

Algorithmen

Vorlesung im WS 2003/2004

Oliver Vornberger

Olaf Müller

Ralf Kunze

Institut für Informatik
Fachbereich Mathematik/Informatik
Universität Osnabrück

Literatur

- Guido Krüger:
GoTo Java 2, Addison-Wesley, 2001.
- Ken Arnold & James Gosling & David Holmes:
Die Programmiersprache Java, Addison-Wesley, 2001.
- Dietmar Ratz & Jens Scheffler & Detlef Seese:
Grundkurs Programmieren in Java. Band 1: Der Einstieg in Programmierung und Objektorientierung, Carl Hanser, 2001.
- David Flanagan:
“Java in a Nutshell, (3. Auflage: Deutsche Ausgabe für Java 1.3), O’Reilly, 2001.
- Judith Bishop:
Java lernen, Addison Wesley, 2001.
- Timothy Budd:
Classic Data Structures in Java, Addison Wesley, 2001.

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 FrankLohmeyer 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.inf.uos.de/~ainf>

Osnabrück, im September 2003

Oliver Vornberger

Olaf Müller

Ralf Kunze

Inhaltsverzeichnis

1	Einführung	13
1.1	Informatik	13
1.2	Algorithmus, Programm, Prozess	14
1.3	Anweisungen, Ablaufprotokoll	14
2	Java	17
2.1	Sprachmerkmale	17
2.2	Variablen	18
2.3	Kontrollstrukturen	18
2.4	Einfache Datentypen	24
2.4.1	Ganze Zahlen (byte, short, int, long)	24
2.4.2	Gleitkommazahlen (float, double)	27
2.4.3	Boolean (boolean)	31
2.4.4	Charakter (char)	32
2.4.5	Typumwandlung	33
2.4.6	Konstanten	36
3	Felder	37
3.1	Feld von Ziffern	38
3.2	Feld von Daten	39
3.3	Feld von Zeichen	40
3.4	Feld von Wahrheitswerten	41
3.5	Feld von Indizes	42
3.6	Feld von Zuständen	43
3.7	Lineare und binäre Suche	45
4	Klassenmethoden	47

5	Rekursion	51
5.1	Fakultät, Potenzieren, Fibonacci, GGT	52
5.2	Türme von Hanoi	53
6	Komplexität und Verifikation	55
6.1	O-Notation	55
6.2	Korrektheit und Terminierung	59
6.3	Halteproblem	62
7	Sortieren	63
7.1	Selection Sort	64
7.2	Bubblesort	65
7.3	Mergesort	66
7.4	Quicksort	69
7.5	Bestimmung des Medians	70
7.6	Heapsort	72
7.7	Untere Schranke für Sortieren durch Vergleichen	76
7.8	Bucket Sort	78
7.9	Radix Sort	79
7.10	Externes Sortieren	81
8	Objektorientierte Programmierung	83
9	Abstrakte Datentypen	91
9.1	Liste	91
9.2	Keller	95
9.3	Exceptions: throw + try + catch	102
9.4	Schlange	105
9.5	Baum	108
9.6	Suchbaum	117
9.7	AVL-Baum	124
9.8	Spielbaum	136
10	Hashing	139
10.1	Offenes Hashing	140
10.2	Geschlossenes Hashing	140

11 Graphen	149
11.1 Implementation von Graphen	151
11.2 Graphalgorithmen	152

Verzeichnis der Java-Programme

AVLBaum.java	127
AVLBaumTest.java	135
ArrayAbzaehltreim.java	42
ArraySchlange.java	106
ArraySchlangeTest.java	107
AusnahmeKeller.java	103
AusnahmeKellerTest.java	104
Automat.java	44
Baum.java	108
Bedingung.java	19
BreitenSuche.java	113
BubbleSort.java	65
BucketSort.java	78
CharKeller.java	100
Collatz.java	17
Datum.java	84
Fakultaet.java	22
Fall.java	20
Feld.java	37
GGT.java	23
Ganzzahl.java	34
GeHashing.java	143
Gleitkomma.java	30
Graph.java	153
Graph_IO.java	154
Graph_Test.java	157

Graph.Tiefensuche.java	155
Graph.TopoSort.java	156
Hanoi.java	53
HashTest.java	146
HeapSort.java	74
Keller.java	95
KellerFehler.java	102
Kind.java	90
Klammer.java	99
Konstanten.java	36
Liste.java	92
Matrix.java	39
Menge.java	119
Merge.java	66
Methoden.java	48
OfHashing.java	142
Parameter.java	49
Person.java	85
PersonTest.java	87
Postfix.java	100
PostfixBaumBau.java	115
PostfixPreorderTest.java	116
PreorderTraverse.java	116
QuickSort.java	69
QuickSortTest.java	70
Rekursion.java	52
Reverse.java	98
Schlange.java	105
Schleife.java	21
SelectionSort.java	64
Sichtbarkeit.java	50
Sieb.java	41
SortTest.java	68
SpielBaum.java	138

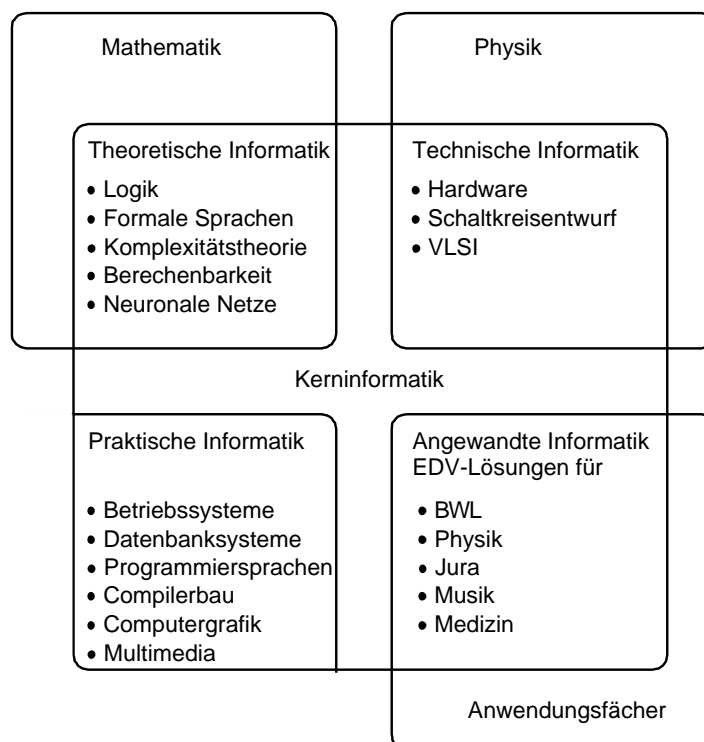
Student.java	86
SuchBaum.java	120
SuchBaumTest.java	122
Suche.java	45
TiefenSuche.java	112
Traverse.java	111
TraverseTest.java	114
Umwandlung.java	35
VerweisBaum.java	110
VerweisKeller.java	97
VerweisListe.java	93
VerweisListeTest.java	94
Zeichen.java	33
Zeichenkette.java	40
Ziffern.java	38

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	→	Maschinenprogramm
if (a > b)	Compiler	0110111011101110111

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)

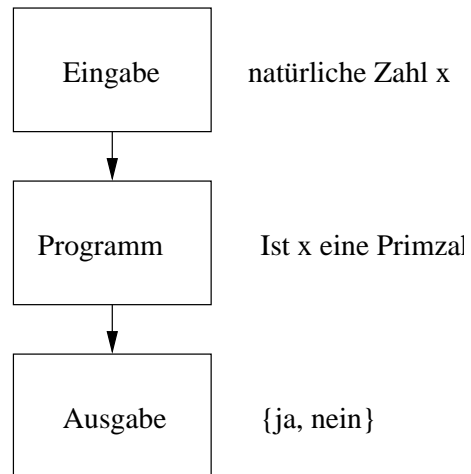
Zeile	x	z
	undef.	undef.
1	3	undef.
2	3	0
3	10	0
4	10	1
3	5	1
4	5	2
3	16	2
4	16	3
3	8	3
4	8	4
3	4	4
4	4	5
3	2	5
4	2	6
3	1	6
4	1	7
5	Ausgabe	7

Programm iteriert Collatz-Funktion

$f : \mathbb{N} \rightarrow \mathbb{N}$

$f(x)$ = 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, z; // definiere 2 Variablen

        x = IO.readInt("Bitte eine Zahl: "); // lies einen Wert ein
        z = 0; // setze z auf 0

        while (x != 1) { // solange x ungleich 1 ist
            if (x % 2 == 0) // falls x gerade ist
                x = x / 2; // halbiere x
            else // andernfalls
                x = 3*x+1; // verdreifache x und add. 1
            z = z+1; // erhoehere z um eins
        }
        IO.println("Anzahl der Iterationen: " + z); // gib z aus
    }
}
```

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 verschiedenen 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*).

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

2.3 Kontrollstrukturen

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


```
/****** 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;

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

public class Schleife {

    public static void main (String [] argv) {

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

        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 {
            x=IO.readInt("Bitte 1<= Zahl <=12"); // lies x ein
        } while (( x < 1) || (x > 12));        // solange x unzulaessig

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

```
/****** 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
 *
 * ggt(x,y) =    x                falls 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  $(x-y) = t*(f1-f2)$ 
 *
 * ggt(x,y) =    x                falls y = 0
 *              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!

$$\begin{array}{r}
 0011 \quad 3 \\
 + 1011 \quad -5 \\
 \hline
 = 1110 \quad -2
 \end{array}$$

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

$$\begin{array}{r}
 0111 \quad 7 \\
 + 0001 \quad 1 \\
 \hline
 = 1000 \quad -8 \quad \text{falsch}
 \end{array}$$

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

00111	7	00011	3
+ 00001	1	11011	-5
<u>0</u> 1000		<u>1</u> 1110	-2
Ergebnis undefiniert		Ergebnis ok!	

Operatoren

+	:	Ganzzahl	×	Ganzzahl	→	Ganzzahl	Addition
-	:	Ganzzahl	×	Ganzzahl	→	Ganzzahl	Subtraktion
*	:	Ganzzahl	×	Ganzzahl	→	Ganzzahl	Multiplikation
/	:	Ganzzahl	×	Ganzzahl	→	Ganzzahl	ganzzahlige Division
%	:	Ganzzahl	×	Ganzzahl	→	Ganzzahl	Modulo = Rest der Division
&	:	Ganzzahl	×	Ganzzahl	→	Ganzzahl	bitweise Und-Verknüpfung
	:	Ganzzahl	×	Ganzzahl	→	Ganzzahl	bitweise Oder-Verknüpfung
^	:	Ganzzahl	×	Ganzzahl	→	Ganzzahl	bitweise XOR-Verknüpfung
<<	:	Ganzzahl	×	Ganzzahl	→	Ganzzahl	Linksshift
>>	:	Ganzzahl	×	Ganzzahl	→	Ganzzahl	Vorzeichen erhaltender Rechtsshift
>>>	:	Ganzzahl	×	Ganzzahl	→	Ganzzahl	Vorzeichen ignorierender Rechtsshift
~	:	Ganzzahl			→	Ganzzahl	Bitweise Negation
-	:	Ganzzahl			→	Ganzzahl	Vorzeichenumkehr
++	:	Ganzzahl			→	Ganzzahl	Inkrement
--	:	Ganzzahl			→	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	XXXXXXXXXXXXXXXXXXXXXXXXXXXX
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

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

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

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:

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

Addition: (Exponenten angleichen, Mantissen addieren)

Beispiel:

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

Problem beim Angleichen der Exponenten:

Beispiel:

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

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:

$$\begin{array}{l}
 .2 \\
 2 \\
 2. \\
 2.0 \\
 2.538 \\
 2.538f \\
 2.5E2 = 2.5 * 10^2 = 250 \\
 2.5E-2 = 2.5 * 10^{-2} = 0.025
 \end{array}$$

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

/** Gleitkommaoperationen
 */

public class Gleitkomma {

    public static void main (String [] argv) {

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

        IO.println("1/2 + 1/4 + 1/8 + ...");
        summe = 0.0; summand = 1.0; // Initialisierungen
        while (summand > 0.00000000000001) { // 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("Naehierung von Pi :");
        pi = 4.0; // setze Naehierung 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 Naehierung
            IO.println( pi, 22, 16); // drucke Naehierung 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; // ver Hundertfache 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	false
true	true	true	true	false	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
	bell	7
\b	backspace	8
\t	horizontal tab	9
\n	newline	10
	vertical tab	11
\f	form feed	12
\r	carriage return	13
\101	Buchstabe A in Oktal-Schreibweise	13
\u0041	Buchstabe A in Hexadezimal-Schreibweise	13
\"	Anführungsstriche	34
\'	Apostroph	39
\\	backslash	92

Operatoren

<, <=, ==, >=, >, != : char × char → 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;

        IO.println( x & y ); // Obacht: jeweils cast auf int !
        IO.println( x | y ); // bitweise Und-Verknuepfung
        IO.println( x ^ y ); // bitweise Oder-Verknuepfung
        IO.println( ~x );    // 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
}

```

```
/****** 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. 9 Stellen
        long    l = 1000000000000000000L; // ca. 18 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 = 1/2;                    // ergibt 0 wegen ganzzahliger Division
        IO.println(d);
        d = 1/2.0;                  // ergibt 0.5 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[][] m = new double[4][3]; // 4x3 Matrix vom Typ double

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

        int[][] dreieck = {{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 , wert, ziffernwert; // Hilfsvariablen

        zeile = IO.readChars("Bitte Ziffernfolge: "); // Liest so lange Zeichen von
                                                    // der Tastatur, bis <Return>

        wert = 0;
        for (i=0; i < zeile.length; i++) {
            c = zeile[i];           // besorge naechstes Zeichen
            ziffernwert = (int) c - (int) '0'; // konvertiere Ziffer zur Zahl
            wert = 10*wert + ziffernwert; // verknuepfe mit bisherigem Wert
        }
        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 = summe + a[i][k]*b[k][j]; // von Zeile i und Spalte j
                c[i][j] = summe;         // die Produktsumme kommt nach 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;

        IO.print(s);

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

            i++;
        }

        IO.println(t);
    }
}
```


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.println(i,10); // gib sie aus, und
                for (j = i+i; j < n; j = j+i) // fuer alle Vielfachen j von i
                    sieb[j] = false; // streiche j aus der Liste
            }
    }
}
```

3.5 Feld von Indizes

```
/* ***** ArrayAbzaehltreim.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 ArrayAbzaehltreim {

    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
$\delta : S \times \Sigma \rightarrow S$	=	Überföhrungsfunktion
$s_0 \in S$	=	Anfangs- oder Startzustand
$F \subseteq S$	=	Endzustände

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

$$\delta^*(s_0, w) \in F$$

$$\delta^*(s_0, x_0x_1x_2 \dots x_k) = \delta(\dots \delta(\delta(\delta(s_0, x_0), x_1), x_2) \dots x_k).$$

Die Wirkungsweise der Überföhrungsfunktion δ kann durch einen Zustandsüberföhrungsgraphen beschrieben werden. In diesem Graphen föhrt 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.

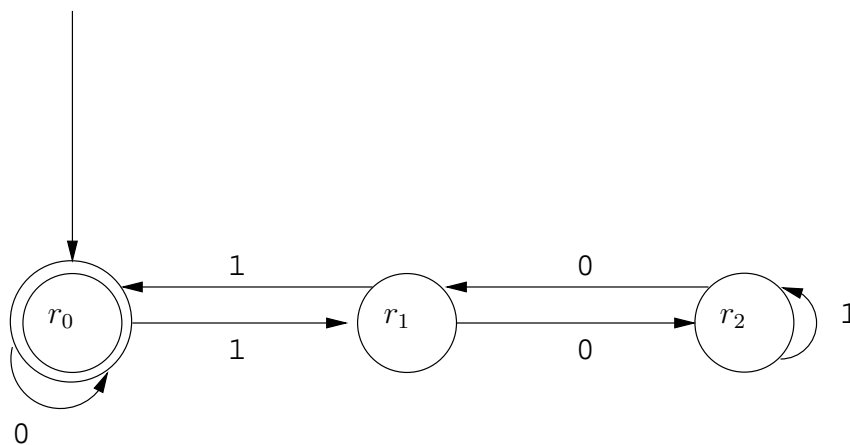
$$S = \{r_0, r_1, r_2\}$$

$$\Sigma = \{ '0', '1' \}$$

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 betragt. An der Kante steht das jeweils vorderste Bit der Dualzahl, die von links nach rechts abgearbeitet wird.



```
/****** Automaten.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

        if (z == 0)
            IO.println("ist durch 3 teilbar");           // gib Ergebnis bekannt
        else
            IO.println("ist nicht durch 3 teilbar");

    }
}
```

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: "); // bestimme das 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.print("Minimum = " + min); // gib Ergebnis bekannt
        IO.println(" an Position " + index);

        a = IO.readInts("Bitte Zahlenfolge: "); // suche in einer Folge
        x = IO.readInt ("Bitte zu suchende Zahl: "); // ein bestimmtes 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);

        a = IO.readInts("Bitte sortierte Zahlenfolge: "); // suche in sortierter Folge
        x = IO.readInt ("Bitte zu suchende Zahl: "); // ein bestimmtes Element
        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");// gib Ergebnis bekannt
        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 =

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

Korollar: $2^k - 1$ Zahlen verursachen höchstens k Schleifendurchläufe, da nach k Halbierungen die Intervalllänge 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 außerhalb derselben Klasse aufgerufen werden. Die Zugriffsrechte werden über sogenannte *Modifier* geregelt: Durch den Modifier `private` wird der Zugriff nur von der eigenen Klasse ermöglicht, durch den Modifier `public` darf von einer beliebigen anderen Klasse zugegriffen 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
    // und ohne Parameter
    public static void bitte() {
        IO.println("Bitte Eingabe: ");
    }

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

    // Methode mit Rueckgabewert
    // mit einem Integer-Parametern
    // lokale Variablen
    // berechne 2 hoch n
    // liefere Ergebnis ab
    public static int zweihoch(int n) {
        int i, h = 1;
        for (i=0; i < n; i++) h = h * 2;
        return h;
    }

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

    // Methode ohne Rueckgabewert
    // oeffentlich aufrufbar
    // rufe bitte auf
    // rufe sterne auf
    // rufe zweihoch auf
    // rufe ggt auf
    public static void main (String [] argv) {
        bitte();
        sterne(3*5+7);
        IO.println("2 hoch 7 ist " + zweihoch(7));
        IO.println("ggt(28,12) = " + ggt(28,12));
    }
}

```



```

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

import AlgoTools.IO;

/** Uebergabe von Arrays an Methoden
 */

public class Parameter {

    public static void zeige(int[] a) {
        for (int i=0; i < a.length; i++)
            IO.print(a[i],5);
        IO.println();
    }

    public static int summe(int[] a) {
        int s = 0;
        for (int i=0; i < a.length; i++)
            s = s + a[i];
        return s;
    }

    public static int[] kopiere(int[] a) {
        int[] b = new int[a.length];
        for (int i=0; i < a.length; i++)
            b[i] = a[i];
        return b;
    }

    public static void reset(int[] a) {
        for (int i=0; i < a.length; i++)
            a[i] = 0;
    }

    public static void main(String [] argv) {
        int[] folge = {1,4,9,16,25,36};
        int[] noch_eine_folge;
        zeige(folge);
        IO.println("Summe = " + summe(folge));
        noch_eine_folge = kopiere(folge);
        zeige(noch_eine_folge);
        reset(folge);
        zeige(folge);
    }
}

```

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

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

public class Sichtbarkeit {

    private static int grenze;           // globale Variable in dieser Klasse
    private static int i = 100;         // globale Variable in dieser Klasse

    public static void setze() {        // Methode ohne Parameter
        grenze = i;                     // setze globale grenze auf 42
    }

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

    public 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();
    }

    public 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 {

    public 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
    }

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

    public 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
    }

    public 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>0) {
            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).`

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

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:

$$\begin{array}{rcl} f(n) & = 2 \cdot f(n-1) + 1 & = 2 \cdot (2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1 \\ & \uparrow & \uparrow \\ & \text{Rekursionsgleichung} & \text{Induktionsannahme} \end{array}$$

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 $f, g : \mathbb{N} \rightarrow \mathbb{N}$

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 $f = O(g)$

⇒ 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	...	2^n	3^n	$n!$
↑	↑	↑	↑	↑			↑		↑
binäre Suche	lineare Suche	schlaues Sortieren	dummes Sortieren	Gleichungs- system lösen			alle Teil- mengen		alle Permu- tationen

Annahme: 1 Schritt dauert $1 \mu\text{s} = 0.000001 \text{ s}$

$n =$	10	20	30	40	50	60
n	$10 \mu\text{s}$	$20 \mu\text{s}$	$30 \mu\text{s}$	$40 \mu\text{s}$	$50 \mu\text{s}$	$60 \mu\text{s}$
n^2	$100 \mu\text{s}$	$400 \mu\text{s}$	$900 \mu\text{s}$	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

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

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 $\sum_{i=1}^n i \cdot \frac{1}{2^i} \leq 2 \Rightarrow O(1)$

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 $s = AAA\dots AB$
 $p = A\dots AB$

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 $n \rightarrow \infty$ und $\alpha > 1$ geht $\frac{1}{\alpha^{n-2}} \rightarrow 0$.

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

\Rightarrow 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 $1.62^n \approx 15.000$
 $2^n \approx 1.000.000$.

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.

Syntaktisch lassen sich diese Zusicherungen am einfachsten als Kommentare formulieren.

Z.B. `/* i > 0 ∧ z = i2 */`

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

```
i = i - 1;
⇒ /* i ≥ 0 ∧ 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 ∧ x2 ≤ n) = Schleifeninvariante P */
while (z <= n) /* Q */ {
  /* z = (x+1)2 ∧ y = 2x+1 ∧ x2 ≤ n ∧ z ≤ n ⇒ (x+1)2 ≤ n */
  x = x + 1;
  /* z = x2 ∧ y = 2x-1 ∧ x2 ≤ n */
  y = y + 2;
  /* z = x2 ∧ y = 2x+1 ∧ x2 ≤ n */
  z = z + y;
  /* z = x2 + 2x + 1 ∧ y = 2x+1 ∧ x2 ≤ n */
  /* (z = (x+1)2 ∧ y = 2x+1 ∧ x2 ≤ n) = P */
}
/* (z = (x+1)2 ∧ y = 2x+1 ∧ x2 ≤ n ∧ z > n) = P ∧ ¬Q */
/* x2 ≤ 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 $z > n$

⇒ Abbruch gesichert

⇒ 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.

⇒ 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 ≥ 0 */
x = 2; y = n; z = 1;
/* z · xy = 2n ∧ y ≥ 0 */
while (y > 0) {
  /* z · xy = 2n ∧ y > 0 */
  if (y % 2 == 1) {
    /* z · xy = 2n ∧ y > 0 ∧ y ungerade */
    z = z * x;
    /*  $\frac{z}{x} \cdot x^y = 2^n \wedge y > 0 \wedge y$  ungerade */
    y = y - 1;
    /*  $\frac{z}{x} \cdot x^{y+1} = 2^n \wedge y \geq 0$  */
  } else {
    /* z · xy = 2n ∧ y > 0 ∧ 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 · xy = 2n ∧ y ≥ 0 */
  }
  /* z · xy = 2n ∧ y ≥ 0 */
}
/* z · xy = 2n ∧ y ≥ 0 ∧ y ≤ 0 */
/* z · xy = 2n ∧ y = 0 ⇒ z = 2n */
IO.println (z);
/* Ausgabe: 2n */

```

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

```
public 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:

```
import AlgoTools.IO;

public class Quer {

    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, wenn das Programm Quer.class auf den String q angesetzt wird ? D.h.

```
java Quer < Quer.java
```

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

        int i, j, pos, min; // 2 Indizes, Position, Minimum

        for (i=0; i<a.length-1; i++) { // durchlaufe Array
            pos = i; // Index des bisher kleinsten
            min = a[i]; // Wert des bisher kleinsten
            for (j=i+1; j<a.length; j++) // durchlaufe Rest des Array
                if (a[j] < min) { // falls kleineres gefunden,
                    pos = j; // merke Position des kleinsten
                    min = a[j]; // merke Wert des kleinsten
                }
            a[pos] = a[i]; // speichere bisher kleinstes um
            a[i] = min; // 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

$$\begin{aligned}
 & n - 1 + n - 2 + n - 3 + \dots + 1 \\
 & = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)
 \end{aligned}$$

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

Bemerkung: Diese Vorgehensweise wird *Divide & Conquer* (Teile und Beherrsche) genannt, da das ursprüngliche Problem zunächst in unabhängige Teilprobleme zerlegt wird und danach die Teillösungen wieder zusammengeführt werden.

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++];
        }

        while (i<a.length) c[k++] = a[i++]; // ggf.: Rest von Folge a
        while (j<b.length) c[k++] = b[j++]; // ggf.: Rest von Folge b

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

Analyse von Mergesort (und ähnlich gelagerten Rekursionen)

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

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

Zeige: $f(n) \leq (c_1 + c_2) \cdot n \cdot \log n + c_1$

Verankerung:

$n = 1 \Rightarrow f(1) \leq c_1$ nach Rekursion

$$f(1) \leq (c_1 + c_2) \cdot 1 \cdot \log 1 + c_1$$

Induktionsschluss

Sei bis $n - 1$ bewiesen

$$f(n) \leq 2 \cdot f(\frac{n}{2}) + c_2 \cdot n$$

↑

Rek.

$$\leq 2 \cdot [(c_1 + c_2) \cdot \frac{n}{2} \cdot \log \frac{n}{2} + c_1] + c_2 \cdot n$$

↑

Induktionsannahme

$$= 2 \cdot [(c_1 + c_2) \cdot \frac{n}{2} \cdot (\log n - 1) + c_1] + c_2 \cdot n$$

$$= (c_1 + c_2)n \cdot \log n - (c_1 + c_2) \cdot n + 2c_1 + c_2 \cdot n$$

$$= [(c_1 + c_2)n \cdot \log n + c_1] + [c_1 - c_1 \cdot n]$$

$$\leq (c_1 + c_2)n \cdot \log n + c_1$$

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 aufrufen.
        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
    }
}

```

Die Analyse der Laufzeit von Quicksort stützt sich auf folgende Rekursionsungleichung für die Anzahl der Schritte $f(n)$ bei n zu sortierenden Daten im günstigsten Fall (Partitionen sind immer gleich groß):

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

Daraus ergibt sich $f \in O(n \cdot \log n)$. Diese Komplexität gilt auch für den durchschnittlichen Fall; es lässt sich zeigen, dass die Zahl der Schritte nur um den konstanten Faktor 1.4 wächst.

Im ungünstigsten Fall (bei entarteten Partitionen) gilt $f \in O(n^2)$.

```

/***** 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

```

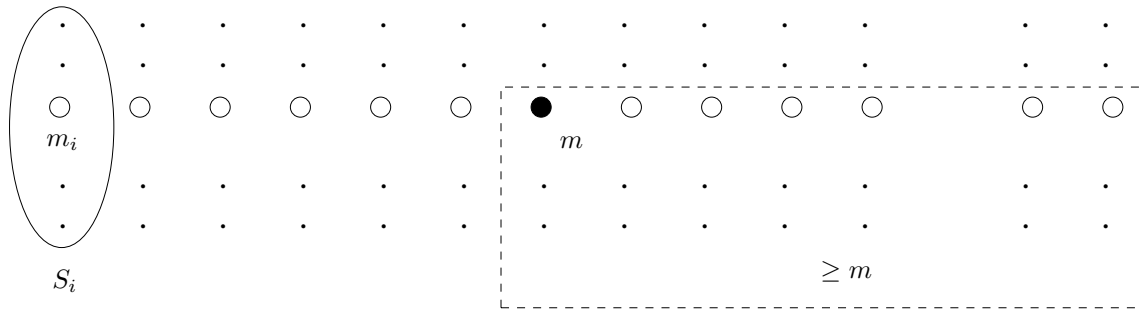
public static int select (int[] S, int k) {
    /* Liefert aus dem Feld S das k-t kleinste Element. */

    int[] A, B, C;

    int n = |S|;
    if (n < 50) return k-t kleinstes Element per Hand;
    else {
        zerlege S in Gruppen zu je 5 Elementen  $S_1, S_2, \dots, S_{\frac{n}{5}}$ ;
        Bestimme in jeder Gruppe  $S_i$  den Median  $m_i$ ;
        Bestimme den Median der Mediane:
         $m = \text{select}(\bigcup_i m_i, \frac{n}{10})$ ;
         $A = \{x \in S \mid x < m\}$ ;
         $B = \{x \in S \mid x == m\}$ ;
         $C = \{x \in S \mid x > m\}$ ;
        if ( $|A| \geq k$ ) return select (A, k);
        if ( $|A|+|B| \geq k$ ) return m;
        if ( $|A|+|B|+|C| \geq k$ ) 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:

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

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

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

Beweis durch Induktion:

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

Sei bis $n - 1$ bewiesen

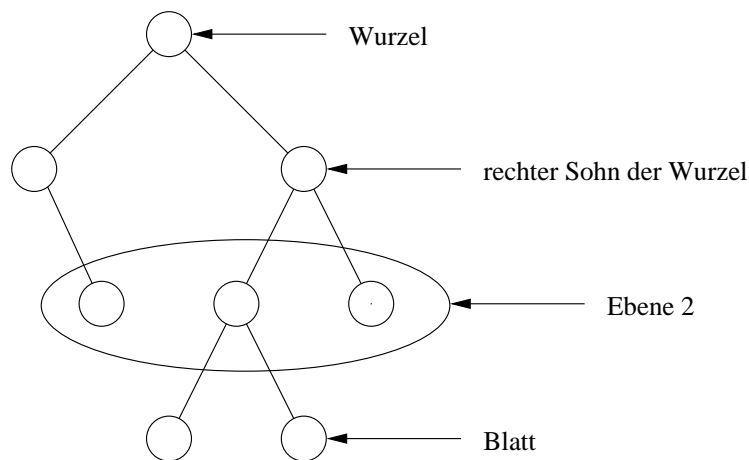
$$\begin{aligned} f(n) &\leq c \cdot n + f(\frac{n}{5}) + f(\frac{3}{4}n) \\ &\quad \uparrow \\ &\text{Rekursionsgl.} \\ &\leq c \cdot n + 20 \cdot c \cdot \frac{n}{5} + 20 \cdot c \cdot \frac{3}{4} \cdot n \\ &\quad \uparrow \\ &\text{Ind.-Annahme} \\ &= 1 \cdot c \cdot n + 4 \cdot c \cdot n + 15 \cdot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

7.6 Heapsort

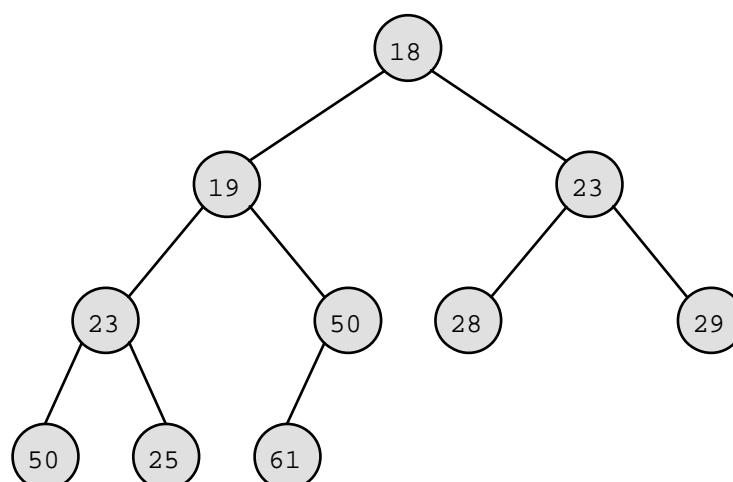
Baum und Heap

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 $0, 1, \dots, n - 1$ 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_0, \dots, a_{n-1} im Array a .

```

Baue einen Heap mit den Werten aus a;
for (i=0; i<n; i++) {
  liefere Minimum aus der Wurzel;
  entferne Wurzel;
  reorganisiere Heap;
}

```

Idee für Wurzelentfernen:

Entferne "letzten" Knoten im Heap und schreibe seinen Schlüssel in die Wurzel.

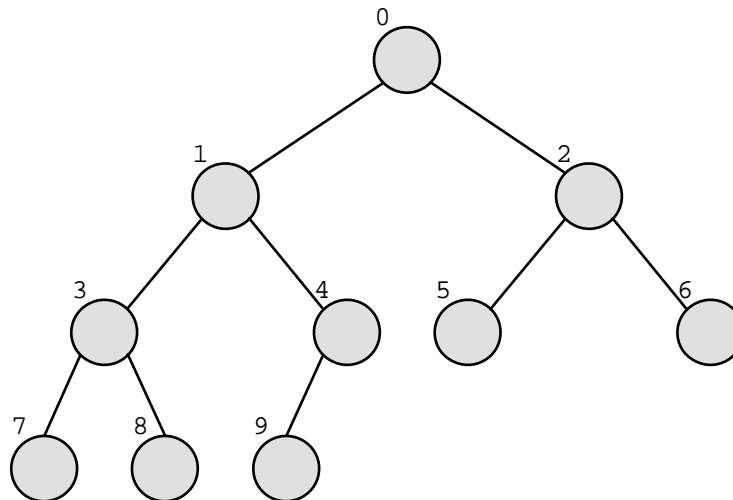
Vertausche so lange Knoten mit "kleinerem" Sohn, bis Heapbeziehung eintritt.

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

```
int[] a = new int [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 .

```

/***** 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)/2; l>=0; l--) // beginnend beim letzten Vater
            sift(a,l,n-1); // jeweils Heap reparieren

        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: $\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \dots < 2$

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

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

für best, average und worst case.

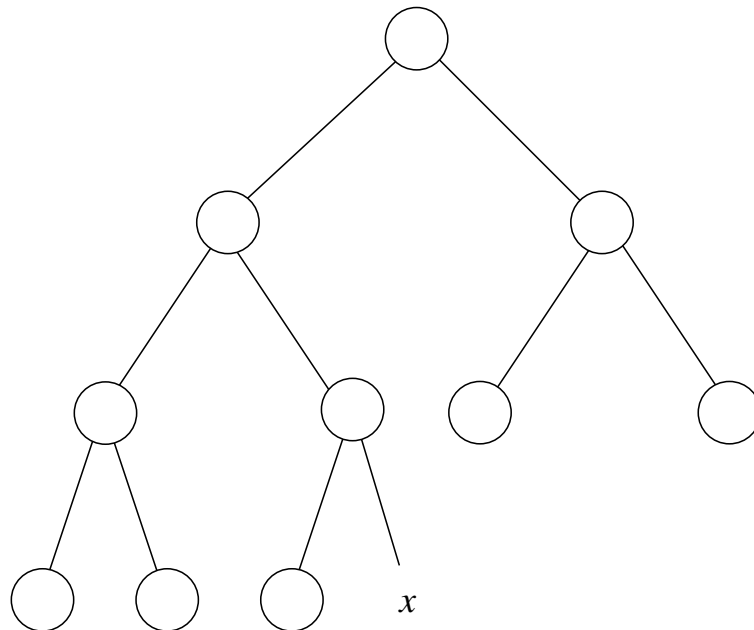
Weitere Einsatzmöglichkeit des Heaps

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.



```

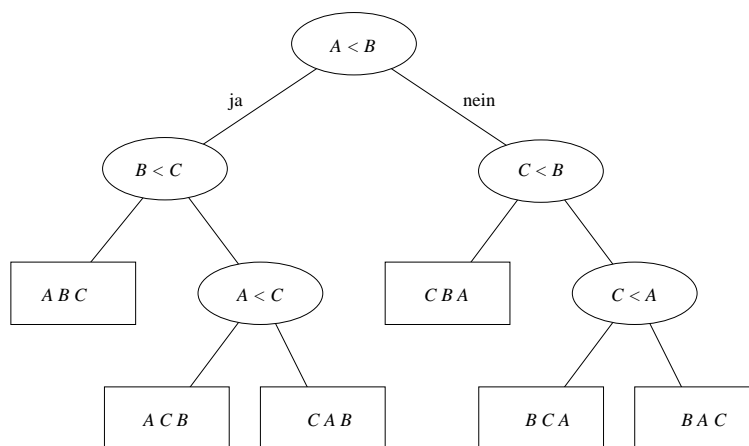
i = neuer Index;
while (a[i] < a[vater(i)]) {
    tausche a[i] mit a[vater(i)];
    i = vater(i);
}

```

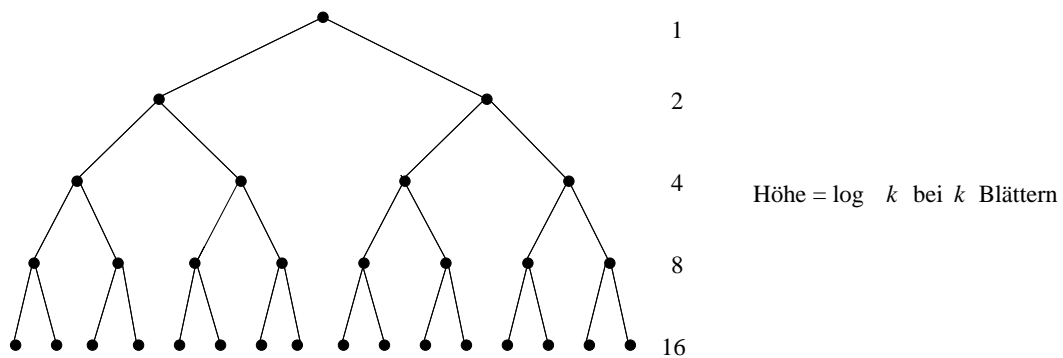
Aufwand $O(\log n)$

7.7 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.



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

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

\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.

7.8 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; // Alphabetgrosse 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.print("sortiert mit Bucket-Sort: "); // Ergebnis ausgeben
        IO.println(ergebnis);
    }
}
// Aufwand: O(n) + O(N) bei n(=a.length) zu sortierenden Zeichen
// aus einem N-elementigen Alphabet

```

7.9 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}, \dots, W_0 gebildet.

W_N enthält die unsortierten Wörter in der gegebenen initialen Reihenfolge.

W_j mit $j \in [0, \dots, N - 1]$ enthält alle Wörter, aufsteigend sortiert bezüglich der Positionen $j + 1, \dots, N$.

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):

```

W4: hefe bach gaga cafe geha
W3: gaga geha hefe cafe bach
W2: bach hefe cafe gaga geha
W1: bach cafe gaga hefe geha
W0: bach cafe gaga geha hefe

```

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:

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

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 Buchstaben einmal verteilt und einmal aufgesammelt, jeweils in konstanter Zeit.

Bei Wörtern **unterschiedlicher** Länge bilde zunächst

$\text{laenge}[j] = \{w \mid \text{Länge von Wort } w = j\}$

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, \dots, 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.

7.10 Externes Sortieren

Problem: Sortieren auf Medien mit sequentiellm Lese- und Schreibzugriff.

Lösung: Wiederholtes Mischen von bereits sortierten Teilfolgen.

Ein *Lauf* ist eine monoton nicht fallende Teilfolge.

$$\begin{array}{cccccc} \underbrace{3 \quad 7 \quad 8}_{\text{Lauf}} & \underbrace{2 \quad 9}_{\text{Lauf}} & \underbrace{4}_{\text{Lauf}} & \underbrace{1 \quad 5 \quad 6}_{\text{Lauf}} \end{array}$$

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 Instanzieren 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:	<code>public</code>	<code>protected</code>	paketsichtbar	<code>private</code>
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)
 * und 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); // kreierte Datum 15.08.1972
        d = new Datum (1972);     // kreierte 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 2001 - geb_datum.jahr; // liefert das Lebensalter
    }

    public static void main (String [] argv) {

        Person p;           // deklarriere 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; // deklarriere 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 [] argv) {

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

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

        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
    }
}

```

Statisches Binden bezeichnet den Vorgang, einem Objekt zur Übersetzungszeit den Programmcode seiner Methoden zuzuordnen; **dynamisches Binden** bezeichnet den Vorgang, einem Objekt zur Laufzeit den Programmcode seiner Methoden zuzuordnen.

Folgende Fälle können auftreten:

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

Wird übersetzt und zwei Instanzen werden angelegt.

```
IO.println(p.fach);
```

Führt zu einem Übersetzungsfehler, da Instanzen vom Typ `Person` nicht über das Datenfeld `fach` verfügen.

```
IO.println((Student)p.fach);
```

Wird übersetzt, da eine Instanz vom Typ `Student` über das Datenfeld `fach` verfügt. Führt aber zu einem Laufzeitfehler, da `p` zur Laufzeit nicht auf einen Studenten zeigt.

```
p = s;
```

Wird übersetzt und ausgeführt. `p` zeigt nun auf einen Studenten.

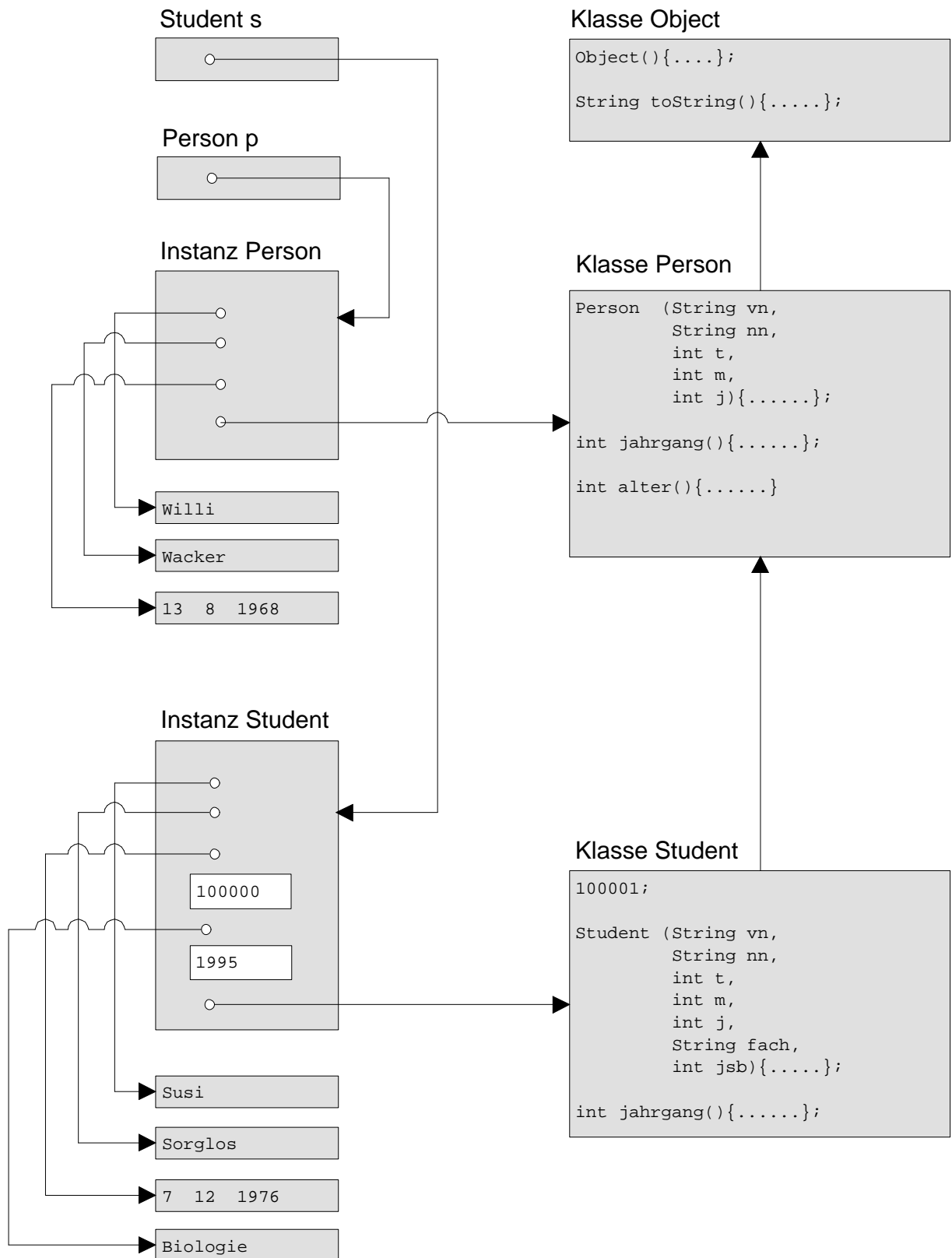
```
IO.println((Student)p.fach);
```

Wird übersetzt und ausgeführt, da `p` nun auf eine Instanz vom Typ `Student` zeigt, welche über das Datenfeld `fach` verfügt.

```
IO.println(p.jahrgang());
```

Wird übersetzt und ausgeführt. Es kommt die Methode `jahrgang()` aus der Klasse `Student` zum Einsatz. Hinweis: Beim *Statischen Binden* wäre bereits zur Übersetzungszeit dem Objekt `p` die Methode `jahrgang` der Klasse `Person` zugeordnet worden.

Auf der folgenden Seite verdeutlicht eine Grafik das Zusammenspiel zwischen den Klassen `Person` und `Student` und ihrer Instanzen. Auf dem Laufzeitkeller führt von der Variable `s` ein Verweis auf die Speicherfläche, welche eine Instanz vom Typ `Student` aufnimmt. Da einige Bestandteile von `Student` (nämlich `vorname`, `nachname`, `geb_datum` und `fach`) selbst wieder Objekte sind, führen dorthin weitere Verweise, die Instanzvariablen `matr_nr` und `jsb` werden hingegen explizit gespeichert. Ein weiterer Verweis (gezeichnet von links nach rechts) führt zu den Klassenvariablen und Klassenmethoden; von dort sind die Oberklassen erreichbar, von denen geerbt wurde.



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

/** Klasse Kind
 * bestehend aus Nummer und Verweis auf Nachbarkind
 * mit Konstruktor zum Anlegen eines Kindes
 * ermöglicht Implementierung eines Abzaehleims 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; // deklarriere 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

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.

Interfaces enthalten nur Methodenköpfe und Konstanten. Ein Interface stellt eine Schnittstelle dar und legt damit die Funktionalität seiner Methoden fest, ohne diese zu implementieren. Dies geschieht 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.

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`:

<code>empty</code>	: <code>Liste</code>	→ <code>boolean</code>	liefert <code>true</code> , falls <code>Liste</code> leer
<code>endpos</code>	: <code>Liste</code>	→ <code>boolean</code>	liefert <code>true</code> , wenn <code>Liste</code> abgearbeitet
<code>reset</code>	: <code>Liste</code>	→ <code>Liste</code>	das erste Listenelement wird zum aktuellen
<code>advance</code>	: <code>Liste</code>	→ <code>Liste</code>	der Nachfolger des akt. wird zum aktuellen
<code>elem</code>	: <code>Liste</code>	→ <code>Objekt</code>	liefert das aktuelle Element
<code>insert</code>	: <code>Liste</code> × <code>Objekt</code>	→ <code>Liste</code>	fügt vor das aktuelle Element ein Element ein; das neu eingefügte wird zum aktuellen
<code>delete</code>	: <code>Liste</code>	→ <code>Liste</code>	löscht das aktuelle Element; der Nachfolger wird zum aktuellen

```

/***** Liste.java *****/
/** Interface fuer den ADT Liste */

public interface Liste {

    public boolean empty();           // true, wenn Liste leer, false sonst

    public boolean endpos();         // true, wenn Liste am Ende, false sonst

    public void reset();             // rueckt an den Anfang der Liste

    public void advance();           // rueckt in Liste eine Position weiter

    public Object elem();            // liefert Inhalt des aktuellen Elements

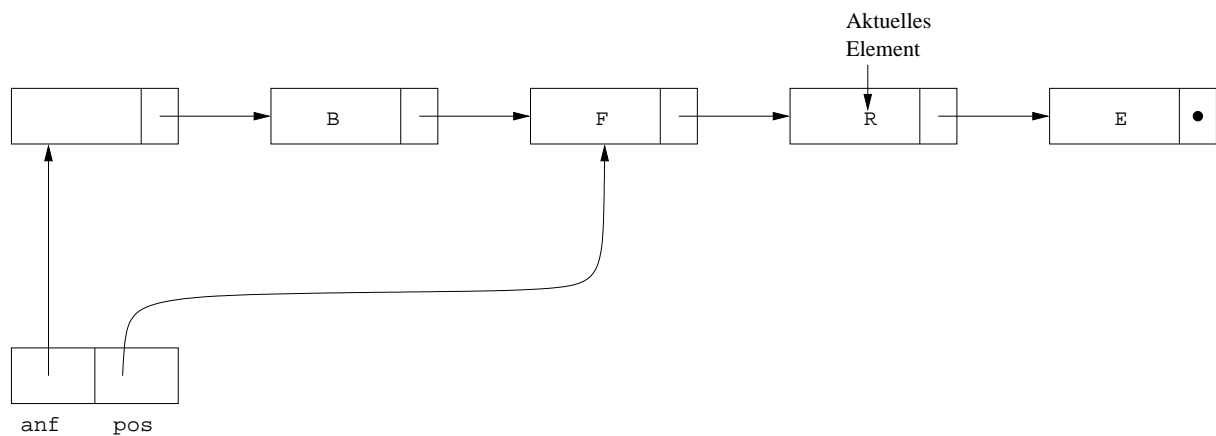
    public void insert(Object x);    // fuegt x vor das aktuelle Element ein
                                     // und macht x zum aktuellen Element

    public void delete();            // entfernt aktuelles Element und macht
                                     // seinen Nachfolger zum aktuellen Element

}

```

Konzept zur 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.

Hinweis zur Fehlerbehandlung:

In den folgenden Implementationen wird nach Eintreten einer fehlerhaften Situation durch Werfen einer `RuntimeException` ein Programmabbruch verursacht. Eine erweiterte Fehlerbehandlung (die auch das Abfangen von Fehlern vorsieht) wird in Kapitel 9.3 vorgestellt.

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

/** Implementation des Interface Liste mithilfe von Verweisen */

public class VerweisListe implements Liste {

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

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

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

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

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

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

    public void advance() { // rueckt in Liste vor
        if (endpos()) throw new
            RuntimeException("in VerweisListe.advance");
        pos = pos.next;
    }

    public Object elem() { // liefert aktuelles Element
        if (endpos()) throw new
            RuntimeException("in VerweisListe.elem");
        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()) throw new
            RuntimeException("in VerweisListe.delete");
        pos.next = pos.next.next;
    }
}
```

```

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

/** Testet die Implementation der VerweisListe */

public class VerweisListeTest {

    public static void main (String [] argv) {

        Liste l = new VerweisListe();           // kreierte Liste
        Student s;                             // deklarriere Student

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

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

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

        s = new Student("Susi", "Sorglos", 10, 7, 1973, "JUR", 1996); // kreierte 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:

```

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

```

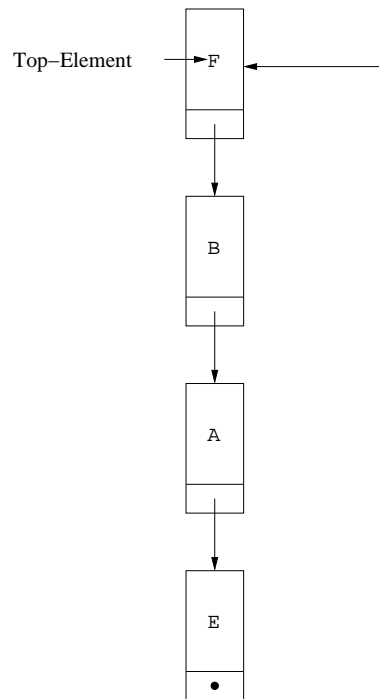
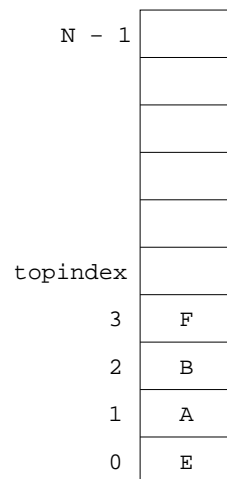
Semantik der Kelleroperationen (LIFO: Last in, first out):

- 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 .

```

/***** Keller.java *****/
/** Interface fuer den ADT Keller */
public interface Keller {
    public boolean empty(); // liefert true, falls Keller leer, false sonst
    public void push(Object x); // legt Objekt x auf den Keller
    public Object top(); // liefert oberstes Kellerelement
    public void pop(); // entfernt oberstes Kellerelement
}

```

Implementation eines Kellers mit Verweisen**Implementation eines Kellers mit einem Array (LIFO: Last in, first out)**


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

/** Implementation eines Kellers mithilfe von Verweisen */

public class VerweisKeller implements Keller {

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

    private KellerEintrag top; // verweist auf obersten KellerEintrag

    public VerweisKeller() { // 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 Top-Element
        if (empty()) throw new
            RuntimeException("in VerweisKeller.top");
        return top.inhalt;
    }

    public void pop() { // entfernt Top-Element
        if (empty()) throw new
            RuntimeException("in VerweisKeller.pop");
        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 VerweisKeller();           // 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
 *  und markiert die erste fehlerhafte Position
 */

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 VerweisKeller(); // Keller fuer Zeichen

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

        for (i=0; i < c.length && !fehler; i++){

            switch (c[i]) {

                case '(' : // oeffnende Klammer
                case '[' : k.push(new Character(c[i])); break; // auf den Keller

                case ')' : if (!k.empty() && // runde Klammer zu
                    ((Character)k.top()).charValue()=='(') // ueberpruefen
                    k.pop(); // und vom Keller
                    else fehler = true; break; // sonst Fehler

                case ']' : if (!k.empty() && // eckige Klammer zu
                    ((Character)k.top()).charValue()=='[') // ueberpruefen
                    k.pop(); // und vom Keller
                    else fehler = true; break; // sonst Fehler

                default: // beliebiges Zeichen
            }
            IO.print(" "); // weiterruecken in
        } // der Ausgabe

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



```

        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: throw + try + catch

Über das Auslösen einer *RuntimeException* hinaus besteht in Java die Möglichkeit, eigene Fehlerklassen zu definieren. Dies bedeutet, daß bei Vorliegen eines fehlerhaften Zustands eine speziell entworfene 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, daß in den Methoden `top` und `pop` anstelle der `RuntimeException` 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 ******/
/** Implementation eines Kellers unter Verwendung von eigenen Exceptions
    (Obacht: ohne Bezug auf das Interface Keller ! ) */
public class AusnahmeKeller {

    private static class KellerEintrag {
        Object inhalt;           // Inhalt des KellerEintrags
        KellerEintrag next;     // Verweis 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 Integer gecastet
 *  und mithilfe der Methode intValue() ausgewertet.
 *  Obacht: Nach der Schleife wird ein Fehler verursacht !
 */

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();

            k.pop();                                        // verursache Fehler

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

        IO.println("Hier geht es im Programmfluss weiter.");
    }
}
```


9.4 Schlange

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

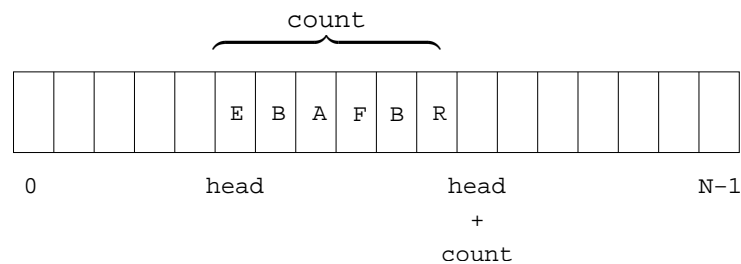
Schnittstelle des ADT Schlange:

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

```

/***** Schlange.java *****/
/** Interface zum ADT Schlange */
public interface Schlange {
    public void enq( Object x );           // Fuegt x hinten ein
    public void deq();                     // Entfernt vorderstes Element
    public Object front();                 // Liefert vorderstes Element
    public boolean empty();                // Testet, ob Schlange leer ist
}
  
```

Konzept zur Implementation einer Schlange mit einem Array:



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

/** Implementation des Interface Schlange mit Hilfe eines Arrays */

public class ArraySchlange implements Schlange {

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

    public ArraySchlange(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()) throw new RuntimeException("in ArraySchlange.enq");
        inhalt[(head+count)%inhalt.length]=x; // Element einfuegen
        count++;                       // Anzahl inkrementieren
    }

    public void deq() {               // Entfernt vorderstes Element
        if (empty()) throw new RuntimeException("in ArraySchlange.deq");
        inhalt[head] = null;           // Verweis auf null setzen
        head = (head + 1) % inhalt.length; // Anfang-Index weiterruecken
        count--;                       // Anzahl dekrementieren
    }

    public Object front() {           // Liefert Element,
        if (empty()) throw new RuntimeException("in ArraySchlange.front");
        return inhalt[head];           // welches am Anfang-Index steht
    }
}
```

```
/* ***** ArraySchlangeTest.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 ArraySchlangeTest {

    public static void main(String [] argv) {

        Schlange q = new ArraySchlange(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 Wurzelement

```

/***** Baum.java *****/
/** Interface Baum mit 7 Methoden. */

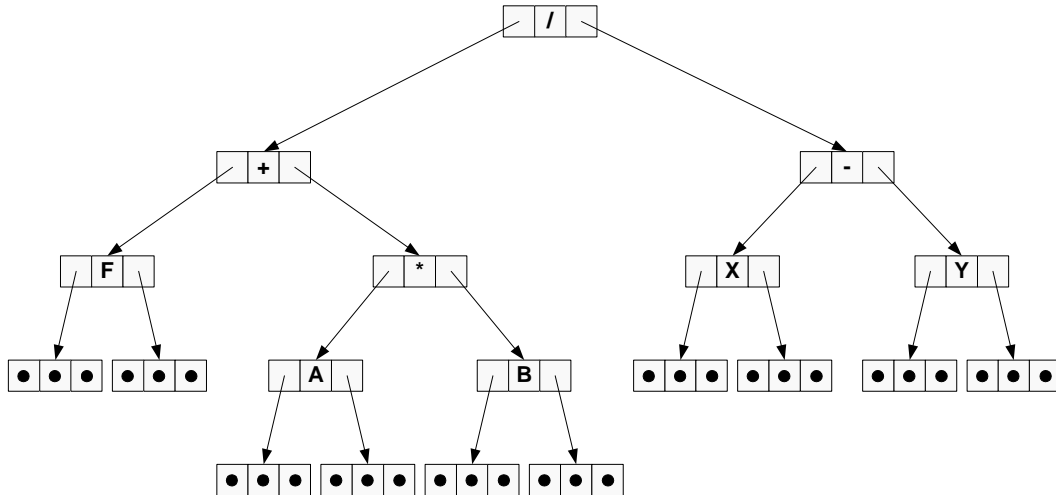
public interface Baum {

    public boolean empty ();           // liefert true, falls Baum leer ist
    public Baum left ();              // liefert linken Teilbaum
    public Baum right ();             // liefert rechten Teilbaum
    public Object value ();           // liefert Objekt in der Wurzel
    public void setValue(Object x);   // setzt x in die Wurzel
    public void setLeft(Baum b);      // haengt b als linken Teilbaum an
    public void setRight(Baum b);     // haengt b als rechten Teilbaum an
}

```

Konzept zur Implementation eines Baumes mit Verweisen

(Obacht: Aus technischen Gründen hat jedes Blatt zwei Verweise auf leere Bäume. Hierdurch vereinfachen sich gewisse Operationen auf dem später noch einzuführenden Suchbaum.)



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))

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

/** Klasse VerweisBaum mit einem Konstruktor und 7 Methoden.
 * Ein VerweisBaum besteht aus den Datenfeldern inhalt, links, rechts. */

public class VerweisBaum implements Baum{

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

    public VerweisBaum () { // konstruiert einen leeren Baum
        inhalt = null; // kein Inhalt
        links = null; // kein linker Sohn
        rechts = null;} // kein rechter Sohn

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

    public Object value () { // liefert Objekt in der Wurzel
        if (empty()) throw new
        RuntimeException("in VerweisBaum.value");
        return inhalt;}

    public Baum left () { // liefert linken Teilbaum
        if (empty()) throw new
        RuntimeException("in VerweisBaum.left");
        return links;}

    public Baum right () { // liefert rechten Teilbaum
        if (empty()) throw new
        RuntimeException("in VerweisBaum.right");
        return rechts;}

    public void setValue(Object x) { // setzt x in die Wurzel
        if (empty()) { // falls der Baum leer ist
            links = new VerweisBaum(); // lege ein Blatt an
            rechts= new VerweisBaum();
        }
        inhalt = x;} // setze x in Wurzel

    public void setLeft(Baum b) { // haengt b als linken Teilbaum an
        if (empty()) throw new
        RuntimeException("in VerweisBaum.setLeft");
        links = b;}

    public void setRight(Baum b) { // haengt b als rechten Teilbaum an
        if (empty()) throw new
        RuntimeException("in VerweisBaum.setRight");
        rechts = b;}
}
```

```
/****** 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 Knoteninhalte 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 Knoteninhalte 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 Knoteninhalte 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 VerweisKeller(); // 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 ArraySchlange(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 binären Baums mit Operanden in
 *  den Blättern und Operatoren in den inneren Knoten:
 */

public class TraverseTest {

    public static void main(String[] argv) {

        VerweisBaum a = new VerweisBaum(); a.setValue(new Character('A'));
        VerweisBaum b = new VerweisBaum(); b.setValue(new Character('B'));
        VerweisBaum m = new VerweisBaum(); m.setValue(new Character('*'));
        VerweisBaum f = new VerweisBaum(); f.setValue(new Character('F'));
        VerweisBaum p = new VerweisBaum(); p.setValue(new Character('+'));
        VerweisBaum x = new VerweisBaum(); x.setValue(new Character('X'));
        VerweisBaum y = new VerweisBaum(); y.setValue(new Character('Y'));
        VerweisBaum n = new VerweisBaum(); n.setValue(new Character('-'));
        VerweisBaum d = new VerweisBaum(); d.setValue(new Character('/'));
        m.setLeft(a); m.setRight(b);      p.setLeft(f); p.setRight(m);
        n.setLeft(x); n.setRight(y);      d.setLeft(p); d.setRight(n);

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

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

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

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

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

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

    }
}

```

```

/***** 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 {

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

        Baum b; // Hilfsbaum
        char c; // Zeichen

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

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

            c = ausdruck[i]; // besorge naechstes Zeichen
            b = new VerweisBaum(); // erzeuge leeren Baum
            b.setValue(new Character(c)); // mache daraus ein Blatt

            if (c=='+' || c=='-' || c=='*' || c=='/'){ // falls c ein Operator ist

                b.setRight((Baum)k.top()); k.pop(); // haenge rechten Sohn ein
                b.setLeft ((Baum)k.top()); k.pop(); // haenge linken Sohn ein

            }

            k.push(b); // lege Baum auf Keller
        }
        return (Baum)k.top(); // gib Ergebnisbaum 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();
    }
}

```

```

/***** PreorderTraverse.java *****/
import java.util.Enumeration;
/** Schrittweise Preordertraversierung eines Baumes mit Interface Enumeration */
public class PreorderTraverse implements Enumeration{

    private Keller k; // Keller zum Zwischenspeichern

    public PreorderTraverse(Baum wurzel) { // Konstruktor,
        k = new VerweisKeller(); // besorge Verweiskeller
        if (!wurzel.empty()) k.push(wurzel); // initialisiert Keller
    }

    public boolean hasMoreElements() { // liefert true
        return !k.empty(); // falls noch Elemente
    } // im Keller sind

    public Object nextElement() {
        Baum b = (Baum)k.top(); // hole Baum vom Keller
        k.pop(); // entferne Top-Element
        if (!b.right().empty()) k.push(b.right()); // lege rechten Sohn auf Keller
        if (!b.left().empty()) k.push(b.left()); // lege linken Sohn auf Keller
        return b.value(); // liefere Element ab
    }
}

/***** PostfixPreorderTest.java *****/
import AlgoTools.IO;
import java.util.Enumeration;

/** Klasse PostfixPreorderTest konstruiert unter Verwendung von PostfixBaumBau
    einen Baum aus dem eingelesenen String und gibt ihn in Preorder-Notation
    aus unter Verwendung von PreorderTraverse */
public class PostfixPreorderTest {

    public static void main (String [] argv) {

        char[]zeile=IO.readChars("Bitte Postfix-Ausdruck:");// lies Postfixausdruck
        Baum wurzel =PostfixBaumBau.postfixBaumBau(zeile); // konstruiere daraus
        // einen Baum

        IO.println("Umwandlung in Prefix-Notation:");
        PreorderTraverse pre = new PreorderTraverse(wurzel);// erzeuge Enumeration
        while (pre.hasMoreElements() { // solange Elemente da
            Character c = (Character) pre.nextElement(); // besorge Element
            IO.print(c); // gib Wert 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 .

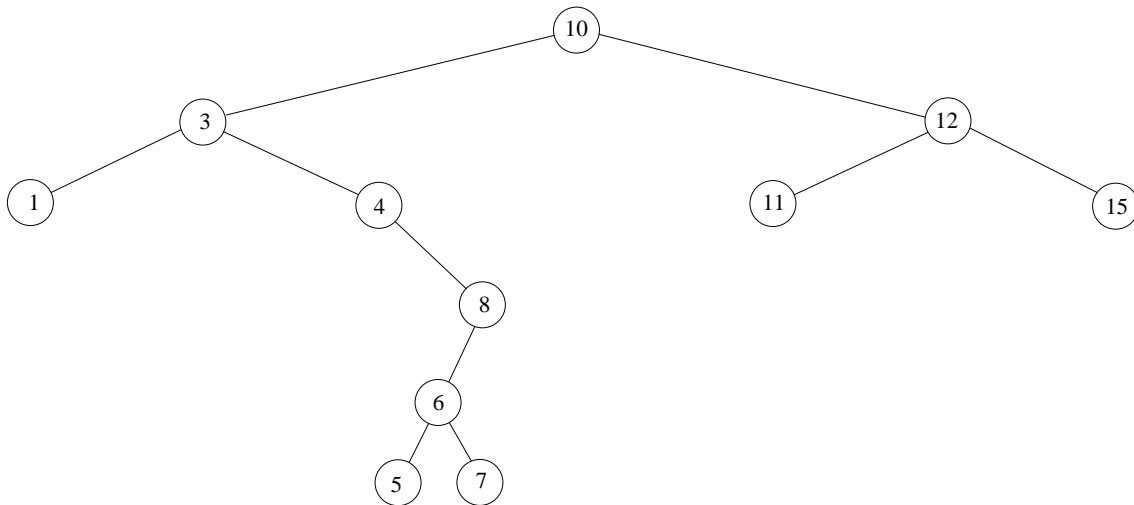
Der Aufwand aller Suchbaumoperationen ist proportional zur Anzahl der Knoten auf dem Wege von der Wurzel bis zu einem Blatt.

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



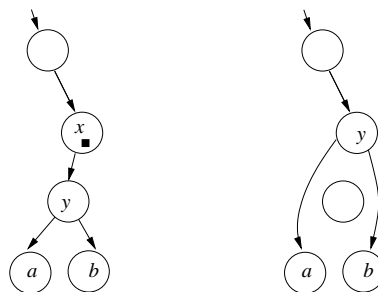
In den folgenden drei Grafiken wird mit einem gefüllten Kreis der NULL-Zeiger visualisiert und mit einem gefüllten Quadrat der Verweis auf einen leeren Baum (bestehend aus einem Knoten mit drei Nullzeigern).

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

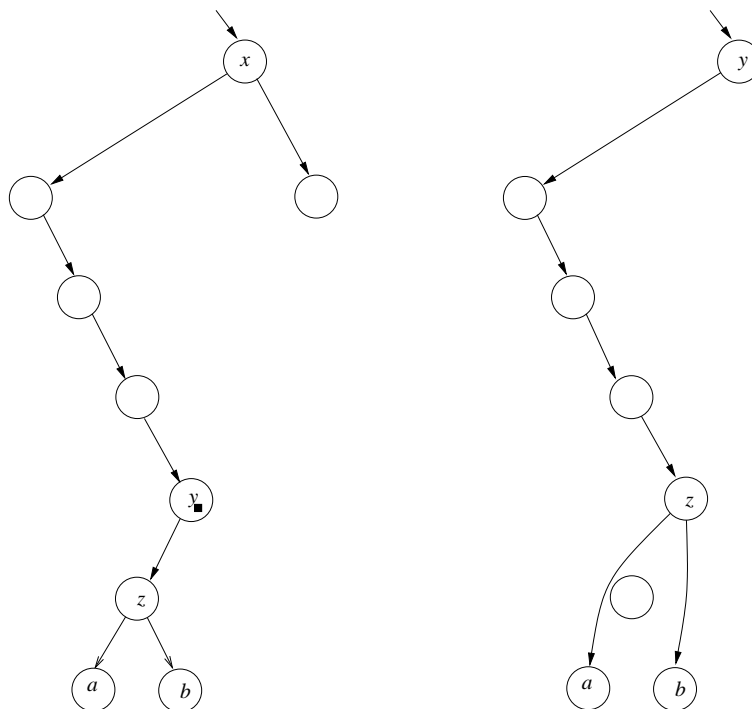
Löschen eines Knotens ohne Söhne



Löschen eines Knotens mit einem Sohn



Löschen eines Knotens mit zwei Söhnen




```

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

/** Implementation eines binären Suchbaums über Comparable-Objekten.
    Bereitgestellt werden die im Interface Menge angekuendigten Methoden
    lookup, insert und delete als öffentliche Methoden.

    Zur Implementation wird von allen dreien die private Methode find benutzt,
    die fuer die Navigation im Baum zustaendig ist.

    Die Methode delete verwendet zusaetzlich noch die private Methode findMax.
*/

public class SuchBaum extends VerweisBaum implements Menge {

    private SuchBaum find(Comparable x) { // liefert den Suchbaum
                                           // mit x in der Wurzel
                                           // (ggf. leerer Baum)
        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);
    }

    public Comparable lookup(Comparable x){ // Sucht x im SuchBaum
                                           // falls gefunden:
        return (Comparable)find(x).inhalt; // liefert Comparable-Object
                                           // null sonst
    }

    public boolean insert(Comparable x) { // fuegt x in SuchBaum ein
                                           // liefert true, falls erfolgreich
                                           // false sonst

        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; // setze Inhalt auf x
            s.links = new SuchBaum(); // neuer leerer SuchBaum links
            s.rechts = new SuchBaum(); // neuer leerer SuchBaum rechts
            return true;
        }
        else return false;
    }
}

```



```
public boolean delete(Comparable x) { // loescht x aus SuchBaum,
                                     // liefert true, falls erfolgreich
                                     // false sonst

    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 = // ermittele das Maximum
                ((SuchBaum)s.left()).findMax(); // im linken Teilbaum
            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;
    }
}

private SuchBaum findMax() { // sucht im nichtleeren SuchBaum
                             // liefert den SuchBaum
                             // mit dem Maximum in der Wurzel

    SuchBaum hilf = this;

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

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

/** Testet den SuchBaum mit Character-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 Character(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: ");

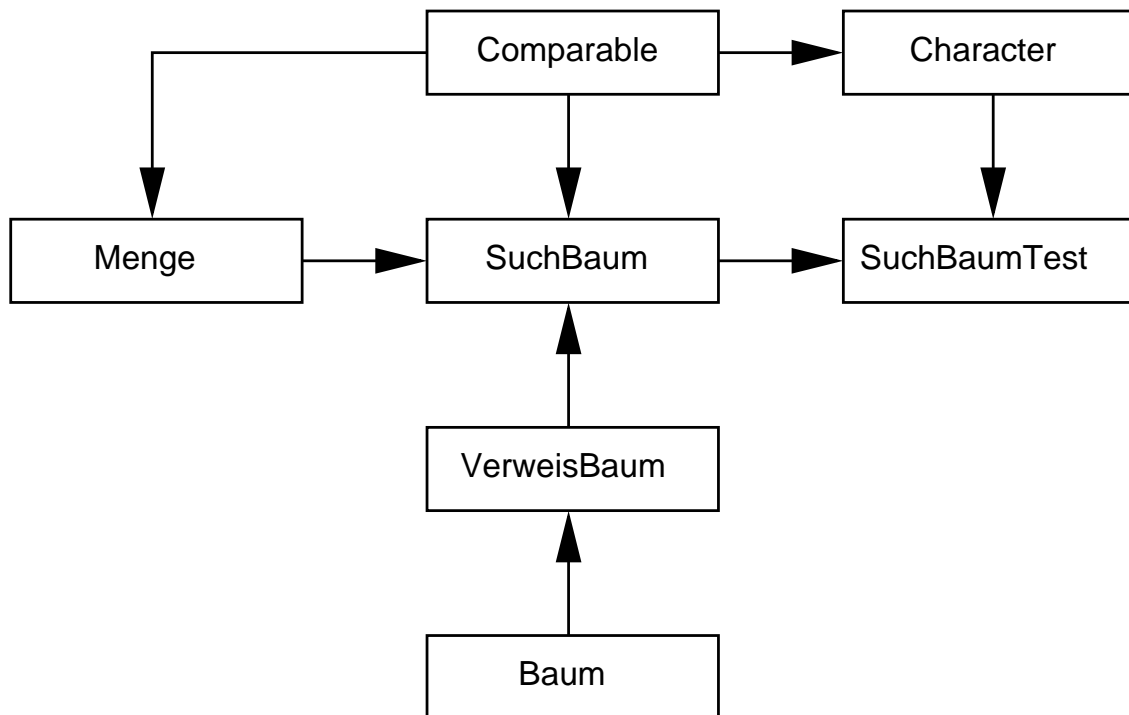
        for (int i=0; i<eingabe.length; i++) { // Elemente im SuchBaum suchen
            Comparable c = s.lookup(new Character(eingabe[i]));
            if(c == null)
                IO.println(eingabe[i] + " konnte nicht gefunden werden");
            else
                IO.println(c + " gefunden");
    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

        for (int i=0; i<eingabe.length; i++) { // Elemente im SuchBaum loeschen
            if (s.delete(new Character(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();
        }
    }
}
```

Abhängigkeiten

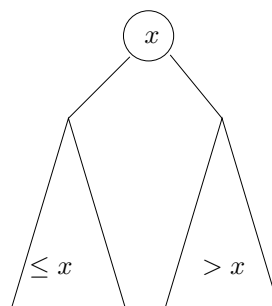
Folgende Grafik verdeutlicht die Abhängigkeiten der zum Suchen von Characters verwendeten Klassen und Interfaces:



Multimenge

Zur Verwaltung einer Multimenge (Elemente sind ggf. mehrfach vorhanden) kann ein Suchbaum wie folgt verwendet werden:

1. Möglichkeit: Elemente doppelt halten



2. Möglichkeit: Zähler im Knoten mitführen

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

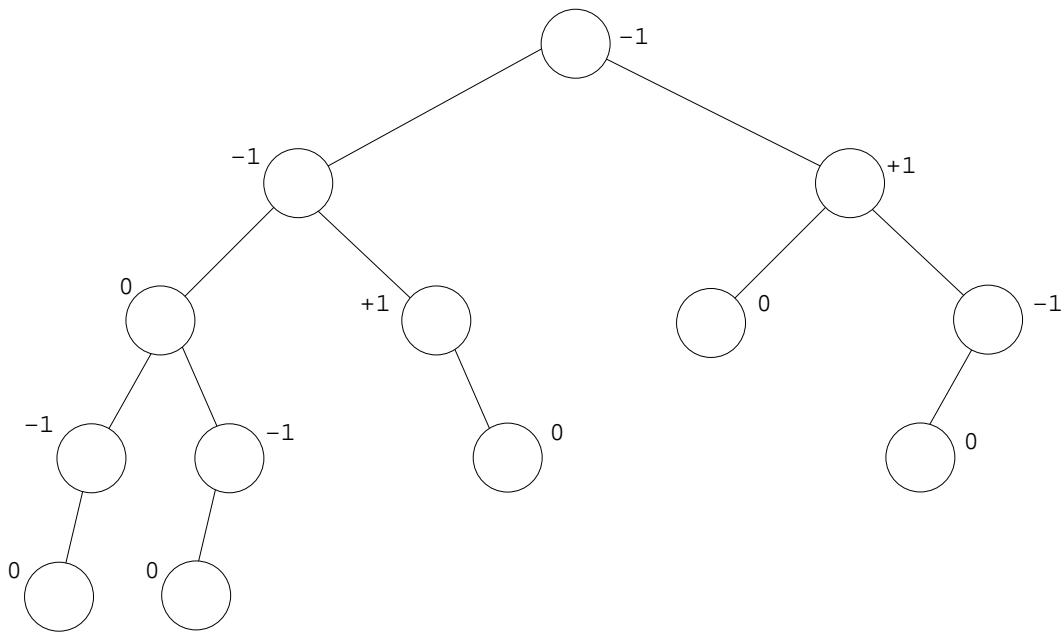
Beim Löschen: Zähler herunterzählen, sofern mehrfach da, sonst entfernen.

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 $bal(x) = \text{Höhe des rechten Teilbaums von } x \text{ minus Höhe des linken Teilbaums von } x$.

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.

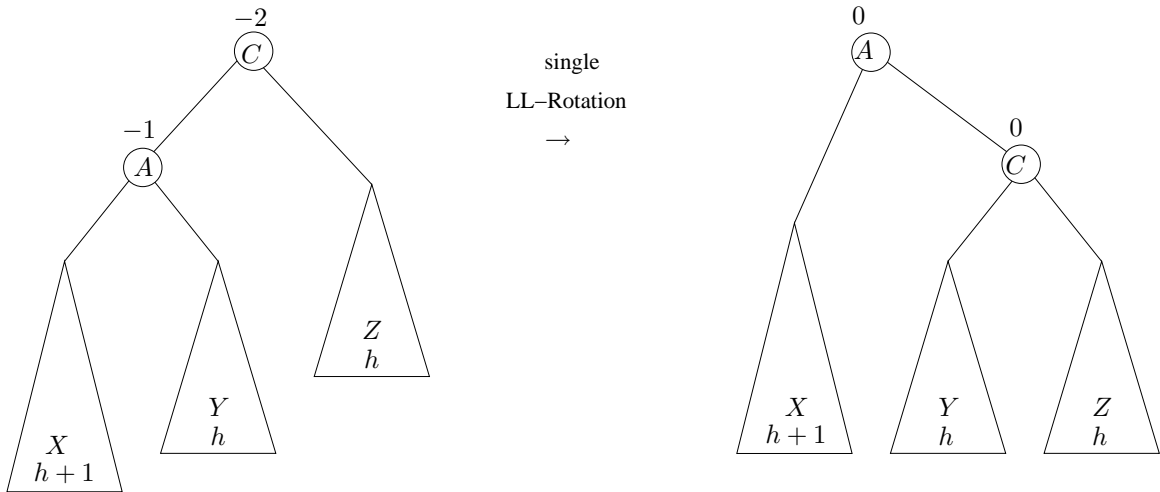
Während das Einfügen eines einzelnen Schlüssels *höchstens eine* Rotation erfordert, kann das Löschen eine Rotation für *jeden* Knoten entlang des Weges zur Wurzel verursachen.

Rotationen für AVL-Baum bei linksseitigem Übergewicht

Single LL-Rotation

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

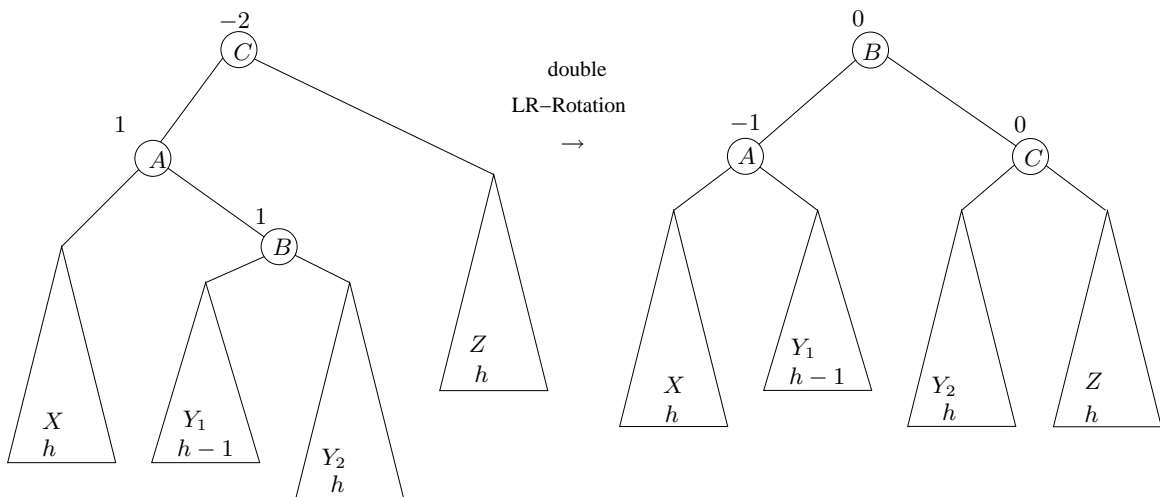
Bei Löschen im Teilbaum Z : Höhe des gesamten Baums vorher und nachher gleich oder nacher um eins kleiner.



Double LR-Rotation

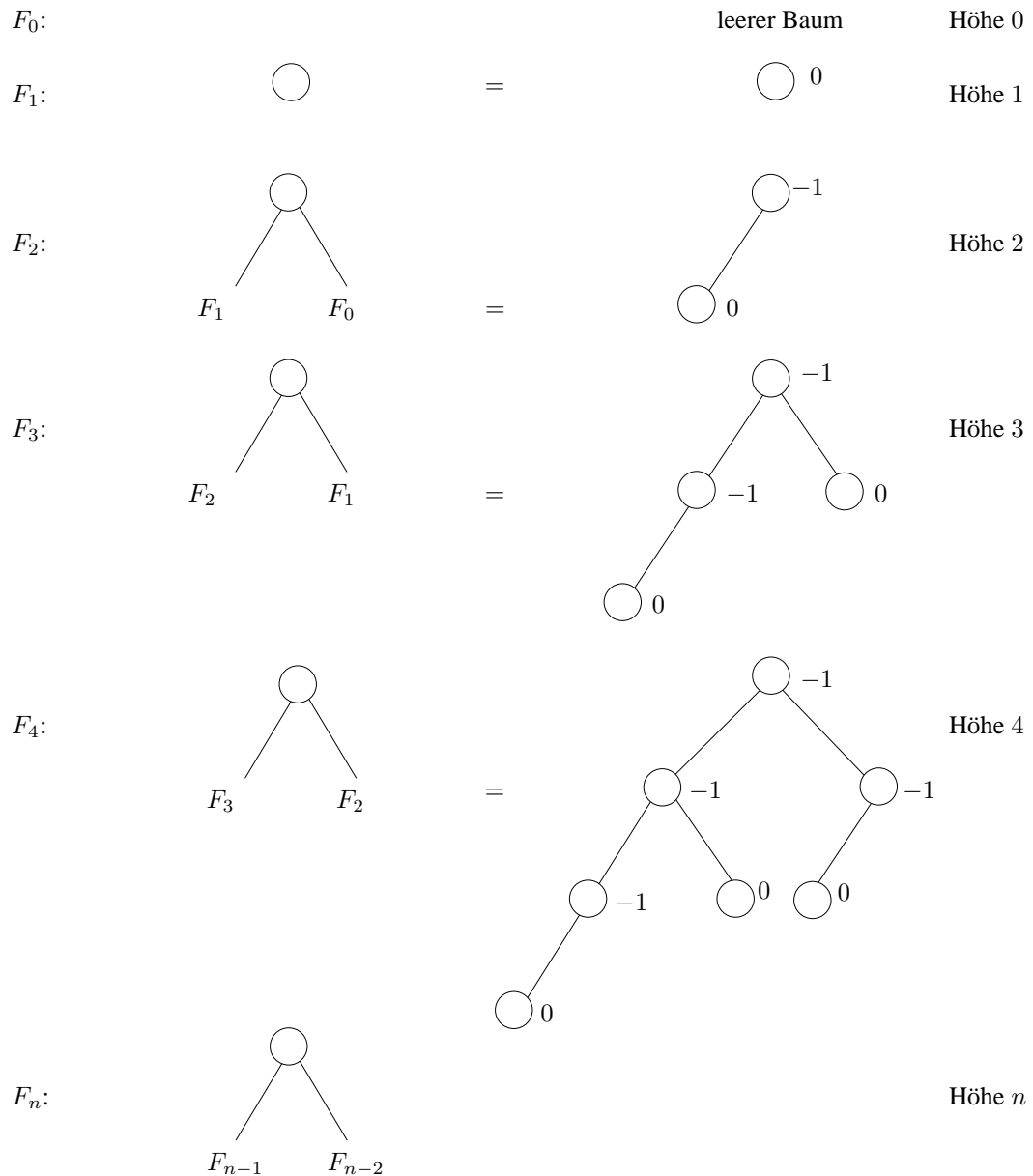
Bei Einfügen in Teilbaum Y_1 oder Y_2 : Höhe des gesamten Baums vorher und nachher gleich.

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, fuer jeden seiner Knoten unterscheiden sich
 * die Hoehen seiner beiden Teilbaeume maximal um eins.
 */

public class AVLBaum extends SuchBaum implements Menge {

    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 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.value();         // wird mit Wurzel
    rechts.setValue(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.value();          // wird mit Wurzel
    links.setValue(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 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 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
            if (links.empty()) {
                inhalt = null;
                links = null;
            }
            else {
                inhalt = links.value(); // ersetze Knoten durch linken Sohn
                links = links.left();  // Kein linker Sohn mehr
            }
            balance = 0; // Knoten ist Blatt
            s.unbal = true; // Hoehe hat sich geaendert
        } else if (links.empty()) { // Kein linker Sohn
            if (rechts.empty()) {
                inhalt = null;
                rechts = null;
            }
            else {
                inhalt = rechts.value(); // ersetze Knoten durch rechten Sohn
                rechts = rechts.right(); // 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 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();
    }
}
```

```
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
        if (links.empty()) {
            inhalt = null;
            links = null;
        }
        else {
            inhalt = links.value(); // ersetze Knoten durch linken Sohn.
            links = links.left(); // Kein linker Sohn mehr
        }
        balance = 0; // Knoten ist Blatt
        s.unbal = true; // Teilbaum wurde kuerzer
    }
    return ersatz; // Gib Ersatz-Objekt zurueck
}
}
```

```
/****** 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 Character(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 Character(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): ");
        }

    }

    /** Der AVL-Baum wird liegend mit Werten und Balancen ausgegeben. */
    public static void printAVLBaum(Baum b, int tiefe) {
        if (! b.empty()) {
            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.

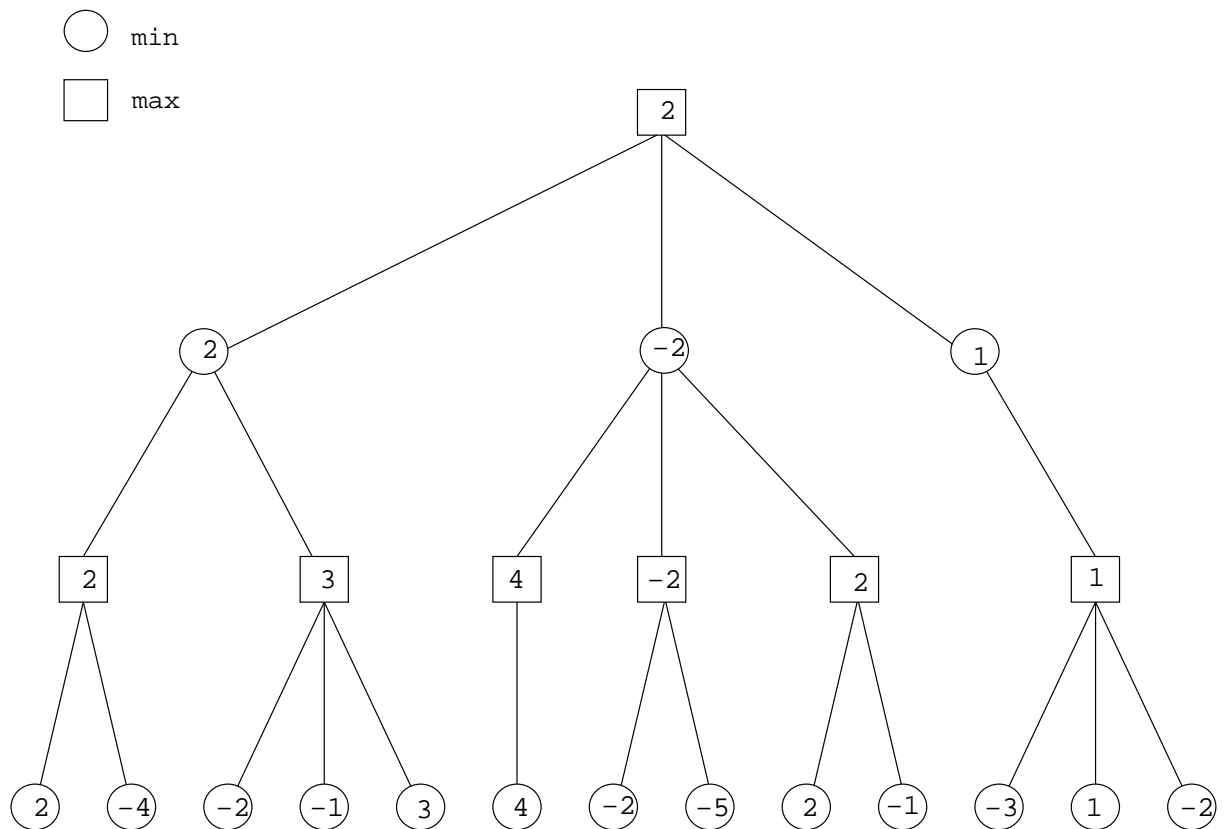
Der Wert eines Blattes wird bestimmt durch eine statische Stellungsbewertung.

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




```

/***** 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

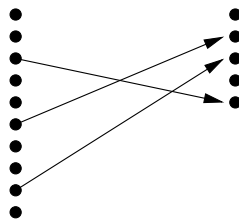
Hashing

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



mögliche Elemente

N Adressen

Gegeben Adressen von 0 bis $N - 1$.

Sei x ein beliebiges Objekt. Dann ist

```
String s = x.toString();
```

seine Stringrepräsentation.

Sei $x = x_{n-1}x_{n-2} \dots x_1x_0$ ein String, dann ist

$$f(x) = \left(\sum_{i=0}^{n-1} x_i \right) \text{MOD } N$$

eine Hashfunktion.

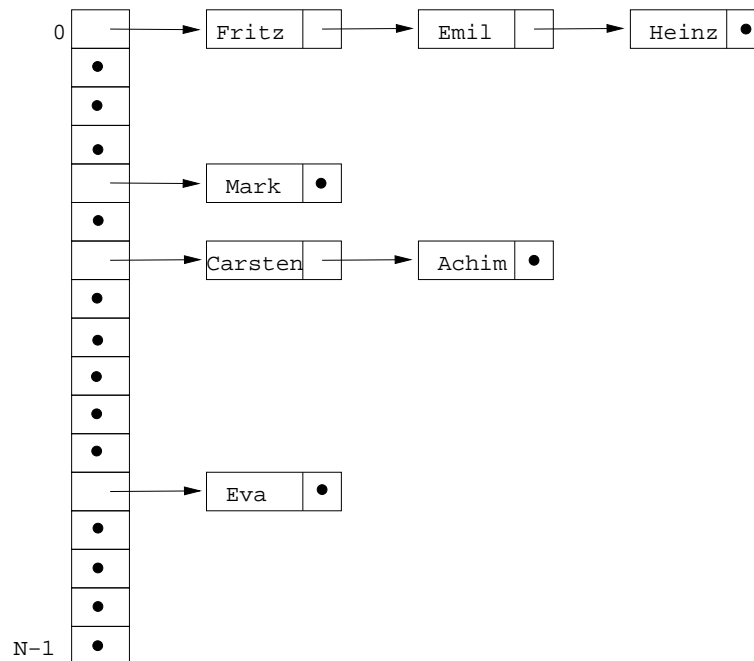
Gilt: $f(x) = f(y)$, so liegt eine *Kollision* vor, die bei *offenem* und *geschlossenem* Hashing unterschiedlich behandelt wird.

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 Zuständen
                        // LEER, BELEGT und GELOESCHT
```

Falls $y = f(x)$ schon belegt ist, so suche für x einen Alternativplatz.

$y + 1, y + 2, y + 3, y + 4, \dots$ *lineares Sondieren*

$y + 1, y + 4, y + 9, y + 16, \dots$ *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.:

gesucht: $f : \{\text{braun, rot, blau, violett, türkis}\} \rightarrow \mathbb{N}$

Länge(w)	=	5	3	4	7	6
Länge(w) - 3	=	2	0	1	4	3

$\Rightarrow f(w) = \text{Länge}(w) - 3 \in [0..4]$ ist perfekte Hashfunktion.

```

/***** OfHashing.java *****/

/** Implementation des Interface Menge durch Array von Listen */

public class OfHashing implements Menge {

    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 VerweisListe(); // 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
            // und Objekt noch nicht gefunden
            ((Comparable)l.elem()).compareTo(x) != 0)
            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 Menge
 * durch ein geschlossenes Hashing mit einem Array von Objekten.
 */

public class GeHashing implements Menge {

    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 eingetragenen 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 erhoehrt.

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 String als Eintraege in der HashTable
 *  und ein Objekt der Klasse GeHashing als Instanz des Interface Menge
 */
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(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(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(s))
                IO.println(s + " wurde geloesch");
            else IO.println(s + " wurde nicht geloesch");
            IO.print(h);
            s = IO.readString("Bitte String: ");
        }
    }
}
```

Laufzeit bei geschlossenem Hashing

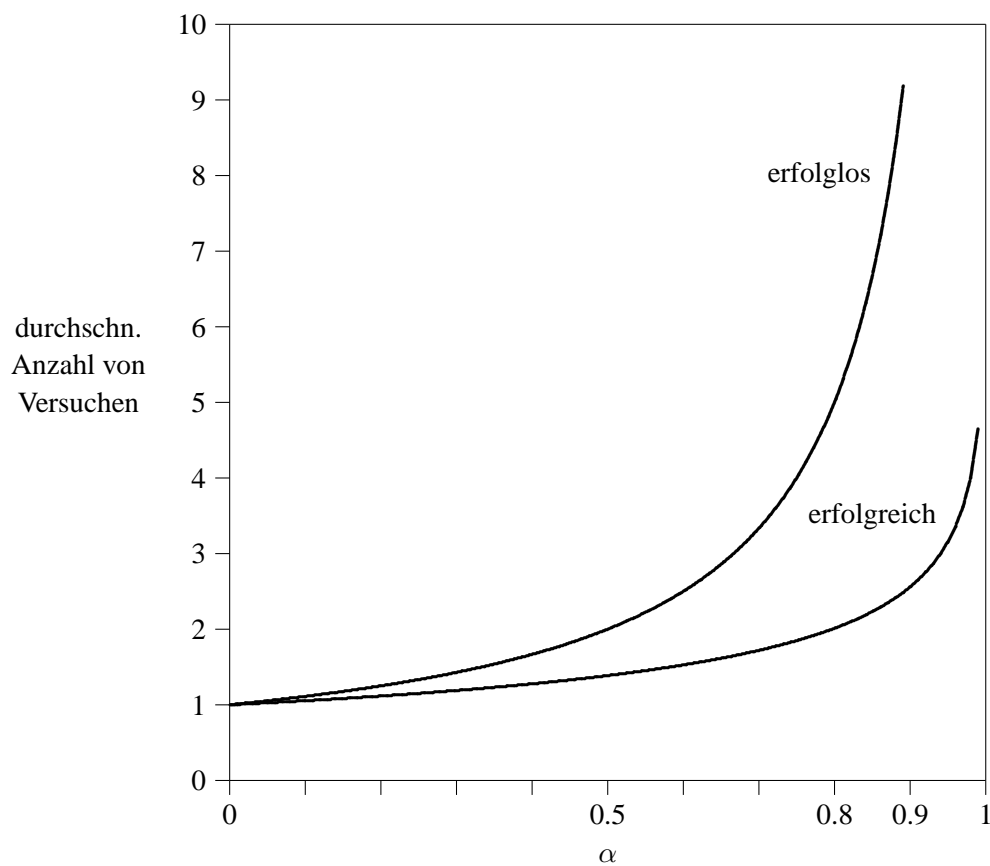
Sei n die Anzahl der in der Hashtabelle zur Zeit gespeicherten Objekte, sei N die Anzahl der möglichen Speicherpositionen.

Sei $\alpha = \frac{n}{N} \leq 1$ 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 $\alpha = 0.8$

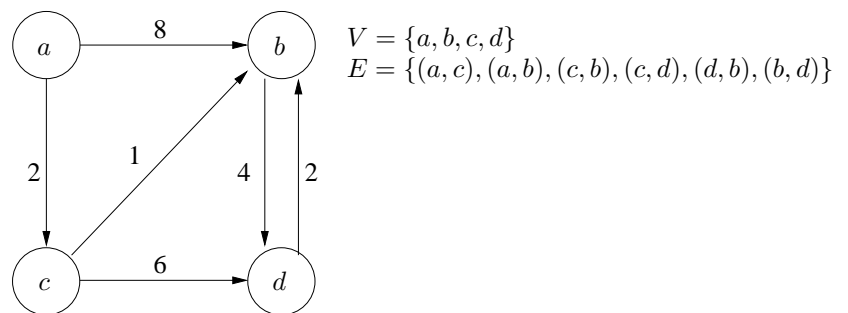
d.h., in 2 Schritten wird von 1.000.000 Elementen aus einer 1.250.000 großen Tabelle das richtige gefunden. (Zum Vergleich: der AVL-Baum benötigt dafür etwa 20 Vergleiche.)



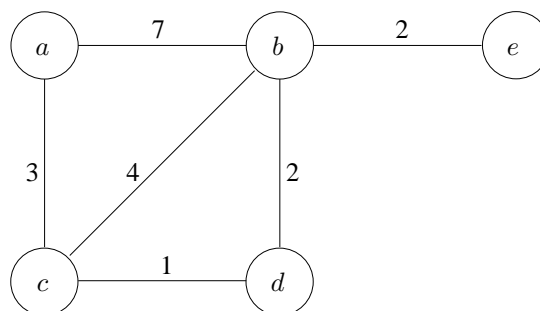
Kapitel 11

Graphen

Ein *gerichteter Graph* $G = (V, E)$ besteht aus *Knotenmenge* V
und *Kantenmenge* $E \subseteq V \times V$

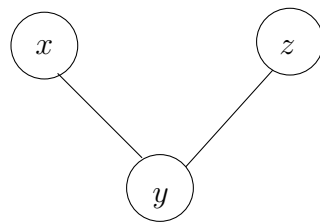
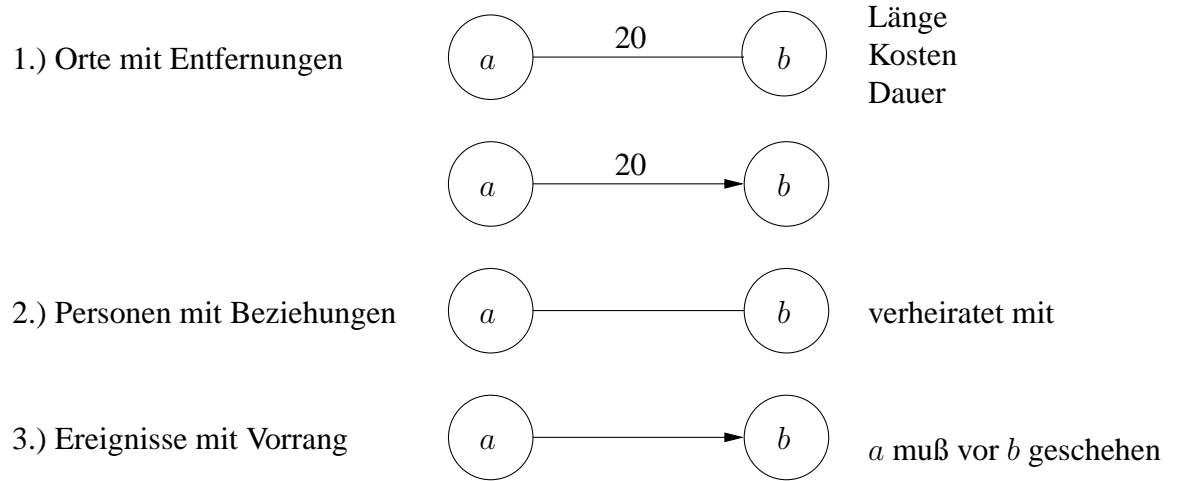


Ein *ungerichteter Graph* $G = (V, E)$ besteht aus *Knotenmenge* V
und *Kantenmenge* $E \subseteq P_2(V) = 2\text{-elem. Teilmengen von } V$.

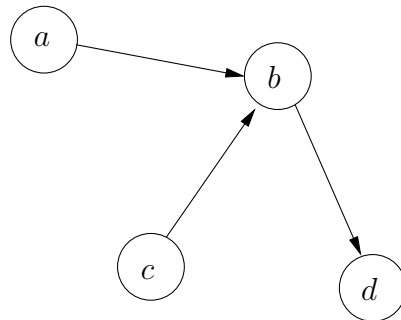


Kanten können gewichtet sein durch eine *Kostenfunktion* $c : E \rightarrow \mathbb{Z}$.

Mit Graphen können zwischen Objekten ($\hat{=}$ Knoten) binäre Beziehungen ($\hat{=}$ Kanten) modelliert werden.



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 149 ergibt sich:

Index	Knoten
0	<i>a</i>
1	<i>b</i>
2	<i>c</i>
3	<i>d</i>

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 = $O(|V|^2)$.

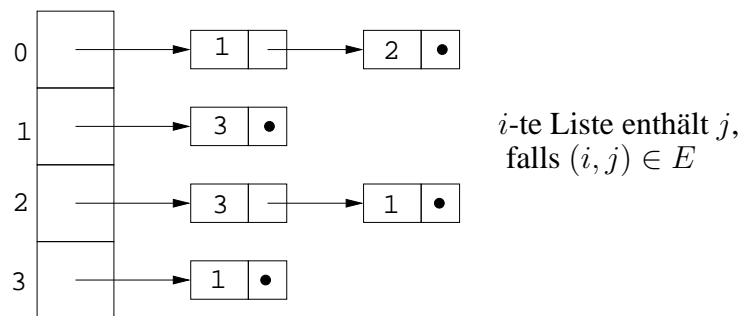
Direkter Zugriff auf Kante (i, j) in konstanter Zeit möglich.

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



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 $f: V \rightarrow \mathbb{N}$ zu finden, die mit den Kantengerichtungen 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.


```
/* ***** 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 VerweisListe[n]; // besorge Platz fuer Array mit n Listen
        for (int i=0; i < n; i++) // besorge n mal
            g[i] = new VerweisListe(); // 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
    }
}
```

```

/***** 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 ArraySchlange(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);
        }

    }

}
```