

INFORMATIK ZUSAMMENFASSUNG

Informatik = Problemlösen

Algorithmus

Def.1:

“Ein Algorithmus ist eine Vorschrift oder eine Anleitung zur Lösung einer Aufgabenstellung, die so präzise formuliert ist, dass man sie im Prinzip „mechanisch“ ausführen kann.”

Def.2:

“Ein Algorithmus ist eine präzise (d.h. in einer festgelegten Sprache abgefasste), endliche Beschreibung eines allgemeinen (vom Ausführenden interpretierbaren) Verfahrens zur Lösung einer Aufgabe / eines Problems unter Verwendung ausführbarer, elementarer Verarbeitungsschritte”

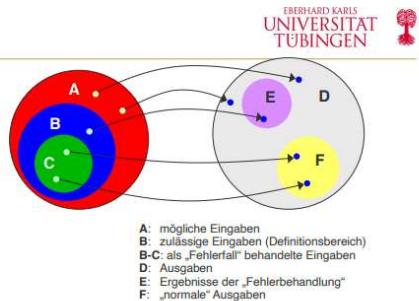
Eigenschaften Algorithmen:

- Die Anleitung ist **ausführbar und reproduzierbar** (das heißt, sie ist „operativ“)
- Besteht aus **Teilalgorithmen** (nicht mehr verfeinerte Ta. = elementare Operationen)
- Algo. Muss **hinreichend genau sein** = alle Schritte und deren Ausführungsreihenfolge muss feststehen.
- **Endlichkeit** der Beschreibung: Der vollständige Algorithmus muss in einem endlichen Text formuliert und aufgeschrieben sein.
- **Termination** des Algo.: muss endlich viele Schritte haben und zu einem Ende kommen.
- **Korrektheit**: Algo ist korrekt, wenn es für jede Eingabe aus dem Definitionsbereich der Funktion, die richtige Ausgabe erzeugt
- **Robustheit**: wie viele (vom problemlösenden Algorithmus aus gesehen) falsche Eingaben vom Programm aus erkannt und gemeldet werden.

Siehe Schaubild:

Korrektheit

- Ein Programm ist **korrekt**, wenn
 $B - C \rightarrow E$ („Fehlerfall“) und
 $C \rightarrow F$ („Normalfall“)
wie definiert implementiert sind.

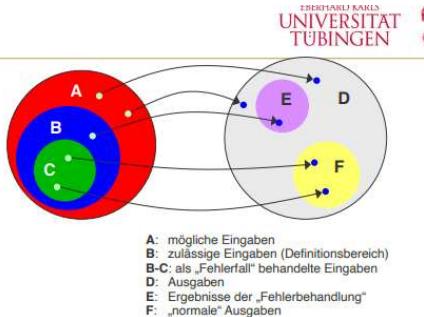


Anmerkungen:

- Ein Programm reagiert irgendwie auf jede denkbare Eingabe (die nicht im Definitionsbereich der Funktion liegen).
 - Dies ist die Abbildung $A - B \rightarrow D = (E \cup F)$.
 - Dieser Bereich könnte auch leer sein.
- Für die Korrektheitsbetrachtung ist es unwichtig, ob Teile der definierten Eingaben ($B - C$) als „falsch“ angesehen werden

Robustheit

- Die **Robustheit** ist eine Aussage darüber, wie viele (vom problem-lösenden Algorithmus aus gesehen) falsche Eingaben vom Programm aus **erkannt und gemeldet** werden ($|B - C| \geq \text{card}(B - C)$).



- „Extremwerte“:
 - Nicht robust ist ein Programm, in dem eine Behandlung von Eingaben, die eine Fehlermeldung als Ausgabe verursachen müssen, nicht vorgesehen ist ($B = C$ und $A - B$ nicht leer).
 - Ein Programm ist maximal robust, wenn es keine Eingabe gibt, die das Programm zu Fehlreaktionen veranlassen kann ($A = B$; Idealfall).

Ausführung eines Algorithmus = Prozess

- Ausführende Instanz = Prozessor

Prozessor muss:

- Sprache in dem Algo. Verfasst ist kennen
- Elementare Operationen verstehen und ausführen

Verschiedene Ausführungswege von TeilAlgo.:

- **nacheinander** (sequentiell, seriell)

- **gleichzeitig** (parallel)

- **beliebig** (kollateral)

Strukturelemente (Beschreibung eines Algo.):

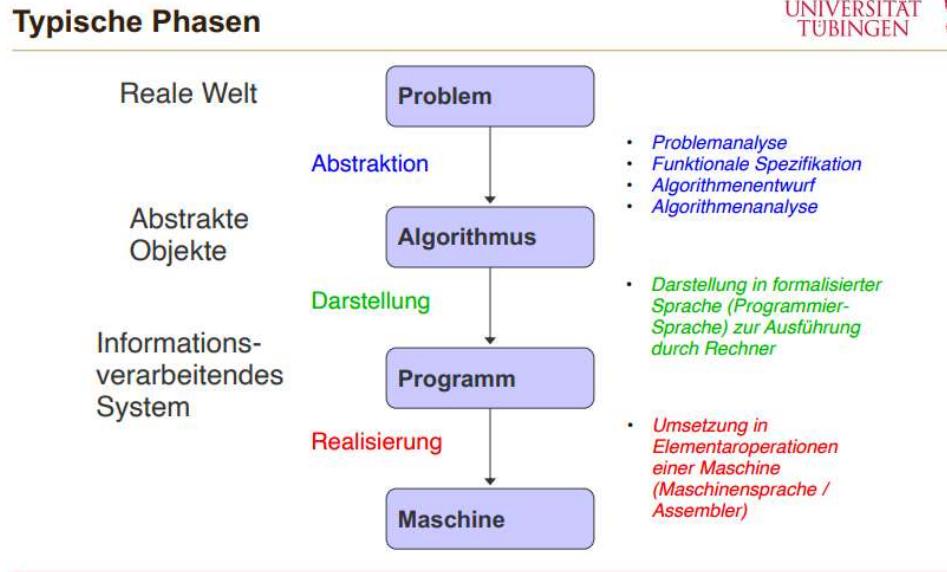
- Befehl bzw. Operation

- Befehlsequenz

- Wiederholungen von Befehlen oder Befehlsequenzen

-Bedingte Auswahl

Typische Phasen:



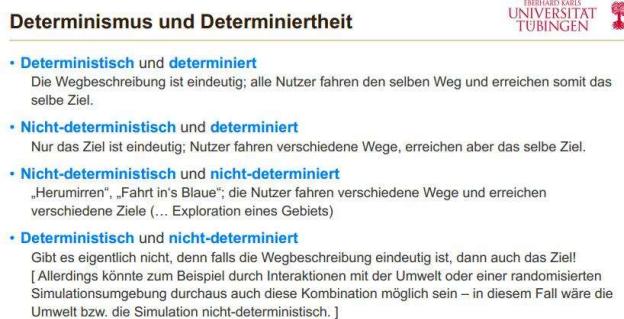
Bekannte Problemlösung:

-Divide and Conquer: -> Zerteilung in kleine Teilprobleme und diese Gleichzeitig ausführen
(Vorlesung Experiment Jüngste Person)

Determinismus und Determiniertheit

-deterministisch: Bei einer Eingabe immer den gleichen Pfad/Folge von auszuführenden Schritten

-determiniert: Bei gleicher Eingabe immer gleiche Ausgabe aber Pfad kann anderst sein.



Java Grundbegriffe

Programm = Folgen von Anweisungen

Methoden = Unterprogramme, die eine bestimmte Aufgabe in wiederverwendbare Form umsetzen.

Aufruf = Methode -> Stelle nachdem Aufruf

Klasse = "Bauplan" für eine Reihe von gleichartigen Objekten; die Klasse spezifiziert welche Attribute und Methoden ein Objekt besitzt

Objekt = Instanz einer Klasse

-Java besteht aus Reihe von Klassen/Methoden z.b.: "main"

-Name der Klasse muss dem Namen der Datei entsprechen.:

Class Helloworld -> Helloworld.java

- System.out.println() macht einen Zeilenumbruch am Ende des Textes der ausgegeben wurde.

Bezeichner

-Namen (identifier) für Datenobjekte:

wobei das erste Zeichen keine Ziffer sein darf.

Außerdem gibt es einige reservierte Wörter, die nicht als Bezeichner verwendet werden dürfen:

Teiler Bestandteil der Sprache Java:

```
abstract  default  if          private    this
boolean   do       implements  protected  throw
break     double   import      public     throws
byte      else     instanceof  return    transient
case      extends  int         short     try
catch     final   interface   static    void
char      finally  long        strictfp  volatile
class     float   native      super    while
const     for     new         switch
continue  goto   package    synchronized
```

Variablen:

Deklaration von Variablen:

```
int      index;  
String   name;  
boolean  isGreen;  
double   price;
```

Wertzuweisung:

```
index    = 5;  
name     = "Ben";  
isGreen  = false;  
price    = 3.5;
```

Ein Gleichheitszeichen bedeutet, dass der Wert des Ausdrucks auf der rechten Seite der Variable auf der linken Seite zugewiesen wird (anders als in der Mathematik)

Java kennt so genannte einfache Typen (primitive types):

byte, short, int, long :	für ganze Zahlen (Integer-Werte)
float, double :	für Gleitkommazahlen (reelle Zahlen in der verfügbaren Rechnergenauigkeit)
char :	Zeichen aus einem Zeichensatz (Unicode)
boolean :	logische Werte (true, false)

Eine Variable kann in Java ab dem Zeitpunkt ihrer Deklaration bis zum Ende der Methode verwendet werden.

```
int i;  
  
i = 3;      OK  
j = 4;      Fehler (bei Übersetzung)!  
int j;  
j = 6;      OK
```

-Variablen können in einer Klasse auch außerhalb von Methoden deklariert werden.

->Dann muss der Deklaration (vorerst) ein static vorangestellt werden. (Dadurch ist die Variable überall in der Klasse verwendbar. Es ist eine „Klassen-Variable“.)

```

public class Prog {
    static int k = 5;
    public static void main(String[] args) {
        k = 7;
        ...
    }
}

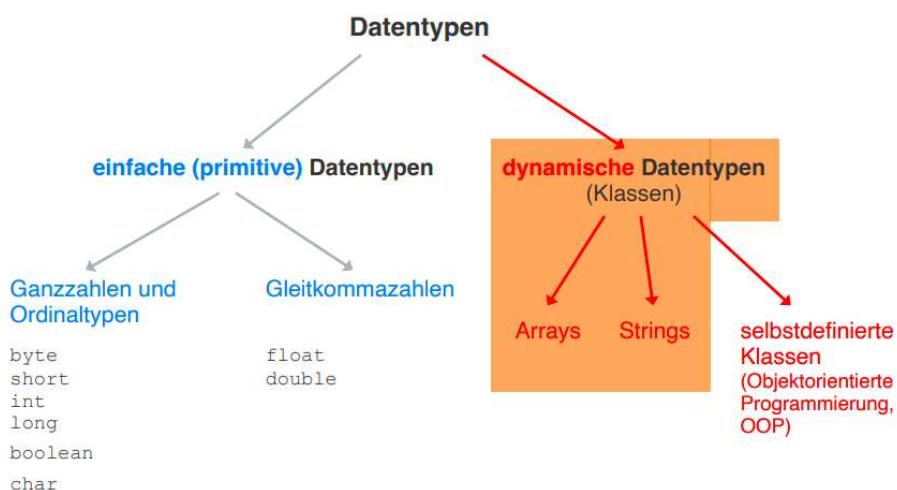
```

Literale

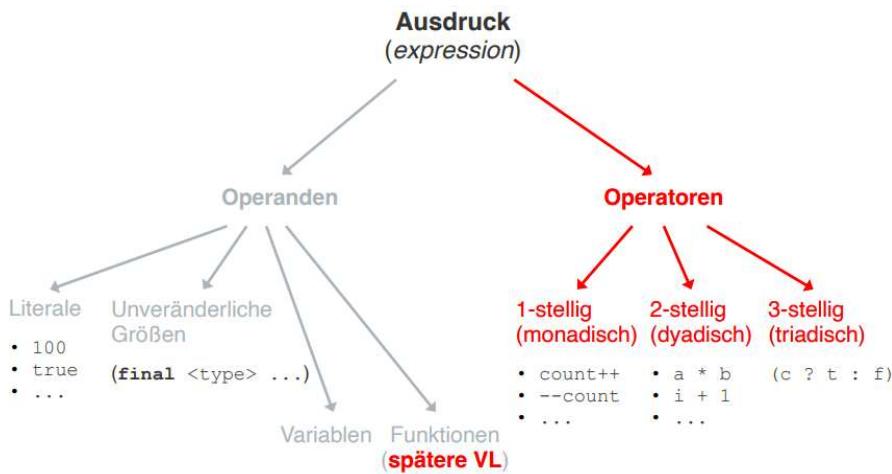
- Literale sind fest definierte konstante Werte, die innerhalb von Ausdrücken verwendet werden können Bsp: 674, 3.14, true, false, null
- Häufig soll eine Größe festgelegt werden, die einen festen unveränderlichen, d.h. während der Programmlaufzeit konstanten, Wert besitzt.
- Mit dem reservierten Java-Schlüsselwort final kann eine Konstante festgelegt werden.

Bsp.: final int MAX_COUNT = 10;

Konstanten werden in Groß-Buchstaben geschrieben. Bei Zusammensetzungen wird der Bezeichner mit Unterstrich zusammengefügt



Ausdrücke sind Kompositionen von Operanden und Operatoren, entsprechend mathematischer Gleichungen.



Aufgrund der sequenziellen Analyse und Auswertung kann die Variable selbst auch auf der rechten Seite auftreten (Verwendung ihres alten Wertes), bevor der neue Wert wieder an dieselbe Variable zugewiesen wird:

<code>a = 5 * 3;</code>	Inhalt von a: 15
<code>a = (a + 7) % 12;</code>	Inhalt von a: (15+7)%12 = 10

Weiterhin gelten die folgenden **Kurzschrifweisungen**:

<code>i -= a;</code>	entspricht:	<code>i = i - a</code>
<code>i *= a;</code>	entspricht:	<code>i = i * a</code>
<code>i /= a;</code>	entspricht:	<code>i = i / a</code>
<code>i %= a;</code>	entspricht:	<code>i = i % a</code>

- Für **Zählvariablen** werden häufig folgende Operatoren verwendet
 - Inkrement (`i++`) oder (`++i`) (entspricht `i += 1`)
 - Dekrement (`i--`) oder (`--i`) (entspricht `i -= 1`)

VORSICHT: Auswertung von Ausdrücken:

- **Inkrement** des Werts der Variable i:
 - `i++` nach Verwendung des Wertes von i
 - `++i` Inkrement vor Verwendung des Wertes von i
- **Dekrement** des Werts der Variable i:
 - `i--` nach Verwendung des Wertes von i
 - `--i` Dekrement vor Verwendung des Wertes von i

<code>int a = 6, b, c;</code>	
<code>b = a++ * 3;</code>	Inhalte der Variablen: b = 18 und a = 7
<code>c = ++a / 2;</code>	Inhalte der Variablen: a = 8 und c = 4

Tipp: Verwende **`++` / `--` -Operatoren** nur in Einzelanweisungen,
nicht in Ausdrücken!

Pre increment

$++v$

1. v wird manipuliert
" $v \rightarrow v+1$ "
2. v wird "gelesen"

$$v = 2; \quad$$

$$m = \underline{\underline{++v}}; \quad$$

$$m == 3 \quad$$

$$v == 3$$

Post increment

$v++$

1. v wird "gelesen"
2. v wird manipuliert

$$v = 2;$$

$$m = \underline{\underline{v++}}; \quad$$

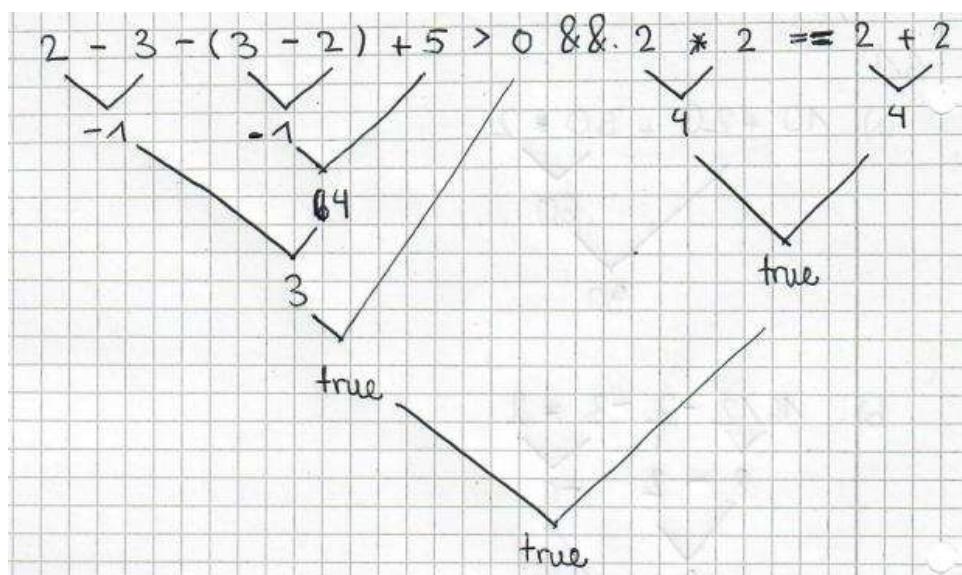
$$m == 2$$

$$v == 3$$

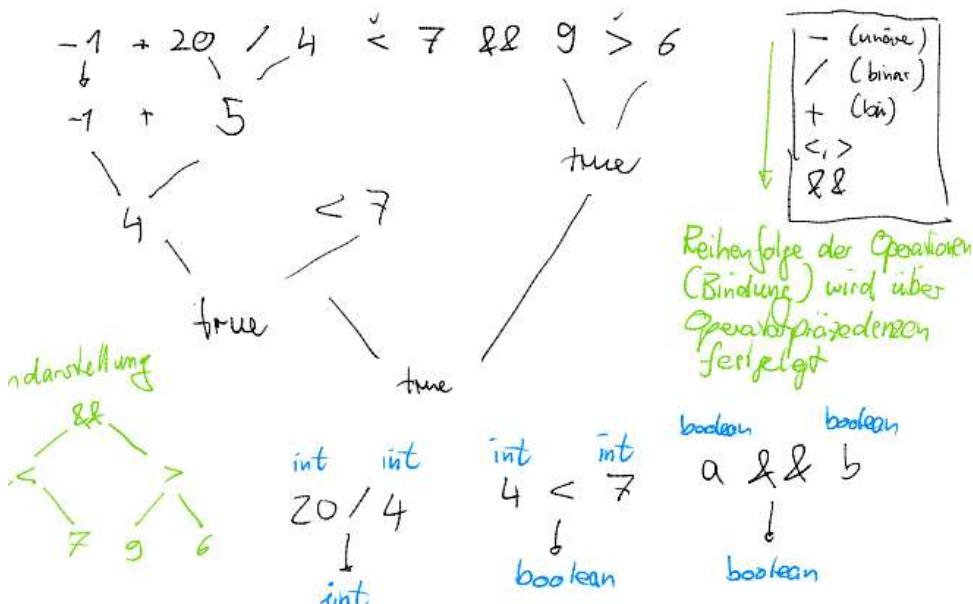
Anweisungen:

Imperativ = "schrittweise" Anweisungsverarbeitung -> **Variablenzuweisung** als zentrales Element

Beispiel wie eine Operation hier Schrittweise bearbeitet wird: (so auch in Klausur wahrscheinlich)



Beispiel aus der Vorlesung:



WICHTIG DIE REIHENFOLGE FÜR AUSWERTUNG BEACHTEN:

Die **Präzedenzregeln** der Anwendung von Operatoren bei der Auswertung von Ausdrücken sind folgendermaßen:

- 1 Einstellige Operationen: `++, --` (**Inkrement, Dekrement**),
(unäre)
positives (+) / negatives (-) Vorzeichen,
`!` (Negation), `~` (Bitweises Komplement)
(`<type>`) (**Typ casting**)
- 2 Multiplikative Operationen: `*`, `/`, `%`
- 3 Additive Operationen: `+`, `-`
- 4 Bitshift Operationen: `<<`, `>>`, `>>>`
- 5; 6 Vergleichsoperationen: `<`, `>`, `<=`, `>=`; `==`, `!=`
- 7 – 9 Bitweise Operationen: `&`, `^`, `|`
- 10 Konjunktion (logisch): `&&` (**logisch**)
- 11 Disjunktion (logisch): `||` (**logisch**)
- 12 Ternärer Operator: `? :`
- 13 Zuweisungsoperatoren: `=, +=, -=, *=, /=, %=, etc.`

Das gleiche und dasselbe

Bei primitiven Typen Prüfung auf Gleichheit via ==

`Int a = 10;`

`Int b = 20;`

`If(a == b) {...}`

Bei Objekttypen ist == problematisch

`String s1 = new String("abc");`

`String s2 = new String("abc");`

`Boolean v = (s1 == s2);`

--> Prüft dasselbe (gleicher Zeigerwert) aber nicht auf Gleichheit des Inhalts

Lösung: "equals"

- Stammt aus Object
- Bedeutung von Gleichheit kann individuell spezifiziert werden -> s1.equals(s2)

IF, else

Beispiel:

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

    int user = 21;

    if (user <= 18) {
        System.out.println("User is 18 or younger");
    }
    else if (user > 18 && user < 40) {
        System.out.println("User is between 19 and 39");
    }

    else {
        System.out.println("User is older than 40");
    }
}
```

“Else if” nur wenn erste “if” = false ist und “else” nur wenn “if und else if” = false sind.

Dangeling-if-Problem: Einrücken macht nix bei if else -> immer geschweifte Klammern benutzen

Ansonsten gibt's auch

TenaryOperator: (bei if else)

variable = (condition) ? expressionTrue : expressionFalse;

Anstatt so:

```
int time = 20;
if (time < 18) {
    System.out.println("Good day.");
} else {
    System.out.println("Good evening.");
}
```

Geht auch:

```
int time = 20;
String result = (time < 18) ? "Good day." : "Good evening.";
System.out.println(result);
```

Also String s = ()? () : ();

Switchcase:

- Die Verzweigung kann primitive byte-, short-, char- und int-Werte, Enums und Strings prüfen -> also nur ganzzahlige Typen
- kann keine long, float oder gar Objekte prüfen.
- Die switch-case-Verzweigung entspricht einer Mehrfachverzweigung mit if.
- Der Wert hinterm switch wird mit jedem case verglichen, wenn keins geht -> default

Beispiel:

```
public class SwitchCaseClass {  
  
    public static void main(String[] args){  
        int i=2;  
  
        switch(i){  
            case 0:  
                System.out.println("i ist null");  
                break;  
            case 1:  
                System.out.println("i ist eins");  
                break;  
            case 2:  
                System.out.println("i ist zwei");  
                break;  
            case 3:  
                System.out.println("i ist drei");  
                break;  
            default:  
                System.out.println("i liegt nicht zwischen null und drei");  
                break;  
        }  
    }  
}
```

Es gehen auch mehrere cases:

```
int n;

switch ( n ) {
    case 1:
        System.out.println("Die Zahl ist 1.");
        break;
    case 2:
    case 4:
    case 8:
        System.out.print("Die Zahl ist 2, 4 oder 8.");
        System.out.println("(Vielfache von 2)");
        break;
    case 3:
    case 6:
    case 9:
        System.out.print("Die Zahl ist 3, 6, oder 9");
        System.out.println("(Vielfache von 3)");
        break;
    case 5:
        System.out.println("Die Zahl ist 5.");
        break;
    default:
        System.out.println("Die Zahl ist 7 oder außerhalb des " +
                           "Sbereiches 1 bis 9.");
}
```

For Schleife

Kennst ja

```
1 | for(int i=1;;i++) {
2 |     System.out.println(i);
3 |     if(i%10==0) {
4 |         break;
5 |     }
6 | }
```

Hier wird innerhalb der Schleife eine Abbruchbedingung erstellt, geht auch.

Oder:

```
for (int i = 0; i <= 10; i = i + 2) {
    System.out.println(i);
}
```

Weiter Variante:

For (int x : array) {

...

}

Hier kannst auch über das array iterieren hast aber nicht index wie bei for each.

While Schleife

```
let x = 1;
while (x < 5) {
    console.log ('Lauf ' + x );
    x = x + 1;
}
```

Die do..while-Schleife wiederholt eine Anweisung, bis eine Bedingung nicht mehr erfüllt ist

```
do {
    Anweisungen;
} while (Bedingung)
```

Es gibt immer wieder Ausnahmesituationen, in denen es effizienter ist, die while-Schleife vor ihrem Ende zu verlassen:

- Entweder um den Rest des Code-Blocks auszulassen und sofort den nächsten Schleifendurchlauf zu beginnen,
- oder um die Schleife vorzeitig abzubrechen.

Dafür stellt Javascript die Anweisungen break (Schleife sofort abbrechen) und continue (sofort wieder an den Anfang der Schleife) zur Verfügung.

while vs. do..while Schleife

- Bei der do..while Schleife wird der Schleifenkörper auf alle Fälle einmal ausgeführt, auch wenn die Bedingung bereits zu Beginn nicht erfüllt ist. (Hinweis: Aus diesem Grund ist die do..while Schleife nur mit großer Vorsicht anzuwenden, da sie eine häufige Fehlerquelle darstellt)

Datentypen

Ganze Zahlen

Zahlen mit positivem und negativem Wertebereich

- Wertebereiche für ganze Zahlen

• byte	[-128, ..., 127]
• short	[-32.768, ..., 32.767]
• int	[-2.147.483.648, ..., 2.147.483.647]
• long	[-2 ⁶³ , ..., 2 ⁶³ -1]

- Byte 8 bit/ short 16 bit / int 32 bit / long 64 bit

Vorsicht: Arithmetische Operationen dürfen den Wertebereich des Datentyps nicht überschreiten.

-->Konsequenzen: Bei einer Überschreitung kommt es je nach Programmiersprache zum

- Programmabbruch (Überlauf) oder
- die Variable erhält einen anderen (falschen) Wert

Weiter Notizen:

Int b = 0x.... durch den Präfix wird Hexidezimal angefangen

Int c = 01234 mit 0 initiert man ein Octal /Ausgabe wäre “668”

Binärsystem/-zahlen

- Computer arbeiten nicht im Dezimalsystem sondern im Binärsystem (nach G.W. Leibniz). Computer kennen nicht, wie der Mensch, die 10 Ziffern 0, 1, ..., 9 sondern nur die zwei Ziffern 0 und 1.

- Im Binärsystem lässt sich auf ähnliche Weise rechnen wie im Dezimalsystem

Zahl in ein Binärdarstellung:

1. Teile die Zahl mit Rest durch 2.
2. Der Divisionsrest ist die nächste Ziffer (von rechts nach links).
3. Falls der (ganzzahlige) Quotient = 0 ist, bist du fertig,
andernfalls nimm den (ganzzahligen) Quotienten als neue Zahl
und wiederhole ab (1).

Bsp.: Die Dezimalzahl 67 wird ins 2er-System umgewandelt

$$67 : 2 = 33 \text{ Rest: } 1$$

$$33 : 2 = 16 \text{ Rest: } 1$$

$$16 : 2 = 8 \text{ Rest: } 0$$

$$8 : 2 = 4 \text{ Rest: } 0$$

$$4 : 2 = 2 \text{ Rest: } 0$$

$$2 : 2 = 1 \text{ Rest: } 0$$

$$1 : 2 = 0 \text{ Rest: } 1$$

Resultat: 1000011

Binär wieder zurück in Zahl (hexa):

$$1 \cdot 1 = 1$$

$$1 \cdot 2 = 2$$

$$0 \cdot 4 = 0$$

$$0 \cdot 8 = 0$$

$$0 \cdot 16 = 0$$

$$0 \cdot 32 = 0$$

$$1 \cdot 64 = 64$$

$$\rightarrow 1 + 2 + 64 = 67$$

$$\begin{array}{r}
 -4 + 6 : \quad 1011 \\
 \quad \quad \quad 0110 \\
 \hline
 \quad \quad \quad 0001 \\
 \quad \quad \quad \rightarrow \text{ist eins} \\
 \quad \quad \quad \text{aber noch 1 Übertrag also}
 \end{array}$$

$$\begin{array}{r}
 0001 \\
 0001 \\
 \hline
 \end{array}$$

0010 --> ist jz zwei und richtig

Darstellung von Vorzeichen :

bei einem Byte das erste Zeichen: 1 -> negativ, 0 -> positiv

Zahl in Einerkomplementdarstellung:

1. Zahl in Binärform bringen
2. Alle bits invertieren ('0 wird zu 1 und 1 zu 0)
3. Vorzeichen 0 oder 1 hinzufügen

1111 und 0000 sind stellen beide die 0 dar im Einerkomplement

Kann sein z.B. bei $-4 + 6$ beim verrechnen eine 1 immer rüber und dann hat man ja 1 eins vorne mehr aber die verrechnen einfach nochmal (aber wenn du mehr als 4 stellen oder so hast -> einfach halt vorne ran weiter verrechnen)

Bsp +5 -> Binär: 0101 Einerkomplementdarstellung: 1010

Zahl in Zweirkomplementdarstellung:

1. Zahl in Binärform bringen
2. Hinzufügen der führenden Null (um das Vorzeichenbit zu kennzeichnen)
3. Alle Bits invertieren
4. 1 addieren

Bsp +5 -> <-Falsch

1. Binär: 0101
2. Invertieren: 1010
3. 0 hinzufügen: 01010
4. 1 addieren: 01011

Weitere Bsp.:

Also 5 in Binär ist einfach nur 101

In Einerkomplement entweder 0 oder 1 je nach Vorzeichen:

$$+5 = 0101 \quad -5 = 1101$$

und dann invertieren:

$$+5 = 1010 \quad -5 = 0010$$

Manchmal wird gefragt on byte = 8 stellen oder 4 bit = 4 Stellen etc. dann einfach nullen adden vorne.

also in byte wäre zweierkomplement:
 $+5 = 0000\ 1011$

Beim Rechnen zuerst alles in Zweierkomplement bringen und dann halt verrechnen !ACHTUNG: Wenn man mit negativen Zahlen rechnet nicht führende 1 oder null hinzufügen sondern negativ machst du ja indem du invertierst und 1 addest siehe Bsp unten

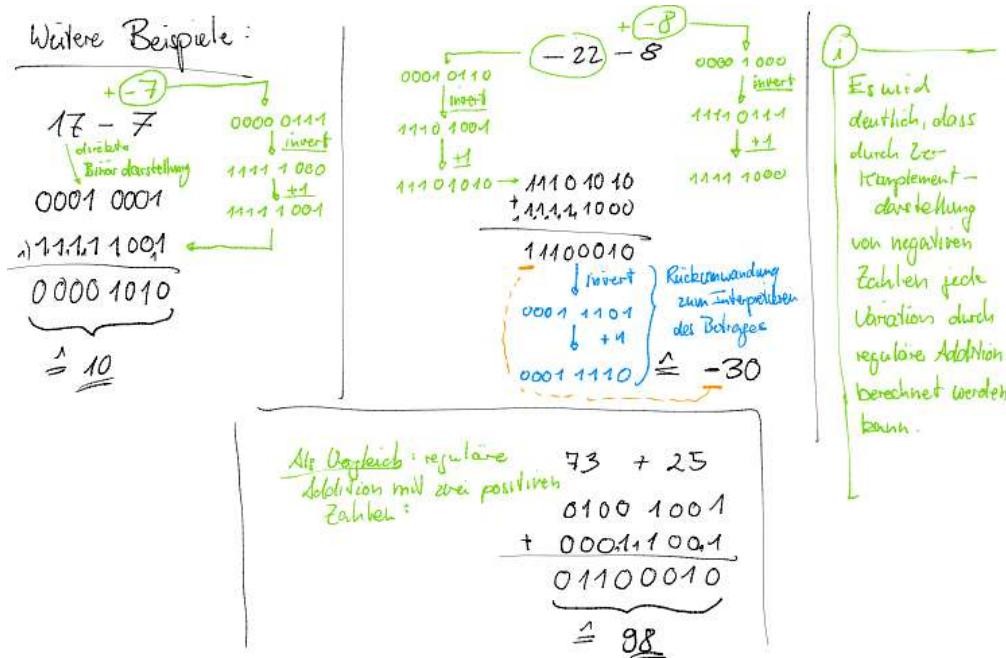
In Zweierkomplement
das oben einfach machen und eine 1 addieren.

$$\begin{array}{r}
 +5 = \\
 1010 \\
 1
 \end{array}$$

$$\begin{array}{r}
 1011
 \end{array}$$

!!!Wenn mal negatives Rauskommt wie und Bsp, dann Wieder rückwandeln 1. Invertieren 2. +1 dazurechnen

Beim Rest Rechnen von unten nach oben lesen ist von rechts nach links



Gleitkommazahlen (float, double)

- Näherungsweise Darstellung der rationalen Zahlen
- Wertebereich float [$-3.4028235 \times 10^{38}$; 3.4028235×10^{38}]

double [$-1.797693 \times 10^{308}$; 1.797693×10^{308}]

- Zahlen die ein “.” enthalten werden von java automatisch als double interpretiert Bsp.: -1.5 oder 100.00

Formatangaben: Eine Zahl kann als float gekennzeichnet werden, wenn ihr ein ‘f’ (‘F’) angehängt wird Bsp.: 1f 1.e12f 3.14F

Mit ‘d’ (‘D’) wird sie als double gekennzeichnet Bsp.: 5d 17.208E-3d 2.D

Spezielle Werte

- Positiv / negativ Unendlich:
`Float.NEGATIVE_INFINITY`
`Float.POSITIVE_INFINITY`
- Not-a-Number
`Float.NaN`
- Bsp.: Division durch Null

```

float fval1 = 10.0;
float fval2 = 0.0;
float fval3 = 0.0;
float fres1, fres2;

fres1 = fval1 / fval2;   Ausgabe: Infinity
fres2 = fval2 / fval3;   Ausgabe: NaN
    
```

- Demo: InfinityAndNotANumber.java

Primitive Datentypen in Java:

void
boolean
char
byte
short
int
long
float
double
(String)

Vorsicht bei Vergleichen:
Es gilt immer NaN ungleich NaN

- Die Mantisse besitzt eine feste Anzahl von Stellen.
- Je größer die Zahl vor dem Dezimalpunkt, desto weniger Nachkommastellen sind darstellbar.

- Sind nicht alle Nachkommastellen einer Zahl darstellbar, so wird auf die nächste darstellbare Zahl gerundet.

Wahrheitswerte (boolean)

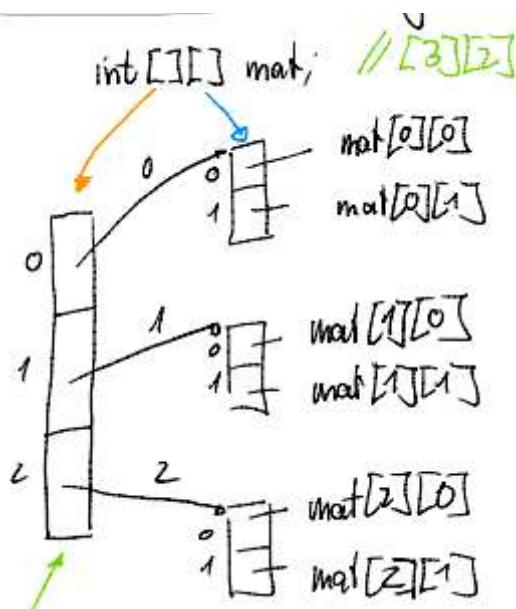
Wahrheitstabelle:

		Negation	Konjunktion und (and)	Disjunktion oder (or)	exklusives oder (Xor)
Boolean a	Boolean b	$\neg a$	$a \&& b$	$a \mid\mid b$	$a \wedge b$
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

Array (Felder)

Bsp.: $\text{int[]} \text{ array} = \underline{\text{new}} \text{ int}[10];$ mit wert
gegebenen Elter
 $\text{int[]} \text{ array} = \underline{\text{new}} \text{ int}[] \{1, 2, 3, 4, 5\}$

Mehrdimensionale Arrays:



- Geht auch mit mehr `int[][][]... 2-Diemensionales Array kann man als Matrize sehen.`
- Subarray können auch Nulls sein/ nicht jeder Zweig muss Elemente haben

- String is eig ein array aus char (str.charAt(3) -> char eines Strings an der Stelle 3)

Enumerations

Wenn man einschränken will was einer Methode oder so gegeben wird/
Variablen Einschränkung durch vordefinierte KONSTANTEN im enum

```
public class Main {  
    enum Level {  
        LOW,  
        MEDIUM,  
        HIGH  
    }  
}
```

Wie Accessen mit: Level.MEDIUM zum Beispiel:

```
public static void main(String[] args) {  
    Level myVar = Level.MEDIUM;      Hier kannst du dem Enum Ding was deklarieren  
    System.out.println(myVar);  
}  
}
```

Hier kannst du Einschränken was die Methode nimmt (also hier nur die Enums)

Weiteres Beispiel:

```
enum Level {  
    LOW,  
    MEDIUM,  
    HIGH  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Level myVar = Level.MEDIUM;  
  
        switch(myVar) {  
            case LOW:  
                System.out.println("Low level");  
                break;  
            case MEDIUM:  
                System.out.println("Medium level");  
                break;  
            case HIGH:  
                System.out.println("High level");  
                break;  
        }  
    }  
}
```

```
public void printLevel(Level x) {  
    System.out.println("Dein Level ist: " + x);  
}
```

Hier kannst dem Enum auch mehrere Dinger zuweisen noch aber brauchst auch Konstruktor dann wo halt die einzelnen Sachen nimmt :

```
enum Wochentag {  
    MONTAG("Erster Tag der Woche", true),  
    DIENSTAG("Zweiter Tag der Woche", true),  
    MITTWOCH("Dritter Tag der Woche", true),  
    DONNERSTAG("Vierter Tag der Woche", true),  
    FREITAG("Fünfter Tag der Woche", true),  
    SAMSTAG("Sechster Tag der Woche", false),  
    SONNTAG("Letzter Tag der Woche", false);  
  
    // Instanzvariablen  
    private final String beschreibung;  
    private final boolean arbeitstag;  
  
    // Konstruktor  
    private Wochentag(String beschreibung, boolean arbeitstag) {  
        this.beschreibung = beschreibung;  
        this.arbeitstag = arbeitstag;  
    }  
  
    // Getter-Methoden  
    public String getBeschreibung() {  
        return beschreibung;  
    }  
  
    public boolean isArbeitstag() {  
        return arbeitstag;  
    }  
}
```

LOW.name() -> liefert Name des enumElements

LOW.ordinal() -> liefert “interne” Nummer des enumElements hier 0

Import/Packages

```
import package.name.Class; // Import a single class  
import package.name.*; // Import the whole package
```

Static Methoden/Variablen

- kann in Klasse verwendet werden, ohne dass eine Instanz der Klasse erstellt werden muss.

-Vorteile von static -> memory speicher wird gesavet

Static Variable:

- Wenn eine Variable als "static" deklariert wird, wird sie einmalig im Speicher erstellt und kann von jeder Instanz der Klasse verwendet werden.
- Änderungen an einer statischen Variable wirken sich auf alle Instanzen der Klasse aus.

Beispiel:

Static Methoden:

- Statische Methoden sind spezielle Funktionen, die direkt aufgerufen werden können, ohne dass eine Instanz der Klasse erstellt wird.
- Statische Methoden können nur auf andere statische Methoden oder Variablen zugreifen und nicht auf Instanzvariablen.

Static Methoden Beispiel:

```
public class Main {  
    // Static method  
    static void myStaticMethod() {  
        System.out.println("Static methods can be called without creating  
objects");  
    }  
  
    // Public method  
    public void myPublicMethod() {  
        System.out.println("Public methods must be called by creating  
objects");  
    }  
  
    // Main method  
    public static void main(String[ ] args) {
```

```

myStaticMethod(); // Call the static method
// myPublicMethod(); This would output an error

Main myObj = new Main(); // Create an object of Main
myObj.myPublicMethod(); // Call the public method
}
}

```

Call-by-Value

- Form des Methodenaufrufs, bei dem Kopien von Werten und nicht die Werte selbst als Parameter übergeben werden.

```

public static int inc(int x) {
    x = x + 1;
    return x; [Diese Zuweisung verändert
               a (Argument von inc) nicht!]
}

```

"Polymorphie" auf die Fähigkeit von Objekten und Klassen, sich je nach Kontext unterschiedlich zu verhalten.

Polymorphie (Early Binding)

Public static int f1(int a, boolean b, double c)

Definition Sigantur besteht aus:

- **Methodename "f1"**
- **Anzahl Typen und der Reihenfolge der formalen Parameter "int", "boolean", "double"**

Public static int **f1(int a, boolean b, double c)**

Nicht aus:

- Parameternamen
- Rückgabetypr

~~Public static int f1(int a, boolean b, double c)~~

Methodenüberladung

In Namensraum (Klasse) existieren mehrere Methoden mit **dem selben Namen** aber unterschiedlicher Signatur

--> Die Zuordnung, welche Methode aufgerufen wird, geschieht zur Übersetzungszeit und bleibt dann fix --> Early Binding

Java Speicher-organisation

Der Heap-Speicher ist der Bereich im RAM, in dem Java Objekte und Arrays speichert, die während der Programmausführung erstellt werden

- Hier liegt alles was mit "new" erzeugt wird -> Speicherallokierung
- Bsp.: int[] a = new int [10] "a" ist Adresse bzw. Pointer / Array-Daten liegen irgendwo auf Heap
- Heap-Größe durch VM begrenzt (Virtual Machine)
- Garbage Collection hält Ordnung (löscht nicht mehr verwendete Daten und gibt Speicher frei)

Stack: Der Stack-Speicher ist wie ein Stapel von Aufgaben, den Java verwendet, um zu wissen, welche Methode gerade ausgeführt wird und wo sie ihre Variablen und lokalen Daten speichert.

- Staplespeicher Elemente werden oben draufgelegt. Zugriff nur auf das oberste Element.
- Organisiert in "Frames" Jeder Methodenaufruf hat einen eigenen "Frame" -> Kontext
- Frame enthält - lokale Variablen - Rücksprungadresse
- Nur oberster Frame zugreifbar
- Beim Verlassen eines Methodenaufrufs (Rücksprung) wird aktueller Frame gelöscht

Obejktorientierte Programmierung

Zentrales Element: Klasse -> Bauplan/Schablone für Konkrete Objekte/Instanzen

Public = überall global sichtbar

Private = nur in der spezifischen Klasse

--> Bsp um Werte nur zu lesen sie aber nicht zu verändern (get/set-Methoden)

- Konstruktor (ist standartmäßig vorhanden)
- Spezialisierter Konstruktor (mit Argumenten) -> wenn ein spezial Konstruktor eingeführt wird und standart Konstruktor nicht wird nur spezial Konstruktor genutzt.

	Klasse	Objekt
Variable	<p>Klassenvariable</p> <ul style="list-style-type: none"> • statische Variable • existiert einmal <p>Bsp:</p> <pre>class Foo { static int a; Foo.a = 10; }</pre>	<p>Instanzvariable</p> <ul style="list-style-type: none"> • existiert je Objektinstanz <p>Bsp:</p> <pre>class Foo { int a; Foo foo = new Foo(); foo.a = 10;</pre> <p>innerhalb der Klasse (in Instanzmethode):</p> <pre>this.a = 10;</pre>
Methode	<p>Klassenmethode</p> <ul style="list-style-type: none"> • statische Methode • wird im Kontext der Klasse gerufen <p>Bsp:</p> <pre>class Foo { static void bar() {...} } foo.bar();</pre>	<p>Instanzmethode</p> <ul style="list-style-type: none"> • wird im Kontext einer Objektinstanz gerufen <p>Bsp:</p> <pre>class Foo { void bar() {...} } this.bar();</pre> <p>innerhalb der Klasse (in Instanzmethode):</p> <pre>this.bar();</pre>

Standartkonstruktor:

Wahl und Verlieferung Konstruktoren

Standardkonstruktor

```
public class Foo {
    public Foo() {
    }
}
```

Foo f = new Foo();

Spezialisierte Konstruktoren

```
public Foo(int a, int b)
```

}

→ Standardkonstruktor abaktiviert
(muss explizit definiert werden)

```
Foo f = new Foo(1, 2);
```

```
public Foo(double a, int b) {
```

}

i Konstruktionen können wie Methoden überladen werden

Private Konstruktor:

-Instanzen können nur noch innerhalb der Klasse erzeugt werden (Regulation der Insanzerzeugung siehe Singleton)

Private Konstruktor

```
public Foo {
    private Foo() {
    }
}
```

Folge: Instanzen können nur noch innerhalb der Klasse erzeugt werden.

→ Lässt sich zur Regulation der Instanzierung einsetzen

Bsp. Singleton Erweiterungsmuster

Singleton → Erlaubt nur eine Instanz einer Klasse

public class Singleton {
 private Singleton() { ... }
}

private static Singleton instance = null;

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

Singleton s = Singleton.getInstance();

Singleton s2 = Singleton.getInstance();

Aufruf außerhalb der Klasse nicht möglich

Aufrufe

liefen die

selbe

Instanz

Mann kann auch anstatt Daten in Arrays speichern mit Klassen sowas wie records in Racket machen und von da eben aufrufen.

UML-Klassendiagramm (Unified Modeling Language)

Hund
- name : String - gewicht : int - hundZaehler : int
+ bellen() : void + setName(nameHund : String) : void + getName() : String + setGewicht(gewichtHund : int) : void + getGewicht() : int

- Sichtbarkeit protected mit: # Sichtbarkeit default (package) mit: ~
- Im UML Klassendiagramm werden Klassenvariablen mit Hilfe eines Unterstrichs gekennzeichnet.
- Abstrakte Sachen werden kursiv gekennzeichnet

- Extends normales pfeil
- Implements (von Interface) gestrichelter pfeil

Erstellen Sie ein UML-Klassendiagramm, welches folgende Typen umfasst:

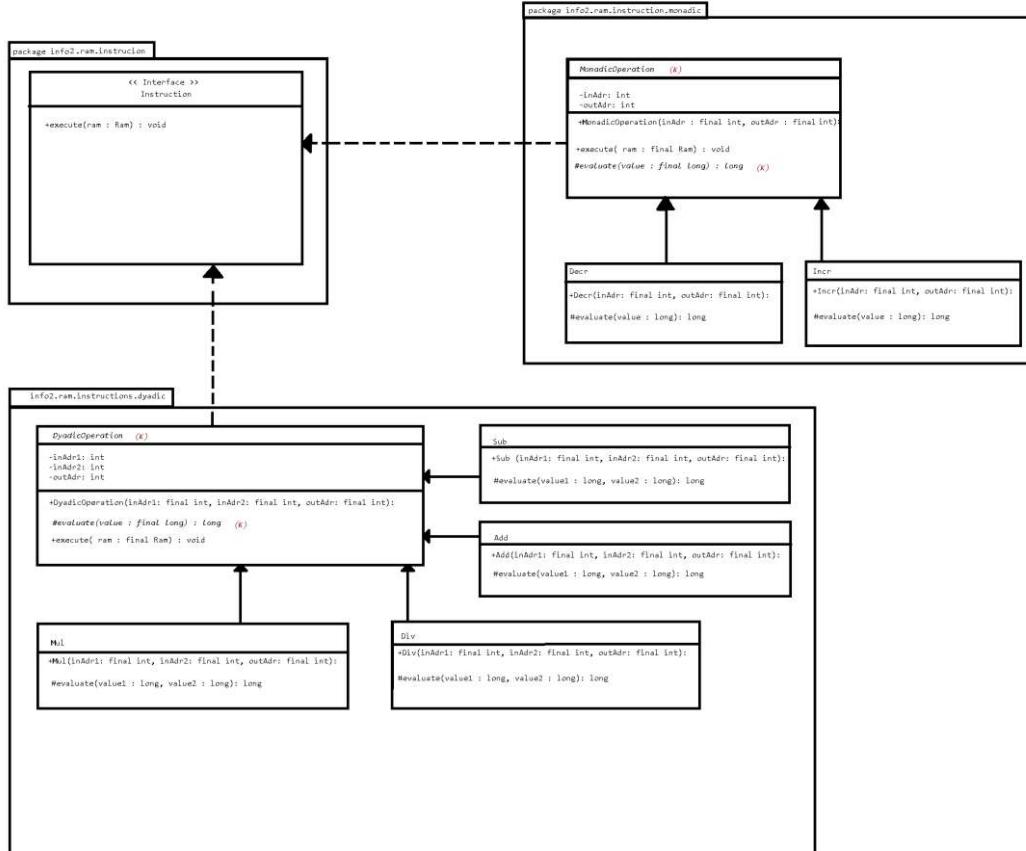
Instruction,
MonadicOperation,
Incr,
Decr,
DyadicOperation,
Add, Sub, Mul and Div.

(K) = Kurziv

Ignorieren Sie alle hier

nicht aufgeführten Klassen. Berücksichtigen Sie ferner die nachfolgenden Punkte:

- Zeichnen Sie alle wichtigen Abhängigkeiten der Typen untereinander ein.
- Stellen Sie Methoden (inkl. Konstruktoren) und Instanzvariablen inklusive Sichtbarkeiten dar.
- Erfassen Sie auch die jeweiligen Packages (und Subpackages) der genannten Typen.

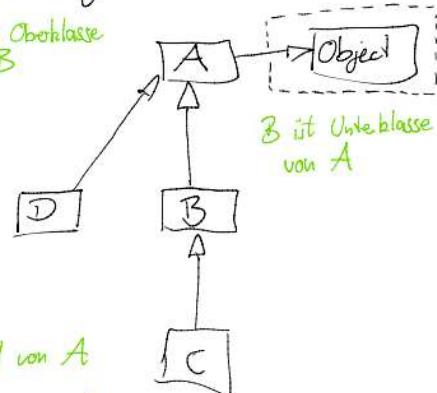


Vererbung

Vererbung

Synonyme:
Oberklasse, Superklasse, Elternklasse, Basisklasse

A ist Oberklasse von B



public class B extends A {
};

public class C extends B {
};

public class D extends A {
};

Berbt von A
B erweitert A

! Jede Klasse erbt von Object!

- Jedes B erbt ein vollständiges A (Alles was A kann, kann auch b inkl. Variablen)
- Man kann auch eine Variable von A nehmen und in B deklarieren
- Mit "super" wird alles davor/ Oberklasse adressiert
- Mit "this" immer das jetzige wo man drin ist

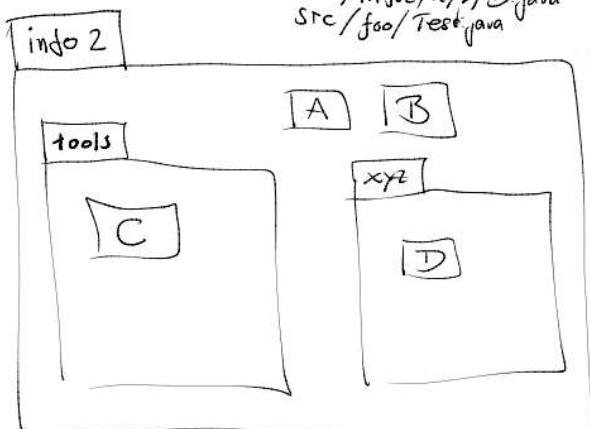
Packages

Packages

Ordnerstruktur:

```

src/info2/A.java
src/info2/B.java
src/info2/tools/C.java
src/info2/xyz/D.java
src/foo/Test.java
  
```



A.java

package info2;

:

import info2.A;

1 Zeile in Datei
indirekt
Package zugeordnet

ist nötig,
weil A

auf der Seite von
info2 verwendet
werden soll

C.java

package info2.tools;

- Struktureinheiten, welche Klassen die miteinander Zusammenhängen gruppiert.
- Packagename immer kleingeschrieben
- Wenn man nicht im Package ist muss man "import" nutzen
- Man kann auch Subpackages bilden in einem package (bsp. Subpackage tools --> info2.tools)

- Wenn man nicht public/private Sichtbarkeit schreibt bei Methoden ist automatisch sichtbarkeit "Package" -> kann nur im Package benutzt werden (auch nicht subpackage)
- Sichtbarkeit "protected" Methode ist im eigenen Package aber auch von allen erbenden Unterklassen sichtbar.

Überschreiben von Methoden (override)

Überschreiben von Methoden (override)

```
public class Base {
    public String foo() {
        return ...
    }
}

public class Child extends Base {
    @Override optionale (aber wichtige) Annotation
    public String foo() {
        ...
    }
}
```

• spezialisierte Implementierung von foo() überschreibt die Impl. in Klasse Base
• Bei Aufruf wird immer die zuletzt überschriebene Impl. aufgerufen

- damit ist foo überschrieben
- Base b = new Child(); --> b.foo() foo von Child
- Base d = new Base(); --> d.foo() foo von Base
- Überschreiben von Methoden ist weitere Ausprägung von Polymorphie in Java
- METHODEN DIE NICHT ÜBERSCHRIEBEN WERDEN KÖNNEN: private, static, final
- Mit "super.foo()" kann man nach dem override wieder auch aufs foo() von Base zugreifen

Final-Methoden

- Final verhindert, dass Methoden später nochmal überschrieben werden darf.
- Geht auch bei Klassen -> was bedeutet keine weitere Klasse darf von dieser erben

Vorteile von Final

- Optimieren/schnellerer Code
- Umgehen von Late-Binding zu viel schnelleren Early-Binding

Early Binding vs Late Binding

Early Binding (static Binding)

"zur Übersetzungszeit"

Betrifft:

Methodenüberladung
Konstruktorüberladung

Beim Early Binding weiß der Compiler bereits zur Kompilierungszeit, welche Methode aufgerufen wird, und es gibt keinen zusätzlichen Aufwand zur Laufzeit, um die geeignete Methode zu bestimmen.

Late Binding das Gegenteil: erst zur Laufzeit wird bestimmt

- Statische Methoden
- Private Methoden
- Final Methoden
- Auswahl bei überladenen Methoden

Late Binding (dynamic Binding)

“zur Laufzeit”

Betrifft:

- Alle Methoden, die überschrieben werden können
- Methodenüberschreibung
- Vererbung

Abstrakte Klassen

- Eine abstrakte Klasse wird durch das Schlüsselwort `abstract` angezeigt
- Eine abstrakte Methode besteht nur aus einer Methodendeklaration, ohne Methodendefinition
- Eine Klasse, die mindestens eine abstrakte Methode enthält, ist auch abstrakt (muss als `abstract` gekennzeichnet werden)

Beispiel:

```
public abstract class A {
    public abstract String getName();
    public void test() {
        System.out.println(
            "Name: " + this.getName());
    }
}
```

**1 Deklaration der Methode
2 Definition der Methode

A a = new ~~A()~~; Nur konkrete (nicht-abstrakte) Klassen dürfen instantiiert werden.

Hier wird abstract Methode konkretisiert:

```

public class B extends A {
    @Override
    public String getName() {
        return "B";
    }
}

A b = new B();
b.test();

```

C
Kurs

abstrakte Methode
getName wird durch
Implementierung konkret

Jetzt erlaubt, da alle
abstrakten "Versprechen"
erfüllt sind!

Interfaces (Schnittstellen/ Schnittstellenklassen)

- Vollständig Abstrakt
- Schnittstelle zwischen unserem code und anderem code

```

public interface Foo {
    int bar();
}

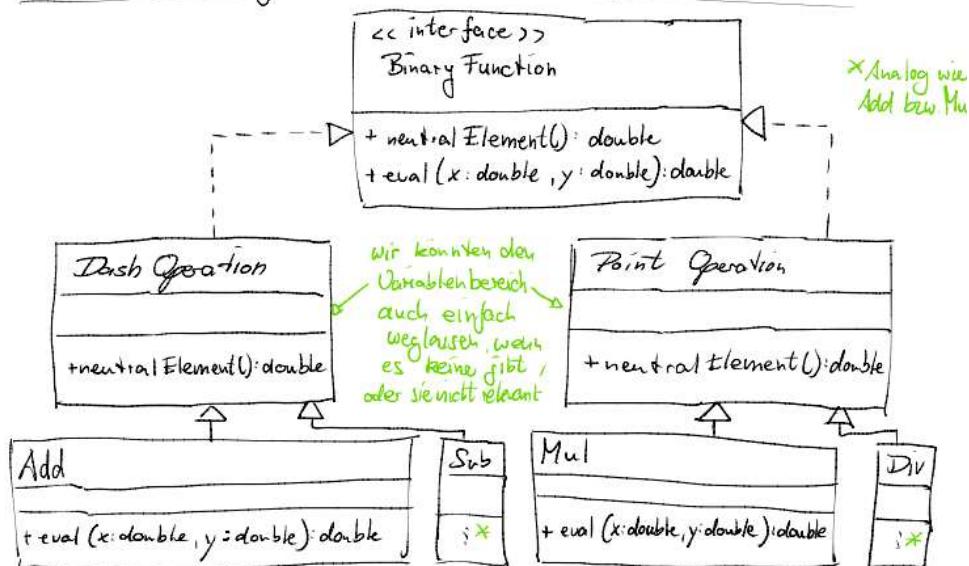
public class Kuchen implements Foo {
    @Override
    public int bar() {
        return 42;
    }
}

Foo foo = new Kuchen();
System.out.println(foo.bar());

```

Beispiel UML Diagramm:

UML - Klassendiagramm für die wesentlichen Fancy Lib-Komponenten



Übersicht extends vs implements

Klasse erbt von Klasse

- Extends, nur eine

Klasse erbt von Interface (implementiert)

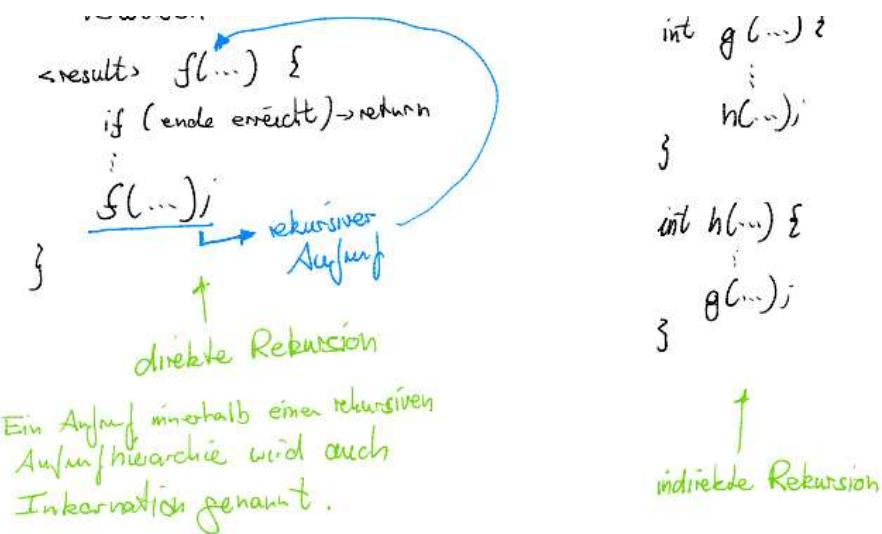
- Implements, mehrere erlaubt

Interface erbt von Interface

- Extends, mehrere erlaubt

Rekursionen

Funktionen (Methoden) die auf sich selbst verweisen



Rekursive Definition:

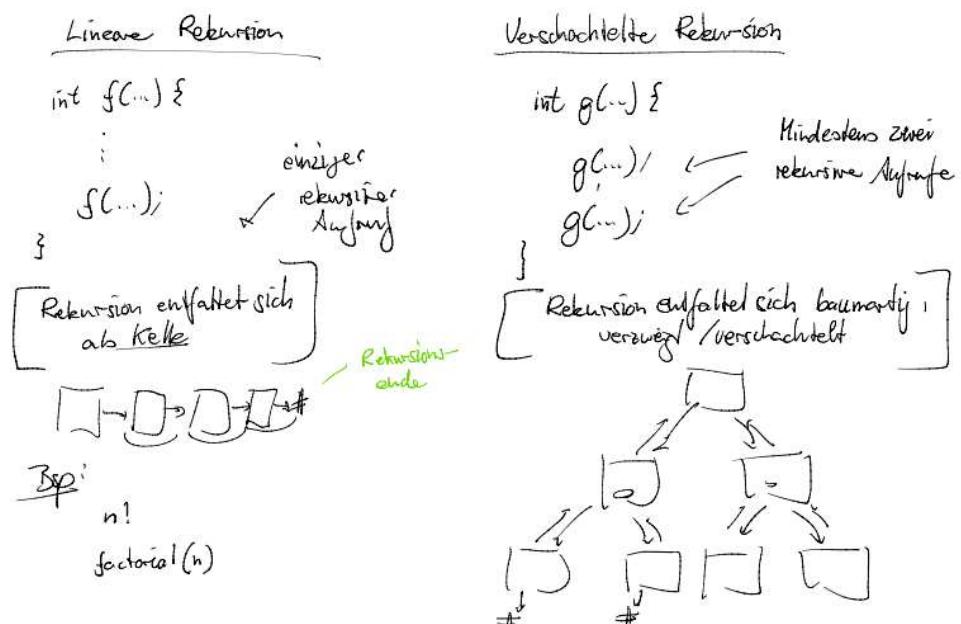
$$n! = \begin{cases} 1 & \text{falls } n=0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

Java

```
public int factorial(int n) {
    if (n <= 0) {
        return 1;
    }
    return n * factorial(n-1);
}
```

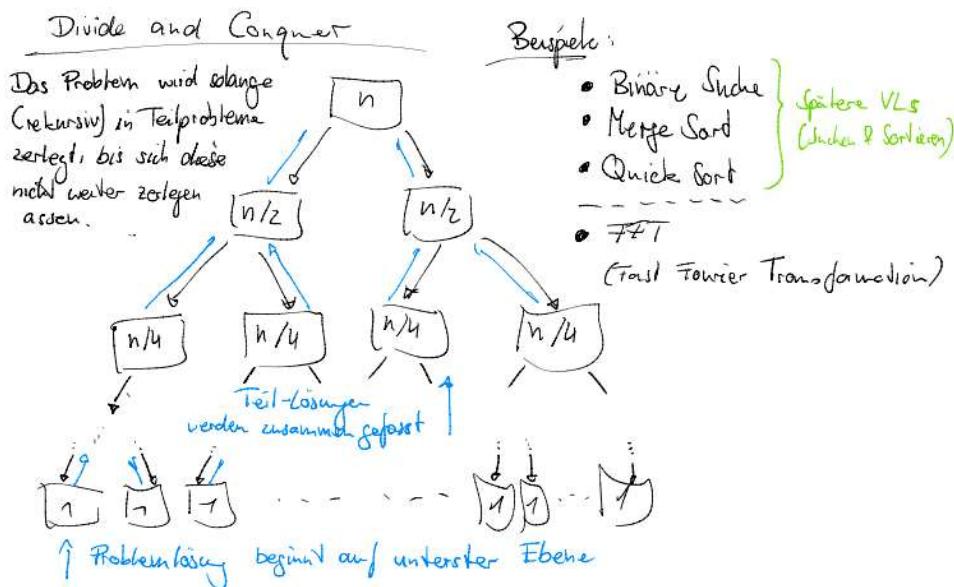
rekursiver Aufruf

Arten von auftauchender Rekursion:



Divide and Conquer

- Das Problem wird solange (rekursiv) in Teilprobleme zerlegt, bis sich diese nicht weiter zerlegen lassen.



Backtracking

- Typischerweise (rekusives) Verfahren zur Problemlösung (Lösungssuche)

Lösung wird schrittweise aufgebaut

Grundsätzliche Struktur von Backtracking:

```

find Solution (<Teillösung>){           V
    if (<Teillösung> ist eine zulässige Lösung){      |
        Lösung ausgeben;
        ggf. Suche beenden;
    }
    für alle möglichen nächsten Erweiterungen der Teillösung{      |
        find Solution (<erweiterte Teillösung>);      |
    }
}
  
```

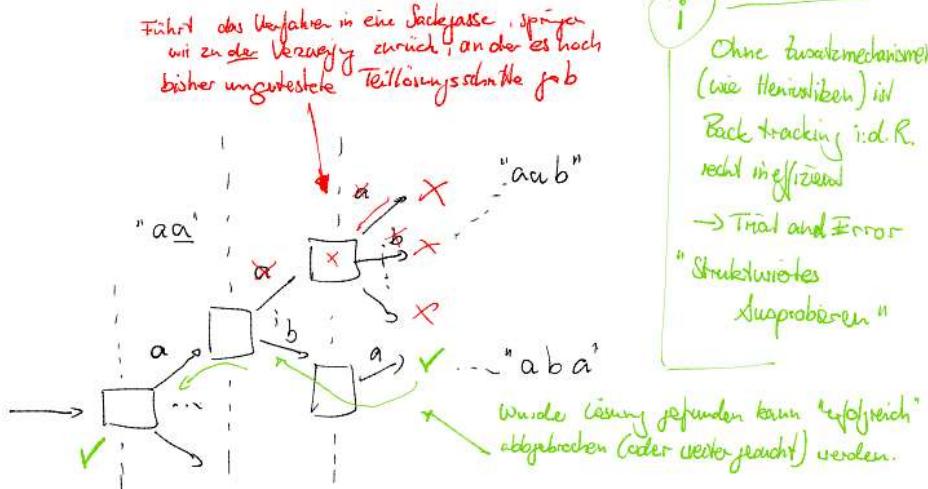
Veranschaulichung Backtracking:

- Führt das Verfahren in eine Sackgasse, springen wir zu der Verzweigung zurück, an der es noch bisher ungetestete Teillösungsschritte gab.

+ Wurde Lösung gefunden kann "erfolgreich" abgebrochen (oder weiter gesucht) werden.

(Ohne Zusatzmechanismen ist Backtracking i.d.R recht ineffizient -> Trial and Error
"Strukturiertes Ausprobieren"

Backtracking ("Zurückverfolgen")



Dynamische Arrays

Für dynm. Arrays benötigen wir:

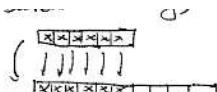
- Klasse, die Array kapselt
- Initiales Array (ggf. Zähler)
- Wenn "add" oder andere schreibende Zugriffsmethoden zur Überschreitung der Größe führt,

--> interner Array vergrößern ohne Daten zu verlieren

Idealerweise ist das alles noch außen hin nicht sichtbar, sonder geschieht automatisch und vor uns "versteckt"

Dynamisches Array weiter Anmerkung:

- Zugriff an beliebige Position sehr schnell (konstante Laufzeit) $\text{get}(i) \Leftrightarrow [i]$
- Daten liegen zusammenhängend im Speicher
- Das Einfügen eines neuen Elements kann eine Arrayvergrößerung auslösen -> Umzug auf ein neues größeres Array (kopieren der alten Daten notwendig)



- Dyn. Arrays sind oft **nicht sehr Speichereffizient**, da (je nach Anwendung) viel reservierter Speicher ungenutzt bleibt (also entweder machst ja Größe vom array vergrößern und machst das einmal und dann ist halt viel speicher unbenutzt oder du

machst es mehrere male immer wenn max größer überschritten wird (z.b immer um 1 nur vergrößern) aber dann hat man halt viele zusätzliche Prozesse die alles Verlangsamen)

- Remove "mittendrin" erfordert das Versetzen aller nachfolgenden Elemente

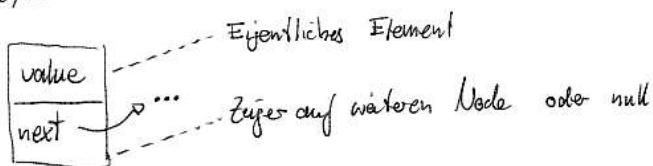


- Rüber Kopieren Allgemein ins neue Array dauert auch länger je größer das Array

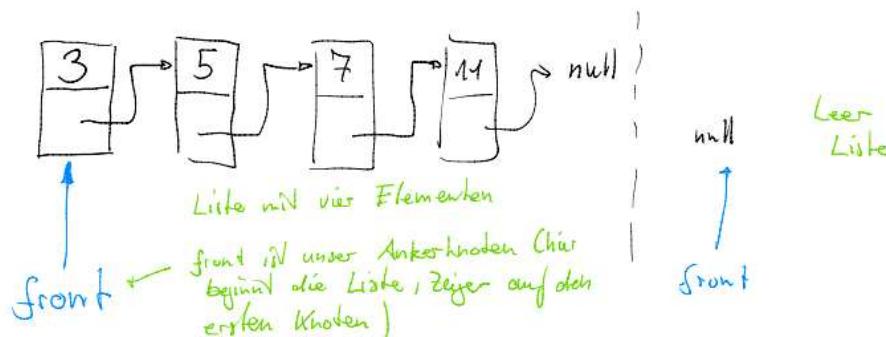
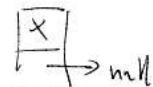
(linear) einfache verkettete Listen

(Linear) einfache verkettete Listen

Node / List Node



Null erfüllt uns die Semantik "es gibt keinen Nachfolge knoten"



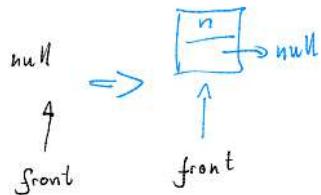
- Listen können kreuz und quer im Speicher liegen

Arbeit mit Listen

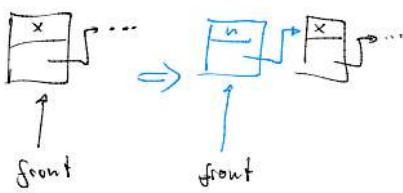
Arbeiten mit Listen

Beispiel: addFront(n) (Element am Anfang einfügen)

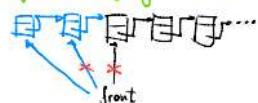
Fall 1) Liste leer



Fall 2) Liste nicht leer

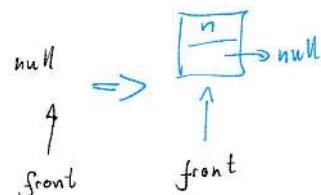


Mehrfacher Aufruf von addFront

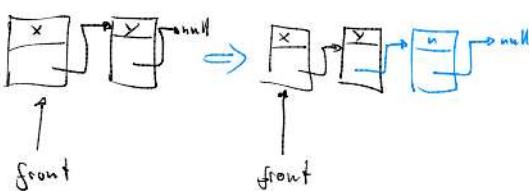


Beispiel: addBack(n) (Element am Ende einfügen)

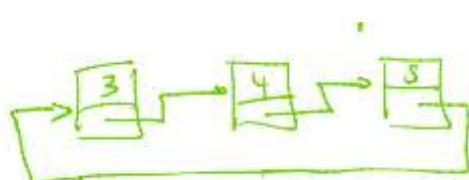
Fall 1) Liste leer



Fall 2) Liste nicht leer



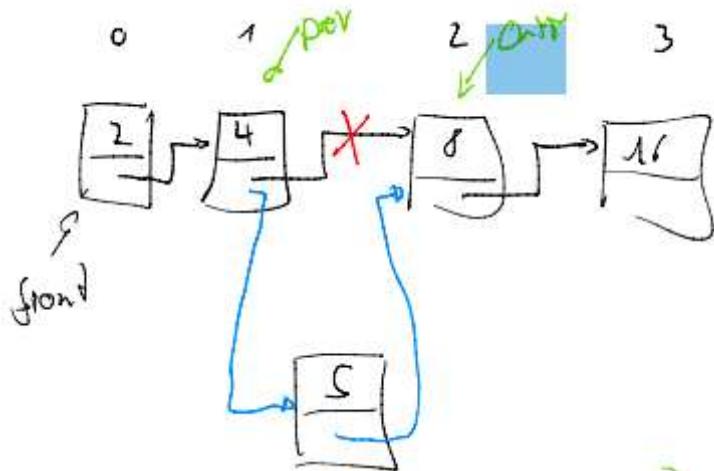
- Man kann Listen auch zu Ringstrukturen (Ring-Listen machen indem man den letzten Knoten wieder auf ersten Knoten zeigt:



Insert-At:

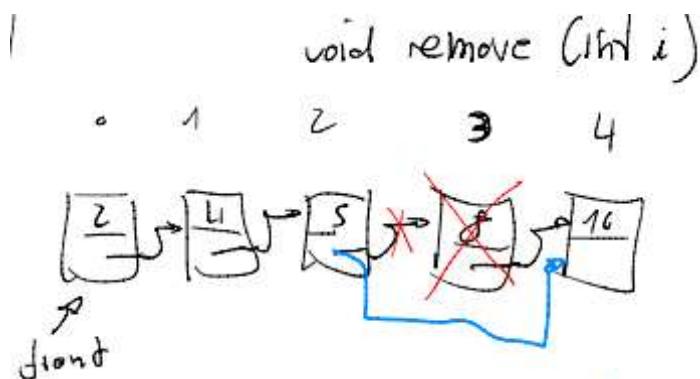
- Position finden (prev merken)
- Knoten erzeugen
- Next-Zeiger von prev auf neuen Knoten setzen
- Next-Zeiger von neuem Knoten auf curr setzen

- Fals Einfügen am Anfang -> front setzen



Remove:

- Position finden (prev merken)
- Next-Zeiger von prev auf nächsten Knoten curr.next setzen
- Falls löschen am Anfang -> front setzen



Doppel verkettete Listen

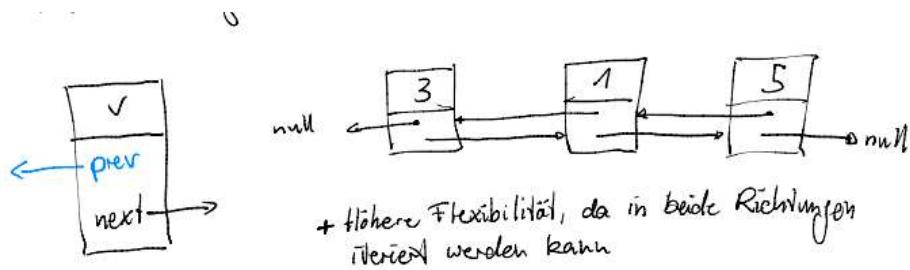
Erweiterung des Knotentyps um weitere Zeiger

Vorteile:

- Höhere Flexibilität, da in beide Richtungen iteriert werden kann
- Einige Operationen leichter, da immer auch der Vorgänger bekannt ist

Nachteil:

- Mehr Arbeit bei der Knotenverwaltung

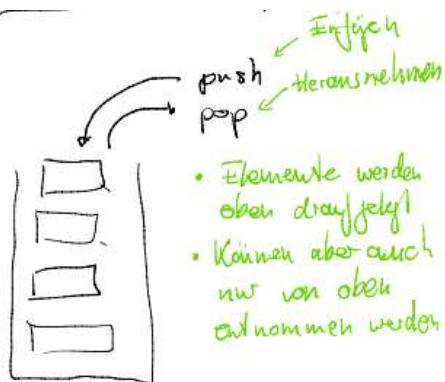


Stack (Stapelspeicher)

Einfügen: "push" Rausnehmen: "pop"

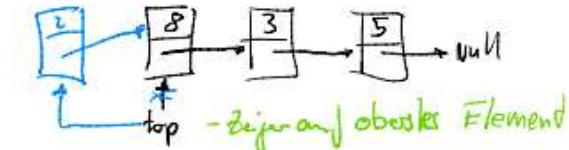
-Elemente werden oben draufgelegt können aber auch nur von oben entnommen werden

Konzept: LIFO Last-in First-out (Das letzte was rein ging geht, als auch erstes wieder raus)

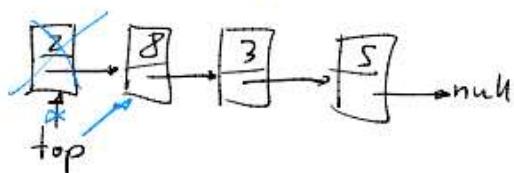


Stack als Liste

Push ()



pop ()

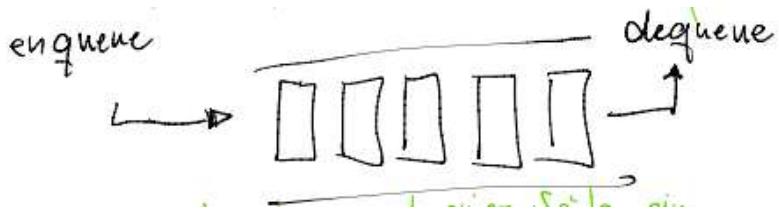


Queue (Schlange/Warteschlange)

Einfügen: "enqueue" Rausnehmen: "dequeue"

-Elemente gehen auf einer Seite rein und auf der anderen raus

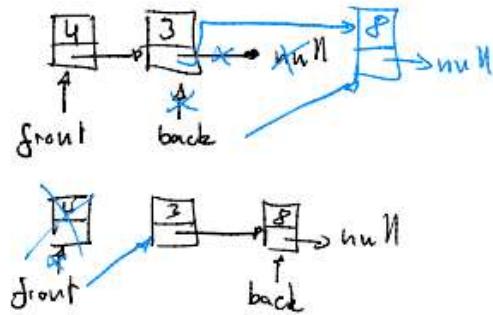
Konzept: FIFO First-in First-out (Was als erstes rein geht, geht als erstes wieder raus)



Queue als Liste (mit zusätzl. Zeiger back)

enqueue(8)

dequeue()



Array vs Listen

Arrays (nicht-dynamisch)

- + Einfügen am "Ende" einfach (falls genug Platz)
- + Wahlfreien Zugriff [index] -> sehr schnell
- + Implizierte Topologie (keine zusätzliche Verwaltungsstrukturen)
- Beliebiges Einfügen/Löschen schwierig
- Begrenzte Größe (gilt nicht für dyn. Arrays)

Listen

- + Einfügen/ Löschen an beliebiger Stelle einfach
- + Beliebig erweiterbar
- Für wahlfreien Zugriff (auf i-tes Element) muss Liste sequentiell durchlaufen werden
- Explizite Topologie (next-Zeiger...)

Bäume

- Ein Baum besteht aus Knoten, die über Kanten verbunden sind
- Ein Knoten kann mehrere Nachfolgeknoten haben
- Ein Knoten hat maximal einen Vorgängerknoten

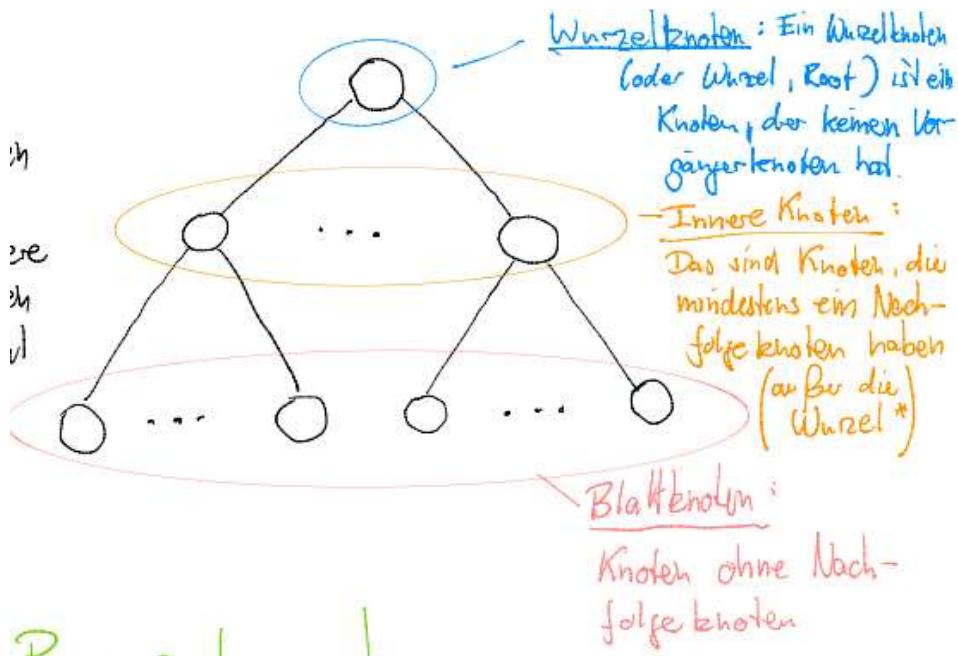
- Es sind keine Zyklen erlaubt, d.h. es muss eine Wurzel geben

Wurzelknoten: Knoten der kein Vorgänger hat

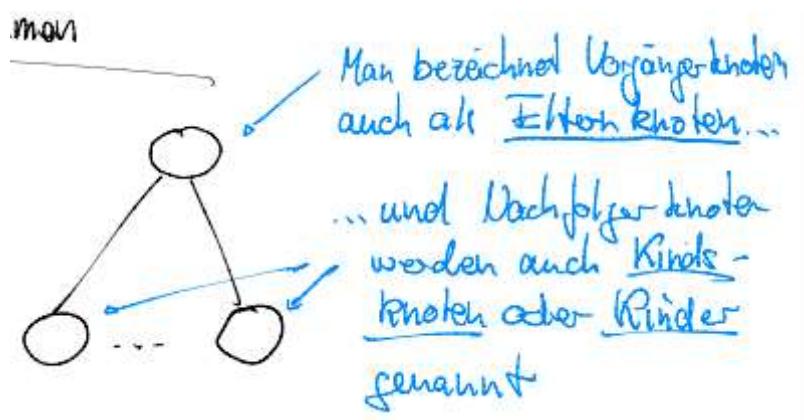
Innerknoten: Knoten mit mind. ein Nachfolgeknoten (außer Wurzel)

Blattknoten: Knoten ohne Nachfolgeknoten

(Baum kann man auch als Liste verstehen: der next nicht ein Zeiger, sondern eine Liste von Zeigern)



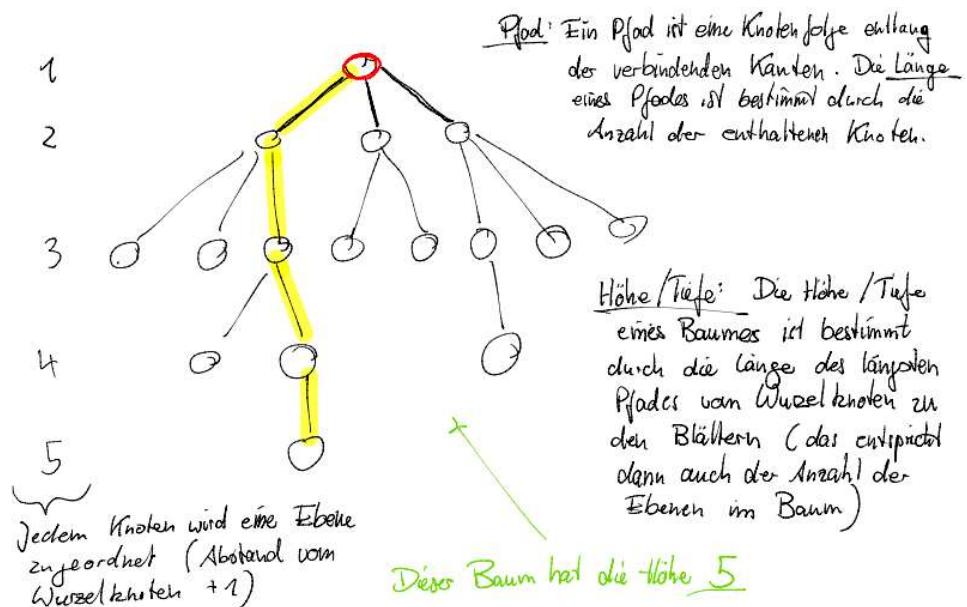
Vorgängerknoten = Elternknoten / Nachfolgen = Kinderknoten



Jeder innere Knoten bestimmt einen eigenen Unterbaum, in dem er der Wurzelknoten ist (rekursive Struktur)

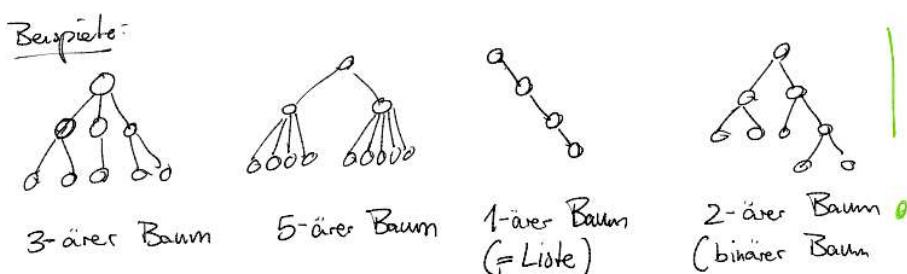
Pfad: eine Knotenfolge entlang der verbindenden Kanten (Länge eines Pfades = Anzahl der enthaltenen Knoten)

Höhe/Tiefe = Länge des längsten Pfades (von Wurzel bis zum Blatt) = Anzahl der Ebenen des Baums



Verzweigungsgrad eines Knotens (Ausgangsgrad) = Anzahl der Kinder des Knotens

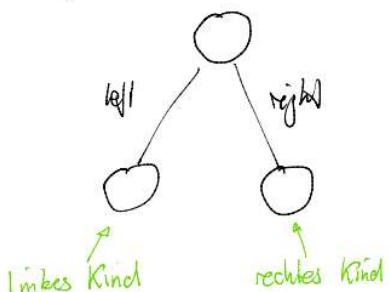
Wir sprechen von einem n-ären Baum, wenn n der maximale Verzweigungsgrad aller Knoten des Baumes ist.



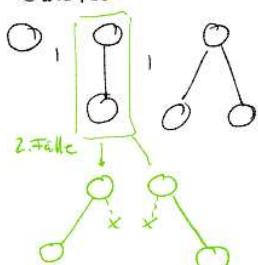
Binäräume

Binäräume

- Binärbaum ist ein 2-äser Baum
- Jeder Knoten hat maximal zwei Kinder!



Substrukturen, die uns erwarten



Traversierung von Binäräumen

-um Elemente vom Baum zu verarbeiten müssen wir baum irgendwie durchlaufen (d.h. Knoten besuchen)

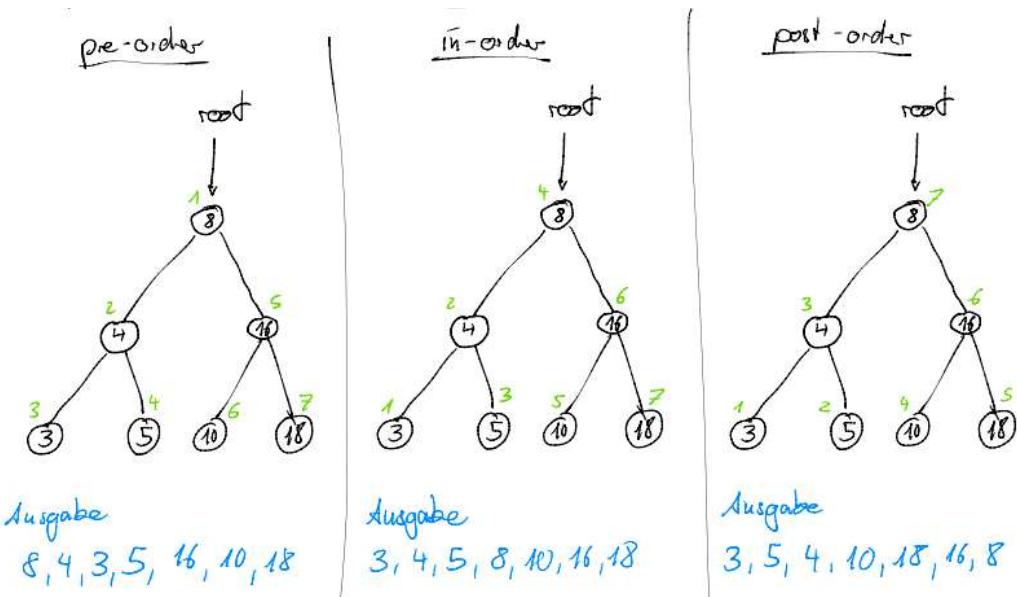
Zwei Typen von Traversierung

- Tiefentraversierung (depth-first)
- Breitentraversierung (breadth-first)

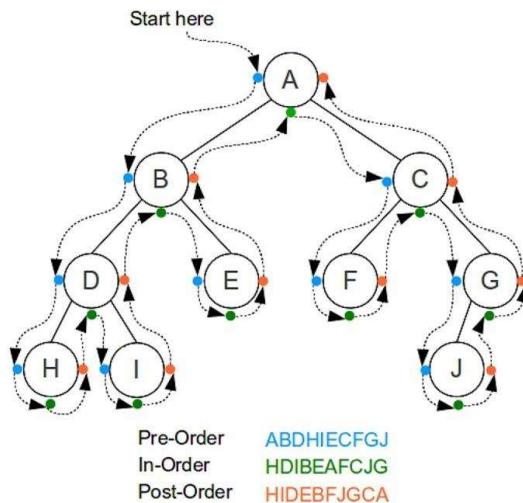
Tiefentraversierung

3 Arten:

<u>pre - order</u> (Besuch vor Abstieg)	<u>in - order</u> (Besuch zwischen Abstieg im linken und rechten Teilbaum)	<u>post - order</u> (Besuch nach dem Abstieg in beide Teilbäume)
<pre>traverse (node) { if (node == null) return; visit (node); traverse (node.left); traverse (node.right);</pre>	<pre>traverse (node) { if (node == null) return; traverse (node.left); visit (node); traverse (node.right);</pre>	<pre>traverse (node) { if (node == null) return; traverse (node.left); traverse (node.right); visit (node);</pre>



Simplest Trick to find PreOrder InOrder PostOrder



Breitentraversierung

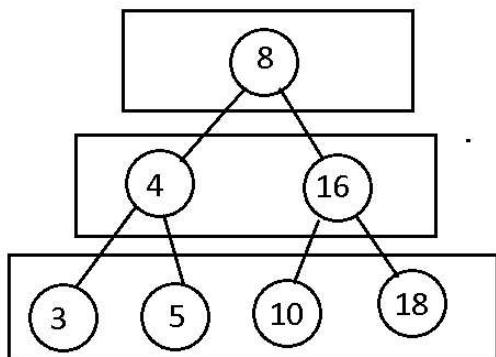
Vorgehensweise:

Oberseter Knoten wird gequeuet

--> wird rausgenommen und abgegeben

--> Dann werden die Kinder in die queue gemacht

--> soweiter bis keine weiteren Knoten



1. \rightarrow

2. \rightarrow

visit 8

3. \rightarrow
 16 4

4. \rightarrow
 16 ↘
 4

5. \rightarrow
 5 3 16 ↘
 16

6. \rightarrow
 18 10 5 3

Ausgabe: 8 4 16 3 5 10 18

Sortierte Binärbäume

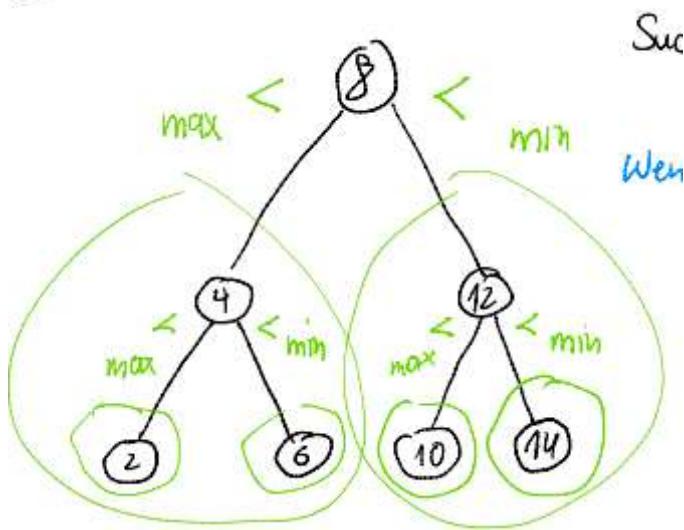
Ein Binärbaum T ist sortiert/geordnet wenn:

- T ein Blatt (oder leer)

Oder wenn T-left bzw T-right existiert:

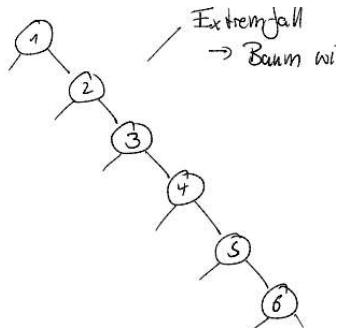
- Das größte Element in T-left ist kleiner als das Element von T
- Das kleinste Element in T-right ist größer als das Element von T
- T-left und T-right sind sortiert

Wenn man weiß ein Baum ist sortiert dann braucht man weniger Schritte/Knoten besuchen weil man instant kucken ob das zu suchende Element größer oder kleiner ist als T und dann direkt in den entsprechenden rechten oder linken Teilbaum rein und wieder.



Auch sortierte Binäräbäume können "entarten"

Bsp.: Baum wird zu Liste (Sortiertheit bringt bei Suche kein Vorteil mehr)



Balanciertheit

Höhe von T = $h(T)$

$$-h(T) = 1 + \max(h(T\text{-left}), h(T\text{-right}))$$

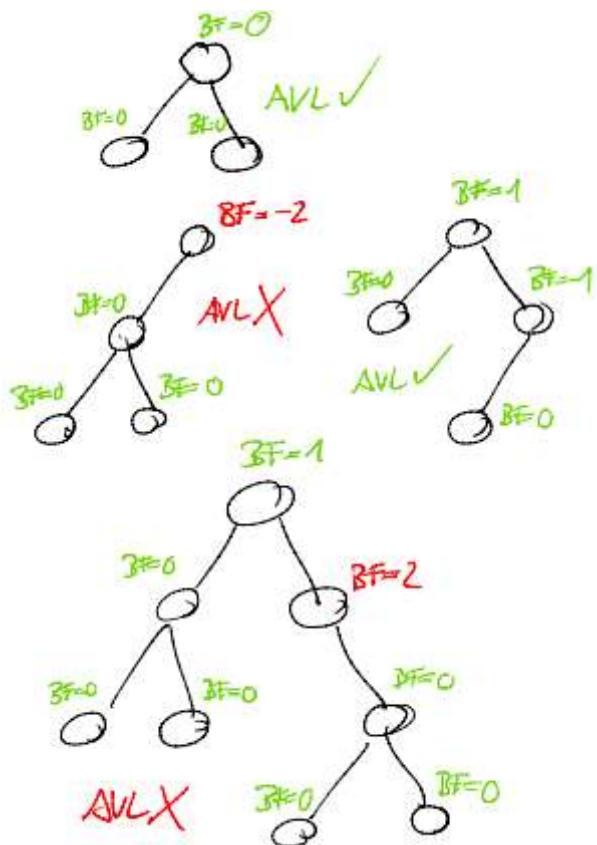
BalanceFaktor:

$$BF(T) = h(T\text{-left}) - h(T\text{-right})$$

AVL-Baum ist es dann, wenn:

Für alle Knoten n aus T gilt: $-1 \leq BF(n) \leq 1$ (AVL-Eigenschaft)

Hier unten beim Letzen Baum vorsicht also immer alle Teilbäume checken

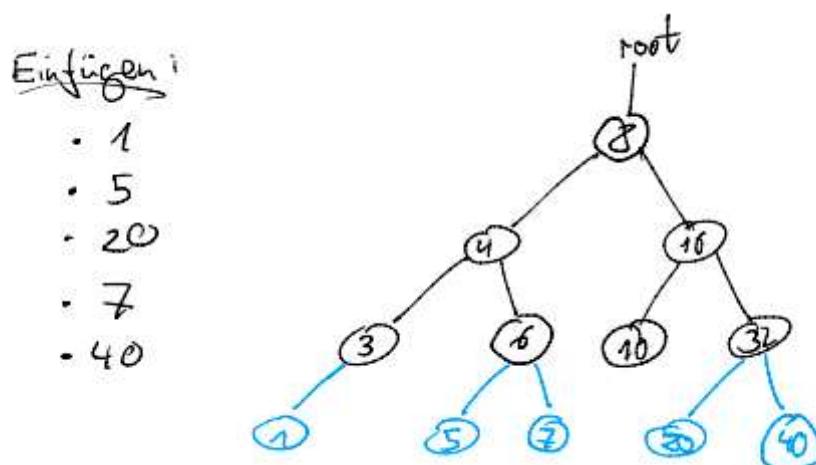


Sortiert Einfügen

Vorgehen:

- Ist x kleiner als aktueller Knotenwert, such in linken Teilbaum weiter
- Ist x größer als aktueller Knotenwert, suche in rechten Teilbaum weiter
- Ist linker bzw. Rechter Teilbaum leer, Element als neuen Knoten einfügen
- (ist $x ==$ Knotenwert z.B ignorieren)

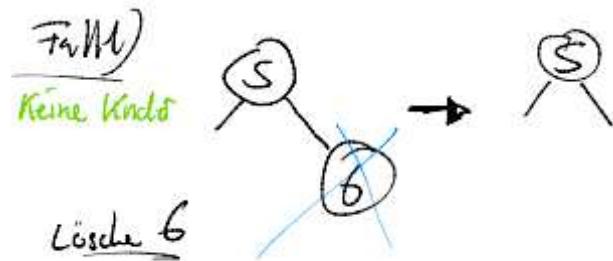
(Auch durch sortiertes Einfügen kann Baum "entarten")



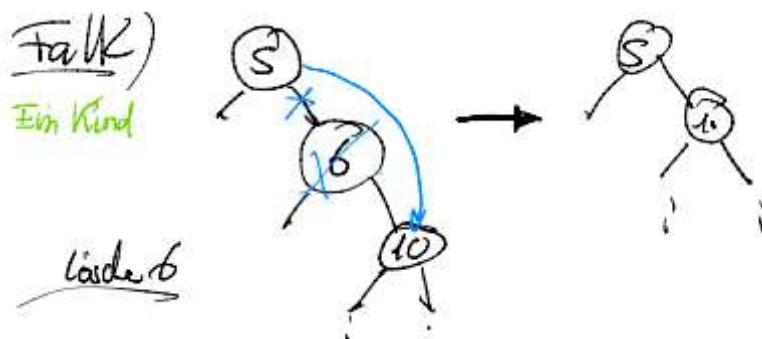
Sortiertes Löschen

Mehrere Fälle:

- 1) Keine Kinder

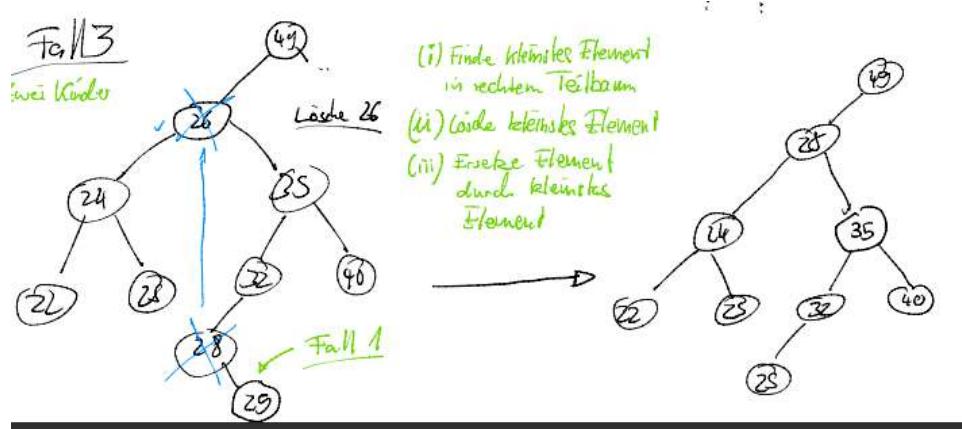


- 2) Ein Kind



- 3) Zwei Kinder

- Finde kleinstes Element in rechtem Teilbaum
- Lösche kleinstes Element
- Ersetze Element durch kleinstes Element



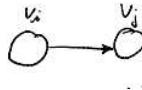
Graphen

Ein Graph G ist ein Tupel $G = (V, E)$ oder $G(V, E)$

- V (Vertices) ist die Menge der Knoten.
- E ist die Menge der Kanten.

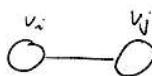
Bei den Kanten unterscheidet man zwischen Graph mit gerichteten Kanten (gerichteter Graph)

Kante ist ein Typ
(gerichtet)
 $E \subseteq V \times V$ oder $E \subseteq \{(v_i, v_j) \mid v_i, v_j \in V\}$
d.h. ein $e = (v_i, v_j) \in E$ bezeichnet eine Kante
von v_i nach v_j mit $(v_i, v_j) \neq (v_j, v_i)$ fällt v_j



Und einem ungerichteten Graph

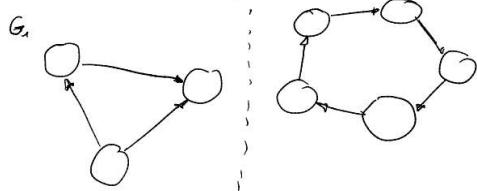
Kante ist eine
Menge (ungerichtet)
 $E \subseteq \{\{v_i, v_j\} \mid v_i, v_j \in V\}$
d.h. ein $e = \{v_i, v_j\} \in E$ bezeichnet eine Kante
zwischen v_i und v_j mit $\{v_i, v_j\} = \{v_j, v_i\}$



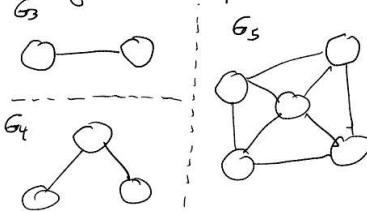
Wir betrachten hier nur Graphen ohne "Schlingen" d.h.
für alle Kanten gilt $v_i \neq v_j$ (irreflexiv)

Beispiele

Gerichtete Graphen G_2

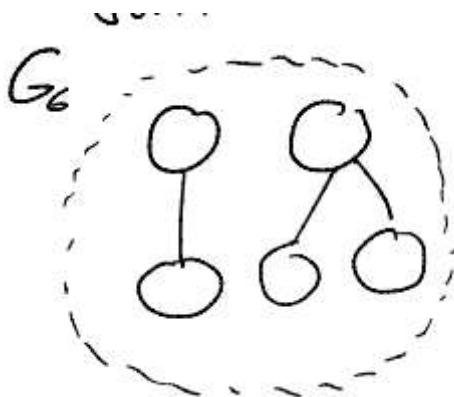


ungerichtete Graphen



Weitere Anmerkungen:

Graphen können auch nicht zusammenhängend sein



Mann kann Graphen als Verallgemeinerung von Bäumen verstehen:

- Mehrere Vorgänger von Bäumen erlaubt
- Ggf. Zyklen erlaubt

Graphen können wie Listen/Bäume durch dynamische Knotenstrukturen modelliert werden z.B.

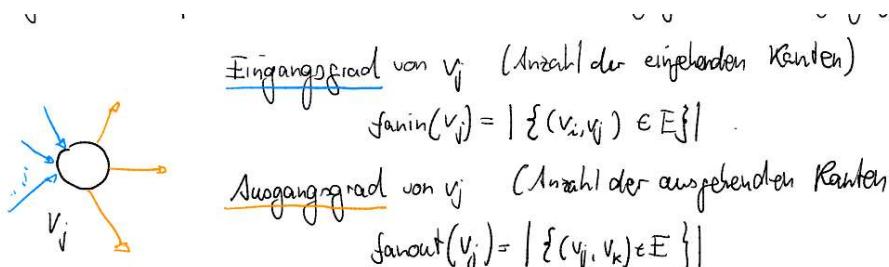
```
public class Node {
    public int value;
    public Edge[] neighbors;
}

public class Edge {
    public Node first;
    public Node second;
}
```

Gerichtete Graphen

Unterscheidung zwischen Eingangs- und Ausgangsgrad

(Anzahl der Eingehenden/Ausgehenden Kanten)

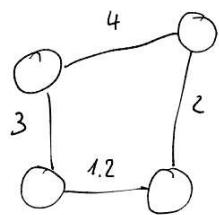


Gewichteter Graph

Ein gewichteter Graph ist ein Tupel $G = (V, E, W)$ mit

- (V, E) ist ein (gerichteter/ ungerichteter) Graph
 - $W: E \rightarrow M$ z.B. $M = \text{Reelle Zahlen } \mathbb{R}$
- > Eine Abbildung die **jedem Knoten ein Gewicht** zuordnen kann

Bsp.



(So können wir z.b. Distanzen (kantenlängen) zwischen Knoten ausdrücken (Auch: Neuronale Netze))

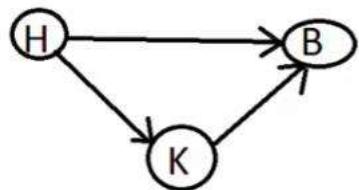
Alternative Darstellung: Adjazenz-Matrizen

Sei $G = (V, E)$ ein Graph dann ist die Adjazenz Matrix A eine **$n \times n$ -Matrix**.

$n = \text{Anzahl der Knoten}$. Matrix enthält nur 0 und 1.

Ist **G ungerichtet**, dann ist A immer symmetrisch d.h. $A = A^T$

Ist **G gerichtet**, dann kann A auch unsymmetrisch sein.



$$G = (V, E)$$

$$V = \{H, K, B\}$$

$$E = \{(H, K), (K, B), (H, B)\}$$

Menge der Knoten

Menge der Kanten also von einem Knoten zum anderen

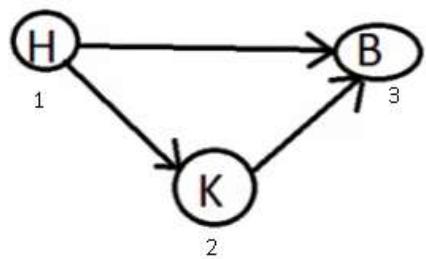
$$G = (\{H, K, B\}, \{(H, K), (H, B), (K, B)\})$$

in Matrix:

3x3 Matrix weil 3 Knoten.

Knoten bezeichnen wenn es wie hier Buchstaben sind.

Wenn ein Knoten existiert dann 1 wenn nicht dann 0.



Adjazenz Matrix

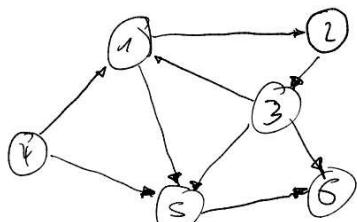
	1	2	3
1	0	1	1
2	0	0	1
3	0	0	0

Bsp. Erste Zeile: Es gibt eine Kante von 1 nach 3 und von 1 nach 2 aber nicht von 1 auf 1

Wären jz oben da beim Graph keine Pfeile also ungerichtet. Dann würde Matrix so aussehen:

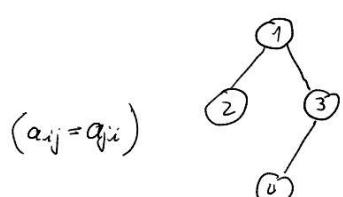
0 1 1
1 0 1
1 1 0

Bsp. gerichteter Graph:



$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 & 1 & 1 \\ 4 & 1 & 0 & 0 & 0 & 1 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 1 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

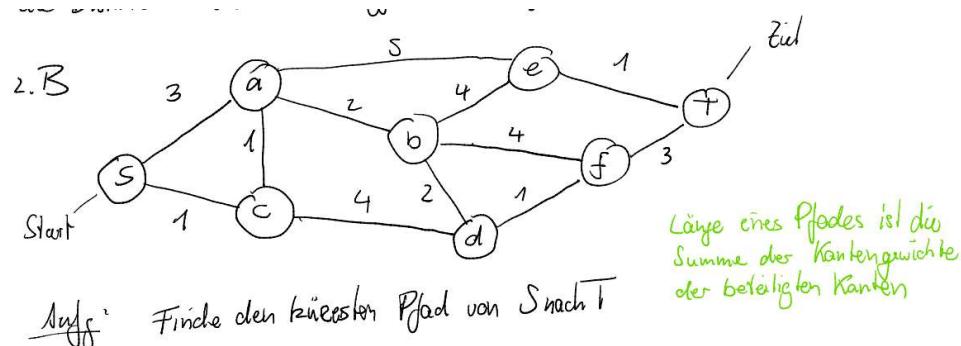
Bsp. ungerichteter Graph



$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 1 & 1 & 0 \\ 2 & 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 & 1 \\ 4 & 0 & 0 & 1 & 0 \end{bmatrix} \quad A = A^T$$

Algorithmen auf Graphen

Gegen sei ein ungerichteter, gewichteter Graph mit positiven Kantengewicht (ein Gewicht = Distanz zwischen Knoten)

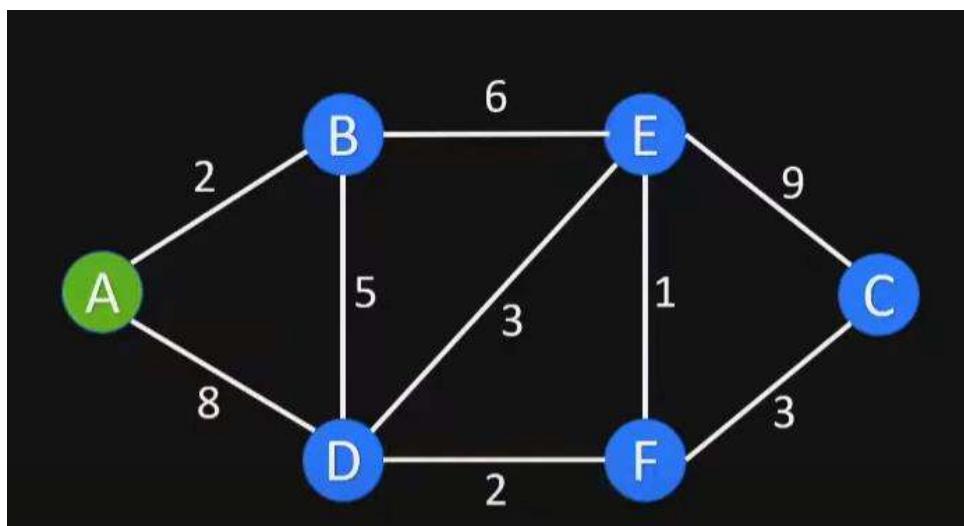


Lösung dazu: **Dijkstra- Algorithmus**

Dijkstra- Algorithmus

Kürzester Weg von einem Knoten zu allen anderen Knoten

Bsp.:



Schritte:

1. Wir müssen uns merken welche Knoten noch nicht besucht worden sind

Also zwei Listen: Visited Nodes: [] Unvisited Nodes [A, B, C, D, E, F]

2. Allen Knoten eine gewisse Distanz zuweisen die sich im Verlauf ändern wird

A auf 0 setzen

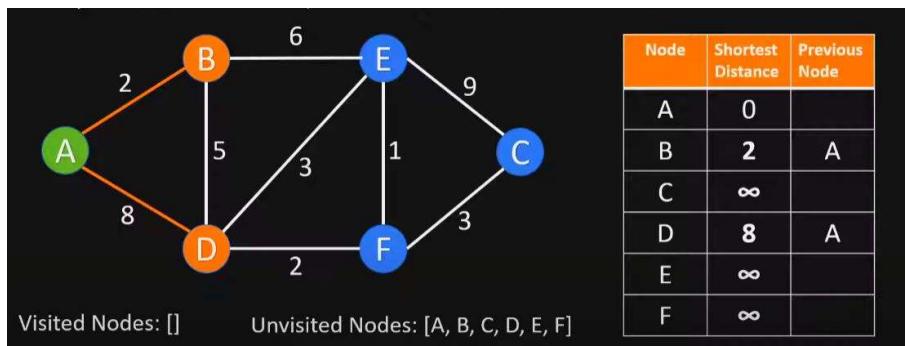
Alle Anderen Knoten auf unendlich Setzen.

Node	Shortest Distance	Previous Node
A	0	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	

(Distanz von A zu A ist 0, was anderes wissen wir am Anfang nicht)

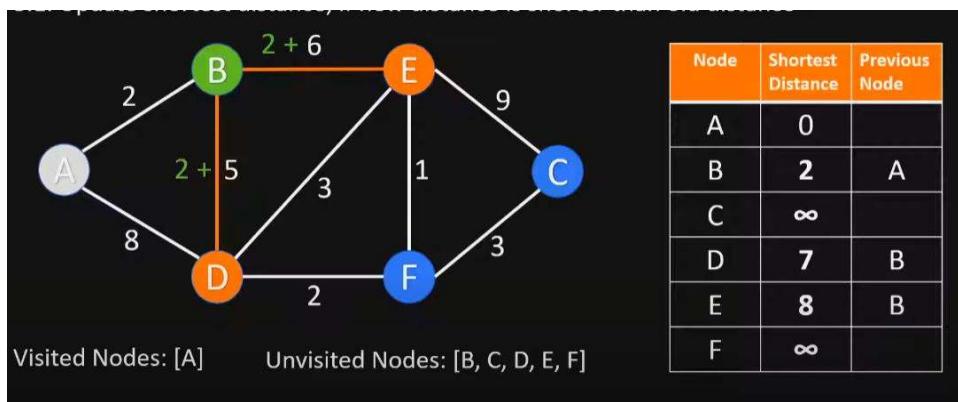
Zweite Spalten: Den letzten Knoten der uns zu dieser Distanz führte (Am Anfang leer)

3. Wir besuchen nun B und C. Distanz zwischen A und B ist 2/ A und C ist 8

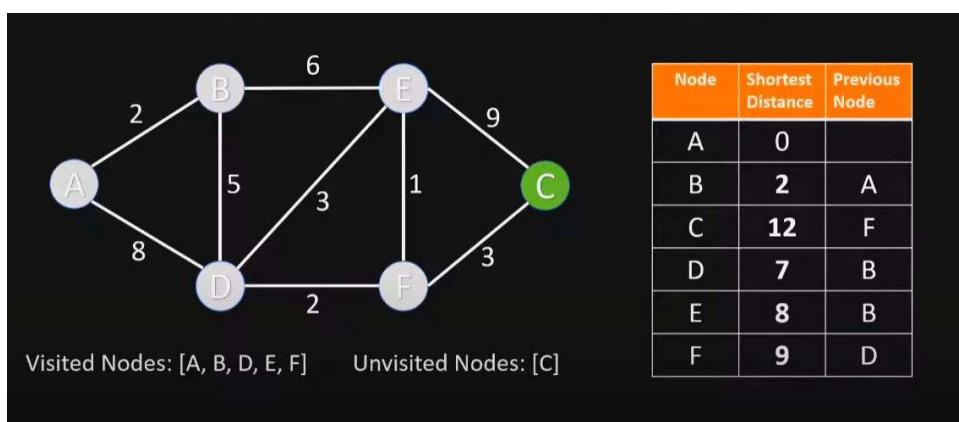
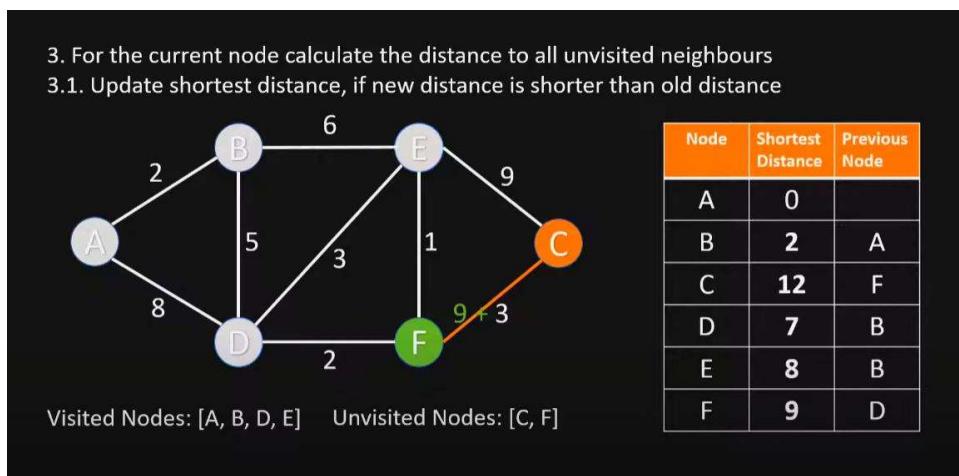
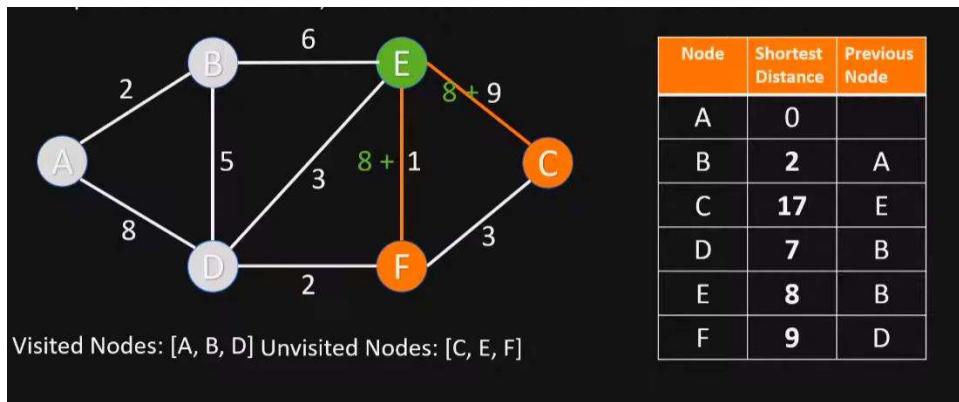
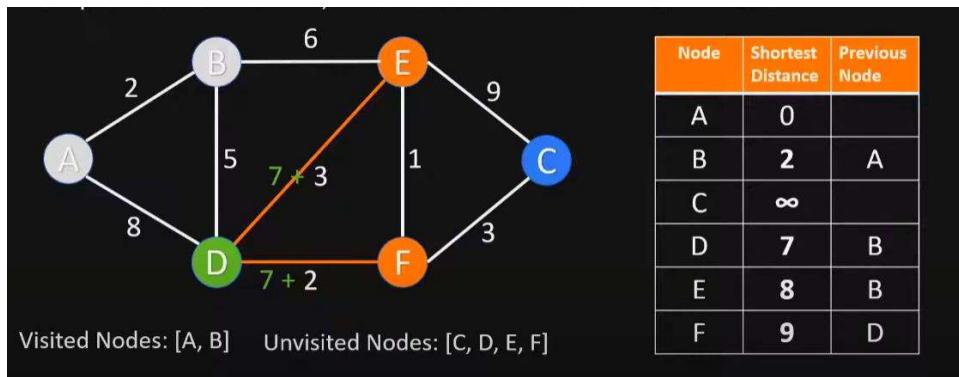


4. A wurde nun besucht und kann in die Liste der besuchten. Jetzt nehmen wir den nächsten Knoten der bearbeitet werden soll: Hier nehmen wir den Knoten aus der unvisited Liste mit der zurzeit geringsten Distanz. A hat geringste Distanz wurde aber schon besucht. Dh. Wir nehmen B mit Distanz 2. Wir führen Schritt 3) nochmal aus.

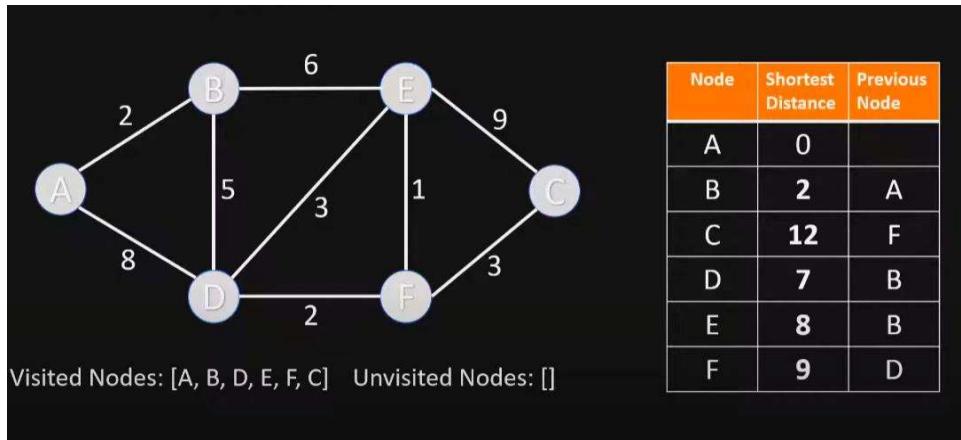
Die benachbarten sind D mit Distanz 5 und E mit Distanz 6. Weil B ja schon Distanz 2 hat müssen wir $2+6 / 2+5$. Wenn jz die neue Distanz für D kleiner ist als das was in Tabelle steht dann überschreiben wir die Distanz. Also für D ist ja jetzt Distanz 7 und das ist kleiner als 8 -> also unserer neuer Distanzwert für D. Wenn die neue Distanz größer als die in Tabelle, dann nicht überschreiben.



5. Und so weiter:

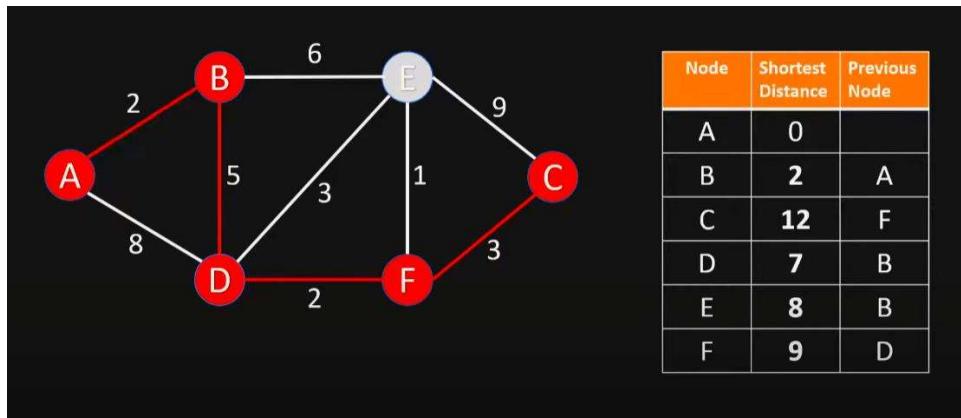


Ch hat keine unvisited Nachbaren mehr deswegen sind wir fertig!



Wenn man jz den kürzesten Pfad von A nach C will einfach bei Tabelle kucken:

Prev Node von C ist F -> Prev von F ist D -> von D ist B -> von B ist A



Java Generics

Den Typ bei Methoden/Klassen beliebig lassen.

(Generische Programmierung hat das Ziel einen möglichst hohen Grad an Wiederverwendbarkeit von Softwarekomponenten (meist Bibliotheken) zu erreichen:

- Algorithmen sollen möglichst allgemein formuliert sein,
- sodass diese für möglichst viele Datentypen verwendet werden können)

```
Integer[] intArray = {1, 2, 3, 4, 5};
Double[] doubleArray = {5.5, 4.4, 3.3, 2.2, 1.1};
Character[] charArray = {'H', 'E', 'L', 'L', 'O'};
String[] stringArray = {"B", "Y", "E"};
```

Anstatt für jeden ArrayTyp eine eigene Methode zu machen (weil du kannst ja nicht mit ner Methode die intarray übergeben bekommt auch auf z.b double array iterieren), machen wir halt nur eine und da wo der Typ ist kommt so etwas wie eine variable "T" meistens kann auch anders sein: S, V, W...

```
public static <Thing> void displayArray(Thing[] array) {

    for(Thing x : array) {
        System.out.print(x+" ");
    }
    System.out.println();
}
```

Oder wenn nicht void dann T selber halt als Typ der rausgegeben wird:

```
public static <Thing> Thing getFirst(Thing[] array) {
    return array[0];
}
```

Oder wenn man (hier beispiel ein Game wo man die einzelnen Sachen da drawen will) irgend ein Typ hat.

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

    Player player = new Player();
    Enemy enemy = new Enemy();
    Item item = new Item();
    Tree tree = new Tree();

    draw(player);
    draw(enemy);
    draw(item);
    draw(tree);
}

public static <Thing> void draw(Thing x) {
    x.draw();
}
```

Generic Klassen

Hier wo du von beliebigen Typen eine Klasse erstellen kannst anstatt jz für jedes mal für jeden Typen eigene Klasse

```
public class MyGenericClass <Thing>{  
    Thing x;  
  
    MyGenericClass(Thing x){  
        this.x = x;  
    }  
  
    public Thing getValue() {  
        return x;  
    }  
}
```

Um dann eine neue Instanz zu machen kann man einfach:

```
MyGenericClass<Integer> myInt = new MyGenericClass<>(1);  
MyGenericClass<Double> myDouble = new MyGenericClass<>(3.14);  
MyGenericClass<Character> myChar = new MyGenericClass<>('@');  
MyGenericClass<String> myString = new MyGenericClass<>("Hello");
```

Generic Array:

Für Integer bzw. Int kann man auch beliebige Typen

```
ArrayList<Integer> myFriends = new ArrayList<>();
```

Was wenn man mehrere beliebige Typen generic haben will?:

Ganz einfach mehrere auflisten:

Aber dann muss man auch:

Also kann man auch zwei verschiedene auf einmal siehe das 4. hier.

Bound Types

Wenn man begrenzen will was für T reinkommen soll:

Bsp bezogen auf oben die Klasse

Hier darf also für das erste T nur irgend eine Nummer eingeügt werden (Number kommt hier implementiert in Java /JAVADOC etc.) man könnte da halt auch was eigenes einbauen.

Komplexität

-Zeitkomplexität = Laufzeitkomplexität

--> Wie viel Zeit benötigt eine Algo. Zur Berechnung.

-Speicherkomplexität

--> Wie viel Speicherplatz wird zur Berechnung benötigt wird.

Laufzeit Messung abhängig von:

-Echter Hardware

-Betriebssystem

-Compiler (-Version)

-Allgemeine Systemauslastung

-etc.

Laufzeitanalyse eines Programms

-elementare Operationen : alle gleichwertig (= 1 Schritt = 1 abstrakte Zeiteinheit)

Beispiele für elem. Op.: Zuweisung, Vergleiche, Arithmetische Operationen

-Schleifen vervielfältigen den Aufwand der Schritte innerhalb der Schleife

Nur der Dominante Teilterm zählt der rest ist eig egal für Laufzeitbestimmung

Big-O –Notation

Seien f, g Funktionen mit $f, g : N \rightarrow N$ dann gilt

- $F(n)$ element von $O(g(n))$

($O(g(n))$ ist eine Menge von Funktionen)

-Innerhalb von $O(\dots)$:

Möglichst kleine und einfache Funktionen (nur domin Term)

Keine irrelevanten Konstanten ($2n$ weglassen ok, n^2 weglassen nicht ok)

Rechenregeln

Wichtige Klassen von Funktionen:

Weitere Anmerkungen

- $\log^k n = (\log n)^k$
- Man nennt ein Algo. Effizient wenn dessen Berechnung polynomiell viel Laufzeit benötigt

P = Menge der Probleme, die mit einer deterministischen Rechenmaschiene in Polynomialzeit gelöst werden können

NP = Menge der Probleme, die mit einer nicht-deterministischen Rechenmaschiene in Polynomialzeit gelöst werden können

- Interessantes Problem (gelöst),
Gibt *Das wäre schlecht!*
 $P = NP$ oder $P \subset NP$?

Man geht von P ist Teilmenge von NP aus.

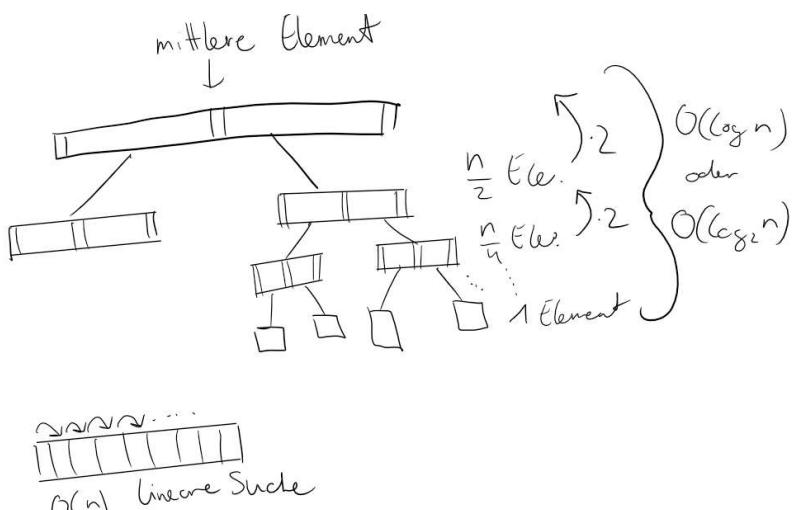
Suchen und Sortieren

Binärsuche

Gegeben ein aufsteignd sortiertes Array ($a_0, a_1, a_2, \dots, a_{n-1}$)

Rekursive Strategie:

- 1) Wir wählen mittleres Element
- 2) Entspricht mit. El. Dem key -> fertig!
- 3) Ist gesuchter Key kleiner als mitl. El. -> suche rekursiv im linken Teilarray
- 4) Ist gesuchter Key größer als mitl. El. -> suche rekursiv im rechten Teilarray
- 5) Lässt sich array nicht weiter zerlegen -> nicht gefunden



($O(\log n)$ ist viel schneller als $O(n)$) !

Sortieren

Für das Sortieren erhalten wir als Eingabe ein Array oder eine Folge von Schlüsseln (an denen ggf. Auch noch Nutzdaten hängen)

Eingabe array = [0, ..., n-1] (könnte grundsätzlich unsortiert sein) (n = Anzahl im Array El.)

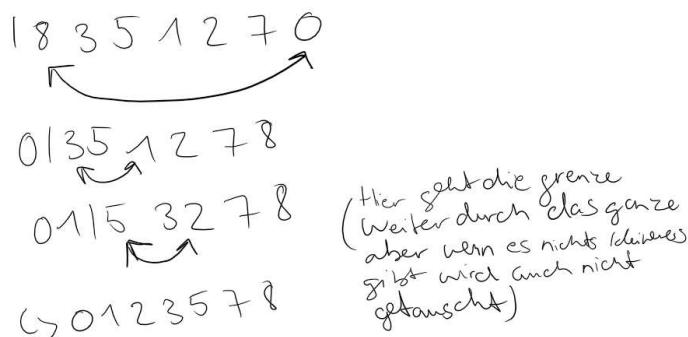
Ziel/Ausgabe: Die Werte in dem Array sollen der Gestalt verschoben oder getauscht werden, sodass für alle gilt:

$0 \leq i \leq (n-1)$ gilt (aufsteigend sortiert) (oder eben absteigend dann die < Verstauschen zu >)

Selection Sort

Strategie:

- 1) Wir starten mit $i = 0$
- 2) Wähle Position des kleinsten Elements (minindex) aus restlicher Folge $i \leq \text{minindex} < n$
- 3) Tausche kleinstes Element mit Element ganz links: $a[i]$ tausch mit $a[\text{minindex}]$
- 4) Verschiebe linke Grenze um eine Position nach rechts
- 5) Wiederhole solang es Elemente gibt



Worstcase laufzeit $O(n^2)$

Insertion Sort

Strategie:

- 1) Aktuelles Element $i = 1$
- 2) Suche richtige Position j in neuer Folge (Die neue Folge wird am Anfang der alten Folge aufgebaut)
 - Neue Folge ist stets sortiert
 - Richtige Position: $a[j-1] \leq a[i]$ und $a[j] > a[i]$ oder $j = 0$
 - Merke Element = $a[i]$, verschiebe alle Elemente zwischen j und $i-1$ um eine Stelle nach vorne/nach rechts, setze $a[j] = \text{Element}$ (an richtige Pos einfügen)
- 3) Setze $i++$

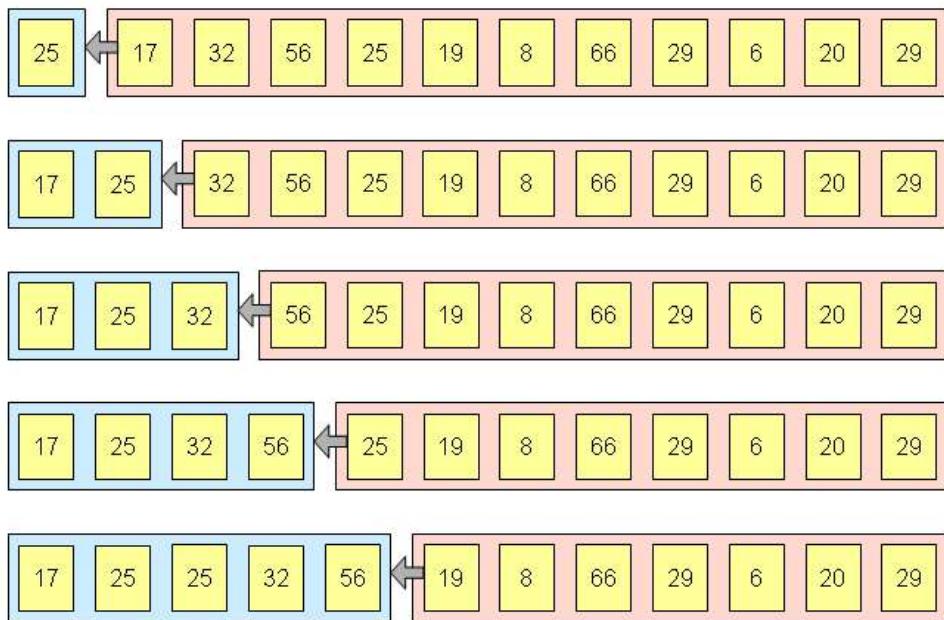
4) Wiederhole solange es Elemente gibt ($i < n$)

Worstcase/Averagecase: $O(n^2)$

Bestcase: $O(n)$ wenn array schon geordnet

Insertion Sort: Java Code

```
public static void intInsertionSort (int [] a) {  
    for (int i=1; i < a.length; i++) {  
        int temp = a[i];  
        int j;  
        for (j = i - 1; j >= 0 && temp < a[j] ; j--)  
            a[j + 1] = a[j];  
        a[j + 1] = temp;  
    }  
}
```

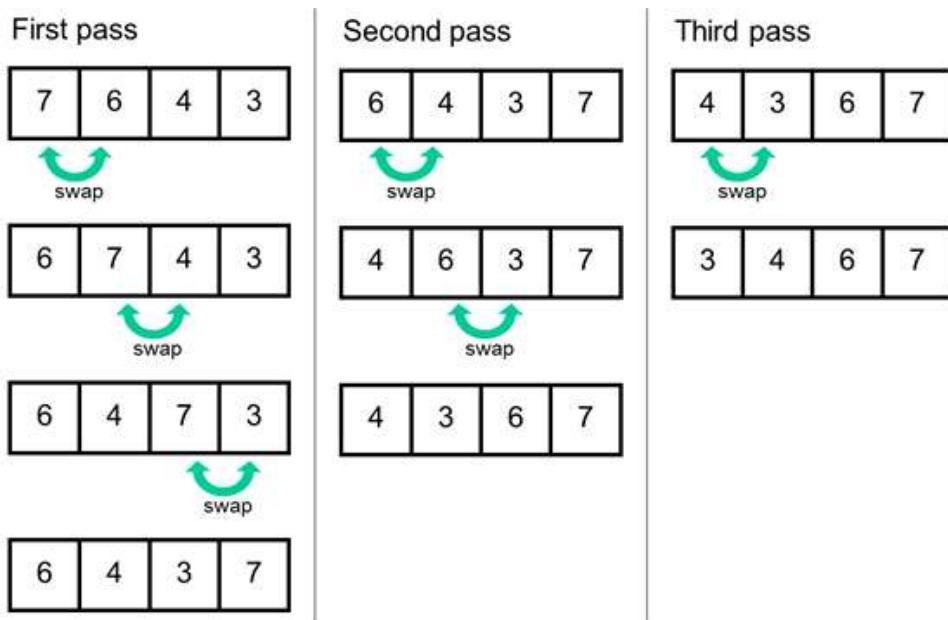


Bubblesort

i = index für Element ganz rechts

i = n - 1 (n = Anzahl der Elemente)

- 1) Wir starten ganz links im array und durchlaufen es ($0 \leq j < i$)
- 2) Dabei tauschen wir immer das aktuelle Element $a[j]$ mit dem nachfolgenden Elment $a[j+1]$ wenn $a[j] > a[j+1]$
- 3) Nach einem Durchlauf liegt das große Elment ganz rechts (i)
- 4) Wiederhole solange $i > 0$ (oder keine swaps mehr)



Worstcase: $O(n^2)$

Alle 3 SortingAlgos haben selbe laufzeit von $O(n^2)$ aber sind in der praxis verschieden schnell !

Autoboxing

```
// primitive types
//----- //-----
// boolean      Boolean
// char         Character
// int          Integer
// double       Double

// autoboxing = the automatic conversion that the Java
// unboxing = the reverse of autoboxing. Automatic con

Boolean a = true;
Character b = '@';
Integer c = 123;
Double d = 3.14;
String e = "Bro";

a.|
```

The screenshot shows a Java code editor with a tooltip for the `booleanValue()` method of the `Boolean` class. The tooltip reads: "Returns the value of this Boolean object as a primitive boolean value". Below the code, there is a list of methods for the `Boolean` class, including `booleanValue()`, `compareTo()`, `equals()`, `getClass()`, `hashCode()`, `notify()`, `notifyAll()`, `toString()`, `wait()`, and `wait(timeoutMillis)`.

Also anstatt primitive Datentypen zu nutzen kann man wenn man referenzierten Datentypen nutzt halt vlt verschiedene Funktionen/Methoden aus den Nutzen. Schon Intuitiv mit String immer gemacht.

Beispiel im Bild wurde boolean a erstellt und jz kannst z.b: a.compareTo(false); --> false weil a ja true war etc. Solche Sachen haben auch die anderen

Wie man die Schreibt halt: alles ausgeschrieben und Groß also von "int"... zu "Integer"...

Autoboxing = wenn die Variable die man deklariert hat von einem primitive n Datentypen, automatisch zu refferenzierten konvertet werden kann. Also Dass z.B.: Boolean a = true; kein Fehler ausgibt da true ein "boolean" ist und der Typ eig nicht übereinstimmt.

Unboxing = das gleiche nur zurück: von referenziert zu primitiv

Deswegen geht auch:

```
25
26     if(a==true) {
27         System.out.println("This is true");
28     }
29
30 }
```

The screenshot shows a Java code editor with a terminal window below it. The terminal window displays the output of the program: "his is true". The code itself is a simple conditional statement that prints "This is true" to the console if the variable 'a' is true.

Obwohl a ja mit referenzierten "Boolean" getypt wurde wird von java automatisch erkannt und unboxed so wird a also primitiver boolean gesehen.

Achtung:

```
int a = 1234;
//würde gehen
double cobj = a;
//geht nicht!
Dobule cobj = a;|
```

Stabilität Sortierverfahren:

wenn bei der Sortierung gleichwertige Elemente ihre Reihenfolge zu einander beibehalten

- (Irrelevant wenn gleichwertige Elemente nicht unterscheidbar aber wenn z.B nach key sortiert wird aber 2 unterschiedliche Key gleiche Element haben)

- kann durch Erweiterung der Ordnungsrelation kompensiert werden

Insertionsort, Bubblesort sind stabil

Selectionsort nicht (aber kann man am anfang das min Element ganz nach vorne)

Mergesort (easysplit/hardjoin) stabil

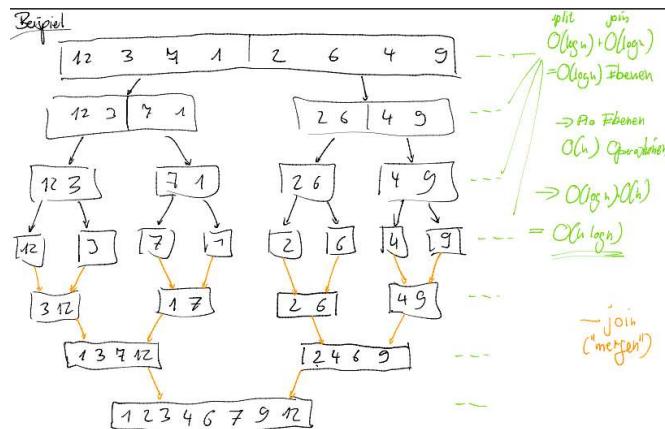
- Benötigt zusätzlichen Speicher $O(n)$

- Laufzeit immer $O(n \log n)$

Teilst nach prinzip devide and conquer alles so lange auf bis nur einzelne Elemente

Dann wieder Verschmelzen immer 2 benachbarte Teile zueinander und sortieren

Das so lange bis nur noch ein Element (die sortierte Liste/Array) da ist



Quicksort (hard split/easy join) nicht stabil

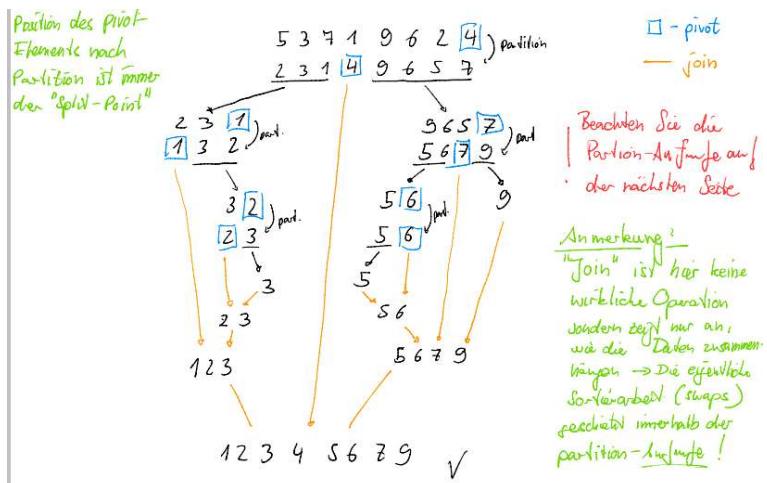
- Laufzeit $O(n \log n)$ Worstcase: $O(n^2)$ -> wenn pivot ungeünstig gewählt wird (pivot letztes Ele und Liste ist schon sortiert)

- Wählt ein Pivot und teilt Elemente entsprechend auf:

links alle Ele < pivot

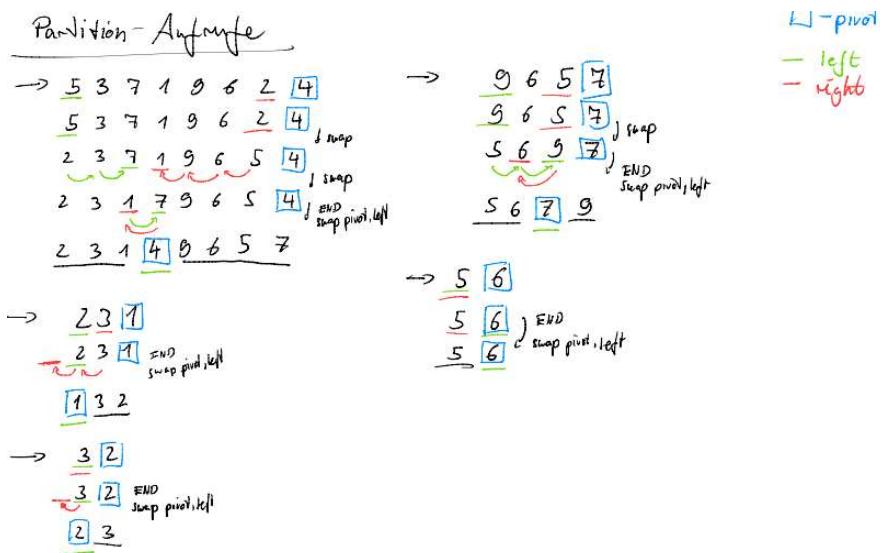
rechts alle Ele \geq pivot

Dann nochmal bis alles Element sortiert ist (einzelne Ele oder 2er Ele sortiert)



Beachten Sie die Partition-Aufzüge auf der nächsten Seite

Anmerkung:
"Join" ist hier keine wirkliche Operation sondern zeigt nur an, wie die Daten zusammen hängen. → Die eigentliche Sortierarbeit (swaps) geschieht innerhalb der partition-Aufzüge!



Man fängt von links an kuck ist i Ele größer als 4? Wenn ja also 5

Man fängt von rechts an kucken ist Ele kleiner gleich als 4? (pivot selber, also 4 wird übersprungen)
Wenn ja also 2

---> man tauscht pos 5 und 2

-> so weiter bis die suche sich trifft zwischen 1 und 7

-> wir wissen ja pivot liegt im rechten Teilbereich (es muss das kleinste oder gleich sein, deswegen kann es im Teilbereich rechts ganz nach links wandern) und ist somit pivot/punkt wo man splitten kann und das ganze nochmal im linken bzw. Rechten Teilbaum machen.

Bsp für nicht stabil:

Partition-Anfang:

