Wichtige Befehle in Java:
Länge von Arrays: .length
Objekte inhaltlich mittels equals vergleichen, mittels == werden Referenzen verglichen.

1 Algorithmisches Denken

Definition (*Informatik*)

Kunstwort aus den 60er Jahren, das die Assoziationen Informatik gleich Information und Technik oder Information und Mathematik erwecken sollte

Definition (*Algorithmus*)

Folge einfacher Anweisungen, die folgende Eigenschaften aufweist:
Endlichkeit, Terminierung, eindeutige Reihenfolge, eindeutige Wirkung

Vorgehensweise bei der Lösung von Programmierproblemen:
→ Problem verstehen und abstrahiert beschreiben
→ Problemlösung in algorithmischer Form beschreiben
→ Algorithmus in Programmiersprache implementieren (Hochsprache)
→ Übersetzung des Programms in Maschinencode (Compiler)
→ Überprüfung: beschreibt die Spezifikation wirklich das Problem?
→ Nachweis, dass Algorithmus das spezifierte Problem korrekt löst (Semantik)
→ Setzt das Programm den Algorithmus korrekt um?
→ Aufwandsanalyse

2 Grundlagen der Programmierung

2.1 Variablen

Definition (*Variablendeklaration*)

legt Datentyp und Namen einer Variable fest
bei Übersetzung wird Variable im Speicher erzeugt und Speicher reserviert (je nach Datentyp)
Datentyp dient dazu, Speicherinhalt korrekt zu interpretieren

Bezeichner können aus kleinen und großen lat. Buchstaben, Ziffern, _ und $ bestehen, jedoch keine Leerzeichen und keine Schlüsselwörter. Sollten mit Kleinbuchstaben beginnen und selbsterklärend sein (mnemonische Namen).

Konstanten werden am Anfang eines Blocks definiert und in Großbuchst. geschrieben, _ zur Worttrennung

2.2 Datentypen

Datentyp: Menge von Daten gleicher Art

Definition (*primitive Datentypen*)

ganze Zahlen:	byte:	$-2^7 =<$	byte	$=< 2^7 - 1$
	short:	$-2^{15} =<$	short	$=< 2^{15} - 1$
	int:	$-2^{31} =<$	int	$=< 2^{31} - 1$
	long:	$-2^{63} =<$	long	$=< 2^{63} - 1$ (Anhängen von L)
Fließkommazahlen:	dargestellt gemäß IEEE, float (32 bit), double (64 bit)			
	⇒ für genaue Rechnungen ungeeignet!			
Zeichen:	char (zwischen ' ', Zahlen in ASCII oder UNICODE interpretiert)			
boolescher Wert:	boolean (wahr oder falsch)			

Entscheidungsgehalt: Anzahl an Bit, die man benötigt, um jedes Element einer Menge eindeutig mit einem Bitmuster zu identifizieren:

$H(M) = \lceil log_2(|M|) \rceil$

Zahlen in Binär: vorangestelltes 0b/0B
Zahlen in Hex: vorangestelltes 0x
Zahlen in Oktal: vorangestellte 0

Aufzählungstyp enum < Name > {ELEMENT1, ELEMENT2, ...}

Array: zusammengesetzter Datentyp, endliche Folge von Werten eines Datentyps
Deklaration: int[] a={viele viele bunte Werte} oder int[ ] a=new int[10] ACHTUNG: Indizierung beginnt bei 0.

2.3 Operatoren und Ausdrücke

+	$x + y$	Addition	>	$x > y$	x größer y
−	$x - y$	Subtraktion	>=	$x \geq y$	x größer gleich y
*	$x + y$	Multiplikation	<	$x < y$	x kleiner y
/	$x - y$	Division	<=	$x \leq y$	x kleiner gleich y
%	$x + y$	Modulo	==	$x == y$	x gleich y
$a++$	$++a$	Inkrementierung	!=	$x! = y$	x ungleich y
$a--$	$--a$	Dekrementierung	&&	||	logisches Und/Oder
^		Bitoperator entweder-oder	&	|	Bitoperator Und/Oder

Bit-Verschiebung:
<< left shift. Bsp: 15<<3 0000 1111 wird zu 0111 1000
>> right shift sign. Beachtet Vorzeichen, von links altes Vorzeichenbit einschieben.
>>> right shift no-sign. ignoriert Vorzeichen, von links neue Nullen einschieben.

Auswertungsreihenfolge, von unten nach oben:

Postfix-Operatoren	[] . (params) expr++ expr–
unäre Operatoren	++expr –expr +expr -expr !
Erzeugung oder Typumwandlung	new (type) expr
Multiplikationsoperatoren	* / %
Additionsoperatoren	+ -
Verschiebeoperatoren	<< >> >>>
Vergleichsoperatoren	< > <= >= instanceof
Gleichheitsoperatoren	== !=
Bitoperator Und	&
Bitoperator exklusives Oder	^
Bitoperator inklusives Oder	|
logisches Und	&&
logisches Oder	||
Fragezeichenoperator	?
Zuweisungsoperatoren	= += -= *= /= %= >>= <<= >>>= &= ^= |=

2.4 Typkonvertierung

implizite Typkonvertierung: Automatische Konvertierung durch Compiler zu größeren Datentypen
explizite Typkonvertierung: Konvertierung zu kleinerem Datentypen, muss von Programmierer explizit gefordert werden.
Typkonvertierungsoperator: (Zieldatentyp) Wert

3 Ablaufstrukturen und Methoden

if-else:
if (<Bedingung>) {
<Anweisungen>
}else {
<Anweisungen>
}

switch:
switch (<Ausdruck>) {
case <konstanter Ausdruck>: <Anweisung>: break;
...
default: <Anweisung>;
}

while (<Bedingung>) {
<Anweisungen>
}

do {
<Anweisungen>
} while (<Bedingung>);

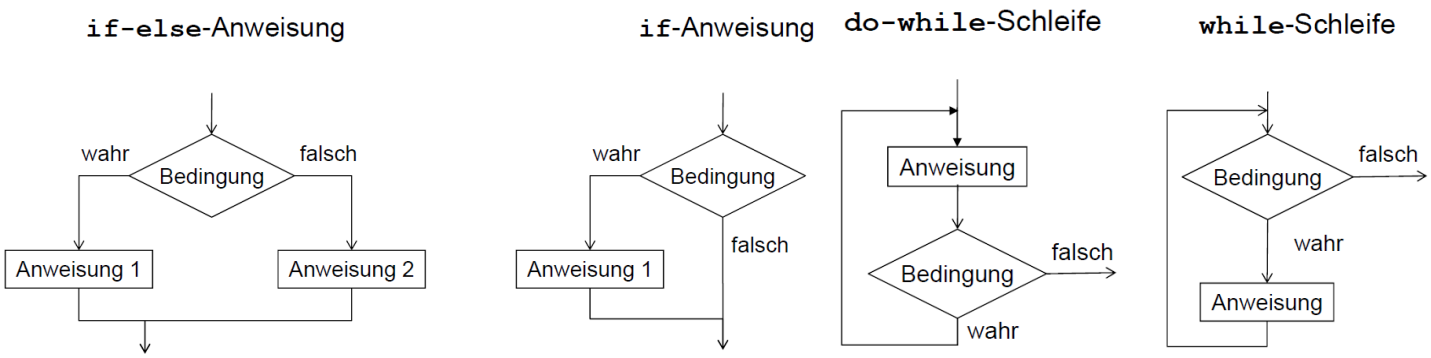
while-Schleifen, wenn die Anzahl an Iterationen unbekannt ist.

for:
for (<Initialausdruck>; <Bedingung>; <Inkrementausdruck>) {
<Anweisungen>
}
for-Schleifen, wenn die Anzahl an Iterationen bekannt ist.

bedingter Ausdruck:
allgemein: <Bedingung> ? <Ausdruck1> : <Ausdruck2>
statt
int max;
if ($i > j$) max= i; else max=j;

könnte man auch schreiben:
int max = ($i > j$) ? i : j;

3.1 Ablaufdiagramme



3.2 Methoden

Deklaration:
<Sichtbarkeitsmodifikator> <Rückgabetyp> <Methodenname> (Param 1, ..., Param n){
Anweisungen;
}

Methodenaufruf: <Objektname>.<Methodenname>(<Eingabewerte>)

4 Rekursion

Definition (*Rekursion*)

Eine Funktion heißt rekursiv, wenn zur Berechnung von f diese Funktion f wieder benutzt wird.
Eine Methodendeklaration f(x) { ...} heißt rekursive Methodendefinition, gdw. f im Block {...} aufgerufen wird. Funktion ruft sich selbst auf ⇒ Rekursionsschritt

Voraussetzungen: Selbstähnlichkeit, Monotonieeigenschaft, Beschränktheit

Eine Funktion heißt linear rekursiv, wenn sie in jedem Zweig einer Fallunterscheidung ihres Rumpfes höchstens einmal aufgerufen wird.
Eine Methode f heiß endrekursiv, wenn der rekursive Methodenaufruf stets die letzte Aktion des Zweigs zur Berechnung von f ist. Kann unmittelbar entrekursiviert werden: rekursiv besser zum Aufschreiben, iterativ schneller in der Ausführung.
Eine Methode f heißt kaskadenartig rekursiv, wenn in einem Zweig einer Fallunterscheidung im Rumpf von f zwei oder mehr Aufrufe von f auftreten.
Zwei Methoden f und g heißen verschränkt (auch: wechselseitig) rekursiv, wenn die Methodendeklaration von f einen Aufruf der Methode g enthält und die Methodendeklaration von g einen Aufruf der Methode f enthält.
Eine Methode f heißt verschachtelt rekursiv, wenn man zur Bestimmung der Parameter des rekursiven Aufrufs einen rekursiven Aufruf der Funktion ausführen muss.

Terminierungsbeweis:
finde ganzzahlige Terminierungsfunktion und beweise, dass t bei jedem Rekursionsschritt streng monoton fällt und nach unten beschränkt ist.

4.1 Entrekursivierung

$f(x) \quad = g(x)$ falls $praed(x)$ (nicht rekursiver Basisfall)
$\quad\quad = f(r(x))$ sonst (rekursiver Aufruf)

arg = x; // Hilfsvariable
while (!$praed(arg)$) {
    arg = $r(arg)$; // modifiziere Parameter
}
return $g(arg)$; // nicht-rekursiver Basisfall

5 Rekursion im Einsatz

1. Durchreichen von Zwischenergebnissen
2. Dynamisches Programmieren:

Mithilfe von einer Liste werden alle schon berechneten Werte gespeichert(erhöhter Speicherbedarf)
Vorgehen:
→ rekursive Version implementieren
→ Tabelle zum Zwischenspeichern der Werte erzeugen und initialisieren
→ rekursive Version um Lookup/Memoization erweitern
  → Tabelle als zusätzliche Eingabe
  → zu berechnenden Wert in der Tabelle nachschlagen (Lookup)
  → falls bereits berechnet: Wert direkt zurückgeben
  → ansonsten:
    → Wert rekursiv berechnen
    → Wert in der Tabelle zwischenspeichern (Memoization)
    → Wert zurückgeben

Backtracking, Rücksetzverfahren:
systematisches Durchsuchen des Suchraums, dabei Anwendung von trial-and-error
Grundgerüst:

```
static int[ ][ ] backtrack(int[ ][ ] state)  //z.B. Sudoku
   if (isFinal(state)) {
      return state;
    } else {
      // moegliche Erweiterungen aufzaehlen
      int[] candidates = getExtensions(state);
      for (int i = 0; i ¡ candidates.length; i++) {
         int c = candidates[i];
         state = apply(state, c); //z.B. Ziffer setzen
         if (backtrack(state) != null) { // Rekursion
            return state;
         }
         state = revert(state, c); //z.B. Ziffer loeschen
      }
      return null;
    }
```

6 Asymptotische Aufwandsanalyse

Qualitätskriterien für Algorithmen:
Korrektheit, hohe Verständlichkeit, einfache Implementierbarkeit, geringstmöglicher Bedarf an Ressourcen

Der exakte Aufwand eines Algorithmus (z.B. als Anzahl Rechenschritte bis zur Terminierung oder als verbrauchte Zeit) kann nur schwer als Funktion von Umfang und Eingabewerten allgemein angegeben werden. Wichtiger ist die ungefähre Größenordnung, mit der der Aufwand in Abhängigkeit vom Umfang der Eingabe wächst.
Eine Elementaroperation entspricht genau einer Instruktion des zugrunde liegenden Von-Neumann-Rechners und hat das Kostenmaß 1.

Definition (*O-Notation (asympt. obere Schranke), Groß-O*)

$f(n) \in O(g(n)) : f(n)$ wächst höchstens so schnell wie $g(n)$
$O(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ | \exists n_0 \in \mathbb{N}, c \in \mathbb{R}, c > 0 : \forall n \geq n_0 \ f(n) \leq c \cdot g(n)\}$

Definition (Ω - *Notation*)

$f(n) \in \Omega(g(n)) : f(n)$ wächst mind. so schnell wie $g(n)$
$\Omega(g(n)) = \{h(n) | \exists c > 0 \ \exists n_0 > 0 \ \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq h(n)\}$

Definition (Θ - *Notation*)

$f(n) \in \Theta(g(n)) : f(n)$ wächst ebenso so schnell wie $g(n)$
$\Theta(g(n)) = \{h(n) | \exists c_1 > 0 \ \exists c_2 > 0 \ \exists n_0 > 0 \ \forall n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq h(n) \leq c_2 \cdot g(n)\}$

Definition (*Rechenregeln für die O-Notation*)

Seien $f(n) \in O(r(n)), g(n) \in O(s(n))$ und $c > 0$ konstant. Dann gilt:
a) $f(n) + g(n) \in O(r(n) + s(n)) = O(max(r(n), s(n)))$
b) $f(n) \cdot g(n) \in O(r(n) \cdot s(n))$
c) $c \cdot f(n) \in O(r(n))$
d) $f(n) \pm c \in O(r(n))$

7 Objektorientierte Programmierung: Klassen und Objekte

Datentyp bisher: Daten gleicher Sorte, z.B. ganze Zahlen, Wahrheitswerte, Zeichenketten.
Jetzt: Konzentration auf das Klassen- und Objektkonzept, das Grundlage der objektorientierten Programmierphilosophie ist.

Definition (*Klasse*)

Klasse (auch Objekttyp): Konzept der objektorientierten Modellierung und Programmierung, mit dem neue Datentypen definiert werden können.
Die Deklaration einer Klasse stellt damit eine Art Bauanleitung oder Schablone für sog. Objekte (auch: Exemplare, Instanzen) dieser Klasse dar.
Klassennamen sollen selbsterklärend sein und die Lesbarkeit des Programms fördern sowie Substantive sein und mit einem Großbuchstaben beginnen.

Konkret wird in der Klassendeklaration festgelegt:
- aus welchen Datenkomponenten (sog. Attributen) ein Objekt besteht
- welche Methoden, z.B. zur Manipulation der zugehörigen Attributwerte, zur Verfügung stehen und
- welche Konstruktoren zur Erzeugung von Objekten (auch Instanziierung) verwendet werden können

Definition (*Attribute*)

Attribute sind Datenkomponenten, aus denen Objekte einer Klasse bestehen. Deklaration und Benennung erfolgen analog zu Variablen. Erlaubt sind Attribute primitiven und zusammengesetzten Typs.

Definition (*Konstruktoren*)

Konstruktoren sind Operationen, die es ermöglichen, Objekte einer Klasse zu instanziieren, d.h. anzulegen oder zu erzeugen. Alle Konstruktoren einer Klasse haben denselben Namen, nämlich den der Klasse, zusätzlich können sie Parameter haben. Ein Rückgabetyp wird nicht angegeben.

1. explizite Konstruktoren
Sie werden bei der Klassendeklaration angegeben und i.d.R. dazu genutzt, die Attributwerte des zu erzeugenden Objekts auf für den Anwendungszweck sinnvolle Startwerte zu setzen. Explizite Konstruktoren können Parameter haben. Eine Klasse kann mehrere explizite Konstruktoren haben, die aber unterschiedliche Parameterlisten haben müssen.
2. impliziter Konstruktor (auch Standard-Konstruktor, Default-Konstruktor)
Wird kein expliziter Konstruktor angegeben, so legt der Übersetzer automatisch einen impliziten Konstruktor an. Ein impliziter Konstruktor hat immer die Form <Klassenname>(){ }.

Ein objektorientiertes Programm in Java besteht aus Folge von Klassendeklarationen, die jeweils Attribut-, Konstruktor- und Methodendeklarationen enthalten.
In einer der Klassen muss die Methode main deklariert sein, mit der die Ausführung des Gesamtprogramms startet.
Klasse, die die Methode main deklariert, nimmt i.d.R. Sonderrolle ein, nämlich die der Spezifikation des Hauptprogramms und nicht die der Deklaration des Datentyps einer Klasse von Objekten. In der Regel beginnt dort die Erzeugung der für die Problemlösung relevanten Objekte anhand der Klassendeklarationen. Diese Objekte verwalten eigene Daten (Werte der Attribute) und sie reagieren auf Nachrichten (Aufrufe von Methoden, die in der Klasse des Objekts deklariert wurden).
Problemlösung durch schrittweise Objekterzeugung, - verknüpfung und - kommunikation.

Definition (*Objekt*)

Objekt ist Datenstruktur, deren Datentyp (Struktur, Verhalten) durch eine Klasse festgelegt ist. Objekt ist konkrete Ausprägung einer Klasse, deshalb auch Instanz genannt.
Klassendeklaration legt fest, welche Attribute und Methoden für ein Objekt der Klasse zur Verfügung stehen und in welcher Form eine Instanziierung stattfinden kann.
Während eine Klasse die Attribute i.d.R. nur deklariert, werden bei einem Objekt die Attribute durch den verwendeten Konstruktor oder durch Methodenaufrufe mit Attributwerten belegt.

Definition (*Objektvariablen*)

Attribute von primitivem Typ sind Behälter im Objekt. Sie verhalten sich wie die Ihnen bekannten Variablen.
Attribute vom Typ einer Klasse (sog. Objektvariablen) können einen Verweis/eine Referenz auf ein Objekt enthalten.
Bei der Deklaration einer Objektvariable wird eine Variable angelegt, die einen Verweis auf ein Objekt der zugehörigen Klasse aufnehmen kann (deshalb auch Referenzvariable).
Bei der Deklaration wird noch keine Objektinstanz erzeugt, der Wert der neu angelegten Variable ist eine sog. Null-Referenz.

Definition (*Instanziierung*)

Instanziierung bezeichnet die Erstellung eines neuen Objekts einer Klasse. Das Objekt wird erzeugt, d.h. der entsprechende Platz im Speicher reserviert. Verweis auf dieses Objekt bzw. auf den Speicherbereich wird zurückgegeben.
Form: Schlüsselwort new gefolgt von Konstruktoraufruf.

Definition (*Klassenattribut/Klassenmethode*)

Klassenattribut (auch: statisches Attribut) ist Sonderfall, bei dem sich alle Objekte der Klasse denselben Attributwert teilen. Schreiboperationen in einem Objekt wirken sich damit auf alle anderen Objekte aus.
Schlüsselwort static.
Zugriff auf Klassenattribute von außen hat die Form <klassenname>.<attributname> oder <objektname>.<
von innen <attributname>

Klassenmethoden sind auch ohne instanziiertes Objekt verwendbar, da sie einer Klasse und nicht einem Objekt zugeordnet sind und bieten damit die Möglichkeit, Methoden direkt auf einer Klasse selbst auszuführen.
Man verwendet sie als Getter und Setter für Klassenattribute oder wenn für die Methode kein Objekt der Klasse benötigt wird oder generell zu einer Klasse keine Objekte instanziiert werden sollen. So sparsam wie möglich zu verwenden, da eig. nicht der objektorientierten Philosophie entsprechend.

Definition (*Sichtbarkeit von Information*)

Oft ist nur wichtig, was ein Objekt leistet, nicht aber, wie es realisiert ist. Es werden Programmiertechniken benötigt, mit denen man den Zugriff auf Attribute und Methoden von Objekten einschränken kann.
Sichtbarkeits-Modifizierer:
keiner (paketsichtbar): aus den Klassen desselben Paktes
public: aus allen Klassen heraus
private: nur aus definierender Klasse heraus
protected: aus definierender Klasse, allen Unterklassen und Klassen desselben Pakets

Definition (*Datenkapselung*)

Bezeichnet das Verbergen von Programmierdetails vor dem direkten Zugriff von Außen. Direkter Zugriff auf interne Datenstruktur wird unterbunden und erfolgt stattdessen über wohl definierte Schnittstellen (Black-Box-Modell).

Definition (*Wrapper-Klassen*)

Viele Methoden erwarten Parameter von Typ Object. Um diesen Methoden Werte eines primitiven Datentyps zu übergeben, stell Java für jeden Datentyp eine sog. Wrapper-Klasse zur Verfügung, die diesen Wert kapselt.

7.1 UML - Unified Modelling Language

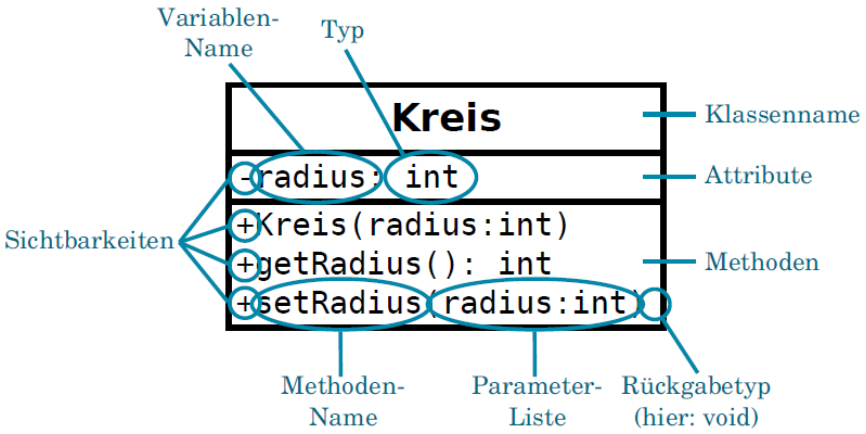
Assoziation: setzt Objekte von genau 2 Klassen zueinander in Beziehung
Einfache Linie zw. den Klassen, Name an der Mitte dieser notiert, Anzahlangaben (mit wie vielen Objekten der gegenüberliegenden Assoziationsseite ist je ein Objekt mind./höchst. verbunden)

Multiplizitäten: geben an, wie viele Objekte der an der Assoziation beteiligten Klassen jeweils miteinander in Beziehung stehen können.
0..* wird mit Liste angegeben

Aggregation: Spezialform der Assoziation, modelliert eine "hat-Beziehung"
UML-Darstellung: unausgefüllte Raute am Beziehungsende auf der Seite des Ganzen

Komposition: Spezialform der Aggregation, bei der ein Bestandteil genau zu einem Ganzen gehört und ohne dieses nicht existieren kann.
UML-Darstellung: ausgefüllte Raute am Beziehungsende auf der Seite des Ganzen

7.1.1 Klassenkarten



Abstrakte Klassen: über dem Klassennamen <<abstract>>
Interdaces: über dem Klassennamen <<interface>>
statische Methoden und Attribute durch Unterstreichen kennzeichnen

8 OOP: Klassenbeziehungen, Polymorphie, Module

1. objektorientierte Analyse
Ziel: allgemeines Systemmodell erstellen ohne Implementierungsdetails. Beschreibung, was das System machen soll und Erstellung eines OOA-Modells
2. objektorientierter Entwurf, obj. orient. Design
Planung und Beschreibung, wie das geplante System die Anforderungen realisiert
3. objektorientierte Programmierung
Umsetzung in Quelltest der verwendeten objektorientierten Programmiersprache

8.1 Vererbung

Mechanismus, der es ermöglichst, sog. Unterklassen aus einer gegebenen Klasse abzuleiten.
Die Unterklasse erbt dann von der (Ober-)Klasse die Attribute und Methoden, kann aber auf private Methoden und Attribute nicht zugreifen. Es können weitere Attribute und Methoden hinzukommen, die der Spezialisierung dienen.
Ein Objekt der Unterklasse ist auch ein Objekt der Oberklasse.
UML: Linie von Unter- zur Oberklasse mit unausgefülltem Dreieck als Pfeilspitze auf Seite der Oberklasse.
Schlüsselwort super ermöglicht den expliziten Bezug auf Attribute oder Methoden der direkten Oberklasse der Klasse, in der es verwendet wird.

8.2 Polymorphismus

Polymorphie = Fähigkeit, verschiedene Gestalt anzunehmen
Beispiel: polymorphe Methoden: haben gleichen Namen, tun aber etwas Unterschiedliches.

Überladen:
denselben Namen, aber unterschiedlicher Signatur (unterschiedlicher Rückgabetyp reich nicht)

Überschreiben:
liegt vor, wenn es zu einer Methode, die in einer Klasse deklariert ist, eine Methode in einer Unterklasse gibt, die denselben Namen, diesselbe Parameterliste und denselben Resultattyp hat.
Anwendung: Obj. der Oberklasse: Methode d. Oberkl., Obj. der Unterklasse: Methode d. Unterkl.
Auf eine überschriebene Instanzmethode der Oberklasse kann im Inneren der überschriebenen Klasse per super zugegriffen werden. Klassenmethoden können nicht überschrieben werden.

Methodenauswahl in Java:
1. Finden der anwendbaren und zugreifbaren Methoden
2. Auswahl der spezifischsten Methode

8.2.1 Polymorphe Variablen

Typ: Eigenschaft von Variablen und Ausdrücken, definiert Mindestforderung bzgl. anwendbarer Operationen, rein syntaktisch festgelegt.

Klasse: stellt Konstruktor für Objekte bereit, definiert Signatur oder Implementierung der Operationen. Objekt gehört zu der Klasse, mit deren Konstruktor es konstruiert wurde. Mit der Klassendefinition wird ein gleichnamiger Typ eingeführt.

Polymorphe Variablen in typsicheren Sprachen:
Typsicherheit:
Es dürfen nur Methoden aufgerufen werden, die schon beim statischen Typ von m verfügbar sind, da der dynamische Typ erst zur Laufzeit feststeht, der Compiler jedoch zur Übersetzungszeit garantieren muss, dass die entsprechende Methode bzw. das entsprechende Attribut tatsächlich existiert.
Eine polymorphe Variable kann im Laufe der Ausführung eines Programms Referenzen auf Objekte verschiedener Klassen haben. Sie hat einen
- statischen Typ: wird durch Angabe der Klasse bei der Deklaration angegeben und kann bei der Übersetzung überprüft werden. Ändert sich nicht.
- dynamischen Typ: wird durch die Klasse des Objekts angegeben, auf den die Variable zur Laufzeit zeigt. Zur Laufzeit bekannt, kann sich ändern.

dynamische Bindung bei Instanzmethoden: zur Laufzeit wird in Abhängigkeit von der Tatsache, ob das Objekt eine Instanz der Ober- oder einer Unterklasse ist, die jeweilige Version der Methode aufgerufen.
statische Bindung bei Klassenmethoden: es wird immer die Methode des Typs genutzt als der die Variable deklariert ist.

Verdecken von (Klassen-)Attributen und Klassenmethoden:
bedeutet, dass ein in der Unterklasse deklariertes Attribut denselben Namen aufweist wie ein entsprechendes Attribut der Oberklasse, oder dass eine in der Unterklasse deklarierte Klassenmethode diessebe Signatur aufweist, wie eine entsprechende Klassenmethode der Oberklasse.
Hierbei wird innerhalb der Unterklasse das Attribut der Oberklasse durch das namensgleiche Attribut der Unterklasse verdeckt, oder die Klassenmethode der Oberklasse durch die signaturgleiche Klassenmethode der Unterklasse verdeckt.
Regel: beim Zugriff auf (Klassen-)Attribute und Klassenmethoden ist immer der statische Typ an der Zugriffsstelle der relevante.

(Instanz-) Methoden	*(Instanz-) Attribute*
▪ ***Überladen von (Instanz-) Methoden*** – innerhalb einer Klasse oder innerhalb einer Klasse und einer ihrer Unterklassen – (min.) zwei (Instanz-) Methoden, die denselben Namen aber ansonsten unterschiedliche Signaturen aufweisen – es wird die (Instanz-) Methodenversion verwendet, die am besten zu den Argumenttypen an der Aufrufstelle passt	
▪ ***Überschreiben von (Instanz-) Methoden*** – innerhalb einer Klasse und einer ihrer Unterklassen – zwei (Instanz-) Methoden, die dieselbe Signatur **(bis auf evtl. engeren Resultattyp)** aufweisen – zur Laufzeit wird in Abhängigkeit von der Tatsache, ob es sich um Objekt der Ober- oder der Unterklasse handelt ***(dynamischer Typ an der Zugriffsstelle)***, die entsprechende (Instanz-) Methodenversion verwendet ***(dynamische Bindung)***	▪ ***Verdecken von (Instanz-) Attributen*** – innerhalb einer Klasse und einer ihrer Unterklassen – zwei (Instanz-) Attribute, die denselben Namen aufweisen – zur Laufzeit wird immer die (Instanz-) Attributversion des in der Variable deklarierten Typs ***(statischer Typ an der Zugriffsstelle)*** verwendet ***(statische Bindung)***
Klassenmethoden	***Klassenattribute***
▪ ***Überladen von Klassenmethoden:*** – innerhalb einer Klasse oder innerhalb einer Klasse und einer ihrer Unterklassen – (min.) zwei Klassenmethoden (oder Konstruktoren), die denselben Namen aber ansonsten unterschiedliche Signaturen aufweisen – es wird die Klassenmethoden-/Konstruktorversion verwendet, die am besten zu den Argumenttypen an der Aufrufstelle passt	
▪ ***Verdecken von Klassenmethoden:*** – innerhalb einer Klasse und einer ihrer Unterklassen – zwei Klassenmethoden, die dieselbe Signatur aufweisen – zur Laufzeit wird immer die Klassenmethodenversion des in der Variable deklarierten Typs ***(statischer Typ an der Zugriffsstelle)*** verwendet ***(statische Bindung)***	▪ ***Verdecken von Klassenattributen*** – innerhalb einer Klasse und einer ihrer Unterklassen – zwei Klassenattribute, die denselben Namen aufweisen – zur Laufzeit wird immer die Klassenattributversion des in der Variable deklarierten Typs ***(statischer Typ an der Zugriffsstelle)*** verwendet ***(statische Bindung)***

8.3 Schnittstellen

Mit einer Schnittstelle (Schlüsselwort interface) wird in Java definiert, welche Methoden eine diese Schnittstelle implementierende Klasse (Schlüsselwort implements) mindestens haben muss.
In Interface können Konstanten, Signaturen von Methoden deklariert werden, alles andere ist nicht zulässig. Konstantendeklarationen haben automatisch die Modifizierer static, final und public, Methodendeklarationen abstract und public. Deshalb brauchen diese nicht angegeben werden.

Auflösung potentieller Mehrfachvererbung bei Schnittstellen:
Methoden: Werden 2 gleich benannte Methoden über versch. Pfade ererbt, so wird bei identischen Signaturen eben diese Signatur übernommen, und bei unterschiedl. Signaturen beide übernommen und dann als überladene Methoden behandelt.
Konstanten: werden 2 gleich benannte Konstanten über versch. Pfade geerbt, meldet der Übersetzer Fehler, nur wenn die Konstante verwendet wird. Ein Interface kann eine Konstante deklarieren, die eine/mehrere geerbte Konstanten gleichen Namens verdeckt.

Eine abstrakte Klasse ist eine Art Oberklasse mit nur partieller Implementierung, restliche Implementierung muss in einer Unterklasse nachgeholt werden.

8.4 Pakete und Klassenbibliotheken

Paket: Zusammenstellung von als zusammengehörig betrachteten Java-Klassen und Interfaces.
Paket besitzt einen Namen, der - wie von Dateiverzeichnissen her bekannt - hierarchisch aufgebaut sein kann. Der Paketname definiert Namensraum für die im Paket vorkommenden Klassendeklarationen und sollte nur aus Kleinbuchstaben bestehen.

Um außerhalb eines Pakets eine Klasse oder Schnittstelle zu verwenden, gibt es zwei Möglichkeiten:
1. Zugriff auf Klasse in einem Paket über den qualifizierten Namen
2. Paketname mittels import bekannt machen.

9 Robustes Programmieren

9.1 Fehlerquellen

Strategien zum Umgang mit Fehlern bei der Programmentwicklung:
1. Identifizierung möglicher Fehlerquellen mit Checklisten
2. Absicherung von Quellcode gegen fehlerhafte Verwendung
3. Testen von Programmen
4. Formale Verifikation von Programmteilen
5. Toleranz gegen verbleibende Fehler

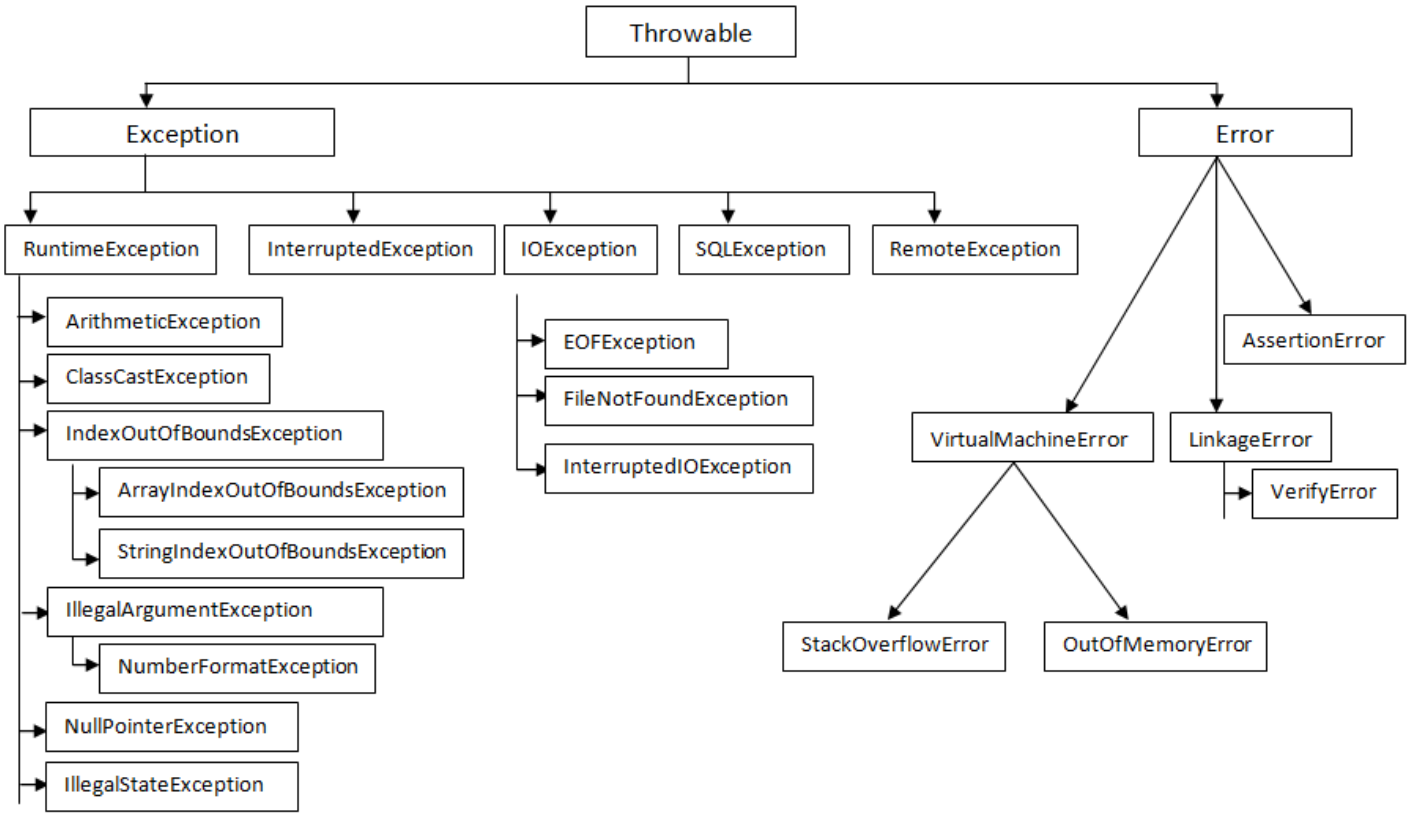
Fehlervermeidung durch gemeinsame Programmentwicklung oder gemeinsames "Walkthrough"

9.2 Fehlerbehandlung

Ausgabe von Fehlermeldungen: System.err als Ausgabe an das Konsolenfenster

Eine Ausnahme (engl. exception) ist ein Ereignis, durch das die normale Ausführungsreihenfolge unterbrochen wird.
Man sagt: eine Exception wird geworfen (throw), wenn sie ausgelöst wird, und wird gefangen, wenn sie behandelt wird.
Jeder Ausnahme wird ein Ausnahmetyp zugeordnet. Ausnahmetypen erben alle von der vordefinierten Klasse java.lang.Throwable mit genau zwei direkten Unterklassen:
java.lang.Error (= serious problem) zeigt Probleme bei der Programmausführung an, die das Programm i.d.R. nicht selbst beheben kann/sollte.
java.lang.Exception ist die Oberklasse, unter der alle Ausnahmetypen zusammengefasst werden, die das Programm selbst behandeln möchte/sollte.

Alle Exceptions ohne RuntimeExceptions sind checked exceptions, alles andere aus Throwable sind unchecked exceptions. Checked Exceptions müssen behandelt oder weitergegeben werden.



Eigene Ausnahmeklassen erstellen:
die Klasse sollte eine Unterklasse von java.lang.Exception sein.
Ein Ausnahmeobjekt sollte den Fehlerzustand aufnehmen können, also muss man entsprechende Attribute vorsehen.

Ausnahmen behandeln mit try-catch:
try-Block umfasst Code, der Ausnahmen auslösen kann.
Wird in einem try-Block eine Ausnahme geworfen, wird die Ausführung des try-Blocks sofort abgebrochen.
Dann wird der Typ der Ausnahmeobjekts mit den Typangaben der nachfolgenden catch-Klauseln verglichen (von oben nach unten).
Der erste catch-Block, dei dem der Ausnahmetyp bzgl. instanceof zur Typmenge passt, behandelt die Ausnahme und gibt dem Ausnahmeobjekt einen Namen.
Passt keiner der catch-Blöcke oder löst der catch-Block selbst eine Ausnahme aus, so wird die Ausführung abgebrochen und die letzte Ausnahme an den Aufrufer übergeben.
Falls keine Ausnahme ausgelöst wurde oder nach Bearbeitung der passenden catch-Klausel wird mit den nachfolgenden Anweisungen fortgesetzt.
Manchmal soll im Anschluss an den Code im try-Block ein Stück Code auf jeden Fall ausgeführt werden.
Solcher Code kann am Ende in einem sog. finally-Block angegeben werden.

Ausnahmen weitergeben:
Möglichkeit 1: Ausnahme in der verwendenden Methode mit try-catch abfangen
Möglichkeit 2: Ausnahme an aufrufende Methode weiterleiten:
<ModifierList> <MethodName>(<ParameterList>) throws <ExceptionClass^1>, ..., <ExceptionClass^n>

9.2.1 Stacktrace

Ein Stacktrace zeigt die Methodenschachteln auf dem Stack.
Hilfreich für die Rückverfolgung einer Ausnahme: in welcher Methode wurde die Ausnahme geworfen? von wo aus wurde diese Methode aufgerufen?
Falls Exception in Java nicht gefangen wird, bricht Programm ab und Stacktrace wird ausgegeben.
Kann auch manuell ausgegeben werden: printStackTrace()

9.3 Zusicherungen

Während der Programmentwicklung ist es möglich, Bedingungen zu formulieren, die am Beginn oder am Ende einer Methode oder während der Ausführung einer Schleife gelten müssen, damit die Methode korrekt arbeitet (sog. Vor- bzw. Nachbedingung, Schleifeninvariante).
Solche Bedingungen können als Behauptungen während der Programmentwicklung (genauer: in der Testphase) automatisiert mittels sog. Zusicherungen (engl. assertions) geprüft werden.
Bei der Ausführung beim Anwender sollten sie nicht eingesetzt und korrekter Programmablauf sollte dort anders sichergestellt werden.
assert Expression1 : Expression2
Expression1 ist boolescher Ausdruck und Expression2 ein Fehlermeldungsausdruck.

9.4 wp-Kalkül

Zur Analyse eines Algorithmus A:
Formulierung einer Nachbedingung Q:
- logischer Ausdruck über den Variablen des Algorithmus
- Anforderung an Methode, soll nach Abarbeitung von A erfüllt sein.

Bestimmung der Vorbedingung P:
- logischer Ausdruck über den Eingabewerten von A
- Welche Eigenschaften müssen die Eingabewerte haben, damit nach der Ausführung von A die Nachbedingung Q auch tatsächlich erfüllt ist?

A ist korekt, falls alle Eingabewerte, die tatsächlich auftreten können, P erfüllen.

wp = weakest precondition, schwächste Vorbedingung
Seien Algorithmus A und Nachbedingung Q gegeben. Das schwächste Prädikat P (also das, das bei möglichst vielen Eingabedaten x wahr ergibt), für das gilt P{A}Q gilt, heißt wp(A,Q):=P
Das wp-Kalkül ist ein sog. Verifikationskalkül (verwenden rein syntaktische Regelwerke zur Ableitung zu verifizierender Aussagen, Regeln sind rein mechanisch anwendbar)

Formale Definition:
E: Menge der möglichen Eingabedaten
O: Menge der möglichen Ausgabedaten
Vorbedingung P: Prädikat auf E
Nachbedingung Q: Prädikat auf E × O
A: Algorithmus, der Eingabedaten in Ausgabedaten überführt.
→ ist also eine Abbildung A: E → O
→ A(x) bezeichnet die Ausgabedaten zu den konkreten Eingabedaten x
Falls gilt $\forall x \in E : P(x) \rightarrow Q(x, A(x))$, dann schreibt man $\{P\}A\{Q\}$
Das schwächste Prädikat P mit $\{P\}A\{Q\}$ heißt wp(A, Q).

Allgemeines Vorgehen:
Rückwärtsanalyse: A schrittweise von hinten abarbeiten, also in jedem Schritt letzte Anweisung X in A verarbeiten. Dazu Auswirkungen von X auf Nachbedingung bestimmen (was muss vor X gelten, damit nach X Nachbedingung erfüllt ist?). Ergebnis wird neue Nachbedingung für restliche Anweisungen in A.

Zuweisungen:	in Q werden alle Vorkommen durch neuen Wert ersetzt.
In-/Dekrement-Operatoren:	nach/vor der eigentlichen Substitution behandeln.
Fallunterscheidungen:	Bedingung wahr: b erfüllt und X führt zur Nachbedingung
	Bedingung falsch: b nicht erfüllt und Y führt zur Nachbedingung
Datentypen:	Vereinfachung in Datentyp der Variable, explizit hinschreiben

9.4.1 Korrektheit von Schleifen

2 Werkzeuge zum Korrektheitsbeweis von Schleifen:
1. Schleifeninvariante: muss vor/nach jedem Durchlauf wahr sein, beschreibt Verhalten der Schleife
2. Schleifenvariante: Funktion über Programm-Variablen, Beweis der Terminierung

Schleifeninvariante:
Sei eine Schleife der Form while (b) {A;} gegeben. Ein Prädikat I ist eine gültige Schleifeninvariante, falls gilt $\{I \wedge b\} \rightarrow wp(A, I)$
Falls vor der Ausführung des Rumpfes die Invariante I und die Schleifen-Bedingung b erfüllt sind (d.h. die Schleife wird auch tatsächlich ausgeführt), dann muss nach Ausführung des Rumpfes zumindest die Invariante erfüllt sein.
Eine geeignete Schleifeninvariante I hat zusätzlich folgende Eigenschaften:
- das Prädikat I ist vor der Schleife erfüllt
- die Nachbedingung Q lässt sich nach der Schleife aus I implizieren

Schleifenvariante (auch: Terminierungsfunktion)
Schleife muss nach endlich vielen Durchläufen abbrechen
1. V ist eine geeignete Funktion auf den in der Schleife verfügbaren Variablen
2. V ist ganzzahlig
3. V ist streng monoton fallend
4. V ist nach unten beschränkt ($\{I \wedge b \wedge V\}A\{c \leq V < z\}$ mit $c, z \in \mathbb{Z}$)

10 Grundlegende Datentypen

10.1 Spezifikation von Datentypen

Abstrakter Datentyp (ADT):
Festlegung der auf den Typ anwendbaren Operationen (Schnittstelle)
Festlegung der Wirkung der Operationen

Man legt fest:
- adt: Namen des ADTs
- sorts: Liste verwendeter Datentypen
- ops: Schnittstellen der Operationen: Name, Parameter-Typen und Ergebnis-Typ
- axs: Axiome zur Beschreibung der Semantik: so allgemein wie möglich, so ausführlich wie nötig

Dies geschieht in der Signatur (Bsp.):

adt	IntSet			
sorts	IntSet, Int, Boolean			
ops	create:			$\rightarrow$ IntSet
	insert:	IntSet $\times$ Int		$\rightarrow$ IntSet
	delete:	IntSet $\times$ Int		$\rightarrow$ IntSet
	contains:	IntSet $\times$ Int		$\rightarrow$ Boolean
	isEmpty:	IntSet		$\rightarrow$ Boolean
axs	isEmpty(create)	= true		
	isEmpty(insert(x,i))	=false		
	insert(insert (x,i),i)	= insert(x,i)		
	contains(create, i)	= false		
	contains(insert(x,j),i)	= true	falls i=j	
		contains (x,i)	sonst	
end	IntSet			

Vorteile:
Man kann Datentyp genau soweit festlegen, wie gewünscht, ohne Implementierungsdetails vorwegzunehmen. Die Sprache, in der Axiome beschrieben werden, folgt formalen Regeln. Daher sind Entwurfswerkzeuge konstruierbar, die Spezifikation prüfen oder Prototyp erzeugen können.

Nachteile:
Bei komplexen Anwendungen wird die Zahl der Gesetze sehr groß. Es ist nicht immer leicht, anhand der Gesetze die intuitive Bedeutung eines Datentyps zu erkennen. Insbesondere ist es schwierig zu prüfen, ob die Menge der Gesetze vollständig ist

Umsetzung in Java-Code:
ops: Schnittstellen der Java-Methoden
axs: Umsetzung der Java-Methoden

Konstruktoren:	mit create und insert lassen sich alle mögl. Objekte erzeugen. Zwingend erforderlich.
Hilfskonstrukt.:	erzeugen auch Datenobjekte des Typs, aber nicht zwingend erforderlich.
Normalform:	konstruiert durch minimale Zahl von Konstruktoraufrufen.

10.2 Generische/Parametrisierte Klassen

Generische bzw. parametrisierte Klassen bieten sich an, wenn man feststellt, dass man mehrere Klassen für verschiedene Typen, aber mit ansonsten identischem Code benötigt.

Vorteile: Code muss nur einmal geschrieben werden. Platzhalter kann durch einen beliebigen Typ ersetzt werden, der so verwendet werden kann, wie der Platzhalter im Code der generischen Klasse.

Durch Angabe des Typplatzhalters T in spitzen Klammern wird Klasse zur generischen bzw. parametrisierten Klasse. Mehrere Typparameter werden durch Kommata getrennt angegeben. Typparameter können innerhalb der Klasse als Typen von Variablen oder Rückgabewerten oder als Parameter von Methoden genutzt werden.

Mögliche Typen für Typparameter können eingeschränkt werden: nur Klassen die von bestimmter Klasse erben oder von bestimmtem Interface implementieren.
Einschränkung mittels extends

Typparameter dürfen nicht in statischen Elementen verwendet werden, da verschiedene Instanzen einer Klasse mit unterschiedlichen generischen Typen dennoch diesselben statischen Elemente verwenden.

10.3 Elementare Listen

Listen enthalten endliche Folgen von Objekten eines gegebenen Grundtyps in einer definierten Ordnung. Jedes Element hat Vorgänger und Nachfolger (außer 1. und letztes Element). Listen dürfen Duplikate enthalten. Man unterscheidet unsortierte und sortierte Listen.
Listeneintrags-Objekte haben folgende Struktur:
Verweis auf den Listeneintrag + Verweis auf den Nachfolger des Listeneintrags
Liste besteht aus verketteten Listeneintrags-Objekten, die jeweils vorne angefügt werden

Stapel/Keller (Stacks)
Ein Stapel/Keller (engl. Stack) ist ein Spezialfall der elementaren Liste.
Stapelprinzip: man kann nur oben etwas drauflegen oder herunternehmen.
Nur oberstes Element ist sichtbar.
Zuletzt eingefügtes Element wird als erstes entnommen: Last-In-First-Out
(Warte-) Schlangen (Queues)
Schlangen sind spezielle Listen, bei denen Elemente nur an einem Ende (vorne) entnommen und nur am anderen Ende (hinten) angehängt werden.
Schlangen sind sog. First-In-First-Out Datenstrukturen.

11 Verkettete Listen, dynamische Arrays, Mengen

Einfach verkettete Liste: jedes Element kennt seinen direkten Nachfolger.
Zweifach verkettete Liste: jedes Element kennt zusätzlich seinen direkten Vorgänger.

11.1 Innere, geschachtelte und lokale Klassen

1. innere Klasse:
nicht-statische Klasse innerhalb einer anderen Klasse
Methoden der inneren Klasse können auf alle Elemente der äußeren Klasse zugreifen.
Ein Objekt der inneren Klasse ist immer abhängig von einem Objekt der äußeren Klasse, d.h. es muss ein Objekt der äußeren Klasse existieren, um eines der inneren instanziieren zu können. Deshalb können innere Klasse keine statischen Elemente besitzen.

2. geschachtelte Klasse:
statische Klasse innerhalb einer anderen Klasse
wie statische Methoden können geschachtelte Klassen auf statische Elemente der sie enthaltenden Klasse zugreifen, nicht aber auf deren Instanzvariablen.
Objekte sind nicht von einem Objekt der äußeren Klasse abhängig und können statische Elemente besitzen.

3. lokale Klasse:
nicht-statische Klasse innerhalb einer Methode

11.2 Einfach verkettete Listen

Liste besteht aus verketteten Listeneintrags-Objekten mit sog. Wächter-Objekt. next der Wächterelements zeigt auf den vordersten Listeneintrag. Letzter Listeneintrag verweist auf Wächter → Zyklus

Durchlaufen einer verketteten Liste:

1. per for-Schleife

```
Medium elem;
for (int i = 0; i < medList.size(); i++) {
    elem = medList.get(i);
    elem.print();
}
```

2. per for-each-Schleife

```
for (BibMitglied bm : bmList) {
    bm.print();
}
```

3. Listen-Iterator:

```
Iterator<Medium> iter = medList.iterator();
Medium elem;
while (iter.hasNext()) {
    elem = iter.next();
    elem.print(); // mache etwas mit dem Listenelement
}
```

11.3 Dynamische Arrays

	einfach verkettete Listen	Standard-Arrays
Speicherbedarf:	ändert sich dynamisch mit der Anzahl der enthaltenen Objekte immer nur so viel wie nötig Speicherbedarf für Zeiger	wird einmal festgelegt und ist dann nicht veränderbar ggfs. Speicherverschwendung bei nicht voll belegtem Array
Element suchen:	lineare Suche	binäre Suche (wenn sortiert)
Zugriff i-tes Element:	Liste von vorne bis zur betreffenden Stelle durchlaufen	direkter Zugriff möglich

Kombination der Vorteile ergibt sog. dynamisches Array

11.4 Mengen

Ziel:
Mengen so darstellen, dass die klassischen Operationen Hinzufügen und Entnahme von Elementen, Vereinigung, Durchschnitt und Differenz effizient ausgeführt werden können.
Mengen enthalten keine Duplikate von Elementen.
Elemente haben keine feste Reihenfolge.

1. Implementierung mit einfach verketteter Liste ohne Sortierung der Elemente
Mengenelemente werden in nicht-sortierter, einfach verketteter Liste gehalten.
Neue Elemente werden vorne an der zugrundeliegenden Liste angefügt, sofern sie nicht bereits enthalten sind (Test erfordert Durchlaufen der Liste per linearer Suche).

2. Implementierung mit einfach verketteter Liste mit Sortierung der Elemente
wenn die Elemente in der Menge aufsteigend sortiert sind, dann muss man nicht die ganze verkettete Liste durchlaufen, um nach Duplikaten zu suchen und die Mengenoperationen sind schneller, weil ein Reißverschlussverfahren verwendet werden kann.

3. Implementierung mit dynamischem Array mit Sortierung der Elemente
Bei der Implementierung einer Menge mit Hilfe einer einfach verketteten Liste profitierten allein die Mengenoperationen von der Sortierung.
Vom Suchaufwand her ist eine Implementierung mithilfe eines Arrays eine günstigere Lösung, da direkt auf die einzelnen Elemente zugegriffen werden kann.
Die Speicherung als Array spart Platz, weil keine Verzeigerung nötig ist. Sie kostet Laufzeit, weil beim Änderung der Größe Kopierarbeit anfällt.

4. Implementierung als Bitvektor ohne Sortierung der Elemente

11.5 Streutabellen

Idee: aus dem Wert eines zu speichernden Mengenelements wird seine Adresse im Speicher berechnet.
Speicher zur Aufnahme der Mengenelemente ist Streutabelle der Größe m, wobei m i.d.R. sehr viel kleiner als der Wertebereich ist. Die zu speichernden Mengenelemente haben einen eindeutigen Schlüssel.
Erforderlich ist die Hash-Funktion $h : G \to S$. h ist nicht injektiv, möglichst surjektiv, sollte Schlüssel gleichmäßig auf den Bereich der Tabellenindizes verteilen und sollte effizient zu berechnen sein. h bildet ein Datenelement auf einen Hashwert ab, benötigt dazu einen Schlüssel, der ein Element eindeutig identifiziert. Der errechnete Hashwert wird als Index in die Tabelle verwendet, das Element wird in entsprechendem Bucket der Tabelle gespeichert.
Belegungsfaktor BF = tatsächlich eingetragene Schlüssel / Indexmenge
BF $< 50\ \% \to$ Verschwendung von Speicherplatz, BF $> 80\% \to$ höhere Kollisionswahrscheinlichkeit.

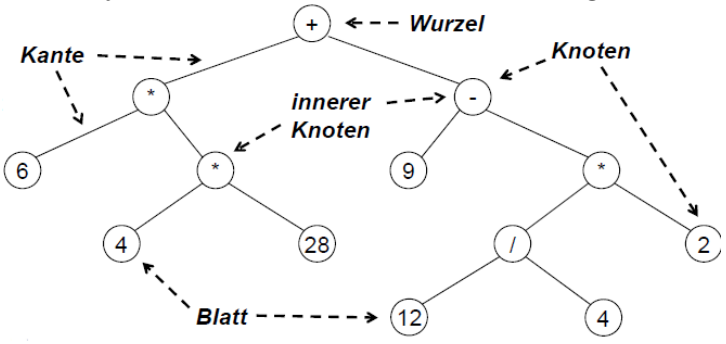
Offenes Hashing: Behälter kann beliebig viele kollidierende Schlüssel aufnehmen (verkettete Liste).
Geschlossenes Hashing: Behälter kann nur kleine konstante Anzahl von kollidierenden Schlüsseln aufnehmen.

12 Bäume

12.1 Allgemeine Bäume

Bäume erlauben es, hierarchische Beziehungen zwischen Objekten darzustellen.

Liste: jedes Element hat einen Nachfolger
Baum: jedes Element hat mehrere Nachfolger



Höhe eines Knotens: Anzahl der Kanten des längsten Pfades zu einem Blatt.
Höhe des Baumes: Höhe der Wurzel
Bäume enthalten keine Zyklen.
Verzweigungsgrad: Zahl der mögl. Nachfolger pro Element
X ist Vater/Mutter von Y, fall es eine Kante von X nach Y gibt.
X ist Kind von Y, falls es eine Kante von Y nach X gibt.
X ist sibling eines Knoten Y, falls X und Y denselben direkten Vorgänger haben.
X ist indirekter Vorgänger/Nachfolger von Y, falls es einen Pfad von X nach Y/Y nach X gibt.
X ist ein Blatt, falls X keine Kinder hat.
X ist ein innerer Knoten, falls X mind. ein Kind hat.

Definition (*Binärbäume*)

Jeder Knoten hat nur 2 Nachfolger
Der leere Baum ist ein Binärbaum.
Wenn x ein Knoten ist und A und B Binärbäume sind, dann ist auch (AxB) ein Binärbaum.
x heißt Wurzel, A linker Teilbaum, B rechter Teilbaum.
Die Wurzeln von A und B heißen Kinder/Söhne von x, x ist ihr Elternknoten/Vaterknoten.
Ein Knoten, dessen beide Kinder leere Bäume sind, heißt Blatt.

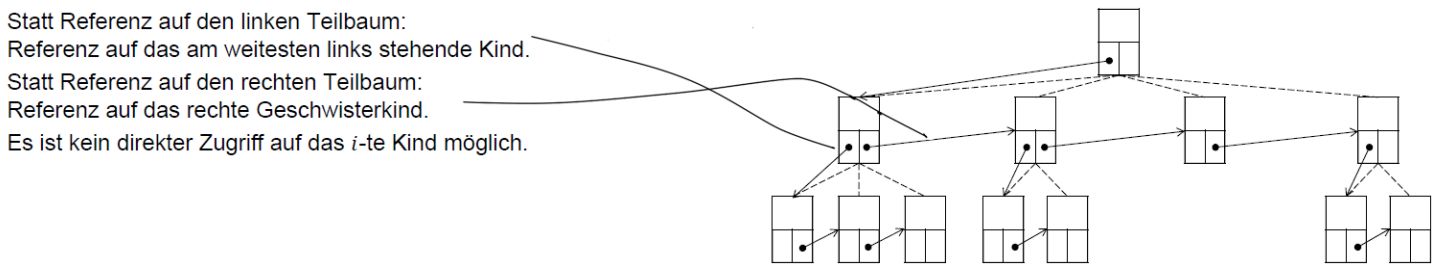
adt	BinTree		
sorts	BinTree, E, Boolean		
ops			
	create:		→ Bintree
	maketree:	BinTree × E × BinTree	→ BinTree
	value:	BinTree	→ E
	left, right:	BinTree	→ BinTree
	leaf:	E	→ BinTree
	isEmpty:	BinTree	→ Boolean
axs			
	left(maketree(l,e,r))=l		
	right(maketree(l,e,r))=r		
	value(maketree(l,e,r))=e		
	leaf(v)= maketree(create,v,create)		
	isEmpty(create)=true		
	isEmpty(maketree(l,e,r)=false		
	...		
end	BinTree		

Definition (*Allgemeine Bäume*)

Anzahl der Kinder eines Knotens ist beliebig.
Keine festen Positionen für Kinder.
Rekursive Definition:
1. Der leere Baum ist ein (allgemeiner) Baum.
2. Wenn x ein Knoten ist und A1, ... , An Bäume sind, dann ist auch das n+1-Tupel (x, A1, ... , An) ein Baum.

Jeder Knoten speichert eine ArrayList von Referenzen auf seine Kinder

andere Interpretation der Referenzen:



12.2 Binäre Suchbäume

Bäume kann man sich als strukturelles Abbild des Teile-und- Herrsche-Prinzips vorstellen, wenn man beim Zugriff jeweils pro Knoten nur ein Kind weiterverfolgt.
Eine solche Datenstruktur in Form eines Baums wird Suchbaum genannt. Ein binärer Suchbaum hat max. zwei Kinder je Knoten.

Traversierung, Besuchsreihenfolgen:
Breitensuche: Geschwister vor Kindern besuchen, noch zu besuchende Kntoen in Queue
Tiefensuche: Kinder vor Geschwistern besuchen, noch zu besuchende Knoten auf Stack

Inorder:	Preorder:	Postorder:
1. linker Unterbaum	1. aktueller Knoten	1. linker Unterbaum
2. aktueller Knoten	2. linker Unterbaum	2. rechter Unterbaum
3. rechter Unterbaum	3. rechter Unterbaum	3. aktueller Knoten

Klassifikation von Suchbäumen:
1. Anzahl m der Kinder pro Knoten (Knotengrad):
- Binärbaum: Anzahl $0 \leq m \leq 2$
- allgemeiner Baum: Anzahl beliebig
2. Speicherort der Nutzdaten:
- natürlicher Baum: Nutzdaten werden bei jedem Knoten gespeichert.
- Blattbaum/hohler Baum: Nutzdaten werden nur in Blattknoten gespeichert
3. Ausgewogenheit/Ausgeglichenheit/Balanciertheit:
- balancierter Baum: die Höhen aller Unterbäume unterscheiden sich höchstens um Δn
- vollständig ausgewogen: ANzahl der Knoten in Unterbäumen unterscheidet sich höchstens um 1
- unbalancierter Baum: keine derartigen Einschränkungen

Suche nach einem Element:
- fange bei der Wurzel an
- solange gesuchter Wert nicht gefunden:
- falls gesuchter Wert kleiner als aktueller Wert: steige nach links ab
- falls gesuchter Wert größer als aktueller Wert: steige nach rechts ab
- falls kein entsprechendes Kind vorhanden: Wert nicht enthalten

Einfügen eines Wertes:
- Suche nach Einfügeposition mittels Suchalgorithmus
- Einhängen des neuen Knotens an dieser Position

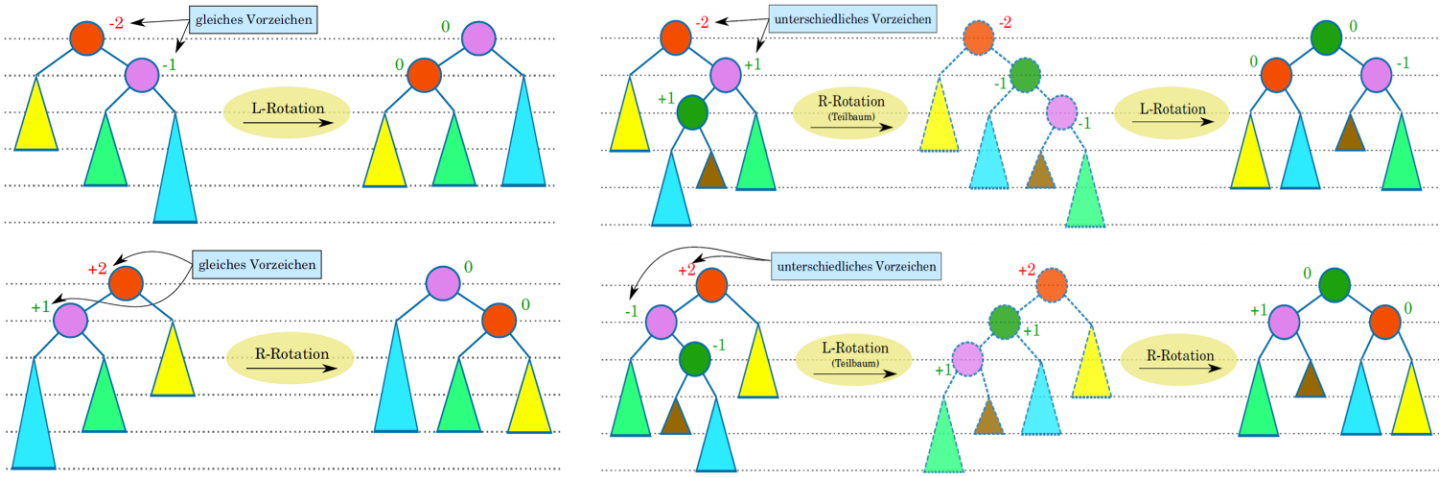
Löschen eines Wertes:
- Suche nach Knoten mittels Such-Algorithmus
- falls zu löschender Knoten Blatt-Knoten ist: aus Baum entfernen
- falls zu löschender Knoten kein Blatt-Knoten ist: Knoten ersetzen durch...
- größten Knoten im linken Teilbaum oder kleinsten Knoten im rechten Teilbaum
- dadurch ggf. weitere Ersetzungen in den Teilbäumen notwendig!

12.3 AVL-Bäume

Ein AVL-Baum ist ein binärer Suchbaum, in dem sich die Höhen seiner zwei Teilbäume höchstens um 1 unterscheiden. Wird dies durch Aktualisierungen verletzt, so muss sie durch eine Rebalancieroperation wieder hergestellt werden.

Einfügen und Löschen wie bei Binärbaum, danach Rebalancierphase:
In jedem Knoten wird dessen Balance geprüft: falls Knoten aus der Balance geraten, wird das durch eine Rebalancieroperation korrigiert.
Balancefaktor eines Knoten: Differenz zw. Höhe des linken und Höhe des rechten Teilbaums
Nach einer Einfügung genügt eine einzige Rotation oder Doppelrotation, um die Strukturinvariante des AVL-Baums (AVL-Bedingung) wiederherzustellen (und dabei die Suchbaumeigenschaft zu erhalten).

Nachteile von AVL-Bäumen: zusätzlicher Platzbedarf in den Knoten zur Speicherung der Höhe oder der Balancefaktoren, komplizierte Implementierung



12.4 Halden

Heap ist ein partiell geordneter Binärbaum.
Max-Heap: Wurzel jedes Teilbaums ist größer als andere Knoten des Teilbaums
Min-Heap: Wurzel jedes Teilbaums ist kleiner als andere Knoten des Teilbaums
Verwendung als Prioritätswarteschlange
effiziente Speicherung in Array möglich

Einfügen eines neuen Elements
- Element an die nächste freie Position in der untersten Ebene einfügen
- falls Ebene voll: Element wird erster Knoten einer neuen Ebene
- solange Halden-Eigenschaft in einem Teilbaum verletzt ist:
  Element entsprechend der Halden-Eigenschaft nach oben wandern lassen

Löschen eines Elements
- zu löschendes Element mit dem letzten Element ersetzen
- solange Halden-Eigenschaft in einem Teilbaum verletzt ist:
- Element entsprechend der Halden-Eigenschaft nach unten wandern lassen
- Min-Heap: Tauschen mit kleinerem Kind
- Max-Heap: Tauschen mit größerem Kind

13 Sortieralgorithmen

internes Sortierverfahren: Alle Datensätze können gleichzeitig im Hauptspeicher gehalten werden. Direkter Zugriff auf alle Elemente ist möglich und erforderlich.
externes Sortierverfahren: Sortieren von Massendaten, die auf externen Speichermedien gehalten werden. Zugriff ist auf einen Ausschnitt der Datenelemente beschränkt
Sortieren durch Auswählen, Einfügen, Fachverteilen oder mittels Divide-and-Conquer-Verfahren
Laufzeit: Vergleichsbasierte Sortierverfahren haben mindestens Laufzeit von O(n log n).
in-situ: Sortierverfahren verwendet nur ein Array
stabiles Sortierverfahren: gleiche Werte ändern ihre relative Reihenfolge nicht.

vergleichsbasierte Sortierverfahren

Bezeichnung	Best Case	Average Case	Worst Case	stabil	in situ
Einfache Sortierverfahren					
Sortieren durch Auswählen (SelectionSort)	$O(n^2)$	$O(n^2)$	$O(n^2)$	(leicht mgl.)	(wenn mit Array)
Sortieren durch Einfügen (InsertionSort)	O(n)	$O(n^2)$	$O(n^2)$	X	(wenn mit Array)
Blasensortierung (BubbleSort)	O(n)	$O(n^2)$	$O(n^2)$	X	X
Verfeinertes Auswählen					
Haldensortieren (HeapSort)	O(n log n)	O(n log n)	**O(n log n)**		X
Teile-&-Herrsche/Divide-&-Conquer-Verf.					
Sortieren durch Verschmelzen (MergeSort)	O(n log n)	O(n log n)	**O(n log n)**	X	(in der Regel nicht)
Sortieren durch Zerlegen (QuickSort)	O(n log n)	O(n log n)	$O(n^2)$		X

nicht-vergleichsbasierte Sortierverfahren

Sortieren durch Fachverteilen	Zeit	stabil	in situ
BucketSort	O(n + m)	X	
RadixSort	O(n · k)	X	

SelectionSort
Lösche nacheinander die Maxima aus einer Liste und füge sie vorne an eine anfangs leere Ergebnisliste an.
InsertionSort
Starte mit erstem Element. Nimm jeweils das nächste Element und füge dieses an der richtigen Stelle in die Reihe ein.
BubbleSort
Das Array wird immer wieder durchlaufen und dabei werden benachbarte Elemente in die richtige Reihenfolge gebracht. Größere Elemente überholen so die kleineren und drängen an das Ende der Folge.
HeapSort
Die n zu sortierenden Elemente werden in eine max-Halde (min-Halde) eingefügt: Aufwand O(n log n). Dann wird n-mal das Maximum (Minimum) aus der Halde entnommen: Aufwand O(n log n).
MergeSort
falls zu sortierende Liste mehr als ein Element beinhaltet:
- teile Liste in zwei kleinere Listen auf
- verfahre rekursiv mit den beiden Teillisten
- Listen mit maximal einem Element sind trivialerweise sortiert
- verschmelze jeweils zwei sortierte Listen (Sortierung in verschmolzener Liste beibehalten)

QuickSort
falls zu sortierende Liste mehr als ein Element beinhaltet:
- wähle ein Pivot-Element (zufälliges oder Median aus x Elementen wählen)
- schiebe alle kleineren Elemente vor das Pivot-Element
- schiebe alle größeren Elemente hinter das Pivot-Element
- verfahre rekursiv mit den beiden Teillisten

BucketSort
Zu sortierende Werte sind ganze Zahlen zw. 0 und m-1. Man nutzt eine Menge von Behältern B_0 bis B_{m-1}, in die man alle Elemente passend einfügt. Am Ende schreibt man die Elemente in die Ergebnisfolge.
RadixSort
Voraussetzung: Elemente sind Zeichenfolgen über endlichem Alphabet, auf dessen Zeichen eine Totalordnung definiert ist. Für jedes Zeichen steht ein Fach zur Verfügung.
Einzelne Stellen aller Elemente von hinten nach vorne abarbeiten.
Partitionieren: Elemente abhängig von der aktuellen Stelle in das passende Fach legen.
Einsammeln: Elemente wieder aus den Fächern nehmen vom 'kleinsten' Fach zum 'größten' Fach.

14 Graphen und Graphalgorithmen

14.1 Grundbegriffe

Ein Graph ist ein Paar $G = (V, E)$, wobei gilt:
1. V ist eine endliche nichtleere Menge (Knoten)
2. E ist eine zweistellige Relation
Ein Pfad von v nach w ist eine endliche Folge von Knoten. Die Länge eines Pfades ist die Anzahl der Kanten im Pfad. Ein Pfad heißt einfach, wenn alle Knoten im Pfad paarweise verschieden sind.

In einem gerichteten Graphen heißt ein Pfad der Länge von v nach v ein Zyklus. Ein Zyklus von v nach v heißt minimaler Zyklus, wenn außer v kein anderer Knoten mehr als einmal vorkommt. Bei ungerichteten Graphen muss ein Zyklus mindestens drei Knoten enthalten.

Die Anzahl der direkten Vorgänger eines Knotens heißt Eingangsgrad des Knotens. Die Anzahl der direkten Nachfolger eines Knotens heißt Ausgangsgrad des Knotens. Bei ungerichteten Graphen spricht man vom Grad des Knotens.
Ein Knoten mit Grad 0/* heißt Quelle. Ein Knoten mit Grad */0 heißt Senke.

Ein gerichteter Graph heißt stark zusammenhängend (oder: stark verbunden), wenn es für alle v, w einen Pfad von v nach w gibt. Bei ungerichteten Graphen lässt man stark weg. Anschaulich: Jeder Knoten ist von jedem anderen Knoten aus erreichbar. Ein gerichteter Graph heißt schwach zusammenhängend, wenn der zugehörige ungerichtete Graph (der durch Hinzunahme aller Rückwärtskanten entsteht) zusammenhängend ist.

Sind G ein Graph und S ein zyklenfreier Teilgraph von G, der alle Knoten von G enthält, und sind G und S beide zusammenhängend, dann heißt S Spannbaum (auch: aufspannender Baum) von G. S ist ein minimaler Spannbaum, falls S ein Spannbaum von G ist, und die Summe der Kantengewichte in S kleiner oder gleich der aller anderen Spannbäume S' von G ist.
Ein Knoten v eines gerichteten azyklischen Graphen (DAG) heißt Wurzel, falls es keine auf ihn gerichteten Kanten gibt.

Ein Graph wird zu einem bewerteten/gewichteten Graphen, indem man eine Gewichtsfunktion ergänzt, die jeder Kante ein positives Gewicht zuordnet. Die Kosten eines Pfades definiert man in gewichteten Graphen als die Summe der Gewichte seiner Kanten.

14.2 Darstellungen von Graphen

Adjazenzmatrix: boolesche $n \times n$ Matrix mit $A_{ij} = true$, falls $(v_i, v_j) \in E$ und $A_{ij} = false$ sonst.
Bei bewerteten/gewichteten Graphen speichert man statt true/false die Kantengewichte in der Matrix. Fehlende Kanten stellt man mit einem Spezialwert für Unendlich dar.
Diese Repräsentation wählen, wenn Tests auf das Vorhandensein von Kanten häufig vorkommen.
Adjazenzliste: Man verwaltet für jeden der n Knoten eine Liste seiner Nachfolger/Nachbarknoten

14.3 Graphdurchlauf

Tiefensuche: entspricht einem Preorder-Durchlauf von G, der jeweils in einem bereits besuchten Knoten von G abgebrochen wird.
Die Tiefensuche durchläuft die Knoten eines Graphen in einer bestimmten Reihenfolge. Wenn jedem Knoten die Position in dieser Reihenfolge zugeordnet wird, ist das eine DFS-Nummerierung.
Wenn v.dfsNr < w.dfsNr, dann ist v Vorfahre von w in T.
Breitensuche: besucht die Knoten von G ebenenweise, also zuerst die Knoten, die über einen Pfad der Länge 1 von der Wurzel aus erreichbar sind, dann über einen Pfad der Länge 2 usw. In einem bereits besuchten Knoten von G wird abgebrochen.

14.4 Kürzeste Wege in Graphen

Dijkstra:
für alle Knoten gespeichert: Distanz, Vorgänger und 'besucht'-Markierung
- Initialisierung:
- Startknoten mit Distanz 0, alle anderen Knoten mit Distanz ∞
- markiere alle Knoten als unbesucht
- solange es noch unbesuchte Knoten im Graphen gibt:
  - wähle den Knoten v mit der geringsten Distanz aus (Min-Heap)
  - markiere diesen Knoten v als besucht
  - für alle unbesuchten Nachbarn u des aktuellen Knoten v
    - berechne die Länge des Pfades über v nach u:
      Gewicht = Pfad vom Startknoten nach v + Gewicht der Kante (v, u)
    - falls das berechnete Gewicht kleiner als die gespeicherte Distanz ist, dann
      a) aktualisiere die Distanz auf den neuen Wert
      b) speichere Knoten v als Vorgänger von u

Floyd:
es existiert eine direkte Kante $v \to w$ genau dann, wenn ein Pfad von v nach w existiert, der nur Knoten aus der Ursprungsmenge als Zwischenknoten verwendet. Der kürzeste derartie Pfad hat Kosten α.

14.5 Minimaler Spannbaum

Algorithmus von Prim:
Wähle einen beliebigen Knoten als Startgraph T.
Solange T noch nicht alle Knoten enthält, führe aus:
Wähle eine Kante e mit minimalen Kosten aus, die einen noch nicht in T enthaltenen Knoten v mit T verbindet. Füge e und v dem Graphen T hinzu.

Algorithmus von Kruskal:
gieriger Algorithmus, in jedem Schritt die kleinste Kante wählen, die 'nichts kaputt macht'
- Eingabe: Graph G, dessen minimaler Spannbaum S bestimmt werden soll
- S enthält zu Anfang nur die Knoten von G und keine Kanten
- sortiere die Kanten von G aufsteigend nach ihrem Gewicht
- solange S noch nicht zusammenhängend ist, betrachte die nächste Kante
- wenn diese zwei getrennte Komponenten von S verbindet, dann zu S hinzufügen, andernfalls nicht

15 Geometrische Algorithmen

Punkt:	n-Tupel im n-dimensionalen Raum (n Koordinaten)
Gerade:	Linie durch zwei beliebige Punkte
Strecke:	Linie durch ihre beiden Endpunkte
Streckenzug:	Folge von Strecken (Kanten), bei der der Endpunkt (Knoten) einer Strecke der Anfangspunkt der nächsten Strecke ist
Polygon:	der Anfangspunkt der ersten Strecke ist Endpunkt der letzten
innen:	Polygon, dessen Kanten sich nicht kreuzen, umrahmt eine Region, deren Punkte innen sind
konvex:	Polygon ist konvex, wenn jede Strecke zwischen zwei seiner Punkte komplett im Inneren des Polygons liegt

Punkt-in-Polygon-Problem
Wähle Punkt 'unendlich' weit außen und verbinde ihn mit dem Punkt P. Gehe auf der Verbindung in Richtung P.
Wenn eine Polygonkante gekreuzt wird und wir außen (a) sind, gehen wir nach innen (i). Wenn eine Polygonkante gekreuzt wird und wir innen (i) sind, gehen wir nach außen (a).

Konstruktion von Polygonen
Punkte zu Polygon verbinden
Radius einer Kreisscheibe über alle Punkte rotieren. Verbinde den letzten zum Polygon hinzugefügten Punkt mir dem nächsten vom Radius überstrichenen Punkt. Als Mittelpunkt einen Randpunkt der Punktmenge wählen.

Konvexe Hülle
Die konvexe Hülle einer Punktmenge ist das kleinste konv. Polygon, das alle Punkte in seinem Innern hat. Beginne mit einem Polygon aus beliebigen drei Punkten der gegebenen Punktmenge. Wähle beliebigen noch nicht bearbeiteten Punkt P (möglichst weit außen). Liegt er außerhalb des bisher erzeugten Polygons, dann modifiziere die bisher gefundene Hülle, sodass sie den neuen Punkt enthält.
Einpackalgorithmus: Beginne mit einem äußersten Punkt (z.B. äußerst links) und Schnur nach unten. Verbinde immer zu dem Punkt, den bei Drehung gegen den Uhrzeigersinn die Einwickelschnur als nächstes erreicht (stützende Kante).

Ballung und nächstes Paar
Problem: Punktmenge soll in homogene Klassen eingeteilt werden.
Zweck: ohne a-priori-Wissen bestimmen, was zusammengehört bzw. was sich ähnlich ist.
Scharfe Klassifikation: Punkte werden zu Klassen zusammengefasst, sodass jeder Punkt genau zu einer Klasse gehört.
Der Heterogenitätsindex einer Klasse gibt an, wie gut die Punkte einer Klasse zusammenpassen.
Der Güteindex einer Klassifikation bewertet eine Klassifikation.
Hierarchische Verfahren:
- zusammenfassend/agglomerativ: zu Beginn enthält jede Klasse genau einen Punkt.
  Füge in jedem Schritt die zwei am wenigsten entfernten Klassen zusammen.
- zerteilend/divisiv: beginne mit einer Klasse, die alle Punkte enthält.
  Teile in jedem Schritt eine Klasse in zwei Unterklassen.

Für jede pro Schritt entstehende Klassifikation wird ein Güteindex berechnet. Wähle die Klassifikation, bei der sich im nächsten Schritt der Güteindex nur noch gering verbessern würde.

Finde nächstes Paar: Diejenigen zwei Punkte einer Punktmenge mit geringstem gegenseitigen Abstand.
Einfacher Algorithmus: Berechne alle Distanzen, wähle das Paar mit der kleinsten Distanz.