Algorithmen

Vorlesung im WS 2000/2001

Oliver Vornberger

Olaf Müller

Praktische Informatik
Fachbereich Mathematik/Informatik
Universität Osnabrück

Literatur

- Guido Krüger:
  Java 1.1 lernen, Addison-Wesley, 1997.
- Ken Arnold & James Gosling:
  Java — Die Programmiersprache, Addison-Wesley, 1996.
- David Flanagan:
  "Java in a Nutshell , (2. Auflage: Deutsche Ausgabe für Java 1.1)", O'Reilly, 1998.

Danksagung

Wir danken ...

... Frau Gerda Holmann für sorgfältiges Erfassen des Textes und Erstellen der Grafiken.

... Frau Astrid Heinze für die Konvertierung gemäß neuer Rechtschreibung.

... Herrn Frank Lohmeyer für die Bereitstellung des Pakets `AlgoTools` von *Java*-Klassen zur vereinfachten Ein- und Ausgabe sowie für die Implementation eines *Java-Applets* zur Simulation der Beispiel-Programme im WWW-Browser.

... Herrn Frank M. Thiesing für die Bereitstellung eines Entwicklungswerkzeuges zur Typesetting-Organisation.

... Herrn Viktor Herzog für die Konvertierung des Skripts nach HTML.

HTML-Version
Der Inhalt dieser Vorlesung kann online betrachtet werden unter
`http://www-lehre.informatik.uni-osnabrueck.de/~ainf`

Osnabrück, im September 2000

Oliver Vornberger Olaf Müller

Inhaltsverzeichnis

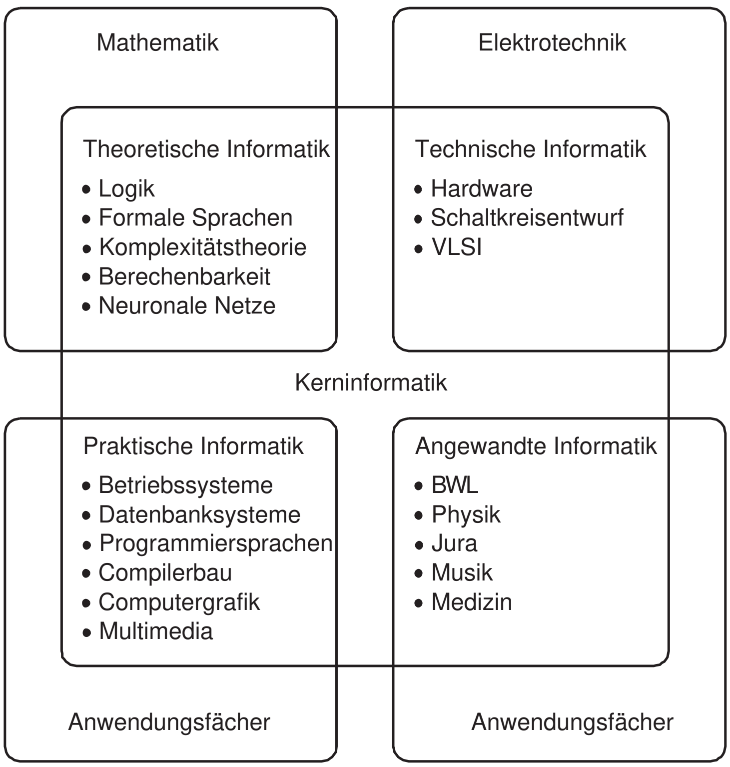
Verzeichnis der Java-Programme

Kapitel 1

Einführung

1.1 Informatik

Wissenschaft von der EDV
Konzepte, unabhängig von Technologie
Formalismus
Interdisziplinärer Charakter



1.2 Algorithmus, Programm, Prozess

Eine endlich lange Vorschrift, bestehend aus Einzelanweisungen, heißt *Algorithmus*.

Beispiele:

Telefonieren:	Hörer abnehmen Geld einwerfen wählen sprechen auflegen
Kochrezept:	Man nehme ...
Bedienungsanleitung:	Mit Schlüssel S die Mutter M auf Platte P festziehen.

Der Durchführende kennt die Bedeutung der Einzelanweisungen; sie werden deterministisch, nicht zufällig abgearbeitet. Endliche Vorschrift bedeutet **nicht** endliche Laufzeit, aber die Beschreibung der Vorschrift muss endlich sein.

Hier: Elementaranweisungen müssen vom Computer verstanden werden.

Programm	$\rightarrow$	Maschinenprogramm
`if (a > b)`	Compiler	`011011101110110111`

Ein für den Compiler formulierter Algorithmus heißt *Programm*.
Ein Programm in Ausführung heißt *Prozess*.

1.3 Anweisungen, Ablaufprotokoll

Elementare Anweisungen

Beispiel: teile x durch 2
erhöhe y um 1

Strukturierte Anweisungen

enthalten Kontrollstruktur, Bedingung, Teilanweisungen.
Beispiele:

```
WENN es tutet
   DANN waehle
   SONST lege auf
```

```
abheben
waehlen
SOLANGE besetzt ist TUE
                      auflegen
                      abheben
                      waehlen
```

Beispiel für einen Algorithmus in umgangssprachlicher Form

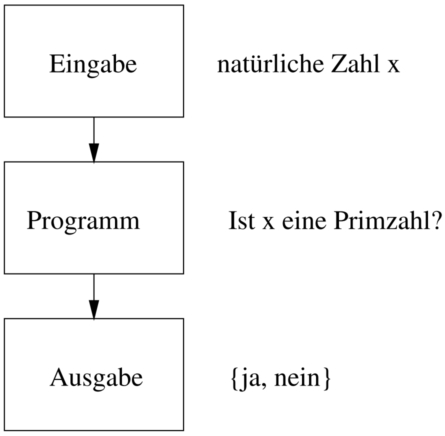
```
lies x
setze z auf 0
SOLANGE x ≠ 1 TUE
   WENN x gerade
      DANN halbiere x
      SONST verdreifache x und erhoehe um 1
   erhoehe z um 1
drucke z
```

Ablaufprotokoll (`trace`)

Zeile	x	z
	undef.	undef.
1	3	undef.
2	3	0
3	10	0
4	10	1
3	5	1
4	5	2
3	16	2
4	16	3
3	8	3
4	8	4
3	4	4
4	4	5
3	2	5
4	2	6
3	1	6
4	1	7
5	Ausgabe	7

Programm iteriert Collatz-Funktion
$f : \mathbb{N} \to \mathbb{N}$
$f(x)$ = Anzahl der Iterationen, um x auf 1 zu transformieren

Typisch: Programm löst Problem



Terminierung, Korrektheit und Effizienz sind nicht algorithmisch zu bestimmen. Dafür ist jeweils eine neue Idee erforderlich.

Kapitel 2

Java

2.1 Sprachmerkmale

```
/***************************  Collatz.java  *********************************/

import AlgoTools.IO;

/**  Berechnet Collatz-Funktion, d.h.
 *   Anzahl der Iterationen der Funktion g: N -> N
 *   bis die Eingabe auf 1 transformiert ist
 *   mit g(x) = x/2 falls x gerade, 3*x+1 sonst
 */

public class Collatz {

  public static void main(String argv[]) {

    int x, zaehler;                            // definiere 2 Variablen
    zaehler = 0;                               // setze zaehler auf 0

    x = IO.readInt("Bitte eine Zahl: ");       // fordere eine Zahl an

    while (x != 1) {                           // solange x ungleich 1 ist
        if (x % 2 == 0)                        // falls x gerade ist
            x = x / 2;                         // teile x durch 2
        else                                   // andernfalls
            x = 3*x+1;                         // nimm x mit 3 mal und add. 1
        zaehler++;                             // erhoehe zaehler um eins
        IO.println(x);                         // gib x aus
    }
    IO.println("Anzahl der Iterationen: " + zaehler);
  }
}
```

Java ist eine objektorientierte Programmiersprache: Die Modellierung der Realwelt erfolgt durch in Klassenhierarchien zusammengefasste Objekte, beschrieben durch Datenfelder und Methoden. Datenfelder sind zur Klasse oder ihren Objekten gehörende Daten; Methoden sind Anweisungsfolgen, die auf den Datenfeldern operieren, um ihren Zustand zu manipulieren.

Der Collatz-Algorithmus als Java-Programm besteht aus der Definition der Klasse `Collatz` mit der Methode `main`. Nach Übersetzung des Programms in den maschinenunabhängigen Bytecode wird die Methode `main` gestartet, sobald die sie umschließende Klasse geladen wurde. Der Quelltext besteht aus durch Wortzwischenräume (Leerzeichen, Tabulatoren, Zeilen- und Seitenvorschubzeichen) getrennte Token. Zur Verbesserung der Lesbarkeit werden Kommentare eingestreut, entweder durch `/*` ... `*/` geschachtelt oder durch `//` angekündigt bis zum Zeilenende. Der Dokumentationsgenerator ordnet den durch `/**` ... `*/` geklammerten Vorspann der nachfolgenden Klasse zu. Die von den Schlüsselwörtern verschieden gewählten Bezeichner beginnen mit einem Buchstaben, Unterstrich (`_`) oder Dollarzeichen (`$`). Darüberhinaus dürfen im weiteren Verlauf des Bezeichners auch Ziffern verwendet werden. Zur Vereinfachung der Ein-/Ausgabe verwenden wir die benutzer-definierte Klasse `AlgoTools.IO` mit den Methoden `readInt` und `println`.

2.2 Variablen

Variablen sind benannte Speicherstellen, deren Inhalte gemäß ihrer vereinbarten Typen interpretiert werden. Java unterstützt folgende "eingebaute" Datentypen (genannt *einfache Datentypen*).

`boolean`	entweder `true` oder `false`
`char`	16 Bit Unicode
`byte`	vorzeichenbehaftete ganze Zahl in 8 Bit
`short`	vorzeichenbehaftete ganze Zahl in 16 Bit
`int`	vorzeichenbehaftete ganze Zahl in 32 Bit
`long`	vorzeichenbehaftete ganze Zahl in 64 Bit
`float`	Gleitkommazahl in 32 Bit
`double`	Gleitkommazahl in 64 Bit

2.3 Kontrollstrukturen

Kontrollstrukturen regeln den dynamischen Ablauf der Anweisungsfolge eines Programms durch Bedingungen, Verzweigungen und Schleifen.

```
/****************************  Bedingung.java  ******************************/

import AlgoTools.IO;

/**  Verzweigung durch Bedingung (if-Anweisung, Bedingungsoperator)
 *
 *   Vergleichsoperatoren:   <    kleiner
 *                           <=   kleiner gleich
 *                           ==   gleich
 *                           >    groesser
 *                           >=   groesser gleich
 *                           !=   ungleich
 *
 *   logische Operatoren:    &&   und
 *                           ||   oder
 *                           !    nicht
 */

public class Bedingung {

  public static void main (String argv[]) {

    int x = -5, y, m;                      // definiere 3 Integer-Variablen
                                           // davon eine initialisiert

    if (x < 0) x = -x;                     // setze x auf Absolutbetrag

    if (x < 0) y = -x; else y = x;         // setze y auf den
                                           // Absolutbetrag von x

    m = IO.readInt("Bitte Monatszahl: ");
    if      ((3 <= m) && (m <=  5)) IO.println("Fruehling");
    else if ((6 <= m) && (m <=  8)) IO.println("Sommer");
    else if ((9 <= m) && (m <= 11)) IO.println("Herbst");
    else if (m==12 || m==1 || m==2) IO.println("Winter");
    else                            IO.println("unbekannte Jahreszeit");

    x = x % 2 == 0 ? x/2 : 3*x + 1;        // bedingter Ausdruck:
                                           // weise der linken Seite
                                           // den Ausdruck vor ':' zu,
                                           // falls Bedingung vor '?' wahr;
  }                                        // sonst Ausdruck hinter ':'
}
```

```
/***************************  Fall.java  ***********************************/

import AlgoTools.IO;

/**  Verzweigung durch Fallunterscheidung (switch/case-Anweisung)
 *
 */

public class Fall {

  public static void main (String argv[]) {

    int zahl  = 42;
    int monat = 11;

   switch (zahl % 10) {// verzweige in Abhaengigkeit der letzten Ziffer von zahl

        case 0:  IO.println("null  "); break;
        case 1:  IO.println("eins  "); break;
        case 2:  IO.println("zwei  "); break;
        case 3:  IO.println("drei  "); break;
        case 4:  IO.println("vier  "); break;
        case 5:  IO.println("fuenf "); break;
        case 6:  IO.println("sechs "); break;
        case 7:  IO.println("sieben"); break;
        case 8:  IO.println("acht  "); break;
        case 9:  IO.println("neun  "); break;
    }


    switch(monat) {                   // verzweige in Abhaengigkeit von monat

        case  3: case  4: case  5:   IO.println("Fruehling"); break;
        case  6: case  7: case  8:   IO.println("Sommer   "); break;
        case  9: case 10: case 11:   IO.println("Herbst   "); break;
        case 12: case  1: case  2:   IO.println("Winter   "); break;
        default:                     IO.println("unbekannte Jahreszeit");
    }

  }
}
```

```
/***************************  Schleife.java  *********************************/

import AlgoTools.IO;

/**  for-Schleife, while-Schleife, do-while-Schleife, break, continue
 */

public class Schleife {

  public static void main (String argv[]) {

    int i, x=10, y=2, summe;                 // 4 Integer-Variablen

    for (i=1; i<=10; i++) IO.println(i*i,6);  // drucke 10 Quadratzahlen

    while (x > 0) {                           // solange x groesser als 0
        x--;                                  // erniedrige x um eins
        y = y + 2;                            // erhoehe y um zwei
    }

    do {
        x++;                                  // erhoehe x um eins
        y += 2;                               // erhoehe y um 2
    } while (x < 10);                         // solange x kleiner als 10

    while (true) {                            // auf immer und ewig
        x /= 2;                               // teile x durch 2
        if (x == 1) break;                    // falls x=1 verlasse Schleife
        if (x % 2 == 0) continue;             // falls x gerade starte Schleife
        x = 3*x + 1;                          // verdreifache x und erhoehe um 1
    }

    IO.println("Bitte Zahlen eingeben. 0 als Abbruch");
    summe = 0;                                // initialisiere summe
    x = IO.readInt();                         // lies x ein
    while (x != 0) {                          // solange x ungleich 0 ist
        summe += x;                           // erhoehe summe
        x = IO.readInt();                     // lies x ein
    }
    IO.println("Die Summe lautet " + summe);

    do {
        IO.print("Bitte Monatszahl: ");       // Aufforderung
        x = IO.readInt();                     // lies x ein
    } while (( x < 1) || (x > 12));           // solange x unzulaessig
  }
}
```

```
/**************************  Fakultaet.java  ********************************/

import AlgoTools.IO;

/**  Berechnung der Fakultaet mit for-, while- und do-while-Schleifen
  *
  *  n! :=   1 fuer n=0,
  *          1*2*3* ... *n sonst
  *
  */

public class Fakultaet {

  public static void main (String argv[]) {

    int i, n, fakultaet;                   // 3 Integer-Variablen

    n = IO.readInt("Bitte Zahl: ");        // fordere Zahl an


    fakultaet = 1;                         // berechne n! mit for-Schleife
    for (i = 1; i <= n; i++)
        fakultaet = fakultaet * i;
    IO.println(n + " ! = " + fakultaet);


    fakultaet = 1;                         // berechne n! mit while-Schleife
    i = 1;
    while (i <= n) {
        fakultaet = fakultaet * i;
        i++;
    }
    IO.println(n + " ! = " + fakultaet);


    fakultaet = 1;                         // berechne n! mit do-while-Schleife
    i = 1;
    do {
        fakultaet = fakultaet * i;
        i++;
    } while (i <= n);
    IO.println(n + " ! = " + fakultaet);
  }
}
```

```
/***************************  GGT.java  **************************************/

import AlgoTools.IO;

/**  Berechnung des GGT
 *
 *   ggt(x,y) =     groesster gemeinsamer Teiler von x und y
 *
 *                  x                 falls x = y
 *   ggt(x,y) =     ggt(x-y, y)       falls x > y
 *                  ggt(x, y-x)       falls y > x
 *
 *                  denn wegen x=t*f1 und y=t*f2 folgt (x-y) = t*(f1-f2)
 *
 *                  x                 falls y = 0
 *   ggt(x,y) =     ggt(y, x mod y)   sonst
 *
 */

public class GGT {

    public static void main (String argv[]) {

        int teiler, a, b, x, y, z;                   // 6 Integer-Variablen

        IO.println("Bitte zwei Zahlen: ");
        a=x=IO.readInt();  b=y=IO.readInt();         // lies 2 Zahlen ein

        teiler = x;                                  // beginne mit einem teiler
        while ((x % teiler != 0) ||                  // solange x nicht aufgeht
               (y % teiler != 0))                    // oder y nicht aufgeht
            teiler--;                                // probiere Naechstkleineren
        IO.println("GGT = " + teiler);

        while (a != b)                               // solange a ungleich b
            if (a > b) a = a - b;                    // subtrahiere die kleinere
            else       b = b - a;                    // Zahl von der groesseren
        IO.println("GGT = " + a);

        while (y != 0) {                             // solange y ungleich 0
            z = x % y;                               // ersetze x durch y
            x = y;                                   // und y durch x modulo y
            y = z;
        }
        IO.println("GGT = " + x);
    }
}
```

2.4 Einfache Datentypen

Der Datentyp legt fest:

- Wertebereich,
- Operationen.

Die Implementierung verlangt:

- Codierung.

einfach = von der Programmiersprache vorgegeben.

2.4.1 Ganze Zahlen (`byte`, `short`, `int`, `long`)

Wertebereich: ganze Zahlen darstellbar in 8, 16, 32, 64 Bits.

Typ	Wertebereich	Länge
`byte`	`-128..127`	8 Bit
`short`	`-32768..32767`	16 Bit
`int`	`-2147483648..2147483647`	32 Bit
`long`	`-9223372036854775808..9223372036854775807`	64 Bit

Codierung

Codierung der positiven Zahlen in Dualzahldarstellung:
Sei

$$x = \sum_{i=0}^{n-1} d_i \cdot 2^i$$

Algorithmus dezimal → dual:

```
while (x != 0){
   if (x%2 == 0) IO.print('0');
            else IO.print('1');
   x = x/2;
}
```

Obacht: Bits werden rückwärts generiert!

Codierung der ganzen Zahlen im 2-er Komplement:

d_3	d_2	d_1	d_0	x
0	1	1	1	7
0	1	1	0	6
0	1	0	1	5
0	1	0	0	4
0	0	1	1	3
0	0	1	0	2
0	0	0	1	1
0	0	0	0	0
1	1	1	1	-1
1	1	1	0	-2
1	1	0	1	-3
1	1	0	0	-4
1	0	1	1	-5
1	0	1	0	-6
1	0	0	1	-7
1	0	0	0	-8

Beispiel zur Berechnung des 2-er Komplements einer negativen Zahl:

Gegeben $-x$	-4
Finde d_i zu x	0100
Invertiere Bits	1011
Addiere 1	1100

Vorteil: Nur ein Addierwerk!

```
    0011    3
  + 1011   -5
  ------------
  = 1110   -2
```

Subtraktion mittels Negierung auf Addition zurückführen. Obacht: Überlauf beachten!

```
    0111    7
  + 0001    1
  --------------------
  = 1000   -8   falsch
```

Trick: Vorzeichenbits verdoppeln, müssen nach der Verknüpfung identisch sein:

00111	7	00011	3
+ 00001	1	11011	-5
01000		11110	-2
Ergebnis undefiniert		Ergebnis ok!	

Operatoren

`+`	:	`Ganzzahl`	$\times$	`Ganzzahl`	$\rightarrow$	`Ganzzahl`	Addition
`-`	:	`Ganzzahl`	$\times$	`Ganzzahl`	$\rightarrow$	`Ganzzahl`	Subtraktion
`*`	:	`Ganzzahl`	$\times$	`Ganzzahl`	$\rightarrow$	`Ganzzahl`	Multiplikation
`/`	:	`Ganzzahl`	$\times$	`Ganzzahl`	$\rightarrow$	`Ganzzahl`	ganzzahlige Division
`%`	:	`Ganzzahl`	$\times$	`Ganzzahl`	$\rightarrow$	`Ganzzahl`	Modulo = Rest der Division
`&`	:	`Ganzzahl`	$\times$	`Ganzzahl`	$\rightarrow$	`Ganzzahl`	bitweise Und-Verknüpfung
`|`	:	`Ganzzahl`	$\times$	`Ganzzahl`	$\rightarrow$	`Ganzzahl`	bitweise Oder-Verknüpfung
`^`	:	`Ganzzahl`	$\times$	`Ganzzahl`	$\rightarrow$	`Ganzzahl`	bitweise XOR-Verknüpfung
`<<`	:	`Ganzzahl`	$\times$	`Ganzzahl`	$\rightarrow$	`Ganzzahl`	Linksshift
`>>`	:	`Ganzzahl`	$\times$	`Ganzzahl`	$\rightarrow$	`Ganzzahl`	Vorzeichen erhaltender Rechtsshift
`>>>`	:	`Ganzzahl`	$\times$	`Ganzzahl`	$\rightarrow$	`Ganzzahl`	Vorzeichen ignorierender Rechtsshift
`~`	:	`Ganzzahl`			$\rightarrow$	`Ganzzahl`	Bitweise Negation
`-`	:	`Ganzzahl`			$\rightarrow$	`Ganzzahl`	Vorzeichenumkehr
`++`	:	`Ganzzahl`			$\rightarrow$	`Ganzzahl`	Inkrement
`--`	:	`Ganzzahl`			$\rightarrow$	`Ganzzahl`	Dekrement

Konstantenbezeichner

`123` `+123` `-123`

Eine führende Null kündigt eine Oktalzahl zur Basis `8` an: `0173`. Eine führende Null mit nachfolgendem `x` oder `X` kündigt eine Hexadezimalzahl zur Basis `16` an: `0x7B`.

2.4.2 Gleitkommazahlen (`float`, `double`)

Die Typen `float` und `double` dienen zur Speicherung von Gleitkommazahlen nach dem IEEE-Standard 754-1985 in der Form Vorzeichen, Exponent, Mantisse.

Codierung

Bei der Codierung für 32 Bits (`float`) benötigt das Vorzeichen 1 Bit, der Exponent 8 Bits, die Mantisse 23 Bits.

X	XXXXXXXX	XXXXXXXXXXXXXXXXXXXXXXX
Vorzeichen s	Exponent e	Mantisse f

$$\text{Wert} = \begin{cases} (-1)^s \times 2^{e-127} \times 1.f & \text{falls } 0 < e < 255 \quad \text{(normalisiert)} \\ (-1)^s \times 2^{-126} \times 0.f & \text{falls } e = 0 \quad \text{(subnormal)} \end{cases}$$

Bei einer normalisierten Darstellung liegt der Wert der Mantisse $1.f$ im Intervall $[1, 2[$.

Hierbei haben Dualzahlen nach dem Komma

$$0.d_{-1}d_{-2}d_{-3}\ldots d_{-k} \text{ die Bedeutung } \sum_{i=-k}^{-1} d_i \cdot 2^i$$

Algorithmus dezimal $\rightarrow$ dual:

Sei Dezimalzahl x gegeben. Bestimme größte 2-er Potenz 2^d mit $2^d \leq x$. Setze $f = (x - 2^d)/2^d$. Offenbar gilt

$$x = (1 + f) \cdot 2^d \text{ mit } 0 \leq f < 1.$$

Bestimme die Dualzahl-Bits von f durch

```
for (i = 0; i < 23; i++) {
   f = f * 2.0;
   if (f >= 1.0) {IO.print('1'); f = f - 1.0;}
      else IO.print('0');
}
```

Beispiel: Sei $x = 13.5$ gegeben. Als Codierung ergibt sich

$$\begin{aligned} s &= 0 \\ f &= (13.5 - 2^3)/2^3 = 0.6875 = \frac{1}{2} + \frac{1}{8} + \frac{1}{16} \\ e &= 3 + 127 = 130 \end{aligned}$$

0	10000010	10110000000000000000000
Vorzeichen	Exponent	Mantisse

Bei einer subnormalisierten Darstellung liegt der Wert der Mantisse $0.f$ im Intervall $[0, 1[$.

Exponent $e = 0$ und Mantisse $f = 0$ repräsentieren die vorzeichenbehaftete Null. Exponent $e = 255$ und Mantisse $f = 0$ repräsentieren das vorzeichenbehaftete Unendlich$(+\infty, -\infty)$. Exponent $e = 255$ und Mantisse $f \neq 0$ repräsentieren die undefinierte Zahl `NaN` (not a number).

Beispiel: $2^{-130} = 1/16 \cdot 2^{-126}$

Daraus folgt

$$\begin{aligned} s &= 0 \\ f &= 0.0625 = 1/16 \\ e &= 0 \end{aligned}$$

Codierung

`0`	`00000000`	`00010000000000000000000`
Vorzeichen	Exponent	Mantisse

Die größte darstellbare positive Zahl liegt knapp unter

$$2 \cdot 2^{254-127} = 2^{128} \approx 10^{38}$$

`0`	`11111110`	`11111111111111111111111`
Vorzeichen	Exponent	Mantisse

Die kleinste darstellbare positive Zahl lautet

$$2^{-23} \cdot 2^{-126} = 2^{-149} \approx 10^{-45}$$

`0`	`00000000`	`00000000000000000000001`
Vorzeichen	Exponent	Mantisse

Bei der Codierung für 64 Bits (`double`) benötigt das Vorzeichen 1 Bit, der Exponent 11 Bits, die Mantisse 52 Bits.

`X`	`XXXXXXXXXXX`	`XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX`
Vorzeichen s	Exponent e	Mantisse f

$$\text{Wert} = \begin{cases} (-1)^s \times 2^{e-1023} \times 1.f & \text{falls } 0 < e < 2047 \quad \text{(normalisiert)} \\ (-1)^s \times 2^{-1022} \times 0.f & \text{falls } e = 0 \quad \text{(subnormal)} \end{cases}$$

Damit liegt die größte darstellbare positive Zahl knapp unter

$$2 \cdot 2^{2046-1023} \approx 10^{308}$$

Die kleinste darstellbare positive Zahl lautet

$$2^{-52} \cdot 2^{-1022} = 2^{-1074} \approx 10^{-324}$$

Operatoren

`+`	:	`Gleitkomma` × `Gleitkomma`	→	`Gleitkomma`	Addition
`-`	:	`Gleitkomma` × `Gleitkomma`	→	`Gleitkomma`	Subtraktion
`*`	:	`Gleitkomma` × `Gleitkomma`	→	`Gleitkomma`	Multiplikation
`/`	:	`Gleitkomma` × `Gleitkomma`	→	`Gleitkomma`	Division
`%`	:	`Gleitkomma` × `Gleitkomma`	→	`Gleitkomma`	Modulo
`++`	:	`Gleitkomma`	→	`Gleitkomma`	Inkrement um 1.0
`--`	:	`Gleitkomma`	→	`Gleitkomma`	Dekrement um 1.0

Multiplikation: (Exponenten addieren, Mantissen multiplizieren)

Beispiel:

$$\begin{array}{ccccc} 12 & * & 20 & = & \\ 1.5\cdot 2^3 & * & 1.25\cdot 2^4 & = & \\ 1.5\cdot 1.25 & * & 2^3\cdot 2^4 & = & \\ 1.875 & * & 2^7 & = & 240 \end{array}$$

Addition: (Exponenten angleichen, Mantissen addieren)

Beispiel:

$$\begin{array}{ccccc} 12 & + & 20 & = & \\ 1.5\cdot 2^3 & + & 1.25\cdot 2^4 & = & \\ 0.75\cdot 2^4 & + & 1.25\cdot 2^4 & = & \\ (0.75+1.25) & * & 2^4 & = & \\ 1 & * & 2^5 & = & 32 \end{array}$$

Problem beim Angleichen der Exponenten:

Beispiel:

$$\begin{array}{ccccc} 1024 & + & \frac{1}{1048576} & = & \\ 1\cdot 2^{10} & + & 1\cdot 2^{-20} & = & \\ 1\cdot 2^{10} & + & 2^{-30}\cdot 2^{10} & = & \\ 1\cdot 2^{10} & + & 0\cdot 2^{10} & = & 1024 \end{array}$$

Bei 23 Bits für die Mantisse ist 2^{-30} nicht mehr darstellbar.

Gleitkommaoperationen stoßen in Java keine Ausnahmebehandlung an. D.h., Division durch Null führt nicht zum Abbruch, sondern ergibt den Wert $+\infty$ bzw. $-\infty$; Null dividiert durch Null ergibt `NaN` (not a number).

Konstantenbezeichner

Beispiele:

`.2`				
`2`				
`2.`				
`2.0`				
`2.538`				
`2.538f`				
`2.5E2`	=	$2.5 * 10^2$	=	250
`2.5E-2`	=	$2.5 * 10^{-2}$	=	0.025

```
/*************************  Gleitkomma.java  *******************************/

import AlgoTools.IO;

/**  Gleitkommaoperationen
 */

public class Gleitkomma {

  public static void main (String argv[]) {

    double summe, summand, pi, nenner, x;  // fuenf 64-Bit-Gleitkommazahlen

    IO.println("1/2 + 1/4 + 1/8 + ...");
    summe = 0.0; summand = 1.0;            // Initialisierungen
    while (summand > 0.0000000000001) {    // solange Summand gross genug
        summand = summand / 2.0;           // teile Summand
        summe   = summe + summand;         // erhoehe Summe
        IO.println(summe, 22, 16);         // drucke Summe auf insgesamt
    }                                      // 22 Stellen mit 16 Nachkommastellen

    IO.println("Naeherung von Pi :");
    pi = 4.0;                              // setze Naeherung auf 4
    nenner = 1.0;                          // setze nenner auf 1
    for (int i=1; i <= 100; i++) {         // tue 100 mal
        nenner += 2.0;                     // erhoehe nenner um 2
        summand = 4.0 / nenner;            // bestimme naechsten summand
        if (i%2 == 1) summand = -summand;  // abwechselnd positiv/negativ
        pi += summand;                     // addiere summand auf Naeherung
        IO.println( pi, 22, 16);           // drucke Naeherung auf insgesamt
                                           // 22 Stellen, 16 Nachkommastellen
    }

    IO.println("Ueberlaufdemo :");
    x = 1.0;                               // setze x auf 1
    while (x < Double.MAX_VALUE) {         // solange kleiner als Maximum
        x = x * 100.0;                     // verhundertfache x
       IO.println(x, 22);                  // drucke 22-stellig
    }                                      // druckt als letzten Wert Infinity

    IO.println( -1.0 / 0.0 );              // ergibt -Infinity
    IO.println(  0.0 / 0.0 );              // ergibt NaN (not a number)

  }
}
```

2.4.3 Boolean (`boolean`)

Der Typ `boolean` dient zum Speichern der logischen Werte wahr und falsch. Die einzigen Konstantenbezeichner lauten `true` und `false`.

Codierung

Mögliche Codierung in einem Byte:

`false`	=	0
`true`	=	1

Operatoren

`&&`	:	`boolean`	×	`boolean`	→	`boolean`	logisches Und mit verkürzter Auswertung
`||`	:	`boolean`	×	`boolean`	→	`boolean`	logisches Oder mit verkürzter Auswertung
`&`	:	`boolean`	×	`boolean`	→	`boolean`	logisches Und mit vollständiger Auswertung
`|`	:	`boolean`	×	`boolean`	→	`boolean`	logisches Oder mit vollständiger Auswertung
`^`	:	`boolean`	×	`boolean`	→	`boolean`	Exklusiv-Oder
`==`	:	`boolean`	×	`boolean`	→	`boolean`	Gleichheit
`!=`	:	`boolean`	×	`boolean`	→	`boolean`	Ungleichheit
`!`	:	`boolean`			→	`boolean`	Negation

P	Q	P && Q	P || Q	P ^ Q	!Q
false	false	false	false	false	true
false	true	false	true	true	false
true	false	false	true	true	
true	true	true	true	false	

Verkürzte Auswertung erfolgt von links nach rechts und bricht frühstmöglich ab:

```
while ((t > 0) && (n % t != b)) {
   t = t - 1;
}
```

De Morgan'sche Regeln:

```
(!p) && (!q) = !(p || q)
(!p) || (!q) = !(p && q)
```

2.4.4 Charakter (`char`)

Wertebereich: alle Zeichen im 16 Bit breiten Unicode-Zeichensatz.

Codierung

Jedes Zeichen wird codiert durch eine 2 Byte breite Zahl, z.B. der Buchstabe `A` hat die Nummer `65` (dezimal) = `00000000 01000001` (binär). Die niederwertigen 7 Bits stimmen mit dem ASCII-Zeichensatz (American Standard Code for Information Interchange) überein.

Zeichen-Literale werden zwischen zwei Apostrophen geschrieben, z.B. `'Q'` oder `'5'` oder `'?'`. Einige Sonderzeichen können durch eine Escape-Sequenz ausgedrückt werden.

Escape-Sequenz	Bedeutung	Codierung
`\n`	Zeilenvorschub mit Wagenrücklauf	10
`\t`	Tabulator	9
`\b`	Backspace	8
`\r`	Wagenrücklauf	13
`\f`	Zeilenvorschub ohne Wagenrücklauf	12
`\\`	Backslash	92
`\'`	Apostroph	39
`\"`	Anführungsstriche	34
`\015`	Wagenrücklauf in Oktal-Schreibweise	13
`\u000D`	Wagenrücklauf in Hexadezimal-Schreibweise	13

Operatoren

`<, <=, ==, >=, >, != : char` $\times$ `char` $\rightarrow$ `boolean`

verlangen Ordnung auf dem Zeichensatz!

Es gilt `0 < 1 <...< 9 <...< A < B <...< Z <...< a < b <...< z < ...`

```
c = IO.readChar();
if ((('A' <= c) && (c <= 'Z')) ||
    (('a' <= c) && (c <= 'z')))
    IO.print ("Das Zeichen ist ein Buchstabe");
```

```
/*************************  Zeichen.java  **********************************/

import AlgoTools.IO;

/**  Umwandlung von Character zur Zahl
 *   Umwandlung von Zahl zum Character
 */

public class Zeichen {

  public static void main (String argv[]) {

    for (int i=0; i<=255; i++) {                // fuer den Latin-1-Zeichensatz
        IO.print(i,7);                          // drucke laufende Nummer
        IO.print( " " + (char) i );             // drucke zugehoeriges Zeichen
        if (i % 8 == 0) IO.println();           // nach 8 Zeichen neue Zeile
    }
    IO.println();

    char c;
    do {
        c = IO.readChar("Bitte ein Zeichen: "); // lies ein Zeichen ein
        IO.print("Der ASCII-Code lautet : ");   // gib den zugehoerigen
        IO.println((int) c,3);                  // ASCII-Code aus
    } while (c != 10 );                         // bis ein Newline kommt
  }
}
```

2.4.5 Typumwandlung

Der Typ eines Ausdrucks wird durch seine Bestandteile und die Semantik seiner Operatoren bestimmt. Grundsätzlich werden sichere, d.h. verlustfreie, Umwandlungen implizit, d.h. automatisch, ausgeführt. Konvertierungen, die ggf. verlustbehaftet sind, verlangen einen expliziten Cast-Operator.

Arithmetische Operationen auf ganzzahligen Werten liefern immer den Typ `int`, es sei denn, einer oder beide Operanden sind vom Typ `long`, dann ist das Ergebnis vom Typ `long`.

Die kleineren Integer-Typen `byte` und `short` werden vor einer Verknüpfung auf `int` umgewandelt. Character-Variablen lassen sich implizit konvertieren.

Ist ein Operand in Gleitkommadarstellung, so wird die Operation in Gleitkomma-Arithmetik durchgeführt. Gleitkommakonstanten ohne Suffix sind vom Typ `double` und erfordern eine explizite Typumwandlung (cast), wenn sie einer `float`-Variable zugewiesen werden.

```
/*************************  Ganzzahl.java  *********************************/

import AlgoTools.IO;

/**  Typumwandlung und Bitoperationen auf ganzen Zahlen
 */

public class Ganzzahl {

  public static void main (String argv[]) {

    byte b;                                   // ganze Zahl mit  8 Bit
    int  n;                                   // ganze Zahl mit 32 Bit

    b = -127;                                 // setze b auf -127;
    n = b;                                    // implizite Typumwandlung
                                              // n hat nun den Wert -127
                                              // codiert auf 4 Bytes

    n = 130;                                  // setze n auf 130
    b = (byte) n;                             // explizite Typumwandlung
    IO.println(b);                            // b hat nun den Wert -126
                                              // codiert auf einem Byte

    n = -2 ;                                  // setze n auf -2
    b = (byte) n;                             // explizite Typumwandlung
    IO.println(b);                            // b hat nun den Wert -2
                                              // codiert auf einem Byte

    n = 1;                                    // initialisiere n
    while (n > 0) {                           // solange n positiv ist
        n = n * 10;                           // verzehnfache n
        IO.println(n, 20);                    // drucke n auf 20 Stellen
    }                                         // letzter Wert ist negativ !

    byte x=-127, y=126;
                                              // Obacht: jeweils cast auf int !
    IO.println( x & y  );                     // bitweise Und-Verknuepfung
    IO.println( x | y  );                     // bitweise Oder-Verknuepfung
    IO.println( x ^ y  );                     // bitweise XOR-Verknuepfung
    IO.println( ~x     );                     // bitweise Negation
    IO.println( x <<  6);                     // Linksshift
    IO.println( x >>  6);                     // Vorzeichen erhalt. Rechtsshift
    IO.println( x >>> 6);                     // Vorzeichen ignor.  Rechtsshift
  }                                           // d.h. Nullen nachschieben
}
```

```java
/*************************  Umwandlung.java  ********************************/

import AlgoTools.IO;

/**  implizite und explizite Typumwandlungen zwischen einfachen Datentypen
 */


public class Umwandlung {

  public static void main (String argv[]) {

    char   c = '?';                  // das Fragezeichen
    byte   b = 100;                  // ca.  3 Stellen
    short  s = 10000;                // ca.  5 Stellen
    int    i = 1000000000;           // ca. 10 Stellen
    long   l = 100000000000000L;     // ca. 15 Stellen
    float  f = 3.14159f;             // ca.  6 Stellen Genauigkeit
    double d = 3.14159265358979;     // ca. 15 Stellen Genauigkeit

    i = s ;                          // implizite Typumwandlung ohne Verlust
    i = c ;                          // implizite Typumwandlung ohne Verlust
    s = (short) c;                   // explizite Typumwandlung ohne Verlust
    s = (short) i;                   // explizite Typumwandlung  mit Verlust
    d = i;                           // implizite Typumwandlung ohne Verlust
    i = (int) d;                     // explizite Typumwandlung  mit Verlust

    d = f;                           // implizite Typumwandlung ohne Verlust
    f = (float) d;                   // explizite Typumwandlung  mit Verlust

    d = 1/10;                        // ergibt 0 wegen ganzzahliger Division
    IO.println(d);
    d = 1/10.0;                      // ergibt 0.1 wegen Gleitkommadivision
    IO.println(d);

    IO.println(2 * b + c );          // Ausdruck ist vom Typ int, Wert = 263
    IO.println(i + 1.5);             // Ausdruck ist vom Typ double
  }
}
```

2.4.6 Konstanten

Benannte Konstanten können innerhalb einer Klasse zur Verbesserung der Lesbarkeit benutzt werden. Sie werden bei der Deklaration mit dem Attribut `final` markiert und erhalten ihren nicht mehr änderbaren Wert zugewiesen.

```
/*************************  Konstanten.java  ********************************/

import AlgoTools.IO;

/**  Einsatz von Konstanten fuer ganze Zahlen, Gleitkomma und Character
 */


public class Konstanten {

  public static void main (String argv[]) {

    final int    MAX_GRAD = 360;                // Integer-Konstante
    final double PI       = 3.141592653589793; // Double Konstante
    final char   PIEP     = 7;                  // Character-Konstante

    int grad;
    double r;

    IO.print(PIEP);                             // erzeuge Piep

    for (grad=0; grad <= MAX_GRAD; grad++) {   // fuer jeden ganzzahligen Winkel
        r = Math.sin (grad * PI / 180);         // berechne Sinus vom Bogenmass
        IO.println(grad + " " + r);             // gib Winkel und Sinus aus
    }
  }
}
```

Kapitel 3

Felder

Mehrere Daten desselben Typs können zu einem Feld (Array) zusammengefasst werden.

```
/***************************  Feld.java  **************************************/

import AlgoTools.IO;

/**  Zusammenfassung mehrerer Daten desselben Typs zu einem Feld
 */

public class Feld {

  public static void main (String argv[]) {

    double[] a;                                 // eindimensionales double-Feld
    int i, j;                                   // Laufvariablen

    a = new double[8];                          // besorge Platz fuer 8 Double

    for (i=0; i < a.length; i++)                // durchlaufe das Feld
        IO.println(a[i]);                       // drucke jedes Feldelement

    double[][] mat = new double[4][3];          // 4x3 Matrix vom Typ double

    for (i=0; i<mat.length; i++) {              // durchlaufe jede Zeile
        for (j=0; j < mat[i].length; j++)       // durchlaufe die Spalten
            IO.print(mat[i][j], 8, 2);          // drucke Matrixelement
        IO.println();                           // gehe auf neue Zeile
    }

    int[][] diagonal = {{1}, {2,3}, {4,5,6}};   // untere linke Dreiecksmatrix
  }
}
```

3.1 Feld von Ziffern

```
/**************************  Ziffern.java  **********************************/

import AlgoTools.IO;

/**  Erwartet eine Folge von Ziffern
 *   und berechnet den zugehoerigen Wert.
 */

public class Ziffern {

  public static void main (String argv[]) {

    char[] zeile;                           // Array fuer Zeile
    char c;                                 // Character-Variable

    int i = 0, wert = 0, ziffernwert;       // Hilfsvariablen

    zeile = IO.readChars("Bitte Ziffernfolge: "); // Liest so lange Zeichen von
                                            // der Tastatur, bis <Return>
    i = 0;
    while (i < zeile.length)   {            // solange Zeile nicht erschoepft
        c = zeile[i];                       // besorge naechstes Zeichen
        ziffernwert = (int) c - (int) '0';  // konvertiere Ziffer zur Zahl
        wert = 10*wert + ziffernwert;       // verknuepfe mit bisherigem Wert
        i++;
    }
    IO.println("Der Wert lautet " + wert);  // gib Wert aus
  }
}
```

3.2 Feld von Daten

```
/***************************  Matrix.java  ***********************************/

import AlgoTools.IO;

/** Multiplikation zweier NxN-Matrizen (ohne Ein- und Ausgabe)
 *
 *       c[i][j]  :=  Summe {k=0 bis N-1}  ( a[i][k] * b[k][j] )
 */

public class Matrix {

  public static void main (String argv[]) {

    final int N = 4;                         // Zeilen- und Spaltenzahl

    double[][] c = new double[N][N];         // eine 4x4 Matrix mit Speicherplatz

    double[][] a = {{  1,  2,  3,  4},       // eine initialisierte 4x4 Matrix
                    {  5,  6,  7,  8},
                    {  9, 10, 11, 12},
                    { 13, 14, 15, 16}};

    double[][] b = {{ 17, 18, 19, 20},       // eine initialisierte 4x4 Matrix
                    { 21, 22, 23, 24},
                    { 25, 26, 27, 28},
                    { 29, 30, 31, 32}};


    int i, j, k;                             // Laufindizes
    double summe;                            // Zwischensumme


    for (i = 0; i < N; i++)                  // Zeilenindex
        for (j = 0; j < N; j++) {            // Spaltenindex
            summe = 0.0;
            for (k = 0; k < N; k++)          // verknuepft werden alle Komponenten
                summe += a[i][k]*b[k][j];    // von Zeile i und Spalte j
            c[i][j] = summe;                 // die Produktsumme steht in c[i,j]
        }
  }
}
```

3.3 Feld von Zeichen

```
/***************************  Zeichenkette.java  *****************************/

import AlgoTools.IO;

/**  Interpretiert zwei eingelesene Zeichenfolgen als Strings
 *   und vergleicht sie.
 */

public class Zeichenkette {

  public static void main (String argv[]) {

    char[] s, t;                          // Felder von Zeichen
    int i;                                // Laufindex

    s = IO.readChars("Bitte einen String: ");
    t = IO.readChars("Bitte einen String: ");

    i = 0;

    while (true) {                        // s und t gelten so lange als gleich,
                                          // bis ein String zu Ende ist oder
                                          // zwei Buchstaben sich unterscheiden

      if ((i==s.length) && (i==t.length)) { IO.println(s + " = " + t); break;}
      if ( i==s.length)                   { IO.println(s + " < " + t); break;}
      if ( i==t.length)                   { IO.println(s + " > " + t); break;}
      if ( s[i] < t[i])                   { IO.println(s + " < " + t); break;}
      if ( s[i] > t[i])                   { IO.println(s + " > " + t); break;}

      i++;
    }
  }
}
```

3.4 Feld von Wahrheitswerten

```
/***************************  Sieb.java  *************************************/

import AlgoTools.IO;

/**  Sieb des Eratosthenes zur Ermittlung von Primzahlen.
 *   Idee: Streiche alle Vielfachen von bereits als prim erkannten Zahlen.
 */

public class Sieb {

  public static void main (String argv[]) {

    boolean[] sieb;                             // boole'sches Array
    int i, j, n;                                // Laufvariablen

    n = IO.readInt("Bitte Primzahlgrenze: "); // fordere Obergrenze an
    sieb = new boolean[n];                      // allokiere Speicherplatz

    for (i = 2; i < n; i++) sieb[i] = true;   // alle sind zunaechst Primzahl

    for (i = 2; i < n; i++)
        if (sieb[i]) {                          // falls i als Primzahl erkannt,
            IO.print(i,6);                      // gib sie aus, und
            for (j = i+i; j < n; j += i)        // fuer alle Vielfachen j von i
                sieb[j] = false;                // streiche j aus der Liste
        }
  }
}
```

3.5 Feld von Indizes

```
/***************************  ArrayAbzaehlreim.java  ***************************/

import AlgoTools.IO;

/**  n Kinder stehen im Kreis, jedes k-te wird abgeschlagen.
 *   Die Kreisordnung wird organisiert durch Array mit Indizes der Nachfolger.
 */

public class ArrayAbzaehlreim {

  public static void main (String argv[]) {

    int[] next;                             // Array mit Indizes
    int i, index, n, k;                     // Laufvariablen

    n = IO.readInt("Wie viele Kinder ? "); // fordere Zahl der Kinder
    k = IO.readInt("Wie viele Silben ? "); // fordere Zahl der Silben an
    next = new int [n];                     // allokiere Platz fuer Index-Array

    for (i = 0; i < n; i++)                 // initiale Aufstellung
        next[i] = (i+1) % n;

    index = n-1;                            // index zeigt auf das Kind vor
                                            // dem ausgezeichneten Kind

    while (next[index] != index) {          // so lange abschlagen, bis jemand
                                            // sein eigener Nachfolger ist
        for (i=1; i< k; i++)                // gehe k-1 mal
            index = next[index];            // zum jeweiligen Nachfolger

        IO.print("Ausgeschieden: ");        // gib den Index des ausgeschiedenen
        IO.println(next[index],5);          // naechsten Nachfolgers aus

        next[index] = next[next[index]];    // setze Index auf neuen Nachfolger
    }
    IO.println("Es bleibt uebrig: " + index);
  }
}
```

3.6 Feld von Zuständen

Ein *endlicher Automat* A ist ein 5-Tupel $A = (S, \Sigma, \delta, s_0, F)$ mit

S	$=$	endliche Zustandsmenge
Σ	$=$	endliches Eingabealphabet
$\delta : S \times \Sigma \to S$	$=$	Überführungsfunktion
$s_0 \in S$	$=$	Anfangs- oder Startzustand
$F \subseteq S$	$=$	Endzustände

Ein Wort $w \in \Sigma^*$ wird akzeptiert, falls

$$\delta^*(s_0, w) \in F$$
$$\delta^*(s_0, x_0 x_1 x_2 \ldots x_k) = \delta(\ldots\delta(\delta(\delta(s_0, x_0), x_1), x_2)\ldots x_k).$$

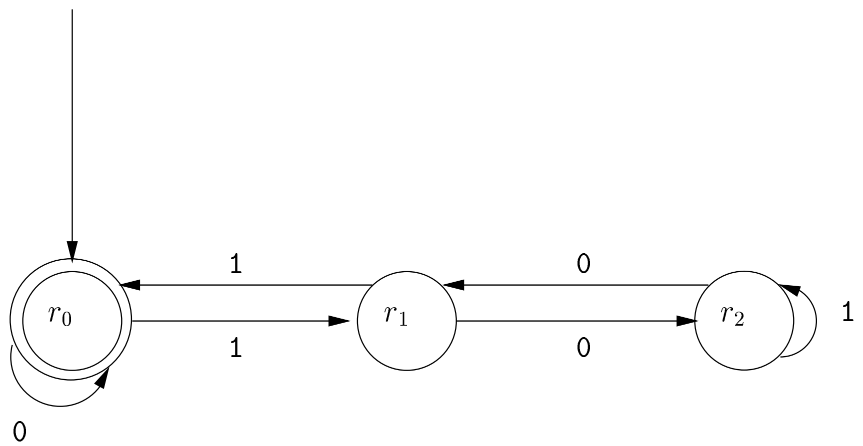
Die Wirkungsweise der Überführungsfunktion δ kann durch einen Zustandsüberführungsgraphen beschrieben werden. In diesem Graphen führt eine mit x beschriftete Kante von Knoten a zu Knoten b, falls gilt: $\delta(a, x) = b$.

Beispiel:

A soll durch 3 teilbare Dualzahlen erkennen.

$S = \{r_0, r_1, r_2\}$
$\Sigma = \{$`'0', '1'`$\}$
Startzustand ist r_0
$F = \{r_0\}$

Die Knoten r_0, r_1 und r_2 charakterisieren die Zustände, wenn der Rest der Division durch 3 0, 1 bzw. 2 beträgt. An der Kante steht das jeweils vorderste Bit der Dualzahl, die von links nach rechts abgearbeitet wird.



```
/***************************  Automat.java  **********************************/

import AlgoTools.IO;

/**  Endlicher Automat mit delta : Zustand x Eingabe -> Zustand.
 *   Ueberprueft, ob der eingegebene Binaerstring durch 3 teilbar ist.
 *   Syntaktisch korrekte Eingabe wird vorausgesetzt.
 *   Zeichen '0' und '1' werden umgewandelt in Integer 0 bzw. 1.
 */

public class Automat {

  public static void main (String argv[]) {

    int[][] delta = {{0,1}, {2,0}, {1,2}};             // Ueberfuehrungsfunktion

    int z = 0;                                         // Zustand

    char[] zeile=IO.readChars("Bitte Binaerstring: "); // fordere Eingabe an

    for (int i=0; i < zeile.length; i++)               // fuer jedes Zeichen
        z = delta[z][zeile[i] - (int) '0'];            // berechne Folgezustand

    IO.println("Ganzzahliger Rest betraegt " + z);     // Gib Ergebnis bekannt
  }
}
```

3.7 Lineare und binäre Suche

```
/**************************  Suche.java  ***********************************/
import AlgoTools.IO;

/**  lineare Suche eines Wertes und des Minimums im Array
 *   und binaere Suche im geordneten Array
 */

public class Suche {

  public static void main (String argv[]) {

    int[] a;                                             // Feld fuer Zahlenfolge
    int i, x, min, index, links, rechts, mitte;

    a = IO.readInts("Bitte Zahlenfolge:      ");
    x = IO.readInt ("Bitte zu suchende Zahl: ");
                                                         // lineare Suche n. Element
    for (i=0; (i < a.length) && (x != a[i]); i++);     // durchlaufe Folge
    if (i==a.length) IO.println("Nicht gefunden");     // gib Suchergebnis bekannt
    else             IO.println("Gefunden an Position " + i);
                                                         // lineare Suche n. Minimum
    min = a[0]; index = 0;                               // Minimum ist erster Wert
    for (i = 1; i < a.length; i++){                      // durchlaufe gesamte Folge
        if (a[i] < min) {                                // falls neues Minimum
            index = i;                                   // merke Position
            min   = a[i];}                               // und Wert des Minimums
    } IO.println("Minimum = " + min + " an Position " + index);

    a = IO.readInts("Bitte sortierte Zahlenfolge: ");// binaere Suche n. Element
    x = IO.readInt ("Bitte zu suchende Zahl:      ");// in sortierter Folge
    mitte = links = 0;                                   // initialisiere links
    rechts = a.length-1;                                 // initialisiere rechts
    while (links <= rechts) {                            // solange nichtleeres Int.
        mitte = (links + rechts) / 2;                    // berechne Mitte
        if (a[mitte] == x) break;                        // richtig gesprungen
        if (a[mitte] < x)  links  = mitte+1;             // zu kurz gesprungen
        else               rechts = mitte-1;             // zu weit gesprungen
    }
    if (links > rechts) IO.println("Nicht gefunden");
    else IO.println("Gefunden an Position " + mitte);
  }
}
```

Analyse der Laufzeit der linearen Suche

Beh.: Die Anzahl der Vergleiche beträgt im ungünstigsten Fall n und im Durchschnitt $\frac{n}{2}$.

Analyse der Laufzeit der binären Suche

Beh.: In einem Schleifendurchlauf schrumpft ein Intervall der Länge $2^k - 1$ auf ein Intervall der Länge $2^{k-1} - 1$.

Beweis:

Sei Länge vom alten Intervall =

$$
\begin{array}{rlcl}
 & 2^k - 1 & = & \text{rechts} - \text{links} + 1 \\
\Rightarrow & 2^k & = & \text{rechts} - \text{links} + 2 \\
\Rightarrow & 2^{k-1} & = & \frac{\text{rechts} - \text{links}}{2} + 1 \\
\Rightarrow & 2^{k-1} - 1 & = & \frac{\text{rechts} - \text{links}}{2} = \frac{\text{rechts} + \text{rechts} - \text{links} - \text{rechts}}{2} \\
 & & = & \text{rechts} - (\frac{\text{links} + \text{rechts}}{2} + 1) + 1 \\
 & & = & \text{neue Intervalllänge im } \texttt{THEN}\text{-Zweig}
\end{array}
$$

Korollar: $2^k - 1$ Zahlen verursachen höchstens k Schleifendurchläufe, da nach k Halbierungen die Intervallänge 1 beträgt.

Beispiel für eine Folge von 31 sortierten Zahlen:

Schleifendurchlauf	1	2	3	4	5
aktuelle Intervalllänge	$2^5 - 1$	$2^4 - 1$	$2^3 - 1$	$2^2 - 1$	$2^1 - 1$
	$= 31$	$= 15$	$= 7$	$= 3$	$= 1$

Anders formuliert:
n Zahlen verursachen höchstens $\log_2 n$ Schleifendurchläufe.

Kapitel 4

Klassenmethoden

Häufig benutzte Algorithmen werden zu so genannten *Methoden* zusammengefasst, die unter Nennung ihres Namens und ggf. mit Übergabe von aktuellen Parametern aufgerufen werden.

Methoden sind entweder objektbezogen (siehe später) oder klassenbezogen. Klassenbezogene Methoden werden durch das Schlüsselwort `static` deklariert und können innerhalb oder (durch Angabe des Schlüsselworts `public`) außerhalb der Klasse aufgerufen werden. Gibt eine Methode einen Wert zurück, so steht sein Typ vor dem Methodennamen, andernfalls steht dort `void`. Als Aufrufmechanismus wird *call-by-value* verwendet, d.h., ein im Methodenrumpf benutzter formaler Parameter wird bei Methodenaufruf mit dem Wert des aktuellen Parameters versorgt.

Innerhalb einer Methode verdecken formale Parameter und lokale Variablen gleich lautende Identifier aus der umschließenden Klasse.

Eine mit dem Schlüsselwort `static` versehene Deklaration einer Klassenvariablen führt eine klassenbezogene Variable ein, welche global in allen Methoden dieser Klasse sichtbar ist.

Methoden können überladen werden durch verschiedene Parametersätze.

```
/***************************  Methoden.java  *********************************/

import AlgoTools.IO;

/**  Klassen-Methoden
 *   mit und ohne formale Parameter
 *   mit und ohne Rueckgabewert
 */

public class Methoden {

                                              // Methode ohne Rueckgabewert
  static void bitte() {                       // und ohne Parameter
    IO.println("Bitte Eingabe: ");
  }

                                              // Methode ohne Rueckgabewert
  static void sterne(int k) {                 // mit einem Integer-Parameter
    int i;                                    // lokale Variable
    for (i=0; i < k; i++)                     // zeichne k Sterne
        IO.print('*');
    IO.println();
  }

                                              // Methode mit Rueckgabewert
  static int hoch(int a, int b) {             // und zwei Integer-Parametern
    int i, h = 1;                             // lokale Variablen
    for (i=0; i < b; i++) h *= a;             // berechne a hoch b
    return h;                                 // liefere Ergebnis ab
  }

                                              // Methode mit Rueckgabewert
  static int ggt(int a, int b) {              // zwei Integer-Parameter
    while (a != b)                            // solange Zahlen verschieden
        if (a > b) a -= b; else b -= a;       // ziehe kleinere von groesserer ab
    return a;                                 // liefere Ergebnis zurueck
  }                                           // aktuelle Parameter unveraendert

                                              // Methode ohne Rueckgabewert
  public static void main (String argv[]) {  // oeffentlich aufrufbar
    bitte();                                  // rufe bitte auf
    sterne(3*5+7);                            // rufe sterne auf
    IO.println("3 hoch 7 ist " + hoch(3,7)); // rufe hoch auf
    IO.println("ggt(28,12) = " + ggt(28,12));// rufe ggt auf
  }
}
```

```
/***************************  Parameter.java  ********************************/

import AlgoTools.IO;

/**  Wirkungsweise der Parameteruebergabe an Methoden
 */

public class Parameter {
                                            // Methode ohne Rueckgabewert
  static void zeige(int[] a) {              // erhaelt Array als Parameter
    for (int i=0; i < a.length; i++)        // gibt Array aus
        IO.print(a[i],5);
    IO.println();
  }
                                            // Methode mit Rueckgabewert
  static int summe(int[] a) {               // erhaelt Array als Parameter
    int s = 0;                              // initialisiert Summe
    for (int i=0; i< a.length; i++)         // durchlaeuft Array
        s = s + a[i];                       // berechnet Summe
    return s;                               // und liefert sie zurueck
  }
                                            // Methode mit Rueckgabewert
  static int[] kopiere(int[] a) {           // erhaelt Array als Parameter
    int[] b = new int[a.length];            // besorgt Platz fuer 2. Array
    for (int i=0; i<a.length; i++)          // durchlaeuft Array a
        b[i] = a[i];                        // kopiert es nach b
    return b;                               // liefert b zurueck
  }
                                            // Methode mit Seiteneffekt
  static void reset(int[] a) {              // erhaelt Array als Parameter
    for (int i=0; i < a.length; i++)        // durchlaeuft Array
        a[i] = 0;                           // und setzt jede Komponente auf 0
  }

  public static void main(String argv[]) {  // erhaelt Strings als Parameter
    int[] folge = {1,4,9,16,25,36};         // deklariere initialisierte Folge
    int[] noch_eine_folge;                  // deklariere leere Folge
    zeige(folge);                           // gib folge aus
    IO.println("Summe = " + summe(folge));  // gib die Summe aus
    noch_eine_folge = kopiere(folge);       // schaffe eine Kopie der Folge
    zeige(noch_eine_folge);                 // gib Kopie aus
    reset(folge);                           // setze folge auf 0
    zeige(folge);                           // gib folge aus
  }
}
```

```
/**************************  Sichtbarkeit.java  *****************************/

import AlgoTools.IO;

/** Sichtbarkeit und Lebensdauer von Variablen.
 *  Ueberladen von Methoden.
 */

public class Sichtbarkeit {

  static int grenze;                        // globale Variable
  static int i = 100;                       // globale Varibale

  static void setze() {                     // Methode ohne Parameter
    int i = 42;                             // lokales i, verschieden von glob. i
    grenze = i;                             // setze globale grenze auf 42
  }


  static void setze(int a) {                // Methode mit Parameter
    grenze = a;                             // setze globale Variable
  }                                         // durch uebergebenen Parameter


  static void zeichne() {                   // Methode ohne Parameter
    for (int i=0; i < grenze; i++)          // lokale Variable laeuft bis
        IO.print('*');                      // zum Wert der globalen Variablen
    IO.println();
  }


  static int hole() {                       // Methode ohne Parameter
    return grenze;                          // liefert momentanen Wert von grenze
  }


  public static void main(String argv[]){ // Methode mit Parameter
    setze();                                // setze globale Variable
    zeichne();                              // nutze globale Variable
    setze(4711);                            // setze globale Variable
    IO.println("Grenze = " + hole());       // hole  globale Variable
  }
}
```

Kapitel 5

Rekursion

Eine Methode (mit oder ohne Rückgabewert, mit oder ohne Parameter) darf in der Deklaration ihres Rumpfes den eigenen Namen verwenden. Hierdurch kommt es zu einem rekursiven Aufruf. Typischerweise werden dabei die aktuellen Parameter so modifiziert, dass die Problemgröße schrumpft, damit nach mehrmaligem Wiederholen dieses Prinzips schließlich kein weiterer Aufruf erforderlich ist und die Rekursion abbrechen kann.

5.1 Fakultät, Potenzieren, Fibonacci, GGT

```
/***************************  Rekursion.java  ********************************/

import AlgoTools.IO;

/** Rekursive Methoden                                                          */

public class Rekursion {

  static int fakultaet (int n) {        //
    if (n == 0)                         //
        return 1;                       //           1         falls n=0
    else                                //  n! :=
        return n * fakultaet(n-1);      //           n*(n-1)!  sonst
  }

  static int zweihoch (int n) {         //
    if (n == 0)                         //           1         falls n=0
        return 1;                       //   n
    else                                //  2   :=      (n-1)
        return 2*zweihoch(n-1);         //           2*2       sonst
  }

  static int fib (int n){               // Jedes Kaninchenpaar bekommt vom
    if (n <=1 )                         // 2. Jahr an ein Paar als Nachwuchs
        return 1;                       //
    else                                // Jahr     0  1  2  3  4  5  6  7  8
        return fib(n-1) + fib(n-2);     // Paare    1  1  2  3  5  8 13 21 34
  }                                     //

  static int ggt (int x, int y) {       //
    if (y == 0)                         //
        return x;                       //             x          falls y=0
    else                                // ggt(x,y) :=
        return ggt(y, x % y);           //             ggt(y,x%y) sonst
  }

  public static void main (String argv[]) {
    int[] a = IO.readInts("Bitte zwei Zahlen: ", 2);
    IO.println(a[0] + " ! =  " + fakultaet(a[0]));
    IO.println("2 hoch " + a[0] + " = " + zweihoch(a[0]));
    IO.println(a[0] + ". Fibonacci-Zahl = " + fib(a[0]));
    IO.println("ggt(" + a[0] + "," + a[1] + ") = " + ggt(a[0],a[1]));
  }
}
```

5.2 Türme von Hanoi

```
/**************************  Hanoi.java  ***********************************/

import AlgoTools.IO;

/**  Tuerme von Hanoi:
 *   n Scheiben mit abnehmender Groesse liegen auf dem Startort A.
 *   Sie sollen in derselben Reihenfolge auf Zielort C zu liegen kommen.
 *   Die Regeln fuer den Transport lauten:
 *   1.) Jede Scheibe muss einzeln transportiert werden.
 *   2.) Es darf nie eine groessere Scheibe auf einer kleineren liegen.
 *   3.) Es darf ein Hilfsort B zum Zwischenlagern verwendet werden.
 */

//           |                       |                        |
//          x|x                      |                        |
//         xx|xx                     |                        |
//        xxx|xxx                    |                        |
//       xxxx|xxxx                   |                        |
//    ---------------         ---------------          ---------------
//        Start (A)              Zwischen (B)               Ziel (C)

public class Hanoi {

  static void verlege (             // drucke die Verlegeoperationen, um
    int n,                          // n Scheiben
    char start,                     // vom Startort
    char zwischen,                  // unter Zuhilfenahme eines Zwischenortes
    char ziel) {                    // zum Ziel zu bringen

    if (n == 1)
        IO.println("Scheibe  1 von " + start + " nach " + ziel);
    else {
        verlege(n-1,start, ziel, zwischen);
        IO.println("Scheibe " + n +" von " + start + " nach " + ziel);
        verlege(n-1,zwischen, start, ziel);
    }
  }

  public static void main (String argv[]) {
    int n;
    do{ n = IO.readInt("Bitte Zahl der Scheiben (n>0): "); } while (n <= 0);
    verlege(n,'A','B','C');
  }
}
```

Analyse der Größe der erzeugten Ausgabe:

Sei $f(n)$ die Anzahl der generierten Verlegebefehle bei Aufruf von

```
verlege (n, start, zwischen, ziel).
```

$$\text{Offenbar:}\quad \begin{array}{rcl} f(1) &=& 1 \\ f(n) &=& f(n-1)+1+f(n-1) \quad = \quad 2\cdot f(n-1)+1 \end{array}$$

d.h., die Wertetabelle beginnt wie folgt:

n	$f(n)$
1	1
2	3
3	7
4	15
5	31
6	63

Verdacht: $f(n) = 2^n - 1$

Beweis durch Induktion

Induktionsverankerung: $f(1) = 1 = 2^1 - 1$

Induktionsschritt: Sei bis $n-1$ bewiesen:

$$f(n) \underbrace{= 2\cdot f(n-1)+1}_{\substack{\uparrow \\ \text{Rekursionsgleichung}}} \quad \underbrace{= 2\cdot(2^{n-1}-1)+1}_{\substack{\uparrow \\ \text{Induktionsannahme}}} = 2^n - 2 + 1 = 2^n - 1$$

Kapitel 6

Komplexität und Verifikation

Nicht:	absolute CPU-Zeit angeben
Nicht:	Anzahl der Maschinenbefehle zählen
Sondern:	Wachstum der Laufzeit in "Schritten" in Abhängigkeit von der Größe der Eingabe = Länge ihrer Codierung = Anzahl der Daten beschränkter Größe oder Länge der Darstellung einer Zahl

6.1 O-Notation

Seien $f, g : \mathbb{N} \to \mathbb{N}$

f ist höchstens von der Größenordnung g

in Zeichen: $f \in O(g)$

falls $n_0,\ c \in \mathbb{N}$ existieren mit

$$\forall\, n \geq n_0 :\ f(n) \leq c \ \cdot \ g(n)\ .$$

Statt $f \in O(g)$ sagt man auch $f = O(g)$

$\Rightarrow$ Wegen des konstanten Faktors c ist die exakte Festlegung eines Schrittes nicht erforderlich.

Beispiel:

Sei $f(n) = 31n + 12n^2$.

Dann ist $f \in O(n^2)$

Natürlich gilt auch: $f \in O(n^7)$

Die wesentlichen Klassen

$\log n$	n	$n \cdot \log n$	n^2	n^3	n^4	$\ldots$	2^n	3^n	$n!$
$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$			$\uparrow$		$\uparrow$
binäre Suche	lineare Suche	schlaues Sortieren	dummes Sortieren	Gleichungs-system lösen			alle Teil-mengen		alle Permu-tationen

Annahme: 1 Schritt dauert 1 μs = 0.000001 s

$n =$	10	20	30	40	50	60
n	10 μs	20 μs	30 μs	40 μs	50 μs	60 μs
n^2	100 μs	400 μs	900 μs	1.6 ms	2.5 ms	3.6 ms
n^3	1 ms	8 ms	27 ms	64 ms	125 ms	216 ms
2^n	1 ms	1 s	18 min	13 Tage	36 J	366 Jh
3^n	59 ms	58 min	6.5 J	3855 Jh	10^8 Jh	10^{13} Jh
$n!$	3.62 s	771 Jh	10^{16} Jh	10^{32} Jh	10^{49} Jh	10^{66} Jh

Für einen Algorithmus mit Laufzeit $O(2^n)$ gilt daher:

> Wächst n um 1, so wächst 2^n um den Faktor 2.
> Wächst n um 10, so wächst 2^n um den Faktor $2^{10} = 1024$.
> Ein 1000-mal schnellerer Computer kann eine um 10 Daten größere Eingabe in derselben Zeit bearbeiten.

Analoge Aussagen sind möglich bzgl. Speicherplatz.
Wir zählen nicht die Anzahl der Bytes, sondern betrachten das Wachstum des Platzbedarfs in Speicherplätzen in Abhängigkeit von der Größe der Eingabe.

Aussagen über *Laufzeit* und *Platzbedarf* beziehen sich auf

- *best case* günstigster Fall
- *worst case* ungünstigster Fall
- *average case* im Mittel

Analyse von `for`-Schleifen

Beispiel:

Minimumsuche in einem Array der Länge n

```
min = a[0];
for (i = 1; i < n; i++){
        if(a[i] < min) min = a[i];
}
```

Laufzeit $O(n)$ für best case
worst case
average case.

Beispiele:

```
for (i = 0; i < k; i++) {
   for (j = 0; j < k; j++) {
      brett[i][j] = 0;
   }
}
```

k^2 Schritte für k^2 Daten
$\Rightarrow O(n)-$Algorithmus

```
for (i = 0, i < k; i++) {
   for (j = 0; j < k; j++) {
      if (a[i] == a[j]) treffer = true;
   }
}
```

k^2 Schritte für k Daten
$\Rightarrow O(n^2)-$Algorithmus

Analyse von `while`-Schleifen

Beispiel:

Lineare Suche im Array

```
i = 0;
while (i < n) && (a[i] != x) {
   i++;
}
```

Laufzeit:	best case	1 Schritt	$\Rightarrow$	$O(1)$
	worst case	n Schritte	$\Rightarrow$	$O(n)$
	average case	$\frac{n}{2}$ Schritte	$\Rightarrow$	$O(n)$

Annahme für den average case:
Es liegt Permutation der Zahlen von 1 bis n vor.
Dann ist die mittlere Anzahl

$$= \frac{1}{n} \sum_{i=1}^{n} i \quad \approx \quad \frac{n}{2}$$

Beispiel:

Suche 0 in einem Array, bestehend aus Nullen und Einsen.

Laufzeit:	best case	1 Schritt	$\Rightarrow$	$O(1)$
	worst case	n Schritte	$\Rightarrow$	$O(n)$
	average case	$\sum\limits_{i=1}^{n} i \cdot \frac{1}{2^i} \leq 2$	$\Rightarrow$	$O(1)$

Obacht:

Alle Laufzeitangaben beziehen sich jeweils auf einen konkreten Algorithmus A (für ein Problem P) = *obere Schranke* für Problem P.

Eine Aussage darüber, wie viele Schritte jeder Algorithmus für ein Problem P mindestens durchlaufen muss, nennt man *untere Schranke* für Problem P.

Beispiel: naives Pattern-Matching

```
char[] s = new char[N];                              // Zeichenkette
char[] p = new char[M];                              // Pattern
```

Frage: Taucht Pattern p in Zeichenkette s auf?

```
for (i = 0; i <= N - M; i++) {                       // Index in Zeichenkette
   for (j = 0; (j < M) && (p[j] == s[i+j]); j++);   // Index im Pattern
   if (j == M) break;                                // Erfolg
}
```

Laufzeit best case: $O(1)$
Laufzeit worst case: $(N-M+1)\cdot M$ Vergleiche für $\begin{array}{rcr} s & = & AAA\ldots AB \\ p & = & A\ldots AB \end{array}$

Sei n die Summe der Buchstaben in Pattern p und String s. Sei $M = x \cdot n$ und $N = (1-x)\cdot n$ für $0 < x < 1$.

Gesucht wird x, welches $((1-x)\cdot n - x\cdot n + 1)\cdot x \cdot n = (n^2+n)\cdot x - 2n^2 \cdot x^2$ maximiert.

Bilde 1. Ableitung nach x und setze auf 0: $0 = n^2 + n - 4n^2 \cdot x$

$\Rightarrow$ Maximum liegt bei $\frac{n^2+n}{4n^2} \approx \frac{1}{4}$

Also können $(\frac{3}{4}n - \frac{1}{4}n + 1)\cdot \frac{1}{4}n = \frac{1}{8}n^2 + \frac{1}{4}n$ Vergleiche entstehen.

$\Rightarrow$ Die Laufzeit im worst case beträgt $O(n^2)$.

Analyse eines rekursiven Programms

Beispiel:

Die Laufzeit von $fib(n)$ beträgt:

$$f(n) = \begin{cases} 1, & \text{falls } n \leq 1 \\ f(n-1) + f(n-2) + 1 & \text{sonst} \end{cases}$$

Offenbar gilt: $f \in O(\alpha^n)$ für ein $\alpha < 2$ (denn $f(n) = f(n-1) + f(n-1)$ würde zu $O(2^n)$ führen).

Gesucht ist also ein α, so dass für große n gilt:

$$\alpha^n = \alpha^{n-1} + \alpha^{n-2} + 1$$

Teile durch α^{n-2}:

$\Rightarrow \alpha^2 = \alpha + 1 + \frac{1}{\alpha^{n-2}}$. Für $n \to \infty$ und $\alpha > 1$ geht $\frac{1}{\alpha^{n-2}} \to 0$.

$\Rightarrow \alpha^2 = \alpha + 1 \Rightarrow \alpha = \frac{1}{2} + \sqrt{\frac{1}{4}+1} = 1.61803$ (gemäß Formel $-\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q}$)

⇒ Das rekursive Fibonacci-Programm hat die Laufzeit $O(1.62^n)$, wobei n der Wert des Inputs ist, nicht seine Länge!

für $n = 20$ ist	1.62^n	≈ 15.000
	2^n	$\approx 1.000.000.$

6.2 Korrektheit und Terminierung

Durch *Testen* kann nachgewiesen werden, dass sich ein Programm für **endlich viele** Eingaben korrekt verhält. Durch eine *Verifikation* kann nachgewiesen werden, dass sich ein Programm für **alle** Eingaben korrekt verhält.

Bei der *Zusicherungsmethode* sind zwischen den Statements so genannte *Zusicherungen* eingestreut, die eine Aussage darstellen über die momentane Beziehung zwischen den Variablen.

Z.B. `/*` $i > 0 \wedge z = i^2$ `*/`

Aus einer Zusicherung und der folgenden Anweisung lässt sich dann eine weitere Zusicherung ableiten:

`i = i - 1;`
⇒ `/*` $i \geq 0 \ \wedge z = (i+1)^2$ `*/`

Bei Schleifen wird die Zusicherung P, die vor Eintritt und vor Austritt gilt, die *Schleifeninvariante* genannt.

```
/* P */
while Q {
   /* P ∧ Q */
   ⋮
   ⋮
   /* P */
}
/* P ∧ ¬Q */
```

Beginnend mit einer ersten, offensichtlich richtigen Zusicherung, lässt sich als letzte Zusicherung eine Aussage über das berechnete Ergebnis ableiten = *partielle Korrektheit.*

Zusammen mit dem Nachweis der *Terminierung* ergibt sich die *totale Korrektheit.*

Beispiel für die Zusicherungsmethode:

```
int n, x, y, z;
do { n = IO.readInt(); } while (n < 0);
/* n ≥ 0 */
x = 0; y = 1; z = 1;
/* (z = (x + 1)^2 ∧ y = 2x + 1 ∧ x^2 ≤ n) = Schleifeninvariante P */
while (z <= n) /* Q */ {
   /* z = (x + 1)^2 ∧ y = 2x + 1 ∧ x^2 ≤ n ∧ z ≤ n ⇒ (x + 1)^2 ≤ n */
   x = x + 1;
   /* z = x^2 ∧ y = 2x − 1 ∧ x^2 ≤ n */
   y = y + 2;
   /* z = x^2 ∧ y = 2x + 1 ∧ x^2 ≤ n */
   z = z + y;
   /* z = x^2 + 2x + 1 ∧ y = 2x + 1 ∧ x^2 ≤ n */
   /* (z = (x + 1)^2 ∧ y = 2x + 1 ∧ x^2 ≤ n) = P */
}
/* (z = (x + 1)^2 ∧ y = 2x + 1 ∧ x^2 ≤ n ∧ z > n) = P ∧ ¬Q */
/* x^2 ≤ n < (x + 1)^2 */
/* x = ⌊√n⌋ */
IO.println (x);
/* Ausgabe: ⌊√n⌋ */
```

Terminierung

Da y immer positiv ist und z immer um y erhöht wird und n fest bleibt, muß irgendwann gelten $z > n$

$\Rightarrow$ Abbruch gesichert

$\Rightarrow$ totale Korrektheit.

Laufzeit

In jedem Schleifendurchlauf wird x um eins erhöht.

Da x bei 0 beginnt und bei $\lfloor\sqrt{n}\rfloor$ endet, wird der Schleifenrumpf $\sqrt{n}$ mal durchlaufen. Der Schleifenrumpf selbst hat eine konstante Anzahl von Schritten.

$\Rightarrow$ Laufzeit $O(n^{\frac{1}{2}})$, wobei n der Wert der Eingabezahl ist!

Weiteres Beispiel für die Zusicherungsmethode:

```
int n, x, y, z;
do { n = IO.readInt(); } while (n < 0);
/* $n \geq 0$ */
x = 2; y = n; z = 1;
/* $z \cdot x^y = 2^n$ */
while (y > 0) {
   /* $z \cdot x^y = 2^n \;\wedge\; y > 0$ */
   if (y % 2 == 1) {
      /* $z \cdot x^y = 2^n \;\wedge\; y > 0 \;\wedge\; y$ ungerade */
      z = z * x;
      /* $\frac{z}{x} \cdot x^y = 2^n \;\wedge\; y > 0 \;\wedge\; y$ ungerade */
      y = y - 1;
      /* $\frac{z}{x} \cdot x^{y+1} = 2^n \;\wedge\; y \geq 0$ */
   /* $z \cdot x^y = 2^n \;\wedge\; y \geq 0$ */
   } else {
      /* $z \cdot x^y = 2^n \;\wedge\; y > 0 \;\wedge\; y$ gerade */
      x = x * x;
      /* $z \cdot (x^{\frac{1}{2}})^y = 2^n \;\wedge\; y > 0 \;\wedge\; y$ gerade */
      y = y/2;
      /* $z \cdot (x^{\frac{1}{2}})^{2y} = 2^n \;\wedge\; y \geq\; 0$ */
      /* $z \cdot x^y = 2^n \;\wedge\; y \geq 0$ */
   }
   /* $z \cdot x^y = 2^n \;\wedge\; y \geq 0$ */
}
/* $z \cdot x^y = 2^n \;\wedge\; y \geq 0 \wedge\; y \leq 0$ */
/* $z \cdot x^y = 2^n \;\wedge\; y = 0 \Rightarrow z = 2^n$ */
IO.println (z);
/* Ausgabe: $2^n$ */
```

Terminierung

In jedem Schleifendurchlauf wird y kleiner $\Rightarrow$ irgendwann einmal Null $\Rightarrow$ Abbruch.

Laufzeit

Die Dualzahldarstellung von y schrumpft spätestens nach 2 Schleifendurchläufen um eine Stelle

$\Rightarrow$ $O(\log n)$ Durchläufe, wobei n der Wert der Eingabezahl ist, d.h. $O(n)$, wenn n die Länge der Dualdarstellung der Eingabe ist.

Hinweis: Der naive Ansatz

```
n = IO.readInt("n = "); x = 1;
for (i = 0; i < n; i++) x = 2*x;
```

hat Laufzeit $O(n)$, wenn n der Wert der Eingabezahl ist, d.h. $O(2^k)$, wenn k die Länge der Dualdarstellung von n ist.

6.3 Halteproblem

Beh.: Es gibt kein Programm, welches entscheidet, ob ein gegebenes Programm, angesetzt auf einen gegebenen Input, anhält.

Beweis durch Widerspruch

Annahme: Es gibt eine Methode

```
static boolean haltetest (char[]s, char[]t)
/* liefert true, falls das durch die Zeichenkette s dargestellte
 * Java-Programm bei den durch die Zeichenkette t dargestellten Eingabedaten
 * anhaelt; liefert false, sonst */
```

Dann lässt sich folgendes Java-Programm in der Datei `Quer.java` konstruieren:

```
public class Quer {

import AlgoTools.IO;

    public static void main(String argv[]) {

        char[] s = IO.readChars();
        if (haltetest(s,s)) while (true);
    }
}
```

Sei q der String, der in der Datei `Quer.java` steht.

Was passiert bei Aufruf von

`java Quer`

und Eintippen von q ?

1. Fall:	Hält an	$\Rightarrow$	`haltetest`(q,q) `== false`
		$\Rightarrow$	`Quer` angesetzt auf q hält nicht!
2. Fall:	Hält nicht	$\Rightarrow$	`haltetest`(q,q) `== true`
		$\Rightarrow$	`Quer` angesetzt auf q hält an!

$\Rightarrow$ Also kann es die Methode `haltetest` nicht geben!

Kapitel 7

Sortieren

Motivation für Sortieren:

1. Häufiges Suchen
   Einmal sortieren, dann jeweils $\log n$ Aufwand.

2. Tritt ein Element in zwei Listen L_1, L_2 auf?
   Sortiere $L_1 \cdot L_2$, dann nach Doppelten suchen!

7.1 Selection Sort

"Hole jeweils das kleinste Element nach vorne"

```
public class SelectionSort {                          // Klasse SelectionSort

  public static void sort(int[] a) {                  // statische Methode sort
    for (int i=0; i<a.length-1; i++) {                // durchlaufe Array
        int k = i;                                    // Index des bisher kleinsten
        int x = a[i];                                 // Wert des bisher kleinsten
        for (int j=i+1; j<a.length; j++)              // durchlaufe Rest des Array
            if (a[j] < x) {                           // falls kleineres gefunden,
                k = j;                                // merke Index
                x = a[j];                             // merke Position
            }
        a[k] = a[i];                                  // speichere bisher kleinstes um
        a[i] = x;                                     // neues kleinstes nach vorne
    }
  }
}
```

Beispiel:

```
4 9 3 2 5
2 9 3 4 5
2 3 9 4 5
2 3 4 9 5
2 3 4 5 9
```

Analyse für *Selection Sort*

Worst case und *best case*:
Zwei ineinander geschachtelte `for`-Schleifen

$$n - 1 + n - 2 + n - 3 + \ldots + 1$$
$$= \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$$

Platzbedarf: $O(n)$
zusätzlich zu den Daten: $O(1)$

Der Algorithmus wird nicht schneller, wenn die Zahlen bereits sortiert sind!

7.2 Bubblesort

Sortieren mit *Bubblesort* = Blasen steigen auf

“Vertausche jeweils unsortierte Nachbarn”

```
4 9 3 2 5
  3 9
    2 9
4 3 2 5 9
3 4
  2 4
3 2 4 5 9

  ⋮
```

```
public class BubbleSort {                           // Klasse BubbleSort

  public static void sort(int[] a) {                // statische Methode sort
    int tmp;                                        // Hilfsvariable zum Tauschen
    boolean getauscht;                              // merkt sich, ob getauscht
    do {
        getauscht = false;                          // nimm an, dass nicht getauscht
        for (int i=0; i<a.length-1; i++){          // durchlaufe Array
            if (a[i] > a[i+1]) {                    // falls Nachbarn falsch herum
                tmp = a[i];                         // bringe
                a[i] = a[i+1];                      // beide Elemente
                a[i+1] = tmp;                       // in die richtige Ordnung
                getauscht = true;                   // vermerke, dass getauscht
            }
        }
    } while (getauscht);                            // solange getauscht wurde
  }
}
```

Analyse von *Bubblesort*

Best case: $O(n)$
Worst case: $O(n^2)$

Die kleinste Zahl wandert in der `for`-Schleife jeweils um eine Position nach links.
Wenn sie zu Beginn ganz rechts steht, so sind $n-1$ Phasen nötig.

Average case: $O(n^2)$

Weitere Verbesserungen möglich (*Shaker Sort*), aber es bleibt bei $O(n^2)$
Grund: Austauschpositionen liegen zu nahe beieinander.

7.3 Mergesort

Idee (rekursiv formuliert):

```
Sortiere die vordere Hälfte der Folge;
Sortiere die hintere Hälfte der Folge;
Mische die beiden sortierten Folgen zu einer sortierten Folge
```

Die Hilfsroutine zum Mischen lautet:

```
public class Merge {                                 // Klasse Merge

  public static int[] merge (int[]a, int[]b) {   // mischt a und b
                                                 // liefert Ergebnis zurueck

    int i=0, j=0, k=0;                           // Laufindizes
    int[] c = new int[a.length + b.length];      // Platz fuer Folge c besorgen

    while ((i<a.length) && (j<b.length)) {       // mischen, bis ein Array leer
      if (a[i] < b[j])                           // jeweils das kleinere Element
          c[k++] = a[i++];                       // wird nach c uebernommen
      else
          c[k++] = b[j++];
    }

    if (i==a.length)                             // falls Folge a am Ende:
      while (j<b.length) c[k++] = b[j++];        // Rest von Folge b uebernehmen
    else                                         // falls Folge b am Ende:
      while (i<a.length) c[k++] = a[i++];        // Rest von Folge a uebernehmen

    return c;                                    // Ergebnis abliefern
  }
}
```

Analyse von *Mergesort* (und ähnlich gelagerten Rekursionen)

$$f(n) \le \begin{cases} c_1 & \text{für } n = 1 \\ 2 \cdot f(\frac{n}{2}) + c_2 \cdot n & \text{sonst} \end{cases}$$

Beh.: $f \in O(n \cdot \log n)$
Zeige: $f(n) \le (c_1 + c_2) \cdot n \cdot \log n + c_1$

Verankerung:

$n = 1 \Rightarrow$ $f(1) \le c_1$ nach Rekursion
$f(1) \le (c_1 + c_2) \cdot 1 \cdot \log 1 + c_1$

Induktionsschluss

Sei bis $n - 1$ bewiesen

$$\begin{aligned} f(n) &\underset{\substack{\uparrow \\ \text{Rek.}}}{\le} 2 \cdot f(\tfrac{n}{2}) + c_2 \cdot n \\ &\underset{\substack{\uparrow \\ \text{Induktionsannahme}}}{\le} 2 \cdot [(c_1 + c_2) \cdot \tfrac{n}{2} \cdot \log \tfrac{n}{2} + c_1] + c_2 \cdot n \\ &= 2 \cdot [(c_1 + c_2) \cdot \tfrac{n}{2} \cdot (\log n - 1) + c_1] + c_2 \cdot n \\ &= (c_1 + c_2) n \cdot \log n - (c_1 + c_2) \cdot n + 2c_1 + c_2 \cdot n \\ &= [(c_1 + c_2) n \cdot \log n + c_1] + [c_1 - c_1 \cdot n] \\ &\le (c_1 + c_2) n \cdot \log n + c_1 \end{aligned}$$

Aber: *Mergesort* benötigt $O(n)$ zusätzlichen Platz!

Iterative Version von *Mergesort* (für $n = 2^k$)

```
l = 1; /* Länge der sortierten Teilfolgen */
k = n; /* Anzahl der sortierten Teilfolgen */
while (k > 1) {
   /* alle Teilfolgen der Länge l sind sortiert */
   /* Sie beginnen bei l · i für i = 0, 1, ..., k − 1 */
   Mische je zwei benachbarte Teilfolgen der Länge l
   zu einer Teilfolge der Länge 2 * l;
   l *= 2; k /= 2;
   /* Alle Teilfolgen der Länge l sind sortiert */
   /* Sie beginnen bei l · i für i = 0, 1, ..., k − 1 */
}
```

Es ergeben sich $\log n$ Phasen mit jeweils linearem Aufwand.

$\Rightarrow O(n \cdot \log n)$

```
/***************************  SortTest.java  **********************************/

import AlgoTools.IO;

/** testet Sortierverfahren
 */

public class SortTest {

  public static void main (String argv[]) {

    int[] a, b, c;                                        // Folgen a, b und c

    a = IO.readInts("Bitte eine Zahlenfolge:      ");   // Folge einlesen
    SelectionSort.sort(a);                                // SelectionSort aufr.
    IO.print("sortiert mit SelectionSort: ");
    for (int i=0; i<a.length; i++) IO.print(" "+a[i]);  // Ergebnis ausgeben
    IO.println();
    IO.println();

    b = IO.readInts("Bitte eine Zahlenfolge:      ");   // Folge einlesen
    BubbleSort.sort(b);                                   // BubbleSort aufrufen
    IO.print("sortiert mit BubbleSort:    ");
    for (int i=0; i<b.length; i++) IO.print(" "+b[i]);  // Ergebnis ausgeben
    IO.println();
    IO.println();

    c = Merge.merge(a,b);                                 // mische beide Folgen
    IO.print("sortierte Folgen gemischt:  ");
    for (int i=0; i<c.length; i++) IO.print(" "+c[i]);  // Ergebnis ausgeben
    IO.println();
  }
}
```

7.4 Quicksort

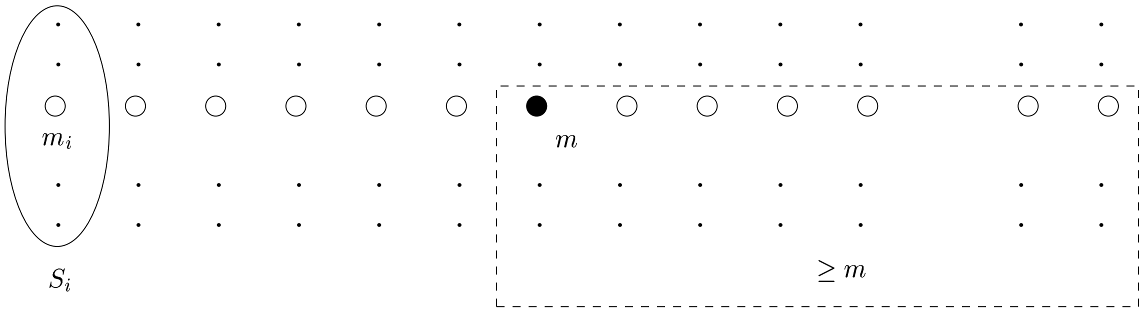
```
/*************************** QuickSort.java *********************************/

/** rekursives Sortieren mit Quicksort
 *  Idee: partitioniere die Folge
 *  in eine elementweise kleinere und eine elementweise groessere Haelfte
 *  und sortiere diese nach demselben Verfahren
 */

public class QuickSort {

  public static void sort (int[] a, int unten, int oben) {
    int tmp ;
    int i = unten;
    int j = oben;
    int x = a[(unten+oben) / 2];                 // Pivotelement, willkuerlich

    do {
        while (a[i] < x) i++;                    // x fungiert als Bremse
        while (a[j] > x) j--;                    // x fungiert als Bremse
        if ( i<=j )  {
            tmp  = a[i];                         // Hilfsspeicher
            a[i] = a[j];                         // a[i] und
            a[j] = tmp;                          // a[j] werden getauscht
            i++;
            j--;
        }
    } while (i <= j);
                              // alle Elemente der linken Haelfte sind kleiner
                              // als alle Elemente der rechten Haelfte

    if (unten < j)  sort(a, unten, j);           // sortiere linke Haelfte
    if (i < oben )  sort(a, i, oben );           // sortiere rechte Haelfte
  }

  // Best case:     O(n * log n), wenn Partitionen immer gleich gross sind
  // Average case:  O(n * log n), um den Faktor 1.4 schlechter als best case
  // Worst case:    O(n * n),     wenn Partitionen entarten: 1 Elem. + n-1 Elem.
}
```

```
/***************************  QuickSortTest.java  ****************************/

import AlgoTools.IO;

/** testet Quicksort
 */

public class QuickSortTest {

  public static void main (String argv[]) {

    int[] a;                                                // Folge a

    a = IO.readInts("Bitte eine Zahlenfolge: ");            // Folge einlesen
    QuickSort.sort(a, 0, a.length-1);                       // QuickSort aufrufen
    IO.print("sortiert mit QuickSort: ");
    for (int i=0; i<a.length; i++) IO.print(" "+a[i]);      // Ergebnis ausgeben
    IO.println();
  }
}
```

7.5 Bestimmung des Medians

```
static double select (double[] S, int k) {
/* Liefert aus dem Feld S das k-t kleinste Element. */
/* Das Feld S habe n Elemente. k ≤ n. */
   double[] A, B, C;


   if (|S| < 50) return k-t kleinstes Element per Hand;
   else {
      n = |S|;
      zerlege S in Gruppen zu je 5 Elementen S_1, S_2, ..., S_{n/5};
      Bestimme in jeder Gruppe S_i den Median m_i;
      Bestimme den Median der Mediane:
         m = select (⋃_i m_i, n/10);
      A = {x ∈ S | x < m};
      B = {x ∈ S | x == m};
      C = {x ∈ S | x > m};
      if (|A| >= k) return select (A, k);
      else if (|A|+|B| >= k) return m;
      else /* if (|A|+|B|+|C| >= k) */ return select(C, k-|A|-|B|);
   }
}
```

Beh.: $|A| \leq \frac{3}{4}n$



d.h. mind. $\frac{1}{4}$ der Elemente von S ist $\geq m$

$\Rightarrow$ höchstens $\frac{3}{4}$ der Elemente von S ist $< m$

$\Rightarrow |A| \leq \frac{3}{4}n$, analog $|C| \leq \frac{3}{4}n$.

Sei $f(n)$ die Anzahl der Schritte, die die Methode `select` benötigt für eine Menge S der Kardinalität n.

Also:

$$f(n) \leq \begin{cases} c & \text{, für } n < 50 \\ \underbrace{c \cdot n}_{\substack{\text{Mediane} \\ \text{der 5er Gruppen}}} + \underbrace{f(\frac{n}{5})}_{\substack{\text{Median} \\ \text{der Mediane}}} + \underbrace{f(\frac{3}{4}n)}_{\substack{\text{Aufruf mit} \\ \text{A oder C}}} & \text{, sonst} \end{cases}$$

Beh.: $f \in O(n)$

Zeige: $f(n) \leq 20 \cdot c \cdot n$

Beweis durch Induktion:

$f(n) \leq c \leq 20 \cdot c \cdot n$ für $n < 50$

Sei bis $n-1$ bewiesen

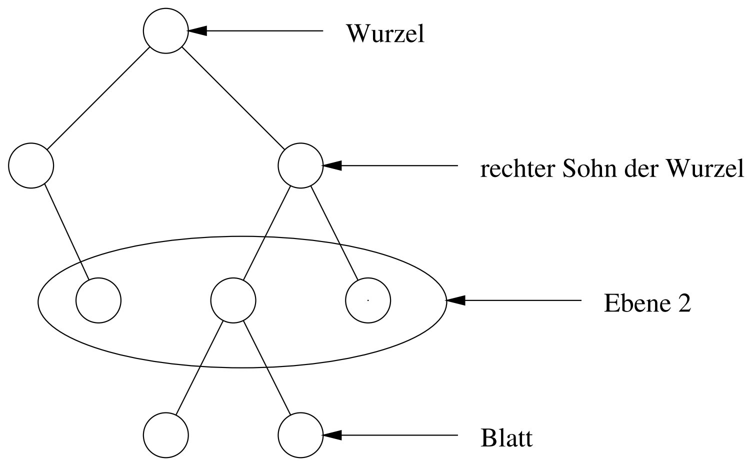
$$\begin{aligned} f(n) &\underset{\substack{\uparrow \\ \text{Rekursionsgl.}}}{\leq} c \cdot n + f(\tfrac{n}{5}) + f(\tfrac{3}{4}n) \\ &\underset{\substack{\uparrow \\ \text{Ind.-Annahme}}}{\leq} c \cdot n + 20 \cdot c \cdot \tfrac{n}{5} + 20 \cdot c \cdot \tfrac{3}{4} \cdot n \\ &= 1 \cdot c \cdot n + 4 \cdot c \cdot n + 15 \cdot c \cdot n \\ &= 20 \cdot c \cdot n \qquad \text{q.e.d.} \end{aligned}$$

7.6 Heapsort

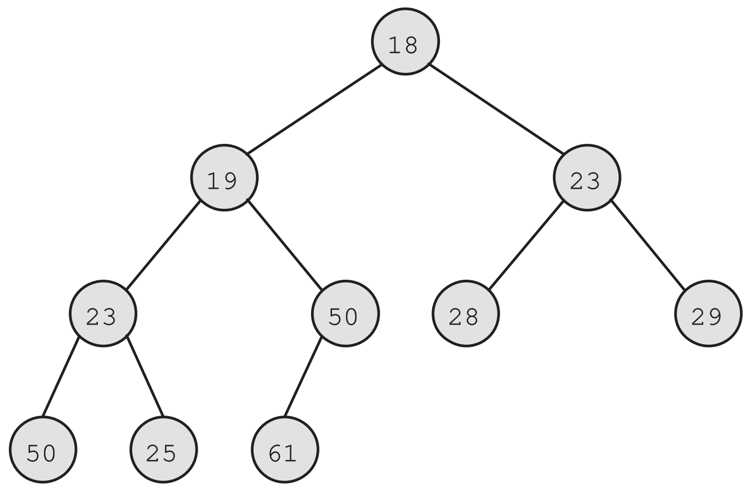
Baum und Heap

Def.: Ein *binärer Baum* ist entweder leer oder besteht aus einem Knoten, dem zwei binäre Bäume zugeordnet sind. Dieser heißt dann *Vater* des linken bzw. rechten Teilbaums. Ein Knoten ohne Vater heißt *Wurzel*. Die Knoten, die x zum Vater haben, sind seine *Söhne*. Knoten ohne Söhne heißen *Blätter*.

Ebene 0 = Wurzel. Ebene $i+1$ = Söhne von Ebene i.



Def.: Ein *Heap* ist ein binärer Baum mit $n+1$ Ebenen, in dem die Ebenen $0, 1, \ldots, n-1$ vollständig besetzt sind; Ebene n ist von links beginnend bis zum so genannten letzten Knoten vollständig besetzt. Die Knoten enthalten Schlüssel. Der Schlüssel eines Knotens ist kleiner oder gleich den Schlüsseln seiner Söhne.



Offenbar steht der kleinste Schlüssel eines Heaps in der Wurzel.

Idee für *Heapsort***:**

Verwendet wird ein Heap als Datenstruktur, die das Entfernen des Minimums unterstützt.

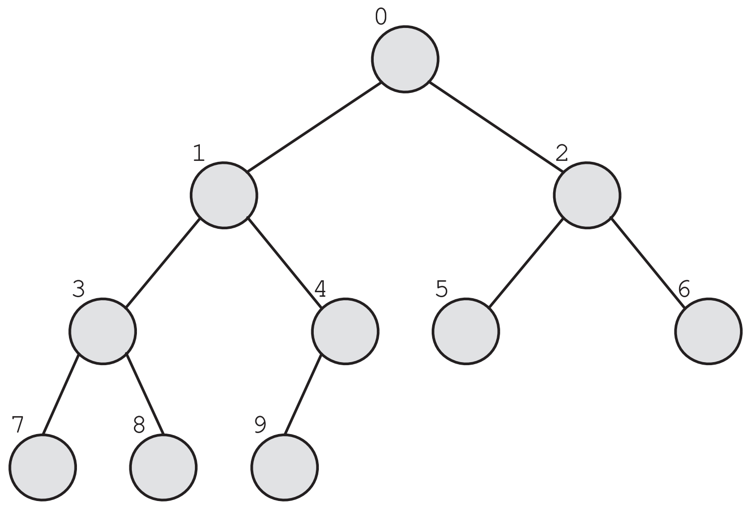
Gegeben seien die Schlüssel $a_1, \ldots, a_n$.

```
Baue einen Heap auf mit den Schlüsseln a1,...,an.
do {
   entferne Wurzel; // = Minimum
   reorganisiere Heap;
} while (Heap ist nicht leer);
```

Idee für Wurzelentfernen:

Entferne "letzten" Knoten im Heap und schreibe seinen Schlüssel in die Wurzel.
Vertausche so lange Knoten mit "kleinerem" Sohn, bis Heapbeziehung eintritt.

Idee für Implementation: Die Knoten werden wie folgt nummeriert:
Wurzel erhält Nr. 0,
linker Sohn von Knoten i erhält die Nr. $(2 \cdot i + 1)$
rechter Sohn von Knoten i erhält die Nr. $(2 \cdot i + 2)$



Im Array

```
double[] a = new double [n];
```

steht in `a[i]` der Schlüssel von Knoten i.

Vorteil: Die Vater/Sohn-Beziehung ist allein aus dem Knotenindex zu berechnen.
$2i + 1$ bzw. $2i + 2$ heißen die Söhne von i
$(i - 1)/2$ heißt der Vater von i.

```java
/**************************  HeapSort.java  ********************************/

import AlgoTools.IO;

/** Iteratives Sortieren mit Heapsort
 *  Entnimm einem Heap so lange das kleinste Element, bis er leer ist.
 *  Die entnommenen Elemente werden im selben Array gespeichert.
 */

public class HeapSort {

  private static void sift (int[] a, int l, int r) {
                                              // repariere das Array a
                                              // in den Grenzen von l bis r
    int i, j;                                 // Indizes
    int x;                                    // Array-Element

    i = l; x = a[l];                          // i und x initialisieren
    j = 2*i+1;                                // linker Sohn
    if ((j<r) && (a[j+1]<a[j])) j++;          // jetzt ist j der kleinere Sohn

    while ((j<=r) && (a[j] < x)) {            // falls kleinerer Sohn existiert
        a[i] = a[j];
        i    = j;
        j    = j*2+1;
        if ((j<r) && (a[j+1]<a[j])) j++;      // jetzt ist j der kleinere Sohn
    }
    a[i] = x;
  }

  public static void sort (int[] a) {         // statische Methode sort
    int l,r,n,tmp;                            // links, rechts, Anzahl, tmp
    n = a.length;                             // Zahl der Heap-Eintraege
    for (l=n/2-1; l>=0; l--) sift(a,l,n-1);   // Heap aufbauen

    for (r=n-1; r>0; r--) {                   // rechte Grenze fallen lassen
        tmp  = a[0];                          // kleinstes Element holen
        a[0] = a[r];                          // letztes nach vorne
        a[r] = tmp;                           // kleinstes nach hinten
        sift(a, 0, r-1);                      // Heap korrigieren
    }
  }
}
```

Aufwand für die Konstruktion eines Heaps

Sei h die Höhe eines Heaps. Sei $n - 1 = 2^h - 1$ die Anzahl der Elemente, z.B. $15 = 2^4 - 1$.

Ebene	Sickertiefe	Anzahl
$h-1$	1	$\frac{n}{2}$
$h-2$	2	$\frac{n}{4}$
$h-3$	3	$\frac{n}{8}$
$\vdots$	$\vdots$	$\vdots$
0	h	$\frac{n}{2^h} = 1$

Anzahl der Schritte:

$$\sum_{i=1}^{h} c \cdot i \cdot \frac{n}{2^i} = c \cdot n \cdot \sum_{i=1}^{h} \frac{i}{2^i}$$

$\Rightarrow$ Aufwand $O(n)$, denn: $\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \ldots < 2$

Aufwand für einmaliges Minimumentfernen: $O(\log n)$

$\Rightarrow$ **Gesamtaufwand:** $O(n) + O(n \cdot \log n) = O(n \cdot \log n)$

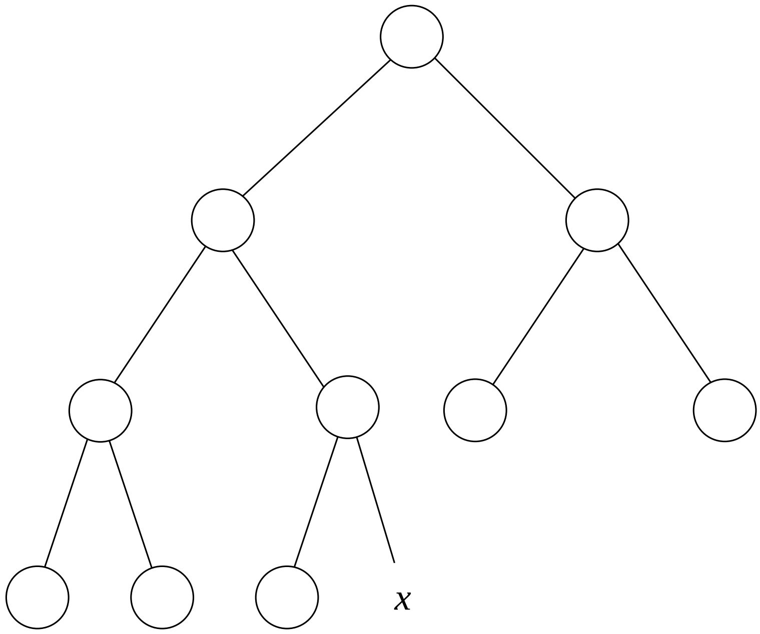
für best, average und worst case.

Weitere Einsatzmöglichkeit des Heaps

Verwende eine dynamisch sich ändernde Menge von Schlüsseln mit den Operationen

- `initheap` legt leeren Heap an
- `get_min` liefert das momentan Kleinste
- `del_min` entfernt das momentan Kleinste
- `insert(x)` fügt x hinzu
- `heapempty` testet, ob Heap leer ist

Idee für Einfügen: (schlecht: von oben nach unten)
besser: Füge neues Blatt mit Schlüssel x an, und lasse x hochsickern.

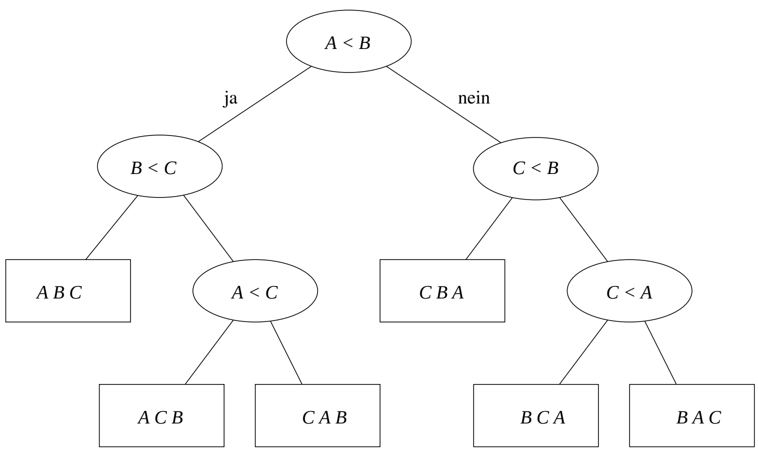



```
i = neuer Index;
while (a[i] < a[vater(i)]){
   tausche a[i] mit a[vater(i)];
   i = vater(i);
}
```

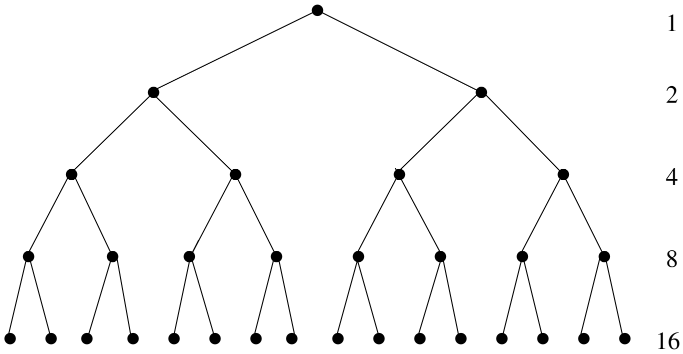
Aufwand $O(\log n)$

Untere Schranke für Sortieren durch Vergleichen

Entscheidungsbaum zur Sortierung von 3 Elementen:
gegeben A, B, C



Der Entscheidungsbaum zur Sortierung von n Elementen hat $n!$ Blätter.



Höhe = log k bei k Blättern

$$n! \geq n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot \frac{n}{2} \geq \left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right)$$

$$\begin{aligned}
\Rightarrow \log n! \quad &\geq \quad \log\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) = \frac{n}{2} \cdot \log\left(\frac{n}{2}\right) = \frac{n}{2}(\log n - 1) \\
&= \quad \frac{n}{2} \cdot \log n - \frac{n}{2} = \frac{n}{4} \cdot \log n + \frac{n}{4} \cdot \log n - \frac{n}{2} \\
&\geq \quad \frac{n}{4} \cdot \log n \text{ für } n \geq 4
\end{aligned}$$

$\Rightarrow$ Ein binärer Baum mit $n!$ Blättern hat mindestens die Höhe $\frac{n}{4} \cdot \log n$.

$\Rightarrow$ Jeder Sortieralgorithmus, der auf Vergleichen beruht, hat als Laufzeit mindestens $O(n \cdot \log n)$. Dies ist eine untere Schranke.

7.7 Bucket Sort

```
/***************************  BucketSort.java  *******************************/

import AlgoTools.IO;

/** Sortieren durch Verteilen auf Buckets (Faecher).
 *  Idee: 1.) Zaehlen der Haeufigkeiten b[i] einzelner Schluessel i;
 *        2.) Buckets durchlaufen und i-ten Schluessel b[i]-mal ausgeben.
 */

public class BucketSort {

  static final int N = 256;                       // Alphabetgroesse N

  public static char[] sort (char[] a) {          // sortiere Character-Array a
                                                  // und liefere Ergebnis zurueck

    char[] fertig = new char[a.length];           // Ergebniszeichenkette
    int[] b = new int[N];                         // N Buckets
    int i, j, k=0;                                // Laufvariablen

    for (i=0; i < N; i++) b[i] = 0;               // setze alle Buckets auf 0

    for (i=0; i < a.length; i++)                  // fuer jedes Eingabezeichen
        b[a[i]]++;                                // zustaendiges Bucket erhoehen

    for (i=0; i < N; i++)                         // fuer jedes Bucket
        for (j=0; j < b[i]; j++)                  // gemaess Zaehlerstand
            fertig[k++] = (char) i;               // sein Zeichen uebernehmen

    return fertig;                                // Zeichenkette zurueckgeben
  }

  public static void main (String argv[]) {

    char[] zeile, ergebnis;                               // 2 Zeichenfolgen

    zeile = IO.readChars("Bitte Zeichenkette:       "); // Zeichenkette einlesen
    ergebnis = sort(zeile);                               // Bucket-Sort aufrufen
    IO.println("sortiert mit Bucket-Sort: "+ergebnis);  // Ergebnis ausgeben
  }
}
// Aufwand: O(n) + O(N)   bei n(=a.length) zu sortierenden Zeichen
//                        aus einem N-elementigen Alphabet
```

7.8 Radix Sort

Idee: Sortieren von Strings über einem endlichen Alphabet durch mehrmaliges Anwenden von Bucket Sort.

Es soll eine Liste von Wörtern mit insgesamt n Buchstaben in $O(n)$ sortiert werden. Pro Buchstabe des Alphabets existiert ein Bucket.

Zunächst wird angenommen, dass alle Wörter die Länge N haben.

Es werden sukzessive Listen $W_N, W_{N-1}, \ldots, W_0$ gebildet.
W_N enthält die unsortierten Wörter in der gegebenen initialen Reihenfolge.
W_j mit $j \in [0, \ldots, N-1]$ enthält alle Wörter, aufsteigend sortiert bezüglich der Positionen $j+1, \ldots, N$.
Das Ergebnis der Sortiervorgänge ist W_0.

```
for (j = N; j > 0; j--) {
   verteile die Wörter aus W_j
   gemäß j-tem Buchstaben
   auf die Buckets;
   sammele Buckets auf nach W_{j-1}
}
```

Beispiel:

Zu sortieren seien die Strings

`hefe bach gaga cafe geha`

Es entstehen nacheinander folgende Listen (unterstrichen ist jeweils der Buchstabe, der im nächsten Schritt zum Verteilen benutzt wird):

$$\begin{array}{ll}
W_4: & \texttt{hef}\underline{\texttt{e}}\ \texttt{bac}\underline{\texttt{h}}\ \texttt{gag}\underline{\texttt{a}}\ \texttt{caf}\underline{\texttt{e}}\ \texttt{geh}\underline{\texttt{a}} \\
W_3: & \texttt{ga}\underline{\texttt{g}}\texttt{a}\ \texttt{ge}\underline{\texttt{h}}\texttt{a}\ \texttt{he}\underline{\texttt{f}}\texttt{e}\ \texttt{ca}\underline{\texttt{f}}\texttt{e}\ \texttt{ba}\underline{\texttt{c}}\texttt{h} \\
W_2: & \texttt{b}\underline{\texttt{a}}\texttt{ch}\ \texttt{h}\underline{\texttt{e}}\texttt{fe}\ \texttt{c}\underline{\texttt{a}}\texttt{fe}\ \texttt{g}\underline{\texttt{a}}\texttt{ga}\ \texttt{g}\underline{\texttt{e}}\texttt{ha} \\
W_1: & \underline{\texttt{b}}\texttt{ach}\ \underline{\texttt{c}}\texttt{afe}\ \underline{\texttt{g}}\texttt{aga}\ \underline{\texttt{h}}\texttt{efe}\ \underline{\texttt{g}}\texttt{eha} \\
W_0: & \texttt{bach cafe gaga geha hefe}
\end{array}$$

Die zugehörigen Buckets lauten:

	$W_4 \rightarrow W_3$	$W_3 \rightarrow W_2$	$W_2 \rightarrow W_1$	$W_1 \rightarrow W_0$
a	gaga geha		bach cafe gaga	
b				bach
c		bach		cafe
d				
e	hefe cafe		hefe geha	
f		hefe cafe		
g		gaga		gaga geha
h	bach	geha		hefe

Beim Aufsammeln werden die Buckets in alphabetischer Reihenfolge durchlaufen und ihre Inhalte konkateniert.

Um beim Einsammeln der Buckets nur die gefüllten anzufassen, ist es hilfreich, zu Beginn

```
char[][] nichtleer;
```

anzulegen. `nichtleer[j]` enthält die Buchstaben, welche in den zu sortierenden Wörtern an j-ter Position vorkommen. Verteile hierzu zunächst Positionen auf Buchstaben:

`pos[x]`=$\{j|$ Buchstabe x kommt an j-ter Position vor $\}$

x	pos[x]				
a	2	2	4	2	4
b	1				
c	3	1			
d					
e	2	4	4	2	
f	3	3			
g	1	3	1		
h	1	4	3		

Nach dem Einsammeln der Buckets werden die Buchstaben auf Positionen verteilt:

j	nichtleer[j]				
1	b	c	g	g	h
2	a	a	a	e	e
3	c	f	f	g	h
4	a	a	e	e	h

Obacht: Bei der Konstruktion von `nichtleer` müssen doppelte Einträge vermieden werden (möglich, da sie hintereinander stehen würden).

Bei Vermeidung der Doppelten hat `nichtleer[j]` folgende Gestalt:

j	nichtleer[j]			
1	b	c	g	h
2	a	e		
3	c	f	g	h
4	a	e	h	

Der Aufwand zur Konstruktion von `nichtleer` beträgt $O(n)$. Jedes Wort wird bzgl. jedes Buchstabens einmal verteilt und einmal aufgesammelt, jeweils in konstanter Zeit.

Bei Wörtern **unterschiedlicher** Länge bilde zunächst

`laenge[j]` $= \{w|$ Länge von Wort $w = j\}$

Der Aufwand hierfür beträgt $O(n)$.

Verteile im j-ten Durchlauf zunächst alle Wörter aus `laenge[j]`;
z.B. `fad` wird bzgl. des 3. Buchstabens verteilt, bevor W_3 verteilt wird.

Der Aufwand zum Verteilen und Aufsammeln der Listen $W_N, \ldots, W_0$ beträgt $O(n)$, da jedes Zeichen einmal zum Verteilen benutzt wird und einmal beim Aufsammeln unter Vermeidung der nichtleeren Buckets zu einem Schritt beiträgt.

7.9 Externes Sortieren

Problem: Sortieren auf Medien mit sequentiellem Lese- und Schreibzugriff.
Lösung: Wiederholtes Mischen von bereits sortierten Teilfolgen.

Ein *Lauf* ist eine monoton nicht fallende Teilfolge.

$$\underbrace{3 \quad 7 \quad 8}_{\text{Lauf}} \ \underbrace{2 \quad 9}_{\text{Lauf}} \ \underbrace{4}_{\text{Lauf}} \ \underbrace{1 \quad 5 \quad 6}_{\text{Lauf}}$$

Gegeben 3 Magnetbänder, mit initial $0, n, 0$ Läufen. Je 2 Bänder mischen ihre Läufe zusammen auf das dritte:

Band A:	0	0	8	3	0	2	1	0
Band B:	21	13	5	0	3	1	0	1
Band C:	0	8	0	5	2	0	1	0
								sortiert

(In jeder Spalte ist die momentane Anzahl der Läufe vermerkt.)

Allgemeine Regel:

Sei $fib(i)$ die i-te Fibonacci-Zahl.

Es habe Band	A	$fib(i)$	Läufe
	B	$fib(i-1)$	Läufe
	C	0	Läufe

dann mische $fib(i-1)$ Läufe von A und B zusammen auf Band C.

Kapitel 8

Objektorientierte Programmierung

Die Modellierung eines Ausschnittes der realen Welt geschieht durch eine Klassenhierarchie, d.h., gleichartige Objekte werden zu Klassen zusammengefasst, von denen durch Vererbung Spezialisierungen abgeleitet werden. Gleichartigkeit bedeutet die Übereinstimmung von objektbezogenen Datenfeldern und objektbezogenen Methoden. Die abgeleitete Klasse erbt von der Superklasse die dort definierten Datenfelder und Methoden, fügt ggf. eigene hinzu und kann ihnen ggf. durch Überschreiben eine neue Bedeutung geben. Jede Klasse besitzt einen oder mehrere Konstruktoren, die für das Instanziieren ihrer Objekte zuständig sind.

Datenfelder, die mit dem Schlüsselwort `static` deklariert werden, heißen *Klassenvariable*. Sie existieren pro Klasse genau einmal (unabhängig von der Zahl der kreierten Instanzen) und übernehmen die Rolle von globalen Variablen, auf die alle Objekte dieser Klasse zugreifen können.

Die Sichtbarkeit von Variablen und Methoden wird mit Hilfe von Modifiern geregelt. Ist ein Element einer Klasse mit keinem der Schlüsselworte `public`, `private` oder `protected` deklariert, dann ist es nur innerhalb von Klassen desselben Pakets sichtbar. Das Standardpaket besteht aus allen Klassen im aktuellen Arbeitsverzeichnis.

Die folgende Tabelle zeigt die Umstände, unter denen Klassenelemente der vier Sichtbarkeitstypen für verschiedene Klassen erreichbar sind.

Erreichbar für:	`public`	`protected`	paketsichtbar	`private`
Dieselbe Klasse	ja	ja	ja	ja
Subklasse im selben Paket	ja	ja	ja	nein
Subklasse in anderem Paket	ja	ja	nein	nein
Keine Subklasse, selbes Paket	ja	ja	ja	nein
Keine Subklasse, anderes Paket	ja	nein	nein	nein

```
/***************************  Datum.java  ***********************************/

import AlgoTools.IO;

/** Klasse Datum
 *  bestehend aus drei Integers (Tag, Monat, Jahr)
 *  mit zwei Konstruktoren zum Anlegen eines Datums
 *  und einer Methode zur Umwandlung eines Datums in einen String
 */

public class Datum {

  int tag;                                        // Datenfeld tag
  int monat;                                      // Datenfeld monat
  int jahr;                                       // Datenfeld jahr

  public Datum (int tag, int monat, int jahr){ // Konstruktor mit 3 Parametern
    this.tag   = tag;                             // initialisiere Tag
    this.monat = monat;                           // initialisiere Monat
    this.jahr  = jahr;                            // initialisiere Jahr
  }

  public Datum (int jahr){                        // Konstruktor mit 1 Parameter
    this(1, 1, jahr);                             // initialisiere 1.1. Jahr
  }

  public String toString(){                       // Methode ohne Parameter
    return(tag+"."+monat+"."+jahr);               // liefert Datum als String
  }

  public static void main (String argv[]) {

    Datum d;                                      // deklariere ein Datum

    d = new Datum (15,8,1972);                    // kreiere Datum 15.08.1972
    d = new Datum (1972);                         // kreiere Datum 01.01.1972
    d.jahr++;                                     // erhoehe Datum um ein Jahr
    IO.println(d.toString());                     // drucke Datum
    IO.println(d);                                // hier: implizites toString()
  }
}
```

```
/***************************  Person.java  **********************************/

import AlgoTools.IO;

/** Klasse Person
 *  bestehend aus Vorname, Nachname, Geburtsdatum
 *  mit einem Konstruktor zum Anlegen einer Person
 *  und zwei Methoden zum Ermitteln des Jahrgangs und des Alters
 */

public class Person {

  String vorname;                                   // Datenfeld Vorname
  String nachname;                                  // Datenfeld Nachname
  Datum  geb_datum;                                 // Datenfeld Geburtsdatum

  public Person                                     // Konstruktor mit
               (String vn,                          // Vorname
                String nn,                          // Nachname
                int t,                              // Geburtstag
                int m,                              // Geburtsmonat
                int j)                              // Geburtsjahr
  {
    vorname   = vn;                                 // initialisiere Vorname
    nachname  = nn;                                 // initialisiere Nachname
    geb_datum = new Datum(t,m,j);                   // initialisiere Geburtsdatum
  }

  public int jahrgang () {                          // Methode
    return geb_datum.jahr;                          // liefert Geburtsjahrgang
  }

  public int alter(){                               // Methode
    return 2000 - geb_datum.jahr;                   // liefert das Lebensalter
  }

  public static void main (String argv[]) {

    Person p;                                       // deklariere eine Person

    p = new Person("Willi","Wacker",22,8,1972);     // kreiere Person
    p.geb_datum.jahr++;                             // mache sie 1 Jahr juenger
    IO.print(p.vorname + " " + p.nachname);         // gib Name aus
    IO.println(" ist "+p.alter()+" Jahre alt.");    // gib Alter aus
  }
}
```

```
/***************************  Student.java  **********************************/

import AlgoTools.IO;

/** Klasse Student, spezialisiert die Klasse Person
 *  durch Klassenvariable next_mat_nr;
 *  durch weitere Datenfelder mat_nr, fach, jsb
 *  durch eigenen Konstruktor und durch eigene Methode jahrgang
 *  welche die Methode jahrgang der Klasse Person ueberschreibt
 */

public class Student extends Person {           // Student erbt von Person

  static int next_mat_nr = 100000;              // globale Klassenvariable

  int mat_nr;                                   // Matrikel-Nummer
  String fach;                                  // Studienfach
  int jsb;                                      // Jahr des Studienbeginns

  public Student                                // Konstruktor mit
               (String vn,                      // Vorname
                String nn,                      // Nachname
                int t,                          // Geburtstag
                int m,                          // Geburtsmonat
                int j,                          // Geburtsjahr
                String fach,                    // Studienfach
                int jsb)                        // Studienbeginn
  {
    super(vn, nn, t, m, j);                     // Konstruktor des Vorfahren
    this.fach   = fach;                         // initialisiere Fach
    this.jsb    = jsb;                          // initialisiere Studienbeginn
    mat_nr      = next_mat_nr++;                // vergib naechste Mat-Nr.
  }

  public int jahrgang() {                       // Methode liefert als Jahrgang
    return jsb;                                 // das Jahr des Studienbeginns
  }

  public static void main (String argv[]) {
    Student s;                                            // deklariere Student
    s = new Student("Willi","Wacker",22,8,1972,"BWL",1995);// kreiere Student
    IO.print(s.vorname + " " + s.nachname);               // gib Name aus und
    IO.println("'s Matrikelnummer lautet: " + s.mat_nr);   // Matrikelnummer
  }
}
```

```
/**************************  PersonTest.java *******************************/

import AlgoTools.IO;

/** Klasse PersonTest
 *  verwendet Instanzen der Klasse Person und der Klasse Student
 */

public class PersonTest {

  public static void main (String args[]) {

    Student s;                                         // Student
    Person p;                                          // Person

    p = new Person("Uwe","Meier",24,12,1971);          // kreiere Person
    s = new Student("Eva","Kahn",15,9,1972,"BWL",1998);  // kreiere Student

    IO.println(p.nachname+"'s Jahrgang: "+p.jahrgang()); // gib Jahrgang aus
                                                       // da p eine Person ist
                                                       // wird Geburtsjahrgang
                                                       // ausgegeben

    IO.println(s.nachname+"'s Jahrgang: "+s.jahrgang()); // gib Jahrgang aus
                                                       // da s ein Student ist
                                                       // wird
                                                       // Immatrikulationsjahr
                                                       // ausgegebn

    p = s;                                             // p zeigt auf s

    IO.println(p.nachname+"'s Jahrgang: "+p.jahrgang()); // gib Jahrgang aus
                                                       // da p auf Student
                                                       // zeigt, wird
                                                       // Immatrikulationsjahr
                                                       // ausgegeben


    if (p instanceof Student)                          // falls p Student ist
      IO.println(((Student)p).fach);                   // gib p's Fach aus

    s = (Student) p;                                   // s zeigt auf p
    IO.println(s.fach);                                // gib s's Fach aus
  }
}
```

```
/***************************  Mitarbeiter.java  ******************************/
/** abstrakte Klasse Mitarbeiter
 *  bestehend aus Name und Vorname und Konstruktor
 *  und einer abstrakten Methode
 */

public abstract class Mitarbeiter {
  String vorname;                            // Vorname
  String nachname;                           // Nachname
  Mitarbeiter() {}                           // Konstruktor
  abstract double monatsBrutto();            // abstrakte Methode
}


/***************************  Arbeiter.java  *********************************/
/** Klasse Arbeiter
 *  bestehend aus Stundenlohn und Stundenzahl
 *  und der Implementierung der abstrakten Methode der Superklasse
 */

public class Arbeiter extends Mitarbeiter {
  int std_lohn;                              // Stundenlohn
  int std_zahl;                              // Arbeitsstunden im Monat

  public double monatsBrutto() {             // berechnet Monatsbrutto
    return std_lohn * std_zahl;              // aus Stundenlohn und Stundenzahl
  }
}


/***************************  Angestellter.java  *****************************/
/** Klasse Angestellter
 *  bestehend aus grundgehalt und ortszuschlag
 *  und der Implementierung der abstrakten Methode der Superklasse
 */

public class Angestellter extends Mitarbeiter {
  int grundgehalt;                           // Grundgehalt
  int ortszuschlag;                          // Ortszuschlag

  public double monatsBrutto() {             // berechnet Monatsbrutto
    return grundgehalt + ortszuschlag;       // aus Grundgehalt und Ortszuschlag
  }
}
```

```
/***************************  Firma.java  ************************************/

/** Klasse Firma
 *  bestehend aus einem Array fuer Mitarbeiter
 */

public class Firma {

  static final int N = 4;                             // Zahl der Mitarbeiter
  static Mitarbeiter[] ma;                            // Array von Mitarbeitern

  public static void main (String argv[]){

    ma                 = new Mitarbeiter[N];    // Platz fuer Mitarbeiter-Array

    ma[0]              = new Arbeiter();        // lege 1. Mitarbeiter an
    ma[1]              = new Angestellter();    // lege 2. Mitarbeiter an
    ma[2]              = new Angestellter();    // lege 3. Mitarbeiter an
    ma[3]              = new Arbeiter();        // lege 4. Mitarbeiter an

    double bruttoSumme = 0.0;                   // initialisiere Bruttosumme

    for (int i=0; i < N; i++)                   // durchlaufe Mitarbeiter-Array
        bruttoSumme += ma[i].monatsBrutto();    // und addiere Monats-Brutto

  }

}
```

```
/***************************  Kind.java  *************************************/

import AlgoTools.IO;

/** Klasse Kind
 *  bestehend aus Nummer und Verweis auf Nachbarkind
 *  mit Konstruktor zum Anlegen eines Kindes
 *  ermoeglicht Implementierung eines Abzaehlreims mit k Silben fuer n Kinder
 */

public class Kind {

  int  nr;                                      // Nummer
  Kind next;                                    // Verweis auf naechstes Kind

  public Kind (int nr) {                        // Konstruktor fuer Kind
    this.nr   = nr;                             // initialisiere Nummer
  }

  public static void main (String argv[]) {

    int i;                                      // Laufvariable
    int n=IO.readInt("Wie viele Kinder ? ");    // erfrage Kinderzahl
    int k=IO.readInt("Wie viele Silben ? ");    // erfrage Silbenzahl

    Kind erster, letzter, index;                // deklariere drei Kinder

    erster = letzter =  new Kind(0);            // kreiere erstes Kind

    for (i=1; i < n; i++){                      // erzeuge n-1 mal
      index = new Kind(i);                      // ein Kind mit Nummer i
      letzter.next = index;                     // erreichbar vom Vorgaenger
      letzter = index;
    }
    letzter.next = erster;                      // schliesse Kreis

    index = letzter;                            // beginne bei letztem Kind
    while (index.next != index) {               // solange ungleich Nachfolger
      for (i=1; i<k; i++) index=index.next;     // gehe k-1 mal weiter
      IO.print("Ausgeschieden: ");              // Gib die Nummer des Kindes aus,
      IO.println(index.next.nr, 5);             // welches jetzt ausscheidet
      index.next = index.next.next;             // bestimme neuen Nachfolger
    }
    IO.println("Es bleibt uebrig: " + index.nr);
  }
}
```

Kapitel 9

Abstrakte Datentypen

Def.: Ein *abstrakter Datentyp* ADT ist eine Datenstruktur zusammen mit darauf definierten Operationen.

Java unterstützt den Umgang mit ADTs durch die Bereitstellung von Klassen und Interfaces.

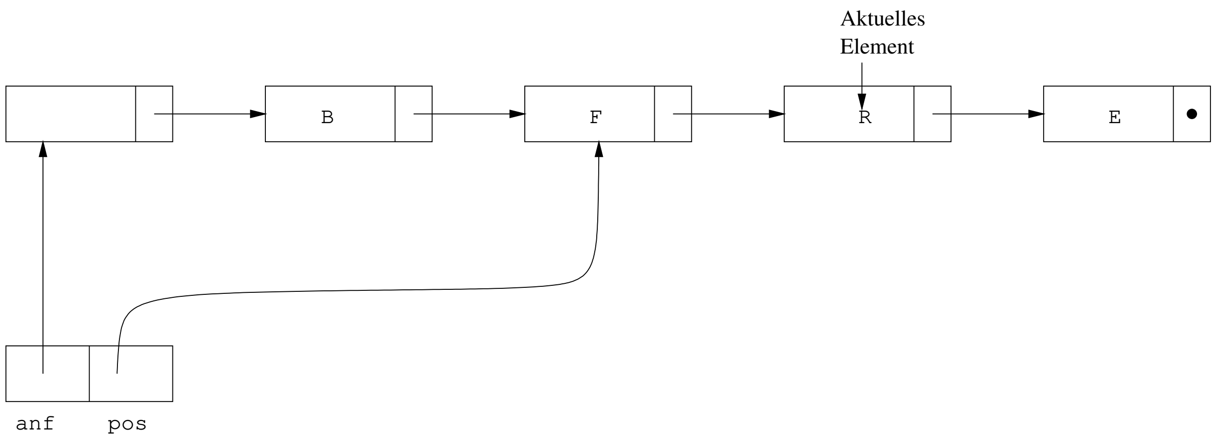
9.1 Liste

Def.: Eine *Liste* ist eine (ggf. leere) Folge von Elementen zusammen mit einem so genannten (ggf. undefinierten) *aktuellen* Element.

Schnittstelle des ADT `Liste`:

`empty`	:	Liste	$\rightarrow$	boolean	liefert true, falls Liste leer
`endpos`	:	Liste	$\rightarrow$	boolean	liefert true, wenn Liste abgearbeitet
`reset`	:	Liste	$\rightarrow$	Liste	das erste Listenelement wird zum aktuellen
`advance`	:	Liste	$\rightarrow$	Liste	der Nachfolger des akt. wird zum aktuellen
`elem`	:	Liste	$\rightarrow$	Objekt	liefert das aktuelle Element
`insert`	:	Liste $\times$ Objekt	$\rightarrow$	Liste	fügt vor das aktuelle Element ein Element ein; das neu eingefügte wird zum aktuellen
`delete`	:	Liste	$\rightarrow$	Liste	löscht das aktuelle Element; der Nachfolger wird zum aktuellen

Implementation einer Liste



`anf` zeigt auf den ersten Listen-Eintrag (leerer Inhalt),
`pos` zeigt auf den Listen-Eintrag **vor** dem Listen-Eintrag mit dem aktuellen Element.

```
/****************************  Liste.java  **********************************/
import AlgoTools.IO;
/** ADT Liste mit empty, endpos, reset, advance, elem, insert, delete        */

public class Liste {

    private static class ListenEintrag {
        Object        inhalt;      // Inhalt des ListenEintrags
        ListenEintrag next;        // zeigt auf naechsten ListenEintrag
    }

    private ListenEintrag anf;     // zeigt auf nullten ListenEintrag
    private ListenEintrag pos;     // zeigt auf ListenEintrag vor dem aktuellen

    public Liste() {               // kreiert eine leere Liste
        pos = anf = new ListenEintrag();
        anf.next  = null;
    }

    public boolean empty() { return anf.next == null; } // true, wenn Liste leer

    public boolean endpos() { return pos.next == null; }// true, wenn am Ende

    public void reset() {  pos = anf; }       // rueckt an den Anfang der Liste

    public void advance() {                             // rueckt in Liste vor
        if (endpos()) IO.error("in advance: am Ende");
        pos = pos.next;
    }

    public Object elem() {         // liefert Inhalt des aktuellen Eintrags
        if (endpos()) IO.error("in elem: kein aktueller ListenEintrag");
        return pos.next.inhalt;
    }

    public void insert(Object x) {                 // fuegt ListenEintrag ein
        ListenEintrag hilf = new ListenEintrag();  // Das neue Listenelement
        hilf.inhalt        = x;                    // kommt vor das aktuelle
        hilf.next          = pos.next;
        pos.next           = hilf;
    }

    public void delete() {                         // entfernt aktuelles Element
        if (endpos()) IO.error("in delete: kein aktueller ListenEintrag");
        pos.next = pos.next.next;
    }
}
```

```
/****************************  ListeTest.java  ******************************/

import AlgoTools.IO;

/** Testet die Klasse Liste mit Studenten
 */

public class ListeTest {

  public static void main (String argv[]) {

    Liste l = new Liste();                                      // kreiere Liste
    Student s;                                                  // deklariere Student

    s = new Student("Willi","Wacker",22,8,1972,"BWL",1995);// kreiere Student
    l.insert(s);                                                // fuege in Liste ein
    l.advance();                                                // eins weiter in l

    s = new Student("Erika","Muster",28,2,1970,"VWL",1994);// kreiere Student
    l.insert(s);                                                // fuege in Liste ein
    l.advance();                                                // eins weiter in l

    s = new Student("Hein","Bloed",18,5,1973,"CLK",1996);  // kreiere Student
    l.insert(s);                                                // fuege in Liste ein
    l.advance();                                                // eins weiter in l

    s = new Student("Susi","Sorglos",10,7,1973,"JUR",1996);// kreiere Student
    l.insert(s);                                                // fuege in Liste ein

    l.reset();                                                  // an den Anfang

    while (!l.endpos()) {                                       // 1.,3.,5.. loeschen
      l.delete();
      if (!l.endpos())
        l.advance();
    }
    l.reset();                                                  // an den Anfang

    while (!l.endpos()) {                                       // Liste ausgeben
      IO.println(((Student)l.elem()).vorname+" "+((Student)l.elem()).nachname);
      l.advance();                                              // ein weiter in l
    }
    l.reset();                                                  // an den Anfang

    while (!l.empty()) l.delete();                              // Liste loeschen
  }
}
```

9.2 Keller

Def.: Ein *Keller* ist eine (ggf. leere) Folge von Elementen zusammen mit einem so genannten (ggf. undefinierten) *Top*-Element.

Schnittstelle des ADT `Keller`:

`push`	:	Keller × Objekt	→	Keller	legt Element auf Keller
`pop`	:	Keller	→	Keller	entfernt oberstes Element
`top`	:	Keller	→	Objekt	liefert oberstes Element
`empty`	:	Keller	→	boolean	liefert `true`, falls Keller leer ist

Semantik der Kelleroperationen:

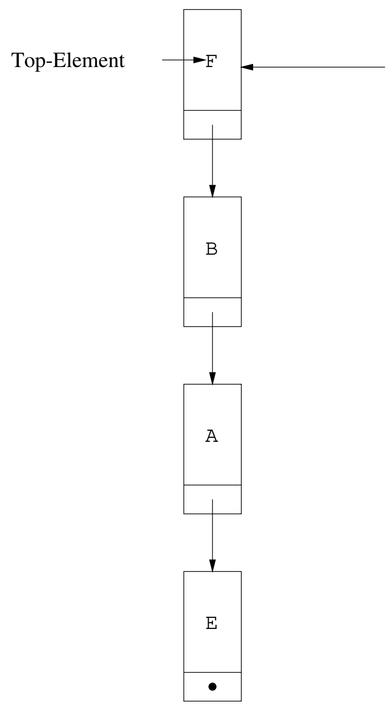
A1) Ein neu konstruierter Keller ist leer.

A2) Nach einer `Push`-Operation ist ein Keller nicht leer.

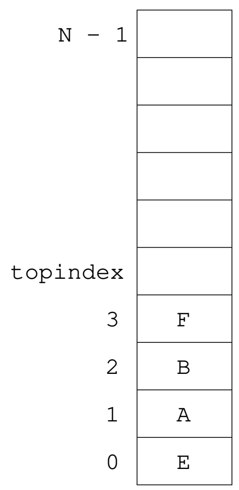
A3) Nach einer `Push-Pop`-Operation ist der Keller unverändert.

A4) Nach der `Push`-Operation mit dem Element x liefert die `Top`-Operation das Element x.

Implementation eines Kellers mit Verweisen



Implementation eines Kellers mit einem Array



```
/****************************  Keller.java  *********************************/

import AlgoTools.IO;

/** Abstrakter Datentyp Keller mit den Methoden push, pop, top, empty
 */

public class Keller {

    private static class KellerEintrag {
        Object inhalt;          // Inhalt des KellerEintrags
        KellerEintrag next;     // zeigt auf naechsten KellerEintrag
    }

    private KellerEintrag top;  // zeigt auf obersten KellerEintrag

    public Keller() {                                      // legt leeren Keller an
        top = null;
    }

    public boolean empty() {                               // liefert true,
        return top == null;                                // falls Keller leer
    }

    public void push(Object x) {                           // legt Objekt x
        KellerEintrag hilf = new KellerEintrag();          // auf den Keller
        hilf.inhalt = x;
        hilf.next   = top;
        top         = hilf;
    }

    public Object top() {                                  // liefert oberstes
        if (empty()) IO.error("in top: Keller leer");      // Kellerelement
        return top.inhalt;
    }

    public void pop() {                                    // entfernt oberstes
        if (empty()) IO.error("in pop: Keller leer");      // Kellerelement
        top = top.next;
    }
}
```

```
/*****************************  Reverse.java  ********************************/

import AlgoTools.IO;

/** Liest eine Folge von ganzen Zahlen ein
 *  und gibt sie in umgekehrter Reihenfolge wieder aus.
 *  Verwendet wird die Wrapper-Klasse Integer,
 *  welche Objekte vom einfachen Typ int enthaelt.
 *  Vor dem Einfuegen in den Keller werden mit new Integer diese Objekte
 *  erzeugt, nach dem Auslesen aus dem Keller werden sie nach int gecastet.
 */

public class Reverse {

  public static void main (String argv[]) {

    Keller k = new Keller();                       // lege leeren Keller an

    int[] a = IO.readInts("Bitte Zahlenfolge:   "); // lies Integer-Folge ein

    for (int i=0; i<a.length; i++)                 // pushe jede Zahl als
        k.push(new Integer(a[i]));                 // Integer-Objekt

    IO.print("Umgekehrte Reihenfolge:");
    while (!k.empty()) {                           // solange Keller nicht leer
        IO.print(" "+((Integer)k.top()).intValue());// gib Top-Element aus
        k.pop();                                   // entferne Top-Element
    }
    IO.println();
  }
}
```

```
/****************************  Klammer.java  ********************************/

import AlgoTools.IO;

/** Ueberprueft Klammerung mit Hilfe eines Kellers
 */

public class Klammer {

  public static void main(String[] argv) {

    char[] c;                               // Eingabezeichenkette
    int i = 0;                              // Laufindex in char[] c
    boolean fehler = false;                 // Abbruchkriterium
    Keller k = new Keller();                // Keller fuer Zeichen

    c = IO.readChars("Bitte Klammerausdruck eingeben: ");
    IO.print("                                ");

    while ((i < c.length) && (!fehler)) {
      switch (c[i]) {
        case '(':                           // '(' und '[' auf den Keller legen
        case '[': k.push(new Character(c[i])); break;
        case ')': if (!k.empty() && ((Character)k.top()).charValue() == '(')
                       k.pop();             // '(' vom Keller nehmen
                  else fehler = true; break;
        case ']': if (!k.empty() && ((Character)k.top()).charValue() == '[')
                       k.pop();             // '[' vom Keller nehmen
                  else fehler = true; break;
        default:  break;                    // beliebiges Zeichen, keine Klammer
      }
      IO.print(" ");                        // naechstes Zeichen
      i++;
    }

    if ((i==c.length) && !fehler && k.empty())
         IO.println("korrekt geklammert");
    else IO.println("^ nicht korrekt geklammert");
  }
}
```

```
/****************************  CharKeller.java  *****************************/
import AlgoTools.IO;

/** Abstrakter Datentyp Character-Keller mit Elementen vom Typ char           */

public class CharKeller extends Keller {

    public CharKeller() {                      // legt leeren CharKeller an
        super();
    }

    public void push(char x) {                 // legt char x auf den Keller
        push(new Character(x));
    }

    public char ctop() {                       // liefert oberstes Kellerelement
        return ((Character)top()).charValue();
    }
}


/****************************  Postfix.java  *******************************/
import AlgoTools.IO;

/** Wandelt Infix-Ausdruck in Postfix-Ausdruck um.
 *  Vorausgesetzt wird eine syntaktisch korrekte Eingabe,
 *  bestehend aus den Operatoren +,-,*,/ sowie den Operanden a,b,...,z
 *  und den oeffnenden und schliessenden Klammern. Beispiel:(a+b)*c-d/f
 *  Ausgabe ist der aequivalente Postfixausdruck.  Beispiel: ab+c*df/-
 *  Verwendet wird ein Character-Keller, der die bereits gelesenen
 *  oeffnenden Klammern sowie die Operatoren speichert.
 */

public class Postfix {

  public static void main(String[] argv) {

    CharKeller k = new CharKeller();                 // Character-Keller
    char[] infix;                                    // Eingabezeile
    char c;                                          // aktuelles Zeichen

    infix = IO.readChars("Bitte Infix-Ausdruck (+,-,*,/,a,...,z): ");
    IO.print("umgewandelt in Postfix:                 ");

    for (int i=0; i<infix.length; i++) {             // durchlaufe Infixstring

        c = infix[i];                                // aktuelles Zeichen
```

```
      switch (c) {

          case '('  :  k.push(c);                    // '(' auf den Stack
                       break;


          case ')'  :  while ( k.ctop() != '(' ) {   // Keller bis vor '('
                           IO.print(k.ctop());       // ausgeben
                           k.pop();                  // und leeren
                       }
                       k.pop();                      // und '(' entfernen
                       break;


          case '+'  :
          case '-'  :  while (!k.empty()             // Keller bis vor erste
                              && k.ctop() != '(') {  // oeffnende Klammer
                           IO.print(k.ctop());       // ausgeben
                           k.pop();                  // und leeren
                       }
                       k.push(c);                    // lege letztes Zeichen ab
                       break;


          case '*'  :
          case '/'  :  if (!k.empty()                // solange Keller
                           && (k.ctop()=='*'         // * enthaelt
                           || k.ctop()=='/')) {      // oder / enthaelt
                               IO.print(k.ctop());   // gib Operator aus
                               k.pop();              // und entferne ihn
                       }
                       k.push(c);                    // lege letztes Zeichen ab
                       break;


          default   :  if (c>='a' && c<='z')         // falls Operand vorliegt
                           IO.print(c);              // direkt ausgeben
      }
    }

    while (!k.empty()) {                             // beim Eingabeende
        IO.print(k.ctop());                          // Keller ausgeben
        k.pop();                                     // und leeren
    }
    IO.println();
  }
}
```

9.3 Exceptions

Die Fehlerbehandlung in Java basiert auf der Idee, daß bei Vorliegen eines fehlerhaften Zustands eine entsprechende Fehlerbedingung geworfen wird und längs der geschachtelten Aufrufhierarchie jede Methode versuchen kann, diese Fehlermeldung zu fangen. Dieses Konzept wird über die Vokabeln `throw`, `try` und `catch` realisiert.

Im folgenden Beispiel wird zunächst eine eigene, von der Klasse `Exception` abgeleitete, Fehlerklasse `KellerFehler` formuliert.

Die Klasse `AusnahmeKeller` unterscheidet sich von der bereits eingeführten Klasse `Keller` nur dadurch, daïn den Methoden `top` und `pop` anstelle des Aufrufs von IO.error nun der eigens definierte `KellerFehler` geworfen wird.

Die Klasse `AusnahmeKellerTest` liest analog zur Klasse `Reverse` eine Zahlenfolge ein und gibt sie in umgekehrter Reihenfolge wieder aus. Durch die Verwendung eines AusnahmeKellers kann ein eventuell auftretender Fehler abgefangen und behandelt werden.

```
/*************************** KellerFehler.java    ****************************/

/** Definition einer eigenen Ausnahme                                         */

public class KellerFehler  extends Exception   {

  public KellerFehler (String s) { super(s); };

}
```

```
/************************** AusnahmeKeller.java *****************************/

/** Abstrakter Datentyp Keller mit den Methoden push, pop, top, empty        */

public class AusnahmeKeller {

    private static class KellerEintrag {
        Object inhalt;          // Inhalt des KellerEintrags
        KellerEintrag next;     // zeigt auf naechsten KellerEintrag
    }

    private KellerEintrag top;  // zeigt auf obersten KellerEintrag

    public AusnahmeKeller () {                         // legt leeren Keller an
        top = null;
    }

    public boolean empty() {                           // liefert true,
        return top == null;                            // falls Keller leer
    }

    public void push(Object x) {                       // legt Objekt x
        KellerEintrag hilf = new KellerEintrag();      // auf den Keller
        hilf.inhalt = x;
        hilf.next   = top;
        top         = hilf;
    }

    public Object top() throws KellerFehler {          // liefert oberstes
        if (empty())                                   // Kellerelement
          throw new KellerFehler("in top: Keller leer");
        return top.inhalt;
    }

    public void pop() throws KellerFehler  {           // entfernt oberstes
        if (empty())                                   // Kellerelement
          throw new KellerFehler("in pop: Keller leer");
        top = top.next;
    }
}
```

```
/********************* AusnahmeKellerTest ********************************/

import AlgoTools.IO;

/** Liest eine Folge von ganzen Zahlen ein
 *  und gibt sie in umgekehrter Reihenfolge wieder aus.
 *  Verwendet wird die Wrapper-Klasse Integer,
 *  welche Objekte vom einfachen Typ int enthaelt.
 *  Vor dem Einfuegen in den Keller werden mit new Integer diese Objekte
 *  erzeugt, nach dem Auslesen aus dem Keller werden sie nach int gecastet.
 */

public class AusnahmeKellerTest {

  public static void main (String argv[]) {

    AusnahmeKeller k = new AusnahmeKeller();        // lege leeren Keller an

    int[]a = IO.readInts("Bitte Zahlenfolge:    "); // lies Integer-Folge ein

    try {                                           // versuche

      for (int i=0; i<a.length; i++)                // pushe jede Zahl als
        k.push(new Integer(a[i]));                  // Integer-Objekt

      IO.print("Umgekehrte Reihenfolge:");
      while (!k.empty()) {                          // solange Keller nicht leer
        IO.print(" "+((Integer)k.top()).intValue());// drucke
        k.pop();                                    // entferne Top-Element
      }
      IO.println();

    } catch (KellerFehler e) {                      // behandle Kellerfehler
        IO.println("Problem im Keller: " + e);
    }
  }
}
```

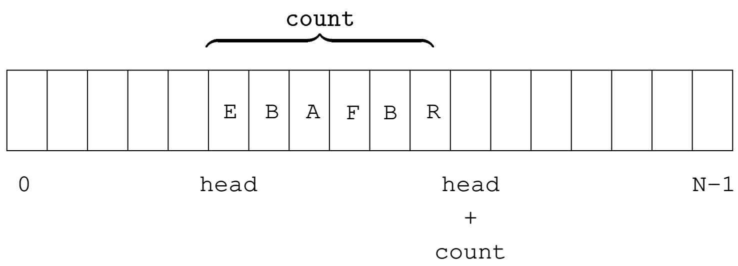
9.4 Schlange

Def.: Eine *Schlange* ist eine (ggf. leere) Folge von Elementen zusammen mit einem so genannten (ggf. undefinierten) *Front*-Element.

Schnittstelle des ADT `Schlange`:

`enq`	:	Schlange $\times$ Objekt	$\to$	Schlange	fügt Element hinten ein
`deq`	:	Schlange	$\to$	Schlange	entfernt vorderstes Element
`front`	:	Schlange	$\to$	Objekt	liefert vorderstes Element
`empty`	:	Schlange	$\to$	boolean	liefert `true`, falls Schlange leer ist

Implementation einer Schlange mit einem Array



```
/*****************************  Schlange.java  ******************************/

import AlgoTools.IO;

/** ADT Schlange mit den Methoden enq, deq, front, empty
 */

public class Schlange {

    private Object[] inhalt;                    // Array fuer Schlangenelemente
    private int head;                           // Index fuer Schlangenanfang
    private int count;                          // Anzahl Schlangenelemente

    public Schlange(int N) {                    // Konstruktor fuer leere Schlange
        inhalt = new Object[N];                 // besorge Platz fuer N Objekte
        head    = 0;                            // initialisiere Index fuer Anfang
        count   = 0;                            // initialisiere Anzahl
    }

    private boolean full() {                    // Testet, ob Schlange voll ist
        return count==inhalt.length;            // Anzahl gleich Arraylaenge?
    }

    public boolean empty() {                    // Testet, ob Schlange leer ist
        return count==0;                        // Anzahl gleich 0?
    }

    public void enq( Object x ) {               // Fuegt x hinten ein
        if (full()) IO.error ("in enq: Schlange ist voll!");
        inhalt[(head+count)%inhalt.length]=x;// Element einfuegen
        count++;                                // Anzahl inkrementieren
    }

    public void deq() {                         // Entfernt vorderstes Element
        if (empty()) IO.error("in deq: Schlange ist leer!");
        head = (head + 1) % inhalt.length;      // Anfang-Index weiterruecken
        count--;                                // Anzahl dekrementieren
    }

    public Object front() {                     // Liefert Element,
        if (empty()) IO.error("in front: Schlange ist leer!");
        return inhalt[head];                    // welches am Anfang-Index steht
    }
}
```

```
/****************************  SchlangeTest.java  **************************/

import AlgoTools.IO;

/** Programm zum Testen der Methoden des ADT Schlange.
 *  Liest Zeichenketten und reiht sie in eine Schlange ein.
 *  Bei Eingabe einer leeren Zeichenkette wird die jeweils vorderste
 *  aus der Schlange ausgegeben und entfernt.
 */

public class SchlangeTest {

  public static void main(String argv[]) {

    Schlange q = new Schlange(100);              // konstruiere Schlange mit
                                                 // Platz fuer 100 Objekte

    IO.println("Bitte Schlange fuellen durch Eingabe eines Wortes.");
    IO.println("Bitte Schlangen-Kopf entfernen durch Eingabe von RETURN.");

    do {                                         // Beginn der Schleife

      String eingabe = IO.readString("Input: "); // fordere String an

      if ( eingabe.length()>0 )                  // falls Eingabe != RETURN

          q.enq(eingabe);                        // fuege in Schlange ein

      else                                       // falls EINGABE == RETURN

          if ( !q.empty() ){                     // sofern Schlange nicht leer
              IO.println("entfernt: " +          // kuendige Ausgabe des
                         q.front() );            // Frontelements an
              q.deq();                           // entferne Frontelement
          }

    } while ( !q.empty() );                      // Ende der Schlangen-Schleife
    IO.println("Schlange ist jetzt leer.");
  }
}
```

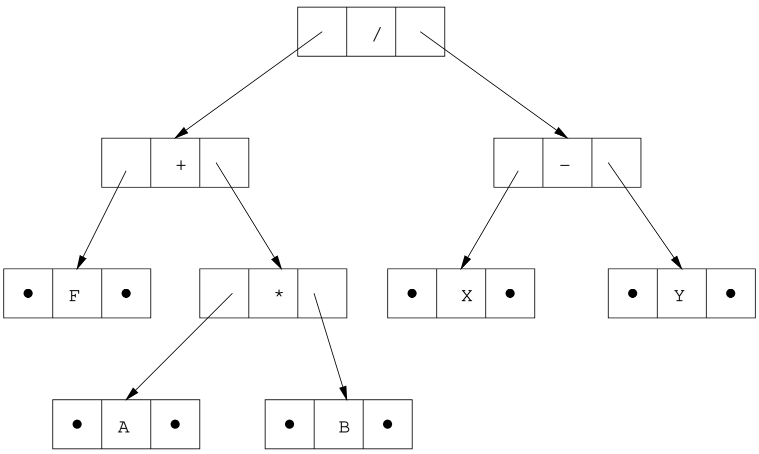
9.5 Baum

Def.: Ein *binärer Baum* ist entweder leer oder besteht aus einem Knoten, dem ein Element und zwei binäre Bäume zugeordnet sind.

Schnittstelle des ADT `Baum`:

`empty`	:	Baum	→	boolean	liefert `true`, falls Baum leer ist
`left`	:	Baum	→	Baum	liefert linken Teilbaum
`right`	:	Baum	→	Baum	liefert rechten Teilbaum
`value`	:	Baum	→	Objekt	liefert Wurzelelement

Implementation eines Baumes mit Verweisen



Traversierungen

Eine Traversierung eines binären Baumes besteht aus dem systematischen Besuchen aller Knoten in einer bestimmten Reihenfolge.

Traversierungen dieses Baumes

Preorder:	`/ + F * A B - X Y`
Inorder:	`F + A * B / X - Y`
Postorder:	`F A B * + X Y - /`
Klammerinorder:	`( ( F + ( A * B) ) / ( X - Y ) )`

```
/**************************  Baum.java  ************************************/
import AlgoTools.IO;

/** Klasse Baum  mit drei Konstruktoren und vier Methoden.
 *  Ein Baum besteht aus den Datenfeldern inhalt, links, rechts.
 */

public class Baum {

  Object inhalt;                            // Inhalt
  Baum links, rechts;                       // linker, rechter Teilbaum

  public final static Baum LEER = new Baum(); // leerer Baum als Klassenkonst.

  public Baum () {                          // konstruiert einen leeren Baum
    inhalt = null;                          // kein Inhalt
    links  = null;                          // keine
    rechts = null;                          // Kinder
  }

  public Baum (Object x) {                  // konstruiert ein Blatt
    this(LEER, x, LEER); }                  // mit Objekt x

  public Baum (Baum l, Object x, Baum r) { // konstruiert einen Baum
    inhalt = x;                             // aus einem Objekt x und
    links  = l;                             // einem linken Teilbaum
    rechts = r; }                           // und einem rechten Teilbaum

  public boolean empty () {                 // liefert true,
    return (inhalt == null);                // falls Baum leer ist
  }

  public Baum left () {                     // liefert linken Teilbaum
    if (empty()) IO.error("in left: leerer Baum");
    return links;
  }

  public Baum right () {                    // liefert rechten Teilbaum
    if (empty()) IO.error("in right: leerer Baum");
    return rechts;
  }

  public Object value () {                  // liefert Objekt in der Wurzel
    if (empty()) IO.error("in value: leerer Baum");
    return inhalt;
  }
}
```

```
/**************************  Traverse.java  *********************************/

import AlgoTools.IO;

/** Klasse Traverse
 *  bestehend aus vier statischen Methoden
 *  zum Traversieren von Baeumen
 */

public class Traverse {

  public static void inorder(Baum b) {        // Inorder-Traversierung
    if (!b.empty()) {                         // falls Baum nicht leer,
        inorder (b.left());                   // steige links ab
        IO.print(b.value());                  // gib Knoteninhalt aus
        inorder (b.right());                  // steige rechts ab
    }
  }

  public static void preorder(Baum b) {       // Preorder-Traversierung
    if (!b.empty()) {                         // falls Baum nicht leer,
        IO.print(b.value());                  // gib Knoteninhalt aus
        preorder(b.left());                   // steige links ab
        preorder(b.right());                  // steige rechts ab
    }
  }

  public static void postorder(Baum b) {      // Postorder-Traversierung
    if (!b.empty()) {                         // falls Baum nicht leer,
        postorder(b.left());                  // steige links ab
        postorder(b.right());                 // steige rechts ab
        IO.print (b.value());                 // gib Knoteninhalt aus
    }
  }

  public static void klammerinorder(Baum b) { // Klammerinorder-Traversierung
    if (!b.empty()) {                         // falls Baum nicht leer
      if (!b.left().empty()) IO.print("(");   // "("
      klammerinorder(b.left());               // linker Sohn
      IO.print(b.value());                    // Wurzel von b
      klammerinorder(b.right());              // rechter Sohn
      if (!b.right().empty()) IO.print(")");  // ")"
    }
  }
}
```

```
/*****************************  TiefenSuche.java  ***************************/

import AlgoTools.IO;

/** Klasse Tiefensuche enthaelt statische Methode tiefenSuche,
 *  die mit Hilfe eines Kellers eine iterative Tiefensuche
 *  auf einem Baum durchfuehrt (= preorder)
 */

public class TiefenSuche {

  public static void tiefenSuche (Baum wurzel) { // starte bei wurzel

    Baum b;                                 // Hilfsbaum

    Keller k = new Keller();                // konstruiere einen Keller

    if (!wurzel.empty()) k.push(wurzel);  // lege uebergebenen Baum in Keller

    while (!k.empty()) {                    // solange Keller noch Baeume enthaelt

        b = (Baum)k.top();                  // besorge Baum aus Keller
        k.pop();                            // und entferne obersten Eintrag

        do {
            IO.print(b.value());            // gib Wert der Baumwurzel aus
            if (!b.right().empty())         // falls es rechten Sohn gibt,
                k.push(b.right());          // lege rechten Sohn auf den Keller
            b = b.left();                   // gehe zum linken Sohn
        } while (!b.empty());               // solange es linken Sohn gibt
    }
  }
}
```

```
/*****************************  BreitenSuche.java  **************************/

import AlgoTools.IO;

/** Klasse BreitenSuche enthaelt statische Methode breitenSuche,
 *  die mit Hilfe einer Schlange eine iterative Breitensuche
 *  auf einem Baum durchfuehrt
 */

public class BreitenSuche {

  public static void breitenSuche (Baum wurzel) {// starte bei wurzel

    Baum b;                               // Hilfsbaum

    Schlange s = new Schlange(100);       // konstruiere eine Schlange

    if (!wurzel.empty()) s.enq(wurzel);   // lege uebergebenen Baum in Schlange

    while (!s.empty()) {                  // solange Schlange nicht leer

        b = (Baum)s.front();              // besorge Baum aus Schlange
        s.deq();                          // und entferne vordersten Eintrag

        IO.print(b.value());              // gib Wert der Baumwurzel aus

        if (!b.left().empty())            // falls es linken Sohn gibt,
               s.enq(b.left());           // haenge linken Sohn an Schlange
        if (!b.right().empty())           // falls es rechten Sohn gibt,
               s.enq(b.right());          // haenge rechten Sohn an Schlange
    }
  }
}
```

```
/****************************  TraverseTest.java  ***************************/
                                                        //         '/'
import AlgoTools.IO;                                    //      /       \
                                                        //      +       -
/** Traversierungen des binaeren Baums mit Operanden in //    /   \   /   \
 *  den Blaettern und Operatoren in den inneren Knoten: //    F   *   X   Y
 */                                                     //       / \
                                                        //       A B
public class TraverseTest {

    public static void main(String[] argv) {

        Baum a     = new Baum(new Character('A'));
        Baum b     = new Baum(new Character('B'));
        Baum mal   = new Baum(a, new Character('*'), b);
        Baum f     = new Baum(new Character('F'));
        Baum plus  = new Baum(f, new Character('+'), mal);
        Baum x     = new Baum(new Character('X'));
        Baum y     = new Baum(new Character('Y'));
        Baum minus = new Baum(x, new Character('-'), y);
        Baum durch = new Baum(plus, new Character('/'), minus);

        IO.print("Preorder:        ");
        Traverse.preorder(durch);         // Ausgabe: /+F*AB-XY
        IO.println();

        IO.print("Inorder:         ");
        Traverse.inorder(durch);          // Ausgabe: F+A*B/X-Y
        IO.println();

        IO.print("Postorder:       ");
        Traverse.postorder(durch);        // Ausgabe: FAB*+XY-/
        IO.println();

        IO.print("Klammer-Inorder: ");
        Traverse.klammerinorder(durch);   // Ausgabe: ((F+(A*B))/(X-Y))
        IO.println();

        IO.print("Tiefensuche:     ");
        TiefenSuche.tiefenSuche(durch);   // Ausgabe: /+F*AB-XY
        IO.println();

        IO.print("Breitensuche:    ");
        BreitenSuche.breitenSuche(durch); // Ausgabe: /+-F*XYAB
        IO.println();
    }
}
```

```java
/*****************************  PostfixBaumBau.java  ************************/

import AlgoTools.IO;

/** Klasse PostfixBaumBau enthaelt statische Methode postfixBaumBau,
 *  die einen Postfix-Ausdruck uebergeben bekommt
 *  und den zugehoerigen Baum zurueckliefert.
 *  Verwendet wird ein Keller ueber Baeumen.
 */

public class PostfixBaumBau {

  private static boolean is_operator(char c) {    // liefert true,
    return (c=='+' || c=='-' || c=='*' || c=='/');// falls c ein Operator ist
  }

  public static Baum postfixBaumBau (char[] ausdruck) { // konstruiert Baum

    Baum links, rechts;                           // Hilfsbaeume
    char c;                                       // aktuelles Zeichen

    Keller k = new Keller();                      // konstruiere einen Keller

    for (int i=0; i < ausdruck.length; i++) {     // durchlaufe Postfix-Ausdruck

        c = ausdruck[i];                          // aktuelles Zeichen

        if (!is_operator(c))                      // falls Operand,
            k.push(new Baum(new Character(c)));   // lege als Baum auf Keller
        else {                                    // falls Operator,
            rechts = (Baum)k.top(); k.pop();      // hole rechten Sohn
            links  = (Baum)k.top(); k.pop();      // hole linken Sohn
            k.push(new Baum(links,new Character(c),rechts)); // lege auf Keller
        }
    }
    return (Baum)k.top();                         // gib Baum im Keller zurueck
  }

  public static void main (String argv[]) {
    char [] zeile = IO.readChars("Bitte Postfix-Ausdruck: ");  // lies Postfix
    Baum wurzel = postfixBaumBau(zeile);          // konstruiere daraus Baum
    IO.print("Inorder lautet:         ");         // kuendige Traversierung an
    Traverse.klammerinorder(wurzel);              // gib in Klammer-Inorder aus
    IO.println();
  }
}
```

```
/****************************  PraefixBaumBau.java  ************************/

import AlgoTools.IO;

/** Klasse PraefixBaumBau enthaelt statische Methode praefixBaumBau,
 *  die einen Praefix-Ausdruck uebergeben bekommt
 *  und den zugehoerigen Baum zurueckliefert.
 */

public class PraefixBaumBau {

  private static boolean is_operator(char c) {    // liefert true,
    return (c=='+' || c=='-' || c=='*' || c=='/');// falls c ein Operator ist
  }

  static int i=0;                                 // Laufindex

  public static Baum praefixBaumBau (char[] ausdruck) {

    Baum links, rechts;                           // Hilfsbaeume
    char c;                                       // aktuelles Zeichen

    c = ausdruck[i++];                            // aktuelles Zeichen

    if (!is_operator(c))                          // falls Operand,
        return new Baum(new Character(c));        // liefere Operand als Baum
    else {                                        // falls Operator,
        links  = praefixBaumBau(ausdruck);        // baue linken Sohn;
        rechts = praefixBaumBau(ausdruck);        // baue rechten Sohn
        return (new Baum(links,new Character(c),rechts)); // verbinde
    }
  }

  public static void main (String argv[]) {
    char [] zeile = IO.readChars("Bitte Praefix-Ausdruck: ");  // lies Praefix
    Baum wurzel = praefixBaumBau(zeile);          // konstruiere daraus Baum
    IO.print("Inorder lautet:        ");          // kuendige Traversierung an
    Traverse.klammerinorder(wurzel);              // gib in Klammer-Inorder aus
    IO.println();
  }
}
```

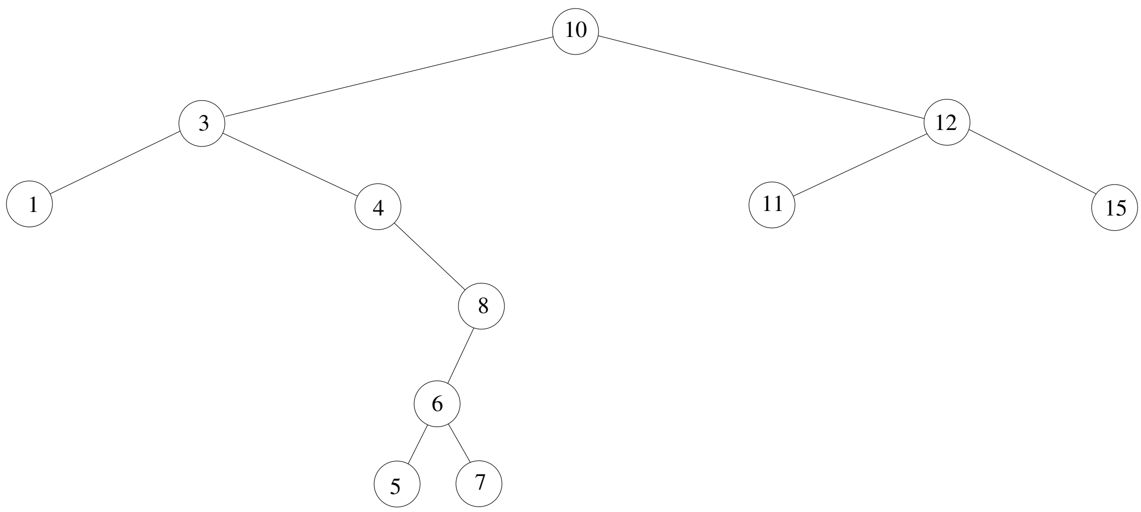
9.6 Suchbaum

Def.: Ein binärer *Suchbaum* ist ein binärer Baum, bei dem alle Einträge im linken Teilbaum eines Knotens x kleiner sind als der Eintrag im Knoten x und bei dem alle Einträge im rechten Teilbaum eines Knotens x größer sind als der Eintrag im Knoten x.

Der Aufwand aller Suchbaumoperationen ist proportional zur Anzahl der Knoten auf dem Wege von der Wurzel bis zu einem Blatt.

Best case:	Hat jeder Knoten 2 Söhne, so hat der Baum bei Höhe h $n = 2^h - 1$ Knoten. Die Anzahl der Weg-Knoten ist $h = \log(n)$.
Worst case:	Werden die Elemente sortiert eingegeben, so entartet der Baum zur Liste, der Aufwand beträgt dann $O(n)$.
Average case:	Für n zufällige Schlüssel beträgt der Aufwand $O(\log(n))$, genauer: Die Wege sind um 39 % länger als im best case.

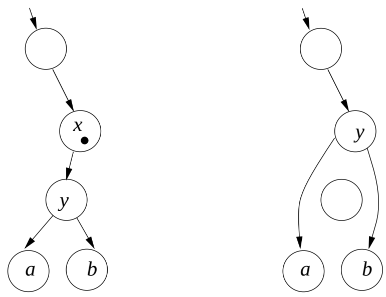
Beispiel für einen binären Suchbaum



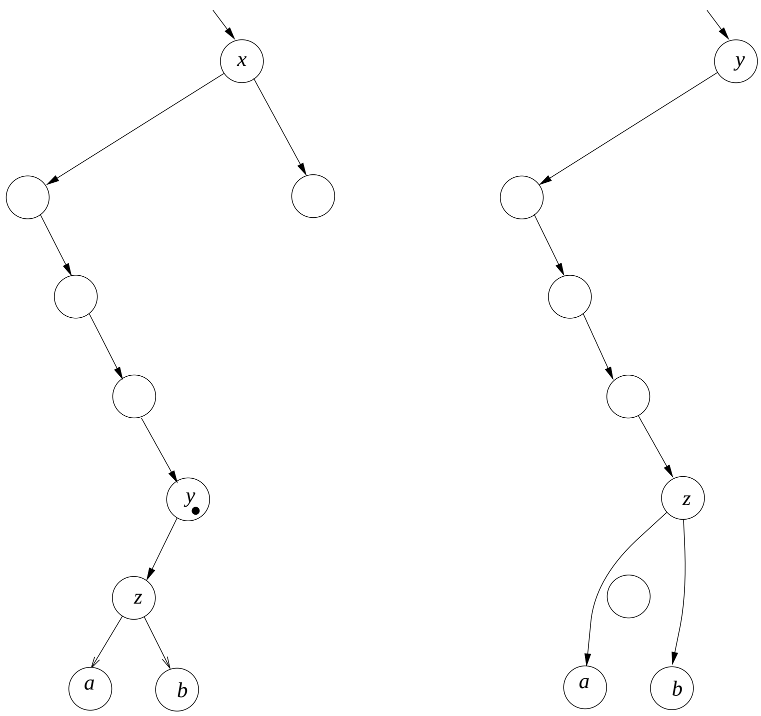
Sei x das Element in dem zu löschenden Knoten des Suchbaums.

Löschen eines Knotens ohne Söhne

Löschen eines Knotens mit einem Sohn



Löschen eines Knotens mit zwei Söhnen



Interfaces

Interfaces enthalten nur Methodenköpfe und Konstanten. Alle in einem Interface definierten Methoden sind implizit `abstract`. Ein Interface stellt eine Schnittstelle dar und legt damit die Funktionalität ihrer Methoden fest, ohne diese zu implementieren. Dies geschieht im Gegensatz zu einer abstrakten Klasse nicht in einer Subklasse, sondern in einer beliebigen Klasse, die dies zuerst in einer `implements`-Klausel deklariert und die dann eine Implementation aller Methoden des Interface bereitstellen muss. Verwendet werden kann ein Interface auch ohne Kenntnis der konkreten Implementation.

```
/***************************  Comparable.java  ******************************/

/**   Das Interface deklariert eine Methode, anhand der sich das
 *    aufgerufene Objekt mit dem uebergebenen vergleicht.
 *    Fuer jeden von Object abgeleiteten Datentyp muss eine solche
 *    Vergleichsklasse implementiert werden.
 *    Die Methode erzeugt eine Fehlermeldung, wenn a ein Objekt einer anderen
 *    Klasse als dieses Objekt ist.
 *
 *    int compareTo(Comparable a)
 *         liefert  0, wenn this == a
 *         liefert <0, wenn this <  a
 *         liefert >0, wenn this >  a
 */
public interface Comparable {

    public int compareTo(Comparable a);
}


/*************************  CharComparable.java  ***********************/

import AlgoTools.IO;

/**   Implementiert die Methode aus dem Interface Comparable
 *    in der fuer den einfachen Datentyp char spezifischen Weise.
 *
 *    int compareTo(Comparable a)
 *         liefert  0, wenn this == a
 *         liefert <0, wenn this <  a
 *         liefert >0, wenn this >  a
 */
public class CharComparable implements Comparable {

  private char c;

  public CharComparable(char c) { this.c = c; }
```

```
  /** vergleicht dies Objekt mit Character-Objekt a und liefert
   *  <0, =0 oder >0
   */
  public int compareTo(Comparable a) {
    if(!(a instanceof CharComparable))    // Ist a ein Objekt dieser Klasse?
      IO.error("CharComparable.compareTo:a("+a+") ist kein CharComparable!");

    // Character-Objekte nach char umwandeln und Differenz zurueckliefern
    return (c - ((CharComparable)a).c);
  }

  public String toString() { return new String("" + c); }
}


/*********************  StringComparable.java  ****************************/

import AlgoTools.IO;

/** Klasse StringComparable
 *    Implementiert Methode aus Compare und vergleicht damit dies Objekt mit
 *    dem uebergebenen auf die fuer Strings spezifische Weise.
 *
 *    int compareTo(Object a)
 *         liefert  0, wenn die Strings gleich sind.
 *         liefert <0, wenn dieser String lexikografisch kleiner String a ist
 *         liefert >0, andersrum
 */

public class StringComparable implements Comparable {

  private String s;

  public StringComparable(String s) { this.s = s; }

  /** vergleicht dies String-Objekt mit String-Objekt a und liefert
   *  <0, =0 oder >0
   */
  public int compareTo(Comparable a) {
    if(! (a instanceof StringComparable)) // Ist a ein Objekt dieser Klasse?
      IO.error("StringComparable.compare:a("+a+") ist kein StringComparable!");

    return s.compareTo(((StringComparable)a).s);
  }

  public String toString() { return s; }
}
```

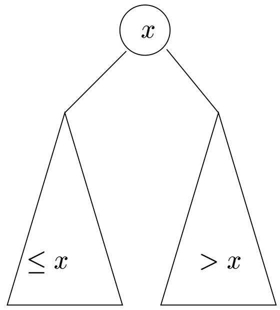
```
/***************************  SuchBaum.java  ******************************/

import AlgoTools.IO;

/** Implementation eines binaeren Suchbaums ueber Comparable-Objekten.
 */


public class SuchBaum extends Baum {


    // sucht x im SuchBaum: liefert den SuchBaum mit x in der Wurzel, ggf. leer
    private SuchBaum find(Comparable x) {
        if (empty())                                      return this;
        if(((Comparable)value()).compareTo(x) == 0)  return this;
        if(((Comparable)value()).compareTo(x) >  0)  return
           ((SuchBaum)left()).find(x);
        else                                              return
           ((SuchBaum)right()).find(x);
    }



    /** Sucht x im SuchBaum: liefert null, wenn x nicht gefunden wurde,
     * sonst Comparable-Objekt x */
    public Comparable lookup(Comparable x) {
        return (Comparable)find(x).inhalt;
    }



    /** fuegt x in SuchBaum ein: liefert true, wenn erfolgreich, sonst false.*/
    public boolean insert(Comparable x) {
        SuchBaum s = find(x);         // SuchBaum mit x in der Wurzel oder leer

        if (s.empty()) { // wenn leer, d.h. x noch nicht im SuchBaum enthalten:
            s.inhalt = x;                  // setzte Inhalt auf x
            s.links  = new SuchBaum();     // neuer leerer SuchBaum links
            s.rechts = new SuchBaum();     // neuer leerer SuchBaum rechts
            return true;
        }
        else return false;
    }
```

```
    /** loescht x aus SuchBaum: liefert true, wenn erfolgreich geloescht,
     *  sonst false */
    public boolean delete(Comparable x) {
        SuchBaum s = find(x);         // SuchBaum mit x in der Wurzel oder leer
        SuchBaum ersatz;              // Ersatzknoten

        if (s.empty()) return false;  // wenn x nicht gefunden: false
        else {                        // wenn x gefunden
            if      (s.left().empty())  ersatz = (SuchBaum)s.right();
            else if (s.right().empty()) ersatz = (SuchBaum)s.left();
            else {                    // Knoten mit x hat zwei Soehne
                ersatz = ((SuchBaum)s.left()).findMax(); // Maximum im linken
                s.inhalt = ersatz.inhalt;                // ersetze Inhalt
                s = ersatz;                              // zu ersetzen
                ersatz = (SuchBaum)ersatz.left();        // Ersatz: linker
            }
            s.inhalt = ersatz.inhalt; // ersetze die Komponenten
            s.links  = ersatz.links;
            s.rechts = ersatz.rechts;
            return true;
        }
    }

    // findet im nichtleeren SuchBaum das Maximum:
    // liefert den SuchBaum mit dem Maximum in der Wurzel
    private SuchBaum findMax() {
        SuchBaum hilf = this;

        while (!hilf.right().empty()) hilf = (SuchBaum)hilf.right();
        return hilf;      // der rechteste Nachfahr von this
    }
}
```

```
/****************************  SuchBaumTest.java  ***************************/
import AlgoTools.IO;

/** Testet den SuchBaum mit CharComparable-Objekten.
 */
public class SuchBaumTest {

    public static void main(String[] argv) {
        SuchBaum s = new SuchBaum();

        char[] eingabe = IO.readChars("Bitte Zeichen fuer insert: ");

        for (int i=0; i<eingabe.length; i++)   // Elemente in SuchBaum einfuegen
            if (s.insert(new CharComparable(eingabe[i])))
                IO.println(eingabe[i] + " eingefuegt");
            else
                IO.println(eingabe[i] + " konnte nicht eingefuegt werden");

        IO.print("Inorder: ");                 // Inorder-Traversierung
        Traverse.inorder(s); IO.println();

        eingabe = IO.readChars("Bitte Zeichen fuer lookup: ");

        for (int i=0; i<eingabe.length; i++) { // Elemente im SuchBaum suchen
            Comparable c = s.lookup(new CharComparable(eingabe[i]));
            if(c == null)
                IO.println(eingabe[i] + " konnte nicht gefunden werden");
            else
                IO.println(c + " gefunden");
        }
        eingabe = IO.readChars("Bitte Zeichen fuer delete: ");

        for (int i=0; i<eingabe.length; i++) { // Elemente im SuchBaum loeschen
            if (s.delete(new CharComparable(eingabe[i])))
                IO.println(eingabe[i] + " geloescht");
            else
                IO.println(eingabe[i] + " konnte nicht geloescht werden");

            IO.print("Inorder: ");             // Inorder-Traversierung
            Traverse.inorder(s); IO.println();
}
        IO.println("Versuche String 'Test' einzufuegen!");
        s.insert(new StringComparable("Test"));
        IO.println("Einfuegen erfolgreich!");
}   }
```

Speichern von Mehrfachexemplaren in einem Suchbaum

1. Möglichkeit: Elemente doppelt halten



2. Möglichkeit: Zähler im Knoten mitführen

Beim Einfügen: Zähler hochzählen, sofern Element schon vorhanden, sonst einfügen.

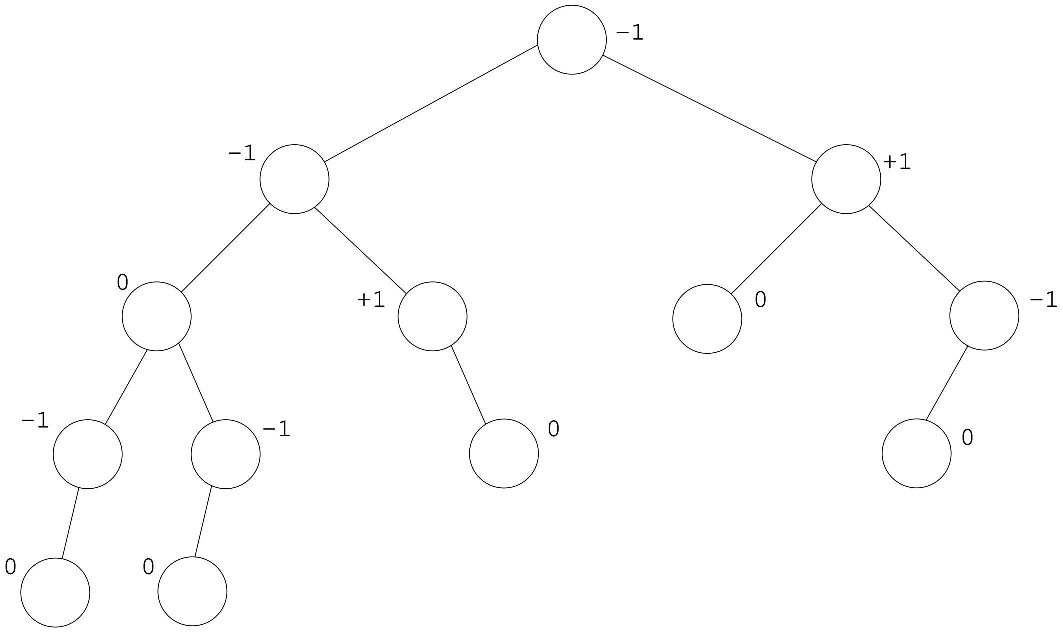
Beim Löschen: Zähler herunterzählen, sofern mehrfach da, sonst entfernen.

9.7 AVL-Baum

(benannt nach Adelson-Velskii und Landis, 1962)

Def.: Ein Knoten eines binären Baumes heißt *ausgeglichen* oder *balanciert*, wenn sich die Höhen seiner beiden Söhne um höchstens 1 unterscheiden.

Def.: Ein binärer Suchbaum, in dem jeder Knoten ausgeglichen ist, heißt *AVL-Baum*.



Sei $bal(x)$ = Höhe des rechten Teilbaums von x minus Höhe des linken Teilbaums von x.

Aufgrund der Ausgeglichenheit ist die Suche in einem AVL-Baum auch im ungünstigsten Fall von der Ordnung $O(\log n)$. Um das zu gewährleisten, muss nach jedem Einfügen oder Löschen die Ausgeglichenheit überprüft werden. Hierzu werden längs des Weges vom eingefügten bzw. gelöschten Element bis zur Wurzel die Balance-Werte abgefragt und durch so genannte *Rotationen* repariert.

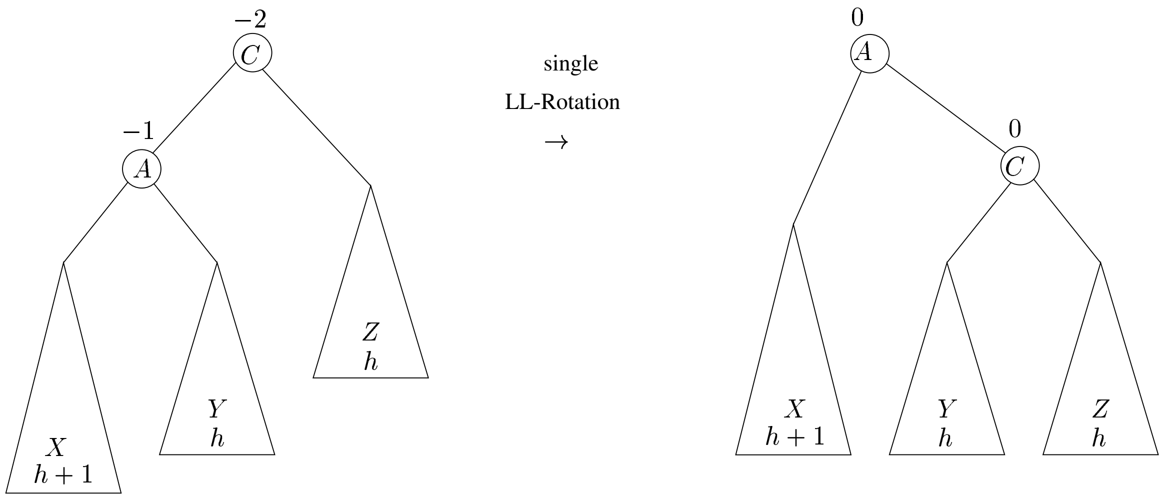
Während das Einfügen eines einzelnen Schlüssels *höchstens eine* Rotation erfordert, kann das Löschen eine Rotation für *jeden* Knoten entlang des Weges zur Wurzel verursachen.

Rotationen für AVL-Baum bei linksseitigem Übergewicht

Single LL-Rotation

Bei Einfügen in Teilbaum X: Höhe des gesamten Baums vorher und nachher gleich.

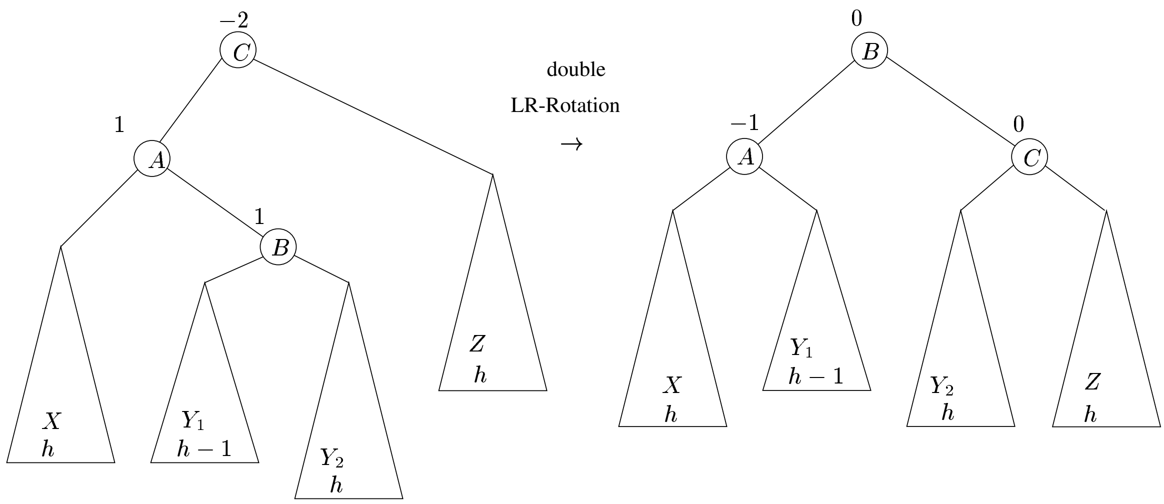
Bei Löschen im Teilbaum Z: Höhe des gesamten Baums vorher und nachher gleich oder nacher um eins kleiner.



Double LR-Rotation

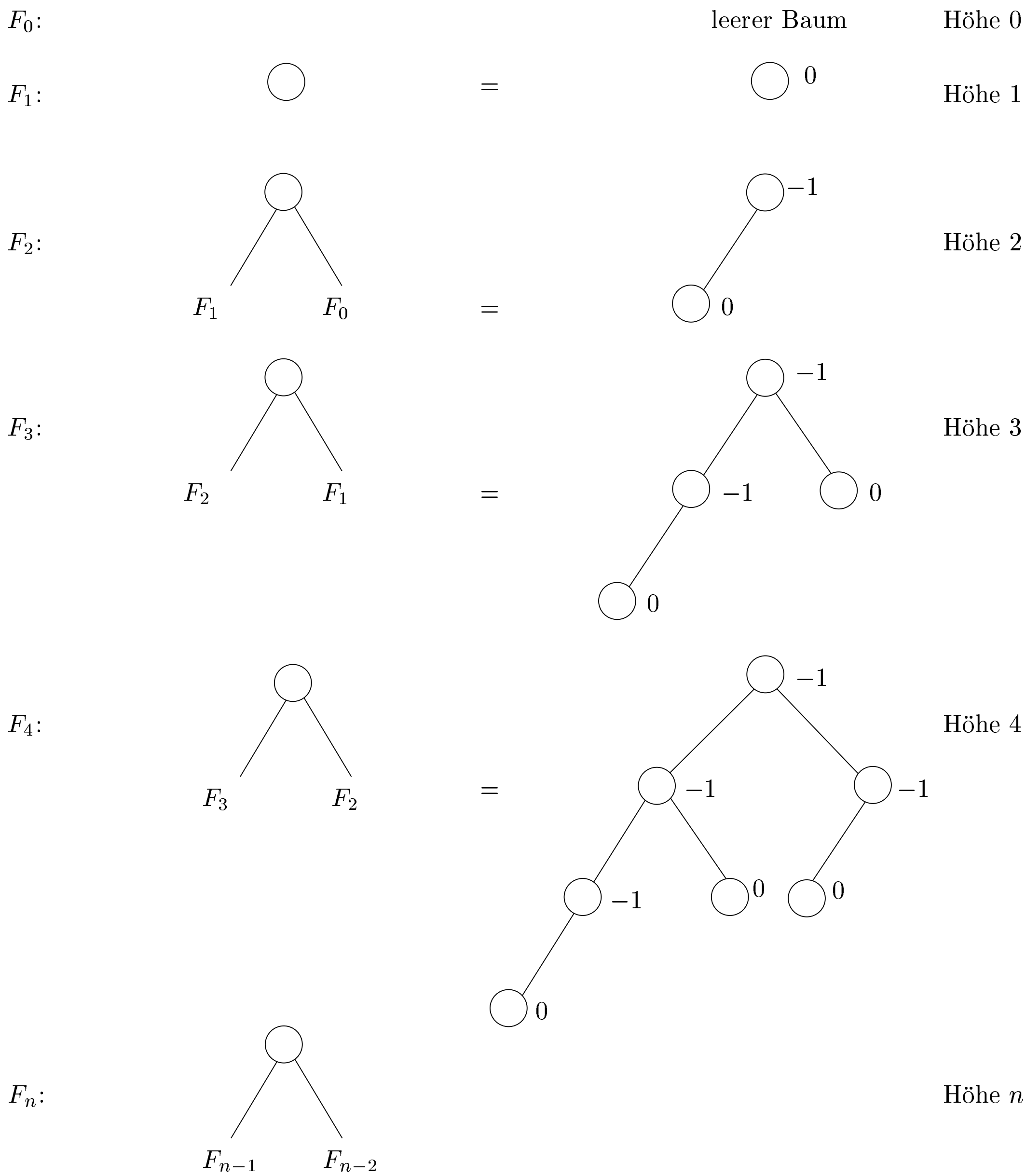
Bei Einfügen in Teilbaum Y_1 oder Y_2: Höhe des gesamten Baums vorher und nachher gleich.

Bei Löschen im Teilbaum Z: Höhe des gesamten Baums nachher um eins kleiner.



Bei rechtsseitigem Übergewicht werden die symmetrischen Rotationen "single RR" und "double RL" angewendet.

Minimal ausgeglichene AVL-Bäume sind *Fibonacci*-Bäume



Satz: Die Höhe eines AVL-Baumes mit n Knoten ist $\leq 1.45 \cdot \log n$, d.h. höchstens 45 % größer als erforderlich.

Bem.: Das Löschen eines Knotens in einem Fibonacci-AVL-Baum führt zu einer Reduktion der Höhe des Baumes. Das Löschen des Knotens mit dem größten Schlüssel erfordert die größtmögliche Zahl von Rotationen: Im Fibonacci-Baum F_n werden dabei $\lfloor \frac{n-1}{2} \rfloor$ LL-Rotationen durchgeführt.

```
/****************************  AVLBaum.java  ********************************/

import AlgoTools.IO;

/** Ein AVLBaum ist ein SuchBaum, bei dem alle Knoten ausgeglichen
 *  sind. Das heisst, die Hoehe aller Teilbaeume unterscheidet sich
 *  maximal um eins.
 */


public class AVLBaum extends SuchBaum {

    private int balance;                 // Balance

    public AVLBaum() {                   // erzeugt leeren AVLBaum

        balance = 0;
    }


    private static class Status {        // Innere Klasse zur Uebergabe eines
                                         // Status in der Rekursion
        boolean unbal;                   // unbal ist true, wenn beim Einfue-
                                         // gen ein Sohn groesser geworden ist
        Status () {                      // Konstruktor der inneren Klasse
            unbal = false;               // Element noch nicht eingefuegt =>
        }                                // noch keine Unausgeglichenheit
    }



    public String toString() {           // fuer Ausgabe: Inhalt(Balance)

        return new String(inhalt + "(" + balance + ")");
    }
```

```
public boolean insert(Comparable x) {// fuegt x in den AVLBaum ein: true,
                                     // wenn erfolgreich, sonst false.
                                     // Kapselt die Funktion insertAVL
    return insertAVL(x, new Status());
}


private boolean insertAVL(Comparable x, Status s) { // Tatsaechliche Methode zum
                                                    // Einfuegen (rekursiv)
    boolean eingefuegt;

    if(empty()) {                    // Blatt: Hier kann eingefuegt werden
        inhalt = x;                  // Inhalt setzen
        links  = new AVLBaum();      // Neuer leerer AVLBaum links
        rechts = new AVLBaum();      // Neuer leerer AVLBaum rechts
        s.unbal = true;              // Dieser Teilbaum wurde groesser
        return true;                 // Einfuegen erfolgreich und
    }                                // dieser Teilbaum groesser

    else if (((Comparable)value()).compareTo(x) == 0)   // Element schon im AVLBaum
        return false;

    else if (((Comparable)value()).compareTo(x) > 0){   // Element x ist kleiner =>
        eingefuegt = ((AVLBaum)left()).insertAVL(x,s);  // linker Teilbaum

        if(s.unbal) {                // Linker Teilbaum wurde groesser
            if (balance == 1) {      // Alte Unausgeglichenheit ausgegl.
                balance = 0;         // => neue Balance = 0
                s.unbal = false;     // Unausgeglichenheit ausgeglichen
                return true;
            }
            else if (balance == 0) { // Hier noch kein Rotieren noetig
                balance = -1;        // Balance wird angeglichen
                return true;
            }
            else {                   // Rotieren notwendig

                if (((AVLBaum)links).balance == -1)
                    rotateLL();
                else
                    rotateLR();
                s.unbal = false;     // Unausgeglichenheit ausgeglichen
                return true;         // => Rueckgabewert
            }                        // angleichen
        }
```

```
    } else {                            // Element ist groesser =>
        eingefuegt = ((AVLBaum)right()).insertAVL(x,s);// rechter Teilbaum

        if(s.unbal) {                   // Rechter Teilbaum wurde groesser
            if (balance == -1) {        // Alte Unausgeglichenheit ausgegl.
                balance = 0;            // => neue Balance = 0
                s.unbal = false;        // Unausgeglichenheit ausgeglichen
                return true;
            }
            else if (balance == 0) {    // Hier noch kein Rotieren noetig
                balance = 1;            // Balance wird angeglichen
                return true;
            }

            else {                      // Rotieren notwendig

                if (((AVLBaum)rechts).balance == 1)
                    rotateRR();
                else
                    rotateRL();
                s.unbal = false;        // Unausgeglichenheit ausgeglichen
                return true;            // => Rueckgabewert
            }                           // angleichen
        }
    }
    return eingefuegt;                  // Keine Rotation => Ergebnis zurueck
}
```

```
public void rotateLL() {

    IO.println("LL-Rotation im Teilbaum mit Wurzel "+ inhalt);

    AVLBaum a1 = (AVLBaum)links;     // Merke linken
    AVLBaum a2 = (AVLBaum)rechts;    // und rechten Teilbaum
                                     // Idee: Inhalt von a1 in die Wurzel
    links = a1.links;                // Setze neuen linken Sohn
    rechts = a1;                     // Setze neuen rechten Sohn
    a1.links = a1.rechts;            // Setze dessen linken
    a1.rechts = a2;                  // und rechten Sohn

    Object tmp = a1.inhalt;          // Inhalt von rechts (==a1)
    a1.inhalt = inhalt;              // wird mit Wurzel
    inhalt = tmp;                    // getauscht

    ((AVLBaum)rechts).balance = 0;   // rechter Teilbaum balanciert
    balance = 0;                     // Wurzel balanciert
}

public void rotateLR() {

    IO.println("LR-Rotation im Teilbaum mit Wurzel "+ inhalt);

    AVLBaum a1 = (AVLBaum)links;     // Merke linken
    AVLBaum a2 = (AVLBaum)a1.rechts; // und dessen rechten Teilbaum
                                     // Idee: Inhalt von a2 in die Wurzel
    a1.rechts = a2.links;            // Setze Soehne von a2
    a2.links = a2.rechts;
    a2.rechts = rechts;
    rechts = a2;                     // a2 wird neuer rechter Sohn

    Object tmp = inhalt;             // Inhalt von rechts (==a2)
    inhalt = rechts.inhalt;          // wird mit Wurzel
    rechts.inhalt = tmp;             // getauscht

    if (a2.balance == 1)             // Neue Bal. fuer linken Sohn
        ((AVLBaum)links).balance = -1;
    else
        ((AVLBaum)links).balance = 0;

    if (a2.balance == -1)            // Neue Bal. fuer rechten Sohn
        ((AVLBaum)rechts).balance = 1;
    else
        ((AVLBaum)rechts).balance = 0;
    balance = 0;                     // Wurzel balanciert
}
```

```
public void rotateRR() {

    IO.println("RR-Rotation im Teilbaum mit Wurzel "+ inhalt);

    AVLBaum a1 = (AVLBaum)rechts;    // Merke rechten
    AVLBaum a2 = (AVLBaum)links;     // und linken Teilbaum
                                     // Idee: Inhalt von a1 in die Wurzel
    rechts = a1.rechts;              // Setze neuen rechten Sohn
    links = a1;                      // Setze neuen linken Sohn
    a1.rechts = a1.links;            // Setze dessen rechten
    a1.links  = a2;                  // und linken Sohn

    Object tmp = a1.inhalt;          // Inhalt von links (==a1)
    a1.inhalt = inhalt;              // wird mit Wurzel
    inhalt = tmp;                    // getauscht

    ((AVLBaum)links).balance = 0;    // linker Teilbaum balanciert
    balance = 0;                     // Wurzel balanciert
}

public void rotateRL() {

    IO.println("RL-Rotation im Teilbaum mit Wurzel "+ inhalt);

    AVLBaum a1 = (AVLBaum)rechts;    // Merke rechten Sohn
    AVLBaum a2 = (AVLBaum)a1.links;  // und dessen linken Teilbaum
                                     // Idee: Inhalt von a2 in die Wurzel
    a1.links = a2.rechts;
    a2.rechts = a2.links;            // Setze Soehne von a2
    a2.links = links;
    links = a2;                      // a2 wird neuer linker Sohn

    Object tmp = inhalt;             // Inhalt von links (==a2)
    inhalt = links.inhalt;           // wird mit Wurzel
    links.inhalt = tmp;              // getauscht

    if (a2.balance == -1)            // Neue Bal. fuer rechten Sohn
        ((AVLBaum)rechts).balance = 1;
    else
        ((AVLBaum)rechts).balance = 0;

    if (a2.balance == 1)             // Neue Bal. fuer linken Sohn
        ((AVLBaum)links).balance = -1;
    else
        ((AVLBaum)links).balance = 0;
    balance = 0;                     // Wurzel balanciert
}
```

```
public boolean delete(Comparable x) {// loescht x im AVLBaum: true,
                                     // wenn erfolgreich, sonst false.
                                     // Kapselt die Funktion deleteAVL
    return deleteAVL(x, new Status());
}

private boolean deleteAVL(Comparable x, Status s) { // Tatsaechliche Methode
                         // zum Loeschen (rekursiv); true, wenn erfolgreich
    boolean geloescht;               // true, wenn geloescht wurde

    if(empty()) {                    // Blatt: Element nicht gefunden
        return false;                // => Einfuegen erfolglos
    }

    else if (((Comparable)value()).compareTo(x) < 0){     // Element x ist groesser =>
                                               // Suche rechts weiter
        geloescht = ((AVLBaum)rechts).deleteAVL(x,s);
        if (s.unbal == true) balance2(s);      // Gleiche ggf. aus
        return geloescht;
    }
    else if (((Comparable)value()).compareTo(x) > 0){     // Element x ist kleiner =>
                                               // Suche links weiter
        geloescht = ((AVLBaum)links).deleteAVL(x,s);
        if (s.unbal == true) balance1(s);      // Gleiche ggf. aus
        return geloescht;
    }
    else {                           // Element gefunden
        if (rechts.empty()) {        // Kein rechter Sohn
            inhalt = links.inhalt;   // ersetze Knoten durch linken Sohn
            links = links.links;     // Kein linker Sohn mehr
            balance = 0;             // Knoten ist Blatt
            s.unbal = true;          // Hoehe hat sich geaendert
        } else if (links.empty()) {  // Kein linker Sohn
            inhalt = rechts.inhalt;  // ersetze Knoten durch rechten Sohn
            rechts = rechts.rechts;  // Kein rechter Sohn mehr
            balance = 0;             // Knoten ist Blatt
            s.unbal = true;          // Hoehe hat sich geaendert
        } else {                     // Beide Soehne vorhanden
            inhalt = ((AVLBaum)links).del(s);   // Rufe del() auf
            if (s.unbal) {           // Gleiche Unbalance aus
                balance1(s);
            }
        }
        return true;                 // Loeschen erfolgreich
    }
}
```

```
private Object del(Status s) {        // Sucht Ersatz fuer gel. Objekt
    Object ersatz;                    // Das Ersatz-Objekt
    if (!rechts.empty()) {            // Suche groessten Sohn im Teilbaum
        ersatz = ((AVLBaum)rechts).del(s);
        if (s.unbal)                  // Gleicht ggf. Unbalance aus
            balance2(s);
    } else {                          // Tausche mit geloeschtem Knoten
        ersatz = inhalt;              // Merke Ersatz und
        inhalt = links.inhalt;        // ersetze Knoten durch linken Sohn.
        links = links.links;          // Kein linker Sohn mehr
        balance = 0;                  // Knoten ist Blatt
        s.unbal = true;               // Teilbaum wurde kuerzer
    }
    return ersatz;                    // Gib Ersatz-Objekt zurueck
}
```

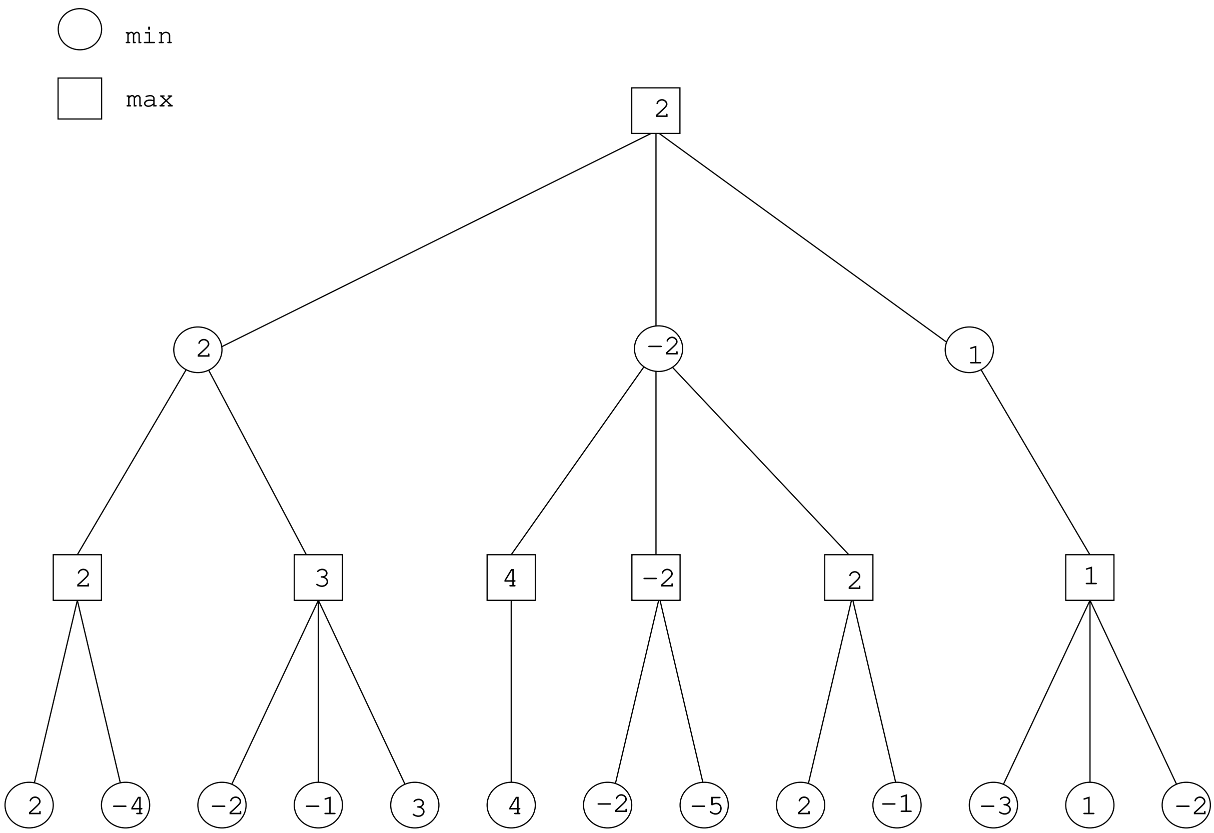
```
    private void balance1(Status s) {    // Unbalance, weil linker Ast kuerzer
        if (balance == -1)
            balance = 0;                 // Balance geaendert, nicht ausgegl.
        else if (balance == 0) {
            balance = 1;                 // Ausgeglichen
            s.unbal = false;
        } else {                         // Ausgleichen (Rotation) notwendig
            int b = ((AVLBaum)rechts).balance; //Merke Balance des rechten Sohns
            if (b >= 0) {
                rotateRR();
                if (b == 0) {            // Gleiche neue Balancen an
                    balance = -1;
                    ((AVLBaum)links).balance = 1;
                    s.unbal = false;
                }
            } else
                rotateRL();
        }
    }


    private void balance2(Status s) {    // Unbalance, weil recht. Ast kuerzer
        if (balance == 1)
            balance = 0;                 // Balance geaendert, nicht ausgegl.
        else if (balance == 0) {
            balance = -1;                // Ausgeglichen
            s.unbal = false;
        } else {                         // Ausgleichen (Rotation) notwendig
            int b = ((AVLBaum)links).balance; // Merke Balance des linken Sohns
            if (b <= 0) {
                rotateLL();
                if (b == 0) {            // Gleiche neue Balancen an
                    balance = 1;
                    ((AVLBaum)rechts).balance = -1;
                    s.unbal = false;
                }
            } else
                rotateLR();
        }
    }
}
```

```
/*****************************  AVLBaumTest.java  ****************************/

import AlgoTools.IO;

/** Klasse zum Testen des AVLBaums: Einfuegen und Loeschen von Character      */

public class AVLBaumTest {

  public static void main(String[] argv) {

    AVLBaum b = new AVLBaum();
    char k = IO.readChar("Char in AVL-Baum einfuegen (Loeschen: \\n): ");
    while (k != '\n') {
      if (b.insert(new CharComparable(k))) IO.println(k + " eingefuegt");
      else                                 IO.println(k + " nicht eingefuegt");
      IO.println("AVL-Baum mit Balancen:");
      printAVLBaum(b, 0);
      k = IO.readChar("Char in AVL-Baum einfuegen (Loeschen: \\n): ");
    }
    IO.println();
    k = IO.readChar("Char im AVL-Baum loeschen (Abbruch: \\n): ");

    while (k != '\n') {
      if (b.delete(new CharComparable(k))) IO.println(k + " geloescht");
      else                           IO.println(k + " nicht geloescht");
      IO.println("AVL-Baum mit Balancen:");
      printAVLBaum(b, 0);
      k = IO.readChar("Char im AVL-Baum loeschen (Abbruch: \\n): ");
    }

    IO.println("Versuche String 'Test' einzufuegen!");
    b.insert(new StringComparable("Test"));
    IO.println("Einfuegen erfolgreich!");
    printAVLBaum(b, 0);
  }

  /** Der AVL-Baum wird liegend mit Werten und Balancen ausgegeben.  */
  public static void printAVLBaum(Baum b, int tiefe) {
    if (! b.empty()) {                     // Wenn Baum nicht leer:
      printAVLBaum(b.right(), tiefe+1);    // rechten Teilbaum ausgeben
      for (int i=0; i<tiefe; i++)          // entsprechend der Rekursions-
        IO.print("    ");                  // tiefe einruecken
      IO.println((AVLBaum)b);              // Wurzel und Balance ausgeben
      printAVLBaum(b.left(), tiefe+1);     // linken Teilbaum ausgeben
    }
  }
}
```

9.8 Spielbaum

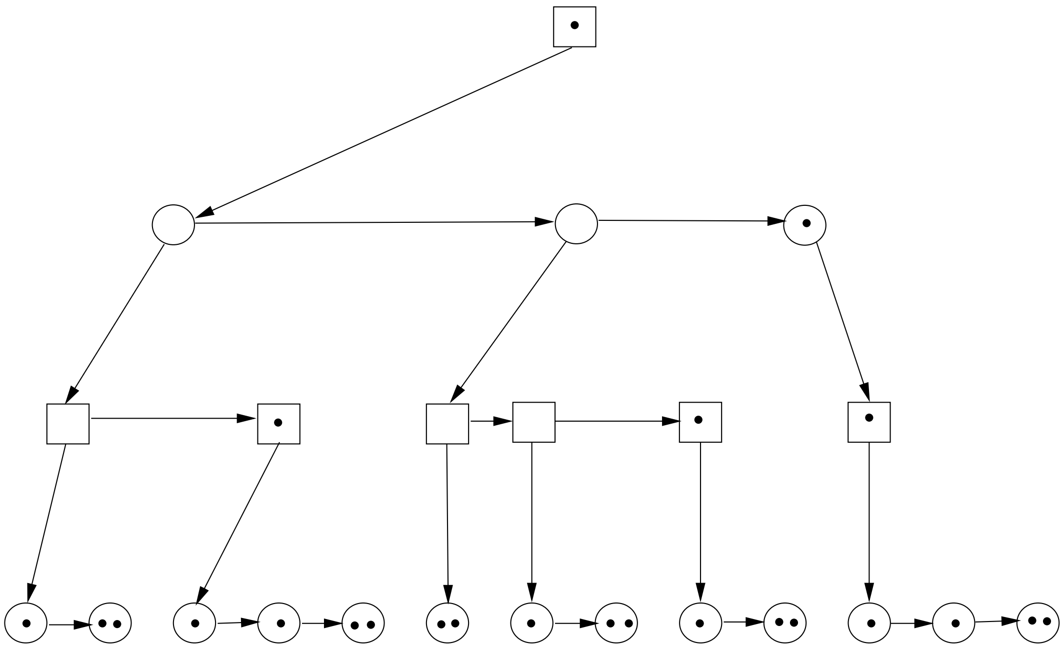
Def.: Ein Spielbaum ist ein Baum mit zwei Typen von Knoten: Minimum-Knoten und Maximum-Knoten.
Die Knoten repräsentieren Spielstellungen.
Der Wert eines Blattes wird bestimmt durch eine statische Stellungsbewertung.
Der Wert eines Minimum-Knotens ist das Minimum der Werte seiner Söhne.
Der Wert eines Maximum-Knotens ist das Maximum der Werte seiner Söhne.
Obacht: Bei Höhe h und Verzweigungsgrad d gibt es d^{h-1} Blätter.
Z.B. 8 Halbzüge mit je 20 Alternativen $\Rightarrow$ 25.600.000.000 Blätter.

Beispiel für einen Spielbaum



Implementation eines nicht-binären Baums

Jeder Knoten hat einen Verweis auf den ältesten Sohn und den nächstjüngeren Bruder.



```java
/***************************  SpielBaum.java  *********************************/

/** Klasse SpielBaum mit der geschachtelten Klasse Stellung
 *  und den Datenfeldern typ, stellung, first, next
 *  und der Methode minmax.
 */

public class SpielBaum {

  class Stellung {}           // Platzhalter fuer Definition der Klasse Stellung
  Stellung stellung;          // Datenfeld Spielstellung
  boolean  maxtyp;            // true, falls Max-Knoten; false, falls Min-Knoten
  SpielBaum first;            // Verweis auf aeltesten Sohn
  SpielBaum next;             // Verweis auf naechstjuengeren Bruder

  public int statisch(){      // Platzhalter fuer Algorithmus zur Bewertung
    return 0;                 // des Knotens aufgrund des Datenfelds stellung
  }

  public int minmax () {                          // wertet Spielbaum aus
    SpielBaum bruder;                             // Hilfsverweis
    int bruderwert, best;                         // Hilfsvariable

    if (first == null)                            // falls Blatt,
        return statisch();                        // werte statisch aus
    else {                                        // falls kein Blatt,
        if (maxtyp) best = Integer.MIN_VALUE;     // je nach Knotentyp
               else best = Integer.MAX_VALUE;     // setze initialen Wert fest

        bruder = first;                           // beginne bei aeltestem Sohn,
        while (bruder != null) {                  // solange es Soehne gibt,
          bruderwert = bruder.minmax();           // bestimme Wert des Knotens
          if ((( maxtyp) && (bruderwert>best))||  // falls Verbesserung
              ((!maxtyp) && (bruderwert<best)))   // beobachtet wurde,
              best = bruderwert;                  // merke Verbesserung
          bruder = bruder.next;                   // gehe zum naechsten Bruder
        }
    }
    return best;                                  // liefere Wert zurueck
  }
}
```

Kapitel 10

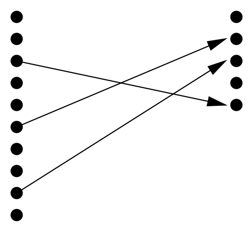
Hashing

Zum Abspeichern und Wiederfinden von Elementen wäre folgende Funktion hilfreich:

```
f: Element -> int
```

Dann könnte Element x bei Adresse $f(x)$ gespeichert werden.

Problem: Anzahl der möglichen Elemente $\gg$ Anzahl der Adressen



Gegeben Adressen von 0 bis $N-1$.
Sei `x` ein beliebiges Objekt. Dann ist

```
String s = x.toString();
```

seine Stringrepräsentation.
Sei $x = x_{n-1}x_{n-2}\ldots x_1x_0$ ein String, dann ist

$$f(x) = \left(\sum_{i=0}^{n-1} x_i\right) \text{ MOD } N$$

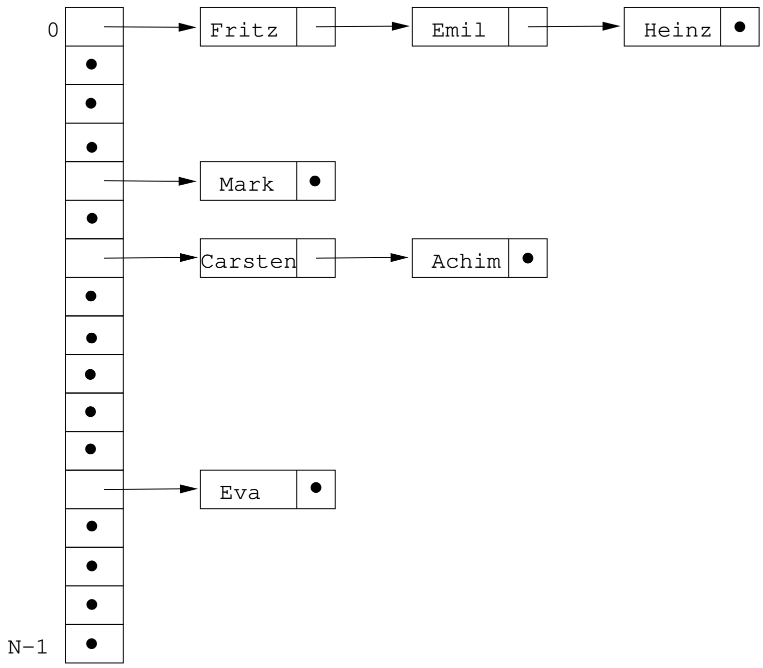
eine *Hashfunktion*.
Gilt: $f(x) = f(y)$, so liegt eine *Kollision* vor, die bei *offenem* und *geschlossenem* Hashing unterschiedlich behandelt wird.

10.1 Offenes Hashing

```
private Liste[] b; // Array von Buckets
                   // Jedes Bucket enthaelt Liste von Comparables
```

Alle Elemente x mit $f(x) = i$ befinden sich in der Liste b$[i]$. Bei N Buckets und n Elementen enthält jede Liste im Mittel $\frac{n}{N}$ Elemente.

Implementation des offenen Hashings



10.2 Geschlossenes Hashing

```
private Comparable[] inhalt; // Array von Comparables
private byte[] zustand;       // Array von Zustaenden
                              // LEER, BELEGT, GELOESCHT
```

Falls $y = f(x)$ schon belegt ist, so suche für x einen Alternativplatz.

$y+1, y+2, y+3, y+4, \ldots$	*lineares Sondieren*
$y+1, y+4, y+9, y+16, \ldots$	*quadratisches Sondieren*
$y + f_2(x)$	*Double Hashing* (Schrittweite wird durch 2. Hashfunktion bestimmt)

Beim linearen und quadratischen Sondieren müssen höchstens $N-1$ Sondierschritte durchgeführt werden. Beim quadratischen Sondieren werden ggf. nicht alle Buckets besucht, aber mindestens $\frac{N}{2}$.

Implementation des geschlossenen Hashings

0	B	Fritz
	B	Emil
	L	
	L	
	B	Mark
	L	
	B	Carsten
	G	
	L	
	B	Heinz
	G	
	L	
	B	Eva
	L	
	L	
	B	Achim
N-1	L	

```
L = LEER
B = BELEGT
G = GELOESCHT
```

Beispiel:

Die beiden Abbildungen ergeben sich durch sukzessives Einfügen der Worte

`Fritz`, `Mark`, `Emil`, `Carsten`, `Ulf`, `Heinz`, `Lutz`, `Eva`, `Achim`

und anschließendes Löschen von

`Ulf`, `Lutz`

für $N = 17$.

Perfekte Hashfunktion

Gesucht wird eine Hashfunktion f, die auf den Elementen keine Kollision verursacht, z.B.:

$$\begin{array}{rcccccccc} \text{gesucht: } f & : & \{\text{braun}, & \text{rot}, & \text{blau}, & \text{violett}, & \text{türkis}\} & \to & \mathbb{N} \\ \text{Länge}(w) & = & 5 & 3 & 4 & 7 & 6 & & \\ \text{Länge}(w) - 3 & = & 2 & 0 & 1 & 4 & 3 & & \end{array}$$

$\Rightarrow f(w) = \text{Länge}(w) - 3 \in [0..4]$ ist perfekte Hashfunktion.

```
/****************************  Hashing.java  ********************************/

/** Das Interface deklariert die Methoden zum Hashing.                        */

public interface Hashing {

  public String toString();                  // liefert den Aufbau der Tabelle
  public Comparable lookup(Comparable x);  // versucht, x nachzuschlagen
  public boolean insert(Comparable x);     // versucht, x einzufuegen
  public boolean delete(Comparable x);     // versucht, x zu loeschen
}


/****************************  OfHashing.java  ******************************/

/** Implementation des Interface Hashing durch Array von Listen              */

public class OfHashing implements Hashing {

  private Liste[] b ;                         // Array fuer Listen

  public OfHashing(int N) {                   // Konstruktor fuer Hashtabelle
    b  = new Liste[N];                        // besorge Platz fuer N Listen
    for (int i=0; i<N; i++)                   // konstruiere pro Index
        b[i]=new Liste();                     // eine leere Liste
  }

  private int hash(Comparable x) {            // berechne Hash-Wert fuer x
    int i, summe = 0;                         // Hilfsvariablen
    String s = x.toString();                  // berechne Stringdarstellung von x
    for (i=0; i<s.length(); i++)              // durchlaufe den String
        summe += s.charAt(i);                 // und addiere alle Zeichen
    return summe % b.length;                  // liefere summe mod Array-Laenge
  }

  public String toString() {                  // liefert den Aufbau der Tabelle
    String s = new String();                  // als String zurueck
    for (int i=0; i< b.length; i++){          // durchlaufe Array
      s += i + " :";                          // notiere Index
      b[i].reset();                           // gehe an Anfang der Liste
      while (!b[i].endpos()) {                // solange noch nicht am Ende
         s += " " + b[i].elem();              // notiere aktuelles Objekt
         b[i].advance();                      // schreite in Liste voran
      }
      s += "\n";                              // notiere Zeilenvorschub
    } return s;                               // liefere Stringdarstellung
  }
```

```
  private boolean find(Liste l, Comparable x) { // sucht x in Liste
    l.reset();                                  // gehe an den Anfang der Liste
    while (!l.endpos() &&                       // solange noch nicht am Ende
          ((Comparable)l.elem()).compareTo(x) != 0) // und Objekt noch nicht gefunden
      l.advance();                              // schreite voran
    return !l.endpos();                         // liefert true, falls gefunden
  }                                             // sonst false

  // Unter Verwendung der Hashfunktion hash und der Hilfsmethode find
  // werden die oeffentlichen Methoden implementiert


  public Comparable lookup(Comparable x) {  // versucht x nachzuschlagen
    int index = hash(x);                        // berechne Hash-Wert
    if (find(b[index],x))                       // falls in Liste gefunden,
        return (Comparable)b[index].elem(); // liefere Verweis auf Objekt
    else return null;                           // liefere Fehlanzeige
  }


  public boolean insert(Comparable x) {       // versucht x einzufuegen
    int index = hash(x);                        // berechne Hash-Wert
    if (!find(b[index],x)){                     // falls nicht in Liste gefunden,
        b[index].insert(x);                     // fuege in Liste ein
        return true;                            // melde Erfolg
    } else return false;                        // melde Misserfolg
  }


  public boolean delete(Comparable x) {       // versucht x zu loeschen
    int index = hash(x);                        // berechne Hash-Wert
    if (find(b[index],x)){                      // falls in Liste gefunden,
        b[index].delete();                      // entferne aus Liste
        return true;                            // melde Erfolg
    } else return false;                        // melde Misserfolg
  }
}
```

```
/****************************  GeHashing.java  ******************************/

/** Implementation des Interface Hashing
 *  durch ein geschlossenes Hashing mit einem Array von Objekten.
 */

public class GeHashing implements Hashing {

  private final static byte LEER      = 0;  // noch nie belegt, jetzt frei
  private final static byte BELEGT    = 1;  // zur Zeit belegt
  private final static byte GELOESCHT = 2;  // war schon mal belegt, jetzt frei

  private Comparable[] inhalt;              // Array fuer Elemente
  private byte[]       zustand;             // Array fuer Zustaende

  public GeHashing(int N) {                 // Konstruktor fuer Hashtabelle
    inhalt  = new Comparable[N];            // besorge Platz fuer N Objekte
    zustand = new byte[N];                  // besorge Platz fuer N Zustaende
    for (int i=0; i<N; i++)                 // setze alle Zustaende
        zustand[i]=LEER;                    // auf LEER
  }

  private int hash(Comparable x) {          // berechne Hash-Wert fuer x
    int i, summe = 0;                       // Hilfsvariablen
    String s = x.toString();                // berechne Stringdarstellung von x
    for (i=0; i<s.length(); i++)            // durchlaufe den String
        summe += s.charAt(i);               // und addiere alle Zeichen
    return summe % inhalt.length;           // liefere summe mod Tabellenlaenge
  }

  public String toString() {                // liefert den Aufbau der Tabelle
    String s = new String();                // als String zurueck
    for (int i=0; i< inhalt.length; i++){   // durchlaufe Tabelle
      s += i;                               // notiere Index
      switch(zustand[i]) {                  // je nach eingetragenem Zustand
        case LEER:      s+="  L  "; break;  // notiere L
        case BELEGT:    s+="  B  "; break;  // notiere B
        case GELOESCHT: s+="  G  "; break;  // notiere G
      }
      if (zustand[i]==BELEGT) s+=inhalt[i]; // notiere Objektdarstellung
      s += '\n';                            // notiere Zeilenvorschub
    }
    return s;                               // liefere Stringdarstellung
  }
```

```
// Unter Verwendung der Hashfunktion hash wird eine Hilfsmethode
// implementiert, welche das Objekt in der Tabelle sucht und durch
// - einen positiven Rueckgabewert den Index angibt, wo es gefunden wurde
// - einen negativen Rueckgabewert den bitweise negierten Index angibt,
//   wo es eingefuegt werden koennte (d.h., es wurde nicht gefunden).
// Implementiert ist ein quadratisches Sondieren: Die Sondierschrittweite d
// wird mit eins initialisiert und in jedem Sondierschritt um zwei erhoeht.

private int find(Comparable x) {          // sucht x in der Hashtabelle
                                          // liefert Position, falls gefunden
                                          // oder negierte Pos. zum Einfuegen
  int versuche         = 0;               // Zahl der Sondierungen
  int d                = 1;               // Sondierschrittweite
  boolean loch_gefunden = false;          // true, falls Loch gefunden
  int lochindex        = 0;               // Vorschlag zum spaeteren Einfuegen
  int index            = hash(x);         // berechne Hash-Wert

  while (!(((zustand[index] == BELEGT)    // Abbruch, falls
   && (inhalt[index].compareTo(x) == 0))  // Element gefunden
   || (zustand[index] == LEER)            // oder leeres Feld gefunden
   || (versuche == inhalt.length))) {     // oder Zahl der Versuche zu gross

                                          // nur wegen spaeterem insert:
      if ((!loch_gefunden) &&             // falls noch kein Loch gefunden
          (zustand[index] == GELOESCHT)){ // und GELOESCHT vorliegt,
          loch_gefunden = true;           // merke, dass Loch gefunden
          lochindex     = index;          // und seine Position
      }

     index = (index + d) % inhalt.length; // mache einen Sondierschritt
     d    += 2;                           // erhoehe Schrittweite
     versuche++;                          // erhoehe Zahl der Versuche
  }

  if ((zustand[index]==BELEGT) &&         // falls belegtes Feld
      (inhalt[index].compareTo(x) == 0))  // und x gefunden
      return index;                       // liefere Position zurueck
  if (loch_gefunden) return ~lochindex;   // liefere Loch (bitweise negiert)
               else return ~index;        // sonst letzte Einfuegeposition
}                                         // als negierte Zahl zurueck
```

```
  // Unter Verwendung der Hilfsmethode find
  // werden die oeffentlichen Methoden lookup, insert, delete implementiert


  public Comparable lookup(Comparable x) {  // versucht, x nachzuschlagen
    int pos = find(x);                      // versuche, x zu finden
    if (pos >= 0) return inhalt[pos];       // falls gefunden: liefere Objekt,
           else return null;                // sonst melde Misserfolg
  }


  public boolean insert(Comparable x) {     // versucht, x einzufuegen
    int pos = find(x);                      // versuche, x zu finden
    if ((pos < 0) &&                        // falls x nicht gefunden
        (zustand[~pos]!=BELEGT)) {          // und falls Platz vorhanden:
       inhalt[~pos] = x;                    // fuege x ein
       zustand[~pos]= BELEGT;               // setze Zustand auf BELEGT
       return true;                         // melde Erfolg
    } else return false;                    // melde Misserfolg
  }


  public boolean delete(Comparable x) {     // versucht, x zu loeschen
    int pos = find(x);                      // versuche, x zu finden
    if (pos >= 0) {                         // falls x gefunden:
        zustand[pos] = GELOESCHT;           // setze Zustand auf GELOESCHT
        inhalt[pos]  = null;                // gib Verweis auf Objekt frei
        return true;                        // melde Erfolg
    } else return false;                    // melde Misserfolg
  }
}
```

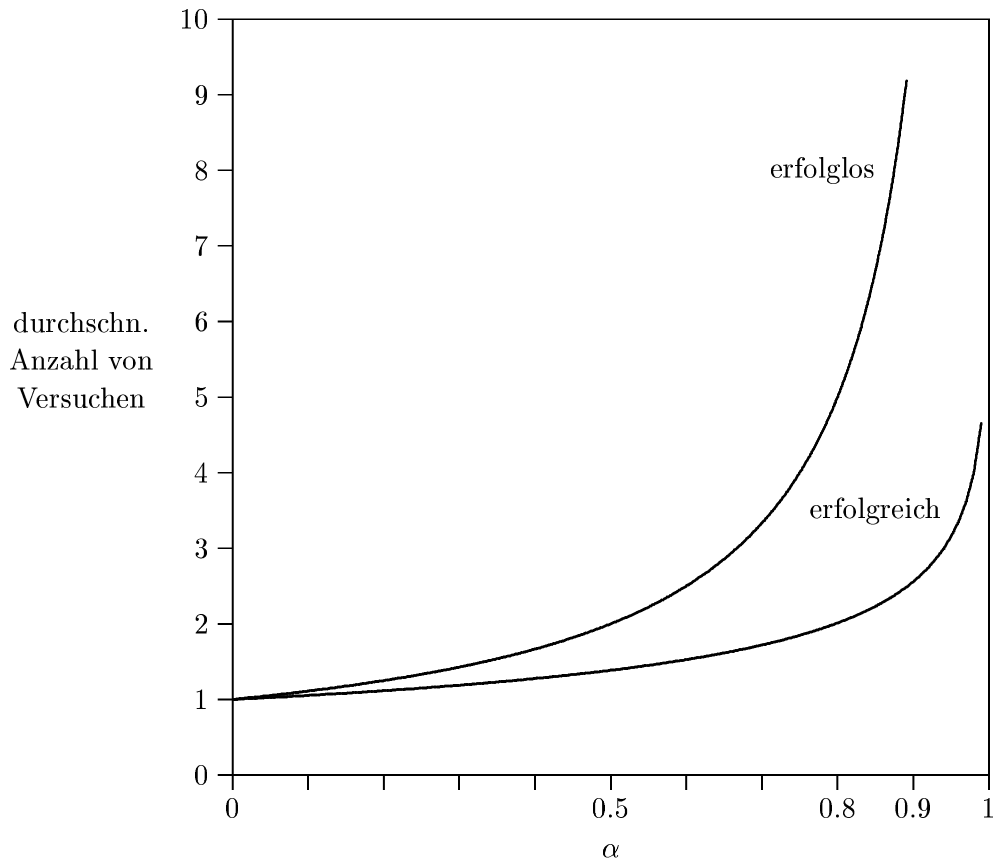
```
/*****************************  HashTest.java  *******************************/

import AlgoTools.IO;

/** Testet die Hash-Tabelle mit String-Objekten. Verwendet werden:
 *  Objekte der Klasse StringComparable als Eintraege in der HashTable
 *  und ein Objekt der Klasse GeHashing als Instanz des Interface Hashing
 */
public class HashTest {

    public static void main(String[] argv) {

        int groesse = IO.readInt("Bitte Groesse der Tabelle angeben: ");
        GeHashing h = new GeHashing(groesse);
        String s;

        IO.println("Gelegenheit fuer INSERT:");
        s = IO.readString("Bitte String (RETURN beendet): ");
        while (s.length()>0) {
            if (h.insert(new StringComparable(s)))
                 IO.println(s + " eingefuegt");
            else IO.println(s + " konnte nicht eingefuegt werden");
            IO.print(h);
            s = IO.readString("Bitte String: ");
        }

        IO.println("Gelegenheit fuer LOOKUP:");
        s = IO.readString("Bitte String (RETURN beendet): ");
        while (s.length()>0) {
            Comparable c = h.lookup(new StringComparable(s));
            if (c != null)  IO.println(c + " wurde gefunden");
            else            IO.println(s + " wurde nicht gefunden");
            IO.print(h);
            s = IO.readString("Bitte String: ");
        }

        IO.println("Gelegenheit fuer DELETE:");
        s = IO.readString("Bitte String (RETURN beendet): ");
        while (s.length()>0) {
            if (h.delete(new StringComparable(s)))
                 IO.println(s + " wurde geloescht");
            else IO.println(s + " wurde nicht geloescht");
            IO.print(h);
            s = IO.readString("Bitte String: ");
        }
    }
}
```

Laufzeit bei geschlossenem Hashing

Sei $\alpha \leq 1$ der Auslastungsfaktor. Dann ergibt sich für die Anzahl der Schritte mit Double-Hashing als Kollisionsstrategie bei

- erfolgloser Suche: $\approx \frac{1}{1-\alpha} = 5.0$, für $\alpha = 0.8$
- erfolgreicher Suche: $\approx -\frac{ln(1-\alpha)}{\alpha} = 2.01$, für $\alpha = 0.8$

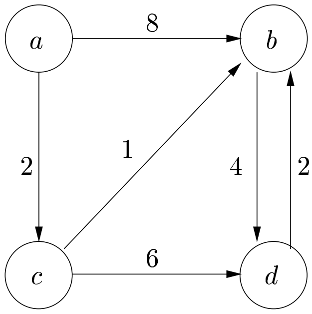
d.h., in 2 Schritten wird von 800.000 Elementen aus einer 1.000.000 großen Tabelle das richtige gefunden.



Kapitel 11

Graphen

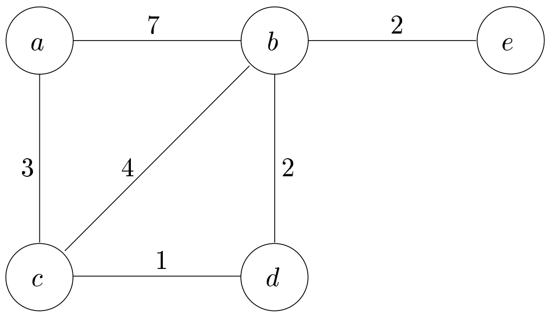
Ein *gerichteter Graph* $G = (V, E)$ besteht aus *Knotenmenge* V und *Kantenmenge* $E \subseteq V \times V$



$V = \{a, b, c, d\}$
$E = \{(a, c), (a, b), (c, b), (c, d), (d, b), (b, d)\}$

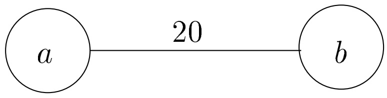
Kanten können gewichtet sein durch eine *Kostenfunktion* $c : E \to \mathbb{Z}$.

Ein *ungerichteter* $G = (V, E)$ besteht aus *Knotenmenge* V und *Kantenmenge* $E \subseteq P_2(V)$ = 2-elem. Teilmengen von V.

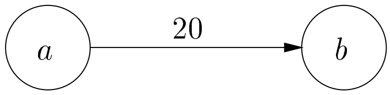


Mit Graphen können zwischen Objekten ($\hat{=}$ Knoten) binäre Beziehungen ($\hat{=}$ Kanten) modelliert werden.

1.) Orte mit Entfernungen



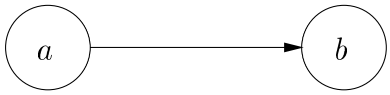
Länge
Kosten
Dauer



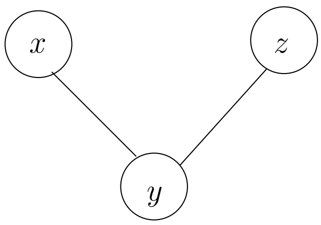
2.) Personen mit Beziehungen

verheiratet mit

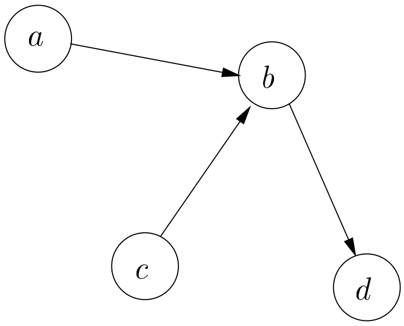
3.) Ereignisse mit Vorrang



a muß vor b geschehen



x ist zu y adjazent
x und y sind Nachbarn
x und z sind unabhängig
Der Grad von y ist 2



a ist Vorgänger von b
b ist Nachfolger von a
Eingangsgrad von b ist 2
Ausgangsgrad von b ist 1

Ein *Weg* ist eine Folge von adjazenten Knoten.
Ein *Kreis* ist ein Weg mit Anfangsknoten = Endknoten.

11.1 Implementation von Graphen

Es sei jedem Knoten eindeutig ein Index zugeordnet. Für den gerichteten Graphen auf Seite 147 ergibt sich:

Index	Knoten
0	a
1	b
2	c
3	d

Implementation durch Adjazenzmatrix

	0	1	2	3
0	0	1	1	0
1	0	0	0	1
2	0	1	0	1
3	0	1	0	0

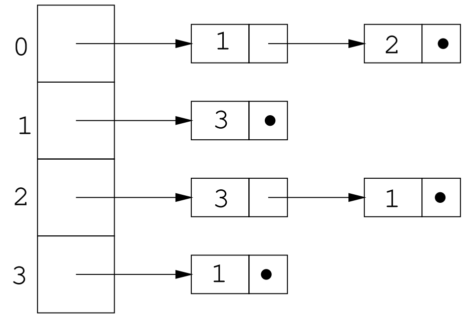
$$m[i,j] := \begin{cases} 1, & \text{falls } (i,j) \in E \\ 0 & \text{sonst} \end{cases}$$

	0	1	2	3
0	0	8	2	∞
1	∞	0	∞	4
2	∞	1	0	6
3	∞	2	∞	0

$$m[i,j] := \begin{cases} c(i,j), & \text{falls } (i,j) \in E \\ 0, & \text{falls } i = j \\ \infty & \text{sonst} \end{cases}$$

Platzbedarf $= O(|V|^2)$.
Direkter Zugriff auf Kante (i,j) in konstanter Zeit möglich.
Kein effizientes Verarbeiten der Nachbarn eines Knotens.
Sinnvoll bei dicht besetzten Graphen.
Sinnvoll bei Algorithmen, die wahlfreien Zugriff auf eine Kante benötigen.

Implementation durch Adjazenzlisten



i-te Liste enthält j,
falls $(i, j) \in E$

Platzbedarf = $O(|E|)$
Kein effizienter Zugriff auf Kante (x, y) möglich.
Sinnvoll bei dünn besetzten Graphen.
Sinnvoll bei Algorithmen, die, gegeben ein Knoten x, dessen Nachbarn verarbeiten müssen.

11.2 Graphalgorithmen

Die auf den folgenden Seiten verwendete Java-Implementation von gerichteten, unbewerteten Graphen stützt sich auf ein Array von Listen, welches in effizienter Weise das Abarbeiten der Nachbarn eines Knotens mit folgenden Methoden erlaubt:

```
void    reset         (int x); // setzt die Adjazenzliste fuer x zurueck
boolean moreNeighbors (int x); // testet, ob es weitere Nachbarn von x gibt
int     nextNeighbor  (int x); // liefert den naechsten Nachbarn von x
```

Die Klasse `Graph_IO` wurde definiert, um Graphen vom Benutzer einzulesen und ihre momentane Struktur anzuzeigen.

Die Klasse `Graph_Tiefensuche` führt auf einem gerichteten Graphen unter Verwendung eines Kellers eine Tiefensuche durch und vergibt Nummern an Knoten in der Reihenfolge, in der sie besucht werden. Bei nicht zusammenhängenden Graphen muß dazu die Suche mehrfach gestartet werden.

Die Klasse `Graph_TopoSort` führt auf einem gerichteten Graphen eine sogenannte topologische Sortierung durch und versucht eine Knoten-Nummerierung f : V $\to \mathbb{N}$ zu finden, die mit den Kantenrichtungen kompatibel ist, d.h. $(x, y) \in E \Rightarrow f(x) < f(y)$.

In der Klasse `Graph_Test` wird ein Graph eingelesen, angezeigt und die beiden Nummerierungen ermittelt.

```java
/***************************  Graph.java  ************************************/

/** Abstrakter Datentyp Graph mit den Methoden number, insert, reset,
    moreNeighbors, nextNeighbor
 */

public class Graph {

  private Liste[] g;                     // g ist ein Array von Listen

  public Graph(int n) {                  // Konstruktor fuer Graph mit n Knoten
    g = new Liste[n];                    // besorge Platz fuer Array mit n Listen
    for (int i=0; i < n; i++)            // besorge n mal
      g[i] = new Liste();                // Platz fuer eine Liste
  }

  public int number() {                  // liefere Zahl der Knoten
    return g.length;
  }

  public void insert(int i, int j){     // fuege fuer Knoten i den Nachbarn j ein
    g[i].insert(new Integer(j));
    g[i].advance();
  }

  public void reset(int i) {             // setze Nachbarschaftsliste i zurueck
    g[i].reset();
  }

  public void reset() {                  // setze Nachbarschaftslisten zurueck
    for (int i=0; i<g.length; i++)
      g[i].reset();
  }

  public boolean moreNeighbors(int i){ // teste, ob i weitere Nachbarn hat
    return !g[i].endpos();
  }

  public int nextNeighbor(int i) {       // liefere den naechsten Nachbarn
    Object o = g[i].elem();              // von Knoten i
    g[i].advance();                      // und ruecke eins weiter
    return ((Integer)o).intValue();      // liefere als int
  }
}
```

```
/*************************  Graph_IO.java        *****************************/

import AlgoTools.IO;

/** Klasse zum Einlesen und zum Ausgeben von Graphen
 */

public class Graph_IO {                              // Ein- und Ausgabe von Graphen

  public static Graph einlesen(){                    // lies Graph ein
    Graph g;                                         // Graph g
    int n = IO.readInt("Zahl der Knoten: ");         // erfrage Zahl der Knoten
    g = new Graph(n);                                // lege Graph an
    int[]a;                                          // array fuer Adjazenzliste
    for (int i=0; i<n; i++) {                        // fuer jeden Knoten
      a = IO.readInts("Nachbarn von "+i+": ");       // erfrage die Nachbarn
      for (int j=0; j<a.length; j++)                 // fuer jeden Nachbarn von i
        g.insert(i,a[j]);                            // fuege ihn bei Knoten i ein
    }                                                //
    g.reset();                                       // setze Graph zurueck
    return g;                                        // liefere Graph ab
  }

  public static void ausgeben(Graph g) {             // gib Graph aus
    for (int i=0; i<g.number(); i++){                // fuer jeden Knoten
      IO.print(i + " : ");                           // drucke seinen Namen
      while (g.moreNeighbors(i)) {                   // fuer jeden Nachbarn
        IO.print(" " + g.nextNeighbor(i));           // drucke dessen Namen
      }                                              //
      IO.println();                                  // Zeilenumbruch
    }                                                //
    g.reset();                                       // setze Adjazenzlisten zurueck
  }

}
```

```
/************************  Graph_Tiefensuche.java **************************/

/**    Tiefensuche auf einem Graphen                                         */

public class Graph_Tiefensuche {               // Tiefensuche auf Graphen

  static int id=0;                             // Variable fuer lfd. Nummer
  static boolean[]besucht;                     // zum Notieren, wer besucht ist
  static int[]ergebnis;

  private static void visit (Graph g, int k ) { // Tiefensuche beginnend bei k
    int x;                                     // Knotenname
    ergebnis[id++]=k;                          // notiere Knoten im Ergebnis
    besucht[k] = true;                         // markiere als besucht

    g.reset(k);                                // setze Nachbarn von k zurueck
    while (g.moreNeighbors(k)) {               // solange es noch Nachbarn gibt
      x =  g.nextNeighbor(k);                  // besorge naechsten Nachbarn
      if (!besucht[x]) visit(g,x);             // starte ggf. neue Rekursion
    }
  }

  public static int[]tiefensuche (Graph g) {    // oeffentliche Methode

    int k;                                     // aktueller Knoten
    besucht = new boolean[g.number()];         // lege array besucht an
    ergebnis= new int[g.number()];             // lege array ergebnis an

    for (k=0; k<g.number(); k++)               // notiere jeden Knoten
      besucht[k]=false;                        // als nicht besucht

    for (k=0; k<g.number(); k++)               // fuer jeden nicht besuchten
      if (!besucht[k]) visit(g,k);             // Knoten starte Rekursion

   return ergebnis;                            // liefere Ergebnis
  }
}
```

```java
/************************  Graph_TopoSort.java *****************************/

/** Topologisches Sortieren eines gerichteten Graphen                       */

public class Graph_TopoSort {                      // Klasse mit einer Methode

  public static int[]sort (Graph g){               // zum topologischen Sortieren

    int i, j;                                      // Knotennamen
    int id=0;                                      // Nummerierungsvariable
    int[]indegree = new int[g.number()];           // Vektor mit Eingangsgraden
    int[]ergebnis = new int[g.number()];           // Vektor fuer Ergebnis

    for (i=0; i<g.number(); i++) indegree[i]=0; // setze indegree auf 0

    g.reset();                                     // setze Adjazenzlisten zurueck
    for (i=0; i<g.number(); i++) {                 // fuer jeden Knoten i
      while (g.moreNeighbors(i)) {                 // solange er Nachbarn hat
        j = g.nextNeighbor(i);                     // hole Namen des Nachbarn
        indegree[j]++;                             // erhoehe dessen Eingangsgrad
      }
    }

    Schlange s = new Schlange(g.number());         // besorge eine Schlange

    for (i=0; i< g.number(); i++)                  // jeder Knoten
      if (indegree[i]==0)                          // mit Eingangsgrad 0
        s.enq(new Integer(i));                     // kommt in die Schlange

    while (!s.empty()) {                           // solange Schlange nicht leer
      i = ((Integer)s.front()).intValue();         // hole Knoten aus Schlange
      ergebnis[id++]=i;                            // vergib naechste ID
      g.reset(i);                                  // setze Knoten i zurueck
      while (g.moreNeighbors(i)){                  // solange i Nachbarn hat
        j = g.nextNeighbor(i);                     // holen Namen des Nachbarn
        indegree[j]--;                             // erniedrige den Eingangsgrad
        if (indegree[j]==0)                        // falls Eingangsgrad jetzt 0
          s.enq(new Integer(j));                   // Knoten in Schlange stellen
      }
    }
    if (id==g.number())                            // falls genuegend IDs vergeben
               return ergebnis;                    // gib Ergebnis zurueck
               else return null;                   // sonst gib null zurueck
  }
}
```

```
/**************************  Graph_Test.java *******************************/
import AlgoTools.*;

/** Programm zum Testen der Graph-Algorithmen
 */

public class Graph_Test  {

  public static void main (String argv[]) {

    int i;
    int[]e;
    Graph g = Graph_IO.einlesen();
    Graph_IO.ausgeben(g);

    e = Graph_Tiefensuche.tiefensuche(g);
    IO.println("Reihenfolge fuer Tiefensuche: ");
    for (i=0; i<e.length; i++) IO.print(e[i],3);
    IO.println();

    e = Graph_TopoSort.sort(g);
    if (e==null)
      IO.println("Graph ist zyklisch");
    else {
      IO.print("Graph ist azyklisch. Reihenfolge: ");
      for (i=0; i<e.length; i++) IO.print(e[i],3);
    }

  }

}
```