Algorithmen und Datenstrukturen

Grundlagen

Definition (*Algorithmus*)

Folge einfacher Anweisungen mit den Eigenschaften:
Endlichkeit, Terminierung, eindeutige Reihenfolge/Wirkung
Algorithmenentwurf:
Top-Down: Gesamtproblem in Teilprobleme zerlegen
Bottom-Up: Gesamtproblem besteht aus Teilproblemen

Variablen

Speichereinheit von Datenwerten. Hat Namen und Datentyp, der angibt, welcher Typ von Werten in ihr gespeichert werden kann. Zur Vorstellung kann man ein Behältermodell nehmen.

Definition (*Variablendeklaration*)

< Datentyp > < Name > = < Wert > ;
Semikolon auch bei Deklaration nicht vergessen
Beispiel: int zahl = 1337;

Definition (*Bezeichner (Name einer Variable)*)

Werden kleingeschrieben, ohne Sonderzeichen/Leerzeichen
Die Namen sollten selbsterklärend sein und Lesbar
Die Namen dürfen keine Schlüsselbegriffe wie "void" sein

Definition (*Verarbeitung*)

Wertzuweisung: < Name > = < Name >;
Eine 0 vor den anderen Zahlen definiert diese als Oktalzahlen
Konstanten:
→ statische: Überall verwendbar
→ lokale: nur in bestimmten Bereich verwendbar
→ Deklaration mit < final > vor dem Datentyp
→ Konventionell mit Capslock deklariert

Datentypen

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	$=< 2^{63} - 1$
Fließkommazahlen:	float (32 bit), double (64 bit)			
	⇒ darstellung gemäß IEEE			
	⇒ für genaue Rechnungen ungeeignet!			
Zeichen:	char (zwischen ' ')			
boolescher Wert:	boolean (wahr oder falsch)			

Umgang mit Boolean:

x	!x	x	y	x&&y	x||y
true	false	false	false	false	false
false	true	false	true	false	true
		true	false	false	true
		true	true	true	true

Definition (*Array*)

zusammengesetzter Datentyp mit endlicher Folge von Werten desselben Datentyps
Deklaration: int[ ] a=new int[Länge des Arrays]{Elemente}
Beispiel: int[ ] a=new int[10]
Deklaration einzelne Werte: int[ ] a={Wert1 , Wert2 , ...}
Mehrdimensionale Arrays: int[ ] a=new int[10][10]
Länge von Arrays: .length
Arrays kann man mit For Schleifen durchsuchen:
int[ ] a=new int[x]
for (int i = 0 , i <= x , i++)
Zugriff auf ein Element am Index i = int[i];

Rechenoperatoren und Ausdrücke

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

++a wird erhöht und dann damit gerechnet
a++ wird erst im Nachhinein erhöht.
→ wichtig wenn eingebunden in Formel/Rechnung
<< bzw. >> ist eine Bitverschiebung. Beispiel:
15 << 3 : (15=) 00001111 wird zu 01111000 (=120)

Operationen für Zeichenketten

Länge: < Name > . length();
Reverenzvergleich: ==
Inhaltsvergleich: < Name1 >.equals(< Name2 >)
Einzelnes Zeichen auslesen: <Name>.charAt(<Zahl>)
Zusammenhängen: Zwischen die Strings ein "+"
Beispiel: String hey = "hey"; String heyy = hey + "y";
Int to String: Integer.toString(<Intvariable>)
Char an Stelle: name.getChar(pos)

Typkonvertierung

Typsicherheit:
auf Operanden können nur die ihrem Typ entsprechenden Operationen angewandt werden.

implizite Typkonvertierung:
Vergrößerung des Wertebereichs
Beispiel: int j = 12; ⇒ double d = j;

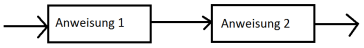
explizite Typkonvertierung:
Verkleinerung des Wertebereichs
Beispiel: double y = (double) x;

Bei Typkonvertierung sollte man aufpassen, da zuerst die rechte Seite betrachtet wird. Deshalb Typkonvertierung nicht in einer Rechnung, sondern davor.

Schleifen mit Ablaufdiagrammen

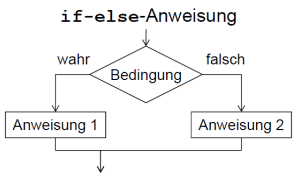
Definition (*Sequenz*)

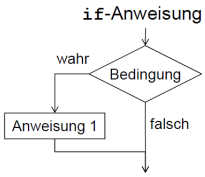
Folge von nacheinander auszuführenden Anweisungen, die mit einem Semikolon voneinander getrennt werden



Definition (*if-else*)

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



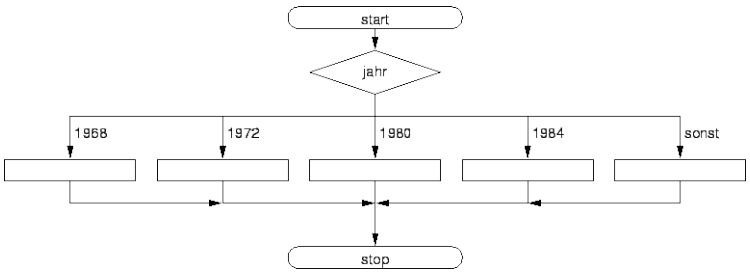


→ Bedinungen sind Boolean Auswertungen (Bsp.: (a == b))
Kurzschreibweise mit ?-Operator:
→ Form: < Bedingung > ? < Ausdruck1 > : < Ausdruck2 >
→ Normal: if (i > j) { max = i; } else { max = j; }
→ Kürzer: max = (i > j) ? i : j;
→ ?-Operator schwächer als == - Operator

Definition (*switch*)

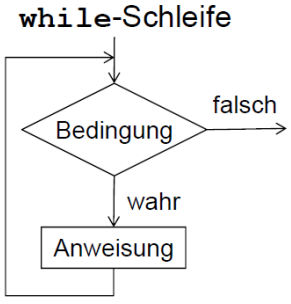
switch (<Ausdruck>) {
case <konstanter Ausdruck>: <Anweisung>: break;
...
default: <Anweisung>;
}

→ wird verwendet anstatt vielen if-else-Verzweigungen
→ sequentielle Abarbeitung bis break oder default Fall

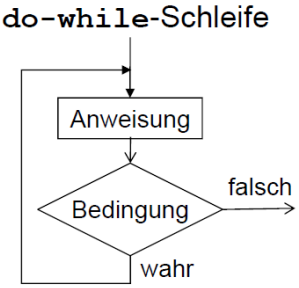


Definition (*while/do-while*)

Grundgedanke: wiederholte Ausführung von Aktionen, bis eine Zielbedingung erfüllt ist.
while (<Bedingung>) {
<Anweisungen>
}



Bei Do-while Schleifen Besonderheit: Die Anweisung wird mindestens einmal ausgeführt.
do {
<Anweisungen>
} while (<Bedingung>);



→ Durch falsche Bedingungen können Endlosschleifen entstehen!

Definition (*for*)

Kann verwendet werden, wenn bereits vor der Ausführung bekannt ist, wie oft der Schleifenrumpf durchlaufen werden soll.
for (<Initialausdruck>; <Bedingung>; <Inkrementausdruck>) {
<Anweisungen>
}
Eine for Schleife ist verwandt mit der while Schleife und kann auch durch eine while Schleife abgebildet werden

→ break: Ausführung der gesamten Schleife wird abgebrochen
→ continue: Aktueller Schleifendurchlauf wird abgebrochen

Methoden

Deklaration:
static < Rückgabetyp> < Methodenname> (Param1, ..., Param n){
Anweisungen;
}
Methodenaufruf: <Methodenname>(<Eingabewerte>)
→ wenn keine Rückgabe: Rückgabetyp ist void
→ Methoden kann man nicht Schachteln

Rekursion

Definition (*Rekursion*)

Vorgehen bei Rekursion:
Das Problem wird in immer kleinere Probleme gelöst, bis man bei dem kleinsten Problem angekommen ist. Das ist der Basisfall. Dann werden all diese Probleme gelöst und zu einer Gesamtlösung zusammengeführt.
Voraussetzung für Rekursion ist, dass die Teilprobleme bzgl. einer gegebenen Ordnung immer kleiner werden, man einen Basisfall hat und die Probleme in gewisser Weise ähnlich sind.
Eine Funktion ist rekursiv, wenn zur Berechnung von f diese Funktion f benutzt wird.
Funktion ruft sich selbst auf ⇒ Rekursionsschritt

linear rekursiv:	von oben nach unten und dann wieder hoch
endrekursiv:	nach unten und Ergebnis steht fest
kaskadenartig rekursiv:	zwei rekursive Aufrufe gleichzeitig nicht verschachtelt
verschachtelt rekursiv:	zwei rekursive Aufrufe ineinander
verschränkt rekursiv:	zwei Methoden die sich gegenseitig aufrufen

→ Rekursionarten können auch kombiniert auftreten.
→ Es ist möglich die Terminierung einer Rekursiven Methode anhand eines Induktionsbeweises über die programmierte mathematische Funktion zu beweisen.
→ Eine weitere Möglichkeit ist, eine ganzzahlige Terminierungsfunktion zu finden, sodass t bei jedem Rekursionsschritt streng monoton fällt und nach unten beschränkt ist.
→ bei Entrekursion wird im Methodenaufruf gerechnet, bei linearer Rekursion im Allgemeinen innerhalb des Returnblockes außerhalb des Methodenaufrufes
→ Vorteile Rekursion: Elegant, leicht lesbar, weniger Fehleranfällig, Terminierbarkeit ist leichter zu beweisen
→ Nachteile Rekursion: Bei hoher Rekursionstiefe extrem viel Speicherbedarf

Speicherreduzierung Rekursion

Durchreichen von Zwischenergebnissen:
→ bei kaskadenartiger Rekursion
→ Im Funktionsaufruf werden Zwischenergebnisse weitergegeben, aus denen sich neue Ergebnisse berechnen lassen.

Dynamisches Programmieren
→ keine kaskadenartige Rekursion
→ es sollte nicht ultra viele Zwischenergenisse geben
→ Zwischenergebnisse werden in Hilfsarray gespeichert
→ Rekursiver Aufruf beschreibt auch das Hilfsarray
→ Benötigt Basisfall der Hilfsarray abfrägt

Backtracking Breitensuche

Man geht immer eine Höhe des Entscheidungsbaumes ab und prüft ob alle Ergebnisse dieser Höhe die Lösung für das Ergebnis sind. Wenn nicht, geht man eine Höhe Tiefer.
Einfache Umsetzung z.b. mit for-Schleifen durchitterieren

Backtracking Tiefensuche

Man geht Ast für Ast des Baumes entlang und prüft ob man eine Lösung für das Problem gefunden hat.
Einfache Umsetzung mit Rekursiven Aufrufen

Backtracking Beispiel

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;}
```

Gierige Algorithmen

Hierbei wird eine Entscheidungsregel gewählt, anhand derer der Algorithmus den Entscheidungsbaum absucht. Die Schwierigkeit hierbei ist das finden einer Entscheidungsregel.

Asymptotische Aufwandsanalyse

Um die Laufzeit eines Programms zu berechnen, berechnet man die Summe der Elementaroperationen (z.b. x = 1), welche das Kostenmaß 1 haben.
Da die exakte Berechnung schwer ist, verwendet man Annäherungswerte:

Definition (*O-Notation (obere Schranke)*)

$f(n) \in O(g(n))$: $f(n)$ wächst höchstens so schnell wie $g(n)$

Definition (Ω - *Notation (untere Schranke)*)

$f(n) \in \Omega(g(n))$: $f(n)$ wächst mind. so schnell wie $g(n)$

Definition (Θ - *Notation*)

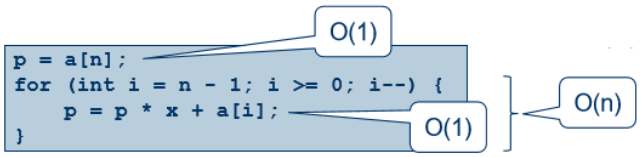
$f(n) \in \Theta(g(n))$: $f(n)$ wächst ebenso so schnell wie $g(n)$

Definition (*Rechenregeln für die O-Notation*)

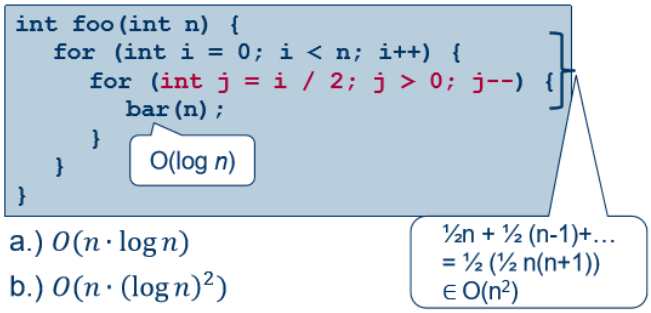
Seien $f(n) \in O(r(n)), g(n) \in O(s(n))$ und $c > 0$ konstant.
Dann gilt:
a) $f(n) + g(n) \in O(r(n) + s(n)) = O(max(r(n), s(n)))$
b) $f(n) \cdot g(n) \in O(r(n) \cdot s(n))$
c) $c \cdot f(n) \in O(r(n))$
d) $f(n) \pm c \in O(r(n))$

Definition (*Laufzeit der Ablaufstrukturen in O-Notation*)

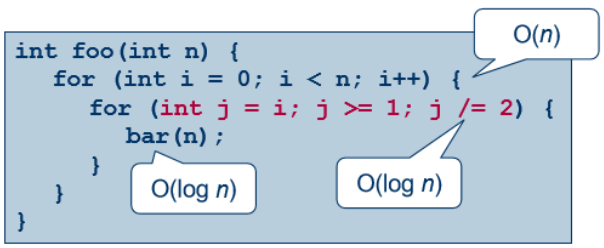
Elementaroperationen: Laufzeit O(1)
Sequenz: Addition der Laufzeit der Sequenzglieder
Konstante Schleife: Konstante kann vernachlässig werden.
Abhängige Schleifen von O(f(n)): O(f(n) * g(n))
Bedingte Anweisung: Addition beider Fälle
Methodenaufruf: Seperate Analyse der Methode



a.) $O(n)$ ✔



a.) $O(n \cdot \log n)$
b.) $O(n \cdot (\log n)^2)$
c.) $O(n^2)$
d.) $O(n^2 \cdot \log n)$ ✔



a.) $O(n \cdot \log n)$
b.) $O(n \cdot (\log n)^2)$ ✔
c.) $O(n^2)$
d.) $O(n^2 \cdot \log n)$

Objektorientierte Programmierung

Ein objektorientiertes Programm in Java besteht aus Klassen, die jeweils Attribut-, Konstruktor- und Methodendeklarationen enthalten. Eine der Klassen muss die main-Klasse sein, welche das Gesamtprogramm ausführt.

Definition (*Klasse*)

Benötigt zur Definition neuer Datentypen, d.h. eine Klasse ist eine Art Bauanleitung für Objekte (auch: Instanzen).
Klassennamen fangen mit einem Großbuchstaben an.

Aufbau einer Klasse:
- Klassendeklaration
- Datenkomponenten (Attributen) eines Objektes
- Methoden z.B. Manipulation der zugehörigen Attributwerte
- Konstruktoren zur Erzeugung von Objekten

Abstrakte Klassen: gekennzeichnet mit "abstract", keine eigenen Objekte, nur als Oberklasse sinnvoll

Klassenschachtelung:
Klassen kann man ineinander deklarieren. Methoden der inneren Klasse können auf Objekte/Methoden der äußeren Klasse zugreifen (auch private).

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 erzeugen Objekte. 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 mit unterschiedliche Parameterlisten haben

2. impliziter Konstruktor
Wird kein expliziter Konstruktor angegeben, so legt der Übersetzer automatisch einen impliziten Konstruktor an. Dieser erstellt automatisch anhand der Variablen Objekte

Beispiel:

```
public class Test {

 static public class Spam {

  int Dieter;                    // Attribute

  public Spam(int Dieter) {
   this.Dieter = Dieter;
  }

  public void run(){
   Dieter = Dieter + 1;
  }
 }

 public static void main( String[] args) {
  Spam penis = new Spam(0);      // Objekt erstellen

  penis.run();                   // Methode auf Objekt

  System.out.print(penis.Dieter);// Zugriff auf einzelne Attribute
 }
}
```

Definition (*Objekt*)

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.
Der Konstruktor belegt die Attribute der Objekte mit Werten.

Definition (*Objektvariablen*)

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*)

Erstellung eines neuen Objekts einer Klasse.
Form: Schlüsselwort new gefolgt von Konstruktoraufruf.
Beispiel: Bibmitglied bm1; bm1 = new Bibmitglied (Attribute des Kosntruktors)
Wertezuweisung: bm2 = bm1 bedeutet, beide verweisen auf den gleiche Speicher.

Satz

Methodenaufruf Methode von anderer Klasse: Klassenname.Methodenname(Attribute)
Innerhalb einer Klasse: Methodenname (Attribut)

Definition (*Klassenattribut/Klassenmethode*)

Klassenattribut: Hierbei teilen sich alle Objekte der Klasse denselben Attributwert. Schreiboperationen in einem Objekt wirken sich damit auf alle anderen Objekte aus.
Deklaration: static <Datentyp> <Name> = <Wert>;
Zugriff von außen: <klassenname>.<attributname>
Zugriff 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 kann sie als Getter und Setter für Klassenattribute verwenden oder Objekte irrelevant für die Methode sind.
Deklaration mit Schlüsselwort static

Definition (*Sichtbarkeit von Information*)

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*)

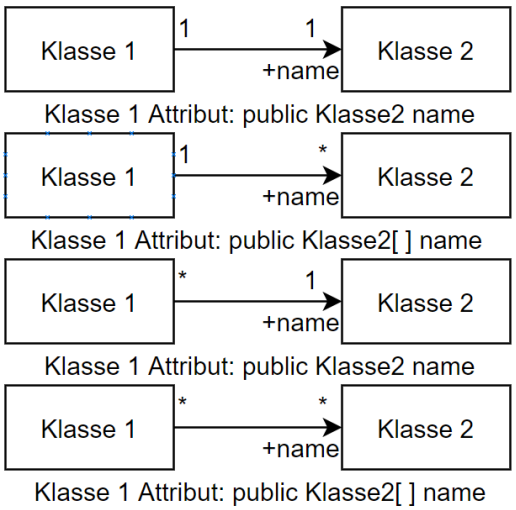
Bezeichnet das Verbergen von Programmierdetails vor dem direkten Zugriff von Außen. Direkter Zugriff auf interne Datenstruktur wird unterbunden und erfolgt stattdessen über wohl definierte Schnittstellen (Black-Box-Modell).

Definition (*Wrapper-Klassen*)

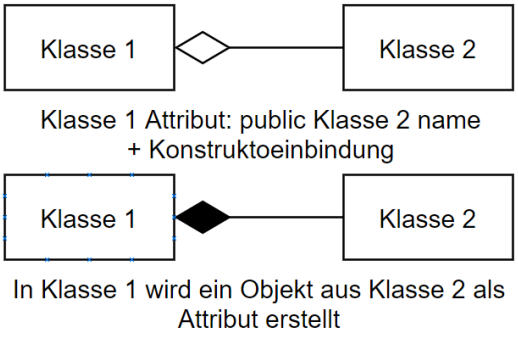
Viele Methoden erwarten Parameter von Typ Object. Um diesen Methoden Werte eines primitiven Datentyps zu übergeben, stell Java für jeden Datentyp eine sog. Wrapper-Klasse zur Verfügung, die diesen Wert kapselt.
Beispiel: Integer ii = new Integer(i);

UML - Unified Modelling Language

Assoziation: setzt Objekte zweier Klassen in Beziehung.
Multiplizitäten: Gibt an wie viele Objekte der beteiligten Klassen jeweils miteinander in Beziehung stehen können.
Sonderfälle:
0...* Beziehung brauch keinen Konstruktoreinbindung
1...* Beziehung braucht Konstruktoreinbindung
0...n Beziehung braucht einen Zähler



Aggregation:
UML-Darstellung: unausgefüllte Raute auf der Seite der Klasse
Komposition:
UML-Darstellung: ausgefüllte Raute auf der Seite der Klasse

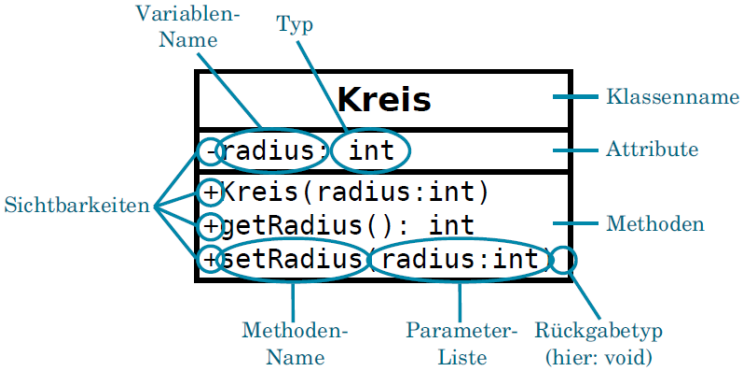


Abstrakte Klassen: über dem Klassennamen <<abstract>>
Interfaces: über dem Klassennamen <<interface>>
Statisch: Eigenschaft ist durch Unterstreichen gekennzeichnet

Sichtbarkeit in UML:
+ = public , - = private , # = protected , ++ = published

Instanzen zählen:
1: Instanzzählattribut a deklarieren
2: Pro Konstruktoraufruf a++

Klassenkarten



Vererbung

Die Unterklasse erbt von der Oberklasse die Attribute und Methoden, kann aber auf private Methoden und Attribute nicht zugreifen. Es können noch weitere Attribute und Methoden hinzukommen. Ein Objekt der Unterklasse ist auch ein Objekt der Oberklasse. Eine Unterklasse kann (in Java) nicht von mehreren Oberklassen erben
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.
Abstract Oberklasse:
<Klassendeklaration> <extends> <Oberklassenname> <{>
Interface Oberklasse:
<Klassendeklaration> <implements> <Oberklassenname

Polymorphismus

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

Überschreiben:
Tritt ein, wenn in Ober- und Unterklasse zwei Methoden mit der gleichen Signatur vorhanden sind.
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.

Polymorphe Variablen

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

Klasse: Konstruiert Objekte und definiert Signatur und Operationen für die Ojekte. Mit der Klassendefinition wird ein gleichnamiger Typ eingeführt.

Typsicherheit:
Es dürfen nur Methoden aufgerufen werden, die schon beim statischen Typ eines Objektes verfügbar sind.
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 Klassenattributen/methoden:
Ober- und Unterklassen Attributname gleich bzw. Ober- und Unterklassen Methodenname gleich.
Unterklassenattribut/methode überdeckt namesgleiches/signaturgleiche Oberklassenattribut/methode
Regel: beim Zugriff auf Klassenattribute und Klassenmethoden ist der statische Typ an der Zugriffsstelle der Relevante.

Robustes Programmieren

Fehlerbenachrichtigungen

Eine Exception unterbricht die normale Ausführungsreihenfolge
java.lang.Error (= serious problem) zeigt Probleme an, die das Programm 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.

Fehlerbehandlungscode

```
Naive Methode:
if (r < 0) {
 System.err.println("Fehler: negatives Argument");
 return -1;
}

Exeption:
if (r < 0) {
 IllegalArgumentException e
 = new IllegalArgumentException(
 "Fehler: negatives Argument");
 throw e;
}

try/catch:
try{
 code;
} catch (IllegalArgumentException e){
 System.err.println(e.getMessage());
} finally{
 < Dieser Code wird auf jeden Fall
   ausgeführt, egal was ist >
}
```

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. Solche Bedingungen lassen sich mit manuell aktivierbaren Assertions umsetzen.
assert Expression1 : Expression2
Expression1 ist boolescher Ausdruck und Expression2 ein Fehlermeldungsausdruck.

Testscode erstellen

Konstruktoren testen:
1: void Methode erstellen
2: Konstruktor aufrufen

Methoden testen:
1: Methode zum Testen erstellen
2: Objekt erzeugen
3: Assert.assertEquals("Fehlermeldung" , gewolltes Ergebnis, Methodenaufruf, bzw. Ergebnis der Methode)

wp-Kalkül

wp = weakest precondition, schwächste Vorbedingung

Regeln

$\rightarrow wp(\text{"}a := 7\text{"}, a = 7) = (7 = 7) = true$
$\rightarrow wp(\text{"}a := 7\text{"}, a = 6) = (7 = 6) = false$
$\rightarrow wp(\text{"}a := 7\text{"}, b = 12) = (b = 12)$
$\rightarrow wp(\text{"}a := b\text{"}, a > 1) = (b > 1)$
$\rightarrow wp(\text{"}s1; s2\text{"}, Q) = wp(\text{"}s1\text{"}, wp(\text{"}s2\text{"}, Q))$ (Sequenzregel)
$\rightarrow wp(\text{"}if\ b\ then\ s1\text{"}, Q) = [b \wedge wp(s1, Q)] \vee [\neg b \wedge Q]$
$\rightarrow wp(\text{"}if\ b\ then\ s1\ else\ s2\text{"}, Q) =$
$[b \wedge wp(s1, Q)] \vee [\neg b \wedge wp(s2, Q)]$

Schleifeninvariante finden

→ Schwer selbst rauszufinden
→ Gilt vor der Schleife
→ Nachbedingung lässt sich daraus implizieren
→ Finden durch Einsetzen

Invariante gilt vorher

→ 1. wp("Vardekl.1 , ... Vardekl.n" , Schleifeninvariante)
→ 2. von rechts nach links einsetzen und auflösen
⇒ Versuchen, nur noch Parameter zu haben
→ 3. Ergebnis = bekannte Vorbedingung

Invariante während Schleife

→ 1. wp("A1 , ... , A2" , I) (A = Algo.zeile, I = Invarianz)
→ 2. von rechts nach links einsetzen und umformen/aufösen
→ 3. Gleichheit mit Invarianz zeigen
→ Hilfsmittel: Fallunterscheidung bzw. Induktion

Nachbedingung gilt

→ 1. ¬ b bestimmen (b = Schleifenbedingung)
→ 2. wp("A,Q") (A = Algo nach Schleife, Q = Nachbedingung)
→ 3. Zeige: $(I \wedge \neg b) \rightarrow wp(\text{"}A, Q\text{"})$

Schleifenvariante finden

→ Benötigt zur totalen Korrektheit
→ Beweist, dass die Schleife terminiert
→ Schleifenbedingung umstellen

Korrektheit von Schleifen

Tipps:
Präinkrement: NACH der Substitution behandeln
Postinkrement: VOR der Substitution behandeln Partielle korrekt:
Schleifeninvariaten existiert und gilt vor der Schleife und Q kann daraus impliziert werden
Total korrekt:
Schleife ist partiall korrekt und Schleifenvariante existiert

Grundlegende Datentypen

Spezifikation von Datentypen

Man legt fest:
- adt: Namen des ADTs
- sorts: Liste verwendeter Datentypen
- ops: Beschreibung nutzbare Operationen
- axs: Axiome zur Beschreibung der Semantik: Basisfälle

Dies geschieht in der Signatur (Bsp.):

adt	IntSet		
sorts	IntSet, Int, Boolean		
ops	create:		→ IntSet
	insert:	IntSet × Int	→ IntSet
	delete:	IntSet × Int	→ IntSet
	contains:	IntSet × Int	→ Boolean
	isEmpty:	IntSet	→ Boolean
axs	isEmpty(create)	= true	
	isEmpty(insert(x,i))	=false	
	insert(insert (x,i),i)	= insert(x,i)	
end	IntSet		

Vorteile:
Man kann Datentyp genau soweit festlegen, wie gewünscht, ohne Implementierungsdetails vorwegzunehmen.

Nachteile:
Bei komplexen Anwendungen unübersichtliche Menge an Gesetzen, deren Nutzen nicht immer leicht ersichtlich ist und deren Vollständigkeit nicht leicht zu prüfen ist.

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

Konstruktoren (erforderlich):
mit create und insert lassen sich alle mögl. Objekte erzeugen.

Klassen:
Zugriff auf andere Klassen mit: Klassenname.Methodenname

Generische/Parametrisierte Klassen

Vorteile: Code muss nur einmal geschrieben werden. Platzhalter kann durch einen beliebigen Typ ersetzt werden.

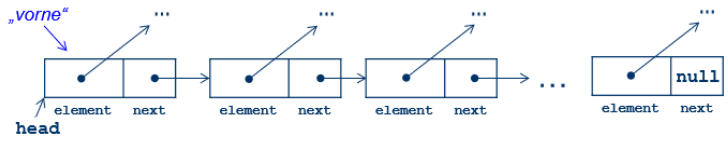
Durch Angabe des Typplatzhalters T in spitzen Klammern wird Klasse zur generischen bzw. parametrisierten Klasse. Mehrere Typparameter werden durch Kommata getrennt angegeben. Typparameter können innerhalb der Klasse als Typen von Variablen oder Rückgabewerten oder als Parameter von Methoden genutzt werden.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

```
public class Container<T> {
 private T value;
 public T setValue(T n) {
    value = n;
} }

// Containter fuer Integer-Zahlen
Container<Integer> i = new Container<Integer>();
int a = 3;
i.setValue(a);

// Container fuer Gleitkommazahlen
Container<Double> d = new Container<Double>();
double b = 0.2;
d.setValue(b);
```

Elementare Listen

→ Listen enthalten endliche Folgen von Objekten
→ Reihenfolge der Objekte hat definierte Ordnung
→ Fast Element hat Vorgänger und Nachfolger
→ Erstes Element ohne Vorgänger, letztes ohne Nachfolger
→ Listen dürfen Duplikate enthalten
→ Man unterscheidet unsortierte und sortierte Listen

Struktur:



Code:

```
public class Listenoperationen {
 Listenkonstruktor head;

 public int erstesElement() {
  if (head != null) {
   return head.element;
  } else {
   return -1;
  }
 }

 public void anhaengen(int x) {
  Listenkonstruktor tmp = new Listenkonstruktor();
  tmp.element = x;
  head = tmp;
  tmp.next = head;
 }

 public boolean istleer() {
  if (head == null) {
   return true;
  }else{
   return false;
  }
 }
}
```

Stapel/Keller (Stacks)
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
(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.

Javas Linked List (java.util.List<E>)

Element anhängen: → boolean add(E elem);
Element an index einfügen → void add(int index, E elem);
Alle Elemente löschen: → void clear();
Objekt enthalten?: → boolean contains(Object o);
Element an Index: → E get(int index);
Liste leer?: boolean isEmpty();
Objekt entfernen: boolean remove(Object o);
Element an index entfernen: → E remove(int index);
Element ersetzen und erhalten: → E set(int index, E elem);
Anzahl der Elemente: → int size();
Tipp: == null nicht immer möglich

Verkettete Listen

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

Listen mit Wächterelement:
Enthält Objekt am "Anfang" welches auf den vordersten Listeneintrag zeigt. Der letzte Listeneintrag zeigt auf das Wächterelement. Der Vorteil ist, dass es weniger Sonderfällt gibt, z.b. Einfügen am Anfang ist wie Einfügen irgendwo.
Im Listenkonstruktor muss dann zusätzlich noch das Wächterelement erzeugt werden.

Durchlaufen verketteter Liste:
for-each-Schleife: andere Form der for-Schleife, mit der sich Listen leichter durchlaufen lassen. Syntax:
for(<Objekttyp> <Laufvariable> : <Behältername>){ }
Beispiel:

```
int[] array = new int[]{1,2,3};
for(int k : array) System.out.println(k);
Output: 1 2 3
```

Werte vergleichen:
1.compareTo(2) = negativ
2.compareTo(2) = 0
3.compareTo(2) = positiv

Iterator:
Möglichkeit zur Navigation durch eine beliebige Datenstruktur. Der Iterator ist dabei eine Art Zeiger.
Verfügbar durch: java.util.Iterator<E>
Ob nächstes Element: → boolean hasNext();
Nächstes Element: → E next();
Element löschen: → void remove();

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
}
```

Aufwand:
Suche/Zugriff: O(n)
Einfügen/Löschen vorne/hinten: O(1)
Einfügen/Löschen Mitte: O(n)
Länge bestimmen: O(1)

Dynamische Arrays

Kombination der Vorteile von Listen und Arrays. Kombination der Vorteile ergibt sog. dynamisches Array. Hierbei dauert es länger, Elemente anzufügen (von O(1) auf O(n)), dafür ist Werte ändern und erhalten schneller(von O(n) aufO(1))

Mengen

Ziel:
Mengen so darstellen, dass die klassischen Mengenoperationen effizient ausgeführt werden können. Mengen enthalten keine Duplikate und Elemente haben keine feste Reihenfolge. Es gibt vier Möglichkeiten der Implementierung:

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

2. Einfach verketteter Liste mit Sortierung:
Wie 1. nur mit dem Vorteil, dass man nicht die ganze Liste durchlaufen muss, wodurch Mengenoperatoren schneller sind. Elemente werden mit dem Reißschlussverfahren eingefügt

3. Dynamischem Array mit 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. Bitvektor ohne Sortierung

Binäre Suche / Lineare Suche

	binäre Suche	lineare Suche
bester Fall	$O(1)$	$O(1)$
schlechtester Fall	$O(\log n)$	$O(n)$
Durchschnitt (erfolgreich)	$O(\log n)$	$O(n)$
Durchschnitt (erfolglos)	$O(\log n)$	$O(n)$

Binäre Suche kann bei Sortierten Datenstrukturen verwendet werden. Dabei wird der zu Durchsuchende Teil Schrittweise halbiert und durch Vergleiche der jeweils richtige Teil weiterverarbeitet, bis man das Element gefunden hat.
Lineare Suche ist einfach das durchgehen von vorne nach hinten

Streutabellen / Hashmaps

Anstatt in Mengen zu speichern, wird das zu speichernde in "Buckets" aufgeteilt. Mit einem Schlüssel lässt sich der "Bucket" ermitteln.

LastFaktor: Einträge / Buckets

Offenes Hashing:
Jeder Bucket stellt eine verkettete Liste dar, d.h. in jeden bucket passen unendlich viele Elemente.
Vorteile: Löschen ist möglich
Nachteile: Benötigt viel Speicherplatz (für die Zeiger) und es können lange Listen entstehen.

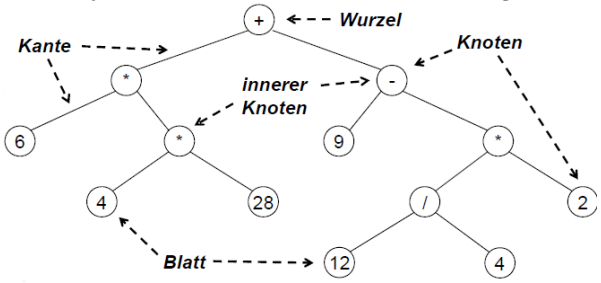
Geschlossenes Hashing:
Nur ein Platz pro Bucket, zum aufteilen hat man eine Funktion (Hashfunktion).
Vorteil: sparsamerer Umgang mit Speicher.
Nachteil: Löschen ist nicht dirket möglich, langsamer als offenes Hashing

Bäume

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 maximal 2 Nachfolger
> Der leere Baum ist 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		

Implementierung:

```
class Entry<E> {
 E element;
 Entry left, right;
 public Entry(Entry<E> l, E x, Entry<E> r) {
  left = l;
  element = x;
  right = r;
 } … }
class BinTree<E> {
 Entry<E> root;
 …      // Konstruktor und Methoden
        // der Klasse BinTree }
```

Definition (*Allgemeine Bäume*)

> Anzahl der Kinder eines Knotens ist beliebig.
> Keine festen Positionen für Kinder.(wie left, right)
> Grad eines Knotens: Anzahl Kinder
> Grad eines Baumes: max. Grad aller Knoten

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. Suchbäume sehen je nach Einfügreihenfolge anders aus.
Binäre Suchbäume haben keine doppelten Einträge

Traversierung, Besuchsreihenfolgen:
Breitensuche: Zuerst eine Höhe, dann die andere
Tiefensuche: Zuerst ein Ast, dann der andere
Besuchreihenfolge bei Tiefensuche:

Inorder:	Preorder:	Postorder:
1. linkers	1. aktuell	1. links
2. aktuell	2. links	2. rechts
3. rechts	3. rechts	3. aktuell

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

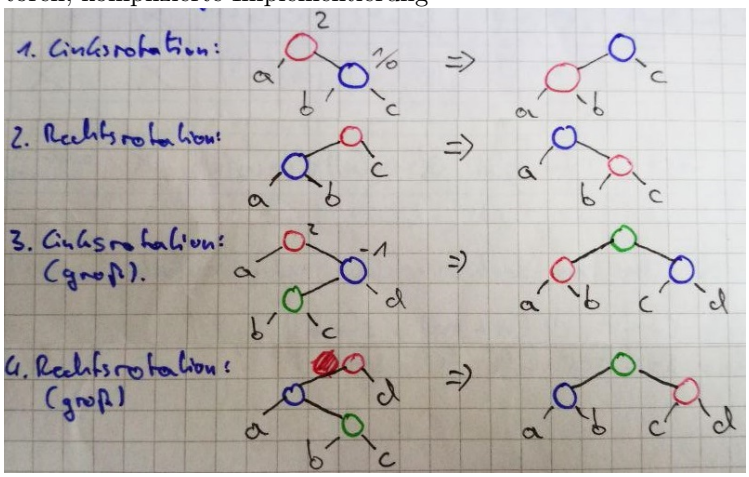
Löschen eines Wertes:
- Suche nach Knoten mittels Such-Algorithmus
- falls zu löschender Knoten Blatt-Knoten ist: aus Baum entfernen
- falls zu löschender Knoten kein Blatt-Knoten ist: Knoten ersetzen durch...
- größten Knoten im linken Teilbaum oder kleinsten Knoten im rechten Teilbaum
- dadurch ggf.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

AVL-Bäume

Muss man nicht programmieren können, hier reicht die Theorie!
Ermöglicht Operationen in O(log(n))
Balancefaktor berechnen: O(n)
Korrektur Aufwand: O(log(n))
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



Halden

Löschen/Einfügen: O(log n)
Heap ist ein partiell geordneter Binärbaum.
Max-Heap: Elternknoten immer größer als Kinderknoten
Min-Heap: Elternknoten immer kleiner als Kinderknoten
Verwendung als Prioritätswarteschlange
effiziente Speicherung in Array möglich
Feldeinbettung: Lesen von oben nach unten, von links nach rechts

Einfügen eines neuen Elements
- Element bei nächster freien untersten Position einfügen
- falls Ebene voll: Element wird erster Knoten neuer 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

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 einziges Array
stabiles Sortierverfahren: gleiche Werte ändern ihre relative Reihenfolge nicht.
Laufzeiten vergleichsbasierter Sortierverfahren:

Bezeichnung	Best	Average	Worst	stabil	in situ
Einfach					
SelectionSort	n^2	n^2	n^2	mgl.	array
InsertionSort	n	n^2	n^2	ja	array
BubbleSort	n	n^2	n^2	ja	ja
Verfeinert					
HeapSort	$n \log n$	$n \log n$	$n \log n$	nein	ja
Divide/Conquer					
MergeSort	$n \log n$	$n \log n$	$n \log n$	ja	nein
QuickSort	$n \log n$	$n \log n$	n^2	nein	ja

Laufzeiten nicht-vergleichsbasierter Sortierverfahren:

Bezeichnung	Zeit	stabil	in situ
BucketSort	$n + m$	ja	nein
RadixSort	$n \cdot k$	ja	nein

SelectionSort
Zwei Listen benötigt: SortList und ResultList
1: Finde Maximum in SortList
2: Füge Maximum an ResultList VORNE dran
3: Lösche Maximum in SortList, Goto 1
InsertionSort
Zwei Listen benötigt: SortList und ResultList
1: Nimm Erstes Element E aus SortList
2: Finde Element E' in ResultList mit E < E'
3: Füge Element E vor E' ein
4: Falls kein E', Füge es am Ende ein
5: Nimm nächstes Element aus SortList, Goto 2
Tipp: Bei Arrays von oben nach unten jeweils nach rechts verschieben (Bsp.: $a[i+1] = a[i]$). Mit .compareTo() vergleichen: 1.comparteTo(2) = negativ, 1.compareTo(1) = 0, 1.compareTo(0) = positiv
BubbleSort
Gegeben: Ein Array oder Liste
Benötigt: Zwei Iteratoren (z.b. for (int i = 0; i < upper; i++)
Benötigt: Zwei Schleifen, do-while und for-Schleife
Benötigt: Schleifenbedingung swapped für do-while
1: swapped = false und Vergleiche Element i mit i+1
2: Tausche Elemente, falls $i > i + 1$
3: Wenn getauscht, setzte Schleifenbedingung auf true
4: Wenn Array / Liste durchiteriert: upper - 1
5: Wenn Swapped = true: Goto 1
HeapSort
Gegeben: z.b. Max-Halde im Array
Benötigt: ArrayLänge: n und for-Schleife(int i = n-1 ; i > 0 ; i–)
1: Max Element mit letztem ArrayElement Index i Tauschen
2: Haldeneigenschaft für 0 bis Index i-1 Widerherstellen
3: Lass For-Schleife weiterlaufen, also Goto 1 mit i- -
MergeSort
Gegeben: Eine Liste
Benötigt: zwei Methoden und Rekursion
1. Äußere Methode:
1: Basisfall: Liste kleiner als 1: Liste zurückgeben
2: Liste halbieren und als zwei Listen speichern
3: Verschränkter Rekursiver Aufruf mit zweiter Methode
Bsp.: return merge(mergesort(left) , mergesort(right))
2. Verschmelz-Methode:
1: Neue return Liste ret erstellen
2: 1te while Schleife, solange übergebene Listen > 0
3: Erste Elemente der Teillisten speicher
4: Kleineres Element KLE an ret HINTEN anhängen
5: KLE in Teilliste löschen, 1te while-Schleife Ende
6: Alle anderen Werte an ret Anhängen
7: Alle anderen Werte bei den Teillisten löschen
Tipp: Verschmelzen benötigt 3 Schleifen
QuickSort
Gegeben: Ein Array
Benötigt 3 Methoden: Quicksort, QuicksortRec, Partition
1. Quicksort(array):
1: Aufruf von QuicksortRec mit Array , 0 , Arraylänge - 1
2. QuicksortRec(array, m , n):
1: Solange m > n
2: Partition aufrufen mit (array , m , n)
3: Ergebnis von Partition speicher z.b. r
4: quicksortRec(a,m,r-1)
5: quicksortRec(a,r+1,n)
3. Partition(array,m,n):
1: Pivotelement wählen und Zwischenspeichern
2: Neue Variablen i = m-1 und j = m anlegen
3: Array von j bis n durchiterieren
4: Wenn a[j] kleiner gleich Pivotelement: i++ ; Index i und j tauschen, Ansonsten j++;
5: Array Ende; Index i+1 und n Tauschen
6: i+1 als neues Pivotelement zurückgeben
Vorteile Quicksort: Kann Schneller als Heapsort/Mergesort sein, da weniger Instruktionen pro Schleifen
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
Gegeben: Menge größerer Zahlen
Bucktes für Wertintervalle.
1: Man nehme rechteste Ziffer
2: Sortierung wie bei Bucketsort
3: Wähle Ziffer weiter links, Goto 2

Graphen und Graphalgorithmen

Definition Graph:
Ein Menge von Knoten, verbunden durch eine Kante.
Kantenrichtung:
Es gibt ungerichtete und gerichtete Graphen. Ungerichtete Graphen können in einen gerichteten überführt werden.
Pfad:
Eine endliche Folge von Knoten von v nach w. Die Länge eines Pfades ist die Anzahl der Kanten im Pfad. Ein Pfad heißt einfach, wenn alle Knoten im Pfad paarweise verschieden sind.
Zyklus:
Pfad von v nach v über andere Knoten. Minimaler Zyklus, wenn außer v kein anderer Knoten mehr als einmal vorkommt. Bei ungerichteten Graphen hat ein Zyklus mindestens drei Knoten.
Vorgänger und Nachfolger:
w direkter Vorgänger von v: Genau eine Kante von w nach v
w direkter Nachfolger von v: Genau eine Kante von v nach w
w Vorgänger von v: Mehrere Kanten von w nach v
w Nachfolger von v: Mehrere Kanten von v nach w
Eingangsgrad: Anzahl der direkten Vorgänger eines Knoten
Ausgangsgrad: Anzahl der direkten Nachfolger eines Knoten
Bei ungerichteten Graphen spricht man vom Grad des Knotens.
Quelle: Keine Vorgänger, nur Nachfolger
Senke: Nur Vorgänger, keine Nachfolger
Zusammenhänge
Stark Zusammenhängend: Jeder Knoten ist von jedem anderen aus erreichbar. Bei ungerichteten Graphen kein "stark" möglich
Ein gerichteter Graph heißt schwach zusammenhängend, wenn der zugehörige ungerichtete Graph zusammenhängend ist.
Teilgraph
Teilgraph: Teilmenge an Kanten und Knoten.
induzierter Teilgraph: Teilmenge an Knoten, von denen alle Kanten übernommen werden
Spannbaum: Zyklenfreier, zusammenhängender, alle Knoten beinhaltender Teilgraph
Wurzel
Ein Knoten v eines gerichteten azyklischen Graphen (DAG) heißt Wurzel, falls es keine auf ihn gerichteten Kanten gibt.
Graphengewichtung
Zuordnung einer Gewichtung an jede Kante
Kosten eines Pfades ist die Summe der Gewichte seiner Kanten
Eulerweg: Enthält alle Kanten genau 1 mal
Eulerzyklus: Eulerweg mit Startknoten = Endknoten

Darstellungen von Graphen

Adjazenzmatrix:
boolesche $n \times n$ Matrix mit true, wenn Kante zwischen Knoten. Bei gewichteten Graphen speichert man die Kantengewichte in der Matrix. Fehlende Kanten stellt man mit einem Spezialwert für Unendlich dar (Double.NaN , Integer.Max_Value/3).
Diese Repräsentation wählen, wenn Tests auf das Vorhandensein von Kanten häufig vorkommen.
Vorteile:
In O(1) feststellbar, ob Kante existiert
Nachteile:
Hoher Platzbedarf, Auffinden aller Knoten dauert O(n), Inizialisierung der Matrix dauert lang
Adjazenzliste:
Verwaltung der Nachbarknoten eines Knoten als Liste
Vorteil: Geringerer Platzbedarf
Nachteil: Test, ob v,w benachbart dauert lange
Adjazenzfeld
Zuerst Knoten, die Auf den Index im Array verweißen, ab dem die Kanten zu anderen Knoten stehen
Vorteil: Sehr geringer Platzbedarf
Nachteil: Einfügen und Löschen sehr aufwändig

Graphendurchlauf

Tiefensuche:
1: Startknoten wählen
2: Neuen, bisher unbesuchten Knoten finden
3: Neuen Knoten als besucht markieren, Goto 2
Tipp: Ähnlich wie Backtracking
DFS-Nummerierung: Nummerierung der Reihenfolge, mit welcher Tiefensuche die Kanten besucht

Breitensuche:
1: Besuche Knoten, deren Pfad Länge 1 von der Wurzel hat
2: Makiere Besuchte Knoten
3: Besuche Knoten, deren Pfad Länge 2 von der Wurzel hat
4: Usw. Abbrechen, wenn bereits besuchter Knoten

Kürzester Weg

Dijkstra:
Aufwand: $O(V \cdot \log(V) + E)$ mit E=Kanten, V=Knoten
1: 0 für Startknoten und ∞ für andere Knoten in die Tabelle
2: Kosten vom Startknoten zu anderen Knoten eintragen
3: Billigsten Weg nehmen, Addition aller Werte eintragen
4: Vom neuen Teilbaum Wege zu neuen Knoten eintragen
5: Goto 3
Tipp: Wenn man nicht mit einem Weg durch alle Knoten kommt, einfach billigsten neuen Ast erschaffen
→ Dijektra ist ein gieriger Algorithmus
Pseudocode:
Startknoten mit Distanz 0, alle anderen Distanz unendlich
Makiere Alle Knoten als unbesucht
solange es noch unbesuchte Knoten gibt:
wähle Knoten mit geringster Distanz
Makiere diesen Knoten v als besucht
Berechne Kanten + Pfad für alle Nachfolger von v

Floyd:
Man erschafft Pseudokanten von einem Knoten zu allen anderen Knoten, indem man den kürzesten Weg über andere Kanten zusammenfasst

Minimaler Spannbaum

Prim:
Aufwand: $O((|E|)+|V| \cdot \log|V|)$ mit E=Kanten und V=Knoten
1: Billigsten Pfad nehmen
2: Neuer Teilbaum
3: Von neuem Teilbaum, billigsten neuen Knoten/Teilbaum nehmen
4: Goto 2
Kruskal:
Aufwand: $O(|E| \cdot \log|E|)$ mit E=Kanten
1: Anfang an billigster Kante
2: nächste billige Kante zu neuem Teilbaum nehmen
3: Wenn keine solche Kante im Anfangsbaum: neuer Teilbaum
→ gieriger Algorithmus
→ gut bei wenig Kanten und vielen Knoten

Geometrische Algorithmen

Punkt in Polygon Problem

Beste Laufzeit: $O(log(n))$
Gegeben: 2 Punkte und 1 Polygon
Frage: Beide Punkte in/außerhalb eines Polygons
Vorgehen: Man zieht einen Punkt unendlich weit nach außen und dann bildet man eine Strecke zum anderen Punkt. Während man die Strecke konstruiert, addiert man alle Schnittpunkte. Bei einer Gerade Anzahl, liegen die Schnittpunkte in der gleichen Ebene.
Sonderfall 1: Beim schneiden einer Kante des Polygons muss betrachtet werden, ob man die Seiten wechselt oder nicht. Geht also die Strecke von innen nach außen, muss dies mitgezählt werden.
Sonderfall 2: Wenn die Strecke auf einer Kante des Polygons liegt, gleiches vorgehen wie bei Sonderfall 1.

Konstruktion von Polygonen

Beste Laufzeit: $O(n\ log(n))$
Gegeben: Punktmenge
Frage: Wie bekommt man garantiert ein Polygon mit überschneidungsfreien Kanten?
Vorgehen: Die Verwendung eines gierigen Algorithmuses bzw. Backtracking ist nicht sinnvoll. Daher: Extrempunkt wählen, zu jeden anderen Punkt Gerade bilden und Steigung bestimmen. Steigungen sortieren und der Reihenfolge nach verbinden.
Sonderfall 1: Gleiche Steigung, anhand des Abstandes wählen
Sonderfall: Zwei Punkte mit 180° Winkel: Neuer Extrempunkt

Konvexe Hülle

Beste Laufzeit: $O(n\ log(n))$
Gegeben: Punktmenge
Frage: Kleinstes Polygon, dass alle Punkte im Inneren hat
Ausdehnungsalgorithmus: Beginne mit Polygon aus der beliebigen Punkten. Schritt: Wähle beliebigen, noch nicht bearbeiteten Punkt (möglichst weit außen). Liegt dieser Punkt außerhalb des existierenden Polygons, modifiziere die Hülle. (Aufwand: n^2)
Einpackalgorithmus: Beginne mit äußerstem Punkt. Schritt: Verbinde immer zu dem Punkt, den der Rotierzeiger als nächstes erreicht. (Aufwand: n^2)
Graham's Einpackalgorithmus: Wähle Extrempunkt e. Lege durch jeden Punkt und e eine Gerade und bestimme Steigung. Sortiere Steigung und verbinde immer wieder den nächsten Punkt in der Steigungsreihenfolge. Sonderfall: 180°, solange Kanten entfernen, bis Winkel zu neuem Punkt kleiner als 180° ist. (Aufwand: $nlogn$)

Ballung und nächstes Paar

Beste Laufzeit: $O(n\ log(n))$
Gegeben: Punktmenge
Gesucht: Einteilung in homogene Klassen
Zweck: Finden was zusammengehört
Klassifikation: Punkte werden zu Klasse zusammengefasst
Heterogenitätsindex: Gibt an, wie gut die Punkte in einer Klasse zusammenpassen
Güteindex: Bewertet Klassifikation
Hierachisches Verfahren:
Zusammenfassen: Zu Beginn enthält jede Klasse genau einen Punkt. Füge in jedem Schritt zwei Klassen zusammen.
Zerteilen: Beginne mit einer Klasse, die alle Punkte enthält, Zertieile diese in jedem Schritt in zwei Unterklassen
Pro Klassifikation wird Güteindex berechnet. Man wähle pro Schritt eine Klassifikation, bei der sich der Güteindex verbessert
Finde nächstes Paar: Punkte mit geringstem Abstand