Algorithmen und Datenstrukturen

Michael Müller

27. November 2012

Inhaltsverzeichnis

1 Grundlagen und Begriffe

Der Begriff **Algorithmus** ist auf den arabischen Mathematiker *Mohammed Al-Chwarizmi* (783-850) zurückzuführen, dessen Buch später (in lateinischer Übersetzung) mit den Worten

Dixit Algorismi...

(*Es sprach Algorismi...*) begann. (Ein weiteres mathematisches Werk heißt *Hisab al-jabr walmuquabala*, aus dem sich das Wort *Algebra* entwickelte.)
Die historische Entwicklung kann grob durch folgende Punkte erläutert werden:

- Algorithmus von Euklid (300 v.Chr.)
- Sieb des Eratosthenes (200 v.Chr)
- Al-Chwarizmi (800 n.Chr.)
- Adam Ries, Buch über Grundrechenarten (1542)
- Gerolamo Cardano, algebraische Algorithmen zur Lösung von Gleichungen (1545)
- Gottfried W. Leibniz (1646-1716), binäres Zahlensystem, Einsicht in das Wesen eines Algorithmus, beispielsweise Unterscheidung von Erzeugungs- und Entscheidungsverfahren

Danach Formalisierung der logischen und mathematischen Grundlagen (Entwicklung eines *Logikkalküls*) durch

- George Boole (1815-1864)
- Gottlob Frege (1848-1925)
- Bertrand Russell (1872-1970)
- Alfred North Whitehead (1861-1947)

(und anderen). Desweiteren ist das von David Hilbert (1862-1943) entwickelte Axiomensystem (der Euklidschen Geometrie) zu erwähnen, in deren Zusammenhang das *allgemeine Entscheidungsproblem* formuliert wurde (*kann entschieden werden, ob eine Aussage aus einem Axiomensystem geschlossen werden kann oder nicht*). Der Nachweis, dass ein solches System widerspruchsfrei ist, existiert nicht. Dies wurde von Kurt Gödel (1906-1978) bewiesen. Ferner hat diese Erkenntnis elementare Auswirkungen auf die Theorie der Algorithmen.
Eine weitere Präzisierung des Algorithmenbegriffs erfolgte durch

- Andrej Markov (1856-1922)

Eine Beschreibung durch automatisch arbeitende Maschinen erfolgte durch

- Alan Mathison Turing (1912-1954)

Seit 1965 werden Laufzeitverhalten/Speicherbedarf von Algorithmen innerhalb der *Komplexitätstheorie* beschrieben.

Im Jahr 2000 wurden von Editoren der Zeitschrift *Computing in Science and Engineering* zehn Algorithmen ausgewählt, die ihrer Ansicht nach die größte Bedeutung für Wissenschaft und Technik im 20. Jahrhundert hatten:

- Monte Carlo Methode (1946, auch Metropolis Algorithmus)
- Simplex-Verfahren (1947)
- Krylov-Verfahren zur Lösung linearer Gleichungssysteme (CG-Verfahren, 1950)
- Matrix-Faktorisierung für lineare Gleichungssysteme (LU-Zerlegung, 1951)
- Fortran Compiler (1957)
- QR-Verfahren zur Berechnung von Eigenwerten (1959/61)
- Quicksort-Algorithmus (1962)
- Schnelle Fourier-Transformation FFT (1965)
- Integer-Relation-Detection Algorithmus (1977)
- Schnelle Multipol-Methode FMM (1987)

Unter einem *Algorithmus* verstehen wir eine Folge von Rechenschritten, die eine Eingabe in eine Ausgabe umwandelt.

Eine exakte Definition dieses Begriffs werden wir hier nicht vorstellen, denn für unsere Zwecke ist nur ein intuitives Verständnis des Algorithmenbegriffs notwendig. Eine formale Definition eines Algorithmus kann mit Hilfe sogenannter *Turingmaschinen* (Alan Turing, 1912-1954) erfolgen. Diese Formalitäten müssen erst dann berücksichtigt werden, wenn ein Nachweis erbracht werden muß, dass zu einem bestimmten Problem **kein** Algorithmus existiert, der dieses Problem löst.

Beispiel 1.1 (Bestimmung Maximum)
Eingabe: Eine Menge von n natürlichen Zahlen $a_1, ..., a_n$.
Ausgabe: Das Maximum der Zahlen $a_1, ..., a_n$.
Vorgehensweise: Speichere a_1 auf einer Variablen max. Vergleiche der Reihe nach $a_2, ..., a_n$ mit max. Ist $a_i > max$ für ein $i \in \{2, ..., n\}$, so initialisiere $max = a_i$. Gebe am Ende max aus.

Für Algorithmen fordert man in der Regel folgende Eigenschaften:

- **Finitheit**
  Die Beschreibung des Algorithmus besitzt eine endliche Länge.
- **Ausführbarkeit**
  Jeder Schritt des Algorithmus muß ausführbar sein.
- **Platzkomplexität**
  Der Algorithmus darf in jedem Schritt nur endlich viel Speicherplatz benötigen
- **Terminierung**
  Der Algorithmus muss nach endlich vielen Schritten anhalten und ein Ergebnis liefern.

- **Determiniertheit**
  Der Algorithmus liefert bei gleichen Startvoraussetzungen stets das gleiche Ergebnis.
- **Determinismus**
  Zu jedem Zeitpunkt ist der nachfolgende Schritt des Algorithmus eindeutig festgelegt.

Der praktischen Umgang mit Problemenlösungen durch Algorithmen führt zur Betrachtung der folgenden Punkte:

(1) **Entwurf** eines Algorithmus zu einem gegebenen Problem

(2) Nachweis der **Korrektheit** des Algorithmus

(3) **Effizienz/Laufzeit** des Algorithmus

(4) **Darstellung/Notation** des Algorithmus

Bei der Entwicklung von Algorithmen können **Entwurfstechniken**, wie zum Beispiel *Divide and Conquer*, *Greedy-Strategien*, *dynamisches Programmieren* oder auch *Branch-and-Bound*-Methoden, zum Einsatz kommen. Diese Techniken weisen alle eine spezielle Charakteristik auf, die an Beispielen vor- bzw. gegenübergestellt werden.
Der Nachweis der **Korrektheit** eines Algorithmus erfordert meist tiefergehende mathematische Kenntnisse und Fähigkeiten. Oft spielen mathematische Verfahren wie das der vollständigen Induktion und auch kombinatorische Ansätze eine große Rolle. Wir werden Korrektheitsbeweise stets vernachlässigt behandeln, nur kurz schematisch darstellen oder auf andere Literatur verweisen.
Bei der Analyse der **Laufzeit** eines Algorithmus ist es zunächst notwendig, eine geeignete Schreibweise zu definieren. Mittels dieser Schreibweise kann das Laufzeitverhalten eines Algorithmus in Abhängigkeit der Eingabe angegeben werden.
Zur **Notation** eines Algorithmus kann man zunächst einen sogenannten **Pseudocode** verwenden. Ein Pseudocode ist kein übersetzbarer Quellcode, er wird aber in der Regel an Programmiersprachen wie beispielsweise C, C++, Java angelehnt. Weitere Darstellungsformen bzw. Diagramme werden wir zu einem späteren Zeitpunkt behandeln.

Bei der Verwendung von Daten innerhalb eines Algorithmus spielt die Speicherform für diese Daten eine bedeutende Rolle. Deshalb ist eine Beschreibung von Speicherformen durch sogenannte **Datenstrukturen** unerlässlich. Wir werden zu einem späteren Zeitpunkt Datenstrukturen wie Listen, Schlangen, Stapel, Bäume kennenlernen.

Zum Abschluss des Kapitels werden wir am Beispiel des Algorithmus von Euklid die praxisrelevanten Punkte *Entwurf, Korrektheit, Effizienz* und *Darstellung* aufzeigen:

Beispiel 1.2 Der Algorithmus von Euklid
(Euklid (360-280 v.Chr., Alexandria): griechischer Mathematiker, sein berühmtestes Werk *Die Elemente* trägt das mathematische Wissen der damaligen Zeit zusammen.)

Eingabe: natürliche Zahlen $0 < b < a$
Ausgabe: größter gemeinsamer Teiler $\mathrm{ggT}(a, b)$

```
EUKLID(a,b)

c=0
while b>0{
     c=a mod b
     a=b
     b=c
}  //Auf a ist ggT(a,b) gespeichert
```

Um die Korrektheit des Algorithmus nachzuweisen ist zunächst zu zeigen, dass der Algorithmus in endlich vielen Schritten ein Ergebnis liefert:
Durch

$$r_0 = a, \quad r_1 = b, \quad r_{n+1} = r_{n-1} \text{mod } r_n$$

wird eine Folge (r_n) nicht-negativer ganzer Zahlen definiert. Wegen

$$r_{n-1} = qr_n + r_{n+1}$$

mit $0 \leq r_{n+1} < r_n$ ist die Folge (r_n) **streng monoton fallend** und beschränkt. Aus diesem Grund terminiert der Algorithmus nach endlich vielen Schritten. Wegen

$$\text{ggT}(r_{n-1}, r_n) = \text{ggT}(qr_n + r_{n+1}, r_n) = \text{ggT}(r_{n+1}, r_n) = \text{ggT}(r_n, r_{n+1})$$

liefert der Algorithmus das richtige Ergebnis.
Um die Laufzeit des Algorithmus anzugeben, werden wir die Anzahl der Rechenoperationen in Abhängigkeit der Eingabegrößen nach oben hin abschätzen. Dazu nehmen wir an, dass der Algorithmus nach n Schritten terminiert:

$$\begin{aligned} r_0 &= q_1 r_1 + r_2 \\ r_1 &= q_2 r_2 + r_3 \\ &\ldots \\ r_{n-2} &= q_{n-1} r_{n-1} + r_n \\ r_{n-1} &= q_n r_n + r_{n+1} \end{aligned}$$

mit $0 < a < b$, $r_0 = a$, $r_1 = b$, $r_n = \text{ggT}(a, b)$ und $r_{n+1} = 0$. Bezeichnet man mit F_n die n-te Fibonacci-Zahl, so kann die Abschätzung

$$r_j \geq F_{n-j+2}$$

für $j = n, ..., 0$ durch vollständige Induktion nach j gezeigt werden:
Für $j = n$ und $j = n - 1$ ist die Behauptung klar. Außerdem ist

$$r_{j-1} = q_j r_j + r_{j+1} \geq r_j + r_{j+1} \geq F_{n-j+2} + F_{n-j+1} = F_{n-j+3}.$$

Damit gilt obige Abschätzung. Beachtet man, dass die k-te Fibonacci-Zahl (in geschlossener Form) durch

$$F_k = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^k - \left(\frac{1-\sqrt{5}}{2} \right)^k \right)$$

berechnet werden kann, so gilt unter Berücksichtigung $b = r_1 \geq F_{n+1}$:

$$\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n = F_n + \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n \leq F_n + 1 \leq F_{n+1} \leq b.$$

Es folgt

$$n \ln\left(\frac{1+\sqrt{5}}{2}\right) \leq \ln(b\sqrt{5})$$

und damit

$$n \leq c \ln(b\sqrt{5})$$

mit

$$c = \ln^{-1}\left(\frac{1+\sqrt{5}}{2}\right).$$

Dadurch ist die Laufzeit nach oben hin abgeschätzt. Insbesondere kann durch obigen Rechenweg gezeigt werden, dass zwei aufeinanderfolgende Fibonacci-Zahlen den *worst case* für den vorgestellten Algorithmus von Euklid bilden.

2 Darstellungsformen für Algorithmen

Ind diesem Kapitel werden wir die wichtigsten Darstellungsformen zur Visualisierung von Algorithmen vorstellen. Wir unterscheiden hierbei:

- sprachliche Formulierungen
- strukturelle Darstellungen

Als sprachliche Darstellung werden wir keine Programmiersprache, sondern eine Metasprache, einen sogenannten **Pseudocode**, vorstellen. Dieser Pseudocode ist kein übersetzbarer Quellcode, er wird aber in der Regel an Programmiersprachen wie beispielsweise C/C++, Java angelehnt. Wir benutzen folgende Konventionen für einen Pseudocode:

- Benennung: `ALGORITHMENNAME(Eingabeparameter)`
- Schleifen
  - `for ... to (downto) ... by ...`
  - `while ... do ...`
  - `do ... while ...`
- Fallunterscheidungen
  - `if ... else ...`
  - `switch(...) case ...`
- Blöcke werden durch geschweifte Klammern {...} gekennzeichnet.
- Kommentare werden durch // gekennzeichnet.
- Wertzuweisung durch: `x = Wert`

- Vergleich durch: `x == y`
- Zugriff auf i-te Komponente eines Feldes `feld` durch: `feld[i]`
  Zugriff auf Länge des Feldes `feld` durch: `feld.laenge`
- Zugriff auf Attribut `x` eines Objektes `obj` durch: `obj.x`
- Rückgabe eines oder mehrerer Werte durch `return`. (Die Prozedur bricht dann unmittelbar ab.)

Beispiel 2.1 Der Pseudocode zur Bestimmung des Maximums von n natürlichen Zahlen kann folgendermaßen formuliert werden. Dabei sind auf dem Feld `feld` die natürlichen Zahlen $a_1, ..., a_n$ abgespeichert.

```
MAX(feld)

max=feld[0]
for i=1 to n-1 { //Durchlaufe Feld
     if feld[i]>max { //Vergleiche mit max
          max=feld[i]  //Initialisiere max neu
     }
}  //Auf max steht das Maximum
```

Bemerkung 2.2 Grundsätzlich unterscheidet man bei der Erstellung von Algorithmen zwischen

- iterativer Programmierung
- rekursiver Programmierung

Die iterative Programmierung verwendet im Gegensatz zur rekursiven Programmierung keine Selbstaufrufe. Diese werden durch Verwendung von Schleifen kompensiert. Iterative Programme lassen sich auch rekursiv formulieren, rekursive Programme können iterativ umgesetzt werden. Oft haben rekursive Programme allerdings den höheren Speicheraufwand.

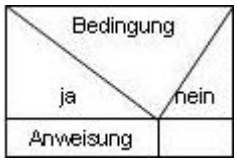
Bemerkung 2.3 Auch bei der Formulierung eines Pseudocodes kann man zwischen **Syntax** (Lehre vom Satzbau) und **Semantik** (Lehre von der inhaltlichen Bedeutung einer Sprache) unterscheiden.

Als strukturelle Daigramme werden **Struktogramme** (Nassi-Shneidermann-Diagramm) und **Programmablaufpläne** (PAP, Flussdiagramm, Programmstrukturplan) betrachtet. Wir stellen zunächst die Grundelemente der Struktogramme vor:

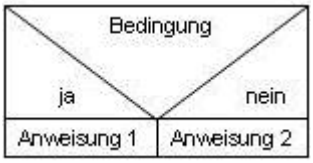
- Anweisung

- Anweisungsfolge

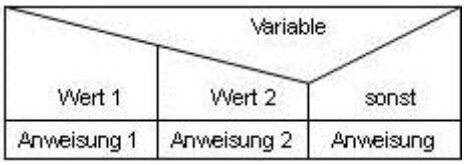
- einfache Verzweigung



bzw.



- mehrfache Verzweigung



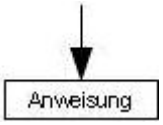
- for-Schleife

- while-Schleife

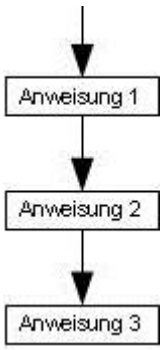
- do-while-Schleife

Bei Programmablaufplänen benutzt man unter anderem folgende Diagramme für obige Befehle bzw. Kontrollstrukturen:

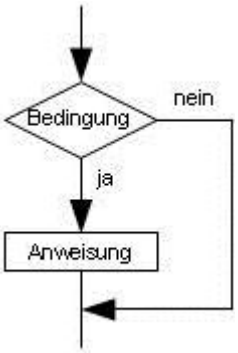
- Anweisung



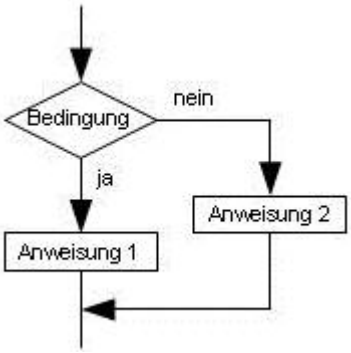
- Anweisungsfolge



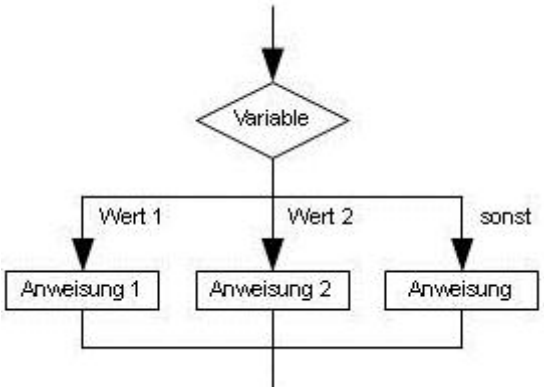
- einfache Verzweigung



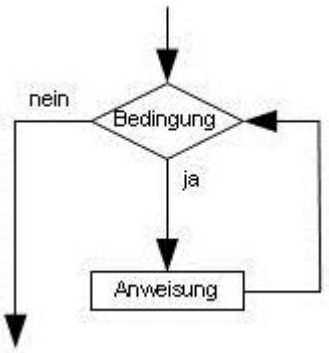
bzw.



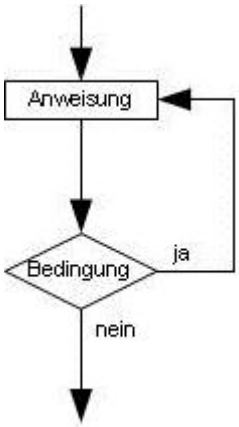
- mehrfache Verzweigung



- for-Schleife bzw. while-Schleife

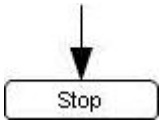


- do-while-Schleife

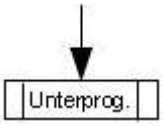


Bei Programmablaufplänen kommen auch unter anderem folgende Symbole zum Einsatz:

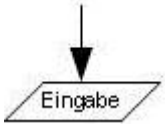
- Start/Stop-Symbol



- Unterprogramm



- Ein-/Ausgabe



3 Laufzeitanalyse von Algorithmen

Bei der Analyse eines Algorithmus geht es darum, ein Maß für die Effizienz des gegebenen Algorithmus hinsichtlich der Rechenzeit bzw. des benötigten Speicherplatzes anzugeben. Meist wird man diesbezüglich von einem allgemeinen idealisierten Modell einer **Maschine mit wahlfreiem Zugriff** (RAM, Random-Access-Machine) ausgehen. Innerhalb dieses Modells werden Anweisungen des Algorithmus nacheinander ausgeführt, Operationen können nicht parallel ablaufen. Das Befehlsrepertoire einer RAM beschränkt sich in der Regel auf Transportbefehle (Laden, Speichern, Kopieren), arithmetische Befehle (Addieren, Subtrahieren, Multiplizieren, Dividieren, Modulo, Ab-/Aufrunden) und Kontrollstrukturen (Wiederholungs-/Entscheidungsanweisungen). Für jeden dieser Befehle ist ein festes Maß an Zeit vorgesehen. Insbesondere wird gefordert, dass die Breite eines Datenwortes begrenzt ist. Wir gehen davon aus, dass für das Ausführen einer Zeile des Pseudocodes ein konstanter Zeitaufwand (der in jeder Zeile verschieden sein kann) notwendig ist.
Durch das RAM-Modell soll nicht versucht werden, Konzepte der Speicherhierarchie des Rechners zu modellieren. Man ist vielmehr daran interessiert, die Charakteristiken eines Algorithmus hinsichtlich dessen Anforderungen an die Resourcen aufzuzeigen.

Beispiel 3.1 Der folgende Algorithmus bestimmt iterativ die n-te Fibonacci-Zahl (für $n \geq 0$):

```
FIB(n)

1  f0=0
2  f1=1
3  if n==0 return f0
4  if n==1 return f1
5  for i=2 to n{
6       erg=f0+f1
7       f0=f1
8       f1=erg
9  }
10 return erg
```

Ist nun c_i der (konstante) Kostenaufwand der Zeile i, so ergibt sich Gesamtaufwand $T(n)$ des obigen Algorithmus durch:

$$T(n) = c_1 + c_2 + c_3 + c_4 + n \cdot c_5 + (n-1) \cdot (c_6 + c_7 + c_8) + c_9 + c_{10}$$

$T(n)$ ist demnach eine Polynomfunktion vom Grad eins, also von der Form

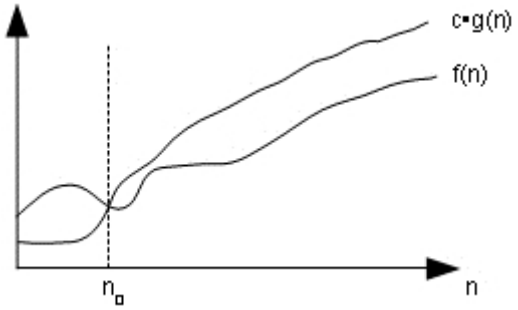
$$T(n) = an + b.$$

Ausschlaggebend für den Wachstumsgrad der Laufzeit ist stets der höchste Exponent (des Polynoms), in diesem Fall also die Eins.

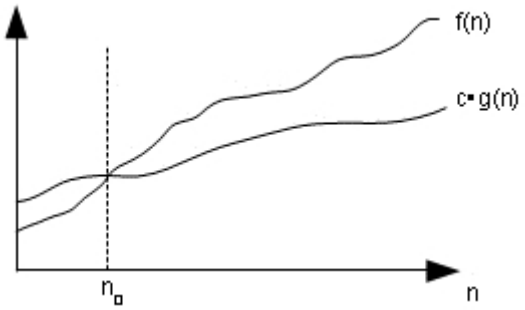
Um das Wachstum einer Laufzeit besser beschreiben zu können, bedient man sich mehrerer spezieller Notationen.

Definition 3.2 *Seien $f, g : \mathbb{N} \to \mathbb{R}^+$ Funktionen.*

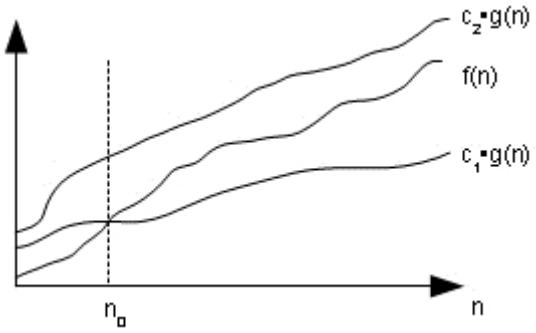
(a) Man sagt, $f(n)$ ist von $O(g(n))$ genau dann, wenn $n_0 \in \mathbb{N}$ und $c \in \mathbb{R}^+$ existieren mit $f(n) \leq cg(n)$ für alle $n \geq n_0$. Gegebenenfalls nennt man $g(n)$ eine **obere asymptotische Schranke**.



(b) Man sagt, $f(n)$ ist von $\Omega(g(n))$ genau dann, wenn $n_0 \in \mathbb{N}$ und $c \in \mathbb{R}^+$ existieren mit $f(n) \geq cg(n)$ für alle $n \geq n_0$. Gegebenenfalls nennt man $g(n)$ eine **untere asymptotische Schranke**.



(c) Man sagt, $f(n)$ ist von $\Theta(g(n))$ genau dann, wenn $n_0 \in \mathbb{N}$ und $c_1, c_2 \in \mathbb{R}^+$ existieren mit $c_1 g(n) \leq f(n) \leq c_2 g(n)$ für alle $n \geq n_0$. Gegebenenfalls nennt man $g(n)$ eine **asymptotisch scharfe Schranke**.



Bemerkung 3.3 (a) Anstatt $f(n)$ *ist in* $O(g(n))$ schreibt man auch $f(n) \in O(g(n))$ bzw. $f(n) = O(g(n))$. Dies gilt auch für die Bezeichnungen Ω und Θ. Man beachte, dass das Gleichheitszeichen bei dieser Notation missbräuchlich verwendet wird.

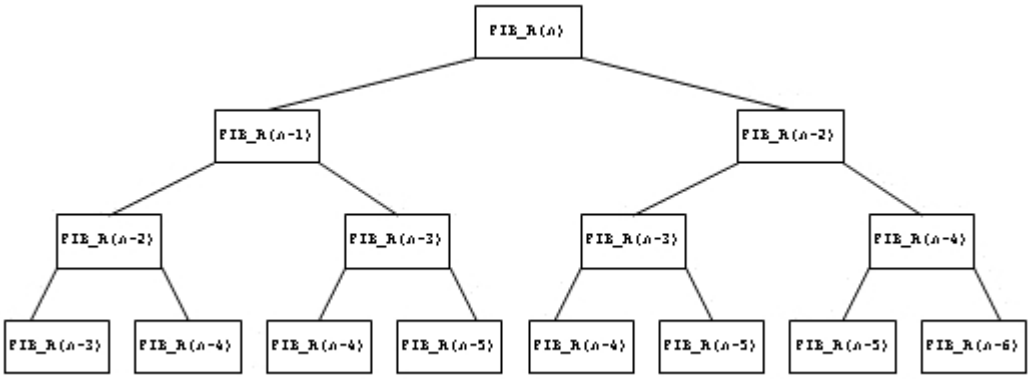
(b) Definition (b) ist dem Buch von Cormen entnommen. Gelegentlich wird obige Definition durch folgendes ersetzt:
Man sagt, $f(n)$ ist von $\Omega(g(n))$ genau dann, wenn $c \in \mathbb{R}^+$ existiert, so dass für alle $n_0 \in \mathbb{N}$ ein $n \geq n_0$ existiert mit $f(n) \geq cg(n)$.

(c) Es ist $f(n) = \Theta(g(n))$ genau dann, wenn $f(n) = \Omega(n)$ und $f(n) = O(g(n))$.

Beispiel 3.4 Der Algorithmus `FIB(n)` hat die Laufzeit $O(n)$ und auch $\Omega(n)$, also auch $\Theta(n)$. Insbesondere kann der Algorithmus zur Berechnung der Fibonacci-Zahlen auch rekursiv formuliert werden:

```
FIB_R(n)

1  if n==0 return 0
2  if n==1 return 1
3  return FIB_R(n-1)+FIB_R(n-2)
```

Der obige Pseudocode ist wesentlich kürzer und übersichtlicher als die iterative Version, doch die Laufzeit ist maßgeblich höher. Dies kann man sich an folgendem Diagramm verdeutlichen, in dem mittels einer Baumstruktur die geschachtelten Rekursionen aufgezeigt werden:



Man kann leicht zeigen, dass für die Laufzeit $f(n)$ von `FIB_R(n)` die Abschätzung $f(n) = \Theta(2^n)$ gilt.

Man hat folgende Bezeichnungen für die Wachstumsgrade der folgenden Funktionen:

- $g(n) = 1$ konstantes Wachstum
- $g(n) = \log(n)$ logarithmisches Wachstum
- $g(n) = n$ lineares Wachstum
- $g(n) = n\log(n)$ $n\log(n)$-Wachstum
- $g(n) = n^2$ quadratisches Wachstum
- $g(n) = n^k$ polynomielles Wachstum

- $g(n) = 2^n$ exponentielles Wachstum

Meist wird eine der oberen Funktionen in Verbindung mit der O, Ω und Θ-Notation verwendet.
Im allgemeinen ist die Laufzeit eines Algorithmen nicht von einer einzelnen Eingabegröße n abhängig. Dies kann man sich am Beispiel eines Sortieralgorithmus klarmachen (Sortieralgorithmen werden im nächsten Kapitel behandelt). Als Eingabeparameter erhält der Sortieralgorithmus ein Feld der Länge n, das sortiert werden soll. Die Laufzeit des Algorithmus wird nun als Funktion $f(n)$ (also in Abhängigkeit von der Länge n) aufgestellt. Ist nun das Feld bereits sortiert, so erhält man für $f(n)$ möglicherweise eine andere Funktionsvorschrift als für ein unsortiertes Feld. Dies macht man zum Anlass, eine Abschätzung der Laufzeit $f(n)$ für folgende Fälle zu betrachten:

Für die Eingabedaten liegt ein

- schlechtester Fall (worst case)
- mittlerer Fall (average case)
- bester Fall (best case)

vor. Der best case bei einem Sortieralgorithmus ist (wohl) der Fall, bei dem das Feld bereits sortiert ist, der worst case könnte der Fall sein, bei dem das Feld in entgegengesetzter Richtung sortiert ist. Ein average case ist meist sehr schwierig anzugeben.
Durch die Angabe $f(n) = O(g(n))$ für die Laufzeit eines Algorithmus ist der worst case (insbesondere die anderen beiden Fälle) mit einbezogen. Durch die Angabe $f(n) = \Omega(g(n))$ für die Laufzeit eines Algorithmus ist der best case (insbesondere die anderen beiden Fälle) mit einbezogen.
Durch Formulierungen *die Laufzeit im best case ist* $f(n) = \Omega(g(n))$ oder *die Laufzeit im worst case ist* $f(n) = \Theta(g(n))$ kann die Angabe der Laufzeit präzisiert werden.

Beispiel 3.5 Der Sortieralgorithmus `BUBBLESORT` funktioniert folgendermaßen: Eingabe ist ein Integer-Feld `feld` der Länge n. Der Algorithmus wird nun mittels einer Schleife (und der Laufvariable `j` die Komponenten `feld[2]` bis `feld[n]` des Feldes durchlaufen. In jedem Schritt vergleicht er die aktuelle Komponente `feld[j]` der Reihe nach mit allen Komponenten `feld[i]`, `i=j-1,j-2,...` (also allen Komponenten links von `feld[j]`), vertauscht die Komponente `feld[i]` mit `feld[j]`, falls `feld[i]<feld[j]` ist und sortiert demnach die aktuelle Komponente an der richtigen Stelle ein. Man kann sich dieses Prinzip vorstellen wie das Aufsteigen verschieden großer Blasen im Wasser. Aus diesem Grund wird der Algorithmus `BUBBLESORT` genannt. Der Pseudocode sieht folgendermaßen aus:

```
BUBBLESORT(feld)

for j=2 to feld.laenge{
     s=feld[j]
     i=j-1
     while i>0 und s<feld[i]{
          feld[i+1]=feld[i]
          feld[i]=s
```

```
          i=i-1
     }
}
```

Die Laufzeit $f(n)$ des BUBBLESORT-Algorithmus ist $f(n) = O(n^2)$, aber es ist $f(n) \neq \Theta(n^2)$. Für die Laufzeit $f(n)$ des BUBBLESORT-Algorithmus **im worst case** ist $f(n) = \Theta(n^2)$.

4 Das Prinzip Teile und Herrsche

Das Paradigma (Denkmuster) *Teile und Herrsche* beschreibt ein Prinzip des Algorithmenentwurfs. Wir werden dieses Prinzip an den Sortieralgorithmen MERGESORT und QUICKSORT erläutern. Der Sortieralgorithmus BUBBLESORT aus dem vorigen Kapitel unterliegt **nicht** dem Teile & Herrsche-Prinzip.

Beim Entwurf eines Algorithmus kann man nach folgender Methodik vorgehen:

(1) Teile das Problem in mehrere Teilprobleme auf, die jeweils kleinere Instanzen des Ausgangsproblems darstellen.

(2) Beherrsche die Teilprobleme rekursiv. Sind die Teilprobleme klein genug, so löse die Teilprobleme auf direktem Weg.

(3) Vereinige die Lösungen der Teilprobleme zur Lösung des ursprünglichen Problems.

Der Algorithmus MERGESORT (Sortieren durch Mischen) funktioniert nach folgendem Prinzip: Teile das Feld in zwei kleinere Felder, sortiere diese und füge dann die kleineren Felder so zusammen, dass das zusammengefügte Feld sortiert ist.
Wir werden zunächst eine Methode MERGE vorstellen, die zwei sortierte Felder zu einem sortierten Feld zusammenfügt:

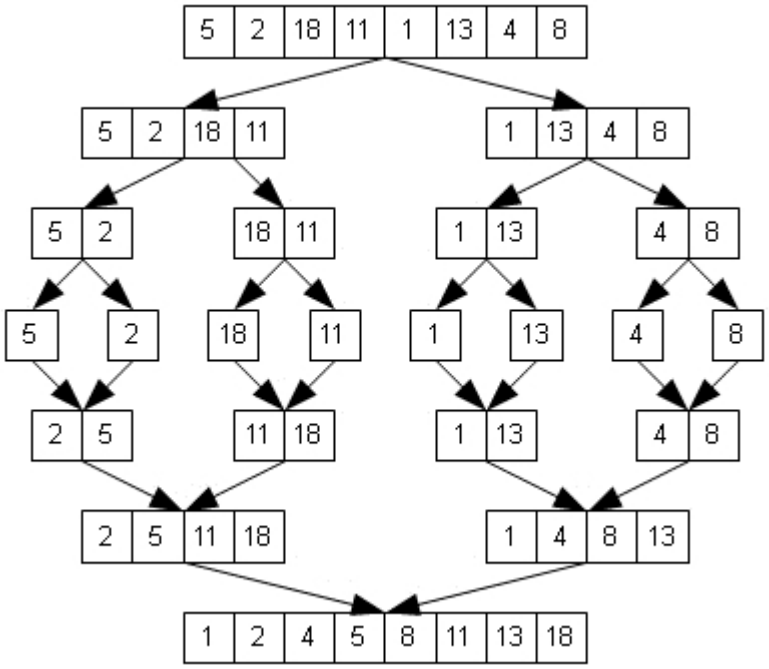
```
MERGE(feld1, feld2)

n=feld1.laenge
m=feld2.laenge
new feld[n+m] // neues Feld der Laenge n+m
i=1
j=1
for k=1 to n+m
     if i>n und j<=m{
          feld[k]=feld2[j]
          j=j+1
     }
     if j>m und j<=n{
          feld[k]=feld1[i]
          i=i+1
     }
     if i<=n und j<=m{
```

```
          if feld2[j]<feld1[i]{
               feld[k]=feld2[j]
               j=j+1
          }
          else {
               feld[k]=feld1[i]
               i=i+1
          }
     }
}
return feld
```

Unter Verwendung von zwei Methoden `FELDLINKS` und `FELDRECHTS`, die ein Feld in zwei kleinere Felder aufteilen, und der oben beschriebenen Methode kann nun der Algorithmus `MERGESORT` formuliert werden:

```
MERGESORT(feld)

if feld.laenge==1 return feld
return MERGE(MERGESORT(FELDLINKS(feld)),MERGESORT(FELDRECHTS(feld)))
```

Die Vorgehensweise des Algorithmus kann durch folgendes Beispiel verdeutlicht werden:



Die obige Variante des Mergesort-Verfahrens ist bezüglich des Speicheraufwandes nicht optimiert. Auch können einige Verzweigungen in anderer Weise erstellt werden, so dass sich die Laufzeit verringert (jedoch nicht maßgeblich). Durch die Vorgabe des Algorithmus in dieser Form sollte in erster Linie das Teile&Herrsche-Prinzip verdeutlicht werden.

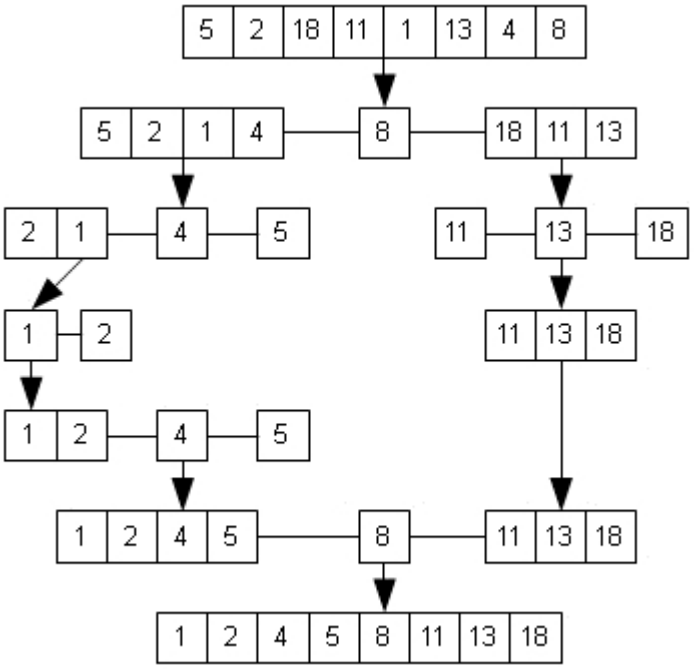
Ein weiteres Sortier-Verfahren ist der sogenannte **Quicksort**-Algorithmus. Dieser Algorithmus basiert ebenfalls auf dem Teile&Herrsche-Paradigma. Hierbei wird grundsätzlich das gegebene Feld mittels eines *Pivotelementes* in zwei neue Felder eingeteilt. Dabei besteht das erste Feld aus allen Elementen (außer dem Pivotelement) des Ursprungsfeldes, welche kleiner gleich dem Pivotelement sind. In dem zweiten Feld sind die Elemente, die größer als das Pivotelement sind. Der Algorithmus kann nun rekursiv die beiden kleineren Felder sortieren und dann zum sortierten Ursprungsfeld zusammensetzen. Das Pivotelement kann bei diesem Vorgang beliebig aus dem Feld gewählt werden.
Der Unterschied zum `MERGESORT`-Algorithmus liegt darin, dass bei `QUICKSORT` der meiste Aufwand im Aufteilen des Feldes, beim `MERGESORT` im Zusammenfügen des Feldes liegt.
Der hier vorgestellte `QUICKSORT`-Algorithmus arbeitet im Gegensatz zum `MERGESORT`-Verfahren **in place**, d.h. er ordnet die Elemente innerhalb des gegebenen Feldes um. Die Eingabeparameter `s` und `t` geben jeweils den ersten bzw. den letzten Index des Feldes an.

```
QUICKSORT(s,t)

     if s<t{
     p=PARTITION(s,t)
     quicksort(s,p-1)
     quicksort(p+1,t)
}
```

Dabei wird die Methode `PARTITION` genutzt, die das gegebene Feld so umsortiert, das alle Elemente, die kleiner gleich dem Pivotelement sind, nach *links* verlagert, alle Elemente, die größer als das Pivotelement sind, nach *rechts* verlagert werden. Insbesondere gibt diese Methode den Index des Pivotelementes zurück. Als Pivotelement wird die Methode zu Beginn stets das Element mit Index `t` wäehlen.

```
PARTITION(s,t)

p=feld[t]
for j=s to t-1{
     if feld[j]<=p{
          VERTAUSCHE(j,s)
          s=s+1
     }
}
VERTAUSCHE(s,t)
return s
```

Hier wird zusätzlich die Methode `VERTAUSCHE(s,t)` genutzt, die die Elemente `feld[s]` und `feld[t]` vertauscht. Das folgende Beispiel veranschaulicht die Vorgehensweise des QUICKSORT-Algorithmus. Dabei zeigen die Pfeile stets auf die gewählten Pivot-Elemente:



5 Datenstrukturen

Problemstellungen aus der Informatik legen nahe, den mathematischen Begriff einer *Menge* mit einzubeziehen. Da Mengen in unserem Zusammenhang meist einiger Veränderungen ausgesetzt sind (beispielsweise durch Löschen/Hinzunehmen von Elementen), spricht man hier von **dynamischen Mengen**. Eine solche dynamische Menge formuliert man zunächst in allgemeiner Form als einen *abstrakten Datentyp*:
Unter einem **abstrakten Datentyp** (ADT) versteht man eine Menge von Objekten zusammen mit einer Menge von Operationen auf den Objekten.
Eine Realisierung eines abstrakten Datentyps mittels einer Programmiersprache nennt man einen **Datenstruktur**. Elementare abstrakte Datentypen sind:

- lineare Listen
- Stapel
- Warteschlangen
- Bäume

Lineare Listen basieren auf dem Konzept einer endlichen Folge (im mathematischen Sinn): $a_0, a_1, a_2, ..., a_n$. Bei linearen Listen betrachtet man grundsätzlich folgende Operationen:

- `EINFUEGEN`: Einfügen eines Elementes in die Liste (evtl. mit Angabe der Stelle)
- `ENTFERNEN`: Entfernen eines Elementes aus der Liste (evtl. mit Angabe der Stelle)
- `SUCHEN`: Suchen eines Elementes in der Liste (mit Rückgabe boolean oder Stelle)
- `ZUGRIFF`: Liefert ein Element an einer Stelle der Liste

Im allgemeinen kann man lineare Listen auf zwei verschiedene Arten implementieren, nämlich durch

- Sequentielle Speicherung
  Listenelemente werden in einem zusammenhängenden Speicherbereich abgelegt (Array).
- Verkettete Speicherung
  Listenelemente werden in Speicherzellen abgelegt, deren Zusammenhang durch Zeiger hergestellt wird.

Wir werden zunächst die abstrakten Datentypen *Stapel* und *Warteschlangen* vorstellen, die spezielle lineare Listen darstellen. Weiterhin wird eine Implementierung der beiden Datentypen als sequenziell gespeicherte Liste erstellt.

Ein **Stapel** (Stack) ist eine Menge von Objekten zusammen mit den folgenden Operationen:
`PUSH`: Hinzufügen eines Elementes an oberste Stelle
`POP`: Löschen des Elementes an oberster Stelle
`TOP`: Gibt oberstes Element des Stapels aus
`STACKEMPTY`: gibt aus, ob der Stapel leer ist

Die `POP`-Operation eines Stapels arbeitet nach dem **LIFO**-Prinzip (Last-in-First-out). Dabei wird das zuletzt hinzugefügte Objekt des Stapels gelöscht. Versucht man, von einem leeren Stapel ein Element zu entnehmen, so spricht man von einem **Unterlauf des Stapels** (stack underflow). Um dies zu vermeiden, kann mittels der Methode `STACKEMPTY` getestet werden, ob der Stapel leer ist. Überschreitet die Höhe des Stapels einen vorgegebenen Wert, so spricht man von einem **Überlauf des Stapels** (stack overflow).
Ein Stapel kann beispielsweise durch ein Feld realisiert werden (hier wird die Stapelhöhe 100 vorgegeben):

```
class Stapel{
private int[] werte=new int[100];
private int lastind=-1;
public void push(int x){
     if (lastind<wert.length-1){
          lastind++;
          werte[lastind]=x;
     }
     else System.out.println("stack overflow");
}
public void pop(){
     if (!stackempty()) lastind=lastind-1;
     else System.out.println("stack underflow");
}
public int top(){
     return werte[lastind];
}
public boolean stackempty(){
```

```
     if (lastind==-1) return true;
     else return false;
}
}
```

Da die Fehlerbehandlung in Java noch nicht behandelt wurde, wurde hier auf eine `try ... catch`-Umgebung verzichtet. Eine Fehlerbehandlung wäre in der Methode `top` zusätzlich notwendig.

Eine **Warteschlange** (Queue) ist eine Menge von Objekten zusammen mit den folgenden Operationen:
`ENQUEUE`: Hinzufügen eines Elementes am Ende der Schlange
`DEQUEUE`: Löschen des Elementes am Kopf der Schlange
`QUEUEEMPTY`: gibt aus, ob die Schlange leer ist
`TOP`: Gibt das Kopfelement der Schlange aus

Die `DEQUEUE`-Operation funktioniert nach dem **FIFO**- Prinzip (First-in-First-out). Dabei wird das Element gelöscht, das zuerst der Warteschlange hinzugefügt wurde. Wie bei einem Stapel werden hier die Begriffe Überlauf bzw. Unterlauf der Schlange benutzt.
Auch eine Schlange kann durch ein Feld realisiert werden (wobei hier ein Feld der Länge 10 benutzt wird). :

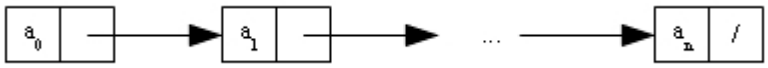
```
class Schlange{
private int[] werte=new int[10];
private int kopf=0;
private int ende=0;
public void enqueue(int x){
     if (!(ende+1-kopf)%10==0){
          werte[ende]=x;
          ende=(ende+1)%10;
     }
     else System.out.println("queue overflow");
}
public void dequeue(){
     if (!queueuempty()) kopf=(kopf+1)%10;
     else System.out.println("queue underflow");
}
public boolean queueempty(){
     if (kopf==ende) return true;
     return false;
}
}
```

Bei dieser Implementierung haben wir auf die Implementierung der Methode `TOP` verzichtet. Auch hierbei ist eine Fehlerbehandlung notwendig.

Bei einer (einfach) verketteten Speicherung linearer Listen besteht die Liste aus einer Fol-

ge von **Knoten**. Dabei enthält jeder Knoten:

- das Listenelement (oder *Schlüssel*)
- ein Zeiger (bzw. Referenz) auf den nächsten Knoten



Dabei symbolisiert der Slash im letzten Kästchen der oberen Graphik den Nullpointer `null`.

Im Gegensatz zu Arrays, deren Größe (ohne Neuinitialisierung) nicht veränderbar ist, kann sich die Größe der Liste bei einer verketteten Speicherung ändern. Diese Flexibilität hat einen erheblichen Vorteil gegenüber Feldern, andererseits ist die Suche nach Elementen der Liste zeitintensiver. Wir stellen eine mögliche Implementierung einer einfach verketteten Liste vor. Dazu erstellen wir zunächst eine Klasse `Knoten`, deren Instanzen die jeweiligen Knoten der Liste darstellen:

```
public class Knoten{
private Object schluessel;
private Knoten next;
public Knoten(Object o){
     this.schluessel=o;
     this.next=null;
}
public Object getSchluessel(){
     return schluessel;
}
public void setSchluessel(Object schluessel){
     this.schluessel=schluessel;
}
public Knoten getNext(){
     return next;
}
public void setNext(Knoten next){
     this.next=next;
}
}
```

Durch die folgende Klasse `Liste` wird eine verkettete Liste implementiert, die zunächst einen Startknoten mit dem Wert "Kopf" enthält. Ferner enthält die Klasse folgende Methoden:

- getFirstElement
  gibt erstes Element der Liste zurück
- getLastElement
  gibt letzes Element der Liste zurück

- addLastElement
  fügt neuen Knoten ans Ende der Liste an
- delete
  löscht letztes Element der Liste
- ausgeben
  gibt Listenelemente aus
- find
  sucht ein Element und gibt Wahrheitswert aus

Die Implementation in Java sieht nun folgendermaßen aus:

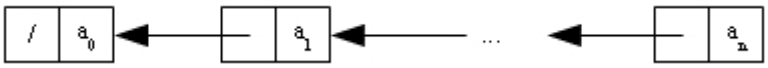
```
public class Liste{
private Knoten start=new Knoten("Kopf");
public Knoten getFirstElement(){
     return this.start
}
public Knoten getLastElement(){
     Knoten k=start;
     while(k.getNext()!=null) k=k.getNext();
     return k;
}
public void addLastElement(Object o){
     Knoten neu=new Knoten(o);
     Knoten last=getLastElement();
     last.setNext(neu);
}
public void ausgeben(){
     Knoten k=start;
     while (k!=null){
          System.out.println(k.getSchluessel());
          k=k.getNext();
     }
}
public void delete(Object o){
     Knoten k=start;
     while (k.getNext()!=null && !k.getSchluessel().equals(o)){
          if (k.getNext().getSchluessel().equals(o)){
               if (k.getNext().getNext()!=null){
                    k.setNext(k.getNext().getNext());
               }
               else{
                    k.setNext(null);
                    break;
               }
          }
```

```
          k=k.getNext();
     }
}
public boolean find(Object o){
     Knoten k=start;
     while (k!=null){
          if (k.getSchluessel().equals(o)){
               return true;
          }
          k=k.getNext();
     }
     return false;
}
}
```

Verkettete Listen können auch andere Formen annehmen. Eine **rückwärts verkettete Liste** besteht aus Knoten, die

- das Listenelement
- ein Zeiger auf den Vorgängerknoten

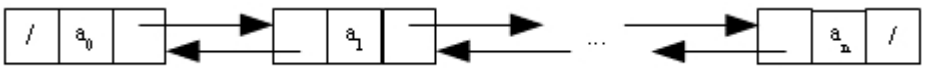
enthalten.



Eine **doppelt verkettete Liste** besteht aus Knoten, die

- das Listenelement
- einen Zeiger auf den Vorgängerknoten
- einen Zeiger auf den Nachfolgerknoten

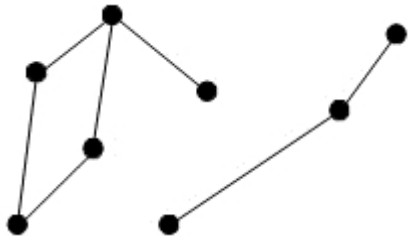
enthalten.



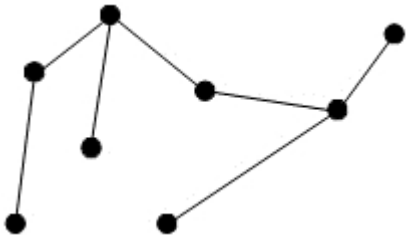
6 Bäume

Wir werden im folgenden die Datenstruktur eines *Baumes* vorstellen. Dazu ist es notwendig, einige Grundbegriffe aus der Graphentheorie zu behandeln.
Ein **ungerichteter Graph** G ist ein Paar (V, E), wobei V eine endliche Menge von **Knoten** und E eine endliche Menge von **Kanten** ist, die Knoten aus V verbinden (V steht für *vertex*, E steht für *edge*). Der Begriff Knoten ist prinzipiell nicht mit der gleichen Bezeichnung bei den verketteten Listen zu verwechseln. Man schreibt (u, v) bzw. (v, u) für eine Kante aus E, die

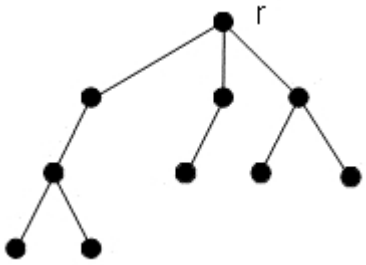
die Knoten u und v verbindet. In ungerichteten Graphen erlauben wir keine Kanten (u, u), die den gleichen Knoten $u \in V$ verbinden (sogenannte *Schlingen*). Außerdem ersparen wir uns an dieser Stelle den eigentlich notwendigen mathematischen Formalismus. Der folgende Graph besteht aus zwei Komponenten und enthält eine geschlossenen Kantenfolge:



Ein Graph G heißt **azyklisch**, falls G keine (echte) geschlossenen Kantenfolge enthält. Ein Graph G heißt **zusammenhängend**, falls jeder Knoten von jedem anderen aus erreichbar ist. Ein **(freier) Baum** ist ein azyklischer, zusammenhängender, ungerichteter Graph:



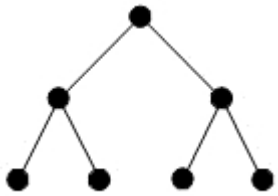
Ein **gewurzelter Baum** ist ein freier Baum, in dem ein Knoten r (als Wurzel) ausgezeichnet ist:



Ist r die Wurzel eines gewurzelten Baumes T und x ein weiterer Knoten, so heißt jeder Knoten auf dem Pfad von r nach x **Vorfahre** von x. Ist y Vorfahre von x, so heißt x **Nachfahre** von y. Ist die Kante (x, y) auf dem Pfad von r nach x (d. h. y ist *direkter* Vorfahre von x), so heißt y **Vater** von x und x heißt **Kind** von y. Haben zwei Knoten den gleichen Vater, so nennt man sie **Geschwister**. Ein Knoten ohne Kinder heißt **Blatt**. Die Anzahl der Kinder eines Knotens x in T heißt **Grad** von x. Die Länge des Pfades von r zu x heißt **Tiefe** von x in T. Alle Knoten gleicher Tiefe bilden eine **Ebene**. Die **Höhe** von T ist die Anzahl der Kanten auf dem längsten Pfad von r zu einem Blatt.
Ein **binärer Baum** T ist eine Struktur auf einer endlichen Menge von Knoten, die

- keine Knoten enthält, oder
- aus drei disjunkten Knotenmengen besteht: einer **Wurzel**, einem binären Baum, der sogenannte **linke Teilbaum** und einem binären Baum, dem sogenannten **rechten Teilbaum**.

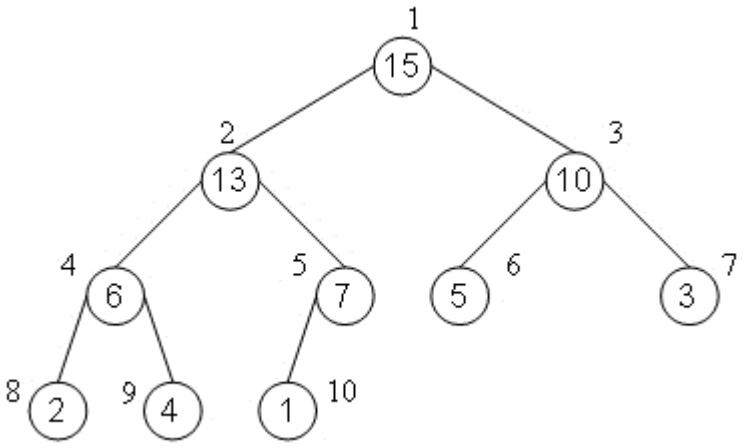
Die Wurzel des linken Teilbaums wird als **linkes Kind**, die Wurzel des rechten Teilbaumes als **rechtes Kind** der Wurzel von T bezeichnet. Hat jeder Knoten von T entweder den Grad 2 oder ist ein Blatt, so ist T ein **vollständiger** binärer Baum. Das folgende Diagramm zeigt einen vollständigen binären Baum der Höhe 2:



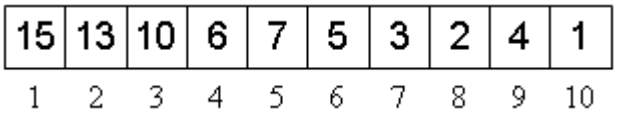
Ein **(binärer) Heap** ist ein (fast vollständiger) binärer Baum, der auf allen Ebenen, bis auf möglicherweise der niedrigsten Ebene, ausgefüllt ist. Für einem Heap werden in der Regel die Operationen

- `PARENT(x)`: Rückgabe des Vaterknotens von `x`
- `LEFT(x)`: Rückgabe des linken Kindknotens von `x`
- `RIGHT(x)`: Rückgabe des rechten Kindknotens von `x`
- `VALUE(x)`: Rückgabe des Wertes von `x`

realisiert.
Ein Heap kann mit einem Array implementiert werden. Dabei werden die Knoten der Reihe nach in jeder Ebene von links nach rechts fortlaufend durchnummeriert.



Die Werte der einzelnen Knoten werden dann in einem Array an den entsprechenden Stellen gespeichert.



Eine Implementierung in Java könnte folgendermaßen aussehen:

```
public class Heap{
private int[] feld;
private int groesse;
public Heap(int[] feld){
     this.feld=feld;
     this.groesse=feld.length-1;
}
public int parent(int i){
     if (i>0) return i/2;
     return -1;
}
public int left(int i){
     if (2*i<=groesse) return 2*i;
     return -1;
}
public int right(int i){
     if (2*i+1<=groesse) return 2*i+1;
     return -1;
}
public int value(int i){
     return feld[i];
}
}
```

Wir werden im folgenden zunächst ein weiteres Sortierverfahren, den `HEAPSORT`-Algorithmus, vorstellen. Beim Entwurf dieses Algorithmus steht eine neue Technik im Vordergrund, nämlich die Verwendung von Datenstrukturen zum Verwalten von Informationen. Aus diesem Grund betrachtet man in der Regel zwei Arten von Heaps, den *Max-Heap* und den *Min-Heap*. Ein **Max-Heap** ist definiert durch die Eigenschaft

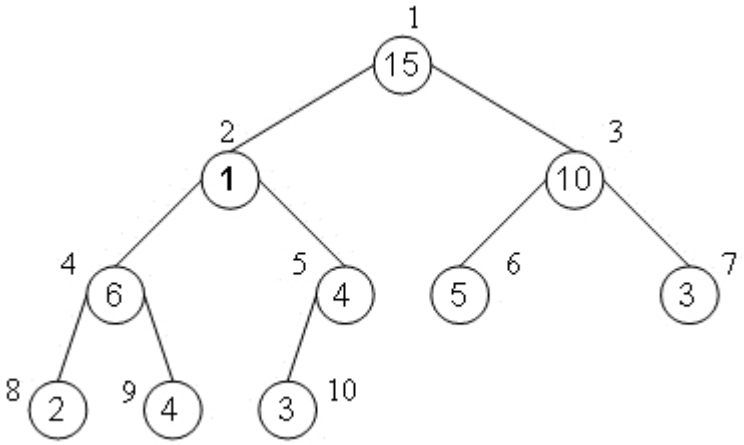
$$\texttt{VALUE}(\texttt{PARENT}(\texttt{x})) \geq \texttt{VALUE}(\texttt{x})$$

für alle Knoten `x`, wobei `x` nicht die Wurzel ist. Das obige Beispiel erfüllt die Max-Heap-Bedingung. Ein **Min-Heap** ist definiert durch die Eigenschaft

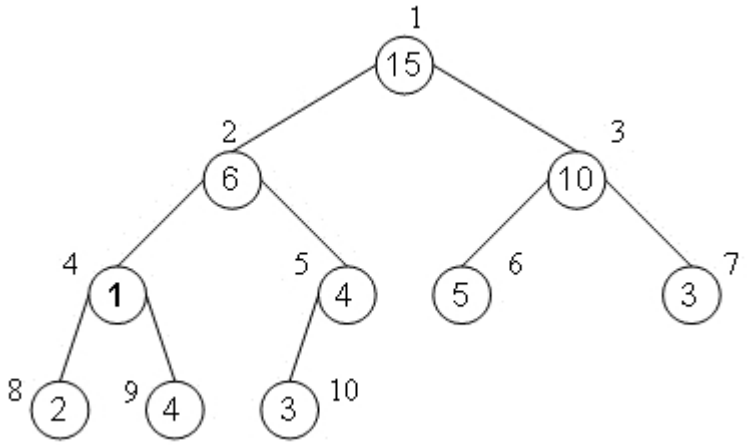
$$\texttt{VALUE}(\texttt{PARENT}(\texttt{x})) \leq \texttt{VALUE}(\texttt{x})$$

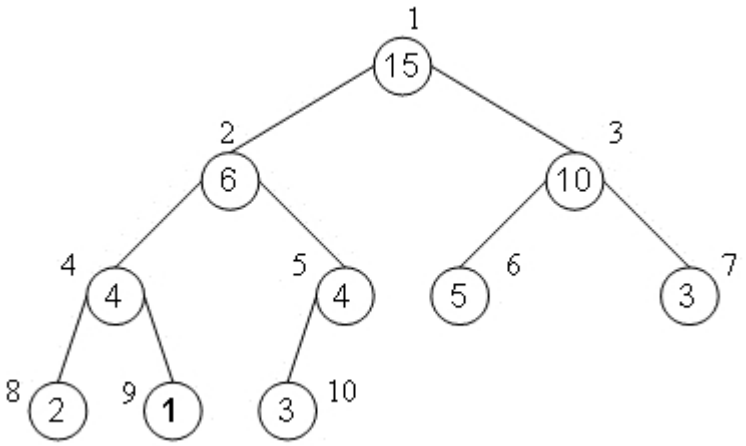
für alle Knoten `x`, wobei `x` nicht die Wurzel ist. Für den `HEAPSORT`-Algorithmus setzt man in der Regel voraus, daß die Daten in der Datenstruktur eines Max-Heap vorliegen. Dazu

muß zunächst sichergestellt sein, daß die Max-Heap-Bedingung gültig ist. Dazu betracht man wiederum die folgenden zwei Methoden:
Der Algorithmus `MAX-HEAPIFY(x)` setzt voraus, daß die binären Teilbäume mit Wurzeln `LEFT(x)` und `RIGHT(x)` bereits die Max-Heap-Bedingung erfüllen. Die Methode `MAX-HEAPIFY(x)` wird dann solange die Wurzel `x` mit dem Kind mit höchstem Wert vertauschen, bis die Max-Heap-Eigenschaft für den gesamten Baum erfüllt ist. Ist beispielsweise folgender Heap gegeben,



so wird die Methode MAX-HEAPIFY(2) schrittweise folgendermaßen vorgehen (die 2 in der Anweisung gibt den Index des Knotens an):





Der resultierende Heap hat dann die Max-Heap-Eigenschaft. Die Prozedur `MAX-HEAPIFY` kann als Instanzmethode der obigen Klasse `Heap` auf folgende Weise umgesetzt werden:

```
public void maxHeapify(int i){
     int l=left(i);
     int r=right(i);
     if (l==-1) l=i;
     if (r==-1) r=i;
     int max;
     if (feld[l]>feld[i] && feld[l]>=feld[r]) max=l;
     else if (feld[r]>feld[i] && feld[r]>feld[l]) max=r;
     else max=i;
     if (max!=i);{
          vertausche(i,max);
          maxHeapify(max);
     }
}
```

Dabei vertauscht die Methode `vertausche(i,max)` die Komponenten `feld[i]` und `feld[max]`. Die Methode `BUILD-MAX-HEAP` überführt einen Heap in einen Max-Heap. Dabei werden alle Knoten, die keine Blätter sind, in umgekehrter Reihenfolge (also von unten) durchlaufen und jeweils die Methode `maxHeapify(i)` angewendet. Beachte: der erste Knoten (in umgekehrter Reihenfolge), der kein Blatt ist (also keine Kinder besizt), hat den Index $\lfloor$`groesse/2`$\rfloor$. Dabei ist $\lfloor$`groesse/2`$\rfloor$ die grösste ganze Zahl, die kleiner-gleich `groesse/2` ist. Dies kann in Java einfach durch `groesse/2` umgesetzt werden, da `groesse` vom Typ `int` ist. Die Methode `BUILD-MAX-HEAP` kann nun folgendermaßen als Instanzmethode der Klasse `Heap` umgesetzt werden:

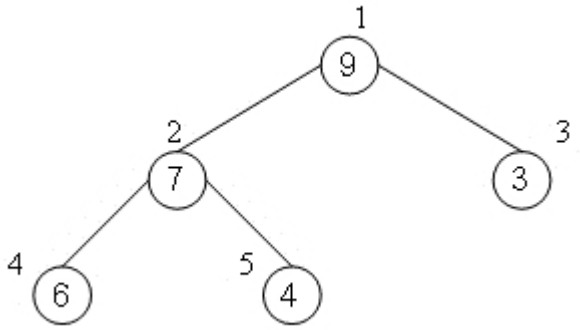
```
public void buildMaxHeap(){
     for (int i=groesse/2; i>=0;i--) maxHeapify(i);
}
```

Der Sortieralgorithmus `HEAPSORT` legt einen Max-Heap zugrunde und funktioniert folgen-

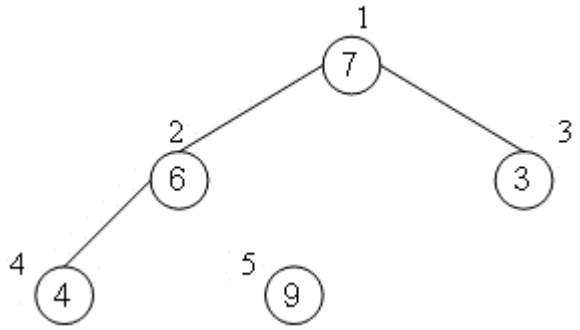
dermaßen:
Die Wurzel des Max-Heaps ist das größte Element des Heaps und wird aus dem Baum entfernt. Dann wird das letzte Element des Heaps auf die Position der Wurzel gesetzt und die Methode `MAX-HEAPIFY(r)` angewendet, wobei `r` die Wurzel des Heaps ist. Die Vorgehensweise wird wiederholt, bis kein Element mehr übrig ist. Die entfernten Elemente sind dann der Reihe nach sortiert.
Der beschriebene Algorithmus kann in folgender Weise umgesetzt werden:

```
public void heapsort(){
     buildMaxHeap();
     for (int i=groesse; i>0; i--){
          vertausche(i,0);
          groesse--;
          maxHeapify(0);
     }
}
```

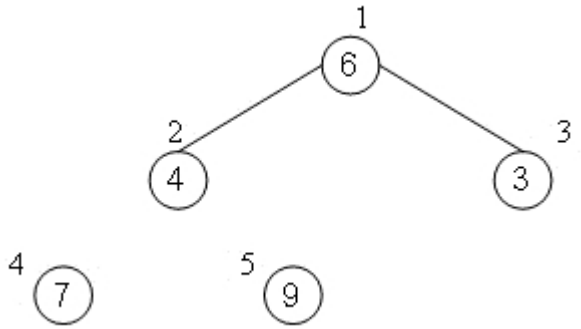
Der Algorithmus wird an folgendem Beispiel illustriert. Dabei wird der Laufindex `i` dem Diagramm angepasst. Für den entsprechende Laufindex der Java-Implementation muß `i` um eins erniedrigt werden. Gegeben ist der Max-Heap



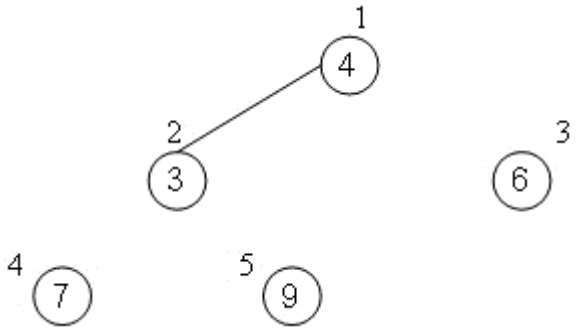
Für `i=5` ergibt sich nach Durchführung von `MAX-HEAPIFY`:



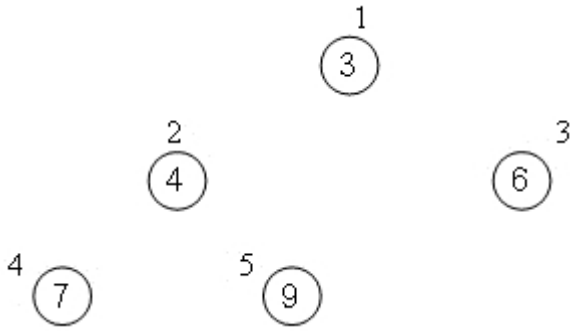
Für `i=4` ergibt sich nach Durchführung von `MAX-HEAPIFY`:



Für `i=3` ergibt sich nach Durchführung von `MAX-HEAPIFY`:



Für `i=2` ergibt sich nach Durchführung von `MAX-HEAPIFY`:



Die Werte sind nun der Größe nach geordnet.

Der Vorteil bei der Verwendung der Datenstruktur eines Baumes bzw. Heaps liegt oft darin, dass bei bestimmten Prozeduren nicht das gesamte Feld, sondern nur jeweils ein Knoten einer Ebene durchlaufen wird. Damit kann sich die Laufzeit von $O(n)$ auf $O(\log(n))$ reduzieren. Die Laufzeit von `MAX-HEAPIFY` angewendet auf einen Knoten der Höhe `h` ist $O(h)$, bzw. $O(\log(n))$, wobei n die Länge des Feldes ist. Für `BUILD-MAX-HEAP` ergibt sich die Laufzeit $O(n \log(n))$. Entsprechend ist die Laufzeit von `HEAPSORT` in $O(n \log(n))$.

Eine weitere wichtige Anwendung der **Datenstruktur** Heap ergibt sich bei der Implementierung einer *Prioritätswarteschlange*. Eine **Prioritätswarteschlange** (Vorrangwarteschlange,

priority queue) ist ein abstrakter Datentyp, der in der Regel aus einer Menge von Objekten und den Operationen:

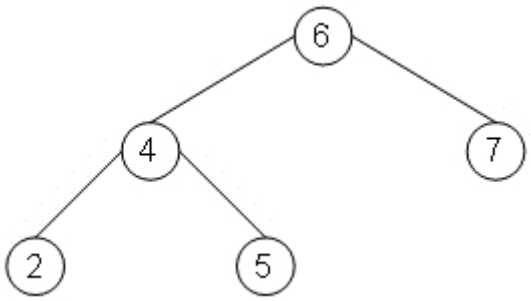
- `INSERT(x)`: Einfügen des Elementes `x` in die Objektmenge
- `MAX`: Gibt Element mit maximalem Schlüssel zurück
- `EXTRACT-MAX`: Entfernt Element mit maximalem Schlüssel und gibt dieses zurück
- `INCREASE-KEY(x,k)`. Erhöht den Wert des Schlüssels von `x` auf den Wert `k`

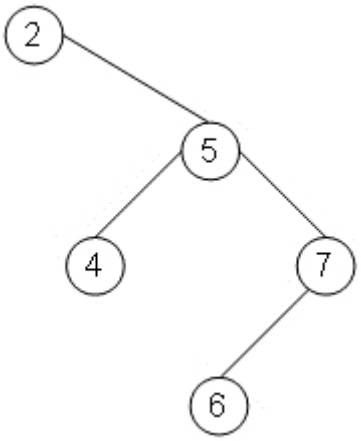
besteht. Dabei hat jedes Element `x` einen Schlüssel, dessen Wert die Priorität des jeweiligen Elementes beschreibt. Die Menge der Schlüssel kann nun durch einen Max-Heap realisiert werden (der dann auch *Max-Prioritätswarteschlange* genannt wird). Die obigen Methoden operieren dann auf den Max-Heap der Schlüssel. Dabei muß stets sichergestellt sein, daß nach Anwendung einer der Operationen die Max-Heap-Eigenschaft gültig ist. Auf eine Java-Implementierung einer Prioritätswarteschlange wird an dieser Stelle verzichtet.

Eine weitere Anwendung binärer Bäume bilden die sogenannten *binären Suchbäume*. Dabei betrachten wir binäre Suchbäume als Datenstruktur, nämlich als spezielle Implementierung eines *Suchbaumes*. Ein **Suchbaum** besitzt eine Menge von Objekten, die in Form einer Baumstruktur dargestellt wird, und die Operationen

- `EINFÜGEN`: Einfügen eines Elementes in den Baum
- `ENTFERNEN`: Löschen eines Elementes aus dem Baum
- `TRAVERSIEREN`: Methode zur systematischen Durchlaufung der Knoten des Baumes
- `SUCHEN`: Suchen eines Elementes im Baum

Wir werden hier nur **binäre Suchbäume** betrachten, für deren Knoten stets folgende Bedingung gelten muß:
Ist `x` ein Knoten des Baumes, so gilt: Ist `y` ein Knoten des linken Teilbaumes von `x`, so ist `VALUE(x)` $\geq$ `VALUE(y)`. Ist `y` ein Knoten des rechten Teilbaumes von `x`, so ist `VALUE(x)` $\leq$ `VALUE(y)`.
Die folgenden beiden Suchbäume repräsentieren die selbe Menge von Objekten:

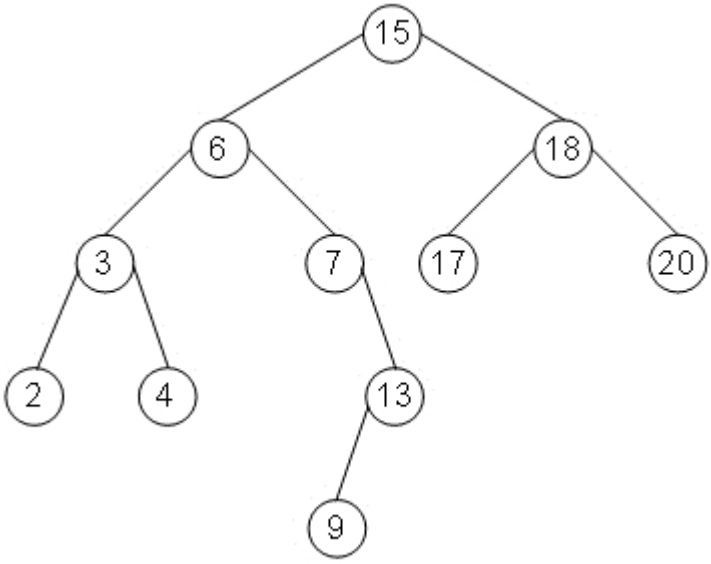




Ist ein binärer Suchbaum gegeben, so kann durch sogenannte **Inorder-Traversierung** die Elementmenge sortiert ausgegeben werden. Die Vorgehensweise ist rekursiv und prinzipiell sehr einfach umzusetzen. Es ist sinnvoll, binäre Suchbäume durch Verkettung mittels Zeigern zu implementieren. Dies werden wir hier nicht tun, wir geben lediglich den Pseudocode zur Inorder-Traversierung als Methode `INORDER-TREE-WALK` an:

```
INORDER-TREE-WALK(x)

if x≠nil{
     INORDER-TREE-WALK(x.LEFT)
     VALUE(x)
     INORDER-TREE-WALK(x.RIGHT)
}
```

Dabei ist `x` die Wurzel des Teilbaumes, der durchlaufen wird.
Die aufgestellte Suchbaum-Bedingung ist maßgeblich für die Operation `SUCHEN`. Ist beispielsweise der binäre Suchbaum



gegeben, so kann der Wert 13 durch Beschreitung des Weges $15 \to 6 \to 7 \to 13$ gefunden werden. Dabei wird jede Ebene genau einmal durchschritten. Die Laufzeit eines solchen Verfahrens ist also in $O(h)$, wobei h die Höhe des Baumes ist. Eine iterative Umsetzung dieses Verfahrens als Pseudocode ist gegeben durch:

```
ITERATIVE-TREE-SEARCH(x,k)

while x≠nil und k≠VALUE(x){
     if k<VALUE(x) x=x.LEFT
     else x=x.RECHTS
}
return x
```

Da die Laufzeit eines solchen Algorithmus von der Höhe des Baumes abhängt, ergibt sich die Frage, wie ein solcher Suchbaum gebildet wird. Ind der Regel wird dieser iterativ gebildet, beginnend mit einem leeren Baum. Dann werden durch Anwendung der Operation `EINFÜGEN` Schritt für Schritt die Knoten dem Baum zugefügt. Hierbei kommt es natürlich an, in welcher Reihenfolge man die Ausgangsdaten einfügt. Dies ist das maßgebliche Kriterium, nach dem sich der Suchbaum entwickelt. Die Operationen `EINFÜGEN` und `ENTFERNEN` werden hier nicht vorgestellt.

7 Greedy-Algorithmen

In diesem Kapitel beschäftigen wir uns mit sogenannten *Greedy-Algorithmen.* Ein solcher Algorithmus besitzt in der Regel die **Greedy-Auswahleigenschaft**, bei der in jedem aktuellen Schritt des Verfahrens die Wahl getroffen wird, die im Moment am besten erscheint. Dabei bleiben Lösungen von Teilproblemen unberücksichtigt. Wir werden uns das Vorgehen eines Greedy-Algorithmus an folgender Problemstellung klarmachen:

Das Aktivitäten-Auswahl-Problem:
Gegeben ist eine Folge von Aktivitäten $a_1, a_2, ..., a_n$ mit jeweils einer Startzeit s_i und einer Endzeit f_i (wobei $0 \leq s_i < f_i < \infty$ gilt). Dabei sind zwei Aktivitäten a_i, a_j **kompatibel**, wenn sich die jeweiligen Zeitintervalle nicht überlappen, also wenn entweder $f_i \leq s_j$ oder $f_j \leq s_i$ gilt. Beim Aktivitäten-Auswahl-Problem soll die maximale Teilmenge paarweise kompatibler Aktivitäten gefunden werden.
Beispielsweise können wir folgende Aktivitäten betrachten:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	14
f_i	4	5	6	7	9	9	10	11	12	14	16

Die Menge $\{a_3, a_9, a_{11}\}$ ist paarweise kompatibel, aber **nicht** maximal. Die Menge $\{a_1, a_4, a_8, a_{11}\}$ ist paarweise kompatibel und maximal, bildet also eine Lösung der Ausgangsproblems. Die Menge $\{a_2, a_4, a_9, a_{11}\}$ bildet ebenfalls eine Lösung. An diesem Beispiel sieht man nun, dass es durchaus mehrere gültige Lösungen für ein gegebenes Problem geben kann.
Das folgende Verfahren liefert zum gegebenen Problem eine maximale Teilmenge paarweise

kompatibler Aktivitäten. Dabei wird vorausgesetzt, dass die gegebenen Daten so geordnet sind, dass die Endzeiten in ansteigender Reihenfolge vorliegen (dies ist bei dem angeführten Beispiel bereits der Fall).
Das Verfahren wird zunächst die erste Aktivität (also a_1) wählen. In jedem weiteren Schritt wird dann die Aktivität gewählt, die als erstes endet und mit der vorigen kompatibel ist. Da die Aktivitäten nach den Endzeiten sortiert sind, ist es ausreichend, in jedem Schritt die erste Aktivität zu wählen, deren Startzeit größer-gleich der Endzeit des Vorgänger ist. Das Verfahren wird nun folgende Aktivitäten wählen (die jeweiligen Start/Endzeiten sind in den Klammern angegeben):

$$\begin{array}{cccc} a_1 & a_4 & a_8 & a_{11} \\ (1-4) & (5-7) & (8-11) & (12-16) \end{array}$$

In jedem Schritt wählt der Algorithmus also das erstbeste passende Objekt. Dabei wird auch auf keine Lösung eines Teilproblems zurückgegriffen. Die Geedy-Auswahleigenschaft ist demnach erfüllt. An diesem Beispiel wird in sehr einfacher Weise die Greedy-Charakteristik (also die Auswahl des *Erstbesten* in jedem Schritt) eines Algorithmus aufgezeigt. Es muß im allgemeinen natürlich noch gezeigt werden, daß ein solches Vorgehen tatsächlich eine korrekte Lösung des Problems erzeugt. Diesen notwendigen Korrektheitsbeweis werden wir hier aber nicht erbringen. Es soll im übrigen nicht unerwähnt bleiben, daß eine solche Greedy-Strategie nicht immer eine korrekte Lösung ermittelt. Am Beispiel des Rucksack.-Problems werden wir dies am Ende des Kapitels erläutern.
Das Verfahren könnte nun iterativ folgendermaßen umgesetzt werden (dabei sind `s` und `t` die Felder der Start- und Endzeiten):

```
GREEDY-ACTIVITY-SELECTOR(s,f)

n=s.laenge
k=f[1]
Ausgabe: 1 //Aktivität 1 wird gewählt und ausgegeben
for i=2 to n{
     if k<=s[i]{
          Ausgabe: i
          k=f[i]
     }
}
```

Eine rekursive Variante des Algorithmus könnte folgendermaßen aussehen (dabei ist `t` der Index der Aktivität, bei der der Algorithmus startet):

```
REKURSIVE-GREEDY-ACTIVITY-SELECTOR(s,f,t)

n=s.laenge
k=f[t]
Ausgabe: t //erste Aktivität wird gewählt und ausgegeben
for i=t+1 to n{
     if k<=s[i]{
          REKURSIVE-GREEDY-ACTIVITY-SELECTOR(s,f,i)
```

```
            break
        }
}
```

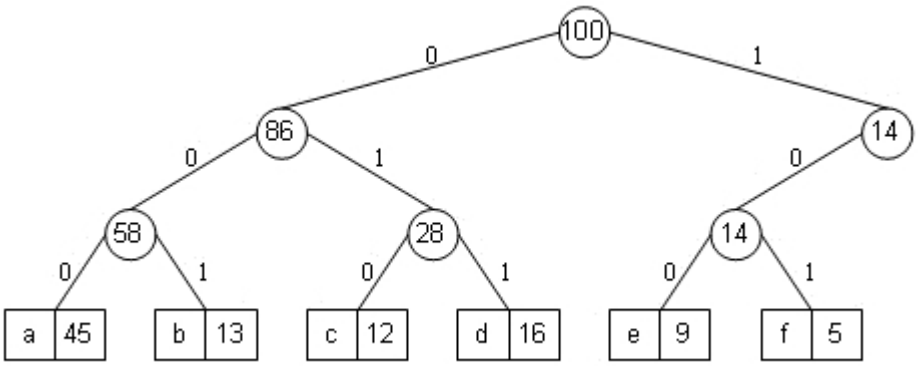
Die Huffman-Codierung:
Wir werden im folgenden einen Algorithmus zur Komprimierung von Daten vorstellen. Dieses Verfahren geht ebenfalls nach einer Greedy-Strategie vor. Wir beginnen mit einem kleinen Beispiel: In einer Zeichenkette der Länge 100 befinden sich ausschließlich die Zeichen *a, b, c, d, e, f*. Jedes dieser Zeichen wird durch eine binäre Zeichenfolge, einen sogenannten **Code** bzw. ein **Codewort**, dargestellt. Verwendet man Codewörter mit fester Länge, so benötigt man zur Codierung von 6 Zeichen mindestens 3 Bit. Es ist aber auch möglich, Codewörter mit variabler Länge zu konstruieren. Mögliche Codierungen sind in folgender Tabelle dargestellt:

Zeichen	a	b	c	d	e	f
Häufigkeit	45	13	12	16	9	5
Code fester Länge	000	001	010	011	100	101
Code variabler Länge	0	100	101	110	1110	1111

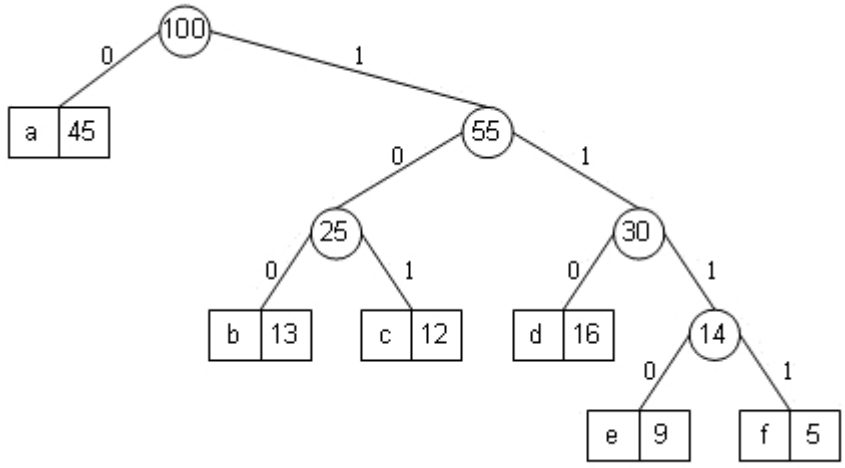
Der obige Code mit variabler Länge entspricht der Huffman-Codierung der gegebenen Zeichen. Es handelt sich hierbei um einen sogenannten **Präfix-Code**, d.h. kein Codewort ist Präfix eines anderen Codewortes. Im gegebenen Beispiel werden bei der Codierung mit fester Länge 300 Bit benötigt. Bei Verwendung des Codes variabler Länge werden insgesamt

$$45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4 = 224 \text{ Bit}$$

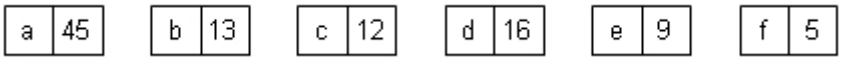
verwendet. Codes variabler Länge bilden also ein adäquates Mittel zur Komprimierung von Daten. Die jeweiligen Codewörter bei der Codierung mit fester Länge kann durch folgenden Baum dargestellt werden:



Dabei sind an jedem Knoten die relativen Häufigleiten eingetragen. Der Baum zum Code mit variabler Länge sieht folgendermaßen aus:

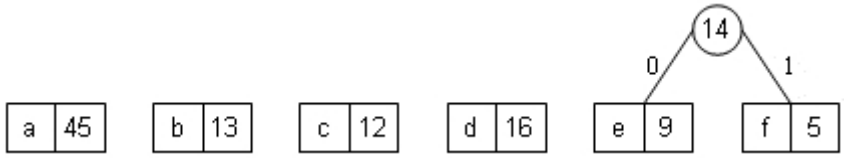


Die Konstruktion eines solchen Huffman-Codes erfolgt durch Erstellung des zugehörigen Baumes beginnend bei den Blättern:

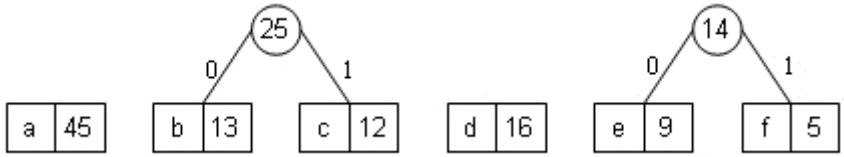


Man betrachtet dann schrittweise die Knoten, die keinen Vaterknoten besitzen und wählt die beiden Knoten mit dem kleinsten Wert. Diese beiden Knoten erhalten einen Vaterknoten mit der Summe der beiden Werte der Kindknoten. Wir erläutern die Vorgehensweise an den folgenden Bäumen:

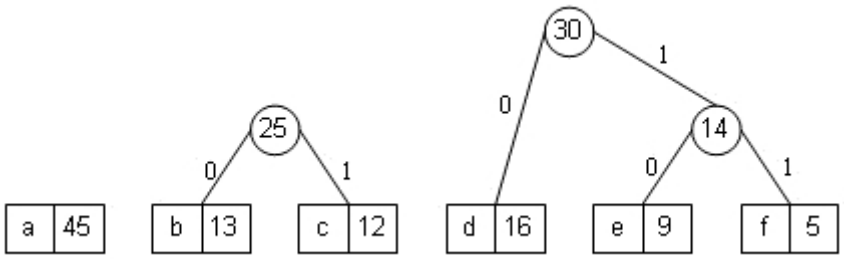
Schritt 1:



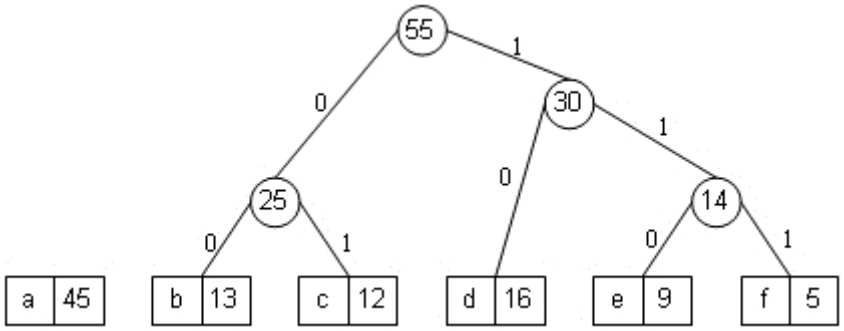
Schritt 2:



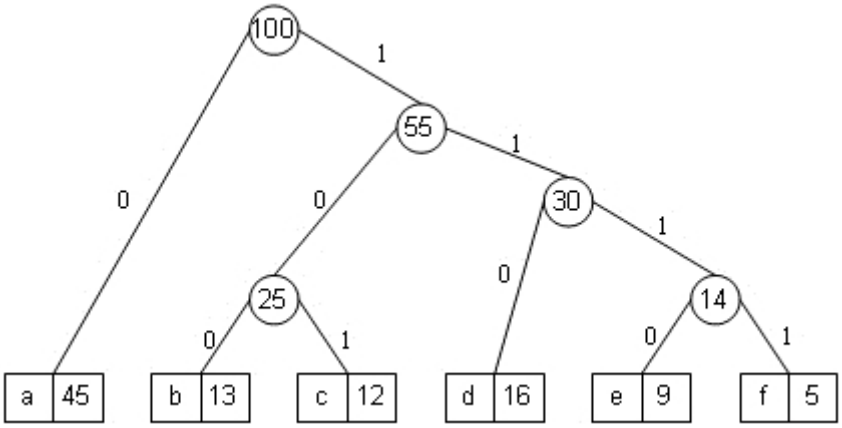
Schritt 3:



Schritt 4:



Schritt 5:



Da das Verfahren in jedem Schritt die Knoten mit den kleinsten Werten berücksichtigt, geht der Algorithmus nach einer Greedy-Strategie vor. Die Korrektheit des Algorithmus wird hier nicht bewiesen. Bei dem Ergebnis handelt es sich um einen Präfix-Code, da jeder Knoten, der für die zu codierenden Zeichen *a, b, c, d, e, f* steht, ein Blatt ist.

Die Verwendung einer Greedy-Strategie führt nicht bei jedem Problem zum Erfolg. Wir wollen dies exemplarisch am *Rucksackproblem* erläutern:
Gegeben sind n Gegenstände mit einem Gewicht $m_1, m_2, ...$ (in Kilogramm) und einem Wert $p_1, p_2, ..., p_n$ (in Euro). Zum Einpacken von Gegenständen steht ein Rucksack zur Verfügung, der maximal mit M Kilogramm belastet werden kann. Der Rucksack soll nun so gepackt werden, dass sich im Rucksack der maximale Wert befindet.

Konkret steht ein Rucksack mit einer Maximalbelastung von 50 Kilogramm zur Verfügung. Außerdem sind drei Gegenstände gegeben:

Gegenstand	A	B	C
Gewicht	40	25	25
Wert	80	35	49

Ein Greedy-Algorithmus zum Lösen des Problems würde wohl folgendermaßen vorgehen: In jedem Schritt wird der Gegenstand in den Rucksack gepackt, der

- die verbleibende Belastbarkeit des Rucksacks nicht überschreitet
- den durchschnittlich höchsten Wert der verbleibenden Gegenstände besitzt.

Würden mehrere Gegenstände obige Bedingungen erfüllen, würde der Gegenstand mit maximalem Gewicht gewählt werden.
In obigem Beispiel würde das Verfahren Gegenstand A wählen, denn A hat den größten Durchschnittswert von 2 Euro/Kilo und überschreitet nicht die Belastbarkeit des Rucksackes. Es könnte kein weiterer Gegenstand mehr gewählt werden. Die optimale Lösung besteht allerdings aus den Gegenständen B und C. Ein Greedy-Algorithmus ist für diese Problemstellung also nicht geeignet.

Literatur

[1] H. Cormen, *Algorithmen - eine Einführung*, Oldenbourg-Verlag 2010

[2] T. Ottmann, P. Widmayer, *Algorithmen und Datenstrukturen*, 4. Auflage, Spektrum Akad. Verlag 2002

[3] R. Sedgewick, *Algorithmen*, Addison Wesley 1991

[4] A. Solymosi, U. Grude, *Grundkurs Algorithmen und Datenstrukturen in Java*, 4. Auflage, Vieweg+Teubner 2008