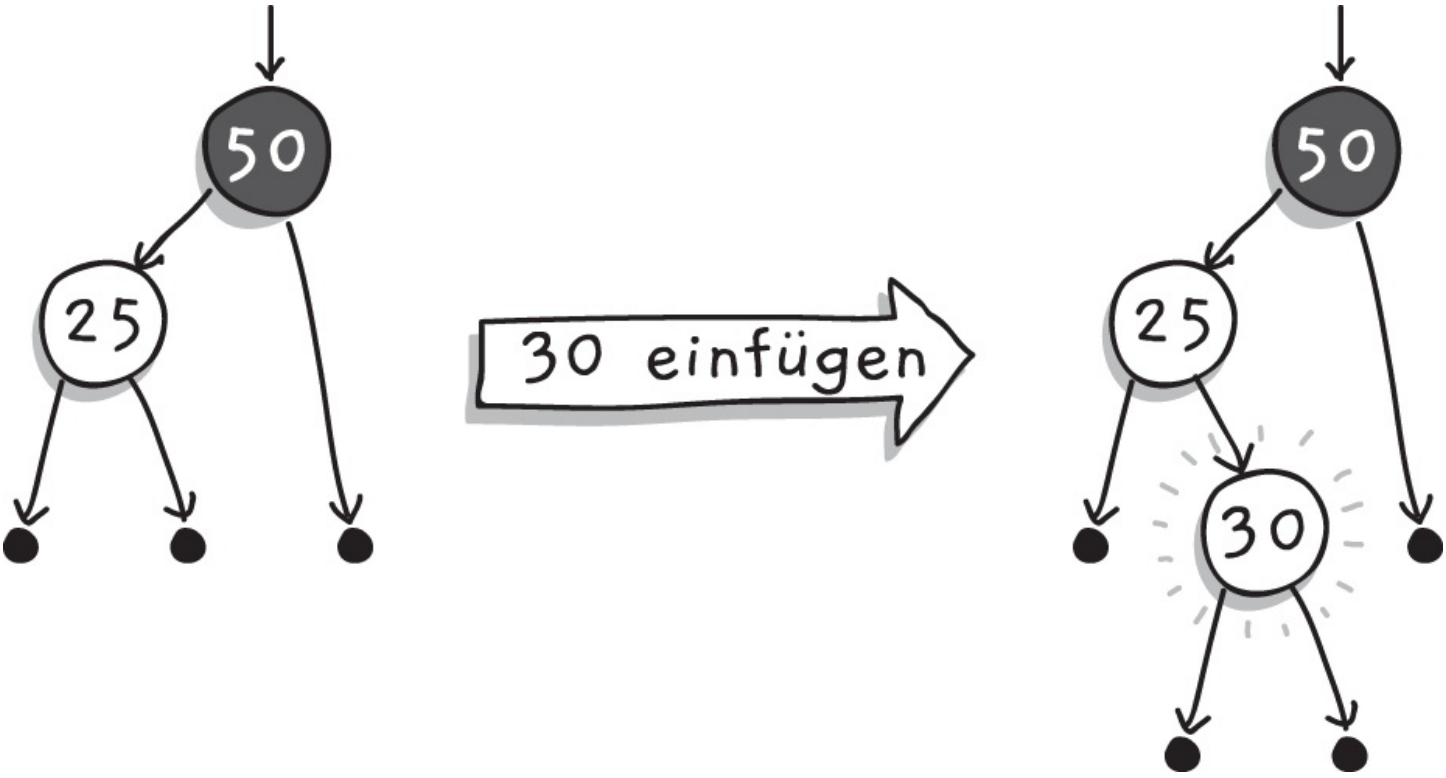


Abbildung 9.25: Ein Rot-Schwarz-Baum gerät aus dem Gleichgewicht.

Mit Rotationen und Umfärben kann das behoben werden (siehe die Abbildungen 9.26 und 9.27).

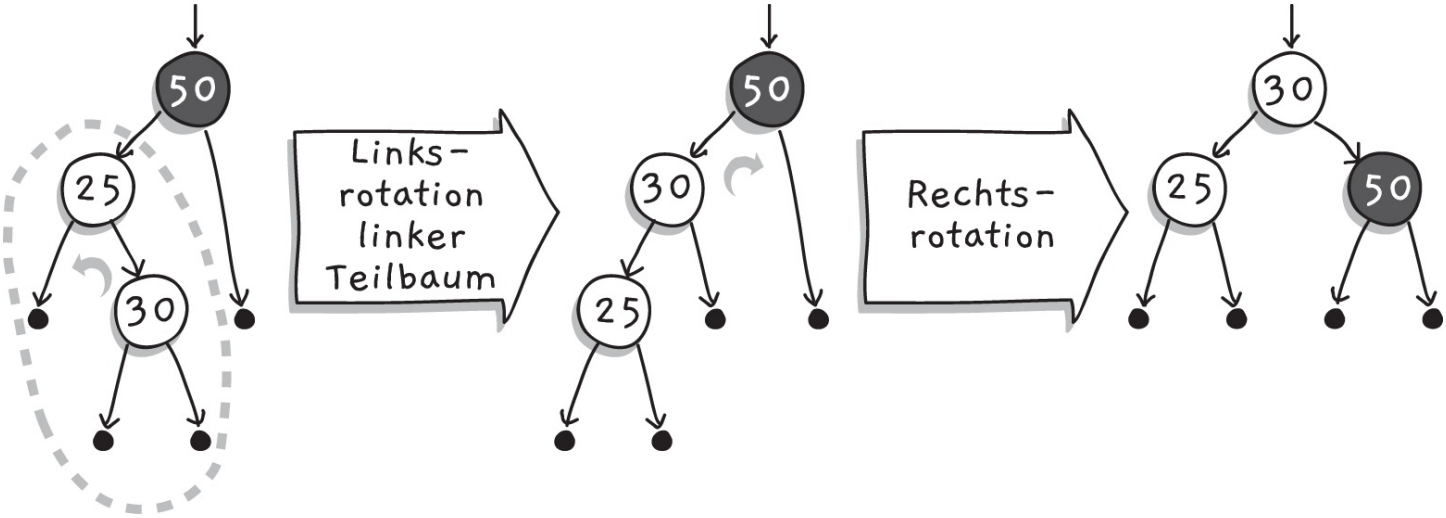


Abbildung 9.26: Zwei Rotationen …



Abbildung 9.27: … und einmal Umfärben bringen ihn wieder ins Gleichgewicht.

Generell kann der Baum mit Rotationen und Umfärben der Knoten wieder in die gewünschte Form gebracht werden. Dabei wird ein Teilbaum mit einer schwarzer Wurzel mit rotem Kind und rotem Enkelkind zu einem Teilbaum mit schwarzer Wurzel und zwei roten Kindern transformiert. Dazu wird wie im Beispiel von Abbildung 9.26 und 9.27 rotiert und umgefärbt.

Im Beispiel sind die roten Knoten linkes Kind und links-rechtes Enkelkind eines schwarzen Knotens. Im Allgemeinen gibt es vier Fälle für eine rot-rote Kind-Enkel-Konfiguration (Abbildung 9.28):

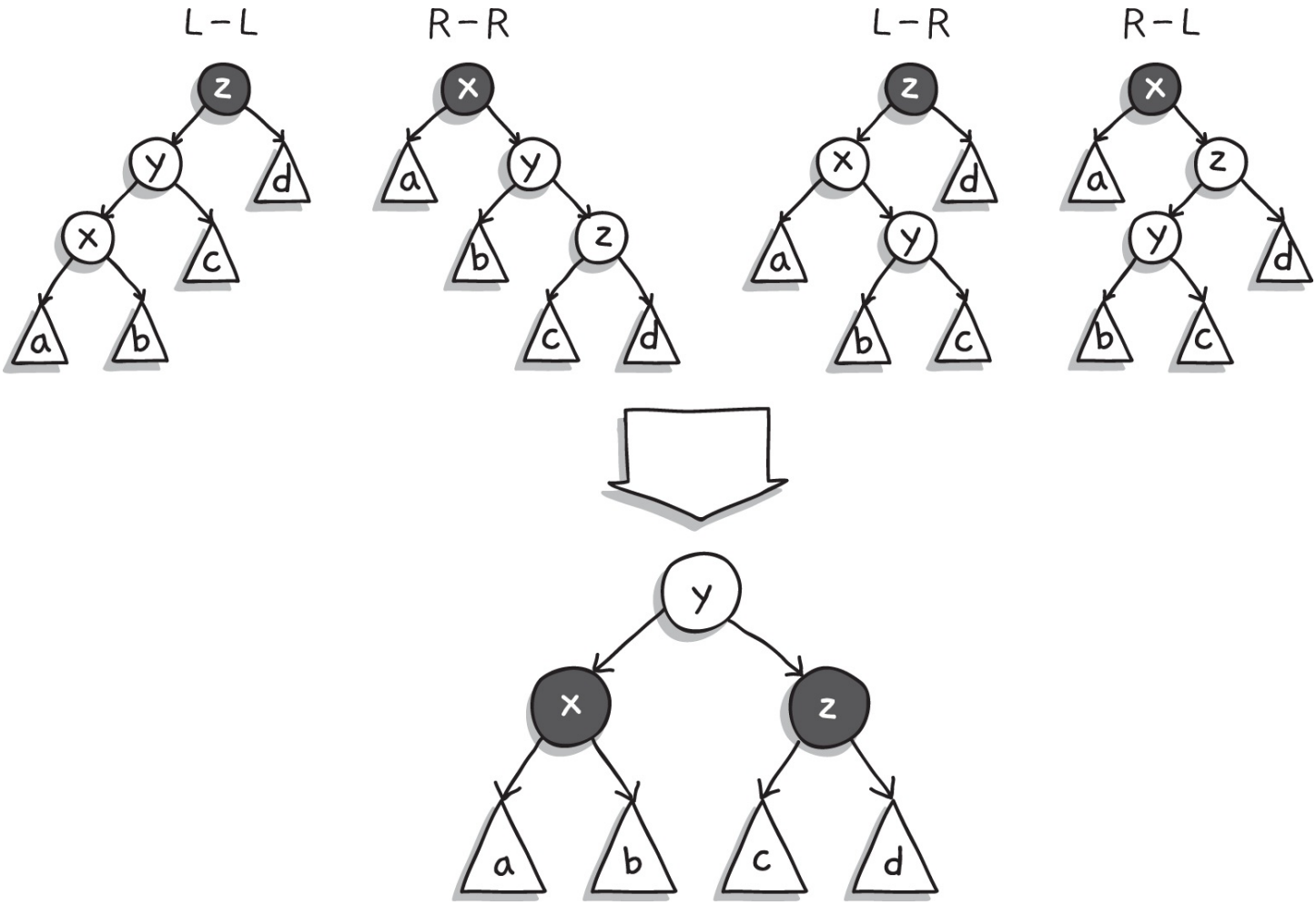


Abbildung 9.28: Transformationen von Rot-Rot-Konstellationen (!! unten Farben vertauschen bei x=>Rot,y=>Schwarz,z=>Rot)

- ✔ L-L: Linkes Kind, linkes Enkelkind rot
- ✔ L-R: Linkes Kind, rechtes Enkelkind rot
- ✔ R-L: Rechtes Kind linkes Enkelkind rot
- ✔ R-R: Rechtes Kind rechtes Enkelkind rot

Von den Unterbäumen unterhalb dieser Konstellationen (a, b, c, d in Abbildung 9.28) nehmen wir an, dass sie schwarze Wurzeln haben. Das ist legitim, denn wir haben unsere Umformungen durch die Funktion balance in Algorithmus InsertRB ja so organisiert, das sie rekursiv Bottom-up ausgeführt werden. Diese vier Konstellationen werden alle in die gleiche Zielstruktur umgeformt.

Der entsprechende Algorithmus besteht im Wesentlichen aus dem Erkennen der Muster L-L, L-R, R-L und R-R. In Programmiersprachen, die Case-Notation/Pattern-Match nicht unterstützen, wird das eventuell etwas unübersichtlich, sollte aber ansonsten keine größeren Probleme verursachen.

Es darf allerdings nicht verschwiegen werden, dass dieser Code funktional ist. Eine imperative Variante wird schnell deutlich komplexer. Damit wird auch eine etwas bessere

Effizienz erreicht. Bei Umfärbungen wird in der funktionalen Version des Algorithmus ein neuer Knoten angelegt. In einer imperativen Version würde man einfach das Farb-Bit eines Knotens verändern.

```
Balance(color, tree1, value, tree2)

    case(color, tree1, value, tree2) =   L-L
        (Black, Node(Red, Node(Red, a, x, b), y, c), z, d)
            return Node(Red, Node(Black, a, x, b), y, Node(Black, c, z, d))

    case(color, tree1, value, tree2) =   R-R
        (Black a, x, Node(Red, b, y, Node(Red, c, z, d)))
            return Node(Red, Node(Black, a, x, b), y, Node(Black, c, z, d))

    case(color, tree1, value, tree2) =   L-R
        (Black, Node(Red, a, x, Node(Red, b, y, c) ), z, d)
            return Node(Red, Node(Black, a, x, b), y, Node(Black, c, z, d))

    case(color, tree1, value, tree2) =   R-L
        (Black, a, x, Node(Red, Node(Red, b, y, c), z, d))
            return Node(Red, Node(Black, a, x, b), y, Node(Black, c, z, d))

    case(color, tree1, value, tree2) =   irgendetwas anderes
        (_, _, _, _)                     keine Balancierung notwendig
            return Node(color, tree1, value, tree2)
```



Rot-Schwarz-Bäume: Imperativ und Funktional

Basierend auf Th. Corman u. a. *Algorithms* werden Algorithmen auf Rot-Schwarz-Bäumen meist in imperativer Form behandelt. Der hier präsentierte Algorithmus basiert auf Chris Okasaki *Functional Pearls, Red-Black-Trees in a Functional Setting*, Journal of Functional Programming 9(4) 471-477. Diese funktionale Version des Algorithmus ist um den Preis einer etwas geringeren Effizienz deutlich klarer. Rot-Schwarz-Bäume sind immer noch eine hoch interessante Datenstruktur mit sehr guter Effizienz, aber leider recht komplizierten Implementierungen.

Kapitel 10

Verbindungen finden

IN DIESEM KAPITEL

- Dijkstras Algorithmus zur Bestimmung der kürzesten Pfade von einem zu allen anderen
- Identifikation verbundener Komponenten mit der Datenstruktur *Disjoint Set*
- Kruskals Algorithmus zur Berechnung des minimalen aufspannenden Baums

Im Leben stellt sich oft die Frage, ob es Verbindungen zwischen zwei Dingen gibt und wenn ja, welche die beste ist. Die Dinge können völlig unterschiedlicher Natur sein und die Verbindungen zwischen ihnen können sich in den unterschiedlichster Arten zeigen. Wenn aber Verbindungen im Spiel sind, dann sind Graphen nicht weit. Sie sind ja das universelle Modell von Dingen, die wie auch immer in Beziehung zueinander stehen. Reden wir also über Knoten und Kanten.

Bei der Suche nach der Verbindung für's Leben geht es darum, benachbarte Knoten zu finden und deren emotionale Distanz zuverlässig zu bestimmen. Bei der Suche nach einer geeigneten Bahn- oder Straßenverbindung sind die direkten Nachbarschaften das eher geringere Problem, die Vielfalt der Reisemöglichkeiten ist das, was die Sache schwierig macht.

In diesem Kapitel betrachten wir zunächst ein klassisches Problem der Wegfindung: Wie findet man in einem Graphen die kürzeste Verbindung von Knoten? Dieses Problem hat vielfältige Anwendungen, nicht nur die, bei denen man heutzutage sein Navi konsultiert.

Das zweite Problem, das wir in diesem Kapitel betrachten wollen, sieht auf den ersten Blick vielleicht etwas weniger alltäglich aus, hat aber ebenfalls eine intuitiv verständliche Grundlage und stellt sich auch in vielfältigen realen Situationen. Es ist das Problem des minimalen Spannbaums, bei dem es darum geht, einen Graphen auf sein essenzielles Skelett zu reduzieren. Der interessante Aspekt hierbei ist die Wahl einer geschickten Datenstruktur, mit der ein einfacher, aber zunächst sehr langsamer Algorithmus effizient und schnell wird.

Beginnen wir mit der Wegfindung.

Kürzeste Pfade

Nehmen wir an, es sei ein Problem zu lösen, das als Suche nach den kürzesten Pfaden in einem Graphen modelliert werden kann. Der Graph hat Knoten und Kanten. Die Natur der Knoten spielt für den Algorithmus keine Rolle. Es handelt sich um eine nicht weiter spezifizierte Menge.

Die Kanten können gerichtet oder ungerichtet sein. Es gibt Algorithmen, die voraussetzen, dass jede Kante in beiden Richtungen durchlaufen werden kann. Das ist hier nicht der Fall. Es ist egal, ob die Kanten gerichtet sind oder nicht. Wir nehmen die Kanten der Einfachheit halber als gerichtet an. Sie können darum durch ein Knotenpaar modelliert werden. Ungerichtete Kanten könnten einfach als zwei Kanten, nämlich eine Hin- und eine Her-Kante modelliert werden.

Jeder Kante ist eine Länge zugeordnet. Das ist eine Markierung in Form eines positiven numerischen Wertes. Wir nehmen an, es handelt sich um natürliche Zahlen. Elemente jeder anderen total geordneten Menge sind aber auch als *Länge* geeignet. Statt *Länge* wird oft auch der Begriff (Kanten-)*Gewicht* verwendet. Wir gehen auch vom allgemeinen Fall eines zyklischen Graphen aus. Bei azyklischen Graphen gibt es einfachere Algorithmen.

 In Kapitel 6 werden die wichtigsten Begriffe der Graphentheorie kurz erläutert.

Alle kürzesten Pfade von einem Start aus

Zu einem gegebenen Startknoten s soll der kürzeste Pfad zu jedem Knoten v in einem Graphen berechnet werden. Manchmal beschränkt man sich auch darauf, nur die Länge des kürzesten Pfades in Erfahrung zu bringen.

Ein einfaches Beispiel ist der Graph in Abbildung 10.1 mit Startknoten a und folgender Lösung (»SP« steht für *shortest path*, also kürzester Weg):

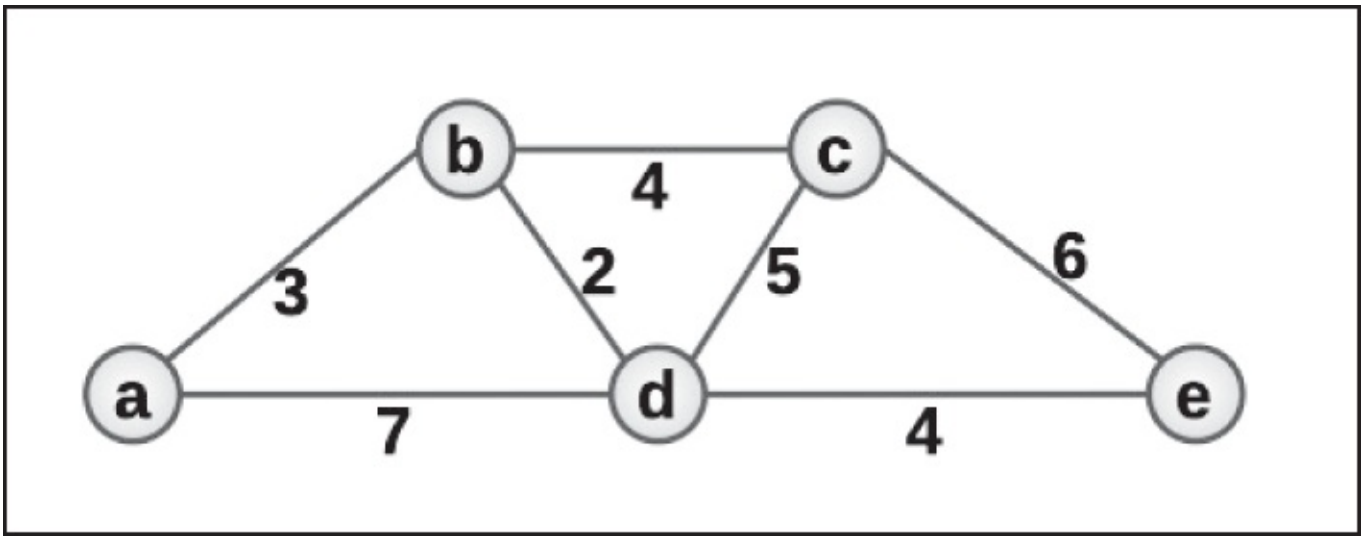


Abbildung 10.1: Ein ungerichteter, gewichteter Graph

$$SP = [a \to 0, b \to 3, c \to 7, d \to 5, e \to 9]$$

Etwas exakter haben wir die Problemstellung:

> **Problem:** Kürzester Pfad zu allen Knoten von einem Startknoten aus
>
> **Eingabe:** Ein Graph $G(V, E, w)$, wobei w eine Abbildung $E \rightarrow \mathbb{N}_0$ ist, sowie ein Startknoten $s \in V$
>
> **Berechne:** Die Länge des kürzesten Pfades $SP(v)$ von s zu jedem Knoten $v \in V$:
>
> $SP(v) = Min\{len(e) \mid e \text{ ist ein Pfad von } s \text{ nach } v\}$
>
> Die Länge ist dabei die Summe der Kantenmarkierungen $w(e)$ für jede Kante im Pfad.

Interessiert nur der kürzeste Pfad zu einem ganz bestimmten Knoten, dann wird der Algorithmus zur Berechnung des Pfades zu allen Knoten einfach abgebrochen, wenn der kürzeste Pfad zu diesem bestimmten Ziel gefunden wurde. Es gibt keinen spezialisierten, auf ein Ziel ausgerichteten Algorithmus, der besser ist als dieser, der für alle Ziele den kürzesten Pfad berechnet.

Für eine Lösungsidee betrachten wir zunächst einmal zwei Extremfälle:

- ✔ Wenn der Graph nur aus dem Startknoten s beseht, dann ist die Sache einfach. s hat die Entfernung 0 zu sich selbst. Damit ist die Sache erledigt (siehe Abbildung 10.2 mit $s = a$).

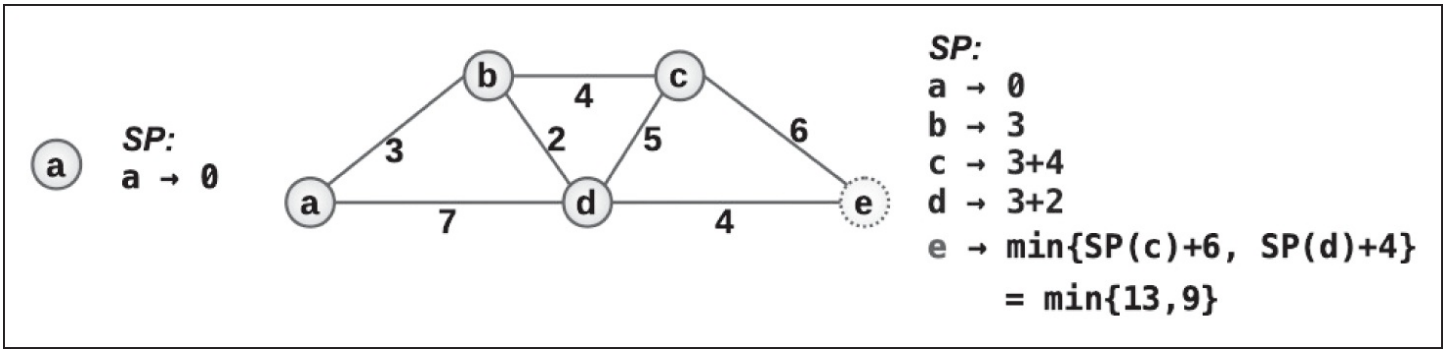


Abbildung 10.2: Kürzeste Wege: Extremfälle

- ✔ Der andere Extremfall ist der, wenn alles getan ist und nur noch ein Knoten k übrig bleibt, für den wir den kürzesten Weg von s aus bestimmen müssen. Wir betrachten dann einfach alle Nachbarn n von k und nehmen das Minimum von $SP(n) + w(n, k)$, also das Minimum von deren Entfernung von s plus der Länge der direkten Verbindung (siehe Abbildung 10.2 mit $k = e$ und Nachbarn $n \in \{c, d\}$).

Damit haben wir einen Anfang und ein Ende der Berechnung. Fehlt nur noch der Weg

dazwischen. Beim ersten und beim letzten Schritt haben wir keine Wahl. Es muss mit dem Startknoten losgehen. Fehlt nur noch ein Knoten, für den der kürzeste Pfad zu berechnen ist, dann ist genau dieser dran. Im Allgemeinen ist das natürlich nicht der Fall. Gehen wir noch einen Schritt weiter zurück und nehmen an, es fehlten noch zwei Knoten. Beispielsweise c und e (siehe Abbildung 10.3).

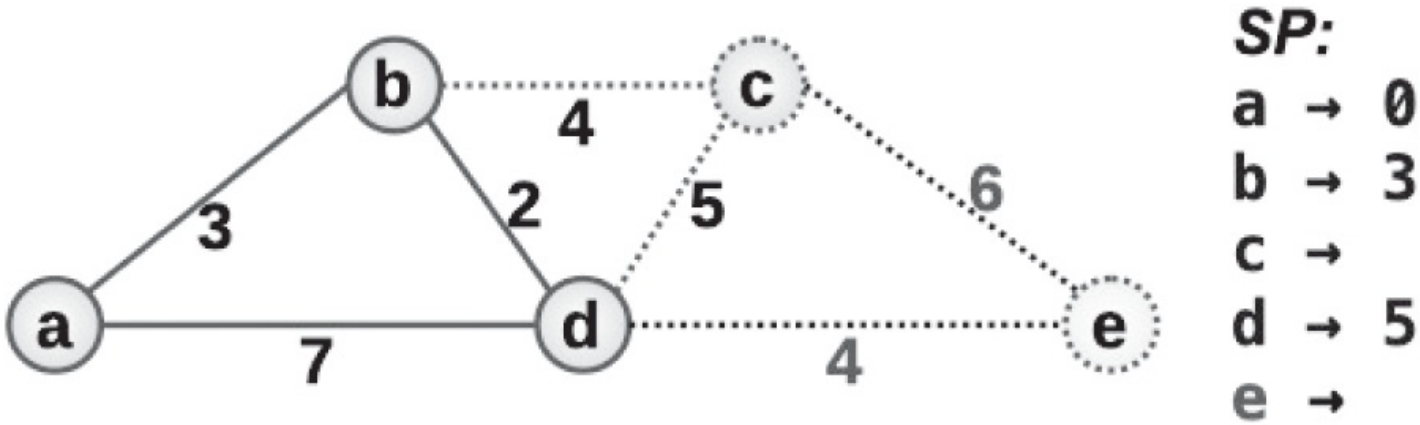


Abbildung 10.3: Kürzeste Wege: Erweiterung einer Teillösung

Zeit, einmal kurz durchzuatmen.



In Teil IV werden algorithmische Techniken betrachtet. Im Kern geht es dabei darum, wie man zu einer Idee für einen Algorithmus kommt. Die erste Technik ist *Erschöpfende Suche*: Probiere einfach alles aus. Alles hieße in unserem Fall alle Pfade, die bei s starten und in einem Knoten enden. Pfade sind eine Teilmenge der Permutationen aller Knoten und Permutationen gibt es ziemlich viele. So etwas macht man nur, wenn man sonst überhaupt keine Idee hat.

Die zweite Technik ist *Teilen und Herrschen*: Versuche, das Problem auf kleinere Probleme der gleichen Art zu reduzieren. Die dritte Technik, *dynamisches Programmieren*, ist eine Variante der zweiten.

Die vierte Technik *Gier* ist etwas, zum dem uns entweder Denkfaulheit oder Verzweiflung treibt – etwa nach zu viel erfolglosem Nachdenken über eine Lösung nach Technik zwei und drei oder weil die CPU bei der Lösungssuche nach Technik eins durchgeschmort ist.

Bis hierher haben wir eine Lösung nach dem Prinzip *Teilen und Herrschen* gesucht. Genauer gesagt *Reduzieren und Herrschen*, denn das Problem wird auf nur ein kleineres, nicht mehrere kleinere, reduziert. Dabei haben wir von zwei Enden her überlegt:

- ✔ Können wir, bei Start mit einem Knoten, die Lösung systematisch durch Hinzunahme weiterer Knoten größer machen, die Lösung also *Bottom-up* durch Hinzunahme weiterer Knoten konstruieren? Oder aber:

- ✔ Können wir die Lösung für n Knoten aus der Lösung für $n-1$ Knoten konstruieren, die Lösung also rekursiv *Top-down* zusammenbauen?

Beide sind die zwei Seiten derselben Medaille: Die Suche nach einer intelligenten systematischen Lösung des Problems.

Gut, weiter geht's. Thema: Wie kann eine Teillösung systematisch vergrößert werden?

Angenommen, es fehlen noch zwei Knoten bis zum vollständigen Graphen, etwa c und e. Für die anderen Knoten v sei das Problem gelöst und $SP(v)$ berechnet.

Jetzt haben wir die Wahl: wir können c oder e hinzunehmen. Nehmen wir zur Abwechslung mal e zuerst und wenden wieder das Verfahren von oben an. Wir berechnen also das Minimum über alle bekannten Pfade, die über eine Direktverbindung zu unserem neuen Knoten e verlängert werden können. In diesen Fall gibt es nur eine Direktverbindung zu Knoten e, deren SP bekannt ist, nämlich die von Knoten d. Diese nehmen wir hinzu und berechnen $SP(e) = 9$ (siehe Abbildung 10.4).

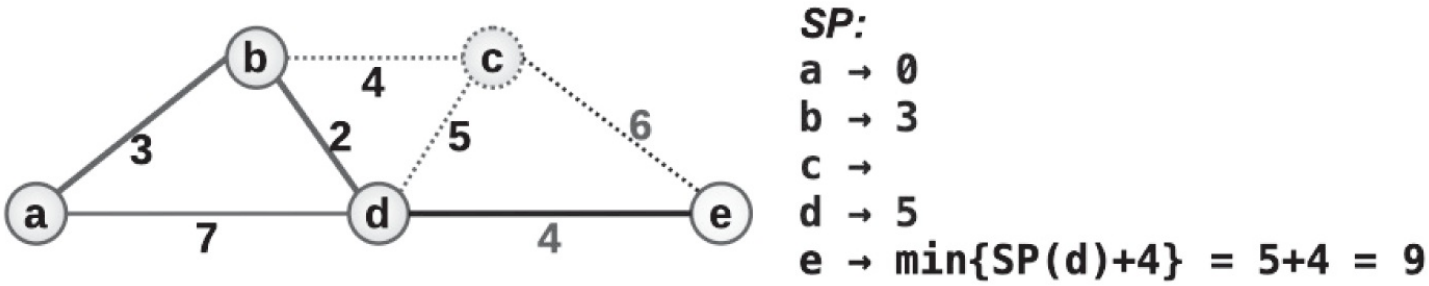


Abbildung 10.4: Kürzeste Weg: Alternativen bei der Erweiterung des Bekannten

Alles bestens. Jetzt kommt noch c mit $SP(c) = 7$ hinzu, was als das Minimum der Länge aller Pfade über seine Direktverbindungen berechnet wird:

$$SP(c) = \\ min\{SP(b)+4, SP(d)+5, SP(e)+6\} = min\{3+4, 5+5, 9+6\} = 7$$

Prima. Scheint bestens zu klappen. Wir können mit dem Startknoten starten und dann einen Knoten nach dem anderen hinzunehmen. Frisch ans Werk. – Hmm, funktioniert das wirklich so? Ist ein erfolgreiches behandeltes Minibeispiel vielleicht doch noch keine Garantie für ein fehlerloses Konzept?

Also nochmal mit etwas mehr Misstrauen. Der Anfang und das Ende sind sicher korrekt. Aber können wir im allgemeinen Fall wirklich irgendeinen der noch unverarbeiteten Knoten einfach so hinzunehmen? So einfach kann das doch nicht sein.

Vom Vertrauten ins Unbekannte

Werfen wir noch einmal einen Blick auf unser kleines Beispiel. Dort war es egal, ob zuerst c oder e zu den Zielen mit gesichertem Pfad hinzugefügt wurde. Liegt das vielleicht doch an besonderen Gegebenheiten im Beispiel?

Präzisieren wir erst einmal unser Verfahren (siehe Abbildung 10.5):

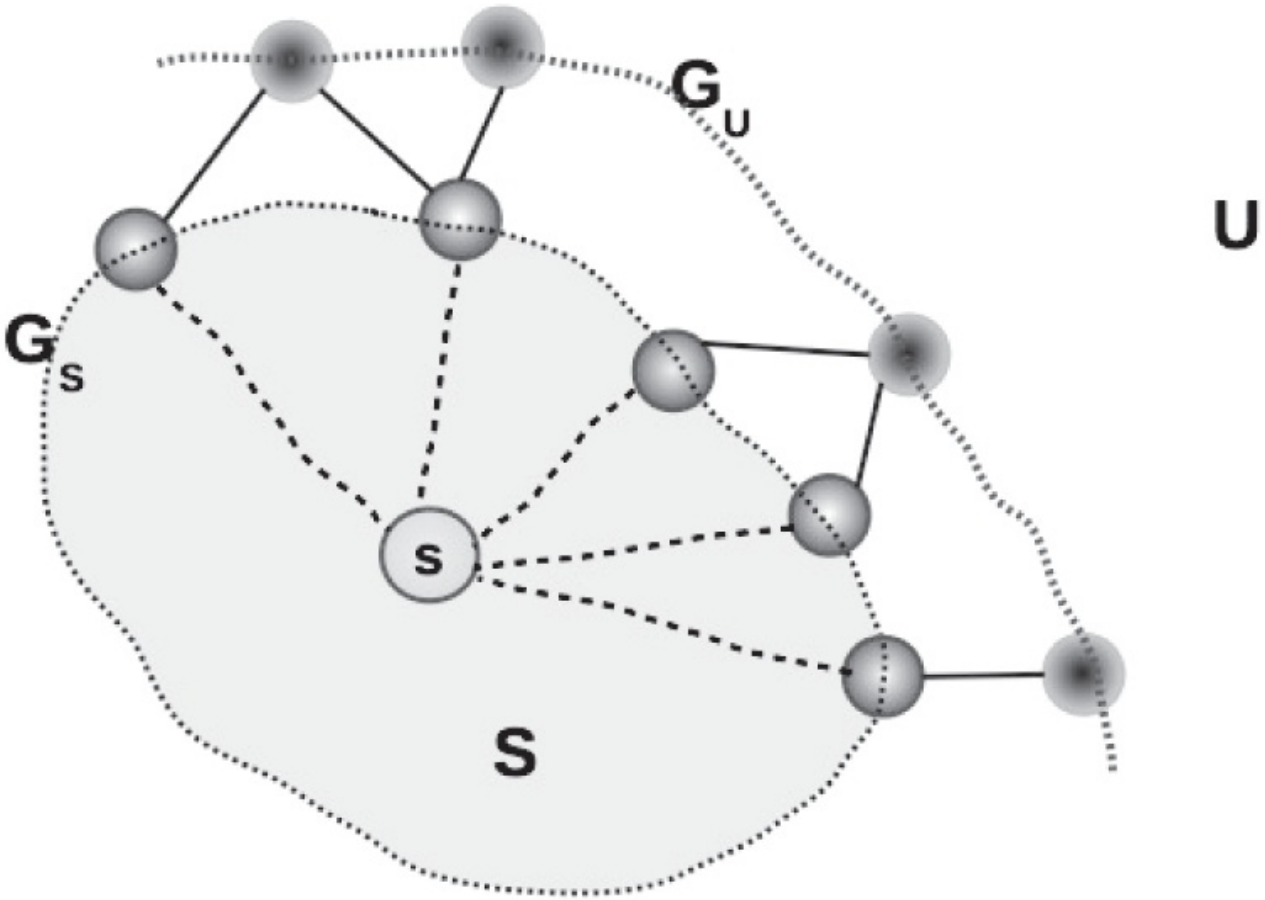


Abbildung 10.5: Kürzeste Weg: Von innen nach außen

- ✔ Teile die Knoten auf in *Inland* S mit bekannter kürzester Entfernung ($t \in S \Rightarrow SP(t)$ bekannt) und den Rest, das *Ausland* U mit unbekanntem $SP(u)$ für $u \in U$.
- ✔ Starte mit $S = \{s\}$ als Menge von Knoten mit bekanntem kürzesten Pfad von s aus. Alle anderen Knoten gehören zum Ausland U .
- ✔ Solange nicht alle verarbeitet sind, nimm schrittweise einen Knoten $u \in U$ zu S hinzu:
  - Bestimme die Menge G_S der *inländischen Grenzstationen*. Das sind die Knoten, die einen direkten Nachbarn haben, der außerhalb von S liegt.
  - Bestimme jetzt die Menge der *ausländischen Grenzstationen* G_U . Das sind die direkten Nachbarn der inländischen Grenzstationen, die im Ausland U , also außerhalb von S liegen.
  - Wähle irgendeine ausländische Grenzstation $u \in U$.
  - Berechne die Entfernung von s zu u über jeden Pfad, der über eine inländische Grenzstation geht.

- Das Minimum dieser Entfernungen ist der kürzeste Pfad von s nach u.
- Mit bekannter kürzester Entfernung von s kann u jetzt zu S hinzugenommen werden und gehört damit auch zum Inland S.

Was kann dabei schiefgehen? Angenommen, wir wählen u als ausländische Grenzstation (siehe Abbildung 10.6).

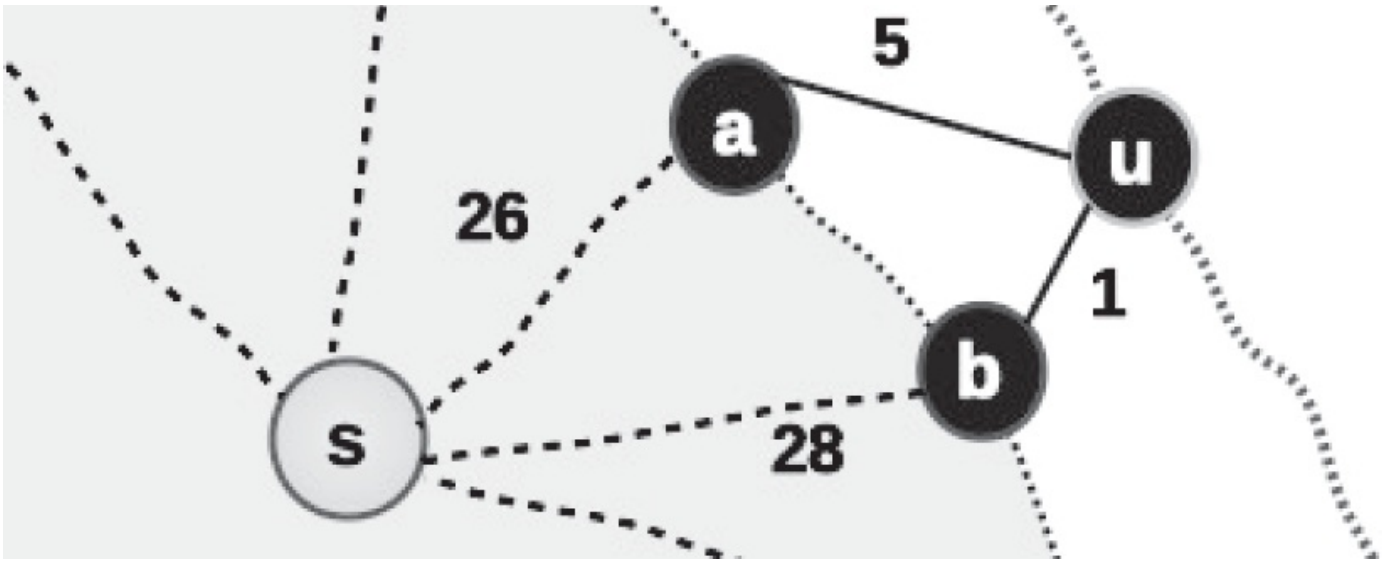


Abbildung 10.6: Lokale Plausibilität des Algorithmus

$SP(u)$ berechnet sich als Minimum von $SP(a) + 5 = 26 + 5$ und $SP(b) + 1 = 28 + 1$. Also ist $SP(u) = 29$. Einen besseren, fast rein inländischen Weg als den über b mit der Länge 29 kann es nicht geben. Jeder andere Weg müsste über eine weitere ausländische Station gehen oder ist länger.

Ups, könnte es vielleicht einen Weg zu u geben, der mit mindestens einer weiteren Station durch das noch unbekannte ausländische Terrain U geht und kürzer als 29 ist? Klar doch. Weiten wir unseren Blick auf die globale Situation (siehe Abbildung 10.7).

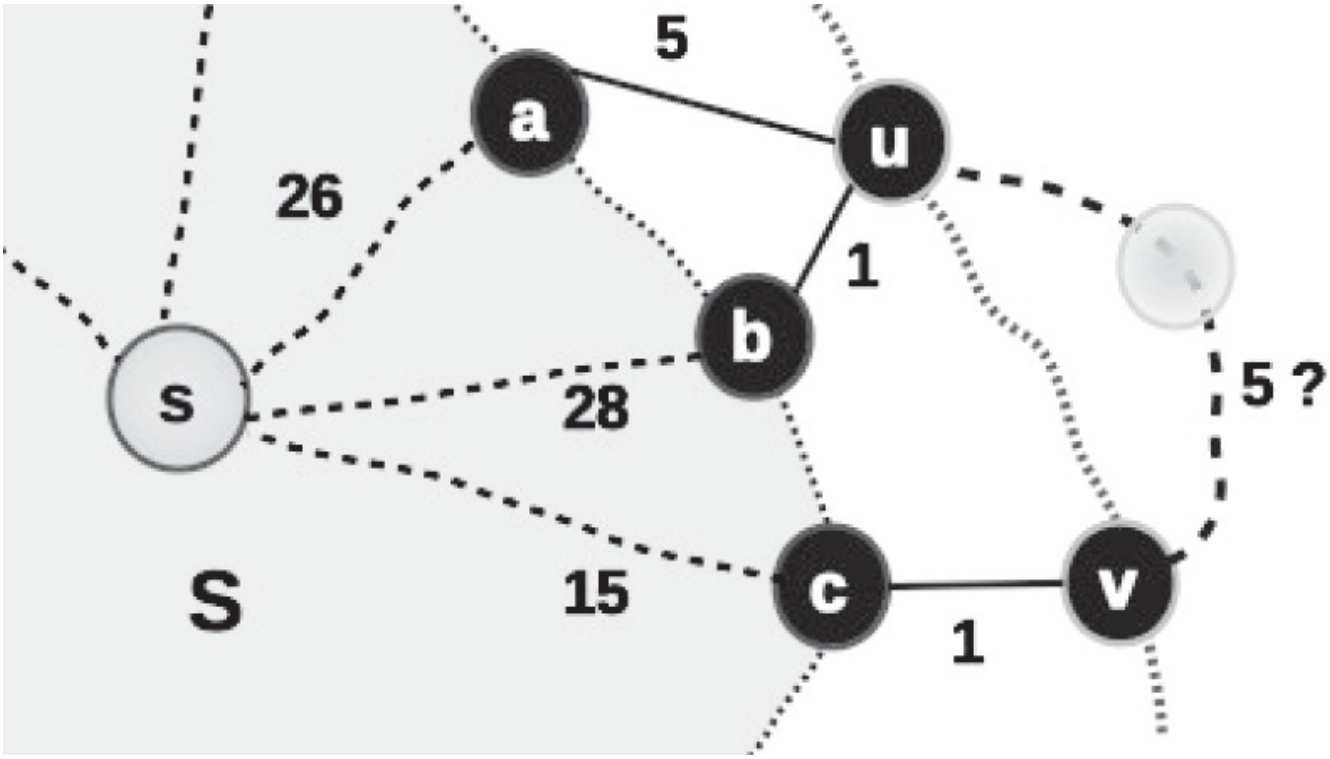


Abbildung 10.7: Globale Plausibilität des Algorithmus

Möglicherweise gibt es also einen Weg, der über andere inländische und ausländische Grenzstationen c und v geht und auf kürzerem Weg nach u führt. c hat keine direkte Verbindung zu u, sonst hätten wir c schon lokal betrachtet. Darum gilt $v \neq u$. Die ausländischen Grenzstationen u und v können aber eine direkte oder indirekte Verbindung haben, die kürzer ist als die über b. Eine Verbindung, die noch unbekannt ist, weil sie durch bisher unerforschtes Gebiet führt.

Bricht unsere Idee damit völlig zusammen? Vielleicht nein, vielleicht haben wir mit u nur den falschen Knoten gewählt. Vielleicht hätten wir vorhin besser v statt u gewählt. Fraglich, ob das etwas genutzt hätte. Es hätte ja das Gleiche passieren können. Auch hier könnte sich wieder ein Weg über einen anderen Knoten durch ausländisches Gebiet als besser herausstellen. Ja, dann fangen wir doch mit diesem Knoten an. Hat das ein Ende? Gibt es eine *beste ausländische Grenzstation*, zu der ein fast rein inländischer Weg führt? Gibt es also einen eindeutig zu bestimmenden Knoten, um den das bekannte Gebiet S erweitert werden kann, ohne Einträge in SP zu machen, die sich später als falsch herausstellen?

Betrachten wir alle ausländischen Grenzstationen $t \in T$ am Rand des Bekannten. Eine von diesen hat den kürzesten, fast rein inländischen Pfad vom Startknoten aus. Also einen Pfad, bei dem nur der letzte Schritt aus dem Bekannten herausführt. Nehmen wir an, es sei v. Ist dieser fast rein inländische Weg auch der absolut beste oder kann nach einer Ausweitung des Inlands S doch noch ein besserer kürzerer Weg zum Vorschein kommen (siehe Abbildung 10.8)?

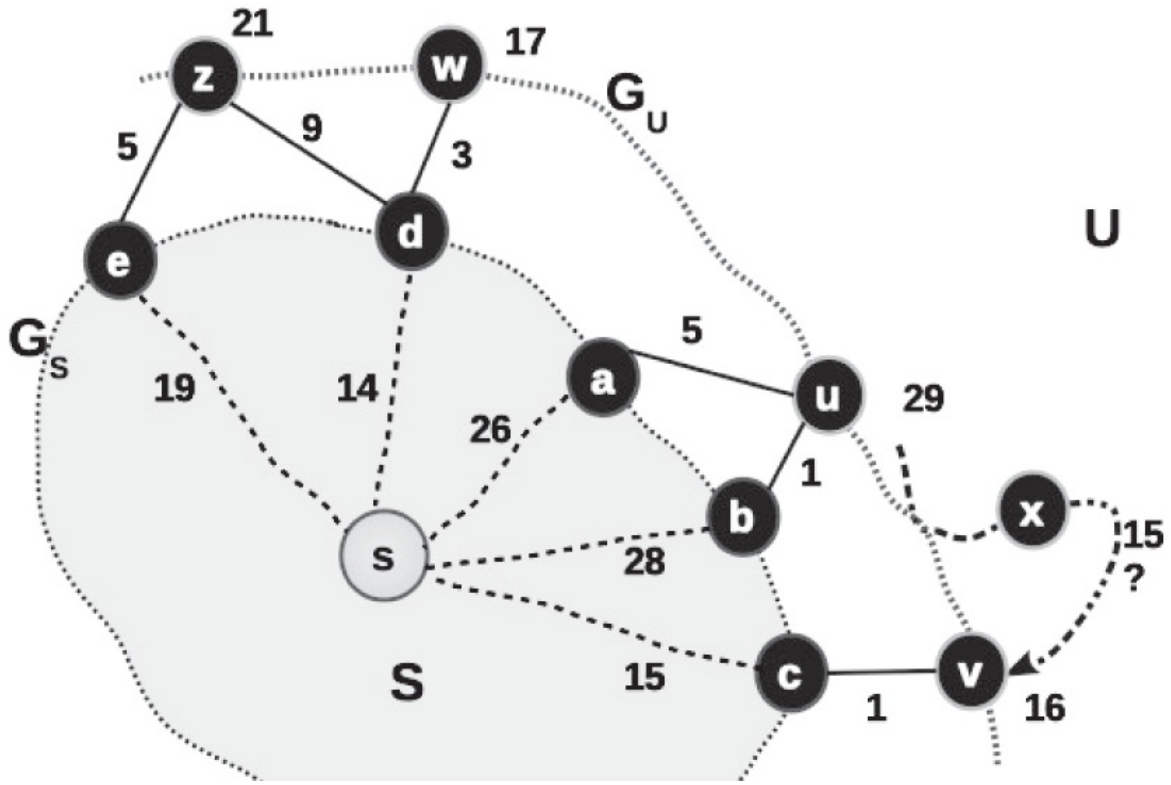


Abbildung 10.8: Gibt es eine geheime Abkürzung zur nächstgelegenen Grenzstation?

Einen kürzeren Weg zu v mit nur einer Strecke im unbekannten Gebiet kann es nicht geben, denn wenn doch, dann hätten wir ihn schon gewählt. Kann es dann einen Knoten x im unbekanntem Gebiet geben, über den es einen kürzeren Weg zu v gibt? Offensichtlich nicht: Wenn es einen gäbe, dann müsste es über uns bekannte Wege zu irgendeiner Grenzstation gehen und dieser Weg wäre mindestens so lang wie einer der Wege zu einer Grenzstation. Von diesen Grenzstationen haben wir aber schon die gewählt, die am nächsten an s liegt. S kann also gefahrlos um die Grenzstation erweitert werden, die am dichtesten von allen an s liegt.

Kürzester Pfad zu allen Knoten

Damit haben wir (endlich) den Algorithmus fertig konzipiert und auch ein ausreichendes Vertrauen in ihn entwickelt. Möglicherweise gibt es noch Optimierungen. Aber wir dokumentieren unsere Erkenntnisse erst einmal in Pseudocode:

ShortestPaths (s, V, E, w)

$S \leftarrow \{s\}$ *Bereich mit bekanntem SP (Inland)*

$U \leftarrow V \setminus \{s\}$ *Bereich mit unbekanntem SP (Ausland)*

$SP \leftarrow [s \rightarrow 0]$ *bekannte Länge kürzester Pfade*

while $U \neq \emptyset$ *transferiere Knoten sukzessive von U nach S*

$BE \leftarrow$ Kanten, die aus S heraus führen

$(u, d) \leftarrow$ am nächsten zu s gelegene Station außerhalb von S, zu der eine Kante in BE führt, zusammen mit ihrer Entfernung d von s

$S \leftarrow S \cup \{u\}$ *u's kürzeste Entfernung von s ist definitiv bekannt*

$U \leftarrow U \setminus \{u\}$ *u kommt zum bekannten Gebiet ...*

$SP \leftarrow SP + [u \rightarrow d]$ *... mit gesichertem SP*

return SP

U ist das unbekannte ausländische Gebiet, S das bekannte inländische. Bekannt bedeutet dabei, dass von jedem Knoten $v \in S$ die Entfernung $SP(s)$ nach s bekannt ist. Solange es noch Stationen in unbekanntem Gebiet U gibt, wähle ein $u \in U$ aus. u soll die beste Station sein von allen in U. Es liegt an der Grenze zu S und ist von denen an der Grenze (und allen anderen) am dichtesten an s. $SP(u)$ kann zweifelsfrei berechnet werden. Also: Berechne $SP(u)$ und verschiebe u von U nach S.

Alle Werte sind problemlos zu berechnen. Für die grenzüberquerenden Kanten ist eine Hilfsfunktion boarderEdges zuständig. Sie schaut alle Kanten an, die von Knoten in S ausgehen und filtert die heraus, deren zweiter Endpunkt außerhalb von S liegt.

Dann wird für jede Station am ausländischen Ende einer solchen Kante berechnet, wie weit sie von s entfernt ist. Das ergibt einen Menge von Paaren mit einem Knoten $outNode$, dessen Entfernung d von s sich über $d = SP(inNode) + w(inNode, outNode)$ berechnet:

$$\{(outNode, d) \mid (inNode, outNode) \in BE,\ d = SP(inNode) + w(inNode, outNode)\}$$

Von diesen wird dann das Paar (u, d) ausgewählt mit der kürzesten Entfernung d. Damit haben wir den Knoten u, der zu S hinzugenommen wird, und auch die Entfernung d, mit der SP zu $SP + [u \rightarrow d]$ erweitert werden kann.

Zusammen als etwas stärker formalisierter Algorithmus:

ShortestPaths (s, V, E, w)

$S \leftarrow \{s\}$ *Bereich mit bekanntem SP (Inland)*

$U \leftarrow V \setminus \{s\}$ *Bereich mit unbekanntem SP (Ausland)*

$SP \leftarrow [s \rightarrow 0]$ *bekannte Länge kürzester Pfade*

while $U \neq \emptyset$ *transferiere alle Knoten in bekanntes Gebiet*

$BE \leftarrow$ **boarderEdges**(S, E) *grenzüberquerende Kanten ...*

$NodesWithDistance \leftarrow$ *.. mit ihren Entfernungen von s, soweit bekannt*

$$\{(outNode, d) \mid (inNode, outNode) \in BE, d = SP(inNode) + w(inNode, outNode)\}$$

$(u, d) \leftarrow$ Minimum $\{(node, distance) \in NodesWithDistance\}$ mit *distance* als Vergleichskriterium

$S \leftarrow S \cup \{u\}$ *u's kürzeste Entfernung von s ist definitiv bekannt*

$U \leftarrow U \setminus \{u\}$ *u kommt zum bekannten Gebiet ...*

$SP \leftarrow SP + [u \rightarrow d]$ *... mit gesichertem SP*

return SP

ShortestPaths berechnet die Länge des kürzesten Pfades in einem Graphen mit den Knoten V, den Kanten E und den Kantenlängen w. Die Hilfsfunktion *boarderEdges* wird verwendet, um die Kanten zu bestimmen, welche die Grenze zwischen bekanntem und unbekanntem Gebiet überqueren. Also solche, die von einem Knoten mit bekanntem kürzesten Pfad zu einem Knoten mit unbekanntem kürzesten Pfad führen:

boarderEdges(V, E) *berechne die Kanten in E, die aus V herausführen*

return $\{(x, y) \in E \mid x \in V, y \notin V\}$

Es ist naheliegend, dass man neben der Länge des kürzesten Pfades auch diesen selbst berechnen will. Diese zusätzliche Information passt gut zu den Zahlwerten in der Abbildung SP. Der Algorithmus kann entsprechend erweitert werden zu einem, der nicht nur die kürzeste Entfernung d berechnet, sondern auch noch den entsprechenden Pfad p: Statt der Paare (Knoten, Distanz) werden Tripel (Knoten, Distanz, Pfad) berechnet und SP bildet Knoten auf Paare (Distanz, Pfad) ab:

ShortestPathsWithPath (s, V, E, w)

$S \leftarrow \{s\}$
$U \leftarrow V \setminus \{s\}$
$SP \leftarrow [s \rightarrow (0, s :: Nil]$ *bekannte kürzeste Pfadlänge mit dem Pfad*
while $U \neq \emptyset$
  $BE \leftarrow$ **boarderEdges**(S, E) *grenzüberquerende Kanten*
  $NodesWithDistanceAndPath \leftarrow$ *.. mit Entfernung und Pfad*
    $\{(outNode, d, p) \mid$ *Grenzstation outNode, im Ausland mit:*
      $(inNode, outNode) \in BE,$
      $d = SP(inNode).1 + w(inNode, outNode),$ *kürzeste Pfadlänge*
      $p = SP(inNode).2 + [outNode])\}$ *kürzester Pfad*
  $(u, d, p) \leftarrow$ Minimum $\{(node, distance, path) \in NodesWithDistanceAndPath\}$
      mit *distance* als Vergleichskriterium
  $S \leftarrow S \cup \{u\}$
  $U \leftarrow U \setminus \{u\}$
  $SP \leftarrow SP + (u \rightarrow (d, p))$
return SP

Das mag auf den ersten Blick etwas kompliziert erscheinen, aber der Kern des Algorithmus ist recht einfach und entspricht dem oben hergeleiteten Verfahren. Er hat allerdings noch ein ordentliches Potenzial an Optimierungsmöglichkeiten. Wird es ausgeschöpft, dann kommt man zu Dijkstras *Single-Source-Shortest-Path*-Algorithmus SSP.

Dijkstras Algorithmus

Der Algorithmus zum Auffinden des kürzesten Pfades zu jedem Knoten in einem Graphen kann optimiert werden. Die wichtigste Aktion ist die Bestimmung des Nachbarn eines Knotens in S, der außerhalb von S liegt und die kürzeste Entfernung zu s hat. Dazu werden alle Nachbarknoten von Knoten in S, die außerhalb von S liegen, bestimmt und ihre Entfernung von s berechnet. Von diesen wird der Knoten mit der kleinsten Distanz ausgewählt.

Gibt es hier eventuell Effizienzgewinne durch die Vermeidung unnötiger Arbeit? Ist es wirklich notwendig, in jedem Schleifendurchlauf alles komplett neu zu berechnen? Es kommt ja nur ein einziger Knoten zu S hinzu und SP wird um einen einzigen Eintrag erweitert. Muss man wirklich wieder alle ausländischen Grenzstationen berechnen, ihre Entfernung von s bestimmen und über alles das Minimum bilden? Ein genauerer Blick lohnt hier sicher.

Von Schritt zu Schritt, in jedem Schleifendurchlauf, ändert sich V durch die Hinzunahme eines neuen Knotens u. Die im Schritt vorher berechneten Pfade und Pfadlängen können sich aber nur dann ändern, wenn das neue u an ihnen beteiligt ist. Es kann sein, dass es von u aus eine bisher unbekannte direkte Verbindung zu einer

altbekannten Grenzstation gibt (v und w im Beispiel von Abbildung 10.9). Deren Wert muss neu berechnet werden. Er kann sich ändern, weil jetzt ein kürzerer Pfad bekannt ist (das ist im Beispiel bei v so), er kann aber auch gleich bleiben (w im Beispiel).

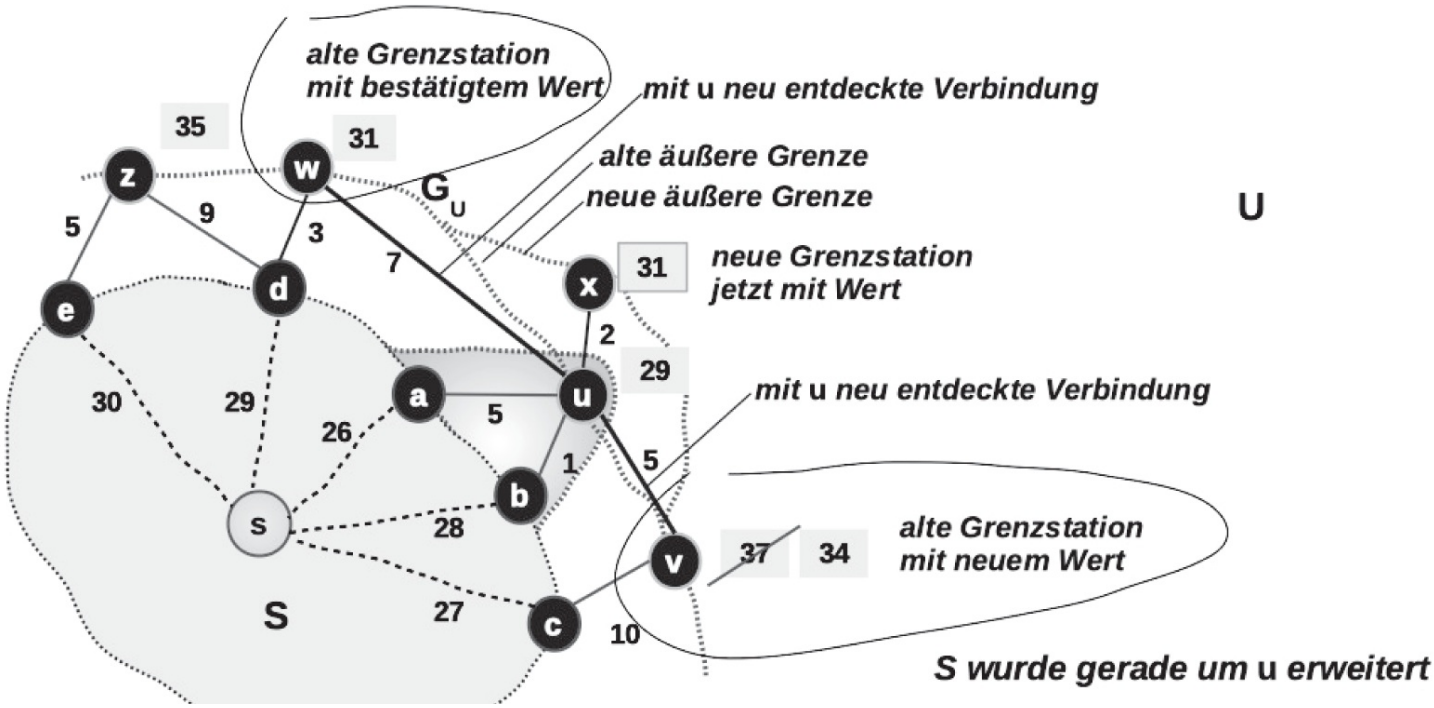


Abbildung 10.9: Nicht alle Pfadlängen müssen neu berechnet werden.

Die Minimierung geht über alle Knoten auf der äußeren Grenze und auch die muss nicht ständig neu berechnet werden. Der innere Bereich S wird in jedem Schritt um einen Knoten u erweitertet. u mutiert dann von einer Grenzstation zu einer im gesicherten Bereich. Die bisher ausländischen direkten Nachbarn von u werden dafür zu Grenzstationen. Statt die ausländischen Stationen an der Grenze immer wieder komplett neu zu berechnen, verwalten wir eine Variable T (siehe Abbildung 10.10). Sie enthält alle Knoten, die nicht zu S gehören, aber über eine Kante direkt mit einem Knoten in S verbunden sind. T wird mit den Nachbarn des Startknotens s initialisiert und mit jedem u , das aus T entnommen wird, um dessen (bisher ausländische) Nachbarn erweitert. Das spart eine Menge Arbeit.

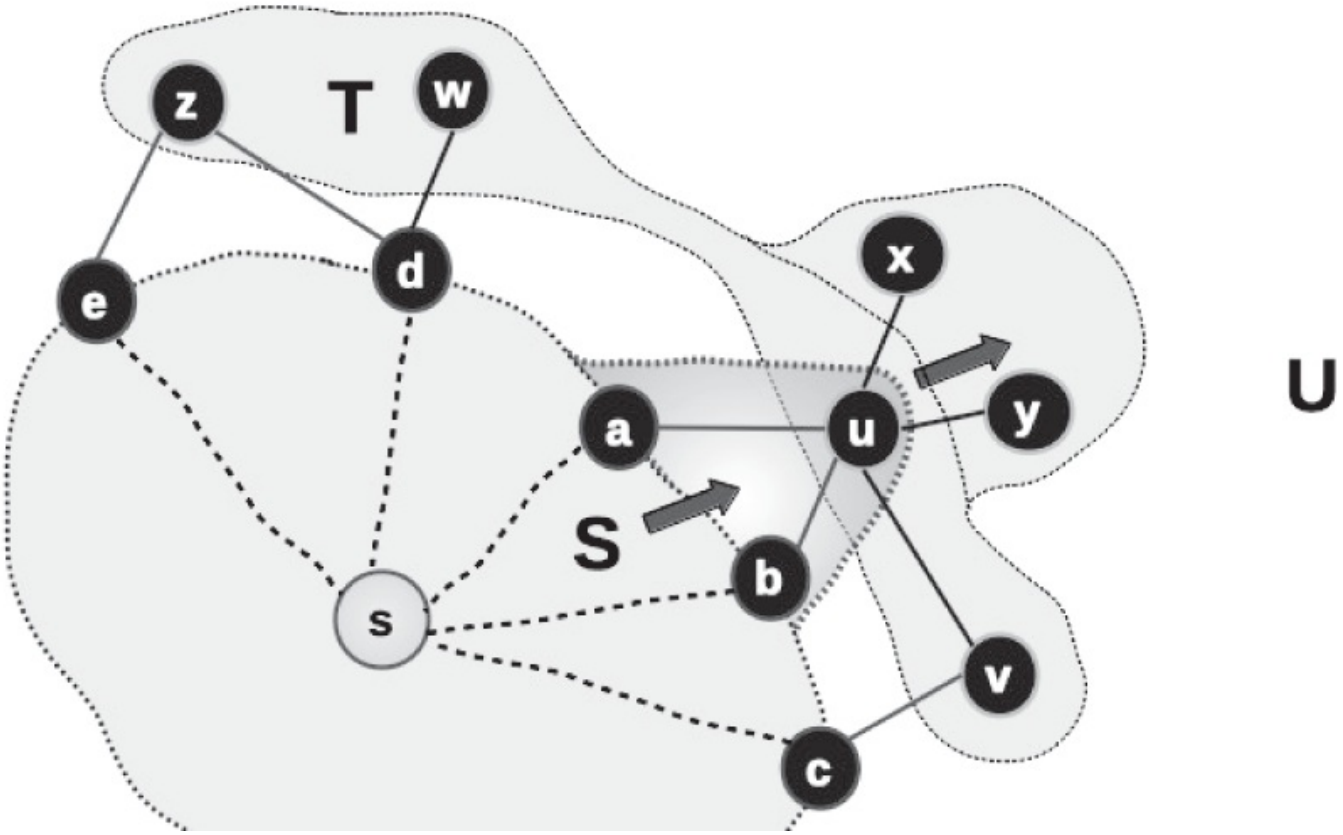


Abbildung 10.10: T mit allen Grenzstationen

Mit T können wir uns auch die Variable U einsparen. U war ja der unerforschte Bereich, der mit allen Knoten außer dem Startknoten initialisiert wurde. U diente zur Kontrolle, ob alle Knoten abgearbeitet wurden. Diese Aufgabe kann aber auch von T übernommen werden. Wenn keine neuen Knoten mehr durch Grenzverschiebungen, also durch Veränderungen von T, entdeckt werden, dann gibt es keinen Knoten mehr, der von s aus erreicht werden kann und noch nicht betrachtet wurde. Gibt es noch weitere Knoten, dann haben sie keine Verbindung zu s und brauchen sowieso nicht betrachtet zu werden.

Damit haben wir folgenden optimierten Algorithmus:

```
SSP (s, E, w)
    S ← {s}                                  Pfade zu s in S sicher, zu t in T nicht sicher
    T ← neighbors(s)                         Knoten am Rand des Unbekannten
    SP ← [s → (0, [s])]                      Knoten mit Entfernung und Pfad
    for n ← neighbors(s)                     Wege zu den Nachbarn in T
        SP ← SP + [s → (w(s, n), [s, n])]

    while T ≠ ∅                              solange es noch Knoten im Unbekannten gibt
        u ← Minimum {v ∈ T}                  u ist bester in T
            mit SP(v).1 als Vergleichskriterium   das Minimum der Entfernung zu s
        S ← S ∪ {u}                          verschiebe u von T nach S
        T ← T \ {u}

        for n ← neighbors(u)
            if n ∈ T                         gibt es eventuell Verbesserungen in SP?
                d ← SP(u).1 + w(u, n)
                if d < SP(n).1               Verbesserung bei n gefunden
                    SP ← [n → (d, SP(u).2 + [n])]    korrigiere Distanz und Pfad
            else if n ∉ S                    n ist neuer Knoten aus dem Unbekannten, kommt zu T
                SP ← [n → (d, p)]            und zu SP mit ersten Werten für
                    mit d = (SP(u).1 + w(u, n),      Distanz und
                    und p = SP(u).2 + [n])]          Pfad
                T ← T ∪ {n}
    return SP
```

neighbors berechnet die direkten Nachbarn eines Knotens. Diese Funktion ist direkt abhängig von der konkreten Speicherung des Graphen und seiner Knoten und Kanten.

In diesem Algorithmus erfüllt SP zwei Aufgaben:

- ✔ Für alle Knoten $x \in S$ enthält es die kürzeste Entfernung des Pfades von s nach x (und den Pfad). Das ist ein fixer und gesicherter Wert. Auf Werten in S entspricht $SP(x)$ hier dem $SP(x)$ in der unoptimierten Version oben.
- ✔ Für die anderen Knoten optimiert SP die Minimierung in jedem Schleifendurchlauf:
    - Für alle Knoten $y \in T$ enthält SP die *bis jetzt bekannte* kürzeste Entfernung des Pfades von s nach y (und den Pfad). Das ist noch kein gesicherter Wert. Er ist Bestandteil der Minimierung entsprechend der Argumentation oben. Kommt ein Knoten u neu zu S, dann ändert sich das Wissen über die Entfernung der Knoten in T. Aber nur für die, die direkte Nachbarn n von u sind. Für alle anderen muss die Entfernung nicht neu berechnet werden. Sie ist die gleiche wie im Schleifendurchlauf vorher.

- Für Knoten $z \in U$, also völlig neu auftretende Knoten aus dem Unbekannten, die in T aufgenommen werden, wird $SP(z)$ erstmalig berechnet.

Insgesamt wird die Entfernung von Knoten in T nicht mehr komplett in jedem Schleifendurchlauf neu berechnet, sondern nur noch, wenn ein Knoten in T neu aufgenommen wird oder sich etwas geändert haben könnte, weil er Nachbar eines Knotens ist, der nach S gewandert ist.

Jeder Knoten $v \in V$ wird genau einmal in S eingefügt. Dabei wird jeweils ein Minimum über die Knoten in V gebildet. Das ergibt $O(|V|^2)$. In der while-Schleife werden alle Kanten untersucht, aber insgesamt wird jede Kante nur einmal betrachtet. Der Algorithmus hat damit die Laufzeitkomplexität $O(|V|^2 + |E|)$. Im Detail wird die Laufzeit stark von der Implementierung der Mengen als Datenstrukturen beeinflusst.

Dieser Algorithmus ist genial und einer der bekanntesten und meist verwendeten Algorithmen, die von E. W. Dijkstra entdeckt wurden. Mit unserer Argumentation wollten wir zeigen, dass auch ein genialer Algorithmus nicht vom Himmel fallen muss, sondern in hartnäckiger Arbeit entwickelt werden kann. Wir wissen natürlich nicht, ob Dijkstra mit den gleichen Überlegungen wie wir zu seinem Algorithmus kam, aber es könnte so gewesen sein wie geschildert.



Algorithmen werden normalerweise nicht zusammen mit den Überlegungen veröffentlicht, die bei ihrer Entwicklung eine Rolle gespielt haben. Das führt oft zu der Fehleinschätzung, dass

- ✔ gute algorithmische Ideen entweder einfach so vom Himmel fallen oder nie kommen und dies wiederum verleitet dazu,
- ✔ den Wert algorithmischer Techniken und harter ausdauernder Arbeit zu unterschätzen.

Datenstrukturen für Dijkstras Algorithmus

Für eine weitere Optimierung sollten noch geeignete Datenstrukturen für die Repräsentation des Graphen gewählt werden. Das gilt besonders dann, wenn der Graph groß ist und viele Knoten und Kanten enthält. Da regelmäßig die Nachbarn eines Knotens durchlaufen werden, bietet sich eine Speicherung als Adjazenzmatrix oder Adjazenzliste an. Statt SP in einer eigenen Datenstruktur zu speichern, könnte $SP(n)$ als Markierung des Knotens n gespeichert werden. Die Berechnung des Minimums könnte beschleunigt werden, wenn T als Prioritätswarteschlange organisiert wird, die die Distanz zu s als Priorität verwendet. Diese Warteschlange kann dann noch effizient in Form eines Heaps organisiert werden (siehe Abbildung 10.11). Dabei ist allerdings zu beachten, dass sich im Heap auch Knoten befinden, deren Distanz zum Startknoten verändert wird. Das ist eine

Operation, die von der Datenstruktur unterstützt werden muss. Speziell verfeinerte Varianten des Heaps können noch weiteres Optimierungspotenzial bieten.

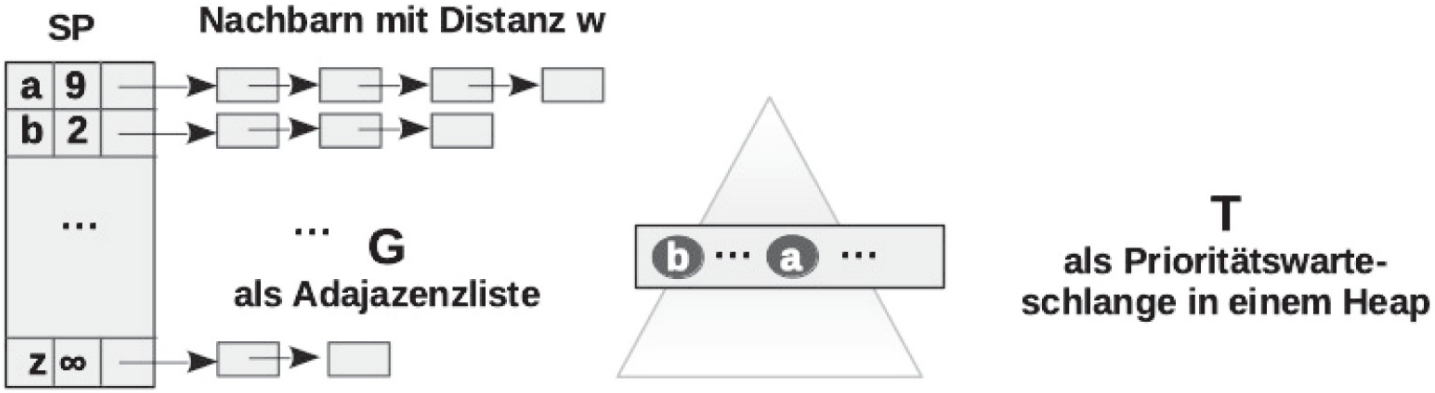


Abbildung 10.11: Datenstrukturen für Dijkstras Algorithmus



In Kapitel 7 werden Heaps und Prioritätswarteschlangen auf Basis von Heaps genauer besprochen.

Verbundenes aufspüren

Im letzten Abschnitt haben wir uns damit beschäftigt, wie die kürzesten Wege von einer Startpunkt s aus zu allen Knoten gefunden werden können. Der Algorithmus setzt voraus, dass es einen Pfad von s aus zu jedem Knoten gibt. Eine verwandte Fragestellung ist, ob jeder Knoten in einem Graphen tatsächlich mit jedem anderen direkt oder indirekt verbunden ist. Wir gehen hier davon aus, dass der Graph ungerichtet ist.

Schaut man einen Graphen mit den Augen an, dann erkennt man meist sofort, ob alle Knoten miteinander verbunden sind. Zumindest dann, wenn er nicht zu groß ist und übersichtlich dargestellt wird. Ist das der Fall, dann sagt man, der Graph habe genau eine *verbundene Komponente*, er ist *verbunden*. Andernfalls besteht er aus mehreren Inseln, seinen verbundenen Komponenten, bei denen es keinen Pfad von einer zur anderen gibt (siehe Abbildung 10.12).

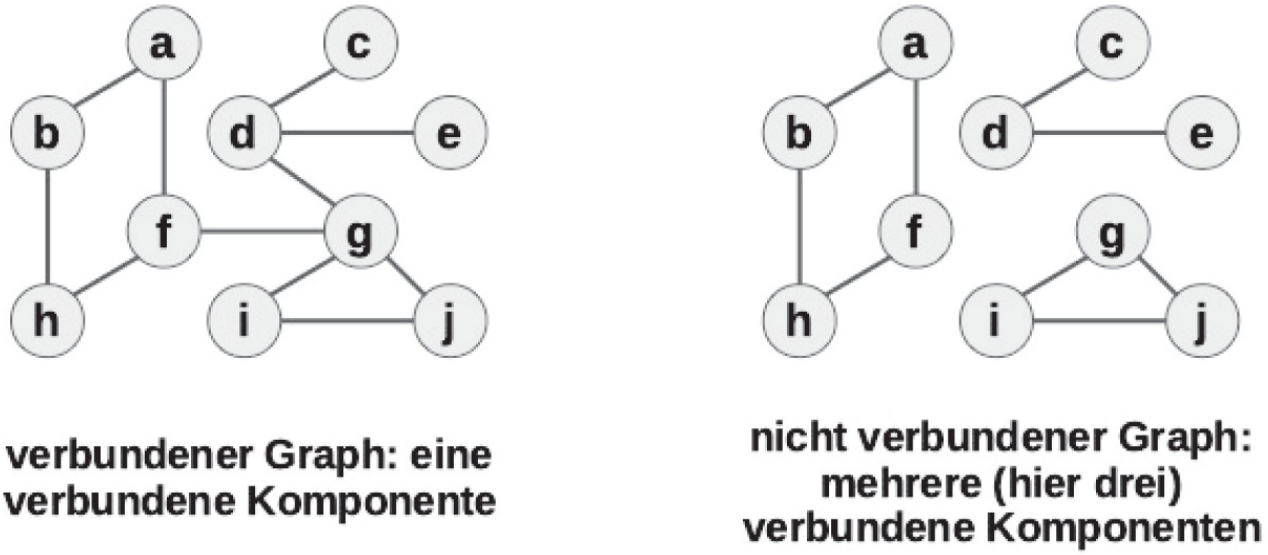


Abbildung 10.12: Verbundene und nicht verbundene Graphen

Eine *verbundene Komponente* eines ungerichteten Graphen $G(V, E)$ ist ein maximaler Teilgraph von G, in dem es einen Pfad von jedem Knoten zu jedem anderen gibt. Ein Graph ist *verbunden* (auch *zusammenhängend*), wenn er sich selbst als einzige verbundene Komponente hat.

Wie bei fast allen Problemen in Zusammenhang mit Graphen sieht man sofort eine geographische Anwendung. Hier könnte es darum gehen, ob man, bei den gegebenen Verbindungen, eventuell von einem Ort zu anderen kommen kann. Andere Anwendungen sind meist weniger offensichtlich, aber oft genauso wichtig. Die verbundenen Komponenten müssen keine Orte sein, die durch Straßen verbunden sind, es könnten auch Freundeskreise (»Blasen«) in sozialen Medien sein oder auch die von einer Spielstellung aus erreichbaren künftigen Spielstellungen.

Verbundene Komponenten identifizieren

Was man mit den Augen ganz einfach erkennen kann, zumindest bei einigermaßen übersichtlichen Graphen (und einigmaßen ungetrübten Augen), ist algorithmisch nicht ganz so einfach. Es fehlt einem Algorithmus ja stets die globale Übersicht. Er muss in der Ebene auf Knoten und Kanten herumkriechen. Es ist aber auch klar, dass ein einfaches Durchsuchen des Graphen nach Ameisenart nicht reichen kann. Ausgehend von einem Knoten und dann immer die Kanten entlang, das kann nicht zur Lösung führen. Wir bleiben dabei ja immer in der Komponente des Startknotens. Andere Komponenten bekommen wir nicht zu sehen und erfahren nicht einmal, ob es sie überhaupt gibt. Ohne ein Minimum an Übersicht geht es also nicht.

Wir nehmen also an, dass wir zumindest einen direkten Zugriff auf alle Knoten und die von ihnen ausgehenden Kanten haben. Das reicht völlig, um den Rest nach Ameisenart

erledigen zu können. Los geht's: Auf jeden Knoten setzen wir eine Ameise und geben ihr den Auftrag, ihr Gebiet zu erkunden und zu markieren. Die Ameise läuft ihr Gebiet ab, indem sie allen Kanten folgt und auf jedem Knoten ihre Markierung absetzt.

Treffen dabei zwei Ameisen zusammen (siehe Abbildung 10.13), dann verzichten sie auf jeden Revierkampf. Sie bilden stattdessen ein Team und markieren ihr Territorium mit einer gemeinsamen Marke.

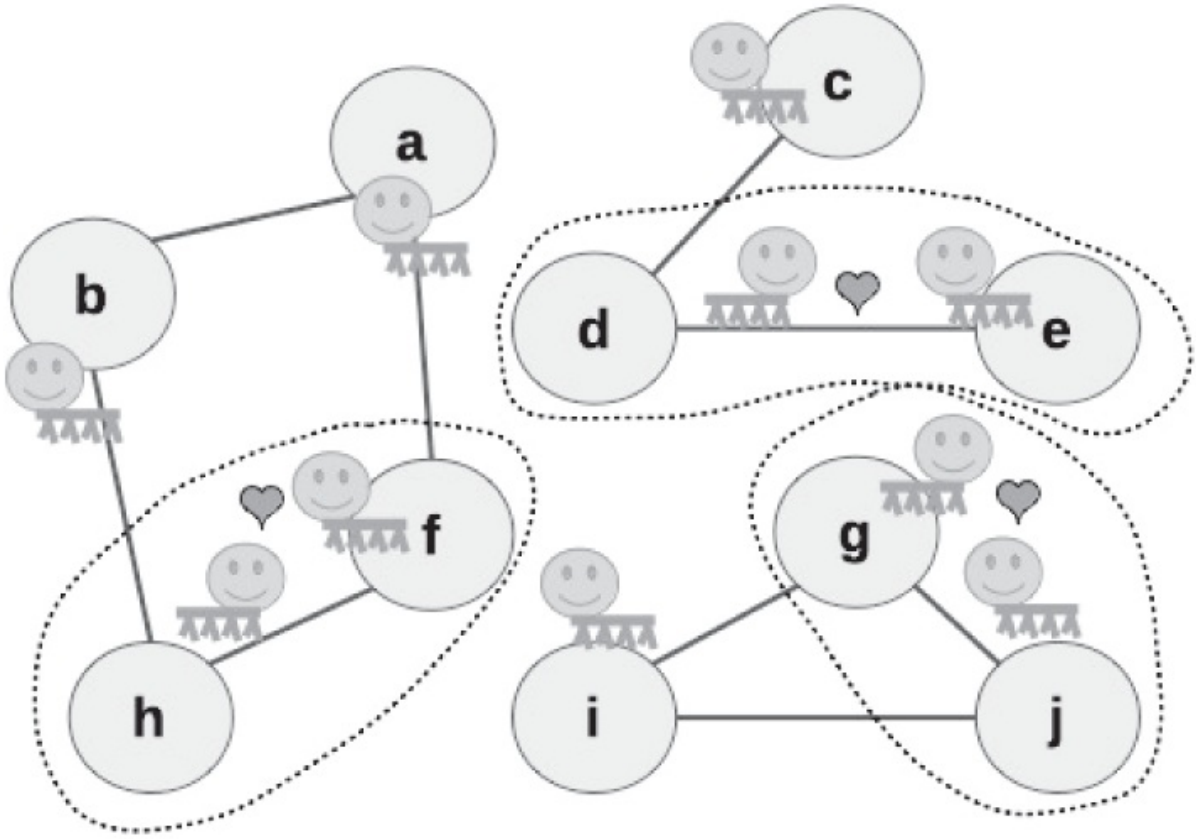


Abbildung 10.13: Aus der Ameisenperspektive verbundene Komponenten finden

Wir beobachten das Treiben der Ameisen eine Zeit lang und stellen fest, dass sie sich schwer damit tun, nicht im Kreis herumzulaufen und sich nach einer Begegnung synchronisiert zu bewegen. Mit etwas mehr Überblicks- und Herrschaftswissen kann die Sache aber deutlich vereinfacht werden.

Mit einem direkten Zugriff auf die Knoten, und zusätzlich auch auf die Kanten, können wir unseren Tierchen das Leben etwas leichter machen. Erstmal werden alle Kanten entfernt. Jeder Knoten bekommt eine Ameise, deren Revier dieser Knoten ist. Jetzt nehmen wir eine Kante nach der anderen und fügen sie zum Graphen hinzu. Schön langsam, damit wir sehen, was passiert. Die Ameisen sehen die Kante und untersuchen sie, und wenn diese sie in eine fremde Region führt, dann vereinigen sie sich und ihre Region mit der anderen Region und deren Bewohnern (siehe Abbildung 10.14).

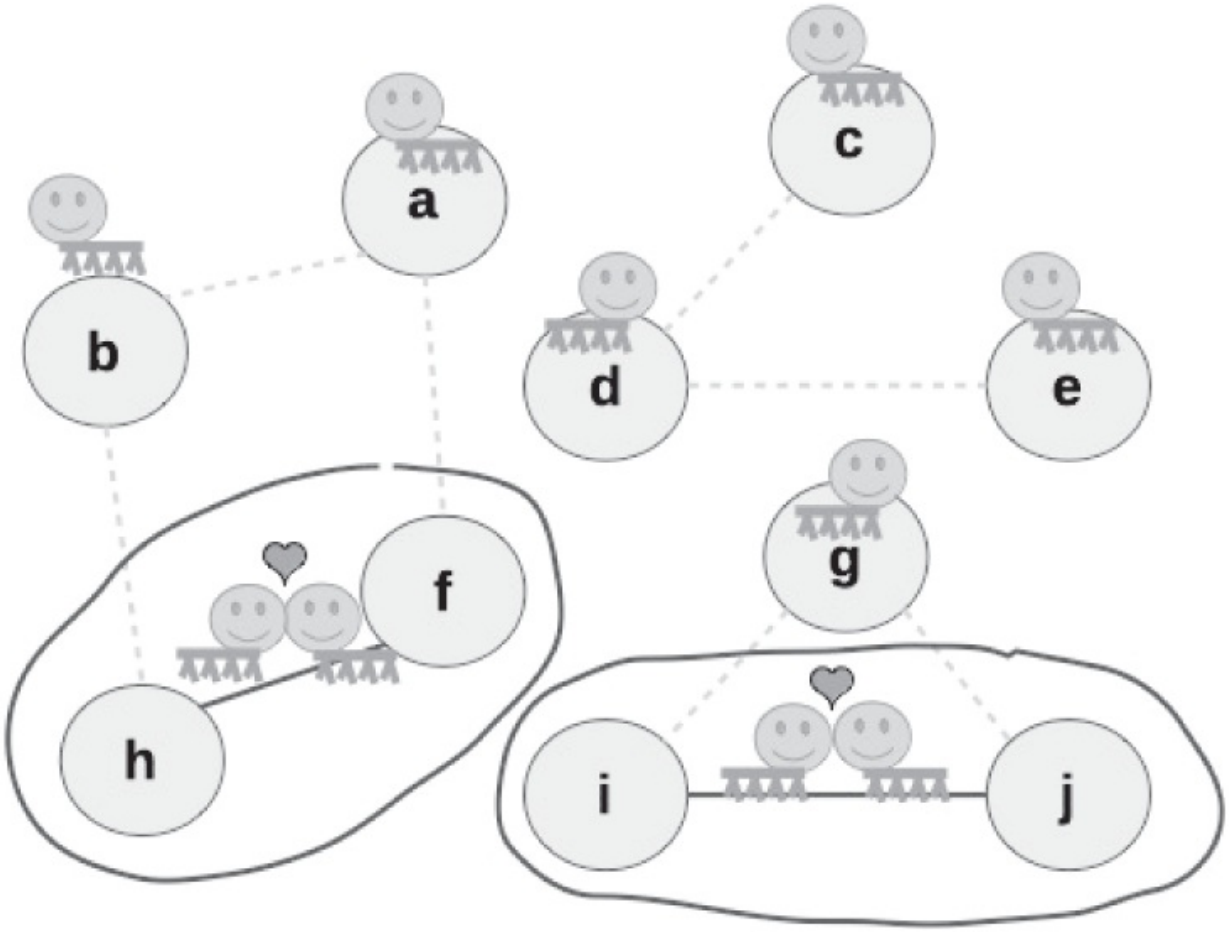


Abbildung 10.14: Ameisen bilden (mit etwas globaler Hilfe) verbundene Komponenten.

Es kann natürlich sein, dass die neu hinzugekommene Kante zu einem Knoten führt, der bereits zur eigenen Region führt (siehe Abbildung 10.15). Dann passiert nichts weiter.

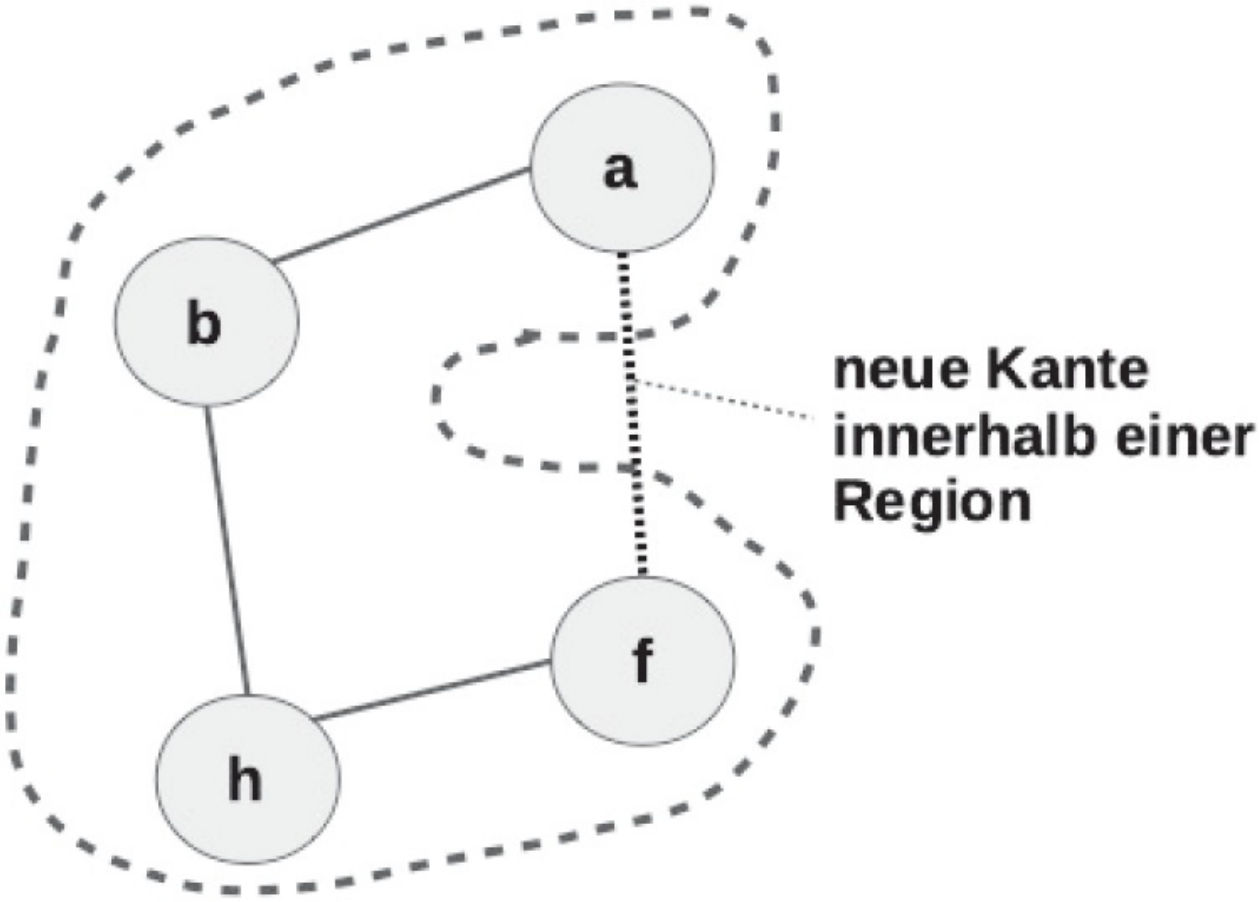


Abbildung 10.15: Manche Kanten führen nicht in andere Regionen.

Der Algorithmus ist recht einfach: Am Anfang bildet jeder Knoten seine eigene Region. Dann werden sukzessive alle Kanten hinzugenommen. Verbindet dabei eine Kante zwei bisher unterschiedliche Regionen, dann werden diese zu einer vereinigt. Das Verfahren endet, wenn alle Kanten abgearbeitet sind. Die verbleibenden Regionen sind die verbundenen Komponenten des Graphen.

Der Algorithmus im Detail ist:

- ✔ Am Anfang bildet jeder Knoten eine Region.
- ✔ Dann werden die vorher entfernten Kanten eine nach der anderen hinzugefügt. Mit jeder hinzukommenden Kante (x, y) werden die Ameisen an den beiden Enden aktiv:
  - Sie beschnuppern sich um festzustellen, ob sie zur gleichen Region gehören.
  - Wenn nein, dann wird ein Vereinigungsfest gefeiert und die beiden Regionen zu einer verschmolzen.

Das Interessanteste an diesem Algorithmus sind die Datenstrukturen, insbesondere die Datenstruktur zur Verwaltung der Regionen. Da gibt es diverse Optionen. Entweder

- werden alle Knoten einer Region mit einer gemeinsamen »Knotenfarbe« markiert oder
- die Zugehörigkeit eines Knotens $v \in V$ zu einer Region $r \in R$ wird in einer Abbildung $V \to R$ gespeichert, oder
- jede Region wird durch die Menge ihrer Mitglieder repräsentiert
- und so weiter.

Und dann haben wir es ja auch nicht nur mit nur einer Region zu tun, sondern mit einer Menge von Regionen, also einer Menge von Mengen, für die eine passende Datenstruktur definiert werden muss – lassen wir die Frage nach der passenden Datenstruktur erst einmal offen.

Stattdessen nehmen wir eine mathematische Haltung an: Regionen werden durch die Menge ihrer Mitglieder definiert. Die wichtigsten Operationen auf diesen Mengen sind:

- MkSet: Erzeuge eine Region.
- FindSetOf: Feststellen, zu welcher Region ein Knoten gehört.
- Unite: Zwei Regionen mit all ihren Knoten zu einer Region vereinigen.

Damit ist der Algorithmus zur Simulation der Ameisen auf einer noch etwas abstrakten Ebne fertig. Abstrakt, weil er mit Regionen als Mengen von Knoten formuliert wird, obwohl ziemlich sicher ist, dass diese Mengen noch durch passende Datenstrukturen ersetzt werden müssen.

```
ConnectedComponents (V, E)
    regions ← { MkSet(v) | v ∈ V }   Menge von Mengen: Jeder Knoten bildet eine Region
    for (x, y) ← E                   für jede Kante (x,y)
        regionOfx ← FindSetOf(x, regions)      suche die Region von x
        regionOfy ← FindSetOf(y, regions)      suche die Region von y
        if regionOfx ≠ regionOfy     bei Bedarf vereinige die Regionen
            regions ← Unite(regions, regionOfx, regionOfy)
    return regions
```

Die Hauptarbeit übernehmen die Hilfsfunktionen. MkSet erzeugt eine Region:

```
MkSet(v)
    return { v }
```

MkSet nimmt einen Knoten und verpackt ihn in einer Menge.

FindSetOf(x , $regions$) sucht in $regions$ nach der Menge, die x enthält:

```
FindSetOf(v, regions)
    for set ← regions
        if set.contains(v)
            return set
```

Da jedes $v \in V$ in genau einer Region enthalten sein muss, liefert FindSetOf immer ein Ergebnis.

Unite($regions$, $regionOfx$, $regionOfy$) entfernt die beiden Regionen $regionOfx$ und $regionOfy$ aus $regions$ und vereinigt sie zu einer neuen Region, die dann zu $regions$ hinzugefügt wird:

```
Unite(regions, region1, region2)
    unitedRegion ← region1 ∪ region2
    return (regions \ {region1, region2}) ∪ {unitedRegion}
```

Alle diese Hilfsfunktionen sind trivial bis einfach. Es war nicht zwingend notwendig, ihre Aktionen in eigenen Funktionen herauszuziehen. Aber mit ihnen werden die Aktionen identifiziert, die auf den »wirklichen« Datenstrukturen arbeiten werden. Die Mengen beschreiben ja erst einmal nur, welche Informationen diese Datenstrukturen enthalten sollen, nicht wie sie real zu organisieren sind.



Unterscheide in Algorithmen die Beschreibung (meist in mathematischer Form) der zu verarbeitenden Informationen von den Datenstrukturen, in denen diese Informationen tatsächlich gespeichert werden.

Datenstrukturen bei der Berechnung verbundener Komponenten

Der Algorithmus operiert im Wesentlichen auf Regionen, das heißt auf Mengen von Mengen von Knoten. Seine Effizienz hängt darum stark davon ab, wie geschickt diese Mengen von Mengen gespeichert werden. Es gibt diverse Möglichkeiten, Mengen in einer Datenstruktur zu speichern. Üblich sind beispielsweise Suchbäume und Hashtabellen.

Unsere Regionen sind Mengen, aber sie haben spezielle Eigenschaften, die sich eventuell ausnutzen lassen. Es werden nur die folgenden Operationen benötigt:

- ✔ *MkSet* erzeugt eine Region aus einem Knoten.
- ✔ *FindSetOf* sucht die Region, zu der ein Knoten gehört.
- ✔ *Unite* ersetzt zwei Regionen, also zwei Knotenmengen, durch ihre Vereinigung.

Die Regionen sind zwar Mengen von Mengen. Es werden aber keineswegs beliebige Mengen von Mengen zusammengefügt. Jede Region

- ✔ besteht nur aus Elementen der Knotenmenge V,
- ✔ alle Regionen sind disjunkt, das heißt, sie haben kein Element gemeinsam, und außerdem ergibt
- ✔ die Vereinigung aller Regionen wieder V.

So etwas nennt man Partitionen von V (siehe Abbildung 10.16). Eigentlich sind es *Partitionierungen* von V, der Begriff *Partitionen* ist im mathematischen Kontext aber geläufiger.

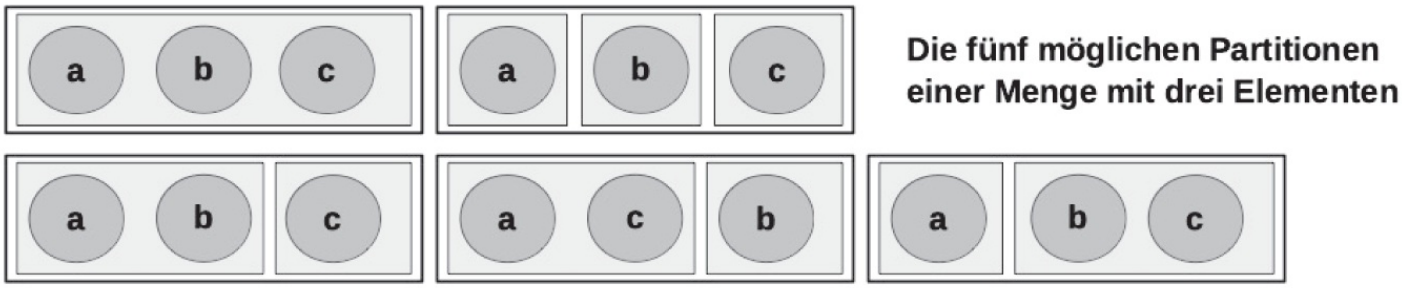


Abbildung 10.16: Partitionen: Aufteilungen einer Menge

Mithilfe einer einfachen Datenstruktur können die Regionen des Graphen – oder allgemeiner Partitionen einer Menge – so abgespeichert werden, dass die beiden Operationen *FindSetOf* und *Unite* effizient auszuführen sind. Die Idee ist nicht wirklich erstaunlich, wir nutzen wieder einmal Bäume. Jede Partition/Region wird durch einen Baum repräsentiert.

Am Anfang ist jeder Knoten ein Baum für sich. Ein Baum, der nur aus einer Wurzel besteht, ist etwas unscheinbar, aber immerhin ein Baum. Werden jetzt zwei Regionen vereinigt, dann werden die Bäume, die sie repräsentieren, zu einem Baum vereinigt. Bei zwei Bäumen, die nur aus einem Knoten bestehen, ernennen wir einfach einen der beiden zum Elternknoten und den anderen zum Kind (siehe Abbildung 10.17).

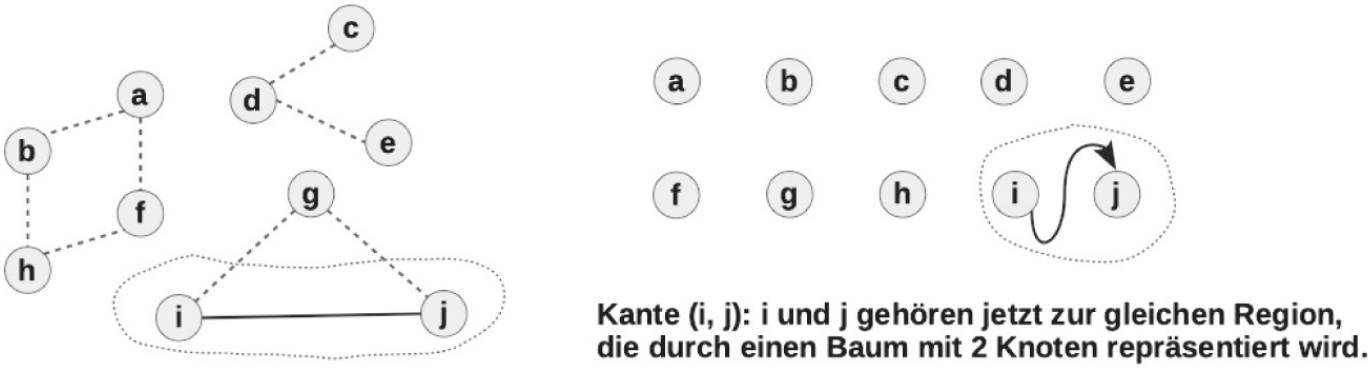


Abbildung 10.17: Zwei Regionen vereinigen

Nach der Vereinigung weiterer Regionen haben wir mehrere kleine Bäume mit jeweils

zwei Knoten (siehe Abbildung 10.18). Die Entscheidung, wer von den beiden die Wurzel und wer das Blatt sein soll, ist dabei völlig willkürlich.

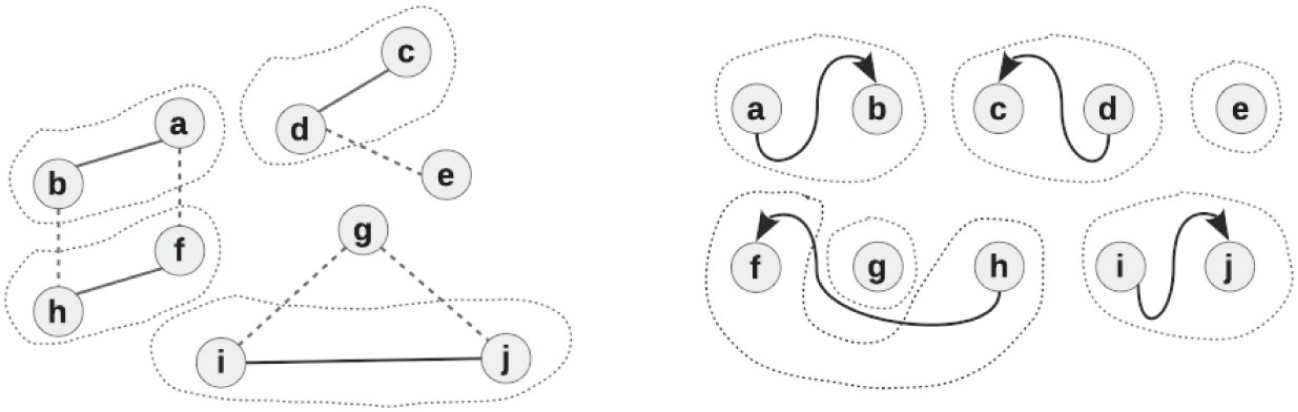


Abbildung 10.18: Nach der Vereinigung einiger Regionen

Mit der Kante (d, e) kommen jetzt zwei Bäume unterschiedlicher Größe zusammen. In dem Fall nehmen wir die Wurzel des größeren Baumes als Wurzel des vereinigten Baums. Diese Strategie minimiert die Baumhöhe. Mit der Kante (a, f) werden zwei weitere Regionen vereinigt. Die Wurzel des einen Baumes wird wieder unter die Wurzel des anderen gehängt. Beide sind gleich groß. Wir wählen willkürlich die Wurzel des einen (a) als Vorgänger der Wurzel (h) des anderen (siehe Abbildung 10.19).

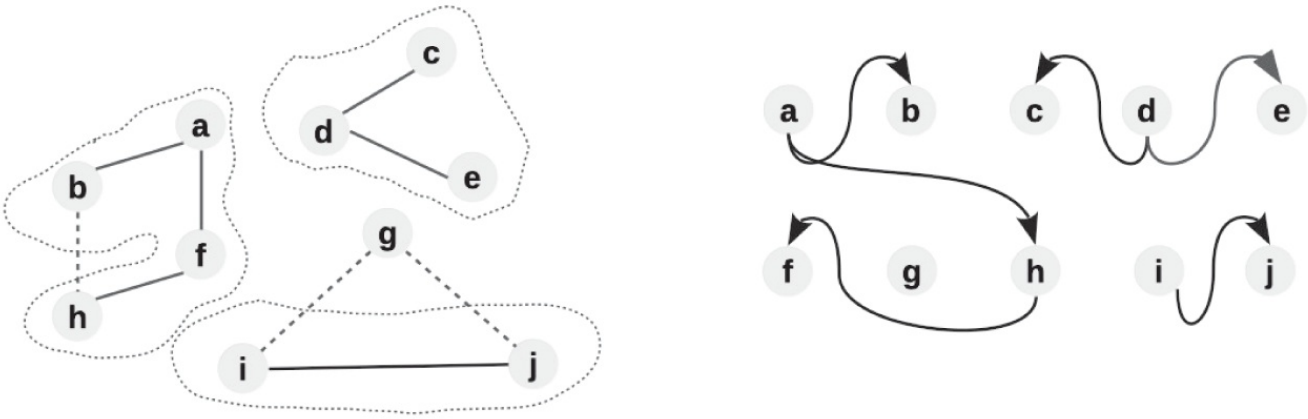


Abbildung 10.19: Kleinere Bäume unter größere hängen

Die Kante (b, h) wird ignoriert, denn b und h sind in der gleichen Region, was man daran erkennt, dass sie die gleiche Wurzel haben. (g, i) fügt g in den Baum mit Wurzel i ein. (g, j) wird ignoriert, da g und j jetzt die gleiche Wurzel i haben. Damit sind alle Kanten abgearbeitet und verbundenen Komponenten als Bäume identifiziert (siehe Abbildung 10.20).

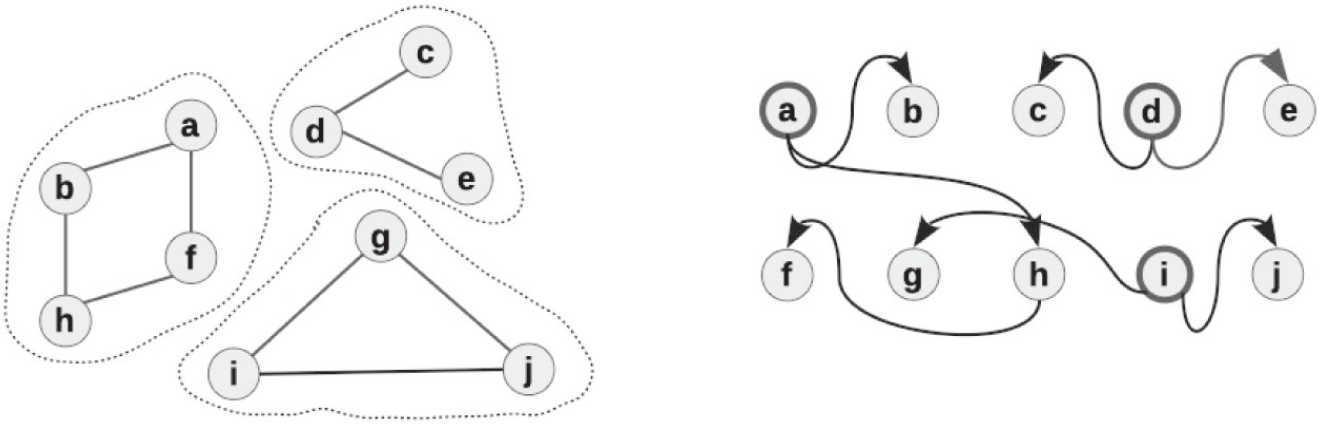


Abbildung 10.20: Die verbundenen Komponenten sind als Bäume identifiziert.

Disjunkte Mengen als Datenstruktur

Die Bäume repräsentieren Partitionen einer festen Kantenmenge. Die Zahl der Kanten ändert sich nicht und darum ist es nicht notwendig, während der Laufzeit des Algorithmus Speicher anzufordern. Das steigert die Effizienz erheblich. Alles kann sich darum in Arrays fester Größe abspielen. Eine entsprechende Datenstruktur ist unter dem Namen *Disjoint Set* bekannt.

Die Umsetzung der Idee kann im Detail variieren, sie ist aber im Prinzip recht naheliegend. Alle Knoten können, zusammen mit weiteren Informationen, in einem Array versammelt werden. Die weiteren Informationen sind sinnvollerweise Informationen über die Position des Knotens in einem der Bäume, also die Eltern-Kind-Beziehung.

Speichert man die Eltern-Kind-Beziehung der Bäume nicht wie üblich in Richtung Elternknoten zu Kind, sondern invers, also rückwärts vom Kind zum Elternknoten, dann ist einem Knoten nur ein anderer zugeordnet. Das kann in einem einfachen Array abgespeichert werden. $a[v]$ enthält den Elternknoten von v. Für unser Beispiel ergibt sich damit ein Array entsprechend Abbildung 10.21. Eine Wurzel lassen wir einfach auf sich selbst zeigen.

a	a
b	a
c	d
d	d
e	d
f	h
g	i
h	a
i	i
j	i

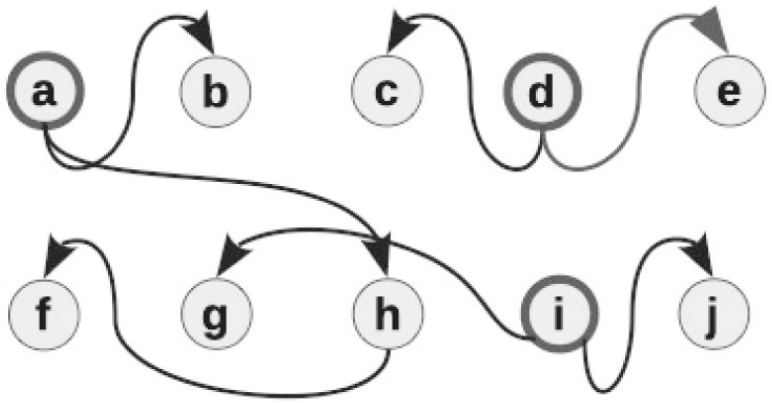


Abbildung 10.21: Bäume invers in einem Array

Diese Art der Speicherung hat für die Anforderungen hier mehrere Vorteile:

- ✔ Pro Knoten wird nur ein Verweis benötigt, der auf den Elternknoten.
- ✔ Der Test, ob zwei Knoten zum gleichen Baum (zur gleichen Menge) gehören, wird zum Test, ob die Kette der Vorfahren beider in der gleichen Wurzel endet.
- ✔ Die Vereinigung von zwei Mengen wird zum Verschmelzen der Bäume, die sie repräsentieren. Dazu suchen wir die Wurzeln der beiden Bäume und hängen, bei unterschiedlich großen Wegen bis zur Wurzel, die Wurzel des kleineren Baumes unter die des größeren.

Die Regionen werden damit als ein einziges Array von Knoten gespeichert. Der Einfachheit halber nehmen wir an, dass dieses Array auch über Knoten adressiert werden kann (a[v] mit $v \in V$ ist möglich). Die Ausgangspartitionierung, bei jeder Knoten in seiner eigenen Menge steckt, wird erreicht mit

```
K: array [V] of V
for v ← V
    K[v] ← v                    jedes v ist Wurzel und zeigt auf sich selbst
```

In fast allen Programmiersprachen können nur natürliche Zahlen als Indizes verwendet werden. Dort müssen die Knoten als natürliche Zahlen verwaltet werden oder die Zuordnung Knoten zu Position muss im Array auf andere Art bewerkstelligt werden,

beispielsweise mit einer Hashfunktion.

Jedes $v \in V$ ist zunächst einmal Wurzel und zeigt damit in K auf sich selbst. Ein zweites über V indiziertes Array **S** nutzen wir, um die Baumgrößen zu verwalten. **S** wird genutzt, um größere von kleineren Bäumen unterscheiden zu können, ohne jedes Mal nachzählen zu müssen.

```
S: array [V] of Int
for v ← V
    S[v] ← 1                    jeder Baum hat die Größe 1
```

Mit dieser Datenstruktur können FindSetOf und Unite sehr effizient realisiert werden:

```
FindSetOf(v)                    liefert die Wurzel des Baums, zu dem v gehört
    while v ≠ K[v]
        v ← K(V)
    return v
```

FindSetOf(v) liefert die Wurzel des Baums, zu dem v gehört. Also den Knoten, der den Baum repräsentiert, der die Menge repräsentiert, zu der v gehört. So konzentriert sollten Datenstrukturen im Idealfall sein (siehe Abbildung 10.22).

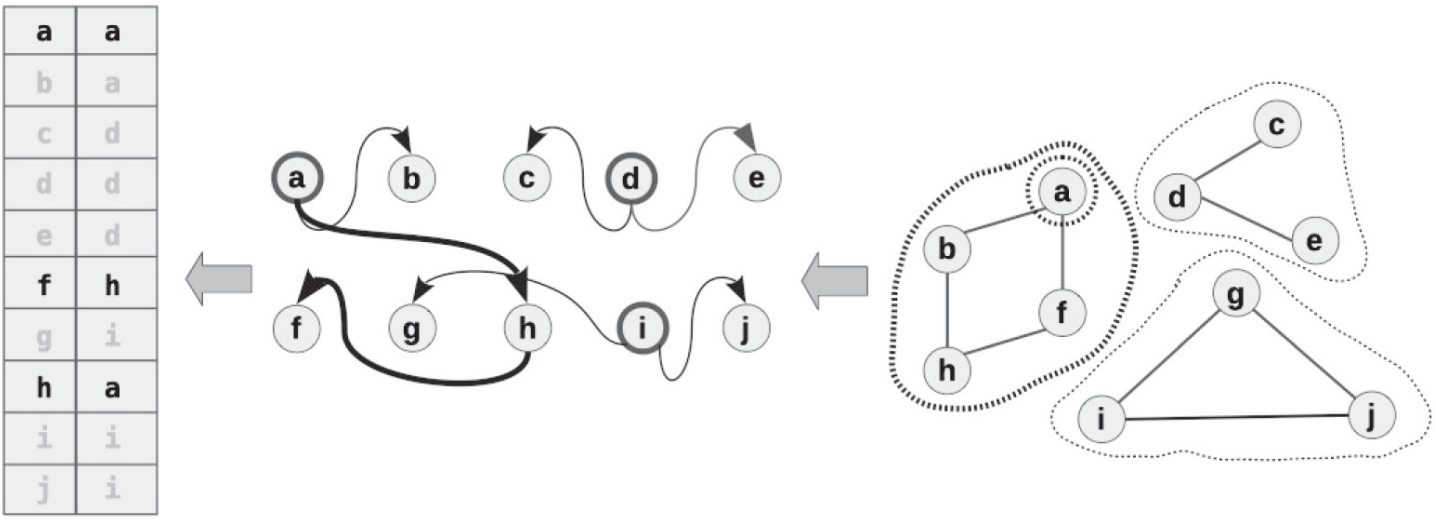


Abbildung 10.22: Menge von Mengen als Menge von Bäumen als Tabelle

Union(u , v) verschmilzt die Bäume, zu denen u und v gehören. Von beiden wird die Wurzel gesucht, dann wird die Wurzel des kleineren unter die Wurzel des größeren gehängt. S hilft bei der Unterscheidung zwischen Groß und Klein und muss nach der Verschmelzung natürlich angepasst werden:

```
Union(u, v)                          verschmilzt die Mengen, zu denen u und v gehören.
    a ← FindSetOf(u)                 a: Repräsentant der Menge zu der u gehört
    b ← FindSetOf(v)                 b: Repräsentant der Menge zu der v gehört
    if S(a) ≥ S(b)                   a repräsentiert die größere Menge (S enthält die Größe)
        S(a) ← S(a) + S(b)           b wird in die Menge, die a repräsentiert, übernommen
        K(b) ← a
    else                             b repräsentiert die größere Menge
        S(b) ← S(b) + S(a)           a wird in die Menge, die b repräsentiert, übernommen
        K(a) ← b
```

MkSet wird als eigenständige Operation nicht mehr benötigt. Seine Funktionalität steckt in der Initialisierung von K und S.

Der auf Mengen operierende Grundalgorithmus im funktionalen Stil von oben kann jetzt an diese optimierten imperativen Datenstrukturen angepasst werden:

```
ConnectedComponents (V, E)
    K: array [V] of V
    for v ← V
        K[v] ← v
S: array [V] of Int
for v ← V
    S[v] ← 1
Union(u, v)                          lokaler Hilfsalgorithmus mit Zugriff auf K und S
    ... wie oben ...
FindSetOf(v)                         lokaler Hilfsalgorithmus mit Zugriff auf K und S
    ... wie oben ...
for (x, y) ← E
    rootx ← FindSetOf(x)
    rooty ← FindSetOf(y)
    if rootx ≠ rooty
        Union(rootx, rooty)
return K
```

Insgesamt steigern diese angepassten Datenstrukturen die Effizienz des Algorithmus ganz erheblich.

Laufzeiten

Die Operation FindSetOf durchläuft einen Baum rückwärts bis zu Wurzel:

```
FindSetOf(v)                    liefert die Wurzel des Baumes, zu dem v gehört
    while v ≠ K[v]
        v ← K(V)
    return v
```

Die Laufzeit der Schleife ist abhängig von der Baumhöhe. Hängt die Baumhöhe von der Eingabegröße ab? Offensichtlich ja: Der Baum besteht aus Knoten des Graphen und der Graph ist die Eingabe des Gesamtalgorithmus. Nehmen wir also an, dass der Graph n Knoten hat. Welche Beziehung besteht nun zwischen der Zahl der Knoten n und der Baumhöhe h, und damit der Zahl der Schleifendurchläufe, im ungünstigsten Fall?

Bezeichnen wir die Zahl der Knoten eines Baumes T als $g(T)$. (g klingt wie *Gewicht*: die übliche Definition für das Gewicht eines Baumes ist zwar die Zahl seiner Blätter, nicht die Zahl seiner Knoten. Das ignorieren wir hier aber einfach mal.) Die *Höhe* $h(T)$ ist die Länge des längsten Pfades von der Wurzel zu einem Blatt. Schlanke Bäume T mit großer Höhe $h(T)$ im Verhältnis zum Gewicht $g(T)$ sind schlecht. Dicke Bäume mit wenig Höhe $h(T)$ im Verhältnis zum Gewicht $g(T)$ sind gut. Wie schlank können die Bäume schlimmstenfalls werden?

Wenn alle Knoten in einer Reihe verkettet sind, dann ist der Baum ganz dünn, er ist zur Liste verkommen: n Knoten sind durch $n-1$ Kanten hintereinander verbunden und es ist $h = n - 1$. Diese Kettenform kann glücklicherweise nicht vorkommen. Wenn zwei Bäume verschmolzen werden, dann wird der kleinere, also der mit weniger Knoten, unter die Wurzel des größeren Baumes gehängt. Die Baumhöhe h erhöht sich nur dann, wenn zwei Bäume vereinigt werden, die gleich viele Knoten haben. Das Gewicht wird verdoppelt und die Höhe h um 1 erhöht. Das legt es nahe, dass die schlechtestmögliche Beziehung von n zu h logarithmisch ist:

$$2^h \leq n$$

beziehungsweise

$$h \leq \log_2 n$$

Die Operation FindSetOf ist damit in $\mathcal{O}(\log n)$. Union hat keine Aktion, die von der Eingabelänge (Zahl der Knoten oder Kanten) abhängt, und ist darum in $\mathcal{O}(1)$.

In ConnectedComponents geht die Schleife

```
for (x, y) ← E
    rootx ← FindSetOf(x)
    rooty ← FindSetOf(y)
    if rootx ≠ rooty
        Union(rootx, rooty)
return K
```

über alle Kanten. In jedem Durchlauf wird neben Operationen mit konstanten Laufzeiten zweimal FindSetOf aufgerufen. FindSetOf ist in $\mathcal{O}(\log n)$, wie wir gerade gesehen haben. n ist die Zahl der Knoten. Wenn der Graph m Kanten hat, dann ist die Laufzeit der Schleife in $\mathcal{O}(m \log n)$. Die Initialisierung der Arrays,

```
ConnectedComponents (V, E)
    K: array [V] of V
    for v ← V
        K[v] ← v
    S: array [V] of Int
    for v ← V
        S[v] ← 1
```

ist abhängig von der Zahl der Knoten und damit in $\mathcal{O}(n)$. Der Algorithmus insgesamt ist dann in $\mathcal{O}(n + m \log n)$. $m \log n$ dominiert n. (Kanten sind Knotenpaare und bei n Knoten kann es $n(n-1)/2$ Knotenpaare geben). Insgesamt haben wir damit eine Laufzeit in $\mathcal{O}(m \log n)$.



In Kapitel 2 werden Laufzeitabschätzungen genauer erläutert, insbesondere die Faustregel 2: *Der stärkste Summand gewinnt.*

Spann mir einen Graphen auf

Verbindungen sind gut. Manchmal ist es aber auch zu viel des Guten. Bei einer Elektroinstallation ist es schön, wenn es viele Steckdosen gibt. Aber von jeder Dose zu jeder anderen und von allen zum Sicherungskasten jeweils eine Leitung zu verlegen, ist sicher zu viel. Es reicht, wenn es eine Verbindung in alle Stockwerke gibt und in diesen dann ein Verteiler platziert wird, von dem aus es zu den einzelnen Stockdosen geht.

Modelliert man die Verkabelung mit den Verteilern und Steckdosen als Graph, dann geht es darum, den Graphen von überflüssigen Kanten zu befreien. Überflüssig sind die Kanten, die nicht zwingend gebraucht werden, um von einem Knoten zu einem anderen zu kommen. Nach dieser Reduktion bleibt ein Grundgerüst des Graphen übrig – seine Essenz (siehe Abbildung 10.23). Diese Grundgerüst wird Spannbaum genannt. Es ist ein Baum, denn wenn alle nicht zwingend notwendigen Kanten entfernt wurden, dann kann es keine zwei Wege zu einem Ziel geben, damit gibt es auch keine Zykel. Ein verbundener ungerichteter Graph ohne Zykel ist immer baumförmig. Der Spannbaum spannt den Graphen mit einer minimalen Zahl von Kanten auf, so wie ein Regenschirm von seinem Gestänge aufgespannt wird. Der Spannbaum ist meist nicht eindeutig bestimmt, es kann also zu einem Graphen mehrere Spannbäume geben.

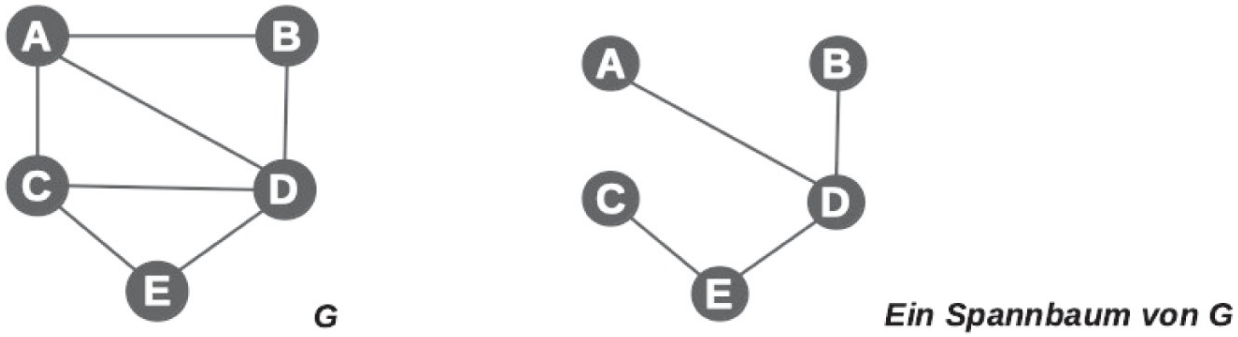


Abbildung 10.23: Spannbaum: Zykelfreier Teilgraph mit allen Knoten

Ein *Spannbaum* eines ungerichteten verbundenen Graphen $G(V, E)$ ist ein zykelfreier verbundener Teilgraph $G(V, E')$ von G, der alle seine Knoten V enthält sowie eine minimale Teilmenge $E' \subseteq E$ der Kanten.

Der Spannbaum kann nur für verbundene Graphen berechnet werden. Dabei geht es darum, möglichst viele Kanten zu eliminieren, ohne dass der Graph dabei in unverbundene Komponenten zerfällt.

Der Spannbaum ist heute wahrscheinlich eines der am häufigsten berechneten Graphenprobleme, da viele Kommunikationsnetze eine entsprechende Berechnung regelmäßig und in kurzen Abständen ausführen, um stets zykelfrei und trotzdem vollständig verbunden zu sein. Der Spannbaum wird generell in Anwendungen benötigt, in denen es um eine kostengünstige Erreichbarkeit geht. Etwa bei Verteilungsproblemen im Transportwesen oder bei Versorgungsnetzen, bei denen alle Stationen mit minimalem Aufwand versorgt werden sollen.

Minimaler Spannbaum

Wenn die Kanten eine Länge haben, dann wird die Sache noch etwas interessanter. Wir können nicht nur einen Spannbaum berechnen, wir können sogar den besten, den minimalen Spannbaum berechnen. Das ist der, dessen Kanten insgesamt die geringste Länge haben.

Der *minimale Spannbaum* eines gewichteten, ungerichteten, verbundenen Graphen $G(V, E, w)$ ist der Spannbaum $G(V, E', w)$ von G mit der kleinsten Summe $\sum_{e \in E'} w(e)$ der Kantenlängen.

Zu einem Graphen kann es auch mehrere minimale Spannbäume geben. Im Beispiel von Abbildung 10.24 ist der minimale Spannbaum eindeutig.

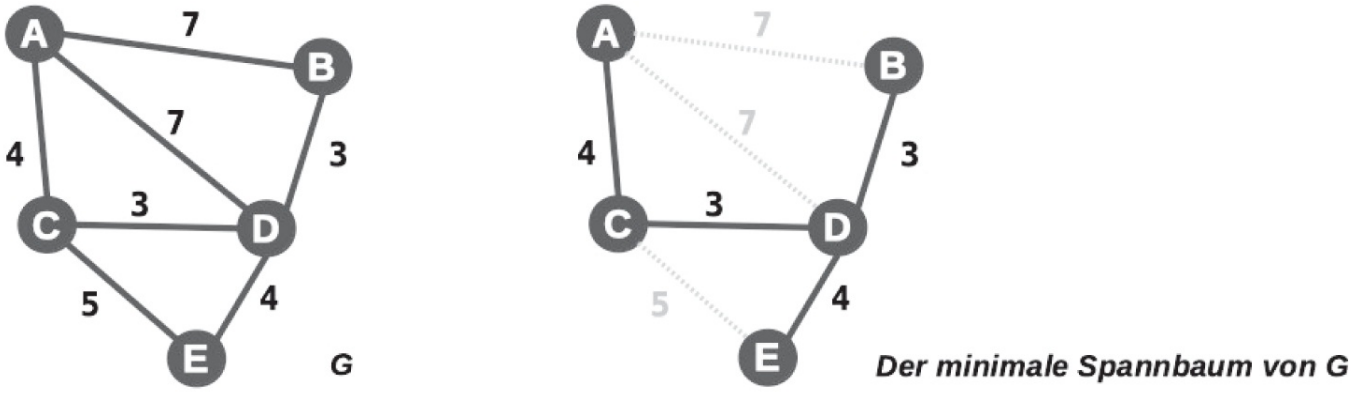


Abbildung 10.24: Ein Graph mit eindeutigem minimalem Spannbaum

Der Spannbaum ist wurzellos

Das Interessante an einem Spannbaum, minimal oder nicht, ist die Tatsache, dass er keine Wurzel hat. Alle seine Knoten sind gleichberechtigt und wir könnten irgendeinen als Wurzel definieren. Mit definierter Wurzel würde er zu einem etwas vertrauteren Wesen, einem Wurzelbaum. Im Beispiel von Abbildung 10.24 könnten wir beispielsweise den Knoten mit den meisten Kanten, also D zur Wurzel erklären. Damit bekämen alle Kanten eine Richtung: weg von der Wurzel (siehe Abbildung 10.25).

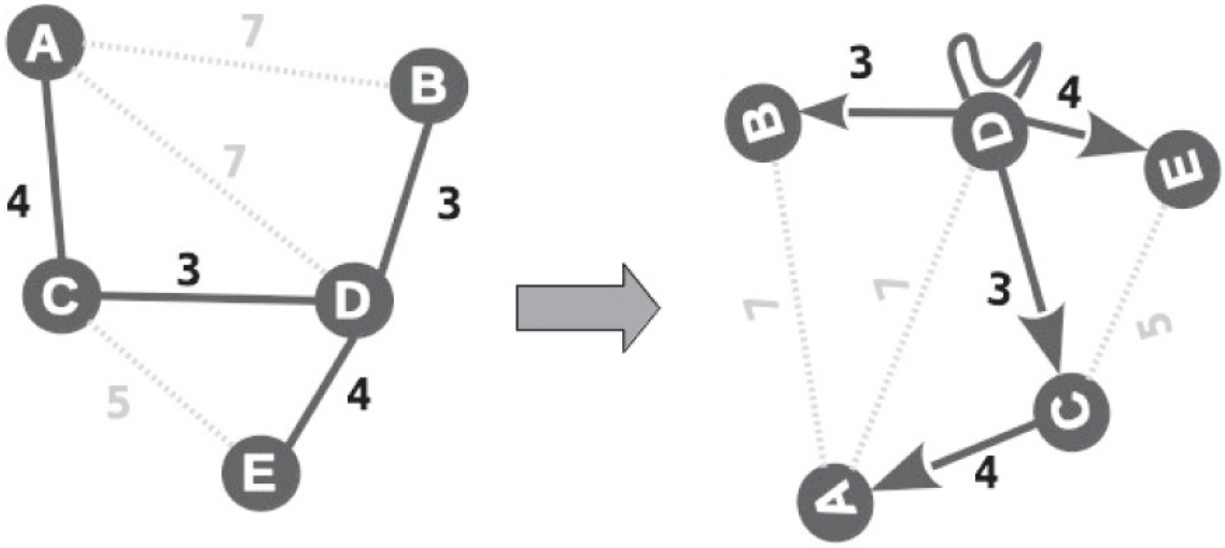


Abbildung 10.25: Einem wurzel- und richtungslosen Baum Orientierung geben

Kruskals Algorithmus

Zur Berechnung des minimalen Spannbaums gibt es mehrere Algorithmen. Einer von ihnen, der Algorithmus von *Kruskal*, basiert auf dem oben besprochen Algorithmus zum Auffinden verbundener Komponenten. Was haben nun verbundene Komponenten mit Spannbäumen zu tun? Wir gehen bei einem Spannbaum doch schon gleich davon aus, das der gesamte Graph verbunden ist. Es gibt da nichts an verbundenen Komponenten zu

finden.

Beim Aufspüren verbundener Komponenten wird der Graph zuerst von allen Kanten befreit und dann wird eine nach der anderen wieder hinzugenommen. Kanten, die unverbundene Regionen verbinden, führen zu Aktionen, die anderen werden ignoriert. Das scheint tatsächlich auch ein sinnvolles Vorgehensweise zu sein, um den Spannbaum zu berechnen: Die Kanten, die zu einer Vereinigung von Regionen führen, gehören wohl zum Spannbaum, die anderen eben nicht.

Experimentieren wir mal ein bisschen mit unserem Graphen aus Abbildung 10.25 und setzen den Algorithmus zum Aufspüren verbundener Komponenten auf ihn an (siehe Abbildung 10.26). Klar, wir wissen schon im Voraus, dass es nur eine verbundene Komponente gibt, aber wir wollen beobachten, welche Bäumchen dabei aufgebaut werden.

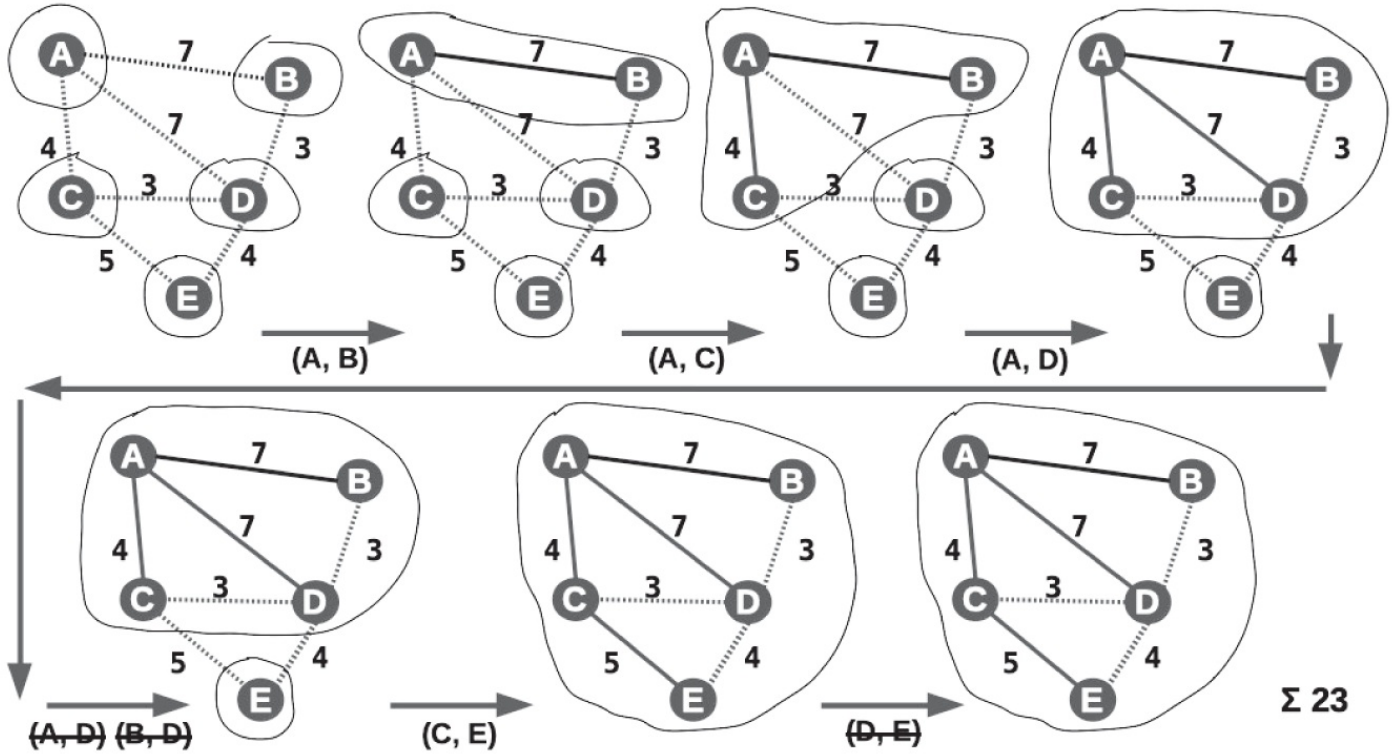


Abbildung 10.26: Spannbaum als Beifang beim Fischen nach verbundenen Komponenten

Der Algorithmus liefert nicht nur zufällig einen Spannbaum. Er macht es notwendigerweise. Jede nicht ignorierte Kante verbindet bis dahin unverbundene Regionen. Sie muss also notwendig zu einem Spannbaum gehören. Ignorierte Kanten verbinden Regionen, die schon verbunden sind. Sie haben im Spannbaum somit nichts zu suchen.

Ein kleines Problem bleibt noch. Es wird zwar ein korrekter Spannbaum berechnet, aber der ist bestenfalls per Zufall minimal. Im Beispiel von Abbildung 10.26 haben wir einen Spannbaum der Länge 23 berechnet, der minimale Spannbaum hat aber die Länge 14.

Es ist klar, dass der berechnete Spannbaum davon abhängt, in welcher Reihenfolge die Kanten verarbeitet werden. Wenn es mehrere Kanten gibt, die zwei bisher unverbundene Regionen verbinden, dann sollten wir nicht irgendeine nehmen, bloß weil sie in der Kantenliste als erste vorkommt. Besser wäre es, die kürzeste zu nehmen. Wenn also zwei Regionen vor der Vereinigung stehen, dann suchen wir sinnvollerweise nach der kürzesten Kante, die sie verbindet.

Hmm, in jedem Schritt erst nach der kürzesten Kante suchen, das macht die Sache dann schon wieder ziemlich aufwendig. Wenn eine Kante hinzugefügt wird, dann müssen wir sicher sein, dass es keine gibt, die kürzer ist und in die gleiche Region führt. Die kürzeste Kante zu suchen, ist sicher kein Problem, aber die kürzeste, die in eine bestimmte Region führt?

Glücklicherweise gibt es eine einfache Lösung für dieses Problem: Wenn wir alle Kanten der Größe nach sortieren und in dieser Reihenfolge untersuchen, dann begegnen uns automatisch die kürzeren vor den längeren. Das gilt auch für eine bestimmte Region. Niemals kommt eine längere Kante, die in eine bestimmte Region führt, vor einer kürzeren, die in die gleiche Region führt. Mit dieser Idee wird der Algorithmus zur Bestimmung des minimalen Spannbaums zu einer einfachen Variante des Algorithmus zur Bestimmung der verbundenen Komponenten:

```
MinSpanTree (V, E, w)
    MinSPT ← ∅
    regions ← { {v} | v ∈ V }
    Esorted ← Sort(E, w)
    for (x, y) ← ESorted
        regionOfx ← FindSetOf(x, regions)
        regionOfy ← FindSetOf(y, regions)
        if regionOfx ≠ regionOfy
            MinSPT ← result ∪ { (x, y) }
            regions ← Unite(regions, regionOfx, regionOfy)
    return MinSPT
```

Mit der Datenstruktur *Disjoint Set* von oben wird dies zu einem effizienten Algorithmus zur Bestimmung des minimalen Spannbaums. Es beginnt mit einer Sortierung der Kanten. Mit einem guten Sortieralgorithmus kann das in $\mathcal{O}(m \log m)$ erledigt werden, wobei m die Zahl der Knoten ist. Die nachfolgenden Aktionen entsprechen denen beim Finden der verbundenen Komponenten. Insgesamt haben wir dann eine Laufzeit in $\mathcal{O}(m \log m + m \log n)$. Wegen $m \leq n^2$ (maximal $n(n-1)/2$ Kanten bei n Knoten) ist $\log m \leq 2 \log n$ und darum auch $m \log m + m \log n \leq 3m \log m$. Die 3 kann in der $\mathcal{O}$-Abschätzung ignoriert werden. Damit kommen wir insgesamt auf eine Laufzeit in $\mathcal{O}(m \log n)$.

Der minimale Spannbaum hat sich so als einfache Variante der Suche nach zusammenhängenden Komponenten herausgestellt. Es hilft also, ein paar Algorithmen zu kennen. Algorithmen können mit viel Intuition entwickelt werden. Ein wenig algorithmische Technik sollte dazu beigemischt werden.

Neben dem Problem geht es auch immer darum, welche Daten wir zur Verfügung haben. Es macht beispielsweise schon einen Unterschied, ob die Knoten und eventuell die Kanten erst mühsam durch das Ablaufen des Graphen aufgespürt werden müssen oder ob sie sofort in einer Übersicht vorliegen.

Ganz allgemein empfiehlt es sich, mit abstrakten Konzepten zu beginnen, mit Mengen, Abbildungen und so weiter, die das Wesen der zur Verfügung stehenden Informationen und ihre Verarbeitung klar zum Ausdruck bringen. Diese abstrakte Version des Algorithmus kann mit einer modernen Programmiersprache oft schon leicht umgesetzt und getestet werden.

Für den Einsatz in der »Produktion« ist diese Version in der Regel zu ineffizient. Da müssen die Datenstrukturen definiert werden, welche die für den Algorithmus benötigten Informationen effizient zugreifbar machen:

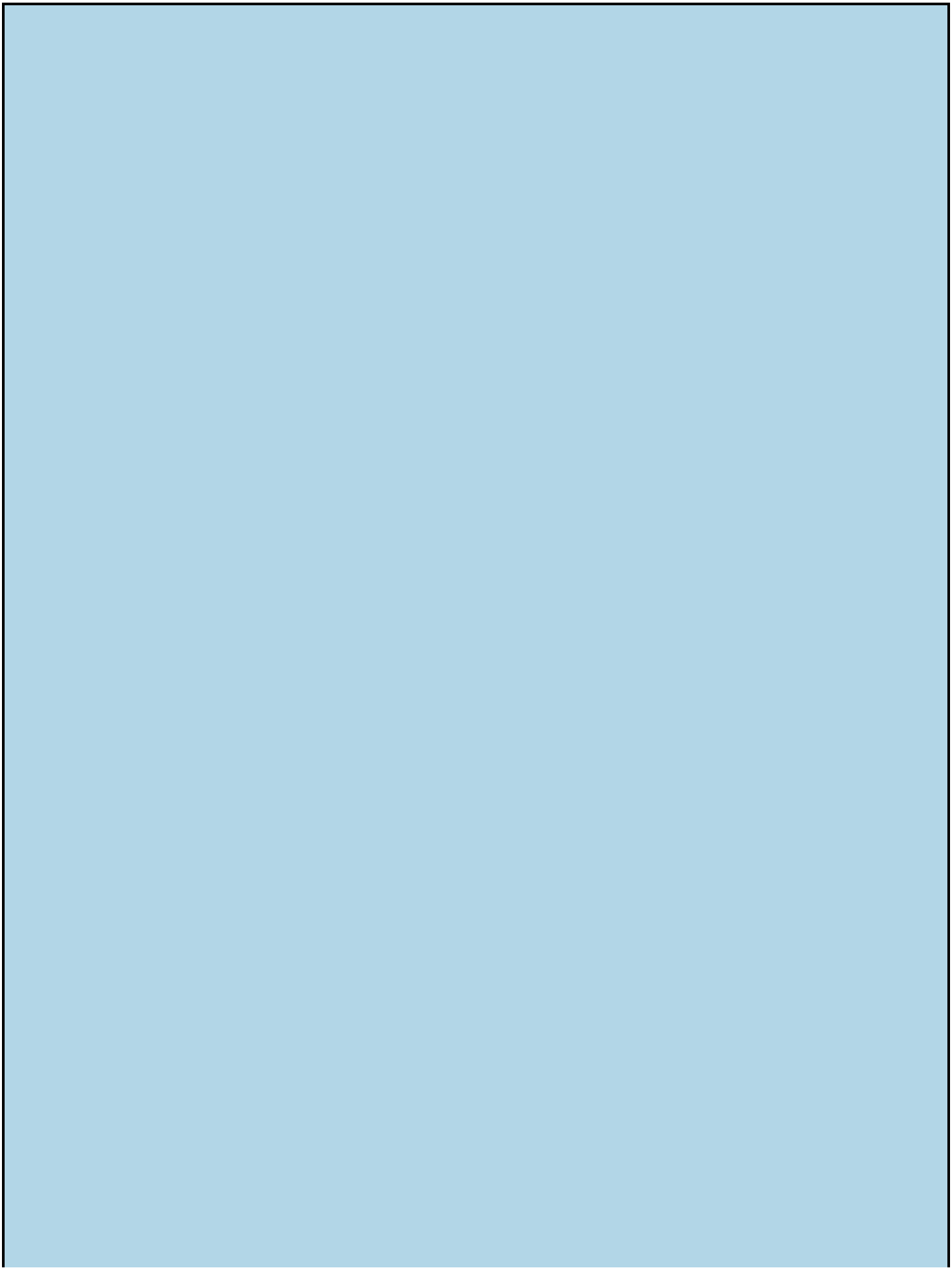

Problem + Daten

- ⇒ abstrakter Algorithmus, der auf Informationen agiert
- ⇒ konkreter Algorithmus, der auf Datenstrukturen agiert.

Teil IV

Algorithmische Techniken


MIR WÄRE ES WIRKLICH LIEBER, DU WÜRDEST EINE ANDERE SUCHMETHODE ALS BRUTE FORCE BENUTZEN!
CHRISTIAN KALKERT



IN DIESEM TEIL …

Denk- und Lösungsansätze bei der Entwicklung von Algorithmen

Erschöpfende Suche: Systematisches Durchsuchen aller Kandidaten

Teilen und Herrschen: Lösungen von einfachen Problemen zu Lösungen komplexerer Probleme ausbauen

Dynamisches Programmieren: Teillösungen systematisch tabellieren und nutzen

Techniken für Näherungslösungen: Heuristiken, Approximationen, Gier

Kapitel 11

Probleme totschlagen

IN DIESEM KAPITEL

- Rohe Kraft
- Erschöpfende Suche
- Backtracking
- Verzweigen und Begrenzen

Wer kennt das nicht. Manchmal hat man einfach keine gute Idee, wie ein Problem zu lösen ist, oder auch einfach keine Zeit, länger über eine mögliche Lösung nachzudenken. Man möchte morgens Milch in seinen Kaffee schütten, aber die neue Milchpackung hat nicht mehr die Lasche, an der man sonst einfach gezogen hat, um sie zu öffnen. Mist. Zu früher Stunde hat man weder Lust noch Zeit lange nachzudenken. – Was ist die Definition von *offen*? Nun, es bedeutet *mit einer Öffnung*. Also Messer raus, reingestochen, Loch drin, offen: Problem gelöst. Mit *roher Kraft* wurde die Definition der Lösung auf die einfachstmögliche Art in einen Algorithmus umgesetzt.

Milchpackungen mögen unter Umständen eine Herausforderung für die Problemlösungskompetenz darstellen, in der Regel ist das Problem aber auch ohne den Einsatz von roher Kraft beherrschbar. Andere Probleme sind da aber schon von anderem Kaliber. Ein wichtiger Anlass, der ansteht, oder auch nur der alltägliche Überlebenskampf, und wir suchen nach dem passenden Outfit. Für dieses Problem gibt es keine Lasche, an der man ziehen kann, oder sonst irgendetwas Systematisches. Wir müssen mühsam jede Kombination an Kleidungsstücken ausprobieren. Eine im wahrsten Sinne des Wortes *erschöpfende Suche*.

Doch halt. Gelegentlich stellt sich heraus, dass eine Teillösung nie und nimmer zu einer Gesamtlösung ausgebaut werden kann. Die schweren Bergschuhe zusammen mit den grünen Leggings können wir beispielsweise kombinieren mit was immer im Kleiderschrank zu finden sein mag, ein passendes Outfit für den Schulabschlussball wird damit nicht zu konstruieren sein. Legen wir sie also wieder zurück. Das ist die Methode *Backtracking*, bei der auf der Suche nach einer Lösung offenkundig ungeeignete Teillösungen frühzeitig verworfen werden. Also werden die Bergschuhe wieder ausgezogen und zurückgestellt (Backtrack) und es geht mit dem nächsten Paar Schuhe, den Ballerinas, weiter.

Manchmal wird auch nicht irgendeine Lösung gesucht, sondern die beste. Man spricht dann von einem *Optimierungsproblem*. Viele Probleme sind Optimierungsprobleme: Suche die beste Geldanlage, die am besten passende Hose und so weiter. Die Suche nach einer optimalen Lösung kann mit einer Variante des Backtrackings optimiert werden: Die gerade angefangene Konstruktion einer Lösung kann abgebrochen werden, wenn klar ist, dass sie zu nichts Besserem führen wird, als zu etwas, das man schon hat. Diese Strategie nennt sich *Verzweigen und Begrenzen*. Rosa Leggings und Wanderstiefel werden, egal womit ich sie kombiniere, beispielsweise zu einem schlechteren Ergebnis führen als die Wanderstiefel mit den grünen Leggings und der Trachtenjacke. Die Suche nach einem Oberteil für diese schräge Kombi kann ich mir also komplett sparen.

Erschöpfende Suche

Ob es um den Partner für's Leben oder einen vermissten Schlüssel geht: Die Lösung mancher Probleme kann nur durch ein mehr oder weniger systematisches Durchsuchen aller Möglichkeiten gefunden werden. Die Suche ist oft nicht ganz so einfach, wie es zunächst scheinen mag, vor allem, wenn sie in absehbarer Zeit zum Erfolg führen soll. Es hilft aber immer, sich auf ein paar nützliche Prinzipien und Strategien zu besinnen.

Das Wichtigste bei einer Suche ist zu wissen, wonach man sucht. Bei der Suche nach dem Schlüssel ist die Sache klar, ich suche den Ort, an dem der Schlüssel liegt. In anderen Fällen ist das nicht ganz so einfach. Da müssen eventuell komplexere Kriterien definiert werden, denen das Gesuchte genügen muss. Manch eine bereut es beispielsweise irgendwann, bei der Partnersuche auf eine ausreichend vollständige Spezifikation der gewünschten Lösung verzichtet zu haben.

Dann muss man natürlich auch wissen, wo gesucht werden muss. Das ist der *Suchraum*. Kennt man den Suchraum, dann muss dieser schließlich noch systematisch und vollständig abgelaufenen und untersucht werden. Auch das ist nicht immer trivial. Bei der Suche nach dem Schlüssel wird schnell mal die Ecke übersehen, an der er liegt, und stattdessen zehnmal an der Stelle nachgeschaut, an der er liegen sollte, aber auch beim zehnten Nachschauen nicht zum Vorschein kommt.


Prinzip und Strategie der *erschöpfenden Suche*:

- Was ist der *Suchraum*? – Definiere die Menge der Dinge die als Lösung des Problems in Frage kommen.
- Was ist eine *Lösung*? – Definiere die Objekte des Suchraums, die als Lösung des Problems akzeptabel sind.
- Organisiere die *Suche*! – Definiere ein Verfahren, das die Elemente des Suchraums aufzählt.

Bei der Suche nach einem verlegten Schlüssel besteht der Suchraum vielleicht aus allen Orten der Wohnung und der Garage. Die Lösung ist der genaue Platz, an dem der vermisste Schlüssel greifbar ist. Das mit dem Suchraum ist immer etwas problematisch. Ist er zu groß, dann dauert die Suche zu lange (siehe Abbildung 11.1). Ist er zu klein, dann ist es unter Umständen zu schwierig, ihn korrekt zu bestimmen. Man kann nicht das Universum nach einem verlegten Schlüssel durchsuchen, und der Ratschlag, nur genau da zu suchen, wo er liegt, ist auch nicht so richtig konstruktiv. Irgendetwas dazwischen ist ein geeigneter Suchraum: groß genug, um das Gesuchte zu enthalten, aber klein genug, um einfach bestimmt werden zu können.

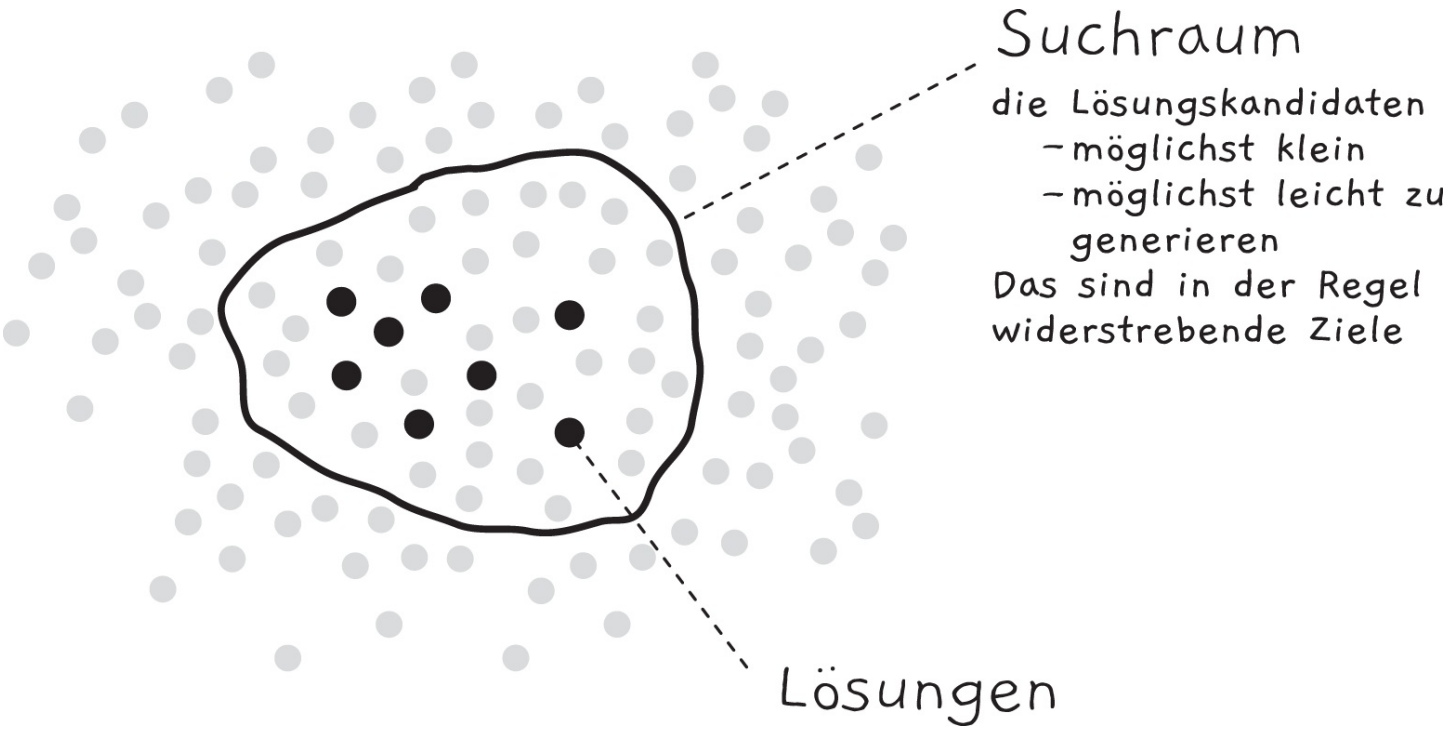


Abbildung 11.1: Suchraum: so klein und einfach zu beschreiben wie möglich

Weiß man, wo und nach was gesucht werden soll, dann muss die Suche noch organisiert werden: Jedes Element des Suchraums muss mindestens einmal und am besten auch nur genau einmal angeschaut werden. Keines darf ausgelassen werden und es sollte auch nicht immer wieder das Gleiche untersucht werden. Insgesamt:


Der Suchraum sollte

- ✔ möglichst klein,
- ✔ möglichst leicht zu bestimmen und
- ✔ möglichst leicht zu durchlaufen sein.

Im Folgenden betrachten wir ein paar Strategien, mit denen man dies konkret erreichen kann.

Die üblichen Verdächtigen: Kombinatorische Objekte

Erschöpfende Suche wird verwendet, wenn einem ansonsten nichts einfällt. »Verhaften Sie die üblichen Verdächtigen!« ist eine etwas plumpe, aber eventuell erfolgversprechende Anwendung der Strategie *erschöpfende Suche* in der Kriminalistik. Man schnappt sich die, die man sich immer schnappt, und verhört sie so lange, bis einer gesteht. Natürlich muss man dazu seine Pappenheimer kennen. Mit etwas Erfahrung ist aber oft klar, wer da in Frage kommt.

In der Welt der Algorithmen gibt es sie auch, die üblichen Verdächtigen. Sie werden *kombinatorische Objekte* genannt. Es sind übliche Kollektionen an Kandidaten, denen man eventuell etwas nachweisen kann. So wie ein guter Kriminalist die schrägen Vögel der Stadt kennen sollte, sollte der Algorithmiker seine üblichen kombinatorischen Objekte kennen.


Der Suchraum für eine erschöpfende Suche besteht sehr oft aus *kombinatorischen Objekten*. Es lohnt sich, mal einen genaueren Blick auf sie zu werfen.

Ihr Name ist Programm. Kombinatorische Objekte sind Kombinationen von Dingen. Beispielsweise sind bilden vom Lottospielen bekannten *6 aus 49*, also die Menge aller Kombinationen von 6 Elementen aus 49 verschiedenen, ein *kombinatorisches Objekt*. Das mathematische Gebiet der Kombinatorik beschäftigt sich mit so etwas. Die Algorithmen bewegen sich in Modellwelten. Suchalgorithmen bewegen sich in modellierten Suchräumen. Die Kombinatorik hat viele der potenziellen Suchräume in Modellwelten analysiert.


Kombinatorische Objekte sind Kollektionen, wie sie in der Kombinatorik (einem Teilgebiet der diskreten Mathematik) beschrieben werden.

Die üblichen Verdächtigen bei einer erschöpfenden Suche ergeben sich oft aus dem Durchlaufen, den *Traversierungen*) einer Datenstruktur. Alle Elemente einer Liste oder eines Arrays. Alle Knoten oder alle Kanten in einem Graphen. Das sind Beispiele für einfache kombinatorische Objekte: Mengen, die »irgendwie« gegeben sind. Ihre Erzeugung hängt nur von der Implementierung der Datenstruktur ab. Gelegentlich wird es aber etwas komplexer. Permutationen, also die Menge der Umsortierungen einer Folge, sind ein Beispiel für komplexere kombinatorische Objekte.

Einige wichtige kombinatorische Objekte sind:

- ✔ eine vorgegebene Menge,
- ✔ die Menge der Permutationen einer Sequenz,
- ✔ die Menge der Teilsequenzen einer Sequenz,
- ✔ die Menge der Teilmengen einer Menge,

- die Menge der Kombinationen von n Elementen aus einer Menge mit m Elementen, und zwar jeweils mit oder ohne Wiederholung,
- alle möglichen Aufteilungen einer Menge in Teilmengen (Partitionen einer Menge).

Für einen Algorithmus, der für eine erschöpfende Suche in einem Suchraum herumschnüffelt, sind zwei Dinge wichtig:

- Wie wird der Suchraum erzeugt?
- Wie viele Elemente hat er?

Die Zahl der Elemente ist wichtig für die Laufzeit. Die Frage ist meist schnell beantwortet. Die Mathematiker haben so etwas genauestens erforscht. Wir müssen es nur nachschlagen.



Die Zahl der Elemente bei einer *Auswahl* von k Elementen aus einer Menge mit n Elementen ist:

Wiederholung	*Reihenfolge relevant*	*Reihenfolge nicht relevant*
erlaubt	n^k	$C_{n-1}^{k+n-1} = \binom{k+n-1}{n-1}$
nicht erlaubt	$P_k^n = \frac{n!}{(n-k)!}$	$C_k^n = \binom{n}{k}$

Dabei ist

$$C_k^n = \binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} = \frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}{k \cdot (k-1) \cdot \ldots \cdot 1}$$

der sogenannte Binomialkoeffizient. Damit haben wir immerhin schon mal die Zahl der Möglichkeiten. Deren Erzeugung ist dann noch einmal etwas anderes: ein eigenständiges algorithmischen Problem.

Lässt man hunderttausend Affen hunderttausend Jahre auf hunderttausend Tastaturen herumhämmern, dann schreibt einer von ihnen vielleicht einen bedeutenden Roman. Ok, so viel Zeit, Affen und Computer haben wir nicht. Die Idee ist aber klar: Auch das tollste Buch ist letztlich nichts anderes als eine von vielen möglichen Aneinanderreihungen von Buchstaben und Zeichen und mit einer erschöpfenden Suche kann man irgendwann schriftstellerische Erfolge feiern. – Irgendwann!

Nehmen wir also mal an, wir seien beim Sortieren so wenig inspiriert wie beim Schriftstellern und müssten nach Art der Affen sortieren. – Sorry, liebe Affen, natürlich seid ihr eigentlich viel intelligenter. Für das Sortieren einer Kollektion von Elementen gibt es viele schlaue Algorithmen. Aber zur Abwechselung wollen wir hier rohe Kraft

anwenden. Einfach so, oder um die CPU ein wenig aufzuwärmen oder um uns ein wenig affig zu fühlen. Sortieren mit roher Kraft geht so:

```
PermutationSort(a)
    for candidate ← Permutations(a)
        if isSorted(candidate)
            return candidate
```

Dabei ist a die Sequenz, die sortiert werden soll. Wir generieren einfach alle möglichen Permutationen (Umgruppierungen) von a und schauen jeweils, ob sie sortiert ist. Wenn ja, dann ist die Sortierung fertig. Wenn nein, dann geht es weiter.

Für eine Abschätzung des Aufwands ist es natürlich zunächst einmal interessant zu wissen, wie groß der Suchraum ist, also wie viele Permutationen einer Sequenz es gibt. Die Permutationen sind eine Auswahl von n aus n ohne Wiederholung mit Beachtung der Reihenfolge. Ein Blick auf den Spickzettel zeigt, wie viele das sind:

$$P_n^n = \frac{n!}{(n-n)!} = \frac{n!}{0!} = \frac{n!}{1} = n!$$

Das sind ziemlich viele. Vor allem wenn n groß ist. Und eine Idee, wie wir sie mit welchem Aufwand erzeugen könnten, haben wir auch noch nicht. Bleiben wir vielleicht doch besser bei den intelligenten Sortierverfahren.

Konzentrierte oder weit ausschweifende Suche

Im Fall der Sortierung ist der Suchraum ziemlich klar. Es gibt da nicht viel zu überlegen. So einfach ist es aber nicht immer. Man kann den Suchraum zu groß wählen, dann ist es meist einfacher, ihn zu erzeugen, aber die Suche dauert lange – oder man macht ihn klein, dann geht die Suche schnell, aber der Suchraum kann nicht so leicht erzeugt werden. Klar: Warum wird der Schlüssel nicht einfach nur da gesucht, wo er liegt?

Betrachten wir dazu ein Beispiel: Die berühmten n Damen auf einem Schachbrett. Bei diesem Problem geht es darum, n Damen auf einem Schachbrett der Größe n so zu platzieren, dass keine irgendeine andere schlägt. Gesucht ist also ein Algorithmus, der n Positionen in einer ($n \times n$)-Matrix so auswählt, dass keine zwei oder mehr Positionen in der gleichen Zeile, gleichen Spalte oder gleichen Diagonalen liegen (siehe Abbildung 11.2).

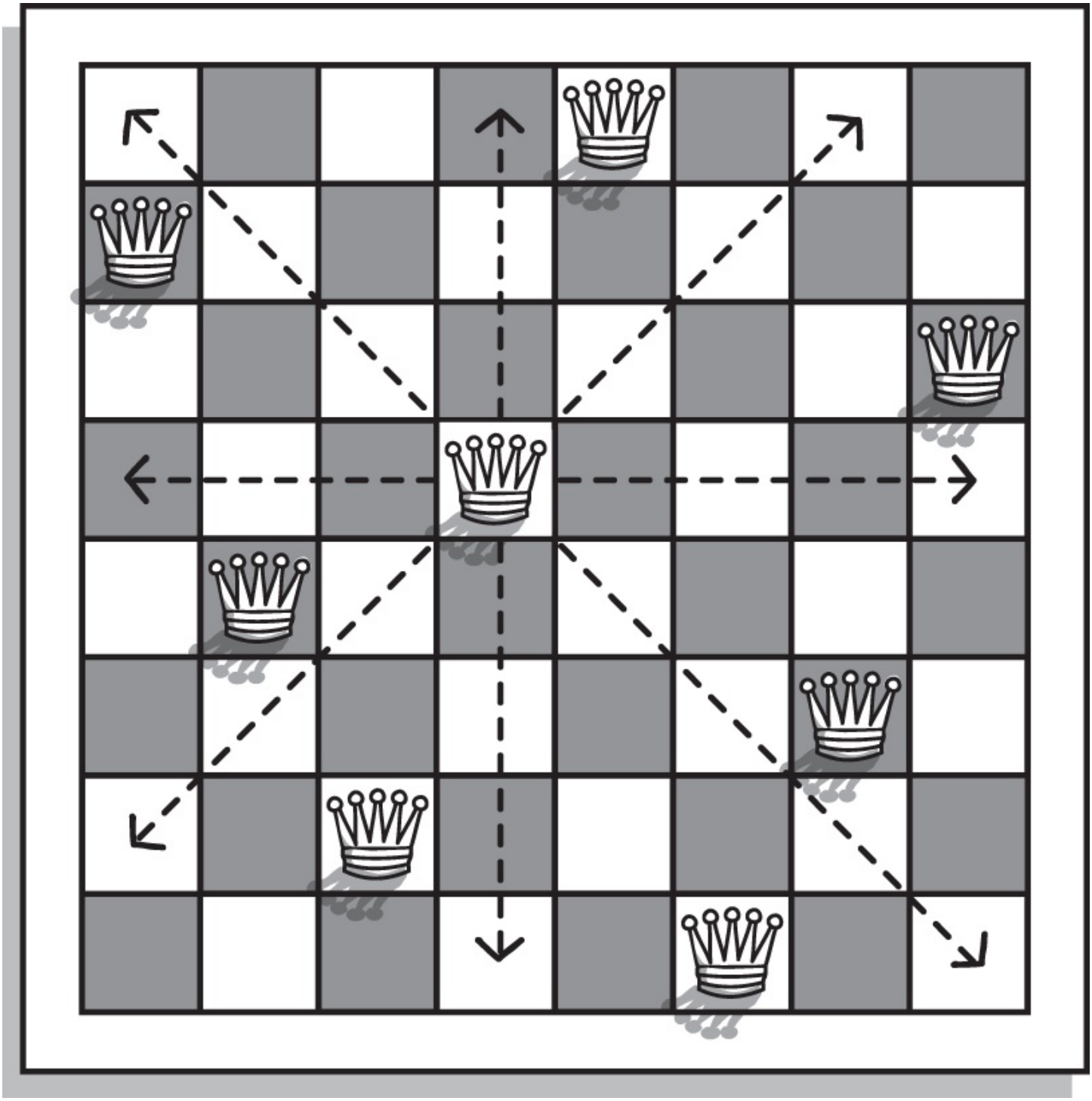


Abbildung 11.2: Acht friedliche Damen, die sich nicht gegenseitig bedrohen

Als Erstes halten wir fest, dass es für die Damen keine einfache Reduzieren-und-Herrschen-Lösung gibt: Eine Lösung des 7-Damen-Problems kann *nicht* zu einer Lösung des 8-Damen-Problems erweitert werden. Das größere Problem ist nicht die Erweiterung eines kleineren Problems der gleichen Art, sondern die Erweiterung eines Problems von leicht anderer Art: Eine Lösung des 8-Damen-Problems ist eine Erweiterung des Problems

*7 Damen auf den ersten 7 Zeilen eines (**8** × **8**)-Feldes.*

Es ist *nicht* eine Erweiterung der Lösung des kleineren Problems der *gleichen* Art

7 Damen auf einem (7 × 7)-Feld.

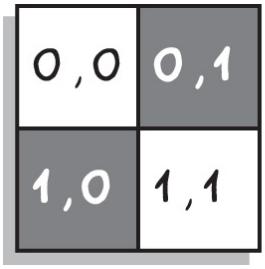
Das kann man lösen, aber die Lösung kann nicht in einfacher Art zu einer Lösung des (8 × 8)-Problems ausgebaut werden. Es gibt natürlich kein absolutes und allgemeingültiges Kriterium für »in einfacher Art«, aber wenn das größere Problem zusätzliche Beschränkungen und Bedingungen an die Lösungen des kleineren stellt, dann ist Teilen und Herrschen eher keine erfolgversprechende Lösungsstrategie.

Wie auch immer. Wir stellen fest, dass wir keine schlaue Idee haben. Schlaue Ideen sind ja meistens Ideen zum Reduzieren/Teilen und Herrschen. Gier hilft auch manchmal, hier aber nicht. Es bleibt nur die rohe Kraft in Form einer erschöpfenden Suche.

Bei den n Damen handelt es sich um ein Auswahlproblem. Probleme wie »6 aus 49« sind *Auswahlprobleme*. *Sechs* Richtige aus 49 Möglichen ist das Gleiche wie *ein* Richtiger aus den $\binom{49}{6}$ möglichen 6er-Kombinationen.

Von allen möglichen $n \cdot n$ Positionen auf dem Feld müssen n so ausgewählt werden, dass sie die Bedingung erfüllen. Angenommen, eine Position wird durch ihre Zeile x und Spalte y bestimmt, dann muss aus den n^2 Listen möglicher Kombinationen von Positionen, die den Suchraum bilden, eine Liste von n Positionen gefunden werden, die die Bedingung erfüllen.

Nehmen wir mal ein kleines (2×2)-Brett mit 4 Positionen (siehe Abbildung 11.3). Es gibt zwar keine Lösung des 2-Damen-Problems, aber der Suchraum ist angenehm klein und überschaubar. Er hat 16 Elemente: »2 aus 4« *mit* Wiederholung ergibt $4^2 = 16$ Möglichkeiten 2 Positionen auszuwählen. Das ist für ein so kleines Feld schon ziemlich viel. Bei einem (8×8)-Feld gibt es 64 verschiedene Positionen, von denen auf $64^8 \approx 2,8 \times 10^{14}$ verschiedene Arten 8 ausgewählt werden können.



[[(0,0)(0,0)], [(0,0)(0,1)], [(0,0)(1,0)], [(0,0)(1,1)],
[(0,1)(0,0)], [(0,1)(0,1)], [(0,1)(1,0)], [(0,1)(1,1)],
[(1,0)(0,0)], [(1,0)(0,1)], [(1,0)(1,0)], [(1,0)(1,1)],
[(1,1)(0,0)], [(1,1)(0,1)], [(1,1)(1,0)], [(1,1)(1,1)]]

Abbildung 11.3: Suchraum für eine Auswahl von 2 von 16 möglichen Positionen

Das ist zu viel. Der Suchraum muss kleiner werden. Ein paar einfache Beobachtungen helfen da weiter:

- ✔ Selbstverständlich darf nicht mehr als eine Dame auf einem Platz stehen.
- ✔ Die Reihenfolge der Positionen spielt keine Rolle.

Der Suchraum kann damit reduziert werden, und zwar auf eine Untersuchung einer

Auswahl von n aus n^2 ohne Wiederholung und ohne Berücksichtigung der Reihenfolge.

Die Größe des Suchraums ist aber nicht das alleinige Kriterium. Es darf auch nicht zu schwierig sein, ihn zu erzeugen. Die 4 möglichen Positionen auf einen (2×2)-Brett sind leicht zu erzeugen:

```
TwoQPositions                    erzeugt alle möglichen Positionen in einen (2 x 2)-Feld
    result ← Nil
    for i ← 0, 1
        for j ← 0, 1
            result ← result + [ (i,j) ]
    return result
```

Das sind die Positionen $[(0,0),(0,1),(1,0,(1,1)]$. Von diesen müssen zwei passende ausgewählt werden. Die entsprechenden Möglichkeiten könnten ebenfalls leicht mit einer geschachtelten Schleife berechnet werden:

```
TwoQSearchSpace                  erzeugt alle möglichen Paare von Positionen
    result ← Nil                 in einen (2 x 2)-Feld
    for p0 ← TwoQPositions()
        for p1 ← TwoQPositions()
            result ← result + [ (p0,p1) ]
    return result
```

Das ergibt die 16 möglichen Arten, 2 Damen auf ein (2 x 2)-Brett zu stellen:

((0,0),(0,0)),	((0,0),(0,1)),	((0,0),(1,0)),	((0,0),(1,1)),
((0,1),(0,0)),	((0,1),(0,1)),	((0,1),(1,0)),	((0,1),(1,1)),
((1,0),(0,0)),	((1,0),(0,1)),	((1,0),(1,0)),	((1,0),(1,1)),
((1,1),(0,0)),	((1,1),(0,1)),	((1,1),(1,0)),	((1,1),(1,1)).

Da stehen natürlich auch schon mal zwei auf einem Platz. Dieser Fall lässt sich leicht eliminieren:

```
TwoQBoards                       alle Möglichkeiten der Auswahl von
    result ← Nil                 2 Positionen ohne Wiederholung
    for p0 ← TwoQPositions()
        for p1 ← TwoQPositions()
            if p1 ≠ p2           nicht 2 auf einem Platz
                result ← result + [ (p0,p1) ]
    return result
```

Das ergibt den Suchraum n aus n^2 ohne Wiederholung, mit

$$\binom{n \cdot n}{n}$$

Elementen. Bei $n = 2$ sind das 12 mögliche Kombinationen:

((0,0),(0,1)),	((0,0),(1,0)),	((0,0),(1,1)),
((0,1),(0,0)),	((0,1),(1,0)),	((0,1),(1,1)),
((1,0),(0,0)),	((1,0),(0,1)),	((1,0),(1,1)),
((1,1),(0,0)),	((1,1),(0,1)),	((1,1),(1,0)).

Jetzt sind die Kombinationen ausgeschlossen, bei denen zwei Damen an einem Platz stehen. Geht es noch kleiner? Im Lösungsraum oben wird noch die Reihenfolge unterschieden, in der die zwei Damen auf das Feld gestellt werden. So sind sowohl ((0,0), (1,1)) als auch ((1,1),(0,0)) Bestandteile des Suchraums, obwohl beide die die gleiche Situation auf dem Feld beschreiben. Können derartige Duplikate auch eliminiert werden? Und wenn's geht, auf einfache Art?

Klar geht das, wir fügen nur die hinzu, die nicht schon in umgekehrter Reihenfolge drin sind:

```
TwoQBoards                          alle Möglichkeiten der Auswahl von Positionspaaren
    result ← Nil                    ohne Wiederholung und Symmetrie
    for p0 ← TwoQPositions()
        for p1 ← TwoQPositions()
            if p0 ≠ p1 and (p1,p0) ∉ result
                result ← result + [(p0,p1)]
        return result
```

Damit können wir den Suchraum reduzieren auf die 6 Elemente

$$\binom{n \cdot n}{n} = \binom{4}{2} = (4 \cdot 3)/2 = 6$$

einer Auswahl von 2 aus 4 ohne Wiederholung (siehe Abbildung 11.4):

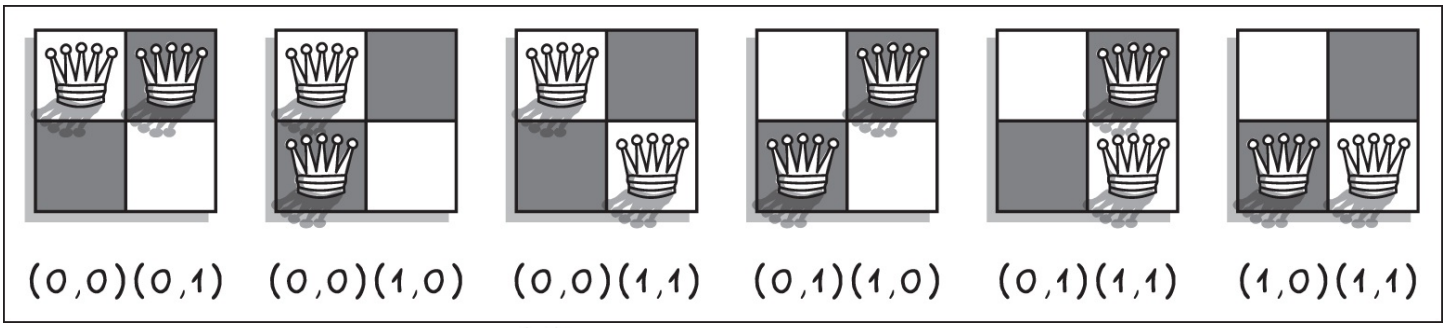


Abbildung 11.4: Suchraum der Größe $\binom{n}{k}$ beim 2-Dame-Problem

((0,0),(0,1)),	((0,0),(1,0)),	((0,0),(1,1)),
	((0,1),(1,0)),	((0,1),(1,1)),
		((1,0),(1,1)).

Wenn wir jetzt noch alle Konfigurationen entfernen, bei denen zwei Damen in einer Zeile, einer Spalte oder einer Diagonalen sind, dann ist der Suchraum auf die Lösung zusammengeschrumpft.

Wir haben die Wahl, weiteren Aufwand in die Verkleinerung des Suchraums zu stecken oder in die Suche selbst. Da gibt es keine scharfe Grenze. Solange es gelingt, einen verkleinerten Suchraum zu erzeugen, sollte verkleinert werden. Ein gutes Kriterium ist immer die Frage, ob auch der verkleinerte Suchraum ein kombinatorisches Objekt darstellt.



Wenn ein Suchraum als kombinatorisches Objekt definiert werden kann (Permutationen, Partitionen einer Menge, Auswahlproblem etc.), dann gibt es mit hoher Wahrscheinlichkeit bekannte Standardalgorithmen, mit denen der Suchraum effizient erzeugt werden kann.

Gehen wir in diesem Sinne noch einen Schritt weiter und sehen uns die Konfigurationen an, bei denen in jeder Zeile genau eine Dame steht. Handelt es sich dann auch noch um ein einfaches kombinatorisches Objekt und wenn ja, um welches?

Da in jeder Zeile genau eine Dame platziert wird, kann die Zeilennummer weggelassen werden. Der Suchraum besteht jetzt aus allen Folge der Länge n von Spaltennummern. Davon gibt es n^n. Bei $n = 2$ sind das vier.

Um welche kombinatorischen Objekte es sich dabei handelt, sieht man leicht an Abbildung 11.5: Es sind die Tupel der Länge n von n verschiedenen Werten.

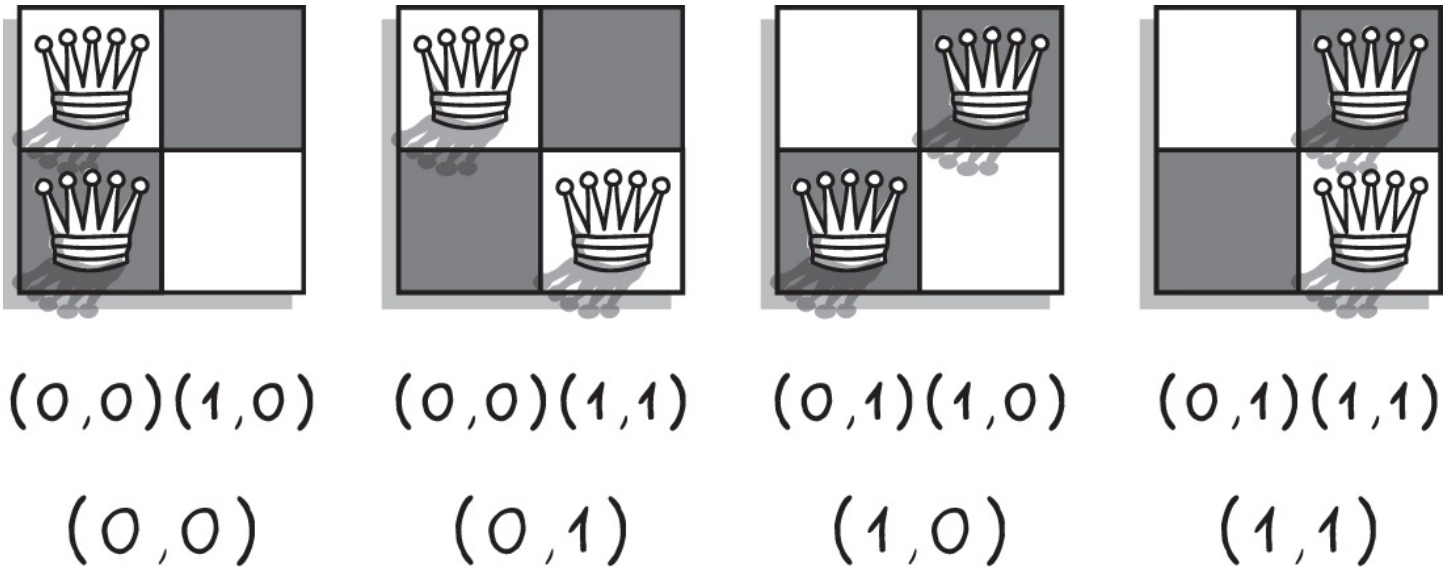


Abbildung 11.5: Suchraum der Größe n^n beim 2-Dame-Problem

Deren Erzeugung ist sehr einfach: Mit n geschachtelten Schleifen können alle n-Tupel erzeugt werden, für $n = 2$ beispielsweise mit zwei geschachtelten Schleifen:

```
TwoQPositions
    result ← Nil                    alle Möglichkeiten der Auswahl von Positionspaaren
    for p0 ← 0 .. 1                 ohne Wiederholung und ohne zwei in einer Zeile
        for p1 ← 0 .. 1             als Paare von Spaltenpositionen
            result ← result + [ (p0, p1) ]
    return result
```

Die erschöpfende Suche nach acht friedlichen Damen

Der Suchraum ist jetzt stark reduziert, auch wenn er im Allgemeinen immer noch eine beachtlich große Zahl an Kandidaten enthält. Jetzt müssen seine Bewohner noch begutachtet werden:

```
TwoQueens                           erzeuge den Suchraum und prüfe seine Elemente
    for board ← TwoQPositions()
        if Ok(board)
            return board
```

Oder etwas kompakter:

```
TwoQueens                           erzeuge den Suchraum und prüfe seine Elemente
    for p0 ← 0 .. 1
        for p1 ← 0 .. 1
            if Ok([p0, p1]) return [p0, p1]
```

Ein $board$ ist eine Folge von Spaltenpositionen und repräsentiert ein Brett, in dem in Zeile i eine Dame an Position $board[i]$ steht. Es ist nur dann OK, wenn alle Werte unterschiedlich sind. Dann gibt es keine zwei Damen in der gleichen Spalte.

Es darf auch keine zwei Damen in der gleichen Diagonalen geben. Das ist etwas kniffliger. Zwei Damen sind in einer Diagonalen, wenn sie zeilenweise so weit von einander entfernt sind wie spaltenweise. Nehmen wir mal eine (5×5)-Matrix und betrachten das Feld $3, 3$ und seine Diagonalen:

1,1	1, 2	1, 3	1, 4	**1,5**
2, 1	**2,2**	2, 3	**2,4**	2, 5
3, 1	3, 2	**3,3**	3, 4	3, 5
4, 1	**4,2**	4, 3	**4,4**	4, 5
5,1	5, 2	5, 3	5, 4	**5,5**

In der Diagonalen von $3, 3$ sind in der zweiten und dritten Zeile (zeilenweise Entfernung 1) jeweils die Spalten 2 und 4 (spaltenweise Entfernung 1). $3, 3$ bedroht darum $2, 2$ und $2, 4$ sowie $5, 2$ und $5, 4$. In der kompakten Speicherung, mit einem Wert pro Zeile, wäre dann beispielsweise $1, 2, 3, 2, 1$ wegen mehrfacher Verstöße gegen das Diagonalenverbot auszusortieren. Ebenso auch $5, 4, 3, 4, 5$.

Der Algorithmus Ok zur Prüfung eines Bretts mit seiner Damenbelegung ist somit:

```
Ok(board)                                  ist die Belegung konfliktfrei?
    for i ← 0 .. board.length−1,           für jede Position i
        for j ← i + 1 .. board.length −1   für jede Position j hinter i
            x ← board[i]                   Position in Zeile i
            y ← board[j]                   Position in Zeile j
            d ← j − i                      Zeilendifferenz der beiden Damen
            if x = y or                    gleiche Spalte
                y = x − d or y = x + d     gleiche Diagonale (Zeilendifferenz = Spaltendifferenz)
                return false
    return true
```

Für ein (8×8)-Brett finden wir dann mit ein paar geschachtelten Schleifen

```
EightQueens
    for p0 ← 0 .. 7
        for p1 ← 0 .. 7
            for p2 ← 0 .. 7
                for p3 ← 0 .. 7
                    for p4 ← 0 .. 7
                        for p5 ← 0 .. 7
                            for p6 ← 0 .. 7
                                for p7 ← 0 .. 7
                                    board ← [p0, p1, p2, p3, p4, p5, p6, p7]
                                    if Ok(board) return board
```

schnell eine der 92 Lösungen für das 8-Damen-Problem.

Iterative und rekursive Erzeugung des Suchraums

Wenn es ums Suchen geht, dann ist die Schleife unsere beste Freundin. Nur in Notfällen werden wir ihr untreu und wenden uns der spröden Rekursion zu.



Es geht nichts über geschachtelte Schleifen! n geschachtelte Schleifen der Länge m sind die einfachste und schnellste Methode, einen Suchraum der Art »n aus m« zu durchlaufen.

Schleifen rekursiv erzeugen

Das mit den Schleifen funktioniert leider nur, wenn n, die Zahl der Schleifen, *statisch* bekannt ist, wenn also der Implementierter des Codes weiß, wie viele Schleifen benötigt

werden und sie darum eintippen kann. Im Pseudocode mögen ja gelegentlich noch »Pünktchen«, oft vornehmer *(Code-)Ellipsen* genannt, der folgenden Art akzeptabel sein:

```
QueensN(n)
    for p_0 ← 0 .. n − 1
      ...                                  Code-Ellipse (Pünktchen)
            for p_{n−1} ← 0 .. n − 1
                board ← [p_0, ... p_{n−1}]
                if Ok(board) return board
```

Solche Anweisungen an den Codierer sind als Teil eines Algorithmus schon etwas zweifelhaft, auch wenn klar ist, was gemeint ist und die Umsetzung auch klar ist: In realem Code wird sich jeder Compiler zurecht beschweren, wenn wir ihm mit so etwas kommen. Da müssen wir die Pünktchen schon selbst ausprogrammieren. Und im Pseudocode sind sie auch nur für die akzeptabel, die genau wissen, dass und wie so etwas in eine »pünktchenfreie Notation« umgesetzt wird.

Die Pünktchennotation in *QueensN* ist als Anleitung zur Konstruktion von Schleifen zu verstehen. Wir können sie für ein gegebenes n, beispielsweise 8, per Hand ausführen. Das ist ein einfacher händisch auszuführender Algorithmus nach der Strategie Reduzieren und Herrschen.



Reduzieren und Herrschen ist ein Spezialfall der Strategie *Teilen und Herrschen*, die in Kapitel 12 behandelt wird. Beim Reduzieren und Herrschen wird ein Problem nicht auf mehrere, sondern auf nur ein kleines Problem zurückgeführt.

Das Problem, $count$ Schleifen zu erzeugen, wird auf das Problem der Erzeugung von ($count - 1$) Schleifen reduziert, das reduzierte Problem wird gelöst und die Lösung zu einer Lösung des Problems für n ausgebaut. Kurz und einfach gesagt – um n Schleifen zu erzeugen, erzeugen wir $n - 1$ Schleifen und hängen dann noch eine dran. Etwas präziser in »Pseudo-Pseudocode«:

```
ErzeugeSchleifen(count, limit)      erzeuge count geschachtelte Schleifen der Form:
  if count = 0                      for 0 ... limit - 1
     return <<if Ok(board) return board>>
  else
     <<for x ← 0 .. limit − 1>> +
     ErzeugeSchleifen(count − 1, limit)
```

Das ist natürlich kein richtiger Algorithmus. Eher eine Anweisung an den Codierer. Es ist aber möglich, daraus einen richtigen Algorithmus zu machen: Nehmen wir wieder unsere Damen und verallgemeinern das Problem auf ein Brett mit beliebiger und erst zur Laufzeit bekannter Größe $n \times n$. Das heißt, n ist jetzt ein Eingabeparameter, keine Konstante.

Der Algorithmus ist rekursiv. Klar, wenn es ernst wird und die kleine Schleife nicht

weiterweiß, dann muss ihre große Schwester, die Rekursion ran. Der Algorithmus arbeitet mit der Akkumulatortechnik, bei der im Akkumulator das gewünschte Ergebnis aufgesammelt wird:

```
QueensN(n, limit)
    return Loops(n, n, [Nil])

Loops(count, limit, acc)              count Schleifen 0 bis limit
    if count == 0                     in acc werden die Konfigurationen (boards) gesammelt
        return [ x ∈ acc | Ok(x) ]        gib die zurück, die Ok sind
    else
        return Loops(                 n – 1 Schleifen und eine hinten dran
            count − 1,
            limit,                    acc erweitert um jeden aus acc ...
            [ l + [x] | l ∈ acc, x ← 0 ... limit − 1])      .. mit angehängtem x
```

Im Akkumulator-Parameter acc werden die zu untersuchenden Konfigurationen aufgesammelt. In jedem Rekursionsschritt wird quasi eine neue Schleife abgearbeitet, indem das bisher konstruierte und in acc aufgesammelte Zwischenergebnis um alle $0 \ \ldots \ limit - 1$ Möglichkeiten erweitert wird. Schließlich sind alle möglichen Konfiguration komplett aufgebaut und können geprüft werden (Verankerung der Rekursion).



Die Akkumulatortechnik wird in Kapitel 3 erläutert. Bei ihrem Einsatz hat eine rekursive Funktion einen Akkumulator-Parameter, in dem das Gesamtergebnis schrittweise, von rekursivem Aufruf zu rekursivem Aufruf aufgebaut wird.

In QueensN wird das Ergebnis in einem Akkumulator-Parameter aufgesammelt. Wem das zu funktional ist, der kann natürlich auch in guter alter Sitte eine globale Variable als Akkumulator verwenden. Der Listenausdruck in der Argumentliste von Loops kann dann auch in geschachtelte Schleifen aufgelöst werden:

```
QueensNI(n)                                  imperative(re) Variante
    acc ← Nil                                Sammelbecken der Ergebnisse
    Loops(n, n, [Nil])
    return acc

Loops(count, limit, board)
    if count = 0
        if Ok(board)
            acc ← acc + [board]
        else
            for x ← 0 ... limit − 1          Schleife statt Listenausdruck
                Loops(count − 1, limit, board + [x] )
```

Einen baumartigen Suchraum rekursiv erzeugen

In Kapitel 5 haben wir die Traversierung von Bäumen betrachtet. Dabei ging es darum, alle in einem Baum gespeicherten Werte aufzuzählen. Jetzt wollen wir ein ähnliches Problem betrachten: Wie können alle Pfade von der Wurzel zu einem Blatt generiert werden?

Bei einem Binärbaum hat man bei jedem Knoten die Wahl, nach links oder rechts zu gehen (siehe Abbildung 11.6). Wenn der Baum eine völlig ausbalancierte komplette Form hat, dann entsprechen diese Pfade bei einem Binärbaum der Höhe n den Möglichkeiten, n Werte aus einer Kollektion von zwei auszuwählen: es wird n -mal wird zwischen links und rechts gewählt. Das kann man natürlich gut mit geschachtelten Schleifen ausdrücken:

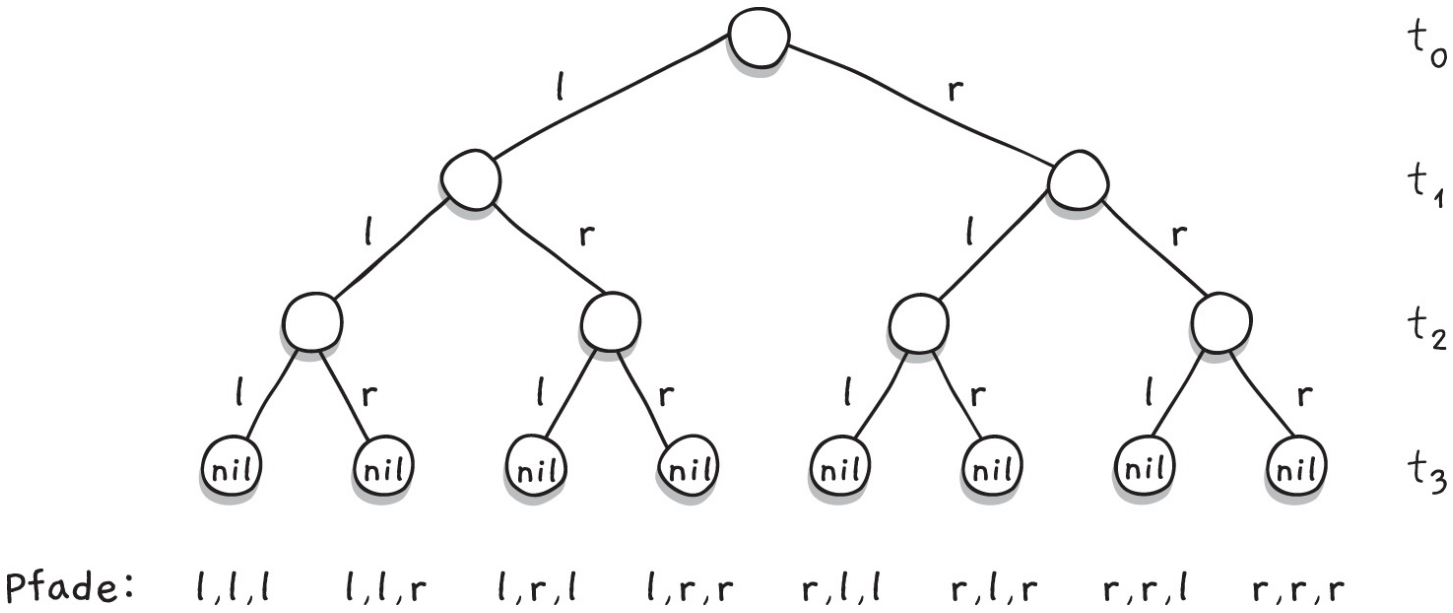


Abbildung 11.6: Suchraum als Pfade in einem Baum

```
AllPaths(t_0, n)                          alle Pfade durch einen perfekten Binärbaum der Tiefe n
    result ← Nil                          ein Pfad: eine Folgen von l-/r-Entscheidungen
    for d_1 ← [l, r]                      für jede Richtung l, r
        t_1 ← t_0.d_1                     gehe in die Richtung
        for d_2 ← [l, r]                  für jede Richtung l, r
            t_2 ← t_1.d_2                 gehe in die Richtung
            ...                           und immer so weiter
            for d_n ← [l, r]
                t_n ← t_{n-1}.d_n         jetzt sind wir ganz unten
                result ← result + [(d_1, d_2, ... d_n)]   Pfad hierher speichern
            return result
```

In jeder Schleife werden alle möglichen Richtungsentscheidungen (l und r) durchlaufen. Der Pfad wird mit diesen Richtungen erweitert und in der innersten Schleife durchläuft $(d_1,\ d_2,\ \ldots\ d_n)$ alle möglichen Pfade von der Wurzel zu einem Blatt. Das kann natürlich nicht in jeder Programmiersprache genau auf diese Weise codiert werden. Eventuell muss man die Knoten als Arrays speichern und hat dann l und r als Indizes 0 und 1.

Es gibt 2^n Möglichkeiten, eine solche Auswahl zu treffen und natürlich ist der Baum selbst bei dieser Berechnung völlig überflüssig. Die Bäume t_i werden im Algorithmus nicht benutzt.

Das sieht etwas anders aus, wenn der Baum nicht so schön gebaut ist und ein Knoten beliebiger Tiefe eventuell keinen oder nur einen Nachfolger hat (siehe Abbildung 11.7). Die möglichen Pfade in einem solchen Baum sind dann nicht mehr einfach ein kombinatorisches Objekt, für dessen Berechnung man den Baum gar nicht benötigt. Der Baum wird gebraucht, er muss die Konstruktion der Pfade steuern. Bei jedem Knoten wird geschaut, ob es noch Nachfolger gibt. Es gibt auch nicht die eine Stelle in der tiefsten Verschachtelung der Schleifen, an der ein Pfad konstruiert wird. Es kann in jeder Tiefe Schluss sein kann. Das Verfahren ist:

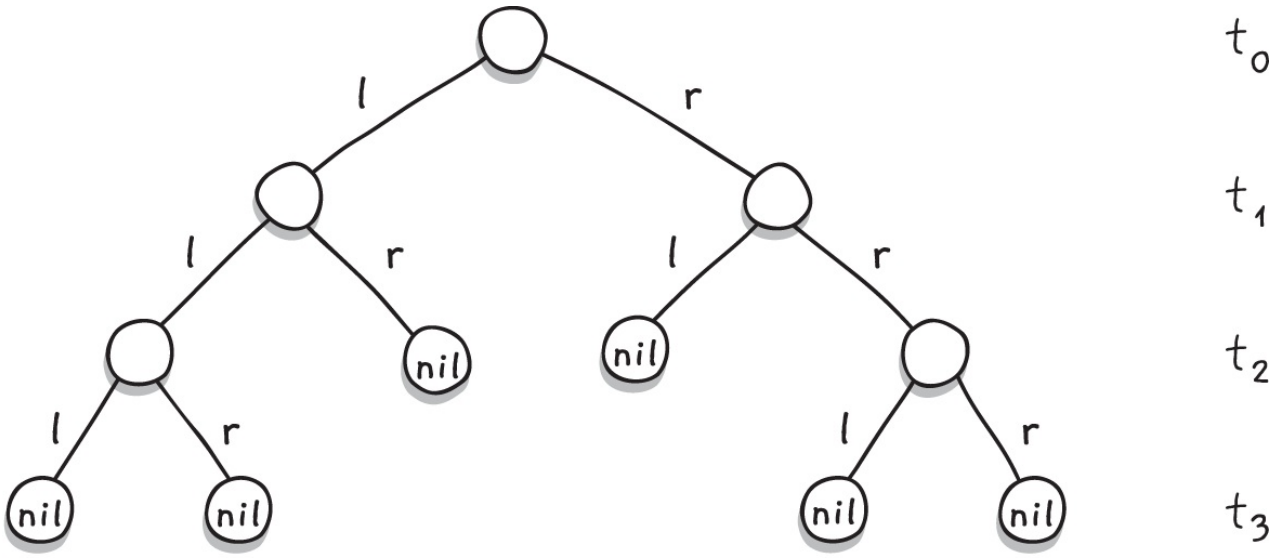


Abbildung 11.7: Suchraum als Pfade in einem nicht perfekten Baum

```
AllPaths(t0)                          alle Pfade durch einen nicht perfekten
    result ← Nil                      Binärbaum beliebiger Tiefe
    if t0 ≠ Nil
        for d1 ← [l, r]               für jede Richtung
            t1 ← t0.d1                gehe in die Richtung
            if t1 = Nil               Blatt erreicht: Pfad zu Ende
                path ← [d1]               definiere Pfad
                result ← result + [path]      speichere Pfad
            else                      weiter geht es
                for d2 ← [l, r]           für jede Richtung: l, r
                    t2 ← t1.d2            gehe in die Richtung
                    if t2 = Nil               Blatt erreicht
                        path ← [d1, d2]           definiere Pfad
                        result ← result + [path]      speichere Pfad
                    else
                        ...                       und so weiter

    return result                     alle Pfade sind abgelaufen und in result gespeichert
```

Die Pünktchen stehen jetzt wieder für eine Codesequenz, von der wir nicht wissen, wie groß sie ist. Wenn ein Baum t_0 übergeben wird, dann muss als Erstes die Tiefe des Baumes bestimmt werden, also die Länge des längsten Pfades. Wenn der übergebene Baum die Tiefe n hat, dann muss erst einmal Code mit n geschachtelten Schleifen erzeugt werden. Uff: Das ist ein Job für Rekursion.

Mit Rekursion können wieder wie oben die Pünktchen ersetzt werden. Während des

Baumdurchlaufs kann der Code zum Durchlaufen nach Bedarf generiert werden. Wir beschränken uns auf die Lösung mit einem globalen Akkumulator, der hier $result$ genannt wird.

```
AllPaths(t)                          alle Pfade bei nicht perfektem Binärbaum mit Wurzel t
    result ← Nil                     globale Variable für APR

    if t ≠ Nil
        APR(t, Nil)                  die rekursive Hilfsfunktion APR füllt result
    return result                    mit allen möglichen Pfaden

APR(t, path)                         path ist ein Pfad von der Wurzel in Richtung Blätter
    if t = Nil                       leerer Baum
        result ← result + [path]     Pfad beendet, speichere ihn
    else
        for d ← [l, r]               weiter geht es in alle Richtungen (d = l, r)
            APR(t.d, path + [d])
```

Auch hier ist die Richtung d wieder der Name einer Record-Komponente. $t.d$ ist also entweder $t.l$ oder $t.r$. Das muss in einer konkreten Implementierung wieder eventuell mithilfe von Arrays codiert werden: $l = 0$, $t = 1$ und $t.d$ als $t[d]$.

Ein Hoch auf die Rekursion. Wieder einmal sieht man, wie sie Dinge vereinfacht.



Die hier beschriebene Methode des Durchwanderns eines Baumes unbekannter Größe und Gestalt zur Zusammenstellung aller Pfade von der Wurzel zu einem Blatt wird häufig als *Backtracking* bezeichnet. Wir bevorzugen die prinzipiell gleichwertige, aber in der Herleitung etwas abweichende Definition im nächsten Abschnitt, die einen stärkeren Schwerpunkt auf die Konstruktion eines Suchraums für eine erschöpfende Suche legt.

Backtracking

Backtracking ist eine einfache Optimierung der erschöpfenden Suche. Bei den n Damen ist diese Optimierung so offensichtlich, dass es schwerfällt, sie nicht anzuwenden. Man muss ja nicht warten, bis die letzte Dame platziert ist, bevor klar ist, dass es sich nicht lohnt, eine ungünstige Konfiguration weiter aufzubauen: Wenn die Damen in der ersten und der zweiten Zeile sich schon gegenseitig schlagen, dann ist klar: Wie immer wir die restlichen Damen setzen werden, es wird keine akzeptable Lösung dabei herauskommen. Wenn eh nichts Vernünftiges herauskommen wird, dann kann man schon gleich aufhören und sich viel Arbeit ersparen.

Eine entsprechende Modifikation ist schnell in die Schleifenlösung eingebaut: Bei jeder

hinzugekommen Dame wird sofort geprüft, ob sie sich mit denen, die schon da stehen, verträgt.

Bei einem (4×4)-Brett hat man dann folgenden Algorithmus:

```
for p0 ← 0 .. 3
    for p1 ← 0 .. 3
        if Ok(p0, p1)
            for p2 ← 0 .. 3
                if Ok(p0, p1, p2)
                    for p3 ← 0 .. 3
                        if Ok(p0, p1, p2, p3)
                            return (p0, p1, p2, p3)
```

Das funktioniert auch mit der rekursiven Variante. Auch hier muss lediglich ein Test eingebaut werden, bevor es in die nächste per Rekursion erzeugte Schleife geht:

```
QueensNBT(n)                    n Queens mit BackTracking
    return LoopsBT(n, n, [Nil]) n Schleifen 0 bis n–1
```

QueensNBT nutzt LoopsBT. Der erste Parameter von LoopsBT ist die Zahl der Schleifen, der zweite ist die Schleifenbegrenzung (von 0 bis $n-1$), der dritte ist ein Akkumulatorargument, ein Sammelbecken für alle möglichen Pfade als Liste von Listen. Es geht los mit einer Liste, die nur die leere Liste enthält. Dieser eine leere Pfad wird dann von LoopsBT auf alle mögliche Arten zu einem guten Pfad zu einem Blatt erweitert.

```
LoopsBT(count, limit, acc)
    if count = 0                    alle Pfade (boards/Bretter) wurden erzeugt und sind in acc
        return [ x ∈ acc | Ok(x)]       gib die guten zurück
    else
        return LoopsBT(
            count−1,                eine Schleife weniger, die noch zu erzeugen ist
            limit,                  unveränderte Schleifenbegrenzung
            [ board + [x] | board ∈ acc, x ← 0 ... limit − 1, and Ok(board + [x])]
        )
```

Der dritte Parameter des rekursiven Aufrufs ist der Akkumulator, der um eine Schleife erweitert wird. Natürlich werden dort keine Schleifen aufgesammelt, sondern das Ergebnis der Aktion von Schleifen: Die Erweiterungen von Pfaden (Brettern) um eine weitere Richtung (Entscheidung für die Platzierung einer Dame).

Der Akkumulator enthält Pfade als Listen $board$ von Richtungsentscheidungen (wo kommt die nächste Dame hin?). In die nächste Rekursion werden die um x erweiterten Pfade $board + [x]$ übernommen, die Ok sind. x durchläuft dabei alle Richtungsentscheidungen einer Schleife. Der Listenausdruck »ist« also die nächste

Schleife.

In einer weniger ambitioniert funktionalen Variante wird der Akkumulator in einer globalen Variablen verwaltet und man setzt eine explizite Schleife ein:

```
QueensNBT(n)
    acc ← Nil                    globale Variable als Sammelbecken aller Pfade (Bretter)
    LoopsBT(n, n, Nil)           rekursiver Aufruf mit einem leeren Pfad
```

Statt alle möglichen Pfade in einem Akkumulator-Parameter aufzusammeln, verwenden wir eine globale Variable acc , die von LoopsBT gefüllt wird. Ein Pfad entspricht einem ganz oder teilweise gefüllten Brett. Der Listenausdruck

$$[board + [x] \mid board \in acc,\ x \leftarrow 0\ \ldots\ limit - 1, \text{and } Ok(board + [x])]$$

beschreibt, wie *alle* Bretter, die noch im Spiel (in acc) sind, um eine Position x erweitert werden. Das kann durch eine explizite Schleife ersetzt werden, in der *ein* Pfad auf alle möglichen Arten erweitert wird.

```
for x ← 0 ... limit − 1
    if Ok(board + [x])
        ... mach weiter mit board + [x]
```

Statt alle akzeptablen Bretter in acc durch die Rekursionen zu schleppen, wird ein Brett (Pfad) in jeder Rekursionsstufe ausgebaut und, wenn es vollständig ist, in acc gespeichert. Das ergibt einen etwas anderen in seiner Grundstruktur imperativen Algorithmus, der allerdings mit *persistenten* Listen als Werten der Variablen $board$ arbeitet:

```
LoopsBT(count, limit, board)
    if count = 0                 fertig: board enthält alle Damen ...
        acc ← acc + [board]      ... und kann im globalen acc gespeichert werden
    else
        for x ← 0 ... limit − 1  Schleife statt Listenausdruck: erweitere den
            if Ok(board + [x])       Pfad auf alle möglichen Arten x, die Ok sind
                LoopsBT(             und mach mit dem erweiterten Pfad weiter
                    count − 1,
                    limit,
                    board + [x])
```

Man beachte diese subtilen Unterschiede der funktionalen Variante oben und der imperativen Variante hier. In der funktionalen Variante laufe ich mit einem Leiterwagen (acc) voller Kleidertüten ($board$) durch die Stadt. Jede Tüte enthält ein partielles Outfit. In jedem Geschäft wird jede Tüte auf alle möglichen Arten erweitert. Das gibt neue Tüten, die die alten auf dem Leiterwagen ersetzen. Das geht so weiter, bis jede Tüte ein vollständiges und akzeptables Outfit enthält. Am Schluss habe ich einen Leiterwagen

voller Tüten mit jeweils einem passablen Outfit.

In der imperativen Variante hier laufe ich mit einer Tüte ($board$) durch die Stadt und wenn immer diese ein vollständiges und akzeptables Outfit enthält, wird sie dupliziert (Persistenz) und dem braven Begleiter (acc) übergeben. Dann geht es zurück zum zuletzt besuchten Geschäft. Dort liegt noch die Tüte in der Version, die sie hatte, als wir das Geschäft betreten haben (Persistenz) und die Suche geht weiter mit der um ein anderes Teil erweiterten Tüte.

Backtracking bringt eine deutliche Beschleunigung der erschöpfenden Suche. Viele Elemente des Suchraums werden erst gar nicht erzeugt, weil schon früh klar wird, dass eine Teillösung mit ihnen nicht zu einer vollständigen Lösung erweitert werden kann. Bergstiefel kombiniert mit pinkfarben Leggings werden kein akzeptables Outfit für den Abiball ergeben, egal welches Oberteil gewählt wird.

Wir kommen nicht in eine andere Laufzeitklasse, aber das Laufzeitverhalten wird im Allgemeinen drastisch verbessert.

Es stellt sich die Frage, ob jede erschöpfende Suche in dieser Art optimiert werden kann.

Kandidaten nicht stückweise bewertbar: kein Backtracking

Nicht jede Suche kann mit per Backtracking optimiert werden. Bei der morgendlichen Suche nach einen passenden Outfit können unpassende Kombinationen aussortiert werden, bevor sie zu einer vollständigen Garderobe ausgebaut wurden. Bei der Suche nach dem Partner für's Leben geht das leider nicht. Die werden ja nicht unter unserer Aufsicht zusammengebaut, sondern kommen am Stück und können darum nur als Ganzes geprüft werden.

Es gibt Situationen, bei denen die Kandidaten zwar Schritt für Schritt konstruiert werden, aber eine Teillösung noch nichts über die Gesamtlösungen aussagt, die aus ihr generiert werden können. Beim Kochen kann beispielsweise aus einem Topf mit kochendem salzigen Wasser, das als solches ungenießbar ist, durchaus ein leckeres Pastagericht entstehen.


Backtracking kann nur angewendet werden, wenn

- ✔ potenzielle Lösungen schrittweise konstruiert werden und wenn
- ✔ noch unvollständige Lösungskandidaten sich schon bewerten lassen.

Die Kandidaten für das n -Damen-Problem wurden schrittweise aufgebaut. Das muss nicht zwingend so gemacht werden. Es gibt ein völlig anderes Verfahren zur Generierung der Kandidaten, das diese am Stück liefert. Wir beobachten dazu zunächst einmal, dass es sich bei den Kandidaten, stellt man sie als Liste der Spaltenpositionen dar, um die Menge aller Permutationen der Länge n von einer Menge von n unterschiedlichen Werten

handelt (siehe Abbildung 11.8).

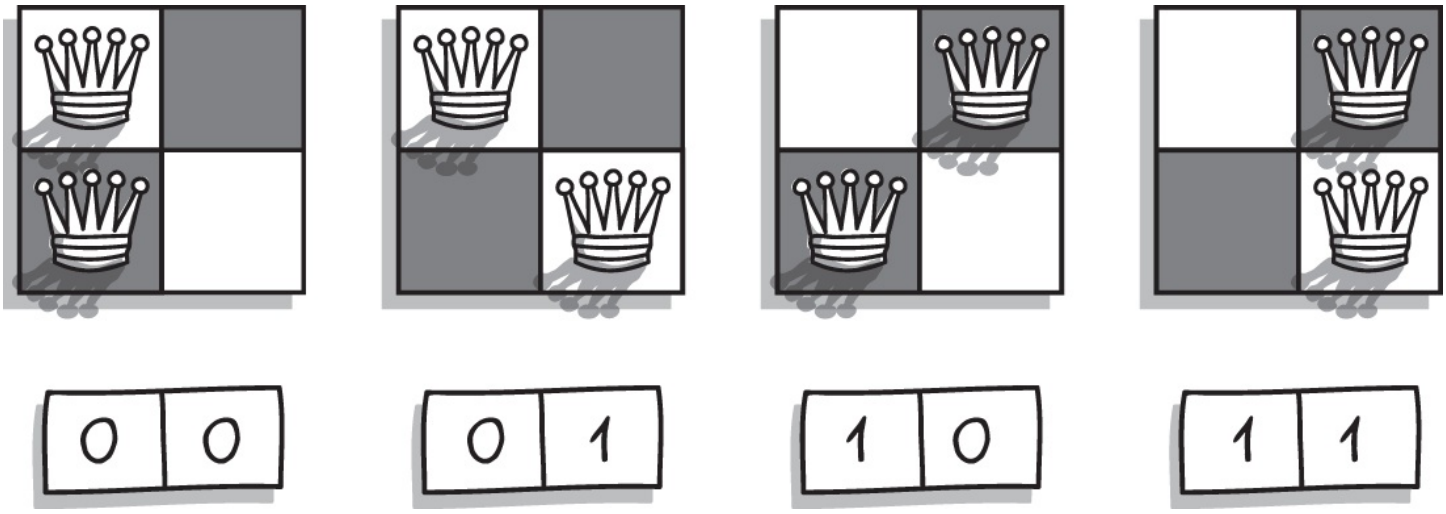


Abbildung 11.8: Suchraum als Tupelmenge

Tupel lassen sich in geschachtelten Schleifen und Schritt für Schritt per Rekursion aufbauen, das haben wir oben auch so gemacht. Für die Erzeugung aller Tupel der Länge n gibt es bei unbekanntem n aber ein besseres Verfahren als das rekursive Verfahren von oben.

Nennen wir es das *Kilometerzähler*-Verfahren zur Erzeugung von Tupeln. Ein Kilometerzähler beginnt mit 000 (Ok, das ist einer für ein Kinderfahrrad) und geht bis 999. Das Bemerkenswerte daran ist, dass alles, was der Zähler anzeigt, vollständig ist. Von 000 bis 999. Es gibt keine Teilfüllungen. Wenn wir darum den Zähler einsetzen, dann ist Backtracking *nicht* möglich.

Eine imperative Lösung für die Erzeugung aller Tupel arbeitet mit einem Array, dem *Kilometerzähler* und wird am besten als Iterator über alle Tupel definiert:

TupelIterator(n) = **record** [
    $a \leftarrow$ **array**[0 .. $n-1$](0...0)
    done $\leftarrow$ **false**
    $j \leftarrow 0$

```
next()
    res ← a
    j ← n − 1
    while j > −1 and a[j] = n − 1        Suche von rechts nach links
        a[j] ← 0                         nach der ersten Stelle, die ohne Überlauf
        j ← j − 1                        erhöht werden kann
    if j = −1                            wenn es keine gibt, sind wir fertig.
        done ← true
    else
        a[j] ← a[j] + 1
        j ← j − 1
    return res
hasNext()
    return not done
]
```

Damit kann die Suche nach einer Lösung des n -Damen-Problems zwar recht effizient, aber nur erschöpfend und ohne Backtracking formuliert werden:

```
NQueensIter(n)
    iter ← TupleIterator(n)
    while iter.hasNext()
        next ← iter.next()
        if Ok(next)
            return next
```

Es gibt gewisse Bezüge zwischen Backtracking und struktureller Rekursion. Bei der strukturellen Rekursion wird die Lösung für das Ganze aus der Lösung für seine Komponenten konstruiert. Beim Backtracking besteht der Suchraum aus Elementen, deren Komponenten etwas über die mögliche Eignung des aus ihnen konstruierten Ganzen aussagen.

Backtracking als Suche im Zustandsraum

Das Aufbauen des Suchraums hat Entscheidungsstellen, die sich später eventuell als Fehlentscheidungen herausstellen, die sich aber jetzt noch nicht als solche zu erkennen geben. Bei jedem Problem, das mit Backtracking gelöst werden kann, hat darum der Lösungsraum eine *baumartige* Struktur mit den Blättern des Baumes als Suchraum. Teillösungen und Lösungen sind die Knoten in diesem Baum. Die Kanten stellen die Entscheidung über eine mögliche Erweiterung der Teillösung dar (siehe Abbildung 11.9).

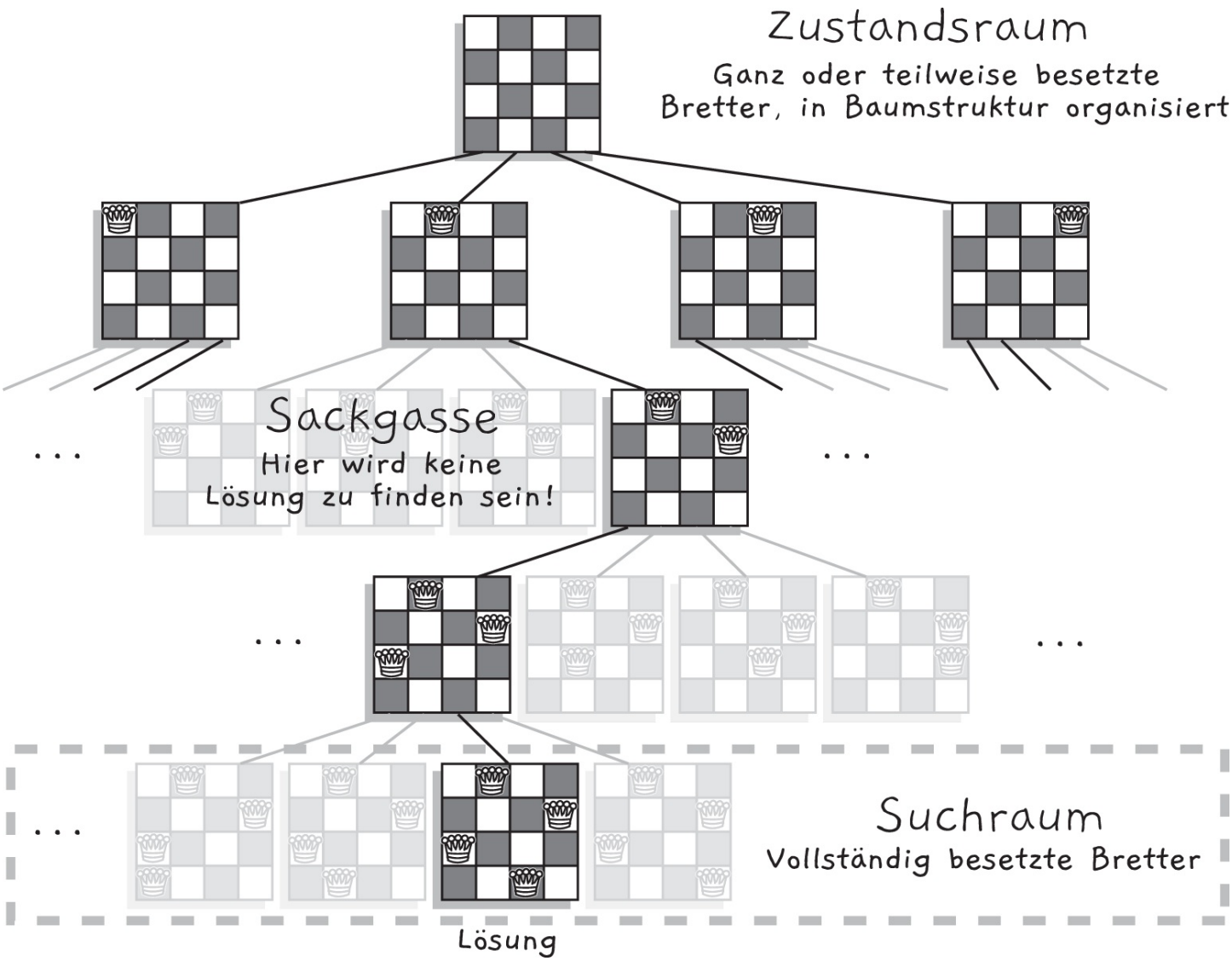


Abbildung 11.9: Suchraum als Blätter des Zustandsbaums

Backtracking-Algorithmen können und werden oft als ein *Wandern im Zustandsraum* interpretiert. Das ist in einer imperativen Formulierung des Algorithmus besonders intuitiv: Man geht von einer Konfiguration zu einer Folgekonfiguration und zieht sich von dort wieder zurück (*backtrack!*), wenn sich der gewählte Weg als Sackgasse erweist.

Eine entsprechende Formulierung des n -Damen-Backtracking-Algorithmus verwendet eine veränderliche Datenstruktur als Repräsentation der aktuellen Position im Zustandsraum. Im folgenden Algorithmus übernimmt $board$ diese Rolle:

```
NQueensBTImp(n)
    solutions ← Nil      globale Variable der Lösungen: boards, die Ok sind
    board ← Nil          globale Variable: board, das wächst und schrumpft
    Solve()              im Ok-Zustand wird es in solutions abgespeichert
    return solutions     alle Lösungen werden zum Schluss zurückgegeben
```

```
Solve()
erweitert board auf alle Arten zu einer Lösung und speichert diese
    if board.length = n          globale Variable board ist vollständig und Ok
        solutions ← solutions + [copy(board)]   aktuellen Inhalt von board kopieren
                                                und in der globalen Variablen solutions speichern
    else
        for x ← 0 ... n − 1      auf alle möglichen Arten
            board.append(x)      board erweitern
            if Ok(board)         falls erweitertes board Ok, ...
                Solve()          ... versuche, es auf alle Arten zu vervollständigen
            board.removeLast     board wieder verkleinern
```

Man beachte, dass es sich hier um eine imperative Umformulierung des Algorithmus AllPaths von weiter oben handelt, bei der alle Pfade in einem Baum unbekannter Größe und Gestalt aufgesammelt werden. Die Variable *board* enthält wechselnde Konfiguration des Bretts und in jedem korrekten und vollständigen Zustand wird ihr Wert in *solutions* abgespeichert. *board* pulsiert quasi und wird dabei gelegentlich in Kopie gespeichert. Der Algorithmus beruht auf einem veränderlichen, nicht-persistenten Wert in *board* . Die Persistenz der gefundenen Lösungen muss darum durch bewusstes Kopieren selbst organisiert werden.

Die Idee des Backtrackings ist diese: Geschachtelte Schleifen über Wahlmöglichkeiten mit frühzeitigem Abbruch bei erkennbar schlechter Wahl. Komplexität entsteht, wenn während der Ausführung des Algorithmus, abhängig von der Eingabe, Schleifen über Wahlmöglichkeiten erzeugt werden müssen. Das geht nur mit Rekursion und kann auf unterschiedliche Arten organisiert werden. Dabei verheddert man sich leicht in Persistenzproblemen.

Das häufig übersehene Persistenzproblem dabei ist: Werden die (Teil-)Lösungen in einer persistenten Datenstrukturen verwaltet, dann werden die verschiedenen Versionen der Lösungspfade automatisch im Hintergrund verwaltet. Bei nicht persistenten Datenstrukturen muss diese Verwaltung mithilfe von Kopieraktionen selbst organisiert werden. Zusammengefasst:

- ✔  *Backtracking ist einfach:* Schleifen über geprüfte Wahlmöglichkeiten, die in der Regel in rekursiven Aufrufen stecken.
- ✔ *Veränderlichkeit und Persistenz:*
    - Veränderlich: Auf- und Abbau der Lösungspfade im Aufrufer der rekursiven Funktion. Lösungen als Schnappschuss (Kopie) speichern.
    - Persistent: Aufbau der Lösungspfade im Aufrufer der rekursiven Funktion. Abbau und Schnappschuss eines Lösungspfads passieren »automatisch«.

Verzweigen und Begrenzen

Backtracking ist eine Optimierung der erschöpfenden Suche, *Verzweigen und Begrenzen* (engl. *branch and bound*) ist eine Anpassung des Backtrackings an Optimierungsprobleme. Auch hierbei muss der Suchraum als Blätter eines rekursiv erzeugbaren Zustandsraums generierbar sein. Die Kandidaten werden aber nicht mit einer Ja-/Nein Entscheidung bewertet, sondern mit einem Qualitätsmaß vermessen und bewertet.

Nehmen wir als Beispiel einen Irrgarten. Eine Ja-/Nein-Entscheidung wäre: *Ist das ein Weg zum Ausgang?* Ein Optimierungsproblem wäre: *Was ist kürzeste Weg zum Ausgang?* mit der Weglänge als Qualitätsmaß. Bei einem Optimierungsproblem gibt es darum sowohl ein Ok-Kriterium der Lösungskandidaten (Weg zum Ausgang?) als auch eine Bewertung der Lösungen (Weglänge).

Erschöpfende und Backtracking-Suche im Irrgarten

Wenn es um Wege geht, dann sind Graphen nicht weit entfernt. Als Datenmodell eines Irrgartens bieten sich darum Graphen an. Jeder Platz im Garten ist ein Knoten und jeder Weg, der zwei Plätze verbindet, eine Kante (siehe Abbildung 11.10).

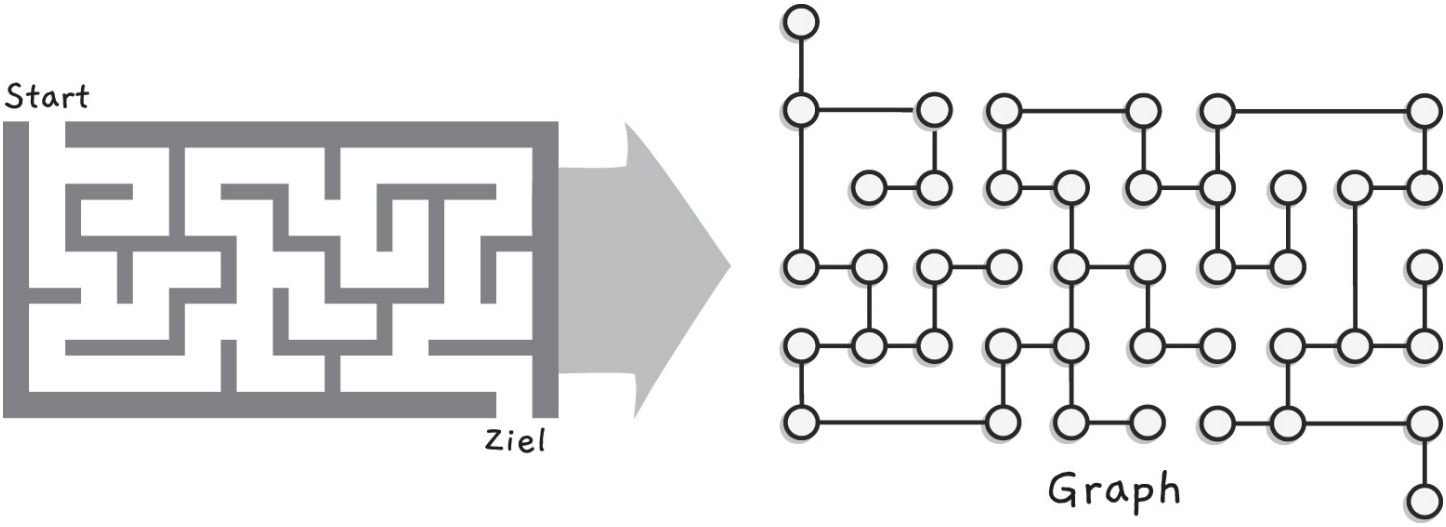


Abbildung 11.10: Graphenmodell eines Irrgartens

Für eine erschöpfende Suche benötigen wir einen Suchraum, der dazu noch systematisch (baumartig) erzeugbar sein muss. Suchen wir den Weg erstmal wirklich mit roher Kraft:

- ✔ Wir erzeugen *alle* Folgen von Knoten.
- ✔ Dann schauen wir, ob sie beim Eingang starten und beim Ausgang enden und
- ✔ ob benachbarte Knoten im Graphen durch eine Kante verbunden sind.

Das ist ein beachtlicher Suchraum, aber nicht unendlich groß. Er sollte keine Zykel

enthalten, darum kann jeder potenzielle Weg höchstens alle Knoten einmal enthalten. Wir können uns also auf die Folgen beschränken, in denen jeder Knoten maximal einmal vorkommt. Das wären alle Permutationen aller Knotenfolgen der Länge 1 bis n, wenn der Graph n Knoten enthält. Ziemlich viele.

Hmm, vielleicht doch lieber nicht so. Der Suchraum der Permutationen hat ja auch keine baumartige Struktur mit sich immer weiter ausdehnenden Teillösungen. Die Intuition liefert uns eine viel bessere Idee zur Organisation des Suchraums.

Wir starten am Eingang und bewegen uns von dort weiter, bis der Ausgang oder eine Sackgasse erreicht ist. Dabei merken wir uns in einer Liste die Plätze (Knoten), die besucht wurden. Ist der Ausgang gefunden, dann enthält unsere Liste den gewünschten Weg. Sind wir in eine Sackgasse geraten, dann geht es zurück, der letzte Knoten in der Liste wird gelöscht und wir probieren es mit einem anderen Weg. Wenn es keinen Weg mehr gibt, den wir noch nicht gegangen sind, dann geht es weiter zurück. Wenn so alles erfolglos durchprobiert wurde, dann gibt es keinen Ausgang.

Die teilweise konstruierten Wege zum Ziel sind unvollständigen Kandidaten. Deren Konstruktion wird abgebrochen, wenn sie nicht erweitert werden können, ohne durch eine Wand zu gehen (siehe Abbildung 11.11).

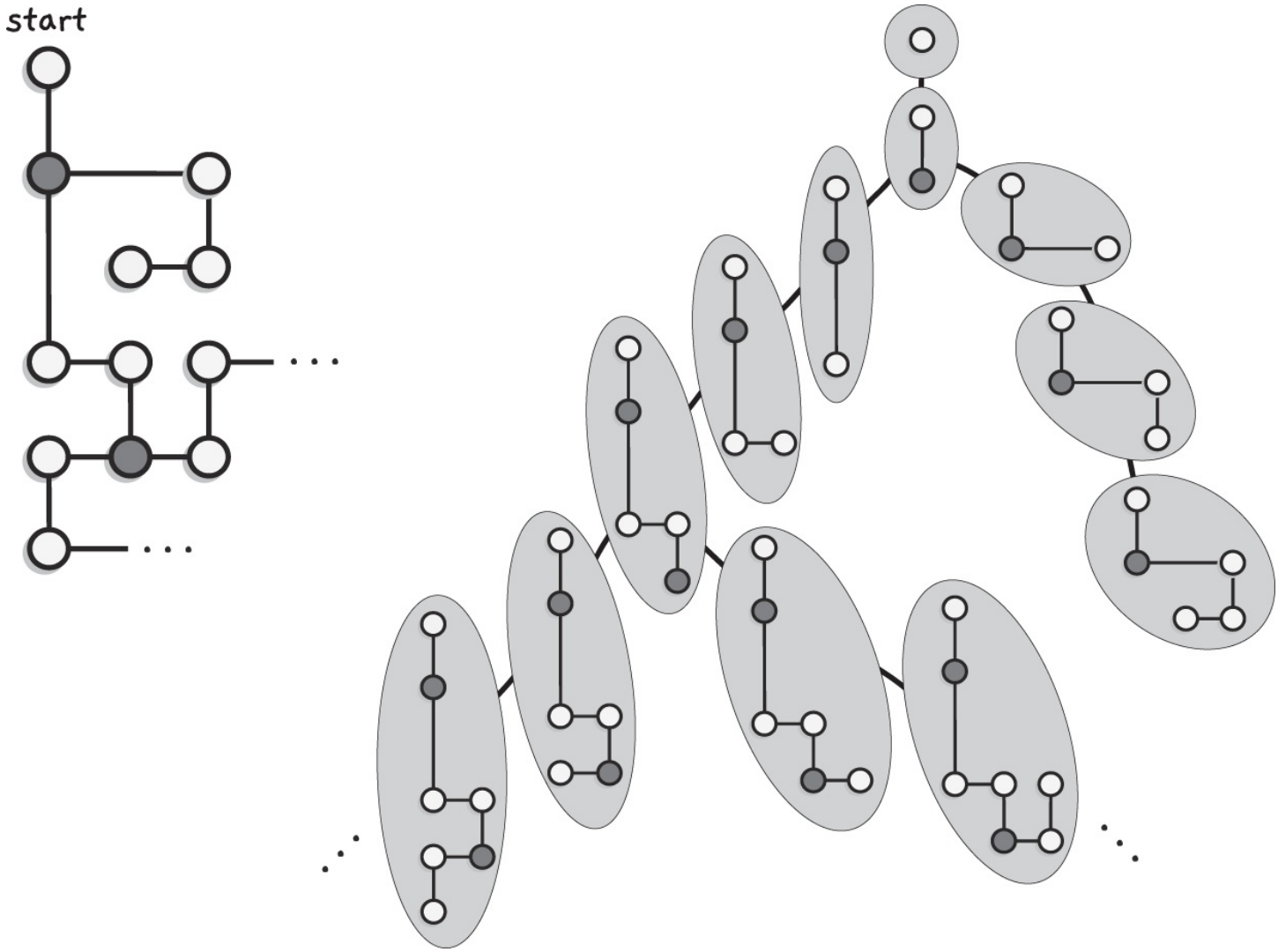


Abbildung 11.11: Baumartiger Aufbau des Suchraums der Wege (Anfang)

Der Algorithmus folgt dem Backtracking-Schema von oben, hier in der imperativen Variante. Die Parameter sind der Graph in Form seiner Knoten V und Kanten E. Außerdem werden der Startknoten $start$ und der Zielknoten $dest$ übergeben. In der Variablen $path$ wird ein Pfad aufgebaut. Er startet mit $start$ und wenn er mit $dest$ endet, kann er zu den Lösungen in $solutions$ hinzugefügt werden. Bei einer Erweiterung von $path$ wird geprüft, ob der neue Knoten nicht schon enthalten ist (keine Zykel) und ob es eine Verbindung vom letzten Knoten des bisherigen Pfades zu ihm gibt.

```
MazeBTImp(V, E, start, dest)
    solutions ← Nil          alle Lösungen, globale Variable für Solve
    path ← [ ]               Variable mit pulsierendem path-Wert (leer am Anfang)
    Solve()                  Solve kopiert und speichert gute
    return solutions         Werte von path in solutions

    Solve()
        if path.last = dest      path ist Ok: kopieren und abspeichern
            solutions ← solutions + [copy(path)]
        else
            for x ← V            alle Möglichkeiten, den path zu erweitern
                if not path.contains(x) and E.contains( (path.last, x) )
                    path.append(x)    Erweiterung Ok: path erweitern
                    Solve()
                    path.removeLast()     und wieder kürzen
```

Hier »pulsiert« der Wert in der Variablen $path$. Immer wenn er eine Lösung darstellt, wird er als Kopie des aktuellen Wertes in $solutions$ gespeichert. Der Algorithmus beruht darauf, dass $path$ veränderlich ist. Wenn ein bestimmter Zustand aufgehoben werden soll, dann muss er kopiert werden. Solche Probleme hat man bei einer rein funktionalen Lösung mit ihren persistenten Datenstrukturen nicht. Alle einmal erzeugten Werte bleiben für immer bestehen. Nichts wird hinterrücks nachträglich geändert. Hier wird absichtlich mit einem veränderlichen nicht-persistenten Wert in $path$ gearbeitet. Wenn Persistenz notwendig ist, weil der Wert in seiner aktuellen Zustand dauerhaft sein soll, dann muss selbst kopiert werden.

So jetzt können wir aus dem Irrgarten herausfinden. Suchen wir noch nach dem kürzesten Weg. Das ist ein Optimierungsproblem.

Optimierungen und Bewertungsfunktionen

Bei den algorithmisch zu lösenden Problemen unterscheidet man

- **Entscheidungsprobleme**

  Ein Entscheidungsproblem ist eine Frage, die mit Ja oder Nein beantwortet werden kann, zum Beispiel: Ist n eine Primzahl?

- **Berechnungsprobleme**

  Ein Berechnungsproblem ist eine Frage, die mit einen Wert beantwortet werden kann, zum Beispiel: Bestimme die Primfaktoren von n .

- **Optimierungsprobleme**

  Ein Optimierungsproblem ist eine Frage nach der besten Lösung, zum Beispiel:

Bestimme den kürzesten Weg von hier zum Bahnhof.

Beim *Verzweigen und Begrenzen* wird die Lösung für ein Optimierungsproblem gesucht. Es ist eine Suchtechnik, die einen baumartig organisierten Suchraum voraussetzt.

- ✔ Jede (Teil-)Lösung wird als das Ergebnis einer Serie von Entscheidungen gedeutet.
- ✔ Der Zustandsraum ist ein Baum mit den Entscheidungen als Kanten.
- ✔ Die Lösungen werden mit einer Bewertungsfunktion **o** bewertet.
- ✔ Eine optimale Lösung l ist eine Lösung, deren Bewertung **o** (l) maximal/minimal ist.

Der wesentliche Gedanke ist dabei, dass die Bewertung **o** der Lösungen auf eine Bewertungsfunktion **o'** von Teillösungen übertragen werden kann. Für jede Teillösung wird ein *Grenzwert* (engl. *bound*) **b** berechnet, der eine Schranke für die Bewertung der Lösungen darstellt, die aus dieser Teillösung generiert werden können.

o' (Teillösung) = **b** $\Rightarrow$ **o** (Gesamtlösung(Teillösung)) $\leq$ **b**

wenn ein maximaler Wert gesucht wird bzw.

o' (Teillösung) = **b** $\Rightarrow$ **o** (Gesamtlösung(Teillösung)) $\geq$ **b**

wenn ein minimaler Wert gesucht wird.

Das Finden einer geeigneten Bewertungsfunktion ist meist der schwierigste Teil des Verfahrens. In unserem Beispiel der Suche nach dem kürzesten Weg wird ein minimal langer Pfad p gesucht und die Bewertungsfunktion ist: **o** $(p) = p$ .length. Das kann leicht auf eine Bewertung von Teillösungen erweitert werden: Die Gesamtpfad wird niemals länger sein als ein Anfang des Pfades. Also nehmen wir einfach

o' (p) = **o** $(p) = p$ .length

Beim Backtracking werden Teillösungen tl generiert. Diese bewerten wir mit **o'** (tl) und vergleichen sie mit der bisher gefundenen besten Lösung l mit ihrer Bewertung **o** (l). Nur wenn **o'** (tl) besser als **o** (l) ist, geht es weiter. Ansonsten wird abgebrochen (siehe Abbildung 11.12). Für die Suche nach dem kürzesten Pfad durch das Labyrinth bekommen wir dann folgenden Algorithmus als Erweiterung des Backtrackings:

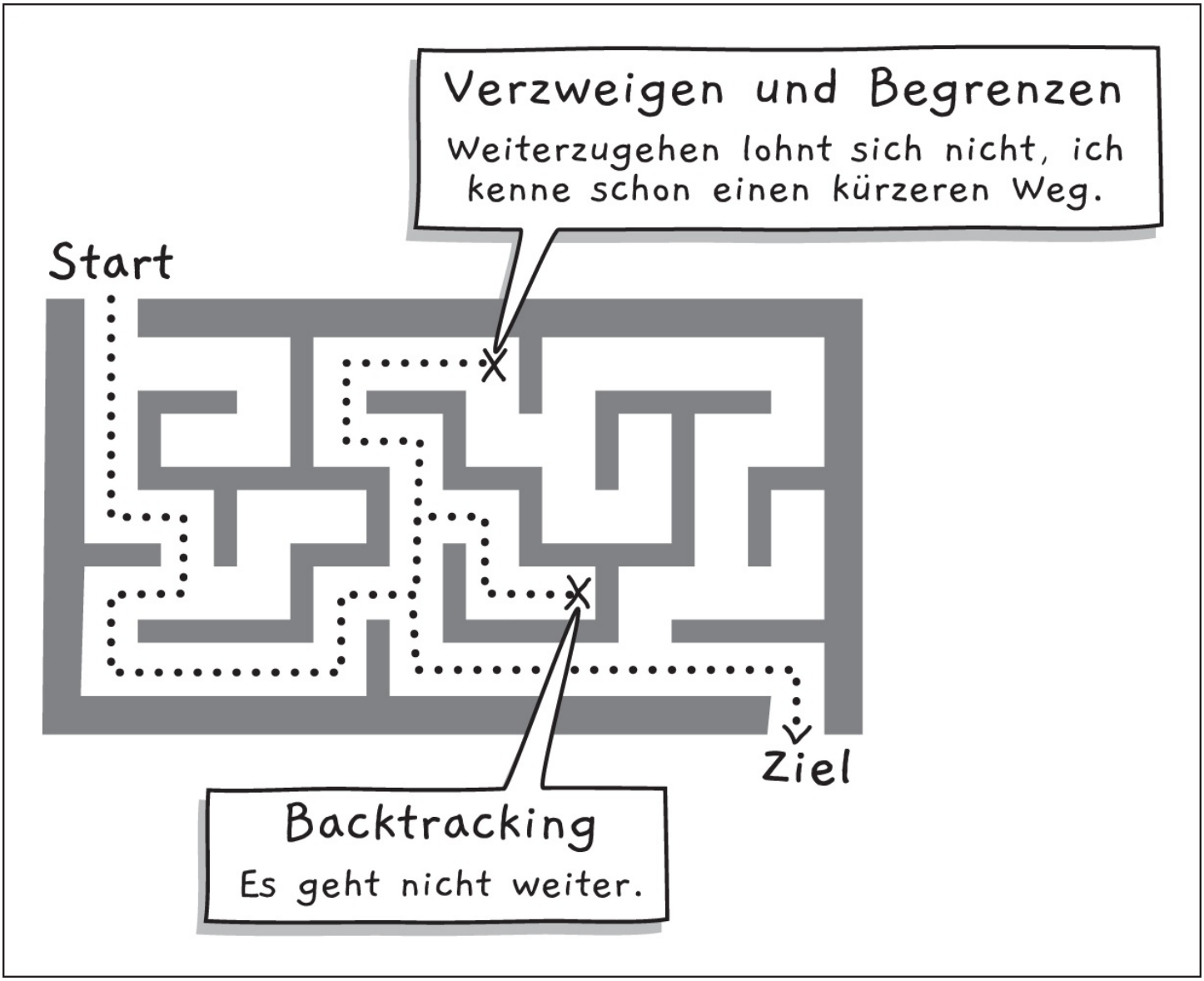


Abbildung 11.12: Backtracking mit Verzweigen und Begrenzen

```
MazeBBImp(V, E, start, dest)
    bestPath ← Nil                  bisher gefundener bester Pfad
    bestLenth ← ∞                   Länge des besten Pfades
    path ← [start]                  pulsierende Pfadvariable, startet als einelementige Liste
    Solve()
    return bestPath
    Solve()                         lokale Funktion hat Zugriff auf alles von MazeBBImp
        if path.last = dest and path.length < bestLength
            bestPath ← copy(path)   Pfad in seinem aktuellen Zustand speichern
            bestLenth ← path.length
        else
            for x ← V               Verzweigen
                if not path.contains(x) and E.contains( (path.last, x) )
                    if path.length ≤ bestLength     Begrenzen
                        path.append(x)              einatmen
                        Solve()                     tauchen
                        path.removeLast             ausatmen
```

Solve hat als lokaler Hilfsalgorithmus Zugriff auf alle Variablen und Parameter von MazeBBImp. In der Variablen $path$ werden Pfade auf- und abgebaut und so alle Wege durch den Graphen durchgespielt. Wenn ein Pfad zum Ziel führt und kürzer ist als der bisher gefundene Weg, dann wird er in Kopie abgespeichert. (Kopieren: punktuelle Persistenz unter der Kontrolle des Algorithmus.)

Die Begrenzung besteht darin, dass die Länge von Teillösungen bewertet wird und zum Abbruch des Lösungsaufbaus führen kann.

Komplexitätsklassen: Schwere Probleme führen zu anstrengender Arbeit

In diesem Kapitel ging es bisher darum, wie mit Problemen zu verfahren ist, für die man keine schlaue Lösungsidee hat und darum nur rohe Kraft bleibt. Mit roher Kraft ist dabei meist die Anstrengung einer erschöpfenden Suche gemeint. Auch wenn das mit dem Suchen nicht ganz so einfach (»roh«) ist, wie es auf den ersten Blick erscheint und oft auch noch recht schlau mit Backtracking oder Verzweigen und Begrenzen optimiert werden kann, so finden doch die entsprechenden Algorithmen nur langsam, mit viel Anstrengung eine Lösung.indexPolynomielle Laufzeit Sie sind, ob mit oder ohne Backtracking beziehungsweise Verzweigen und Begrenzen, in aller Regel nicht polynomiell.

Machen wir vielleicht etwas falsch, wenn wir mit roher Kraft arbeiten? Wenn wir keine

schlaue Idee haben, dann heißt das ja noch lange nicht, dass es prinzipiell keine gibt. Liegt es an uns, an unserem begrenzten Vermögen oder dem der Menschheit insgesamt, dass es keinen besseren Algorithmus gibt, oder können manche Probleme tatsächlich nur mit roher Kraft gelöst werden?



In Kapitel 2 haben wir gesehen, wie man Algorithmen je nach ihrer asymptotischen Laufzeit in unterschiedliche *Laufzeitklassen* einteilen kann.

Die Laufzeitklasse ist also auch ein Maß der Anstrengung, die eine Problemlösung mit einem bestimmten Algorithmus erfordert. Das Problem ist sozusagen die Rennstrecke und die Algorithmen sind die Kaninchen, die auf dieser Strecke um die Wette laufen. Da gibt es langsame wie Egon, der inzwischen völlig verfettet ist, und schnelle, die sich in einer völlig anderen Laufzeitklasse bewegen.

Gibt es ein ähnliche Klassifikation für Probleme wie die Laufzeitkomplexität für Algorithmen? Also so etwas wie ein Maß, mit dem Probleme bewertet werden können. Das Maß müsste etwas darüber aussagen, wie schnell ein Algorithmus bestenfalls sein kann, der es löst. Das wäre dann ein Maß für die Schwierigkeit eines Problems.

Ein solches Maß wäre schon echt nützlich. Man müsste bei einem Problem mit bekanntermaßen hohem Schwierigkeitsgrad nicht über eine mögliche bessere Lösung nachdenken. Ok, während man darauf wartet, dass ein $\mathcal{O}(n^n)$ Algorithmus sein Ergebnis ausgibt, hat man ja viel Zeit zum Nachdenken. Aber wenn klar ist, dass das Nachdenken zu keinem Ziel führt, dann kann man auch in den Garten gehen und leckeren Löwenzahn ernten. Das ist auch anstrengend, aber hat sicher ein Ergebnis. Etwas, das man von unserem Nachdenken über einen besseren Algorithmus vielleicht nicht unbedingt erwarten kann.

Schwer ist, was den Besten schwerfällt

In schwierigem Gelände kommt man nur langsam voran. Für die Schwierigkeit kann es verschiedene Ursachen geben: Es ist steil, es gibt viel Gestrüpp, durch das man sich kämpfen muss, der Boden ist schlammig und so weiter. Bei einfache, Gelände braucht ein guter Läufer für n Kilometer vielleicht $\mathcal{O}(n)$ Stunden. Etwas schwieriger ist es, wenn $\mathcal{O}(n^2)$ Stunden benötigt werden und ganz schwer sind überhängende Felsen, bei denen für n Kilometer $\mathcal{O}(n!)$ Stunden benötigt werden. Nimmt man als Messgröße immer den bekanntermaßen besten Läufer beziehungsweise Kletterer, dann ist man auf der sicheren Seite. Ein Fels, an dem die weltbesten Kletterer schon $\mathcal{O}(n!)$ pro n Kilometer brauchen, wird man wohl zu Recht als schwierig einstufen können.

Genauso kann man die Schwierigkeit eines Problems mit den Laufzeitklassen derjenigen Algorithmen abzuschätzen, die das Problem lösen. Wenn es keine schnellen Algorithmen zur Lösung eines Problems gibt, dann ist es anscheinend schwierig zu lösen. Lassen Sie uns hier nur grob zwischen Problemen unterschieden, die in *polynomieller Zeit* gelöst

werden können, und solchen, bei denen das nicht geht.



Polynomielle Laufzeit bedeutet, dass der Algorithmus in einer Laufzeitklasse $\mathcal{O}(n^k)$ für irgendeine genügend große Zahl k liegt. Dabei muss k eine Konstante sein, darf also nicht von n abhängen. Beispiele dafür sind $\mathcal{O}(n)$, $\mathcal{O}(n^2)$ oder auch $\mathcal{O}(n^{328})$. Auch $\mathcal{O}(1) = \mathcal{O}(n^0)$ ist polynomiell. Taucht n hingegen im Exponenten auf, also zum Beispiel $\mathcal{O}(2^n)$, so handelt es sich nicht mehr um ein Polynom. Mehr über polynomielle und nicht-polynomielle Laufzeiten erfahren Sie auch in Kapitel 2.

Ein Problem, das durch Algorithmen mit polynomieller Laufzeit gelöst werden kann, ist verhältnismäßig »gutartig« (siehe Abbildung 11.13). Wenn es andererseits zur Lösung eines Problems keine Algorithmen mit polynomieller, sondern nur solche mit *superpolynomieller* Laufzeit gibt, dann ist das ein ziemlich »bösartiges« Problem, denn ein solches Laufzeitverhalten führt bei steigender Eingabelänge früher oder später unweigerlich zu schmerzhaften Wartezeiten. Bei der Bewertung der Probleme kommt es nicht darauf an, ob wir einen polynomiellen Algorithmus tatsächlich kennen oder nicht, sondern ob ein solcher Algorithmus *prinzipiell* möglich wäre. Für manche gutartigen Probleme finden wir eventuell erst in 1000 Jahren einen polynomiellen Algorithmus (vielleicht aber auch nie, weil in 1000 Jahren längst die Kaninchen die Herrschaft über die Erde übernommen haben und die natürlich ganz andere Interessen haben als wir Menschen).

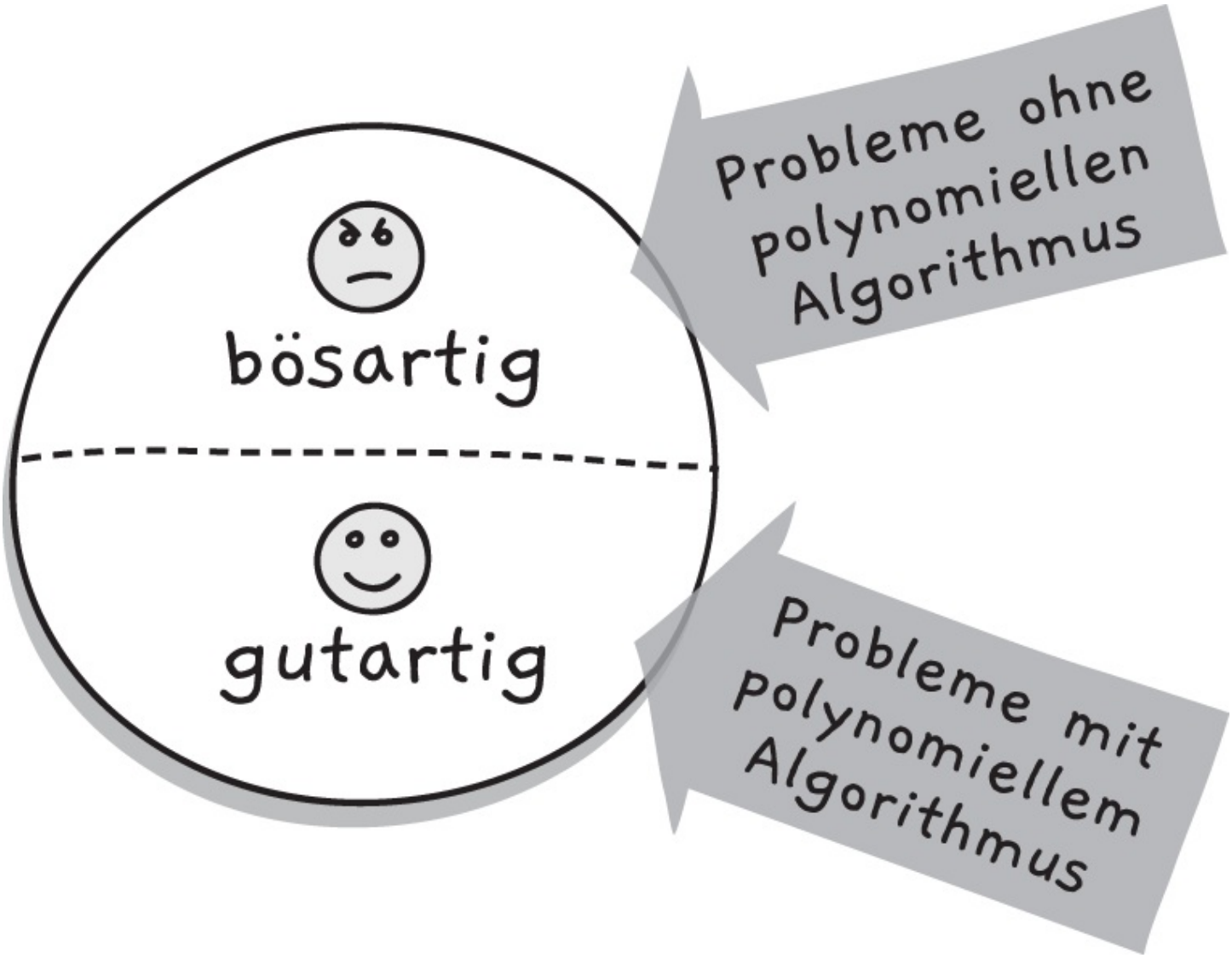


Abbildung 11.13: Bewertung von Problemen

Von den ganz, ganz bösartigen Problemen, die gar nicht berechenbar sind, für die es also überhaupt keinen Algorithmus gibt, noch nicht einmal einen superpolynomiellen, wollen wir hier übrigens gar nicht erst reden.

Vielen Problemen sieht man leider nicht so leicht an, ob sie gut- oder bösartig sind. Sicher, wenn für ein Problem schon ein polynomieller Algorithmus bekannt ist, dann ist das Problem natürlich gutartig. Das ist leicht. Da kann es zusätzlich noch weitere Algorithmen mit superpolynomieller Laufzeit geben, die das Problem ebenfalls lösen, aber das spielt keine Rolle. Solange es auch Lösungen mit polynomieller Laufzeit gibt, ist das Problem »gutartig«.

Bei manchen anderen Problemen kann man auch beweisen, dass es für sie keinen polynomiellen Algorithmus geben kann, beispielsweise wenn allein schon die Ausgabe des Ergebnisses nicht mehr polynomiell ist. Solche Probleme sind dann also eindeutig bösartig.

Dazwischen gibt es aber noch eine Grauzone von Problemen, für die zwar *aktuell* noch

kein polynomieller Algorithmus bekannt ist, bei denen aber auch noch niemand bewiesen hat, dass es keinen solchen Algorithmus geben kann (siehe Abbildung 11.14). In dieser Zone unbekannter Gut- oder Bösartigkeit befinden sich nun ausgerechnet viele interessante Probleme von hoher praktischer Relevanz, die man wirklich gerne lösen möchte, für die aber eben bislang noch niemand einen polynomiellen Algorithmus gefunden hat. Dazu gehören unter anderem viele Planungs- und Kombinationsprobleme. Hier würden wir gerne wissen, ob sich die Suche nach schnelleren Algorithmen überhaupt lohnt oder ob wir es mit bösartigen Problemen zu tun haben, die sich einer polynomiellen Lösung generell entzieht. Ein Beispiel für ein solches Problem werden Sie nun kennenlernen.

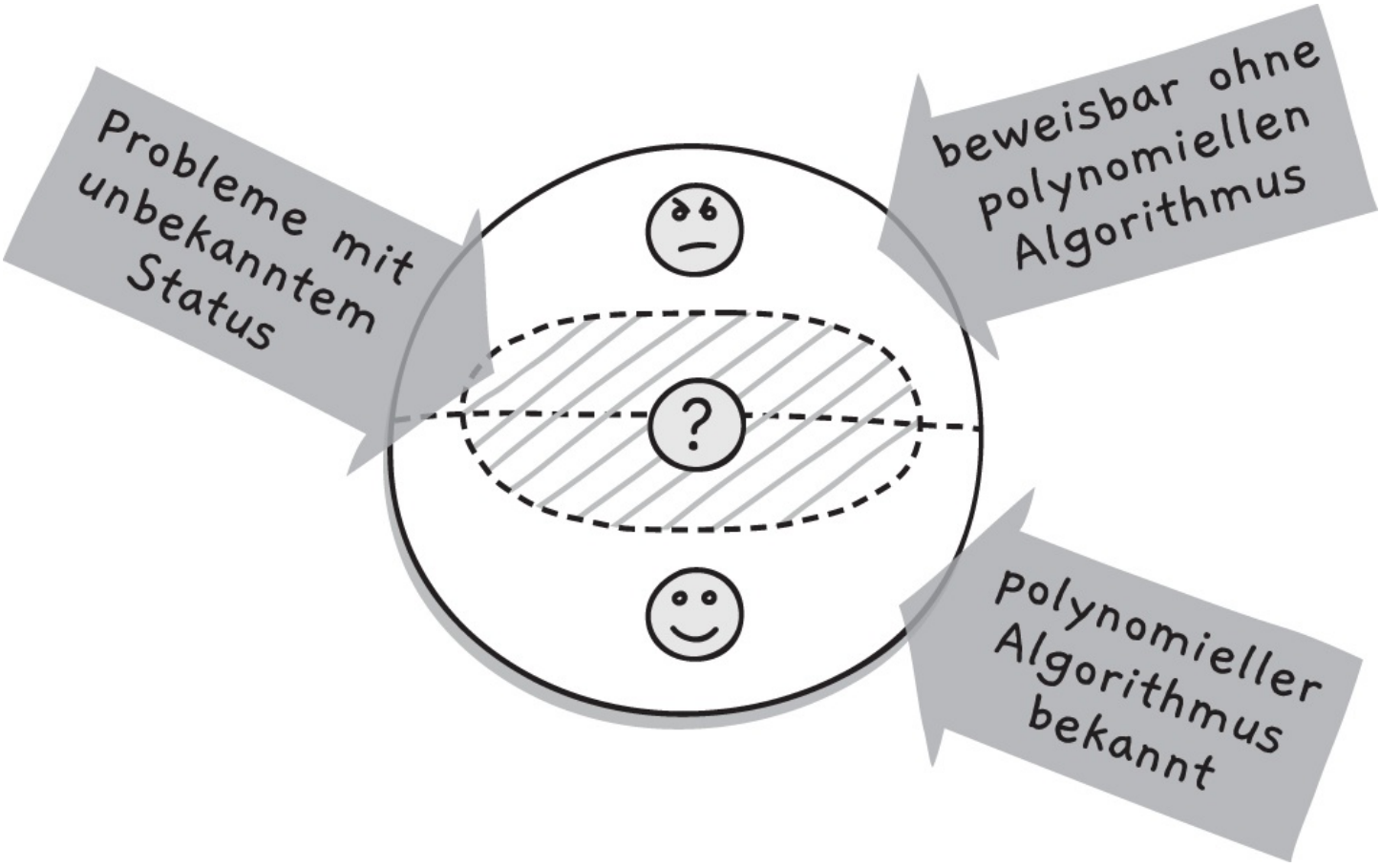


Abbildung 11.14: Status der Bewertung von Problemen

Ein Labyrinth der Kameras

Aus Sicherheitsgründen soll unser Irrgarten mit Kameras ausgerüstet werden. Schließlich könnte ja ein unglücklicher Gefangener bei der erschöpfenden Suche erschöpft zusammenbrechen und Hilfe benötigen. Da sich die Kameras ferngesteuert drehen lassen, bieten sich für ihre Montage die Kreuzungspunkte an; von dort aus kann man jeden Gang überwachen, der zur Kreuzung führt. Leider steht uns nur eine feste Anzahl von k Kameras zur Verfügung. Reichen diese k Kameras aus, um alle Gänge des Labyrinths überwachen zu können?

Diese Aufgabe können wir als Graphenproblem formulieren (Abbildung 11.15): Das Labyrinth sei als Graph gegeben mit einer Menge V aus n Knoten (den Kreuzungspunkten) und einer Menge E von Kanten (den Gängen). Gesucht ist eine Auswahl S von maximal k Knoten aus V, sodass jede Kante E zu einem Knoten aus S führt. Die Menge S enthält also gerade diejenigen Knoten, an denen wir unsere Kameras anbringen müssten, und wenn jede Kante in E mit mindestens einem Knoten verbunden ist, an dem sich eine Kamera befindet, können wir alle Gänge im Labyrinth überwachen. Unsere Aufgabe soll es nun lediglich sein zu überprüfen, ob eine solche Auswahl S von maximal k Knoten überhaupt existiert. Wir haben es hier also mit einem *Entscheidungsproblem* zu tun. Wenn man die Herausforderung liebt, kann man daraus auch ein *Optimierungsproblem* machen und nach der *minimalen* Anzahl k von Kameras fragen, mit der sich gerade noch eine komplette Überwachung des Labyrinths realisieren lässt.

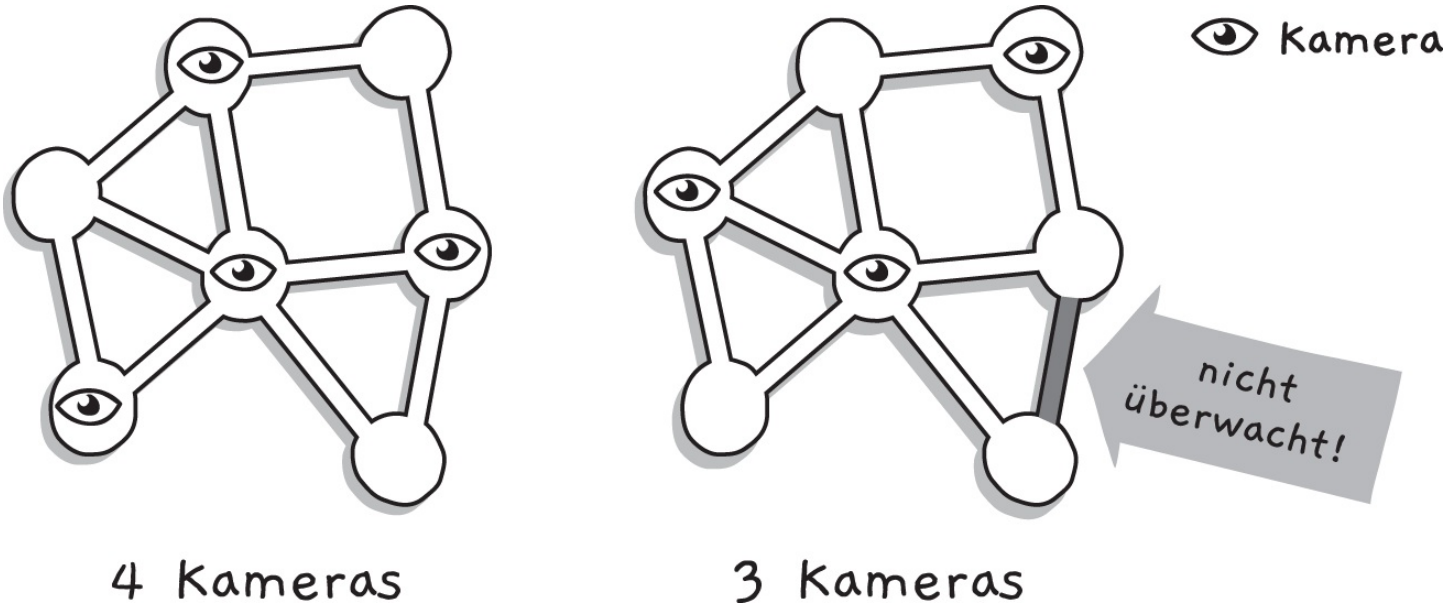


Abbildung 11.15: Man braucht mindestens vier Kameras, um diesen Irrgarten zu überwachen.

Dieses Problem hat den Namen »Knotenüberdeckungsproblem« (engl. *vertex cover problem*) und man kann es mit roher Gewalt lösen. Dazu zählt man einfach alle Teilmengen $S \subseteq V$ mit k Elementen auf und prüft jeweils nach, ob auch alle Kanten des Graphen mit einem Knoten aus S verbunden sind. Wenn man eine passende Teilmenge S findet, gibt man **true** zurück, ansonsten eben **false**. Wenn V insgesamt n Knoten enthält und wir daraus $k \leq n$ Knoten auswählen, so gibt es für diese Wahl $\binom{n}{k}$ Möglichkeiten. Solange das k konstant ist, also unabhängig von n, ist $\binom{n}{k}$ noch mit einem Polynom abschätzbar. Wählen wir jedoch zum Beispiel $k = \frac{n}{2}$, so ist $\binom{n}{n/2}$ ein Produkt aus $\frac{n}{2}$ Faktoren, die jeweils $\geq \frac{n}{2}$ sind, das heißt, es gibt mehr als

$\left(\frac{n}{2}\right)^{\frac{n}{2}}$ viele Teilmengen von V und das ist nicht mehr polynomiell. Böser, böser Algorithmus!

Das nichtdeterministische Orakel

Das Knotenüberdeckungsproblem gehört zu den Problemen, für die keine polynomiellen Algorithmen bekannt sind und von denen man nicht weiß, ob sie nun »gutartig« oder »bösartig« sind. Wie Sie gesehen haben, liegt die Schwierigkeit des Knotenüberdeckungsproblems darin, die richtige Auswahl S von Knoten zu finden. Wenn man sie aber einmal gefunden hat, kann man ganz leicht prüfen, ob die Auswahl S von Knoten auch tatsächlich alle Kanten erreicht. Wie schön wäre es also, wenn wir eine Art »Orakel« hätten, das uns gleichsam auf »magische« Weise eine geeignete Menge S liefern würde! Genauso ein Orakel haben sich die Informatiker ausgedacht und sie nennen es *Nichtdeterminismus*. Ein nichtdeterministischer Algorithmus muss sich in seinem Ablauf nicht in jedem Schritt festlegen. Zur Lösung des Knotenüberdeckungsproblems könnte man zum Beispiel folgenden nichtdeterministischen Algorithmus formulieren:

```
VertexCoverNonDet(V, E, k)      V: Knotenmenge, E: Kantenmenge
                                k: Anzahl der Knoten, die ausgewählt werden dürfen
    S sei eine Teilmenge von V mit k Knoten   ⟸ nichtdeterministisch!
    if S ist mit jeder Kante in E verbunden   S ist gut!
        return true
    else                                      S ist nicht gut
        return false
```

Ob **true** oder **false** ausgegeben wird, hängt natürlich entscheidend von der Wahl der Teilmenge S ab. VertexCoverNonDet legt sich da nicht fest, der Algorithmus ist bei der Wahl nicht determiniert. Der Clou beim Nichtdeterminismus: Wir definieren einfach, dass der Algorithmus »ja« sagt, wenn es eine Wahl von S gibt, bei welcher der Algorithmus **true** zurückgibt, und wenn es keine solche Wahl von S gibt, sagt er »nein«. Das ist so ähnlich, wie wenn der Algorithmus alle Möglichkeiten für S gleichzeitig ausprobieren würde, und wenn eine dieser Mengen S passt, dann ist das Ergebnis »ja« und ansonsten eben »nein«.

Man kann sich auch vorstellen, dass der Algorithmus die Fähigkeit besitzt, das richtige S zu »raten«, was allerdings nur dann funktioniert, wenn man ihn auch anschließend testen lässt, ob das S tatsächlich richtig ist. Nichtdeterminismus ist also ein ziemlich faules Orakel: Damit es einem keinen Quatsch erzählt, muss man immer alles prüfen, was es sagt.

In der Realität gibt es leider keine nichtdeterministischen Maschinen und selbst die Quantencomputer, von denen im Moment überall die Rede ist, beherrschen nach heutigem Verständnis keinen derartigen Nichtdeterminismus. Andererseits würden sich mit

nichtdeterministischen Algorithmen in polynomieller Zeit ziemlich viele interessante Probleme lösen lassen, für die bislang keine »normalen« – also deterministischen – polynomiellen Algorithmen bekannt sind. Wie zum Beispiel das Knotenüberdeckungsproblem (Abbildung 11.16).

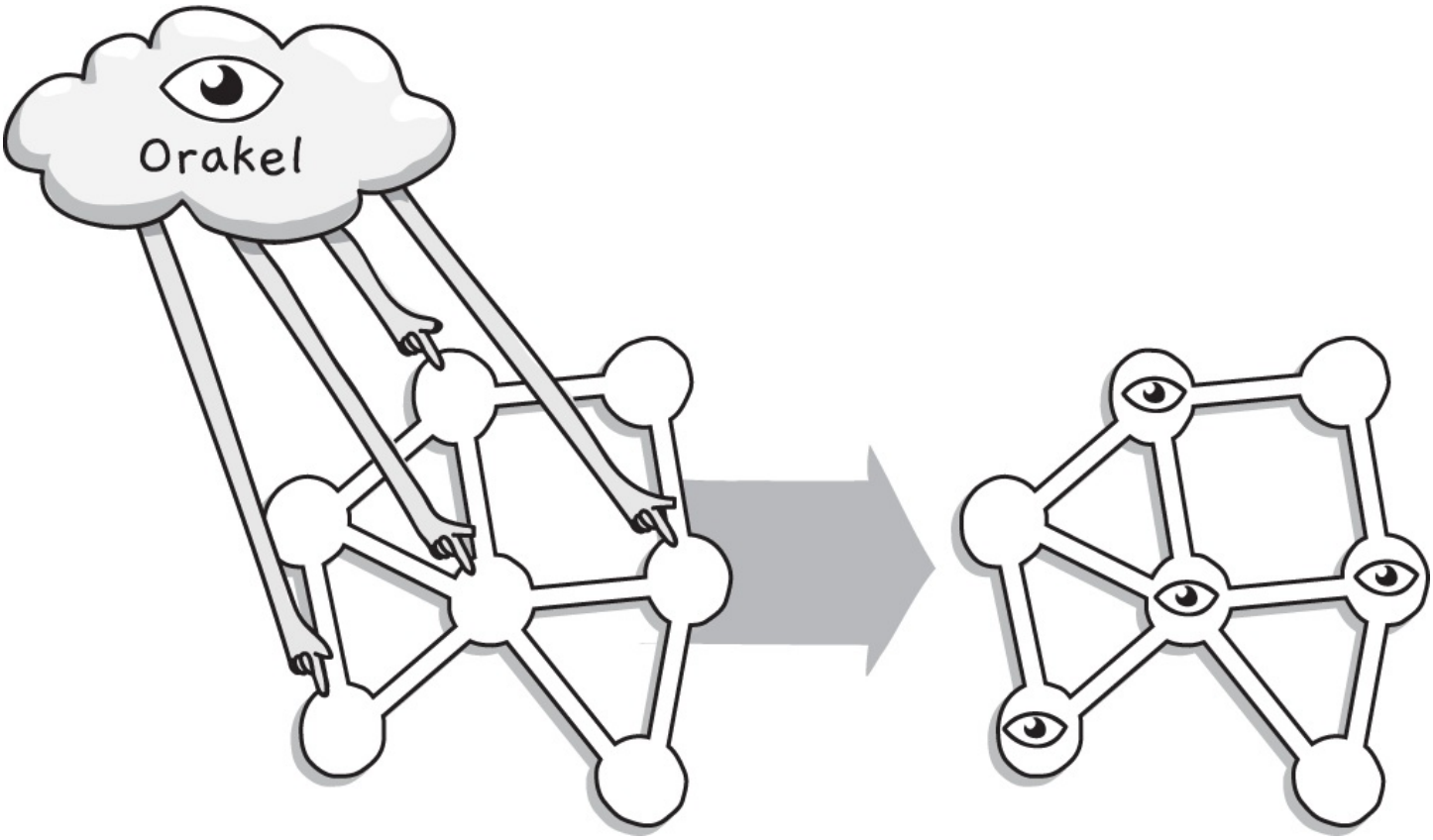


Abbildung 11.16: Ein Orakel wählt die richtigen Standorte für die Kameras aus.

Komplexitätsklassen

Neben den Problemen, die mit einem normalen (also deterministischen) Algorithmus in polynomieller Zeit gelöst werden können, haben wir jetzt also noch die Probleme, die von einem *nicht*deterministischen Algorithmus in polynomieller Zeit gelöst werden können. Je nach Existenz von effizienten Algorithmen der einen oder anderen Art teilt man Probleme in unterschiedliche *Komplexitätsklassen* ein. Bei den Komplexitätsklassen, um die es hier geht, beschränkt man sich dabei üblicherweise auf Entscheidungsprobleme.



Die Komplexitätsklasse **P** (**p**olynomiell) sei die Menge der Entscheidungsprobleme, die von deterministischen Algorithmen in polynomieller Laufzeit gelöst werden können.

Die Komplexitätsklasse **NP** (**n**ichtdeterministisch **p**olynomiell) sei die Menge der Entscheidungsprobleme, die mit *nicht*deterministischen Algorithmen in polynomieller Zeit gelöst werden können.

Probleme in **P** liegen auch in **NP**: Ein deterministischer Algorithmus ist ein

»Sonderfall« eines nichtdeterministischen.

Außerdem gibt es natürlich weiterhin (Entscheidungs-)Probleme, die von keinem polynomiellen Algorithmus, egal ob deterministisch oder nicht, gelöst werden können.

Nun kommt Ihnen das Konzept des Nichtdeterminismus vielleicht etwas seltsam und verschroben vor und es ist ja auch nicht ganz so leicht zu verstehen. Das macht aber nichts, wichtig ist eigentlich nur Folgendes: In **P** liegen die Entscheidungsprobleme, die wir in einigermaßen angemessener Zeit lösen *können*, und in **NP** liegen sehr viele Entscheidungsprobleme, die wir gerne in angemessener Zeit lösen *möchten*. Wie zum Beispiel das Knotenüberdeckungsproblem.

Schwer, schwerer, NP-schwer

Lange haben sich viele kluge Leute die Köpfe darüber zerbrochen, ob **P** $=$ **NP** ist oder nicht. Es hat sich allerdings herausgestellt, dass das eine sehr schwierige Frage ist. Wenn man zum Beispiel beweisen will, dass **P** $=$ **NP** ist, so müsste man ja für *jedes* der unzähligen Probleme in **NP** einen passenden deterministischen Polynomialzeitalgorithmus benennen können, der das Problem löst. Darum sind die Informatiker auf folgenden Kniff gekommen: Wenn es überhaupt ein Problem in **NP** gibt, das *nicht* in **P** liegt, dann doch wohl das *schwerste* Problem in **NP**. Was genau es bedeuten soll, ein »schwerstes« Problem in **NP** zu sein, kann man dabei so definieren:



Ein Problem A heiße **NP-schwer** (engl. »*NP-hard*«), wenn man jedes Problem B aus **NP** in polynomieller Zeit in das Problem A übersetzen kann. Wenn es dann noch gelingt, ein NP-schweres Problem A in polynomieller Zeit zu lösen, kann man auch alle Probleme aus **NP** in polynomieller Zeit lösen, man muss sie dazu ja nur in ein A -Problem verwandeln. Das bedeutet: Sobald wir ein NP-schweres Problem A gefunden haben, müssen wir es nur noch mit einem deterministischen Algorithmus in polynomieller Zeit lösen und schon haben wir bewiesen, dass **P** $=$ **NP** ist.

Klingt nach einem guten Plan. Tatsächlich war bald das erste NP-schwere Problem (das Erfüllbarkeitsproblem) gefunden und danach wurden immer neue NP-schwere Problem entdeckt. Einige davon werden in diesem Buch besprochen (siehe Tabelle 11.1) – und es wird Sie kaum überraschen: Das Knotenüberdeckungsproblem ist ebenfalls NP-schwer. Allerdings ist der Plan nicht ganz aufgegangen, denn leider ist es noch niemandem gelungen, auch nur ein einziges NP-schweres Problem mit einem deterministischen Algorithmus in polynomieller Zeit zu lösen. Das beweist natürlich nicht, dass das nicht geht, aber es demonstriert doch zumindest, dass es sich bei NP-schweren Problem um wirklich *sehr* schwere Probleme handelt.

NP-schweres Problem	*Kapitel*

Rucksackproblem (*knapsack problem*)	8
ganzzahlige lineare Optimierung (*Integer Linear Programming*, ILP)	8
Knotenüberdeckungsproblem (*vertex cover problem*)	11
Problem des Handlungsreisenden (*traveling salesman problem*, TSP)	14
Wechselgeldproblem (*change-making problem*)	14
Mengenüberdeckungsproblem (*set cover problem*)	14

Tabelle 11.1: NP-schwere Probleme in diesem Buch



NP-schwere Probleme sind schwer. Wenn Sie also für ein NP-schweres Problem keinen effizienten Algorithmus finden können, dann seien Sie beruhigt: Das hat bislang noch niemand geschafft.

Weil es schon so viele versucht haben, bislang aber noch alle gescheitert sind, glauben heutzutage die meisten Informatiker, dass man NP-schwere Probleme mit deterministischen Algorithmen gar nicht in polynomieller Zeit lösen kann. Wenn das stimmt, dann ist **P** $\neq$ **NP** (siehe Abbildung 11.17).

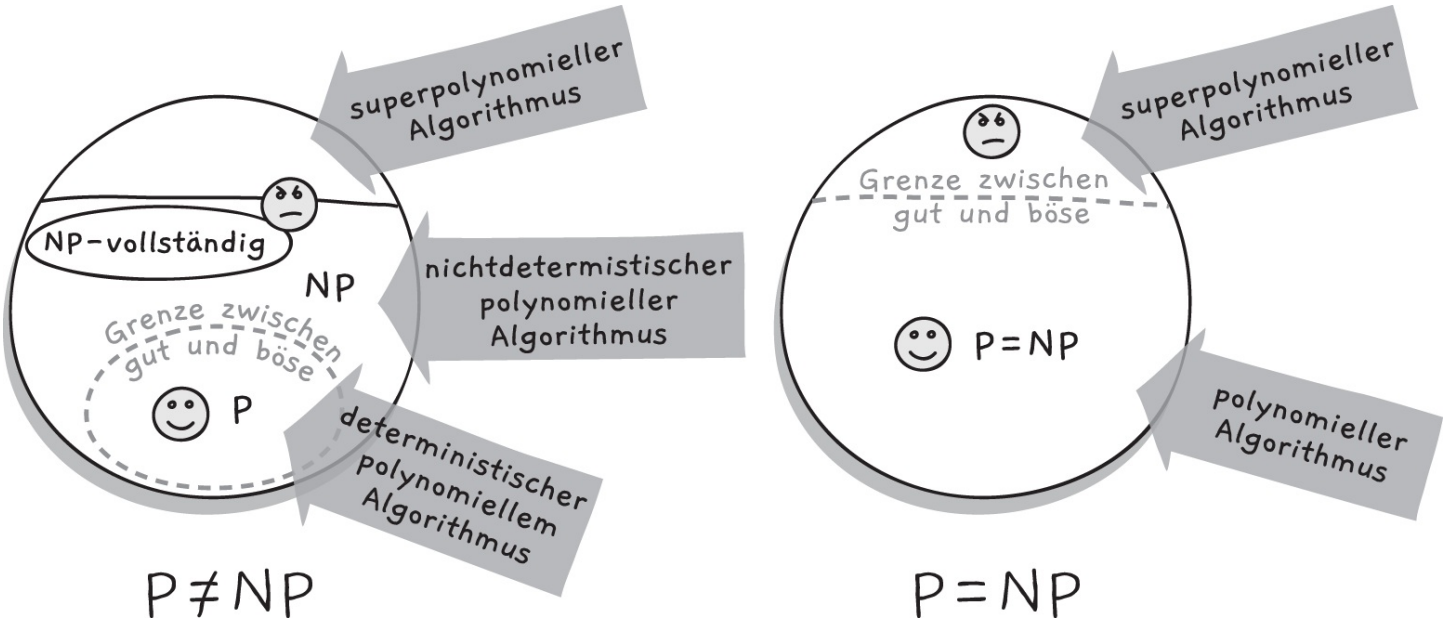


Abbildung 11.17: Wenn P = NP gilt, sind viel mehr Probleme gutartig.



Neben dem Begriff »NP-schwer« ist auch der Ausdruck »*NP-vollständig*« (engl. »*NP-complete*«) gebräuchlich. So werden NP-schwere Probleme genannt, die selbst in der Komplexitätsklasse **NP** liegen. Bei NP-vollständigen Problemen hätte man noch die besten Chancen, sie in Polynomialzeit lösen zu können, denn wenn ein Problem noch nicht einmal in **NP** ist, liegt es ja erst recht nicht in **P** und das macht eine effiziente Lösung doch recht unwahrscheinlich, zumindest bei Entscheidungsproblemen. Sobald man aber die Hoffnung aufgegeben hat, jemals NP-

schwere Probleme in polynomieller Zeit lösen zu können, verliert der Begriff der NP-Vollständigkeit allerdings an Bedeutung. Wir verwenden darum in diesem Buch zumeist einfach den Ausdruck »NP-schwer«.

Wie man mit schweren Problemen umgeht

Für manche Probleme gibt es nur Lösungen, die viel Anstrengung kosten. Wir können also ganz beruhigt sein, wenn wir für sie keine besseren Lösungsansatz finden, als rohe Kraft. Die Schraube soll in die Wand? Schweres Problem: Hammer raus und draufgehauen! Es ist sicher nicht sinnvoll, das Unmögliche zu versuchen und über eine bessere Methode nachzudenken, wenn es keine gibt. Das darf aber nicht dazu verführen, die Flinte zu früh ins Korn zu werfen, und man muss dabei auch nicht nur herumsitzen und auf Inspiration warten.

NP-schwer $\neq$ hoffnungslos

Angenommen, Sie sollen ein algorithmisches Problem lösen und stellen fest, dass es NP-schwer ist. In diesem Fall ist es vielleicht nicht die beste Idee, weiter nach Algorithmen zu suchen, die das Problem in polynomieller Zeit lösen können. Trotzdem müssen Sie noch nicht aufgeben, denn die Lage ist bei Weitem nicht hoffnungslos. Hier ist eine kleine Liste mit Ideen, wie Sie weiter vorgehen könnten.

Problembereich einschränken

Überlegen Sie zunächst, ob Ihr Problem tatsächlich NP-schwer ist. Das Problem, das Sie eigentlich lösen wollten, entstammt ja vermutlich einer Anwendung, zum Beispiel: »Wo soll ich die Kameras anbringen, um meinen Irrgarten komplett überwachen zu können?«. Um dieses Problem zu lösen, haben Sie es umformuliert, zum Beispiel in ein Graphenproblem, das sich dann leider als NP-schwer herausgestellt hat. Das bedeutet aber nicht, dass ihr ursprüngliches Kameraproblem ebenfalls NP-schwer ist. Vielleicht haben Sie ja ein einfaches Problem in ein schwieriges verwandelt. Das kann zum Beispiel leicht passieren, wenn Sie das Problem allgemeiner formuliert haben, als es Ihre Anwendung tatsächlich erfordert. Viele Labyrinthe sind nämlich gar keine allgemeinen Graphen, sondern schlichte Bäume. Wenn man sich auf Bäume beschränkt, ist das Knotenüberdeckungsproblem aber gar nicht mehr NP-schwer, sondern in polynomieller Zeit lösbar.

So kann man NP-schwere Probleme weniger schwer machen: Man legt sich auf bestimmte Arten von Eingaben fest. Also zum Beispiel nur auf Bäume. Oder man beschränkt beim Knotenüberdeckungsproblem die Anzahl der erlaubten Kameras auf eine möglichst kleine Konstante k. Das nennt man »Parametrisierung«: Was zuvor noch von der Eingabe abhing oder durch den Algorithmus bestimmt werden musste, wird nun einfach festgelegt und dadurch vereinfacht sich eventuell das Problem. Wenn zum Beispiel nur $k = 1$ erlaubt ist, müssten alle Kanten mit einem einzelnen Knoten

verbunden sein, um den Irrgarten überwachen zu können, und ob das der Fall ist, lässt sich ganz leicht prüfen.



Das Knotenüberdeckungsproblem in Bäumen

Für Bäume lässt sich das Knotenüberdeckungsproblem wie folgt ziemlich einfach lösen (siehe Abbildung 11.18): Zunächst identifiziert man alle Blätter im Baum. In einem Baum führt zu jedem Blatt nur eine Kante, und um diese Kante zu überwachen, muss man entweder im Blatt eine Kamera aufstellen oder in dem Elternknoten, mit dem diese Kante noch verbunden ist. Wir wählen die Elternknoten, denn Elternknoten können mit mehreren Blättern verbunden sein und dann sparen wir Kameras. Nun entfernen wir sowohl die Blätter als auch ihre Elternknoten aus dem Baum, wodurch ein oder mehrere Restbäume entstehen. Dabei entfernen wir auch alle Kanten, die mit den Blättern oder Elternknoten verbunden sind, aber das macht nichts, denn alle diese Kanten werden ja bereits von den Kameras in den Elternknoten überwacht. Mit den kleineren Bäumen verfahren wir nun genauso wie zuvor und das wiederholen wir so lange, bis alle Bäume verschwunden sind und höchstens Wurzeln übrig geblieben sind. Dieser Algorithmus lässt sich so realisieren, dass er mit einer polynomiellen Laufzeit zurechtkommt.

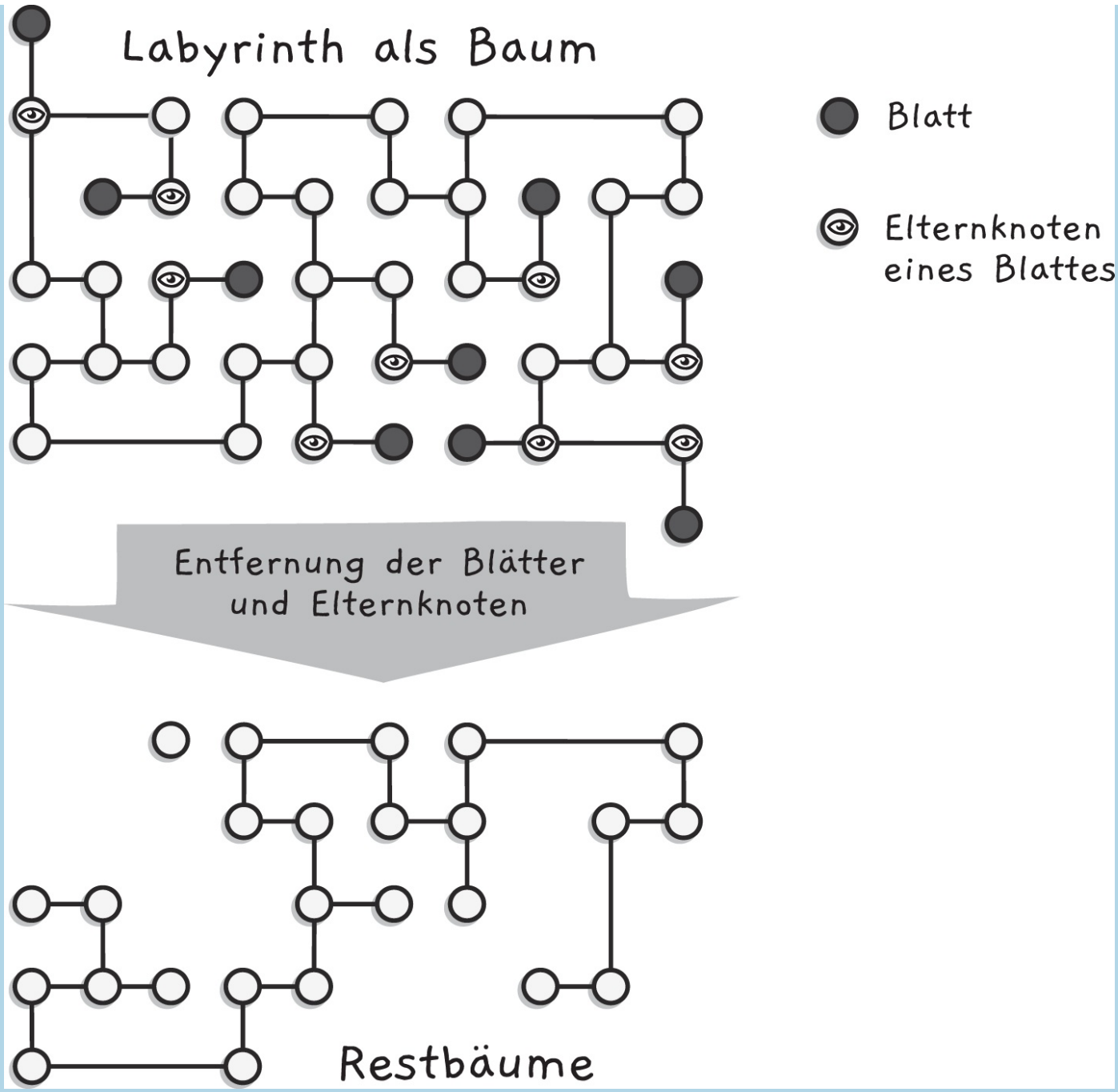


Abbildung 11.18: Lösung des Knotenüberdeckungsproblems im Baum

Näherungslösungen berechnen

Eine weitere Methode, der schwierigen Lösung eines NP-schweren Problems zu entgehen, besteht darin, es nur näherungsweise zu lösen. Das kann insbesondere bei NP-schweren Optimierungsproblemen sinnvoll sein. Wenn man statt der minimalen Anzahl von 100 Kameras, die man für die Überwachung eines großen Irrgartens bräuchte, 102 Kameras verwendet, dann ist das zwar nicht optimal, aber immerhin noch so nah dran, dass wir damit vielleicht noch zufrieden sein können. Eine mehr oder weniger gute Heuristik gibt es für fast alle NP-schweren Probleme.

Schwieriger wird es allerdings, wenn man von seinem Näherungsverfahren erwartet, dass

es einen garantiert nicht zu weit vom Optimum wegführt, denn NP-schwere Probleme lassen sich nicht alle gleich gut approximieren. Für einige NP-schwere Probleme gibt es Methoden, mit denen man beliebig gute Näherungen an das Optimum erreichen kann, und das in polynomieller Zeit. Für andere NP-schwere Probleme hingegen sind gute Näherungen mitunter so schwer zu finden, dass man das Problem auch gleich exakt lösen könnte, anstatt es nur anzunähern.



In Kapitel 8 wird ein *Approximationsschema* am Beispiel des Rucksackproblems gezeigt (siehe den Abschnitt *Sich dem Optimum annähern* in Kapitel 8). Mehr über Approximationsalgorithmen erfahren Sie in Kapitel 14.

Superpolynomielle Laufzeiten beschleunigen

Wenn man partout nicht darum herumkommt, ein NP-schweres Problem in all seiner Gewichtigkeit lösen zu müssen, so wird man sich dafür wohl mit einem superpolynomiellen Algorithmus begnügen müssen. Das heißt aber nicht, dass simple rohe Gewalt für die Lösung des Problems auch immer schon das Nonplusultra wäre. Nicht jede exponentielle Laufzeit ist nämlich gleich schlimm und zwischen den Laufzeitklasse $\mathcal{O}(2^n)$ und $\mathcal{O}(1.1^n)$ besteht ein großer Unterschied. Auch in diesem Fall kann es sich also durchaus lohnen, seinen Verstand ein wenig anzustrengen und nach möglichst guten Lösungsansätzen für das Problem zu suchen.



Als nützliche Methoden zu Beschleunigung der Lösung NP-schwerer Probleme haben sich unter anderem Verzweigen und Begrenzen (*branch and bound*, in diesem Kapitel) sowie dynamisches Programmieren (Kapitel 13) herausgestellt.

Mit manchen NP-schweren Problemen haben sich Fachleute auch schon intensiv beschäftigt und dann gibt es mitunter auch recht gute Software zur Lösung dieser Probleme. In Kapitel 8 sehen Sie, wie man zum Beispiel das Rucksackproblem in ein ganzzahliges lineares Optimierungsproblem (ILP) verwandeln könnte, um es anschließend mit einem »ILP-Solver« lösen zu können.

Gute Ideen sind kein Hexenwerk

Das Nachdenken über Algorithmen ist ein kreativer Akt, ganz egal ob man es mit gut- oder mit bösartigen Problemen zu tun hat. Aber dabei ist man nicht ausschließlich auf sein angeborenes Genie angewiesen. Es gibt schon gewisse Tricks und Denkmuster, denen man folgen kann, um so seiner Genialität ein wenig auf die Sprünge zu helfen. Die erschöpfende Suche ist eine davon. Man nennt diese Tricks und Denkmuster *algorithmische Techniken* und es geht bei ihnen um allgemeine Ansätze, Paradigmen, Strategien, Muster bei der Lösung algorithmischer Probleme.

Die erschöpfende Suche kann hier als Beispiel dafür dienen, wie die Gedanken in die geordneten Bahnen einer Lösungsstrategie geleitet werden können. Man handelt die

Thematik systematisch anhand übersichtlicher Fragestellungen ab:

- ✔ Lösungsmenge: Was charakterisiert eine Lösung des Problems?
- ✔ Suchraum: Zu welcher größeren Menge gehört die Lösungsmenge?
- ✔ Mit welcher Methode können alle Elemente des Suchraums erzeugt werden?
- ✔ Können die Elemente des Suchraums baumartig aus potenziellen Teillösungen aufgebaut werden?
- ✔ Wenn ein baumartiger Aufbau des Lösungsraums möglich ist:
    - Kann man einer Teillösung eventuell frühzeitig ansehen, ob sie zu einer Lösung führen kann? Wenn ja, dann setze Backtracking ein.
    - Handelt es sich um ein Optimierungsproblem und kann man auf Basis einer Teillösung Aussagen über die Qualität der aus ihnen erzeugten Gesamtlösungen treffen, dann setze Verzweigen und Begrenzen ein.

Bei der algorithmischen Umsetzung der Lösungsideen kann man ebenfalls nach erprobten Mustern vorgehen:

- ✔ Sind die Grenzen des Suchraums statisch bekannt (Konstanten im Quellcode des Programms), dann verwende geschachtelte Schleifen.
- ✔ Wenn nicht, verwende ein Rekursionsmuster für Backtracking.

Ein derart systematisches Vorgehen ist aber nicht auf die erschöpfende Suche beschränkt. Im nächsten Kapitel geht es um die algorithmische Technik *Teilen und Herrschen*. Auch das ist als eine Anleitung zum Nachdenken über Problemlösungen zu verstehen: Würde es etwas helfen, wenn das gleiche Problem für eine kleinere Probleminstanz gelöst wäre? Auch hier mit ermutigendem Optimierungspotenzial. So wie Backtracking die erschöpfende Suche ein bisschen verbessert, so kann Teilen und Herrschen mit dynamischer Programmierung verbessert werden. In dem Fall ist das Verbesserungspotenzial sogar richtig dramatisch.

Auch *Gier* hilft, nicht nur im sonstigen Leben, sondern auch beim Nachdenken über Algorithmen. Sie ist eine weitere algorithmische Technik. Die Gedanken werden dabei in Richtung der Frage gelenkt, was bringt mich hier und jetzt spontan meinem Ziel näher? Lassen wir uns also weiter auf dieses spannende Thema ein.

Kapitel 12

Teilen und Herrschen

IN DIESEM KAPITEL

Teilen und Herrschen

Mastertheorem

Binärsuche

Aufgaben auf Mitarbeiter abwälzen

Manche Probleme sind so groß, dass es besser ist, sie erst einmal in kleinere Teilprobleme aufzuteilen. Wenn man dann auch noch schlau ist und die Lösung dieser Teilprobleme auf andere abwälzt, kann man vielleicht sogar schon am frühen Nachmittag Feierabend machen und den Rest des Tages im Freibad verbringen. Dass von dieser Form der Faulheit nicht nur der Einzelne profitiert, sondern sich damit in manchen Fällen auch der Arbeitsaufwand insgesamt reduzieren lässt, werden Sie in diesem Kapitel sehen.

Das Einwohnermeldeamt von Bürokrazien

Im schönen Land Bürokrazien gibt es ein Einwohnermeldeamt, das für jeden Bürger eine Akte verwaltet. Wann immer eine Regierungsstelle etwas über einen Bewohner Bürokraziens in Erfahrung bringen will, stellt sie eine Anfrage. Zuständig ist immer der Direktor des Amtes – Regierungsanfragen sind schließlich wichtig und darum automatisch Chefsache. Man muss ihm nur einen Namen schicken und er besorgt die Akte. Ein paar Ratschläge für eine effiziente Organisation der Aktenablage könnte das Einwohnermeldeamt gerne dem Kapitel 9 entnehmen. Leider gibt es in Bürokrazien nur wenige Informatiker, dafür umso mehr Beamte und Verwaltungsangestellte. Die Beamten sind für besonders verantwortungsvolle Aufgaben zuständig, was sich unter anderem daran zeigt, dass jeder von ihnen genau zwei Mitarbeiter hat, denen er Weisungen erteilen kann. Diese Untergebenen können ihrerseits Beamte sein oder sie sind Angestellte. Die Angestellten haben keine eigenen Mitarbeiter und müssen ihre Arbeit selbst erledigen.

Eine Anfrage an das Amt läuft so ab: Der Direktor – natürlich ein Beamter – erhält eine Anfrage und leitet sie direkt an seine beiden Mitarbeiter zur Bearbeitung weiter. Sobald einer der Mitarbeiter mit der gesuchten Akte zu ihm zurückkommt, geht sie über den langen Dienstweg an den Auftraggeber. Wie der Direktor verfahren auch alle anderen Beamten: Sie tun nichts weiter als die Anfragen an ihre Mitarbeiter weiterzuleiten und

deren Antworten an den jeweiligen Vorgesetzten zurückzugeben. Ganz unten in der Hierarchie arbeiten die Angestellten. Ihre Arbeit ist nicht viel schwerer als die der Beamten, denn jeder Angestellte verwaltet genau eine Akte. Sobald die Anfrage einen Angestellten erreicht, muss dieser nur auf seine Akte schauen, prüfen, ob sie angefordert worden ist, und sie gegebenenfalls dem Vorgesetzten übergeben (siehe Abbildung 12.1).

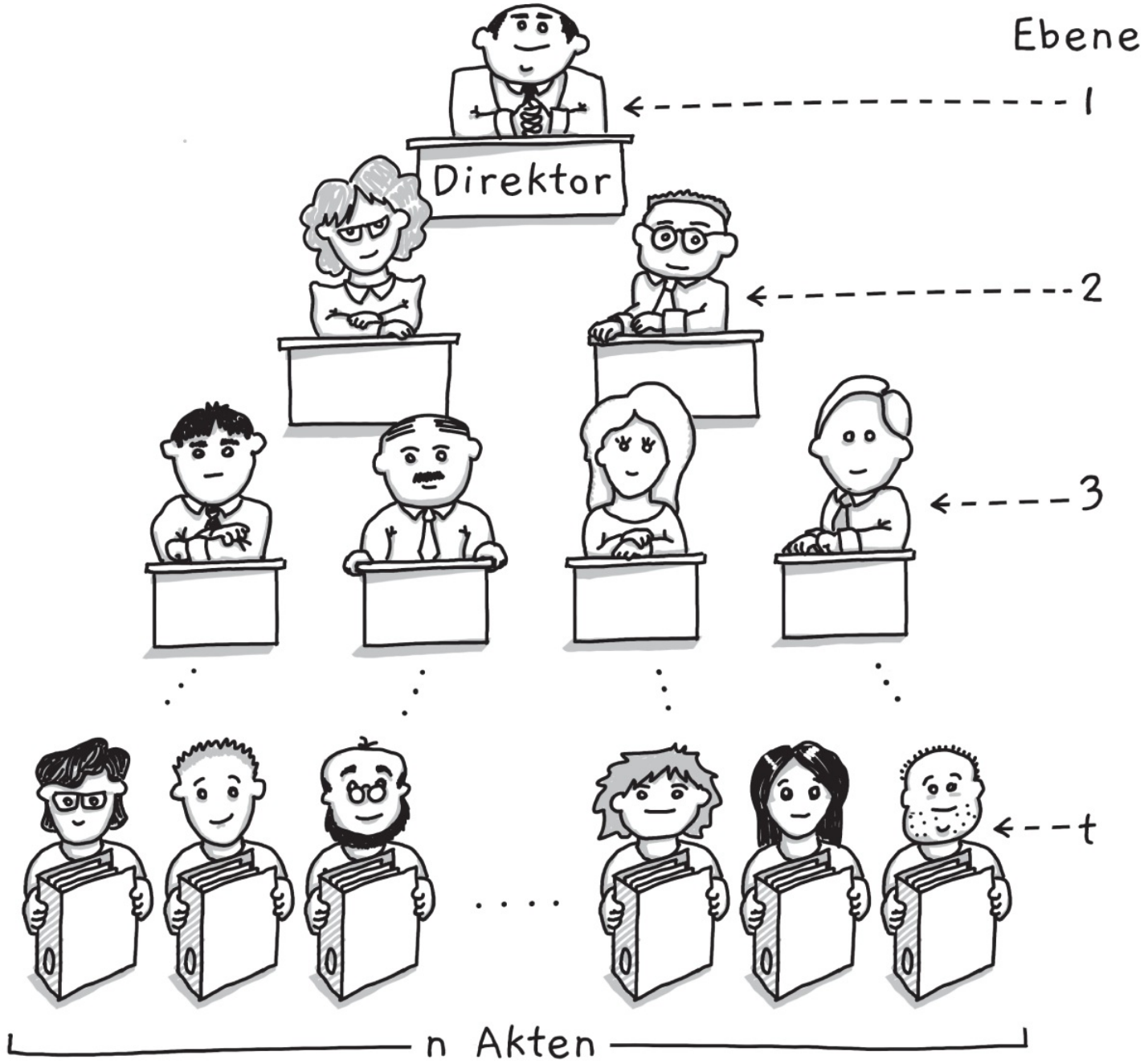


Abbildung 12.1: Hierarchie im Einwohnermeldeamt von Bürokrazien

Wenn das Einwohnermeldeamt also n Akten verwalten soll, so braucht es auch n Angestellte und zusätzlich $n-1$ Beamte im hierarchisch strukturierten Verwaltungsapparat. Dass es gerade $n-1$ Beamte sind, kann man ganz leicht

einsehen: Jeder der n Angestellten soll ja einen Vorgesetzten haben und das muss natürlich ein Beamter sein. Jeder Beamte ist Vorgesetzter von zwei Mitarbeitern des Amtes, muss jedoch andererseits auch selbst einen Vorgesetzten bekommen. Mit jedem neu eingestellten Beamten verringert sich die Anzahl der Mitarbeiter ohne Vorgesetzte also nur um eins. Erst wenn $n-1$-te Beamte eingestellt worden sind, hat nur noch ein Beamter keinen Vorgesetzten und das ist dann der Direktor.

Angesichts dieses Aufwands ist unmittelbar klar, warum Bürokrazien dringend auf Arbeitskräfte aus dem Ausland angewiesen ist.

Das Prinzip Teilen und Herrschen

Obwohl das Einwohnermeldeamt offenbar nicht sehr effizient organisiert ist, steckt dennoch eine nützliche algorithmische Methode dahinter, nämlich das Prinzip »Teilen und Herrschen«.



Bei *Teilen und Herrschen* wird eine Aufgabe gelöst, indem man sie

1. aufteilt,
2. die Teile löst und anschließend
3. aus den Lösungen der Teile die Lösung für die Gesamtaufgabe erstellt.

Für die Lösung der Teile gibt es dann zwei Möglichkeiten: Entweder ist ein Teilproblem klein genug, dass man die Lösung ganz leicht ermitteln kann oder man wendet wiederum – *rekursiv* – »Teilen und Herrschen« an, das heißt, man teilt das Teilproblem in noch kleinere Teilprobleme auf.

Kleines Beispiel: Teilen und Zählen

Hier ist ein einfaches Beispiel für dieses Prinzip: Wir wollen zählen, wie viele Vorkommen eines Wertes x in einem Array a zwischen den Positionen l und r vorkommen mit $l \leq r$, also wie viele der Werte $a[l], a[l+1], \ldots, a[r]$ gleich x sind. Das können wir mit »Teilen und Herrschen« folgendermaßen anstellen:

```
countTH(a, l, r, x)
    a: Array von n Werten; l, r: Positionen in a, l ≤ r
    x: gesuchter Wert
    if l = r then                          Abbruchbedingung: nur ein Element im Array
        if a[l] = x then return 1          wenn dieses Element = x ist, gib 1 zurück ...
        else return 0                      ... sonst gibt 0 zurück
    else
        m ← (l+r)/2 aufgerundet            m: Mittelposition im Array
        c_l ← countTH(a, l, m − 1, x)      zähle Vorkommen von x im linken Teilarray
        c_r ← countTH(a, m, r, x)          zähle Vorkommen von x im rechten Teilarray
        return c_l + c_r                   gib Gesamtzahl der Vorkommen von x zurück
```

An diesem Algorithmus können Sie die folgenden Punkte beobachten, die für Teilen-und-Herrschen-Algorithmen typisch sind:

- ✔ Der Algorithmus countTH ist rekursiv, das heißt, er ruft »sich selbst« erneut auf. Obwohl man jeden rekursiven Algorithmus auch iterativ – also ganz ohne Rekursion – formulieren kann, bietet sich beim intrinsisch rekursiven Prinzip Teilen und Herrschen eine rekursive Konstruktion natürlich an.
- ✔ Wie jeder rekursive Algorithmus hat auch countTH eine Abbruchbedingung. Bei jedem rekursiven Aufruf wird der Algorithmus auf einem kleineren Arrayabschnitt aufgerufen. Sobald das Array nur noch einen Wert enthält, lohnt sich ein weiteres Aufteilen aber nicht mehr. In der Hierarchie des Einwohnermeldeamts ist man dann ganz unten angekommen, bei den Verwaltungsangestellten, die keine Untergebenen haben. Da heißt es dann: Selbst arbeiten und den Wert im Array mit x vergleichen.
- ✔ Ein Aufruf von countTH an einem Blatt des Rekursionsbaums, der also nur auf einem einzelnen Wert des Arrays a arbeitet, »weiß« gar nicht, wie groß das Array ursprünglich mal gewesen ist. Die Arbeit, die ein Verwaltungsangestellter zu tun hat, hängt also nicht von der Eingabelänge n ab und lässt sie sich darum in $\mathcal{O}(1)$ erledigen.
- ✔ Die Beamten an den inneren Knoten des Rekursionsbaums haben im Fall von countTH auch nicht viel zu tun. Der *Zusatzaufwand*, also die Zeit, die der Algorithmus abzüglich der rekursiven Aufrufe braucht, ist bei countTH ebenfalls $\mathcal{O}(1)$. Bei anderen Algorithmen, die nach dem Prinzip vom Teilen und Herrschen arbeiten, ist das mitunter anders und wirkt sich dann natürlich auf die Laufzeit aus, die der Algorithmus insgesamt benötigt.



Viele Leute finden rekursive Algorithmen ziemlich schwer zu verstehen. Sobald man nämlich versucht, einem Algorithmus über mehrere verschachtelte Aufrufe

hinweg in die Untiefen der Rekursion zu folgen, verliert man früher oder später fast unweigerlich den Überblick. Unser Tipp: Denken Sie wie ein Beamter aus Bürokrazien! Wer als Beamter in Bürokrazien Aufgaben an seinen Untergebenen delegiert, kann darauf zählen, dass der Untergebene seiner Pflicht getreulich nachkommen wird. Wie er das konkret anstellt, ist seine Sache und interessiert seinen Chef nicht. Der Chef muss nur wissen, wie man aus korrekten Resultaten der Untergebenen ein korrektes Resultat für das Gesamtproblem erstellen kann. Konkret gesagt: Wenn Sie einen rekursiven Algorithmus verstehen wollen, so folgen Sie *nicht* den rekursiven Aufrufen, sondern gehen Sie darüber hinweg in der Annahme, dass die rekursiven Aufrufe das tut, was sie tun sollen. Wenn jeder Aufruf korrekt funktioniert, wenn alle seine rekursiven Aufrufe (die »Untergebenen«) korrekt funktioniert haben, so ist auch der ganze Algorithmus korrekt. Vorausgesetzt natürlich, dass der rekursive Abstieg auch irgendwann einmal endet; also achten Sie außerdem darauf, ob es im Algorithmus eine wirksame Abbruchbedingung gibt, die dafür sorgt, dass der Algorithmus anhält.

Laufzeiten bei Teilen und Herrschen

Die Laufzeit von countTH lässt sich noch relativ leicht ermitteln: Wird countTH mit einem Array der Länge n aufgerufen, so entspricht die Gesamtzahl der (rekursiven) Aufrufe von countTH der Zahl der Knoten in einem Binärbaum mit n Blättern. Wie wir oben schon argumentiert haben, brauchen wir zusätzlich zu den n »Verwaltungsangestellten« an den Blättern noch $n-1$ »Beamte« an den inneren Knoten, countTH wird also insgesamt ($n+n-1=2n-1$)-mal aufgerufen. Jeder Aufruf benötigt eine Zeit von $\mathcal{O}(1)$, also läuft der Algorithmus insgesamt in $\mathcal{O}(2n-1)=\mathcal{O}(n)$. Dieselbe Laufzeitklasse würden wir natürlich auch erreichen, wenn wir einfach in einer Schleife über das Array laufen und die Vorkommen von x zählen. Tatsächlich lohnt es sich bei diesem Problem gar nicht, das Prinzip Teilen und Herrschen zu benutzen. Es gibt jedoch etliche Anwendungen, die sehr von diesem algorithmischen Prinzip profitieren; verschiedene Beispiele sehen Sie im Abschnitt *Weitere Beispiele für Teilen und Herrschen*.

Die Laufzeit eines Teilen-und-Herrschen-Algorithmus wird maßgeblich durch die folgenden Größen bestimmt:

- ✔ Die Anzahl a der Teile, in die man das Ursprungsproblem aufteilt. Bei den meisten Algorithmen ist $a=2$, das heißt, man teilt das Problem in zwei Teile auf. Manchmal ist aber auch eine Aufteilung in mehr als zwei Teile sinnvoll. Beim Problem der Matrixmultiplikation, über das wir gleich noch mehr erfahren werden (siehe Abschnitt *Matrizen multiplizieren*), sind es beispielsweise $a=4$ Teile.
- ✔ Den Faktor b, um den die Teilprobleme kleiner als das Ursprungsproblem sind. Im

Beispiel oben ist $b = 2$: Wenn wir countTH mit einem Array der Länge n aufrufen, so ruft der Algorithmus sich rekursiv getrennt für die linke und für die rechte Hälfte auf, also jeweils für Arrays der Länge $\frac{n}{2}$.

- ✔ Den Zusatzaufwand $f(n)$, also die Zeit, die ein Aufruf bei der Eingabelänge n benötigt, *abzüglich* der Zeit, die für die rekursiven Aufrufe selbst aufgewendet wird. Das ist zum Beispiel der Aufwand, den der Algorithmus dazu benötigt, die Abbruchbedingung zu checken, die Aufgabe aufzuteilen und am Ende aus den Teillösungen die Gesamtlösung zu berechnen.

Man kann für die Laufzeit $T(n)$ des Teilen-und-Herrschen-Algorithmus beim Aufruf mit Eingabelänge n folgende Gleichung aufstellen:

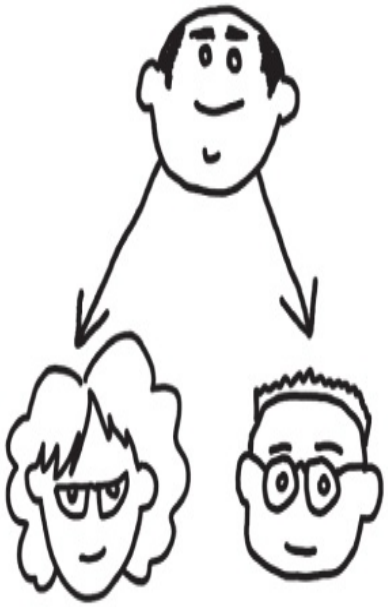
$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n).$$

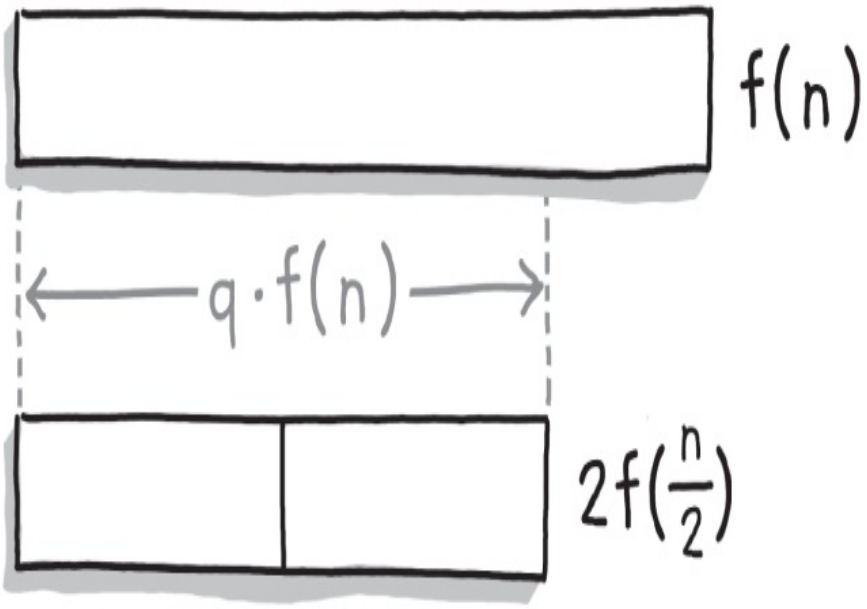
Auf Deutsch gesagt: »Die Laufzeit bei Eingabelänge n für die rekursiven Aufrufe setzt sich aus folgenden Termen zusammen: erstens a -mal die Laufzeit bei Eingabelänge $\frac{n}{b}$ und zweitens der Zusatzaufwand $f(n)$.« Das Komplizierte an dieser Gleichung ist, dass sie genauso rekursiv ist wie der Algorithmus selbst. Das T kommt sowohl links als auch rechts von der Gleichung vor, allerdings mit verschiedenen Argumenten, einmal mit n und einmal mit $\frac{n}{b}$. Eine solche *Rekursionsgleichung* zu lösen, ist nicht unbedingt leicht. Da gibt es verschiedene Methoden. Am einfachsten ist es, das sogenannte *Mastertheorem* anzuwenden, das Sie im Folgenden kennenlernen werden.

Das Mastertheorem

Die wichtigste Frage bei der Bestimmung der Laufzeit ist: »Wer arbeitet mehr? Der Chef oder alle seine direkten Untergebenen zusammen?« Betrachtet man alle Mitarbeiter, die auf der gleichen Stufe in der Hierarchie arbeiten, als eine Gruppe, so kann man die Frage auch anders formulieren: »Welche Hierarchieebene leistet die meiste Arbeit?«. Bei countTH ist die Antwort leicht: Der Chef arbeitet so viel, wie *jeder einzelne* seiner Untergebenen, also *weniger* als seine beiden Untergebenen *zusammen*. Da in den unteren Ebenen der Hierarchie mehr Leute arbeiten als weiter oben, leisten sie also auch insgesamt mehr Arbeit. Denkbar wäre aber auch, dass der Chef *gleich viel* arbeitet wie alle seine Untergebenen zusammen oder dass er sogar *mehr arbeitet* als sie. Allerdings wäre der Direktor des Einwohnermeldeamts in Bürokrazien sicherlich gegen diese für ihn ungünstige Form der Arbeitsteilung. Schauen wir uns die verschiedenen Fälle doch einmal genauer an.

Fall 1: Der Chef arbeitet mehr





Angenommen, der Chef würde *mehr* Arbeit leisten als alle seine direkten Untergebenen zusammen. Nehmen wir einmal an, dass die Untergebenen zusammen höchstens einen festen Bruchteil $q < 1$ so viel wie ihr Chef arbeiten. Wenn zum Beispiel $q = 0,6$ ist, dann leisten die Untergebenen nur 60 % der Arbeit ihres Chefs, bei zwei Untergebenen müsste also zum Beispiel jeder nur 30 % so viel tun wie sein Chef, bei drei Untergebenen jeder nur jeweils 20 %. Wir wollen annehmen, dass das gleiche q für alle anderen Beamten in der Hierarchie gilt: Das Verhältnis der Arbeit des Chefs zur Gesamtarbeit seiner Untergebenen ist immer kleiner oder gleich q und dieses q ist kleiner 1, denn die Untergebenen arbeiten ja zusammen *weniger* als ihr Chef.

Die Gesamtlaufzeit ausgerechnet

Jetzt wollen wir uns einmal anschauen, wie viel Arbeit jede Hierarchieebene im Einwohnermeldeamt zu tun hat. Die Arbeit, die der Direktor (Ebene 1) selbst zu tun hat, hatten wir den Zusatzaufwand $f(n)$ genannt. Auf Ebene 2 leisten die Mitarbeiter dann also insgesamt eine Arbeit von höchstens $q \cdot f(n)$, also maximal ein q-tel der Arbeit ihres Chefs, des Direktors. Noch eine Ebene weiter, auf Ebene 3, tut jeder wieder nur höchstens den q-ten Bruchteil der Arbeit seines jeweiligen Chefs, also insgesamt wird dort nicht mehr als $q^2 \cdot f(n)$ geleistet. Noch eine Ebene tiefer sind es $q^3 \cdot f(n)$, noch eine tiefer $q^4 \cdot f(n)$ und so weiter. Wenn wir nun die Gesamtarbeit über alle Hierarchieebenen hinweg aufsummieren, so bekommen wir also

$$\begin{aligned} T(n) &\leq f(n) + q \cdot f(n) + q^2 \cdot f(n) + q^3 \cdot f(n) + q^4 \cdot f(n) + \ldots \\ &= f(n) \cdot (1 + q + q^2 + q^3 + q^4 + \ldots) \end{aligned}$$

Die Summe endet natürlich, sobald die unterste Ebene – die Ebene der Angestellten – erreicht ist. Wenn wir aber einmal so tun, als würde es in der Hierarchie immer tiefer und tiefer gehen, so wird aus der Summe »$1 + q + q^2 + q^3 + q^4 + \ldots$« eine sogenannte *geometrische Reihe*. Für $q < 1$ gibt es eine Formel, mit der man die Gesamtsumme der geometrischen Reihe leicht ausrechnen kann, nämlich:

$$1 + q + q^2 + q^3 + q^4 + \ldots = \sum_{i=0}^{\infty} q^i = \frac{1}{1-q}.$$

Wenn wir das in die Gleichung oben einsetzen, bekommen wir für die Gesamtlaufzeit

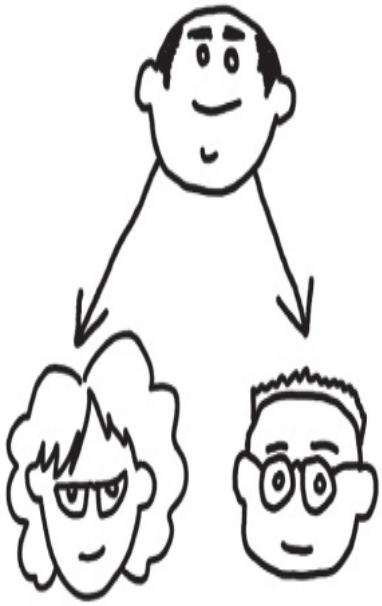
$$T(n) \leq f(n) \cdot \sum_{i=0}^{\infty} q^i = \frac{1}{1-q} \cdot f(n) = \mathcal{O}(f(n))$$

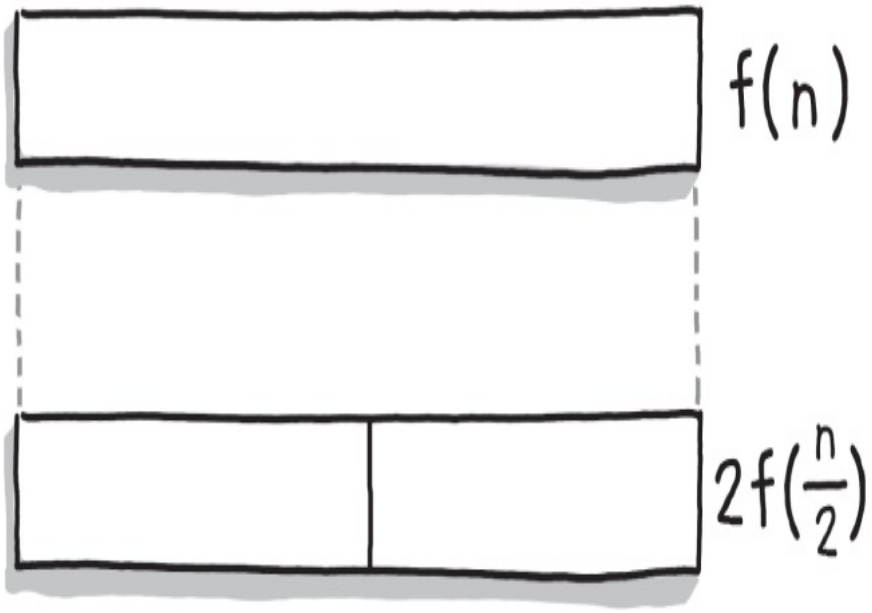
Die Laufzeit $T(n)$ des Algorithmus, also die Gesamtarbeit *aller* Mitarbeiter zusammengenommen, ist dann ein festes Vielfaches der Arbeit $f(n)$ des Direktors. Wenn zum Beispiel $q = 0,6$ ist, dann ist $T(n) \leq \frac{1}{1-q} \cdot f(n) = 2,5 \cdot f(n)$. Da q eine Konstante ist, liegt $T(n)$ damit in der gleichen Laufzeitklasse wie der Zusatzaufwand $f(n)$.



Wenn der Chef *mehr* arbeitet als alle seine direkten Untergebenen zusammen, dann dominiert die Arbeit des Chefs die Gesamtarbeit. In diesem Fall ist die Gesamtlaufzeit $T(n)$ in $\mathcal{O}(f(n))$, das heißt, sie ist höchstens um einen konstanten Faktor größer als der Zusatzaufwand $f(n)$.

Fall 2: Der Chef arbeitet gleich viel





Nun nehmen wir an, dass der Chef genauso viel arbeitet wie seine direkten Untergebenen. Das bedeutet, dass auf jeder Hierarchieebene die gleiche Menge Arbeit geleistet wird. Der Direktor (Ebene 1) leistet ja $f(n)$, in Ebene 2 darunter wird ebenfalls $f(n)$ gearbeitet, die Ebene 3 arbeitet insgesamt auch $f(n)$ und so weiter. Wenn es insgesamt t Hierarchieebenen gibt, so ist die Gesamtarbeit $T(n)$ also $t \cdot f(n)$.

So viele Hierarchieebenen gibt es

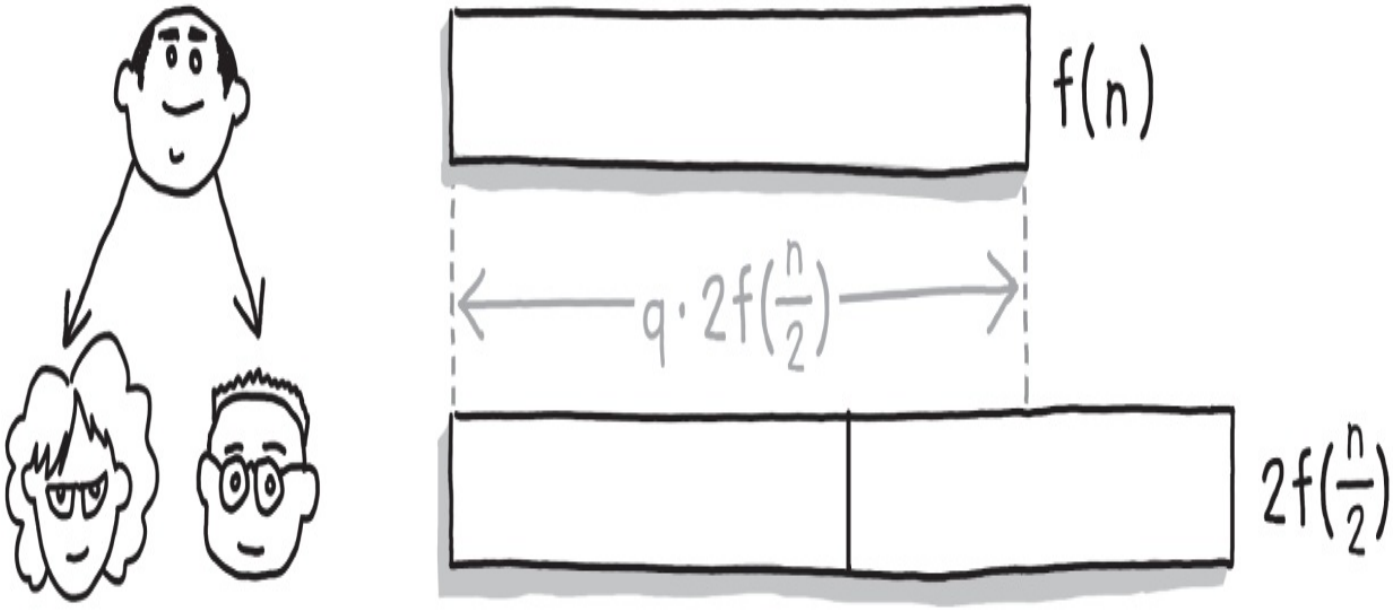
Die Anzahl der Ebenen in der Hierarchie hängt von n ab. Die Länge der Eingabe n wird ja mit jeder Hierarchieebene um den Faktor $b > 1$ kleiner und das wiederholt sich so lange, bis eine Eingabelänge von 1 erreicht ist und die Aufgabe von einem einfachen Angestellten erledigt werden kann. Angenommen, man braucht t Hierarchieebenen, das heißt, n wurde $(t-1)$-mal durch b geteilt. Bei jedem Aufteilen kommt ein weiterer Faktor b dazu, das heißt, wir haben n insgesamt durch b^{t-1} geteilt. Eine Ebene weniger hätte nicht ausgereicht: Wenn wir n nur durch b^{t-2} geteilt hätten, so wäre eine Eingabelänge größer als 1 herausgekommen, das heißt $\frac{n}{b^{t-2}} > 1$. Wenn man das nach t auflöst, so ergibt sich $t < \log_b(n) + 2$.

Dabei ist »$\log_b(n)$« der Logarithmus zur Basis b von n. Die Gesamtarbeit ist mithin

$$T(n) = t \cdot f(n) \leq (\log_b(n) + 2) \cdot f(n) = \mathcal{O}(\log(n) \cdot f(n)).$$

Wenn der Chef *genauso viel* arbeitet wie seine direkten Untergebenen zusammen, dann muss man die Arbeit des Chefs mit der Anzahl der Hierarchieebenen multiplizieren, um den Gesamtaufwand zu bestimmen. Die Anzahl der Hierarchieebenen wächst logarithmisch mit n. In diesem Fall ist die Gesamtlaufzeit $T(n)$ also $\mathcal{O}(\log(n) \cdot f(n))$, das heißt, sie ist um einen $\log(n)$-Faktor größer als der Zusatzaufwand $f(n)$.

Fall 3: Der Chef arbeitet weniger



Lassen Sie uns nun annehmen, dass die direkten Untergebenen des Chefs zusammen mehr arbeiten als der Chef selbst. Diesmal arbeitet der Chef also nur den Anteil q der Gesamtarbeit seiner direkten Untergebenen, wieder mit $q < 1$. Dann lässt sich wieder die gleiche Rechnung aufstellen wie im Fall 1, diesmal aber in der Hierarchie »von unten nach oben«. Nehmen wir einmal an, alle Angestellten in der untersten Ebene haben

zusammen $g(n)$ viel zu tun. Dann wird eine Ebene höher nur $q \cdot g(n)$ geleistet, darüber nur $q^2 \cdot g(n)$ und so weiter. Mit der gleichen Rechnung wie bei Fall 1 kann man dann zeigen, dass die Gesamtarbeit $T(n) \leq \frac{1}{1-q} \cdot g(n)$ ist.

Bleibt noch die Frage zu klären, wie groß $g(n)$ ist. Da jeder Angestellte nur auf einer Eingabe der Länge 1 arbeitet, ist sein Arbeitsaufwand unabhängig von n, also $\mathcal{O}(1)$. Der Gesamtaufwand $g(n)$ hängt also vor allem von der Anzahl der Angestellten ab, die auf der untersten Hierarchieebene des Amtes arbeiten.



So viele Angestellte arbeiten auf der untersten Ebene

Jeder Beamte hat a direkte Untergebene, das heißt, die Zahl der Mitarbeiter pro Hierarchieebene wächst mit jeder Ebene um den Faktor a. Wie wir oben gesehen haben, ist die Anzahl t der Hierarchieebenen kleiner als $\log_b(n) + 2$, es gibt also höchstens $a^{t-1} < a^{\log_b(n)+1}$ Angestellte auf der untersten Ebene t. Mit etwas fieser Logarithmenrechnerei bekommt man am Ende Folgendes heraus:

$$\text{Anzahl der Angestellten} \leq a \cdot n^{\log_b(a)} = \mathcal{O}(n^{\log_b a})$$



Wenn der Chef *weniger* arbeitet als seine direkten Untergebenen zusammen, wird die Gesamtarbeit durch die Anzahl der Angestellten in der untersten Hierarchieebene bestimmt. In diesem Fall ist die Gesamtlaufzeit $T(n)$ dann $\mathcal{O}(n^{\log_b a})$.

Gibt es noch weitere Fälle?

Das Mastertheorem wird vermutlich darum »Mastertheorem« genannt, weil man es bei ganz vielen Gelegenheiten anwenden kann. Dennoch hat es seine Grenzen. Manchmal kann es nämlich passieren, dass zum Beispiel ganz oben in der Hierarchie die Chefs zwar weniger arbeiten als ihre Untergebenen, die Chefs weiter unten in der Hierarchie aber wiederum mehr arbeiten müssen als ihre direkten Untergebenen. Dann passt keiner der drei Fälle des Mastertheorems und Sie können das Mastertheorem folglich nicht benutzen. Wenn Sie in anderen Büchern nachlesen, so finden Sie andere Versionen des Mastertheorems, die etwas umfassender sind als hier beschrieben. Auch diese Versionen können jedoch nicht alle denkbaren Möglichkeiten abdecken und dann bleibt einem nichts übrig, noch einmal selbst über die Laufzeit seines Algorithmus nachzudenken.

So bestimmt man, welcher Fall vorliegt

Der Chef leistet den Zusatzaufwand $f(n)$, während seine a Untergebenen jeweils den Zusatzaufwand $f\left(\frac{n}{b}\right)$ zu erbringen haben, alle Untergebenen zusammen arbeiten also

$a \cdot f\left(\frac{n}{b}\right)$. Um zu bestimmen, welcher der drei Fälle des Mastertheorems vorliegt, müssen wir darum $f(n)$ mit $a \cdot f\left(\frac{n}{b}\right)$ vergleichen. Was von beidem ist größer?



Lassen Sie uns als Beispiel einen polynomiellen Zusatzaufwand von der Form $f(n) = c \cdot n^k$ für beliebige positive Zahlen c und k anschauen. Dann beträgt die Gesamtarbeit der Untergebenen

$$a \cdot f\left(\frac{n}{b}\right) = a \cdot c \cdot \left(\frac{n}{b}\right)^k = c \cdot n^k \cdot \frac{a}{b^k} = f(n) \cdot \frac{a}{b^k}.$$

Je nachdem, ob $\frac{a}{b^k}$ kleiner, gleich oder größer als 1 ist, haben wie es dann mit Fall 1, Fall 2 oder Fall 3 zu tun.

Lassen Sie uns das für $a = b = 2$ nochmal zusammenfassen: Weil es nicht so generell ist wie das Mastertheorem, geben wir ihm spaßeshalber den Namen »Bachelortheorem«.

Das »Bachelortheorem«:

Bei einem Teilen-und-Herrschen-Algorithmus, bei dem die Eingabe immer möglichst gleichmäßig in zwei Teile aufgeteilt wird, ergibt sich bei polynomiellem Zusatzaufwand der Form $f(n) = c \cdot n^k$ die asymptotische Laufzeit $T(n)$ bei Eingabelänge n wie folgt:

$$T(n) = \begin{cases} \mathcal{O}(n^k) & \text{, wenn } k > 1 \quad \text{(Fall 1)} \\ \mathcal{O}(n \cdot \log n) & \text{, wenn } k = 1 \quad \text{(Fall 2)} \\ \mathcal{O}(n) & \text{, wenn } k < 1 \quad \text{(Fall 3)} \end{cases}$$

Die drei Fälle des »Bachelortheorems« sind in Abbildung 12.2 bildlich dargestellt für $k = 2$ (»quadratischer Zusatzaufwand«), $k = 1$ (»linearer Zusatzaufwand«) und $k = 0$ (»konstanter Zusatzaufwand«).

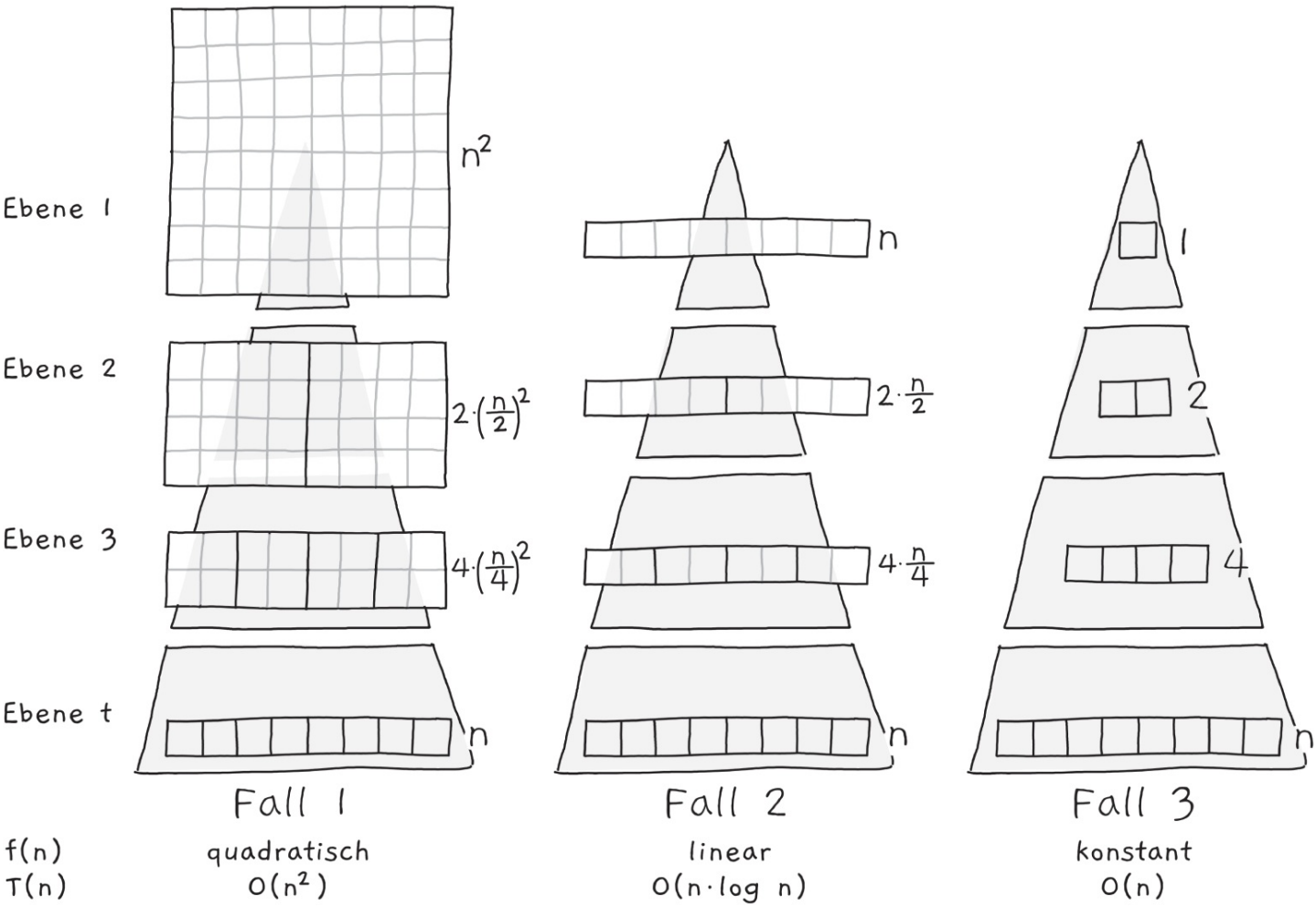


Abbildung 12.2: Die drei Fälle des »Bachelortheorems« für quadratischen, linearen und konstanten Zusatzaufwand

Binärsuche

Jetzt haben wir ausreichend mathematisches Handwerkszeug zusammengesammelt und können uns ein paar Anwendungen des Prinzips Teilen und Herrschen anschauen und dabei das Mastertheorem anwenden. Die erste Anwendung heißt »Binärsuche« oder auch »binäre Suche« und betrifft die schnelle Suche in sortierten Arrays.

In Listen zu suchen, kann eine langwierige Angelegenheit sein: Man muss die Liste Element für Element durchgehen und prüfen, ob man gefunden hat, was man sucht. Steht das gewünschte Element ganz am Ende der Liste oder – noch schlimmer – ist es am Ende gar nicht in der Liste enthalten, so muss man die ganze Liste durchgehen, bevor man mit dem Suchen fertig ist. Bei n Elementen in der Liste hat eine solche *lineare Suche* eine Laufzeit von $\mathcal{O}(n)$.

Der Suchbaum in einfach

In Kapitel 9 finden Sie verschiedene Möglichkeiten, wie man Mengen von Werten geschickter abspeichern kann, nämlich zum Beispiel in einem *Suchbaum*. Man fängt an der Wurzel an, dort steht ein Element, mit dem man seinen Suchwert vergleichen kann.

Wenn man den gesuchten Wert gefunden hat, dann ist man schon fertig. Ansonsten muss man in einem der beiden Unterbäume weitersuchen: Sucht man etwas Kleineres, so muss man im linken Baum schauen, sucht man hingegen etwas Größeres, kann man das nur im rechten Teilbaum finden. Wenn dieser Baum schön balanciert ist, so hat er bei n Knoten eine logarithmische Tiefe und die Suche dauert bloß $\mathcal{O}(\log n)$.

Nun ist ein Suchbaum eine ziemlich aufwendige Konstruktion. Da muss man sich eine Wurzel merken, Werte in Knoten speichern und Zeiger auf linke und rechte Kinder. Doch diese Speicherung als verkettete Datenstruktur lohnt sich auf jeden Fall, wenn man neue Elemente zu der Menge hinzufügen will oder auch Elemente wieder aus ihr löschen möchte. In diesem Fall kann man den Suchbaum umstrukturieren, ohne die restlichen Werte in der Menge verschieben zu müssen, einfach durch »Umhängen« der Zeiger. Manchmal müssen gespeicherte Mengen aber kaum je geändert werden, man will nur in ihnen suchen, und das möglichst effizient. In diesem Fall bietet sich eine andere, unkompliziertere Form der Speicherung an, nämlich die Ablage in einem sortierten Array.

Speichert man die Werte aufsteigend sortiert in einem Array, so kann man nämlich darin suchen, als wäre es ein Suchbaum (siehe Abbildung 12.3). Bei einem perfekt sortierten Suchbaum liegt der Wert an der Wurzel nämlich genau in der Mitte aller in der Menge gespeicherten Werte und diese Stelle kann man in einem sortierten Array leicht finden. Wenn man seine gesuchte Zahl mit dem Wert in der Mitte des Arrays vergleicht, dann gibt es drei Fälle:

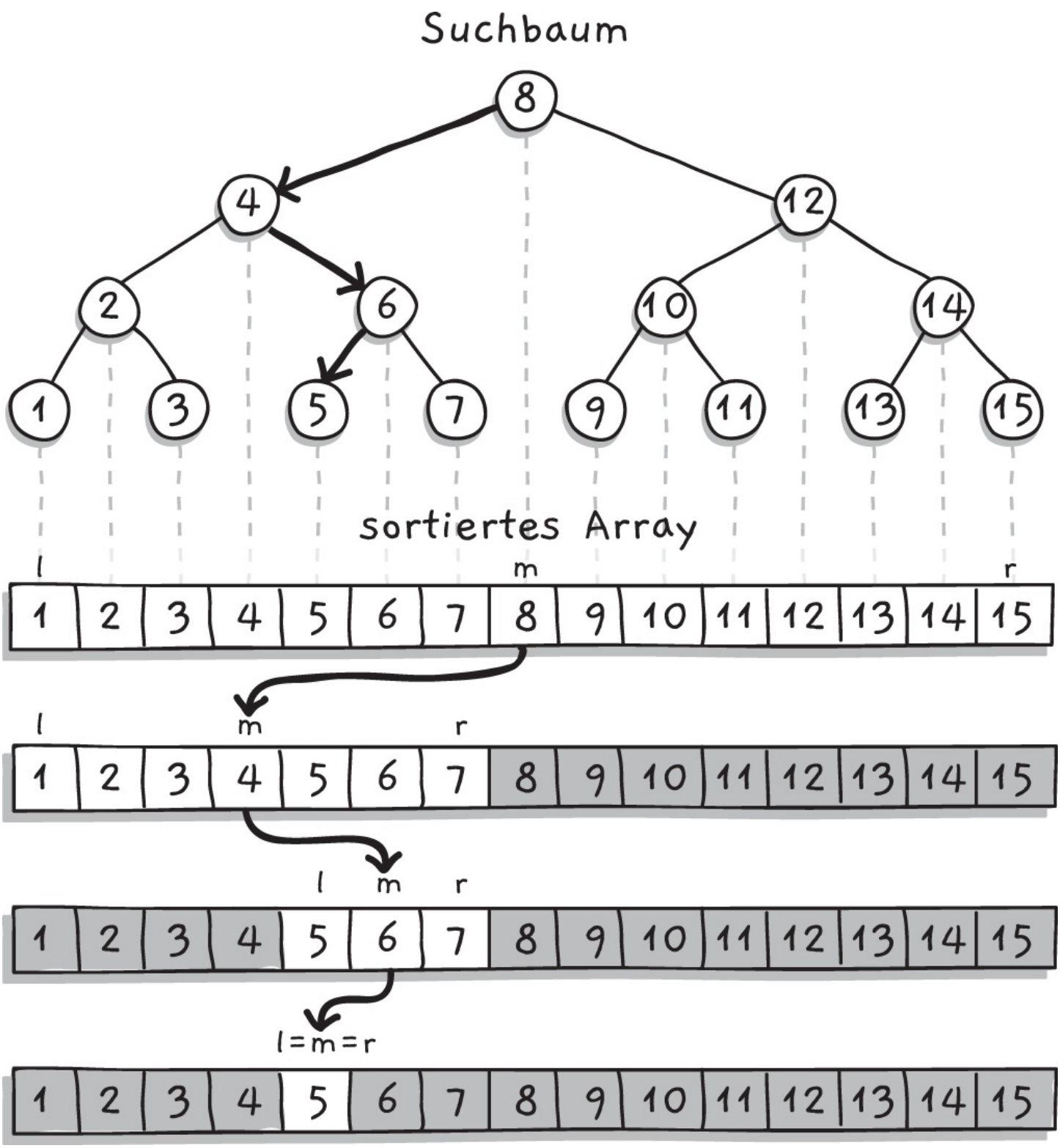


Abbildung 12.3: Suche nach dem Wert 5 in einem Suchbaum und in einem sortierten Array

- ✔ Entweder der Wert in der Mitte ist zufällig gerade der, den man gesucht hat. Dann hat man ihn gefunden und ist fertig.
- ✔ Oder man sucht einen *kleineren* Wert, dann muss dieser sich im *linken* Teilarray befinden. Der Suchraum, also der Teil des Arrays, den wir noch durchsuchen müssen, schränkt sich also auf etwa die Hälfte ein.

- ✔ Oder man sucht einen *größeren* Wert, dann muss dieser sich im *rechten* Teilarray befinden. Auch hier halbiert sich der Suchraum.

Diese Suchmethode wird *binäre Suche* oder kurz *Binärsuche* genannt, und zwar weil sich in jedem Schritt der Suchraum halbiert:

```
BinarySearch(a, x)          a: sortiertes Array mit n Elementen, x: gesuchter Wert
    l ← 1                   l: linker Rand des Suchraumes
    r ← n                   r: rechter Rand des Suchraumes
    while l ≤ r             solange der Suchraum noch nicht leer ist
        m ← (l+r)/2, abgerundet   m: Mitte zwischen l und r
        if x = a[m]         Wenn x gefunden wurde: ...
            return m        ... fertig!
        if x < a[m]         Wenn x in der linken Hälfte des Suchraums liegt: ...
            r ← m − 1       ... Suchraum auf linke Hälfte eingrenzen
        else                Ansonsten liegt x in der rechten Hälfte des Suchraums
            l ← m + 1       ... Suchraum auf rechte Hälfte eingrenzen
    return "x ist nicht in a enthalten"
```

Der Algorithmus liefert die Indexposition des ersten gefundenen Treffers zurück – oder eine Fehlermeldung, wenn sich der gesuchte Wert x nicht im Array befindet. Die Laufzeit von BinarySearch folgt aus dem Mastertheorem: Die Arbeit wird nicht aufgeteilt, sondern es wird nur einer der beiden Untergebenen befragt ($a = 1$). Jeder Mitarbeiter hat aber nur maximal die Hälfte des Suchbereichs seines Chefs ($b = 2$). Der Zusatzsaufwand hält sich in Grenzen ($f(n)$ ist $\mathcal{O}(1)$). Insgesamt wird auf jeder Ebene der Hierarchie gleich viel gearbeitet (Fall 2 des Mastertheorems). Daraus schließen wir: Die Binärsuche läuft in $\mathcal{O}(\log(n) \cdot f(n)) = \mathcal{O}(\log n)$. Ja genau, das hatten wir erwartet, denn das ist ja auch die Laufzeit für die Suche im balancierten Suchbaum.

Grenzen des Suchbereichs

Wenn im Array a der gesuchte Wert x mehrfach vorkommt, kann man mit BinarySearch *eines* seiner Vorkommen entdecken. Wenn wir *alle* Vorkommen von x in a kennen wollen, müssten wir von dieser Trefferposition aus nach links und recht im Array weiterlaufen, bis wir den linken und rechten Rand des Bereichs in a gefunden haben, in dem x gespeichert ist (siehe Abbildung 12.4). Im Extremfall, dass im Array ausschließlich der Wert x steht, er also n -mal gespeichert ist, bekommt man dadurch im *worst case* wieder eine lineare Laufzeit von $\mathcal{O}(n)$ für die Suche heraus.

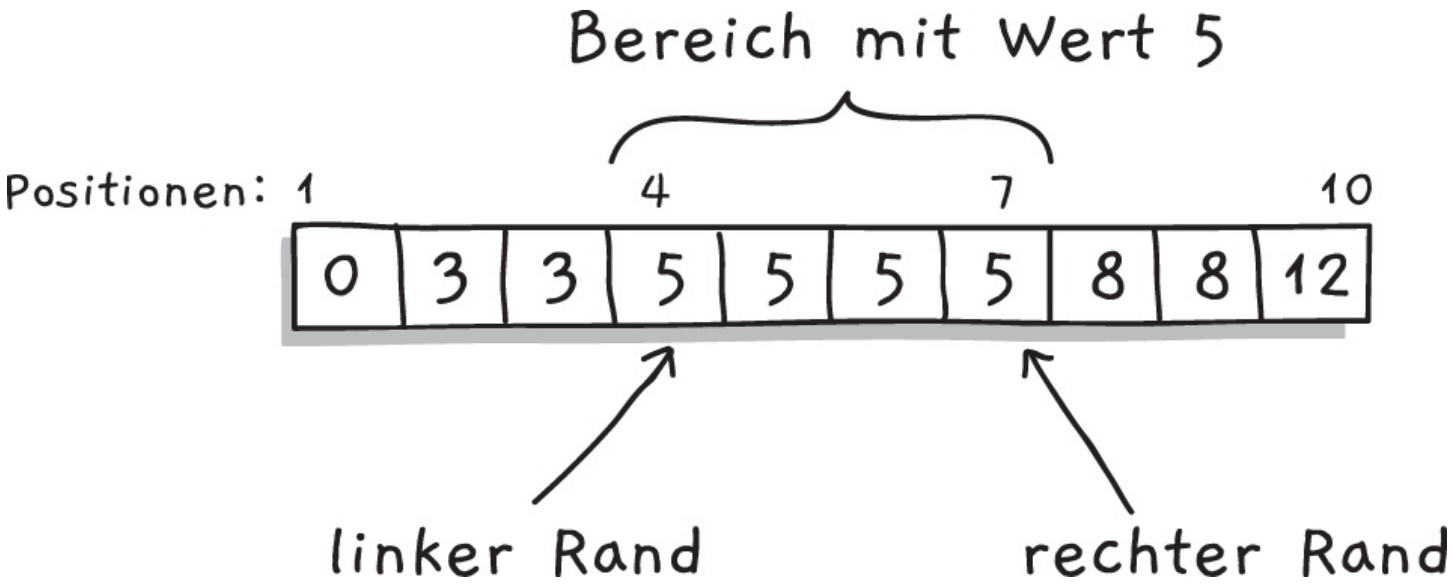


Abbildung 12.4: Linker und rechter Rand aller Vorkommen des Wertes 5 im Array

Mit ein paar kleinen Änderungen an der Binärsuche können wir aber dafür sorgen, dass die Suchzeit nicht mehr von der Anzahl der Treffer abhängt. Statt nur irgendein Vorkommen zu suchen, sorgen wir dafür, dass die Binärsuche den linken Rand des Bereichs findet, in dem die Vorkommen von x gespeichert sind. Das Gleiche können wir auch für den rechten Rand machen. Mit zwei binären Suchen, einmal für den linken Rand und einmal für den rechten Rand, also insgesamt in einer Laufzeit von $2 \cdot \mathcal{O}(\log n) = \mathcal{O}(\log n)$, können wir also den gesamten Bereich aller Vorkommen von x in a identifizieren, und zwar selbst wenn die Anzahl der Vorkommen sehr groß ist. Sollte der gesuchte Wert x gar nicht in a vorkommen, erkennen wir das daran, dass der »linke« Rand ein Feld weiter rechts liegt als der »rechte« Rand.

Für die Suche nach dem Rand müssen wir zunächst den Fall aus BinarySearch werfen, der die Suche abbricht, sobald x gefunden wurde. Wenn wir x finden, dann suchen wir nun stattdessen einfach weiter, bis der Suchraum komplett zusammengeschrumpft ist und die Größe 1 hat. Wenn $x = a[m]$ ist, dann liegt der linke Rand links von m und der rechte Rand rechts von m. Bei der Suche nach dem linken Rand müssen wir diesen Fall also genauso wie den Fall $x < a[m]$ behandeln und den rechten Rand verschieben. Suchen wir hingegen nach dem rechten Rand, ist der Fall $x = a[m]$ so zu behandeln wie $x > a[m]$, das heißt, wir suchen weiter rechts im Array und verschieben also den linken Rand. Am Ende steht der linke Rand des Trefferraums in der Variablen l; der rechte Rand in der Variablen r.

```
BinarySearchLeft(a, x)          a: sortiertes Array mit n Elementen, x: gesuchter Wert
    l ← 1
    r ← n
    while l ≤ r
        m ← (l+r)/2, abgerundet     m: Mitte zwischen l und r
        if x ≤ a[m]                 hier wird nun auch der Fall x = a[m] behandelt
            r ← m − 1               weiter links suchen
        else
            l ← m + 1               weiter rechts suchen
    return l                        am Ende steht der linke Rand in l
BinarySearchRight(a, x)         a: sortiertes Array mit n Elementen, x: gesuchter Wert
    l ← 1
    r ← n
    while l ≤ r
        m ← (l+r)/2, abgerundet     m: Mitte zwischen l und r
        if x < a[m]
            r ← m − 1               weiter links suchen
        else                        der Fall x = a[m] wird hier behandelt
            l ← m + 1               weiter rechts suchen
    return r                        am Ende steht der rechts Rand in r
```

Dass diese Algorithmen korrekt sind, kann man sich recht leicht klarmachen. Lassen Sie uns einen kurzen Beweis für die Korrektheit von BinarySearchLeft skizzieren, die Korrektheit des anderen Algorithmus lässt sich ganz ähnlich zeigen. Zu jedem Zeitpunkt gelten im Algorithmus folgende *Schleifeninvarianten*:

Für alle $1 \leq i < l$ ist $a[i] < x$ *(alle Felder links von l sind $< x$)*

Für alle $r < i \leq n$ ist $a[i] \geq x$ *(alle Felder rechts von r sind $\geq x$)*

Ganz am Anfang, bevor die **while**-Schleife das erste Mal durchlaufen wird, sind diese Aussagen wahr. Mit Blick auf Abbildung 12.5 lässt sich auch leicht sehen, dass wenn die Schleifeninvarianten am Anfang der Schleife gelten, sie am Ende der Schleife wieder gelten. Denn entweder liegt der erste Fall vor ($x \leq a[m]$) oder der zweite Fall ($x > a[m]$). In beiden Fällen werden die Grenzen so verschoben, dass anschließend die Schleifeninvarianten ihre Gültigkeit behalten. Der Algorithmus bricht ab, sobald der Suchraum leer ist, also $l > r$ wird. In diesem Fall ergibt sich genau das Bild in Abbildung 12.5 und der gesuchte linke Rand steht in l.

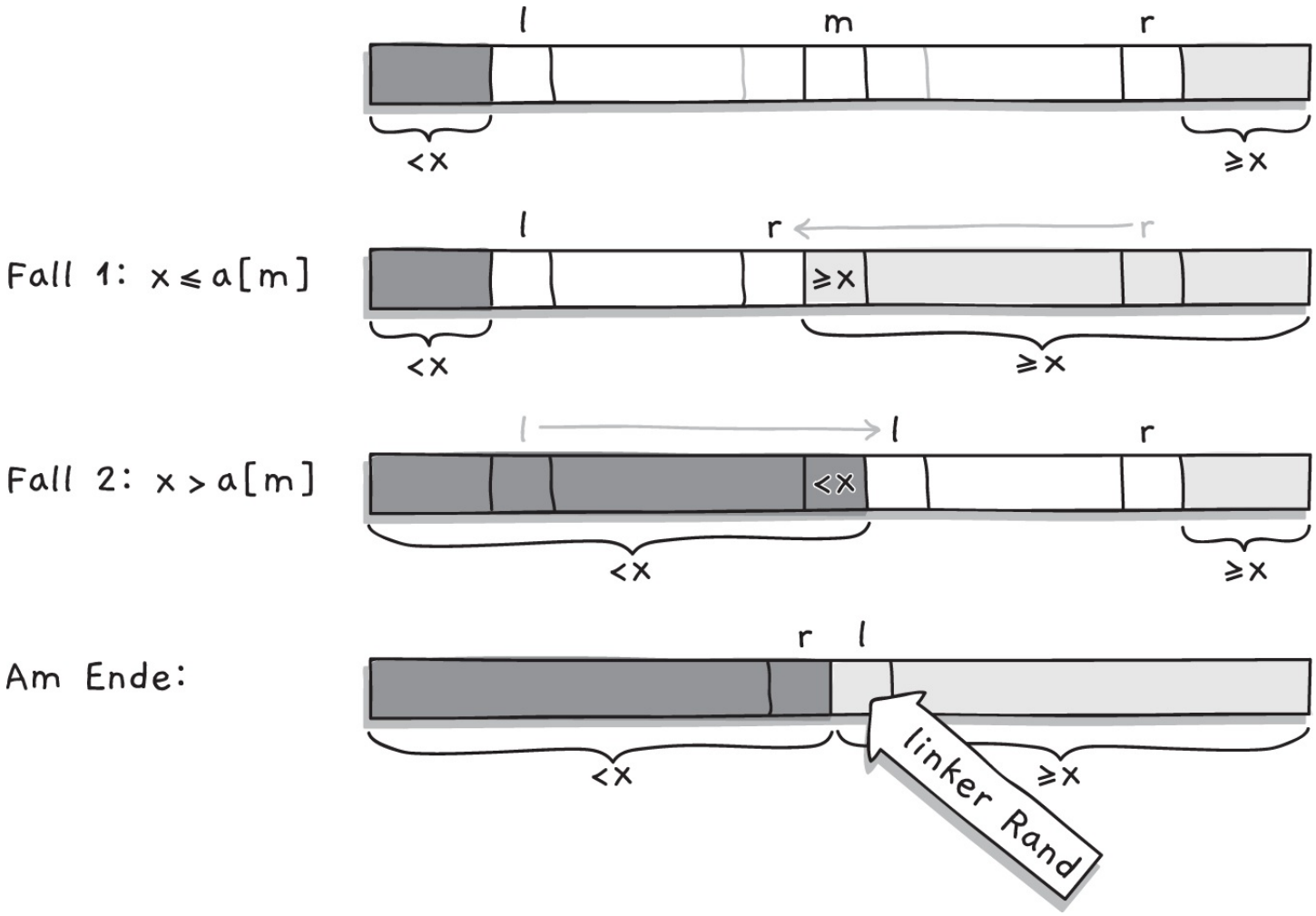


Abbildung 12.5: Die Suche nach dem linken Rand des Trefferbereichs


Mehr zum Thema Korrektheitsbeweise und Schleifeninvarianten finden Sie in Kapitel 1 im Abschnitt *Korrektheit beweisen*.

Weitere Beispiele für Teilen und Herrschen

Als einfaches Beispiel für einen Teilen-und-Herrschen-Algorithmus haben Sie bereits weiter oben in diesem Kapitel countTH kennengelernt. Da hatte jeder Beamte nur $f(n) = \mathcal{O}(1)$ viel zu tun. Also arbeiteten alle Beamten gleich viel, und da jeder von ihnen zwei Untergebene hat, arbeiten diese zusammen mehr als der Chef allein. Wir haben es also mit Fall 3 des »Bachelortheorems« zu tun und die Laufzeit von countTH ist demnach – wie Sie oben bereits gesehen haben – $\mathcal{O}(n)$.

Lassen Sie uns nun ein paar Algorithmen ansehen, die das Prinzip »Teilen und Herrschen« anwenden und deren Laufzeiten mit dem Mastertheorem (oder wenn möglich mit dem »Bachelortheorem«) ausrechnen.

Sortieren

Das Sortieren von Werten in einem Array mit Teilen und Herrschen haben wir bereits in Kapitel 7 ausführlich behandelt. Der Algorithmus MergeSort (Abschnitt *MergeSort: Geteiltes Leid ist halb sortiert*) bekommt als Eingabe das Array a sowie eine linke Grenze l und eine rechte Grenze r des Bereichs in a, der sortiert werden soll. Wenn in diesem Bereich nur ein einzelner Wert steht (also wenn $l = r$), ist nichts zu tun und die Rekursion bricht ab – die »Angestellten« haben hier also einen ausgesprochen leichten Job. Die »Beamten« andererseits haben schon etwas mehr Arbeit: Wenn mehr als ein Wert zwischen l und r im Array steht, dann teilt der Algorithmus diesen Bereich in zwei möglichst gleich große Teilbereiche, die dann jeweils mit rekursiven Aufrufen von MergeSort sortiert werden. Die eigentliche Arbeit kommt anschließend: Die zwei von den jeweiligen »Untergebenen« bereits vorsortierten Arrayhälften müssen zu einem Bereich »verschmolzen« werden. Der »Beamte« verwendet dazu den Algorithmus Merge (Abschnitt *Sortierte Teilarrays verschmelzen mit Merge*): Er geht gleichzeitig beide sortierten Listen durch, nimmt immer den kleineren der beiden nächsten Werte heraus und schreibt ihn in eine neue Liste. Am Ende müssen dann alle Werte wieder zurück in den ursprünglichen Arraybereich kopiert werden. Wenn insgesamt n Werte zwischen l und r im Array stehen, dauert Merge also $\mathcal{O}(n)$; das ist ein linearer Zusatzaufwand. Nach dem »Bachelortheorem«, Fall 2, ist somit die Laufzeit von MergeSort $\mathcal{O}(n \cdot \log n)$.

Matrizen multiplizieren

Eine Matrix A ist eine Tabelle von Zahlen, das heißt, es gibt mehrere Zeilen und mehrere Spalten und in der i-ten Zeile und der j-ten Spalte steht eine Zahl $A[i,j]$. Zwei Matrizen A und B mit gleich vielen Zeilen und gleich vielen Spalten kann man leicht addieren, indem man zu jeder Zahl A die Zahl an der gleichen Stelle in B addiert. Wenn also $C = A + B$ sein soll, dann muss für alle Zeilen i und alle Spalten j gelten, dass $C[i,j] = A[i,j] + B[i,j]$. Lassen Sie uns einmal annehmen, dass die Matrizen quadratisch sind, das heißt, dass es gleich viele Spalten wie Zeilen gibt, nämlich jeweils d. Solche Matrizen nennt man auch *($d \times d$)-Matrizen*. Für die Addition von zwei Matrizen A und B muss man jede der $n = d^2$ Zahlen für C ausrechnen und das benötigt eine Zeit von $\mathcal{O}(d^2) = \mathcal{O}(n)$.

Nun kann man Matrizen nicht nur addieren, sondern auch multiplizieren. Wenn man zwei Matrizen A und B mit jeweils d Zeilen und d Spalten miteinander multipliziert, so kommt wieder eine Matrix C mit d Zeilen und Spalten heraus. Für jede Zahl in C muss man eine Summe aus d Zahlen berechnen. Beispielsweise berechnen man $C[i,j]$, indem man die Zahlen in der i-ten Zeile von A nacheinander mit den Zahlen in der j-ten Spalte von B multipliziert und alle diese Produkte aufaddiert. In Formeln ausgedrückt:

$$C[i,j] = \sum_{z=1}^{d} A[i,z] \cdot B[z,j]$$

Die Berechnung jedes einzelnen Wertes in C benötigt also eine Zeit von $\mathcal{O}(d)$, und da man ja d^2 dieser Werte berechnen muss, dauert eine Matrixmultiplikation somit $\mathcal{O}(d^3) = \mathcal{O}\left(n^{\frac{3}{2}}\right) = \mathcal{O}(n^{1,5})$.

Lassen Sie uns mal sehen, ob wir das mit Teilen und Herrschen nicht schneller hinbekommen. Damit wir es leichter mit dem Teilen haben, wollen wir annehmen, dass die Zahl d der Spalten/Zeilen eine Zweierpotenz ist. Unser Algorithmus teilt die beiden Matrizen A und B jeweils in $b = 4$ gleich große Teilmatrizen auf, die $\frac{d}{2}$ Zeilen und Spalten haben. Jeden der vier Teile unserer Ergebnismatrix C können wir dann, wie in Abbildung 12.6 gezeigt, aus Produkten der Teilmatrizen von A und B berechnen. Insgesamt brauchen wir dafür acht Multiplikationen und vier Additionen von Matrizen mit $\frac{d}{2}$ Zeilen und Spalten. Für die Multiplikationen verwenden wir nach dem Prinzip »Teilen und Herrschen« $a = 8$ rekursive Aufrufe des Algorithmus; die Additionen bedeuten einen linearen Zusatzaufwand von $f(n) = \mathcal{O}(n)$. Jeder Untergebene hat nur ein Viertel der Arbeit seines Chefs zu tun, allerdings hat jeder Chef acht Untergebene, die zusammen doppelt so viel wie ihr Chef arbeiten müssen. Wir haben es somit mit dem Fall 3 des Mastertheorems zu tun, die Laufzeit ist also $\mathcal{O}(n^{\log_b a}) = \mathcal{O}(n^{\log_4 8}) = \mathcal{O}(n^{1,5})$.

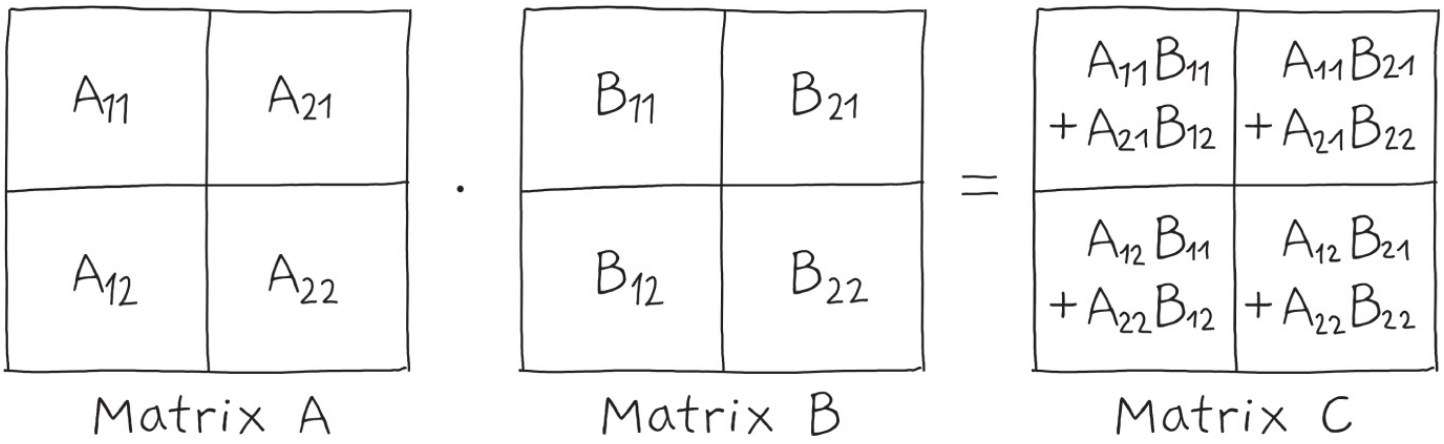


Abbildung 12.6: Matrixmultiplikation mit Teilen und Herrschen

Wenn man Teilen und Herrschen auf diese Weise anwendet, bringt das also leider gar keinen Vorteil gegenüber der herkömmlichen Methode. Der Mathematiker Arnold Strassen hatte jedoch Ende der 1960er Jahre eine Idee, wie man die vier Teile von C mit *sieben* statt acht Matrixmultiplikationen berechnen kann. Zwar braucht Strassens Algorithmus dafür mehr Matrixadditionen, nämlich 18 statt 4, der Zusatzaufwand bleibt aber trotzdem linear in n. Auch mit $b = 7$ Untergebenen liegt immer noch Fall 3 des Mastertheorems vor und durch den eingesparten rekursiven Aufruf sinkt die Laufzeit des

Algorithmus auf $\mathcal{O}(n^{\log_b a}) = \mathcal{O}(n^{\log_4 7}) = \mathcal{O}(n^{1{,}41})$.

Minimaler Punktabstand

Bislang sind in diesem Buch nur die Fälle 2 und 3 des Mastertheorems zur Anwendung kommen. Gute Beispiele für den Fall 1 sind nämlich leider rar, denn wenn der Chef ohnehin mehr tun muss als seine Untergebenen zusammen, so kann er oft genauso gut ganz auf die Untergebenen verzichten und alles selbst machen. Lassen Sie es uns bei folgendem Problem dennoch versuchen.

Gegeben sei eine Reihe von n Punkten, die in einem Array a gespeichert sind. Wir suchen nun den minimalen Abstand zwischen zwei unterschiedlichen Punkte in a. Mit dem Prinzip »rohe Gewalt« ließe sich das Problem zum Beispiel so lösen:

```
minDistBF(a)                          a: Array von n Punkten
    mindist ← ∞                       initialisiere mindist
    for i ← 1, ..., n − 1             Zähle alle Paare von Punkten auf
        for j ← i + 1, ..., n
            dist ← Abstand zwischen a[i] und a[j]
            mindist ← min(mindist, dist)    den minimalen Abstand merken
    return mindist
```

Der Algorithmus zählt einfach alle Paare $a[i]$ und $a[j]$ von Punkten, berechnet den Abstand zwischen ihnen und merkt sich das Minimum. Hier gibt es keine Arbeitsteilung, einer macht alles allein und er braucht dafür die Zeit $\mathcal{O}(n^2)$.

Nun wollen wir das Prinzip »Teilen und Herrschen« benutzen (Abbildung 12.7):

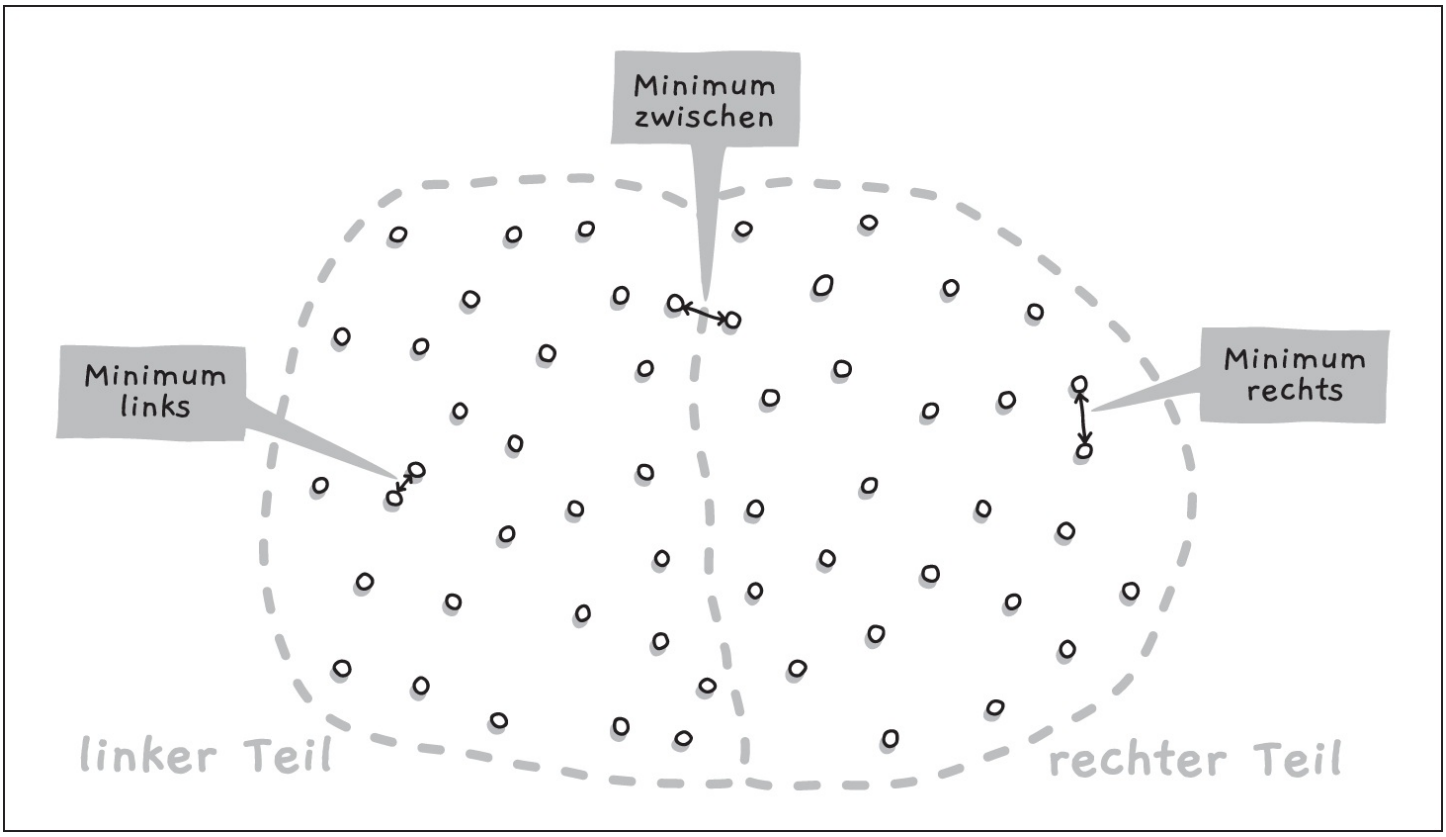


Abbildung 12.7: Minimaler Punktabstand mit Teilen und Herrschen

```
minDistTH(a, l, r)                    a: Array von n Punkten; l, r: Positionen in a, l ≤ r
    if l = r then                     Abbruchbedingung
        return ∞                      bei nur einem Punkt gibt es keinen Abstand
    else
        m ← (l+r)/2, aufgerundet
        mindist ← min(minDistTH(a, l, m − 1), minDistTH(a, m, r))
        for i ← 1, ..., m − 1         zähle alle Punkte der linken Hälfte auf und ...
            for j ← m, ..., n         ... kombiniere sie mit den Punkten der rechten Hälfte
                dist ← Abstand zwischen a[i] und a[j]
                mindist ← min(mindist, dist)    den minimalen Abstand merken
        return mindist
```

Nun kann der »Chef« einen Teil der Arbeit auf seine beiden Untergebenen abwälzen. Der eine berechnet den minimalen Abstand zwischen zwei Punkten in der einen Hälfte des Arrays, der andere den minimalen Abstand zwischen zwei Punkten in der anderen Hälfte. Nun muss der Chef »nur noch« den Abstand von jedem Punkt der einen Hälfte zu jedem Punkt aus der anderen Hälfte berechnen. Wenn es n Punkte in a gibt, ist jede der Hälften $\frac{n}{2}$ groß, der Chef muss also $\frac{n}{2} \cdot \frac{n}{2} = \frac{n^2}{4} = \mathcal{O}(n^2)$ viele Abstände berechnen. Da der Zusatzaufwand hier quadratisch ist, liegt Fall 1 des Bachelortheorems vor, das heißt, die Laufzeit des Algorithmus ist insgesamt ebenfalls $\mathcal{O}(n^2)$.

Nebenbei wollen wir anmerken, dass das nicht die schnellste Methode ist, um das Problem zu lösen. Auf jeden Fall kann man auf diese Weise mit Teilen und Herrschen keinen Vorteil gegenüber der einfachen *Brute-Force*-Methode erzielen. Es ist aber trotzdem schön, dass wir darüber geredet haben.

Kapitel 13

Dynamisches Programmieren

IN DIESEM KAPITEL

Dynamisches Programmieren: »Vom Kleinen zum Großen«.

Als Eisenbahnunternehmer reich werden.

Das Stringalignment-Problem.

Ein profitabler Bauauftrag

Endlich zahlt sich Ihre jahrelange Lobbyarbeit aus: Um die jungen, aufstrebenden Großstädte der Westküste mit den Metropolen an der Ostküste zu verbinden, hat der Kongress beschlossen, quer durch das Land eine Eisenbahnlinie bauen zu lassen. Die Streckenführung wurde bereits festgelegt und wie man in Abbildung 13.1 sehen kann, haben einige sehr einflussreiche Politiker dafür gesorgt, dass ihre jeweiligen Wahlbezirke ebenfalls von dem Projekt profitieren werden. Sie als zukünftiger Eisenbahnmagnat haben zum Glück gute Kontakte in die Politik und konnten sich den Bauauftrag bereits im Vorfeld sichern. Noch besser: Sie können selbst auswählen, welchen Teil des Bauauftrags Sie übernehmen. Die Planer haben die Eisenbahnlinie in $n = 14$ gleich lange Bauabschnitt aufgeteilt und Sie müssen nur noch entscheiden, welchen Teil der Gesamtstrecke Sie bauen wollen. Allerdings muss es ein einzelner, zusammenhängender Teil der Gesamtstrecke sein; Sie dürfen sich also nicht einfach nur die Rosinen herauspicken. Sie entscheiden, von wo bis wo Sie den Auftrag übernehmen werden, und den Rest der Strecke baut dann eben jemand anders. Mindestens einen Bauabschnitt sollten Sie aber auf jeden Fall bauen, das ist Ehrensache. Für jeden fertig gestellten Bauabschnitt bekommen Sie vom Staat 30 Millionen Dollar (30 M$) ausgezahlt. Leider sind die dafür notwendigen Arbeiten je nach Bauabschnitt mit sehr unterschiedlichen Kosten verbunden. Geht eine Strecke zum Beispiel durch ein Gebirge oder Sumpfgebiet oder führt sie über einen großen Fluss, so können die Erschließungs- und Baukosten die Einnahmen bei Weitem übertreffen. Dennoch kann es sich durchaus lohnen, einen solchen Bauabschnitt zu übernehmen, etwa wenn sowohl davor als auch danach sehr lukrative Bauabschnitte liegen. Ihre Ingenieure rechnen mit folgenden Kosten pro Bauabschnitt: 29 M$, 34 M$, 27 M$, 12 M$, 29 M$, 38 M$, 28 M$, 31 M$, 20 M$, 35 M$, 110 M$, 10 M$, 27 M$ und 32 M$. Welchen Streckenteil möchten Sie bauen?

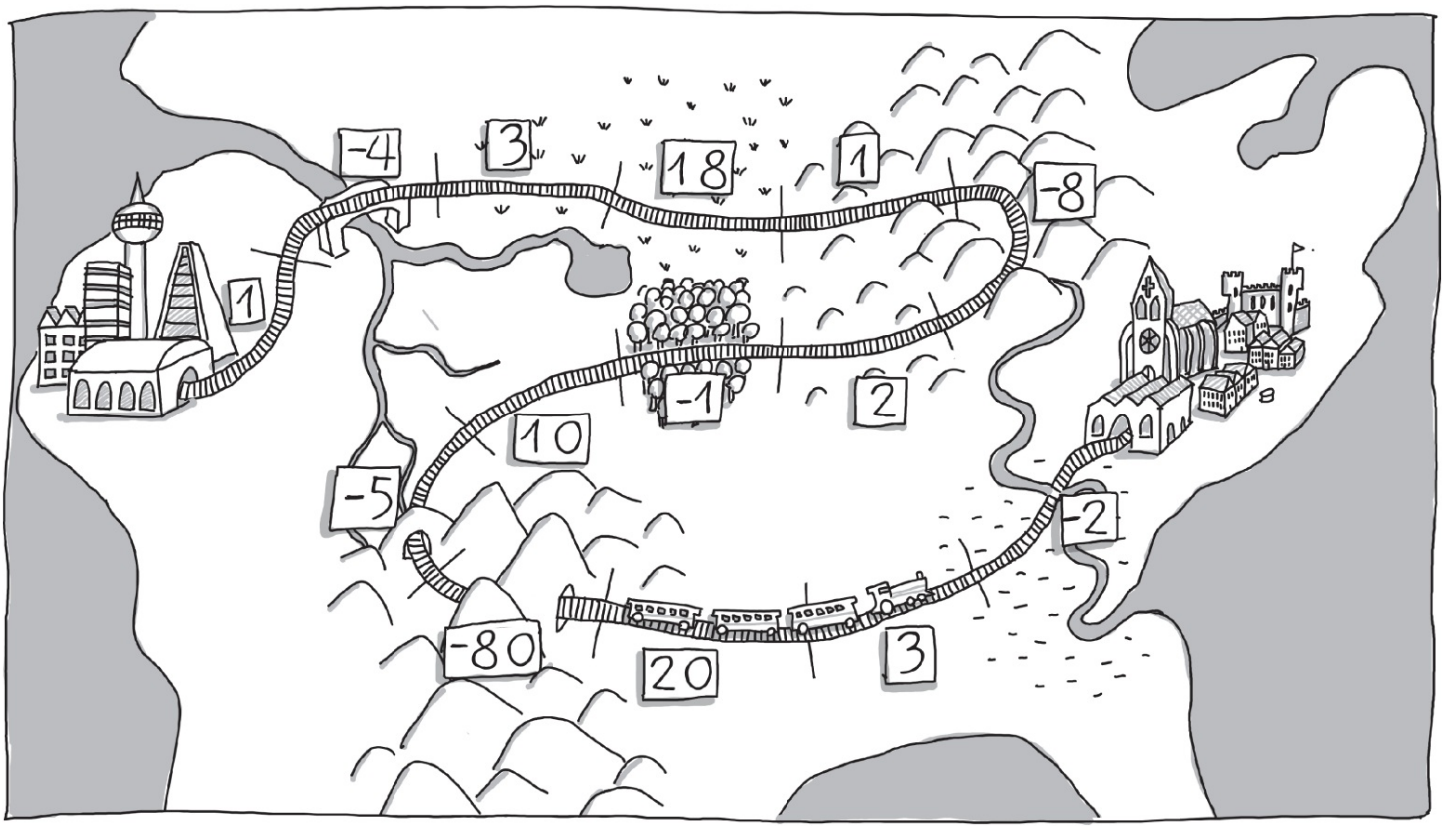


Abbildung 13.1: Verlauf der Bahnstrecke und der Profit pro Bauabschnitt in M$

Das Maximale-Teilsumme-Problem

Wenn wir aus dem Erlös und den Kosten den jeweiligen Gewinn in M$ pro Bauabschnitt ausrechnen und in einem Array a speichern, so können wir das Problem wie folgt formulieren: Gegeben ist ein Array a mit n Zahlen $a[1], a[2], \ldots, a[n]$. In unserem Beispiel wären etwa

$$a = [1, -4, 3, 18, 1, -8, 2, -1, 10, -5, -80, 20, 3, -2]$$

Die negativen Zahlen in a stehen für Streckenabschnitte, deren Bau ein Verlustgeschäft wäre. Gesucht sind zwei Positionen l und r im Array mit $1 \leq l \leq r \leq n$, bei denen die Teilsumme der Zahlen in a zwischen der l-ten und der r-ten Position, also

$$a[l] + a[l+1] + \ldots + a[r] = \sum_{i=l}^{r} a[i]$$

möglichst groß ist. Für die maximale Teilsumme im Array a wollen wir auch kurz MaxSum schreiben. Um uns die Aufgabe ein wenig leichter zu machen, interessieren wir uns zunächst nur für diese maximale Teilsumme selbst. Die beiden Positionen l und r hingegen sollen uns erst einmal egal sein; die können wir später immer noch ausrechnen (siehe Abschnitt *Welche Bahnstrecke bauen Sie?*).

Gier hilft nicht

Nun könnte Ihnen spontan der Bauabschnitt 12 ins Auge fallen, bei dem ein Gewinn von

20 M$ winkt, der höchste Gewinn, den man mit dem Bau eines einzelnen Abschnitts erreichen kann. Wenn wir den angrenzenden Bauabschnitt 13 noch dazu nehmen, können wir hier einen Gewinn von 23 M$ erzielen. Die ausgewählte Strecke noch weiter auszuweiten, lohnt sich für uns dann allerdings nicht. Bauabschnitt 14 ist ein Verlustgeschäft und der Bau von Abschnitt 11 wäre so furchtbar teuer, dass wir diesen Bauabschnitt auf gar keinen Fall übernehmen möchten, obwohl jenseits der hohen Berge noch ein paar durchaus verlockende Bauabschnitte liegen.

Optimal ist diese Wahl allerdings nicht; besser wäre es, wenn Sie sich für die Abschnitte 3 bis 9 entscheiden würden, denn damit läge Ihr Gewinn bei $3+18+1-8+2-1+10=25$ M$. Tatsächlich ist das in unserem Beispiel die maximale Teilsumme. Nach dem Prinzip *Gier* vorzugehen, also einfach den profitabelsten Bauabschnitt zu wählen und so lange es sich lohnt, nach beiden Seiten auszuweiten, führt also nicht unbedingt zum optimalen Ergebnis.

Rohe Gewalt hilft eher

Bevor man nach eleganten Lösungen sucht, ist es oft lehrreich, einem Problem erst einmal mit roher Gewalt (also »*brute force*«) zu begegnen, um zu sehen, wie weit man damit kommt.



Mehr zur Methode der rohen Gewalt und erschöpfenden Suche finden Sie im Kapitel 11, in dem beschrieben wird, wie man gepflegt Probleme »totschlägt«.

Wir probieren also einfach alle möglichen Teilsummen aus und merken uns, welche davon die größte ist. Dazu brauchen wir drei ineinander verschachtelte Schleifen: Eine äußere Schleife für den linken Rand l des Teilabschnitts im Array, eine mittlere Schleife für den rechten Rand r und außerdem eine weitere, innere Schleife i, mit der wir alle Positionen zwischen l und r durchgehen und die Zahlen in a zusammenzählen.

```
maxSubSumBF(a)                  a: Array von n Zahlen
    max ← −∞                    max: größte bislang gefundene Teilsumme, anfangs −∞
    for l ← 1, ..., n           l: Anfang des Teilarrays
        for r ← l, ..., n       r: Ende des Teilarrays
            sum ← 0             sum: wird die Summe der Zahlen im Teilarray, anfangs 0
            for i ← l, ..., r   laufe durch alle Zahlen im Teilarray ...
                sum ← sum + a[i]    ... und addiere sie
            if sum > max then   wenn eine größere Teilsumme gefunden wurde, ...
                max ← sum       ... merke sie dir
    return max
```

Durch die dreifach geschachtelte Schleife bekommt maxSubSumBF eine Laufzeit von $\mathcal{O}(n^3)$. Sie werden sicherlich nicht überrascht sein, wenn wir an dieser Stelle schon einmal vorwegnehmen, dass das mit der richtigen Methode noch deutlich besser geht.

Inkrementelle Gewalt ist weniger roh

Betrachtet man maxSubSumBF genauer, so bemerkt man sehr schnell, dass der Algorithmus oft Zahlen zusammenzählt, die er vorher schon einmal addiert hat. Angenommen, der Algorithmus ist gerade mit dem Durchlauf Nummer $r-1$ der mittleren Schleife fertig. Im anschließenden r -ten Durchlauf wird nun die innere Schleife wiederum alle Zahlen von $a[l]$ bis $a[r-1]$ zusammenzählen, bevor am Ende – als einziger Unterschied zum vorangegangenen Durchlauf der mittleren Schleife – der Wert $a[r]$ zur Summe hinzugezählt wird. Das können wir besser machen. Wann immer wir r um eins erhöhen, nehmen wir den bisherigen Wert **sum** und addieren einfach den zusätzlichen Wert $a[r]$ hinzu. In der Informatik nennt man ein solches Vorgehen *inkrementell*: Aus einem alten Ergebnis wird durch eine kleine Änderung das neue Ergebnis gemacht. Das spart Zeit. Mit diesem Trick werden wir eine der drei Schleifen los und können somit die Laufzeit von $\mathcal{O}(n^3)$ auf $\mathcal{O}(n^2)$ drücken:

```
maxSubSumInc(a)                 a: Array mit n Zahlen
    max ← −∞                    max: größte bislang gefundene Teilsumme, anfangs −∞
    for l ← 1, ..., n           l: Anfang des Teilarrays
        sum ← 0                 sum: wird die Summe der Zahlen im Teilarray, anfangs 0
        for r ← l, ..., n       r: Ende des Teilarrays
            sum ← sum + a[r]    berechne inkrementell die neue Teilsumme
            if sum > max then   wenn eine größere Teilsumme gefunden wurde, ...
                max ← sum       ... merke sie dir
    return max
```

Auch dieser Algorithmus probiert also alle möglichen Teilarrays durch. Letztlich ist das immer noch rohe Gewalt, nur etwas klüger als bei maxSubSumBF.

Die Idee, eine Lösung »inkrementell« aus einer vorherigen Lösung auszurechnen, wird uns in diesem Kapitel noch weiter beschäftigen. Im Array a stehen ja n Zahlen. Wenn es weniger Zahlen wären, so wäre die Aufgabe vermutlich leichter zu lösen. Vielleicht könnten wir ja die Lösung für die Aufgabe mit n Zahlen *inkrementell* aus einer Lösung für die Aufgabe aus den ersten $n-1$ Zahlen im Array ausrechnen. Das werden wir als Nächstes versuchen.

Ein Stück abschneiden und Herrschen

Unser nächste Ansatz erinnert etwas an das Prinzip »Teilen und Herrschen«, nur teilen wir das Array a nicht wie sonst üblich in der Mitte, sondern ganz nah am Rand. Wir lassen zunächst die letzte Zahl aus dem Array a weg, ermitteln die Lösung für das etwas kleinere Array mit nur $n-1$ Zahlen und berechnen daraus dann eine Lösung für das ganze Array mit n Zahlen.

Die maximale Teilsumme der ersten i Zahlen im Array a wollen wir MaxSum_i nennen. Es ist unser Plan, MaxSum_n mithilfe von MaxSum_{n-1} auszurechnen. Dabei

gibt es zwei Möglichkeiten:

1. Entweder ist in MaxSum_n das letzte Element $a[n]$ aus dem Array enthalten oder
2. $a[n]$ ist *nicht* in MaxSum_n enthalten.

Im ersten Fall ist der rechte Rand r des Teilarrays mit der Wertesumme MaxSum_n ganz am Ende des Array, also $r = n$. Die maximale Summe eines Teilarrays, das bis zum Ende läuft, nennen wir $\mathrm{MaxEndSum}_n$. In Fall 1 ist also $\mathrm{MaxSum}_n = \mathrm{MaxEndSum}_n$ (vergleiche Abbildung 13.2).

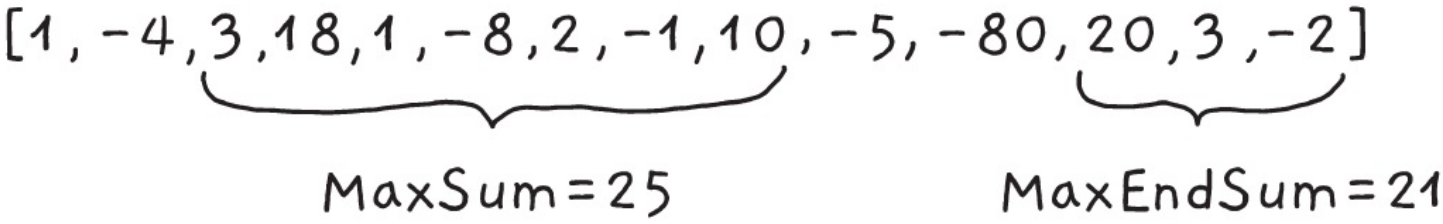


Abbildung 13.2: Maximale Teilsumme und maximale Endsumme

Der zweite Fall ist leicht, denn wenn sich die maximale Teilsumme durch das Hinzunehmen von $a[i]$ gar nicht geändert hat, dann bedeutet das schlicht und einfach $\mathrm{MaxSum}_n = \mathrm{MaxSum}_{n-1}$.

Welcher der zwei Fälle trifft denn nun zu? Das hängt offenbar davon ab, was größer ist, $\mathrm{MaxEndSum}_n$ oder MaxSum_{n-1}. Entweder macht man den größten Gewinn mit einem Teilarray, welches das letzte Element $a[n]$ enthält, oder eben mit einem Teilarray, das $a[n]$ *nicht* enthält. MaxSum_n ist also das Maximum von $\mathrm{MaxEndSum}_n$ und MaxSum_{n-1}.

Nun müssen wir nur noch einen Weg finden, die maximale Endsumme $\mathrm{MaxEndSum}_n$ auszurechnen. Wieder wollen wir die letzte Zahl vom Array abschneiden und versuchen, $\mathrm{MaxEndSum}_n$ *inkrementell* aus $\mathrm{MaxEndSum}_{n-1}$ zu berechnen. Das klappt auch tatsächlich sehr gut, für $\mathrm{MaxEndSum}_n$ gibt es nämlich ebenfalls nur zwei Möglichkeiten:

1. Entweder besteht $\mathrm{MaxEndSum}_n$ nur aus der letzten Zahl $a[n]$ oder
2. $\mathrm{MaxEndSum}_n$ ist die Summe aus mehreren Zahlen, entsteht also aus $\mathrm{MaxEndSum}_{n-1}$ plus dem angehängten $a[n]$.

Da die maximale Endsumme eben eine *End*summe ist, muss ja das letzte Element $a[n]$ immer mit dabei sein. Und wieder gewinnt der Fall, bei dem die größere Zahl herauskommt: $\mathrm{MaxEndSum}_n$ ist das Maximum von $a[n]$ und $\mathrm{MaxEndSum}_{n-1} + a[n]$.

Wenn wir das noch einmal alles zusammenfassen, bekommen wir die folgenden *Rekursionsgleichungen* heraus:

$$\begin{aligned}
\text{MaxSum}_n &= \max(\text{MaxEndSum}_n, \text{MaxSum}_{n-1}) \\
\text{MaxEndSum}_n &= \max(a[n], \text{MaxEndSum}_{n-1} + a[n]) \\
\text{MaxSum}_1 &= a[1] \\
\text{MaxEndSum}_1 &= a[1]
\end{aligned}$$

Die beiden letzten Gleichungen sind sozusagen die *Abbruchbedingungen*. Wenn es überhaupt nur einen einzelnen Bauabschnitt gibt ($n = 1$) und man mindestens einen Abschnitt bauen muss, dann fällt die Wahl ziemlich leicht.

Aus den Rekursionsgleichungen lassen sich nun ohne Umschweife zwei rekursiver Algorithmen stricken, die gemeinsam die Aufgabe lösen:

```
MaxSum(a, n)                                  a: Array mit ≥ n Zahlen
    if n = 1 then return a[1]                 Abbruchbedingung
    else return max(MaxEndSum(a, n), MaxSum(a, n − 1))    Rekursion

MaxEndSum(a, n)                               a: Array mit ≥ n Zahlen
    if n = 1 then return a[1]                 Abbruchbedingung
    else return max(a[n], MaxEndSum(a, n − 1) +a[n])      Rekursion
```

Das lässt sich zwar kurz und knackig hinschreiben, hat allerdings einen Nachteil. Man ist damit nämlich auch nicht schneller als mit maxSubSumInc.

Wird der Algorithmus MaxSum auf einem Array a der Länge n gestartet, so ruft er sich selbst rekursiv für alle Arraylängen $n-1$, $n-2$, $n-3$, $\cdots$, 2, 1 auf. Nach dem gleichen Muster führt ein Aufruf von MaxEndSum(a, n) zu $n-1$ rekursiven Aufrufen von MaxEndSum. MaxEndSum hat also eine Laufzeit von $\mathcal{O}(n)$, und da jeder Aufruf von MaxSum zu einem Aufruf von MaxEndSum führt, hat MaxSum eine Laufzeit von $\mathcal{O}(n^2)$.

In Abbildung 13.3 kann man erkennen, wo das Problem steckt: MaxEndSum wird mit derselben Eingabe mehrfach aufgerufen. Für $n-1$ nämlich zweimal, für $n-2$ dreimal, für $n-3$ viermal und so weiter. Wenn es gelänge, den Algorithmus für jede Arraylänge nur einmal aufrufen zu müssen, dann hätten wir es geschafft und könnten die maximale Teilsumme in Linearzeit ausrechnen.

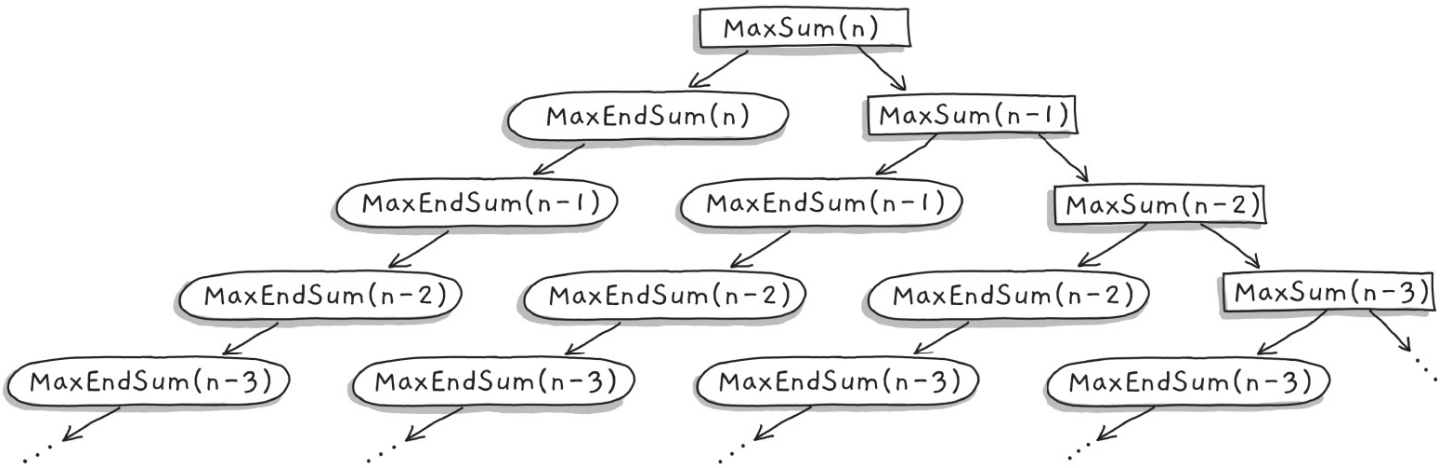


Abbildung 13.3: MaxEndSum wird $\mathcal{O}(n^2)$-mal aufgerufen.

Zwischenergebnisse merken

Für das Problem, dass Werte mehrfach berechnet werden, gibt es eine einfache Lösung. Sobald ein Wert einmal berechnet wurde, speichert man ihn in einer geeigneten Datenstruktur, und sobald er erneut benötigt wird, zaubert man ihn aus dieser Datenstruktur wieder herbei. In unserem Fall würde es reichen, sich für jede Arraylänge i den Werte von MaxEndSum(a, i) zu merken. Dafür bietet sich ein Array der Länge n an; wir wollen es Mem (wie *mem*ory) nennen.

Am Anfang ist Mem noch völlig leer, das heißt, die Werte in den Zellen $Mem[i]$ sind noch nicht definiert. Wann immer wir MaxEndSum mit einer Arraylänge i aufrufen, prüft der Algorithmus, ob $Mem[i]$ bereits berechnet worden ist. Wenn nicht, so muss der Wert mithilfe der Rekursionsgleichungen neu berechnet werden. Anschließend wird der einmal berechnete Wert in $Mem[i]$ gespeichert. Wenn bei einem späteren Aufruf festgestellt wird, dass $Mem[i]$ schon definiert ist, spart sich der Algorithmus die Neuberechnung und gibt direkt diesen Wert zurück.

```
Mem sei ein Array                          Mem: globale Variable zum Merken der Ergebnisse

MaxSum(a, n)                               a: Array mit ≥ n Zahlen
    if n = 1 then return a[1]              Abbruchbedingung
    else return max(MaxEndSum(a, n), MaxSum(a, n − 1))      Rekursion

MaxEndSum(a, n)                            a: Array mit ≥ n Zahlen
    if Mem[n] ist definiert then           wurde der Wert schon berechnet?
        return Mem[n]                      ja: spare die Neuberechnung
    else                                   nein: berechne den Wert
        if n = 1 then res ← a[1]           Abbruchbedingung
        else res ← max(a[n], MaxEndSum(a, n − 1) +a[n])      Rekursion
        Mem[n] ← res                       Ergebnis für später merken
        return res
```

Beim ersten Aufruf von MaxSum wird erstmals MaxEndSum für die volle Länge n des Arrays aufgerufen. MaxEndSum ruft sich selbst in der Folge rekursiv für alle Arraylängen $n-1$, $n-2$, $n-3$, $\cdots$, 2, 1 auf, berechnet jeweils die Werte neu und merkt sie sich in Mem. Sobald MaxEndSum(a, n) also zum ersten Mal zurückkommt, ist Mem komplett gefüllt. Wenn nun anschließend nach und nach MaxSum für Arraylängen $n-1$, $n-2$, $n-3$, $\cdots$ aufgerufen wird, kann MaxEndSum ohne erneuten rekursiven Abstieg den Wert für die maximale Endsumme zurückliefern. Insgesamt wird MaxSum also n-mal und MaxEndSum $(\leq 2 \cdot n)$-mal aufgerufen. Die Laufzeit ist somit $\mathcal{O}(n)$.

Das ist ein großer Erfolg! Mit dieser Technik – sie wird *Memoisation* genannt – ist es uns gelungen, einen Algorithmus zur Lösung des Maximale-Teilsumme-Problems mit optimaler asymptotischer Laufzeit zu finden (vergleiche Abbildung 13.4). Schneller geht es nicht, aber einfacher. Denn bisher ist der Algorithmus noch etwas umständlich. Gleich sehen Sie, wie es einfacher geht.

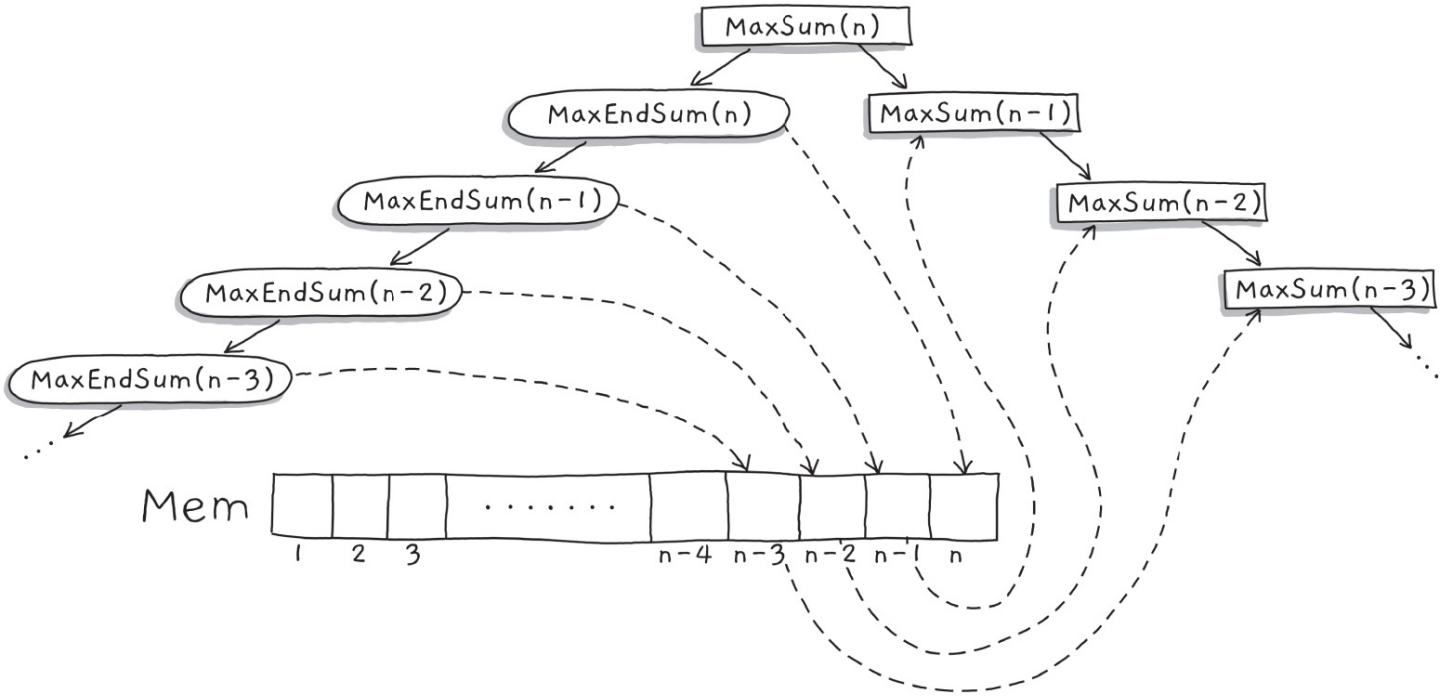


Abbildung 13.4: Memoisation verhindert doppelte Arbeit.

Den Algorithmus vom Kopf auf die Füße stellen

Bevor man MaxSum_n ausrechnen kann, muss man MaxSum_{n-1} ausgerechnet haben, und davor muss man MaxSum_{n-2} ausgerechnet haben und so weiter. Das gleiche gilt für die maximale Endsumme: Vor MaxEndSum_n braucht man MaxEndSum_{n-1} und davor MaxEndSum_{n-2} und so weiter. Bisher regeln wir alle diese Abhängigkeiten durch rekursive Aufrufe, dabei ist es doch recht naheliegend, das ganze iterativ hinzuschreiben. Bisher fangen wir mit unserem Ziel MaxSum_n an und hangeln uns dann in unzähligen Aufrufen bis zur Abbruchbedingung herunter, bevor wir

dann nach und nach alle wichtigen Werte ausrechnen. Lassen Sie uns das doch einfach umdrehen: Zuerst fangen wir mit dem einfachen Fall MaxSum_1 an, rechnen daraus MaxSum_2 aus, dann MaxSum_3 und so weiter, bis wir am Ende MaxSum_n ausrechnen können. Dabei legen wir alle maximalen Teil- und Endsummen in einem Array ab, damit wir einmal berechnete Ergebnisse später für die Berechnung der nächsten Werte zur Verfügung haben.

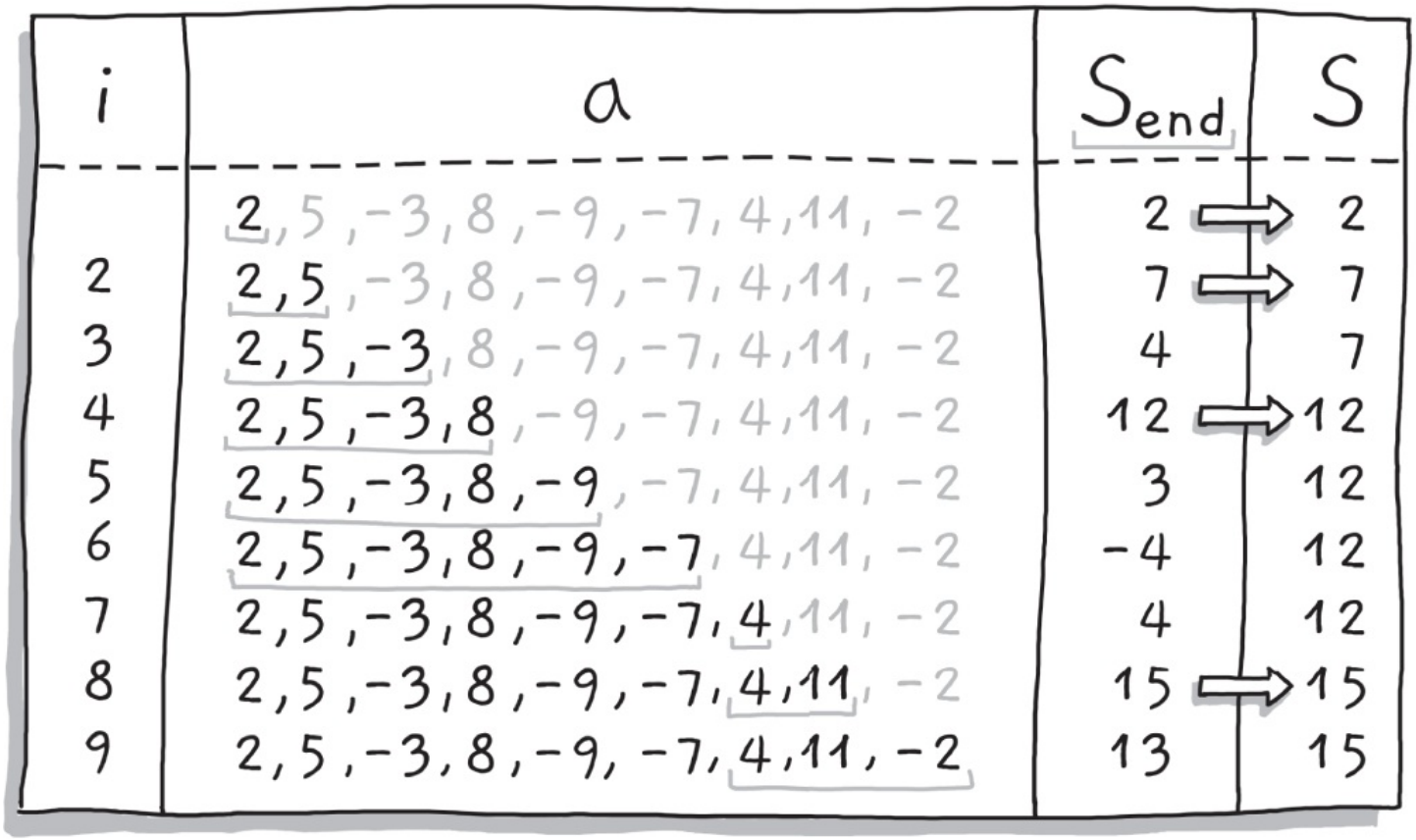
```
maxSumGrowing(a)                    a: Array mit n Zahlen
    MaxSum und MaxEndSum seien zwei Arrays
    MaxSum[1] ← a[1]                initialisiere MaxSum und MaxEndSum
    MaxEndSum[1] ← a[1]
    for i ← 2 ... n                 durchlaufe a und berechne die nächsten Werte
        MaxEndSum[i] ← max(a[n], MaxEndSum[i − 1] + a[n])
        MaxSum[i] ← max(MaxEndSum[i], MaxSum[i − 1])
    return MaxSum[n]
```

Der ultimative Maximale-Teilsumme-Algorithmus

Der Algorithmus maxSumGrowing verbraucht für die Speicherung von Zwischenwerten in den beiden Arrays **MaxSum** und **MaxEndSum** noch relativ viel Platz. Das ist eigentlich unnötig, denn für die Berechnung von **MaxSum** [n] brauchen wir zum Beispiel lediglich **MaxSum** [$n - 1$], alle anderen Einträge in **MaxSum** werden gar nicht mehr benötigt. Tatsächlich kommen wir mit nur zwei einzelne Variablen S und S_{end} aus, die wir in jeder Runde neu aus den alten Werten berechnen.

```
Kadane(a)                           a: Array mit n Zahlen
    S ← a[1]                        S: maximale Teilsumme
    S_end ← a[1]                    S_end: maximale Endsumme
    for i ← 2 ... n                 durchlaufe alle Zellen in a und ...
        S_end ← max(a[i], S_end + a[i])     ... berechne neue maximale Endsumme
        S ← max(S_end, S)           ... berechne neue maximale Teilsumme
    return S
```

Der Algorithmus ist nach seinem Erfinder Joseph Kadane benannt. Abbildung 13.5 zeigt an einem Beispiel, wie dieser Algorithmus arbeitet. Kadanes Algorithmus verwendet ein algorithmisches Prinzip namens *dynamisches Programmieren*. Um dieses Prinzip soll es im weiteren Verlauf dieses Kapitels gehen.



i	a	S_{end}	S
	2,5,-3,8,-9,-7,4,11,-2	2	2
2	2,5,-3,8,-9,-7,4,11,-2	7	7
3	2,5,-3,8,-9,-7,4,11,-2	4	7
4	2,5,-3,8,-9,-7,4,11,-2	12	12
5	2,5,-3,8,-9,-7,4,11,-2	3	12
6	2,5,-3,8,-9,-7,4,11,-2	-4	12
7	2,5,-3,8,-9,-7,4,11,-2	4	12
8	2,5,-3,8,-9,-7,4,11,-2	15	15
9	2,5,-3,8,-9,-7,4,11,-2	13	15

Abbildung 13.5: Beispielablauf für Kadanes Algorithmus

Probleme wachsen lassen

Das algorithmische Prinzip, das Sie im vorangegangenen Abschnitt bei den Algorithmen maxSumGrowing und Kadane kennengelernt haben, hat den etwas nichtssagenden Namen »dynamisches Programmieren«.

Das Prinzip des dynamischen Programmierens

Beim *dynamischen Programmieren* beginnt man mit der Lösung von kleinen Teilproblemen und weitet diese Lösungen iterativ auf immer größere Teilprobleme aus. Dabei merkt man sich einmal gefundene Teillösungen immer mindestens so lange, wie sie noch für die Lösung größerer Teilprobleme benötigt werden.

Die Methode wird insbesondere bei Optimierungsproblemen angewendet, wenn es also zum Beispiel darum geht, die kleinste, größte oder beste einer Reihe von Auswahlmöglichkeiten zu finden. Das Prinzip ist aber auch bei anderen Aufgaben nützlich, wie Sie zum Beispiel weiter unten im Fall der Fibonacci-Zahlen sehen können.

Dynamisches Programmieren verglichen mit Teilen und Herrschen

Beim dynamischen Programmieren versucht man, die Lösung eines großen Problems aus den Lösungen kleinerer Teilprobleme aufzubauen. Dynamische Programmieren verwendet also das Prinzip »Teilen und Herrschen«. Allerdings unterscheiden sich Algorithmen, die dynamisches Programmieren einsetzen, auch in einigen Aspekten von typischen Algorithmen nach dem Konzept »Teilen und Herrschen«:

Typisches Teilen und Herrschen	*Typisches dynamisches Programmieren*
Top-down: beginnt mit dem Zielproblem und reduziert die Problemgröße so lange, bis die Abbruchbedingung erreicht wird	*Bottom-up*: beginnt mit der Lösung kleiner Probleme und weitet die Problemgröße immer mehr aus, bis am Ende das Zielproblem gelöst ist
Rekursiv: muss Zwischenergebnisse nicht extra abspeichern	*Iterativ*: speichert Zwischenergebnisse mindestens so lange, wie sie für nachfolgende Berechnungen noch benötigt werden
Teilen in der Mitte: die Aufgaben werden möglichst gleichmäßig aufgeteilt	*Teilen am Rand*: zur Lösung einer Aufgabe werden die Lösungen von etwas kleineren Aufgaben verwendet

Beispiel 1: Minimum

Ein ganz einfaches Beispiel für einen Algorithmus, der das Prinzip des dynamischen Programmierens einsetzt, haben Sie schon in Kapitel 2 kennengelernt. Wir wollen in einer Reihe von n Bauarbeitern denjenigen herausfinden, der am wenigsten wiegt. Kurz gesagt geht es also darum, durch paarweise Vergleiche das Minimum in einer Liste zu finden. Der Algorithmus Methode2 im Abschnitt *Methode 2: Der Schwerere scheidet aus* in Kapitel 2 geht nach dem Prinzip »der Schwerere scheidet aus« vor: Die Bauarbeiter müssen sich nacheinander auf der Waage mit dem bisher leichtesten Kollegen vergleichen lassen. Dass es sich dabei um eine Anwendung des Prinzips der dynamischen Programmieren handelt, lässt sich wie folgt einsehen: Wenn wir die ersten i Männer betrachten und den leichtesten von ihnen herausnehmen, dann wollen wir das Gewicht dieses leichtesten Mannes Min_i nennen. Für $i = 1$ ist Min_i einfach das Gewicht des ersten Mannes. Für $i > 1$ können wir Min_i einfach wie folgt ausrechnen:

$$\text{Min}_i = \min(\text{Min}_{i-1}, \text{Gewicht des } i\text{-ten Mannes})$$

Lassen Sie uns daraus einen Algorithmus machen, der das Minimum aller n Werte in einem Array a berechnet.

```
min(a)                                  a: Array mit n Werten
    Min sei ein Array mit n Werten
    Min[1] ← a[1]                       fange mit dem kleinsten Problem an
    if n > 1 then                       wenn es noch weitere Werte in a gibt ...
        for i ← 2, ..., n               ... gehe alle Werte in a der Reihe nach durch
            Min[i] ← min(Min[i − 1], a[i])      berechne Lösung für i
    return Min[n]
```

Man kann sehen, dass zur Berechnung von **Min** [i] nur der Wert **Min** [$i - 1$] des um eins kürzeren Arrays benötigt wird. Darum können wir Platz sparen und uns nur noch die Lösung des letzten Problems in einer einzelnen Variablen **Min** merken:

```
min(a)                                  a: Array mit n Werten
    Min ← a[1]                          Min: kleinster bisher gesehener Wert
    if n > 1 then                       wenn es noch weitere Werte in a gibt ...
        for i ← 2, ..., n               ... gehe alle Werte in a der Reihe nach durch
            if a[i] < Min then          sobald ein kleinerer Wert in a gefunden wurde ...
                Min ← a[i]              ... merke ihn dir
    return Min
```

Ein Beispiel ist in Abbildung 13.6 zu sehen, einmal mit Array **Min** (oben) und einmal in der platzsparenden Variante (unten). In beiden Versionen hat der Algorithmus eine Laufzeit von $\mathcal{O}(n)$.

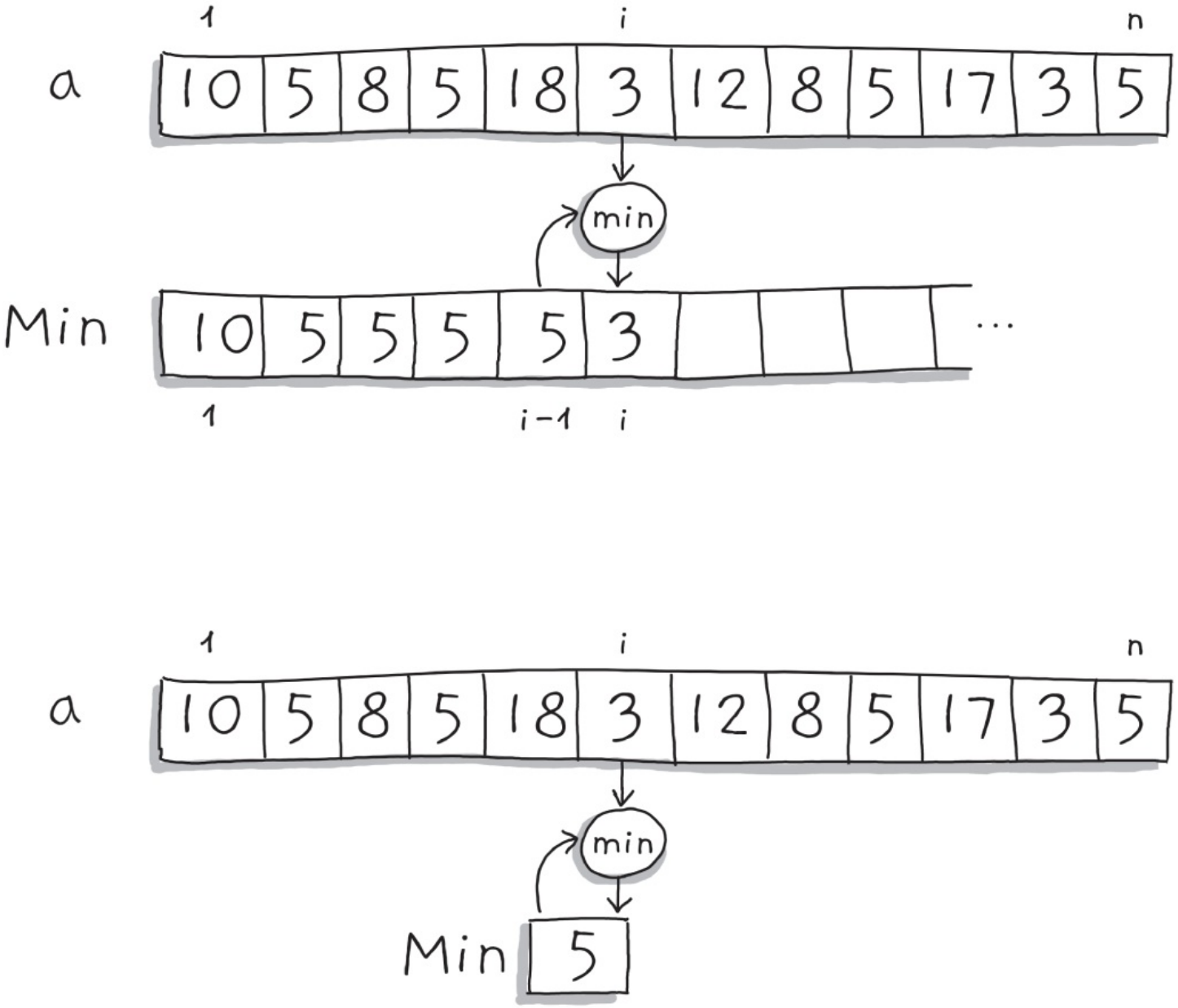


Abbildung 13.6: Minimum mit dynamischem Programmieren

Beispiel 2: Fibonacci-Zahlen

Das folgende Beispiel ist nach Leonardo Fibonacci benannt, der Anfang des 13. Jahrhunderts Rechenmeister in Pisa war. Nehmen wir an, ein Kaninchen wirft einmal im Monat Junge, die ihrerseits zwei Monate nach ihrer Geburt erstmals Junge bekommen können (siehe Abbildung 13.7). Gehen wir der Einfachheit halber davon aus, dass jeder Wurf aus genau einem Kaninchenpaar besteht, dann können wir nämlich einfach die Kaninchenpaare zählen. Zu Anfang (Zeitpunkt 1) haben wir zwei frisch geborene Kaninchen. Einen Monat später (Zeitpunkt 2) ist das Paar ausgewachsen und einen weiteren Monat später (Zeitpunkt 3) hat sich die Zahl der Kaninchen verdoppelt. Nach drei Monaten sind es dann schon drei Paare, nach vier Monaten fünf Paare, nach fünf Monaten acht Paare und so weiter. Die »Kaninchenaufgabe« besteht nun darin, zu einem gegebenen Monat n die Anzahl Fib_n der Kaninchenpaare zu berechnen. Die Zahl Fib_n wird *die* n *-te Fibonacci-Zahl* genannt.

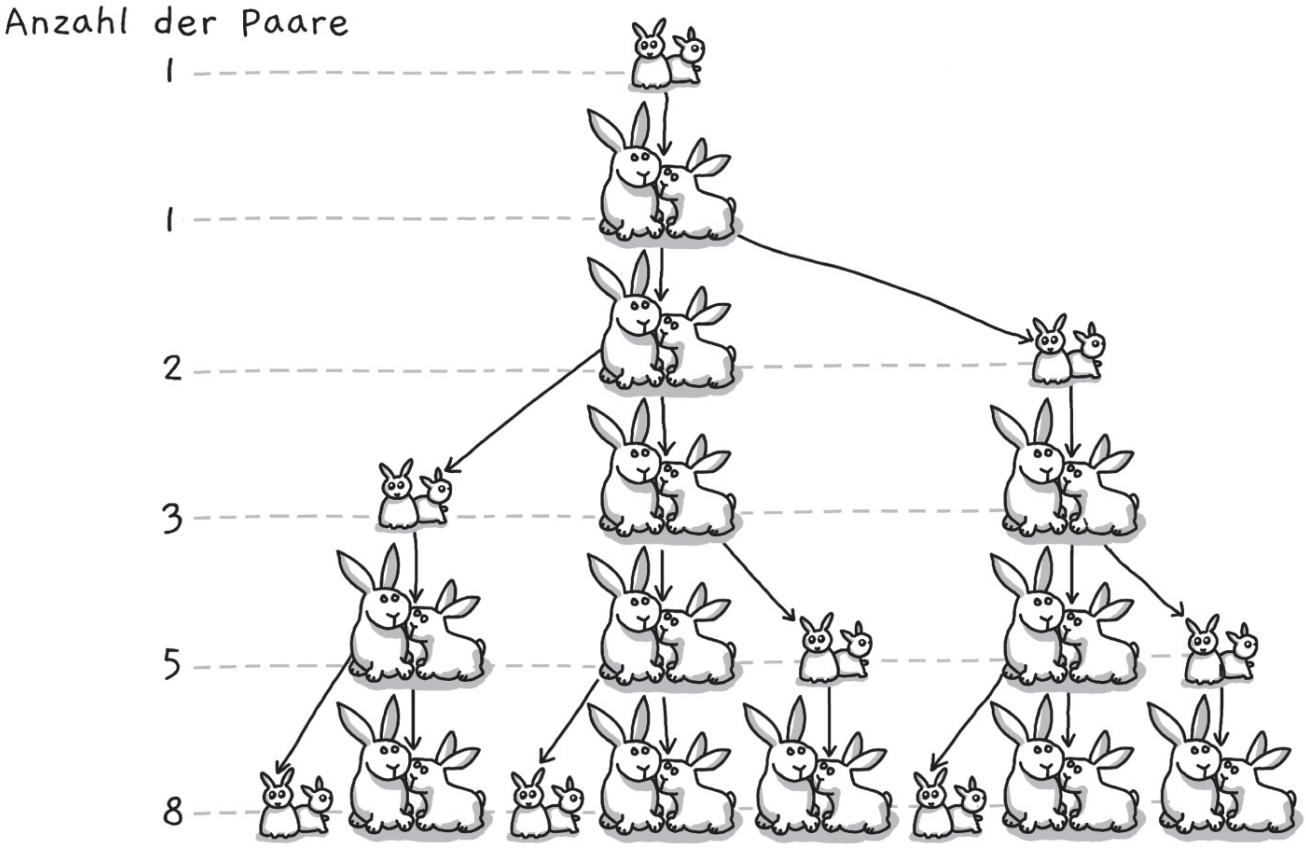


Abbildung 13.7: Fibonaccis Kaninchen vermehren sich wie die Karnickel.

Für die ersten zwei Zeitpunkte 1 und 2 schreiben wir das einfach hin: $\mathrm{Fib}_1 = \mathrm{Fib}_2 = 1$. Zu einem Zeitpunkt $n \geq 3$ gibt es zum einen alle Kaninchen, die es schon einen Monat vorher gegeben hat (aus ethischen Gründen sterben Kaninchen in dieser Aufgabe nicht), also Fib_{n-1} Stück. Außerdem kriegen alle Kaninchen, die zum Zeitpunkt $n-1$ bereits ausgewachsen waren, zum Zeitpunkt n Nachkommen. Das sind alle Kaninchen, die schon zum Zeitpunkt $n-2$ gelebt haben, also Fib_{n-2} Exemplare. Insgesamt gibt es also zum Zeitpunkt n

$$\mathrm{Fib}_n = \mathrm{Fib}_{n-1} + \mathrm{Fib}_{n-2}$$

Kaninchen.

Aus dieser Rekursionsgleichung kann man natürlich wieder einen rekursiven Algorithmus machen:

```
fib(n)
    if n ≤ 2 then return 1          Abbruchbedingung
    else return fib(n − 1) + fib(n − 2)      Rekursion
```

Wenn Sie einmal einen wirklich, wirklich langsames Programm erleben möchten, dann programmieren Sie diesen Algorithmus und versuchen Sie zum Beispiel einmal die fünfzigste Fibonacci-Zahl damit zu berechnen. Wenn Ihnen das zu schnell geht, dann versuchen Sie es eben mit der hundertsten Fibonacci-Zahl. Die Laufzeit von fib(n) ist nämlich exponentiell in n und das macht schon bei mäßig großem n keine echte Freude mehr.

Abhilfe schafft – Sie werden es sicher erraten haben – die Methode des dynamischen Programmierens. Lassen Sie uns die Fibonacci-Zahlen hintereinander in einem Array namens **Fib** speichern. Die Zahlen im Array lassen sich ganz leicht nacheinander ausrechnen.

```
fib(n)
    if n ≤ 2 then return 1          Abbruchbedingung
    Fib[1] ← 1                      Initialisierung von Fib
    Fib[2] ← 1
    for i ← 3, ..., n               nacheinander alle Werte im Array berechnen
        Fib[i] ← Fib[i − 1] + Fib[i − 2]
    return Fib[n]
```

Auch dieser Algorithmus lässt sich platzsparend hinschreiben. Wir müssen uns lediglich die zwei letzten Werte **f1 = Fib** $[i-2]$ und **f2 = Fib** $[i-1]$ des Arrays merken. In der nächsten Runde wird die Summe von **f1** und **f2** das neue **f2**. Dabei muss man ein wenig aufpassen, dass man zuerst das alte **f2** nach **f1** verschiebt, bevor man **f2** mit dem neuen Wert überschreibt.

```
fib(n)
    if n ≤ 2 then return 1          Abbruchbedingung
    f1 ← 1                          Initialisierung von f1 ...
    f2 ← 1                          ... und f2
    for i ← 3, ..., n               die Schleife (n − 2)-mal wiederholen
        f ← f1 + f2                 f = Fib_i berechnen
        f1 ← f2                     das alte f2 nach f1 verschieben
        f2 ← f                      f2 für die nächste Runde neu setzen
    return f2
```

Abbildung 13.8 verdeutlicht die beiden Varianten von fib mit dynamischem Programmieren, einmal mit hohem (oben) und einmal mit niedrigem (unten) Platzverbrauch. Beide Varianten laufen in $\mathcal{O}(n)$.

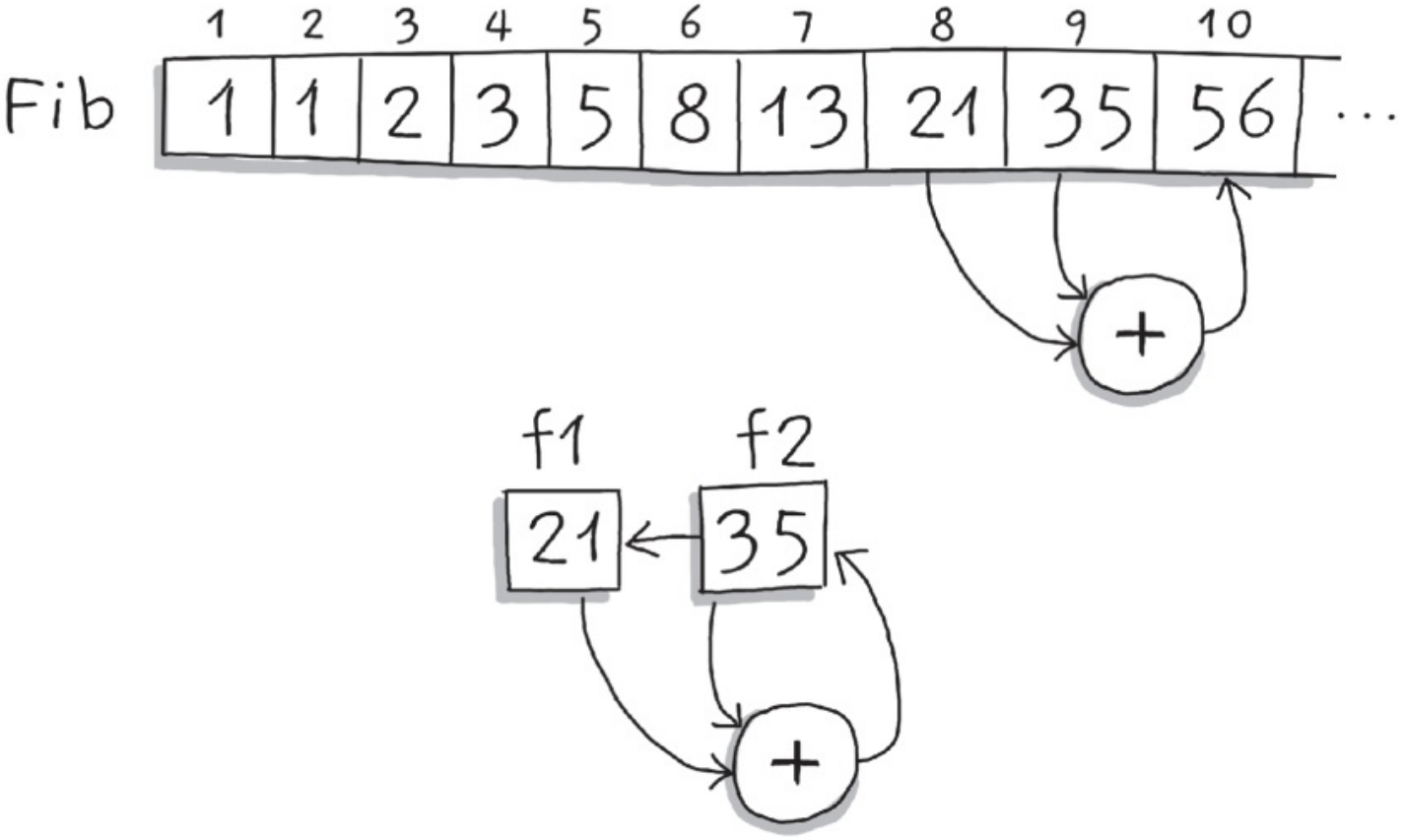


Abbildung 13.8: Berechnung der Fibonacci-Zahlen mit dynamischem Programmieren

Beispiel 3: Rucksack packen

Eine weitere, schon etwas kompliziertere Anwendung für das Prinzip des dynamischen Programmierens haben Sie bereits in Kapitel 8 beim Thema »Rucksack packen« im Abschnitt *Probleme langsam wachsen lassen* kennengelernt. Ziel des Aufgabe ist es, einen Rucksack mit möglichst wertvollen Objekten zu füllen, ohne dabei die maximale Tragkraft T des Rucksacks zu überschreiten. Zur Auswahl stehen n Objekte obj_1, obj_2, obj_3, $\cdots$, obj_n; T sei eine natürliche Zahl. Der Algorithmus KnapsackDP füllt eine Tabelle **Tab** mit $T+1$ Zeilen und $n+1$ Spalten. Dabei bedeutet **Tab** [t , i] so viel wie »der maximale Gesamtwert einer Auswahl aus den ersten i Objekte obj_1, obj_2, $\cdots$, obj_i , die sich zusammen in einen Rucksack mit der maximalen Tragkraft t packen lassen«. Zur Berechnung verwenden wir folgende Rekursionsgleichung:

$$\text{Tab}[t,i] = \max \begin{cases} \text{Tab}[t - \text{gewicht}_i, i-1] + \text{wert}_i \\ \text{Tab}[t, i-1] \end{cases}$$

Dabei sollen gewicht_i das Gewicht und wert_i der Wert des i -ten Objekts obj_i sein. Der maximale Gesamtwert, den man mit einer Beladung des Rucksacks erreichen kann, steht am Ende in der Tabellenzelle **Tab** [T , n]. Den fertigen Algorithmus können Sie im Abschnitt *Probleme langsam wachsen lassen* in Kapitel 8 nachlesen. Ein zweiter Teil des Algorithmus KnapsackDP dient dazu, eine optimale Auswahl an Objekten zu berechnen. Dies wird durch »Backtracking« erreicht. Auf diese Methode werden wir noch einmal im Abschnitt *Der Weg zum Optimum* zurückkommen.

Vergleich von Texten

Im Zeitalter der Autokorrektur kann es leicht zu Missverständnissen kommen. Schnell ins Mobiltelefon eingetippt könnte zum Beispiel aus »Bitte lautlos schreiten« ganz leicht »Bitte laut losschreien« werden. Offenbar muss nur ein Leerzeichen an die falsche Stelle wandern und ein »t« wegfallen und schon wird aus einer Ermahnung zur ruhigen Bewegung eine Aufforderung zum Lärmen.

Etwas Ähnliches passiert auch in der Biologie. Gene sind so etwas wie lange Texte, die nur aus den Buchstaben A, C, G und T bestehen. Im Lauf der Evolution verändern sich diese Texte, sie mutieren. So können sich einzelne Buchstaben verändern, beispielsweise ein A in ein C verwandeln (*Substitution*), oder es kann ein Buchstabe wegfallen (*Löschung*) oder neu hinzukommen (*Einfügung*). Es gibt noch weitere Möglichkeiten, wie sich Gensequenzen verändern können, aber diese drei *Punktmutationen* genügen bereits, dass sich jedes Gen im Prinzip in jedes andere verwandelt haben könnte. Betrachtet man die Sequenzen sich entsprechender Gene eines entfernten Vorfahren, sagen wir mal vom *Australopithecus*, und die vom modernen Menschen, so kann man fragen: »Wie hat sich das eine aus dem anderen entwickelt?« Leider war damals niemand dabei, der heute noch darüber berichten könnte, und so ist die Antwort auf diese Frage natürlich schwierig. Zwischen den wenigen fossilen Überresten früher Menschen, die Forscher bislang finden konnten, liegen riesige Zeiträume, in denen jede Menge Evolution passiert sein könnte.

Die Editierdistanz

Nun sind Mutationen recht selten und noch seltener werden sie von der Evolution angenommen und setzen sich in einer Population durch. Darum könnte man auch fragen: »Wie viele Punktmutationen muss es mindestens gegeben haben, damit sich aus dem einen Gen das andere Gen entwickeln konnte?« Man nennt die gesuchte Zahl auch die Editierdistanz zwischen den beiden Gensequenzen.



Die *Editierdistanz* (auf Englisch: *edit distance*, auch *Levenshtein-Distanz* genannt) zwischen zwei Strings ist die *minimale* Anzahl an Substitutionen, Einfügungen oder Löschungen einzelner Buchstaben, die man benötigt, um den einen String in den anderen umzuwandeln.

Das Gute an der Editierdistanz ist, dass man zur ihrer Berechnung keine Zeitmaschine oder riesige Mengen an Fossilien benötigt, sondern lediglich die beiden Gensequenzen – und einen geeigneten Algorithmus. Natürlich gibt die Editierdistanz keinen Aufschluss über die tatsächlichen historischen Begebenheiten, die letztlich zur Umwandlung einer Gensequenz in eine andere geführt haben. So könnte sich das deutsche Wort »Fenster« durch nur drei Änderungen aus dem Lateinischen Wort »Fenestra« entwickelt haben (Abbildung 13.9), in Wahrheit verlief die Entwicklung jedoch etwas umständlicher. Dennoch ist die Editierdistanz sehr nützlich bei der Suche nach ähnlichen Strings, egal ob es sich dabei um Gene oder irgendeine andere Art von Text handelt. Wenn zum Beispiel die Wikipedia ihre Administratoren darüber unterrichten will, wie stark ein Benutzer einen Artikel verändert hat, dann rechnet sie eine Art Editierdistanz aus. Bleibt nur die Frage: Wie macht man das?

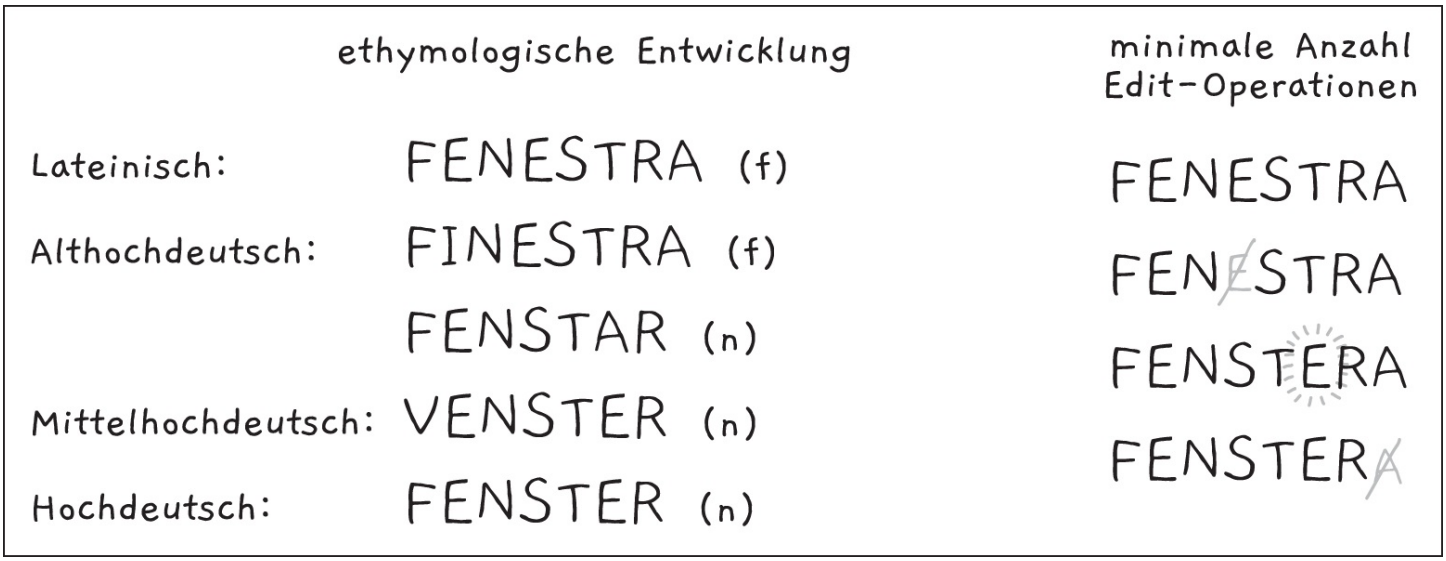


Abbildung 13.9: Die Editierdistanz entspricht nicht unbedingt den historischen Tatsachen.

Problem: Editierdistanz

Eingabe: zwei Strings a und b

Berechne: die Editierdistanz zwischen a und b, also die minimale Anzahl an Substitutionen, Einfügungen oder Löschungen einzelner Buchstaben, die man benötigt, um a in b umzuwandeln

Strings alignieren

Bevor wir das Problem lösen, lassen Sie es uns noch einmal umformulieren. Stellen Sie sich vor, sie wollen die beiden Strings a = »LIEBEN« und b = »FLIEGE« miteinander vergleichen. Dann können Sie die Wörter einfach untereinander schreiben und die Unterschiede zählen, also die Anzahl der unterschiedlichen Buchstaben bestimmen (siehe Abbildung 13.10). In diesem Fall wären alle sieben Buchstaben unterschiedlich. Wenn für die Umwandlung des einen Strings in den anderen ausschließlich Substitutionen

verwendet werden dürfen, so würde man also sieben Änderungen benötigen. Sind auch Löschungen und Einfügungen möglich, geht es sogar mit weniger Änderungen. Dafür erlauben wir, dass man in den Strings spezielle Lückenzeichen »–«, die sonst nicht in den Strings vorkommen dürfen, einfügen darf, und zwar an beliebigen Stellen, also auch am Anfang oder am Ende. Danach müssen beide Strings wieder gleich lang sein, damit wir sie untereinander schreiben und die Buchstaben vergleichen können. Wenn beide Strings untereinander geschrieben sind wie in Abbildung 13.10, so nennt man das ein *Alignment* der beiden Strings.



Abbildung 13.10: Drei unterschiedliche Alignments von »LIEBEN« und »FLIEGE«



Ein *Alignment* zwischen zwei Strings a und b erhält man, wenn man in die Strings an beliebigen Stellen Lückenzeichen »–« einfügt, bis sie gleich lang sind, und sie danach übereinander schreibt, sodass in jeder Spalte von beiden Strings je ein Buchstabe (oder Lückenzeichen) steht.

Ein Alignment zwischen zwei Strings ist *optimal*, wenn es möglichst wenig Spalten im Alignment gibt, die unterschiedliche Buchstaben oder ein Lückenzeichen enthalten.

Ein Lückenzeichen in der oberen Zeile entspricht dem Einfügen eines Buchstabens bei der Umwandlung von »LIEBEN« zu »FLIEGE«, steht hingegen unten ein Lückenzeichen, so entspricht dies dem Löschen eines Buchstabens. Wenn wir es schaffen, durch eventuelles Einfügen von Lückenzeichen in beide Strings die Zahl der Unterschiede zwischen den Strings zu minimieren, haben wir ein *optimales* Alignment zwischen den beiden Strings gefunden. Die Anzahl der Unterschiede im optimalen Alignment ist dabei gerade die Editierdistanz zwischen den Strings. Anders ausgedrückt: Statt des Editierdistanz-Problems können wir auch das Alignment-Problem lösen.

Problem: optimales Stringalignment

Eingabe: zwei Strings a und b

Berechne: die Anzahl der Unterschiede in einem optimalen Alignment zwischen a und b (entspricht der Editierdistanz zwischen a und b)

Im Moment interessieren wir uns nur für die Anzahl der Unterschiede im optimalen

Alignment; später werden wir noch herausfinden, wie man das optimale Alignment selbst bestimmt (siehe Abschnitt *Das optimale Stringalignment*).

Arbeitsteilung auf der Alignmentbaustelle

Angenommen, es soll ein optimales Alignment zwischen »LIEBEN« und »FLIEGE« berechnet werden. Zum Glück müssen Sie nicht alles alleine machen, sondern sind nur für die letzte Spalte des Alignments verantwortlich. Für Sie gibt es nur drei Möglichkeiten, zwischen denen Sie sich entscheiden müssen (siehe Abbildung 13.11:



Abbildung 13.11: Drei Möglichkeiten, wie die letzte Spalte eines Alignments aussehen kann

1. Entweder stehen in der letzten Spalte des Alignments die beiden letzten Buchstaben von »LIEBEN« und »FLIEGE«, das heißt, die letzte Spalte ist $\binom{N}{E}$.
2. Oder in der letzten Spalte des Alignments steht oben der letzte Buchstabe von »LIEBEN« und unten ein Lückenzeichen, das heißt, die letzte Spalte ist $\binom{N}{-}$.

3. Oder in der letzten Spalte des Alignments steht oben ein Lückenzeichen und unten der letzte Buchstabe von »FLIEGE«, das heißt, die letzte Spalte ist $\binom{-}{E}$.

Prinzipiell gäbe es noch die Möglichkeit, dass in der letzten Spalte sowohl oben als auch unten Lückenzeichen stehen, aber eine derartige Alignmentspalte würde nur unnötige Kosten verursachen und kommt darum in einem optimalen Alignment ohnehin nicht vor.

In allen drei Fällen verursacht die letzte Spalte also einen Unterschied im fertigen Alignment; nur wenn »LIEBEN« und »FLIEGE« mit demselben Buchstaben enden würden, hätten man ohne einen Unterschied in der letzten Spalte auskommen können. Weitere Unterschiede im Gesamtalignment hängen nun davon ab, wie gut sich die Teile der beiden Strings alignieren lassen, die nicht in der letzten Spalte stehen. Im Fall 1 wäre dies also ein Alignment von »LIEBE« und »FLIEG«, im Fall 2 ein Alignment von »LIEBE« und »FLIEGE« und im Fall 3 ein Alignment zwischen »LIEBEN« und »FLIEG«. Zum Glück müssen Sie sich nicht um diese optimalen Alignments kümmern, dafür sind Ihre Kollegen zuständig. Sobald Ihre Kollegen aber herausgefunden haben, wie viele Unterschiede es in jedem dieser drei Alignment geben würde, können Sie entscheiden, wie Sie die letzte Spalte Ihres Alignments gestalten sollten. Natürlich nehmen Sie nämlich die Variante, bei der die Gesamtzahl der Unterschiede herauskommt.

Optimale Alignments mit dynamischem Programmieren

Es ist also ziemlich leicht, ein optimales Alignment zwischen zwei Strings a und b zu berechnen, wenn man dabei auf optimale Alignments zwischen Strings zurückgreifen kann, die etwas kürzer als a und b sind. Aus dieser Beobachtung werden wir nun einen Algorithmus konstruieren. Dabei gehen wir nach dem Prinzip des dynamischen Programmierens vor, das heißt, wir fangen mit kleinen Alignments an und weiten diese nach und nach aus, bis wir am Ende ein optimales Alignment der kompletten Strings a und b berechnen können. Die Zwischenergebnisse speichern wir in einer Tabelle, die wir **Tab** nennen wollen. Nehmen wir an, dass a aus n Buchstaben besteht und b aus m Buchstaben, dann soll **Tab** $n+1$ Zeilen und $m+1$ Spalten haben (siehe Abbildung 13.12). Die Tabellenzelle **Tab** [i , j] in der i -ten Zeile und der j -ten Spalte soll dabei folgende Zahl speichern:

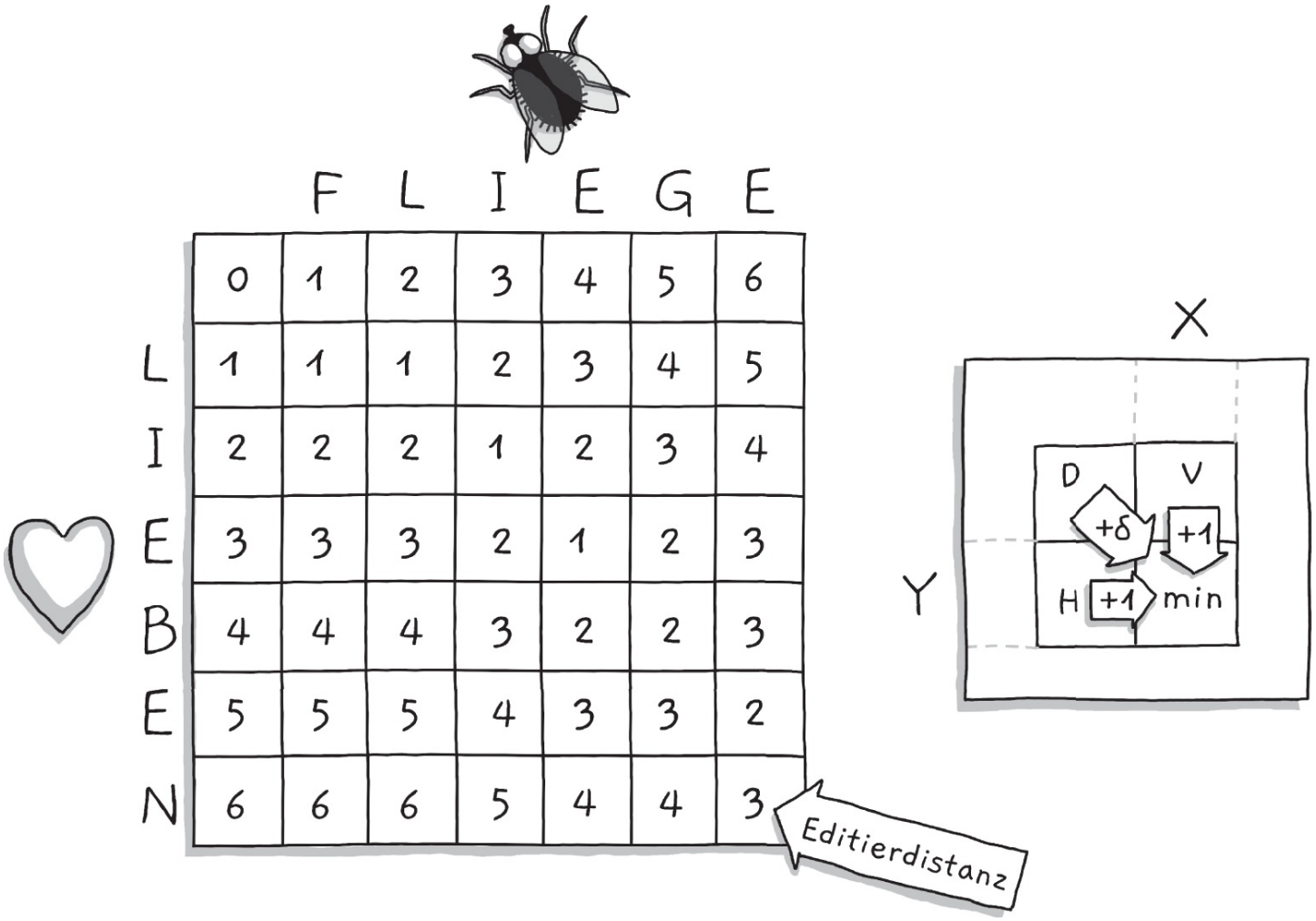


Abbildung 13.12: Das Schema (rechts), nachdem die Tabelle **Tab** (links) ausgefüllt wurde, um zwei Strings zu alignieren

> **Tab** [i , j] = »Die minimale Anzahl von Unterschieden in einem Alignment zwischen den ersten i Buchstaben von a und den ersten j Buchstaben von b «

Sind also zum Beispiel a = »LIEBEN« und b = »FLIEGE«, dann steht in **Tab** [3, 5] die Lösung des Alignmentproblems der beiden Strings »LIE« (die drei ersten Buchstaben von a) und »FLIEG« (die fünf ersten Buchstaben von b).

Rekursionsgleichungen

Wie Sie oben gesehen haben, gibt es bei der Berechnung von **Tab** [i , j] drei Fälle, die man berücksichtigen muss:

1. Entweder schreibt man $a[i]$, also das i -te Zeichen von a , und $b[j]$, also das j -te Zeichen von b , in eine gemeinsame Alignmentspalte. Je nachdem, ob diese beiden Zeichen ungleich oder gleich sind, verursachen sie einen Unterschied im optimalen Alignment oder eben nicht. Die Anzahl der Unterschiede, die man beim Alignment der restlichen $i-1$ Zeichen von a und $j-1$ Zeichen in b mindestens braucht, kann man in Tab in der $(i-1)$-ten Zeile und der $(j-1)$-ten Spalte ablesen.

Insgesamt kann man also schreiben:

$$\mathsf{Tab}[i,j] = \mathsf{Tab}[i-1,j-1] + \begin{cases} 0 & \text{, wenn } a[i] = a[j] \\ 1 & \text{, sonst} \end{cases}$$

2. Man kann auch das i -te Zeichen von a mit einem Lückenzeichen in eine Alignmentspalte schreiben. Das verursacht einen Unterschied. Die ersten $i-1$ Zeichen von a werden dann mit den ersten j Zeichen von b alignieren; das bedeutet weitere **Tab**[$i-1$, j] Unterschiede. Insgesamt ergibt sich in diesem Fall

   $$\mathsf{Tab}[i,j] = \mathsf{Tab}[i-1,j] + 1$$

3. Umgekehrt könnte man auch das j -te Zeichen von b mit einem Lückenzeichen in eine Alignmentspalte schreiben. Auch das verursacht einen Unterschied. Die ersten $j-1$ Zeichen von b werden dann mit den ersten i Zeichen von a alignieren; das bedeutet weitere **Tab**[i, $j-1$] Unterschiede. Insgesamt ergibt sich in diesem Fall dann

   $$\mathsf{Tab}[i,j] = \mathsf{Tab}[i,j-1] + 1$$

Am Ende wählt man für **Tab** [i , j] natürlich das Minimum der drei Fälle aus.

Abbruchbedingungen

Wenn man am oberen Rand der Tabelle **Tab** ($i = 0$) beziehungsweise an ihrem linken Rand ($j = 0$) ist, kann man zur Berechnung von **Tab** [i , j] natürlich nicht mehr auf ein weiter oben beziehungsweise weiter links liegendes Feld zurückgreifen. Das muss man zum Glück aber auch nicht, da man die Tabellenfelder an den Rändern auch ganz leicht direkt ausrechnen kann. Angenommen, wir wollen die ersten j Zeichen von b mit den ersten $i = 0$ Zeichen von a alignieren. Dann stehen in diesem Alignment in der oberen Zeile nur Lückenzeichen, das heißt, die Anzahl der Unterschiede ist j :

$$\mathsf{Tab}[0,j] = j$$

Entsprechend für $j = 0$:

$$\mathsf{Tab}[i,0] = i$$

Der Needleman-Wunsch-Algorithmus

Jetzt haben wir alle Gleichungen zusammen. Lassen Sie uns nun das Prinzip des dynamischen Programmierens anwenden. Statt die Rekursionsgleichungen in einem rekursiven Algorithmus zu verwenden, was sehr langsam wäre, füllen wir die Tabelle **Tab** einfach zeilenweise aus:

```
NeedlemanWunsch(a, b)          a: String mit n Zeichen, b: String mit m Zeichen
    Tab sei eine Tabelle mit n + 1 Zeilen und m + 1 Spalten
    for j ← 0, ..., m          initialisiere oberste Zeile von Tab
        Tab[0, j] ← j
    for i ← 1, ..., n          fülle Tab zeilenweise
        Tab[i, 0] ← i          erste Spalte initialisieren
        for j ← 1, ..., m      restliche Felder in der i-ten Spalte durchlaufen
            D ← Tab[i − 1, j − 1]      Möglichkeit 1: »diagonal«
            if a[i] ≠ b[j]             wenn die beiden Buchstaben unterschiedlich sind, ...
                D ← D + 1              ... erhöht sich die Zahl der Unterschiede im Alignment
            V ← Tab[i − 1, j] + 1      Möglichkeit 2: »vertikal«
            H ← Tab[i, j − 1] + 1      Möglichkeit 3: »horizontal«
            Tab[i, j] ← min(D, V, H)   optimale Möglichkeit auswählen
    return Tab[n, m]           Editierdistanz steht am Ende im Feld rechts unten
```

Der Algorithmus ist übrigens nach seinen beiden Erfindern benannt.

Statt zeilenweise hätte man die Tabelle zum Beispiel auch spaltenweise oder diagonal ausfüllen können. Wichtig ist nur eins: Bevor man **Tab** [i , j] berechnen will, muss man vorher alle Felder berechnet haben, die man dafür braucht, nämlich **Tab** [$i-1$, $j-1$] (das Feld »links oben«, also »diagonal«), **Tab** [$i-1$, j] (das Feld »oben«, also »vertikal«) und **Tab** [i , $j-1$] (das Feld »links«, also »horizontal«).

Da jedes Tabellenfeld nur einmal berechnet wird und die Berechnung pro Tabellenfeld nicht von den Längen n und m der beiden Strings abhängt, hat der Algorithmus eine Gesamtlaufzeit von $\mathcal{O}(n \cdot m)$.

Die Editierdistanz können wir nun also in recht kurzer Zeit berechnen und damit auch die minimale Anzahl an Unterschieden, die ein Alignment von zwei gegebenen Strings haben kann. Was noch fehlt, ist das Alignment selbst. Wie man ein optimales Alignment berechnet, erfahren Sie im nächsten Abschnitt. Lassen Sie uns aber zunächst noch einmal auf das Eisenbahnproblem zurückkommen.

Der Weg zum Optimum

Das Prinzip des dynamischen Programmierens ist eine Methode, die vor allen Dingen bei Optimierungsproblemen nützlich ist, also bei Aufgaben, bei denen es darauf ankommt, einen Zielwert zu optimieren. Beim Berechnen des Minimums (Abschnitt *Beispiel 1: Minimum*) ist der Zielwert offensichtlich: Hier möchten wir, dass der gefundene Wert möglichst klein ist. Beim Maximale-Teilsumme-Problem (Abschnitt *Der ultimative Maximale-Teilsumme-Algorithmus*) wollten wir hingegen den Wert der Teilsumme maximieren, beim Rucksackproblem (Abschnitt *Beispiel 3: Rucksack packen*) ist es die

Summe der Werte einer Auswahl an Objekten und das Stringalignment-Problem im vorangegangenen Abschnitt minimiert die Anzahl der Unterschiede im Alignment. Bislang haben wir uns in allen diesen Fällen damit begnügt, diese Zielgröße zu berechnen. Oft möchte man aber auch wissen, wie man zu dieser Zielgröße gelangt. Im Fall vom Minimum wäre das zum Beispiel die Frage danach, *an welcher Stelle* im Array das Minimum steht (siehe Tabelle 13.1). Das Ziel kennen wir also, aber nun wollen wir uns damit beschäftigen, wie man dort hinkommt.

Aufgabe	*Zielgröße*	*Der Weg zum Ziel*
Minimum	minimaler Zahlenwert	Position des Minimums im Array
Teilsummenproblem	maximale Teilsumme	Grenzen des optimalen Teilarrays
Rucksackproblem	maximale Wertesumme	optimale Auswahl an Objekten
Alignmentproblem	minimale Unterschiede	optimales Stringalignment

Tabelle 13.1: Verschiedene Optimierungsprobleme, die mit dynamischem Programmieren gelöst werden können

Entscheidungen merken

Wenn man seinen Weg durch ein Labyrinth finden will und vielleicht auch wieder zurück, dann kann es ungemein hilfreich sein, sich an jeder Abzweigung zu merken, wo man lang gegangen ist. Genau das gleiche müssen wir tun, wenn wir nicht nur das Ziel berechnen wollen, sondern auch wissen möchten, wie man es erreicht.

Wo ist das Minimum?

Bei der Suche nach dem Minimum ist es leicht, den Weg zum Ziel zu benennen: Sobald wir ein neues Minimum gefunden haben, merken wir uns, an welcher Stelle im Array dieses Minimum das erste Mal vorgekommen ist. Die Position des aktuellen Minimums speichern wir in der Variablen **MinPos**:

```
argmin(a)                    a: Array mit n Werten
    Min ← a[1]               Min: kleinster bisher gesehener Wert
    MinPos ← 1               Position von Min in a
    if n > 1 then            wenn es noch weitere Werte in a gibt, ...
        for i ← 2, ..., n    ... gehe alle Werte in a der Reihe nach durch
            if a[i] < Min then   sobald ein kleinerer Wert in a gefunden wurde, ...
                Min ← a[i]       ... merke ihn dir ...
                MinPos ← i       ... und auch seine Position
    return MinPos
```

So wie er hier steht, ist der Algorithmus noch etwas umständlich, denn eigentlich muss man Min gar nicht mehr speichern, wenn man sich schon MinPos gemerkt hat. Eine etwas knappere Variante des Algorithmus finden Sie in Kapitel 7 auf im Abschnitt *SelectionSort: So lange wählen, bis es passt*.

Welche Bahnstrecke bauen Sie?

Den maximalen Profit, den man mit dem Bau einer Bahnstrecke erzielen kann, hatten wir im Abschnitt *Der ultimative Maximale-Teilsumme-Algorithmus* mit dem Algorithmus Kadane berechnet. Um den Profit tatsächlich realisieren zu können, müsste man natürlich auch wissen, welchen Bauabschnitt man denn dafür nun bauen muss. Wenn wir also ein Array a mit n Zahlen gegeben haben, wollen wir die beiden Grenzen l und r mit $1 \leq l \leq r \leq n$ bestimmen, bei denen die Summe $a[l] + a[l+1] + \ldots + a[r]$ maximal wird. Dazu müssen wir uns wieder unsere Entscheidungen merken. Entscheidungen trifft der Algorithmus immer dann, wenn er bei der Berechnung von S_{end} und S ein Maximum berechnet. Damit Sie das besser erkennen können, sehen Sie hier den Algorithmus noch einmal mit ausführlichen if-Anweisungen:

```
Kadane(a)                          a: Array mit n Zahlen
    S ← a[1]                       S: maximale Teilsumme
    S_end ← a[1]                   S_end: maximale Endsumme
    for i ← 2 ... n                durchlaufe alle Zellen in a und ...
        if S_end + a[i] ≥ a[i]     ... wenn sich eine Verlängerung der Endsumme lohnt, ...
            S_end ← S_end + a[i]   ... verlängere die aktuelle Endsumme
        else                       ansonsten ...
            S_end ← a[i]           ... fange die neue Endsumme hier an
        if S_end > S               wenn eine neue maximale Teilsumme gefunden wurde, ...
            S ← S_end              ... wird sie in S gespeichert
    return S
```

In jedem Durchlauf der **for**-Schleife prüft der Algorithmus, ob er die aktuelle Endsumme verlängert oder ob es sich eher lohnt, die bisherige Endsumme zu verwerfen und ganz neu anzufangen. Wichtig ist, dass sich der Algorithmus merkt, wo die aktuelle maximale Endsumme anfängt. Dazu verwenden wir eine Variable l_{end}. Sobald der Algorithmus anschließend in der zweiten **if**-Anweisung feststellt, dass mit der aktuellen maximalen Endsumme eine neue maximale Teilsumme gefunden worden ist, muss er sich Anfang und Ende dieser maximalen Teilsumme merken. Dafür verwendet er die beiden Variablen **l** und **r**. Der Algorithmus sieht dann so aus:

```
Kadane(a)                                   a: Array mit n Zahlen
    S ← a[1]                                S: maximale Teilsumme
    S_end ← a[1]                            S_end: maximale Endsumme
    l ← 1                                   l: linke Grenze der maximalen Teilsumme
    r ← 1                                   r: rechte Grenze der maximalen Teilsumme
    l_end ← 1                               l_end: linke Grenze der maximalen Endsumme
    for i ← 2 ... n                         durchlaufe alle Zellen in a und ...
        if S_end + a[i] ≥ a[i]              wenn sich eine Verlängerung der Endsumme lohnt, ...
            S_end ← S_end + a[i]            ... verlängere die aktuelle maximale Endsumme
        else                                ansonsten ...
            S_end ← a[i]                    ... fange die neue maximale Endsumme hier an
            l_end ← i                       linke Grenze der maximale Endsumme merken
        if S_end > S                        wenn eine neue maximale Teilsumme gefunden wurde, ...
            S ← S_end                       ... wird sie in S gespeichert
            l ← l_end                       linke Grenze der maximalen Teilsumme merken
            r ← i                           rechte Grenze der maximalen Teilsumme merken
    return (l, r)                           Grenzen des optimalen Teilarrays zurückgeben
```

Den Pfad zurückfinden

Bei der Suche nach dem Minimum oder dem maximalen Teilarray genügen eine Handvoll Variablen zur Speicherung der optimalen Lösung. Anders ist das beim Rucksackproblem oder beim Alignieren von Strings. Eine Lösung des Rucksackproblems besteht in einer möglichst guten Auswahl von Objekten. Wenn insgesamt n Objekte zur Auswahl stehen, kann die Lösung also bis zu n Objekte enthalten, die man sich alle merken muss. Im Prinzip ist es zwar möglich, in jedem Feld der Tabelle **Tab** , in der die Lösungen der Zwischenergebnisse gespeichert werden, zusätzlich zur maximalen Wertesumme auch die zu dieser Summe gehörende optimale Objektauswahl zu speichern, nur wäre das mit einigem Speicheraufwand verbunden. Es gibt eine sparsamere und elegantere Möglichkeit: Statt jeweils die ganze Auswahl zu speichern, muss man sich in jeder Zelle der Tabelle nur die Entscheidung bezüglich des in dieser Spalte untersuchten Objekts merken. Sobald man die gesamte Tabelle ausgefüllt hat, kann man vom Zielfeld ganz rechts unten den Pfad der Entscheidungen zurückverfolgen und auf diese Weise die Auswahl konstruieren. Diesen Pfad rückwärts durch die Tabelle nennt man *Backtrace*. Tatsächlich kommt diese Methode ganz ohne zusätzlichen Speicherbedarf aus, da sich in jeder Zelle der vollständig ausgefüllten Tabelle stets die Entscheidung, die zum Wert in dieser Zelle geführt hat, rekonstruieren lässt und man darum auch weiß, in welche Richtung der Backtrace anschließend weitergehen muss.

Das haben Sie noch nicht verstanden? Kein Problem: In den nachfolgenden Abschnitten wird Ihnen das Prinzip des *Backtrace* am Beispiel des Stringalignment-Problems ausführlich und in aller Schönheit präsentiert.



Wie sich beim Rucksackproblem die optimale Objektauswahl entlang eines Pfades durch die Tabelle **Tab** ermitteln lässt, wird in Kapitel 8 im Abschnitt *Probleme langsam wachsen lassen* beschrieben.

Das optimale Stringalignment

Angenommen, wir haben für die beiden Strings »LIEBEN« und »FLIEGE« den Algorithmus NeedlemanWunsch (siehe den entsprechenden Abschnitt in diesem Kapitel) bis zum Ende durchgeführt und die gesamte Tabelle **Tab** ausgefüllt. Die minimale Anzahl an Unterschieden in einem Alignment können wir nun im Tabellenfeld unten rechts ablesen: Es sind drei (siehe Abbildung 13.13).

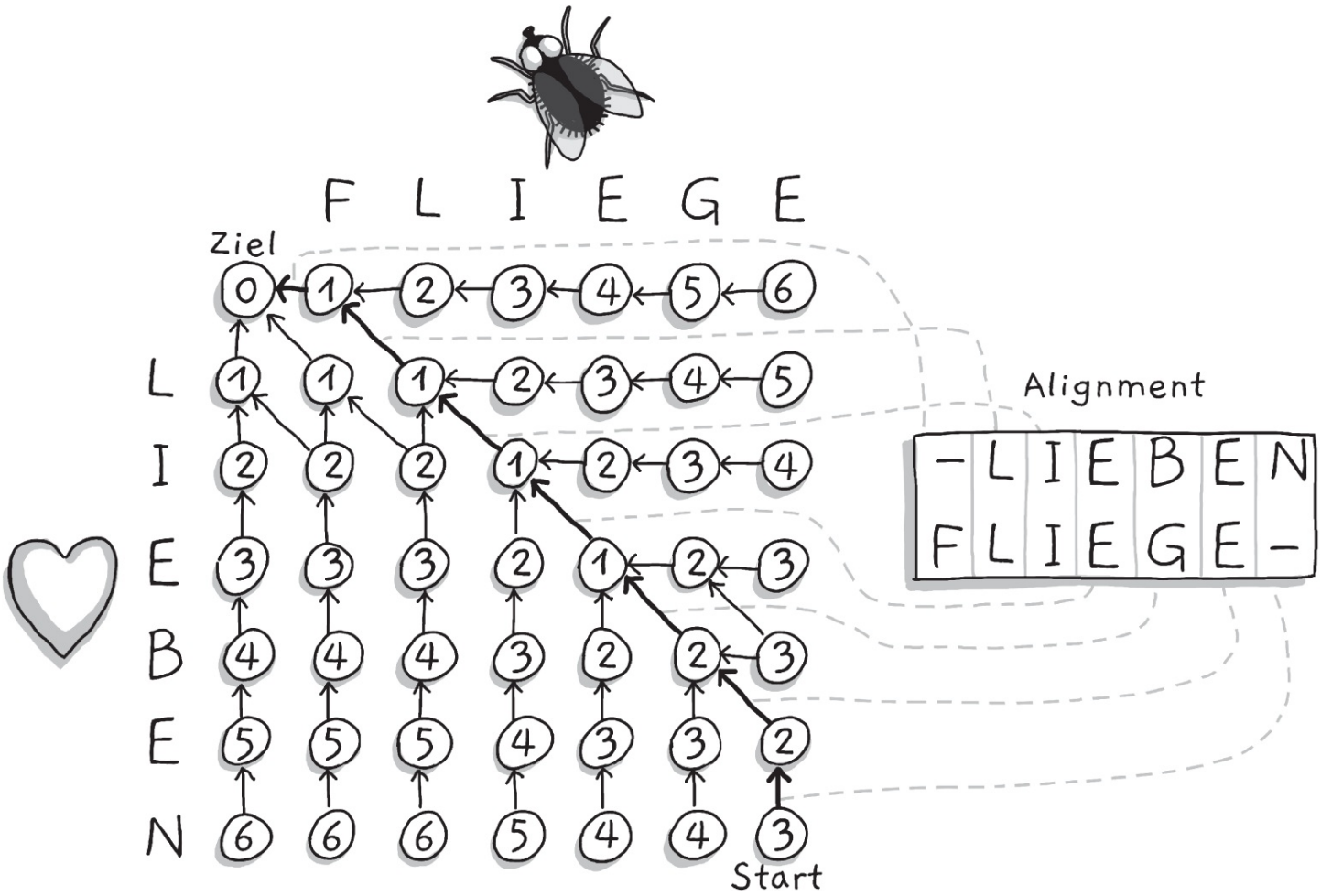


Abbildung 13.13: Die Tabelle als Graph dargestellt, durch den der Backtrace verläuft

Nun wollen wir auch ein Alignment bestimmen, das mit nur drei Unterschieden auskommt. Dazu beginnen wir mit der letzten Spalte des Alignments. Sie kennen die Geschichte bereits (siehe Abschnitt *Arbeitsteilung auf der Alignmentbaustelle*): Es gibt dafür drei Möglichkeiten, nämlich entweder $\binom{N}{E}$, $\binom{N}{-}$ oder $\binom{-}{E}$. Für welche Möglichkeit wir uns entscheiden, hängt von den Unterschieden in den »Restalignments« ab. Wie man aus **Tab** ablesen kann, lassen sich »LIEBE« und »FLIEG« mit drei

Unterschieden alignieren, bei »LIEBE« und »FLIEGE« sind es nur zwei Unterschiede und bei »LIEBEN« und »FLIEG« vier Unterschiede. Die wenigsten Unterschiede bekommen wir also bei der zweiten Option und entscheiden uns deswegen für sie. Die letzten Spalte steht damit fest: $\binom{N}{-}$ (siehe Abbildung 13.14).

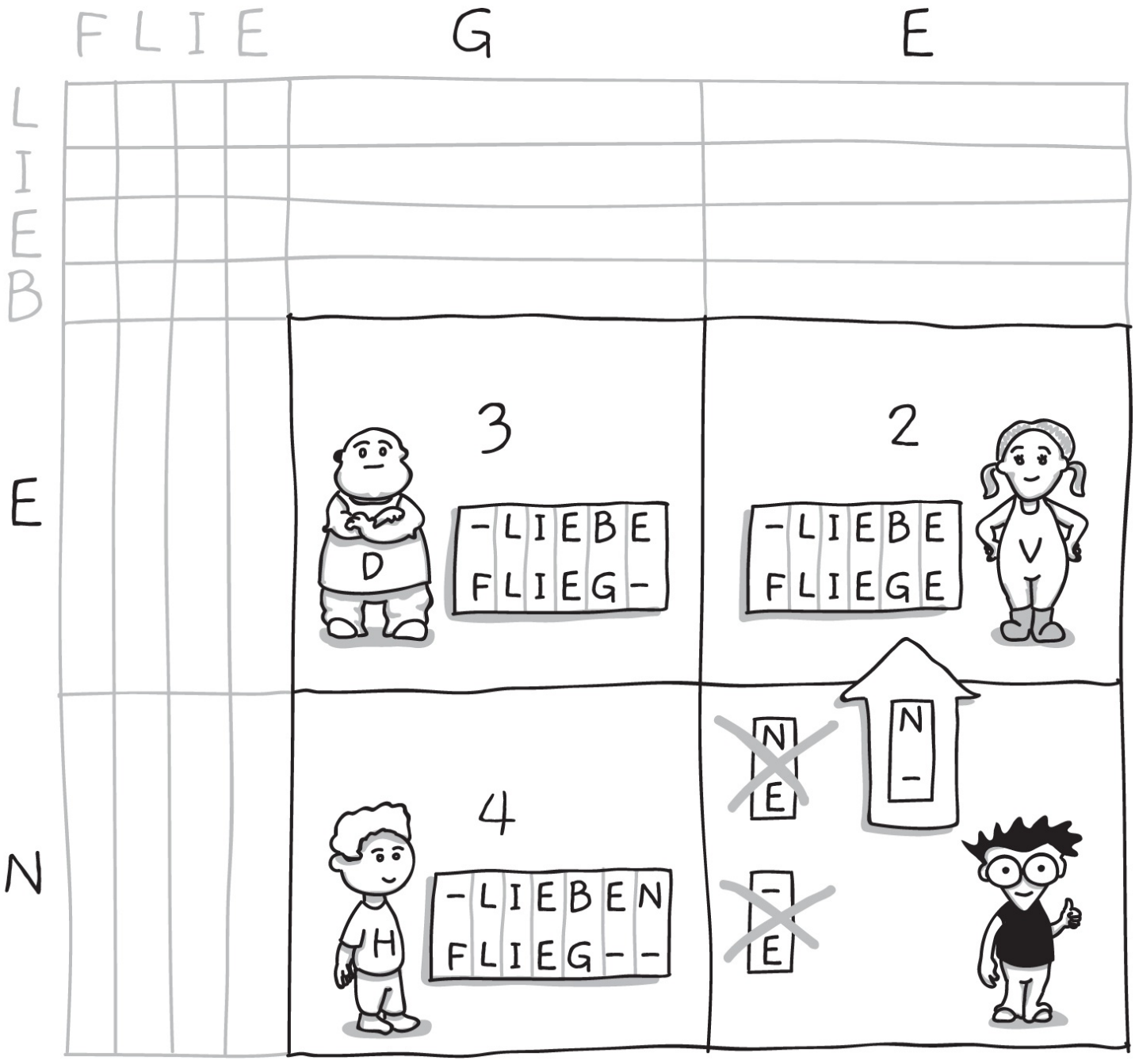


Abbildung 13.14: »Vertikal« ist die beste Option

Nun müssen wir mit dem Rest weitermachen, das heißt, wir brauchen ein optimales Alignment zwischen »LIEBE« und »FLIEGE«. Wir folgen dem Pfad in der Tabelle von der 3 unten rechts senkrecht nach oben zur 2. Wieder stehen drei Möglichkeiten zur

Auswahl, diesmal sind es $\binom{E}{E}$, $\binom{E}{-}$ und $\binom{-}{E}$. Hier entscheiden wir uns für die erste Option, also für den diagonalen Weg nach links oben. Das setzen wir so lange fort, bis wir schließlich in der Tabelle ganz oben links angekommen sind. Bei jedem Schritt haben wir eine neue Spalte des optimalen Alignments produziert und dem bisherigen Alignment vorangestellt.

Zum Abschluss sehen Sie die Methode noch einmal in Pseudocode. Die Berechnung von **D**, **V** und **H** läuft dabei genauso wie im Algorithmus NeedlemanWunsch. Bevor wir die drei Werte berechnen, müssen wir aber erst einmal prüfen, ob wir bereits am oberen oder linken Rand der Tabelle angekommen sind und eine der Abbruchbedingungen greift. Sind wir am oberen Rand angekommen ($i = 0$), so geht es nur links weiter, das heißt, die restlichen Buchstaben von b müssen mitaligniert werden. Sind wir andererseits am linken Rand ($j = 0$), so geht es nur oben weiter und die restlichen Buchstaben von a werden mitaligniert.

getAlignment(a, b, Tab) — *a: String mit n Zeichen, b: String mit m Zeichen; Tab: Tabelle, berechnet mit NeedlemanWunsch(a, b)*

- i ← n — *i: aktuelle Zeile in Tab; beginne unten*
- j ← m — *j: aktuelle Spalte in Tab; beginne rechts*
- A ← $()$ — *A: Alignment, am Anfang leer*
- **while** (i > 0) oder (j > 0) — *solange man noch nicht oben links angekommen ist*
  - **if** i = 0 — *am oberen Rand der Tabelle:*
    - A ← $\binom{-}{b[j]}$ A — *erzeuge neue Alignmentspalte*
    - j ← j − 1 — *gehe eins nach links*
  - **else if** j = 0 — *am linken Rand der Tabelle:*
    - A ← $\binom{a[i]}{-}$ A — *erzeuge neue Alignmentspalte*
    - i ← i − 1 — *gehe eins nach oben*
  - **else**
    - D ← Tab[$i-1, j-1$] — *Möglichkeit 1: »diagonal«*
    - **if** $a[i] \neq b[j]$ — *wenn die beiden Buchstaben unterschiedlich sind, ...*
      - D ← D + 1 — *... erhöhe die Zahl der Unterschiede im Alignment*
    - V ← Tab[$i-1, j$] — *Möglichkeit 2: »vertikal«*
    - H ← Tab[$i, j-1$] — *Möglichkeit 3: »horizontal«*
    - **if** D = Tab[i, j] — *wenn »diagonal« optimal ist:*
      - A ← $\binom{a[i]}{b[j]}$ A — *erzeuge neue Alignmentspalte*
      - i ← i − 1 — *gehe diagonal nach oben ...*
      - j ← j − 1 — *... links*
    - **else if** V = Tab[i, j] — *wenn »vertikal« optimal ist:*
      - A ← $\binom{a[i]}{-}$ A — *erzeuge neue Alignmentspalte*
      - i ← i − 1 — *gehe eins nach oben*
    - **else** — *sonst ist »horizontal« optimal:*
      - A ← $\binom{-}{b[j]}$ A — *erzeuge neue Alignmentspalte*
      - j ← j − 1 — *gehe eins nach links*
- **return** A — *Alignment fertig*

Kapitel 14

Näherungslösungen

IN DIESEM KAPITEL

Heuristiken

Approximation

Gier

Heuristiken

Der Begriff *Heuristik* ist ein vornehmes Wort für das, was man gemeinhin als »Faustregel« bezeichnet, also eine Regel, die zwar nicht immer stimmen muss, sich aber in der Praxis bewährt hat und deren Anwendung häufig zu ausreichend richtigen oder guten Ergebnissen führt. Ein Beispiel für eine Faustregel wäre: »große Männer sind schwerer als kleine Männer«. Das ist im Allgemeinen natürlich falsch, denn kleine dicke Männer können durchaus schwerer als große dünne Männer sein, aber von der Tendenz ist an der Faustregel natürlich etwas Wahres dran.

Derartige nützliche, wenn auch etwas unsichere Regeln kann man sich bei Algorithmen auf verschiedene Weise zunutze machen. Generell gibt es Heuristiken in zwei unterschiedlichen Geschmacksrichtungen:

- Eine Heuristik kann eine *Strategie* sein, die einem dabei hilft, schneller oder mit geringerem Aufwand zur richtigen Lösung zu kommen, beispielsweise durch eine geschickte Wahl der Reihenfolge, in der gesucht wird.

  Wenn Sie Ihren Haustürschlüssel suchen, dann wird es in den meisten Fällen ratsamer sein, zunächst in Ihren Taschen, auf Ablagen oder am Schlüsselbrett zu suchen als zum Beispiel im Kühlschrank oder Backofen. Natürlich könnte sich ihr Schlüssel *im Prinzip* überall versteckt haben, doch es gibt Orte, die schon etwas naheliegender sind als andere, und dann ist es eine gute Idee, zunächst an den Stellen nachzuschauen, wo sich Schlüssel am liebsten aufhalten. Wenn Sie alles absuchen, werden Sie ganz sicher früher oder später Ihren Schlüssel finden. Egal ob mit oder ohne Heuristik. Nur mit Heuristik haben Sie eben bessere Chancen, dass es früher geschieht. Die Heuristik könnte sich allerdings auch irren, etwa wenn der Schlüssel – aus welchen skurrilen Gründen auch immer – doch im Gefrierfach liegen sollte. Wenn man Pech hat, kann sich durch solch eine Heuristik die Suchzeit verlängern, ganz scheitern wird die Suche

jedoch nicht.

- Mit Heuristik kann andererseits auch eine Methode gemeint sein, die das Problem nicht exakt löst, sondern mit der man sich einer Lösung nur mehr oder weniger gut *annähert*.

  Heuristiken dieser Art werden oft bei Optimierungsproblemen angewendet, die generell schwierig zu lösen sind, beispielsweise bei NP-schweren Problemen. Mit einer solchen Heuristik lassen sich die Komplexität der Berechnung verringern und die Laufzeit verkürzen, allerdings zu dem Preis, dass man eventuell kein optimales Ergebnis mehr erhält. Eine mögliche Heuristik für die Entscheidung, was man aus seinem Leben machen soll und welche Ausbildung und welcher Beruf der richtige ist, wäre zum Beispiel: »Mach genau das, was deine Eltern dir raten!« Mit einer geeigneten Heuristik kann man auch bei schweren Aufgaben oft zu einer guten Lösung kommen, in vielen Fällen vielleicht sogar zum Optimum gelangen, doch dafür gibt es keine Garantie. Manche Heuristiken liefern zuweilen sogar ziemlich schlechte Lösungen ab. Die Eltern können sich ganz furchtbar irren und am Ende steht man mit einer Ausbildung da, die nicht zu einem passt, oder einem Beruf, den es gar nicht mehr gibt.



Der Begriff *Heuristik* steht in der Informatik für eine Methode, mit der Hilfe von »Faustregeln« zu praktikablen Lösungen zu kommen. Der Erfolg einer Heuristik ist dabei nicht garantiert.

Auf den nächsten Seiten werden Sie verschiedene Beispiele für beide Arten von Heuristiken kennenlernen.

Interpolationssuche

Dinge zu finden, ist eigentlich ziemlich einfach: Wenn man etwas sucht, muss man eben genau da nachschauen, wo es sich gerade befindet. Schon ist man fertig. Ein klein wenig schwieriger wird es natürlich, wenn man keine Ahnung hat, wo sich das Gesuchte befindet. Genau hier können dann Heuristiken ins Spiel kommen, denn auch wenn wir vielleicht nicht mit Sicherheit wissen, an welcher Stelle genau wir fündig werden, so haben wir eventuell doch eine grobe Vorstellung davon, wo sich die Suche am ehesten lohnen könnte. Mit einer guten Heuristik lässt sich außerdem auch ein bereits gut funktionierender Suchalgorithmus noch weiter beschleunigen.

Ein Beispiel für eine schnelle Suchmethode ist die *binäre Suche*, mit der man in logarithmischer Zeit Werte in sortierten Arrays finden kann. Bei ihr wird der Suchraum, also der Teil des Arrays, den wir noch durchsuchen müssen, in jedem Schritt halbiert. Dafür betrachtet man immer den Eintrag in der Mitte des Suchraums und vergleicht ihn mit dem gesuchten Wert. Wenn etwas Größeres gesucht wird als der Wert in der Mitte, muss es offenbar in der weiter hinten liegenden Hälfte des Suchraums stecken, ansonsten

in der vorderen Hälfte.



Die Binärsuche ist eine Anwendung des Prinzips »Teilen und Herrschen« und wird in Kapitel 12 behandelt.

```
BinarySearch(a, x)                 a: sortiertes Array mit n Elementen, x gesuchter Wert
    l ← 1                          l: linker Rand des Suchraums
    r ← n                          r: rechter Rand des Suchraums
    while l ≤ r                    solange der Suchraum noch nicht leer ist
        m ← (l+r)/2, abgerundet    m: Mitte zwischen l und r
        if x = a[m]                wenn x gefunden wurde, ...
            return m               ... fertig!
        if x < a[m]                wenn x in linker Hälfte des Suchraums liegt, ...
            r ← m − 1              ... Suchraum auf linke Hälfte eingrenzen
        else                       ansonsten liegt x in rechter Hälfte des Suchraums
            l ← m + 1              Suchraum auf rechte Hälfte eingrenzen
    return "x ist nicht in a enthalten"
```

Der Algorithmus hat bereits eine prima Laufzeit, doch das ist kein Grund, ihn nicht verbessern zu wollen. Zum Beispiel mit einer Heuristik. Stellen Sie sich vor, Sie möchten in einem Tierlexikon den Begriffen »Wildschwein« nachschlagen. Die Einträge in einem Lexikon sind sortiert, also bietet sich eine Binärsuche an. Doch kaum jemand würde in einem solchen Fall das Lexikon in der Mitte aufschlagen. Vielleicht wenn Sie nach »Maus« suchen, aber bei »Wildschwein« würden Sie es sicher erst einmal weiter hinten im Lexikon versuchen. Dabei wenden Sie eine Faustregel an, die etwa so lauten könnte: »Wörter, deren Anfangsbuchstaben im Alphabet weit hinten stehen, findet man am ehesten weit hinten im Lexikon«. Das ist sehr vage und kann natürlich auch falsch sein, beispielsweise wenn Sie sich als Lexikon ausgerechnet »Brumms Tierleben in vier Bänden, Band 4: Wildschaf bis Zwergziege« ausgesucht haben, denn Wildschweine sollten in diesem Buch ziemlich weit vorne zu finden sein. Wir sollten uns bei unserer Suche also auch an den beiden Grenzen des Suchraums orientieren.

Lassen Sie uns diese Idee genauer ausformulieren. Gegeben sei ein Array a, in dem n Werte gespeichert sind. Die Werte im Array sind aufsteigend sortiert. Unser Suchraum ist der Teil des Arrays, der zwischen den zwei Positionen l und r (mit $l \geq r$) im Array liegt. Da die Werte sortiert sind, ist $a[l]$ die kleinste Zahl im Suchraum und $a[r]$ die größte. Alle Werte, die im Array an Positionen zwischen l und r stehen, haben Werte zwischen $a[l]$ und $a[r]$. Unsere Heuristik geht nun davon aus, dass diese Werte etwa gleichverteilt zwischen den Grenzen $a[l]$ und $a[r]$ liegen. In diesem Fall liegen die Punkte der einzelnen Werte in Abbildung 14.1 etwa auf der *Interpolationsgeraden* zwischen den beiden Grenzen (links in der Abbildung). Die von uns geschätzte Position m, an der wir als Nächstes nach x suchen, sollte so zwischen dem linken Rand l und

dem rechten Rand r des Suchraums platziert, wie der gesuchte Wert x zwischen dem minimalen Wert $a[l]$ und dem maximalen Wert $a[r]$ liegt. Geometrisch betrachtet heißt das: Der Wert x wird an der Interpolationsgeraden auf die Position m »gespiegelt«, genauer gesagt: Wenn man von x aus waagerecht bis zur Interpolationsgeraden und von dort senkrecht nach unten bis auf die x-Achse läuft, dann trifft man auf m. Dies entspricht der folgenden Gleichung:

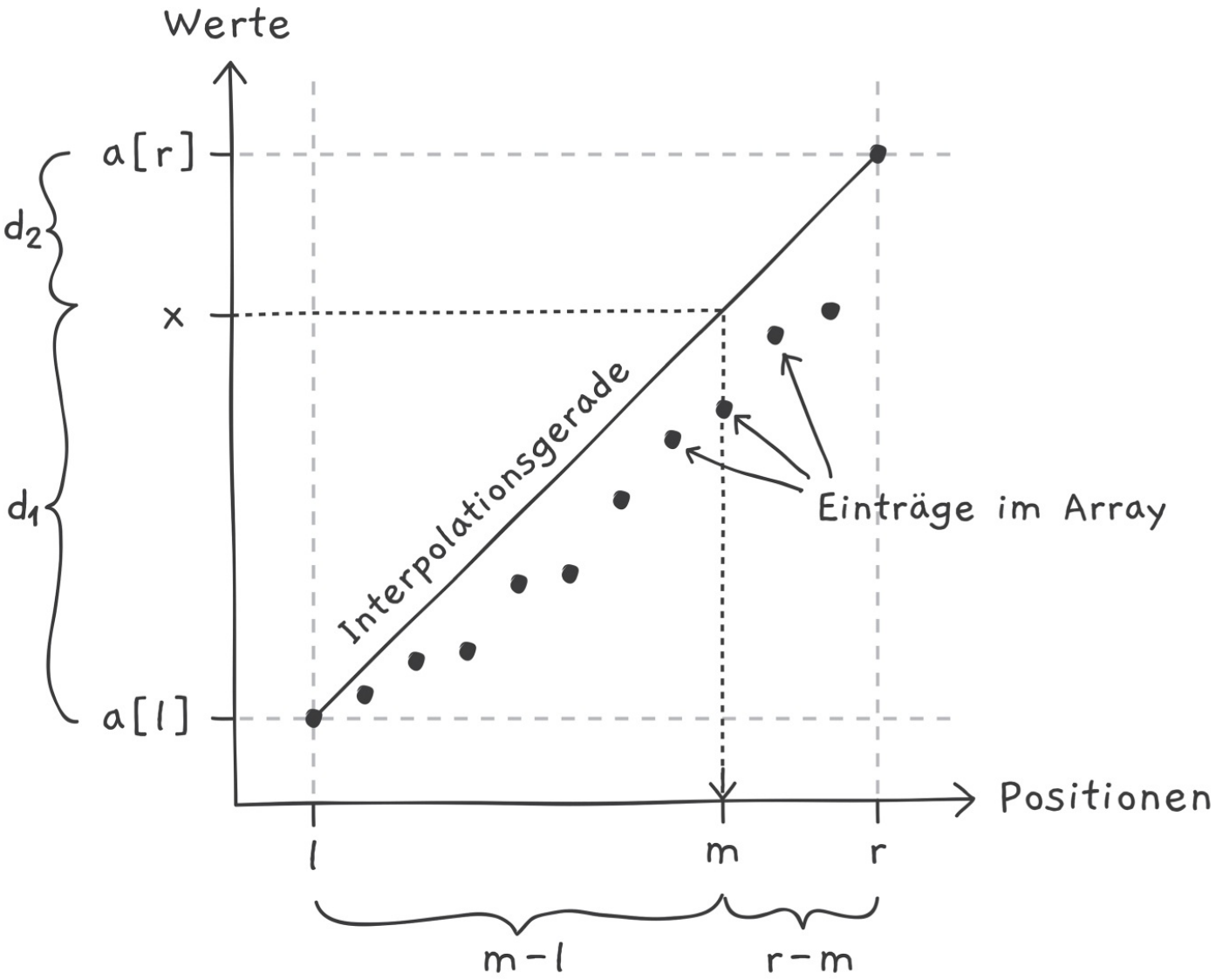


Abbildung 14.1: Interpolationssuche: Der Wert x wird als Nächstes an Position m gesucht.

$$\frac{d_1}{d_2} = \frac{m-l}{r-m}$$

wobei $d_1 = x - a[l]$ und $d_2 = a[r] - x$. Löst man diese Gleichung nach m auf, so ergibt sich

$$m = \frac{d_1 \cdot r + d_2 \cdot l}{d_1 + d_2}$$

Da Positionen in einem Array immer ganze Zahlen sind, müssen wir m natürlich noch runden. Wenn wir diese Formel anwenden wollen, muss der Algorithmus außerdem mit ein paar kruden Sonderfällen klarkommen, beispielsweise wenn $a[l] = a[r]$ und damit $d_1 + d_2 = 0$ ist. Am Ende ergibt sich dann der folgende Algorithmus:

```
InterpolationSearch(a, x)          a: sortiertes Array mit n Elementen, x gesuchter Wert
    l ← 1                          l: linker Rand des Suchraums
    r ← n                          r: rechter Rand des Suchraums
    while l ≤ r                    solange der Suchraum noch nicht leer ist
        if x < a[l] or x > a[r]    wenn x zu klein oder zu groß ist, ...
            return "x ist nicht in a enthalten"      ... ist die Suche erfolglos
        if a[l] = a[r]             wenn es nur noch einen Wert im Suchraum gibt ...
            if a[l] = x            ... und das der gesuchte Wert x ist, ...
                return l           ... x gefunden, fertig!
            else                   ansonsten kommt x in a nicht vor ...
                return "x ist nicht in a enthalten"  ... und die Suche endet erfolglos
        else                       ansonsten weitersuchen:
            d1 ← x − a[l]          Interpolationsmethode anwenden, um m zu berechnen
            d2 ← a[r] − x
            m ← (d1·r+d2·l)/(d1+d2), abgerundet
        if x = a[m]                wenn x gefunden wurde, ...
            return m               ... fertig!
        if x < a[m]                wenn x in linker Hälfte des Suchraums liegt, ...
            r ← m − 1              ... Suchraum auf linke Hälfte eingrenzen
        else                       ansonsten liegt x in rechter Hälfte des Suchraums
            l ← m + 1              Suchraum auf rechte Hälfte eingrenzen
    return "x ist nicht in a enthalten"
```

Liegen die schwarzen Punkten in Abbildung 14.1 genau auf der Interpolationsgeraden, sind die im Array gespeicherten Werte also gleichmäßig auf den Wertebereich verteilt, dann funktioniert der Algorithmus perfekt und man trifft schon beim ersten Versuch genau auf die richtige Position m, an der x in a gespeichert ist, vorausgesetzt, x ist auch in a gespeichert. Sind die Werte im Array hingegen extrem ungleich auf dem Wertebereich verteilt, so kann es passieren, dass InterpolationSearch wiederholt an der völlig falschen Stelle sucht. Dadurch verschlechtert sich die *Worst-Case*-Laufzeit: Der Algorithmus läuft dann nicht mehr in $\mathcal{O}(\log n)$, sondern nur noch in $\mathcal{O}(n)$.

Beim Einsatz von Heuristiken geht man also oft das Risiko ein, dass sich in manchen Fällen die Laufzeit auch verschlechtern könnte. Bei der hier vorgestellten

Interpolationssuche wäre die asymptotische Laufzeitschranke noch zu retten, indem man zum Beispiel Suchpositionen m verbietet, die zu nah an den Suchraumrändern l und r liegen. Mit diesem Trick ließe sich dann wieder eine Laufzeit von $\mathcal{O}(\log n)$ erreichen. Im Vergleich mit der normalen Binärsuche ist der Einsatz der Heuristik jedoch stets mit zusätzlichem Aufwand verbunden, nicht zuletzt durch die relativ aufwendige Berechnung der Suchposition m. Ob sich das in der Anwendung tatsächlich lohnt, lässt sich durch theoretische Überlegungen allein schlecht sagen; man müsste es einfach mal in der Praxis ausprobieren.

Heuristisches Verzweigen und Begrenzen

Wenn man das gesuchte Objekt auch stets sofort als solches erkennt, sobald man es gefunden hat, dann ist klar, dass jeder flüchtige Hinweis auf den Aufenthaltsort die Suche beschleunigen könnte. Es gibt aber auch Aufgaben, bei denen man kaum umhin zu kommen scheint, alle überhaupt möglichen Orte zu durchsuchen. Aufgaben also, die eine *erschöpfende Suche* erfordern. Wenn Sie zum Beispiel aus einem Irrgarten entkommen wollen, ist jede Heuristik willkommen, die ihnen dabei helfen kann. Möchten Sie hingegen den *kürzesten* Ausweg finden, spielt es eigentlich kaum eine Rolle, ob Sie früher oder später das erste Mal aus dem Labyrinth rauskommen, denn bevor Sie fertig sind, müssen Sie ja ohnehin noch den Rest des Labyrinths nach eventuell kürzeren Wegen durchforsten.

Dennoch können auch in einem solchen Fall Heuristiken hilfreich sein, nämlich dann, wenn Sie das Prinzip »Verzweigen und Begrenzen« (Englisch: *branch and bound*) verwenden. Dieses Prinzip beruht auf der Idee, dass man sich die Suche in einem bislang unbekannten Bereich sparen kann, wenn man bereits weiß, dass dort nichts Interessantes zu holen ist. Wenn wir beispielsweise wissen, dass es im Labyrinth einen 200 Meter langen Weg vom Startpunkt bis zum Ausgang gibt, brauchen wir uns um alle Teile des Labyrinths, die weiter als 200 Meter vom Ausgang entfernt liegen, nicht mehr zu kümmern. Jeder Ausweg, der durch eines dieser weit vom Ausgang entfernten Gebiete verläuft, muss länger als 200 Meter sein (siehe Abbildung 14.2). Jeder Weg aus dem Labyrinth gibt uns eine obere Schranke für die gesuchte minimale Weglänge und je kleiner diese obere Schranke ist, desto umfangreichere Teile des Labyrinths darf ich ignorieren und desto schneller läuft der Algorithmus. Eine Heuristik, die uns hilft, möglichst schnell einen möglichst kurzen Weg aus dem Labyrinth zu entdecken, wäre also auch bei diesem Optimierungsproblem hochgradig nützlich, ganz gleich ob es sich bei dieser Heuristik um Wegweiser oder einen Ariadnefaden handelt oder ob man sich schlicht an die Faustregel »bevorzuge Gänge, die etwa in die Richtung des Ziels weisen« hält.

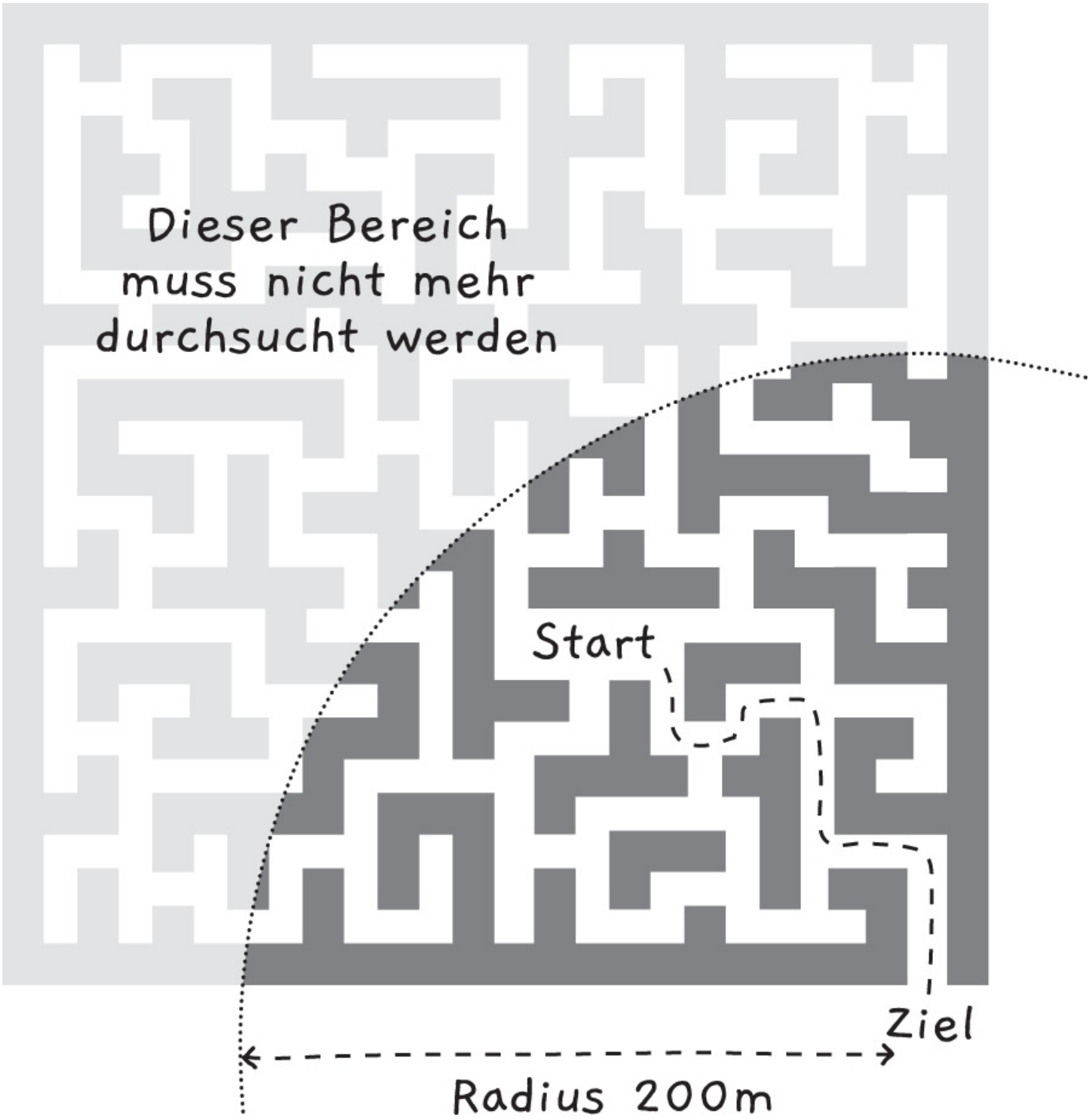


Abbildung 14.2: Kürzester Weg aus einem Labyrinth: Die Länge eines bereits entdeckten Weges beschränkt den Suchraum.

Als weiteres Beispiel für den nützlichen Einsatz von Heuristiken in Branch-and-Bound-Algorithmen wäre das Rucksackproblem zu nennen (siehe den Abschnitt *Den Suchraum begrenzen* im Kapitel 8). Beim Rucksackproblem muss der Algorithmus einen Rucksack von begrenzter Tragkraft mit möglichst wertvollen Objekten füllen. Bei dem von uns vorgestellten Ansatz würde es sich anbieten, besonders attraktive Objekte möglichst weit oben im Suchbaum anzusiedeln. Auswahlkombinationen, die diese attraktiven Objekte beinhalten, würden dadurch früher in Erwägung gezogen, und da es mit attraktiven

Objekte auch tendenziell einfacher sein sollte, eine Rucksackbeladung mit hohem Gesamtwert zu erreichen, sollte man auf diese Weise schneller zu guten unteren Schranken kommen, mit denen sich der Suchraum einschränken lassen kann.



Die algorithmischen Methoden »erschöpfende Suche« und »Verzweigen und Begrenzen« werden ausführlich in Kapitel 11 behandelt. Das Rucksackproblem als weiteres Beispiel für »Verzweigen und Begrenzen« finden Sie in Kapitel 8.

Der A-Algorithmus*

In Kapitel 10 haben Sie Dijkstras Algorithmus kennengelernt, der eine sehr geschickte Methode darstellt, kürzeste Wege in Graphen zu finden. Die Kanten des Graphen sind dabei mit Weglängen ≥ 0 beschriftet. Von einem gegebenen Startknoten s aus traversiert man den Graphen, entdeckt und untersucht nach und nach alle von s aus erreichbaren Knoten und berechnet dabei die Länge der kürzesten Wege von s zu jedem untersuchten Knoten. Sobald wir die Länge des kürzesten Pfades zum gegebenen Zielknoten z bestimmt haben, brechen wir die Suche ab. Der Algorithmus ähnelt dabei einer Breitensuche, nur mit dem Unterschied, dass die Knoten nicht einfach nach dem FIFO-Prinzip (*first in first out*) in einer Warteschlange gespeichert und wieder daraus entfernt werden, sondern immer derjenige Knoten als nächster an die Reihe kommt, der von allen am nächsten am Startknoten dran liegt.



Dijkstras Algorithmus wird ausführlich in Kapitel 10 vorgestellt. Der besseren Übersichtlichkeit halber haben wir den Pseudocode hier etwas knapper dargestellt.

```
Dijkstra(s, z)                    s: Startknoten, z Zielknoten
    D[s] ← 0                      D: Map, speichert minimale Weglängen
    T ← {s}                       T: Menge der Knoten, die noch untersucht werden müssen
    while T ≠ ∅                   solange noch Knoten nicht untersucht sind:
        v ← ein Knoten in T mit minimalem D[v]
        if v = z                  Ziel erreicht:
            return D[z]           Länge des kürzesten Weges von s nach z zurückgeben
        untersuche v              alle von v aus erreichbaren Knoten w durchlaufen und ...
                                  ... dabei eventuell D[w] berechnen und w in T eintragen
        entferne v aus T          v ist fertig untersucht und kann raus aus T
```

Der Algorithmus verwendet eine Map D, um die Längen der kürzesten bislang gefundenen Wege vom Startpunkt s zu den bislang entdeckten Knoten v zu speichern. T ist die Menge der bereits entdeckten, aber noch nicht untersuchten Knoten, also sozusagen unsere »To-do-Liste«.

Heuristische Knotenwahl

Raum für Verbesserungen des Algorithmus bietet sich bei der Auswahl des nächsten

Knotens v, den wir untersuchen wollen. Wüssten wir bereits, wo der kürzeste Weg vom Startpunkt s zum Ziel z durch den Graphen verläuft, dann könnten wir einfach nacheinander die Knoten auf diesem Pfad untersuchen und könnten uns die Untersuchung aller restlichen Knoten sparen. Stattdessen wählt Dijkstras Algorithmus stur immer den Knoten aus, der am wenigsten weit von s entfernt ist. Das kann man mit einem Wasserhahn vergleichen, den man am Startpunkt s aufdreht (siehe Abbildung 14.3). Das Wasser breitet sich in alle Richtungen gleichmäßig aus und nach und nach werden alle Knoten nass, und zwar in der Reihenfolge ihres Abstands zu s. Dem Algorithmus ist es dabei völlig gleich, in welche Richtung das Wasser fließt – er flutet den Graphen einfach gleichmäßig in alle Richtungen. Da wäre es doch schlauer, wenn wir den Graphen sozusagen ein wenig auf das Ziel hin neigen könnten, damit das Wasser eher dorthin fließt als von dort weg. Das erreichen wir dadurch, dass wir bei der Auswahl des nächsten Knotens v aus T nicht mehr nur darauf achten, wie weit er vom Startpunkt entfernt liegt, sondern auch, wie weit der Rest des Weges bis zum Ziel sein wird. Das Problem ist nur, dass wir nicht genau wissen, wie weit es genau von v bis zum Zielknoten z ist. Dafür muss dann natürlich eine Heuristik herhalten, genauer: eine Funktion **rest** (v, z), welche die die Länge des Weges von v nach z abschätzt. Wir müssen also nur eine Zeile im Algorithmus ändern und schon erhalten wir den sogenannten A*-Algorithmus (ausgesprochen »A Stern« oder auch auf Englisch »*a star*«):

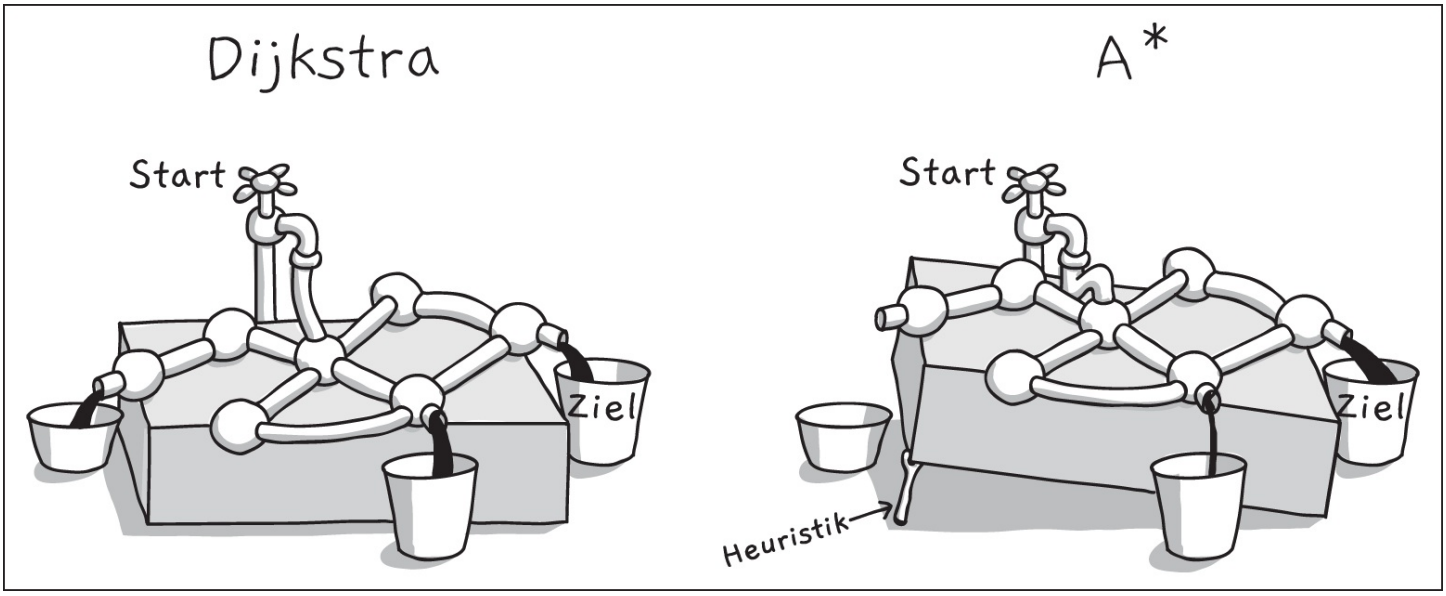


Abbildung 14.3: Beim A*-Algorithmus fließt das Wasser eher in Richtung Ziel.

```
Astar(s, z)                                          s: Startknoten, z Zielknoten
    D[s] ← 0                                         D: Map, speichert minimale Weglängen
    T ← {s}                                          T: Menge der Knoten, die noch untersucht werden müssen
    while T ≠ ∅                                      solange noch Knoten nicht untersucht sind:
        v ← ein Knoten in T mit minimalem D[v]+rest(v, z)   ⇐ hier Heuristik!
        if v = z                                     Ziel erreicht:
            return D[z]                              Länge des kürzesten Weges von s und z zurückgeben
        untersuche v                                 alle von v aus erreichbaren Knoten w durchlaufen und ...
                                                     ... dabei eventuell D[w] berechnen und w in T eintragen
        entferne v aus T                             v ist fertig untersucht und kann raus aus T
```

Aber wie genau soll man die Länge des Weges zwischen zwei Knoten schätzen? Wären die Knoten im Graphen in Wirklichkeit Orte auf einer Landkarte oder, allgemeiner ausgedrückt, Punkte auf einer Fläche oder im Raum, so kann man als Schätzung einfach die *euklidische Distanz* zwischen den Punkten verwenden. Oder, auf Deutsch gesagt, die »Luftlinie«.



Schätzung des Restwegs

Je nach Problemstellung könnte man die Schätzfunktion **rest** auch ganz anders definieren. Allerdings muss man dabei etwas aufpassen, damit der Algorithmus auch korrekt bleibt. Er muss nämlich dafür sorgen, dass nur solche Knoten v zur Untersuchung ausgewählt werden, zu denen bereits der kürzeste Weg gefunden worden ist, bei denen also $D[v]$ bereits seinen korrekten Wert hat. Nur deswegen dürfen wir den Algorithmus abbrechen, sobald wir den Zielknoten z aus T gewählt haben. Außerdem wird dadurch sichergestellt, dass ein einmal untersuchter und aus T entfernter Knoten v niemals wieder in T aufgenommen wird, der Algorithmus also »gierig« bleibt, mit allen Vorteilen, die Gier für die Laufzeit mit sich bringt. (Noch mehr zum Thema »Gier« erfahren Sie weiter hinten in diesem Kapitel.)

Angenommen, auf dem kürzesten Weg von s zum Knoten v liegt vor v ein Knoten u (siehe Abbildung 14.4). Dann müssen wir dafür sorgen, dass v erst untersucht wird, wenn u bereits untersucht und dabei der kürzeste Weg nach v gefunden worden ist, das heißt, es muss gelten:

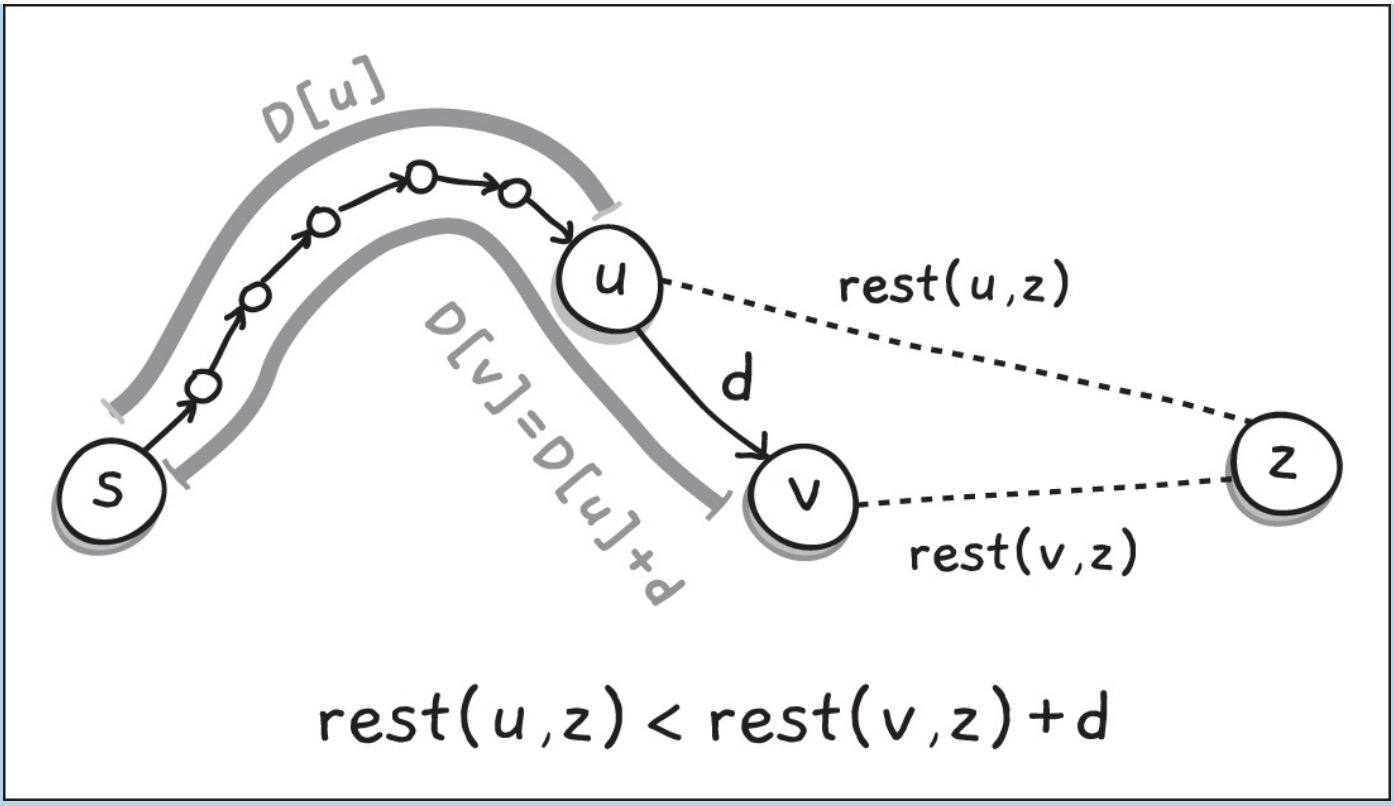


Abbildung 14.4: Dreiecksungleichung für die Schätzfunktion rest

$$D[u] + \text{rest}(u, z) \leq D[v] + \text{rest}(v, z).$$

Die Länge der Kante von u nach v wollen wir d nennen. Da der kürzeste Weg von s nach v über u führt, ist

$$D[v] = D[u] + d.$$

Setzen wir beides ineinander ein, so hebt sich D weg und wir erhalten

$$\text{rest}(u, z) \leq \text{rest}(v, z) + d.$$

Der kürzeste Weg von u nach z sollte also keinesfalls als länger eingeschätzt werden, als man bräuchte, wenn man zunächst von u nach v läuft und von dort aus weiter nach z wandert. Wählt man für **rest** wie erwähnt die euklidische Distanz $dist$, so folgt das aus der sogenannten *Dreiecksungleichung*, nach der für beliebige Punkte a, b und c gilt, dass $dist(a, b) \leq dist(a, c) + dist(c, b)$.

Approximation

Es gibt Probleme, deren Lösung wirklich sehr anstrengend ist, bei denen es kaum besser geht als mit einer erschöpfenden Suche. Ob mit oder ohne Backtracking, die Laufzeit ist und bleibt exponentiell. Zu diesen Problemen gehören zum Beispiel die NP-schweren Probleme, über die wir schon in Kapitel 11 gesprochen haben. Bei solchen Problemen ist eine Lösung oft nur für kleine Probleminstanzen in realistischer Zeit machbar.

Im wirklichen Leben stellen sich oft solche und noch schwierigere Probleme, aber wir kommen mit ihnen trotzdem meist ganz gut zurecht. Man nimmt es einfach nicht so

genau. Eine vernünftige Lösung mit überschaubarem Aufwand zu erreichen, ist in der Regel besser, als mit unvernünftigem Aufwand der perfekten Lösung nachzujagen. Das gilt beim Hausputz, wo wir mal fünf gerade sein lassen, ebenso wie bei der Suche nach der Urlaubswohnung mit dem besten Preis-Leistungs-Verhältnis. Statt auf der richtigen Lösung für ein Problem zu bestehen, suchen wir nach einer Annäherung, oder wie man auch sagt, einer *Approximation*. Die Approximation wird mit einem Approximationsalgorithmus berechnet, der eventuell komplett anders arbeitet als der »richtige« Algorithmus und eine nicht perfekte Lösung berechnet – das aber in annehmbarer Zeit.


Mehr über NP und NP-schwere Probleme erfahren Sie in Kapitel 8.

TSP: Die kürzeste Rundreise

Ein Geschäftsmann möchte eine Reihe von Städten besuchen, um dort seine Ware zu verkaufen. Dafür plant er eine große Rundreise (eine »Tour«), in der er ausgehend von seiner Heimatstadt nacheinander in jede der anderen Städte fährt und am Ende wieder in seine Heimatstadt zurückkehrt (siehe Abbildung 14.5). Um Zeit und Geld zu sparen, soll die bei der Rundreise zurückgelegte Strecke so kurz wie möglich sein. In welcher Reihenfolge sollte er die Städte ansteuern, um die Gesamtweglänge zu minimieren?

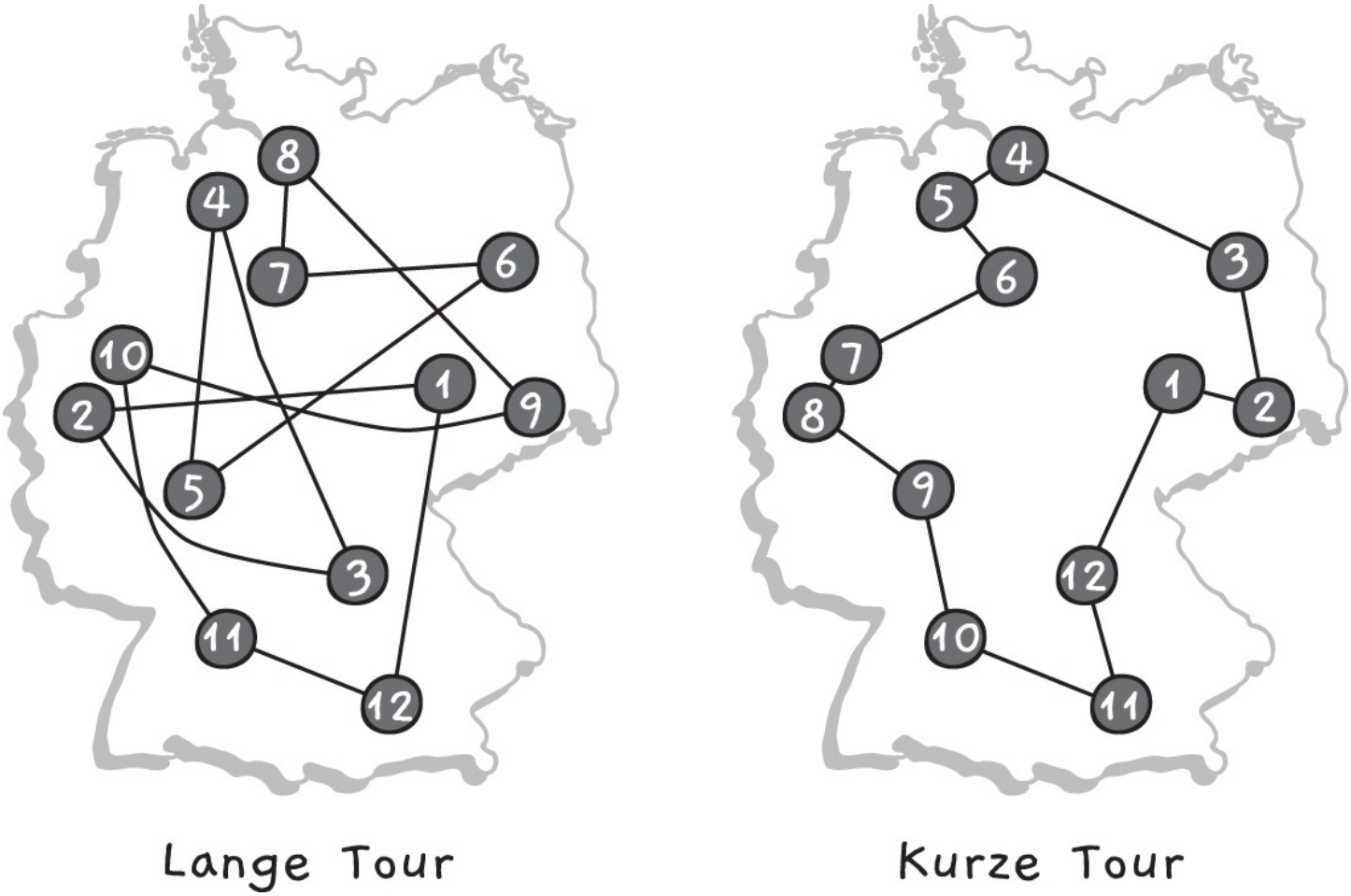


Abbildung 14.5: Zwei unterschiedliche lange Rundreisen zu 12 deutschen Städten

Dieses Problem heißt »Problem des Handlungsreisenden« oder auf Englisch »*traveling salesman problem*« (TSP). Genauer können wir das Problem so formulieren: Es sollen n Städte in einer Rundreise besucht werden. Lassen sie uns diese Städte als n Punkte auf der Landkarte auffassen. Von jedem dieser Städte kann man direkt zu jeder anderen Stadt gelangen, und zwar in gerader Linie. Unser Geschäftsmann reist mit dem Hubschrauber, muss also keinen kurvigen Straßen folgen. Der Weg, den er bei der Reise von einer Stadt zu nächsten Stadt zurücklegen muss, ist also der Abstand (vornehm ausgedrückt: die »euklidische Distanz«) zwischen den beiden Punkten auf der Landkarte. Diese Variante des TSP nennt man darum auch »euklidisches TSP«.

Das TSP ist vermutlich das berühmteste NP-vollständige Problem überhaupt. Alle bislang bekannten Algorithmen zur exakten Lösung des TSP, die also garantiert die kürzeste Tour berechnen, haben Laufzeiten, die exponentiell mit der Zahl n der Städte ansteigt. Der Weltrekord für das größte tatsächlich exakt gelöste TSP liegt bei unter einhunderttausend Städten. Für reale Geschäftsreisen von realen Managern wäre die Lösung des Problems also durchaus noch machbar. Es gibt jedoch zahlreiche in der Praxis relevante Probleme, die sich als ein TSP formulieren lassen und dann teilweise mit Millionen von Städten arbeiten. Statt einer exakten Lösung würde man es dann mit einer möglichst guten Annäherung an das Optimum versuchen. Wir wollen uns jetzt ein paar Methoden anschauen, die man dazu verwenden kann.

Gierige Heuristik

Eine einfache Methode, eine einigermaßen kurze Rundreise durch n Städte zu planen, funktioniert so (siehe Abbildung 14.6): Zunächst suchen wir uns zufällig eine der Städte als Startpunkt für unsere Rundreise aus. Dann wählen wir aus der Menge S der bislang noch nicht besuchten Städte das nächste Ziel aus, und zwar diejenige Stadt, die am *nächsten* an der zuletzt besuchten Stadt liegt. Dies wird so lange wiederholt, bis am Ende alle Städte besucht worden sind und man zurück zum Startpunkt reist:

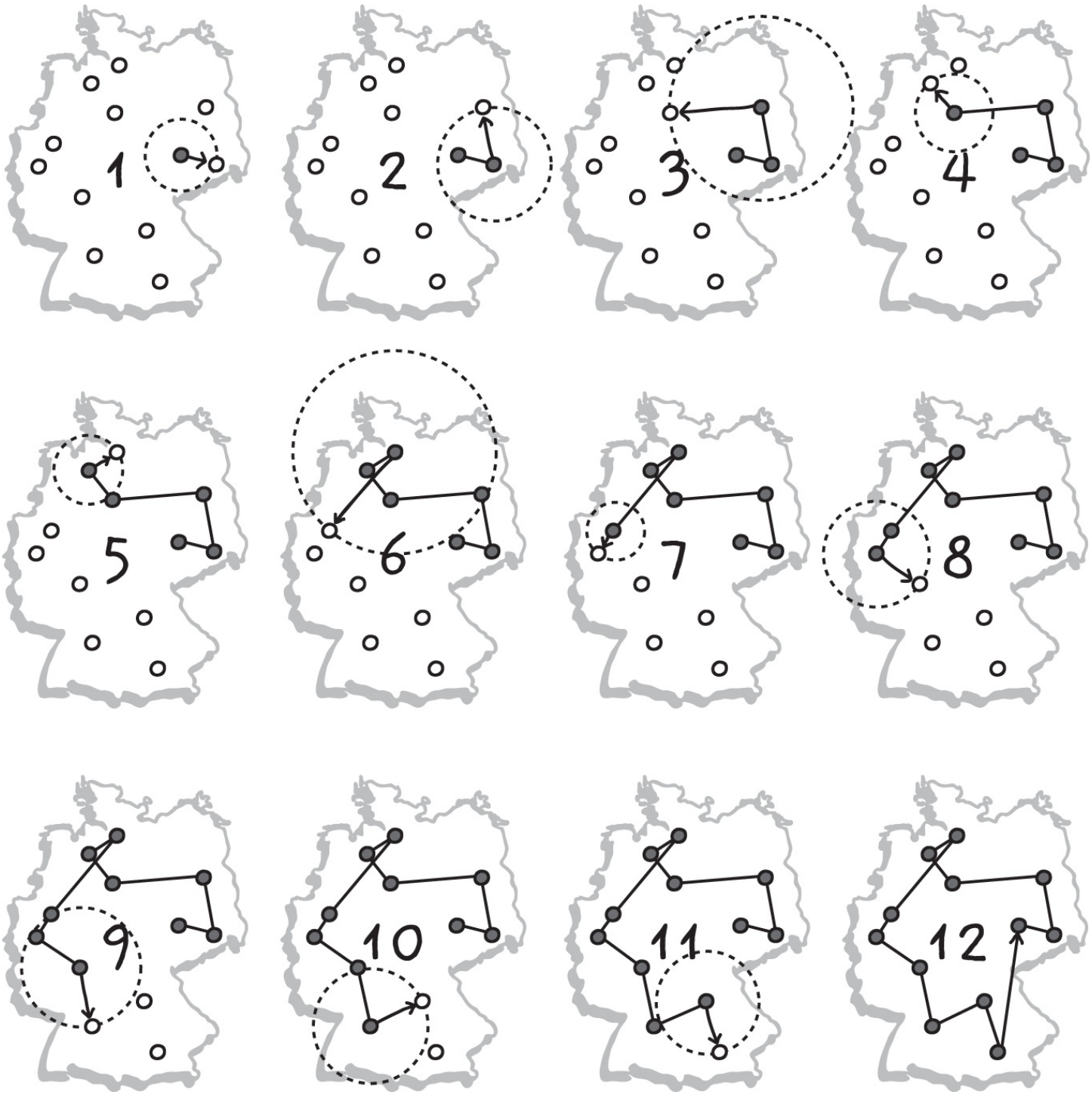


Abbildung 14.6: Näherungsweise Lösung des TSP mit einem Greedy-Algorithmus

```
TSPGreedy(S)                          S: Menge von n Städten
    a ← zufällig gewählte Stadt aus S     a: Stadt, in der man sich gerade befindet
    S.remove(a)                       Startpunkt aus S entfernen
    L ← [a]                           L: Liste speichert die Reihenfolge der Städte der Rundreise
    while S ≠ ∅                       solange noch nicht alle Städte besucht worden sind:
        b ← Stadt in S mit minimalem Abstand zu a     nächste Stadt auswählen
        S.remove(b)                   ausgewählte Stadt aus S entfernen, ...
        L.append(b)                   ... zur Rundreise hinzufügen ...
        a ← b                         ... und zu ihr hinreisen
    return L
```

Mit dieser Methode wird Schritt für Schritt eine Näherungslösung für das TSP konstruiert. Dabei wird eine Heuristik benutzt, denn in jedem Schritt wird diejenige Entscheidung getroffen, die im Augenblick gerade am attraktivsten erscheint, nämlich die am nächsten gelegene Stadt angesteuert. Eine solche Strategie nennt man »gierig«, auf Englisch »*greedy*«. Mit dem algorithmischen Prinzip »Gier« werden wir uns in einem späteren Abschnitt dieses Kapitels noch einmal eingehender beschäftigen. Ein Greedy-Algorithmus entscheidet sich immer für die gerade lokal beste Option und er nimmt eine einmal getroffene Entscheidung später auch nicht zurück, selbst wenn das vielleicht für das Gesamtergebnis besser wäre. Was der gierige Algorithmus einmal hat, das gibt er nicht wieder her. Durch ihre Sturheit können gierige Algorithmen natürlich viel Zeit sparen. Wenn man die Wahl der nächsten Stadt b im Algorithmus einfach durch eine lineare Suche realisiert, hat TSPGreedy eine Laufzeit von $\mathcal{O}(n^2)$.

Lokale Suche

Bei der eben vorgestellten Methode wird mithilfe einer Heuristik Schritt für Schritt eine möglichst gute Näherungslösung konstruiert (*Konstruktionsheuristik*). Eine Alternative dazu wäre, zunächst mit irgendeiner, vielleicht auch schlechten Näherungslösung anzufangen, und dieses dann schrittweise zu verbessern (*Verbesserungsheuristik*). Damit die Suche nach Verbesserungen nicht zu aufwendig wird, zieht man dabei nur kleinere Änderungen der bisherigen Tour in Betracht. Aus diesem Grund heißt die Methode auch *lokale Suche*, denn die Suche nach besseren Lösungen beschränkt sich auf die unmittelbare Nachbarschaft der bislang besten bekannten Lösung des Problems. Im Fall des TSP könnte eine »kleine Änderung« zum Beispiel darin bestehen, dass zwei Städte in der Rundreise ihre Plätze tauschen. Findet man zwei Städte, durch deren Vertauschung sich die Gesamtlänge der Rundreise verkleinert, so vertauscht man die beiden Städte eben und hat damit seine Näherungslösung verbessert. Dies wiederholt man so lange, bis sich keine weitere Verbesserung mehr finden lässt. Am Ende erhält man dann eine Rundreise, deren Länge minimal im Vergleich zu allen anderen Rundreisen in ihrer Nachbarschaft ist – ein sogenanntes *lokales Maximum* (siehe Abbildung 14.7).

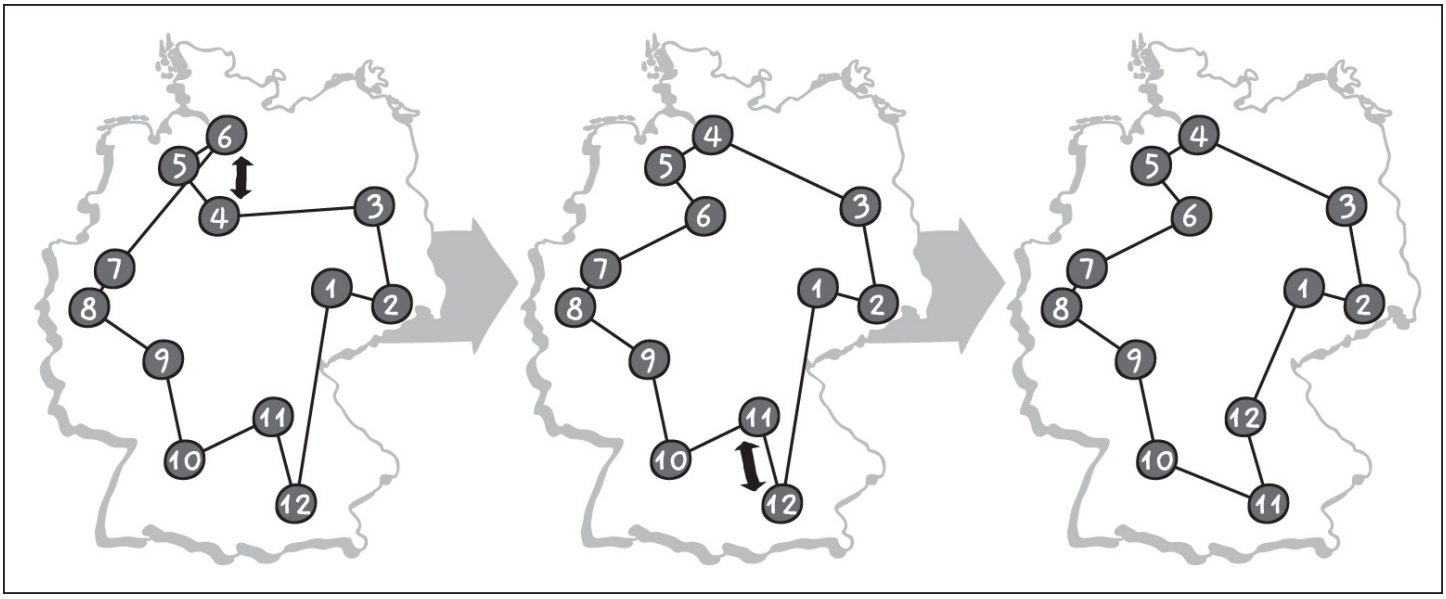


Abbildung 14.7: Schrittweise Verbesserung einer Rundreise durch lokale Suche

```
TSPLocalSearch(S)                    S: Menge von n Städten
    L ← [ ]                          L: Rundreise durch n Städte
    for a ← S                        alle Städte aus der Menge S ...
        L.append(a)                  ... werden zur Rundreise hinzugefügt
    while true                       Folgendes wiederholen:
        L´ ← TSPImproveTour(L)       bessere Rundreise L suchen
        if L = L´                    wenn es keine bessere gibt, ...
            return L                 ... fertig!
        L ← L´                       ansonsten Rundreise weiter optimieren

TSPImproveTour(L)                    L: Reihenfolge von n Städten einer Rundreise
    s_L ← Weglänge von L
    for i ← 1, ..., n − 1            alle Paare (i, j) ...
        for j ← i + 1, ..., n        ... von Knoten aufzählen
            L´ ← Kopie von L         Weglänge der Rundreise berechnen ...
            vertausche L´[i] mit L´[j]   ... nach Vertauschen von i mit j
            s´_L ← Weglänge von L´
            if s´_L < s_L            wenn kürzere Rundreise gefunden worden ist, ...
                return L´            ... diese zurückgeben
    return L                         ansonsten ist L lokal optimal
```

Auch die lokale Suche ist nur eine Approximationsmethode. Zwar erreicht man damit ein lokales Maximum, aber im nicht unbedingt ein globales Maximum.

Approximation ohne Heuristik

Bisher haben wir nur heuristische Methoden zur näherungsweise Lösung von Problemen betrachtet. Doch nicht jede Approximationsmethode ist eine Heuristik. Die Mathematik

ist voll mit Abschätzungen, für die man keine groben »Faustregeln« braucht, sondern die Näherungswerte einfach ausrechnet. Viele Approximationsalgorithmen dieser Art können sogar verbindliche Aussagen darüber machen, wie nah sie dem Optimum kommen.

Approximation der Kreiszahl π

Für die Berechnung der Kreiszahl π (sprich: »pi«), die das Verhältnis von Umfang zu Durchmesser eines Kreises angibt, gibt es keinen exakten Algorithmus, denn π ist eine *irrationale* Zahl, das heißt, man würde unendlich viel Speicher brauchen, um sie abzuspeichern. Man kann die Zahl jedoch approximieren, beispielsweise durch regelmäßige Polygone, wie in Abbildung 14.8 angedeutet. Die Polygone liefern dabei gleichzeitig eine untere und eine obere Schranke für π: Das kleine Polygon liegt ganz innerhalb des Kreises und der Kreis ganz innerhalb des größeren Polygons. Durch den Vergleich der beiden Schranken kann man abschätzen, wie nah man sich dem tatsächlichen Wert für π angenähert hat. Je mehr Ecken die Polygone haben, desto genauer wird die Approximation, und indem man die Eckenzahl immer weiter erhöht, kann man sich π beliebig nah annähern.

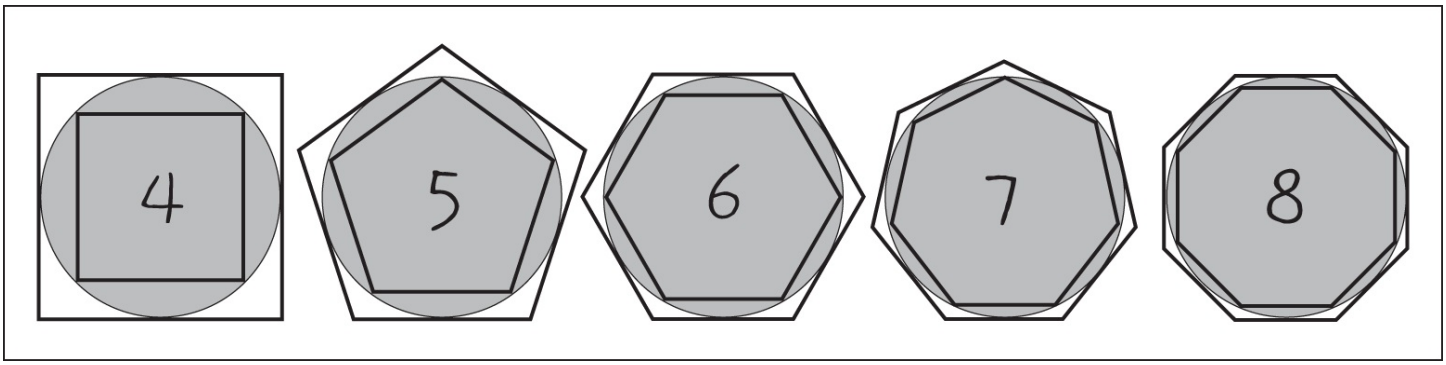


Abbildung 14.8: Approximation der Kreiszahl π mithilfe von Polygonen

Approximationsschemata

Ähnlich wie für π gibt es auch für andere Berechnungsprobleme Methoden, mit denen man sich der exakten Lösung beliebig gut annähern kann. Aber wie misst man eigentlich die Güte einer solchen Annäherung? Lassen Sie uns das für Optimierungsprobleme klären. Optimierungsprobleme versuchen, eine Zielgröße S zu optimieren. Bei einem Maximierungsproblem soll S so groß wie möglich werden, bei einem Minimierungsproblem so klein wie möglich. Das Optimum wollen wir S_{opt} nennen. Bei einem Maximierungsproblem ist also der Wert S einer Näherungslösung kleiner als S_{opt} und bei einem Minimierungsproblem größer als S_{opt}. Das Verhältnis vom Wert S der Approximation zum optimalen Wert S_{opt} kann man als Gütemaß für die Approximation ansehen. Je näher dieses Verhältnis an die Eins herankommt, desto besser ist die Approximation.

Besonders interessant ist es, wenn man eine Möglichkeit findet, wie man zu jeder beliebigen Approximationsgüte $\neq 1$ eine Approximation berechnen kann. Eine solche

Methode nennt man ein *Approximationsschema*. Ein ausführliches Beispiel dafür finden Sie in Kapitel 8 im Abschnitt *Sich dem Optimum annähern*. Dort geht es um das ganzzahlige Rucksackproblem. Genau wie das TSP ist auch das Rucksackproblem NP-schwer, das heißt, alle bislang bekannten exakten Methoden für die Lösung des Rucksackproblems haben eine exponentielle Laufzeit. Das in Kapitel 8 vorgestellte Approximationsschema läuft hingegen in polynomieller Zeit. Das ist beeindruckend und für viele andere NP-schwere Probleme gibt es nachweislich kein derartiges *Polynomialzeit-Approximationsschema*. NP-schwere Probleme lassen sich also unterschiedlich gut approximieren.

Für das euklidische TSP gibt es tatsächlich ebenfalls ein Polynomialzeit-Approximationsschema, doch das ist so umständlich, dass wir hier nicht weiter darüber reden möchten. Stattdessen lernen Sie nun einen Approximationsalgorithmus für das TSP kennen, bei dem die Weglänge S_{approx} der Näherungslösung höchstens doppelt so groß ist wie die Länge S_{opt} der optimalen Rundreise.

Approximation des TSP mit Gütegarantie

Das Problem des Handlungsreisenden kann man sehr leicht auch als Graphenproblem formulieren: Die Knoten im Graphen sind die Städte, jeder Knoten ist mit jedem über eine Kante verbunden und die Länge einer Kanten entspricht dem euklidischen Abstand zwischen den beiden Städten, die diese Kante verbindet. Eine Rundreise wäre dann ein Pfad durch den Graphen, der alle Knoten miteinander verbindet und am Ende wieder zurück zum Ausgangspunkt führt. Gesucht wird beim TSP also ein solcher geschlossener Pfad mit minimaler Gesamtlänge der Kanten.

In Kapitel 10 haben Sie eine Methode kennengelernt, mit der man effizient minimale *Spannbäume* in Graphen berechnen kann, also ein Baum, der alle Knoten im Graphen miteinander verbindet und dabei eine minimale Gesamtlänge der Kanten aufweist. Das TSP ist also dem Problem, minimale Spannbäume zu berechnen, sehr ähnlich. Der einzige Unterschied ist, dass beim einen ein geschlossener Pfad und beim anderen ein Baum gesucht wird.

Wenn wir bei einem geschlossenen Pfad eine der Kanten weglassen, dann ist der Pfad danach zwar nicht mehr geschlossen, verbindet aber immer noch alle Knoten im Graphen miteinander. Ein solcher offener Pfad ist aber auch ein Baum und damit steht schon einmal fest: Die Länge des minimalen Spannbaums ist höchstens so groß wie die Länge der kürzesten Rundreise.

Nun kann man umgekehrt aus einem Spannbaum wieder eine Rundreise machen (siehe Abbildung 14.9): Ausgehend von einem beliebigen Knoten traversiert man den Spannbaum einfach in einer Tiefensuche. Versteht man die Tiefentraversierung als ein Auf- und Ablaufen durch den Baum, so entspricht sie einem Weg durch den Graphen, der die doppelte Länge des Spannbaums besitzt (Abbildung 14.9, Mitte). Wann immer man

bei der Tiefensuche einen neuen Knoten entdeckt, wird dieser zum nächsten Ziel der Rundreise. Dabei kann man ihn natürlich direkt ansteuern, ohne den umständlichen Weg über den Spannbaum zu nehmen (Abbildung 14.9, rechts). Dadurch verkürzt sich die Länge der Rundreise eventuell. Insgesamt ergibt sich daraus die folgende Ungleichung:

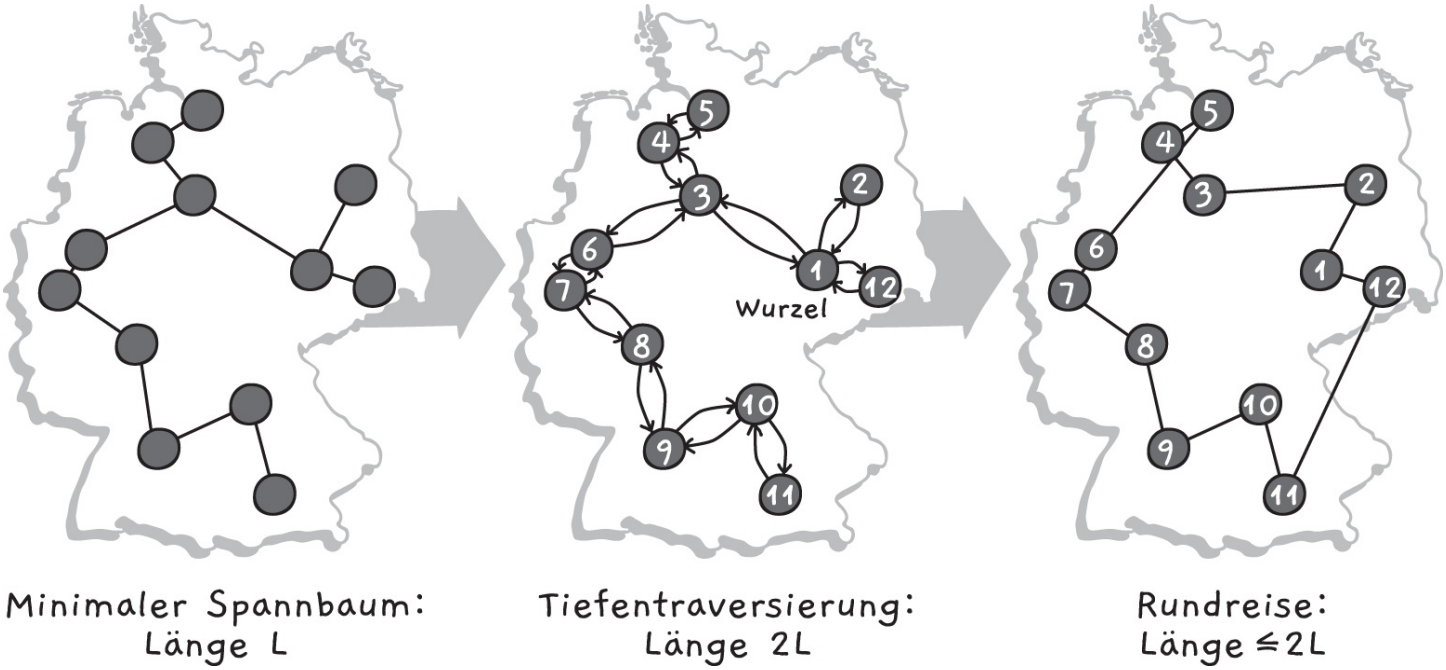


Abbildung 14.9: Approximation eines TSP mit einem minimalen Spannbaum

$$
\begin{aligned}
&\text{Länge der berechneten Rundreise} \\
&\leq \text{Länge der Tiefentraversierung des Spannbaums} \\
&\leq 2 \times \text{Länge des Spannbaums} \\
&\leq 2 \times \text{Länge der kürzesten Rundreise}
\end{aligned}
$$

Die Approximation ist also höchstens doppelt so lang wie die optimale Rundreise.


Minimale Spannbäume werden in Kapitel 10 berechnet, über die Tiefentraversierung von Bäumen erfahren Sie in Kapitel 5 mehr.

Gier

Im lokalen Biomarkt gib es wunderbare nahrhafte und gesunde Rohstoffe, die mit Sorgfalt und Geduld zubereitet eine ideale Kost ergeben. Klar, so ernähren wir uns immer, es sei denn, die Zeit ist gerade mal knapp und der Hunger groß. Dann wird schon mal auf die ideale Lösung zugunsten einer Approximation verzichtet und das, was gerade zu finden ist und nach viel Kalorien und Geschmack aussieht, in sich hineingestopft.

Diese Art der Problemlösung ist eine wichtige algorithmische Technik. Man nennt sie

Gier. Manchmal ist sie ungesund. Aber wir können nicht anders. Mutter Natur hat uns so ausgestattet. In einer unangenehmen oder bedrohlichen Situation geht man den Weg, der am schnellsten weg von unerwünschten hin zum gewünschten Zustand führt. Fehlt uns etwas, dann wird der Mangel mit möglichst großen und fetten Brocken beseitigt. Mit etwas mehr Überblick und dem Gedanken an noch ferne Konsequenzen hätte man sich anders und besser verhalten. Aber hier ist hier und heute ist heute. Was morgen woanders ist, das sehen wir dann und dort.

Das gleiche Prinzip wendet man auch in Situationen an, die nichts mit Essen oder Geld zu tun haben. Wollen wir mal ohne Navigationssystem schnell zu einem bestimmten Ziel kommen, dann nehmen wir einfach immer an jeder Weggabelung den Weg, der möglichst genau in die Himmelsrichtung führt, in der unser Ziel liegt. Auch das ist eine Art von Gier. Was führt jetzt gerade möglichst effektiv weg vom Unerwünschten hin zum Erwünschten? Da wird schon mal ein nicht optimaler Weg gewählt und eventuell landet man sogar in einer Sackgasse, aus der nur mit erheblichem Mehraufwand wieder herauszukommen ist (siehe Abbildung 14.10). Aber im Großen und Ganzen ist Gier eine effektive algorithmische Strategie, die mit vergleichsweise wenig Aufwand zu einem passablen Ergebnis führt.

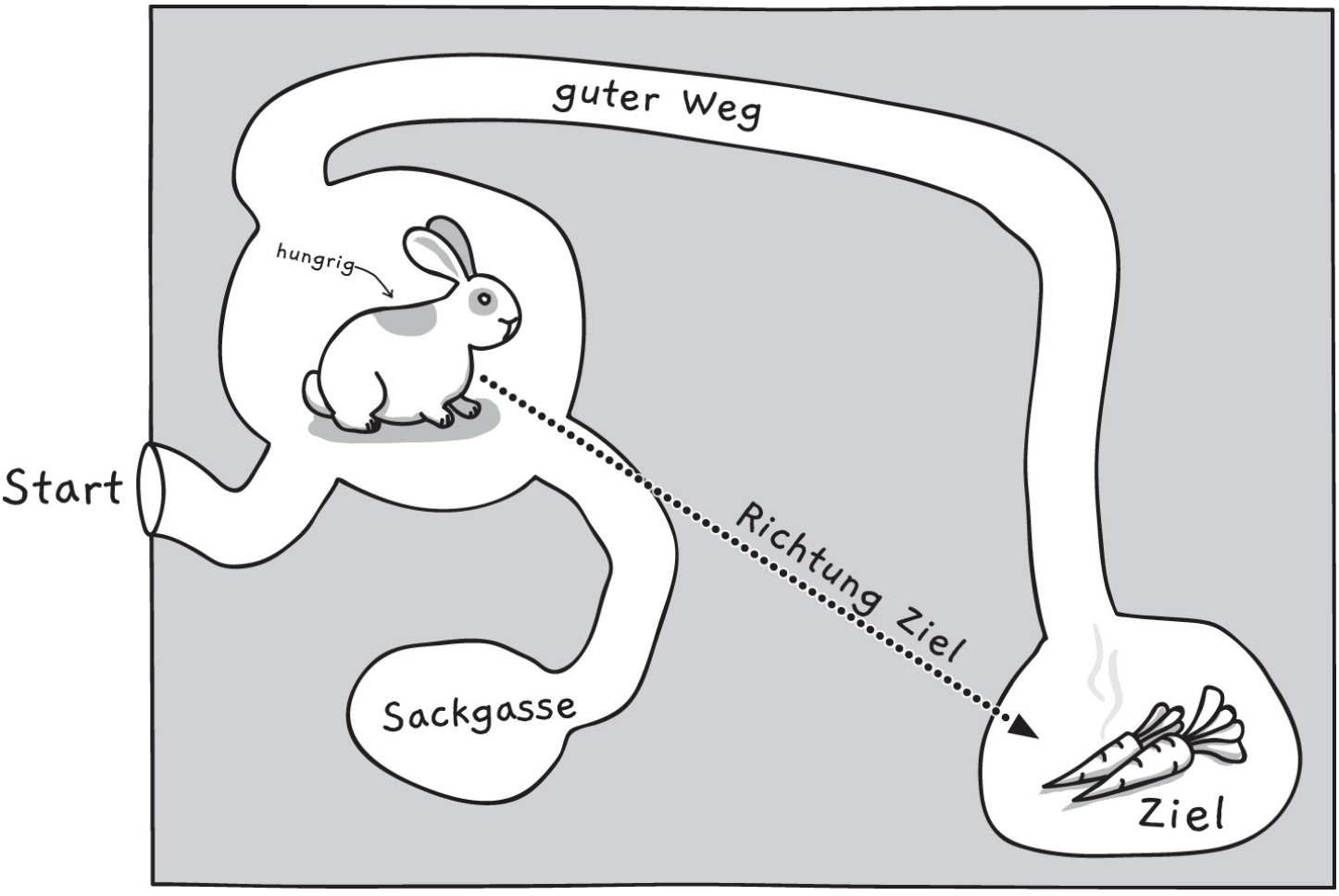


Abbildung 14.10: Wenn Egon gierig in Richtung Ziel läuft, landet er in der Sackgasse.

Bei einem Irrgarten funktioniert Gier nicht besonders gut. Das ist kein Wunder. Irrgärten

werden ja gerade so konstruiert, dass das dieses einfache Verfahren eben nicht zum Ziel führt. Aufs Ganze gesehen ist ein gieriges Vorgehen aber recht erfolgreich. An jedem Entscheidungspunkt entscheidet man auf Basis der aktuellen lokal leicht zugänglichen Informationen, ohne an die globale Situation oder die langfristigen Folgen der Entscheidung zu denken.

Gier: Ich will reich werden. Ich kaufe Bitcoin-Aktien, sofort, ohne weiter nachzudenken! Ich will Möhren. Die Möhren sind unten rechts, also hoppele ich nach unten rechts, sofort, ohne weiter nachzudenken!


Bei der Anwendung der Technik *Gier* wird

- ✔ eine Lösung systematisch Schritt für Schritt aus Teillösungen aufgebaut und
- ✔ bei jedem Schritt die erreichte Teillösung nach lokalen Kriterien optimal verbessert (welche der möglichen nächsten Aktionen bringt uns am dichtesten an die Gesamtlösung?).

Ein typisches Beispiel für einen gierigen Algorithmus haben wir in Kapitel 8 für die näherungsweise Lösung des Rucksackproblems kennengelernt. Beim Rucksackproblem muss man eine Auswahl aus einer Reihe von Objekten treffen, sodass die ausgewählten Objekte alle noch in den Rucksack passen und ihr Gesamtwert maximal ist. Ein einfacher Ansatz besteht darin, die Objekte zunächst absteigend nach ihrer Attraktivität zu sortieren, also beispielsweise nach ihrem Wert, und sie dann in dieser Reihenfolge in den Rucksack zu packen, bis dieser voll ist. Diese Vorgehensweise ist ganz klar gierig: Was wir einmal ausgesucht und in den Rucksack eingepackt haben, geben wir nicht wieder her (die besten Stücken liegen ganz unten!). Der Algorithmus ist geradezu ein Paradebeispiel für eine gierige Heuristik. Wie aus dem Beispiel in Abbildung 8.1 zu erkennen ist, ist die Methode allerdings nicht immer optimal. Das ist typisch für viele gierige Verfahren. Sie werden im Folgenden weitere Beispiele für gierige Approximationsverfahren kennenlernen.

Das Wechselgeldproblem

Das *Wechselgeldproblem* ist ein Problem, das Kassierer ständig lösen müssen.

Gegeben ist das Wechselgeld m (eine ganze Zahl), das zurückzugeben ist: Bestimme dazu die kleinstmögliche Menge an Münzen, mit der das Wechselgeld gezahlt werden kann (siehe Abbildung 14.11). Gelegentlich versuchen Kunden, das Problem für den zu bezahlenden Betrag zu lösen und den Preis mit exakt passenden Münzen zu bezahlen. Mit abnehmender Sehschärfe nimmt die Wahrscheinlichkeit einer solchen Problemlösung jedoch ab.

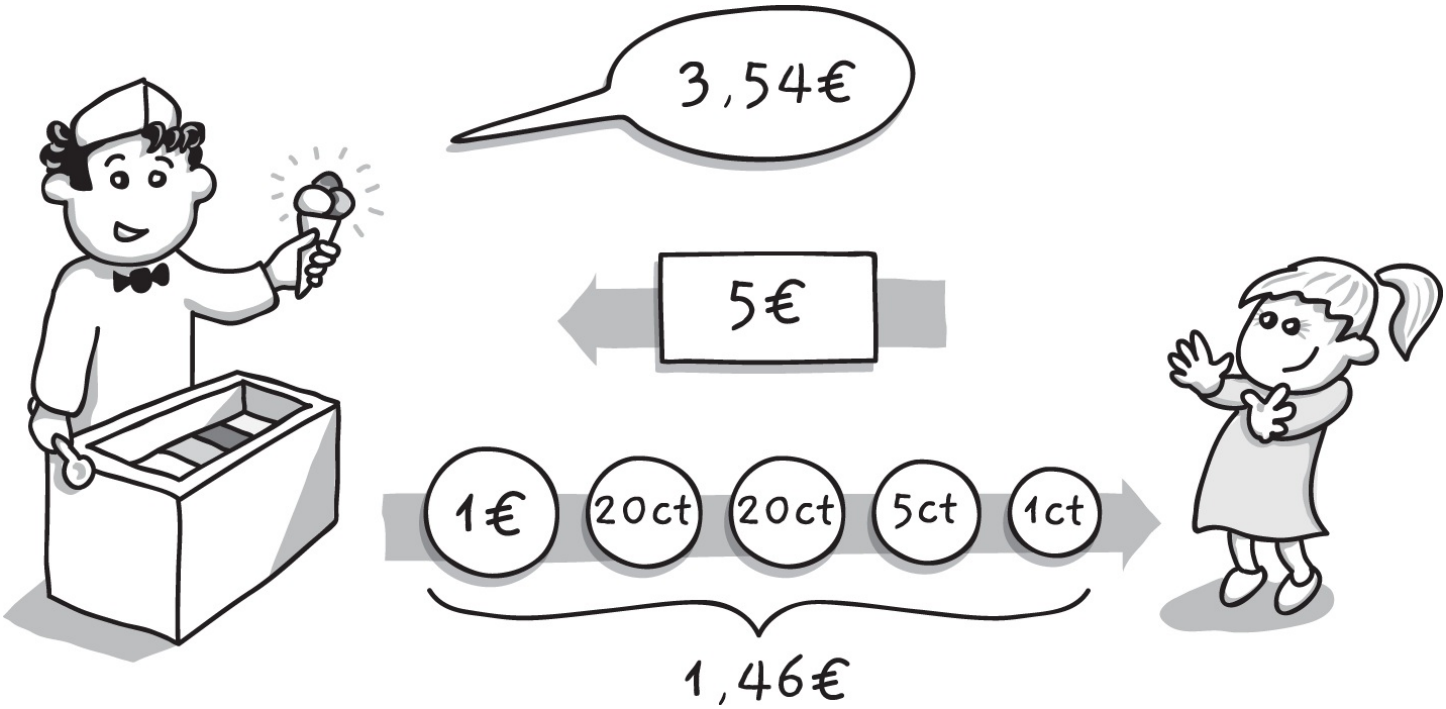


Abbildung 14.11: Das Wechselgeldproblem für $m = 146$

Das Wechselgeldproblem ist ein Optimierungsproblem. Es wird das Optimum in Form einer möglichst kleinen Zahl an Münzen gesucht, die zusammen einen bestimmten Betrag ergeben. Es kann wie folgt präzisiert werden:

Angenommen, es gibt n verschiedene Münzen mit einen Münzwert $d_i \in \{d_1, \ldots, d_n\}$. Jeder Wert von i steht dabei für eine bestimmte Münzart. Bei Euro-Münzen hätten wir beispielsweise die Münzarten $1, 2, \ldots, 9$ mit den folgenden Cent-Werten:

$$d_1 = 1, d_2 = 2, d_3 = 5, d_4 = 10, d_5 = 20, d_6 = 50, d_7 = 100, d_8 = 200, d_9 = 500$$

Für das Wechselgeld nehmen wir für jedes i, also für jede Münzart, jeweils

$$k_i \in \mathbb{N}_0$$

Münzen vom Wert d_i (k_i kann dabei 0 sein). Die Summe der Münzwerte soll genau m und die Zahl der Münzen soll minimal sein (siehe Abb. 14.12).

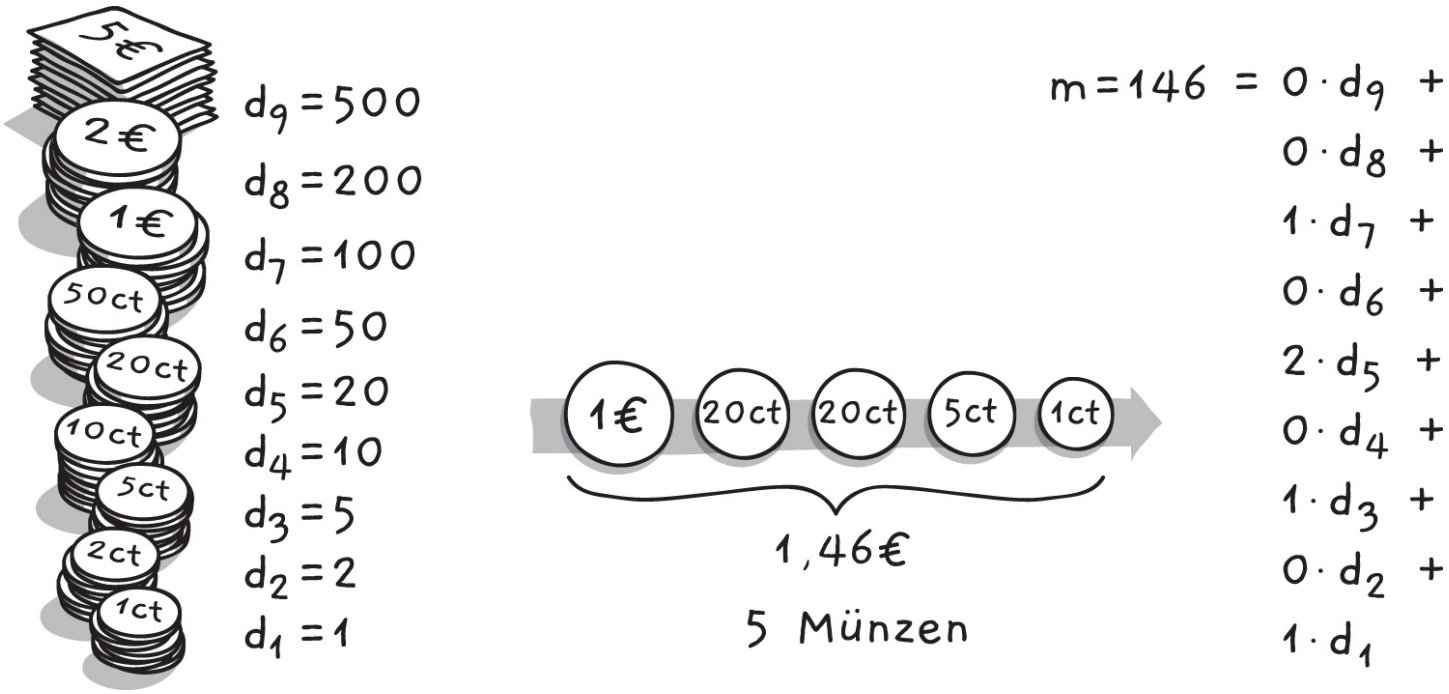


Abbildung 14.12: Mindestens fünf Münzen für das Wechselgeld von 1,46 €

Zur Lösung des Problems müssen wir also

- zu einer Folge der Länge n $d_i \in \{d_1, \ldots, d_n\}$, $d_i \in \mathbb{N}$ und einem Wert $m \in \mathbb{N}_0$
- eine gleich lange Folge $\{k_1, \ldots, k_n\}$, $k_i \in \mathbb{N}_0$ finden,

sodass

$$\sum_{i=1}^{n} (k_i \cdot d_i) = m$$

und dabei die k_i so wählen, dass

$$\sum_{i=1}^{n} k_i$$

minimal ist.

Zur Lösung des Problems gibt es einen schönen Algorithmus nach dem Prinzip *Reduzieren und Herrschen*, der sich mit *dynamischer Programmierung* sogar noch dramatisch beschleunigen lässt.

Wechselgeldproblem mit Teilen und Herrschen lösen

Die Strategie Teilen und Herrschen basiert auf der Überlegung, wie die Lösung einer großen Probleminstanz aus einer oder mehreren Lösungen von kleineren Probleminstanzen konstruiert werden kann. In diesem Fall ist das Problem *bestimme die minimale Zahl an Münzen, deren Betrag* m *ist*.

Ein kleineres Problem der gleichen Art ist eines, bei dem m kleiner ist.

Rekursionshypothese: Bei der Konstruktion einer Lösung für m gehen wir davon aus, dass irgendeine Magie uns die Lösung für jedes $m' < m$ liefert.

Wie könnte diese Magie genutzt werden?

Offensichtlich so:

Sei $c(m)$ die minimale Zahl an Münzen, deren Wert m ist. $c(m)$ ist der gesuchte Wert.

Der Betrag m kann dadurch zustande kommen, dass eine Münze zu einer kleineren Zahl an Münzen (für einen geringeren Betrag und minimal für diesen Betrag) hinzugenommen wurde.

Es gibt dabei n verschiedene Möglichkeiten, zum Wert m zu kommen. Von jeder Münzart i könnte eine Münze hinzugekommen sein:

$c(m) = c(m - d_i) + 1$ für jedes $i \in \{1, \ldots, n\}$

Das heißt, zu einer kleinen Münzkollektion wurde eine Münze hinzugenommen. Die kleinere Münzkollektion ist dabei minimal für ihren Betrag und per *Rekursionshypothese* sind wir in der Lage, $c(m - d_i)$ zu berechnen. Dabei ist $c(0) = 0$, denn wir benötigen offensichtlich 0 Münzen, um zum Betrag 0 zu kommen.

Insgesamt:

Rekursionsverankerung: $c(0)$ = 0

Rekursionsschritt: $c(m)$ = $min\{c(m - d_j) + 1 \mid$ für alle j mit $d_j < m\}$

Aus diesen Überlegungen kann ein Algorithmus konstruiert werden, der dazu noch mit dynamischer Programmierung optimiert werden kann.

Gieriges Geldwechseln

Im Alltag sind wir nicht so recht auf rekursive Algorithmen getrimmt. Wenn wir an der Supermarktkasse stehen, mit einer langen Schlange ungeduldiger Kunden hinter uns, wäre ein Kassierer, der das optimale Wechselgeld mit Bleistift und Papier in einer kleinen Tabelle nach dem Prinzip der dynamischen Programmierung berechnet, sicher eine Zeit lang eine interessante Erscheinung, die Anlass gibt, über die Laufzeitklasse von Rentnern und Kassieren bei der Suche nach Kleingeld und Wechselgeld nachzudenken. Die Filialleiterin würde dem Ganzen aber gewiss bald ein Ende bereiten.

Wir verlassen uns dann doch besser auf die Strategie, mit der uns Mutter Natur ausgestattet hat: die Gier. Das Ziel ist das Erreichen des gewünschten Betrags mithilfe der zur Verfügung stehenden Münzen. Besonders schnell kommt man dahin, wenn man immer die größtmögliche Münze nimmt, die kleiner oder gleich dem noch fehlenden Betrag ist (siehe Abbildung 14.13).

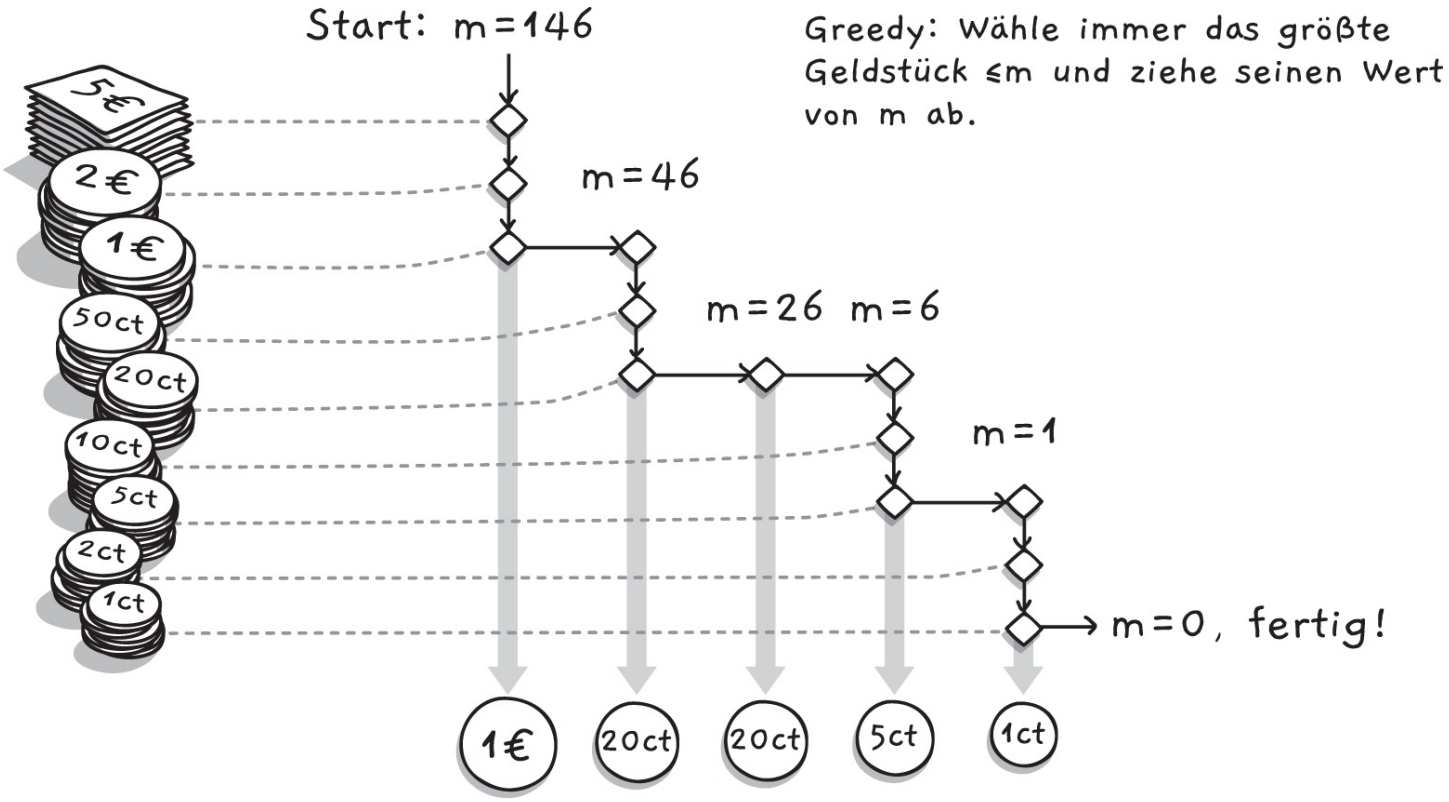


Abbildung 14.13: Gierig Münzen sammeln, bis 1,46 € erreicht ist

Die Strategie Gier führt allerdings auch beim Wechselgeldproblem nicht immer zum optimalen Ergebnis. Angenommen, wir haben die Münzwerte $d_1 = 1$, $d_2 = 10$, $d_3 = 25$ und wollen damit 30 erreichen. Mit Gier bekommen wir die Lösung

$$30 = 25 + 1 + 1 + 1 + 1 + 1 \qquad \text{6 Münzen}$$

Die beste Lösung ist aber

$$30 = 10 + 10 + 10 \qquad \text{3 Münzen}$$

Es hängt von der Münzstückelung ab, ob Gier in jedem Fall zu optimalen Lösung führt. Fast alle Währungen haben eine Münzstückelung, bei der Gier immer zum Erfolg führt.

Das Problem der Mengenüberdeckung

Im Abschnitt über das Wechselgeldproblem sind wir bereits *Approximationen* und *Approximationsalgorithmen* begegnet. Das Wechselgeldproblem wird im alltäglichen Leben mit Gier gelöst. Gier liefert nicht immer das optimale Ergebnis, ist aber einfach und schnell.

Das Beispiel ist aber vielleicht nicht völlig überzeugend, da es einen Algorithmus nach dem Prinzip der dynamischen Programmierung gibt, der es exakt und effizient löst. Zur exakten Lösung des Mengenüberdeckungsproblems gibt aber keinen effizienten Algorithmus für eine exakte Lösung. Es gibt jedoch einen gierigen Approximationsalgorithmus, der gute Ergebnisse liefert.

Um was geht es aber überhaupt? Dachdeckungen braucht man. Wer aber braucht denn

Mengenüberdeckungen?

Angenommen, Sie wollen bei der Versteigerung der Frequenzen nach dem künftigen Mobilfunk-Standard 7G mitmachen und müssen dazu nachweisen, dass Sie eine angemessene Netzabdeckung liefern können. Dabei wollen Sie natürlich auch keine Verluste machen und müssen darum berechnen, was das Ganze kostet. Ein wesentlicher Faktor bei den Kosten eines Mobilfunksystems ist der Betrieb der Mobilfunkstationen. Von diesen sollen natürlich ausreichend viele, aber dabei so wenige wie möglich im Einsatz sein.

Sie wollen natürlich auch nicht selbst neue Masten bauen, das haben andere schon gemacht oder andere mögen das machen. Es gibt also schon Stationen und die Frage ist, welche davon genutzt werden sollen, um eine vollständige Abdeckung des Gebiets zu erreichen, das Sie bedienen wollen.

Jede Station deckt eine bestimmte Region ab und dabei gibt es natürlich auch Überschneidungen. Wir erstellen also zunächst einmal eine Liste aller Stationen mit den von ihnen abgedeckten Gebieten.

Station	*Abgedeckte Gebiete*
S-0	R-92, R-17, R-10, R-11, R12, R-13
S-1	R-981, R-742, R-743, R-812, R-0
…	…
S-98	R-1, R-2, R-980, R-881, R-882, R-257, R-740, R-741, R-742
S-99	R-512, R-513, R-514, R-1, R-2, R-12, R-13

Zu jeder Station gehört mithin eine Teilmenge der Menge aller Regionen. Wir suchen jetzt die kleinstmögliche Menge an Stationen, die das gesamte Gebiet abdeckt. Nehmen wir an, es gibt 100 Stationen S-0 bis S-99 und 1000 Regionen R-0 bis R-999. Wir brauchen dann eine Auswahl der Stationen, die so beschaffen ist, dass die Vereinigung der zugehörigen Regionen die Menge aller Regionen ist.

Identifiziert man die Stationen S mit den Regionen, die sie abdecken, dann kann man das Problem etwas präziser so beschreiben:

Problem: optimale Mengenüberdeckung

Eingabe: eine Menge S von Teilmengen einer Menge R

Berechne: eine minimale Auswahl $T \subseteq S$ derart, dass $(\bigcup_{t \in T} t) = R$

Mit roher Kraft kann das Problem exakt gelöst werden. Wir probieren jedes mögliche T (jeweils eine Mengen von Stationen) aus, ob es die gewünschte Abdeckung liefert. Bei denen, die alle Regionen abdecken, zählen wir die Stationen und nehmen die mit der

geringsten Zahl an Stationen.

Das ist die Lösung eines Optimierungsproblems mit erschöpfender Suche. Der Suchraum ist die Menge aller 2^n Teilmengen der n Stationen. Das ist in jedem Fall exponentiell $O(2^n)$. Ein entsprechender *Rohe-Kraft*-Algorithmus agiert, wie zu erwarten, mit reiner, roher Kraft:

```
Setcovering (S, R)          S ist eine Menge von Teilmengen von R
    result ← ∅
    count ← ∞
    for T ← {T | T ⊆ S}
        if (⋃ t) = R and |T| < count
           t∈T
            count ← |T|
            result ← T
    return result
```

Eine *gierige* Lösung des Problems ist recht naheliegend:

- ✔ Wähle so lange eine Station nach der anderen aus, bis die gewünschte völlige Abdeckung erreicht ist.
- ✔ Nimm dabei als nächste immer die Station, die die meisten der noch nicht abgedeckten Regionen abdeckt.

Dieser Algorithmus benötigt $O(n^2)$ Schritte. In jedem Schritt werden alle restlichen Stationen untersucht, um die Station mit der größten Abdeckung herauszufinden. Statt eines exponentiellen Rohe-Kraft-Algorithmus haben wir einen gierigen Algorithmus mit polynomieller Laufzeit. Schon besser.

```
SetcoveringG (S, R)         S ist eine Menge von Teilmengen von R
    result ← ∅              hier werden die Stationen gesammelt
    notCovered ← R          die noch nicht abgedeckten Regionen
    sRemaining ← S          die noch ungenutzten Stationen
    while notCovered ≠ ∅    solange es nicht abgedeckte Regionen gibt
        sWithCoverage ← max({(s, x) | s ∈ sRemaining, x = |notCovered ∩ s|})
            mit x als Vergleichskriterium
        s ← sWithCoverage.1     s: die Station mit der größten Abdeckung
        notCovered ← notCovered \ s    Station = Menge von abgedeckten Regionen
        sRemaining ← sRemaining \ {s}
        result ← result ∪ {s}
    return result
```

Der Algorithmus identifiziert Stationen und die von ihnen abgedeckten Regionen. Eine Station »ist« eine Menge von Regionen. Das macht die Präsentation des Algorithmus

übersichtlich. In einer Implementierung wird man natürlich nicht völlig naiv mit einer Datenstruktur arbeiten wollen, die eine allgemein verwendbare Datenabstraktion »Menge« zur Verfügung stellt (außer vielleicht in einem Prototyp), sondern eine, welche die besonderen Anforderungen des Algorithmus berücksichtigt.

Der gierige Algorithmus liefert eine gute Näherung an die exakten Lösung bei akzeptabler Laufzeit. Das Beispiel zeigt, dass Gier nicht der perfekte Ersatz für unendlichen Fleiß ist, aber man sich mit ihr viel Mühe ersparen kann und dabei im Allgemeinen ganz gut über die Runden kommt.

Gier in Perfektion

Gier kennt kein Zurück: Was sie einmal in ihren Besitz gebracht hat, lässt sie nicht mehr los. Wie wir gesehen haben, führt das in vielen Fällen dazu, dass dabei am Ende nicht die allerbeste Lösung herauskommt.

Es gibt aber auch Situationen, in denen die Gier uns mit ihrer »unsichtbaren Hand« zielstrebig zum Optimum führt. Ein Prachtbeispiel für einen Algorithmus, der zwar gierig ist, aber trotzdem stets die beste Lösung liefert, sind der Algorithmus von Dijkstra und seine Weiterentwicklung A*; wir hatten die beiden bereits weiter oben vorgestellt. Ausgehend vom Startknoten werden neue Knoten entlang der Kanten des Graphen entdeckt. Dabei wird »gierig« immer der naheliegendste Knoten (bei Dijkstras Algorithmus) beziehungsweise derjenige mit der besten Prognose (bei A*) als nächster ausgewählt. Sobald ein Knoten einmal untersucht wurde, wandert er aus dem Grenzbereich T heraus und wird in die Menge S der sicheren Knoten »aufgenommen«. Danach muss der Knoten nicht noch einmal untersucht werden. Was S einmal geschluckt hat, das spuckt die Menge nicht wieder aus – ein untrügliches Kennzeichen von Gierigkeit.

Kruskals Algorithmus aus Kapitel 10 zum Berechnen des minimalen Spannbaums ist ebenfalls ein gieriger und dennoch *exakter* Algorithmus. Das Ziel dieses Algorithmus ist es, alle Knoten in einer Menge über möglichst kurze Verbindungen (»Kanten«) miteinander zu vernetzen. Der Algorithmus sortiert darum zunächst alle möglichen Kanten aufsteigend nach ihrer Länge, geht anschließend die Kanten der Reihe nach durch und ergänzt nach und nach den unvollständigen Graphen so lange, bis es keine unverbundenen Komponenten mehr gibt. Die Gesamtlänge der Kanten soll möglichst gering sein, also ist es naheliegend, an jeder Entscheidungsstelle – welche Kante kommt hinzu? – kurze Kanten zu bevorzugen. Der Algorithmus ist gierig: Eine einmal ausgewählte Kante wird nicht mehr aus der Auswahl entfernt. Tatsächlich führt die Methode zum optimalen Ergebnis, nämlich dem beziehungsweise einem minimalen Spannbaum.

Huffman-Codierung

Zum Abschluss wollen wir noch ein weiteres Beispiel für einen gierigen und doch exakten

Algorithmus kennenlernen.

Bei einer Codierung werden die Zeichen eines Textes in Codeworte umgewandelt. Bei einer Binärcodierung werden beispielsweise Buchstaben und Ziffern in Folgen von Nullen und Einsen umgewandelt. Man kann dabei jedes Zeichen in ein Codewort fester Länge umwandeln. Beispielsweise werden durch den bekannten ASCII-Code einem Zeichen 7 Bits zugeordnet. A etwa erhält das Codewort 1000001. Ursprünglich war das achte Bit ein Prüfbit. Später wurde es zur Erweiterung des codierbaren Zeichensatzes genutzt. A hat dann den 8-Bit Code 01000001.

Bei einen Code mit Codeworten fixer Länge wird für häufige Zeichen genauso viel Platz gebraucht, wie für Zeichen, die nur sehr selten vorkommen. Geschickter wäre es, wenn Zeichen, die oft vorkommen, möglichst kurze Codeworte haben, dafür könnte man den seltenen Zeichen einen längeren Code zumuten.

Der Morsecode basiert auf diesem Prinzip. Er hat zwei Basis-Zeichen: *An* und *Aus*. Dabei können die unterschiedlichsten Dinge an oder aus sein: eine Lichtquelle, eine Stromquelle, ein Funksignal, ein Ton und so weiter. Die Information steckt darin, dass das Signal lang oder kurz an oder aus ist. Das wird üblicherweise mit Punkten und Strichen codiert:

- ✔ · Punkt: kurz an
- ✔ – Strich: lang an
- ✔ Leerraum: kurz aus
- ✔ / Trennzeichen: lang aus

Zeichen werden mit unterschiedlich langen Codierungen dargestellt. Ein E ist nur ein Punkt (·). Der Code für Q dagegen ist deutlich länger (– – ·–). Die Länge richtet sich nach der Häufigkeit der Buchstaben in einem englischsprachigen Text.

Bei einem Code fester Länge ist es einfach, die einzelnen Codeworte auseinanderzuhalten. Alle 8 Bit beginnt beispielsweise im ASCII-Code ein neuer Buchstabe. So einfach geht das nicht bei einem Code mit unterschiedlicher Länge der Codeworte. Die Codeworte können nur dann sicher erkannt werden, wenn kein Codewort der Anfang eines anderen ist. Das nennt man die *Fano-Bedingung*. Ein Code erfüllt die Fano-Bedingung, wenn kein Codewort ein Präfix (Anfang) eines anderen ist.

Der Morsecode erfüllt die Fano-Bedingung auf sehr einfache Art: jedes Codewort, also die Codierung eines Zeichens, wird mit einer langen Pause als Trennzeichen beendet. Das kann man natürlich auch geschickter machen. Wie beispielsweise die Huffman-Codierung, die in vielen Verfahren zur Datenkompression zum Einsatz kommt.

Bei einer Huffman-Codierung wird jedem Zeichen eines Textes ein Binärcode zugeordnet, derart, dass häufig vorkommende Zeichen einen möglichst kurzen Code haben. Um eine

eindeutige Erkennbarkeit zu gewährleisten, darf kein Codewort der Präfix eines anderen Codeworts sein. Das heißt, die Codierung eines Zeichens darf nicht der Anfang der Codierung eines anderen Zeichens sein. Die Codierung

$$A \to 0,\ B \to 10,\ C \to 110,\ D \to 1110,\ E \to 11110$$

erfüllt diese Bedingung. Der Morsecode benötigt ein Trennzeichen, um die Fano-Bedingung erfüllen zu können. Ohne Trenner wäre der Code für E ($\cdot$) der Anfang vieler anderer Codewörter, beispielsweise von A :

$$E \to \cdot / \; A \to \cdot - /$$

Natürlich ist nicht die Codierung eines Textes das Problem, sondern die Festlegung der Codierung: Welches Zeichen wird wie codiert? Dazu müssen wir natürlich die Wahrscheinlichkeit der einzelnen Zeichen kennen. Der Einfachheit halber gehen wir von einer Binärcodierung aus. Das Verfahren kann aber für beliebige Codierungen verwendet werden.

Die Huffman-Codierung löst das Problem der optimalen Codierung:

Problem: optimale Codierung

Eingabe: ein Zeichensatz, bei dem jedem Zeichen die Wahrscheinlichkeit des Vorkommens in einem Text zugeordnet ist

Berechne: eine optimale Codierung, das heißt eine Zuordnung Zeichen $\to$ Codewort, bei der kein Codewort Präfix eines anderen Codeworts ist und Texte in Sequenzen minimaler Länge übertragen werden

Die Lösung wird als Binärbaum konstruiert, dessen Blätter die Zeichen des Textes repräsentieren und die Wege zu den Blättern (0 für »links weiter«, 1 für »rechts weiter«) die jeweiligen Codeworte (siehe Abbildung 14.14). Durch diese Konstruktion ist garantiert, dass die Codeworte keine gemeinsamen Präfixe haben und der Code damit die Fano-Bedingung erfüllt.

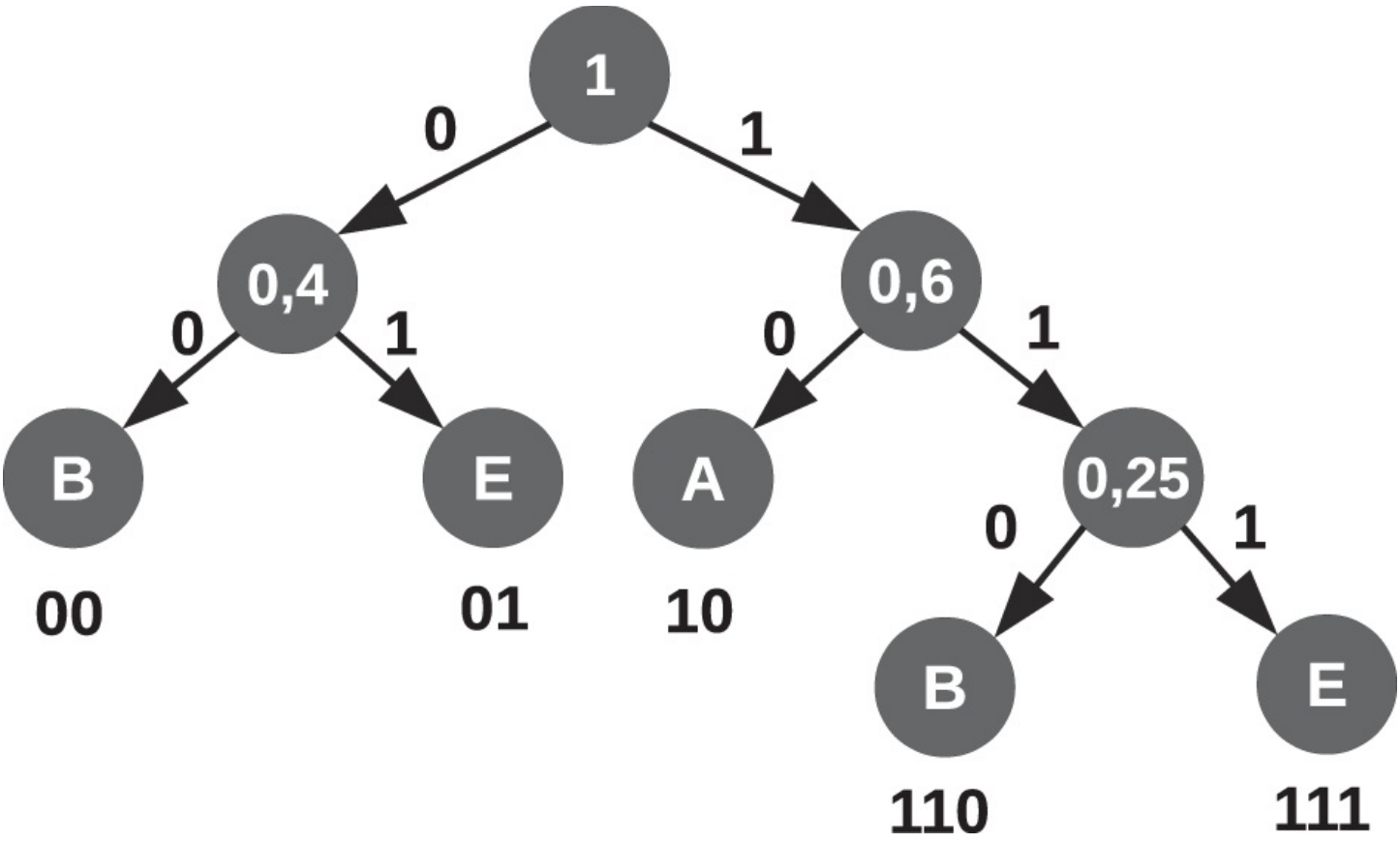


Abbildung 14.14: Huffman-Codierung

Bei der Konstruktion des Baumes wird das Prinzip Gier angewendet. Wenn auch gewissermaßen rückwärts: Möglichst lange Codeworte für seltene Zeichen. Es geht in jedem Schritt um möglichst geringe Wahrscheinlichkeiten.

Der Baum wird in einem Verfahren konstruiert, dass in verblüffender Art dem Verfahren zur Konstruktion eines minimalen Spannbaums ähnelt.



In Kapitel 10 wird ein Algorithmus besprochen, mit dem der minimale Spannbaum eines Graphen berechnet wird. Der Algorithmus beginnt mit den einzelnen Knoten des Graphen und verknüpft diese dann sukzessive miteinander zu immer größeren Bäumen – so lange, bis am Schluss alle in einem Baum zu finden sind.

Das Verfahren zur Berechnung einer Huffman-Codierung ist:

- ✔ Wir beginnen damit, dass jedem Zeichen ein Baum zugeordnet wird, der aus nur einem Knoten besteht. Jeder Baum hat eine zugeordnete Häufigkeit. Das ist am Anfang die Häufigkeit des in ihm enthaltenen Zeichens.
- ✔ Dann führen wir die zwei Bäume mit der geringsten Häufigkeit zu einem neuen Baum zusammen, dessen Häufigkeit ist die Summe der Häufigkeiten der zusammengeführten Bäume.
- ✔ Dieses Zusammenführen wird so lange wiederholt, bis nur noch ein Baum übrig

bleibt.

Betrachten wir ein kleinen Zeichensatz, der aus den Zeichen A, B, C, D und E besteht. Die Häufigkeit/Wahrscheinlichkeit der Zeichen sei:

Zeichen	*A*	*B*	*C*	*D*	*E*
Häufigkeit	0.35	0.1	0.2	0.2	0.15

Im ersten Schritt wird jedes Zeichen ein Baum (siehe Abbildung 14.15). Anschließend werden die Bäume mit der geringsten Häufigkeit vereinigt (siehe Abbildung 14.16). Danach haben B und E die geringste Häufigkeit und werden vereinigt (siehe Abbildung 14.17).



Abbildung 14.15: Starte mit einem Wald voll winziger Bäume.

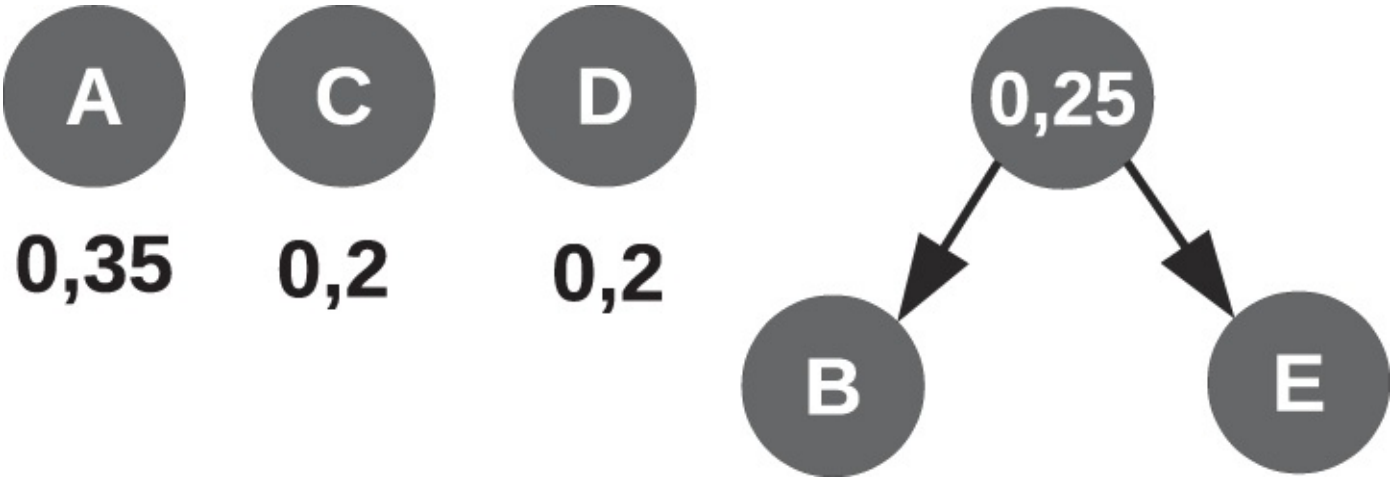


Abbildung 14.16: Vereinigung von B und E

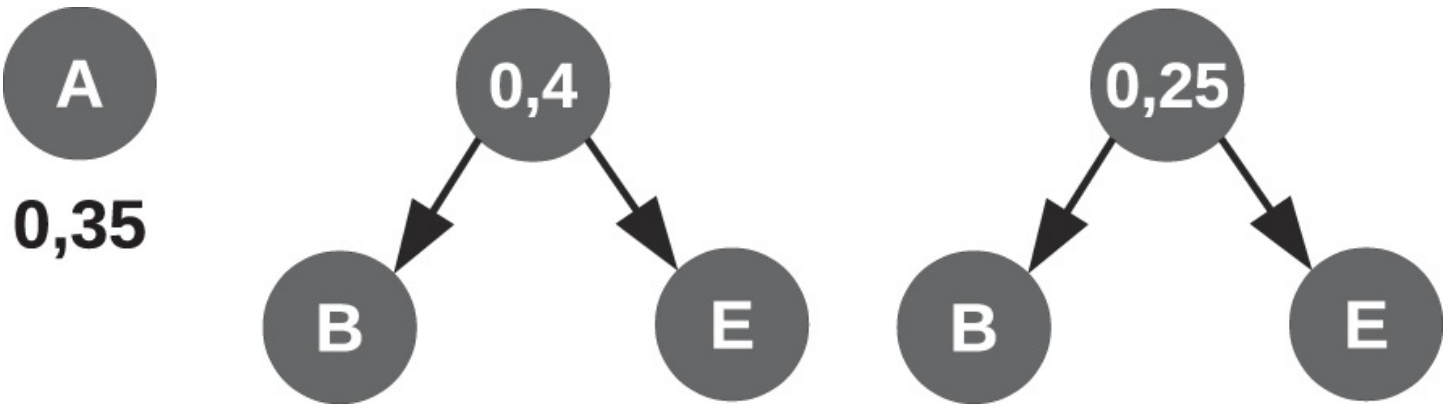


Abbildung 14.17: Vereinigung der nächsten Bäume mit der geringsten Wahrscheinlichkeit

So geht es weiter, bis wir schließlich alle Bäume zu einem einzigen vereinigt haben. Bei diesem definieren dann die Pfade von der Wurzel zu den Blättern die Codierung des Zeichens im Blatt, und zwar mit »gehe nach links« $\rightarrow 0$ und »gehe nach rechts« $\rightarrow 1$.

Die Bäume könnten zur Beschleunigung des Verfahrens in einer Prioritätswarteschlange gespeichert werden. Die Konstruktion des Codes hätte dann die Laufzeiteffizienz $O(n \log n)$, wobei n die Zahl der Zeichen ist.

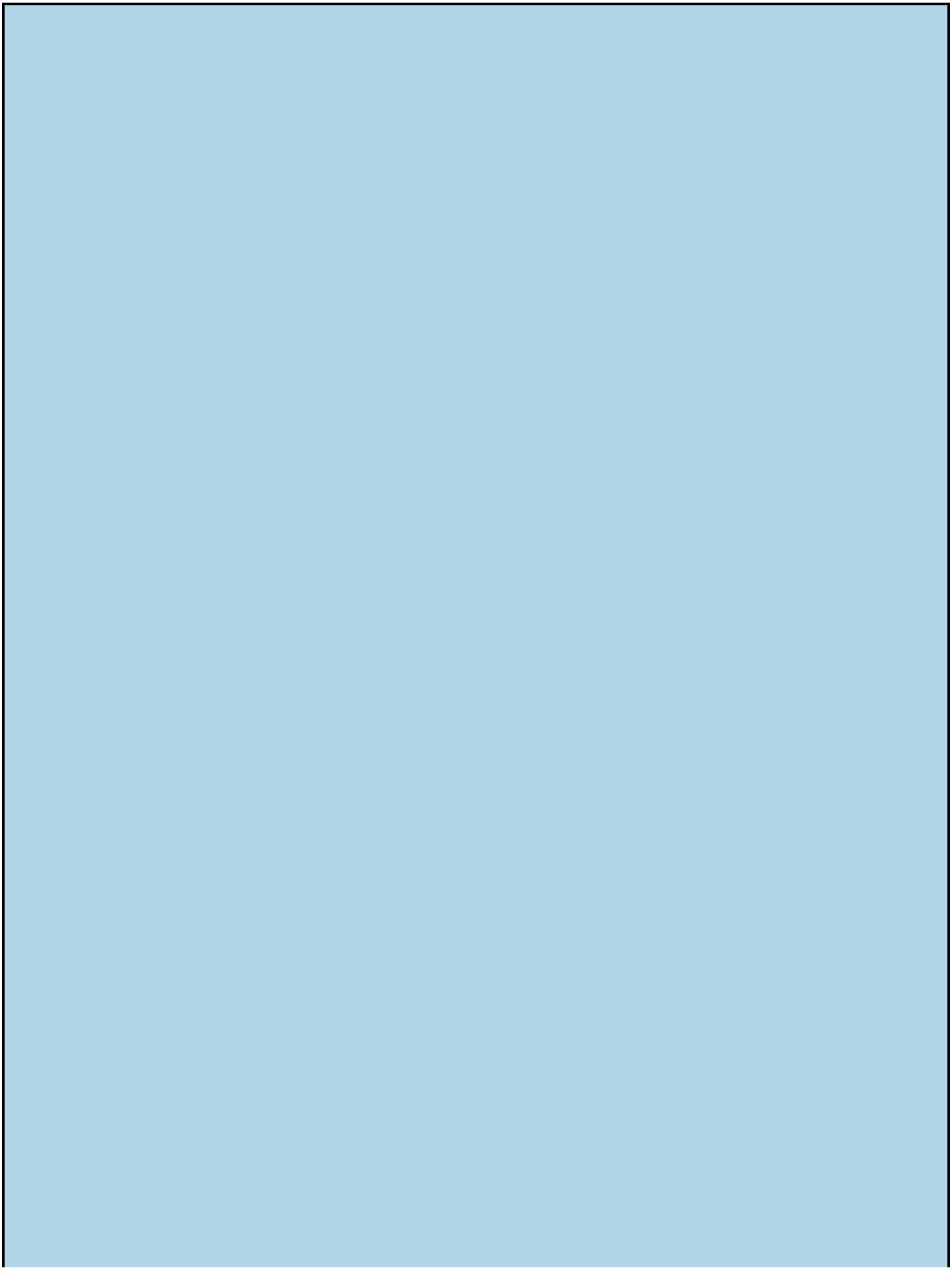
Teil V

Der Top-Ten-Teil





Besuchen Sie uns auf www.facebook.de/fuerdummies!



IN DIESEM TEIL …

Zehn wichtigste Datenabstraktionen und Datenstrukturen

Zehn wohlgemeinte Ratschläge für angehende Algorithmikerinnen und Algorithmiker

Kapitel 15

Zehn Datenabstraktionen und Datenstrukturen

IN DIESEM KAPITEL

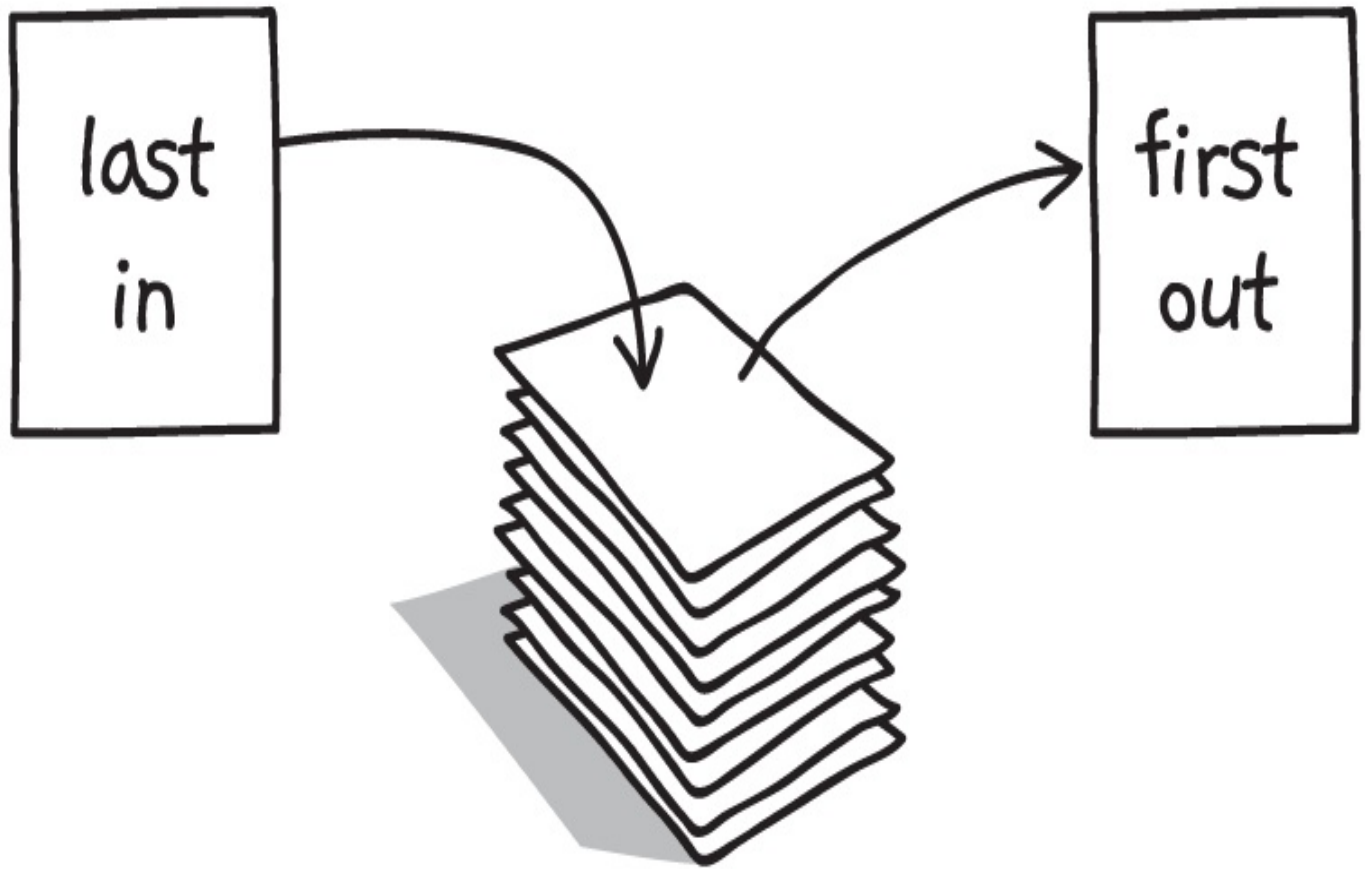
- Stapel
- Warteschlange und Prioritätswarteschlange
- Liste, Array
- Menge, Verzeichnis, Relation
- Graph, Baum

Die Realisation von Datenabstraktionen gehört zu den wichtigsten Aufgaben, die Algorithmen und Datenstrukturen zu erledigen haben. Die Daten werden in der Struktur gespeichert, die Algorithmen realisieren die Zugriffsoperationen mit den von den Nutzern gewünschten Eigenschaften.

Bei manchen Programmiersprachen sind solche Datenabstraktionen Teil der Sprache, bei anderen gehören sie zu der mitgelieferten oder üblichen »Standardbibliothek«. Für Java gab es fast von Anfang an das *Collections Framework*, für C++ schon früh die *Standard Template Library*, Python hat die *Python Standard Library*. Wem das nicht reicht, der findet leicht weitere, traditionsreiche Bibliotheken wie etwa Googles *Guava*.

Diese Kollektionen von Typen sind wichtige Bestandteile jeder Anwendungsentwicklung. Sie nehmen den Entwicklern viel Arbeit mit der Entwicklung von Algorithmen von Datenstrukturen ab. Sie bieten aber auch gute Beispiele für Algorithmen von Datenstrukturen. Die Klasse *TreeMap* des *Collections Framework* von Java ist beispielsweise als Rot-Schwarz-Baum (siehe Abschnitt *Rot-Schwarz-Bäume* in Kapitel 9) implementiert. So etwas möchte ein ausgelasteter Anwendungsentwickler nur in Ausnahmefällen selbst leisten beziehungsweise sein Arbeitgeber bezahlen.

Stapel



- ✔ **Datenmodell:** eine Liste/Sequenz, auf die nur an einem Ende zugegriffen werden kann
- ✔ **Datenabstraktion:** eine Kollektion mit den Operationen »ablegen« (*push*), »entnehmen« (*pop*) und »Test, ob leer« (*empty*). Das zuletzt Hinzugefügte kann als Erstes entnommen werden.
- ✔ **Datenstruktur:**
    - Array mit einem Index, der die Befüllung anzeigt
    - verkettete Liste
    - algebraischer Datentyp Liste

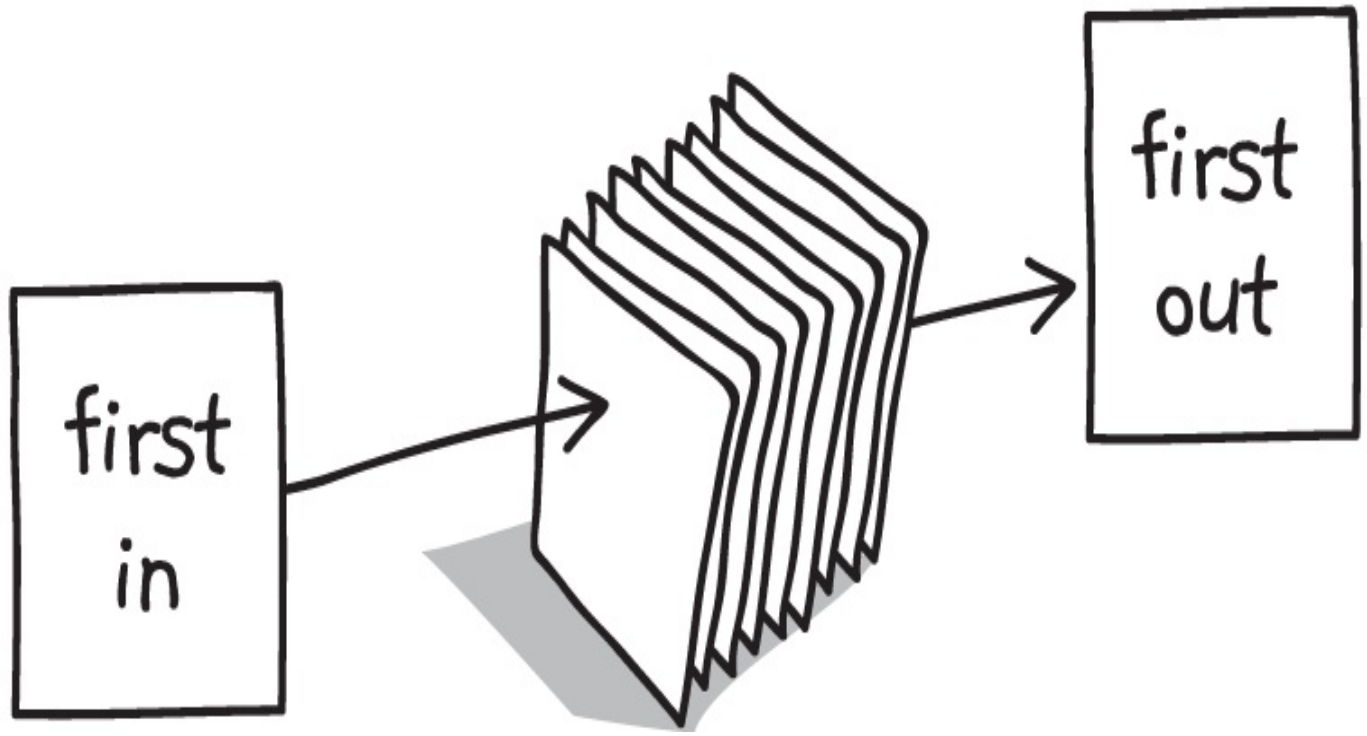
Auf einem Stapel kann man etwas ablegen und das zuletzt Abgelegte wieder entnehmen. Das ist eine Kollektion mit einem sehr einfachen Service. Trotzdem sind Stapel eine wichtige und vielseitig verwendbare Datenabstraktion. Viele Algorithmen benötigen Stapel. Jeder rekursive Algorithmus wird mithilfe von Stapeln realisiert.

Stapel gibt es imperativer und funktionaler Variante, mit beschränkter und unbeschränkter Größe. Meist wird die imperative Variante verwendet: Ablegen erzeugt keinen neuen größeren Stapel, sondern verändert einen existierenden. Als Datenstruktur kommen verkettete Listen und Arrays in Frage. Bei unbeschränkter Größe ist die verkettete Liste die Datenstruktur der Wahl. Hat der Stapel eine beschränkte Größe, dann kann man alle

Elemente in einen Array fester Größe ablegen. Beides kann auch kombiniert werden. Dann hat man eine verkettete Liste von Arrays als Datenstruktur hinter der Stapel-Schnittstelle.

Ein unbeschränkter Stapel kann befragt werden, ob er leer ist, man kann dort immer eine Element ablegen und ein nicht-leerer Stapel kann sein zuletzt hinzugefügtes Element liefern. Das ist das »natürliche« Verhalten einer ganz bestimmte Datenstruktur: der Liste als algebraischer Datenstruktur: entweder »leer« oder »Kopf und Rest der Liste«.

Warteschlange



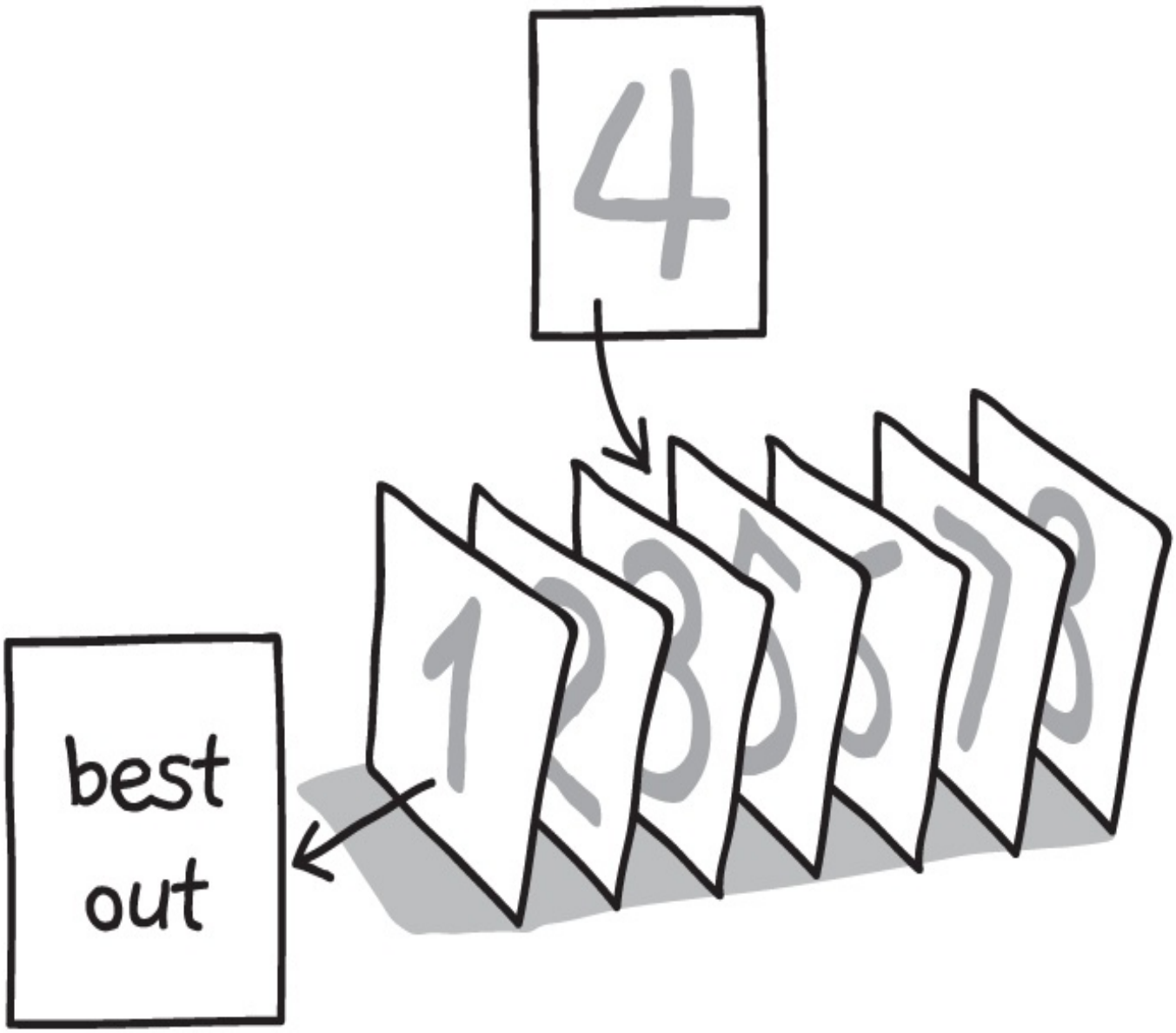
- ✔ **Datenmodell:** eine Liste/Sequenz, auf die an beiden Enden zugegriffen werden kann – an einem lesend, am anderen schreibend
- ✔ **Datenabstraktion:** eine Kollektion mit den Operationen »hinzufügen« (*enqueue*), »entnehmen« (*dequeue*) und »Test, ob leer« (*empty*). Das zuerst Hinzugefügte wird als Erstes entnommen.
- ✔ **Datenstruktur:**
  - veränderliche verkettete Liste
  - zwei persistente Listen: eine zum Hinzufügen, eine zum Entnehmen und gelegentlichen Umkopieren

Warteschlangen sind Kollektionen, durch die Werte geschleust werden: an einem Ende rein, am anderen Ende raus. Ähnlich wie beim Stapel stellt man sich unter einer

Warteschlange eine Liste mit beschränkter Funktionalität vor (Datenmodell). Im Gegensatz zu Stapeln haben Warteschlangen keinen algebraischen Datentyp, der in seinem »natürlichen« Verhalten direkt dem einer Warteschlange entspricht.

Imperative Implementierungen der Datenabstraktion *Warteschlange* können die Daten *am Platz* verändern. In einem funktionalen Kontext müssen die Datenstrukturen persistent sein. Modifikationen bestehender Datenstrukturen sind nicht erlaubt. Modifikationen an einem Ende können effizient realisiert werden. Die Warteschlange mit ihren beiden Ende erfordert dann andere Tricks, um die Zahl der Kopieroperationen zu minimieren.

Prioritätswarteschlange



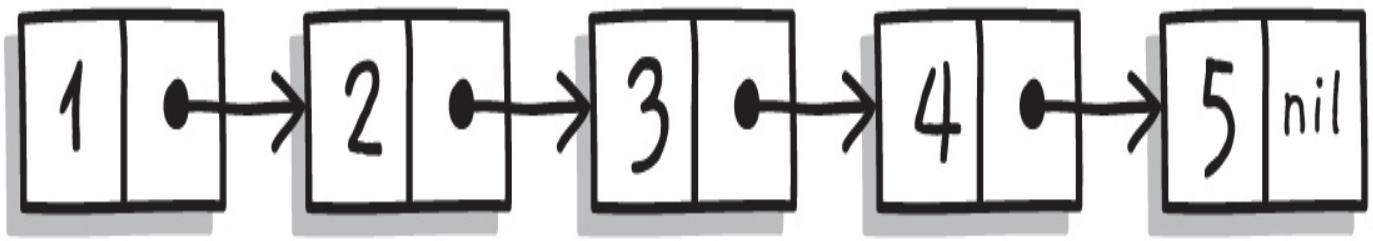
- ✔ **Datenmodell:** eine Liste/Sequenz, auf die an beiden Enden zugegriffen werden kann – an einem lesend, am anderen schreibend. Die enthaltenen Elemente werden nach

Priorität sortiert. »Priorität« ist dabei ein vergleichbarer Wert, der den Elementen zugeordnet wird, meistens eine natürliche Zahl mit »je kleiner, je besser«.

- ✔ **Datenabstraktion:** wie bei Warteschlangen Kollektion mit den Operationen »hinzufügen« (*enqueue*), »entnehmen« (*dequeue*) und »Test, ob leer« (*empty*).
- ✔ **Datenstruktur:**
  - veränderliche verkettete Liste
  - als Heap organisierter Baum
  - als Heap organisiertes Array

Wird eine Array oder eine Liste verwendet, dann hat man die Wahl zwischen der Aufrechterhaltung einer sortierten Reihenfolge und der Suche nach dem kleinsten Wert in einer unsortierten Kollektion. Die Datenstruktur Heap realisiert, ob als Baum oder Array, den effizienten Kompromiss einer halb-sortierten Kollektion.

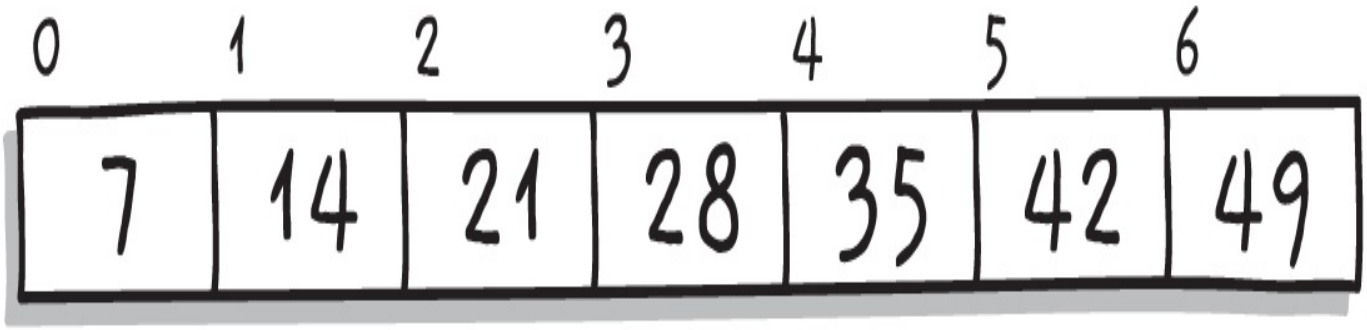
Liste



- ✔ **Datenmodell:** eine Menge von Werten, denen jeweils eine Position zugeordnet ist
- ✔ **Datenabstraktion:** Sequenz mit Zugriff über einen Index
- ✔ **Datenstruktur:**
  - Array
  - verkettete Liste
  - Trie

Bei den imperativen Implementierungen der Datenabstraktion *Liste*, den Arrays und verketteten Listen, können Werte »am Platz« verändert werden. So etwas ist nicht erlaubt bei den persistenten Datenstrukturen, die in der funktionalen Programmierung genutzt werden. Wir sehen eine Steigerung in der Reihe *Stapel* (Modifikationen an einem Ende) über *Warteschlange* (Modifikationen an beiden Enden) zu *Liste* (Modifikation an jeder Stelle). Um die Zahl der Kopieroperationen zu minimieren, werden persistente Listen mit wahlfreiem Zugriff oft als Baumstrukturen wie dem Trie realisiert.

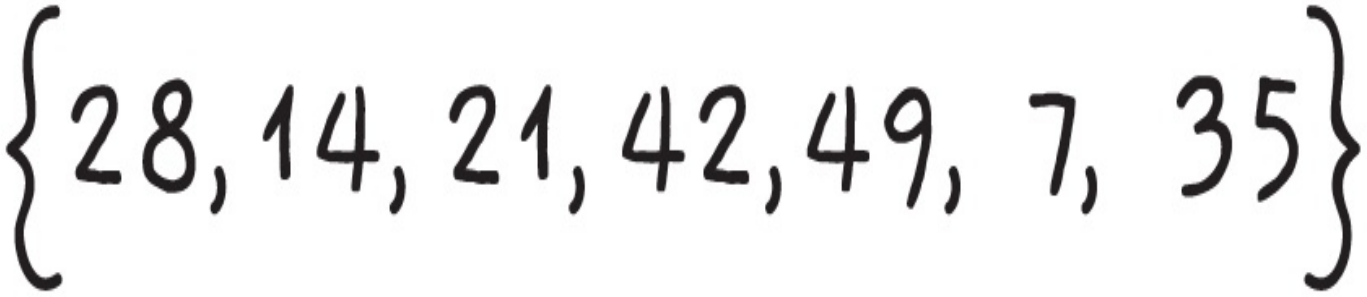
Array



- ✔ **Datenmodell:** eine Liste/Sequenz, auf die an jeder Position zugegriffen werden kann
- ✔ **Datenabstraktion:** lesender und schreibender Zugriff über einen Index
- ✔ **Datenstruktur:**
  - Array

Arrays sind eine Datenstruktur, welche die Datenabstraktion Liste mit wahlfreiem Zugriff direkt unterstützt.

Menge



- ✔ **Datenmodell:** eine ungeordnete Kollektion, die ein Element enthalten kann oder nicht. Den Elementen einer Menge ist weder eine Reihenfolge noch eine Anzahl zugeordnet.
- ✔ **Datenabstraktion:** imperativ: »Einfügen« eines nicht enthaltenen Elements, »Test, ob ein Element enthalten ist«, »Löschen« eines enthaltenen Elements; funktional: Realisation der mathematischen Mengenoperationen
- ✔ **Datenstruktur:**
  - Suchbaum
  - Hashtabelle

Mengen sind ein zentrales mathematisches Konzept und als solches sehr wichtig als Datenmodell. Als Datenabstraktion spielen sie nur eine geringe Rolle. So wie Python (Klasse *Set* mit einer Hashtabelle als Datenstruktur) und Java (*TreeSet* (Suchbaum-basiert) und *HashSet* (Hashtabellen-basiert)) bieten manche Sprachen Datenabstraktionen für Mengen an. In der Regel werden aber direkt die Datenstrukturen genutzt, die zu den Anforderungen des jeweiligen Algorithmus passen. Eine Liste oder ein Array ist ausreichend, wenn es nur darum geht, Daten zusammenzustellen. Es bringt nicht viel, die Daten in eine Datenabstraktion einzupacken, die aufpasst, dass jedes Element nur einmal vorhanden ist und die die Reihenfolge der Elemente wegabstrahiert.

Etwas anders sieht es aus, wenn die Menge als »Massenspeicher« betrachtet wird. Das Vermeiden von Duplikaten ist sinnvoll: warum sollte eine Information zweimal gespeichert werden. Das Fehlen einer Reihenfolge ist dann ebenfalls sinnvoll. Ohne eine von der Anwendung vorgegebene Reihenfolge wie bei Listen ist die Implementierung frei von der Verpflichtung, eine bestimmte Reihenfolge einzuhalten und kann die Daten beliebig speichern, etwa als Hashtabelle oder Suchbaum.

Verzeichnis

- ✔ **Datenmodell:** eine gespeicherte Abbildung/Funktion, also eine eindeutige (funktionale) Zuordnung von Daten zu anderen Daten
- ✔ **Datenabstraktion:** lesender und schreibender Zugriff über einen Schlüssel
- ✔ **Datenstruktur:**
  - (Balancierte) Suchbäume
  - Hashtabelle

Ein Verzeichnis ist ein Sonderfall einer Menge. Eine gespeicherte Abbildung, die Schlüsselwerte auf Datenwerte abbildet, ist eine Menge an Schlüssel-Wert-Paaren. Aktuelle Programmiersprachen haben so etwas im Angebot und dieses Angebot wird in der Anwendungsentwicklung ausgiebig genutzt. In Python werden Verzeichnisse *Mapping Type* oder *dictionary* (dict) genannt. Java bietet eine *HashMap* (Verzeichnis mit Hashtabelle als Datenstruktur) und *TreeMap* (Verzeichnis mit Rot-Schwarz-Baum als Datenstruktur) an.

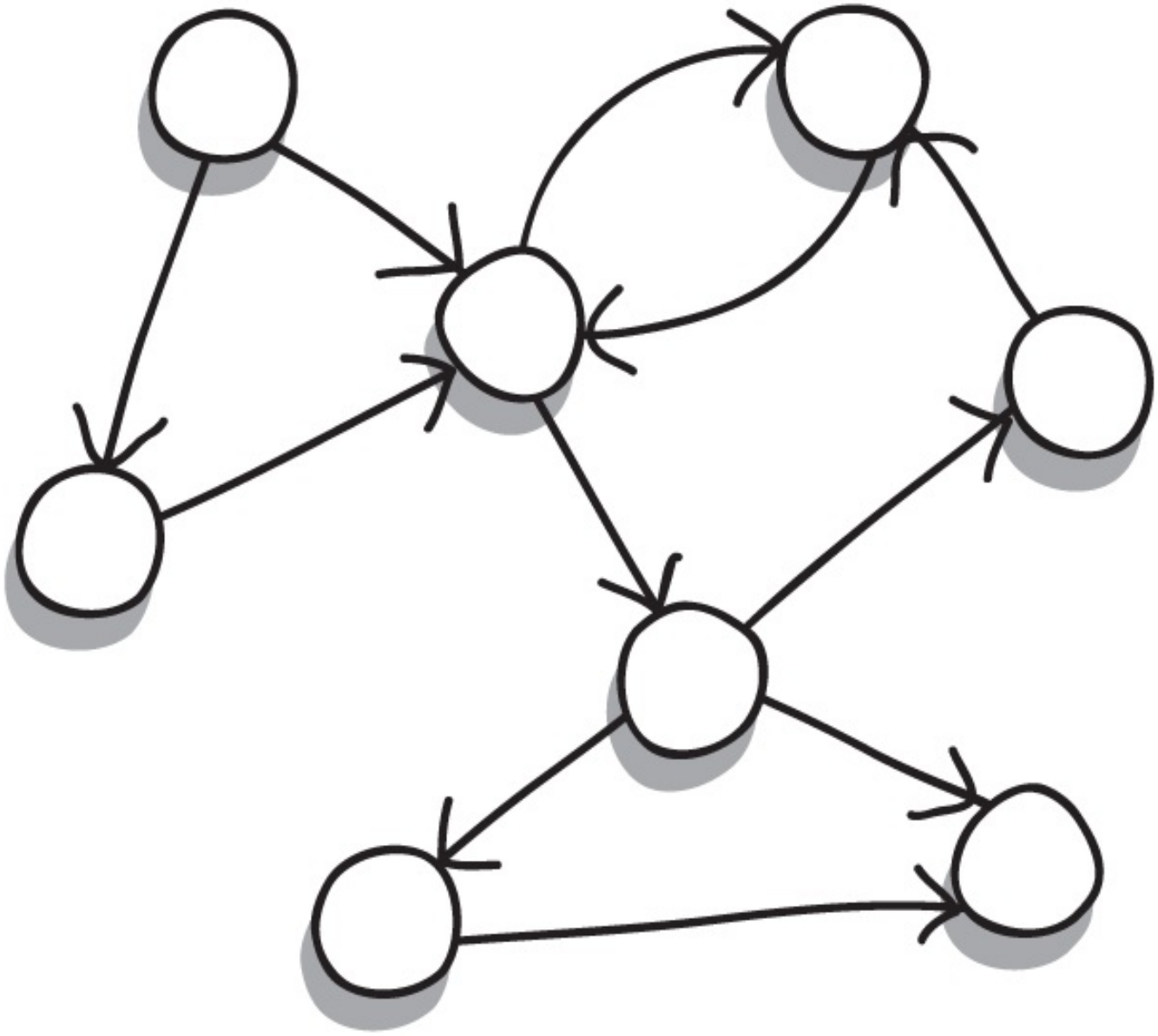
Verzeichnisse sind primär eine imperative Datenabstraktion, man erwartet, dass die Kollektion sich ändert, wenn etwas hinzugefügt wird. In einem funktionalen Kontext benötigt man eine persistente Variante, bei der Hinzufügen eine neue Version des Verzeichnisses erzeugt. Baumbasierte Implementierungen haben damit kein besonderes Problem. Persistente Hashtabellen werden ebenfalls eingesetzt. Sie bestehen aus Hashtabellen, die in Tries organisiert werden.

Relation

- ✔ **Datenmodell:** Teilmenge des Kreuzprodukts mehrerer Mengen, also eine Menge von Tupeln gleicher Länge mit Komponenten, die positionsweise zu den gleichen Mengen gehören und die gleiche Bedeutung haben
- ✔ **Datenabstraktion:** Tabellen einer relationalen Datenbank mit den entsprechenden Operationen
- ✔ **Datenstruktur:**
  - Adjazenzlisten und Adjazenzmatrizen bei binären Relationen
  - Suchbäume und Hashtabellen

Das Datenmodell einer Relation wird beim Entwurf von Algorithmen oft verwendet. Die Kanten eines Graphen sind eine Relation. Das heißt, wenn immer Graphen im Spiel sind, arbeitet man mit einer Relation. Als Datenabstraktion sind Relationen in aktuellen Programmiersprachen praktisch nicht existent, es sei denn, es handelt sich um eine Datenbanksprache.

Graph

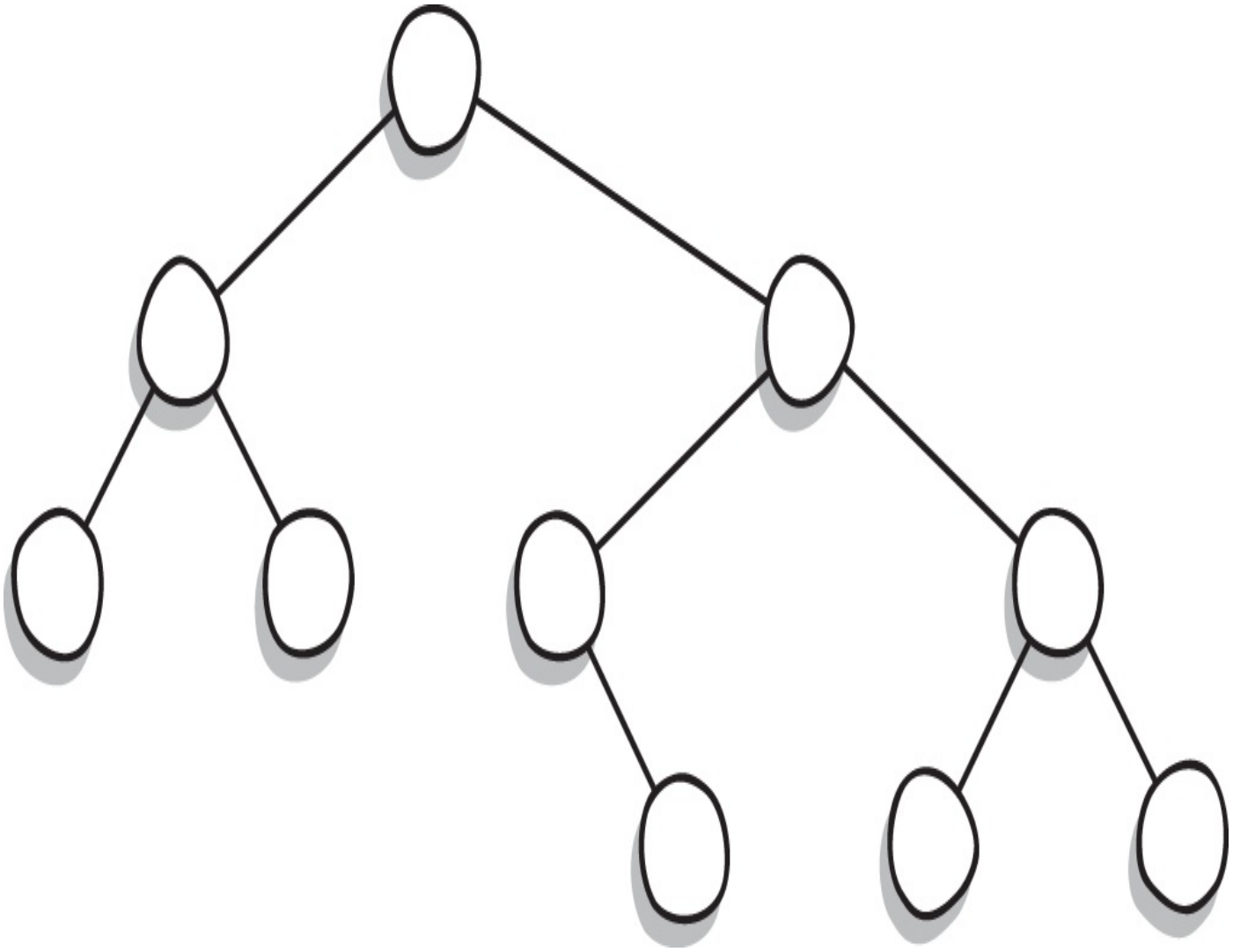


- ✔ **Datenmodell:** Objekte (Knoten), die in einer binären Relation (Kanten) zueinander stehen
- ✔ **Datenabstraktion:** Menge von Knoten und Kanten, die erweitert und inspiziert werden kann (»Knoten und Kanten hinzufügen«, »alle Nachbarn eines Knotens feststellen« und so weiter)
- ✔ **Datenstruktur:**
  - Adjazenzlisten und Adjazenzmatritzen
  - Suchbäume und Hashtabellen

Das Datenmodell *Graph* ist universell, die Datenabstraktion *Graph* ist eher unüblich. Keine der gängigen Programmiersprachen hat bisher Graphen in ihre Standardbibliothek oder gar in ihren Sprachkern aufgenommen. Allerdings gibt es etliche Drittanbieter, die Graphen als Datenabstraktion (*Graphenbibliotheken*) anbieten. Neuerdings gewinnen

Graphendatenbanken an Popularität. Es handelt sich dabei um Datenbanken, die auf Graphen statt auf Relationen als Datenmodell basieren.

Baum



- ✔ **Datenmodell:** Sonderfall eines Graphen: hierarchisch strukturierte Informationen
- ✔ **Datenabstraktion:** unüblich
- ✔ **Datenstruktur:**
  - hierarchisch verkettete Strukturen, algebraische Datentypen
  - Suchbäume

Bäume sind das Datenmodell für alle hierarchisch organisierten Informationen. Algebraische Datentypen könnte man als Datenabstraktionen betrachten. Sie verbergen ja die konkrete Implementierung der Strukturen. Allerdings sind sie zu schwach, um wirklich als Datenabstraktion akzeptabel zu sein.

Die bekannteste Datenabstraktion für Bäume ist sicher das *Document Object Model*: Eine Schnittstelle, um durch die hierarchische Struktur von XML- und HTML-Dokumente zu navigieren und sie zu modifizieren. In manchen GUI-Bibliotheken finden sich ebenfalls Komponenten, mit denen über eine Schnittstelle auf baumartige Strukturen zugegriffen werden kann, ohne in direkten Kontakt mit deren interner Implementierung zu kommen. Datenabstraktionen für Bäume sind ansonsten aber unüblich.

Unter den Datenstrukturen gehören Bäume dagegen zu den unermüdlichsten Arbeitstieren, die an fast allen Stellen zu finden sind.

Kapitel 16

Zehn Ratschläge, wenn (bevor) der kleine Frust kommt

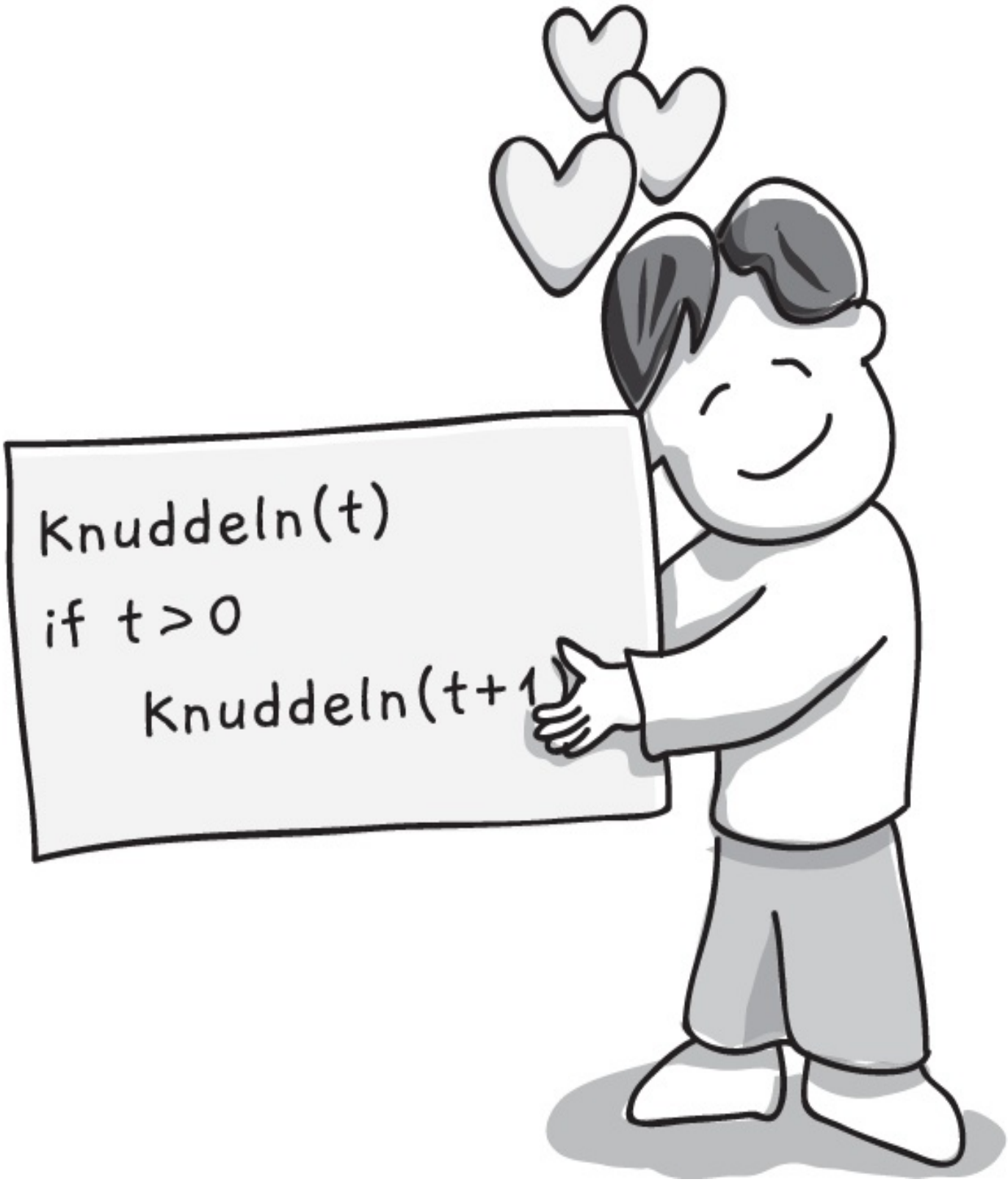
IN DIESEM KAPITEL

Zehn weise Ratschläge gegen den Frust

Wenn man sich darauf einlässt, dann macht es Spaß. Das gilt für viele Dinge, auch für Algorithmen und Datenstrukturen. Trotzdem kann es passieren, dass es mal nicht so spaßig ist. In solchen Situationen kommen dann die zu Wort, die sich für Realisten halten: »Das Leben ist kein Ponyhof!« oder »Arbeit ist Arbeit und wird immer Arbeit bleiben!«.

Na prima, aber das hilft genauso wenig weiter, wie der Enthusiasmus der ewig Begeisterten. Was hilft, ist eine Analyse der Ursache des Missvergnügens. Oft sind ein Mangel an Geduld, ein Missverständnis oder eine Fehleinschätzung die Ursache des Frustes. Wir haben hier mal die zehn wichtigsten Spaßbremsen und die Maßnahmen zu ihrer Beseitigung aufgelistet.

Rekursion ist deine Freundin



Aber Freundschaften müssen eingegangen und gepflegt werden.

Die Schleife ist die kleine Schwester der Rekursion, ein unreifes Ding das schnell laufen kann, aber wenig Lebenserfahrung hat. Es ist entspannt mit ihr, Sie müssen sich aber irgendwann auch auf die Rekursion einlassen, die strahlende Königin auf dem Ball der Algorithmen. Das fängt erst einmal mit Übung und Gewöhnung an und geht mit mehr

Übung weiter.

Am einfachsten ist die strukturelle Rekursion, bei der die Struktur des Algorithmus der (rekursiven) Struktur der Daten folgt. Listen mit ihrer rekursiven Definition als erstem Element gefolgt vom Rest oder Bäume als Knoten mit zwei Unterbäumen eignen sich hervorragend als Übungsmaterial.

Zur Vertiefung des Verständnisses der Rekursion ist die Umformung von Schleifen in rekursive Funktionen und umgekehrt bestens geeignet. Im Abschnitt *Funktion folgt Struktur* in Kapitel 3 werden diese Themen behandelt. Aber nur lesen reicht nicht. Selber machen macht schlau.

Wenn diese Hürde gemeistert ist, dann können Sie sich an fortgeschrittenere rekursive Algorithmen wagen. *Teilen und Herrschen* beziehungsweise *Reduzieren und Herrschen* sind hier die Schlagworte.

Mathematik ist einfach



Aber natürlich nur, wenn es sich um einfache Mathematik handelt.

Glücklicherweise braucht man für Algorithmen und Datenstrukturen in aller Regel nur einfache Mathematik. Ein wenig Kombinatorik hilft auch. Zunächst sollte man die mathematische Notation beherrschen. Speziell die in der elementaren Mengenlehre verwendeten Symbole sollten nicht als magische Zeichen erscheinen. Auch hier hilft üben und es schadet nicht, wenn man mal bei Wikipedia nachschaut oder jemanden fragt, der sich mit der Sache auskennt.

Lassen Sie sich nicht auf die Ausrede ein, dass schwierige Mathematik Ihnen das Leben schwer macht, schon gar nicht, wenn diese Ausrede von Ihnen selbst kommt.

Pseudocode ist verstehbar

Aber nur, wenn man sich auf ihn einlässt und sich mit ihm beschäftigt.

Pseudocode ist keine Programmiersprache und schon gar keine besonders einfache. Er richtet sich an menschliche Leser. Das hat den Vorteil, dass viele Dinge nicht explizit ausformuliert werden müssen, sondern sich aus dem Kontext ergeben. Das macht die Kommunikation effektiv. Natürlich muss man sich erst einmal daran gewöhnen und es ergibt sich nicht immer alles aus dem Kontext. Manchmal versteht man den Kontext auch anders als andere – in dem Fall hilft es, die Unklarheiten zu thematisieren. Auch gegenüber sich selbst. Nur die Unklarheiten können geklärt werden, derer man sich bewusst ist.

Es gibt eine sehr gute Methode, mit der Missverständnisse und Unklarheiten aufgedeckt werden können: Das ist die Implementierung von Pseudocode in einer oder besser noch in verschiedenen Programmiersprachen. Wie bei jeder Fremdsprache hilft nur das intensive Üben. Widerstehen Sie der Versuchung, fertige Programme aus dem Internet zu übernehmen. Nehmen Sie stattdessen einen in Pseudocode beschrieben Algorithmus und übersetzen Sie ihn in ein Programm. Fangen Sie mit kleinen und einfachen Algorithmen an und steigern Sie sich dann langsam, aber gewaltig …

Trauen Sie sich auch, die Rückrichtung zu gehen. Schreiben Sie Pseudocode hin. Mit der Hand, ohne Tastatur. Ja das geht: mit einem Stift auf Papier. Wenn Sie kein Papier haben, ist ein Tablett-Computer mit Stift auch Ok.

Abstraktion ist gut

Manche verbinden den Begriff *Abstraktion* mit *unverständlicher, trockener Theorie* und ducken sich schon ängstlich, wenn sie das Wort nur hören. Tatsächlich geht es bei einer Abstraktion aber immer um eine Vereinfachung. Dinge werden vereinfacht, indem sie auf ihren wesentlichen Kern reduziert werden.

Die realen Dinge, mit denen wir es zu tun haben, sind immer die zu verarbeitenden Daten. Manches an diesen Daten ist wesentlich und wichtig, einiges andere ist unwesentlich. Wenn das Unwesentliche entfernt ist und nur noch das Wesentliche bleibt, dann ist das keine Datenmasse ohne jede Struktur. Es bleiben immer noch genügend Beziehungen zwischen den Einzeldaten. Diese Beziehungen machen das Datenmodell aus. Es ist letztlich Ihre Entscheidung, was das geeignete Datenmodell ist: Eine Liste, ein Baum, ein Graph, eine Relation, eine Menge. Ein absolutes Richtig oder Falsch gibt es da nicht. Abhängig von den Daten und deren geplanter Verarbeitung gibt es ein *passt gut* oder *passt nicht so gut*.

Auch das ist eine Sache der Übung und Erfahrung. Modellieren Sie einfach mal eine »Datenlage« als Liste, Baum oder Graph und dann bemerken Sie schon, wie das passt oder auch nicht.

Sei auch mal funktional

Aber vergiss oder missachte das Imperative nicht.

Viele Algorithmen werden mit Pseudocode ausgedrückt, der Notationen nutzt, die man als Programmierer funktionalen Programmiersprachen zuordnen würde. Ist man mit diesem Programmierstil nicht vertraut, dann kann das zu Verwirrung und Frustration führen. Die Dinge sind aber in aller Regel leicht verständlich und wiederholen sich. Man kann sie lernen.

Taucht im Pseudocode so etwas auf wie

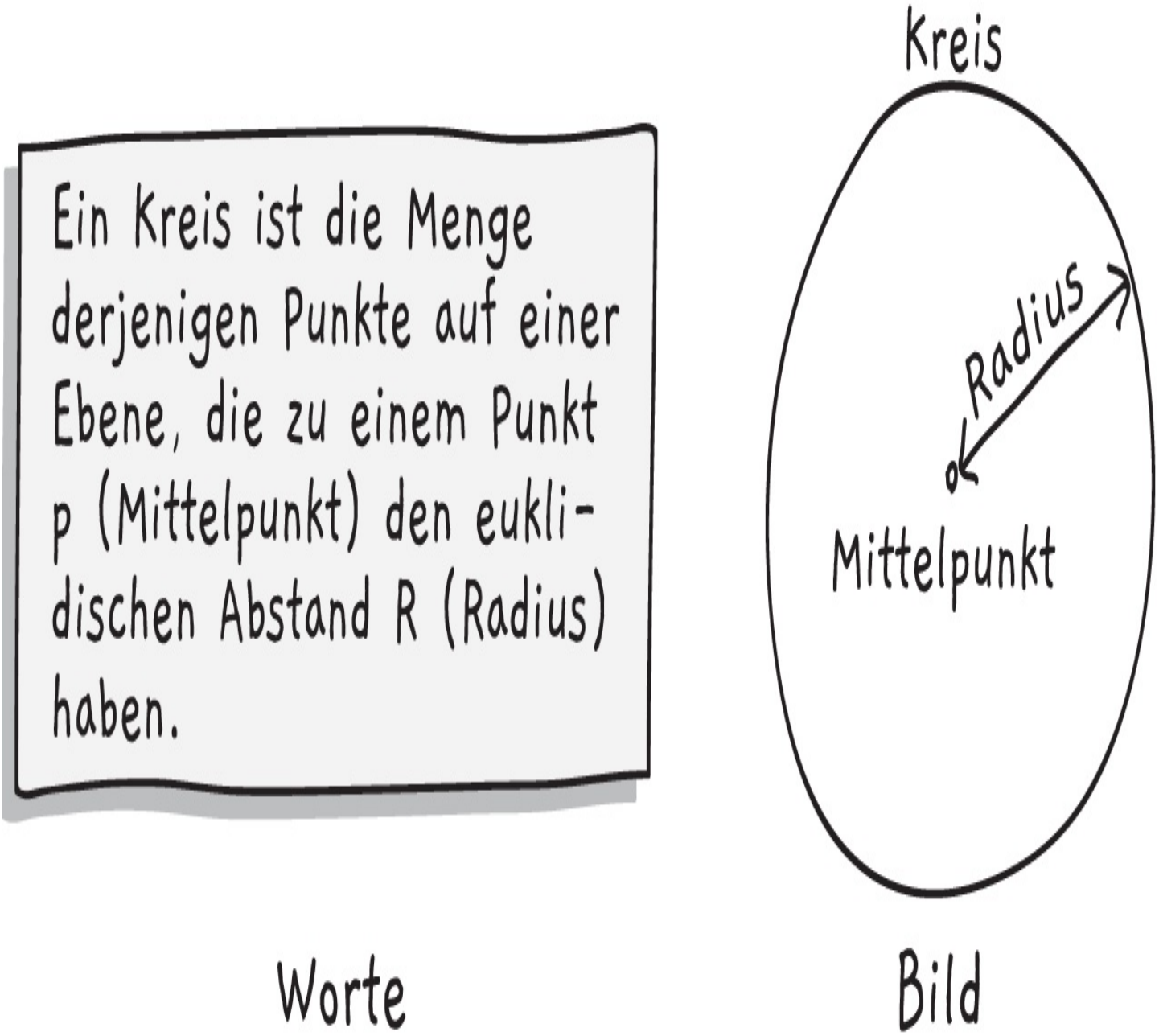
$$min\{c(m - d_j) + 1 \mid \text{ für alle } j \text{ mit } d_j < m\}$$

dann gibt es dafür nicht nur eine leichte Erklärung, sie kann auch von allen verstanden werden.

Entwickler mit Erfahrungen in funktionaler Programmierung sind mit solchen Konstrukten vielleicht etwas vertrauter als andere, aber das heißt nicht, dass dieser Pseudocode mithilfe funktionaler Sprachkonstrukte realisiert werden soll oder muss. Er sagt nichts über irgendwelche Sprachkonstrukte aus, sondern nur das, was tatsächlich zu lesen ist: »Für jedes j , für das der Wert d_j kleiner als m ist, berechne $c(m - d_j)$ und von all diesen Werten nimm den kleinsten.« Das kann man in jeder Programmiersprachen implementieren.

Hat man Probleme mit dieser Ausdrucksweise und ist mehr der handwerkliche Typ, der die CPU gerne mal zum Glühen bringt, dann kann eine kurze Beschäftigung mit einer funktionalen Programmiersprache aber schon mal zum Klick verhelfen.

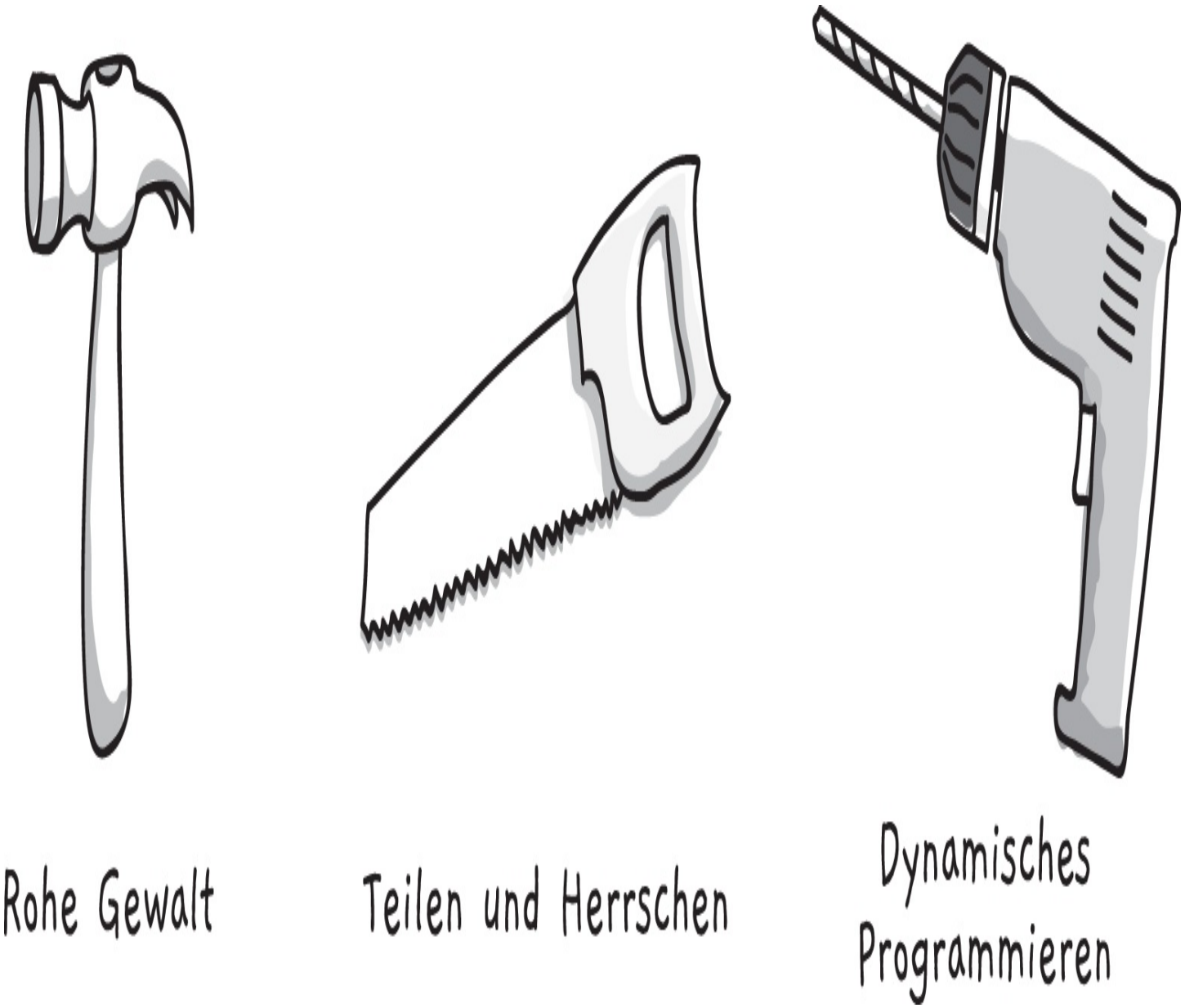
Ein Bild sagt mehr als 1000 Worte



Illustrieren Sie Ihre Ideen mit Zeichnungen. Datenstrukturen können nur visuell verstanden werden. Auch hier ist Handarbeit angesagt: Papier und Bleistift oder noch besser Tafel und Kreide oder irgendwelche elektronische Simulationen von Tafel und Kreide. Sie können können Ihre Ideen dann viel besser mit anderen diskutieren, und das ist natürlich immer eine gute Idee.

Das Bildliche muss auch nicht auf Datenstrukturen beschränkt bleiben. Viele Algorithmen sind nicht übermäßig komplex, ihr Ablauf kann nachgespielt werden, in der Fantasie, auf Papier oder in der Wirklichkeit.

Vieles ist solides Handwerk



Aber es geht nicht (nur) um Tricks.

Einige Algorithmen sind völlig klar und einfach. Andere sind höchst trickreich, man kann sie verstehen, ja, aber ganz offen, das wäre uns nicht eingefallen. Das ist kein Beinbruch, die wenigsten von uns sind ein neuer Dijkstra, Hoare oder Knuth. Und selbst die haben ihre Algorithmen nur selten über Nacht entwickelt. Manchmal hat es sogar Jahre gedauert, bis ein Algorithmus in seiner endgültigen korrekten und eleganten Fassung präsentiert werden konnte.

Man kann also auch schon einmal einige Tage mit einem Algorithmus verbringen, ohne sich dessen schämen zu müssen. Die Arbeit an einem Algorithmus muss auch nicht mit dem ersten erfolgreichen Test beendet werden. Überlegen Sie sich Argumente zur Korrektheit, warum tut er das, was er tun soll? Beginnen Sie dabei mit kleinen und einfachen Problemen. Dann sollte er schnell sein. Da helfen Experimente und schließlich sollte er noch elegant präsentiert werden.

Es geht auch um Kreativität

Es gibt Dinge, die muss man einfach kennen. SelektionSort, InsertionSort, QuickSort, MergeSort muss man kennen. Warteschlagen, Stapel, Hashtabellen, Bäume muss man kennen.

Das ist das kleine Einmaleins. Interessant wird es, wenn Kreativität gefragt ist. Kreativität ist das Thema der algorithmischen Techniken. Nehmen Sie diese Thematik ernst! Fragen Sie sich bei gegebenen Problem explizit, ob eine Lösung möglich ist, die auf *roher Kraft*, *Gier*, *Teilen und Herrschen* oder *dynamische Programmierung* basiert und wie sie aussehen würde. Die Probleme müssen dazu nicht kompliziert sein. Auch hier gilt: üben hilft.

Unterscheide Datenmodell und Datenstruktur

Das Datenmodell beschreibt die Struktur der Daten aus der Sicht der Anwendung. Eine Datenstruktur ist die Organisation der Daten im Speicher. Ein Haufen zusammenhangloser Fakten ist eine Menge (Datenmodell). Man speichert sie in einer Liste, einem Array, einer Hashtabelle oder einem Baum. Das sind Datenstrukturen.

An diesen Unterschied muss man sich als Programmierer erst einmal gewöhnen. Am besten geht das durch Übung. Dazu kann man einfach mal den Blickwinkel wechseln und ein Gedankenexperiment wagen. Ein Graph ist einem Menge von Knoten und Kanten. Kanten sind Paare von Knoten. Wie wäre es, wenn die Knoten in einem Baum und die Kanten in einer Hashtabelle stecken? Oder umgekehrt?

Was schwierig aussieht, ist oft auch schwierig

Wenn ein Programm vom Compiler nicht akzeptiert wird oder nicht die erwartete Ausgabe liefert, dann hat man etwas falsch gemacht. Diese Eindeutigkeit fehlt bei Pseudocode und ganz allgemein scheint es in dem ganzen Fach manchmal einfach an Eindeutigkeit zu fehlen. Das liegt nicht immer nur an der eigenen Begriffsstutzigkeit. Vieles ist tatsächlich schwierig, mehrdeutig und voller Querbezüge.

Die Präzisierung des Begriffs *Algorithmus*, also die Frage, was man sagen darf, wenn man einen tatsächlich ausführbaren Algorithmus beschreibt, war beispielsweise am Anfang des zwanzigsten Jahrhundert ein Thema, das die schlausten Vertreter der zukünftigen Informatik intensiv diskutierten, und aus dem die Theorie der Berechenbarkeit und

Komplexität entstanden ist. Man muss sich also keineswegs wundern, wenn man nicht sofort versteht, was in Pseudocode gesagt werden darf und was nicht.

Problematisch ist auch der beständige Wechsel der Abstraktionsstufe sowie der Perspektive. Eine »Liste« – ist das eine verkettete Liste mit Vor- und Rückwärtszeigern, ein algebraischer Datentyp mit erstem Element und Rest, eine Datenabstraktion, hinter der eine Trie-Variante steht oder eine ArrayList wie in Java oder Python, oder tatsächlich ein Array, oder, oder … Programmierer haben da kein Problem. Es gibt ihre Sprache und eventuell noch eine API und das definiert eine genaue Abstraktionsstufe, auf der alles konstruiert wird.

Man muss sich also nicht wundern, wenn man nicht jeden Wechsel der Perspektive und des Abstraktionsgrads gleich mitbekommt. Da hilft am besten ein herzhaftes »Halt, wovon genau ist gerade die Rede?«

Stichwortverzeichnis

$\mathcal{O}$ (Landau-Symbol) 1

Ω (Landau-Symbol) 1

ω (Landau-Symbol) 1

ε-Approximation 1

← (Zuweisung) 1

Θ (Landau-Symbol) 1

π (Kreiszahl pi) 1, 2

… (Notation) 1, 2

:: (Notation) 1, 2, 3

= (Gleichheit) 1

A

A*-Algorithmus 1–2

Abbruchbedingung 1

Abschätzung 1

Adjazenz 1

Adjazenzliste 1, 2

Adjazenzmatrix 1, 2, 3

Adresse 1

ADT (abstrakter Datentyp) 1, 2

    und alg. DT 1

ägyptische Multiplikation 1, 2

Akkumulator 1, 2

Akkumulatortechnik 1, 2

alg. DT (algebraischer Datentyp) 1, 2, 3, 4, 5
    für Ausdrücke 1
    für Listen 1
    und ADT (abstrakter Datentyp) 1
    und funktionale Programmierung 1
    und veränderliche Strukturen 1
    und Zeiger 1, 2
    und zyklische Strukturen 1
Algorithmen 1–2
    Effizienz 1
    funktional 1
    imperativ 1
    Korrektheit 1–2
    randomisiert 1–2
algorithmische Technik
    Übersicht 1
    Approximation 1–2
    dynamisches Programmieren 1–2
    Gier 1
    Teilen und Herrschen 1–2
Alignment 1–2, 3–4
Amortisierung 1, 2–3, 4, 5
Append 1
Approximation 1–2
    heuristisch 1–2
    NP-schwere Probleme 1
    Rucksackproblem 1
Approximationsschema 1, 2
Array 1, 2
    assoziativ 1
    sortiert 1

asymptotische Laufzeit 1
Ausdrucksbaum 1
Auswahl 1
Auswahlproblem 1
average case 1, 2
azyklisch 1

B

Bachelortheorem 1
Backtrace 1–2
Backtracking 1–2, 3, 4–5
Balancierung 1
Baum 1, 2
    (2,4)- 1
    als Modell hierarchischer Strukturen 1
    Ausdrucks- 1
    ausgeglichen 1
    AVL 1
    B-Baum 1
    balanciert 1
    binär *siehe* Binärbaum
    Datenmodell 1
    entartet 1
    Gewicht 1
    Höhe 1
    k-balanciert 1
    Rot-Schwarz- 1
    Rotation 1
    Tiefe 1
    Traversierung 1–2
bedingte Anweisung 1

Berechenbarkeit 1, 2
Berechnungsproblem 1, 2
best case 1
Beweis 1–2
Binärbaum 1, 2
    als alg. DT 1
    Anzahl der Blätter 1
    im Array gespeichert 1
Binärsuche 1–2, 3
    Interpolationssuche 1–2
Bottom-up 1
branch and bound *siehe* Verzweigen und Begrenzen
break 1
Breitensuche 1
brute force *siehe* rohe Kraft 1
BubbleSort 1
Bucket 1

C

case 1
Collatz-Algorithmus 1
Cons 1
CountingSort 1–2

D

Datenabstraktion 1, 2, 3, 4–5
    funktional 1, 2, 3, 4
    imperativ 1, 2
Datenmodell 1
    Liste 1

Datenstruktur 1
atomar 1
persistent 1–2, 3
verkettet 1, 2
zusammenhängend 1
Datentyp
abstrakt 1
algebraischer *siehe* alg. DT
Dictionary 1
Dijkstras Algorithmus 1–2, 3–4
mit Heuristik *siehe* A*-Algorithmus
Disjoint Set 1, 2
divide and conquer *siehe* Teilen und Herrschen
divide et impera *siehe* Teilen und Herrschen
DP *siehe* dynamisches Programmieren
Dreiecksungleichung 1
dynamisches Programmieren 1–2
Definition 1
Fibonacci-Zahlen 1
Maximale-Teilsumme-Problem 1
Minimum 1, 2, 3
Rucksackproblem 1–2, 3
und Teilen und Herrschen 1

E

Editierdistanz 1
Effizienz 1
Eimer 1
Ellipse 1
else 1
Endrekursion 1

Entscheidungsbaum 1
Entscheidungsproblem 1, 2, 3
erschöpfende Suche 1, 2–3, 4
Erwartungswert 1
euklidische Distanz 1
experimentelle Algorithmik 1
exponentielle Laufzeit 1, 2

F

Fädelung 1
Fakultät 1
Faustregeln *siehe* Heuristik
Fibonacci-Zahlen 1
for 1
For-Comprehension 1

G

gaußsche Summenformel 1
gemischte Strategie 1
geometrische Reihe 1
GGT (Größter Gemeinsamer Teiler) 1–2
Gier 1–2
    Maximale-Teilsumme-Problem 1
    Rucksackproblem 1–2
    Traveling-Salesman-Problem 1
Gleichheit 1, 2, 3
GnomeSort 1

Graph 1, 2, 3
azyklisch 1
Datenstruktur 1–2
gerichtet 1
gewichtet 1, 2
markiert 1, 2
Traversierung 1
ungerichtet 1
verbunden 1
Groß-O-Notation 1–2

H

Halteproblem 1
Handlungsreisender *siehe* Traveling-Salesman-Problem
Hasen *siehe* Kaninchen
Hashfunktion 1, 2, 3
und Gleichheit 1
Hashing 1–2
universell 1
Hashtabelle 1, 2
Heap 1, 2–3
extractMin 1
push 1
sortieren 1
und Prioritätswarteschlangen 1
HeapSort 1–2
Heuristik 1–2
approximativ 1–2
gierig 1
lokale Suche 1
zur Laufzeitbestimmung 1–2

Huffman-Codierung 1

I

if 1
ILP (Integer Linear Programming)
siehe lineare Optimierung
In-place-Sortierung 1
inkrementell 1
InsertionSort 1, 2, 3, 4, 5
Interpolationsgerade 1
Interpolationssuche 1–2
Irrgarten 1
Iterator 1
    Baum 1–2
    Breitentraversierung 1
    und alg. DT 1

K

Kadanes Algorithmus 1, 2
Kaninchen
    Bruno 1
    Egon 1, 2, 3, 4, 5
    gierig 1
    Harvey 1, 2, 3
    im Eimer 1
    Tierschutz 1
    vampirisch 1
    Vermehrung 1
    Zuchtverein 1, 2

Kante 1
gewichtet 1
Markierung 1, 2
parallel 1
Knoten 1
Knotenüberdeckungsproblem 1–2
Lösung für Bäume 1
Kollision 1
kombinatorische Objekte 1
Komplexitätsklassen 1–2
NP 1
P 1
konstante Laufzeit 1, 2
Konstruktionsheuristik 1
kontrahierende Summe 1
Korrektheit 1–2
Beweis 1–2
Kreiszahl π 1, 2
Kruskals Algorithmus 1, 2
kubische Laufzeit 1, 2

L

Labyrinth 1
Landau-Symbole 1

Laufzeit 1
  abschätzen 1
  amortisiert 1, 2–3, 4
  asymptotisch 1
  Average-Case- 1
  berechnen 1
  Best-Case- 1
  bestimmen 1–2
  experimentell bestimmen 1
  exponentiell 1, 2
  Faustregeln 1–2
  konstant 1, 2
  kubisch 1, 2
  linear 1, 2–3, 4
  logarithmisch 1
  mittlere 1
  obere Schranke 1
  polynomiell 1, 2
  quadratisch 1, 2, 3
  sublinear 1
  superpolynomiell 1
  Vergleich 1–2
  Worst-Case- 1
Laufzeitanalyse 1
  randomisiert 1–2
Laufzeitklassen 1–2
leichtester Bauarbeiter 1
Levenshtein-Distanz *siehe* Editierdistanz
Lexikon
  Binärsuche 1
  Interpolationssuche 1

lineare Laufzeit *siehe* Linearzeit
lineare Optimierung 1–2
    ganzzahlig 1
Lineare Programmierung (LP) *siehe* lineare Optimierung
lineare Suche 1
Linearzeit 1, 2–3, 4
    Sortierverfahren 1–2
Liste 1, 2
    als Datenmodell 1
    verkettet 1
    Verknüpfung 1
Listenausdrücke 1
logarithmische Laufzeit 1
Logarithmus 1, 2, 3, 4
lokale Suche 1
lokales Maximum 1
Luftlinie *siehe* euklidische Distanz

M

Markierung 1
Maschinenmodell 1
Mastertheorem 1, 2–3
    Bachelortheorem 1
    Fall 1 1, 2
        Beispiel 1
    Fall 2 1, 2, 3
    Fall 3 1, 2
Matrixmultiplikation 1–2

Maximale-Teilsumme-Problem 1–2
    Definition 1
    dynamisches Programmieren 1
    Gier 1
    Kadanes Algorithmus 1, 2
    rohe Kraft 1
    Teilen und Herrschen 1
Maximum *siehe auch* Minimum mit DP
    lokal 1
Memoisation 1
Menge 1, 2, 3, 4
    als alg. DT 1
    als charakteristischer Vektor 1
    als Datenabstraktion 1, 2
    als Datenstruktur 1
    als funktionale Datenabstraktion 1
    als imperative Datenabstraktion 1
    Disjunktion 1
    Partition 1
Merge-Algorithmus 1–2
MergeSort 1–2, 3–4
minimaler Punktabstand 1–2
Minimum
    mit DP 1, 2, 3
    mit roher Gewalt 1
Modellmaschine 1
Modellwelten 1
Multiplikation
    ägyptische 1, 2
    Matrizen 1–2

N

Needleman-Wunsch-Algorithmus 1, 2

new 1

Nichtberechenbarkeit 1

Nichtdeterminismus 1–2

Nil 1, 2

Notation, 1, 2, 3

- ← (Zuweisung) 1
- … 1, 2
- break 1
- case 1
- else 1
- for 1, 2
- if 1
- new 1
- null 1
- return 1
- while 1

NP (Komplexitätsklasse) 1

NP-schwer 1–2, 3

- Lösung 1–2
- Liste von Problemen 1

NP-vollständig 1

null 1

O

Omega 1

Optimierungsproblem 1, 2–3, 4, 5, 6, 7
Definition 1
ILP 1
lokales Maximum 1
Orakel 1
Ordnungsrelation 1

P

P (Komplexitätsklasse) 1
Partition 1, 2
Partitionsalgorithmus 1–2
randomisiert 1
Permutation 1, 2, 3, 4
PermutationSort 1
Persistenz 1–2, 3, 4, 5, 6
Pfad 1
kürzester 1
Pivot 1, 2
polynomielle Laufzeit 1
Postorder 1
Prioritätswarteschlange *siehe* Priority Queue
Priority Queue 1, 2, 3, 4
mit Heap 1
Problem
Berechnungsproblem 1
Entscheidungsproblem 1
Lösung 1–2
NP-schwer 1–2
NP-vollständig 1
Optimierungsproblem 1
Probleminstanz 1

Pseudocode 1–2

Q

quadratische Laufzeit 1, 2
QuickSort 1, 2, 3, 4–5
    Partitionsalgorithmus 1–2
    randomisiert 1

R

RadixSort 1–2
Random Access Machine (RAM) 1, 2, 3, 4, 5
    Eigenschaften 1
Randomisierung 1–2
    beim Sortieren 1
Rechenzeit *siehe* Laufzeit
Record 1, 2
    und Datenabstraktion 1
Reduzieren und Herrschen 1, 2, 3, 4, 5, 6
Rekursion
    Definition 1
    End- 1, 2, 3, 4, 5, 6
    linear 1, 2
    strukturelle 1–2, 3, 4, 5
    Suchraumerzeugung 1
    verstehen 1
Rekursionsgleichung 1
Relation 1, 2
Ressourcen von Algorithmen 1
return 1
rohe Gewalt *siehe* rohe Kraft

rohe Kraft 1, 2–3
Berechnung des Minimums 1
inkrementell 1
Maximale-Teilsumme-Problem 1
Rucksackproblem 1–2
Rotation
links 1
rechts 1
Rucksackproblem 1–2
Anwendungen 1
Approximation, 1–2
Definition 1
dynamisches Programmieren 1
mit Backtracking 1–2
mit branch and bound 1–2
mit brute force 1–2
mit dynamischem Programmieren 1–2
mit Gier 1–2

S

Schlüssel 1
Schleifen 1
geschachtelte 1
in Graphen 1
Schleifeninvariante 1
Schwarzhöhe 1
SelectionSort 1–2
Sequenzalignment *siehe* Stringalignment
SortByDigit 1–2
Sortierproblem 1
Sortierschlüssel 1, 2

Sortierverfahren 1–2
Übersicht 1
BubbleSort 1
CountingSort 1–2
GnomeSort 1
HeapSort 1–2
in Linearzeit 1–2
In-place- 1
InsertionSort 1
MergeSort 1–2, 3–4
mit Teilen und Herrschen 1–2, 3–4
obere Laufzeitschranke 1–2
PriorityQueueSort 1
QuickSort 1–2
RadixSort 1–2
SelectionSort 1–2
SortByDigit 1–2
stabil 1
topologisch 1
vergleichsbasiert 1
Spannbaum 1, 2
minimal 1, 2, 3
stabiles Sortieren 1
Stapel 1, 2, 3, 4
als Datenabstraktion 1
Strassens Algorithmus 1–2
Strategie, heuristisch 1
Stringalignment 1–2, 3–4

Struktur
durchlaufen 1
verkettet 1
zyklisch 1
sublineare Laufzeit 1
Suchbaum 1, 2, 3, 4
binärer 1, 2
Suche
binär 1–2
linear 1
Suchraum 1
Summenformel 1
superpolynomiell 1

T

Tabelle 1
Teilen und Herrschen 1, 2, 3, 4, 5, 6–7
Definition 1
Interpolationssuche 1
Laufzeit 1–2
Matrixmultiplikation 1–2
Maximale-Teilsumme-Problem 1
minimaler Punktabstand 1–2
Sortieren 1–2, 3–4
und dynamisches Programmieren 1
verstehen 1
Theta (Landau-Symbol) 1
Top-down 1
topologische Sortierung 1
Traceback *siehe* Backtrace
Transitivität 1

Traveling-Salesman-Problem 1
Approximation mit Gütegarantie 1
lokale Suche 1
Traversierung 1, 2
abstrakte Kollektionen 1
Breiten- 1–2
extern 1
extern und intern 1
Graph 1–2
intern 1
Liste 1
mit einem Iterator 1
Postorder- 1
Strategien 1
Tiefen- 1
Trichotomie 1
Trie 1–2
und Persistenz 1
TSP *siehe* Traveling-Salesman-Problem
Tupel 1, 2
Erzeugung 1
Turing-mächtig *siehe* Turing-vollständig
Turing-vollständig 1, 2
Turingmaschine 1, 2
Typ
algebraischer Datentyp 1
Zeiger 1

U

unsichtbare Hand 1

V

Verbesserungsheuristik 1

Vergleiche 1, 2

vergleichsbasierte Sortierverfahren 1

Vertex *siehe* Knoten

Vertex Cover *siehe* Knotenüberdeckungsproblem

Verzeichnis 1, 2, 3

- als Trie 1

Verzweigen und Begrenzen 1, 2–3

- Heuristik 1–2
- Rucksackproblem 1–2

W

Warteschlange 1, 2, 3, 4

- in einem Heap 1
- Prioritäts- 1, 2, 3, 4, 5

Wechselgeldproblem 1

while 1

worst case 1

Wurzelbaum 1

Z

Zeiger 1, 2

- auf Zeiger 1
- null- 1, 2

Zeit *siehe* Laufzeit

Zufall 1

Zustandsraum 1

Zuweisung 1

Zykel 1

WILEY END USER LICENSE AGREEMENT

Besuchen Sie www.wiley.com/go/eula, um Wiley's E-Book-EULA einzusehen.

LERNEN EINFACH GEMACHT



Algorithmen und Datenstrukturen

für dummies®



Mit Algorithmen und Datenstrukturen programmieren

Typische Probleme lösen

Algorithmische Ideen als Pseudocode dokumentieren

Andreas Gogol-Döring
Thomas Letschert