# Mitschriften Informatik B

## Inhaltsverzeichnis

Mitschriften Informatik B ........ 1
Inhaltsverzeichnis ........ 1
Organisatorisches ........ 5
  Tutorien ........ 5
  Test ........ 5
  Klausur ........ 5
  Sommertutorium ........ 5
Themenüberblick ........ 5
Graphentheorie ........ 5
  Definition von Graphen ........ 5
  Darstellung von Graphen ........ 5
    Beispiel für einen Graphen mit Adjazenzmatrix-Darstellung ........ 6
  Beispiele für Aufgabenstellungen, die mit Graphen lösbar sind ........ 6
    1. 4-Problem: 4 Farben reichen für jede Landkarte ........ 6
    2. Welche Graphen treten als Ecken-Kanten-Gerüst von Polyedern auf? ........ 6
    3. Euler'sche Graphen ........ 6
    4. Hamilton'sche Graphen ........ 7
    5. Das TSP-Problem ........ 7
    6. Planare Graphen ........ 7
    7. Computer-Netzwerke ........ 7
  Teilmengen von Graphen ........ 8
    Untergraph ........ 8
    Komplementärer Graph ........ 8
    Isomorphe Graphen ........ 8
  Beispiele für Graphen ........ 8
  Zusammenhang und Abstand in ungerichteten Graphen ........ 8
    Charakterisierung bipartiter Graphen ........ 9
Bäume ........ 9
  Durchmustern eines Baums ........ 10
    Einschub Queue ........ 10
    Durchmustern eines Baums: Breitensuche (BFS; broad first search) ........ 10
    Durchmustern eines Baums: Tiefensuche (DFS; deep first search) ........ 11
  DAGs (Directed, Acyclic Graphs; Gerichtete, azyklische Graphen) ........ 12
    Verfahren 1 mit brute-force ........ 12
    Verfahren 2 mit DFS ........ 12
  Minimal ausspannende Bäume ........ 13
    Generischer MST-Algorithmus ........ 13
    Beispiel-Graph ........ 14
    MST Algorithmus von Prim ........ 14
    MST-Algorithmus von Kruskal ........ 15
Laufzeitabschätzung ........ 15
  Betrachtung des Worst Case ........ 15
    Ziel ........ 15
  Wachstum von Funktionen und die $\mathcal{O}$-Notation ........ 16
  Laufzeitanalyse der $\mathcal{O}$-Notation ........ 16
    Beispiel für eine Laufzeitanalyse ........ 16
    Zeigen des Ergebnisses der Kaufzeitanalyse ........ 16
  Laufzeit einiger wichtiger Funktionen ........ 17

Definition: $f(n) = o(g(n))$ .... 17
Kochrezepte .... 17
Beispiele für den Aufwand verschiedener Algorithmen für das gleiche Problem .... 18
Funktionale versus Imperative Programmierung .... 18
Beispiel ggT(x,y) .... 19
Definition Progamm .... 19
Welche Sprache versteht der Rechner? .... 19
Assembler .... 19
Beispiel für Assembler-Code .... 19
Höhere Sprachen .... 19
Programmparadigmen .... 19
Funktionale (applikative) Programmierung .... 19
Vertreter .... 19
Eigenschaften Applikativer Algorithmen .... 20
Beispiele für Rekursion .... 20
Türme von Hanoi .... 20
Idee .... 20
Analyse .... 20
Implementation in Java .... 20
ggT .... 21
Beispiel für nicht-offensichtliche Semantik .... 21
Probleme bei Rekursionsalgorithmen .... 21
Imperative Programmierung .... 21
Vertreter .... 21
Eigenschaften .... 21
Basiskonzepte .... 22
Zustand eines Rechners .... 22
Ausdrücke (entspricht Termen bei der Funktionalen Programmierung) .... 22
Komplexe Anweisungen .... 22
Prinzipielle Aufbau eines Imperativen Programms .... 22
Kurzfassung Imperatives Programm .... 22
Beispiele für Imperative Programmierung .... 22
Fakultätsfunktion .... 22
Fibonaccizahlen .... 23
ggT1 .... 23
ggT2 .... 23
xyz .... 23
Problem der Korrektheit und der Terminierung von Programmen .... 23
Terminierung der Schleife .... 23
Beispiel für ein Problem der Terminierung der Schleife .... 23
Java .... 24
Geschichte .... 24
Entwurfsziele .... 24
Links zu Geschichte von Java .... 24
Was ist Java? .... 24
Wie arbeitet Java? .... 24
Aufbau eines Java-Programms .... 24
Beispiel für ein Java-Programm: Fibonacci .... 25
Java Syntax .... 26
Primitive Datentypen (Basistypen) .... 26
Tabelle für Datentypen .... 26
2-Komponenten-Darstellung von Zahlen .... 26
Syntax für die Namen und Bezeichner für Variablen, Klassen, Methoden .... 27

- Beispiele für die Syntax von Variablendeklarationen 27
- Referenzdatentypen 27
  - Erstes Beispiel Arrays (Felder) 27
  - Beispiel 1 27
  - Beispiel 2 27
  - Beispiel 3 27
- Kopieren von Feldern 28
  - Zeichenketten 28
  - Einige Stringmethoden 28
  - Beispiel für Umgang mit Zeichenketten 28
  - Die Klasse `java.long.StringBuffer` 28
- Operatoren und Kontrollfluss bei Java 28
  - Anweisungsoperator = 28
  - Datoperator 29
  - Arithmetische Operatoren `+`, `-`, `*`, `/`, `%` (modulo) 29
  - Vergleichsoperatoren <, >, <=, >=, == (Gleichheit), `!=` (Ungleichheit) 29
  - Logische Operatoren 29
  - Einstellige Operatoren 29
  - "Abgekürzte" Schreibweisen 29
  - Bitweise Operatoren 29
  - Casting 30
- Einige Beispiele für typische Java-Entitäten 31
  - Beispiel für ein Objekt 31
  - Beispiel für Instanz eines Objekts 31
  - Beispiele für Methoden 31
  - Beispiele für Manipulation des Objekts 31
- Klassendefinition in Java 31
- Methoden 32
  - Spezielle Methoden 32
  - Methode `main` 32
  - Beispiel zur Definition einer Methode 32
- Ziele beim Softwareentwurf 33
- Charakteristika beim OO-Entwurf 33
- Vererbung, Beschatten/Überschreiben und Abstrakte Klassen/Interfaces 33
  - Beschatten (Shadowing) von Instanzvariablen und Statischen Methoden 34
  - Überschreiben von Methoden 34
  - Abstrakte Klassen und Interfaces 35
  - Mehrfachvererbung 35
  - Zusammenfassung Vererbung und Klassenhierarchien 35
- Polymorphie 36
- Analyse: Amortisierte Komplexität 37
  - Die Aggregat-Methode 37
  - Die Accounting-Methode 37
  - Die Potentialfunktion-Methode 37

Abstrakte Datenstrukturen (ADTs) 37
- Queues, Verkettete Listen und Deques 37
  - Queues (Warteschlangen) 37
  - Verkettete Listen 39
  - Deques 39
- Vektoren, Listen, Sequenzen 39
  - Vektor als ADT mit Methoden 40
  - Listen 40
  - Sequence 41

Bäume als Datenstruktur ........ 41
Einige Eigenschaften gewurzelter Bäume ........ 41
try catch finally Mechanismus ........ 42
Beispiel zu Exceptions ........ 42
Gewurzelte Bäume als ADT ........ 42
Methoden zum ADT Gewurzelter Baum ........ 42
Implementierung von `depth` als Methode von `TreeNode` ........ 43
Implementierung von `height` als Methode von `TreeNode` ........ 43
Traversieren von Bäumen ........ 43
Binäre Bäume ........ 43
Methoden bei Binären Bäumen ........ 43
Grössenverhältnisse bei binären Bäumen mit der Höhe h ........ 44
Implementierung von Binären Bäumen ........ 44
Prioritätswarteschlangen ........ 45
Einfügen in dn Heap ........ 45
Löschen ........ 45
Lineare Datenstruktur vs. Baumartiger Datenstruktur ........ 45
Heapsort läuft in O(nlog2n) ........ 45
ADT Dictionaries (Wörterbücher) ........ 46
Ein Wörterbuch als ADT ........ 46
Einfache Implementierung eines Wörterbuches ........ 46
Binärsuche ........ 46
Suchbäume ........ 47
Pseudocode für Methode `TreeSearch(k,v)` ........ 47
Methode Einfügen `insertItem(k, e)` ........ 47
Methode Enferne Knoten mit Schlüssel k ........ 47
Wörterbücher ........ 47
Beispiel Inverses Telefonbuch ........ 47
AVL-Bäume (nach Adelson-Velski, Landis, 1962) ........ 48
Wiederherstellen der AVL-Struktur textuell ........ 48
Wiederherstellen der AVL-Struktur graphisch ........ 48
Kosten der Rotationen bei AVL-Bäumen ........ 49
ADTs um den Such-Aufwand gering zu halten ........ 50
Wörterbücher ........ 50
Balancierte Suchbäume ........ 50
Hashing ........ 50
Erweiterung Graphentheorie ........ 51
Kürzeste Wege in Graphen (Dijkstra) ........ 51
Algorithmus von Dijkstra ........ 51

# Organisatorisches

## Tutorien

| | | | |
|---|---|---|---|
| Maria: | Mi | 12-15 | R046 |
| Tomasz: | Mo | 12-14 | R055 |
| Tomasz: | Mo | 14-16 | R055 |
| Wolfgang: | Di | 8:30-10 | R055 |
| Adrian H.: | Mi | 12-14 | R119, Arnim-3 |
| Marco B.: | Fr | 10-12 | R053 |

## Test

Mittwoch, 12.6.02 in der Vorlesung

## Klausur

12.7.02, 8:15 Uhr

## Sommertutorium

Inf A und Inf B vom 29.7.-2.8.02 bei Wolfgang Mulzer

# Themenüberblick

Algorithmen / Datenstrukturen
Objekt-orientierte Programmierung
Java
Alles nach Möglichkeit an graphischen Beispielen

# Graphentheorie

## Definition von Graphen

Graphen beschreiben die Beziehung zwischen einer Menge von Objekten (≡ Knoten, *vertex*) und ihrer Beziehung (≡ Kanten, *edges*) zueinander.

Definition: Ein endlicher, ungerichteter Graph ist ein Paar (V,E) mit einer endlichen Knotenmenge V und einer Menge E (= Menge von Knotenpaaren) {u,v} mit u,v ∈ V. Dabei gilt {u,v} = {v,u}.

Definition: Ein endlicher, gerichteter Graph ist ein Paar (V,E) mit einer endlichen Knotenmenge V und einer Menge E (= Menge von Knotenpaaren) {u,v} mit u,v ∈ V. Dabei gilt nicht {u,v} = {v,u} !

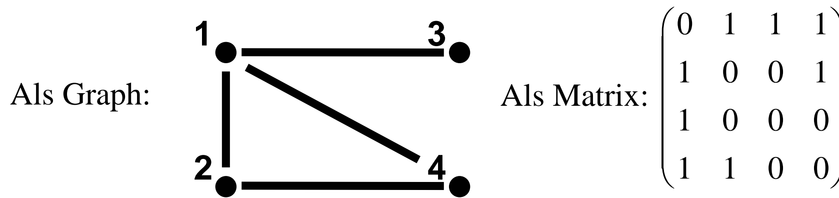
Kanten {u,u} heißen Schleifen (*loops*).
Mehrfachkanten: Mehrere Kanten zwischen einem Knotenpaar
Schlichter Graph: Keine Mehrfachkanten, keine Loops

## Darstellung von Graphen

| | |
|---|---|
| 1. Graphische Darstellung | Knoten = Punkte einer Ebene<br>Kanten = Verbindende Kurven |
| 2. Adjazenz-Listen | Eine Adjazenzliste ist die Liste der Nachbarknoten für einen gegebenen Knoten; die Gesamtheit er Adjazentlisten ist eine Graphen-Darstellung. Bei gerichteten Graphen werden nur die Nachbarn, auf die gezeigt wird, aufgenommen. |

3. Adjazenz-Matrix Für n Knoten wird eine n×n-Matrix gebildet. Gibt es einer Verbindung, wird ein i,j-Eintrag auf 1 gesetzt, sonst auf 0.

### Beispiel für einen Graphen mit Adjazenzmatrix-Darstellung

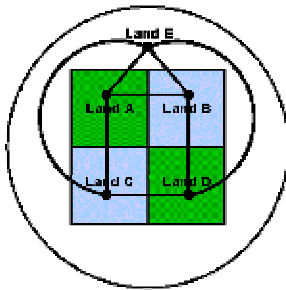



Meist sind Adjazenz-Listen günstiger, da sie nur tatsächlich existierende Kanten verwalten. Adjazenz-Matrizen verwalten auch nicht-existierende Kanten; sie werden als Nullen eingetragen. Die Größe der Adjazenz-Matrix in also in jedem Fall $n^2$.

## *Beispiele für Aufgabenstellungen, die mit Graphen lösbar sind*

### 1. 4-Problem: 4 Farben reichen für jede Landkarte

Die Knoten repräsentieren die Länder, die Kanten die gemeinsamen, nicht-trivialen Grenzen (bedeutet, die Länder berühren sich mehr als nur an einem Punkt). Es konnte 1978 (der erste mathematische Beweis mit notwendiger Computerhilfe!) gezeigt werden, dass 4 Farben in jedem Fall ausreichen (APPEL / HAKEN, 1978).
Hier reichen sogar nur drei Farben aus:

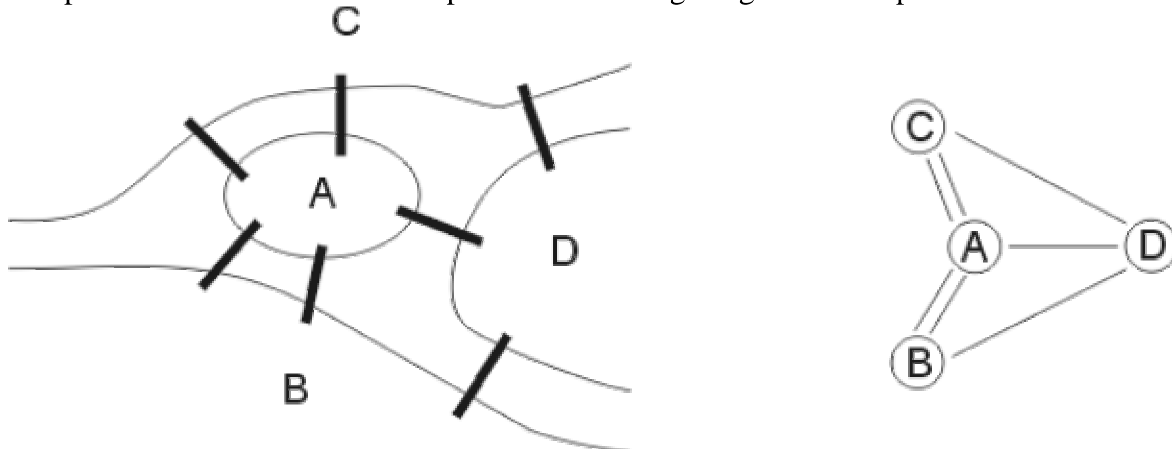



### 2. Welche Graphen treten als Ecken-Kanten-Gerüst von Polyedern auf?

(Anwendung in der Optimierung)

### 3. Euler'sche Graphen

Bedingung für Euler'sche Graphen: Gerade Anzahl von Kanten an jedem Knoten und der Graph muss zusammenhängend sein.
Beispiel für einen Euler'schen Graphen ist das Königsberger Brückenproblem:




Jede Kante soll genau einmal benutzt werden! Das Königsberger Brückenproblem (EULER, 1937) ist nicht lösbar, da die Bedingungen für Euler'sche Graphen (gerade Anzahl Kanten bei Knoten D nicht vorhanden!) nicht erfüllt sind.

## 4. Hamilton'sche Graphen

Hier soll jeder Knoten genau einmal durchlaufen werden. Die Entscheidung, ob ein Hamilton'scher Graph vorliegt, ist ein NP-vollständiges Problem (###).

## 5. Das TSP-Problem

Untermenge der Hamilton'schen Graphen. Der kürzeste Hamilton'sche Graph soll gefunden werden. Dazu werden den einzelnen Kanten Kosten zugeordnet, die es zu minimieren gilt.

## 6. Planare Graphen

Welche Graphen lassen sich in der Ebene mit sich nicht überschneidenden Kurven zeichnen?
Satz von Kuratowski: Ein Graph G ist planar genau dann, wenn er keine Unterteilungen des $K_5$ oder des $K_{3,3}$ als Teilgraphen enthält (KURATOWSKI, 1930)

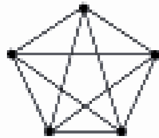
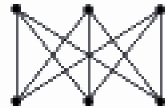



Illustration zu Kuratowski: Graphen mit $K_5$ und $K_{3,3}$

## 7. Computer-Netzwerke

Finden der kürzesten Verbindungen und Schaltnetze.
Definition und wichtige Graphen
G = (V,E)

Definition: Grad eines Knotens $v \in V$ ist die Anzahl $deg_G(v)$ seiner Nachbarn

Inzident: Kante und Knoten stoßen zusammen
Adjazent: Miteinander verbundene Knoten

Definition: Gerichteter Graph

$indeg_G(v)$ = Anzahl der in v ankommenden Kanten
$outdeg_G(v)$ = Anzahl der in v startenden Kanten

*Satz: (Handschlag-Lemma):*
G = (V,E) ungerichtet: $\sum_{v \in V} \deg v = 2|E|$

(Summe der Grade im Graphen, Faktor 2 kommt daher, dass beide Knoten gezählt werden).
Korollar aus Handschlag-Lemma: In jedem ungerichteten Graphen ist die Anzahl der Knoten mit ungeradem Grad gerade.
Beweis:

$$\sum_{v \in V} \deg v = \sum_{v \in V(gerade)} \deg v + \sum_{v \in V(ungerade)} \deg v = 2|E|$$

$\sum_{v \in V(gerade)} \deg v$ muss gerade sein, da es eine Summe gerader Zahlen ist. Das Ergebnis ist auch gerade, da Faktor 2 darin enthalten ist. Eine gerade Zahl kann nur entweder aus einer Summe von zwei geraden Zahlen oder aus einer Summe von zwei ungeraden Zahlen entstehen.

## Teilmengen von Graphen

### Untergraph

Definition: $G'=(V',E')$ ist Untergraph von $G=(V,E)$ falls $V'\subseteq V$ und $E'\subseteq E$.

Definition: Der Graph G' ist induzierter Untergraph von G, falls $V'\subseteq V$, $E'\subseteq E$ und $E'=E\cap\{\{u,v\}\mid u,v\in V\}$
(induziert bedeutet, das alle Kanten der Knoten im Untergraph, die im "Obergraph" vorkommen, auch im Untergraph vorkommen müssen)

### Komplementärer Graph

Komplementärer Graph: Alle Kanten werden zu Nicht-Kanten, alle Nicht-Kanten werden zu Kanten.

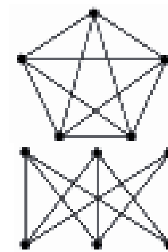
Definition: Das Komplement von $G=(V,E)$ ist $G^C=(V,E^C)$ mit $\{u,v\}\in E \Leftrightarrow \{u,v\}\notin E^C$.

### Isomorphe Graphen

Isomorphe Graphen: Gleich aussehende Graphen (Die Nummerierung der Knoten ist egal)

Definition: $G=(V,E)$ und $G'=(V',E')$ heißen isomorph, wenn eine Bijektion existiert (d. h. sie können ineinander umgewandelt werden) $f:V\to V'$ und $\{u,v\}\in E \Leftrightarrow \{\varphi(u),\varphi(v)\}\in E'$

## Beispiele für Graphen



1. Vollständiger Graph $K_n$, $n\geq 1$
   (alle knoten sind mit jedem knoten verbunden)
2. Vollständige, bipartite Graphen $K_{n,m}$, $n,m\geq 1$
   Untergraphen von vollständigen, bipartiten Graphen heißen selbst bipartit.
3. Der Hyperwürfel $Q_N$ hat folgende Knotenmenge:
   Knoten: alle 0-1 Strings der Länge n
   Kanten: genau dann, wenn der Hamming-Abstand zwischen den Knoten 1 ist (heißt, sie unterscheiden sich in einem einzigen Bit). Die Anzahl der Knoten ergibt sich aus der Länge n des Strings, also $2^n$ Knoten, daraus folgt $2^n \bullet n \bullet 2|E|$ $\quad |E|=n\bullet 2^{n-1}$ Kanten
4. Einfacher Weg: $P_n$, $n\geq 0$hat n+1 Knoten
5. Kreis $C_n$ Weg: $P_{n-1}$ + Kante$\{v_n,v_0\}$
6. G ist k-regulär: Alle Knoten haben Grad k (ein Kreis wäre also 2-regulär, vollständige Graphen sind n-1 – regulär
7. Bäume: Zusammenhängende Graphen ohne Kreise

## Zusammenhang und Abstand in ungerichteten Graphen

Definition: $G=(V,E)$ mit $u,v\in V$

v ist von u erreichbar, wenn es einen Weg gibt, der Untergraph von G ist und u mit v verbindet.

Die Erreichbarkeit ist eine binäre Äquivalenz-Relation:

1. reflexiv [jeder Knoten ist von sich selbst aus erreichbar]
2. symmetrisch [wenn u von v erreichbar ist, dann ist auch v von u erreichbar]
3. transitiv [wenn u von v erreichbar ist und w von v erreichbar ist, dann ist w auch von u erreichbar]

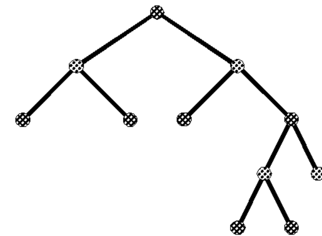
Die Erreichbarkeitsrelation zerlegt V in sogenannte Äquivalenz-Klassen, diese heißen hier Zusammenhangskomponenten des Graphen.
Wege bestehen aus Kanten…

Definition: G heißt zusammenhängend, wenn es nur eine Äquivalenzklasse im Graphen gibt.

Definition: G=(V,E)

Der Abstand $d_G(u,v)$ = Länge (entspricht Anzahl der Kanten) eines kürzesten Weges zwischen v und u. Ist ein Knoten nicht erreichbar, dann gilt $d_G(u,v) = \infty$.

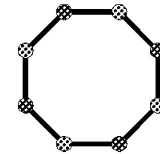
### Charakterisierung bipartiter Graphen



Satz: G ist bipartit, genau dann wenn alle Kreise, die Untergraphen sind, gerade Längen haben. Bäume müssten demnach bipartit sein

Beweis: G ist zusammenhängend

G=(V,E), V=A …. Kanten nur zwischen A und B
Sei C ein bipartiter Kreis in G, C benutzt A und B abwechselnd, also Anzahl der Knoten gerade und gerade Anzahl von Kanten in diesem Kreis (gerade Länge)



Fixieren eines beliebigen Knotens $u \in V, A = \{v \in V \mid d(u,v)\, gerade\}$ (alle Knoten werden darein gesetzt, die geraden Abstand zum fixierten Knoten haben. B = alle anderen Knoten)

Zu Zeigen: Innerhalb von A bzw. B dürfen keine Kanten sein

Indirekter Beweis:
Annahme: In B gibt es eine Kante {v,w}
Behauptung: Der Abstand d(u,v) = d(u,w) (v,w ∈ Knotenmenge von B, u ∈ Knotenmenge von A) sowohl d(u,v) und d(u,w) müssen ungerade sein. zeichnung 9.

Definition: Ein Baum ist ein ungerichteter, zusammenhängender Graph ohne Kreise.

Definition: Ein ungerichteter Graph heißt Wald, wenn alle Zusammenhangskomponenten Bäume sind.

Satz: G = $(V,E)_{zusammenhängend}$, wenn

1. keine Kreise
2. zwischen 2 Knoten nur 1 Weg
3. |E|=|V|-1

# Bäume

Definition: Ein zusammenhängender, ungerichteter Graph ohne Kreise heißt Baum.

Definition: Ein Wald ist ein Graph, bei dem alle Untergraphen Bäume sind.

Definition: Wenn $G = (V,E)$ ein ungerichteter Graph ist, dann heißt $T = (V,E')$ (ein Baum auf V) aufspannender Baum für G

Satz: Folgende Bedingungen sind äquivalent:

(1) G ist ein Baum, $G=(V,E)$

(2) Je zwei Knoten sind durch genau 1 Weg verbunden

(3) $|V|=|E|+1$

Beweis (indirekt):
(1)=>(2) [es seien u und v mit zwei wegen verbunden, dann ist dort ein Kreis => verboten wegen der Definition eines Baums
(2)=>(1) [angenommen, G sei kein Baum, also mit Kreis, dann finde ich immer zwei verschiedene Wege, die die beiden Knoten miteinander verbinden
(1)=>(3) Knoten von Grad 1 heißen Blätter
Zeigen, dass es Blätter gibt:

Betrachten eines längsten Weges in G: Alle Nachbarn von $u_1$ sind auf dem Weg, nur $u_2$ ist Nachbar, da es sonst einen Kreis gäbe. Entferne Blatt und die inzidente Kante, dann gibt es einen neuen Baum, bis ein einzelner Punkt am Ende steht
(3)=>(1) Sei $T=(V,E')$ der aufspannende Baum fur $G$. Also $|V|-|E'|=1$
Außerdem $E' \leq E$ und $|V|-|E|=1$. Daraus folgt: $E'=E$.

## *Durchmustern eines Baums*

### Einschub Queue

Ein Queue ist eine Datenstruktur mit einer linearen Ordnung, bei der neue Element hinten angesetzt werden und vom Kopf an entfernt/abgearbeitet werden (FIFO).
Methoden: **`enqueue`** ('rin in Schlange), **`dequeue`** ('raus aus Schlange), **`isEmpty`** (is' die Schlange leer?)

### Durchmustern eines Baums: Breitensuche (BFS; broad first search)

Ziel: Jeder Knoten des Graphen soll einmal systematisch vom Startknoten s aus durchmustert werden. Der Graph sei in Adjazenzlisten gegeben.
An der Position *s* werden alle benachbarten Knoten in eine Warteschlange (Queue) gesteckt. Dann gehe man zum ersten Knoten *u* der Warteschlange und nehme alle Nachbarn von *u* in die Warteschlange auf (sofern sie noch nicht in der Warteschlange sind). Nachdem der Knoten abgearbeitet, werden sie aus der Warteschlange entfernt. Dadurch wird gleichzeitig ein Baum in einem unbekannten Graphen aufgespannt, das Ergebnis ist in jedem Fall ein Baum. Der Algorithmus liefert gleichzeitig die Abstände und die kürzesten Wege vom Startknoten *s* gratis [Damit könnte man eine Distanzmatrix füllen].
Weiterhin enthalten kürzeste Wege wiederum die kürzesten Wege. Also: Die Kodierung von kürzesten Wegen findet durch Abspeichern des Vorgängers von jedem Knoten auf dem kürzesten Weg. Das spart das Abspeichern des gesamten Weges für jeden Knoten.

### Idee

Knoten haben 3 verschiedene Zustände:

- Noch nicht entdeckt (in der Farbe weiß)
- Knoten entdeckt (Farbe grau), diese werden in der Warteschlange verwaltet
- Knoten abgearbeitet (Farbe schwarz)

Pseudo-Code zu Breitensuche (BSF) für einen Graphen G mit dem Startpunkt s):

```
Comment  01-04Initialisierung fuer die Knoten ungleich s
01  for jede Ecke u∈V(G)\{s}
02     do Farbe[u]← weiß
03         d[u] ← unendlich
04         pi[u] ← NIL
Comment  05-08 Initialisierung fuer s und Warteschlange Q
05  Farbe[s] ← grau
06  d[s]=0
07  Q ← {s}
Comment  Eigentliches Durchgehen durch den Baum
08  pi[s] ← NIL
Comment pi(Knoten) enthaelt den Vorgaenger auf
Comment dem kuerzesten Weg von s nach u
09  while Q ≠ 0
10     do u ← Kopf(Q)
11         for jeden Nachbarn v∈Adj[u]
12            do if Farbe(v)=weiß
13                then Farbe(v) ← grau
14                     d[v] ← d[u]+1
15                     pi[v] ← u
16                     Setze v ans Ende der Warteschlange
17                Entferne Kopf aus Q
18                Farbe(u) ← schwarz
```

Finden des kürzesten Wegs von *u* nach *s*: Finde den Vorgänger pi(*u*), finde den Vorgänger pi(pi(*u*)) etc., bis pi(…)=*s* (dies ist ein Beispiel für einen Greedy-Algorithmus [Zwischenlösungen erweitern sich zu einer vollständigen Lösung und werden nie mehr verändert. Die Lösung baut sich "gierig" auf]).

### Durchmustern eines Baums: Tiefensuche (DFS; deep first search)

Start in s, gehen zum ersten weißen Knoten, weiter gehen zum nächsten weißen Knoten (also kein systematisches abarbeiten aller weißen Knoten). LIFO-Verfahren. Es muss noch eine globale Zeituhr, die indiziert, _wann_ der Knoten entdeckt wurde, mit verwaltet werden. **d(u)** ist Entdeckungszeit und **f(u)** ist Zeit des Schwarzwerdens. Der Knoten v komme nach dem Knoten u, dann weiß ich **d(v)=d(u)+1** und **f(v)=f(u)-1** (da v auf dem Rückweg nach u vorher durchlaufen wird).

Pseudo-Code zu Tiefensuche (DSF) für einen Graphen G mit dem Startpunkt s):

```
Comment d(u) ist Entdeckungszeit
Comment f(u) ist Zeit des Schwarzwerdens

Comment Alle Knoten weiss setzen
01  for jeden Knoten u∈V(G)
02     do Farbe(u) ← weiss

Comment Zeit auf Null setzen
03  Zeit ← 0

Comment Besuchen aller Knoten und registrieren,
Comment wenn sie weiss sind
04  for jeden Knoten u∈V(G)
05     do if Farbe(u)=weiss
```

```
06    DFS-visit(u)
Comment DFS-visit ist eine Funktion, die die
Comment Aktion beim In-die-Tiefe-gehen beschreibt

function DFS-visit
  Comment Was zu tun ist, wenn der Knoten besucht ist
  01  Farbe(u) ← grau
  Comment Updaten des Zeiteintrages fuer die
  Comment Entdeckung des Knotens
  02  d(u)← Zeit ← Zeit+1
  Comment Rekursiver Aufruf fuer die Nachbarn des
  Comment aktuellen Knotens
  03  for jeden Knoten v∈Adj(u)
  04    do if Farbe(u)=weiss
  05       then pi(v)=u
  06         DFS-visit(v)
  07  Farbe(u) ← schwarz
  08  f(u) ← Zeit ← Zeit+1
```

zeichnung

Das Fette sind die Baum-Kanten (*T*) von Knoten-grau nach Knoten-weiß. Die Back-Kanten (*B*) gehen von Knoten-grau nach Knoten-grau (in der Zeichnung die einzige von 3,4 nach 1,12). Die Forward-Kanten gehen von Knoten-grau nach Knoten-schwarz (von Vorgänger zu Nachfolger im Baum in der Zeichnung die einzige von 1,12 nach 5,6). Die restlichen Kanten heißen Cross-Kanten (alle anderen Kanten).

### Methoden zu DFS:

Hauptmethoden: **push** (ein Objekt wird oben auf den Stapel gelegt)
**pop** (oberstes Element entfernen)
Nebenmethoden: **size** (wie viele Elemente hat der Stapel)
**isEmpty** (Boole-Wert)
**top** (gibt das oberste Element, ohne es zu entfernen)

## *DAGs (Directed, Acyclic Graphs; Gerichtete, azyklische Graphen)*

### Definition azyklisch: Keine gerichteten Kreise

Aufgabe: Finde topologische Sortierung eines DAG d. h. Nummerierung der Knoten, so dass $(u,v) \in E$, dann $u < v$ in der Nummerierung steht.

### Verfahren 1 mit brute-force

Suche Knoten, von dem nur Kanten ausgehen (Start in irgendeinem Knoten; Prüfen, ob ein Knoten nur ausgehende Kanten hat [wenn ja, dann ist das der Knoten 1]; den nächsten Knoten gegen den Strich suchen, rekursiv)

### Verfahren 2 mit DFS

Immer, wenn Knoten schwarz wird, ausgeben, das ergibt eine inverse topologische Sortierung.

Zu Zeigen: $(u,v) \in E \Rightarrow f(v) < f(u)$

Beweis: Wenn die Kante (u,v) von DFS untersucht wird, ist u grau, aber v ist nicht grau (zu zeigen). Wenn v weiß: $f(v) < f(u)$, v ist unter u. Wenn v schwarz ist: $f(v) < f(u)$ ???.

Der Knoten v ist nicht grau, da:

Fakt: G ist DAG ⇔ DFS produziert keine Back-Kanten.

Beweis:
Fall 1: => ist klar ???
Fall 2:<=

Angenommen, G hat einen gerichteten Zyklus
Habe v auf Zyklus mit geringster d(v), dann sei u der Vorgänger von v
Wenn v als erstes grau wird, sind alle anderen noch weiß.
Wenn es dann bei u angekommen ist, dann ist v noch grau und u auch grau.
Eine Kante zwischen grau und grau heißt Back-Kante, und die darf nicht sein nach Definition von DAG.

Anwendung z. B. in der Computer-Graphik: Welche Objekte kann ich von einem bestimmten Standpunkt aus sehen. Dadurch kann die Rendering-Reihenfolge bestimmt werden. Die "hintersten" Objekte werden zuerst gemalt.
Anwendungen für FIFO (BFS) sind Abstände berechnen, Anwendungen für LIFO (DFS) ist topologisches Sortieren (z. B. im Job Scheduling, Projektmanagement oder Komponieren der grafischen Ausgabe [painters algorithm]).

## *Minimal ausspannende Bäume*

Minimal ausspannende Bäume = minimum spanning tree (MST)
$G=(V,E)$ ungerichtet, zusammenhängend, die Kanten haben verschiedene Längen durch eine Gewichtungsfunktion $w: E \to \mathbb{R}$.
Aufgabe: Suche aufspannenden Baum $T=(V,E')$, $E' \subseteq E$ mit minimalem Gewicht $w(T)=\sum_{e \in E'} w(e)$.

### Generischer MST-Algorithmus

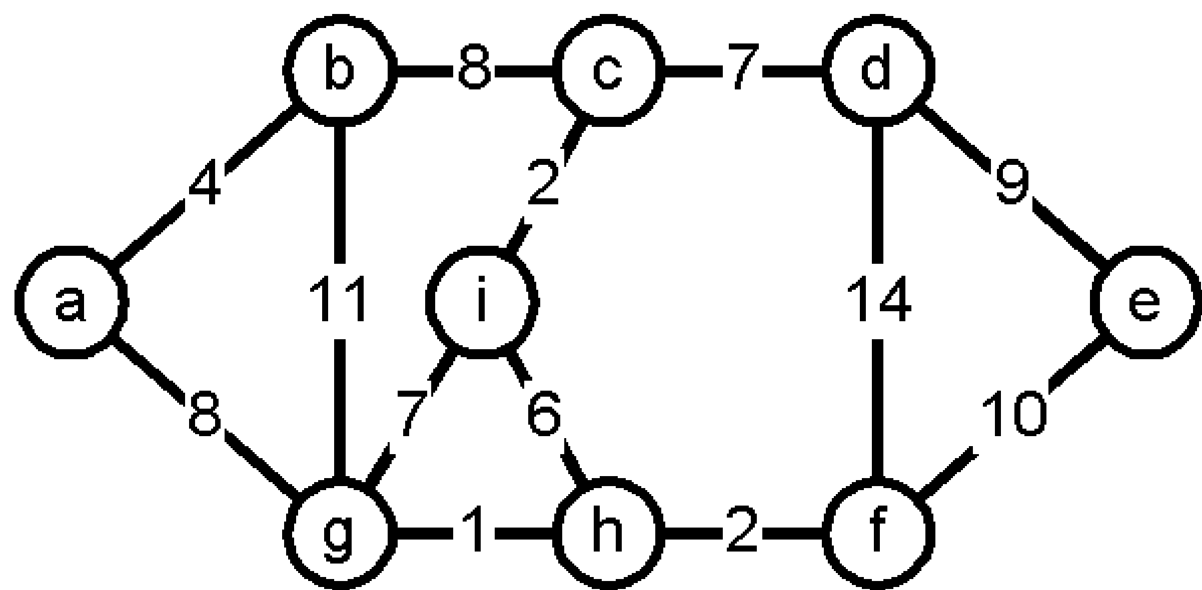
Definition: Schnitt von G ist Zerlegung der Knotenmenge (S,V\S), d. h. der Graph wird zweigeteilt.

Die Kantenmenge A respektiert Schnitt (S,V\S) falls keine Kante aus A Knoten von S mit Knoten aus V\S verbindet (also keine Kante von dem einen Teil des zerschnittenen Graphen in den anderen).
Die Kantenmenge $A \subseteq E$ heißt sicher, wenn es MST $T=(V,E')$ gibt mit $A \subseteq E'$.
G ist ungerichtet, zusammenhängend $w: E \to R$
A ist eine sichere Kantenmenge, die einen Schnitt (S,V\S) respektiert.
Sei $(u,v)$ eine leichteste Kante mit $u \in S$, $v \in V \setminus S$.

Behauptung: $A \cup \{(u,v)\}$ ist sicher.

Beweis: Sei T ein MST, der A enthält. Annahme: $(u,v) \notin T$. Die Hinzunahme von $(u,v)$ zu T erzeugt den Kreis C -> wir betrachten in C alle Kanten (x,y) mit x elem S, y elem V\S -> es gibt wenigstens eine solche Kante (u',v'). Noch Voraussetzung: $w(u,v) \le w(u',v')$
Streichen von (u',v') aus $T \cup \{(u,v)\}$ -> das ist ein MST.

### Beispiel-Graph




### MST Algorithmus von Prim

#### Idee

Lassen Baum von Startknoten aus wachsen

#### Datenstruktur

Prioritätswarteschlange verwalten alle noch nicht erreichten Knoten w zusammen mit Schlüssel key(w), der angibt, was das leichteste Gewicht einer Kante von einem bereits erreichten Knoten zu w ist.

#### Funktionalitäten

Verwalten einer Menge S, x ∈ S, key(x):

- Insert (S,x)
- Minimum(S)
- Extract_Min(S)
- Pi-Zeiger zur Darstellung des Baumes

#### Pseudocode MST-Prim (G,w,r)

*r* sei Startknoten

```
01  Q ← V(G)
02  for jedes u ∈ Q
03    key(u) ← ∞
04  key(r) ← 0
05  pi(r) ← NIL
06  while Q≠{}
07    do u ← Extract_Min(Q)
08    for jedes v ∈ Adj(u)
09      do if v ∈ Q und w(u,v)<key(v)
10         then pi(v) ← u, key(v) ← w(u,v)
```

### MST-Algorithmus von Kruskal

Zuerst werden die Kanten aufsteigend nach dem Gewicht geordnet. Wenn es für die gerade betrachtete Kante einen Schnitt gibt, dann wird diese Kante zur Lösung hinzu genommen.

#### Idee

- Sortiere Kanten nach steigendem Gewicht
- Prüfe nacheinander, ob die Kante die leichteste über einen Schnitt ist, dann wird sie aufgenommen

Im Allgemeinen sind die Zwischenlösungen Wälder, die zu einem Baum zusammenwachsen (greedy algorithm). Das Problem ist jetzt, sicher zu stellen, wie man einen Kreis verhindert. Das wird über die Datenstruktur Union-Find gelöst. Anfänglich stehen alle Knoten allein, wenn sich die ersten Teilbäume bilden, werden sie zusammengefasst. Dann wird ein Knoten als "Chef" bestimmt. Dies wird jedem Knoten des Teilbaums mitgeteilt. Bei jedem neuen Knotenpaar, das geprüft wird, wird verglichen, ob die Knoten den gleichen Chef haben. Wenn nein, dann kann die Kante gezogen werden.

#### Funktionalitäten

- MakeSet(v) Aus $v \in V$ wird eine Menge gemacht
- Union(u,v) Vereinigt die beiden Mengen, zu denen die Knoten $u$ und $v$ gehören.
- FindSet(v) Bestimme die Menge, zu der $v$ gehört (wer ist der "Chef"?)

#### Pseudocode

```
01  A ← ∅
02  for jedes v ∈ V
03    do MakeSet(v)
04  Sortiere Kanten in steigender Reihenfolge
05  for jede Kante(u,v) in sortierter Reihenfolge
06    do if FindSet(u) ≠ FindSet(v)
07      then A ← A ∪ {(u,v)}
08        Union(u,v)
09  return A
```

## Laufzeitabschätzung

Soll Rechner-unabhängig getan werden. Dazu muss die Elementaroperation (z. B. Addition, Bitoperation etc.) festgelegt werden.
Weiterhin kann die Laufzeit abhängig von der Eingabe bestimmt werden:

1. Beste Laufzeit
2. Mittlere Laufzeit (=Wahrscheinlichste Laufzeit)
3. Schlechteste Laufzeit

### *Betrachtung des Worst Case*

worstcase=max{T(|I|) | I Instanz der Größen}
T(n): Konkrete Laufzeit
I: Instanz der (Eingabe-)Größen

#### Ziel

T(n) soll möglichst gut durch untere und obere Schranken angenähert werden, dabei sollen die Konstanten vernachlässigt werden. Dies entspricht dem Wachstum von T(n) bei $n \to \infty$. Der Streifen, der durch die Funktion für die obere Schranke der Laufzeit und die Funktion für die untere Schranke gegeben ist, soll möglichst schmal werden. Dabei muss T(n) oft nur für einen Eingabebereich von n ermittelt werden.

## Wachstum von Funktionen und die $\mathcal{O}$-Notation

Definition: $f, g : \mathbb{N} \to \mathbb{R}^+$

(a) $f(n) \in \Theta(g(n))$ $f$ und $g$ haben dasselbe asymptotische Wachstum

$\exists$ const. $c_1, c_2 > 0$, $\exists n_0 \in \mathbb{N}$

$\forall n > n$: $c_1(g(n)) \leq f(n) \leq c_2(g(n))$

(b) $f(n) \in \mathcal{O}(g(n))$

$\exists c > 0, \exists n_0 : \forall n > n_0, f(n) \leq c \cdot g(n)$

(c) $f(n) \in \Omega(g(n))$

$\exists c > o, \exists n_0 : \forall n > n_0, f(n) \geq c \cdot g(n)$

Anmerkung:

1. Schreiben oft $f(n) = \mathcal{O}(g(n))$ statt $f(n) \in \mathcal{O}(g(n))$, wie es eigentlich korrekt wäre
2. Es gibt Funktionen, die unvergleichbar sind, bezüglich O-Notation
   $f(n) = n, g(n) = n^{1+\sin(n)}$
3. O-Notation immer kritisch hinterfragen (Konstantengröße!)

## Laufzeitanalyse der $\mathcal{O}$-Notation

Konkreter Rechner und konkrete Implementation
Idealisierter Rechner (Turing-Maschine, RAM)
Von Konstanten abstrahieren (o-Notation) (prinzipielles Wachstum von Funktionen

$f, g : \mathbb{N} \to \mathbb{R}^+$

$f(n) = \Theta(g(n))$

$\exists c_1, c_2 > 0$, $\exists n_0 \forall n \geq n_0$

$$\underbrace{c_1 g(n) \leq f(n)}_{f(n)=\Omega(g(n))}, \underbrace{f(n) \leq c_2 g(n)}_{f(n)=\mathcal{O}(g(n))}$$

### Beispiel für eine Laufzeitanalyse

1) $2^{100} = \mathcal{O}(1)$ weil $2^{100} =< 2^{100} * 1$; $(n+1)^2 = n^2 + 2n + 1$

2) Für beliebige $a, b \in \mathbb{R}$, $b > 0$

$f(n) = (n+a)^b$, $f(n) = \Theta(n^b)$

### Zeigen des Ergebnisses der Kaufzeitanalyse

$f(n) = \mathcal{O}(n^b)$, Brauchen $c_2$, $n_0$

$(n+a)^b \leq c_2 \cdot n^b$ für $n \geq n_0$

Wir setzen $c_2 = 2^b$, die Ungleichung ist erfüllt, wenn $n + a \leq 2n$

FEHLENDES !!! ???

## Laufzeit einiger wichtiger Funktionen

| | |
|---|---|
| 1 | Konstanter Aufwand |
| $\log_2 n$ | Teile und Herrsche (z. B. binäre Suche) |
| $\sqrt{n}$ | (z. B. Separatoren in planaren Graphen ???) |
| n | Jede Stelle einer Eingabe auslesen |
| $n^2$ | Insertion Sort |
| $2^n$ | Untersuchung aller Teilmengen einer Eingabe (brute force) |
| n! | Untersuchung aller Permutationen der Eingabe |
| Summen | Hintereinanderausführung |
| Produkte | Verschachtelte Schleifen |

**Definition:** $f(n) = o(g(n))$

o ist hier ein kleines o !

$g(n)$ ist obere Schranke, aber wächst viel schneller als $f(n)$

$\Leftrightarrow \forall c > o \exists n_0 \forall n \geq n_0,\ f(n) \leq c \cdot g(n)$

$f(n) = \omega(g(n))$

$\Leftrightarrow \forall c > o \exists n_0 \forall n \geq n_0$

$c \cdot g(n) \leq f(n)$

Beispiel:

$f(n) = 12n^2 + bn$

$f(n) = o(n^3)$

Merke: Von links nach rechts in o-Relation: 1, log(n), $\sqrt{n}$, n, $n^2$, $n^3$, $2^n$, n!

**Kochrezepte**

1) $$\begin{matrix} d(n) = \mathcal{O}(f(n)) \\ e(n) = \mathcal{O}(g(n)) \end{matrix} \Rightarrow \begin{matrix} d(n) \cdot e(n) = \mathcal{O}(f(n) \cdot g(n)) \\ d(n) + e(n) = \mathcal{O}(f(n) + g(n)) \end{matrix}$$

2) Gleiches Wachstum bleibt bei Logarithmierung erhalten

$f(n) = \Theta(g(n)) \Rightarrow \log_2 f(n) = \Theta(g(n))$

Aber: $$\begin{matrix} f(n) = o(g(n)) \\ \sqrt{n} = o(n) \end{matrix} \not\Rightarrow \begin{matrix} \log_2 f(n) = o(\log_2 g(n)) \\ \log(\sqrt{n}) = \frac{1}{2}\log(n) = \mathcal{O}(\log(n)) \end{matrix}$$

3) Wachstum bei Standardfunktionen

Seien 0<a<b

- $n^a = o(n^b)$ "höhere Potenzen wachsen schneller"
- $(\log_2(n))^b = o(n^a)$ "auch für sehr große b und kleine a.
- $n^b = o(2^{a \cdot n})$ "Exponentialfunktion wächst schneller als Polynome"

4) Stirling-Formel: $\sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n \leq n! \leq \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^{n - \frac{1}{12}n}$

also $2^n = o(n!)$ und $n! = o(n^n)$

- bei Analyse von Sortierverfahren brauchen wir $\log_2 n! = \Theta(n \log_2 n)$ (Aufwand für das bestmögliche Sortierverfahren)

  $\left(\frac{n}{2}\right)^{\frac{n}{2}} < n! < n^n$

**Beispiele für den Aufwand verschiedener Algorithmen für das gleiche Problem**

Beispiel: Asymptotische Laufzeitbestimmung von Algorithmus

Gegeben: Array X der Länge $n$

$X[0], \ldots, X[n-1]$ Einträge (hier: Zahlen)

Gesucht: Berechne Durchschnitte der Präfixe

$$A[i] = \frac{\sum_{j=0}^{i} X[j]}{i+1}$$

Algorithmus 1: Präfix_Durchschnitt 1 (X)

```
for i←0 to n-1
  a←0
    for j←0 to I do
      a←a+X[j]
      A[i]←a/(i+1)
return A
```

Problem dieser Implementierung

- Initialisierung + Return A $\mathcal{O}$(n). Wir fangen jedesmal neu mit dem Durchzählen an.
- 2 geschachtelte Schleifen mit Zählern $i$ und $j$
- $n$ Mal Zuweisung `a←0, A[i]←a/(i+1)`
- Erhöhen und Testen von $i$

Innere Schleife insgesamt: 1+2+3+…+n entspricht $\mathcal{O}$(n)

Dann: $\binom{2}{n} = \frac{n(n+1)}{2}$ $\mathcal{O}\binom{2}{n} = \mathcal{O}\left(\frac{n(n+1)}{2}\right)$ also insgesamt: $\mathcal{O}\left(n^2\right)$

Algorithmus 2: Präfix_Durchschnitt 2 (X)

```
s←0
for i←0 to n-1 do
  s←s+X[i]
  A[i] ←s/i+1
return A
```

- A initialisieren / ausgeben -> $\mathcal{O}(n)$
- Pro Schleife 2 Anweisungen + Updates von $i$ -> $\frac{\mathcal{O}(n)}{\mathcal{O}(n)}$

Insgesamt Aufwand $\mathcal{O}(n)$

# Funktionale versus Imperative Programmierung

Algorithmenbegriff: Algorithmus ist eine eindeutige Beschreibung eines in mehreren Schritten durchgeführten Datenbearbeitungsvorgangs.

Weitere Begriffe: Endliche Beschreibung, determiniert, terminieren

### *Beispiel ggT(x,y)*

1. Falls x=y dann ggT(xy)=x
2. Sonst
3. falss x>y ersetze x durch x-y
4. sonst ersetze y durch y-x
5. und weiter mit 1

Es gilt $x, y \in \mathbb{N}$.

ggT(x,y) ist größte ganze Zahl die x und y teilt (das ist kein Algorithmus!)

### *Definition Progamm*

Formulierung eines Algorithmus und einer Datenstruktur in einer konkreten Programmiersprache. Diese muss syntaktisch (welche Zeichenfolgen sind gültige Programme/Befehle) und semantisch (was macht Sinn, ist es eindeutig?) korrekt sein. Compiler finden nur syntaktische, nicht aber semantische Fehler ("Computer sind dumm")!

## Welche Sprache versteht der Rechner?

Rechner erkennt und bearbeitet nur Instruktionen in seiner Maschinensprache (durch die konkrete Hardware definierter Satz von Binärcodes für Elementaroperationen)

### *Assembler*

Assemblersprachen: Mnemonische Darstellung solcher Instruktionen
Die Programmierung auf Maschinenebene ist nicht wirklich gut nachvollziehbar, sie ist umständlich, maschinenbezogen, nicht problembezogen und fehleranfällig. Sie hat aber sehr gute Laufzeiten.

#### Beispiel für Assembler-Code

```
LDE #3    //   Lade Inhalt von Register 3 in Akkumulator
ADD 7     //   Addiere 7 hinzu
JMP LABEL //   Springe zu Label
```

### *Höhere Sprachen*

Höhere Programmiersprachen liefern Programme mit folgenden Eigenschaften:

- abstrakter (problembezogen)
- kürzer
- besser lesbar
- weniger Hardware-abhängig

Compiler/Interpreter erzeugen Maschinensprache aus dem Programm in Hochsprache.

### *Programmparadigmen*

Programmierparadigmen: funktional/applikativ (Haskell), imperativ (C), logisch (Prolog), objektorientiert (Java), parallel (Java, C++)

## Funktionale (applikative) Programmierung

### *Vertreter*

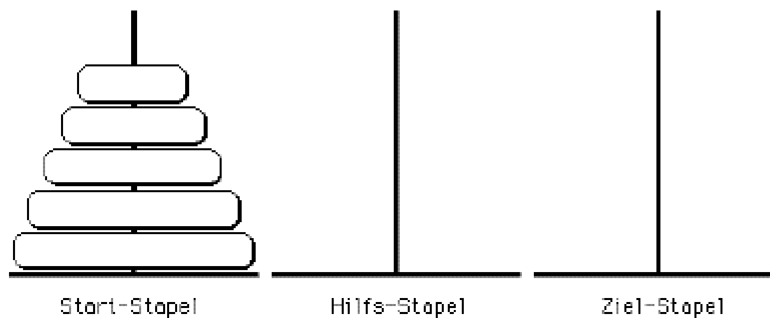
Vertreter: Haskell, Lisp, Scheme

## Eigenschaften Applikativer Algorithmen

Applikative Algorithmen: Definition zusammengesetzter Funktionen durch Terme mit Unbestimmten.
Erst die Termauswertung legt tatsächlich die Reihenfolge der Elementarschritte fest.
Es gibt keine Möglichkeit zur direkten Speicherverwaltung (insbesondere gibt es keine Variablen).
Sie unterstützt Rekursion hervorragend.

## Beispiele für Rekursion

### Türme von Hanoi




Ein Turm ist aus mehreren Klötzen aufgebaut, wobei die jeweils kleineren auf den größeren liegen.
Die Aufgabe besteht darin, den links vorgegebenen Turm rechts neu aufzubauen.
Es darf immer nur ein Turmteil bewegt werden und es dürfen nur kleinere auf größeren Puzzleteilen zu liegen kommen.
Das Spiel wird umso schwerer, je höher der Turm ist.

### Idee

Für n Scheiben

1. Obere n-1 Scheiben nach C mit Hilfe von B
2. Letzte Scheibe von A nach B
3. n-1 Scheibhen von C nach B mit Hilfe von A

### Analyse

$2^n$-1 Bewegungen sind erforderlich

### Implementation in Java

```
/**************************************************************/
/*  Die Türme von Hanoi                          Lizenz: GPL  */
/*                                                            */
/*  (c) 2002  Roland Illig <1illig@informatik.uni-hamburg.de>  */
/**************************************************************/

public class hanoi {

    private static void bewege (char a, char b, char c, int n)
    // Bewegt n Scheiben von Turm a nach Turm c und benutzt als
    // Zwischenspeicher Turm b.
    {
       if (n == 1)
           System.out.println("Lege die oberste Scheibe von " +
                  "Turm " + a + " auf Turm " + c + ".");
```

```
        else {
            bewege(a, c, b, n-1);
            bewege(a, b, c, 1);
            bewege(b, a, c, n-1);
        }
    }

    public static void main (String[] args)
    {
        bewege('a', 'b', 'c', 5);
    }

}
```

### *ggT*

Implementation in Haskell

```
ggt :: Int -> Int -> Int
ggt x y
  | x == y       = x
  | x>y          = ggT (x-y) y
  | otherwise    = ggT x (y-x)
```

Die Semantik ist nicht immer offensichtlich bei Funktionalen Programmiersprachen!

#### Beispiel für nicht-offensichtliche Semantik

```
f :: Int -> Int
f n
  | n>100       = n-10
  | otherwise  = f (f (n+1))
```

### *Probleme bei Rekursionsalgorithmen*

Es ist nicht immer einfach zu entscheiden, ob der Algorithmus terminiert.
Pseudocode:

```
f(x)= if x=1 then 1 else f(g(x))
g(x)= if even(x) then x/2 else 3x+1
```

Ablauf:
Eingabe 1 -> 1
2 -> 1
3 -> f(10) -> f(5) -> f(16) -> f(8) -> f(4) … -> 1
Bis heute ist nicht bekannt, ob der Algorithmus jemals terminiert.
Stichwort Ackermann-Funktionen.

## Imperative Programmierung

### *Vertreter*

Vertreter: C, Pascal, Fortran, Cobol, … , Java

### *Eigenschaften*

Verbreitetste Art, Algorithmen zu formulieren
Basieren auf abstraktem Rechnermodell mit erweiterten sprachlichen Mitteln
Ausführung von Unterprogrammen (Prozeduren)
Sequentielle Abarbeitung (Ausnahme: Sprünge im Programm)

## Basiskonzepte

Variable + Anweisung
Variablen sind Speicherplätze für Werte und bestehen aus Namen (z. B. x) und veränderlichem (sic!) Wert. Der Variable ist immer ein Typ zugeordnet.
t ist ein Term ohne Unbestimmte und w(t) sein Wert.
x:=t ist die Wertzuweisung
Nach Ausführung x=w(t), zu Beginn ist x=⊥ (NIL; leer)

### Zustand eines Rechners

Welche Variablen haben welche Werte
Wertzuweisung ändert Zustand

### Ausdrücke (entspricht Termen bei der Funktionalen Programmierung)

Die Auswertung der Ausdrücke ist zustandsabhängig

### Komplexe Anweisungen

Sequenz: Sind alfa1, alfa2 Anweisungen, so ist die Anweisung (alfa1; alfa2) die Hintereinanderausführung
Selektion: alfa1, alfa2 Anweisungen, B Boolscher Term
if B then alfa1 else alfa2
Iteration: alfa Anweisung, B Boolscher Term
while B do alfa (Schleife)
Wichtig: Iteration ist das Gegenstück zu rekursiven Funktionsaufrufen bei Funktionalen Sprachen.
Programmiersprachen mit obigen Ausdrucksmitteln sind universell (d. h. ALLE Algorithmen lassen sich darin formalisieren)!

## Prinzipielle Aufbau eines Imperativen Programms

```
(nur Typen int,bool)
  <Programmnamen>
  var x,y … : int ; p,q … : bool    // Variablendeklaration
  input x1, … , xn                  // Eingabevariablen
    α1
    α2
    αn
  output y1, … ,ym                  // Ausgabevariablen
```

## Kurzfassung Imperatives Programm

Ausführung eines Imperativen Programms

- Folge von Bassischritten (Wertzuweisungen)
- Diese entsteht mittels Selektion/Iteration basierend auf Booleschen Tests und ist zustandsabhängig
- Die Semantik ist die Gesamttransformation des Zustands

## Beispiele für Imperative Programmierung

### Fakultätsfunktion

```
var x,y : int
input x;
y:=1; while x>1 do y:=y*x;x:=x+1
output y
```

Mit dieser Realisierung fac(x) wird fac(x>=0) = x!, fac(x<0) = 1

### Fibonaccizahlen

```
var x,a,b,c : int;
input x
a=1; b=1 ;
while x>0 do c:=a+b; a:=b;b:=c;x:=x-1
output a
```

### ggT1

```
var x,y : int ;
input x,y;
while x#y do
  while x> y do x:=x-y
  while y>x do y:=y-x
output x
```

### ggT2

```
var x,y,r:int;
input x,y;
r:=1;
while r#0 do
  r:=x mod y ; x:=y ; y:=r
output x
```

### xyz

```
var w,x,y,z : int ;
input x ;
z:=1,w:=1,y:=1;
while w=<x do
  z:=z+1;w:=w+y+z;y:=y+2;
output z
```

Berechnet was? Nämlich $z = \lfloor \sqrt{x} \rfloor$ (die nach oben offenen, eckigen Klammern bedeuten den ganzzahligen Anteil von $\sqrt{x}$ ).

## *Problem der Korrektheit und der Terminierung von Programmen*

Das Problem, was an diesem Programm behandelt werden soll, ist der Nachweis der Korrektheit und der Terminierung.

### Terminierung der Schleife

Konstruieren einer Folge von Werten $u_0$, $u_1$, … mit

- beim ersten Schleifendurchlauf ist $u_0>0$
- ein Schleifenaufruf berechnet (implizit) $u_{i-1}$ aus ui
- für alle i : $0=<u_{i+1}<u_i$
- $u_{i+1}<0$ heisst schleife verlassen
- Differenz $u_i-u_{i+1}$ ist größer als festes $\delta<0$ (trivial bei ganzen Zahlen)

### Beispiel für ein Problem der Terminierung der Schleife

Am Beispiel xyz: u:=x-w
Haben

1. w=<x zu Beginn der Schleife, also $u_0>=0$
2. Sei u Wert vor Schleifendurchlauf, u' danach

$$u' = x-(w+y+z)-x-w-\underbrace{y}_{positiv!}-2 \not\leq u$$

# Java

## Geschichte

Sun implementiert 1990 die Sprache Oak für interaktives Fernsehen etc. Eine Sprache für Embedded Systems.

### Entwurfsziele

- plattformunabhängig
- Zwischencode soll entstehen und ausgeführt werden durch den Interpreter
- OO
- Orientierung an C und C++

1993 erblüht das WWW mit dem ersten Browser "Mosaic". Damit wurde Oak auf Web-Anwendungen umorientiert und wurde auf Java umgetauft. Manche sagen Java stünde für Just Another Vague Acronym…
1995 gab es die erste Referenzimplementierung und den HotJava-Browser (selbst in Java geschrieben). In diesem Jahr gab es auch die erste Unterstützung von Java-Applets durch Netscape V 2.0 . Dies bedeutete den Durchbruch. Es folgte ein explosionsartiger Erfolg
1998 gibt es das Java2 JDK von Sun kostenfrei. IBM, Oracle und M$ unterstützen die Java-Plattform.

### Links zu Geschichte von Java

http://www.uni-karlsruhe.de/~iam/html/java/GeschichteJava.html (Einfacher Text, no frills)
http://www-vs.informatik.uni-ulm.de/Lehre/VerteilteSysII_WS9900/java/ (Folien einer Einführungsvorlesung)

## Was ist Java?

- Volle Programmiersprache (nicht nur für Internet-Anwendungen)
- Klare, kompakte Sprache (vermeidet die Komplexität von C und C++
- Architekturneutral durch Verwenden eines plattformunabhängigen Zwischencodes (das ist der Bytecode)
- Interpretierte Sprache
- Syntax ist C-ähnlich

## Wie arbeitet Java?

1. Erstellen einer Quellcode-Datei **`name.java`** .
2. Der Compiler **`javac`** überprüft Syntax, Variablendeklaration etc. und erstellt **`name.class`** . Das ist der Bytecode. Die Kompilierung geht mit **`javac name.java`** .
3. Zur Ausführung wird der Bytecode in **`name.class`** durch die Java Virtual Machine (JVM) **`java`** ausgeführt (Syntax: **`java name`**. Das Suffix **`.class`** braucht hier nicht mit angegeben zu werden.). Die JVM ist dann natürlich Plattform-spezifisch.

## Aufbau eines Java-Programms

Das Hauptstrukturierungsmittel eines Java-Programmes sind die Klassen, Kapseln, Mengen von Variablen und Methoden (so heißen in Java Prozeduren/Funktionen).

## *Beispiel für ein Java-Programm: Fibonacci*

`fibonacci.java`:

```
class Fibonacci {
  /* gibt die Fibonacci-Zahlen < 50 aus */
```

mehrzeiliger Kommentar. Hierbei gibt es noch den Kommentar mit `//` , der geht dann bis zum Ende der Zeile

```
public static void main(String[] args){
```

**main** ist immer Name der Hauptmethode. **public**, **static** und **void** sind Modifizierer der Methode. In den Klammern nach der Methode sind die Typen, hier eine Liste von Strings, mit den Argumenten **args** )

```
  /* Variable lo für die niedrigere Zahl */
  int lo = 1 ;
```

**lo** und **hi** sind Variablen mit davor angegebenem Typ.

```
  int hi = 1 ;  // Erste beide Fibonacci-Zahlen
  System.out.println(lo) ;
```

Ausgabe

```
  while (hi<50){
```

die geschweifte Klammer kapselt eine Anweisung(sgruppe)

```
    System.out.println(hi) ;
    hi = lo + hi ;  // neuer hi-Wert
    lo = hi - lo ;  // neuer lo-Wert ist alter hi-Wert
    }
  }
}
```

**Beispiel: Imperative und rekursive Realisiuerng der Fakultätsfunktion in Java**

```
import algaj.IOUtils    /* Klasse mit vereinfachter Aus-
                                              /Eingabe */
public class FacImperative{
  public static int factorial (int x) {
    int y=1 ;
    while (x>1) {y=y*x ; x=x-1}
    return y ;
  }
  public static void main (String[] args){
    int z ;
    System.out.print("Zahl: ") ;
    z=IOUtils.red Int() ;
    System.out.println("Fakultät("+z+")="++factorial(z)) ;
    }
  }
```

```
public class FacRecursive{
  public static int factorial(int x){
    if (x<=1)
      return 1 ;
    else
      return x*factorial(x-1) ;
    }
  }
```

## *Java Syntax*

Datentypen in Java: Strenge Typisierung, jede Variable braucht einen Typ
Deklaration von Variablen:

- Festlegung des Typs
- Initialisierung (explizit durch Wertangabe, oder implizit durch default-Wert)
- Bestimmung der Sichtbarkeit

### Primitive Datentypen (Basistypen)

- numerische Typen für Ganz- und Gleitkomma-Zahlen
- Typ für Character
- Boolscher Typ

### Tabelle für Datentypen

| Datentyp | Größe | Wertebereich | Default-Wert |
|---|---|---|---|
| boolean | 1 Bit | true oder false | false |
| char | 2 Byte | Unicode-Character | "/u0000" (Zahl in Hex) |
| byte | 8 Bit | 8 Bit Integer mit Vorzeichen | 0 |
| short | 16 Bit | 16 Bit Integer mit Vorzeichen | 0 |
| int | 32 Bit | 32 Bit Integer mit Vorzeichen | 0 |
| long | 64 Bit | 64 Bit Integer mit Vorzeichen | 0 |
| float | 32 Bit | 32 Bit Gleitkomma mit Vorzeichen | +0.0f * |
| double | 64 Bit | 64 Bit Gleitkomma mit Vorzeichen | +0.0 * |

*: Java nimmt automatisch double an, wenn kein f hinter 0.0 gesetzt wird.
Die Datentypen float und double sind nach dem IEEE 754-1985-Standard implementiert (http://research.microsoft.com/~hollasch/cgindex/coding/ieeefloat.html)

### 2-Komponenten-Darstellung von Zahlen

$B^n$ $b_{n-1},\ldots, b_0$
$0 \leq \sum b_i 2^i \leq 2^n - 1$. Daraus folgt der Werte-Bereich $-2^n$ bis $2^n$-1, dabei ist n die Anzahl der zu Verfügung stehenden Bytes.

### Syntax für die Namen und Bezeichner für Variablen, Klassen, Methoden

Sie sind frei wählbar, aber

- Keine Schlüsselwörter
- Nicht mit Ziffern beginnen
- Groß-/Kleinschreibung wird beachtet (STARKE Empfehlung, dabei systematisch vorzugehen)
- Literalen sind Konstanten

### Beispiele für die Syntax von Variablendeklarationen

```
int eineVariable=42 ;
float x,y ;
char c1='A', c2='B' ;
```

## *Referenzdatentypen*

Alle Nichtbasistypen sind Referenztypen. D. h. Variablen dieser Typen enthalten Referenzen (Verweise) auf den Speicherort der Daten, nicht auf die Daten selbst! Dabei ist null=Default-Wert.

### Erstes Beispiel Arrays (Felder)

Ein Array ist eine Folge von Variablen ein und desselben Datentyps, der nicht primitiv sein muss. Es wird durch Anhängen von [] an den Datentyp oder den Bezeichner definiert (**`int einFeld[] ;`** oder **`int[] nocheinFeld ;`** .). Matrizen können ebenfalls definiert werden: **`float[][] matrix ;`** . Für die Variablen muss Speicherplatz zu Verfügung gestellt werden (Allokation) mittels der Anweisung **`int[] field=new int[20]`** . Die 20 Einträge sind von 0-19 durchnummeriert (Java fängt IMMER bei 0 mit dem Zählen an). Die Initialisierung mit Literalen hat folgende Syntax: **`int[] feld={3,7,8}`** . Der Zugriff auf die Arraylänge geschieht mit **`feld.length`** .

Wichtig: Bei Referenzvariablen erfolgt der Vergleich (z. B. mittels ==) und Zuweisung (=) auf die Referenzen, es werden also keine Kopien angelegt wie bei den Basistypen.

### Beispiel 1

```
int x=3 ;
int y=x ; //y ist 3
y=y+1 ; y ist 4, x ist 3
```

### Beispiel 2

```
int[] feld1={3,4,8,7} ;
int[] feld2=feld1 // Beide Variablen verweisen auf denselben
Speicherbereich!
feld2[2] // feld1[2] ist dann auch 8
```

### Beispiel 3

Arrays von Arrays sind auch zulässig:

```
int[][] pascalTriangle={{1},{1,1},{1,2,1},{1,3,3,1}} ;
```

## *Kopieren von Feldern*

Es gibt zwei Möglichkeiten. Die erste Möglichkeit ist ein elementweises Kopieren. Eleganter ist die spezielle Methode **arraycopy** (eine sogenannte Dienstmethode):

```
int[] feld1={2,4,8,10} ;
int[] feld2=new int[feld1.length] ;
int pdest=0,psrc=0 ;   // Positionen in Ziel und Quelle
System.arraycopy(feld1,psrc,feld2,pdest,feld1.length) //
Kopiert feld1.length Elemente von psrc nach feld2 in pdest
```

### Zeichenketten

Zeichenketten werden in Form der Klasse **java.lang.String** unterstützt.

Erzeugen:
```
String s1=new String("Carli") ;
String s2="Camurçi" ;
String s3=s2 ;
```

Bei String-Variablen ist es möglich, auch ohne das **new** bei der Erzeugung zu arbeiten.

### Einige Stringmethoden

**int length()** // liefert Länge von Zeichenketten
**char charAt(int idx)** // liefert Char bei Pos idx
**int compareTo(String1 String2)** // vergleicht lexikographisch; liefert -1 bei kleinerem Wert
**int indexOf(char ch)** // Position des ersten Auftretens von **ch**
**int indexOf(String s)**
**String substring(int begin, int end)** // liefert Substring von **begin** bis **end**

Konkatenation: **s3=s1+" und "+s2**

### Beispiel für Umgang mit Zeichenketten

Folgende Routine soll den String "und" durch ein "&" ersetzen:

```
String und="und" ;          // Der String namens und hat den Wert "und"
int pos=s3.indexOf(und) ;
String
s4=s3.substring(0,pos)+"&"+s3.substring(pos+und.length()) ;
System.out.println(s4) ;
```

### Die Klasse java.long.StringBuffer

Sie ist ähnlich wie **String**, aber das Ändern des **Strings** ist leichter möglich (Einfügen, Anhängen,…).
Einfache Konvertierung zur Klasse **String** .

## *Operatoren und Kontrollfluss bei Java*

In Ausdrücken werden die Variablen bzw. Literalen durch Operatoren verbunden.

### Anweisungsoperator =

**<Variable>=<Ausdruck>**
Der Variablen wird ein Ausdruck zugewiesen.

### Datoperator

Wird benutzt, um Methoden aufzurufen, die mit der Objektinstanz assoziert sind.

```
<ObjektReferenz>.Methodenname [Parameter]
```

Das gilt auch für Variablen:

```
<ObjektReferenz>.(Variablenname)
```

### Arithmetische Operatoren `+`, `-`, `*`, `/`, `%` (modulo)

### Vergleichsoperatoren `<`, `>`, `<=`, `>=`, `==` (Gleichheit), `!=` (Ungleichheit)

Die Vergleichsoperatoren können auch für Objektreferenzen verwendet werden.

Aber:

```
int[] a={1,2};
int[] b=(int[] a.clone[];
a==b
```

Hier wird das Ergebnis des Vergleichs **`false`** !

### Logische Operatoren

&& (und)
|| (oder)
Das Ergebnis wird von links beginnend ausgewertet. Sobald das Ergebnis feststeht, wird die Auswertung gestoppt.

### Einstellige Operatoren

**`!`** (logische Negation)
**`++`** (Inkrement)
**`--`** (Dekrement)
Die einstelligen Operatoren sind in Präfix-Notation oder Postfix-Notation einsetzbar.
Präfix-Notation: **`++a`** (erst erhöhen, dann rechnen)
Postfix-Notation: **`a++`** (erst rechnen, dann erhöhen)
Beispiel mit **`a1=42`**, **`a2=42`**, **`b1`** und **`b2`** :

```
b1=a1++    // b1=42, a1=43
b1=++a2    // b2=43, a2=43
```

### "Abgekürzte" Schreibweisen

Weiterhin gibt es "abgekürzte" Schreibweisen für Wertzuweisungen:
**`a`** ***`op`*** **`= x`** wird als **`a=a`** ***`op`*** **`x`** interpretiert. Also: **`a+=5`** bedeutet das Gleiche wie **`a=a+5`** .

### Bitweise Operatoren

~ Bitweises Komplement (also wird für 1 eine 0 eingesetzt, und umgekehrt)
& AND
| OR
∧ XOR
<< Bitshift nach links, mit Nullen auffüllen
>> Bitshift nach rechts, mit Nullen auffüllen

### Casting

Mit Casting wird der Wechsel des Variablentyps bezeichnet. Damit kann eine String-Variable in eine Numerische Variable überführt werden, und umgekehrt.

Syntax: `(<gewünschter Typ>) <Variable>`

Beispiele:

```
double d1 = 3.1;
double d2=3.999
int i1=(int)d1;      // Damit wird i1=3
int i2=(int)d2;      // Damit wird i2=3
```

Beispiele in Ausdrücken:

```
double dresult;
int i1=3;
int i2=6;
dresult=(double)i1/(double)i2     // dresult=0.5
dresult=i1/i2                     // dresult=0.0
```

.FEHLENDES

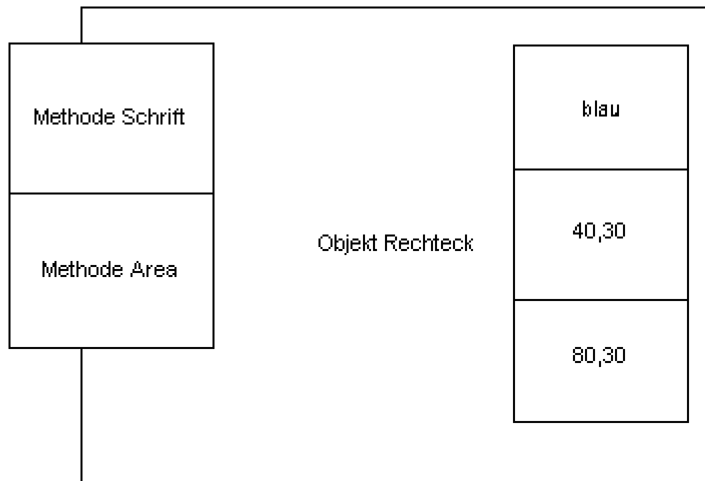
fehlt noch: Objekte/Methoden/Klassen + Objektorientierung
klassische imperative Programmierung:
prozedurale Sichtweise, d. h. Programme bestehen aus Prozeduren Funktionen die Daten manipulieren
Nachteil: Struktur der Daten muss bekannt sein, keine logische Verbindung zwischen Daten und Operationen (z. B. Sortieralgorithmen für verschiedene Datentypen einzeln)
Objekte: besondere Daten- und Programmstrukturen, die Eigenschaften und darauf definierte Operationen (Methoden) besitzen
Kapselung: Verbergen von inneren Eigenschaften der Objekte, Zugriff nur über bestimmte eigene Methoden der Objekte damit sind die Methoden die Schnittstelle (Stichwort information hiding – wichtig in der Programmentwicklung)
Die Grundzüge des objektorientierten Programmierens

- Abstraktion
- Kapselung
- Modularität

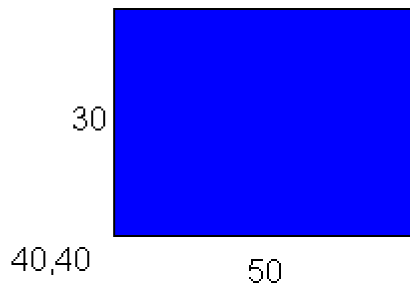
Das bedeutet, die Trennung von Daten und Operationen aufzuheben. Im Mittelpunkt steht nicht mehr das "wie" (prozedurale Sichtweise) sondern das "Was" (=die in der Anwendung existierenden Objekte)
Beispiele
1. Zeichenprogramm für geometrische Objekte wie Liniensegmente/Kreise/Rechtecke. Hier sind die Objekte die geometrischen Objekte. Jedes Objekt hat Eigenschaften:
- Segment (Anfangspunkt, Endpunkt)
- Kreis (Mittelpunkt, Radius
Jedes Objekt "weiss" wie es gezeichnet wird.
Zur Darstellung einer Zeichnung werden die Zeichenmethoden der einzelnene Objekte aufgerufen.
2. Sortieren von Objekten
Definition: Objekt=Identität (Eigenschaft, die es von anderen unterscheidet)
Attribute (statische Eigenschaften zur Darstellung des Zustandes)
Methoden (dynamische Eigenschaften, die das Verhalten beschreiben)
Objekt versus Wert: Die Änderung von Attributen führt nicht dazu, dass die Identität verändert wird (beispielsweise macht eine Namensänderung keinen neuen Menschen).

## *Einige Beispiele für typische Java-Entitäten*

### Beispiel für ein Objekt




### Beispiel für Instanz eines Objekts




### Beispiele für Methoden

Berechene Fläche: Area = h*w
Shift(dx,dy): x=x+dx; y=y+dy;

### Beispiele für Manipulation des Objekts

"Nachricht" an dieses Rechteck: "Verschiebe um Vektor (10,20)"
Der Aufruf verändert das Attribut "Position"
Definition einer Klasse: Menge von Objekten mit gleichen Attributen und gleichen Methoden.
Definition Instanz: Konkrete Ausprägung des Objekts einer Klasse (zb ist ein einzelnes Rechteck eine instanz der klasse rechteck).

## *Klassendefinition in Java*

- In der ersten Zeile kommt die **`package`**-Anweisung
  Zu welchem Paket (Sammlung von Klassen, Unterpaketen) soll die Klasse gehören
- Zugriff auf andere Pakete durch die **`import`**-Anweisung (Ausnahme **`java.lang`** )
- Vor Schlüsselwort class stehen optional Modifzierer
  **`final`**: Klasse kann keine Unterklassen haben
  **`public`**: Klasse kann durch alles im selben Paket, oder nach Importieren der Klasse instanziiert werden, beziehungsweise erweitert werden.
  friendly: Default-Modifier; alles im selben Paket

Der Klassenname kommt vor die Klassendefinition in Form **`classname {definition}`** .
Die Attribute werden als Variablen deklariert (brauchen also einen Typ).

Optional kann die Sichtbarkeit festgelegt werden:

- **`public`** Jeder kann zugreifen
- **`protected`** nur vom selben Paket, beziehungsweise von den abgeleiteten Unterklassen kann zugegriffen werden
- **`private`** Nur Methoden derselben Klasse dürfen zugreifen
- **`friendly`** Default

Gebrauch: **`static`** assoziiert die Variable mit der ganzen Klasse und nicht mit einzelnen Instanzen
final nimmt einen initialen wErt an, der danach nicht mehr verändert werden kann (zb konstantenzuweisungen wie pi=4 [dies beispiel funktionierte bis vor kurzem im us-bundesstaat indiana, pi war dort gesetzlich auf 4 festgelegt])

## *Methoden*

[<Methode_Modifier>]<returnTyp><Methodenname>([Parameter]){Rumpf}
Hier gibt es Modifier wie bei Instanzvariablen: **`final`**, **`static`**, **`abstract`** (Methode hat keinen Rumpf). Der return-Typ kann **`void`** sein, bedeutet, dass es keinen Rückgabetyp gibt.

### Spezielle Methoden

Konstruktoren haben den gleichen Namen wie die Klasse. Es können mehrere Konstruktioren existieren, sie brauchen dann aber unterschiedliche Parametersätze
Konstruktoren werden mit **`new`** aufgerufen, die Schlüsselwörter **`abstract`**, **`static`** und **`final`** funktionieren nicht.

### Methode `main`

Sie wird von anderen Klassen genutzt und ist notwendig für ein stand-alone Programm. Die Methode main hat immer
**`public static void main (String[] args){Rumpf}`**. Sie kann auch für andere Klassen zusätzlich definiert werden, was für die Fehlersuche nützlich ist.

### Beispiel zur Definition einer Methode

```
package geom;                                   Einordnung in das Paket
// Klassen
   public class Rectangle{                      Definition der Klasse Rechtecke
     int x,y,w,h;                               Deklaration der Instanzvariablen
                                                (Attribute)
     int color;
     public Rectangle(){x=y=0;w=h=10;}  Konstruktor-Definition
     public Rectangle(){x=y=20;w=h=10;color=2}      Konstruktor-
                                                Definition
     public Rectangle(int xp,int yp,int wi,int hi)
       {x=xp;y=yp;h=hi;w=wi;}                   Konstruktor für generisches Rechteck
// Methoden
  // Methode shift
  public void shift(int dx, int dy){x+=dx;y+=dy;} Definition der
                                                       Methode shift
  // Methode area
  public int area(){return w*h}
}
```

Anwendungen

1. Konstruktoraufruf
   `Rectangle r1=new Rectangle();` erzeugt ein Rechteck mit den Default-Werten
   `Rectangle r2=new Rectangle(1,10,1,10);`
2. Zugriff auf die Objektvariable/Eigenschaften mit dem dot-Operator
   `r1.shift(10,20);` Nur die Position des Objekts wird verändert auf das mit `r1` referenziert wird.

Nur die Position des Objekts wird verändert auf das mit `r1` referenziert wird.
Wenn Attribute `public` sind, dann geht auch `int xpos=r1.x; r1.y=10;` .

3. Bei Konflikten zwischen Attribute und Parametern hilft das Schlüsselwort `this`
   ```
   public void shift(int x,int y)
     {this.x+=x;this.y+=y;}
   ```
4. Die Klasseneigenschaften müssen außerhalb der eigenen Klasse durch Voranstellen des Klassennamens identifiziert werden.
5. Java hat eine Vielzahl von vordefinierten Klassen, die zu Bibliotheken zusammengefasst sind.

Einige Java-Pakete: `java.lang` (String-Klassen, Math-Klasse), `java.io`, `java.net` (zu finden unter www.javasoft.com)

## Ziele beim Softwareentwurf

Die Software soll die Eigenschaften Robustheit, Anpassungsfähigkeit und Wieder-verwendbarkeit haben.

## Charakteristika beim OO-Entwurf

Klassenkonzept: Die Eigenschaften und Methoden werden zusammengefasst
Vererbung: Die Eigenschaften und Methoden einer gegebenen Klasse werden auf neue Klassen übertragen
Polymorphie: Ein Objekt kann Instanz verschiedener Klassen sein
Fehlerbehandlung

## Vererbung, Beschatten/Überschreiben und Abstrakte Klassen/Interfaces

Eine Klassenstruktur ohne Vererbung scheint erst einmal effizient:
Klasse Student, Klasse Angestellte, … , Klasse Alumni, … , Klasse Tutor.
Sie lassen sich auch leicht durch kopieren und Einfügen erzeugen. Jedoch müssen spätere Änderungen gleichsinnig in jeder Klasse gemacht werden. Weiterhin wird der logische Zusammenhang nicht deutlich. Besser wird das Problem mit der Einführung einer neuen Oberklasse `Person` gelöst. Sie vererbt ihre Eigenschaften an die untergeordneten Klsassen, wobei auch die logischen Beziehungen sichtbar werden: Der `Student` ist eine `Person`.
Dabei gibt es dann einige Regeln:
Erlaubt in Unterklasse `Student`:

1. Neue Instanzvariablen einführen (z. B. `Stipendium`)
2. Neue Methoden (z. B. `getStip`)
3. Existierende Methoden von Person werden überschrieben (overriding)

Nicht erlaubt:

1. Felder von `Person` entfernen
2. Methoden von `Person` entfernen (Anmerkung: Lässt sich aber einfach dadurch lösen, dass man eine Methode, die man entfernen möchte, mit einer Methode, die gar nichts tut, überschreibt)

Zur Implementation werden die Instanzvariablen **name**, **age**, **address** und **phone** von **Person** . Sie werden als **private** deklariert, da ansonsten Unterklassen die Werte der Oberklasse einfach ändern könnten. Das wäre ein großes Sicherheitsrisiko!
Die Unterklasse **Student** hat keinen Zugriff auf die **private**-Variablen von **Person**, sie können aber mit **public**-Methoden von Person erreicht werden. Zusätzlich deklariert **Student** die Variable **stipendium** .

### Beschatten (Shadowing) von Instanzvariablen und Statischen Methoden

Die übernommen Instanzvariablen können von einer Unterklasse neu belegt werden. Das Gleiche gilt für statische Methoden der übergeordneten Klasse.

### Überschreiben von Methoden

Dies geht bei gleichem Methodennamen und Parametern der in der Unterklasse definierten Methode.

Beispiel zum Überschreiben von Methoden

```
class A {                                 // Oberklasse
      int i=1                             // Feld
      int f(){return i;}                  // Instanzvariable
      static char g(){return 'A';}        // statische Klassenmethode
}
class B extends A {                       // Definition der Unterklasse
      int i=2                             // Beschattet Feld i in A
      int f(){return -i;}                 // Überschreibt Methode aus A
      static char g(){return 'B';}        // Beschattet statische Methode aus A
}
public class OverridingTest {
      public static void main (String[]args) {
      B b=new B();                        // neues Objekt von B
      System.out.println(b.i);            // ergibt 2
      System.out.println(b.f());          // ergibt -2
      System.out.println(b.g());          // ergibt 'B'
      System.out.println(B.g());          // ergibt 'B' <- besser, da g()
                                             static ist! Damit ist es nicht
                                             abhängig von den Instanzen, sondern
                                             nur von der Klasse
      A a=(A)b;                           // Casting von b auf A
      System.out.println(a.i);            // ergibt 1 ! Die Referenz in A
                                             entscheidet!
      System.out.println(a.f());          // ergibt -2. Die Referenz ist immer
                                              noch auf B.f() .
      System.out.println(a.g());          // ergibt 'A'
      System.out.println(A.g());          // ergibt 'A'
```

Probleme und deren Lösung beim Overriding

Der Instanztyp hinter der Referenz steht erst zur Laufzeit fest (-> dynamic method lookup / dynamic binding / late binding)

Schutz vor Überschreiben durch Modifier:

1. Oberklassen als `final` deklarieren
2. Einzelne Methoden als `final` deklarieren
3. Methoden als `private` deklarieren
4. Methoden als `static` deklarieren (nur Shadowing beim Überschreiben)

## Abstrakte Klassen und Interfaces

Abstrakte Klassen sind ganz oder teilweise offen implementiert. Der Modifier `abstract` steht vor der Klasse und vor nicht implementierten Methoden. Da gibt es dann keine `{}`-Klammern und `;` schließt die Definition ab. Sie haben keine Konstruktoren (der Befehl `new` ist nicht möglich) und davon abgeleitete Klassen sind ebenfalls abstrakt.
Die Idee ist, dass abstrakte Methoden als Platzhalter fungieren und letztlich von den Unterklassen realisiert werden.

### Beispiel für eine abstrakte Klassse

Die Oberklasse ist `Shape` , und die Unterklassen sind `Rectangle` , `Square` und `Circle` .

```
public abstract class Shape {
      public abstract doublearea();     // kein Rumpf der Methoden
      public abstract doubleperimeter();
      public double semiperimeter() {
           return perimeter()/2.0;}
}
public class Circle extends Shape {
      private double radius
      public Circle (double rad)
           {radius=rad;}
      public double area()
           {return Math.PI*radius*radius;}
      public double perimeter()
           {return 2.0*Math.PI*radius;}
```

## Mehrfachvererbung

Die Mehrfachvererbung ist im Allgemeinen in Java nicht erlaubt. Wenn der Tutor einmal von `Angestellte` und einmal von `Student` erbt, dann könnte er z. B. zweimal einen Namen zugewiesen bekommen.
Das Problem kann mit interfaces gelöst werden. In `interface`-Klassen gibt es nur abstrakte Methoden und `static` und `final`-Felder. Sie dienen zur Datenabstraktion. Dazu muss das Schlüsselwort `interface` vor die Klassendefinition (`public interface` blablubb) gesetzt werden. Alle hierin definierten Methoden sind also implizit abstrakt und es gibt keine Konstanten.
###fehlendes###

## Zusammenfassung Vererbung und Klassenhierarchien

1. Jedes Objekt der Unterklasse ist Objekt der Oberklasse, nie umgekehrt
2. Einfachvererbung und Mehrfachvererbung: Java hat keine Mehrfachvererbung (kann über Interfaces gelöst werden)
   Jede Klasse bei Java hat genau eine Oberklasse (Ausnahme: `java.lang` hat keine Oberklasse)
3. Wird eine Unterklasse von einer Oberklasse abgeleitet, so muss beim Erzeugen eines Objektes auch der Konstruktor der Oberklasse aufgerufen werden. Es gibt also ein sogenanntes constructor chaining bis zur Klasse `java.lang.Object`)

Jede abgeleitete Klasse soll einen eigenen Konstruktor definieren. Standardmäßig wird der Oberklassen-Konstruktor verwendet.

Umsetzen in Java-Code

```
class Student extends Person{
  public Student(String n,int ag,String ad,String ph,double
st)
    {super(n,ag,ad,ph); stip=st}
      private double stip;
}
```

Die **super**-Konstruktormethoden können mit den Parametern aufgerufen werden, die einen Oberklassenkonstruktor matchen. Das geht nur in der 1. Zeile sonst wird der default-Konstruktor verwendet.

4. Objekte der Unterklasse können wie Objekte der Oberklasse verwendet werden, z. B. in Anweisungen:

```
Student marco = new Student("Marco",18,….,1000.0);
Person p = marco;    // zulässig
```

Als Konsequenz: Die Objekte der Klasse `java.lang.Object` können Instanzen jeder anderen Klasse enthalten. Dabei geht aber die Typ-Information verloren (kann durch `instanceof` [`if p instanceof Student` …] wieder geholt werden).
Schöne Syntax:

```
public class Unterklasse extends Oberklasse{
/* alles, was nicht aufgelistet ist, wird unverändert von der
Oberklasse übernommen */
/* evt. ist der Konstruktor davon ausgenommen */
/* public-Teil:
/* Konstruktoren, falls default-Konstruktor nicht passt,
geänderte Oberklasse-Methoden und zusätzliche Methoden */
/* private-Teil: */
/* Zusätzliche Attribute und Methoden */
}
```

Eine Oberklasse-Methode wird durch die Eigenschaft `static` vor der Modifikation durch eine Unterklasse geschützt:

```
public static boolean isOlder(Person p1, Person p2);
  {return p1.getAge()>p2.getAge();}
  Person p=new Person(…);
  Student s=new Student(…);
  Angest a=new Angest(…);
```

-> Mögliche Anfragen: `isOlder(p,s)`, `isOlder(s,a)`

## *Polymorphie*

Ein Konzept kann unterschiedliche Formen annehmen

- Überladen eines Operators **+** : Dabei gibt es den gleichen Methodennamen in einer Klasse, aber verschiedene Parameter)
- Überschreiben (gleiche Methodennamen, gleiche Parametersätze in verschiedenen Klassen
- Typkonversion (Beschatten)

Variablen einer Oberklasse können durch gleichnamige Variablen der Unterklasse beschattet werden (der Typ der Referenz entscheidet, auf welche Variable zugegriffen wird).

Allerdings kann in den Methoden von C (unterste Unterklasse; Unterklasse von B, welches Unterklasse von A ist) auf verschiedene Variablen namens *x* zugegriffen werden:

```
x            // Feld x in C
this.x    // Feld x in C
super.x  // Feld x in B
((B)this).x  // Feld x in B
((A)this).x  // Feld x in A
super.super.x // SYNTAXFEHLER !!!
```

FEHLENDES !!!

## *Analyse: Amortisierte Komplexität*

Vereinfacht: Was kosten Push-Operationen?
Die Array-Größe sei eine Zweier-Potenz

Die Kosten der i-ten push-Operation: $c := \begin{cases} i \text{falls} (i-1) \text{2erPotenz} \\ 1 \text{sonst} \end{cases}$

$$\sum_{i=1}^{n} c_i = n + \sum_{j=0}^{\lfloor \log_2 n \rfloor} 2^j < n + 2n = 3n$$

### Die Aggregat-Methode

Die Aggregat-Methode bei der amortisierten Analyse: Eine Folge von n Operationen kostet im ungünstigsten Fall $T(n)$. Die amortisierten Kosten pro Operation sind $\frac{T(n)}{n}$, hier 3.

### Die Accounting-Methode

[nachlesen]
Sie stellt für die einzelnen Operationen verschiedene amortisierte Kosten in Rechnung. Für einige Operationen sind die Kosten höher als der tatsächliche Aufwand. Der Überschuss wird bestimmten Objekten zugeordnet und hilft teure Operationen zu bezahlen.
Annahme: Ein Push kosten eigentlich 1 €, das Verdoppeln des Arrays von k auf 2k kostet also k €. Die amortisierten Kosten für Push sind dann also 3 €. In Wirklichkeit kostet das Einfügen eines Wertes bei $2^{i-1}$ in einen Array nur 1 Euro, also kann ich pro Einfügen 2 Euro auf die Seite legen. Beim nächsten Push sind $2^i$ Euro für das Kopieren erforderlich.

### Die Potentialfunktion-Methode

[nicht im Rahmen der Vorlesung]

# Abstrakte Datenstrukturen (ADTs)

## *Queues, Verkettete Listen und Deques*

### Queues (Warteschlangen)

Datgenstruktur, bei der Elemente "am Ende" eingefügt werden. Das Element, welches am längsten drin ist, kann entfernt werden (FIFO-Prinzip im Gegensatz zum Stack, bei dem das LIFO-Prinzip gilt).

#### Hauptmethoden

**`enqueue(obj)`** Einfügen am Ende
**`dequeue(obj)`** Front-Element entfernen/ausgeben; Fehlermeldung wenn leer

### Zusatzmethoden

**size()**
**isEmpty()**
**front()** Ausgabe des Front-Elementes ohne Entfernen und Fehlermeldung

### Queues in Java

Als Interface

```
public interface Queue{
     public int size();  // optional
     public boolean isEmpty;
     public Object front() throws QueueOverflowException; CHECK
     public void enqueue(Object);
     public Object dequeue() throws DequeueEmptyException CHECK
}
```

### Implementierung als lineares Array der Grösse N

[Zeichnung]
Markieren mit f erste belegte Zelle, mit r die letzte belegte Zelle

```
enqueue(obj){
     if (r==N) throw QueueOverflowException;
     Q[r]=obj;
     r++;}
dequeue(obj){
     if (f==r) throw QueueEmptyException;
     Object obj=Q[f];
          Q[f]=null; f++;
          return obj;}
```

Allerdings ist bei dieser Form der Implementierung das Problem, dass die Daten durch den Queue wandern, und irgendwann das Ende erreicht ist. Dann wird eine "Voll"-Fehlermeldung ausgegeben, obwohl der Queue leer ist. Das kann man damit verhindern, indem man den Queue zirkulär implementiert, d. h., wenn das Ende erreicht ist, dann wird wieder vorne eingefügt.

### Implementierung als zirkuläres Array der Grösse N

[Zeichnung]

```
enqueue(obj){
     if ((r+1)%N==f) throw QueueOverflowException;
```

**r+1** und nicht **r** wird verwendet, um zwischen ganz leer und voll zu unterscheiden. Also bleibt eine Stelle immer frei. Das %-Zeichen steht für Modulo.

```
     Q[r]=obj;
     r=(r+1)%N;}
dequeue(obj){
     if (f==r){throw QueueEmptyException;
     Object obj=Q[f];
          Q[f]=null;
          f=(f+1)%N;
          return obj;}
}

size(){
return (N-f+r)%N;}
```

Die Methoden haben die Komplexität $\mathcal{O}(1)$, der Speicherbedarf ist $\mathcal{O}(N)$.

### Verkettete Listen

Die Verkettete Liste ist der Prototyp einer dynamischen Datenstruktur. Sie besteht aus einer Kollektion von "Knoten" in linearer Ordnung. Jeder Knoten ist eine Instanz einer Klasse Node. Die Instanz hat 2 Referenzen: 1. Auf "gespeichertes" Objekt und 2. auf den nächsten Knoten. **`head`** referenziert den ersten Knoten, **`tail`** referenziert den letzten Knoten.
[Zeichnung]
Dabei kann am Ende und am Anfang billig eingefügt werden.
Nur am Anfang kann billig gestrichen werden, da das vorletzte Element nicht direkt erreichbar ist!
Eine Queue als einfach verkettete Liste: Methode **`vornStreichen()`** und **`hintenEinfügen()`** . Einen Stack als einfach verkettete Liste: Methode **`vornStreichen()`** und **`vornEinfügen()`** .

### Deques

Deques unterstützen das Einfügen und Streichen vorne und hinten.

#### Der ADT für Deques

Hauptmethoden
insertFirst(obj)
insertLast()
removeFirst()
removeLast()

Implementierung von Deques mit doppelt verketteten Listen erfordert noch die Methoden **`previousReference()`** und die speziellen Nodes **`header`** und **`trailer`** (dies sind sentinel Knoten. sentinel steht für Wächter. Die sentinel Knoten werden auch dummy Knoten genannt).
[Zeichnung]
DLNode hat folgende Methoden:

**`setElement(Object obj)`**
**`setNext(DLNode newNode)`**
**`setPrev(DLNode newPrev)`**
**`getElement()`** Rückgabe von Objekt
**`getNext()`** Rückgabe von DLNode
**`getPrev()`** Rückgabe von DLNode

Mit doppelt verketteten Listen sind alle Methoden von Deques in $\mathcal{O}(1)$ durchführbar. Ein leerer Deque hat zwei Elemente, den **`header`** , der auf **`trailer`** zeigt und umgekehrt.

## *Vektoren, Listen, Sequenzen*

Grundlegende Konzepte: Rang, Position
Definition: $e \in S$ ist die lineare Ordnung von n Elementen
rank(e) = # Elemente vor e in S
(Achtung: Das muss nicht mit Array implementiert werden!
Bisher als ADT: Stack Queue, Deque
Implementiert mit Array, einfachen bzw. doppelt verketteten Listen, dynamischen Arrays. Typisch dafür ist der Zugriff nur am Anfang bzw. Ende. Jetzt kann der Zugriff, Einfügen und Löschen an beliebiger Stelle erfolgen.

### Vektor als ADT mit Methoden

| | |
|---|---|
| **`elemAtRank(r)`** | Element auf Rang r |
| **`replaceAtRank(r)`** | Ersetze Element auf Rang r |
| **`insertAtRank(r)`** | Füge ein auf Rang r |
| **`removeAtRank(r)`** | Entferne von Rang r |
| **`size()`** | Länge des Vektors (=Anzahl der Elemente) |
| **`isEmpty()`** | Wahrheitswert; wahr, wenn der Vektor leer ist |

+ Fehlermeldungen

### Listen

Hier wird die "Position" (=Posten) benötigt.
- Mittel, um Elemente direkter anzusprechen
- Betrachten Liste als Container, der Positionen enthält und diese in linearer Ordnung hält
- Eine Position ist relativ, sie hat ein davor und ein danach

Achtung: Die Position mit Element e ändert sich nicht, auch wenn sich der Rang von e ändert (Der Posten ist ein Eintrag in einer Liste, in welcher Reihenfolge die Einträge getätigt werden, ist egal). Sie ändert sich auch nicht, wenn e durch ein anderes Element ersetzt wird. Die Position ist eine ADT mit der Methode element().

#### Listen-Methoden

| | |
|---|---|
| **`first()`** | ergibt Position |
| **`last()`** | ergibt Position |
| **`isFirst(p)`** | Boolescher Wert |
| **`isLast(p)`** | Boolescher Wert |
| **`before(p)`** | Position |
| **`after(p)`** | Position |
| **`replace(p,e)`** | gib Element von p aus und füge e ein |
| **`swapElements(p,q)`** | vertausche die Elemente an den Positionen p und q |
| **`insertFirst(e)`** | füge e als erstes Element in S ein |
| **`insertLast(e)`** | füge e als letztes Element in S ein |
| **`insertBefore(p,e)`** | füge p vor e ein |
| **`insertAfter(p,e)`** | füge p nach e ein |
| **`remove(p,e)`** | Ausgabe und Löschen des Elements von p |

#### Beispiele für Listenoperationen

| Operation | Output | Liste S |
|---|---|---|
| **insertFirst(8)** | $p_1(8)$ | (8) |
| **insertAfter($p_1$,5)** | $p_2(5)$ | (8,5) |
| **insertBefore($p_2$,3)** | $p_3(3)$ | (8,3,5) |
| **insertFirst(9)** | $p_4(9)$ | (9,8,3,5) |

#### Implementierung mit doppelt verketteten Listen

- Die Nodes der verketteten Liste implementieren die ADT Position
- Alle Operationen sind mit $\mathcal{O}(1)$ durchführbar!

Pseudo-Code für **insertAfter(p,e)**

```
Schaffe neuen Node v;
v.setElement(e);
v.setPrev(p);  // p Vorgänger von v
v.setNext(p.getNext()); // alter Nachfolger von p ist
Nachfolger von v
(p.getNext().setPrev(v);
p.setNext(v);
return v;
```

### Sequence

$$\text{ADT Sequence} = \text{Liste} + \text{Vector} + \begin{cases} \text{Position...atRank}(\text{int } r) \\ \text{int} \text{...........rankOf}(\text{Position} p) \end{cases}$$

Vorteile einer Array-Implementation

**atRank(r)**, **rankOf(p)**, **elementAtRank(r)** in $\mathcal{O}(1)$ (gegen $\mathcal{O}(n)$ bei verketteten Listen)

Vorteile von einer Implementation mit Verketteten Listen

**insertAfter**, **insertBefore**, **remove** in $\mathcal{O}(1)$ (gegen $\mathcal{O}(n)$ bei Arrays)

Der Nachteil bei beiden Implementation ist, dass **insertElementAtRank** und **removeElementAtRank** teuer sind.

## *Bäume als Datenstruktur*

Sie verwalten hierarchisch geordnete Daten (z. B. Stammbäume, Dateisysteme, Gesetzbücher…)
Wir betrachten hier gewurzelte Bäume (rooted trees), d. h. Bäume mit ausgezeichneten Knoten r (nämlich die Wurzel).

### Einige Eigenschaften gewurzelter Bäume

- Die Wurzel (Knoten *r*) induziert die Vorgänger-Relation auf den Baum
- Für jeden Knoten *v* gibt es genau einen Weg zur Wurzel
- Die Knoten auf diesem Weg heißen Vorgänger bzw. Vorfahren von *v* (einschließlich *v*)
- Der Vater von *v* ist der direkte Vorfahre von *v*
- Mit *w* wird das Kind von *v* bezeichnet
- Innere Knoten sind Knoten mit Kindern
- Äußere Knoten oder Blätter sind Knoten ohne Kinder
- Die Höhe oder Tiefe eines Knotens bezeichnet den Abstand zur Wurzel
- Die Tiefe lässt sich rekursiv definieren

Fehlerbehandlung in Java
Object – thorowable (erorr oder xception – RunTimeException

Error: schwerer Fehler, führt im Allgemeinen zum Abbruch
Exception: Situationen, die man behandeln kann (aber nicht muss)
Exceptionmechanismus in Java (b. eckel "thinking in java")
Exception: Signal, das auf Fehler/Ausnahmesituation hinweist
Um darauf hinzuwiesen, wird Exception geworfen (**throw**)
To catch an exception: Die Situation behandeln
Alle Exceptions sind Objekte

Nicht von RunTimeException abgeleitete Exceptions heißen "checked". Sie können nur in Methoden aufgerufen werden, in denen sie deklariert sind.
Instanzen von Exceptions werden mit **`new`** erzeugt

### try catch finally Mechanismus

Exceptions werden in einem **`try`**-Block abgeschickt
Auf **`try`**-Block folgen ein oder mehrere **`catch`**-Blöcke, die jeweils einen Exceptionstyp behandeln.
Exceptions, die in ihrer Methode abgefangen werden, werden nach oben in die aufrufende Methode weitergereicht. Falls **`main`**-Methode nicht abfängt, bricht das Programm ab.
Nach einem **`catch`** Block folgt optional ein **`finally`**-Block. Dessen Anweisungen werden auf jeden Fall ausgeführt, wenn ein **`try`**-Block begonnen wurde. Der **`finally`**-Block eignet sich also gut zum Aufräumen, wie dem Schließen von Dateien, Trennen von Verbindungen etc.)

### Beispiel zu Exceptions

```
try{s=q.slope(p)}
catch (InfiniteSlopeException exc){
if (p.y==q.y) {throws new EqualPointException}
if (p.x<q.x) s=Double.Positive


public class IntPoint{
     public int x,y ;
public double slope  (IntPoint p){
throw InfiniteSlopException{if (this.x==p.x) throw new
InfiniteSlopeException ("zu steil");}
double sl int (double) (p.y-this.y) / (double) ( p.x-this.x);
return sl;
```

## *Gewurzelte Bäume als ADT*

In Arithmetischen Ausdrücken (Aufdröseln von Boolschen Termen) oder Entscheidungsstrukturen (jeder Knoten eine Frage mit zwei oder mehreren Antworten) werden gewurzelte Bäume angewendet.
Die Elemente werden in Baumknoten gespeichert, diese Knoten sind wie Positionen bei Listen relativ zu den Nachbarn definiert.

### Methoden zum ADT Gewurzelter Baum

| | |
|---|---|
| **`root(v)`** | Gibt Wurzelknoten zurück |
| bei einzelnen Knoten: | |
| **`Object element(v)`** | |
| **`TreeNode parent(v)`** | Elternknoten von **`v`** |
| **`TreeNode[] children(v)`** | Liste mit den Kindern von **`v`** |
| **`isRoot(v)`** | Boole-Wert, ob der Knoten die Wurzel ist |
| **`isLeaf(v)`** | Boole-Wert, ob der Knoten ein Blatt ist |
| **`set Element(v,e)`** | Schreibt e in den Knoten **`v`** |
| **`addChild(TreeNode v)`** | neues Kind unter **`v`** |

### Implementierung von `depth` als Methode von `TreeNode`

```
int depth() {
     int d=0;
     if (!isRoot()) {    // das ! steht für das Gegenteil
          d=1+parent().depth()
     return d;}
}
```

### Implementierung von `height` als Methode von `TreeNode`

```
int height() {
     int h=0
     if (!isLeaf()) {
          for (i=0; i<children().length; i++) {
               if (h<children[i].height() {
                    h=i;}
     }
          return h;
}
```

## *Traversieren von Bäumen*

Das Traversieren von Bäumen ist das Aufzählen der Knoten eines Baums.
Preorder-Traversierung: Für jeden Teilbaum besuche <u>zuerst</u> Wurzel v und dann rekursiv die Teilbäume unter den Kindern von v
Postorder-Traversierung: Besuche <u>zuerst</u> die Teilbäume der Kinder, dann v
Inorder-Traversierung (nur bei binären Bäumen)

## *Binäre Bäume*

Geordneter Baum, jeder innere Knoten hat genau 2 Kinder (??? Nicht auch 1 Kind?)

- gewurzelt
- innere Knoten haben 2 Kinder
- Postorder-, Preorder-, Inorder-Traversierung
  - Alle O(n)
  - Rekursiv definiert
- Die Euler-Tour verallgemeinert alle 3 Traversierungen

1. Postorder: Der Wert bei den Knoten hängt vom Wert bei den Kindern ab (z. B. bei der Höhenberechnung)
2. Preorder
3. Inorder
4. (a) Sortieren von Einträgen im Binären Suchbaum
5. (b) Zeichnen von binären Bäumen: Knoten v->Koordinaten x(v)= Anzahl der Knoten von v in der Inorder-Traversierung, y(v) ist die Tiefe von v.

### Methoden bei Binären Bäumen

**`leftChild(v)`**
**`rightChild(v)`**
**`sibling(v)`**
**`expandExternal(v)`** fügt 2 Kinder an Blatt an
**`removeAboveExternal()`**

### Grössenverhältnisse bei binären Bäumen mit der Höhe h

(0) Anzahl der Blätter = 1+Anzahl der inneren Knoten (Beweis mit Induktion über h)
(1) h+1 =< Anbzahl der Blätter =< $2^h$
(2) h+1 =< Anbzahl der inneren Knoten =< $2^h$-1
(3) 2h+1 =< Anbzahl aller Knoten =< $2^{h+1}$
(4) $\log_2(n+1)-1 \le h \le \frac{n-1}{2}$

### Implementierung von Binären Bäumen

#### Array-basierte Implementierung

Diese Implementierung verwendet eine kanonische Nummerierung der Knoten. Das bedeutet, dass die Knoten im vollständigen binären Baum von der Wurzel beginnend einfach ebenenweise durchgezählt werden. Das gilt auch, wenn der Baum nicht vollständig ist. Das hat den Vorteil, dass der Index der Kinder zu denen der Väter in Beziehung steht: $Index_{Kindlinks}=2*Index_{Vater}$ und $Index_{Kindrechts}=2*Index_{Vater}+1$.

#### Verkettete Strukturen für binäre Bäume

<table>
<tr><td rowspan="2">Verweis auf den Element-Wert des linken Kindes</td><td>Verweis auf den Element-Wert des Vaters</td><td rowspan="2">Verweis auf den Element-Wert des rechten Kindes</td></tr>
<tr><td>Element-Wert</td></tr>
</table>

#### Verkettete Strukturen für allgemeine Bäume

<table>
<tr><td>Verweis auf den Element-Wert des Vaters</td><td rowspan="2">Element-Wert</td></tr>
<tr><td>Verweis auf Container für Kinder</td></tr>
</table>

#### Simulation von allgemeinen Bäumen durch binäre Bäume

Regeln: Das erste Kind wird linker Sohn, Weitere Kinder werden rechte Kinder unter dem 1. Sohn. Die freien Stellen werden mit Dummy-Knoten gefüllt.

#### Entscheidungsbäume

Die inneren Knoten werden mit Ja/Nein-Fragen bestückt und die Blätter enthalten die Antworten/Lösungen.
Das Standard-Beispiel ist bergleichsbasiertes Sortieren. Dabei hält jedes Blatt eine der möglichen Reihenfolgen (n! verschiedene).

#### Die Untere Schranke für vergleichsbasiertes Sortieren

A sei irgendein Sortier-Algorithmus, $T_A$ der Entscheidungsbaum dazu. Die Fragestellung ist, wie $T_A$ bei der Eingabe $a_1, a_2, \dots, a_n$? Weiterhin nehmen wir an, dass jedes Blatt verschieden von allen anderen ist (die Elemente sind paarweise verschieden). Eine konkrete Eingabe ergibt den Weg zu einem Blatt. Die Weglänge entspricht dann der Mindestanzahl der notwendigen Operationen und somit der Komplexität. Der worstcase ist der laengste Weg zu einem Blatt. Die Topologie des Baumes wird durch den Sortieralgorithmus bestimmt. Für alle binären Bäume gilt: h+1=< Zahl Blätter =< $2^h$. Also muss $T_A$ mindestens n! Blätter haben, also $2^h$>=n! => h >=$\log_2$(n!)>$\log_2$((n/2)^(n/2)) = (n/2)$\log_2$(n/2)=Omega(n$\log_2$n).

## Prioritätswarteschlangen

Die Daten haben zusätzlichen Schlüssel (keys). Die Schlüssel haben eine lineare, totale Ordnung (Zahlen oder Strings) =< linear.

1. $\exists k, k' \;\; k \le k'$ oder $k' \le k$ (total)
2. $\exists k \;\; k \le k$ (reflexiv)
3. $\exists k, k'$ ( $k \le k'$ und $k' \le k$ ) (antisymmetrisch)
4. $\exists k, k', k'' \;\; \left(k \le k' \wedge k' \le k''\right) \Rightarrow k \le k''$

FEHLENDES

Satz: Ein Heap, der n Elemente speichert, hat die Höhe $h = \left\lceil \log_2 (n+1) \right\rceil$

Beweis: $1+2+4+...+2^{h-2} + \underbrace{1}_{\text{mindestens 1 Eintrag auf vorletztem Level}} \le n \le 1+2+...+ \underbrace{2^{h-1}}_{\text{vorletztes Level voll}}$

$2^{h-1} \le n \le 2^h - 1$

$2^{h-1} + 1 \le n+1 \le 2^h$ (logarithmieren)

### Einfügen in dn Heap

1. Finde erste freie Stelle
2. Füge Element dort ein und stelle Heapinvarianten wieder her
3. ggf. schrittweises Vertauschen des neuen Schlüssels nach oben bis kein Konflikt mehr, einmal
4. Aufwand: O(1) für das Vertauschen und insgesamt O(log($n$)) Aufwand

### Löschen

Die Wurzel hat den minimalen Schlüssel

1. Entferne das "Loch"
2. Füge letztes Element in "Loch" ein und versichern
3. Aufwand ist O(log($n$))

Satz: SelectionSort mit Heaps (Heap-Sort) hat den Aufwand O($n$log($n$)).
Beweis: 1. leerer Heap, 2. n-Mal einfügen, 3. n-Mal streichen.
Eine Heap-Konstruktion ist auch in linearer Zeit möglich. Das kann durch eine bottom-up Methode erreicht werden $n=2^h-1$

## Lineare Datenstruktur vs. Baumartiger Datenstruktur

Die Implementierung einer Prioritätswarteschlange kann mit geordneten oder ungeordneten Sequenzen erfolgen. Eine weitere Methode sind Heaps (Einfügen und Miffff in O(log2n) I

### Heapsort läuft in O(nlog2n)

1. Phase: Baue Heap bottom-up (O(n))
2. Phase: n Mal das Minimum entfernen $\mathcal{O}\left(n \log_2 n\right)$, in Summe $\sum = \mathcal{O}\left(n \log_2 n\right)$

Die Phase 2 ist nicht in $o\left(n \log_2 n\right)$ möglich! Heap Sort ist nämlich vergleichsbasiert, deswegen ist $\Omega\left(n \log_2 n\right)$ die untere Schranke

Max-Heap statt Min Heap ist völlig analog!

## *ADT Dictionaries (Wörterbücher)*

Datenstruktur zur Verwaltung von Items mit Key
Einfügen, Streichen, Suchen bzgl. Schlüssel möglich
Beispiele sind Lexika, Telefonbücher, Kundendatei etc.
Es werden geordnete und ungeordnete Wörterbücher unterschieden (dann zusätzliche Methoden)

### Ein Wörterbuch als ADT

| | | |
|---|---|---|
| Container-.Methode | **`int size()`** | |
| Container-.Methode | **`boolean isEmpty()`** | |
| Container-.Methode | **`Object[] elements()`** | |
| Query-Methode | **`Object findElement(key k)`** | // Element mit Schlüssel k, sonst **`no_such_key`** |
| Query-Methode | **`Object[] findAllElements(key k)`** | // alle Elemente |
| Update-Methode | **`void insertItem(key k, object c)`** | |
| Update-Methode | **`Object remove(key k)`** | //Element mit Schlüssel k streichen, oder **`no_such_key`** |
| Update-Methode | **`Object removeAll(key k)`** | // alle Elemente streichen |

Zusätzlich bei geordneten Wörterbüchern:

| | |
|---|---|
| **`key closestKeyBefore(key k)`** | // Schlüssel eines Elements max =< k |
| **`Object closestElementBefore(key k)`** | |

### Einfache Implementierung eines Wörterbuches

1. Ungeordnete Sequenz: Suchen/Entfernen hat O(n), Einfügen hat O(1). Beispiel: Log-Dateien
2. Array-basierte geordnete Sequenz: Suchen hat O(log2n) (Binärsuche), Einfügen/Entfernen hat O(n). Beispiel Lookup-Tables
3. Geordnete Wörterbücher mit verketteter Liste: Finden O(n) danach Einfügen/Steichen in O(1)

### Binärsuche

Idee: Bei jedem Schritt wird der Suchbereich halbiert

| Schritt | Suchraumgröße |
|---|---|
| 0. | n |
| 1. | n/2 |
| 2. | n/4 |
| i. | $n/2^i$ |
| $\log_2 n$ | 1 |

Pseudocode for Binary Search

Algorithm **`BinarySearch(S, k, low, high)`**

```
if low > high then
     return NO_SUCH_KEY
else
     mid <-(low+high)/2
     if k=key(mid) then
          return key(mid)
     else
```

```
          if k<key(mid) then
               return BinarySearch(S, k, low, mid-1)
          else
               return BinarySearch(S, k, mid+1, high)
```

## *Suchbäume*

Ein Suchbaum ist ein binärer Baum. Das bedeutet das jeder innere Knoten ein Item (key) des Wörterbuchs hält. Für die keys im linken Teilbaum unter v gilt v =< Schlüssel in v. Für die keys im rechten Teilbaum unter v gilt v => Schlüssel in v. Die Blätter sind Platzhalter.

### Pseudocode für Methode `TreeSearch(k,v)`

Input: key k, Knoten v
Output: Knoten w im Unterbaum T(v) <- gewurzelt in v
w ist der innere Knoten mit dem Schlüssel k oder auch, anders ausgedrückt, w ist als Blatt bei Inorder-Traversierung zwischen den Schlüsseln<k und den Schlüsseln>k.

```
if v Blatt then
     return v;
if k=key(v) then
     return v
else
     if k<key(v) then
          return TreeSearch(k,T.leftChild(v));
     else return TreeSearch(k,T.rightChild(v));
```

### Methode Einfügen `insertItem(k, e)`

Sei w ein gefundener Knoten bei `TreeSearch(k,T.root())`

1. Fall w ist Blatt: Speicherung von k,e in w, danach 2 neue Blätter für w. Die Information wird in den Inneren Knoten gehalten, die Blätter sind immer leer.
2. Fall w ist innerer Knoten: Zwei Möglichkeiten: grösster Schlüssel =< k oder kleinster Schlüssel >= k

### Methode Enferne Knoten mit Schlüssel k

Fall 1: Ein Kind von v ist Blatt
Fall 2: Beide Kinder sind innere Knoten
Durch Einfügen/Entfernen ändert sich die Gestalt des Baumes (Höhe und/oder Knotenstruktur). Deswegen müssen Routinen implementiert werden, die dafür sorgen, dass der Baum möglichst balanciert bleibt.

## *Wörterbücher*

Sie bestehen aus Items=Key+Element
Methoden sind Suchen, Einfügen, Streichen
Implementation mit Listen oder mit Suchbäumen

### Beispiel Inverses Telefonbuch

10-stellige Telefonnummern
R=$10^{10}$-1, dabei werden nur n<<R benutzt. Anfragen sind Nummer nach Name.
Lösungsmöglichkeiten

1. Array der Größe $10^{10}$: Die Antwortzeit ist $\mathcal{O}(1)$, aber der Speicherverbrauch ist immens.
2. Balancierte Suchbäume: Speicher ist $\mathcal{O}(n)$ und die Antwortzeit ist $\mathcal{O}(\log_2 n)$
3. Hashing

# AVL-Bäume (nach Adelson-Velski, Landis, 1962)

Die Höhe eines Knotens ist die Weglänge zu einem tiefsten Blatt unter *v*.

Definition: T ist ein AVL-Baum, wenn er die Höhen-Balance-Eigenschaft hat, d. h. $\forall v \in T$. Die Differenz der Höhen seiner Kinder ist =< 1.

Satz: Die Höhe eines AVL-Baumes mit n Einträgen ist $\mathcal{O}(\log_2 n)$ und damit $\Theta(\log_2 n)$.

Beweis: n(h)=minimale Anzahl innerer Knoten eines AVL-Baumes mit Höhe h.
n(1)=1
n(2)=2

$$n(h)=1+n(h-1)+n(h-2)>fib(h)=\left(\frac{1+\sqrt{5}}{2}\right)^h>\sqrt{2}^h$$

daraus folgt, dass Einfügen/Suchen/Streichen in $\mathcal{O}(\log_2 n)$-Zeit erfolgen kann.

Aber: Die Balance-Eigenschaft kann dabei verloren gehen.

## Wiederherstellen der AVL-Struktur textuell

1. Bei Einfügen von Knoten *v* bleibt der Baum balanciert:
   Betrachte den Weg von *v* zur Wurzel
   Sei *z* der erste Knoten mit Höhendifferenz 2
   Nachfolger von *z* seien *y* und *x*
2. Bei Einfügen von Knoten *v* entsteht Ungleichgewicht:
   auf dem Weg zur Wurzel von *v* aus wird erstmals bei *z* Konflikt erzeugt: *y* sei Kind mit größerer Höhe und *x* sei Kind von *y* mit größerer oder gleicher Höhe

## Wiederherstellen der AVL-Struktur graphisch

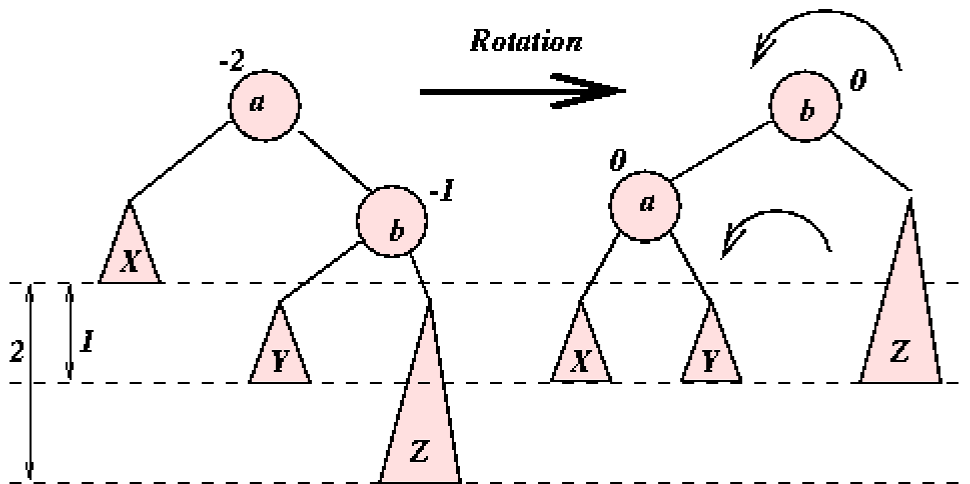
Beim Einfügen von Knoten in AVL-Bäumen, kann es in der Hinsicht zu Problemen kommen, dass nach dieser Operation kein AVL-Baum mehr vorliegt. Wenn man also die "Balance" als Differenz zwischen der Höhe des linken und rechten Teilbaumes ansieht, so kann sich beispielsweise aus

Derartige Effekte können durch die sogenannte *Rotation* oder auch *Doppelrotation* "repariert" werden. Verletzungen des AVL-Merkmals können also beispielsweise auftreten beim

1. Einfügen eines Knoten in den linken Teilbaum des linken Sohnes,
2. Einfügen eines Knoten in den rechten Teilbaum des linken Sohnes,
3. Einfügen eines Knoten in den linken Teilbaum des rechten Sohnes,
4. Einfügen eines Knoten in den rechten Teilbaum des rechten Sohnes,

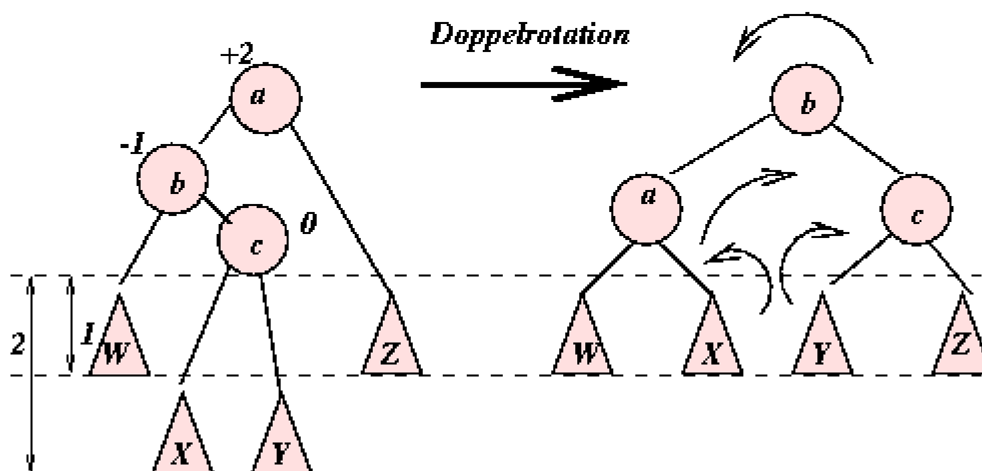
Die Rotationsoperationen seien bildhaft dargestellt:

## Rotation




Hierbei werden also zunächst eine Rotation mit dem rechten Kind nach links und dann eine analoge Operation nach rechts vorgenommen.

## Doppelrotation




Hierbei wird mit dem linken Kind die Doppelrotation nach rechts begonnen und die Rotation dann spiegelbildlich nach links doppelt ausgeführt. Sie wird zum Beispiel in den oben genannten Einfügefällen für die zweite und dritte Art verwendet.

Dazu siehe auch ein Applet unter www.ibr.cs.tu-bs.de/lehre/ss98/audii/applets/avlbaum/

Ziel der ausgeglichenen bzw. AVL-Bäume ist es, stets eine durchschnittliche Suchzeit von $\mathcal{O}(\log_2 n)$ zu erreichen.

### Kosten der Rotationen bei AVL-Bäumen

Alle Operationen (außer **`size`**, **`isEmpty`**) haben 2 Phasen: down (jeweils Kosten von $\mathcal{O}(1)$ pro Level; die Anzahl der Level ist immer $\mathcal{O}(\log_2 n)$ und up mit updates (jeweils Kosten von $\mathcal{O}(1)$ pro Level; die Anzahl der Level ist immer $\mathcal{O}(\log_2 n)$)

| Operation | Kosten |
|---|---|
| `size`, `isEmpty` | $\mathcal{O}(1)$ |
| `findElement`, `insertItem`, `removeElement` | $\mathcal{O}(\log_2 n)$ |

## *ADTs um den Such-Aufwand gering zu halten*

### Wörterbücher

### Balancierte Suchbäume

#### AVL-Bäume

#### Rot-Schwarz-Bäume

Der Rot-Schwarz-Baum ist ein binärer Suchbaum mit gefärbten Knoten (rot oder schwarz)
Regeln:

1. Jeder Knoten ist rot oder Schwarz
2. Jedes Blatt ist schwarz
3. Wenn ein Knoten Rot ist, dann sind die Kinder schwarz
4. Die Wurzel ist schwarz
5. Jeder Weg von einem Knoten zu einem tieferen Blatt hat gleichviele schwarze Knoten

#### (2,4)-Bäume

Keine Binärbäume, die inneren Knoten haben 2,3 oder 4 Kinder

## *Hashing*

Hashfunktionen wurden in der Informatik für schnelle Sortier- und Suchverfahren eingeführt. Die Geschwindigkeitssteigerung wurde durch die Komprimierung der Eingabedaten erreicht: Unterschiedliche Eingaben unterschiedlicher Länge erhalten gleiche Ausgabewerte (Hashwerte) gleicher Länge.
Dabei wird eine dynamische Menge S verwaltet, die sich in einem Universum U befindet. U hat sehr viele Elemente im Vergleich zu S.

###fehlendes###

Beim Hashing stellen sich 2 Probleme:
1. Welche Hashfunktion verwendet man?
Divisionsmethode: $h(k) = k \bmod m$, dabei muss $m$ prim sein
Beispiel zur Divisionsmethode
$h'(k) = k \bmod 13$, es soll die Liste (18, 41, 22, 44, 59, 32, 31, 73) verarbeitet werden:

|  |  | 41 |  |  | 18 | 44 | 59 | 32 | 31 | 73 |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

Dabei stellt sich das Problem des Clustering. Alternativ dazu kann man Chairing durchführen: Dabei wird ist ein Eintrag in eine Hashtabelle ein Verweiss auf eine verkettete Liste. Der Belegungsfaktor ist n/m (erreichte Listenlänge).

Die erwarteten Kosten der Operationen sind für die erfolgreiches Suche $\mathcal{O}\left(1+\frac{n/m}{2}\right)$, für die erfolglose Suche sind sie $\mathcal{O}\left(1+\frac{n}{m}\right)$

2. Wie behandelt man Kollisionen (mehrere Schlüssel in eine Position)
   - Offenes Adressieren: $h: U \times \{0,1,...,m-1\} \to \{0,...,m-1\}$; Wenn ein Slot schon existiert, dann suche weiter nach einem freien Slot!

Die Sondierungsfolge (sollte Permutation aus $S_m$ sein):
$\langle h(k,0), h(k,1), h(k,2), ..., h(k,m-1), \rangle$

- Lineares Sondieren: $h': U \to \{0,...,m-1\}$, Hashfunktion $h(k,i) = (h'(k)+i) \bmod m$
- Quadratisch: $h(k,i) = (h'(k)+c_1 i + c_2 i) \bmod m$

Weitere Möglichkeiten sind Dynamisches Hashing (die Tabellengröße wird zur Laufzeit angepasst) und Universelles Hashing (die Hashfunktion wird zufällig gewählt).

# Erweiterung Graphentheorie

## *Kürzeste Wege in Graphen (Dijkstra)*

Ein Beispiel ist das Ausrechnen des kürzesten oder des billigsten Weges von A nach B im Bahnnetz. Es sei $G = (V, E)$ ein gerichteter Graph

$w: E \to R$ ist die Gewichtsfunktion (z. B. Länge, Kosten,…)

$$w(Weg) = \sum_{e \text{auf.Weg}} w(e)$$

Start *s*, Ziel *t*. Die Aufgabe ist die Bestimmung des kürzesten (leichtesten) Wegs von *s* nach *t*.
Single-source-shortest-paths: Kürzeste Wege von *s* zu allen anderen Knoten!
Die Breitensuche ist ein Spezialfall von Single-source-shortest-paths mit Weg-Gewichten von 1. Die zentrale Eigenschaft von Kürzesten Wegen ist, dass sie wiederum Kürzeste Wege als Teilwege enthalten (diese Eigenschaft gilt nicht für längste Wege, also auch TSP-Problem!)

- Zentrale Eigenschaft: Kürzeste Wege enthalten wiederum Kürzeste Wege als Teilmenge
- Bestimmen eines Baums von kürzesten Wegen von s zu anderen Knoten. Diese werden als π-Zeiger abgespeichert: $\pi(v)$ Vorgänger von v auf kürzestem Weg von s nach v

### Algorithmus von Dijkstra

Greedy-Verfahren, analog zum Prim-MST
In jedem Schritt vergrößern wir die Menge S der Knoten, von denen wir die kürzesten Wege schon bestimmt haben. Ein Knoten $v \in V \setminus S$ (noch nicht erreicht) bekokmmt einen Schlüssel $d[v]$=Länge des bisher bekannten kürzesten Weges zu *v* (Initialisierungswert ist $\infty$). Die Kosten in $V \setminus S$ sind in einer Prioritätswarteschlange bezüglich $d[\ ]$ abgelegt. Dann wird das Minimum aus $V \setminus S$ verwendet.
Relaxierungsschritt: Überprüfe für alle Nachbarn v von u, ob $d[u] + w(u,v) < d[v]$. Wenn das gilt, dann update von $d[v]$ nach $\pi[v] = u$

Pseudocode für Dijkstra

```
01   Für v ∈ V(G)
02        do   d[v]<-∞
03             π[v]<-NIL
04   d[s]<-0
05   S<-Ø
06   Q<-V(G)
07   while Q≠ Ø
08   do   u<-Extract_Min(Q)
09        S<-Sv{u}
10        Für jedes v elem Adj(u)
11             do if     d[v]>d[u]+w(u,v)
12                  then d[v]<-d[u]+w,v)
13                       π[v]<-u
```