

Algorithmen

Vorlesung im WS 2012/2013

Oliver Vornberger

Institut für Informatik
Fachbereich Mathematik/Informatik
Universität Osnabrück

Literatur

- David Flanagan:
Java In A Nutshell (5. Auflage, für Java 5.0), O'Reilly, 2005
- Guido Krüger:
Handbuch der Java-Programmierung, Addison-Wesley, 2011
Online-Fassung unter <http://www.javabuch.de>
- Robert Sedgewick:
Algorithmen in Java, Teil 1-4, Addison Wesley, 2003
- Mark Allen Weiss:
Data Structures & Problem Solving Using Java, Addison Wesley, 2009
- Christian Ullenboom:
Java ist auch eine Insel, 10. Auflage, Galileo Computing, 2011,
Onlinefassung: <http://openbook.galileocomputing.de/javainsel/>
- Robert Sedgewick, Kevin Wayne:
Einführung in die Programmierung mit Java, Pearson, 2011

Danksagung

Ich danke ...

- ... Herrn Frank Lohmeyer für die Bereitstellung des Pakets `AlgoTools` von *Java*-Klassen zur vereinfachten Ein- und Ausgabe sowie für die Implementation eines *Java-Applets* zur Simulation der Beispiel-Programme im WWW-Browser.
- ... Frank Thiesing, Olaf Müller, Ralf Kunze, Dorothee Langfeld, Patrick Fox, Nicolas Neubauer, Mathias Menninghaus und Sebastian Büscher für die inhaltliche Mitarbeit.
- ... 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 2012

Olivier vonberg

Inhaltsverzeichnis

1	Einführung	11
1.1	Informatik	11
1.2	Algorithmus, Programm, Prozess	12
1.3	Anweisungen, Ablaufprotokoll	12
2	Java	15
2.1	Sprachmerkmale	15
2.2	Variablen	16
2.3	Kontrollstrukturen	16
2.4	Einfache Datentypen	22
2.4.1	Ganze Zahlen (byte, short, int, long)	22
2.4.2	Gleitkommazahlen (float, double)	25
2.4.3	Boolean (boolean)	29
2.4.4	Charakter (char)	30
2.4.5	Typumwandlung	32
2.4.6	Konstanten	33
3	Felder	35
3.1	Feld von Ziffern	36
3.2	Feld von Daten	37
3.3	Feld von Zeichen	38
3.4	Feld von Wahrheitswerten	39
3.5	Feld von Indizes	40
3.6	Feld von Zuständen	41
3.7	Lineare und binäre Suche	43
4	Klassenmethoden	45

4.1	Methodenaufrufe	46
4.2	Parameterübergabe an Arrays	47
4.3	Sichtbarkeit	48
4.4	Fehlerbehandlung	49
5	Rekursion	51
5.1	Fakultät, Potenzieren, Fibonacci, GGT	52
5.2	Türme von Hanoi	53
6	Komplexität, Verifikation, Terminierung	55
6.1	O-Notation	55
6.2	Korrektheit und Terminierung	59
6.3	Halteproblem	62
7	Sortieren	63
7.1	SelectionSort	64
7.2	Bubblesort	65
7.3	MergeSort	66
7.4	QuickSort	69
7.5	Bestimmung des Medians	71
7.6	HeapSort	73
7.7	Zusammenfassung von Laufzeit und Platzbedarf	77
7.8	Untere Schranke für Sortieren durch Vergleichen	78
7.9	BucketSort	79
8	Objektorientierte Programmierung	81
8.1	Sichtbarkeit von Datenfeldern	81
8.2	Erste Beispiele	83
8.3	Binden	87
8.4	Referenzen	89
8.5	Wrapperklassen	90
9	Abstrakte Datentypen	93
9.1	Liste	93
9.2	Keller	97
9.3	Schlange	104

9.4	Baum	107
9.5	Suchbaum	116
9.6	AVL-Baum	123
10	Hashing	135
10.1	Offenes Hashing	136
10.2	Geschlossenes Hashing	136
11	Java Collection Framework	141
11.1	Collection	141
11.2	Map	143
12	Graphen	145
12.1	Implementation von Graphen	147
12.2	Graphalgorithmen für Adjazenzmatrizen	149
12.3	Implementation für gerichtete Graphen durch Adjazenzlisten	151
12.4	Traversieren von Graphen	158
12.5	Topologisches Sortieren	159
12.6	Kürzeste Wege	161
12.7	Hamiltonkreis	163
12.8	Maximaler Fluss	165
12.9	Implementation für ungerichtete Graphen durch Adjazenzlisten	168
12.10	Minimaler Spannbaum	171
12.11	Matching	175
12.12	Chinese Postman	176

Verzeichnis der Java-Programme

Collatz.java	15
Bedingung.java	17
Fall.java	18
Schleife.java	19
Fakultaet.java	20
GGT.java	21
Ueberlauf.java	24
Gleitkomma.java	28
Zeichen.java	31
Umwandlung.java	32
Konstanten.java	33
Feld.java	35
Ziffern.java	36
Matrix.java	37
Zeichenkette.java	38
Sieb.java	39
ArrayAbzaehltreim.java	40
Automat.java	42
Suche.java	43
Methoden.java	46
Parameter.java	47
f.java	48
Rekursion.java	52
Hanoi.java	53

SelectionSort.java	64
BubbleSort.java	65
Merge.java	66
SortTest.java	68
QuickSort.java	69
QuickSortTest.java	71
HeapSort.java	75
BucketSort.java	79
Datum.java	83
DatumTest.java	83
Person.java	84
PersonTest.java	84
Student.java	85
StudentTest.java	85
PersonStudentTest.java	86
Kind.java	89
VerweisAbzaehltreim.java	89
Liste.java	94
Eintrag.java	94
VerweisListe.java	95
VerweisListeTest.java	96
Keller.java	97
VerweisKeller.java	99
Reverse.java	100
Klammer.java	101
CharKeller.java	102
VerweisCharKeller.java	102
Postfix.java	102
Schlange.java	104
ArraySchlange.java	105
ArraySchlangeTest.java	106
Baum.java	107
Knoten.java	108

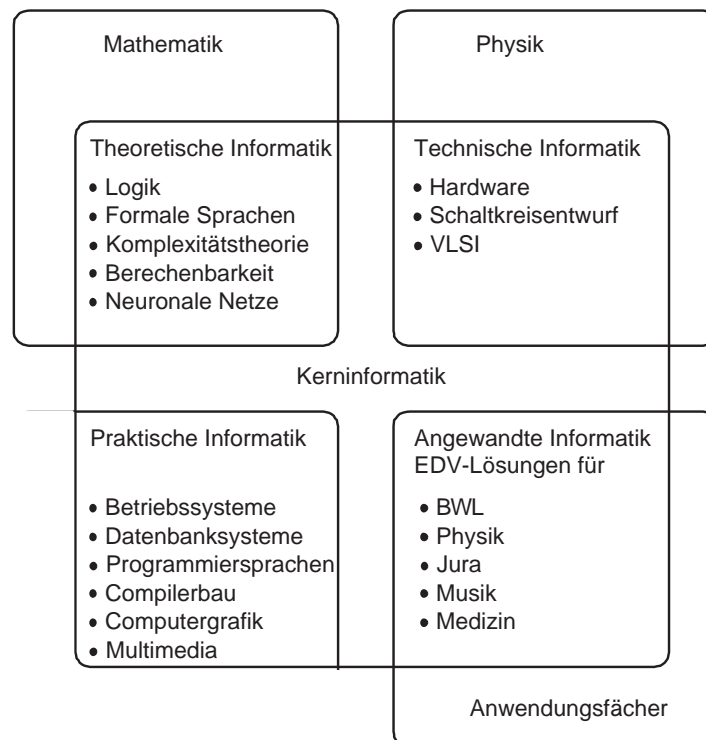
VerweisBaum.java	109
Traverse.java	110
TiefenSuche.java	111
BreitenSuche.java	112
TraverseTest.java	113
PostfixBaumBau.java	114
PreorderTraverse.java	115
PostfixPreorderTest.java	115
Menge.java	118
SuchBaum.java	119
SuchBaumTest.java	121
AVLKnoten.java	126
AVLBaum.java	126
AVLBaumTest.java	134
Floyd.java	149
FloydTest.java	150
Vertex.java	151
Edge.java	151
Graph.java	153
GraphIO.java	154
Result.java	155
GraphTest.java	156
GraphTraverse.java	158
TopoSort.java	160
Dijkstra.java	162
Hamilton.java	164
UndiVertex.java	168
UndiEdge.java	168
UndiGraph.java	169
UndiGraphIO.java	170
Kruskal.java	173
KruskalTest.java	174

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	011011101110110111

Ein für den Compiler formulierter Algorithmus heißt *Programm*.

Ein Programm in Ausführung heißt *Prozess*.

1.3 Anweisungen, Ablaufprotokoll

Elementare Anweisungen

Beispiel: teile x durch 2
erhöhe y um 1

Strukturierte Anweisungen

enthalten Kontrollstruktur, Bedingung, Teilanweisungen.

Beispiele:

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

```

abheben
waehlen
SOLANGE besetzt ist TUE
    auflegen
    abheben
    waehlen

```

Beispiel für einen Algorithmus in umgangssprachlicher Form

```

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

```

Ablaufprotokoll (trace)

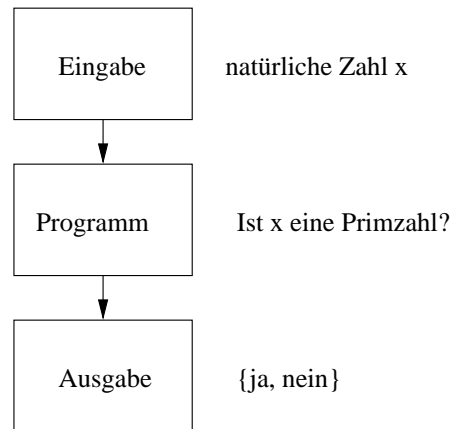
Zeile	x	z
	undef.	undef.
1	3	undef.
2	3	0
3	10	0
4	10	1
3	5	1
4	5	2
3	16	2
4	16	3
3	8	3
4	8	4
3	4	4
4	4	5
3	2	5
4	2	6
3	1	6
4	1	7
5	Ausgabe	7

Programm iteriert Collatz-Funktion

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

$f(x)$ = 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; // erhoehe 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, 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

        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++) {           // drucke 10 mal
            IO.print(i, 6);                // Zahl i auf 6 Stellen
            IO.print(" zum Quadrat = ");  // das Wort zum Quadrat
            IO.println(i*i, 6);           // das Quadrat von i auf 6 Stellen
        }
    }
}
```

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

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

public class Fakultaet {

    public static void main (String [] argv) {

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

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

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

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

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

```

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

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

public class GGT {

    public static void main (String [] argv) {

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

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

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

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

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

```

2.4 Einfache Datentypen

Der Datentyp legt fest:

- Wertebereich,
- Operationen,
- Konstantenbezeichner

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	
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.

```

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

/** Integer-Ueberlauf */

public class Ueberlauf {

    public static void main (String [] argv) {

        int 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 !
    }
}

```


2.4.2 Gleitkommazahlen (float, double)

Gleitkommazahlen werden durch Vorzeichen, Mantisse und Exponent beschrieben und erreichen damit deutlich größere Absolutwerte als Integerzahlen und können auch gebrochene Zahlen codieren. Genauer: Für eine gegebene Zahl x werden Mantisse m und Exponent e gesucht mit der Eigenschaft

$$x = m \cdot 2^e$$

Das heißt: Der Wert der Gleitkommazahl ergibt sich aus dem Produkt von Mantisse und der Zweierpotenz des Exponenten. Da es für ein gegebenes x mehrere zueinander passende Mantissen und Exponenten gibt, wählt man die normalisierte Form mit einer Mantisse $1 \leq m < 2$. Da eine normalisierte Mantisse m immer mit einer 1 vor dem Komma beginnt, reicht es aus, ihren Nachkommawert als reduzierte Mantisse $f = m - 1.0$ abzuspeichern.

Java verwendet zur Kodierung von Vorzeichen, Mantisse und Exponent den IEEE-Standard 754-1985, den wir hier aus didaktischen Gründen etwas vereinfacht darstellen.

Codierung

Bei einer Codierung für 32 Bits (float) werden für das Vorzeichen 1 Bit, für den Exponenten 8 Bits und für die reduzierte Mantisse 23 Bits vorgesehen.

X	XXXXXXXX	XXXXXXXXXXXXXXXXXXXXXXXXXXXX
Vorzeichen s	Exponent e	reduzierte Mantisse f

Die Codierung des Vorzeichens geschieht über 0 = positiv und 1 = negativ, die Codierung des ganzzahligen Exponenten erfolgt im 2-er Komplement, die Codierung der nichtganzzahligen Mantisse erfolgt als Dualzahl mit Nachkommastellen. Hierbei werden die Dualziffern nach dem Komma gewichtet mit 2er-Potenzen der Form 2^i mit negativen i . D.h.

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

Zu gegebenem x lässt sich eine normalisierte Mantisse und der dazu passende Exponent wie folgt finden:

Bestimme die größte 2-er Potenz 2^e mit $2^e \leq x$. Setze $m = x/2^e$. Offenbar gilt

$$x = m \cdot 2^e \text{ mit } 1 \leq m < 2.$$

Zur Bestimmung der k Nachkommastellen der reduzierten Mantisse $f := m - 1.0$ eignet sich folgender Algorithmus

```
for (i = 0; i < k; 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. Offenbar gilt

$$13.5 = 1.6875 \cdot 2^3$$

Als Codierung ergibt sich

$$\begin{aligned} s &= 0 \\ e &= 3 \\ m &= (13.5)/2^3 = 1.6875 \\ f &= 1.6875 - 1.0 = 0.6875 = \frac{1}{2} + \frac{1}{8} + \frac{1}{16} \end{aligned}$$

0	00000011	101100000000000000000000
Vorzeichen	Exponent	reduzierte Mantisse

Für die Spezialfälle wird vereinbart: Das vorzeichenbehaftete Unendlich ($\pm\infty$) wird durch den maximal möglichen Exponent $e = +127$ kodiert. Die vorzeichenbehaftete Null wird durch den minimal möglichen Exponent $e = -128$ kodiert.

Die größte darstellbare positive Zahl im vereinfachten float-Format liegt knapp unter

$$2 \cdot 2^{126} = 2^{127} \approx 10^{38}$$

0	01111110	111111111111111111111111
Vorzeichen	Exponent	reduzierte Mantisse

Die kleinste darstellbare positive Zahl im vereinfachten float-Format lautet

$$1.0 \cdot 2^{-127} = 2^{-127} \approx 10^{-38}$$

0	10000001	000000000000000000000000
Vorzeichen	Exponent	reduzierte Mantisse

Bei der Codierung für 64 Bits (`double`) sind für das Vorzeichen 1 Bit, für den Exponenten 11 Bits und für die reduzierte Mantisse 52 Bits vorgesehen.

X	XXXXXXXXXXXX	XX
Vorzeichen s	Exponent e	reduzierte Mantisse f

$$\text{Wert} = \begin{cases} (-1)^s \cdot 2^e \cdot 1.f & \text{falls } -1024 < e < 1023 \\ \pm\infty & \text{falls } e = 1023 \\ \pm 0 & \text{falls } e = -1024 \end{cases}$$

Damit liegt die größte darstellbare positive Zahl im vereinfachten double-Format knapp unter

$$2 \cdot 2^{1022} = 2^{1023} \approx 10^{308}$$

Die kleinste darstellbare positive Zahl im vereinfachten double-Format lautet

$$1.0 \cdot 2^{-1023} = 2^{-1023} \approx 10^{-308}$$

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:

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

```

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

import AlgoTools.IO;

/** Gleitkommaoperationen
 */

public class Gleitkomma {

    public static void main (String [] argv) {

        double summe, summand, pi, nenner, x; // fuenf 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

        // Pi/4 = 1 -1/3 +1/5 -1/7 ...
        // Pi = 1 -4/3 +4/5 -4/7 ...
        summe = 4.0; // setze Naehderung auf 4
        nenner = 1.0; // setze nenner auf 1
        IO.println("Naehderung von Pi :"); // gib Naehderungswerte aus
        for (int i=0; i <= 10000; i++) { // tue 10000 mal
            nenner += 2.0; // erhoehe nenner um 2
            summand = 4.0 / nenner; // bestimme naechsten summand
            if (i%2 == 0) summand = -summand; // abwechselnd positiv/negativ
            summe = summe + summand; // addiere summand auf Naehderung
            IO.println(summe, 22, 16); // drucke Naehderung auf insgesamt
        } // 22 Stellen, 16 Nachkommastellen

        IO.println("Ueberlaufdemo :");
        x = 2.0; // setze x auf 1
        while (x < Double.MAX_VALUE) { // solange kleiner als Maximum
            x = x * x; // quadriere x
            IO.println(x,30); // drucke in wiss. Notation
        } // druckt als letzten Wert Infinity

        IO.println("Teilung durch 0:");
        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	true

Verkürzte Auswertung erfolgt von links nach rechts und bricht frühestmöglich 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
\a	bell	7
\b	backspace	8
\t	horizontal tab	9
\n	newline	10
\v	vertical tab	11
\f	form feed	12
\r	carriage return	13
\101	Buchstabe A in Oktal-Schreibweise	65
\u0041	Buchstabe A in Hexadezimal-Schreibweise	65
\"	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++) {
            IO.print(i,7);
            IO.print( " " + (char) i );
            if (i % 8 == 0) IO.println();
        }
        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 != '\n' ); // bis ein Newline kommt
    }
}

```

33 !	34 "	35 #	36 \$	37 %	38 &	39 '	40 (
41)	42 *	43 +	44 ,	45 -	46 .	47 /	48 0
49 1	50 2	51 3	52 4	53 5	54 6	55 7	56 8
57 9	58 :	59 ;	60 <	61 =	62 >	63 ?	64 @
65 A	66 B	67 C	68 D	69 E	70 F	71 G	72 H
73 I	74 J	75 K	76 L	77 M	78 N	79 O	80 P
81 Q	82 R	83 S	84 T	85 U	86 V	87 W	88 X
89 Y	90 Z	91 [92 \	93]	94 ^	95 _	96 `
97 a	98 b	99 c	100 d	101 e	102 f	103 g	104 h
105 i	106 j	107 k	108 l	109 m	110 n	111 o	112 p
113 q	114 r	115 s	116 t	117 u	118 v	119 w	120 x
121 y	122 z	123 {	124	125 }	126 ~	127	128
161 ¡	162 ¢	163 £	164 ¤	165 ¥	166 ¦	167 §	168 ¨
169 ©	170 ª	171 «	172 ¬	173 ®	174 ¯	175 ¯	176 °
177 ±	178 ²	179 ³	180 ´	181 µ	182 ¶	183 ·	184 ¸
185 ¹	186 º	187 »	188 ¼	189 ½	190 ¾	191 ¿	192 À
193 Á	194 Â	195 Ã	196 Ä	197 Å	198 Æ	199 Ç	200 È
201 É	202 Ê	203 Ë	204 Ì	205 Í	206 Î	207 Ï	208 Ð
209 Ñ	210 Ò	211 Ó	212 Ô	213 Õ	214 Ö	215 ×	216 Ø
217 Ù	218 Ú	219 Û	220 Ü	221 Ý	222 Þ	223 ß	224 à
225 á	226 â	227 ã	228 ä	229 å	230 æ	231 ç	232 è
233 é	234 ê	235 ë	236 ì	237 í	238 î	239 ï	240 ð
241 ñ	242 ò	243 ó	244 ô	245 õ	246 ö	247 ÷	248 ø
249 ù	250 ú	251 û	252 ü	253 ý	254 þ	255 ÿ	

Teile der Ausgabe vom Programm zeichen.java

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.

```

/***** 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 = 3.141592653589793; // 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    = (char) 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/360.0*2*PI);     // 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[5];         // besorge Platz fuer 5 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
    }
}
```

Beispiel: $wert(4728) = (((4*10)+7)*10+2)*10+8$

3.2 Feld von Daten

```

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

/** Multiplikation zweier NxN-Matrizen
 *
 *      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;

        int[][] a = {{ 1, 2, 3, 4},          // initialisiere Matrix A
                    { 5, 6, 7, 8},
                    { 9, 10, 11, 12},
                    { 13, 14, 15, 16}};

        int[][] b = {{ 17, 18, 19, 20},    // initialisiere Matrix B
                    { 21, 22, 23, 24},
                    { 25, 26, 27, 28},
                    { 29, 30, 31, 32}};

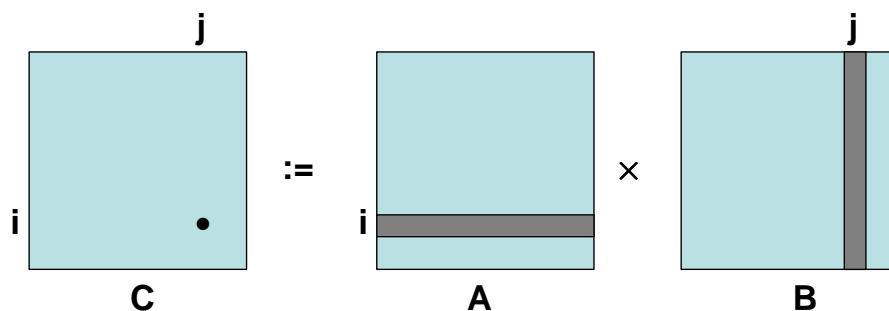
        int[][] c = new int[n][n];        // besorge Platz fuer Matrix

        int i, j, k;                      // Laufindizes

        for (i = 0; i < n; i++)           // Zeilenindex
            for (j = 0; j < n; j++) {     // Spaltenindex
                c[i][j]=0;                 // setze Ergebnis auf 0
                for (k = 0; k < n; k++)    // verknuepfe alle Komponenten
                    c[i][j]=c[i][j]+a[i][k]*b[k][j]; // von Zeile i und Spalte j
            }
    }
}

```

c_{ij} errechnet sich aus i -ter Zeile von A und j -ter Spalte von B:



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: ");

        IO.print(s);

        for (i=0; i<s.length &&      // solange s noch nicht am Ende
              i<t.length &&        // solange t noch nicht am Ende
              s[i]==t[i];          // solange beide noch gleich
              i++);                // gehe eine Position weiter

        if (i==s.length && i==t.length) IO.print("="); // gleichzeitig zu Ende
        else if (i==s.length)          IO.print("<"); // s kuerzer als t
        else if (i==t.length)          IO.print(">"); // t kuerzer als s
        else if (s[i]<t[i])              IO.print("<"); // s kleiner als t
        else                            IO.print(">"); // t kleiner als s

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

Resultierende Ordnung im Telefonbuch:

...
MAI
MAIER
MEIER
MEYER
...

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

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

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

        for (i = 2; i < n; i++)
            if (prim[i]) {
                IO.println(i,10); // falls i als Primzahl erkannt,
                for (j = i+i; j < n; j = j+i) // fuer alle Vielfachen j von i
                    prim[j] = false; // streiche j aus der Liste
            }
    }
}

```

Beispiel für $n=20$ (mit - markiert sind die Vielfachen der Primzahlen):

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
		-		-		-	-	-		-		-	-	-		-		-

Hinweis: Der oben gezeigte Algorithmus lässt sich noch beschleunigen, indem beim Herausstreichen der Vielfachen von i nicht bei $i+i$, sondern erst bei $i*i$ begonnen wird, da die kleineren Vielfachen von i bereits zum Streichen verwendet worden sind. Dabei ist darauf zu achten, dass $i*i$ unterhalb von n liegt:

```
for (j=i*i; 0<j && j<n; j=j+i)
```

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

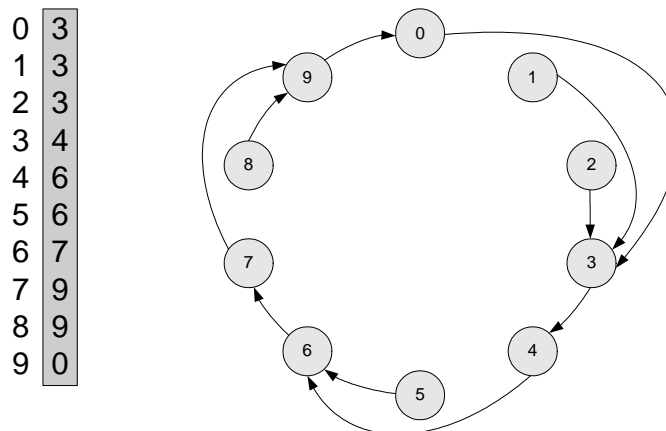
        for (i = 0; i < n; i++)   // initiale Aufstellung
            next[i] = (i+1) % n;

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

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

```



Array next und Verweisstruktur nach dem 4. Abschlagen bei n=10 und k=3

3.6 Feld von Zuständen

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

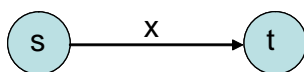
S	=	endliche Zustandsmenge
Σ	=	endliches Eingabealphabet
$\delta : S \times \Sigma \rightarrow S$	=	Überföhrungsfunktion
$s_0 \in S$	=	Anfangs- oder Startzustand
$F \subseteq S$	=	Endzustände

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

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

$$\delta^*(s_0, x_0x_1x_2 \dots x_{n-1}) = \delta(\dots \delta(\delta(\delta(s_0, x_0), x_1), x_2) \dots x_{n-1})).$$

Die Wirkungsweise der Überföhrungsfunktion δ kann durch einen Zustandsüberföhrungsgraphen beschrieben werden. In diesem Graphen föhrt eine mit x beschriftete Kante von Knoten s zu Knoten t , falls gilt: $\delta(s, x) = t$.



Beispiel: A soll durch 3 teilbare Dualzahlen erkennen. Hierzu spendieren wir drei Zustände mit der Bedeutung Rest 0, Rest 1 und Rest 2. Folgende Fälle können auftreten, wenn ein String w , dessen Rest als 0, 1 oder 2 bekannt ist, verlängert wird mit dem Zeichen 0 bzw. mit dem Zeichen 1:

w	w0	w1
$3x + 0$	$6x + 0$	$6x + 1$
Rest 0	Rest 0	Rest 1
$3x + 1$	$6x + 2$	$6x + 3$
Rest 1	Rest 2	Rest 0
$3x + 2$	$6x + 4$	$6x + 5$
Rest 2	Rest 1	Rest 2

Also ergibt sich folgender Automat:

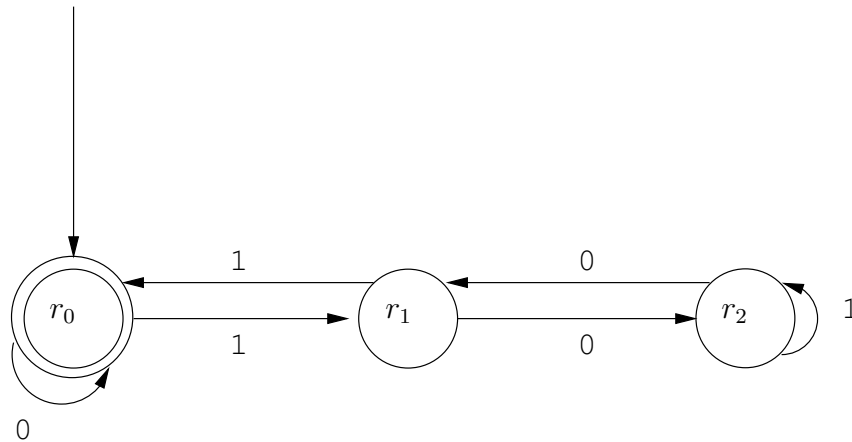
$$S = \{r_0, r_1, r_2\}$$

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

Startzustand ist r_0

$$F = \{r_0\}$$

Dazu gehört ein Zustandsüberföhrungsgraph mit den Knoten r_0 , r_1 und r_2 , welche die Zustände beschreiben, wenn für den bislang eingelesenen String die Division durch 3 den Rest 0, 1 bzw. 2 ergibt. An der Kante steht das jeweils nächste Bit der Dualzahl, die von links nach rechts abgearbeitet wird.



```

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

import AlgoTools.IO;

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

public class Automat {

    public static void main (String [] argv) {

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

        int s=0; // Nummer des Zustands

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

        for (int i=0; i < zeile.length; i++) // fuer jedes Zeichen

            s = delta[s][zeile[i]-'0']; // wende einmal delta an
                                     // dabei Zeichen
                                     // in Zahlen umwandeln

        if (s==0) IO.println("ist durch 3 teilbar"); // Automat akzeptiert
            else IO.println("ist nicht durch 3 teilbar"); // Automat lehnt ab
    }
}

```

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
        links = 0; // initialisiere links
        rechts = a.length-1; // initialisiere rechts
        mitte = (links + rechts)/2; // initialisiere mitte
        while (links <= rechts && a[mitte] != x) { // solange noch Hoffnung
            if (a[mitte] < x) links = mitte+1; // zu kurz gesprungen
            else rechts = mitte-1; // zu weit gesprungen
            mitte = (rechts+links)/2; // neue Mitte
        }
        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 in der Programmiersprache Java *call-by-value* verwendet, d.h., ein im Methodenrumpf benutzter formaler Parameter wird bei Methodenaufruf mit dem Wert des aktuellen Parameters versorgt und innerhalb der Methode kann nur auf den Wert zugegriffen werden.

Methoden können durch verschiedene Parametersätze überladen werden, d.h. der Name einer Methode wird mehrmals verwendet, wobei sich die formalen Parameter unterscheiden müssen. Die Kombination aus Methodennamen und den formalen Parametern bezeichnet man auch als *Signatur*.

Eine mit dem Schlüsselwort `static` versehene Deklaration einer Klassenvariablen führt eine klassenbezogene Variable ein, welche in allen Methoden dieser Klasse sichtbar ist. Auch eine Klassenvariable kann mit unterschiedlichen *Modifiern* versehen werden: Durch den Modifier `private` wird der Zugriff nur von der eigenen Klasse ermöglicht, durch den Modifier `public` darf auch von einer beliebigen anderen Klasse zugegriffen werden.

4.1 Methodenaufrufe

```

/***** 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-Parameter
    // 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
    // mit zwei Integer-Parametern
    // 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));
    }
}

```

4.2 Parameterübergabe an Arrays

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

/** Uebergabe von Arrays an Methoden
 */

public class Parameter {

    public static void zeige(int[] a) { // Methode ohne Rueckgabewert
        for (int i=0; i < a.length; i++) // erhaelt Array als Parameter
            IO.print(a[i],5); // gibt Array aus
        IO.println();
    }

    public static int summe(int[] a) { // Methode mit Rueckgabewert
        int s = 0; // erhaelt Array als Parameter
        for (int i=0; i< a.length; i++) // initialisiert Summe
            s = s + a[i]; // durchlauft Array
        return s; // berechnet Summe
    } // und liefert sie zurueck

    public static int[] kopiere(int[] a) { // Methode mit Rueckgabewert
        int[] b = new int[a.length]; // erhaelt Array als Parameter
        for (int i=0; i<a.length; i++) // besorgt Platz fuer 2. Array
            b[i] = a[i]; // durchlauft Array a
        return b; // kopiert es nach b
    } // liefert b zurueck

    public static void reset(int[] a) { // Methode mit Seiteneffekt
        for (int i=0; i < a.length; i++) // erhaelt Array als Parameter
            a[i] = 0; // durchlauft Array
    } // und setzt jede Komponente auf 0

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

4.3 Sichtbarkeit

Innerhalb einer Methode verdecken formale Parameter und lokale Variablen gleich lautende Identifier aus der umschließenden Klasse. So kann ein Variablenname mehrmals innerhalb einer Klasse erscheinen, z.B. als Klassenvariable, lokale Variable oder als formaler Parameter.

Soll innerhalb einer Methode auf eine Variable zugegriffen werden, wird diese erst innerhalb der Methode gesucht (lokale Variable oder formaler Parameter). Ist dort die entsprechende Variable nicht vorhanden wird nach einer entsprechenden Klassenvariable gesucht.

```

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

/** Sichtbarkeit von Variablen- und Methodennamen */

public class f { // Klasse f

    public static int f; // klassenbezogene Variable f

    public static void f(){ // Klassenmethode f (null-stellig)
        f = 42; // referiert klassenbezogene Variable
    }

    public static int f(int n) { // Klassenmethode f (einstellig)
        int f=1; // lokale Variable f
        for (int i=1; i<=n; i++) f = f*i; // berechnet Fakultaet
        return f;
    }

    public static int f(int x, int y){ // Klassenmethode f (zweistellig)
        while (x != y) // berechnet ggt von x und y
            if (x>y) x = x-y; else y=y-x;
        return x;
    }

    public static void setze(int k){ // Klassenmethode setze
        f = k;
    }

    public static int hole() { // Klassenmethode hole
        return f;
    }

    public static void main(String[] argv){
        int f=5; // lokale Variable f
        f(); // Aufruf von nullstelligem f
        f = f(f); // Aufruf von einstelligem f
        f = f(f,f); // Aufruf von zweistelligem f
        setze(42); // deponiere in Klassenvariable
        IO.println(hole()); // hole von Klassenvariable
    }
}

```


4.4 Fehlerbehandlung

Generell ist der Programmierer verpflichtet seine Methoden robust zu implementieren. Darunter versteht man, dass eine Methode mit allen Werten, die ihr übergeben werden können, sinnvolle Resultate liefert.

Man betrachte folgende Klassenmethode `fakultaet()`, die alle positiven Zahlen von 1 bis zum übergebenen Wert aufmultipliziert und das Ergebnis zurückliefert:

```
public static int fakultaet(int n) {
    int f = 1;
    for (int i=1;i<=n;i++) {
        f = f * i;
    }
    return f;
}
```

Falls für `n` ein Wert kleiner als 0 übergeben wird, liefert die Methode das (offenbar falsche) Ergebnis 1 zurück. Falls für `n` ein zu großer Wert übergeben wird, entsteht ein Integer-Overflow. Korrekt wäre es, wenn die Methode dem Benutzer mitteilen kann, ob der übergebene Parameter im korrekten Wertebereich liegt. Dies könnte über den Rückgabewert geschehen (z.B. -1); besser wäre es jedoch, das Ergebnis der Berechnung von der Fehlermeldung zu trennen. Hierzu kann in Java eine Methode eine `RuntimeException` werfen. Wird diese `RuntimeException` nicht weiter behandelt, führt sie zu einem Programmabbruch. Dies ist eine recht drastische Maßnahme, allerdings ist es besser als eine Berechnung mit fehlerhaften Werten durchzuführen, ohne es dem Benutzer mitzuteilen.

Die Fehlerbehandlung mit einer `RuntimeException` kann wie folgt aussehen:

```
public static int fakultaet(int n) {
    if (n<0) throw new RuntimeException("Obacht: Parameter negativ");
    if (n>12) throw new RuntimeException("Obacht: Parameter zu gross");
    int f = 1;
    for (int i=1;i<=n;i++) {
        f = f * i;
    }
    return f;
}
```

Bei der Erzeugung einer `RuntimeException` kann eine Fehlermeldung angegeben werden. Diese wird bei Programmabbruch auf der Kommandozeile ausgegeben. Die erzeugte `RuntimeException` wird dann mit dem Schlüsselwort `throw` aus der Methode zum Aufrufer der Methode *geworfen* und die Abarbeitung der Methode endet an dieser Stelle.

Generell sollte eine Methode fehlerhafte Eingaben ablehnen, aber auch der Benutzer der Methode sollte vorher sicherstellen, dass erst gar keine fehlerhaften Werte an eine Methode übergeben werden.

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
    }

    // Leonardo da Pisa, *1180
    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 muster (int n) { // schreibt 2 hoch n Kreuze
        if (n>0) {
            muster(n-1);
            IO.print('#');
            muster(n-1);
        }
    }

    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]));
        muster(10); IO.println();
    }
}

```

5.2 Türme von Hanoi

```

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

/**
 * Türme 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 für n Scheiben, d.h. bei Aufruf von

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

Da für $n > 0$ zwei neue Aufrufe mit verkleinertem n sowie eine Druckzeile entstehen, gilt offenbar:

$$f(n) := \begin{cases} 0 & \text{falls } n = 0 \\ 2 \cdot f(n-1) + 1 & \text{falls } n > 0 \end{cases}$$

d.h., die Wertetabelle beginnt wie folgt:

n	$f(n)$
0	0
1	1
2	3
3	7
4	15
5	31
6	63

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

Beweis durch Induktion

Induktionsverankerung: $f(0) = 0 = 2^0 - 1$

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, Verifikation, Terminierung

- 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	\dots	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
$\log(n)$	$2.3 \mu\text{s}$	$3.0 \mu\text{s}$	$4.9 \mu\text{s}$	$5.3 \mu\text{s}$	$5.6 \mu\text{s}$	$5.9 \mu\text{s}$
n	$10 \mu\text{s}$	$20 \mu\text{s}$	$30 \mu\text{s}$	$40 \mu\text{s}$	$50 \mu\text{s}$	$60 \mu\text{s}$
$n \cdot \log(n)$	$23 \mu\text{s}$	$60 \mu\text{s}$	$147 \mu\text{s}$	$212 \mu\text{s}$	$280 \mu\text{s}$	$354 \mu\text{s}$
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:

```

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

```

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

```

int treffer=0;
for (i = 0, i < k-1; i++) {
    for (j = i+1; j < k; j++) {
        if (a[i] == a[j]) treffer++;
    }
}

```

$(k \cdot (k - 1))/2$ 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, darunter die gesuchte Zahl

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?

```
boolean erfolg=false;
for (i = 0; i < N - M + 1 && !erfolg; i++) {           // Index in Zeichenkette
    for (j = 0; (j < M) && (p[j] == s[i+j]); j++);     // Index im Pattern
    if (j == M) erfolg=true;                           // 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 p und 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: $0 = n^2 + n - 4n^2 \cdot x$

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

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 $f(n) = f(n-1) + f(n-2) + 1$ 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} :

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

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

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

$$\text{für } n = 20 \text{ ist } \begin{array}{ll} 1.62^n & \approx 15.000 \\ 2^n & \approx 1.000.000. \end{array}$$

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.

```
//  $i > 0 \wedge z = i^2$ 
```

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

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

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

```
//  $P$   
while  $Q$  {  
    //  $P \wedge Q$   
    :  
    :  
    //  $P$   
}  
//  $P \wedge \neg Q$ 
```

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);
//  $n \geq 0$ 
x = 0; y = 1; z = 1;
//  $x^2 \leq n \wedge z = (x+1)^2 \wedge y = 2x+1$  = Schleifeninvariante  $P$ 
while (z <= n) /*  $Q$  */ {
    //  $x^2 \leq n \wedge z = (x+1)^2 \wedge y = 2x+1 \wedge z \leq n$ 
    //  $(x+1)^2 \leq n \wedge z = (x+1)^2 \wedge y = 2x+1$ 
    x = x + 1;
    //  $x^2 \leq n \wedge z = x^2 \wedge y = 2x-1$ 
    y = y + 2;
    //  $x^2 \leq n \wedge z = x^2 \wedge y = 2x+1$ 
    z = z + y;
    //  $x^2 \leq n \wedge z = x^2 + 2x + 1 \wedge y = 2x + 1$ 
    //  $x^2 \leq n \wedge z = (x+1)^2 \wedge y = 2x+1$  = Schleifeninvariante  $P$ 
}
//  $x^2 \leq n \wedge z = (x+1)^2 \wedge y = 2x+1 \wedge z > n$  =  $P \wedge \neg Q$ 
//  $x^2 \leq n < (x+1)^2$ 
//  $x = \lfloor \sqrt{n} \rfloor$ 
IO.println (x);
// Ausgabe:  $\lfloor \sqrt{n} \rfloor$ 

```

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

```

Terminierung

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

Laufzeit

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

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

Hinweis: Der naive Ansatz

```

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

```

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 `haltetest` in der Klasse `Fee`:

```
public class Fee {
    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 (Fee.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 `haltetest(q, q) == false`
 \Rightarrow Quer angesetzt auf q hält nicht!
- 2. Fall: Hält nicht \Rightarrow `haltetest(q, q) == true`
 \Rightarrow Quer angesetzt auf q hält an!

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

Kapitel 7

Sortieren

In diesem Kapitel werden zahlreiche Algorithmen zum Sortieren vorgestellt. Die Verfahren eignen sich gut, um grundsätzliche Programmierstrategien wie *Greedy*, *Divide & Conquer* und Rekursion zu behandeln. Eine *Greedy*-Strategie versucht durch *gieriges* Vorgehen das jeweils kurzfristig bestmögliche zu erreichen. Bei *Divide & Conquer* wird zunächst das gegebene Problem in Teilprobleme zerlegt, dann deren Lösungen errechnet und diese dann anschließend zur Gesamtlösung zusammengeführt. Rekursive Verfahren lösen die ursprüngliche Aufgabenstellung auf einer reduzierten Problemgröße mit demselben Lösungsansatz und konstruieren aus dieser Teillösung die Gesamtlösung.

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 SelectionSort

“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 *SelectionSort*

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 (*ShakerSort*), 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

$$\begin{aligned} &= 2 \cdot [(c_1 + c_2) \cdot \frac{n}{2} \cdot (\log n - 1) + c_1] + c_2 \cdot n \\ &= (c_1 + c_2)n \cdot \log n - (c_1 + c_2) \cdot n + 2c_1 + c_2 \cdot n \\ &= [(c_1 + c_2)n \cdot \log n + c_1] + [c_1 - c_1 \cdot n] \\ &\leq (c_1 + c_2)n \cdot \log n + c_1 \end{aligned}$$

Aber: MergeSort benötigt $O(n)$ zusätzlichen Platz!

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

```

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

```

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

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

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

/** testet Sortierverfahren
 */

public class SortTest {

    public static void main (String [] argv) {

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

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

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

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

7.4 QuickSort

```

import AlgoTools.IO;

/***** 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) {                // oeffentliche Methode
        quicksort(a, 0, a.length-1);                // startet private Methode
    }

    private static void quicksort (int[] a, int unten, int oben) {
        int tmp ;                                    // Hilfsvariable
        int i = unten;                               // untere Intervallgrenze
        int j = oben;                                // obere Intervallgrenze
        int mitte = (unten + oben) / 2;              // mittlere Position
        int x = a[mitte];                            // Pivotelement

        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 Array-Haelfte sind kleiner
        // als alle Elemente der rechten Array-Haelfte

        if (unten < j) quicksort(a, unten, j);      // sortiere linke Haelfte
        if (i < oben) quicksort(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):

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

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); // 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

```

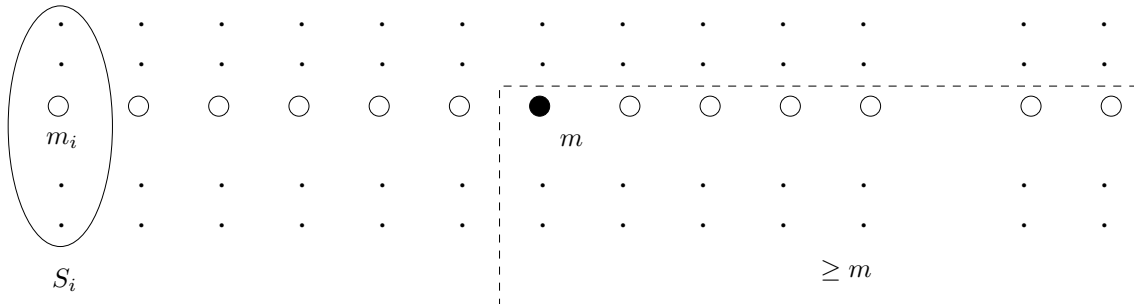
public static int select (int[] S, int k) {
    /* Liefert aus dem Feld S das k-t kleinste Element. */

    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 = select ( $\bigcup_i m_i, \frac{n}{10}$ );
        A = {x ∈ S | x < m};
        B = {x ∈ S | x == m};
        C = {x ∈ S | x > m};
        if (|A| >= k) return select (A, k);
        if (|A|+|B| >= k) return m;
        if (|A|+|B|+|C| >= k) return select (C, k-|A|-|B|);
    }
}

```

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



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

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

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

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

Also:

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

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

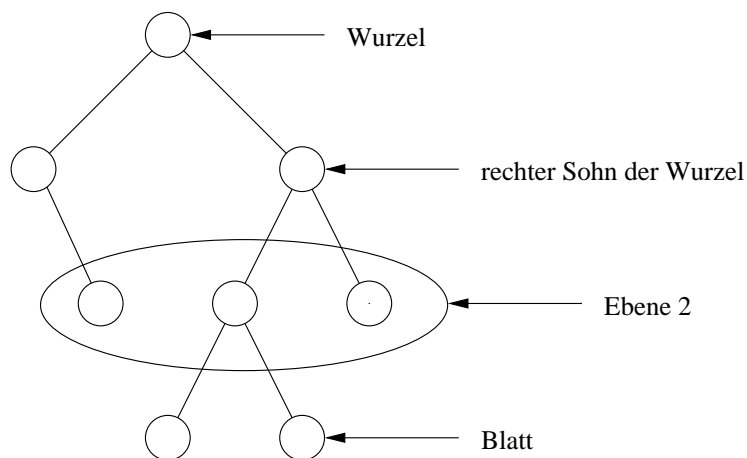
$$\begin{aligned} f(n) &\leq c \cdot n + f\left(\frac{n}{5}\right) + f\left(\frac{3}{4}n\right) \\ &\quad \uparrow \\ &\text{Rekursionsgl.} \\ &\leq c \cdot n + 20 \cdot c \cdot \frac{n}{5} + 20 \cdot c \cdot \frac{3}{4} \cdot n \\ &\quad \uparrow \\ &\text{Ind.-Annahme} \\ &= 1 \cdot c \cdot n + 4 \cdot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

7.6 HeapSort

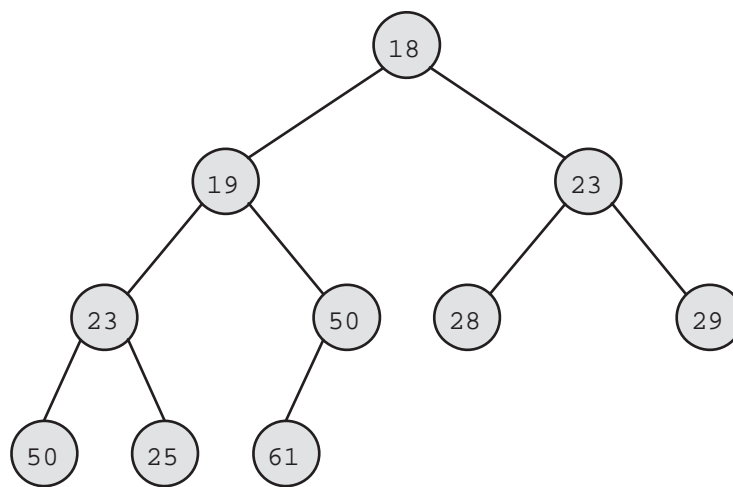
Baum und Heap

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

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



Def.: Ein *Heap* ist ein binärer Baum mit h Ebenen, in dem die Ebenen $0, 1, \dots, h - 2$ vollständig besetzt sind; die letzte Ebene $h - 1$ 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.

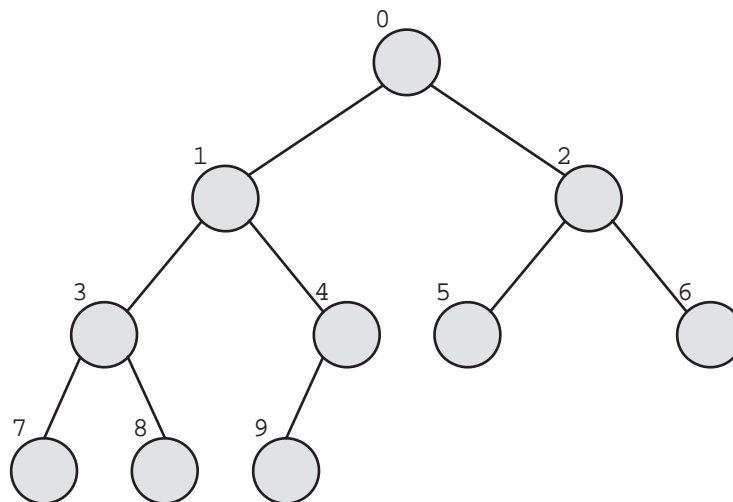
Vertausche so lange Knoten mit "kleinerem" Sohn, bis Heapbeziehung eintritt.

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;

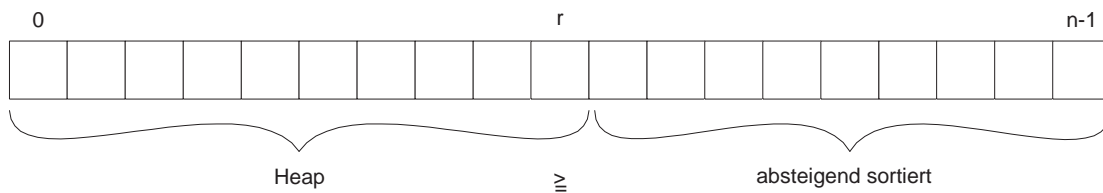
/** Iteratives Sortieren mit Heapsort
 * Entnimm einem Heap so lange das kleinste Element, bis er leer ist.
 * Die entnommenen Elemente werden im selben Array gespeichert.
 */

public class HeapSort {

    private static void sift (int[] a, int l, int r) { // repariere 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 = 2 * i + 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 $n \leq 2^h - 1$ die Anzahl seiner Elemente. Z.B. Ein Heap mit $h = 4$ Ebenen kann maximal $n = 2^4 - 1 = 15$ Knoten haben.

Ebene	Sickertiefe	Anzahl der Knoten dieser Ebene
0	h	1
\vdots	\vdots	\vdots
$h - 3$	3	$\frac{n}{8}$
$h - 2$	2	$\frac{n}{4}$
$h - 1$	1	$\frac{n}{2}$

Anzahl der Schritte, beginnend bei vorletzter Ebene ($h - 2$), endend bei Ebene 0:

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

\Rightarrow Aufwand $O(n)$, denn: $\frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \dots < 1.5$

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

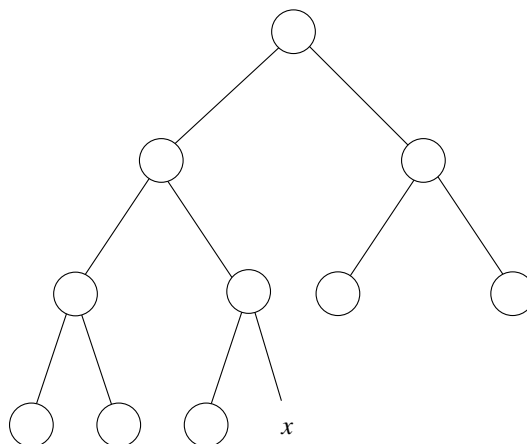
Gesamtaufwand: $O(n) + O(n \cdot \log n) = O(n \cdot \log n)$ für best, average und worst case.

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: Füge neues Blatt mit Schlüssel x an, und lasse x hochsickern.



Aufwand: $O(\log n)$

7.7 Zusammenfassung von Laufzeit und Platzbedarf

	Best	Average	Worst	zusätzlicher Platz
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
MergeSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n)$
QuickSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n^2)$	$O(\log n)$
HeapSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(1)$

Der lineare zusätzliche Platzbedarf beim Mergesort-Algorithmus wird verursacht durch die Methode `merge`, welche für das Mischen zweier sortierter Folgen ein Hilfs-Array derselben Größe benötigt.

Der logarithmische zusätzliche Platzbedarf beim Quicksort-Algorithmus wird verursacht durch das Zwischenspeichern der Intervallgrenzen für die noch zu sortierenden Array-Abschnitte, die jeweils nach dem Aufteilen anhand des Pivot-Elements entstehen.

Beispiel:

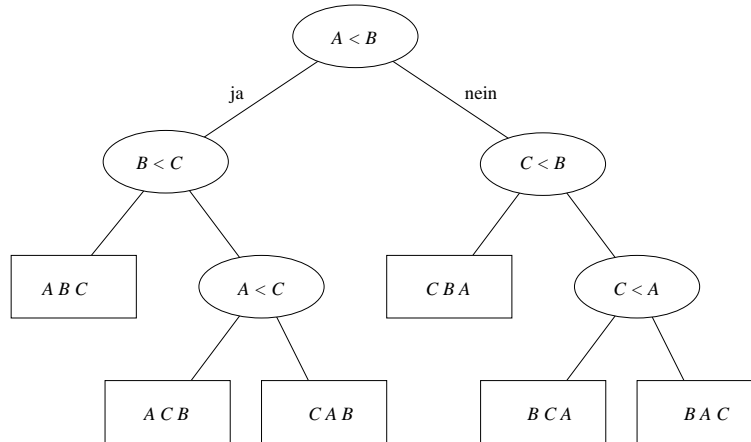
Sei ein Array gegeben mit insgesamt 16 Daten. Der initiale Aufruf von Quicksort wird versorgt mit den Arraygrenzen $[0, 15]$. Folgende Sequenz von gespeicherten Intervallgrenzen entsteht:

```
[0, 15]
[0, 7] [8, 15]
[0, 7] [8, 11] [12, 15]
[0, 7] [8, 11] [12, 13] [14, 15]
[0, 7] [8, 11] [12, 13]
[0, 7] [8, 11]
[0, 7] [8, 9] [10, 11]
[0, 7] [8, 9]
[0, 7]
[0, 3] [4, 7]
[0, 3] [4, 5] [6, 7]
[0, 3] [4, 5]
[0, 1] [2, 3]
[0, 1]
```

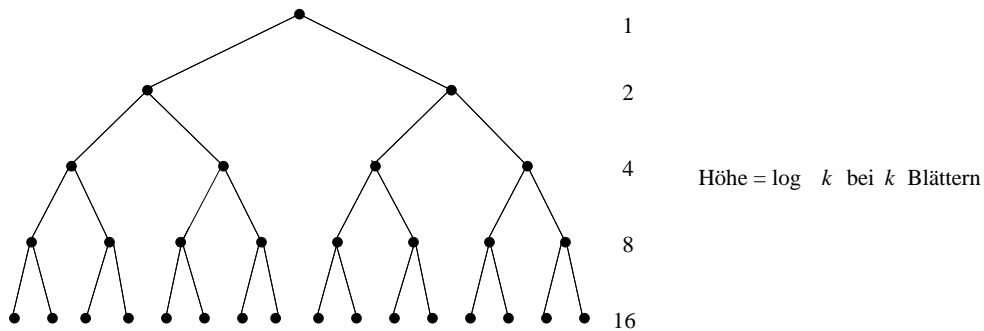
Die maximale Ausdehnung liegt bei einer Folge von jeweils halbierten Intervallgrenzen; davon gibt es logarithmisch viele.

7.8 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.

7.9 BucketSort

Es folgt ein Sortierverfahren, welches ohne Vergleichen arbeitet und daher die Voraussetzungen für die Ableitung der unteren Schranke $O(n \cdot \log n)$ nicht erfüllt und sie daher unterbieten kann. *BucketSort* sortiert einen endlichen, zuvor bekannten Schlüsselbereich durch Verteilen auf Buckets. Eine Erweiterung dieser Idee für das Sortieren von Strings führt zum Sortierverfahren *RadixSort*, welches hier nicht behandelt wird.

```

/***** 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 {

    private static final int N = 256;           // Alphabetgroesse N

    public static void sort (char[] a) {       // sortiere Character-Array a

        int[] b = new int[N];                 // N Buckets
        int i, j, k;                           // 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

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

    }

    public static void main (String [] argv) {

        char[] zeile;                          // Zeichenfolgen

        zeile = IO.readChars("Bitte Zeichenkette: "); // Zeichenkette einlesen
        sort(zeile);                            // Bucket-Sort aufrufen
        IO.print("sortiert mit Bucket-Sort: "); // Ergebnis ausgeben
        IO.println(zeile);

    }
}
// Aufwand: O(n) + O(N) bei n(=a.length) zu sortierenden Zeichen
// aus einem N-elementigen Alphabet

```


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. Eine abgeleitete Klasse erbt von der Oberklasse die dort definierten Datenfelder und Methoden, fügt ggf. eigene hinzu und kann ihnen ggf. durch Überschreiben eine neue Bedeutung geben.

Objekte werden durch Konstruktoren erzeugt. Jede Klasse besitzt einen oder mehrere Konstruktoren, die für das Instanzieren ihrer Objekte zuständig sind. Wird kein eigener Konstruktor definiert, existiert automatisch der vom System bereitgestellte Default-Konstruktor (ohne Parameter).

8.1 Sichtbarkeit von Datenfeldern

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

Ein Datenfeld, welches ohne das Schlüsselwort `static` deklariert wird, ist eine sogenannte Instanzvariable. Eine Instanzvariable existiert je Instanz (also je Objekt) genau einmal und kann entsprechend für jede Instanz einen anderen Wert annehmen. Eine Instanzvariable stellt sozusagen eine Eigenschaft eines Objektes dar.

Hinweis: Variablen, die innerhalb einer Methode deklariert werden, bezeichnet man als *lokale Variablen*.

Die Sichtbarkeit von (Instanz- und Klassen-) 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.

Unter einem Paket versteht man alle Klassen, die in einem bestimmten Verzeichnis liegen. Sinnvollerweise werden logisch zueinander gehörige Klassen in ein Verzeichnis gepackt. Diese Verzeichnisse werden wiederum logisch zusammengehörig in weitere Verzeichnisse geschachtelt, so dass eine ganze Verzeichnisstruktur entsteht. Nach Java-Konvention sollten die Paketnamen (und damit die Verzeichnisse) immer klein geschrieben sein.

Das Standardpaket besteht aus allen Klassen im aktuellen Arbeitsverzeichnis. Eigene Pakete können beispielsweise angelegt werden, indem man

```
package a.b.c;
```

am Anfang einer Klasse schreibt und diese Klasse auch in dem angegebenen Pfad `a/b/c` ablegt. Eine so definierte Klasse kann nach dem Übersetzen durch

```
java a.b.c.MeineKlasse
```

aufgerufen werden, wobei der Aufrufer sich in dem Verzeichnis befindet, welches das Verzeichnis `a` enthält.

Die Klasse `MeineKlasse` kann von anderen Klassen durch

```
import a.b.c.MeineKlasse;
```

importiert werden.

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
andere Klasse im selben Paket	ja	ja	ja	nein
Subklasse in anderem Paket	ja	ja	nein	nein
Keine Subklasse, anderes Paket	ja	nein	nein	nein

- Elemente des Typs `public` sind in der Klasse selbst, in Methoden abgeleiteter Klassen und für den Aufrufer von Instanzen der Klasse sichtbar.
- Elemente des Typs `protected` sind in der Klasse selbst und in Methoden abgeleiteter Klassen sichtbar. Der Aufrufer einer Instanz der Klasse hat nur Zugriff, wenn er in demselben Paket definiert wurde.
- Elemente vom Typ `private` sind nur in der Klasse selbst sichtbar.
- Elemente ohne Modifier gelten als *friendly* und werden als `protected` eingestuft mit der Einschränkung, dass sie in Unterklassen anderer Pakete unsichtbar sind.

8.2 Erste Beispiele

Im Folgenden werden beispielhaft einige Klassen gezeigt, in denen die obigen Konzepte veranschaulicht werden.

```

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

/** 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 t, int m, int j){           // Konstruktor mit 3 Parametern
        tag  = t;                                // initialisiere Tag
        monat = m;                               // initialisiere Monat
        jahr = j;                                // 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
    }

}

/***** DatumTest.java *****/

import AlgoTools.IO;

/** Klasse DatumTest, testet die Klasse Datum */

public class DatumTest {

    public static void main(String[] argv) {
        Datum d;                                // deklariere ein Datum
        d = new Datum (15,8,1972);              // kreiere Datum 15.08.1972
        d = new Datum (1972);                   // kreiere Datum 01.01.1972
        d.jahr++;                               // erhoehe Datum um ein Jahr
        IO.println(d.toString());               // drucke Datum
        IO.println(d);                          // hier: implizites toString()
    }
}

```

```

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

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

import java.util.GregorianCalendar;

public class Person {

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

    public Person (String vn, // Konstruktor mit 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
        int jetzt = new GregorianCalendar().get(GregorianCalendar.YEAR);
        return jetzt - geb_datum.jahr; // liefert das Lebensalter
    }

    public String toString() { // Methode, ueberschreibt toString
        return vorname + " " + nachname; // liefert Vor- und Nachname
    }
}

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

/** Klasse PersonTest, testet Klasse Person */

public class PersonTest {

    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); // gib Vor- und Nachname aus
        IO.println(" ist "+p.alter()+" Jahre alt."); // gib Alter aus
    }
}

```

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

/** Klasse Student, spezialisiert die Klasse Person
 * durch statische 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;           // statische 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 f,                             // Studienfach
         int jsb)                               // Studienbeginn
    {
        super(vn, nn, t, m, j);                // Konstruktor des Vorfahren
        fach = f;                               // 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
    }
}

/* ***** StudentTest.java ***** */

import AlgoTools.IO;

/** Klasse StudentTest, testet die Klasse Student */

public class StudentTest {

    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);                             // gib Name aus und
        IO.println(" hat die Matrikelnummer " + s.mat_nr); // Matrikelnummer
    }
}
```

```

/***** PersonStudentTest.java *****/

import AlgoTools.IO;

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

public class PersonStudentTest {

    public static void main (String [] argv) {

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

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

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

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

        p = s;           // p zeigt auf s

        IO.println(p + "'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
    }
}

```

Innerhalb einer Vererbungshierarchie können Instanzmethoden überschrieben werden. So wird in der Klasse `Person` die Methode `jahrgang()` definiert. In der Klasse `Student` soll sich diese Methode aber anders verhalten, daher wird sie in der Klasse `Student` überschrieben.

8.3 Binden

Wenn mehrere Methoden mit gleichem Namen und unterschiedlicher Parameterliste existieren, ist die Methode *überladen*.

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

Bei der Programmiersprache Java werden alle Instanzmethoden dynamisch gebunden. Der Compiler prüft lediglich, ob die entsprechende Methode vorhanden ist, aber erst zur Laufzeit wird festgestellt, von welchem Typ das Objekt ist und die entsprechende Methode wird ausgewählt. So kann eine Referenz vom Typ `Person` sowohl auf ein `Person` Objekt verweisen, als auch auf ein `Student` Objekt. Erst zur Laufzeit wird festgestellt, um welches Objekt es sich handelt und die entsprechende Methode wird ausgewählt.

Alles andere (Klassenmethoden, Klassenvariablen, Instanzvariablen) wird statisch gebunden, also bereits zur Compilezeit ausgewählt und *fest verdrahtet*. Der Compiler wählt also die Klassenmethode oder Variable aus, die dem Typ der Referenz entspricht.

Folgende Fälle können z.B. 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);
```

Der Compiler versucht das Datenfeld `fach` statisch an das Objekt zu binden. Dies führt zu einem Übersetzungsfehler, da Instanzen vom Typ `Person` nicht über das Datenfeld `fach` verfügen.
- ```
IO.println(((Student)p).fach);
```

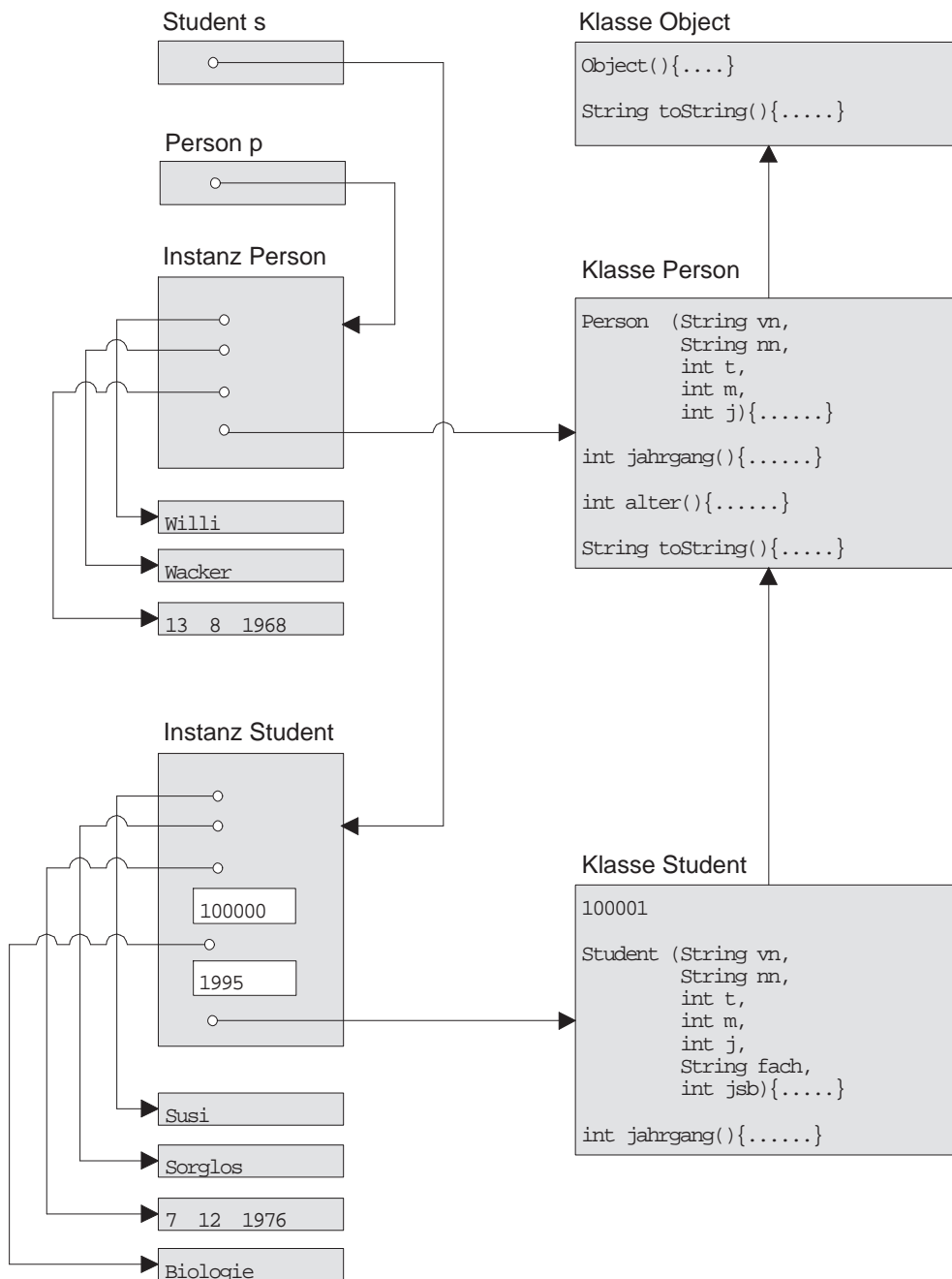
Der Compiler versucht das Datenfeld `fach` statisch an das Objekt zu binden. Der Code wird übersetzt, da dem Compiler mit dem Cast vom Programmierer eine Instanz vom Typ `Student` garantiert wird, welche über das Datenfeld `fach` verfügt. Das Beispiel führt aber zu einem Laufzeitfehler, wenn `p` zur Laufzeit nicht auf einen Studenten verweist.
- ```
p = s;
```

Wird übersetzt und ausgeführt. `p` verweist nun auf einen Studenten.
- ```
IO.println(((Student)p).fach);
```

Wird übersetzt und ausgeführt, da `p` nun auf eine Instanz vom Typ `Student` verweist, welche über das Datenfeld `fach` verfügt.
- ```
IO.println(p.jahrgang());
```

Die Methode `jahrgang()` wird dynamisch zur Laufzeit gebunden. Der Quellcode 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.

Die folgende Grafik verdeutlicht das Zusammenspiel zwischen den Klassen `Person` und `Student` und ihren 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 dort hin 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.



8.4 Referenzen

Referenzen auf Objekte modellieren die in konventionellen Programmiersprachen vorhandenen Zeiger. Um dieses Konzept zu demonstrieren, implementieren wir den in Kapitel 3 vorgestellten Abzählreim nun mit objektorientierter Programmierung. Der an Array-Position i notierte Index des Nachfolgerkind j wird realisiert durch ein Kind-Objekt mit Nr. i , welches eine Referenz enthält auf ein Kind-Objekt mit Nr. j .

```

/***** Kind.java *****/
/** Klasse Kind
 * bestehend aus Nummer und Verweis auf Nachbarkind
 * mit Konstruktor zum Anlegen eines Kindes */

public class Kind {
    int nr; // Nummer
    Kind next; // Verweis auf naechstes Kind
    public Kind (int nr) { // Konstruktor fuer Kind
        this.nr = nr; // initialisiere Nummer
    }
}

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

/** Klasse VerweisAbzaehltreim
 * implementiert einen Abzaehltreim mit k Silben fuer n Kinder mit Verweisen
 * verwendet dabei Objekte vom Typ Kind */

public class VerweisAbzaehltreim {

    public static void main (String [] argv) {

        int i; // Laufvariable
        int n=IO.readInt("Wie viele Kinder ? "); // erfrage Kinderzahl
        int k=IO.readInt("Wie viele Silben ? "); // erfrage Silbenzahl
        Kind erster, letzter, index; // deklariere drei Kinder
        erster = letzter = new Kind(0); // kreierte 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);
    }
}

```

8.5 Wrapperklassen

In der Programmiersprache Java existieren einige *primitive* Datentypen, wie z.B.: `int`, `double`, `boolean`, `char`, usw. Für jeden dieser primitiven Datentypen existiert eine entsprechende Klasse, die einem primitiven Datentyp entspricht. So existiert in Java die Klasse `Integer`, `Double`, `Boolean`, `Character`, usw.

Hinweis: Zeichenketten sind auch Objekte. Diese sind Instanzen der Klasse `String`. Wenn also im Programmcode eine Zeichenkette mit "Meine Zeichenkette" definiert wird, erzeugt man indirekt ein neues Objekt, an das auch Methodenaufrufe geschickt werden können.

Zu einem primitiven Datentyp kann man sehr leicht das entsprechende Objekt erhalten. Im folgenden Beispiel werden zwei `Integer` Objekte angelegt, ein Wert inkrementiert und danach beide Werte ausgegeben.

```
int i = 23;
Integer j = new Integer(i);
Integer k = new Integer(42);
j = new Integer(j.intValue()+1);

IO.println(j.intValue());
IO.println(k.intValue());
```

Da hierbei eine Objektstruktur um einen primitiven Datentyp *gewickelt* wird, nennt man diese Klassen auch *Wrapperklassen*.

Will man den Wert eines `Integer` Objektes erhöhen, ist dies, wie im obigen Beispiel deutlich wird, recht umständlich. Dies liegt daran, dass einem Wrapperobjekt kein neuer Wert zugewiesen werden kann. Diese Objekte nennt man auch *immutable*, also unveränderbar.

In Java 5.0 wurde der Umgang mit diesen Wrapperklassen etwas vereinfacht. So ist es nun möglich einer Objektreferenz vom Typ einer Wrapperklasse direkt einen entsprechenden primitiven Datentyp zuzuweisen:

```
int i = 23;
Integer j = i;
Integer k = 42;
j++;

IO.println(j);
IO.println(k);
```

Diese direkte Zuweisung nennt man *Boxing*. Der Wert wird also in die Schachtel des Wrappers gepackt. Umgekehrt kann auch der Wert eines Wrapperobjektes direkt als primitiver Datentyp zurückgeliefert werden. Diesen Vorgang nennt man *Unboxing* und beides zusammen wird als *Autoboxing* bezeichnet.

Diese vereinfachte Schreibweise ist durchaus bequem, man sollte allerdings auch auf die *Nebenwirkungen* achten. Schreibt man in seinem Quellcode z.B.: `Integer j = 23;` wird durch das Compilieren daraus folgender Code: `Integer j = new Integer(23);`.

Implementiert man z.B. eine `for`-Schleife, so könnte man folgendes programmieren:

```
for (Integer i=0;i<1000;i++) {
    IO.println(i.intValue());
}
```

Dieser Quellcode sieht auf den ersten Blick gut aus, allerdings werden hier ständig neue Objekte erzeugt, wodurch das Programm langsamer wird. In Wirklichkeit sieht obige Schleife nämlich wie folgt aus:

```
for(Integer i=new Integer(0);
    i.intValue()<1000;
    i=new Integer(i.intValue()+1)) {
    IO.println(i.intValue());
}
```

Es ist also immer darauf zu achten, wo man `Boxing` verwendet und ob man sich durch eine vereinfachte Schreibweise ggf. Nachteile einhandelt.

Eine weitere Neuerung in Java 5.0 ist die *for-each*-Schleife. Wird zum Beispiel ein Array von `Integer`-Objekten erzeugt, kann dieses wie gewohnt mit einer `for`-Schleife durchlaufen werden, oder aber mit einer *for-each*-Schleife:

```
Integer [] a = new Integer[10];

for (int i=0; i<10;i++) {           // Array mit Objekten/Werten fuellen
    a[i] = i;                       // Obacht: Boxing
}

for (int i=0; i<10;i++) {
    IO.println(a[i]);
}

for(Integer i: a) {                 // fuer jedes Integer-Objekt aus a
    IO.println(i);                  //
}
```

In der dritten Schleife verweist die Referenz `i` nach und nach einmal auf jedes Objekt im Array.

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 als Typdeklaration 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

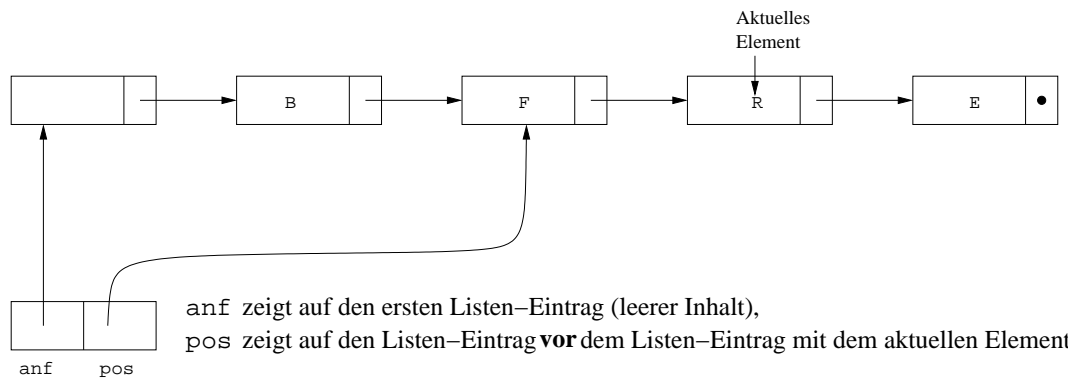
    public void insert(Object x);     // fuegt x vor das aktuelle Element ein
                                        // und macht x zum aktuellen Element

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

}

```

Konzept zur Implementation einer Liste:



Hinweis zur Fehlerbehandlung:

In den folgenden Implementationen wird nach Eintreten einer fehlerhaften Situation eine `RuntimeException` geworfen. Auf das in Java mögliche eigene Exceptionhandling wurde hier zur besseren Lesbarkeit der Quellcodes verzichtet.

```

/***** Eintrag.java *****/

/** Implementation eines Eintrags fuer die VerweisListe und den VerweisKeller
 */

public class Eintrag {

    Object inhalt;           // Inhalt des Eintrags
    Eintrag next;           // Verweis auf naechsten Eintrag

}

```

```
/****** VerweisListe.java *****/
/** Implementation des Interface Liste mithilfe von Verweisen
 */

public class VerweisListe implements Liste {

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

    public VerweisListe() {       // kreiert eine leere Liste
        pos = anf = new Eintrag();
        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() {         // rueckt an Anfang
        pos = anf;
    }

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

    public Object elem() {        // liefert aktuelles Element
        if (endpos()) throw new
            RuntimeException(" am Ende der Liste ");
        return pos.next.inhalt;
    }

    public void insert(Object x) { // fuegt ListenEintrag ein
        Eintrag hilf = new Eintrag(); // 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(" am Ende der Liste ");
        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 abarbeiten
            IO.println(l.elem());             // Student drucken
            l.advance();                     // einmal vorruecken
        }

        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` (Prinzip LIFO: Last in, first out):

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

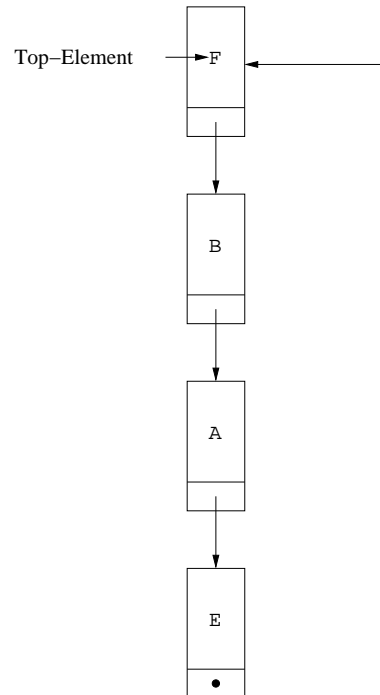
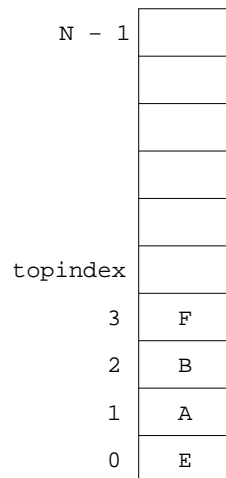
Semantik der Kelleroperationen:

- A1) Ein neu konstruierter `Keller` ist leer.
- A2) Nach einer `Push`-Operation ist ein `Keller` nicht leer.
- A3) Nach einer `Push-Pop`-Operation ist der `Keller` unverändert.
- A4) Nach der `Push`-Operation mit dem Element x liefert die `Top`-Operation das Element x .

```

/***** 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 ***** */

/** Implementation eines Kellers mithilfe von Verweisen
 */

public class VerweisKeller implements Keller {

    private Eintrag 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
        Eintrag hilf = new Eintrag();   // auf den Keller
        hilf.inhalt = x;
        hilf.next    = top;
        top          = hilf;
    }

    public Object top() {               // liefert Top-Element
        if (empty()) throw new
            RuntimeException(" Keller ist leer ");
        return top.inhalt;
    }

    public void pop() {                 // entfernt Top-Element
        if (empty()) throw new
            RuntimeException(" Keller ist leer ");
        top = top.next;
    }
}
```

Die folgenden beiden Beispiele zeigen, dass zum Speichern von Werten aus primitiven Datentypen zunächst das Verpacken der Werte mithilfe der bereits erwähnten *Wrapperklassen* erforderlich ist. So wird durch den Konstruktor `new Integer(i)` ein Integer-Objekt erzeugt mit dem (nicht mehr veränderbaren) Wert `i`. Der Wert dieses Objekts kann über die Methode `intValue()` ermittelt werden.

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

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

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

            switch (zeile[i]) {

                case '(': // oeffnende Klammer
                case '[': k.push(new Character(zeile[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");
    }
}

```

Um die Lesbarkeit der Programme zu erhöhen, soll nun das Einpacken und Auspacken der Wrapper-Klassen-Objekte gekapselt werden. Hierzu leiten wir vom Interface `Keller` das Interface `CharKeller` ab und von der Klasse `VerweisKeller` leiten wir die Klasse `VerweisCharKeller` ab. Erforderlich wird eine neue Methode `ctop()`, da Java nicht zwei identische Signaturen mit unterschiedlichem Rückgabewert erlaubt. Getestet wird die Implementation in der Klasse `Postfix`, welche einen eingelesenen Infix-Ausdruck in die entsprechende Postfix-Notierung überführt.

```

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

public interface CharKeller extends Keller {
    public void push(char c);    // legt char auf den Keller
    public char ctop();        // liefert oberstes Kellerelement als char
}

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

public class VerweisCharKeller extends VerweisKeller implements CharKeller {

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

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

/***** Postfix.java *****/

import AlgoTools.IO;

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

public class Postfix {

    public static void main(String[] argv) {

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

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

```

```

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

    c = infix[i]; // aktuelles Zeichen
    switch (c) {

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

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

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

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

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

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

```

9.3 Schlange

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

Schnittstelle des ADT Schlange (Prinzip FIFO: First in, First out):

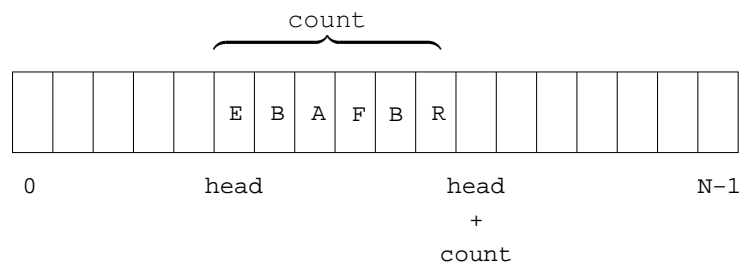
```
empty : Schlange      → boolean   liefert true, falls Schlange leer ist, false sonst
enq   : Schlange × Objekt → Schlange  fügt Element hinten ein
front : Schlange      → Objekt    liefert vorderstes Element
deq   : Schlange      → Schlange  entfernt vorderstes Element
```

```

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

```

Konzept zur Implementation einer Schlange mit einem Array:




```
/* ***** ArraySchlange.java ***** */

/** 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(" Schlange ist voll ");
        inhalt[(head+count)%inhalt.length] = x; // Element einfuegen
        count++;                      // Anzahl inkrementieren
    }

    public void deq() {               // Entfernt vorderstes Element
        if (empty()) throw new RuntimeException(" Schlange ist leer ");
        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(" Schlange ist leer ");
        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 s = new ArraySchlange(100);           // konstruiere Schlange mit
                                                         // Platz fuer 100 Objekte

        String eingabe;

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

        do {

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

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

                s.enq(eingabe);                          // fuege in Schlange ein
                IO.println(" wird eingefuegt.");

            } else {                                    // falls EINGABE == RETURN

                if (!s.empty()) {
                    IO.println(s.front()+" wird entfernt"); // gib Frontelement aus
                    s.deq();                               // entferne Frontelement
                }

            }

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

        IO.println("Schlange ist jetzt leer.");
    }
}
```

9.4 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

`value` : Baum → Objekt liefert Wurzelement

`left` : Baum → Baum liefert linken Teilbaum

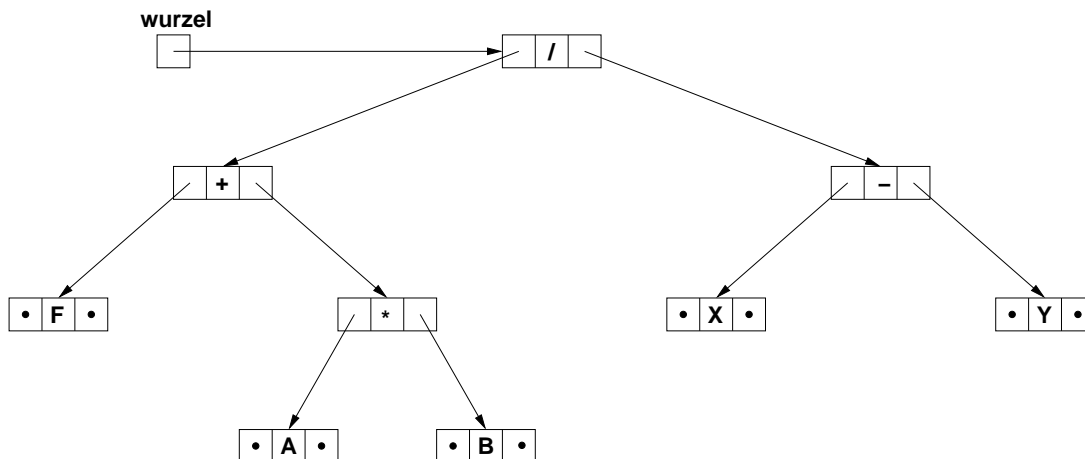
`right` : Baum → Baum liefert rechten Teilbaum

```
/* ***** Baum.java ***** */
/** Interface Baum mit 4 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
}
```

Hinweis: Um Bäume nicht nur auszulesen, sondern auch verändern zu können, müssen die (noch zu definierenden) Konstruktoren der Implementation verwendet werden.

Konzept zur Implementation eines Baumes mit Verweisen

Ein Baum besteht aus einer Menge von Knoten. Der `VerweisBaum` enthält daher einen Verweis auf den Wurzelknoten oder einen `null`-Verweis, wenn der Baum leer ist. Ein Knoten enthält neben einem Verweis auf den Inhalt jeweils einen Verweis auf den linken und rechten Sohn, sofern sie vorhanden sind.



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))

```

/***** Knoten.java *****/

/** Klasse Knoten mit einem Konstruktor
 */

public class Knoten {

    Object inhalt;                                // Knoteninhalt
    Knoten links, rechts;                       // linker, rechter Teilbaum

    public Knoten(Object x) {                   // konstruiere Knoten
        inhalt = x;                             // mit Inhalt x
        links = rechts = null;                 // Nullverweis fuer die Soehne
    }
}
  
```

```

/***** VerweisBaum.java *****/

/** Klasse VerweisBaum mit vier Konstruktoren und vier Methoden.
 *
 * Ein VerweisBaum enthaelt einen Verweis auf den Wurzelknoten, der auf weitere
 * Knoten verweisen kann, die den linken und rechten Teilbaum repraesentieren.
 */

public class VerweisBaum implements Baum {

    Knoten wurzel; // Wurzel des Baums

    public VerweisBaum() { // konstruiert einen leeren Baum
        wurzel = null; // Wurzel verweist auf nichts
    }

    public VerweisBaum(Object x) { // konstruiert ein Blatt
        wurzel = new Knoten(x); // lege Knoten mit Inhalt x an
    }

    public VerweisBaum(VerweisBaum l, Object x, VerweisBaum r) { // konstr. Baum
        wurzel = new Knoten(x); // lege Knoten mit Inhalt x an
        if (l != null) // wenn l nicht null, setze
            wurzel.links = l.wurzel; // wurzel von l als linken Sohn
        if (r != null) // wenn r nicht null, setze
            wurzel.rechts = r.wurzel; // wurzel von r als rechten Sohn
    }

    private VerweisBaum(Knoten k) { // konstruiert einen Baum
        wurzel = k; // aus uebergebenem Knoten
    } // (nur fuer internen Gebrauch)

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

    public Object value() { // liefert Element in der Wurzel
        if (empty())
            throw new RuntimeException(" Baum ist leer ");
        return wurzel.inhalt;
    }

    public Baum left() { // liefert linken Teilbaum
        if (empty())
            throw new RuntimeException(" Baum ist leer ");
        return new VerweisBaum(wurzel.links); // Erzeuge Baum, der linken Sohn
    } // als Wurzelknoten enthaelt

    public Baum right() { // liefert rechten Teilbaum
        if (empty())
            throw new RuntimeException(" Baum ist leer ");
        return new VerweisBaum(wurzel.rechts); // Erzeuge Baum, der rechten Sohn
    } // als Wurzelknoten enthaelt
}

```

```

/***** 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 iterativ organisierte 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()) // lege uebergebenen
            k.push(wurzel); // 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 iterativ organisierte 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()) // lege uebergebenen
            s.enq(wurzel); // 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(new Character('A'));
        VerweisBaum b = new VerweisBaum(new Character('B'));
        VerweisBaum m = new VerweisBaum(a, new Character('*'), b);
        VerweisBaum f = new VerweisBaum(new Character('F'));
        VerweisBaum p = new VerweisBaum(f, new Character('+'), m);
        VerweisBaum x = new VerweisBaum(new Character('X'));
        VerweisBaum y = new VerweisBaum(new Character('Y'));
        VerweisBaum n = new VerweisBaum(x, new Character('-'), y);
        VerweisBaum d = new VerweisBaum(p, new Character('/'), 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

        VerweisBaum b, l, r; // Baeume
        Object x; // Objekt
        char c; // Zeichen

        Keller k = new VerweisKeller(); // Keller

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

            c = ausdruck[i]; // besorge naechstes Zeichen
            x = new Character(c); // erzeuge Objekt

            if (c!='+' && c!='-' && c!='*' && c!='/') // falls c ein Operand ist
                b = new VerweisBaum(x); // erzeuge Blatt
            else { // ansonsten ist c Operator
                r = (VerweisBaum) k.top(); k.pop(); // besorge rechten Sohn
                l = (VerweisBaum) k.top(); k.pop(); // besorge linken Sohn
                b = new VerweisBaum(l,x,r); // erzeuge VerweisBaum
            }
            k.push(b); // lege Verweisbaum 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() {
        if (k.empty()) throw new RuntimeException("in nextElement: Keller leer");
        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

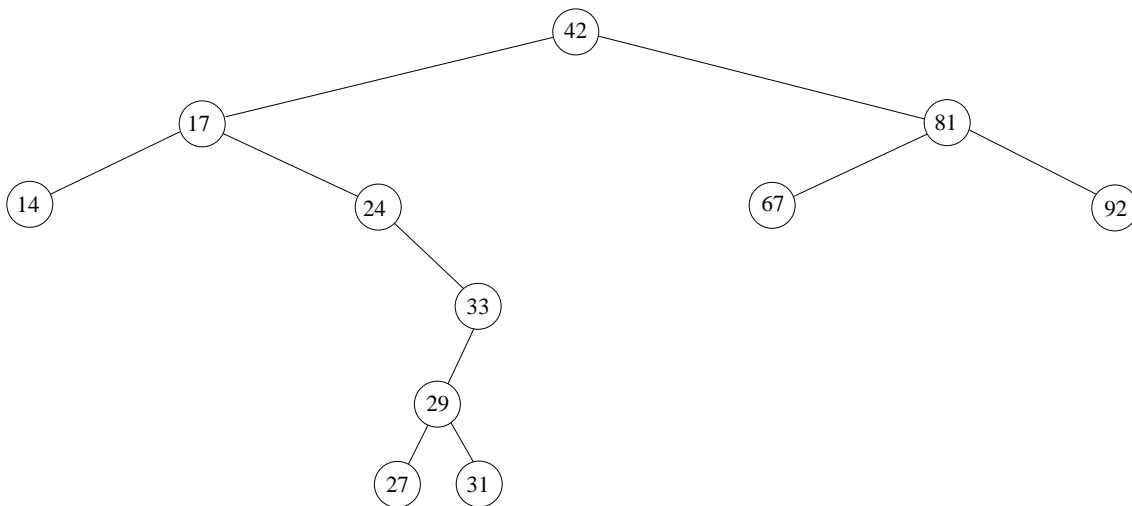
        IO.println("Umwandlung in Prefix-Notation:");
        Enumeration e = new PreorderTraverse(wurzel); // erzeuge Enumeration
        while (e.hasMoreElements()) { // solange Elemente da
            Object x = e.nextElement(); // besorge Element
            IO.println(x); // gib Element aus
        } // Obacht: nicht alle
        IO.println(); // Objekte eignen sich
    } // zur Darstellung !
}

```

9.5 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 .

Beispiel für einen binären Suchbaum



Suchbaumoperationen

Die oben genannte Suchbaumeigenschaft erlaubt eine effiziente Umsetzung der Suchbaumoperationen **lookup**, **insert** und **delete**. *lookup* sucht nach einem gegebenen Objekt durch systematisches Absteigen in den jeweils zuständigen Teilbaum. *insert* sucht zunächst die Stelle im Baum, bei der das einzufügende Objekt platziert sein müsste und hängt es dann dort ein. *delete* sucht zunächst die Stelle, an der das zu löschende Objekt vermutet wird und klinkt es dann aus dem Baum aus. Je nach Zahl der Söhne ist dieser Vorgang unterschiedlich kompliziert (siehe nächste Seite).

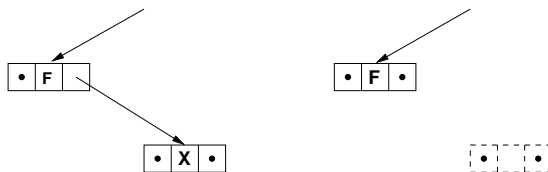
Komplexität

Der Aufwand der Suchbaumoperationen ist jeweils proportional zur Anzahl der Knoten auf dem Wege von der Wurzel bis zu dem betroffenen Knoten.

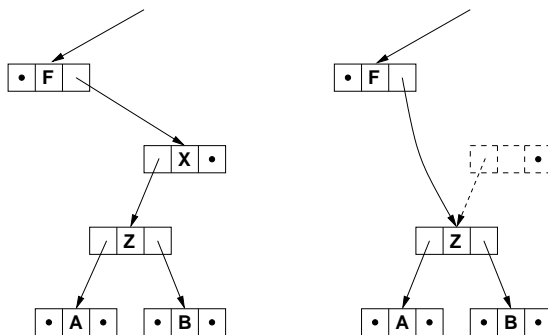
- Best case: Hat jeder Knoten 2 Söhne, so hat der Baum bei Höhe h $n = 2^h - 1$ Knoten. Die Anzahl der Weg-Knoten ist $h = \log(n)$.
- Worst case: Werden die Elemente sortiert eingegeben, so entartet der Baum zur Liste, der Aufwand beträgt dann $O(n)$.
- Average case: Für n zufällige Schlüssel beträgt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

In den folgenden drei Grafiken wird mit einem schwarzen Punkt der null-Verweis visualisiert.
 Sei x das Element in dem zu löschenden Knoten des Suchbaums.

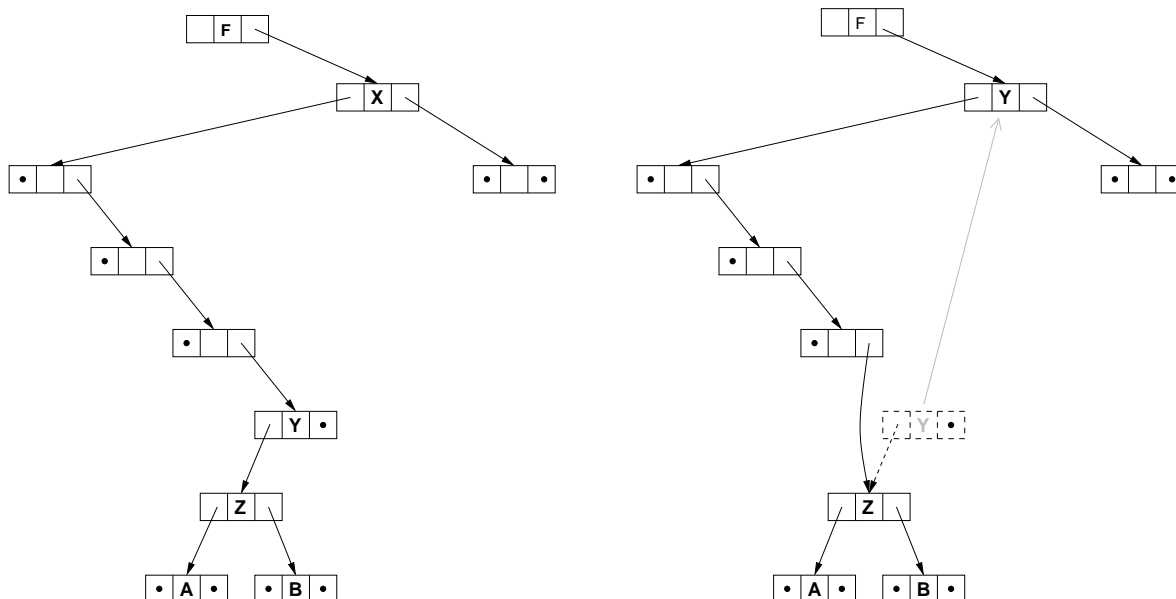
Löschen eines Knotens ohne Söhne



Löschen eines Knotens mit einem Sohn



Löschen eines Knotens mit zwei Söhnen




```

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

/** Implementation eines binären Suchbaums über Comparable-Objekten.
 *
 * Bereitgestellt werden die im Interface Menge angekuendigten Methoden
 * lookup, insert und delete als öffentliche Methoden.
 *
 * Die Methode delete verwendet zusätzlich noch die private Methode findMax.
 */

public class SuchBaum extends VerweisBaum implements Menge {

    public Comparable lookup(Comparable x) {
        // sucht x im SuchBaum
        Knoten k = wurzel; // beginne bei wurzel
        while (k != null) { // solange ein Knoten vorliegt
            if (x.compareTo(k.inhalt) < 0) // vergleiche x mit Knoteninhalt
                k = k.links; // steige je nach Vergleich in den
            else if (x.compareTo(k.inhalt) > 0) // linken oder rechten Teilbaum hinab
                k = k.rechts;
            else // oder (d.h. bei Gleichheit)
                return (Comparable)k.inhalt; // liefere das Objekt zurueck
        }
        // Wenn kein Knoten mehr vorhanden
        return null; // (letztes k ist Blatt) liefere null
    }

    public boolean insert(Comparable x) { // versucht x einzufuegen

        if (wurzel == null) { // Wenn Baum leer
            wurzel = new Knoten(x); // fuege neuen Knoten ein
            return true; // melde Erfolg
        }
        else { // andernfalls
            Knoten vater = null;
            Knoten k = wurzel; // Hilfsknoten
            while (k != null) { // solange ein Knoten vorliegt
                vater = k; // merke aktuellen Knoten als Vater
                if (x.compareTo(k.inhalt) < 0) // falls x < k.inhalt
                    k = k.links; // steige nach links ab
                else
                    if (x.compareTo(k.inhalt) > 0) // falls x > k.inhalt
                        k = k.rechts; // steige nach rechts ab
                    else // falls x und k.inhalt gleich,
                        return false; // melde Misserfolg
            }

            if (x.compareTo(vater.inhalt) < 0) // falls x kleiner als vater
                vater.links = new Knoten(x); // haenge neuen Knoten links an
            else // falls x groesser als vater
                vater.rechts = new Knoten(x); // haenge neuen Knoten rechts an
            return true; // melde Erfolg
        }
    }
}

```

```

public boolean delete(Comparable x){ // loescht x aus SuchBaum

    Knoten vater = null, // Hilfsknoten vater und sohn
           sohn = wurzel; // sohn wird mit wurzel initialisiert

                                // suche zu loeschendes Element:
while (sohn != null && // solange sohn noch vorhanden und
       x.compareTo(sohn.inhalt) !=0){ // x nicht gefunden
    vater = sohn; // gehe eine Ebene tiefer:
    if (x.compareTo(sohn.inhalt)<0) // sohn wird zum neuen vater,
        sohn = sohn.links; // linker oder rechter sohn des alten
    else // sohns wird neuer sohn, je nachdem in
        sohn = sohn.rechts; // welchem Teilbaum x sein muesste
}

                                // finde Ersatzknoten fuer Element:
if (sohn != null) { // wenn sohn != null, wurde x gefunden
    Knoten ersatzKnoten; // Ersatzknoten, ersetzt sohn im Baum

    if (sohn.links == null) // wenn linker Sohn von sohn leer,
        ersatzKnoten = sohn.rechts; // Ersatzknoten ist re. Sohn von sohn
    else if (sohn.rechts == null) // wenn rechter Sohn von sohn leer,
        ersatzKnoten = sohn.links; // Ersatzknoten ist li. Sohn von sohn
    else { // wenn beide Soehne vorhanden,
        ersatzKnoten = sohn; // Ersatzknoten ist sohn selbst
                                // hole groesstes Elem. im li. Teilbaum
        Comparable tmp = (Comparable)(findMax(ersatzKnoten.links).inhalt);
        delete(tmp); // loesche dieses Element aus dem Baum
        ersatzKnoten.inhalt = tmp; // setze dieses Element in ersatzKnoten
    }

                                // Setze ersatzKnoten in Baum ein:
    if (vater == null) // Sonderfall: Element war in wurzel
        wurzel = ersatzKnoten; // ersatzKnoten ist neuer Wurzelknoten
    else if (x.compareTo(vater.inhalt) < 0) // andernfalls untersuche, ob
        vater.links = ersatzKnoten; // ersatzKnoten neuer linker oder
    else // rechter Sohn des Vaters wird
        vater.rechts = ersatzKnoten;
    return true; // melde Erfolg
}
else // Element wurde nicht gefunden,
    return false; // melde Misserfolg
}

private Knoten findMax(Knoten t) { // liefert Knoten mit maximalen Element

    while (t.rechts != null) // solange rechter Sohn vorhanden,
        t = t.rechts; // steige mit t in den re. Teilbaum ab
    return t; // liefere t
}
}

```



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

/** Testet den SuchBaum mit Character-Objekten. */

public class SuchBaumTest {

    public static void main(String[] argv) {

        SuchBaum s = new SuchBaum();

                                                // Elemente in SuchBaum einfüegen
        char[] eingabe = IO.readChars("Bitte Zeichen fuer insert: ");

        Comparable x = new Character('?');
        s.insert(x);

        for (int i = 0; i < eingabe.length; i++) {

            if (s.insert(new Character(eingabe[i])))
                IO.println(eingabe[i] + " eingefuegt");
            else
                IO.println(eingabe[i] + " konnte nicht eingefuegt werden");
        }

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

                                                // Elemente im SuchBaum suchen
        eingabe = IO.readChars("Bitte Zeichen fuer lookup: ");

        for (int i = 0; i < eingabe.length; i++) {

            Character c = (Character) s.lookup(new Character(eingabe[i]));

            if (c == null)
                IO.println(eingabe[i] + " konnte nicht gefunden werden");
            else
                IO.println(c + " gefunden");
        }

                                                // Elemente aus dem SuchBaum loeschen
        eingabe = IO.readChars("Bitte Zeichen fuer delete: ");

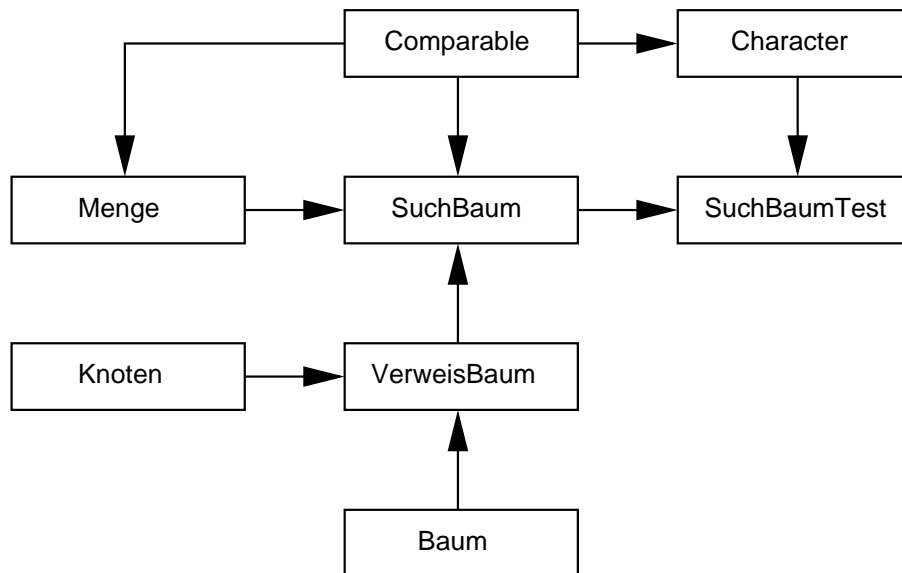
        for (int i = 0; i < eingabe.length; i++) {

            if (s.delete(new Character(eingabe[i])))
                IO.println(eingabe[i] + " geloescht");
            else
                IO.println(eingabe[i] + " konnte nicht geloescht werden");

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

Abhängigkeiten

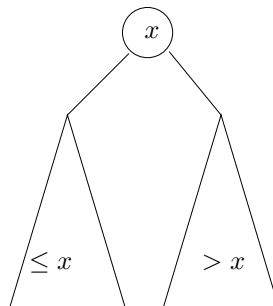
Folgende Grafik verdeutlicht die Abhängigkeiten der zum Suchen von `Character` verwendeten Klassen und Interfaces:



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.

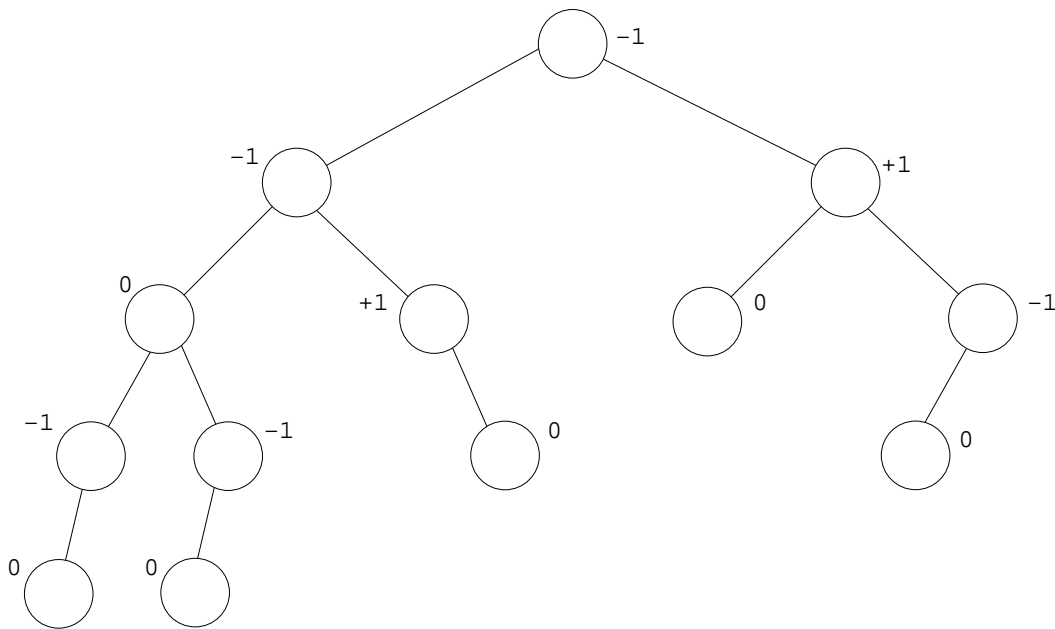
Beim Löschen: Zähler herunterzählen, sofern mehrfach da, sonst entfernen.

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

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 und ggf. wiederhergestellt werden. Hierzu werden längs des Weges vom eingefügten bzw. gelöschten Element bis zur Wurzel die Balance-Werte abgefragt und durch so genannte *Rotationen* repariert.

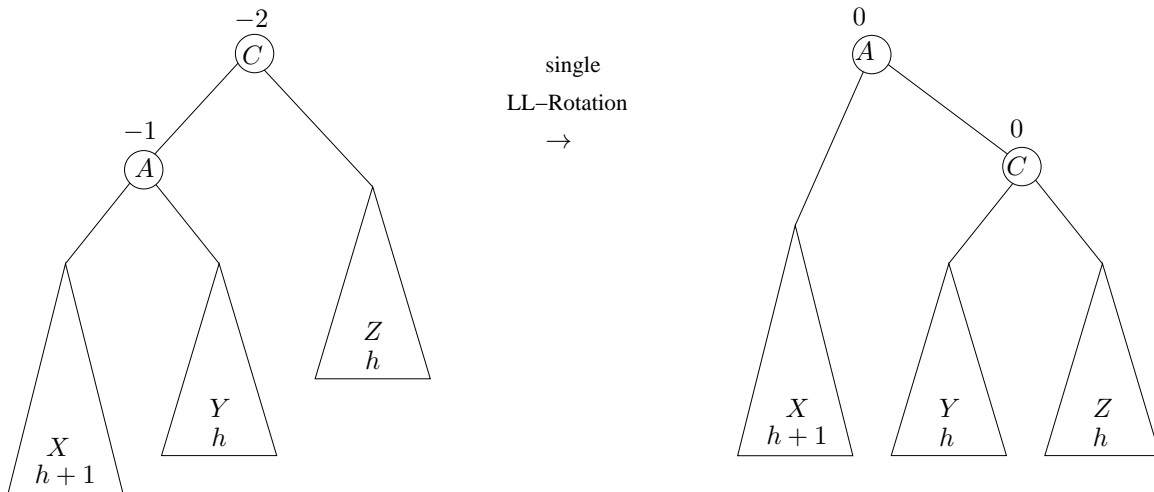
Während das Einfügen eines einzelnen Schlüssels *höchstens eine* Rotation erfordert, kann das Löschen eine Rotation für *jeden* Knoten entlang des Weges zur Wurzel verursachen.

Rotationen für AVL-Baum bei linksseitigem Übergewicht

Single LL-Rotation

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

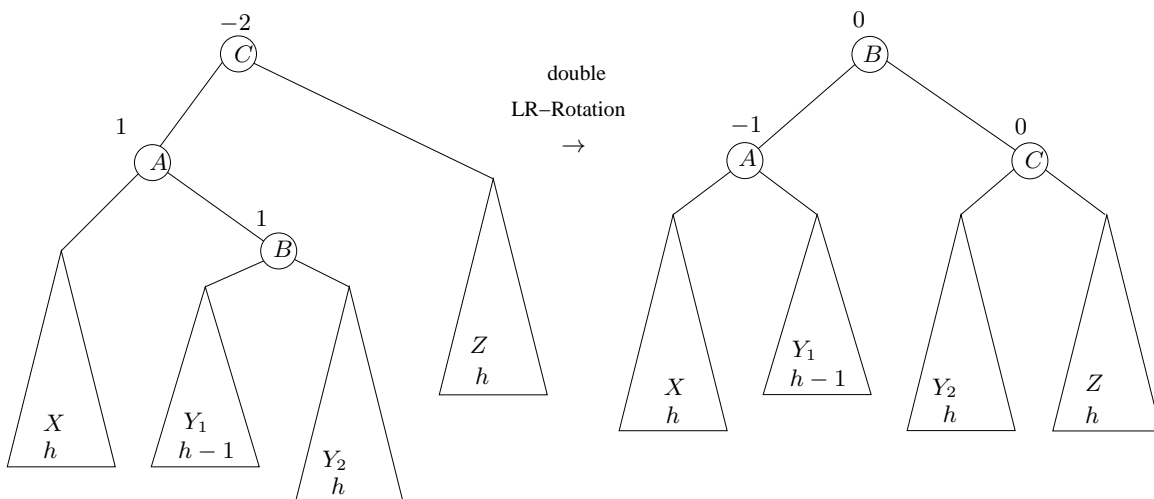
Bei Löschen im Teilbaum Z : Höhe des gesamten Baums vorher und nachher gleich oder nachher um eins kleiner.



Double LR-Rotation

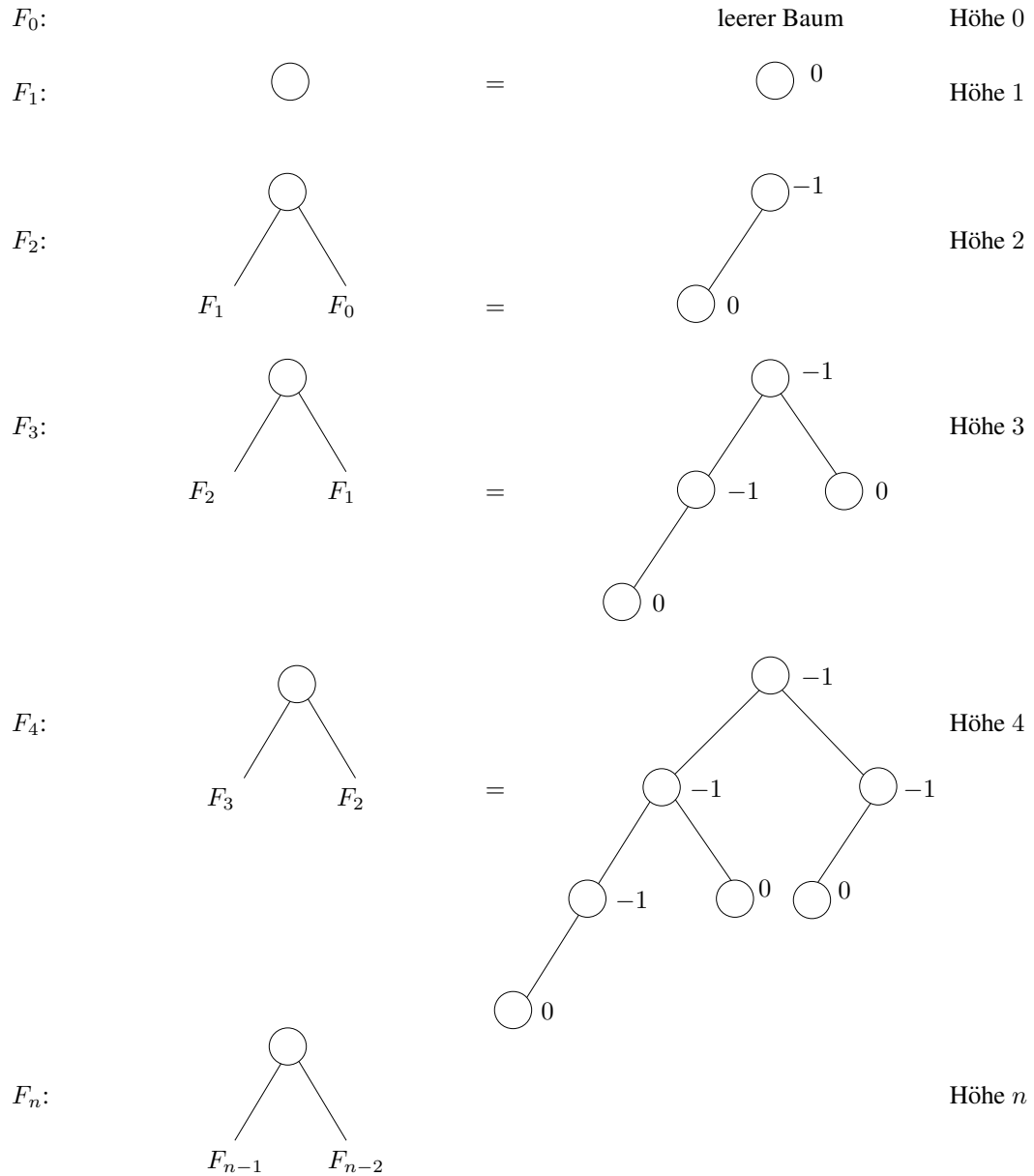
Bei Einfügen in Teilbaum Y_1 oder Y_2 : Höhe des gesamten Baums vorher und nachher gleich.

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 Eintrag erfordert die größtmögliche Zahl von Rotationen: Im Fibonacci-Baum F_n werden dabei $\lfloor \frac{n-1}{2} \rfloor$ LL-Rotationen durchgeführt.

```

/***** AVLKnoten.java *****/

/** Klasse AVLKnoten mit einem Konstruktor.
 *
 * Die Klasse AVLKnoten erweitert die vorhandene Klasse Knoten um das Datenfeld
 * balance, die im AVLBaum fuer jeden Knoten abgespeichert wird.
 */

public class AVLKnoten extends Knoten {

    public int balance;                // Balance

    public AVLKnoten(Object x) {       // erzeugt einen Knoten mit Inhalt x
        super(x);                     // Konstruktoraufruf von Knoten
    }

    public String toString() {         // fuer Ausgabe: Inhalt(Balance)
        return inhalt + "(" + balance + ")";
    }
}

/***** 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 {

    private class Status {            // Innere Klasse zur Uebergabe eines
        // Status in der Rekursion
        private boolean unbalanciert; // unbalanciert ist true, wenn beim
    }                                 // Einfuegen ein Sohn groesser geworden
        // ist.
        // simuliertes Call-by-Reference

    public static void printAVLBaum(Baum b, int tiefe) {

        if (!b.empty()) {             // Wenn Baum nicht leer:
            printAVLBaum(b.right(), tiefe+1); // rechten Teilbaum ausgeben
            for (int i=0; i<tiefe; i++) // entsprechend der Rekursions-
                IO.print("    ");      // tiefe einruecken
            IO.println(((VerweisBaum)b).wurzel); // Wurzel und Balance ausgeben
            printAVLBaum(b.left(), tiefe+1); // linken Teilbaum ausgeben
        }
    }
}

```

```

public boolean insert(Comparable x) { // fuegt x in den AVLbaum ein: true,
                                     // wenn erfolgreich, sonst false.
                                     // Kapselt die Methode insertAVL
    return insertAVL(x, (AVLKnoten)wurzel, null, new Status());
}

                                     // tatsaechliche rekursive Methode zum
                                     // Einfuegen
private boolean insertAVL(Comparable x, AVLKnoten k, AVLKnoten v, Status s) {

    boolean eingefuegt;

    if (k == null) { // Unter einem Blatt:
                    // Hier kann eingefuegt werden.
        if (v == null) // kein Vater:
            wurzel = new AVLKnoten(x); // Einfuegen in der Wurzel
        else { // Vater vorhanden:
            if (x.compareTo(v.inhalt) < 0) // Element mit Inhalt von Vater vergl.
                v.links = new AVLKnoten(x); // und als entsprechend als linken oder
            else // rechten Sohn einfuegen
                v.rechts = new AVLKnoten(x);
            s.unbalanciert = true; // Teilbaum wurde groesser
        }
        return true; // Einfuegen erfolgreich
    }

    else if (x.compareTo(k.inhalt) == 0) { // Element schon im AVLbaum vorhanden
        return false; // Einfuegen nicht erfolgreich
    }

    else {
        if (x.compareTo(k.inhalt) < 0) { // Element ist kleiner als Knoteninhalt
            eingefuegt = insertAVL(x, (AVLKnoten)k.links, k, s);
                                     // Setze Suche im linken Teilbaum fort

            if (s.unbalanciert) { // Falls linker Teilbaum hoeher wurde

                if (k.balance == 1) { // alte Unausgeglichenheit ausgeglichen
                    k.balance = 0; // => neue Balance = 0
                    s.unbalanciert = false; // Teilbaum ist jetzt ausgeglichen
                }

                else if (k.balance == 0) // Noch keineRotation noetig
                    k.balance = -1; // Balance wird angeglichen

                else { // Balance wird angeglichen
                    if ((AVLKnoten)k.links.balance == -1)
                        rotateLL(k);
                    else
                        rotateLR(k);
                    s.unbalanciert = false; // Teilbaum ist danach ausgeglichen
                }
            }
        }
    }
}

```

```
else { // Element groesser als Knoteninhalt

    eingefuegt = insertAVL(x, (AVLKnoten)k.rechts, k, s);
                                // Setze Suche im rechten Teilbaum fort

    if (s.unbalanciert) { // Falls recht. Teilbaum groesser wurde
        if (k.balance == -1) { // alte Unausgeglichenheit ausgeglichen
            k.balance = 0; // => neue Balance = 0
            s.unbalanciert = false; // Teilbaum ist jetzt ausgeglichen
        }
        else if (k.balance == 0) // Noch keineRotation noetig
            k.balance = 1; // Balance wird angeglichen

        else { // Balance wird angeglichen
            if (((AVLKnoten)k.rechts).balance == 1)
                rotateRR(k);
            else
                rotateRL(k);
            s.unbalanciert = false; // Teilbaum ist danach ausgeglichen
        }
    }
}
return eingefuegt; // true, falls im linken oder rechten
} // Teilbaum eingefuegt, sonst false
}
```



```
private void rotateLL(AVLKnoten k) {

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

    AVLKnoten a1 = (AVLKnoten)k.links;    // Merke linken
    AVLKnoten a2 = (AVLKnoten)k.rechts;    // und rechten Teilbaum

                                        // Idee: Inhalt von a1 in die Wurzel k
    k.links = a1.links;                  // Setze neuen linken Sohn
    k.rechts = a1;                       // Setze neuen rechten Sohn
    a1.links = a1.rechts;                // Setze dessen linken und
    a1.rechts = a2;                      // rechten Sohn

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

    ((AVLKnoten)k.rechts).balance = 0;   // rechter teilbaum balanciert
    k.balance = 0;                       // Wurzel k balanciert
}

private void rotateLR(AVLKnoten k) {

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

    AVLKnoten a1 = (AVLKnoten)k.links;    // Merke linken
    AVLKnoten a2 = (AVLKnoten)a1.rechts;  // und dessen rechten Teilbaum

                                        // Idee: Inhalt von a2 in die Wurzel k
    a1.rechts = a2.links;                // Setze Soehne von a2
    a2.links = a2.rechts;
    a2.rechts = k.rechts;
    k.rechts = a2;                       // a2 wird neuer rechter Sohn

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

    ((AVLKnoten)k.links).balance = (a2.balance == 1) ? -1 : 0;
                                        // neue Balance fuer linken Sohn

    ((AVLKnoten)k.rechts).balance = (a2.balance == -1) ? 1 : 0;
                                        // neue Blance fuer rechten Sohn

    k.balance = 0;                       // Wurzel k ist ausgeglichen
}
```

```

private void rotateRR(AVLKnoten k) {

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

    AVLKnoten a1 = (AVLKnoten)k.rechts; // Merke rechten
    AVLKnoten a2 = (AVLKnoten)k.links;  // und linken Teilbaum

                                     // Idee: Inhalt von a1 in die Wurzel k
    k.rechts = a1.rechts;             // Setze neuen rechten Sohn
    k.links = a1;                      // Setze neuen linken Sohn
    a1.rechts = a1.links;             // Setze dessen rechten und
    a1.links = a2;                    // linken Sohn

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

    ((AVLKnoten)k.links).balance = 0; // linker teilbaum balanciert
    k.balance = 0;                   // Wurzel k balanciert
}

private void rotateRL(AVLKnoten k) {

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

    AVLKnoten a1 = (AVLKnoten)k.rechts; // Merke rechten
    AVLKnoten a2 = (AVLKnoten)a1.links; // und dessen linken Teilbaum

                                     // Idee: Inhalt von a1 in die Wurzel k
    a1.links = a2.rechts;             // Setze Soehne von a2
    a2.rechts = a2.links;
    a2.links = k.links;
    k.links = a2;                     // a2 wird neuer linker Sohn

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

    ((AVLKnoten)k.rechts).balance = (a2.balance == -1) ? 1 : 0;
                                     // neue Balance fuer rechten Sohn

    ((AVLKnoten)k.links).balance = (a2.balance == 1) ? -1 : 0;
                                     // neue Balance fuer rechten Sohn

    k.balance = 0;                   // Wurzel k ist ausgeglichen
}

```

```

public boolean delete(Comparable x) { // loescht x im AVLBaum: true, wenn
    // erfolgreich, sonst false
    // Kapselt die Methode deleteAVL
    return deleteAVL(x, (AVLKnoten)wurzel, null, new Status());
}

// tatsaechliche rekursive Methode zum
// Loeschen
private boolean deleteAVL(Comparable x, AVLKnoten k, AVLKnoten v, Status s) {

    boolean geloescht; // true, wenn geloescht wurde

    if (k == null) // Unterhalb eines Blattes:
        return false; // Element nicht gefunden, es wurde
        // nicht geloescht

    else if (x.compareTo(k.inhalt) < 0) { // Element x kleiner als Knoteninhalt

        geloescht = deleteAVL(x, (AVLKnoten)k.links, k, s);
        // Setze Suche im linken Teilbaum fort

        if (s.unbalanciert) // Falls linker Teilbaum kleiner
            balance1(k, s); // geworden ist, gleiche Hoehe an

        return geloescht; // true, falls im linken Teilbaum
    } // geloescht wurde

    else if (x.compareTo(k.inhalt) > 0) { // Element x groesser als Knoteninhalt

        geloescht = deleteAVL(x, (AVLKnoten)k.rechts, k, s);
        // Setze Suche im rechten Teilbaum fort

        if (s.unbalanciert) // Falls rechter Teilbaum kleiner
            balance2(k, s); // geworden ist, gleiche Hoehe an

        return geloescht; // true, falls im rechten Teilbaum
    } // geloescht wurde

    else { // Knoten k enthaelt Element x
        // Wenn k keine Soehne hat
        if (k.rechts == null && k.links == null) {

            if (v == null) // Hat k keinen Vater, ist k die Wurzel
                wurzel = null; // Setze Wurzel auf null
            else { // k hat Vater

                if (x.compareTo(v.inhalt) < 0) // Ist k linker Sohn,
                    v.links = null; // loesche linken Sohn von Vater v
                else // Ist k rechter Sohn,
                    v.rechts = null; // loesche rechten Sohn von v
                s.unbalanciert = true; // Hoehe hat sich geaendert
            }
        }
    }
}

```

```

else if (k.rechts == null) { // Wenn k nur linken Sohn
    if (v != null) { // und einen Vater hat
        if (x.compareTo(v.inhalt) < 0) // setze ihn als neuen linken oder
            v.links = k.links; // rechten Sohn des Vaters
        else
            v.rechts = k.links;

        s.unbalanciert = true; // Hoehe hat sich geaendert
    }
    else // Wenn k keinen Vater hat war k Wurzel
        wurzel = k.links; // Dann wird linker Sohn neue Wurzel
}
else if (k.links == null) { // Wenn k nur rechten Sohn
    if (v != null) { // und einen Vater hat,
        if (x.compareTo(v.inhalt) < 0) // setze ihn als neuen linken oder
            v.links = k.rechts; // rechten Sohn des Vaters
        else
            v.rechts = k.rechts;

        s.unbalanciert = true; // Hoehe hat sich geaendert
    }
    else // Wenn k keinen Vater hat war k Wurzel
        wurzel = k.rechts; // Dann wird rechter Sohn neue Wurzel
}
else { // k hat zwei Soehne
    // rufe im li. Sohn del() auf
    k.inhalt = del((AVLKnoten)k.links, k, s);
    if (s.unbalanciert) // Falls linker Teilbaum kleiner
        balancel(k, s); // geworden ist, gleiche Hoehe an
}
return true; // es wurde geloescht
}
}

private Comparable del(AVLKnoten k, AVLKnoten v, Status s) {
    // Suche Ersatz fuer gel. Element
    Comparable ersatz; // Ersatz-Element

    if (k.rechts != null) { // Wenn rechter Sohn vorhanden,
        // suche dort weiter und loesche
        ersatz = del((AVLKnoten)k.rechts, k, s);
        if (s.unbalanciert) // Falls rechter Teilbaum kleiner
            balance2(k, s); // geworden ist, gleiche Hoehe an
    }
    else { // kein rechter Sohn von k vorhanden

        ersatz = (Comparable)k.inhalt; // Cast, da k.inhalt vom Typ Object

        if (((Comparable)k.inhalt).compareTo(v.inhalt) > 0)
            v.rechts = k.links; // setze linken Sohn von k an die
        else // Stelle von k
            v.links = k.links;
        s.unbalanciert = true; // Hoehe hat sich geaendert
    }
    return ersatz; // Gib Ersatz-Element zurueck
}

```

```

}
private void balance1(AVLKnoten k, Status s) {
    // Unbalance, weil linker Ast kuerzer
    switch (k.balance) {
        case -1: // Balance geaendert, Hoehe nicht
            k.balance = 0; // ausgeglichen
            break;
        case 0: // Ausgeglichen
            k.balance = 1;
            s.unbalanciert = false;
            break;
        case 1: // Ausgleichen (Rotation) notwendig
            int b = ((AVLKnoten)k.rechts).balance; // Balance des linken Sohnes
            if (b >= 0) {
                rotateRR(k);
                if (b == 0) {
                    k.balance = -1; // Gleiche Balancen an
                    ((AVLKnoten)k.links).balance = 1;
                    s.unbalanciert = false;
                }
            }
            else
                rotateRL(k);
    }
}

private void balance2(AVLKnoten k, Status s) {
    // Unbalance, weil rechter Ast kuerzer
    switch (k.balance) {
        case 1: // Balance geaendert, Hoehe nicht
            k.balance = 0; // ausgeglichen
            break;
        case 0: // Ausgeglichen
            k.balance = -1; // Ausgeglichen
            s.unbalanciert = false;
            break;
        case -1: // Ausgleichen (Rotation) notwendig
            int b = ((AVLKnoten)k.links).balance; // Balance des rechten Sohnes
            if (b <= 0) {
                rotateLL(k);
                if (b == 0) {
                    k.balance = 1; // Gleiche Balancen an
                    ((AVLKnoten)k.rechts).balance = -1;
                    s.unbalanciert = false;
                }
            }
            else
                rotateLR(k);
    }
}
}

```

```
/****** 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();

                                // In den AVLBaum einfuegen
        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:");
            AVLBaum.printAVLBaum(b, 0);
            k = IO.readChar("Char in AVL-Baum einfuegen (Loeschen: \\n): ");
        }

        IO.println();

                                // Aus dem AVLBaum loeschen
        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:");
            AVLBaum.printAVLBaum(b, 0);
            k = IO.readChar("Char im AVL-Baum loeschen (Abbruch: \\n): ");
        }
    }
}
```

Kapitel 10

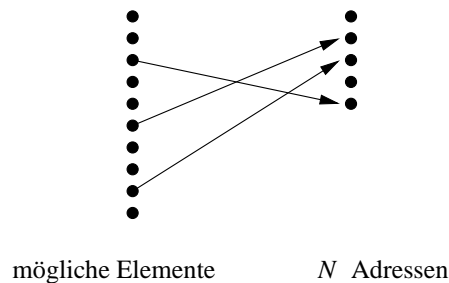
Hashing

Zum Abspeichern und Wiederfinden von Objekten wäre folgende Funktion hilfreich:

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

Dann könnte Objekt x bei Adresse $f(x)$ gespeichert werden.

Problem: Anzahl der möglichen Elemente \gg Anzahl der Adressen



Gegeben Adressen von 0 bis $N - 1$.

Sei x ein beliebiges Objekt. Dann ist

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

seine Stringrepräsentation.

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

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

ein Beispiel für eine (sehr einfache) *Hashfunktion*.

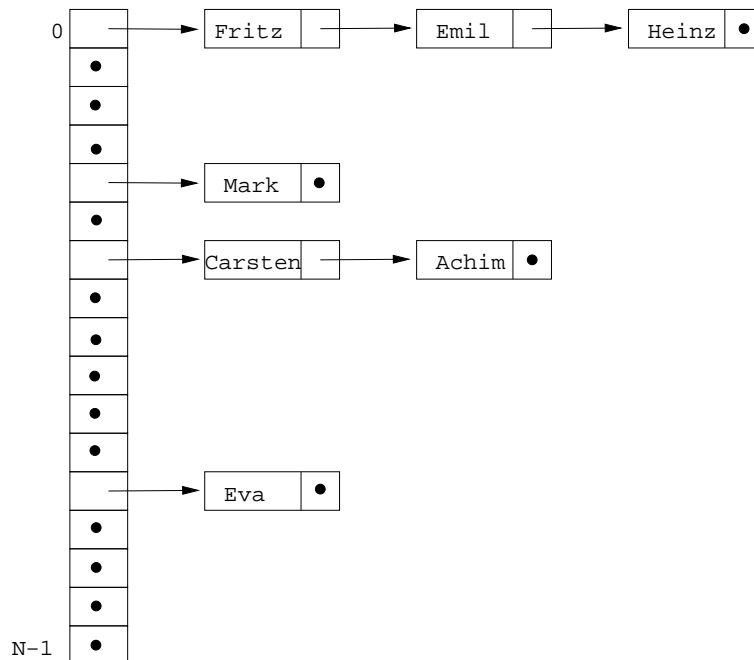
Gilt: $f(x) = f(y)$, so liegt eine *Kollision* vor, die bei *offenem* und *geschlossenem* Hashing unterschiedlich behandelt wird.

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

$y + f_2(x), y + 2 \cdot f_2(x), \dots$ Double Hashing (Schrittweite wird durch 2. Hashfunktion bestimmt)

Alle Berechnungen werden jeweils $\text{mod}(N)$ durchgeführt. Insgesamt werden höchstens $N - 1$ Sondierschritte verursacht. Beim quadratischen Sondieren werden ggf. nicht alle Buckets besucht, aber mindestens \sqrt{N} und höchstens $N/2$.

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	
N-1	B	Achim
	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 $N = 17$ (Beim geschlossenen Hashing wurde quadratisches Sondieren angewendet).

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
    }
    ...
    public boolean empty() { // testet, ob Tabelle leer
        ...
    }
    public Comparable lookup(Comparable x) { // versucht x nachzuschlagen
        ...
    }
    public boolean insert(Comparable x) { // versucht x einzufuegen
        ...
    }
    public boolean delete(Comparable x) { // versucht x zu loeschen
        ...
    }
}

/***** GeHashing.java *****/
/** Implementation des Interface Menge 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
    }
    ...
    public boolean empty() { // testet, Ob Tabelle leer
        ...
    }
    public Comparable lookup(Comparable x) { // versucht, x nachzuschlagen
        ...
    }
    public boolean insert(Comparable x) { // versucht, x einzufuegen
        ...
    }
    public boolean delete(Comparable x) { // versucht, x zu loeschen
        ...
    }
}

```

Laufzeit bei geschlossenem Hashing

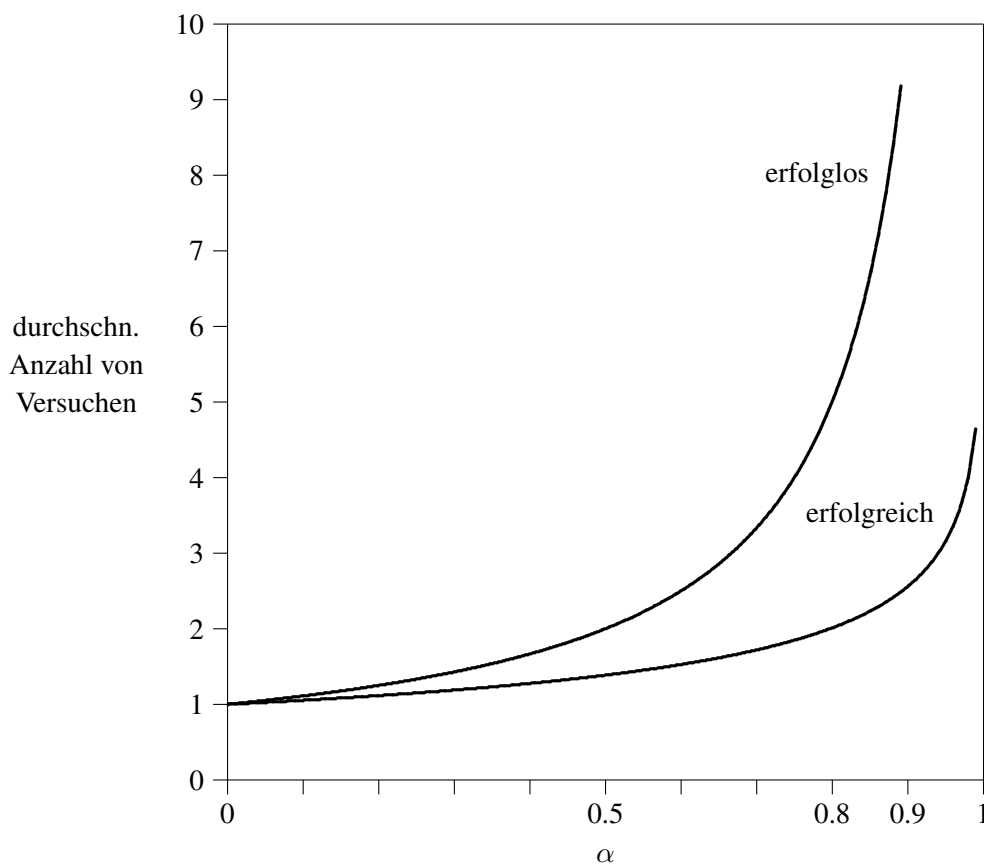
Sei n die Anzahl der in der Hashtabelle zur Zeit gespeicherten Objekte, sei N die Anzahl der möglichen Speicherpositionen.

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.)



Vergleich mit AVL-Baum

	AVL-Baum	Geschlossenes Hashing
Laufzeit	logarithmisch	konstant
Speicherbedarf	dynamisch wachsend	in Sprüngen wachsend
Sortierung	möglich durch Traversierung	nicht möglich

Kapitel 11

Java Collection Framework

Viele der in den vorigen Kapiteln vorgestellten Abstrakten Datentypen werden von der Sprache Java in ähnlicher Form angeboten. Es handelt sich dabei um Interfaces und dazu passende Implementationen, die im *Java Collection Framework* zusammengefasst sind.

11.1 Collection

Das Interface `Collection` verwaltet eine Sammlung von (nicht notwendigerweise verschiedenen) Objekten. Zum Testen auf inhaltliche Gleichheit muss die Object-Methode `equals` vom Anwender geeignet überschrieben werden. Folgende Methoden existieren (unter anderem):

```
boolean add(Object o);           // fuegt Objekt o hinzu
boolean contains(Object o);      // testet, ob es Objekt x gibt
                                   // mit x.equals(o)
boolean remove(Object o);       // entfernt ein Objekt x
                                   // mit x.equals(o)
boolean isEmpty();              // testet, ob Collection leer
Iterator iterator();            // liefert einen Iterator
```

Ein Iterator ist ein Objekt, welches das Durchlaufen aller Objekte einer Collection erlaubt mithilfe seiner Methoden

```
boolean hasNext();              // liefert true, falls noch Objekte da
Object next();                  // liefert naechstes Objekt
void remove();                  // entfernt zuletzt geliefertes Objekt
```

Vom Interface `Collection` wird das Interface `Set` abgeleitet, welches eine Menge ohne Duplikate verwaltet. `Set` verfügt nur über die geerbten, aber über keine zusätzlichen Methoden.

Vom Interface `Collection` wird außerdem abgeleitet das Interface `List`, welches eine sequentiell angeordnete Sammlung von Objekten verwaltet. Zu den geerbten Methoden kommen (unter anderem) hinzu

```
boolean add(int i, Object o); // fuegt Objekt o bei Position i ein
Object get(int i );          // liefert Objekt an Position i
int indexOf(Object o);       // liefert Position von Objekt o
```

Vom Interface `Set` wird das Interface `SortedSet` abgeleitet, welches auf der zu verwaltenden Menge eine Ordnung verlangt, d.h. die Objekte sollten entweder das Interface `Comparable` implementieren oder über die `compare`-Methode eines `Comparator`-Objektes verglichen werden können.

Für das Interface `Set` gibt es die Implementation `HashSet()`, welche die bereits bekannten Methoden `add`, `contains` und `remove` über eine Hashorganisation löst. Der Konstruktor erlaubt die (optionale) Angabe einer initialen Kapazität. Hashing in Java basiert auf der Methode `hashCode()`, mit der beliebige Objekte eine ganze Zahl zugeordnet bekommen mit der Eigenschaft, dass diese innerhalb eines Programmlaufs bei mehrmaligem Aufrufen für dasselbe Objekt identisch sein muss und dass die `hashCodes` für zwei Objekte, die bzgl der `equals`-Methode gleich sind, übereinstimmen müssen.

Die Klasse `TreeSet` implementiert das Interface `SortedSet`. Hier werden die Objekte in einer Baumstruktur verwaltet.

Die Klasse `LinkedList` implementiert das Interface `List`. Hier werden die Objekte in einer doppelt verzeigerten Liste verwaltet.

Die Klasse `Stack` (entspricht dem Keller) verwaltet eine Menge von Objekten und erweitert das Interface `Collection` unter anderem mit den Methoden

```
Object push(Object x); // legt ein Objekt oben auf den Stack
Object pop();          // entfernt das oberste Objekt vom Stack
Object peek();         // liefert das oberste Objekt vom Stack
boolean empty();       // testet, ob der Stack leer ist
```

Die Klasse `PriorityQueue` (entspricht dem Heap) verwaltet eine Menge von vergleichbaren Objekten und erweitert mit ihrer Implementation das Interface `Collection` unter anderem mit den Methoden

```
Object peek();         // liefert das billigste Objekt
Object poll();         // liefert und entfernt das billigste Objekt
```

Generische Klassen erlauben die Einschränkung des Wertebereichs: Beispielsweise wird durch

```
Set <Character> s = new HashSet <Character> ();
for (Character c : s) { // bearbeite c }
```

ein `HashSet` instantiiert, welcher ausschließlich Objekte vom Typ `Character` aufnimmt. Mit einer `for-each`-Schleife können alle Objekte des Sets durchlaufen werden.

11.2 Map

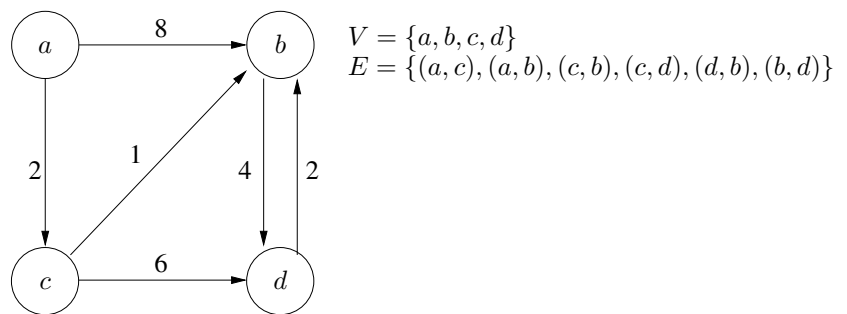
Das Interface `Map` repräsentiert eine Sammlung von sogenannten *Mappings* oder *Assoziationen* zwischen Schlüsseln und Werten. D.h. abgespeichert werden Objekt-Paare, von denen eine Komponente als Schlüssel fungiert und die andere Komponente den zu diesem Schlüssel gehörenden Wert darstellt. Implementiert wird das Interface durch die Klasse `HashMap`. Der Konstruktor erlaubt die (optionale) Angabe einer initialen Kapazität:

```
Map <Integer, String> h =           // besorge eine Map h
    new HashMap<Integer, String>(100); // mit 100 Eintraegen
h.put(42, "Susi");                 // neuer Eintrag mit key 42
if (h.containsKey(42)) ...         // teste, ob key 42 enthalten
if (h.containsValue("Susi")) ...   // teste, ob Wert enthalten
String s = h.get(42);              // besorge Wert zu key 42
h.remove(42);                      // entferne Eintrag zu key 42
Collection <String> c = h.values(); // besorge alle Werte
Iterator <String> iter = c.iterator(); // besorge Iterator
while (iter.hasNext()) {           // solange noch Objekte da
    s= iter.next();                // besorge naechstes Objekt
    IO.println(s);                 // verarbeite Objekt
}
```

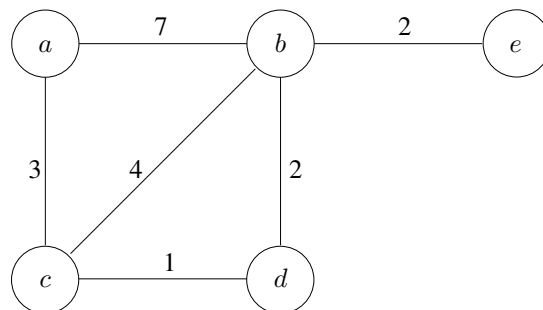

Kapitel 12

Graphen

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

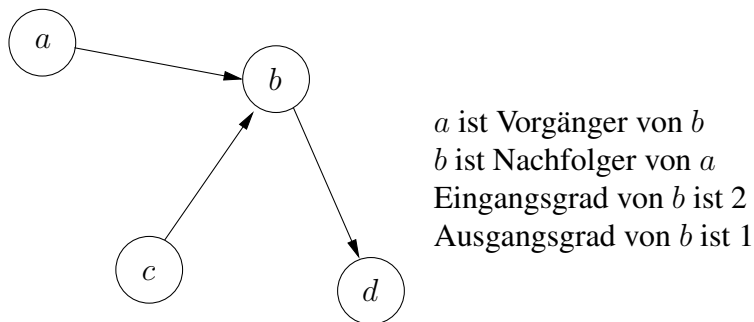
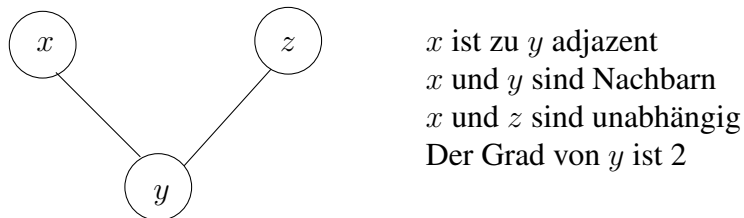
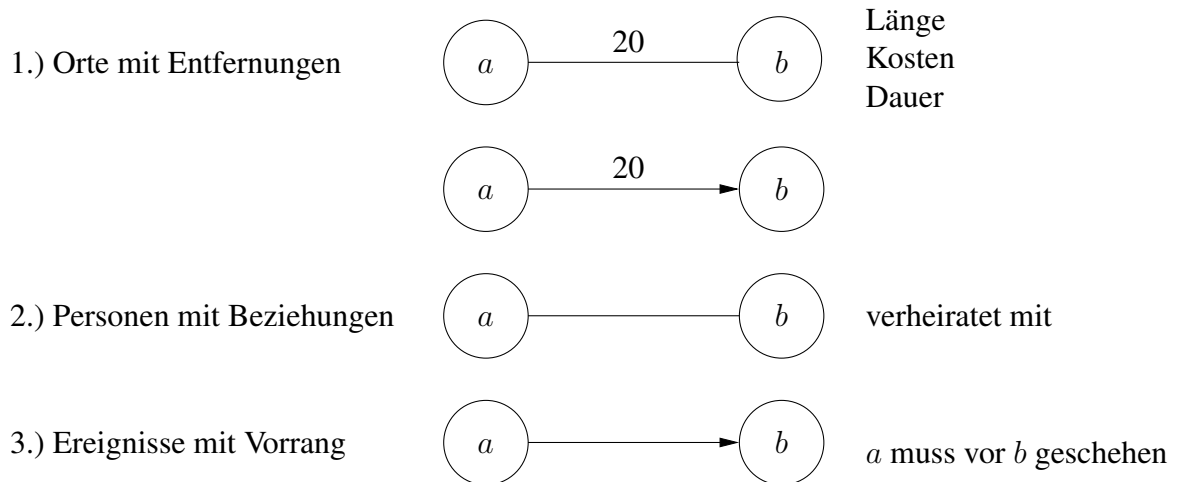


Ein *ungerichteter Graph* $G = (V, E)$ besteht aus *Knotenmenge* V und *Kantenmenge* $E \subseteq P_2(V)$ ($P_2(V)$ sind alle 2-elementigen Teilmengen von V .)



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

Mit Graphen können binäre Beziehungen zwischen Objekten modelliert werden. Die Objekte werden durch die Knoten, die Beziehungen durch die Kanten modelliert.



Wir nennen eine Kante (x,y) inzident zum Knoten x und zum Knoten y . Ein *Weg* ist eine Folge von adjazenten Knoten.

Ein *Kreis* ist ein *Weg*, der zurück zum Startknoten führt.

12.1 Implementation von Graphen

Es sei jedem Knoten eindeutig ein Index zugeordnet. Für den gerichteten Graphen auf Seite 145 ergibt sich:

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

Implementation durch Adjazenzmatrizen

Unbewertete Kanten:

	0	1	2	3
0	0	1	1	0
1	0	0	0	1
2	0	1	0	1
3	0	1	0	0

$$m[i][j] := \begin{cases} 1, & \text{falls } (i, j) \in E \\ 0 & \text{sonst} \end{cases}$$

Bewertete Kanten:

	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.

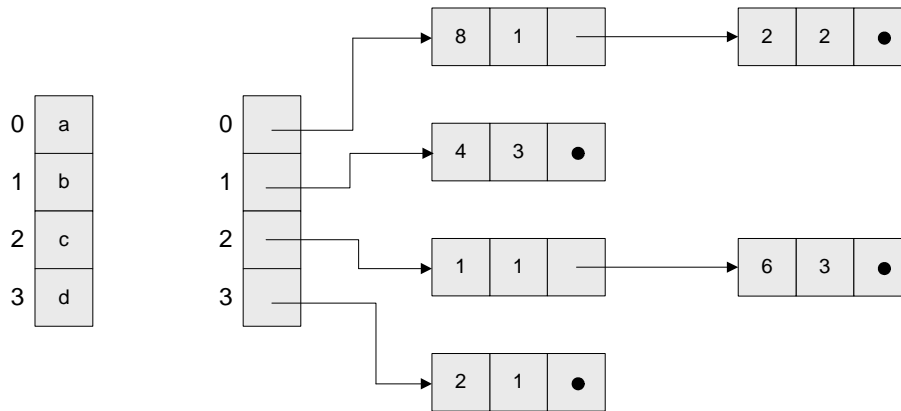
Sinnvoll bei dicht besetzten Graphen, d.h. Graphen mit quadratisch vielen Kanten.

Sinnvoll bei Algorithmen, die wahlfreien Zugriff auf eine Kante benötigen.

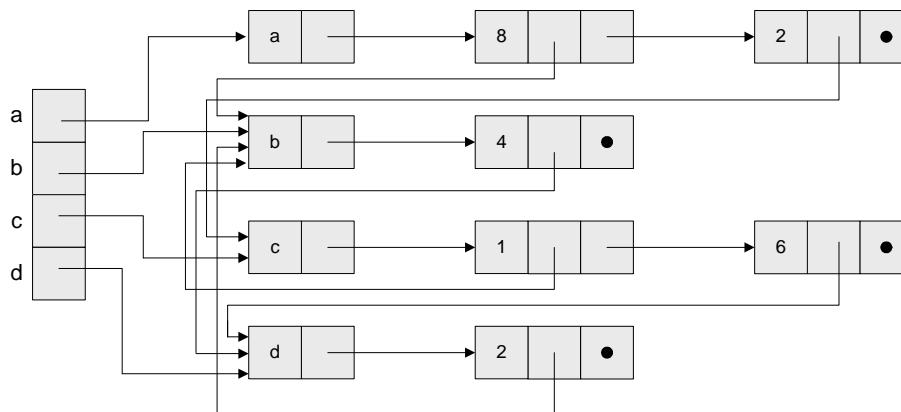
Implementation durch Adjazenzlisten

Wenn die Knoten des Graphen über ihren Index identifiziert werden, bietet sich zusätzlich zum Namens-Array ein Array von Integer-Listen an. Die an Position i beginnende Liste enthält die Nachbarn von i , genauer gesagt, sie enthält die Indizes der Nachbarn von i .

Für bewertete Graphen muss ein Listeneintrag innerhalb der i -ten Liste neben der Knoten-Nummer j auch die Kosten der Kante (i, j) enthalten. Für den gerichteten Graphen auf Seite 145 ergibt sich:



Eine objektorientierte Codierung würde die Knoten als Objekte modellieren, welche ihren Namen als String speichern und ihre Nachbarn als Kantenliste. Die Einträge der Kantenliste bestehen jeweils aus den Kosten der Kante und einem Verweis auf den Knoten. Über ein assoziatives Array (implementiert als Hash-Map) gelangt man vom Namen eines Knotens zu seinem Objekt. Für den gerichteten Graphen auf Seite 145 ergibt sich:



Platzbedarf $= O(|E|)$

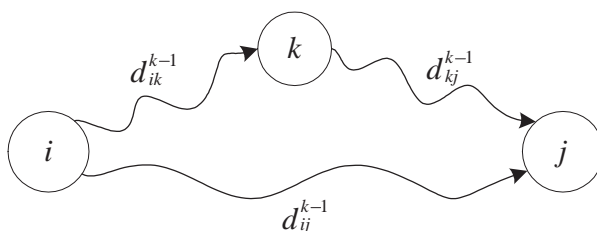
Kein effizienter Zugriff auf Kante (x, y) möglich.

Sinnvoll bei dünn besetzten Graphen, d.h. Graphen mit nur linear vielen Kanten.

Sinnvoll bei Algorithmen, die, gegeben ein Knoten x , dessen Nachbarn verarbeiten müssen.

12.2 Graphalgorithmen für Adjazenzmatrizen

Die Klasse `Floyd` löst das *all-pairs-shortest-path*-Problem mit dem Algorithmus von Floyd. Für eine $(n \times n)$ -Ausgangsmatrix C mit den Kantenkosten werden sukzessive die $(n \times n)$ -Matrizen $D^0, D^1, D^2, \dots, D^{n-1}$ berechnet. Die Matrix D^k enthält die Kosten der kürzesten Wege zwischen zwei Knoten i und j , die als Zwischenknoten nur die Knoten $0, 1, 2, \dots, k$ verwenden. Setze $D^{-1} := C$. Dann lässt sich $D_{i,j}^k$ errechnen durch das Minimum von $D_{i,j}^{k-1}$ und $D_{i,k}^{k-1} + D_{k,j}^{k-1}$.



Um auch die zugehörige Kantenfolge rekonstruieren zu können, wird parallel dazu eine Folge von $(n \times n)$ -Matrizen $P^0, P^1, P^2, \dots, P^{n-1}$ aufgebaut, die an Position $P_{i,j}^k$ den vorletzten Knoten auf dem kürzesten Weg von i nach j notiert, der nur über die Zwischenknoten $0, 1, 2, \dots, k$ läuft.

```

/***** Floyd.java *****/

/** berechnet alle kuerzesten Wege und ihre Kosten mit Algorithmus von Floyd */
/* der Graph darf keine Kreise mit negativen Kosten haben */

public class Floyd {

    public static void floyd (int n,          // Dimension der Matrix
                             double [][] c,  // Adjazenzmatrix mit Kosten
                             double [][] d,  // errechnete Distanzmatrix
                             int    [][] p){ // errechnete Wegematrix

        int i, j, k;                          // Laufvariablen
        for (i=0; i < n; i++) {                // fuer jede Zeile
            for (j=0; j < n; j++) {            // fuer jede Spalte
                d[i][j] = c[i][j];            // initialisiere mit Kantenkosten
                p[i][j] = i;                   // vorletzter Knoten
            }                                   // vorhanden ist nun D hoch -1
        }

        for (k=0; k < n; k++) {                // fuer jede Knotenobergrenze
            for (i=0; i < n; i++) {            // fuer jede Zeile
                for (j=0; j < n; j++) {        // fuer jede Spalte
                    if (d[i][k] + d[k][j] < d[i][j]){ // falls Verkuerzung moeglich
                        d[i][j] = d[i][k] + d[k][j]; // notiere Verkuerzung
                        p[i][j] = p[k][j];         // notiere vorletzten Knoten
                    }                               // vorhanden ist nun D hoch k
                }
            }
        }
    }
}

```


12.3 Implementation für gerichtete Graphen durch Adjazenzlisten

Jeder Knoten der Klasse `Vertex` enthält eine Liste von Kanten; jede Kante der Klasse `Edge` besteht aus Kosten und Zielknoten. Die Klasse `Graph` realisiert den Graph als Assoziation von Knotennamen und Knoten. Die Klasse `GraphIO` liest einen Graph aus einer Datei ein und zeigt seine Adjazenzlisten an. Die Klasse `Result` enthält Routinen zum Anzeigen einer Lösung, welche von dem jeweiligen Algorithmus in den Arbeitsvariablen der Knoten hinterlegt wurde. Die Klasse `GraphTest.java` liest einen gerichteten Graphen ein und wendet verschiedene Graphalgorithmen darauf an.

```

/***** Vertex.java *****/
/** Klasse zur Repraesentation eines Knoten */
import java.util.*;

public class Vertex implements Comparable<Vertex> { // wegen Priority-Queue

    public String      name;           // Name des Knoten           (fix)
    public List<Edge>  edges ;         // Nachbarn als Kantenliste (fix)
    public int         nr;             // Knotennummer           (errechnet)
    public int         indegree;       // Eingangsgrad           (errechnet)
    public double      dist;           // Kosten fuer diesen Knoten (errechnet)
    public boolean     seen;           // Besuchs-Status         (errechnet)
    public Vertex      prev;           // Vorgaenger fuer diesen Knoten (errechnet)

    public Vertex ( String s ) { // Konstruktor fuer Knoten
        name = s; // initialisiere Name des Knoten
        edges = new LinkedList<Edge>(); // initialisiere Nachbarschaftsliste
    }

    public boolean hasEdge(Vertex w) { // testet, ob Kante zu w besteht
        for (Edge e : edges) // fuer jede ausgehende Nachbarkante pruefe
            if (e.dest == w) // falls Zielknoten mit w uebereinstimmt
                return true; // melde Erfolg
        return false; // ansonsten: melde Misserfolg
    }

    public int compareTo(Vertex other) { // vergl. Kosten mit anderem Vertex
        if (other.dist > dist) return -1;
        if (other.dist < dist) return 1;
        return 0;
    }
}

/***** Edge.java *****/
/** Klasse zur Repraesentation einer Kante */
import java.util.*;

public class Edge {

    public Vertex dest; // Zielknoten, zu dem die Kante fuehrt
    public double cost; // Kosten dieser Kante

    public Edge (Vertex d, double c) { // Konstruktor fuer Kante
        dest = d; // initialisiere Zielknoten
        cost = c; // initialisiere Kantenkosten
    }
}

```

}
}


```

/***** Graph.java *****/
import java.util.*;

/** Klasse zur Implementation eines Graphen basierend auf Vertex und Edge */
/* Der Graph wird implementiert als HashMap <String, Vertex>, d.h. als eine */
/* Hashtabelle mit Keys vom Typ String und Values vom Typ Knoten */

public class Graph {

    private Map <String, Vertex> graph;          // Datenstruktur fuer Graph

    public Graph() {                            // leerer Graph wird angelegt
        graph = new HashMap <String, Vertex> (); // als HashMap von String,Vertex
    }

    public boolean isEmpty(){                   // liefert true, falls Graph leer
        return graph.isEmpty();                // mit isEmpty() von HashMap
    }

    public int size(){                          // liefert die Anzahl der Knoten
        return graph.size();                   // mit size() von HashMap
    }

    public Collection <Vertex> vertices(){      // liefert Knoten als Collection
        return graph.values();                 // mit values() von HashMap
    }

    public Vertex getVertex(String s){         // liefere Knoten zu String
        Vertex v = graph.get(s);               // besorge Knoten zu Knotennamen
        if (v==null) {                         // falls nicht gefunden
            v = new Vertex(s);                 // lege neuen Knoten an
            graph.put(s, v);                   // fuege Namen und Knoten in HashMap ein
        }
        return v;                              // liefere gefundenen oder neuen Knoten
    }

    public void addEdge(String source,         // fuege Kante ein von Knotennamen source
                        String dest,         // zu Knotennamen dest
                        double cost) {       // mit Kosten cost
        Vertex v = getVertex(source);        // finde Knoten v zum Startnamen
        Vertex w = getVertex(dest);         // finde Knoten w zum Zielnamen
        v.edges.add(new Edge(w, cost));     // fuege Kante (v,w) mit Kosten cost ein
    }
}

```

```
/****** GraphIO.java *****/

import java.lang.*;
import java.util.*;
import java.io.*;
import AlgoTools.IO;

/** Routinen zum Einlesen eines gerichteten Graphen          */
/* Der Graph wird realisiert durch eine HashMap, welche     */
/* den Namen des Knoten auf den Knoten abbildet            */
public class GraphIO {

    public static Graph readGraph() { // liest Graph aus Datei ein
        Graph g = new Graph();
        try {
            BufferedReader f = new BufferedReader(new FileReader("graph.dat"));
            String zeile;
            while ( (zeile = f.readLine()) != null) {
                StringTokenizer st = new StringTokenizer(zeile);
                String source = st.nextToken();
                String dest   = st.nextToken();
                double cost   = Double.parseDouble(st.nextToken());
                g.addEdge(source,dest,cost);
            }
        } catch (Exception e) {IO.println(e);}
        return g;
    }

    public static void printGraph(Graph g) { // gibt Graph aus
        IO.println("Adjazenzlisten des Graphen:");
        for (Vertex v : g.vertices()) {
            for (Edge e : v.edges) {
                IO.print("(" + v.name + "," + e.dest.name + ")" + e.cost + " ");
            }
            IO.println();
        }
        IO.println();
    }
}
```

```

/***** Result.java *****/

import java.util.*;
import AlgoTools.IO;

/** Routinen zum Anzeigen der Loesungen, kodiert in den Arbeitsvariablen */

public class Result {

    private static void printPath(Vertex dest) {
        if (dest.prev != null) {
            printPath(dest.prev);
            IO.print(" -> ");
        } IO.print(dest.name);
    }

    public static void printPath(Graph g, Vertex w) {
        if (w.dist==Double.MAX_VALUE) IO.println(w.name + " nicht erreichbar");
        else {
            IO.println("Der kuerzeste Weg (mit Gesamtkosten "+w.dist+") lautet: ");
            printPath(w);
        }
        IO.println(); IO.println();
    }

    public static void printHamilton(Graph g, Vertex last) {
        if (last==null) IO.println("Graph hat keinen Hamiltonkreis"); else {
            IO.println("Der Hamiltonkreis lautet:");
            printPath(last);
        }
        IO.println(); IO.println();
    }

    public static void printTopo( Graph g ){
        boolean erfolgreich=true;
        for (Vertex v : g.vertices())
            if (v.nr<0) erfolgreich=false;
        if (erfolgreich) {
            IO.println("Die Sortierung lautet:");
            for (Vertex v : g.vertices())
                IO.println("Knoten " + v.name + " erhaelt Nr. " + v.nr);
        } else IO.println("Graph kann nicht topologisch sortiert werden");
        IO.println();
    }

    public static void printTraverse( Graph g ){
        IO.println("Traversierungsreihenfolge:");
        for (Vertex v : g.vertices()){
            IO.println(v.name + " erhaelt Nr. " + v.nr);
        }
        IO.println();
    }
}

```

```

/***** GraphTest.java *****/
import java.util.*;

/** testet die Graph-Algorithmen */

public class GraphTest {

    public static void main(String [] argv) {

        Graph g = GraphIO.readGraph(); // Graph einlesen
        GraphIO.printGraph(g); // Graph ausgeben

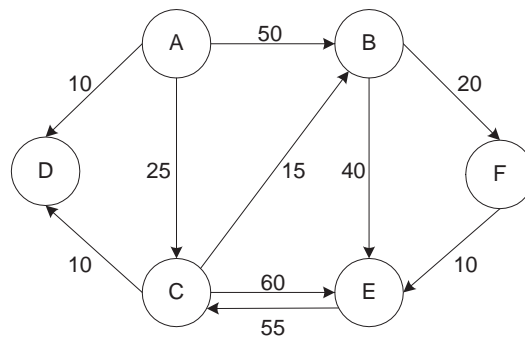
        GraphTraverse.tiefensuche(g); // Tiefensuche-Traversierung
        Result.printTraverse(g); // Traversierung ausgeben

        TopoSort.sortGraph(g); // topologisch sortieren
        Result.printTopo(g); // Sortierung ausgeben

        Dijkstra.dijkstra(g, g.getVertex("A")); // kuerzeste Wege von A berechnen
        Result.printPath(g, g.getVertex("E")); // kuerzesten Weg zu E ausgeben

        Vertex v = Hamilton.hamilton(g, // Hamilton-Kreis berechnen
                                     g.getVertex("A")); // dabei bei A beginnen
        Result.printHamilton(g, v); // Hamilton-Kreis ausgeben
    }
}

```



gerichteter, bewerteter Graph:
gezeigt sind Knotennamen und Kantenkosten

```
A B 50
A C 25
A D 10
B E 85
B F 20
C B 15
C D 10
C E 60
E C 25
F E 10
```

Eingabedatei graph.dat mit Graphbeschreibung

```
Adjazenzlisten des Graphen:

(E,C)25.0
(F,E)10.0
(A,B)50.0 (A,C)25.0 (A,D)10.0
(B,E)85.0 (B,F)20.0
(C,B)15.0 (C,D)10.0 (C,E)60.0

Traversierungsreihenfolge:
D erhaelt Nr. 0
E erhaelt Nr. 1
F erhaelt Nr. 4
A erhaelt Nr. 5
B erhaelt Nr. 3
C erhaelt Nr. 2

Graph kann nicht topologisch sortiert werden

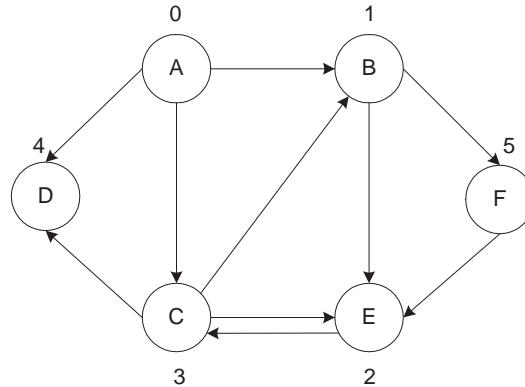
Der kuerzeste Weg (mit Gesamtkosten 70.0) lautet:
A -> C -> B -> F -> E

Graph hat keinen Hamiltonkreis
```

von GraphTest.java erzeugte Ausgabe result.dat

12.4 Traversieren von Graphen

Eine Graphentraverse auf einem gerichteten Graphen lässt sich durch eine rekursiv organisierte *Tiefensuche* implementieren, wobei die Nummern an Knoten vergeben werden in der Reihenfolge, in der sie besucht werden. Bei nicht zusammenhängenden Graphen muss dazu die Suche mehrfach gestartet werden.



Rekursive Tiefensuche: Nummerierung entstanden durch Startknoten A

```

/*****GraphTraverse.java*****/
import java.util.*;

public class GraphTraverse {

    static int id; // Variable zum Numerieren

    public static void tiefensuche(Graph g){
        id = 0; // Initialisiere Zaehler
        for (Vertex v : g.vertices()) // fuer jeden Knoten
            v.seen = false; // markiere als nicht besucht
        for (Vertex v : g.vertices()) // fuer jeden Knoten
            if (!v.seen) visit(v); // falls v nicht besucht: besuche v
    }

    private static void visit(Vertex v) {
        v.nr = id++; // vergib naechste Nummer
        v.seen = true; // markiere als besucht
        for (Edge e : v.edges){ // fuer jede ausgehende Kante
            Vertex w = e.dest; // sei w der Nachbarknoten
            if (!w.seen) visit(w); // falls w nicht besucht: besuche w
        }
    }
}

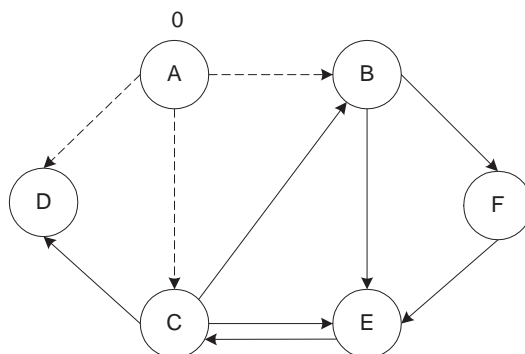
```

12.5 Topologisches Sortieren

Das Problem des Topologischen Sortierens auf einem gerichteten Graphen besteht darin, eine Knoten-Nummerierung $f : V \rightarrow \mathbb{N}$ zu finden, die mit den Kantenrichtungen kompatibel ist, d.h. $(x, y) \in E \Rightarrow f(x) < f(y)$.

Begonnen wird die Nummerierung bei einem Knoten mit Eingangsgrad 0 (der die erste Nummer erhalten kann).

Sei x der Knoten, der zuletzt nummeriert wurde. Dann werden nun die von x ausgehenden Kanten (virtuell) entfernt und somit die (virtuellen) Eingangsgrade der von ihm erreichbaren Knoten vermindert. Verwendet wird dabei eine Schlange, welche solche Knoten speichert, deren virtueller Eingangsgrad im Laufe der Berechnung inzwischen auf 0 gesunken ist und daher für die nächste zu vergebene Nummer infrage kommt. Auf diese Weise wird verhindert, dass immer wieder nach einem Knoten mit Eingangsgrad 0 gesucht werden muss.



Topologisches Sortieren: Knoten A nummeriert, Nachfolgekanten entfernt

```

/***** TopoSort.java *****/

import java.util.*;

/** Topologisches Sortieren eines gerichteten Graphen          */
/* Verwendet wird eine Schlange, welche die Knoten aufnimmt,  */
/* deren Eingangsgrade auf 0 gesunken sind                    */

public class TopoSort {

    public static void sortGraph ( Graph g) {

        for (Vertex v : g.vertices()){ // fuer jeden Knoten
            v.indegree = 0;           // setze seinen Eingangsgrad auf 0
            v.nr = -1;                // vergib ungueltige Nummer
        }

        for (Vertex v : g.vertices()) // fuer jeden Knoten
            for (Edge e : v.edges)    // fuer jeden Nachbarknoten
                e.dest.indegree++;    // erhoehe seinen Eingangsgrad um 1

        List<Vertex> l                // Liste von Knoten, die
            = new LinkedList<Vertex>(); // inzwischen den Eingangsgrad 0 haben

        for (Vertex v : g.vertices()) // jeden Knoten mit Eingangsgrad 0
            if (v.indegree==0) l.add(v); // fuege hinten in die Liste ein

        int id = 0;                  // initialisiere Laufnummer
        while (!l.isEmpty()) {        // solange Liste nicht leer
            Vertex v = l.remove(0);    // besorge und entferne Kopf aus Liste
            v.nr = id++;              // vergib naechste Nummer
            for (Edge e : v.edges){    // fuer jede ausgehende Kante
                Vertex w = e.dest;    // betrachte den zugehoerigen Knoten
                w.indegree--;         // erniedrige seinen Eingangsgrad
                if (w.indegree==0)    // falls der Eingangsgrad auf 0 sinkt
                    l.add(w);        // fuege Knoten hinten in Liste ein
            }
        }
    }
}

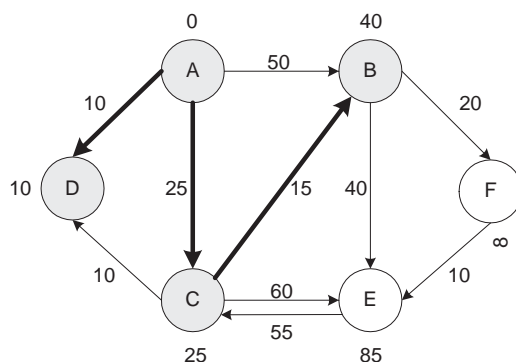
```


12.6 Kürzeste Wege

Das Problem der kürzesten Wege besteht darin, in einem gerichteten, mit nichtnegativen Kosten gewichteten Graphen die kürzesten Wege von einem Startknoten zu allen anderen Knoten auszurechnen (single source shortest paths).

Der Algorithmus von Dijkstra verwendet dabei eine Priority-Queue (\approx Heap aus Kapitel 7), welche die vorläufige Distanz eines Weges vom Startknoten zu einem Zielknoten in Form eines mit dieser Distanz bewerteten Knotens speichert.

Zu Beginn haben alle Knoten die vorläufige Distanz ∞ ; der Startknoten erhält die endgültige Distanz 0. Jeweils der billigste Knoten aus der Prioritäts-Schlange kann seine endgültige Distanz erhalten und fügt ggf. die durch ihn verkürzten vorläufigen Distanzen zu seinen Nachbarn in Form von Knoten in die Schlange ein. Weil die Priority-Queue die Kosten der in ihr gespeicherten Wege nicht verringern kann, werden Wege, deren Kosten sich verringert haben, als neue Knoten eingefügt und die Schlange enthält daher für manche Knoten Mehrfacheinträge mit unterschiedlich teuren Distanzen. Solche Einträge können nach dem Entfernen aus der Schlange ignoriert werden, wenn der Knoten inzwischen als besucht (seen) markiert wurde.



Single-Source-Shortest-Path:
Grau gefärbt sind bereits markierte Knoten

```

/***** Dijkstra.java *****/

import java.util.*;

/** implementiert den single source shortest path Algorithmus nach Dijkstra */
/** */
/** Es sind nur nichtnegative Kantenkosten zugelassen */
/** */
/** Verwendet wird eine Priority-Queue der Knoten, gewichtet mit den Kosten */
/** des vorlaeufig kuerzesten Weges vom Startknoten bis zu diesem Knoten */

public class Dijkstra {

    public static void dijkstra ( Graph g, Vertex start) {

        PriorityQueue<Vertex> p = // Priority-Queue zum Verwalten der Laenge
            new PriorityQueue<Vertex>(); // des kuerzesten Weges bis zum Knoten

        for (Vertex v : g.vertices()){ // fuer jeden Knoten
            v.dist = Double.MAX_VALUE; // Entfernung ist unendlich
            v.seen = false; // Knoten noch nicht gesehen
            v.prev = null; // Vorgaenger noch nicht ermittelt
        }

        start.dist = 0; // endgueltige Kosten zum Startknoten
        p.add(start); // erster Eintrag in PriorityQueue

        while (!p.isEmpty()){ // solange noch Eintraege in Priority-Queue

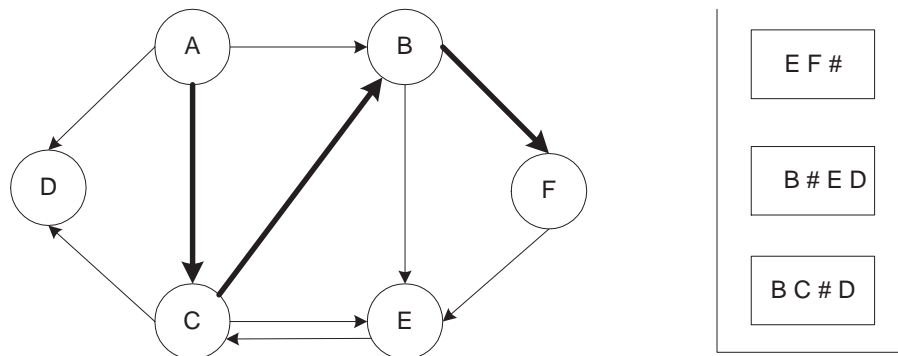
            Vertex v = p.poll(); // billigster Eintrag in PriorityQueue
            if (v.seen) continue; // falls schon bearbeitet: ignorieren
            v.seen = true; // als bearbeitet markieren

            for (Edge e : v.edges){ // fuer jede Nachbarkante e von v tue
                Vertex w = e.dest; // besorge Zielknoten w
                double c = e.cost; // besorge Kosten c zum Zielknoten w
                if (c<0) throw new // falls Kantenkosten negativ
                    RuntimeException("Negativ"); // melde Fehler
                if (w.dist > v.dist + c) { // falls Verkuerzung moeglich
                    w.dist = v.dist + c; // berechne Verkuerzung
                    w.prev = v; // notiere verursachenden Vorgaenger
                    p.add(w); // neuer Eintrag in PriorityQueue
                }
            }
        }
    }
}

```

12.7 Hamiltonkreis

Das Problem des Hamiltonkreises besteht darin, in einem Graphen einen Rundweg zu finden, der jeden Knoten genau einmal besucht und beim Ausgangsknoten wieder endet. Verwendet wird dabei ein Keller von Adjazenzlisten zur Organisation eines Backtracking-Ansatzes. Um die Exploration bei einer oben auf dem Keller liegenden teilweise abgearbeiteten Adjazenzliste an der richtigen Stelle fortführen zu können, bestehen die Kellereinträge aus den Iteratoren der Adjazenzlisten. Diese können jeweils den nächsten, noch nicht ausprobierten Nachbarn liefern.



Suche nach Hamiltonkreis: markiert ist der aktuelle Weg.

Im Keller liegen Iteratoren für Adjazenzlisten, ihr Fortschritt ist markiert mit #

```

/***** Hamilton.java *****/

import java.util.*;

/** sucht einen Hamiltonkreis in einem gerichteten Graphen */
/* verwendet wird ein Keller mit den Adjazenzlisten in Form von Iteratoren */

public class Hamilton {

    public static Vertex hamilton(Graph g, Vertex start) {
        for (Vertex v : g.vertices()) { // fuer jeden Knoten
            v.seen = false; // noch nicht gesehen
            v.prev = null; // noch kein Vorgaenger bekannt
        }
        Iterator iter; // Iteratorvariable
        Stack<Iterator> s = new Stack<Iterator>(); // Stack fuer Adjazenz-Iteratoren
        boolean entschieden = false; // bisher ist noch nichts entschieden
        s.push(start.edges.iterator()); // Nachbarn des Startknoten auf Stack
        start.seen = true; // Startknoten gilt als besucht
        Vertex last = start; // last ist der zur Zeit letzte Knoten
        int k = 1; // Weglaenge = 1

        Vertex v=null; // Hilfsknoten
        while (!entschieden) { // solange noch nicht entschieden
            iter = s.peek(); // oberster Adjazenz-Iterator
            boolean verlaengerbar=false; // bisher keine weiteren gefunden
            while (!verlaengerbar && iter.hasNext()) { // solange Hoffnung besteht
                Edge e = (Edge)iter.next(); // e ist naechste Kante
                v = e.dest; // v ist potentielle Verlaengerung
                if (!v.seen) verlaengerbar=true; // Verlaengerung gefunden
            }
            if (verlaengerbar) { // falls Verlaengerung moeglich
                k++; // erhoehe Weglaenge
                v.seen = true; // markiere v als besucht
                v.prev = last; // trage last als Vorgaenger ein
                last = v; // v ist jetzt der letzte
                if (k==g.size()&&v.hasEdge(start)) // Weg lang genug und Kante zu Start
                    entschieden = true; // endgueltig Kreis gefunden
                else s.push(v.edges.iterator()); // pushe Nachfolger von v auf Stack
            } else { // keine Verlaengerung moeglich
                s.pop(); // entferne oberste Adjazenzliste
                last.seen = false; // last gilt als nicht besucht
                last = last.prev; // Vorgaenger ist nun der letzte
                k--; // Weglaenge erniedrigen
                if (s.empty()) entschieden=true; // endgueltig keinen Kreis gefunden
            }
        }
        if (entschieden && s.empty())
            return null; // kein Kreis gefunden
        else return v; // liefere letzten Knoten des Kreises
    }
}

```

12.8 Maximaler Fluss

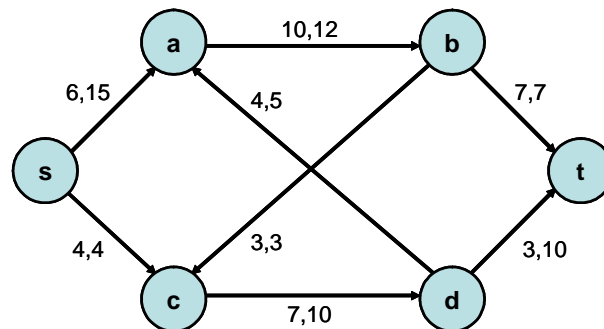
Gegeben sei ein gerichteter Graph $G = (V, E)$, gewichtet mit einer Funktion $c : E \rightarrow \mathbb{N}$. Die Gewichte $c(i, j)$ werden hier als Kantenkapazitäten interpretiert und beschreiben die maximale Zahl von Transporteinheiten, die pro Zeiteinheit durch diese Kante fließen können. Zwei Knoten aus V sind als Quelle s und die Senke t ausgezeichnet. Modelliert werden mit einem solchen Graph die Transportkapazitäten zwischen diversen Orten und gesucht wird der maximale Fluss, der von der Quelle ausgeht, durch die inneren Knoten fließt und an der Senke ankommt. Dabei muss die Flussmenge, die in einen inneren Knoten hineinfließt, übereinstimmen mit der Flussmenge, die aus ihm herausfließt.

Formal: Sei $V(i)$ die Menge aller Vorgängerknoten von Knoten i . Sei $N(i)$ die Menge aller Nachfolgerknoten von Knoten i . Ein **Fluss** ist eine Funktion $f : E \rightarrow \mathbb{N}$ mit:

- $0 \leq f(i, j) \leq c(i, j)$
- für alle $i \in V \setminus \{s, t\}$ gilt: $\sum_{a \in V(i)} f(a, i) = \sum_{b \in N(i)} f(i, b)$,

Gesucht ist nun der maximale Gesamtfluss $F = \sum_{b \in N(s)} f(s, b) = \sum_{a \in V(t)} f(a, t)$.

Im folgenden Graph ist jede Kante beschriftet mit ihrem aktuellen Fluss und ihrer maximalen Kapazität. Es fließen insgesamt 10 Einheiten von der Quelle s zur Senke t .



Um den maximalen Fluss zu berechnen, sucht man zunächst sogenannte erweiternde Wege. Ein **erweiternder** Weg in G ist eine Folge von Knoten, beginnend bei s und endend bei t , wobei für zwei aufeinanderfolgende Knoten i und j gilt:

- es existiert eine Vorwärtskante $(i, j) \in E$ mit $f(i, j) < c(i, j)$ oder
- es existiert eine Rückwärtskante $(j, i) \in E$ mit $f(j, i) > 0$.

Längs eines erweiternden Weges kann der Fluss vergrößert werden, indem man durch die Vorwärtskanten zusätzliche Einheiten fließen lässt und in den Rückwärtskanten den Fluss verringert.

Der Algorithmus von Ford-Fulkerson versucht solange den aktuellen Fluss längs eines erweiternden Weges zu erhöhen, wie dies möglich ist:

```

Sei  $f = 0$  der initiale Fluss;
repeat {
    Suche bzgl.  $f$  einen erweiternden Weg von  $s$  nach  $t$ ;
    falls gefunden: erhöhe  $f$  längs dieses Weges;
} while möglich

```

Die Korrektheit dieses Algorithmus folgt unmittelbar aus dem folgenden Satz:

Ein Fluss f ist maximal \iff Es gibt keinen erweiternden Weg.

\Leftarrow

Sei ein **Cut** des Graphen $G = (V, E)$ definiert als eine Partition der Knotenmenge V in S und \bar{S} , so dass gilt: $s \in S$ und $t \in V \setminus S = \bar{S}$. Sei dann $(S, \bar{S}) := \{(x, y) \in E \mid x \in S, y \in \bar{S}\}$. Die Kapazität eines Cuts ist dann festgelegt durch $c(S) := \sum_{e \in (S, \bar{S})} c(e)$.

Daraus folgt unmittelbar der folgende Satz:

Sei f ein Fluss der Größe F , sei S ein Cut von G

$$\Rightarrow F = \sum_{e \in (S, \bar{S})} f(e) - \sum_{e \in (\bar{S}, S)} f(e).$$

Der Fluss ist also an jedem beliebigen Cut messbar. Aus diesem Satz folgt aber direkt:

Sei f ein Fluss der Größe F , sei S ein Cut von G

$$\Rightarrow F \leq c(S),$$

d.h. der Fluss ist höchstens so hoch wie die Kapazität eines Cuts.

\Rightarrow

Ist ein Fluss f so groß wie die Kapazität eines Cuts S , so ist f ein maximaler Fluss und S ist ein minimaler Cut bezogen auf seine Kapazität $c(S)$.

Wir betrachten nun folgende Situation:

Es gibt in G keine erweiternden Wege mehr. Dann enden also alle von s ausgehenden erweiternden Wege - genauer gesagt deren Anfangsstücke - entweder bei einer saturierten Vorwärtskante oder bei einer Rückwärtskante mit Fluss 0. Durch diese Kanten wird also ein Cut S impliziert, dessen Kapazität gleich dem momentanen Fluss ist. Daher muss der momentane Fluss aufgrund des letzten Satzes maximal sein.

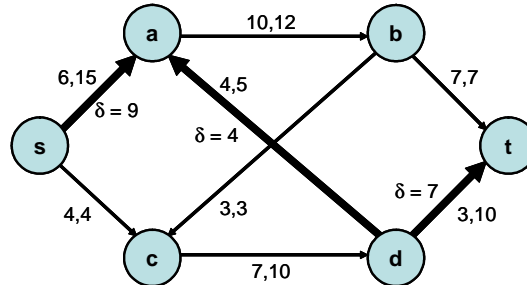
Die Suche nach einem erweiternden Weg wird durch eine Breitensuche organisiert, die beim Knoten s beginnt:

```

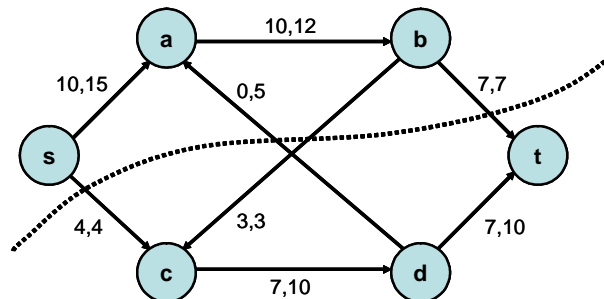
Markiere s;
Sei  $M := \{s\}$  die Menge der noch zu bearbeitenden Knoten;
while M nicht leer und t ist noch nicht markiert {
  Besorge und entferne aus M einen markierten Knoten x
  IF  $\exists$  unmarkiertes y mit  $(x,y) \in E$  und  $f(x,y) < c(x,y)$  THEN
    markiere y und vermerke bei y seinen Vorgänger x;
    notiere  $\delta(x,y) := c(x,y) - f(x,y)$ ;
    nimm y in M auf;
  IF  $\exists$  unmarkiertes y mit  $(y,x) \in E$  und  $f(y,x) > 0$  THEN
    markiere y und vermerke bei y seinen Vorgänger x;
    notiere  $\delta(x,y) := f(y,x)$ ;
    nimm y in M auf;
}
fallst t markiert THEN
  berechne ausgehend von t längs der Vorgänger das maximal mögliche  $\delta$ ;
  erhöhe ausgehend von t längs der Vorgänger den Fluss f um  $\delta$ ;

```

In folgendem Graph ist ein erweiternder Weg fett markiert. Er führt von s über a und d nach t und erlaubt eine Flusserhöhung um $\delta = 4$.



Nach Anwendung des erweiternden Weges entsteht der folgende Graph, mit einem Fluss (und einem Cut) von 14 Einheiten. Da kein weiterer erweiternder Weg existiert, ist der Fluss maximal.



Zur effizienten Implementation der Breitensuche ist es erforderlich, bei jedem Knoten nicht nur die Liste seiner Nachfolger, sondern auch die Liste seiner Vorgänger zu speichern.

12.9 Implementation für ungerichtete Graphen durch Adjazenzlisten

Bei ungerichteten Graphen existiert zwischen zwei Knoten genau eine gewichtete Kante. Daher wird in der Klasse `UndiVertex` von jedem Knoten auf alle adjazenten Kanten verwiesen und in der Klasse `UndiEdge` von jeder Kante auf ihre beiden Endpunkte.

```

/***** UndiVertex.java *****/
import java.util.*;
public class UndiVertex {

    public String          name;          // Name des Knoten          (fix)
    public List<UndiEdge> edges;          // inzidente Kanten          (fix)
    public UndiVertex     chef;          // Repraesentant fuer d. Knoten (errechnet)

    public UndiVertex ( String s ) {      // Konstruktor fuer Knoten
        name = s;                          // initialisiere Name des Knoten
        edges = new LinkedList<UndiEdge>(); // initialisiere adjazente Kanten
    }

    public Collection<UndiVertex>neighbors(){ // liefert die Nachbarn des Knotens
        Collection <UndiVertex> s = new HashSet<UndiVertex>();
        for (UndiEdge e : edges){
            if (e.left != this) s.add(e.left);
            if (e.right != this) s.add(e.right);
        } return s;
    }

    public boolean hasEdge(UndiVertex w) { // testet, ob Kante zu w besteht
        for (UndiVertex v : neighbors()) // fuer jede Kante pruefe
            if (v == w) // falls n mit w uebereinstimmt
                return true; // melde Erfolg
        return false; // ansonsten: melde Misserfolg
    }
}

/***** UndiEdge.java *****/
import java.util.*;
public class UndiEdge implements Comparable<UndiEdge>{

    public UndiVertex left;          // der eine benachbarte Knoten
    public UndiVertex right ;       // der andere benachbarte Knoten
    public double cost;             // Kosten dieser Kante
    public boolean status;          // Zustand der Kante (errechnet)

    public UndiEdge (UndiVertex l, UndiVertex r, double c) { // Konstruktor
        left = l; // initialisiere den einen Nachbarn
        right = r; // initialisiere den anderen Nachbarn
        cost = c; // initialisiere Kantenkosten
        status = false; // Kante ist noch unbearbeitet
    }

    public int compareTo (UndiEdge other) { // vergleiche mit anderer Kante
        return (int)(cost - other.cost); // liefere Ergebnis des Vergleichs
    }
}

```


12.9. IMPLEMENTATION FÜR UNGERICHTETE GRAPHEN DURCH ADJAZENZLISTEN 169

```
/****** UndiGraph.java ******/
/** Klasse zur Implementation eines ungerichteten Graphen */
/* basierend auf UndiVertex und UndiEdge */
/* Der Graph wird implementiert als HashMap <String, Vertex>, d.h. als eine */
/* Hashtabelle mit Keys vom Typ String und Values vom Typ Knoten */

import java.util.*;
public class UndiGraph {

    private Map <String, UndiVertex> graph; // Datenstruktur fuer Graph

    public UndiGraph() { // leerer Graph wird angelegt
        graph = new HashMap<String,UndiVertex>(); // HashMap von String, UndiVertex
    }

    public boolean isEmpty(){ // liefert true, falls Graph leer
        return graph.isEmpty(); // mit isEmpty() von HashMap
    }

    public int size(){ // liefert die Anzahl der Knoten
        return graph.size(); // mit size() von HashMap
    }

    public Collection <UndiVertex> vertices(){ // liefert Knoten als Collection
        return graph.values(); // mit values() von HashMap
    }

    public Collection <UndiEdge> edges() { // liefert Kanten als Collection
        Set s = new HashSet<UndiEdge>(); // erzeuge Menge von Kanten
        for (UndiVertex v : vertices()) { // fuer jeden Graphknoten
            for (UndiEdge e : v.edges) // durchlaufe seine Kanten
                s.add(e); // (doppelte werden vermieden)
        }
        return s;
    }

    public UndiVertex getVertex(String s){ // liefere Knoten zu String
        UndiVertex v = graph.get(s); // besorge Knoten zu Knotennamen
        if (v==null) { // falls nicht gefunden
            v = new UndiVertex(s); // lege neuen Knoten an
            graph.put(s, v); // fuege Namen und Knoten ein
        }
        return v; // liefere gefundenen oder neuen Knoten
    }

    public void addEdge(String source, // fuege Kante ein von Knoten source
                       String dest, // zu Knoten dest
                       double cost) { // mit Kosten cost
        UndiVertex v = getVertex(source); // finde Knoten v zum Startnamen
        UndiVertex w = getVertex(dest); // finde Knoten w zum Zielnamen
        UndiEdge e= new UndiEdge(v,w,cost); // erzeuge ungerichtete Kante
        v.edges.add(e); // fuege Kante (v,w,c) bei Knoten v ein
        w.edges.add(e); // fuege Kante (v,w,c) bei Knoten w ein
    }
}
```

```
/****** UndiGraphIO.java *****/

import java.lang.*;
import java.util.*;
import java.io.*;
import AlgoTools.IO;

/** Routinen zum Einlesen und Ausgeben eines ungerichteten Graphen      */
/* Der Graph wird realisiert durch eine HashMap,                        */
/* welche den Namen des Knoten auf den Knoten abbildet.                */
/* Es wird erwartet, dass jede ungerichtete Kante nur einmal gelesen wird */

public class UndiGraphIO {

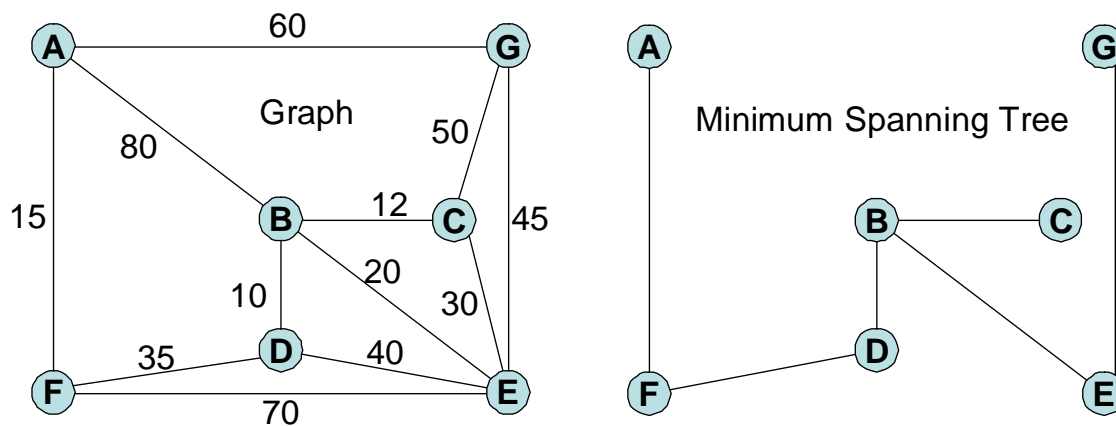
    public static UndiGraph readUndiGraph() { // liest unger.Graph aus Datei ein
        UndiGraph g = new UndiGraph();
        try {
            BufferedReader f = new BufferedReader(new FileReader("undigraph.dat"));
            String zeile;
            while ( (zeile = f.readLine()) != null) {
                StringTokenizer st = new StringTokenizer(zeile);
                String source = st.nextToken();
                String dest = st.nextToken();
                double cost = Double.parseDouble(st.nextToken());
                g.addEdge(source,dest,cost);
            }
        } catch (Exception e) {IO.println(e);}
        return g;
    }

    public static void printUndiGraph(UndiGraph g) { // gibt Graph aus
        IO.println("Adjazenzlisten des Graphen:\n");
        for (UndiVertex v : g.vertices()) {
            for (UndiEdge e : v.edges) {
                if (e.left==v)
                    IO.print("(" + e.left.name + "," + e.right.name + ")");
                else
                    IO.print("(" + e.right.name + "," + e.left.name + ")");
                IO.print(e.cost + " ");
            }
            IO.println();
        }
        IO.println();
    }
}
```

12.10 Minimaler Spannbaum

Gegeben ein ungerichteter, gewichteter Graph $G = (V, E)$, dessen Kanten mit der Funktion $c : E \rightarrow \mathbb{N}$ gewichtet sind. Gesucht wird eine Teilmenge von Kanten mit möglichst geringen Gesamtkosten, die alle Knoten verbindet. Hat der Graph n Knoten, so besteht der Spannbaum aus $n - 1$ Kanten.

Probleme dieser Art entstehen immer dann, wenn ein kostengünstiges Netzwerk gesucht wird, welches z.B. eine Menge von Inseln durch Brücken oder ein Menge von Ortschaften durch Stromleitungen verbindet.



Ein Lösungsverfahren wurde von Kruskal vorgestellt:

```

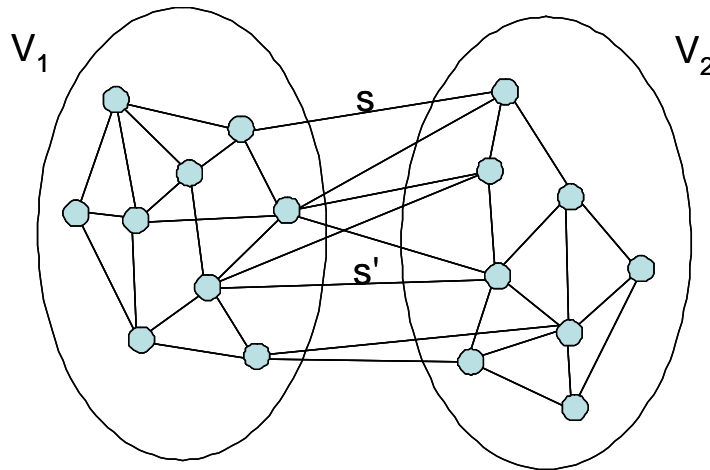
Initialisiere einen Wald von Bäumen durch n isolierte Knoten;
Initialisiere einen Heap mit allen gewichteten Kanten;
while noch nicht genügend Kanten gefunden {
    entferne die billigste Kante (x,y) aus dem Heap;
    falls x und y im verschiedenen Bäumen sind;
    vereinige die beiden Bäume;
}

```

Die Korrektheit des Kruskal-Algorithmus folgt aus dem folgenden

Satz: Gegeben sei eine Partition V_1, V_2 der Knotenmenge V . Sei s die billigste Kante zwischen V_1 und V_2 . Dann gibt es einen MSB (Minimaler Spannbaum) der s enthält. (s ist sogar in allen MSBs enthalten).

Beweis:



Annahme: s sei nicht im MSB enthalten. Dann füge s in den MSB ein. Es entsteht ein Zyklus, der über s' läuft. Entferne nun s' aus dem MSB. Es entsteht ein neuer, billigerer MSB. Dies ist ein Widerspruch zur Minimalität des Ausgangs-MSB.

Zur Verwaltung der Zusammenhangskomponenten notieren wir bei jedem Knoten x in einer Arbeitsvariable `chef` denjenigen Knoten, der zur Zeit als Vorgesetzter für ihn fungiert. Ein Knoten, der sein eigener Vorgesetzter ist, übernimmt die Rolle des Repräsentanten für alle Knoten, die in seiner Zusammenhangskomponente liegen und die über die Variable `chef` direkt oder indirekt auf ihn verweisen. Bei der Vereinigung von zwei Zusammenhangskomponenten mit den Repräsentanten x und y muss nur einer der beiden (z.B. x) den anderen (d.h. y) als neuen Chef eintragen.

Damit die Suche nach dem aktuellen Chef nicht zu immer länger werdenden Verweisketten führt, benutzt man die Technik des *Path halving*: Längs des Weges von einem Knoten x zu seinem Repräsentanten z wird bei jedem Knoten der aktuelle Vater durch den Großvater ersetzt. Dieser Mehraufwand zahlt sich insofern aus, als bei der nächsten Repräsentantensuche der Weg nur noch halb so lang ist.

Hat der Graph n Knoten und m Kanten, dann beträgt der Aufwand für den Kruskal-Algorithmus $O(m \cdot \log(m) \cdot \text{lg}^*(m))$, da maximal jede Kante einmal mit Aufwand $\log m$ aus dem Heap genommen wird. Der zusätzliche Faktor $\text{lg}^*(m)$ bezeichnet eine fast konstante, sehr langsam wachsende Funktion (nämlich die Inverse der Ackermannfunktion, für alle praktischen Werte von m kleiner als 5). Sie wird verursacht durch das Testen mit Path Halving, ob zwei Knoten in derselben Zusammenhangskomponente liegen.

```

/***** Kruskal.java *****/

/** bestimmt einen minimalen Spannbaum nach der Idee von Kruskal */
/** verwendet wird eine PriorityQueue zum Verwalten der gewichteten Kanten */
/** */
/** Um die Suche nach dem Repraesentanten einer Zusammenhangskomponente zu */
/** beschleunigen, wird die Technik des Path-halving eingesetzt: auf dem Weg */
/** zum Repraesentanten wird jeder Vater durch den Grossvater ersetzt */

import java.util.*;

public class Kruskal {

    private static UndiVertex findAndUpdateChef(UndiVertex v) {
        while (v.chef != v) { // solange der Knoten kein Repraesentant ist
            v.chef = v.chef.chef; // ersetze Vater durch Grossvater
            v = v.chef; // steige zum Grossvater hoch
        }
        return v; // liefere den ermittelten Repraesentanten
    }

    public static void kruskal(UndiGraph g) {

        PriorityQueue<UndiEdge> p = // Priority-Queue zum Verwalten
            new PriorityQueue<UndiEdge>(); // aller Kanten gemaess Gewicht

        for (UndiVertex v : g.vertices()){ // jeder Knoten
            v.chef = v; // ist sein eigener Chef
        }

        for (UndiEdge e : g.edges()) { // jede Kante
            e.status = false; // ist noch nicht gewaehlt
        }

        p.addAll(g.edges()); // alle Kanten in die Priority-Queue

        int count = 0; // bisher noch keine Kanten gefunden

        UndiVertex x,y; // 2 Knoten
        while (count < g.size()-1) { // solange noch nicht genug Kanten
            UndiEdge e = p.poll(); // besorge die billigste Kante
            x = findAndUpdateChef(e.left); // bestimme Repraesentanten links
            y = findAndUpdateChef(e.right); // bestimme Repraesentanten rechts
            if (x != y) { // falls in verschiedenen Komponenten
                x.chef = y; // vereinige Komponenten
                e.status = true; // notiere Kante e
                count++;
            }
        }
    }
}

```

```

/***** KruskalTest.java *****/

import java.util.*;
import AlgoTools.IO;

/** testet den Kruskalalgorithmus auf einem ungerichteten Graphen */

public class KruskalTest {

    public static void main(String [] argv) {

        UndiGraph g = UndiGraphIO.readUndiGraph(); // Graph einlesen

        UndiGraphIO.printUndiGraph(g); // Graph ausgeben

        Kruskal.kruskal(g); // berechne Spannbaum

        IO.print("Der minimale Spannbaum "); // gib den errechneten
        IO.println("besteht aus folgenden Kanten:\n "); // Spannbaum aus
        for (UndiEdge e : g.edges()){ // durchlaufe alle Kanten
            if (e.status)
                IO.println("(" + e.left.name + "," + e.right.name + ") " + e.cost);
        }
    }
}

```

Adjazenzlisten des Graphen:

```

(D,B)10.0 (D,E)40.0 (D,F)30.0
(E,B)20.0 (E,C)20.0 (E,D)40.0 (E,F)70.0 (E,G)40.0
(F,A)20.0 (F,D)30.0 (F,E)70.0
(G,A)60.0 (G,C)50.0 (G,E)40.0
(A,B)80.0 (A,F)20.0 (A,G)60.0
(B,A)80.0 (B,C)10.0 (B,D)10.0 (B,E)20.0
(C,B)10.0 (C,E)20.0 (C,G)50.0

```

Der minimale Spannbaum besteht aus folgenden Kanten:

```

(B,C) 10.0
(B,E) 20.0
(B,D) 10.0
(D,F) 30.0
(A,F) 20.0
(E,G) 40.0

```

von KruskalTest.java erzeugte Ausgabe undiresult.dat

12.11 Matching

Ein Matching M in einem ungerichteten Graph $G = (V, E)$ besteht aus einer Menge von unabhängigen Kanten, d.h. Kanten, die keine gemeinsamen Endknoten haben. Ein Matching modelliert mögliche Zuordnungen in einem System von binären Beziehungen, z.B. Spielpartner in einem Tennisturnier.

Bzgl. eines Matchings M wird ein Knoten *frei* genannt, wenn er nicht Endpunkt einer Matchingkante ist; der Knoten heißt *belegt*, wenn er Endpunkt von (genau) einer Matching-Kante ist. In einem gewichteten Graphen werden die Kosten eines Matchings durch die Summe der Gewichte der benutzten Kanten definiert.

M ist ein **Maximal-Matching**, wenn zu M keine weitere Kante hinzugefügt werden kann.

M ist ein **Maximum-Matching**, wenn es keine anderen Matchings mit mehr Kanten gibt.

M ist ein **perfektes Matching**, wenn alle Knoten belegt sind.

M ist ein **Minimum-Cost-Matching**, wenn es von allen perfekten Matchings die geringsten Kosten verursacht.

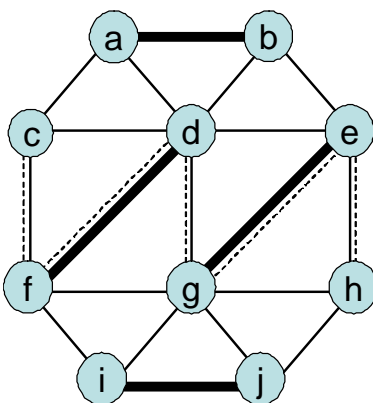
Maximale Matchings können mit einem Greedyverfahren bestimmt werden, indem solange Kanten zwischen freien Knoten gewählt werden, bis dies nicht mehr möglich ist.

Maximum-Matchings können bestimmt werden, indem ausgehend von einem initialen Matching sogenannte erweiternde Wege gefunden werden, längs derer das Matching vergrößert wird. Genauer: Ein alternierender Weg bzgl. eines Matchings M ist eine Sequenz von Kanten mit jeweils gemeinsamen Endpunkten, die abwechselnd zum Matching und nicht zum Matching gehören. Ein erweiternder Weg bzgl. eines Matchings M ist ein alternierender Weg bzgl. M , der bei einem freien Knoten beginnt und bei einem freien Knoten endet. Durch Umtauschen der Matchingkanten in Nicht-Matchingkanten und umgekehrt kann längs des erweiternden Weges das Matching um eine Kante erhöht werden.

Es gilt der Satz:

M ist ein Maximum-Matching \iff es gibt keinen erweiternden Weg mehr.

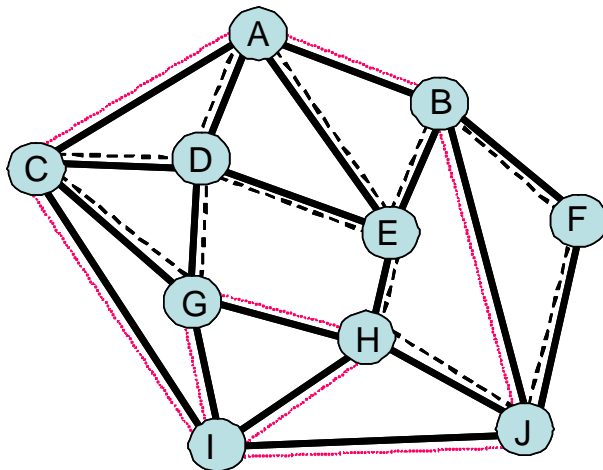
Folgendes Bild zeigt ein maximales Matching bestehend aus vier (dick gezeichneten) Kanten. Dieses Matching lässt sich durch den alternierenden (gestrichelt gezeichneten) Weg über die Knoten c, f, d, g, e, h um eine Kante erweitern. Es besteht dann aus den 5 Kanten $(a, b), (c, f), (d, g), (e, h), (i, j)$ und bildet ein Maximum-Matching.



12.12 Chinese Postman

Vorbemerkung: Ein ungerichteter Graph heißt *Eulergraph*, falls alle Knoten geraden Knotengrad haben. Ein Eulergraph erlaubt immer eine sogenannte *Eulertour*, d.h. einen geschlossenen Kantenzug, in dem jede Kante genau einmal vorkommt. Die Konstruktion einer Eulertour geschieht durch willkürliches Ablaufen noch nicht benutzter Kanten und nutzt aus, dass ein Knoten, der über eine noch nicht benutzte Kante angelaufen wird, immer (wegen seines geraden Knotengrads) eine noch nicht benutzte Kante aufweist, über die der Weg fortgesetzt werden kann. Letztendlich muss diese Kantenfolge wieder beim Ausgangsknoten ankommen. Ggf. müssen mehrere Kantenzüge, die auf diese Weise konstruiert wurden, zu einem Gesamtweg verschmolzen werden.

In folgendem Graph haben alle Knoten geraden Grad. Eine erste Kantenfolge (gestrichelt gezeichnet) beginnt bei F , läuft dann über $J, H, E, D, G, C, D, A, E, B$ und kehrt zu F zurück. Eine weitere Kantenfolge (gepunktet gezeichnet) beginnt bei C , läuft über A, B, J, I, H, G, I und kehrt zu C zurück. Beide Folgen können z.B. am Knoten H verschmolzen werden, indem die gestrichelte Folge bei Ankunft in H zunächst die gepunktete Folge abläuft und dann die gestrichelte Folge weiter führt. Es ergibt sich als Eulertour $F, J, H, G, I, C, A, B, J, I, H, E, D, G, C, D, A, E, B, F$.



Gegeben sei nun ein ungerichteter Graph $G = (V, E)$, gewichtet mit einer Kostenfunktion $c : E \rightarrow \mathbb{N}$.

Das Problem des Chinese Postman besteht darin, eine billigste Kantenrundreise zu bestimmen, d.h. eine geschlossene, möglichst billige Kantenfolge über alle Knoten, in der jede Kante mindestens einmal auftritt. Dies soll die Aufgabe eines Briefträgers modellieren, der kostengünstig alle Straßen einer Stadt abgehen und wieder zum Ausgangspunkt zurückkehren soll.

Die Lösung dieses Problems hängt mit der Euler-Eigenschaft zusammen:

- Ist G ein Eulergraph, so liefert die Eulertour die billigste Postman-Tour.
- Ist G kein Eulergraph, so muss G durch Verdoppelung einiger Kanten in einen Eulergraph G' überführt werden. Gesucht wird daher ein System von kantendisjunkten kürzesten Wegen zwischen denjenigen Knoten in V , die ungeraden Grad haben.

Offensichtlich wird in einer optimalen Lösung jede Kante maximal zweimal durchlaufen. Denn würde

sie öfter durchlaufen, so ließen sich zwei Durchläufe streichen, der Graph bliebe weiterhin eulersch und die Tour wäre verbilligt.

Sei $U := \{n_1, n_2, \dots, n_k\}$ die Menge der Knoten aus V mit ungeradem Knotengrad. Offenbar gilt: k ist gerade, denn in einem Graph ist die Anzahl der Knoten mit ungeradem Kantengrad gerade.

Auf der Menge U errechne mit dem Algorithmus von Floyd die Distanzmatrix d_{ij} , d.h. die Kosten der kürzesten Wege zwischen allen Paaren aus U unter Verwendung aller Knoten und Kanten aus G .

Betrachte nun die Clique dieser k Knoten mit ihren errechneten Kantengewichten d_{ij} .

In dieser Clique suchen wir ein Minimum Cost Matching M . Dieses Matching liefert uns die Knotenpaare, die mit einem kürzesten Weg gemäß Floyd verbunden werden müssen. Dadurch erhalten alle Knoten einen geraden Grad. Wir erhalten also einen Eulergraph, der einen geschlossenen Weg über alle Kanten erlaubt und damit die gesuchte Postman-Tour darstellt.

Es lässt sich nachweisen, dass die zugehörigen Wege alle kantendisjunkt sind. Denn würde eine Kante (x, y) gemeinsam auf dem kürzesten Weg von a nach b und von c nach d genutzt, so könnten durch Entfernen der Kante (x, y) zwei billigere Wege von a über x nach c und von b über y nach d entstehen.

Im folgenden Graphen gibt es vier Knoten mit ungeradem Grad (fett markiert). Sie werden preisgünstig durch kürzeste Wege (gestrichelt markiert) miteinander verbunden. Längs dieser Wege werden nun alle Kanten verdoppelt, so dass dadurch alle Knoten geraden Grad haben. Auf diesem Eulergraph bildet dann eine Eulertour die Lösung des Chinese Postman.

