

# 20.02.24

→ JT - Video (Folio 4) (Folie 20)

WAS IST ein Algorithmus?

↳ das wäre eine typische Prüfungsfrage

↳ z.B. Code, der Daten verschieben

Verfahren zur Lösung eines (oder mehreren gleichartigen Problemen)

↳ sind berechenbar!

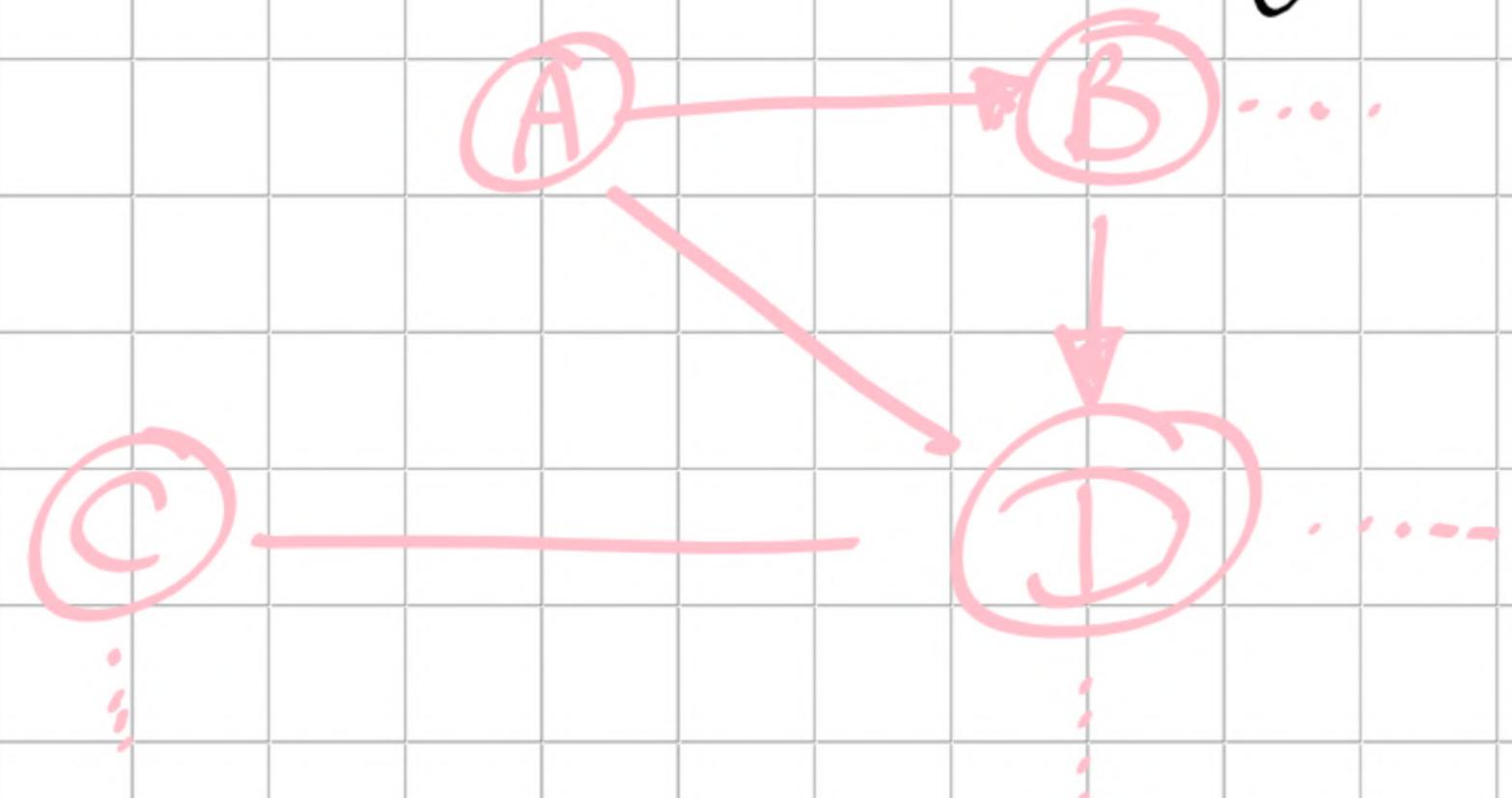
↳ Betonung auf Was, nicht das Wie

↳ Schrittweises Verfahren & TERMINIERT (d.h. hat ein Ende!)

↳ z.B. ARRAY, Liste

WAS IST eine DATENSTRUKTUR?

Ein "PLAN" zur Speicherung & Organisation von Daten. → STRUKTUR



⚠ To remember: RessourcenBEDARF

↳ benötigte Laufzeit (Zeitkomplexität)

↳ Speicherbedarf (Speicherkomplexität)

↳ Menge (10/1000'000 Elemente) & Werte ( $10^0$  oder 1'000'000)

da jeder Rechner anders ist: exakte Wert dieser Komplexität interessiert uns nicht.

aber es ist oftmals ein Trade-off auch

Bsp. Ist  $n$  eine Primzahl, ja/nein?

↳ a) 2, 3, 4, 5, ... über  $\text{sqrt}(n)$  lösbar? → lange Rechenzeit, wenig Speicher

b) 2, 3, 5, 7, 11, ... im Voraus in der Tabelle notieren.

↳ viel Speicher, schnell

- Eine exakte Analyse eines Algos ist unmöglich. Deshalb differenziert man:

- Best CASE
- Worst CASE
- Average CASE

- Komplexität eines Algos beschränkt sich nicht nur auf Zeit & Speicher. Auch z.B. Zeitbedarf? Garbage Collection, Anzahl cores, Compilation, etc.

⚠ immer überlegen: Wie wirkt sich der Algo bei größerem Ressourcenbedarf aus?

Big O-Notationen

damit lässt sich Zeit & Speicherkomplexität von Algos beschreiben

{ auf eine vereinfachte Weise }

"Wie bestimmen wir die Laufzeit des Algos (d.h. Code) in Abhängigkeit der Daten?"

O(n) ← beste Laufzeit, das wollen wir!

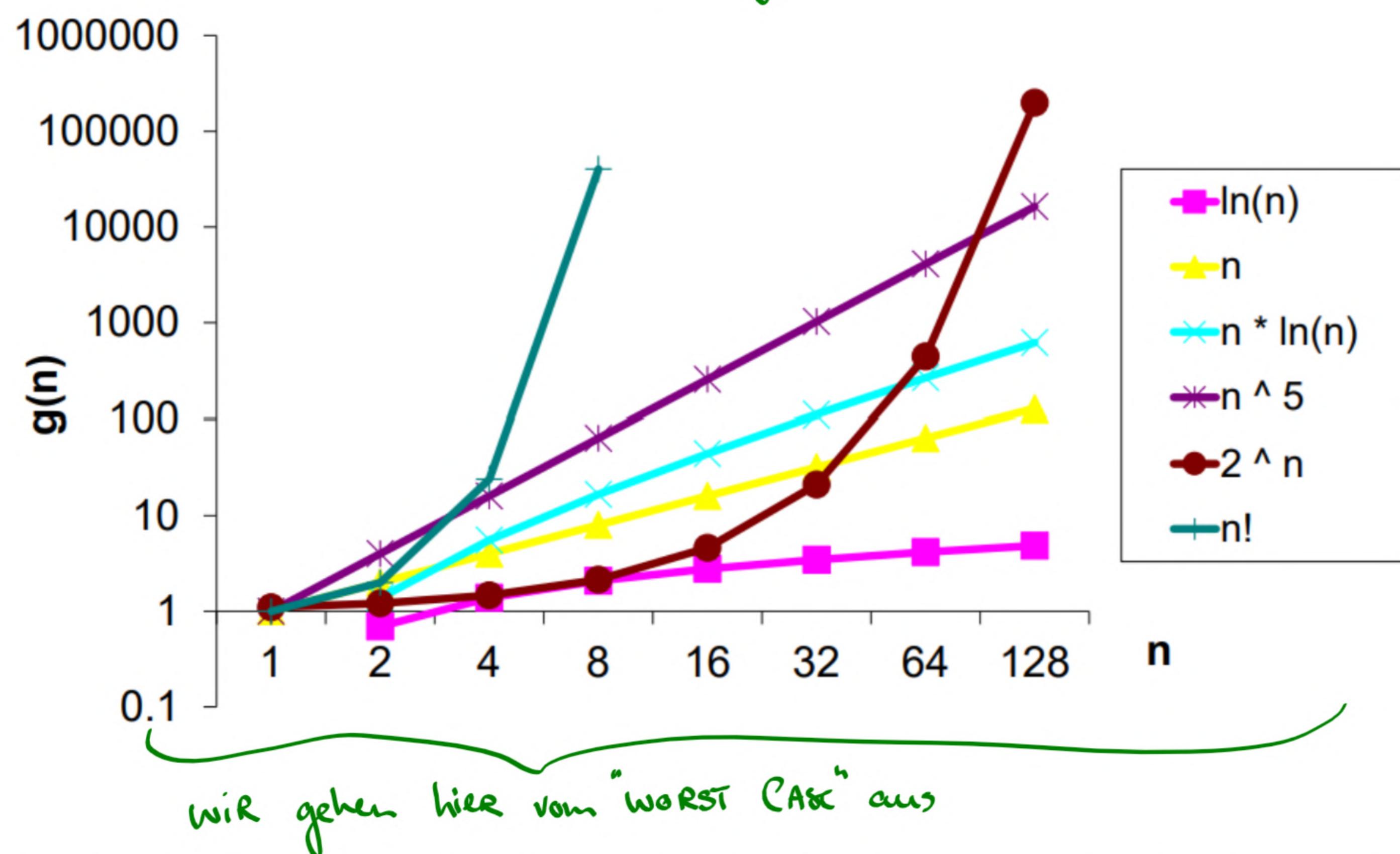
↳ Jedes Element muss 1x angefasst werden  
↳ deswegen ist es lineare, d.h. abhängig von Anzahl Daten. Wenn bei 100 Daten 1min geht, dann bei 200 geht 2 Minuten.

O(n<sup>2</sup>) ← absolut ungute Laufzeit (2)

↳ Jedes Element muss 1x angefasst werden & damit auch jedes andere Element.  
↳ z.B. bei Vergleichen: "Haben wie die Pizza Fungi: 2x im Datenstruktur"

Wie sieht die Kurve der Funktion  $f(n)$  entwickelt bei großem Ressourcenverbrauch?

↳ der Algo



größer, wobei wir mit der Laufzeit rechnen

O(1)

→ Konstant, z.B. HASHING

O(ln(n))

→ Logarithmisch, z.B. BINÄRES SUCHEN

O(n)

→ Linear, z.B. SUCHEN IN TEXTEN

O(n · ln(n))

→ n log n, z.B. RAFFINIERTES SORTIEREN

O(n<sup>m</sup>)

→ Polynomiale, z.B. SIMPLE SORTIEREN

O(d<sup>n</sup>)

→ Exponentielle, z.B. OPTIMIERUNGEN

O(n!)

→ FAKULTÄT, z.B. PERMUTATIONEN

es geht darum, sich selbst wieder aufzurufen

- Rekursion = Ähnlichkeit

- Ein Teil der Matrix ist selbst wieder eine Matrix

Rekursionsbasis (definiert, wann Rekursion terminiert wird)

- Rekursion

Rekursionsvorschrift (mathematische Formel, um Folge von Zahlen & Objekten zu definieren.)

REKURSIONSBASIS = Basisfall auch genannt

- definiert, wann Rekursion terminiert wird
  - definiert, wo die Folge beginnt
- } stellt sicher, dass Rekursion nicht unendlich läuft

REKURSIONSVORSCHRIFT

- beschreibt, wie jedes Element in der Folge basierend auf seinen vorherigen Elementen berechnet wird.

- Rekursion Verwendung: geometrische Muster als Rechnung beschreiben, wobei man beim anschauen der Rechnung nicht direkt auf das Resultat kommt.

↳ Bsp.  $\frac{1 \cdot 5}{5!} \rightarrow$  sieht man direkt: 5.

$\frac{5!}{5!} \rightarrow$  muss man rekursiv lösen:

$$\begin{aligned}
 PR(5) &= 5 \cdot PR(4) \\
 &= 5 \cdot 4 \cdot PR(3) \\
 &= 5 \cdot 4 \cdot 3 \cdot PR(2) \\
 &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot PR(1) \\
 &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120
 \end{aligned}$$

$= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$   
 $= 4 \cdot 6 = 24$   
 $= 3 \cdot 2 \cdot 6$   
 $= 2 \cdot 1 \cdot 2$   
 $= 1$

Rekursive Methode macht sich zuerst selber durch bis zur Rekursionsbasis und rechnet dann alles mit den initialen Werten durch.

Call & Heap Stack

- JVM verwendet für Ausführung Programmes 2 Speicher: Call & Heap Stack.

- Heap: Speicherung von Objekten, d.h. Instanzvariablen bzw. Zustände.

↳ nicht mehr referenzierbare Objekte  $\Rightarrow$  Garbage Collector löscht sie

- Call Stack: Bei Java wird eine Kette von Methoden aufgerufen & bearbeitet. Ursprung: main() - Methode. Jeder Methodenaufruf erfordert Speicher.

↳ aktuelle & lokale Variablen erfordern diesen Speicher.

Jeder neue Methodenaufruf macht Call Stack größer  $\rightarrow$  es wird immer ein neuer Stack Frame angelegt.

↳ Bsp.  $5!$   $\rightarrow$  für jeden Aufruf der Methode neues Stack Frame.

**JEDO ITERATIVE Lösung kann auch REKURSIV gemacht werden & umgedreht.**

↳ d.h. wir können  $5!$  sowohl rekursiv als auch iterativ lösen.

+ Rekursion

- Rekursion

- elegante Problemlösung
- wenig Quellcode
- Programmiersprachen, die nur Rekursion kennen

- viele Methodenaufrufe
- Speicherbedarf Call Stack hoch
- Gefahr Stack Overflow
- langsame Programmablaufzeit

Rekursions-Typen

= primitiv Rekursion

Lineare Rekursion: Aufruf der Methode führt zu höchstens 1x rekursiven Aufruf

Nichtlineare Rekursion:  $\begin{cases} \text{geschachtelt} \\ \text{nicht geschachtelt} \end{cases}$  → Ausführung der Methode führt zu mehr als einem rekursiven Aufruf

Direkte Rekursion: eine rekursive Methode ruft sich direkt selbst auf.

Indirekte Rekursion: eine rekursive Methode ruft sich indirekt selbst auf  $\rightarrow$  ruft zuerst andere Methode auf, die sich dann selbst wieder selbst auf

```

public static boolean isOdd(final int n) {
    if (n == 0) {
        return false;
    } else {
        return isEven(n - 1);
    }
}

public static boolean isEven(final int n) {
    if (n == 0) {
        return true;
    } else {
        return isOdd(n - 1);
    }
}

```

Die beiden Methoden rufen sich immer gegenseitig auf

## Wie implementiere ich eine einfache rekursive Methode?

→ z.B Berechnung der Fibonacci Folge mit Java & 2 Parametern:

```
public static int fiboRec1(int f1, int f2){  
    int fibonacci = f1+f2;  
    if(fibonacci >= 100){  
        System.out.println("Fibonacci succeded 100");  
        return 100;  
    } else {  
        System.out.println(fibonacci);  
        fiboRec1(f1:f2, f2:fibonacci);  
    }  
    System.out.println("Final Fibunacci number: " + fibonacci);  
    return fibonacci;  
};
```

→ Wenn man sich die Ausgabe auf der Console ansieht:  
leicht zu erkennen, dass zuerst Rekursion durchgegangen wird  
und dann wieder hochgerechnet, bzw. die Abarbeitung der Stack  
Frames durchgeht bis die Zahl über 100 geht.

```
--- exec-maven-plugin:3.0.0:exec (default-cli) @ oop_maven_template ---  
2  
3  
5  
8  
13  
21  
34  
55  
89  
Fibonacci succeded 100  
Final Fibunacci number: 89  
Final Fibunacci number: 55  
Final Fibunacci number: 34  
Final Fibunacci number: 21  
Final Fibunacci number: 13  
Final Fibunacci number: 8  
Final Fibunacci number: 5  
Final Fibunacci number: 3  
Final Fibunacci number: 2
```

Console  
Output

27.02.24

Reine Sammlung, wie Steinhaufen. Es interessiert nicht, wo das Zeugs liegt.

Bestimme Reihenfolge, die implizit beibehalten, wie FAHRRADKETTE. Kettenglieder sind miteinander verknüpft & jedes Kettenglied kennt seine 2 NACHBARGlieder.

implizit

Bestimmte Reihenfolge, wird aber SORTIERT wie vollautomatisches Hochregallager.

Elementare Methoden einfügen, suchen, entfernen, ersetzen von Elementen

in welcher Reihenfolge sind die Elemente abgelegt?

Operationen

Welche Operationen kann man auf den DS anwenden?

Operationen in Abhängigkeit einer Reihenfolge/SORTIERUNG (z.B. Maxima, Minima, Attributwerte, Nachfolger, Vorgänger, etc.)

z.B. Arrays: STATISCH, Größe muss bei Erstellung festgelegt werden. → Wie Getränkeflasche, max. Inhalt ist vorgegeben.

STATISCH vs. DYNAMISCHE

Muss man DS am Anfang direkt definieren oder wächst sie dynamisch?

z.B. Bäume & Listen: DYNAMICHE, ändern sich während der Laufzeit. Wie ein Luftballon → mehr oder weniger Luft, je nach Zustand

Explizite/Implizite Beziehungen

Wie stehen die Elemente in der DS in Beziehung zueinander?

Explizit: Beziehungen zwischen den Daten von jedem Element mit Referenz festgehalten. Jedes Element in der Liste enthält einen Verweis auf das nächste Element. (wie FAHRRADKETTE)

Implizit: Beziehungen werden von Außen definiert, z.B. über einen Index. Wie bei einem Bücherregal, wo Bücher geordnet nebeneinander stehen.

Direkter vs. Indirekter Zugriff

Wie kann man auf das gewünschte Element zugreifen?

Direkter Zugriff: Auf jedes einzelne Element direkter Zugriff. Wie Bücherregal, man kann jedes Buch direkt herausnehmen.

Indirekter Zugriff: Kein direkter Zugriff. Allerdings sequentiell nur. Wie Tellerstapel in der Mensa, man kann einen Teller nehmen, aber keinen bestimmten. Möchte man einen bestimmten, muss man alle vorhergehenden Teller umstapeln, bis der gewünschte kommt.

- Die Komplexität (Rechen- und Speicherkapazität, aka Big O) variiert für versch. Operationen. Oft in Abhängigkeit mit Datensumme & Struktur → mehr Daten, mehr Aufwand.

- Bsp. eines CHAR-ARRAY mit max. 16 Elementen.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P

hier ist das Array "voll", also alle Positionen sind belegt & sortiert eingefügt worden

mit max. 8 Elementen.

0	1	2	3	4	5	6	7
B	<leer>	A	M	I	<leer>	<leer>	<leer>

! leere Plätze mittan im Array möglichst vermeiden!

hier Array nur halb voll. Elemente weder sortiert noch fortlaufend eingefügt.

- Eigenschaften eines Arrays:

→ STATISCH: Größe wird bei Initialisierung festgelegt → kann während Laufzeit nicht geändert werden.

→ IMPLIZIT: einzelne Elemente haben keine Beziehungen untereinander, bzw. keine Referenzen aufeinander.

→ DIREKT: auf jedes Element kann direkt zugegriffen werden, weil sie indexiert sind.

→ REIHENFOLGE: Position der Datenelemente bleiben unverändert so wie zugeordnet/unverändert

- Suche in einem Array:

2 Fälle

1) DATEN NICHT SORtiERT, aber fortlaufend, ohne Lücken.

Das Array wird sequenziell durchsucht, d.h. jedes Element nacheinander.

→ Aufwand:  $O(n)$ , da linear abhängig von Datensumme.

0	1	2	3	4	5	6	7
B	G	A	M	I	<leer>	<leer>	<leer>

2) DATEN SORtiERT, fortlaufend gefüllt.

Das Array wird binär durchsucht.

→ Aufwand:  $O(\log(n))$ .

binäre Suche mit 8 Elementen,  
 $\log_2(8) = 3$   
 somit sind maximal 3 Vergleiche notwendig.

- Einfügen eines Elements in einem Array

1) - NICHT SORtiERT

- fortlaufend gefüllt, ohne Lücken.

- Wenn man Index des nächsten freien Platzes kennt:

Aufwand:  $O(1)$ , sonst  $O(n)$ .

0	1	2	3	4	5	6	7
B	G	A	M	I	<leer>	<leer>	<leer>

0	1	2	3	4	5	6	7
B	G	A	M	I	C	<leer>	<leer>

2) - SORtiERT

- man kann Position suchen, wo das Element eingefügt werden würde ( $O(\log n)$ ), aber man muss restliche Elemente nach rechts schieben! Aufwand:  $O(\log(n)) + O(n) = O(n)$

0	1	2	3	4	5	6	7
A	B	G	I	M	<leer>	<leer>	<leer>

0	1	2	3	4	5	6	7
A	B	C	G	I	M	<leer>	<leer>

## - Binäre Suche

**A** Hier müssen die Elemente sortiert sein, weil...

↪ so funktioniert es:

- 1.) Wert in der Mitte der DS herausgesucht.
- 2.) Vergleicht: Ist der Wert grösser/kleiner/gleich gross wie der gesuchte Wert?
- 3.) Wenn grösser, dann werden die Elemente vom Wert aus rechts betrachtet.  
Wenn kleinere, dann " links "
- 4.)

same process again

Hier noch als Algorithmus beschrieben:

- 1.) Datenmenge in der Mitte teilen.
- 2.) Auf Basis des Trennelements entscheiden, ob man in der linken oder Rechten Hälfte weitersucht.
- 3.) Algo rekursiv mit der ausgewählten Hälfte wiederholen.
- 4.) Algo endet, wenn Element = Element oder wenn nur noch 1 Element vorhanden ist.

## - Entfernen eines Elements im ARRAY

1) - NICHT SORTIERT

- fortlaufend befüllt
- wird sequenziell mit  $O(n)$  durchsucht & die Lücke wird mit dem letzten Element  $O(1)$  geschlossen.
- Aufwand:  $O(n) + O(1) = O(n)$

0	1	2	3	4	5	6	7
B	G	A	M	I	<leer>	<leer>	<leer>

0	1	2	3	4	5	6	7
B	I	A	M	<leer>	<leer>	<leer>	<leer>

- SORTIERT

- ARRAY wird mit Aufwand  $O(\log(n))$  durchsucht, aber entstehende Lücke muss direkt links rücken mit  $O(n)$  geschlossen werden.
- Aufwand:  $O(\log(n)) + O(n) = O(n)$

0	1	2	3	4	5	6	7
A	B	G	I	M	<leer>	<leer>	<leer>

0	1	2	3	4	5	6	7
A	B	I	M	<leer>	<leer>	<leer>	<leer>

## - Bilanz von ARRAYS

- Suche in einem Array ist sehr schnell, wenn es sortiert ist. (Anhängen am Schluss geht ebenfalls schnell)
- bieten mit den Indexen einen direkten & einfachen Zugriff auf die Elemente
- leider statisch, funktionieren sehr primitiv, d.h. beim Einfügen müssen Elemente verschoben werden.
- In Java sind Arrays nicht mit Generics kompatibel → Collections meist bevorzugt.

↪ **Jedes Array ist selber ein Objekt.** Deshalb müssen mit `new` instanziert werden. Können auch als Objektreferenzen z.B. an eine Methode übergeben werden.

## - Wie Arrays am besten nutzen?

- Wenn Datenmenge klar beschränkt, von Anfang an bekannt & eher klein.
- Wenige & elementare Datentypen müssen abgelegt werden.
- VORZUZIEHEN: Collections! Sind objektorientierter & erlauben z.B. direkten Zugriff per Index (Bsp. `ArrayList`)
- VERMEIDEN: bei Schnittstellen

## - Was sind Listen?

→ z.B. **LinkedList**:  halten einander fest am Rüssel

enthalten eigentlichen DATENWERT + Referenz auf nächstes Element. = **einfach verketzte Liste.**

**Doppelt verketzte Liste:** 

enthalten eigentlichen DATENWERT + Referenz auf nächstes Element + auf vorheriges Element  
(d.h. man kann von vorne nach hinten iterieren & auch hinten nach vorne. Es gibt Referenzen auf das erste Element (**head**) als auch letzte (**tail**)).

## - Implementieren einer Liste

In 2 Klassen: Klasse selber & Behälter für Elemente

- ↳ **List**:
  - enthält Referenz auf das erste Element (**head**)
  - Hilfsattribut z.B. für Anzahl enthaltener Elemente.
  - Methoden für versch. Operationen.

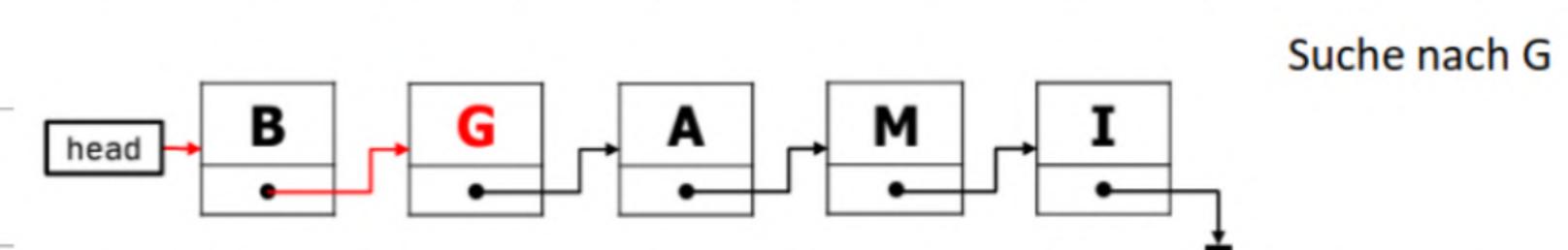
- ↳ **Node / Knoten**:
  - Je nach Listentyp (einfach/doppelt) 1 oder 2 Referenzen auf Vorgänger bzw. Nachfolger
  - Attribut(e) für die effektiv enthaltenen Daten.

## - Eigenschaften von Listen

- **Dynamisch**: Größe der DS passt sich der Anzahl der zu speichernden Elemente an.
- **Explizit**: Haben eine BZ untereinander - sie kennen Nachfolger und bei doppelt verketzten Listen auch Vorgänger.
- **Indirekter Zugriff**: Es kann nicht direkt auf Objekte zugegriffen werden. Es muss von head aus sequenziell vorwärts (& rückwärts) zugegriffen werden.
- **Reihenfolge**: Liste behält Positionen der Datenelemente so wie eingelegt bzw. zugewiesen bei.

## - Suchen eines Elements in Listen

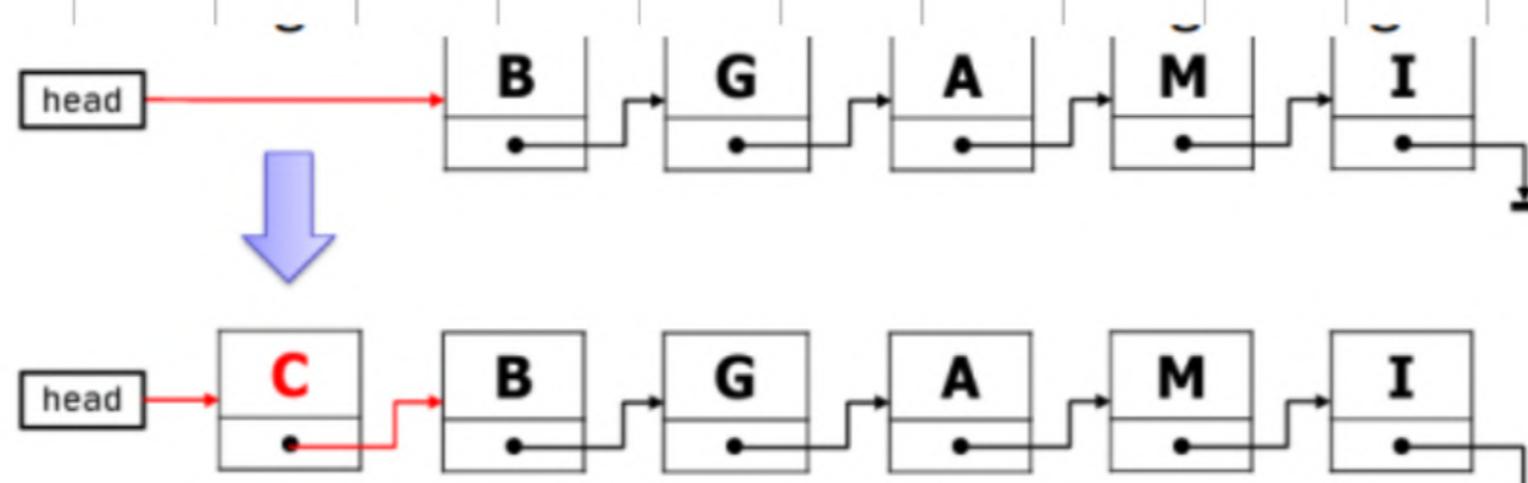
Da kein direkter Zugriff: Aufwand  $O(n)$ , unabhängig ob sortiert/unsortiert, einfacht/doppelt verketzt.



## - Hinzufügen eines Elements in unsortierte Liste

### 1) Einfach verketzte Liste:

- Paul am Anfang.
- Aufwand:  $O(1)$ .



### 2) Doppelt verketzte Liste:

- Paul am Anfang
- Aufwand:  $O(1)$

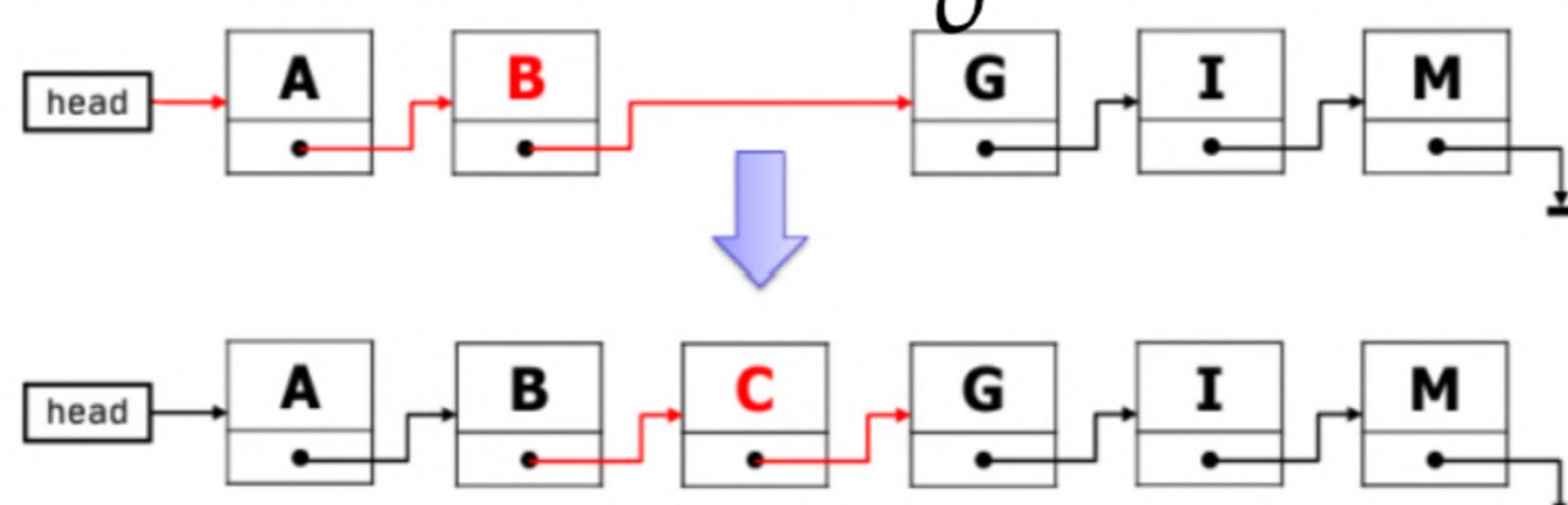
+ Entfernen funktioniert similar.

## - Hinzufügen eines Elements in sortierte Liste

### 1) Einfach verketzte Liste:

- Liste muss zuerst sequenziell durchsucht werden.
- kein Verschieben der Elemente, nur Anpassung der Referenzen.

- Aufwand:  $O(n)$



### 2) Doppelt verketzte Liste:

{  
- "  
" same

## - Listen: Vorteile gegenüber Arrays:

- Dynamisch: nehmen beliebige Datenmenge auf & belegen keinen festen Platz → wachsen & schrumpfen mit Datenmenge mit.
- Perfekt für (sehr) grosse, stark variierende Datenmengen.
- Konstantes & schnelles Einfügen, gerade bei unsortierten Listen
- Sind objektorientiert, unterstützen Generics

## - Stack

Ist ein Stapelspeicher, wie ein Tellerrahmen. Mit `push()` können neue Elemente oben abgelegt werden. Mit `pop()` kann das oberste Element entnommen werden.

### Implementation vom Stack mit Array

- Index des jeweilig letzten Elements wird gespeichert
- `push()`: anhängen am Ende,  $O(1)$
- `pop()`: entnehmen am Ende,  $O(1)$
- Man merkt sich jeweils den Index des letzten Elementes.
- Array ist statisch, Größe somit beschränkt.
- Maximaler Platz ist immer belegt weil bereits reserviert.
- Dadurch ist ein maximal schnelles Einfügen möglich!

### Implementation vom Stack mit Liste

- Eine einfach verkettete Liste reicht aus
- `push()`: einfügen am Anfang der Liste,  $O(1)$
- `pop()`: entnehmen am Anfang der Liste,  $O(1)$
- Einfach verkettet Liste reicht aus.
- Leerer Stack benötigt (fast) keinen Platz.
- Größe dynamisch und nur durch Speicher begrenzt.
- Speicheranforderung für neue Element notwendig, darum im Vergleich zum Array leicht langsamer!

Bei beiden Implementationen ist der Aufwand konstant & somit unabhängig von der Datenmenge.

## - Queue

Ist eine DS, welche Elemente in einer (Warte-)Schlange speichert (wie an der Kasse). Mit `enqueue()`/`offer()` wird Element am Ende der Queue angehängt. Mit `dequeue()`/`poll()` wird Element am Anfang der Queue entnommen.

### Implementation von der Queue mit Array

- Trickreiche Implementation mit (statischem) Array: Man implementiert einen «Ringbuffer» so, dass man die Elemente nicht verschieben muss. Es gibt je einen Index für das erste und das letzte Element welche rotieren. Somit gilt:
  - `enqueue()`: Einfügen «am Ende» (`put`), Aufwand  $O(1)$ .
  - `Dequeue()`: Entnehmen «am Anfang» (`get`), Aufwand  $O(1)$ .
- Wichtig ist: die Indexe dürfen sich nicht gegenseitig überholen!
- Array ist statisch, maximale Größe somit festgelegt.
- Maximaler Platz immer belegt und reserviert.
- Darum aber auch wieder sehr schnell (vgl. Stack)!

### Implementation von der Queue mit List

- Man verwendet eine doppelt verkettete Liste, damit man schnellen Zugriff auf head und tail hat.
- `enqueue()`: Einfügen am Ende der Liste, Aufwand  $O(1)$
- `dequeue()`: Entnehmen am Anfang der Liste, Aufwand  $O(1)$ .
- Leere Queue benötigt (fast) keinen Platz.
- Größe dynamisch und nur durch Speicher begrenzt.
- Speicheranforderung für neue Element notwendig, darum im Vergleich zum Array wieder etwas langsamer!

## - Implementation von ArrayList als Bsp.

Damit jedoch die Implementationen dieser Datenstrukturen mit beliebigen Elementen gut und ohne Fehler geht werden ein paar grundlegende Funktionen benötigt. 1. Muss man bei einer Suche die Objekte auf Gleichheit prüfen können und 2. Muss man bei einer Sortierung auch feststellen können ob ein Objekt grösser, kleiner oder gleich ist. Daraus resultiert dass man `equals()` und `hashCode()` jeweils überschreiben muss und zusätzlich für die Sortierung ggf. Folgende Interfaces implementieren muss: `Comparable<T>` oder `Comperator<T>`

### Beispiel Implementation von ArrayList

```
final List<Temperatur> verlauf = new ArrayList<>();

verlauf.add(new Temperatur(10.2));
verlauf.add(new Temperatur(15.8));
verlauf.add(new Temperatur(21.3));

System.out.println("Anzahl Messwerte: " + verlauf.size());
                                         Anzahl Messwerte: 3

verlauf.set(1, new Temperatur(18.5));

final Temperatur t2 = verlauf.get(1);
System.out.println("Zweiter Messwert: " + t2.getcelsius());
                                         Zweiter Messwert: 18.5
```

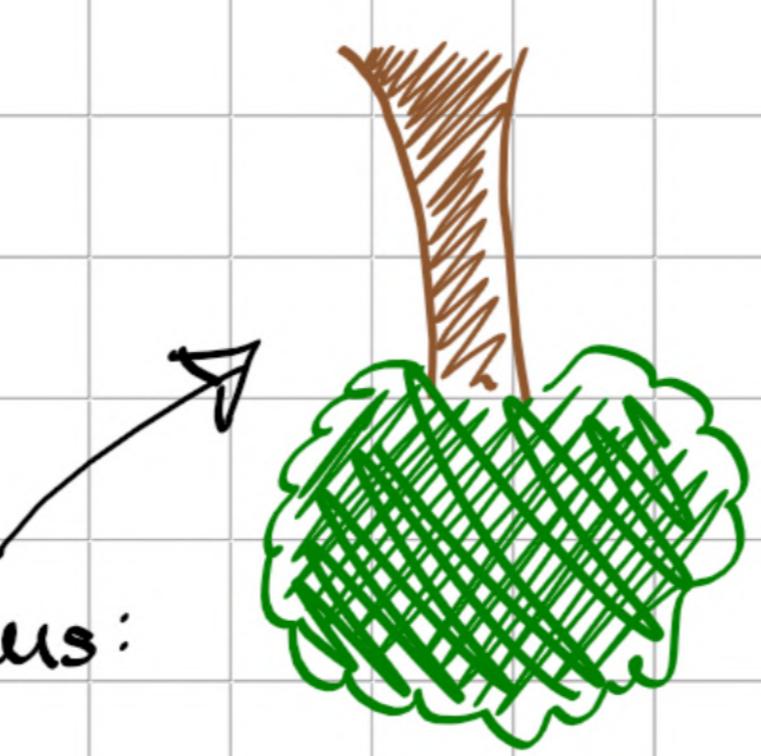
Zuerst wird eine neue `ArrayList` erstellt mit dem Typ `Temperatur` und der wird `verlauf` genannt.

Dann werden drei Temperaturen dem Verlauf hinzugefügt.

Danach wird im `verlauf` an der Stelle 1 eine neue Temperatur eingefügt.

Zum Schluss wird dieser Wert wieder ausgelesen und überprüft ob er korrekt überschrieben wurde.

# 05.03.2024



Bäume sehen in der Informatik so aus:

- Bäume genutzt für 2 Szenarien:
  - 1) Daten haben hierarchische Struktur wie im File Explorer oder Vererbungshierarchie in Java.
  - 2) Daten haben explizite Anordnung um ein bestimmtes Ziel zu erreichen.  
Bspw. man benötigt eine schnelle Suche in einer geordneten Datenmenge.
- Binäre Bäume haben nur 2 Gabelungen. Dafür ist das Einfügen mit mehr Aufwand verbunden.
- Bäume können stark variieren →
  - Anzahl Äste (Gabelungen)
  - unterschiedliche Länge der Äste (Tiefe)
  - die Breite (Grad)
  - die Höhe der Bäume sehr variabel.
- Je nach Anwendungszweck definiert man  $\oplus$  oder  $\ominus$  Restriktionen. → gibt deshalb je nachdem einfache/beschränkte Algorithmen oder Zugriff auf Daten.

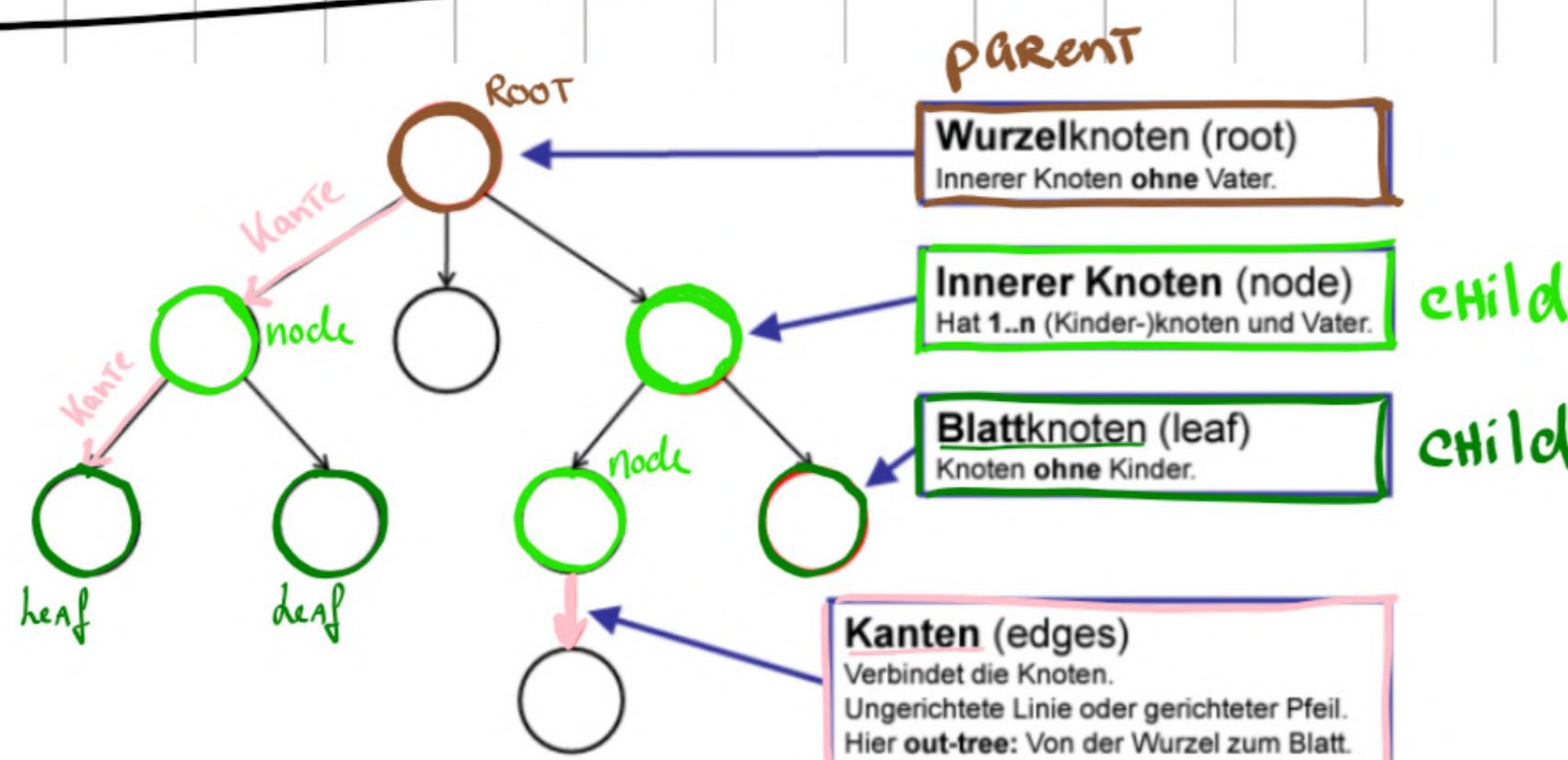
## Gerichtete & ungerichtete Bäume

= in welche Richtung kann ich von den Wurzeln aus navigieren?

↪ Ein normaler Baum ist ungerichtet, man kommt beliebig hoch & beliebig runter. Runter zu den Blättern hat man viele Entscheidungen, wieder hoch zu den Wurzeln ist der Weg sinnvoll.

- |                  |         |   |
|------------------|---------|---|
| <b>Out-Tree:</b> | Nav vom | . Pfeile gehen von der Wurzel aus. → Häufigster Fall! |
| <b>In-Tree:</b>  |         | . Pfeile zeigen zur Wurzel hin → eher seltener Fall.  |

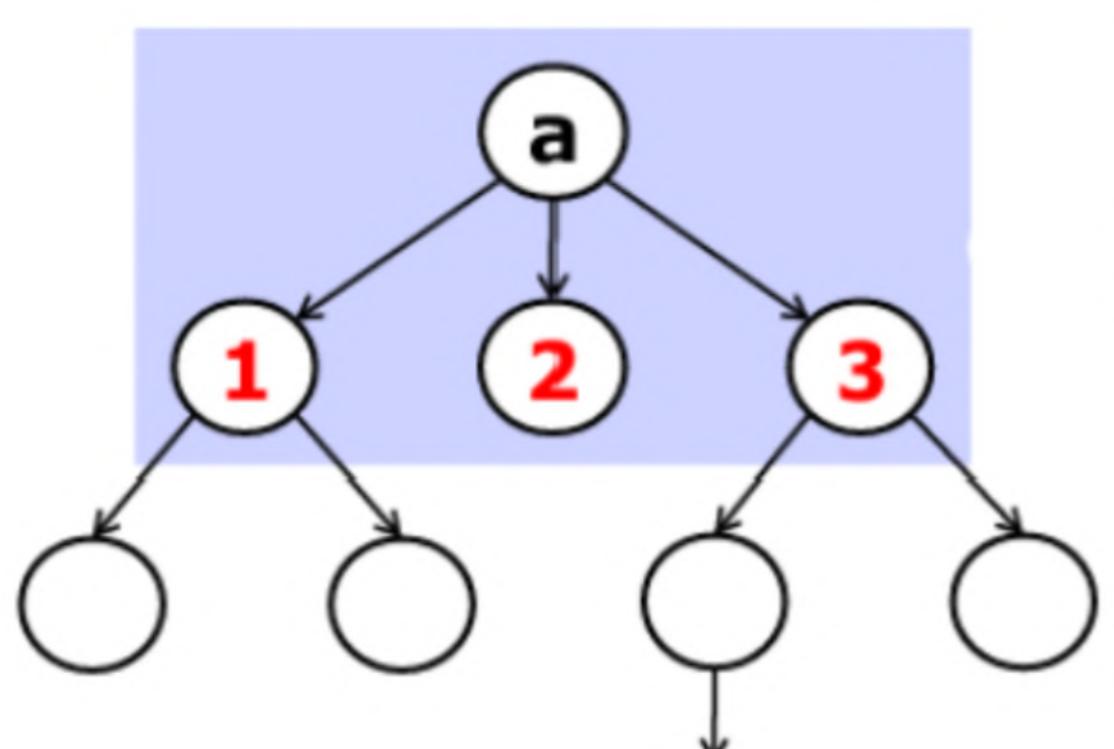
## Baum-ABC



- Bäume = Rekursive DATENSTRUKTUR (alle nodes verweisen auf andere nodes)
- = Monohierarchie: Jeder node ist genau 1 Vorgängernode zugewandt

## Kenngrößen von Bäumen

- Folgende Begriffe kennen: Wurzel, Knoten, Kind, Blatt, Pfad, Tiefe, Niveau, Ordnung, Grad, Gewicht.
- Hauptmerkmal ist **die Ordnung eines Baumes**. Also = Definition wie viele Kinder ein Node maximal haben darf.
- Die konkrete Form eines Baumes kann mit dem Füllgrad beschrieben werden: ausgefüllter Baum, voller Baum, vollständiger Baum



## Ordnung

Die Ordnung bezieht sich darauf, wie viele Children ein Knoten maximal haben darf, diese muss aber nicht zwingend erreicht werden.

→ Knoten a hier enthält mit 3 Kindern am meisten von allen. Also: Der Baum muss mindestens Ordnung von 3 besitzen. 2 max. 3 Children haben.

## GRAD

Bezieht sich darauf wieviele Children ein Knoten aktuell hat.

→ Knoten x hat 2 Kinder. Somit grad 2.

## Pfad (eines Knotens)

Bezeichnet den Weg (respektive Knoten) zu einem bestimmten Knoten/BLATT.

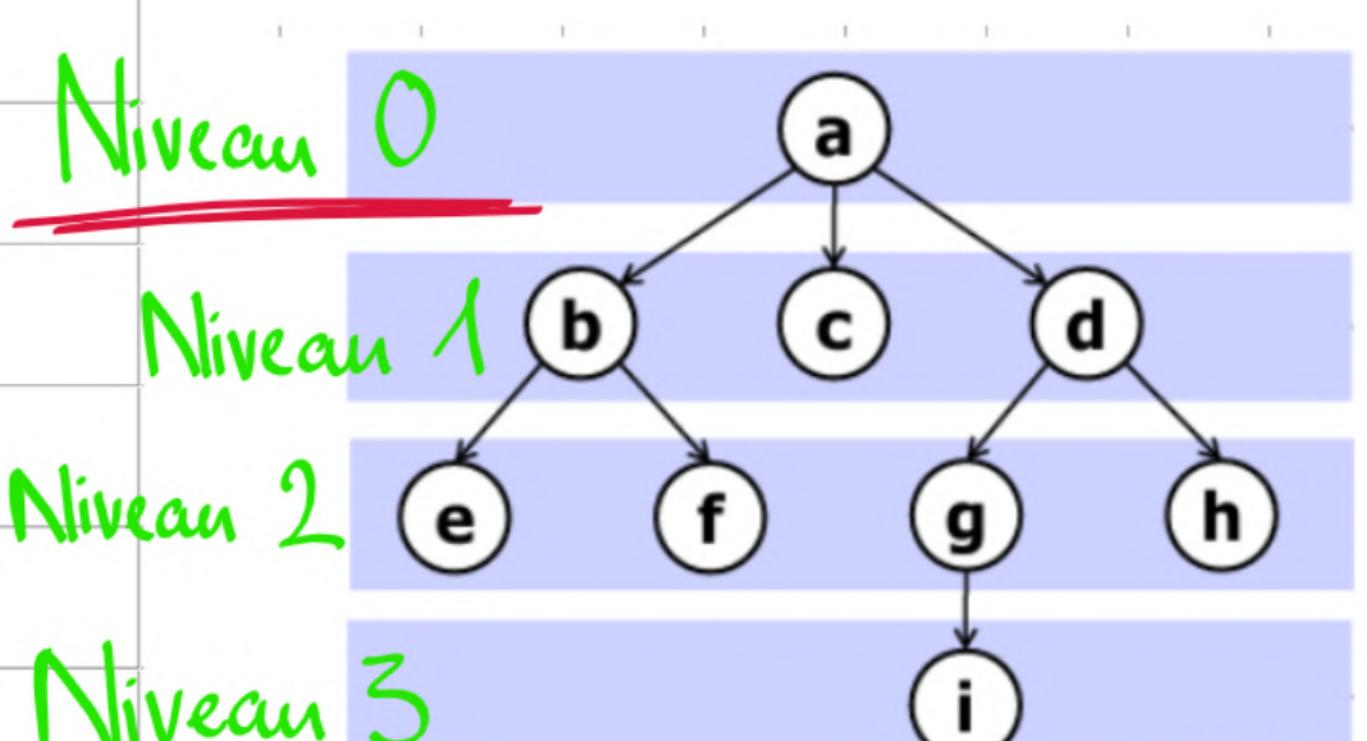
→ Der Pfad zum BLATT ";" lautet  $a \rightarrow d \rightarrow g \rightarrow i$ .

## Tiefe

Länge des Pfades zu einem Knoten, respektive Anzahl Kanten dorthin.

→ i hat eine Tiefe von 3, weil 3 Kanten (Pfeile dorthin)

## Niveaus / Ebenen



Ist die Menge aller Knoten mit der gleichen Tiefe.

## Höhe

Ist die Tiefe des Knotens, welcher am weitesten von der Wurzel entfernt ist. Dabei zählt die Wurzel als 1 (und nicht als 0).

→ Dieser Baum hat eine Höhe von 4.

## Gewicht

Ist Anzahl der im Baum enthaltener Knoten.

→ Anzahl Knoten insgesamt: 9. D.h das Gewicht ist 9.

## Füllgrad "Ausgeführt"

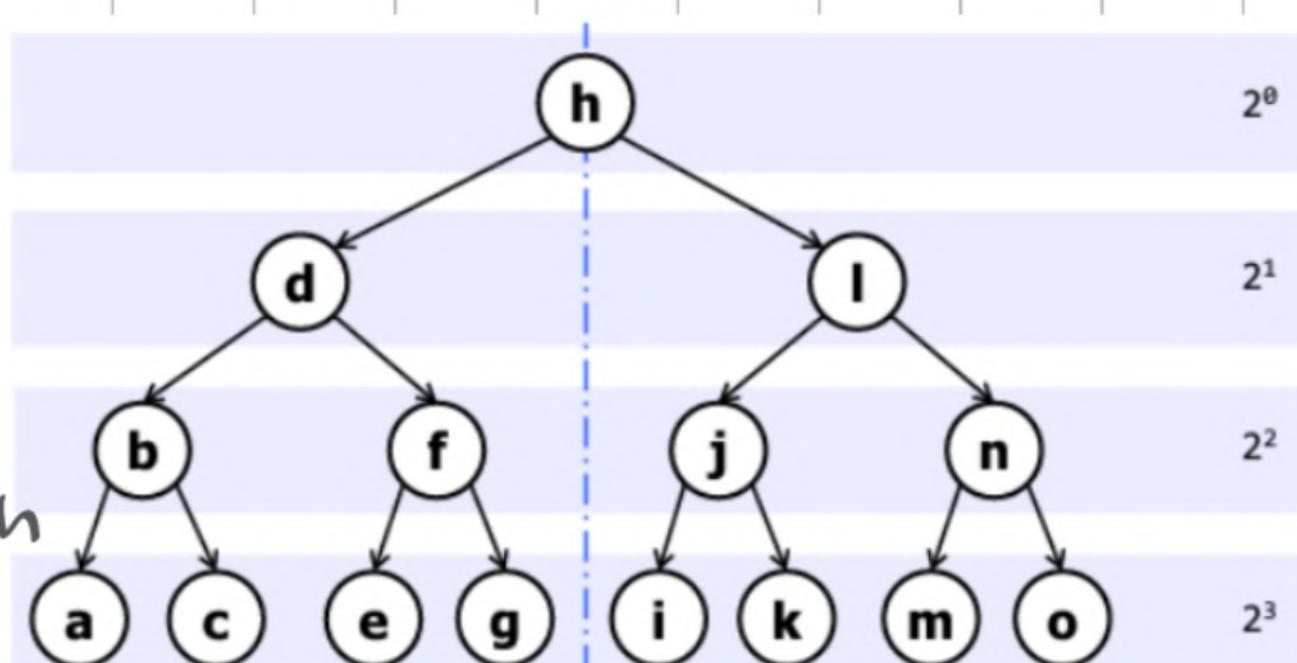
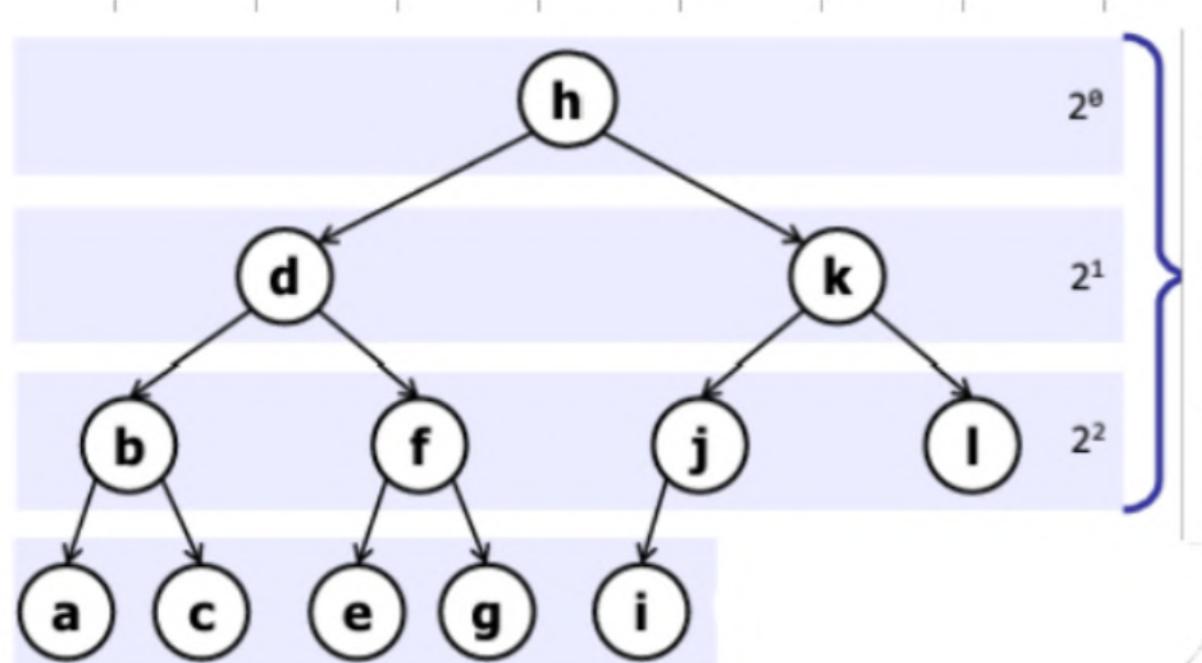
Alle inneren Knoten haben die maximale Anzahl Knoten. D.h jeder Node entspricht der Ordnung! → muss nicht zwingend voll sein! :)

## Füllgrad "Voll"

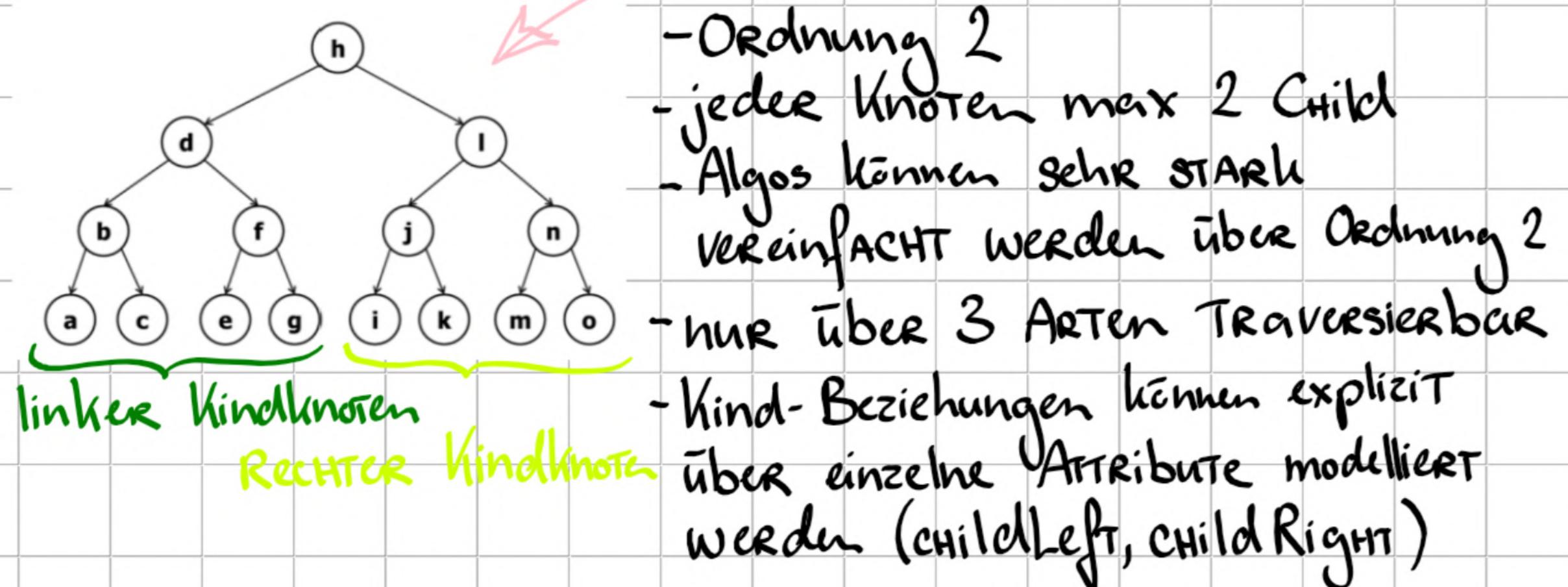
Alle Niveaus sind ausfüllt, und link/rechts-bündig angeordnet ist, aber nur auf einer Seite Children fehlen.

## Füllgrad "Vollständig"

Wenn alle Niveaus ausfüllt sind. Auch: Alle Knoten auf allen Niveaus die maximale Anzahl Children aufweisen.



# Der binäre Baum



DAS IST DER PERFekte binäre Baum!: Komplett, somit ausgeglichen, symmetrisch & balanciert.

## Der Binäre Suchbaum:

Wenn davon gesprochen wird, dann sind Elemente **SORTIERT** eingefügt. Das ist einziger Unterschied zu binären Bäumen.

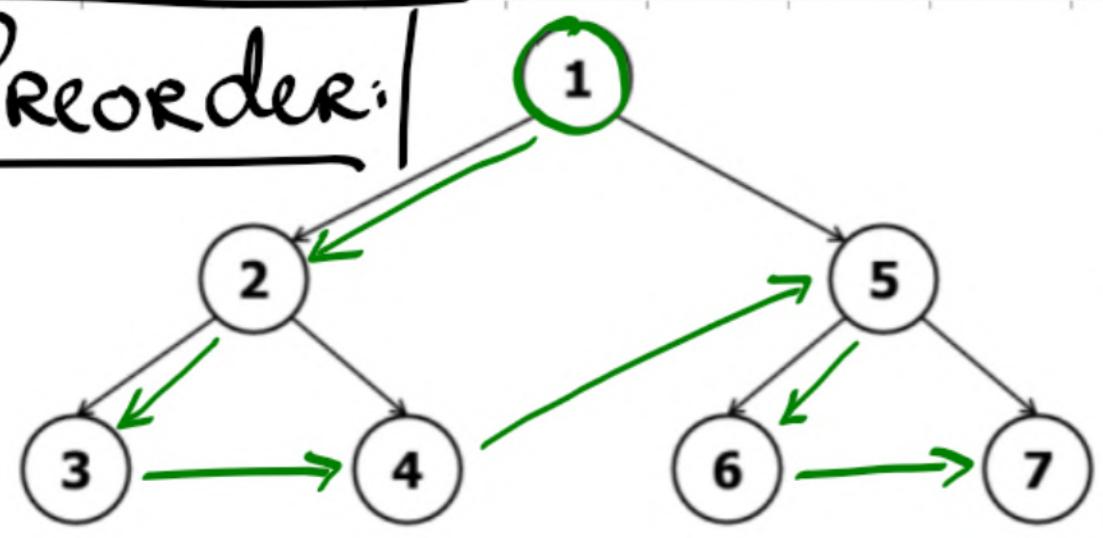
## TRAVERSIEREN - auf 3 unterschiedliche Arten

- Preorder (Hauptreihenfolge)
- Postorder (Nebenreihenfolge)
- Inorder (symmetrische Reihenfolge)

Algos, die diese 3 Traversierungsarten beschreiben, sind alle REKURSIV!  
↳ sind direkt abhängig von Anzahl Knoten, d.h. Aufwand =  $O(n)$ .

↳ nur nutzen, wenn man zwingend ALLE Knoten abarbeiten muss & NICHT für Suche!

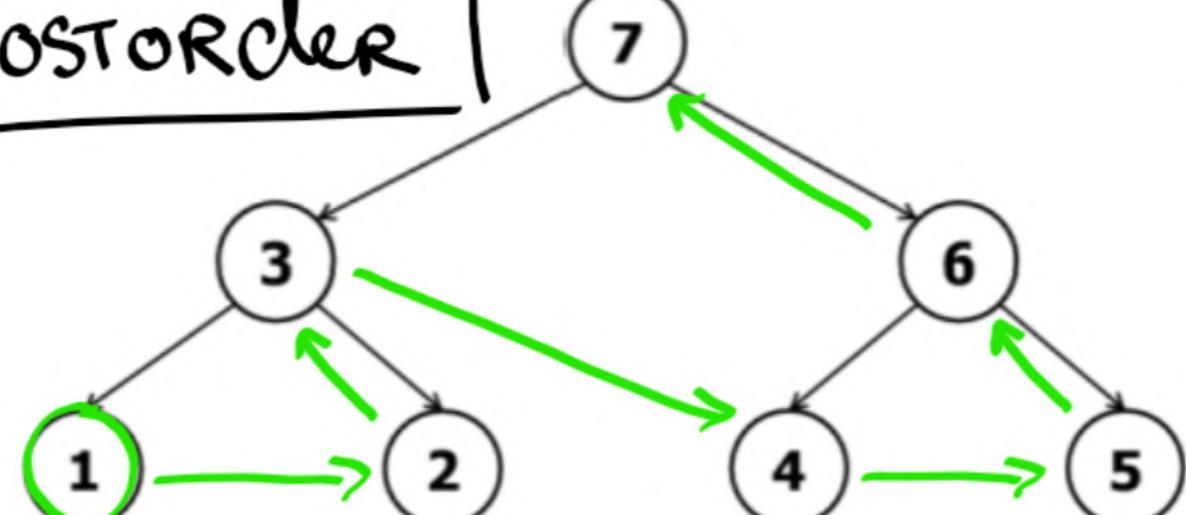
### Preorder:



→ Traversieren von linkem Teilbaum mit `preorder(x.getLeftChild())`.  
→ Rechts mit `" . ." getRightChild()`.

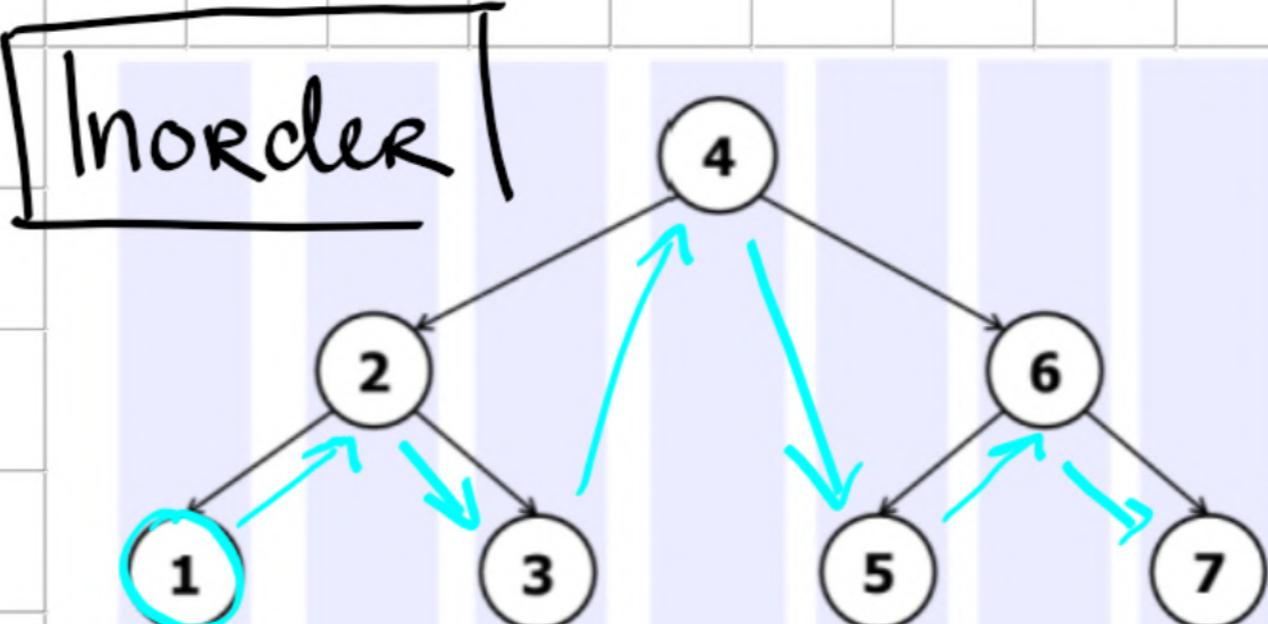
→ Wurzel → Links → Rechts

### Postorder:



→ Links → Rechts → Wurzel

### Inorder:

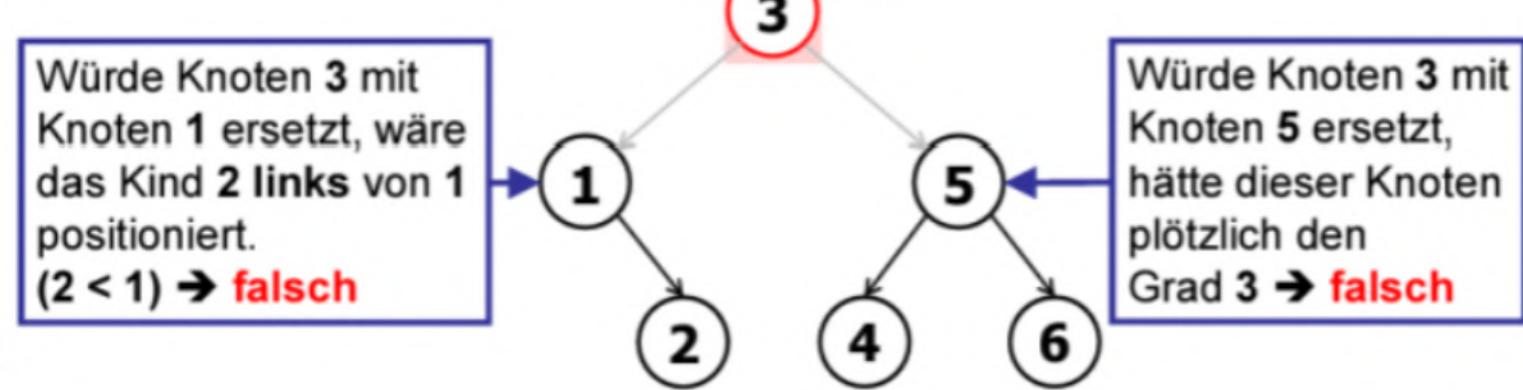


→ Link → Wurzel → Rechts

→ die Inorder-Reihenfolge liefert die enthaltenen Elemente gemäß ihrer Einordnung bzw. Sortierung von links nach rechts zurück.

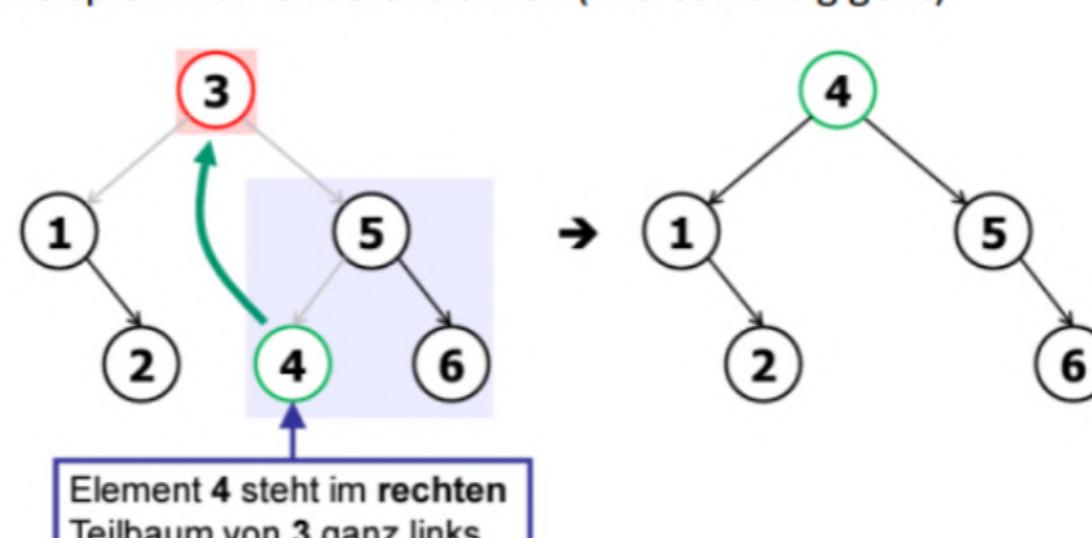
## Dritte Fall beim Entfernen eines Elements

Bsp: Wie wollen das Element 3 entfernen.



so:

Beispiel: Element 3 entfernen (wie es richtig geht).



!Also nicht!

dann ausgewichen, wenn Höhe links & rechts max 1 Unterschied hat.

## BALANCIEREN von BÄUMEN

→ Um optimale Performance zu erreichen, sollte ein Baum eine minimale Anzahl Niveaus zu seinem Gewicht haben. Daraus ergeben sich kürzere Pfade und man ist schneller somit.

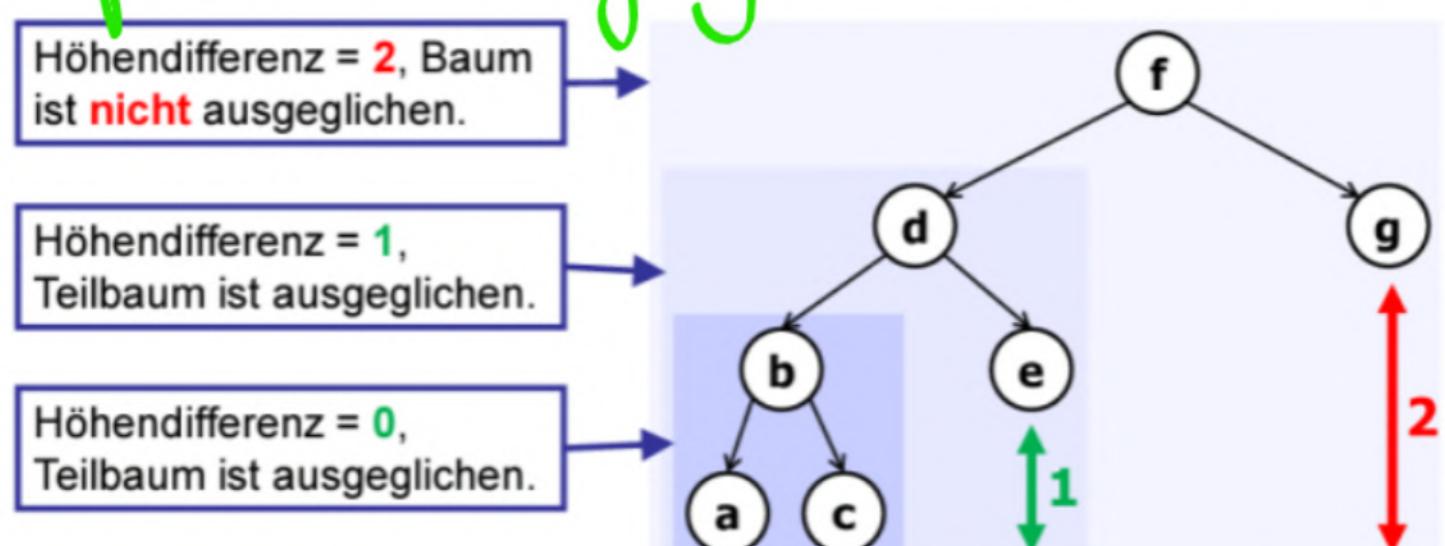
Option 1: Im Rechten Teilbaum Das kleinste Element wählen & an die Stelle tun.

→ DAS ERREICHT man, wenn man Ästen folgt & immer links hält.

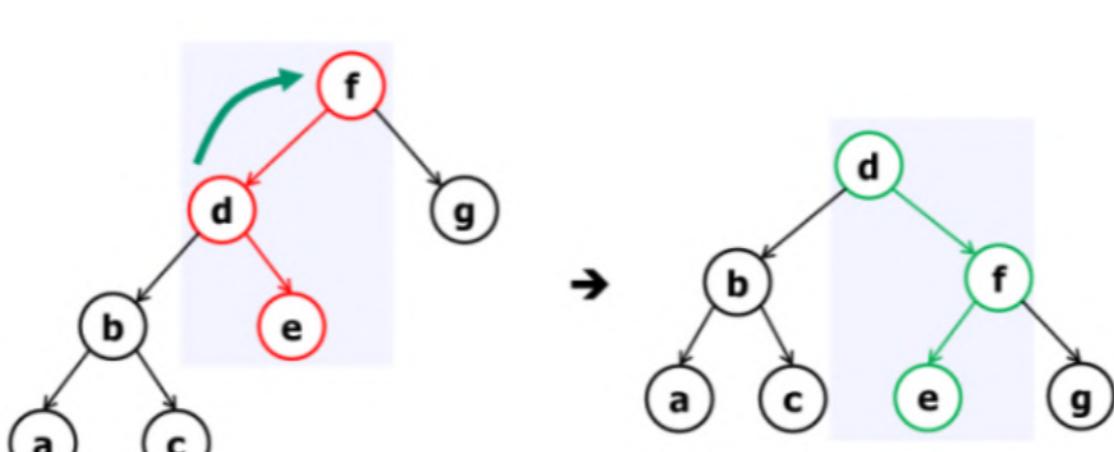
Option 2: Im linken Teilbaum Das grösste Element wählen & an die Stelle tun.

→ DAS ERREICHT man, wenn man Ästen folgt & immer Rechts hält.

Bsp. unausgeglichenes Baum.



Lösung:



Die Knoten d & f werden **ROTiert**. Bei gespiegelten Bäumen geht DAS mit der linken Seite analog.

→ evtl. auch Doppelrotation notwendig.

!Nach jeder Änderung im Baum muss überprüft werden, ob er immer noch ausgeglichen ist!

## Binäre Bäume in Java implementieren

- Über 2 Arten: `TreeSet<E>` und `TreeMap<K,V>`.
- `TreeSet`: Es wird ein navigierbares SorteSet<`E`> implementiert.  
Set: d.h. keine doppelten Objekte.  
Sorted: sortiert, d.h. kann mit Iterator in sortierter Reihenfolge durchlaufen werden (InOrder-Traversierung)
- ↪ nur `Comparable<T>/Comperator<T>`-Interface, somit muss `equals()` implementiert sein.
- `TreeMap`: DS mit Map-Semantik, welche Keys (auch als Set) zur schnellen Suche im Baum speichert.
- Beide Arten basieren auf dem **Red-Black-Baum**.

### Red-Black-Baum:

- 1) Knoten werden abwechselnd ROT/SCHWARZ eingefärbt. Kindknoten eines roten Knotens sind bspw. schwarz.
- 2) Wurzel & Blätter sind SCHWARZ.
- 3) Alle Pfade von der Wurzel zu den Blättern enthalten die gleiche Anzahl von schwarzen Knoten.

Werden diese Regeln verletzt, ist eine Reparatur des Baumes notwendig.

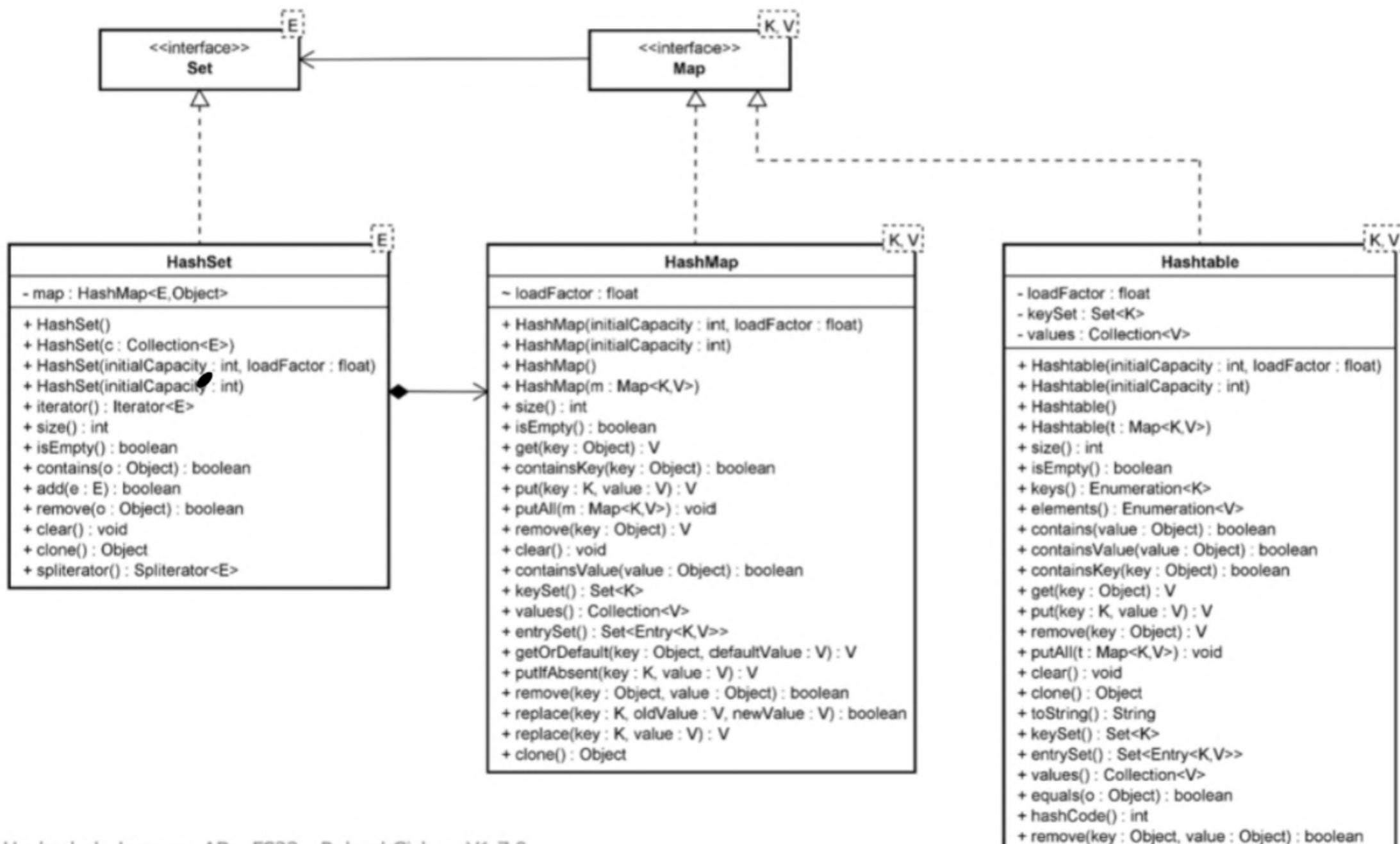
→ Aufwand für die Suche:  $O(\log(n))$ .

# 12.03.2024

- Hashwerte = numerische Datentypen, welche aus vorhandenen Daten berechnet werden kann. Er ist eindeutig & schnell berechnet.

↳ Idee: bei DS berechnet man auf Basis des Inhalts ein **int**, welcher direkt auf Speicherort des Datenobjekts in einer DS zeigt.

- Methoden **equals()** und **HashCode()** sind zu implementieren! Hier aus dem Java Collection Framework:



⚠ Objekte so oft wie möglich **immutable (final)** machen!

⚠ Objekte anstatt verändern lieber löschen & neu erstellen als mutieren.

- Wenn ein final Objekt nicht möglich ist und ein Objekt (oder ein Key) zwingend verändert werden muss, dann entnimmt man es aus der DS, ändert es, fügt es danach wieder ein.  
↳ aufwändig, aber sicher.

Objekte, die in hashbasierter DS eingefügt wurden, dürfen nicht mehr verändert werden!  
Sonst ändert sich ihr Hashwert & andere Elemente können nicht mehr gefunden werden!

## Tipps für die Java-Praxis

### DS & Nebentäufigkeit

- Folgende Klassen aus dem Java Collections Framework nicht mehr verwenden:

→ `java.util.Stack`  
→ `java.util.Vector`  
→ `java.util.Hashtable`

} sind spürbar langsam, weil synchronisiert implementiert. + alt!

↳ lieber so jetzt:

```
final List<Person> syncList =
  Collections.synchronizedList(new ArrayList<>());
syncList.add(new Person()); // Synchronisiert! → ≠ gleichzeitig!
```

- es gibt neben synchronisierten DS auch Collection-Implementationen, die mit atomaren Operationen parallele Operationen erlauben & ohne klassische Synchronisation "thread safe" sind.

→ `java.util.concurrent.ConcurrentMap<K,V>`  
→ `java.util.concurrent.ConcurrentLinkedQueue<E>`  
→ `java.util.concurrent.ConcurrentSkipListSet<E>`

## Hinweise zu equals()

Wird bei der Spezialisierung einer Klasse ein relevantes Attribut ergänzt, kann der **equals**-Kontrakt nicht mehr eingehalten werden. In diesem Fall: equals-Methode nicht überschreiben bei Vererbung! Lieber Vererbung hinterfragen. Methode **hashCode()** sollte dafür immer konsistent überschrieben werden.

1. Fertige implementierte `equals()`-Methode kritisch prüfen, finalisieren & testen.
2. Equals **final** machen.
3. Gleichheit auf Schlüssel-eigen schaffen beschränken.

## Hinweise zu hashCode()

Hashwerte von elementaren Datentypen:

- Int hash = Double.hashCode(doubleValue);
- int hash = (int) value;
- int hash = (value ? 1 : 0);
- int hash = Objects.hash(a1, a2 ...); <- einfachste Variante bleibt jedoch diese

Sind 2 Objekte gemäß der `equals()`-Methode gleich, müssen sie zwingend denselben Hashwert liefern.  
Objekte, welche nicht gleich sind, sollen möglichst versch. Hashwerte liefern (müssen aber nicht).

- es gibt keine perfekten Hashtes.
- Performance von hasht-basierten DS hängt direkt von Quali der Implementation ab.
- der Hashwert darf sich bei wiedeholten Aufrufen von `hashCode()` nicht verändern.

1. hashCode - Methode deshalb Testen.

↳ Implizit: EqualsVerifier

↳ Explizit: Hashwerte von ausgewählten Typischen Objekten kritisch überprüfen.

2. Bei finalen Objekten kann der Hashwert getatched werden → muss somit nur einmal bei Erzeugung des Objekts berechnet werden.

3. Nicht versuchen zu "genial" zu sein. Optimieren sie nicht, make it Dummy-proof

## Gebrauch von final

Davon Gebraucht machen, yes.

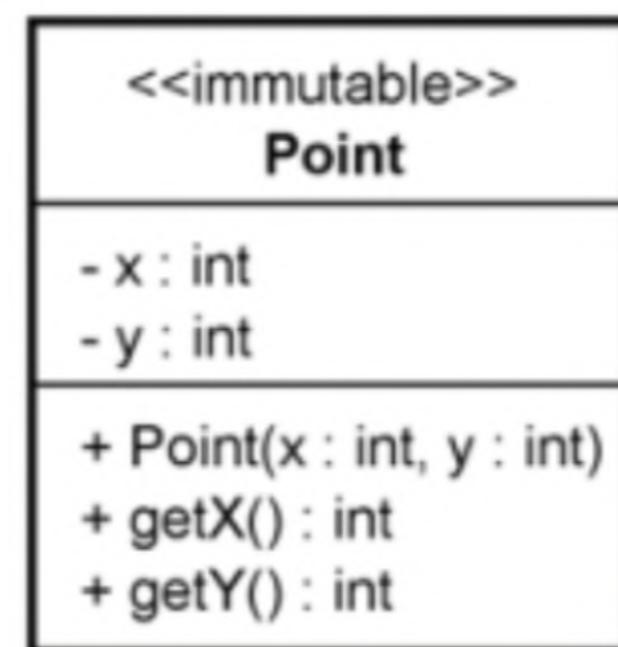
Wenn etw. wirklich verändert werden muss, erzeugt man ein neues Objekt & lässt das alte.

Eigenschaften einer unveränderbaren Klasse:

- Keine Methoden, welche Zustand eines Objektes verändern → SIEHTET!
- Keine Spezialisierung der Klasse möglich. → final Class {}
- Attribute sind alle privat & final. Können also nur bei Erzeugung gesetzt werden

## Visualisierung:

```
public final class Point {  
  
    private final int x;  
    private final int y;  
  
    public Point(final int x, final int y) {  
        this.x = x; this.y = y;  
    }  
  
    public int getX() {  
        return this.x;  
    }  
  
    public int getY() {  
        return this.y;  
    }  
  
    @Test  
    public void testImmutability() {  
        assertImmutable(Point.class);  
    }  
}
```



## Leere Collections, nicht null

weil es ist a) fehleranfälliger  
b) meistens geht die null-Prüfung verloren.

## | RETURN empty arrays or collections, not nulls.

→ IMMER DARAN denken, was für den Nutzer der Klasse am Einfachsten wäre & nicht für mich als Entwickler.

```
private final List<Wine> wineInStock = ...;  
  
public List<Wine> getWines() {  
    if (wineInStock.size() <= 0) {  
        return null;  
    }  
    ...  
}
```



So nicht!

↳ eleganter: Return leere Collection. Via:

`Collections.emptyList()`,  
`.emptySet()`,  
`.emptyMap()`,

} diese Collections sind unmodifizierbar.

↳ man kann nichts reinschreiben.

## Nutzen von generierten Collections

aus Java Collection Framework

- Z.B
  - `LIST<E>` : Generischer Typ mit Typparameter `E`.
  - `String` : Beispiel eines Typparameters.
  - `LIST<String>` : Parametrisierter Typ: "List of Strings"

`List<String> myList = new ArrayList<>();`

↑ Analoges Beispiel: man findet nicht PET im Restmüll wenn man vorher definiert hat, was in den Restmüll kommt.

- auch: keine Raw-Types wie `Object` mehr verwenden. Immer explicit typisieren:

`List myRawList = new ArrayList();`



`List<Object> myList = new ArrayList<>();`



- Versuchen, den bei Generics ab 8 zu auftretenden Warnings (vom Compiler oder der IDE) hartenmäßig auf den Grund zu gehen & das Problem wirklich zu lösen.

- auch: PREFER lists TO ARRAYS.

↳ Arrays sind für eher kleine Datensätze von elementaren Datentypen gut & effizient.

↳ Auf Interfaces sollte man auf Arrays verzichten! → mit generischen Typen nicht so verträglich.

- notfalls auf Third-Party OS zurückgreifen, aber nur wenn überhaupt nicht anderes geht, da → RISIKO.

19.03.2024

in Java heißt das Threading. Task = Thread  
Tasks = Threads

- Nebentäufig = Zwei + Tasks, die zeitgleich bearbeitet werden können
  - ↪ egal ob zuerst eine & dann andere ausgeführt wird
  - ↪ egal ob in umgekehrter Reihenfolge ausgeführt
  - ↪ egal ob gleichzeitig erledigt
  - ↪ Tasks dürfen nicht voneinander abhängen!

→ Wenn Rechner mehr als 1 CPU oder mehrere Rechenkerne hat, passiert die Nebentäufigkeit auch auf Hardwareebene.

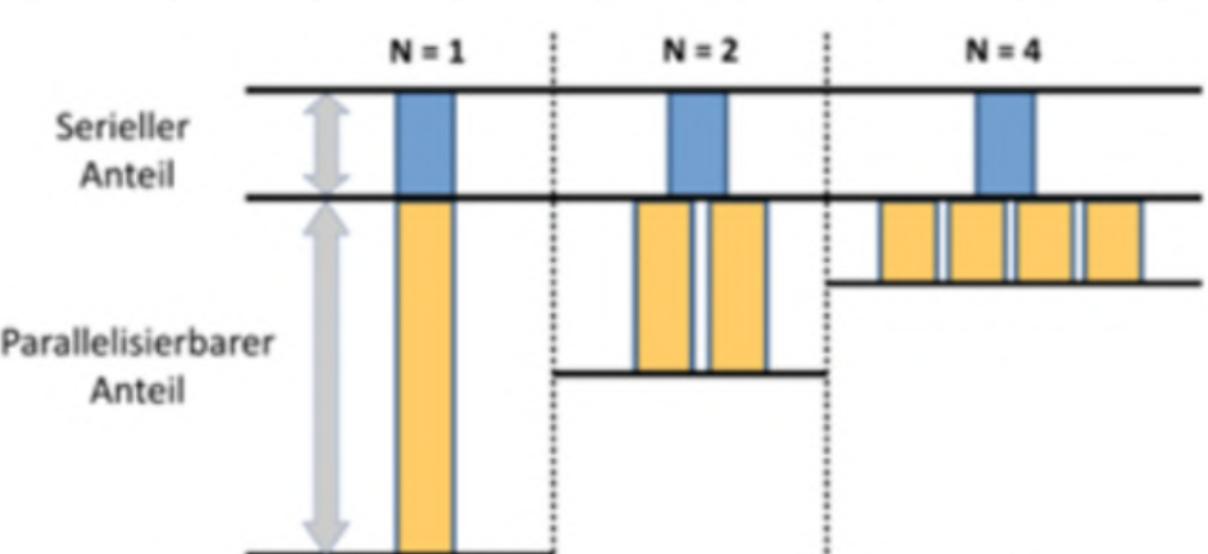
→ Es gibt eine Maßeinheit für den Performance-Gewinn, den man beim Einsatz von Nebentäufigkeit erzielen kann.

Dieser heißt: **Speedup**.

$$S = \frac{T_{\text{seq}} \text{ (Laufzeit mit einem Kern)}}{T_{\text{par}} \text{ (Laufzeit mit mehreren Kernen)}}$$

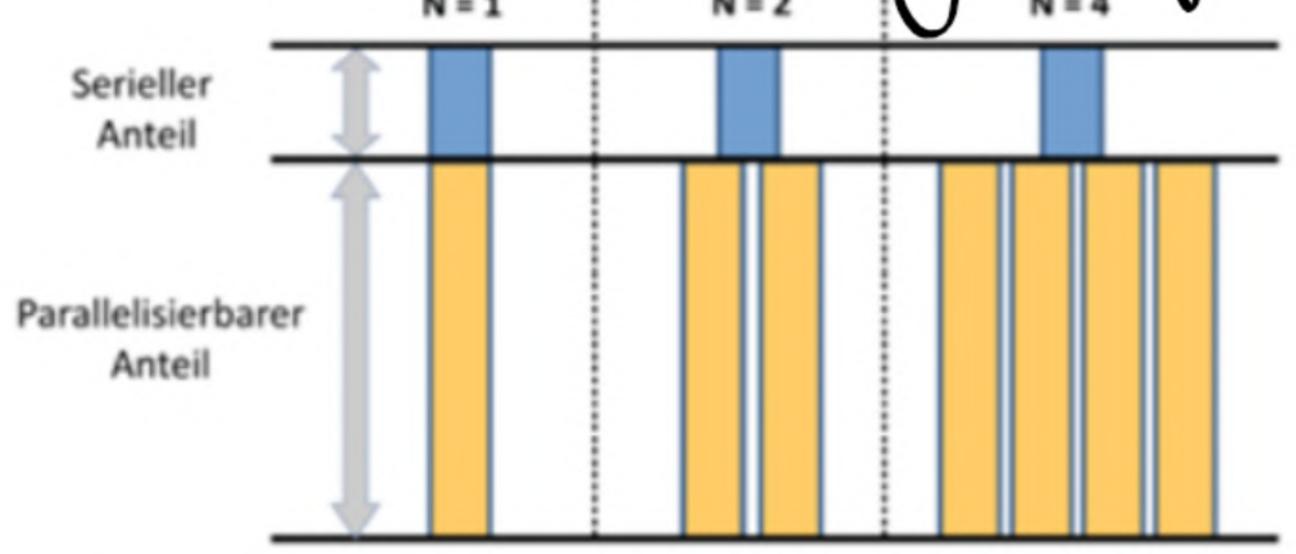
= Zeit, die es für eine bestimmte Aufgabe braucht.  
↪ sollte größer als 1 sein oder mindestens größer als 0!

→ Sichtweise von Amdahl:



Amdahl geht von einer festen vorgegebenen Programm bzw. fixer Problemgröße (→ Fläche des Balken) aus.

→ Sichtweise von Gustafson:



Gustafson betrachtet eine variable Problemgröße (→ Fläche der gelben Balken vergrößert sich) in einem festen Zeitraum

→ Erzeugen & STARTEN von Threads: über den Main-Thread. Ein Java-Programm wird in der JVM (Java Virtual Machine) ausgeführt. Die JVM selbst beansprucht einen Prozess des Betriebssystems. Mit **public static void main(String[] args)** wird der main-Thread in der JVM gestartet. Dann STARTET die JVM weitere Threads dazu.

→ Zugriff auf den ausführenden Thread erhält man über **Thread.currentThread**

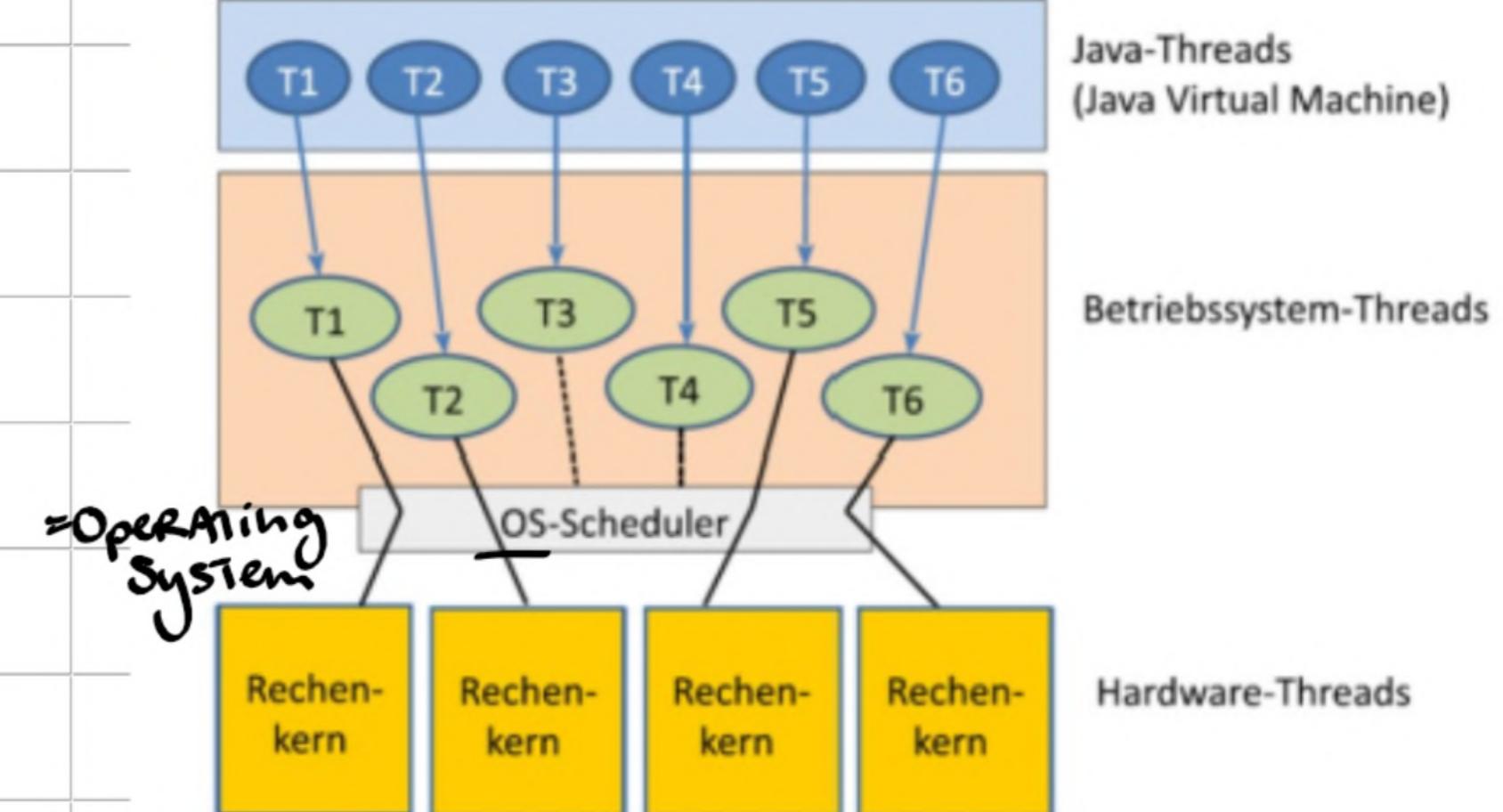
```
public static void main(final String[] args) {
    // Anzahl der Prozessoren abfragen
    final int nr = Runtime.getRuntime().availableProcessors();
    LOG.info("Available processors " + nr);
    // Eigenschaften des main-Threads
    final Thread self = Thread.currentThread();
    LOG.info("Name : " + self.getName());
    LOG.info("Priority : " + self.getPriority());
    LOG.info("ID : " + self.getId());
    LOG.info("Deamon? : " + self.isDaemon());
}
```

↪ entweder über class Thread oder Interface Runnable

→ Threads erzeugen: gemäß API-Dokumentation:

In most cases, Runnable interface should be used if you are only planning to override the run() method and no other Thread methods.

→ Bei der Verwendung von Runnable wird konzeptionell klar zwischen Programmfluss (Thread) und der nebentäufig durchzuführenden Aufgabe (Task) unterschieden.



Das System entscheidet wann welcher Thread ausgeführt wird. Nicht der Programmierer

## Thread Erzeugung via Runnable

→ Interface besitzt nur 1 Methode: **run()**

↳ hier sind Anweisungen implementiert die von Thread abgearbeitet werden.

↳ Instanzen der Task-Klasse werden dem Thread-Objekt über den Konstruktor zugewiesen.

```
public final class MyTask implements Runnable {
    @Override
    public void run() {
        //...Anweisungen - nebenläufig ausführen
    }

    public static void main(final String[] args) {
        final MyTask myTask = new MyTask();
        final Thread thread = new Thread(myTask, "MyTask-Thread");
        thread.start(); // Methoden zum Thread ausführen
    }
}
```

Es empfiehlt sich, Threads immer einen Namen zuzuordnen - es erleichtert die Fehlersuche!

**MASSIV-FUCKING-GIGA-CHAD-PRO-TIPP:**

Dem Thread einen Namen zuzuordnen macht die Fehlersuche extrem einfacher!

→ Im Konstruktor **new Thread()** wird die Methode mitgegeben welche nebenläufig ausgeführt werden sollte.

Main- und Worker Thread – ein Beispiel:

```
@Override
public void run() {
    for (int i = 0; i < 1000; i++) {
        System.out.print("y");
    }
}

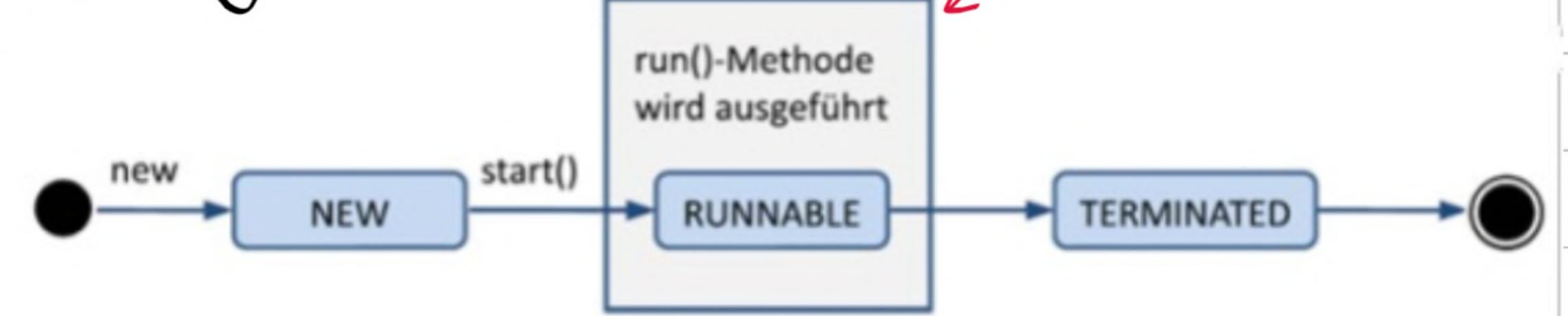
public static void main(final String[] args) {
    //...Anweisungen - siehe Folie 16
    thread.start();
    for (int i = 0; i < 1000; i++) {
        System.out.print("x");
    }
}
```

↓ Screenshot oben

Main Thread  
Worker Thread

Lebenszyklus eines Threads:

wenn hier oo-Schleife eingezeichnet, endet der Thread nie!

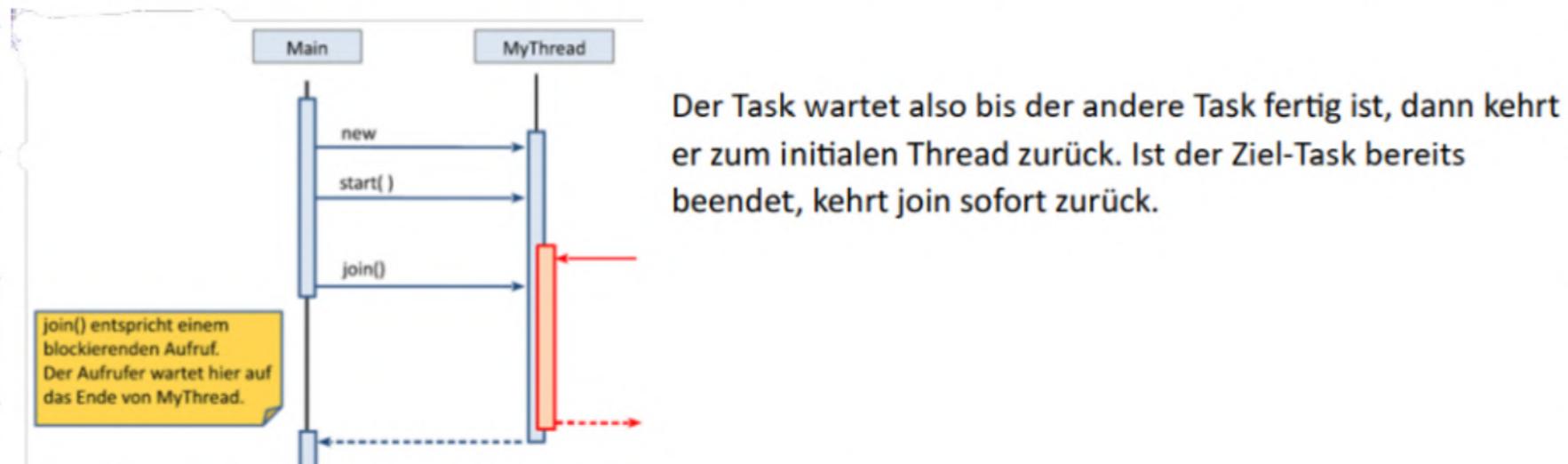


↳ sobald Thread gestartet ist, kann man ihn nicht erneut STARTEN.

↳ man kann aber Zustand von Thread abfragen, bzw. feststellen, ob sich ein Thread-Objekt im RUNNABLE-Zustand befindet (true or false)

→ zwei Tasks können mit **join** voneinander abhängig gemacht werden.

Der aufgerufene Thread wartet also so lange, bis der andere Thread done ist. Die Methode join ist blockierend & es gibt auch join Varianten, bei denen maximale Wartezeit angegeben werden kann.



## Beenden von Threads:

→ Der Thread soll...

- alle benutzten Objekte in konsistenten Zustand gebracht haben. Daten sind geschlossen.
- gemeinsame Ressourcen bereinigt haben (Streams)
- kritische Bereiche freigegeben haben
- keine neuen Aufgaben mehr anfangen.

Etwas muss zusätzlich noch beachtet werden. Mit dem Auslösen der Exception wird das Interrupt-Flag wieder auf false gesetzt. Gegebenenfalls muss das Interrupt-Flag wieder explizit gesetzt werden

```
...
@Override
public void run() {
    // Initialisierungsphase
    try {
        while (Thread.currentThread().isInterrupted() == false) {
            // Arbeitsphase
        }
    } catch (InterruptedException ex) {
        // Thread wurde in einer Wartemethode unterbrochen
        Thread.currentThread().interrupt();
    } finally {
        // Aufräumphase
    }
}
```

Wie reagiert man auf eine InterruptedException?

Nichts tun ist wie immer eine ganz schlechte Idee, das heisst ein leerer catch-Teil ist inakzeptabel.

Mindestens eine Exception Meldung ausgeben!

Man kann exceptions abfangen und terminieren:

- im catch-Teil die Beendung des Threads einleiten; z.B. ein return oder (besser) nochmals die Methode interrupt aufrufen, denn das Interrupt-Flag wurde zurückgesetzt!

Exception abfangen und eventuell nicht terminieren, wenn eine angefangene Arbeitsphase beendet werden soll. Im catch-Teil den Thread aufgrund einer Entscheidung beenden.

# 26.03.2024

z.B gemeinsame Ressource, auf welchen Threads lesen & schreiben (z.B Speicher, Datei, gemeinsame Datenstruktur)

- Was braucht eine gute Lösung für Zugriff auf gemeinsame Ressourcen?
  - ↳ In einem kritischen Bereich darf zu jedem Zeitpunkt max. 1 Thread befinden.
  - ↳ Es dürfen keine Annahmen auf die zugrunde liegende Hardware gemacht werden.
  - ↳ Thread darf andere Threads nicht blockieren (nur wenn er im kritischen Bereich ist.)
  - ↳ Threads dürfen nicht ewig lange warten um in kritischen Bereich einzutreten zu dürfen.

## Beispiel gemeinsamer Counter:

```
public final class SimpleCounter {
    private int count = 0;
    public int increment() {
        count++;
        return count;
    }
}
```

Codeskizze

yield ist der Wunsch an den Compiler den Thread zu wechseln, das passiert aber nur wenn ein Thread bereits im wait Zustand ist.

↳ Beispiel eines Zählers:

- Besitzt einen Zustand repräsentiert durch ein int Attribut.
- Besitzt eine Operation increment, welche den Zustand um 1 erhöht und einen eindeutigen Wert liefert.
- Der Counter kann von mehreren Tasks gleichzeitig benutzt werden.
- Die gemeinsame Ressource ist Counter mit Attribut int count.
- Der Main Thread startet Thread T1 und T2
- Annahme: Thread T1 und T2 laufen auf EINEM Prozessorkern (wie wir wissen darf man diese Annahme eigentlich nicht treffen)

yield ist der Wunsch an den Compiler den Thread zu wechseln, das passiert aber nur wenn ein Thread bereits im wait Zustand ist.

LÄsst man das durchlaufen...

count	Main Thread	Thread T1	Thread T2
0	new Counter();		
1	starten von		
1		count++;	
2		return count;	count++;
2			return count;
3		count++;	count++;
3		return count;	count++;
4		count++;	count++;
4		return count;	count++;
4			return count;
5		count++;	count++;
5		return count;	count++;
5			return count;
6		count++;	count++;
6		return count;	count++;
6			return count;

T1: 1  
T2: 2  
T1: 4  
T1: 6  
T2: 4  
T2: 5  
...

Wo ist die 3?  
Zwischen Zählerstand 2 und 4 ist etwas passiert,  
was nicht gewollt war.

RESULT

Erklärung dazu:

- Die Operation increment ist ein kritischer Abschnitt.
- Befinden sich mehrere Threads gleichzeitig in der Operation increment, überschreiben sich mehrere Threads gleichzeitig den aktuellen Zähler stand, wenn ein Taskwechsel stattfindet. Durch die parallele Ausführung entsteht eine Inkonsistenz!
- Das exakte Verhalten des Schedulers ist nicht deterministisch vorhersagbar, und somit ist das Endresultat ein zufälliges Ergebnis einer sogenannten «race condition».
- Dieses Fehlverhalten kann sehr selten auftreten und darum beim Testen oft nicht entdeckt. Es darf aber NIE ausgeschlossen werden!

Wie lösen wir das Problem?

→ Das Stichwort: **synchronized!**

→ Damit kann Zugriff auf den Lock bekommen (da alle Java-Objekte eine implizite Speerr (=Lock) besitzen).

→ Während man auf den Lock wartet kann der Thread nicht unterbrochen werden.

```
public final class Counter {
    private int count = 0;
}
```

Codeskizze

Lock für das Objekt this

↳ führt zu:

Code ist äquivalent zur synchronisierten Methode

count	Main Thread	Thread T1	Thread T2
0	new Counter();		
1	starten von		
1		count++;	
2		return count;	count++;
2			return count;
3		count++;	
3		return count;	wartet auf den Lock
3			wartet auf den Lock
4		wartet auf den Lock	count++;
4			return count;
4		wartet auf den Lock	
4			wartet auf den Lock

T1: 1  
T2: 2  
T1: 3  
T2: 4  
...

Alles OK

→ man kann **synchronized** auch auf Klassenobjekten benutzen:

```
public final class SingleCounter {
    private static int globalCount = 0;
}
```

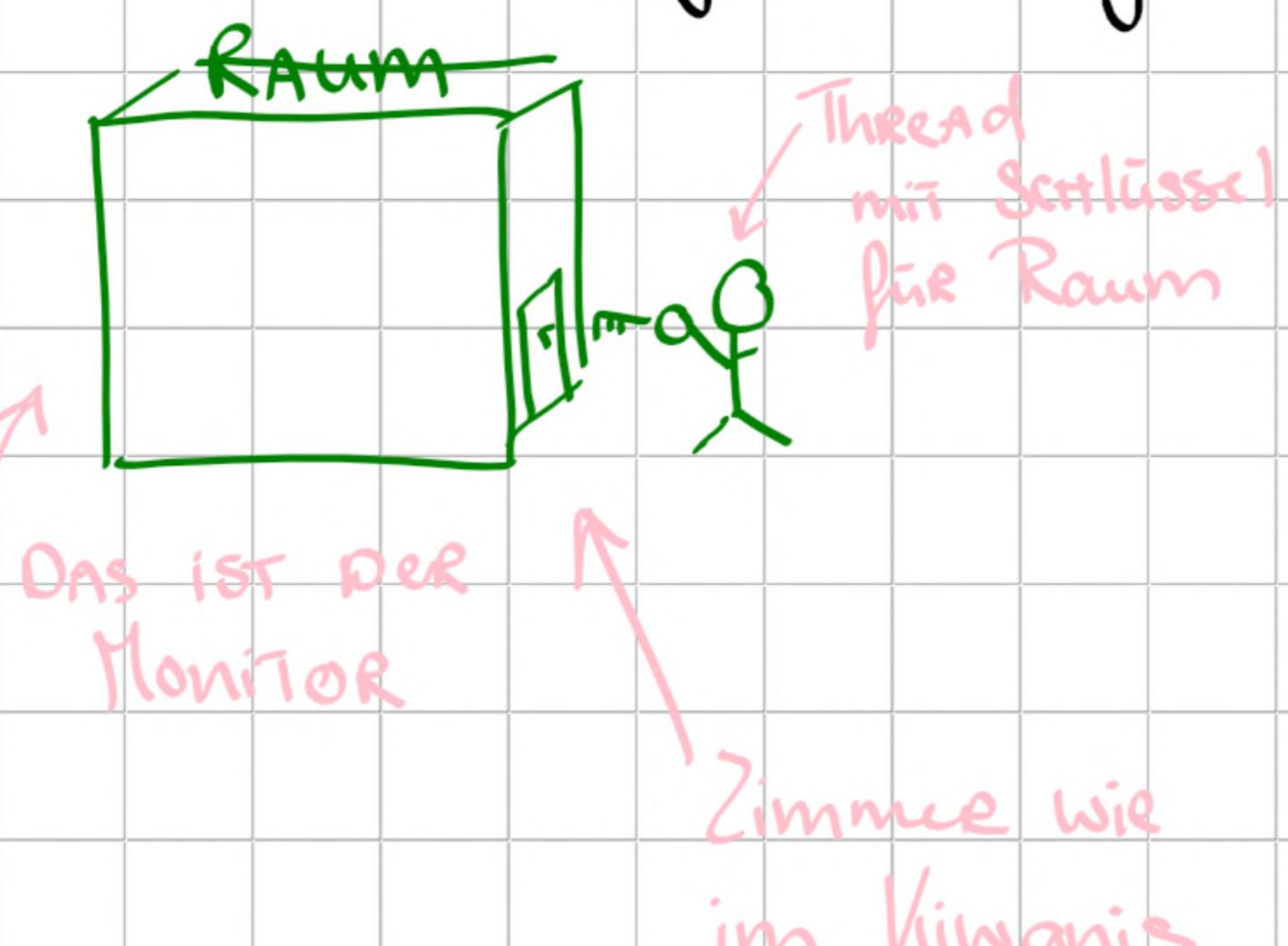
Codeskizze

Lock für das Objekt der Klasse SingleCounter

Code ist äquivalent zur synchronisierten Methode

- Monitor = programmiersprachliches Konzept zur Synchronisation von Zugriffen parallel laufender Prozesse oder Threads auf gemeinsam genutzten Ressourcen.

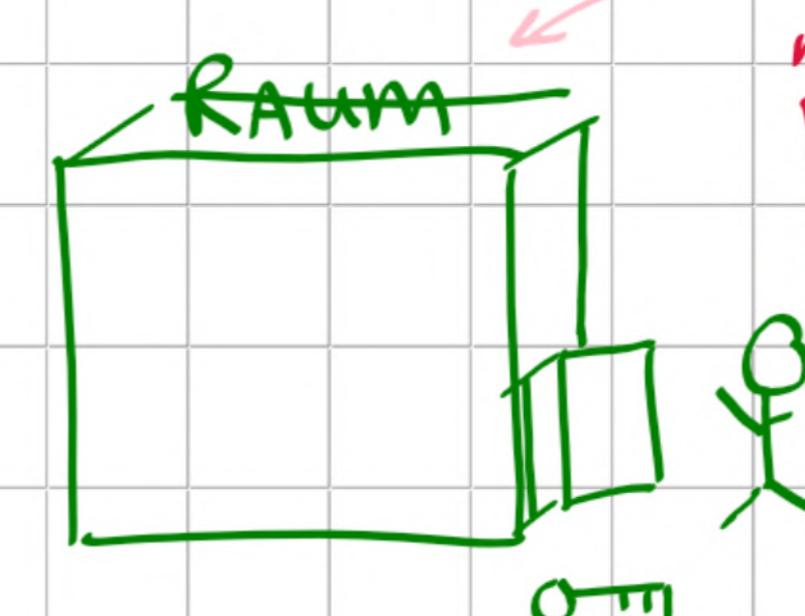
↳ Analogie



Nur 1 Thread kann im Raum sein & auf seine Ressourcen allein zugreifen.

↳ wenn er in den Raum kommt, schließt er Tür ab & behält Schlüssel.

↳ dadurch wird verhindert, dass andere Threads in den Raum kommen können.



Wenn Thread Raum verlässt, lässt er Tür offen. Schlüssel wird zurückgegeben.

Jetzt können andere Threads Raum betreten.

Monitor stellt sicher, dass nur 1 Thread den Raum betreten kann & verhindert so Konflikte & Race Conditions.

In Java:

→ wäre es eine spezielle Klasse

→ alle Daten dieser Klasse müssen private deklariert sein.

→ nur 1 Thread kann zu jedem Zeitpunkt im Monitor aktiv sein.

→ Spez. kann beliebige Anzahl von Bedingungen besitzen.

↳ bei Java kann man aber Attribute auch public machen & nicht alle Methoden müssen synchronized sein.

↳ führt zu unsicherem Umgang.

↳ deswegen kann man 2 Monitore verwenden. Via Reentrant Monitor oder Nested Monitor.

"wieder betreten"

Reentrant Monitor: Eine synchronisierte Methode ruft eine Andere auf, wobei beide synchronisierten Code Abschnitte denselben Object lock-Pool verwenden.

"geschachtelt"

Nested Monitor: Eine synchronisierte Methode ruft eine Andere auf, bei der beide synchronisierten Code Abschnitte verschiedene Object lock-Pools verwenden.

Wenn ein Thread eine synchronisierte Methode/Abschnitt besitzt, wo er den Lock section besitzt. Somit kann er sofort eintreten, ohne lock-Pool zu durchlaufen.

Ohne diese Option würde Rekursion nicht funktionieren.

[In der Programmierung: Ein Thread, der bereits den Monitor (Schlüssel) besitzt, kann den kritischen Bereich (Raum) erneut betreten, ohne dass andere Threads blockiert werden.]

Der Thread kann beliebig viele versch. locks bei sich haben.

↳ führen aber leicht zu Deadlocks, deshalb:

In einem synchronisierten Abschnitt nie eine öffentliche/gesetzte Methode aufrufen, die dazu da ist, überschrieben zu werden.  
d.h. beide Threads warten aufeinander & bleiben blockiert. Programm ist nicht abstürzt, nur für immer blockiert.

Deadlock

= Verklemmung

```
public final class ModuloCounter {
    private volatile int count = 0;
    private final int mod;

    public ModuloCounter(final int mod) {
        this.mod = mod;
    }

    public int getValue() {
        return count;
    }

    public synchronized void increment() {
        count = (count + 1) % mod;
    }

    public synchronized void decrement() {
        count = (count - 1 + mod) % mod;
    }
}
```

Ohne synchronized muss count mit volatile deklariert werden.

Beim Aufruf von getValue wird ein refresh des Caches des Aufrufers für count durchgeführt.

Atomarer, lesernder Zugriff auf count

```
public final class SimpleDeadlockDemo {
    private final Object lock1 = new Object();
    private final Object lock2 = new Object();

    public void methode1() {
        synchronized (lock1) {
            synchronized (lock2) {
                // macht was...
            }
        }
    }

    public void methode2() {
        synchronized (lock2) {
            synchronized (lock1) {
                // macht auch was...
            }
        }
    }
}
```

Codeskizze

Codeskizze

Ein Thread ruft nebenläufig methode1 auf.

Ein anderer Thread ruft nebenläufig methode2 auf.

Achtung: Nested Monitore haben immer Potential für Deadlocks.



Quelle: Hettel/Tran - Nebenläufige Programmierung mit Java

## Schwarzes Loch

→ Hier stellt die Klasse "BlackHole" zwei Methoden `put` & `get` zur Verfügung und `queue` ist die gemeinsame Ressource. 2 Threads werden erzeugt & gestartet. Der eine holt etwas aus dem Loch, der andere wirft etwas hinein.

<pre>public final class BlackHole {     private final BlockingQueue&lt;String&gt; queue;      public BlackHole() {         queue = new LinkedBlockingDeque&lt;&gt;();     }      public synchronized void put(final String thing) {         queue.add(thing);     }     In ein schwarzes Loch etwas     hineingeben...      public synchronized String get() throws InterruptedException {         return queue.take();     }     Aus einem schwarzen Loch etwas     herausnehmen... }</pre>	Codeskizze	<pre>public final class DemoBlackhole {     public static void main(final String[] args) {         final BlackHole blackhole = new BlackHole();         System.out.println("Thread starts...");         new Thread(() -&gt; {             try {                 LOG.info(blackhole.get());             } catch (InterruptedException ex) {                 LOG.debug(ex.getMessage());             }         }, "Blackhole 'getter' thread").start();         new Thread(() -&gt; {             blackhole.put("Sonne, Licht, irgendetwas...");         }, "Blackhole 'putter' thread").start();     } }</pre>	Codeskizze
--	------------	---	------------

Thread holt  
etwas heraus...

Thread gibt  
etwas hinein...

Deadlock

### Wie löst man das Problem?

- METHODEN `put` & `get` NICHT SYNCHRONISIEREN
- Die Klasse `BlockingQueue` IST THREAD-SAFE
- Wenn eine Klasse THREAD-SAFE IST, MUSS SIE DOKUMENTIERT SEIN.

→ ein fremde Methode, die aussenhalb des synchronisierten Bereichs aufgerufen wird, bezeichnet man als **offener Aufruf**.  
↳ verhindern Deadlocks  
↳ verbessern Nebentäufigkeit

↪ möglichst wenig Arbeit in synchronisierten Bereichen durchführen.

# 09.04.2024

→ wenn Zugang zu einem kritischen Abschnitt von bestimmten Bedingungen/Zuständen abhängt, dann reicht **synchronized** nicht aus.

Das Konzept **Warten** wird **Guarded Block** genannt.

= nebenläufiges Warten auf eine bestimmte Bedingung.

```
public synchronized void guardedJoy() {
    while (!joy) { ← Prüfen, ob Bedingung zutrifft...
        try {
            this.wait(); ← ...wenn nicht, warten!
        } catch (InterruptedException e) {
            /* handling... */
        }
    }
    System.out.println("Joy have been achieved!");
}
```

**Wichtig!**

Das Prüfen der Bedingung für die **wait** Methode ist immer mit einer **while** Anweisung auszuführen! Denn die Bedingung könnte sich nebenläufig geändert haben, wenn der Thread wieder aufgeweckt wird.

Siehe Beispiel «SimpleQueue»

→ **Wait** ist eine Methode von **Object**. Deswegen: Threads müssen den Lock des Objekts besitzen um "wait" auf diesem Objekt aufzurufen zu können.

→ Es führen 3 Wege aus dem Wait-Zustand heraus:

- 1) Andere Thread signalisiert Zustandswechsel via **notify** oder **notifyAll**
- 2) Im Argument angegebene Zeit (**timeout**) ist abgelaufen.
- 3) Andere Thread ruft Methode **interrupt** des wartenden Threads auf.

Nochmal: Beim Aufruf von **wait**-Methoden → Iteration verwenden, wenn Warten aufgrund einer Bedingung passiert!!

↪ Bedingungen immer wieder prüfen lassen, das ist wichtig.

→ Nach Aufruf von **wait()** muss ein anderer Thread die wartenden Threads mit **notify** benachrichtigen, dass der **guarded Block** nun frei ist.

```
public synchronized void notifyJoy() {
    joy = true; ← Bedingung setzen...
    this.notifyAll(); ← ...wartende Threads benachrichtigen
}
```

**WICHTIG:** Die Benachrichtigung der **notify/notifyAll** Methoden wird nicht gespeichert, das heißt wenn der pool leer ist und das **notify** gesendet wird, dann passiert einfach nichts. Aber wenn später ein Thread in den lock-pool kommt wird der nicht automatisch ausgeführt: er wartet auf ein **notify** und es kommt zu einem deadlock. Man kann nicht steuern welcher wait-pool tatsächlich aufgeweckt wird. Das Rezept ist, sobald man mehr als einen wait-pool hat, dann wird die **notifyAll** methode aufgerufen, dann wird die Bedingung wieder geprüft und der Thread der die Bedingung erfüllt wird dann ausgeführt.

1. Bedingung in Iteration ausführen
2. Sobald mehr als 2 Threads beteiligt sind, oder mehr als zwei wait-pools, muss ein **notifyAll** aufgerufen werden.

→ auch **notify** & **notifyAll** sind beides Objekte der Klasse **Object**. Threads müssen den Lock des Objekts besitzen um diese Methoden aufzurufen zu können.

→ **public notify void notify()**

↪ weckt genau 1 Thread im Wartezustand auf. Ist random welchen. Gerade bei mehreren Threads ein Problem, weil wenn eine Bedingung auf diesem geweckten Thread nicht passt, müssen alle warten => Deadlock!

→ **public final void notifyAll()**

↪ weckt alle Threads auf im wait-pool (& die einen Lock haben). Dann wird Bedingung gep

## → Regeln:

| notifyAll > notify |

↳ immer besser, kann aber Performance des Programms langsamer machen.

→ wenn notify verwendet wird: Besonders darauf achten, dass Lebendigkeit von Threads erhalten bleibt. Es muss genau der Richtige getroffen werden damit es funktioniert.

→ wait & notify/notifyAll können nicht außerhalb des synchronized Abschnitts aufgerufen werden.  
+ Objekt muss dasselbe sein!

## Implementation mit guarded blocks

Hier implementieren wir einen Ringbuffer, über den Produzenten & Konsumenten (Threads) Daten austauschen.  
Man möchte nicht, dass sich Produzenten & Konsumenten gegenseitig die Daten überschreiben.

- Ringbuffere wird durch Array von Objects realisiert.
- Variable "count" entspricht der aktuellen Anzahl Elemente.
- Variable "tail" zeigt auf den ersten freien Platz zum Einfügen.
- Variable "head" entspricht Index des nächsten verfügbaren Objekts.
- Wenn die Count die Länge des Arrays erreicht hat, wird ein wait() ausgeführt.

```
/*
 * Buffer (First In First Out) mit einer begrenzten Kapazität.
 */
public final class BoundedBuffer<T> {

    private final Object[] data;
    private int head;
    private int tail;
    private int count;
    //...
```

Erweiterung – Daten speichern oder nicht...

Ein Thread kann mit wait(long timeout) wieder aktiv werden, wenn das Timeout (in Millisekunden) abgelaufen ist:

```
public synchronized boolean offer(final T item, final long millis)
    throws InterruptedException {
    while (count == data.length) { ← Warten bis der Thread geweckt wird, oder das Timeout eintritt.
        this.wait(millis);
        if (count == data.length) { ← Falls das Array immer noch voll ist, war's das Timeout - false
            return false;
        }
    }
    count++;
    data[tail] = item;
    tail = (tail + 1) % data.length;
    if (count == 1) {
        this.notifyAll();
    }
    return true; ← Wenn das Element mit Erfolg gespeichert werden konnte - true
}
```

Datenelement speichern:

```
/*
 * Ein Element T speichern. Falls der Buffer voll ist, warten bis
 * ein Platz frei wird.
 * @param item zu speicherndes Element.
 * @throws InterruptedException wenn das Warten unterbrochen wird.
 */
public synchronized void add(final T item)
    throws InterruptedException {
    while (count == data.length) { ← Falls die Anzahl gespeicherten Elemente die Länge des Array erreicht haben - warten.
        this.wait();
    }
    count++;
    data[tail] = item; ← Element speichern und Attribut tail nachführen.
    tail = (tail + 1) % data.length;
    if (count == 1) {
        this.notifyAll(); ← Wenn ein Element ins leere Array gespeichert wurde - wartende Thread benachrichtigen.
    }
}
```

Warum eine Selektion für die Benachrichtigung? Diese deckt die Randbedingung ab (Queue leer bzw. voll)

Wenn ein Element ins leere Array gespeichert wurde, werden mit notifyAll() die wartenden Threads benachrichtigt.

Die add und remove methoden geben die interruptedExceptions weiter, was an sich eine gute Idee ist. Es ist besser die Exceptions dort aufzufangen wo sie aufgerufen werden

Datenelement lesen:

```
/*
 * Ein Element T auslesen. Falls der Buffer leer ist, warten bis
 * ein Platz belegt ist.
 * @return ausgelesenes Element.
 * @throws InterruptedException wenn das Warten unterbrochen wird.
 */
public synchronized T remove() throws InterruptedException {
    while (count == 0) { ← Falls keine Elemente gespeichert sind - warten.
        this.wait();
    }
    count--;
    T item = (T) data[head]; ← Element lesen, Array Feld auf null setzen und Attribut tail nachführen.
    data[head] = null;
    head = (head + 1) % data.length;
    if (count == data.length - 1) {
        this.notifyAll(); ← Wenn ein Element aus dem vollen Array gelesen wurde - wartende Thread benachrichtigen.
    }
    return item;
}
```

## Verlorene Signale

Ein Thread addiert Zahlen & ein Thread wartet auf das Resultat und gibt es aus.

```
private static final Object LOCK = new Object();
private static long sum = 0;

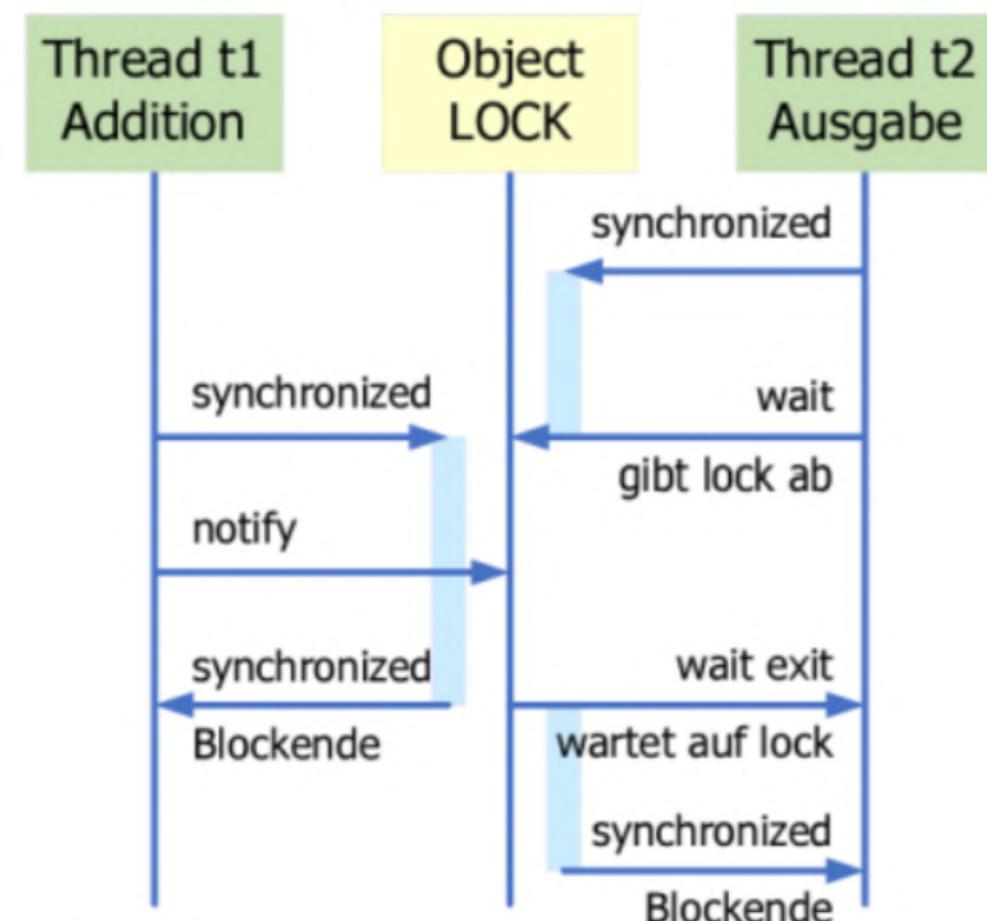
//...

Thread t1 = new Thread(() -> {
    for (int i = 1; i <= 10_000; i++) {
        sum += i;
    }
    LOG.info("calc finished, notifying...");
    synchronized (LOCK) {
        LOCK.notify(); ← Steuerung wartender Threads durch notify.
    }
});
t1.start();
```

```
Thread t2 = new Thread(() -> {
    LOG.info("wait for result...");
    synchronized (LOCK) {
        try {
            LOCK.wait(); ← Thread wartet...
        } catch (InterruptedException e) {
            return;
        }
    }
    LOG.info("sum = " + sum);
});
```

Deadlock Gefahr  
Warum?

→ So sollte dieses Beispiel vorhin funktionieren:



Aber manchmal passiert auch das...

deshalb zusammenfassend:

- 1) Der Aufruf von `notify` & `notifyAll` wird nicht gespeichert.
- 2) Wenn kein Thread wartet, geht das `notify` Signal verloren.
- 3) Es muss eine zeitliche Abfolge zwischen `wait` & `notify` herrschen.
  - ↳ der Aufruf "wait" muss vor dem "notify" erfolgen.
  - ↳ falls keine chronologische Abfolge: Deadlock-Gefahr!

Lösung: Aufrufe (Signale) von "wait" & "notify" müssen gespeichert werden.

Semaphor ↗ Lösung

Implementation eines nach oben nicht begrenzten Semaphors, d.h. Zähler kann unendlich wachsen.

```

public final class Semaphore {
    private int sema; // Semaphor Zähler
    public Semaphore(final int init) {
        sema = init;
    }
    public synchronized void acquire() throws InterruptedException {
        while (sema == 0) {
            this.wait(); ← P Operation: Falls der Zähler den Zustand 0 hat - Thread wartet.
        }
        sema--;
    }
    public synchronized void release() {
        sema++; ← V Operation: Threads werden geweckt und Zähler um 1 erhöht.
        this.notifyAll();
    }
}
  
```

Codeskizze

→ dabei ist egal, ob `acquire` oder `release` zuerst aufgerufen wurde

→ Semaphore verhindern das Verlieren von Signalen, jedoch nur ein Zähler

→ Verwaltet keine gemeinsame Ressourcen.

→ Beachte: schwache & starke Semaphore

Bounded Buffer X Semaphor

Für Implementation braucht es:

- Thread sichere Queue/List für Speicherung der Datenelemente
- Semaphor für Zugriffskontrolle.

```

public final class BoundedBuffer<T> {
    private final ArrayDeque<T> queue;
    private final Semaphore putSema;
    private final Semaphore takeSema;

    public BoundedBuffer(final int n) {
        queue = new ArrayDeque<T>(n);
        putSema = new Semaphore(n);
        takeSema = new Semaphore(0); ← Kapazität des Puffers, bzw. der Queue
    } ← Anfangszustand - leerer Puffer
    ...
  
```

Codeskizze

besitzt einen Pool & garantiert nicht die chronologische, richtige Abarbeitung der Warteschlange.

besitzt FIFO Queue für das Warten, damit first come, first served - Prinzip garantiert ist.

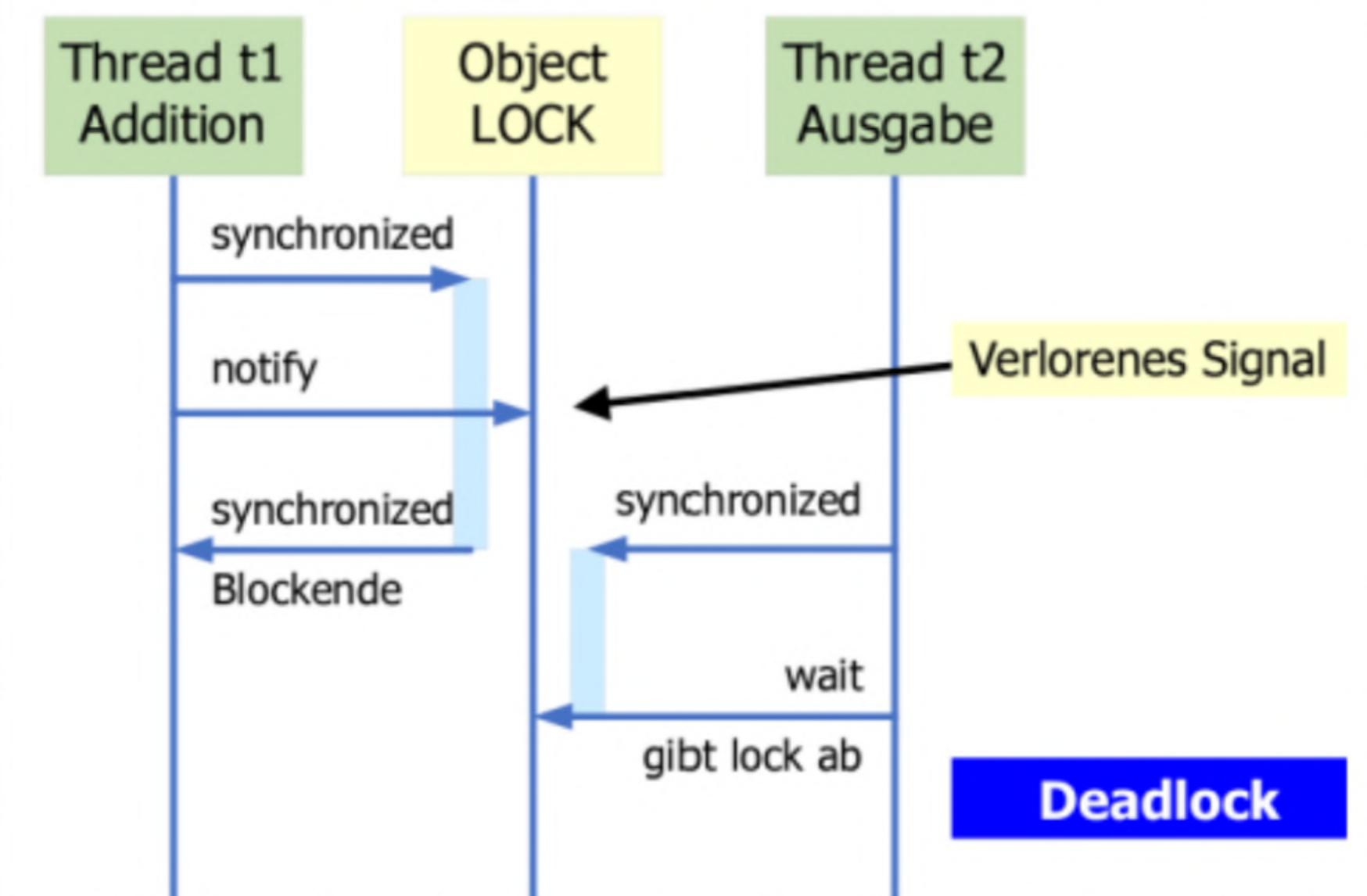
Beim lesen und schreiben gilt das FIFO Prinzip:

```

public void add(final T item) throws InterruptedException {
    putSema.acquire(); ← Warten, falls Queue voll ist.
    synchronized (queue) {
        queue.addFirst(item); ← Element am Anfang der Queue einfügen.
    }
    takeSema.release(); ← Benachrichtigung: Ein Element kann entnommen werden.
}

public T remove() throws InterruptedException {
    takeSema.acquire(); ← Warten, falls Queue leer ist
    T item;
    synchronized (queue) {
        item = queue.removeLast(); ← Element Am Ende der Queue entnehmen.
    }
    putSema.release(); ← Benachrichtigung: Ein Element kann entfernt werden.
    return item;
}
  
```

Codeskizze



Deadlock

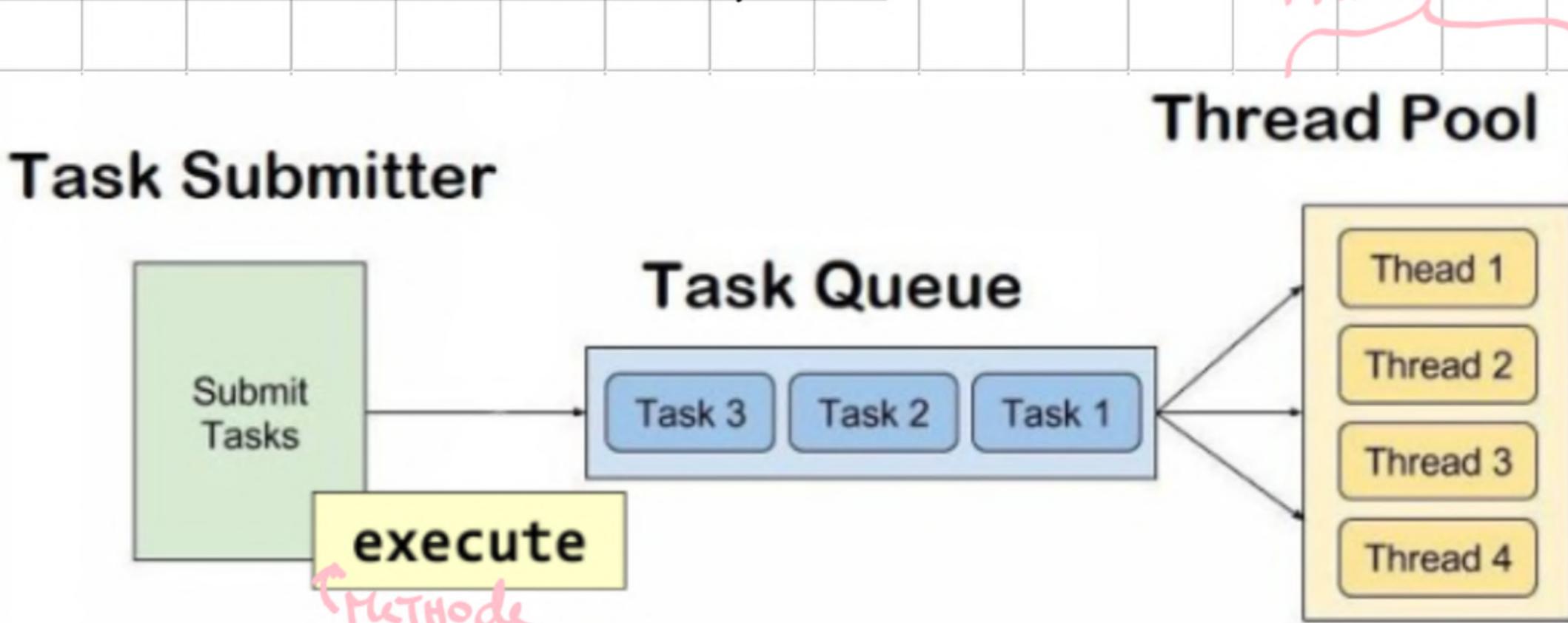
# 09.04.2024

*zur nebenläufigen Ausführung eines Task ist immer ein Thread notwendig*

## Anmerkungen zu nebenläufigen Aufgaben

- die meisten Aufgaben, die nebenläufig ausgeführt werden sollen, sind von kurzer Dauer & treten unregelmäßig auf.
- für jede neue Aufgabe einen Thread erzeugen belastet das Betriebssystem
- Große Anzahl Threads wirkt sich auch negativ auf OS aus  $\Rightarrow$  Sinnvolle Menge der erzeugten Threads beschränken
- Grundsätzlich wird schon mit rudimentären Thread Objekten gearbeitet. Mehr Threadpools
- Beim Erzeugen eines Thread muss ein Objekt vom Typ **Runnable** dem Thread-Konstruktor übergeben werden.
  - ↪ es gibt keine Getter- & Setter
- Nach dem Aufruf von **START** → Thread beginnt mit Abarbeitung der Aufgabe von Typ **Runnable**
- Aufruf von **START** auf Thread-Objekt ist nicht 2x möglich: **IllegalThreadStateException**

## Thread Pool Konzept



- 1) Bei Erzeugung des Pools wird bestimmt wie gross Pool ist (Anzahl Threads)
- 2) Tasks (Objekte des Typs Runnable) werden über **execute** in einer Queue gespeichert (enqueue)
- 3) Worker Thread entnimmt Task aus Task Queue (dequeue) & führt **Run**-Methode aus.

→ es gibt einen "Task-Submitter" der Tasks erzeugt, diese Tasks werden mit **execute** in eine **Task Queue** ausgeladen. Am anderen Ende warten Threads, die in einem Pool warten: "gibt es einen Task für mich?" Dann schnappt er sich ersten Task in Queue & führt ihn aus, holt dann den nächsten.

↪ Vergleichbar mit Post:

Task-Submitter: Paket vorbereiten

Task-Queue: Laufband für Paket

Thread-Pool: Postler

*↑ man kann nicht eigene Mitarbeiter eingestellen, d.h. limitiert*

## Implementierung Thread Pool

- Der Executor stellt ein Interface dar & enthält die Methode **execute** mit einem Parameter vom Typ **Runnable**. Die Schnittstelle (ähnlich zu Threads) ist im Package **java.util.concurrent**.

```

public interface Executor {
    /**
     * Führt die gegebene Aufgabe irgendwann in der Zukunft aus.
     * @param command ausführbare Aufgabe.
     */
    void execute(Runnable command);
}

```

## Erzeugen von Threads

VIA Klasse "PlainThreadPool" implementiert einen Executor. Klasse erzeugt eine festgelegte Anzahl Threads, welche Tasks aus Queue holen & ausführen.

```

public final class PlainThreadPool implements Executor {

    private final BoundedBuffer<Runnable> workQueue;
    private final int nWorkers;

    public PlainThreadPool(final int capacity, final int nWorkers) {
        this.workQueue = new BoundedBuffer<>(capacity);
        this.nWorkers = nWorkers;
        for (int i = 0; i < this.nWorkers; ++i) {
            activate();
        }
    }

    private void activate() {
        final Runnable runLoop = () -> {
            try {
                while (true) {
                    final Runnable r = workQueue.remove();
                    r.run();
                }
            } catch (InterruptedException e) {
                LOG.debug(e);
            }
        };
        final Thread thread = new Thread(runLoop);
        thread.setDaemon(true);
        thread.start();
    }
}

```

Codeskizze

Größe der Queue      Anzahl Worker Threads

Die Methode activate erzeugt und startet einen Worker Thread:

```

private void activate() {
    final Runnable runLoop = () -> {
        try {
            while (true) {
                final Runnable r = workQueue.remove();
                r.run();
            }
        } catch (InterruptedException e) {
            LOG.debug(e);
        }
    };
    final Thread thread = new Thread(runLoop);
    thread.setDaemon(true);
    thread.start();
}

Codeskizze

```

Aufgabe aus der Work Queue holen.  
Aufgabe ausführen.  
Thread terminiert, wenn der main-Thread beendet wird.

## Aufgabe entgegen nehmen

Die Methode execute nimmt eine Aufgabe (vom Typ **Runnable**) entgegen & speichert diese in die Work Queue (vom Typ **BoundedBuffer**) ab. Ist blockierend, wenn Kapazität der Queue voll ist.

```

@Override
public void execute(Runnable command) {
    try {
        workQueue.add(command);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}

```

Codeskizze

## Threadpools

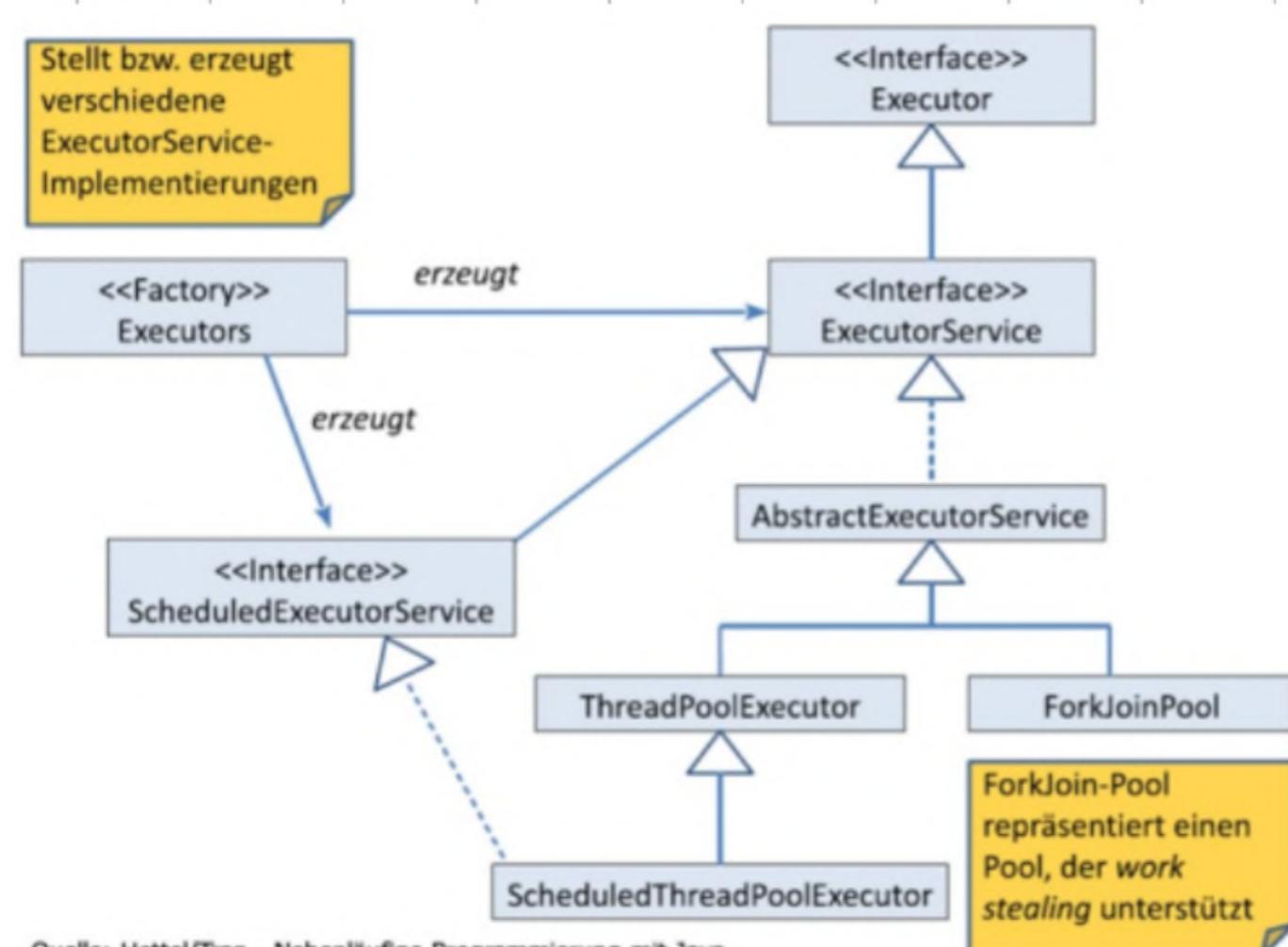
Man kann eigene machen, muss man aber nicht. Es gibt diverse verschiedene Threadpools, bspw.

-> ThreadPoolExecutor

-> ScheduledThreadPoolExecutor

-> ForkJoinPool

So sieht Klassenhierarchie aus:



Factory-Methods der Executors Klasse (nur eine Auswahl)

- newCachedThreadPool()
  - o Diese Fabrikmethode liefert einen Pool wo bei Bedarf neue Worker Threads erzeugt werden.
  - o Unbenutzte Threads bleiben für 60 Sekunden erhalten.
- newFixedThreadPool(int nThreads)
  - o Diese Fabrikmethode liefert einen Pool mit nThreads Threads.
  - o Die Warteschlange für übergebene Aufgaben ist unbeschränkt.
  - o Stirbt ein Thread aufgrund eines Fehlers, wird er durch einen neuen ersetzt.
- newScheduledThreadPool(int corePoolSize)
  - o Diese Fabrikmethode liefert einen ScheduledExecutorService.
  - o Damit werden Aufgaben nach einer gegebenen Verzögerung bzw. periodisch ausgeführt.

## Beispiel Threadpool mit fixer Anzahl Worker

Aufgabe: "Gib 100 Buchstaben auf das Terminal aus.

```
final ExecutorService executor =  
    Executors.newFixedThreadPool(nWorker);  
for (int nTask = 1; nTask <= 4; nTask++) {  
    final char ch = (char) (64 + nTask);  
    executor.execute(() -> {  
        LOG.info("{} starts {}", Thread.currentThread().getName(), ch);  
        for (int i = 0; i < 200; i++) {  
            System.out.print(ch);  
        }  
        System.out.println("");  
        LOG.info("{} finished {}", Thread.currentThread().getName(), ch);  
    });  
}  
TimeUnit.MILLISECONDS.sleep(100);  
LOG.info("shutdown");  
executor.shutdown();
```

Anzahl Worker Thread  
Anzahl Aufgaben  
Aufgabe übergeben  
Thread Pool beenden

# Thread sterben lassen:

```
System.out.print(ch);
if (ch == 'B' & i == 100) {
    Thread.currentThread().stop();
}
System.out.println(" ");
```

## Output:

# Grundsätzliche Empfehlung

- a) Einfache, wenige, lang laufende nebentäufige Threads. Können mit "Thread" & "Runnable" realisiert werden.
  - b) Sobald mit vielen, kurzen, häufigen Threads gerechnet werden muss → Konzept **Executor-Services** nutzen!
  - c) Voneinander unabhängige Aufgaben (fire & forget - Prinzip) sind völlig problemlos.
  - d) Bei Abhängigkeiten (gemeinsame Ressourcen bspw.) ist Synchronisation notwendig! Ansonsten drohen Inkonsistenzen & Fehler.

# WAS PASSIERT, WENN RUN

- 1) Sequenzielle Abarbeitung nWorker = 1;  
    ↳ nach Shutdown werden bereits zugewiesene Aufgaben abgearbeitet. Neue Aufgaben werden mit **RejectedExecutionException** abgewiesen.

↪ shutdown ist kein blockierender Aufruf

## Output:

```
2022-10-18 09:45:42,902 INFO - pool-2-thread-1 starts A
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
2022-10-18 09:45:42,905 INFO - pool-2-thread-1 finished A
2022-10-18 09:45:42,905 INFO - pool-2-thread-1 starts B
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Nur ein Thread am Werken.
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
2022-10-18 09:45:42,906 INFO - pool-2-thread-1 finished B
2022-10-18 09:45:42,907 INFO - pool-2-thread-1 starts C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
2022-10-18 09:45:42,908 INFO - pool-2-thread-1 finished C
2022-10-18 09:45:42,999 INFO - shutdown
```

Dieselbe Demo aber mit zwei Threads:

- Zu beachten ist, dass echt-parallel so viele Threads laufen wie Prozessorkerne oder Threading-Pipes zur Verfügung stehen.

```
2022-10-18 09:53:20,732 INFO - pool-2-thread-1 starts A
2022-10-18 09:53:20,732 INFO - pool-2-thread-2 starts B
AAAAAAAAABBBAAAAAAAAAAABAAAAAAAABAAAAAAAAMAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAA
AAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBAAAAAAABBBBBBBBBBBBBBBBBBAA
AABBBBBABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAABBBBABB BBBB
2022-10-18 09:53:20,737 INFO - pool-2-thread-1 finished A
2022-10-18 09:53:20,737 INFO - pool-2-thread-1 starts C
BBBBBBBCCCCCCCCCCCCCBBBBBCCBBBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
2022-10-18 09:53:20,738 INFO - pool-2-thread-2 finished B
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
2022-10-18 09:53:20,739 INFO - pool-2-thread-1 finished C
2022-10-18 09:53:20.830 INFO - shutdown
```

# SEMESTERWOCHE 7

## Future & Callable - Interfaces

- Typischerweise besitzt ein Runnable Interface kein Rückgabewert (& implementiert eine nebenläufig auszuführende Aufgabe)  
 ↳ wenn Aufgabe Wert zurückliefern muss, geht das nur über spezielles Rückgabe-Attribut

```
public final class RunnableWithReturn<T> implements Runnable {
    private T returnValue;
    private volatile Thread self;
    // ...
    @Override
    public void run() {
        self = Thread.currentThread();
        // ...
    }
    public T get() throws InterruptedException {
        self.join();
        return returnValue;
    }
}
```

Codeskizze

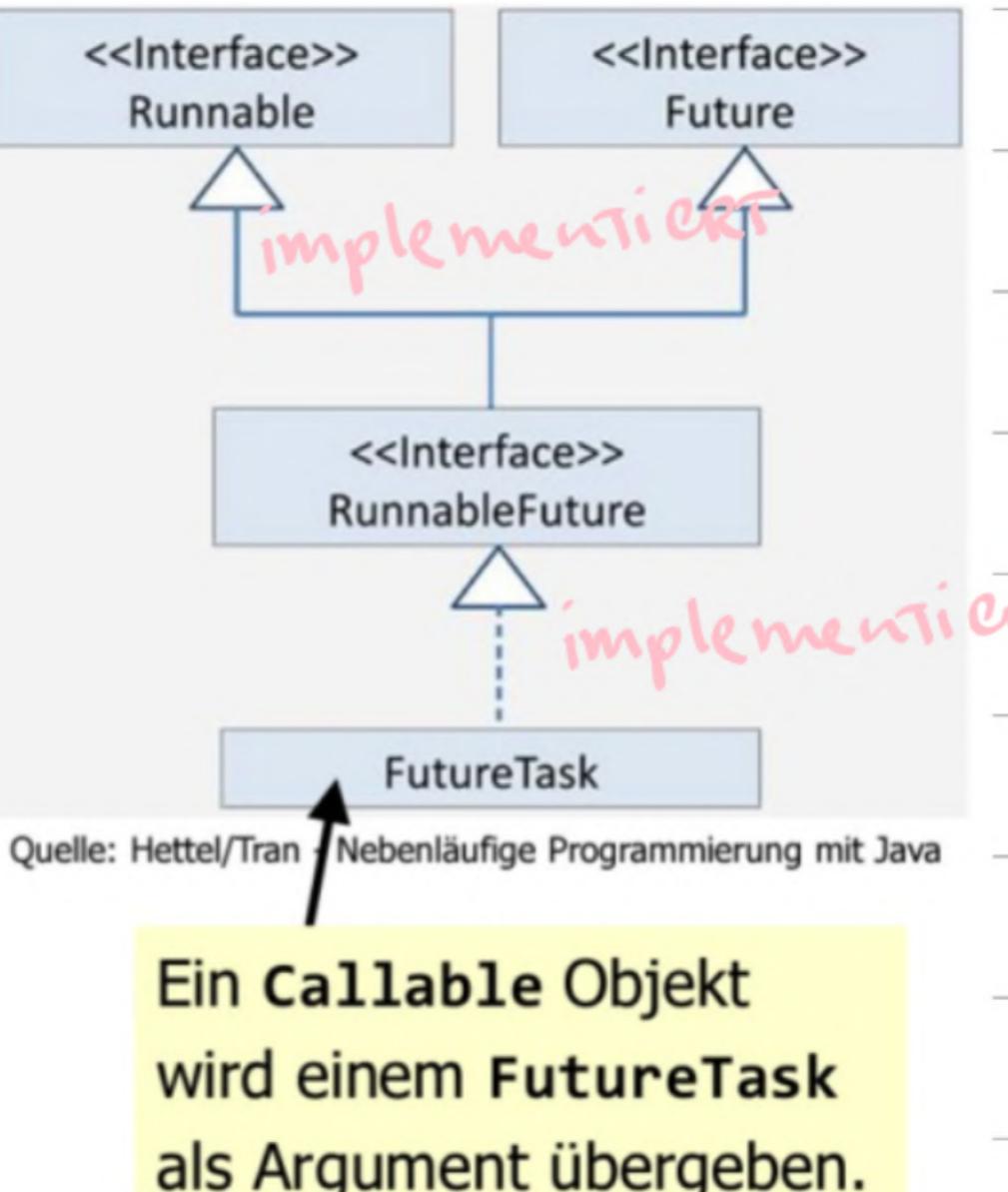
Attribut für Rückgabewert.

Berechnung und Speicherung des Resultats ins Rückgabe Attribut.

Blockierende Abfrage des Rückgabewertes.

- DAS neue Runnable heißt Callable → kennt nur 1 Methode: call().

↳ Kann nun einen Rückgabewert haben. → per Typparameter festlegen.



```
/**
 * A task that returns a result and may throw an exception.
 */
public interface Callable<V> {
    /**
     * Computes a result, or throws an exception if unable
     * to do so.
     * @return computed result
     * @throws Exception if unable to compute a result
     */
    V call() throws Exception;
}
```

Quelle: java.util.concurrent Library

- Mit Interface "Future" kann die Ergebnisrückgabe einer asynchronen Berechnung einheitlich & einfach gemacht werden.  
 → neben Ergebnis kann auch Status der Berechnung abgefragt werden.  
 → FutureTask implementiert das Future Interface + Runnable Interface

## Callables & Executors

- 1) Ein Callable wird dem ExecutorService über Methode submit() zum ausführen übergeben.
- 2) Die Methode liefert ein FutureTask-Objekt zurück.
- 3) Über FutureTask-Objekt kann Rückgabe erfragt werden.

```
final Callable<Integer> sumTask = () -> {
    int sum = 0;
    for (int i = 1; i <= 10000; i++) {
        sum += i;
    }
    Callable
    return sum; ← Rückgabewert.
};
final ExecutorService executor = Executors.newCachedThreadPool();

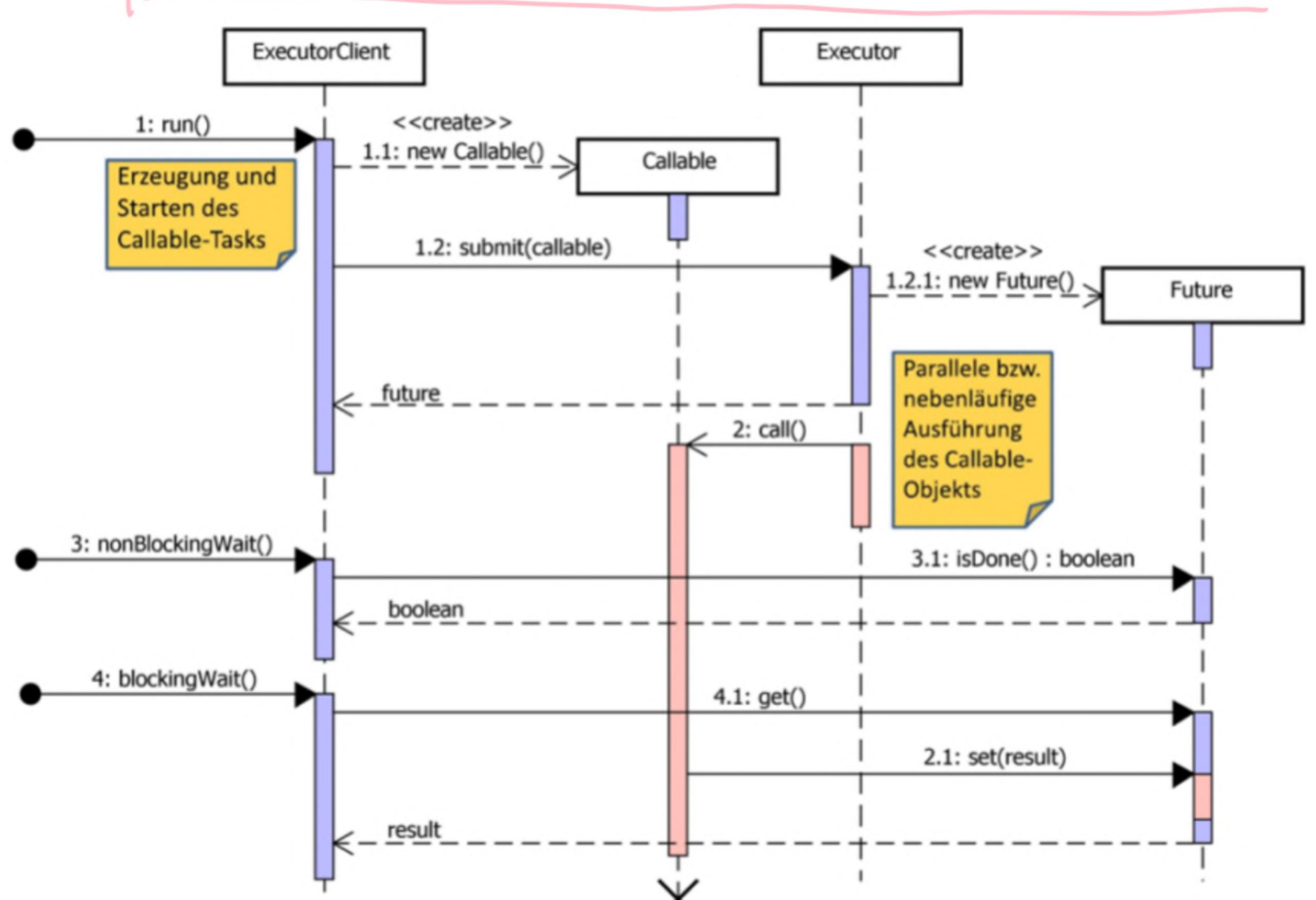
final Future<Integer> future = executor.submit(sumTask);
LOG.info("Summe: " + future.get());
```

Codeskizze

Executor erzeugen und Callable Objekt als Task aufgeben.

Blockierende Abfrage des Rückgabewertes via FutureTask Objekt.

## Funktionsweise Future-Patterns:



## FUTURE - PATTERN

- IST ein Entwurfsmuster, das in der asynchronen Programmierung verwendet wird, um das Ergebnis einer asynchronen Berechnung darzustellen, bevor es TATSÄCHLICH verfügbar ist. Ermöglicht Darstellung von Ergebnissen, wenn sie benötigt werden (anstatt warten bis auf Ende der Berechnung)

## Tasks an einen ExecutorService übergeben:

### - Future<?> submit (Runnable Task)

- ↳ Das zurückgegebene Future-Objekt wird verwendet um `isDone`, `cancel` und `isCancelled` aufzurufen
- ↳ `get` liefert bei Fertigstellung "null"

### - Future<T> submit (Runnable Task, T result)

- ↳ hier liefert `get` des Future-Objekts das vorgegebene Result-Objekt als Ergebnis zurück.

### - Future<T> submit (Callable<T> Task)

- ↳ Future-Objekt zurückgeliefert mit dem Ergebnis der Berechnung abgeholt werden kann.

## Future<T>

→ bietet auch `get(long timeout, TimeUnit unit)` an → Aufrüfer kann maximale Wartezeit angeben.  
 ↳ Wenn Ergebnis nach vorgegebener Zeit nicht verfügbar, wird `TimeoutException` geworfen.

→ mit `isDone` kann Bearbeitungsstatus abgefragt werden.

→ mit `cancel(boolean mayInterruptIfRunning)` kann man abbrechen.

↳ Wenn Task noch nicht gestartet ist, wird er nicht ausgeführt.

↳ Wenn Task mittan in Abarbeitung, kann ein Interrupt gesendet werden.

↳ Tasks müssen so implementiert sein, dass sie ein Interrupt berücksichtigen!

→ mit `isCancelled` kann man prüfen, ob Task abgesprochen wurde.

## Beispiel - ARRAY sortieren mit Callable

Das nebenläufige Sortieren soll ein Callable übernehmen, als Resultat wird das sortierte Array zurückgegeben.

```
public final class ArraySortTask implements Callable<byte[]> {
    private final byte[] array;

    public ArraySortTask(byte[] array) {
        this.array = Arrays.copyOf(array, array.length);
    }

    @Override
    public byte[] call() {
        Arrays.sort(array);
        return array;
    }
}
```

Rückgabe des asynchron, d.h.  
nebenläufig, ausgeführten  
Tasks ist ein byte-Array.

## Beispiel - ARRAY-Sortier-Task aufgeben

Das Programmmodell sieht folgendermassen aus:

1. Arbeit an den ExecutorService übergeben (Fragen was jemand für eine Pizza haben will)
2. Etwas anderes machen (Schonmal alles bereitlegen)
3. Das Resultat abholen (Resultat abholen und gewünschte Pizza machen)

```
final byte[] array = new byte[16];
new Random().nextBytes(array);
```

byte Array erzeugen und  
mit Zufallszahlen füllen,  
Callable-Objekt erzeugen.

```
final Callable<byte[]> task = new ArraySortTask(array);
final ExecutorService executor = Executors.newCachedThreadPool();
```

1. `final Future<byte[]> result = executor.submit(task);`
2. //...hier etwas anderes machen
3. `final byte[] sortedArray = result.get();`

Codeskizze

```
LOG.info("lowest value = " + sortedArray[0]);
LOG.info("highest value = " + sortedArray[sortedArray.length-1]);
```

## Wie geht man mit Fehlern aus nebenläufig ausgeführten Tasks um?

Bsp: Task mit Division durch Null wird mit execute an einen Pool gesendet.

```
final ExecutorService executor = Executors.newCachedThreadPool();
executor.execute(() -> System.out.println(1 / 0));
```

die Exception wird aber nicht beim "submit" geworfen, sondern bei "get"

Sobald Task ausgeführt wird → Exception geworfen:

```
Exception in thread "pool-1-thread-1"
java.lang.ArithmetricException: / by zero
...
```

```
final ExecutorService executor = Executors.newCachedThreadPool();
final Future<?> future =
    executor.submit(() -> System.out.println(1 / 0));
try {
    future.get();
} catch (InterruptedException | ExecutionException ex) {
    LOG.debug(ex);
}
```

Codeskizze

```
2023-03-25 09:45:42,908 DEBUG - java.util.concurrent.ExecutionException:
java.lang.ArithmetricException: / by zero
```

Damit man über die Ausnahme in Kenntnis gesetzt wird, sollte die Exception weitergegeben werden.

```
final ExecutorService executor = Executors.newCachedThreadPool();
final Future<?> future = executor.submit(() -> {
    try {
        System.out.println(1 / 0);
    } catch (Exception ex) {
        LOG.debug(ex);
        throw ex;
    }
});
try {
    future.get();
} catch (InterruptedException | ExecutionException ex) {
    LOG.debug(ex);
}
```

Codeskizze

## Atomarer Zugriff mit atomic Variablen

→ nur mit 32-bit Variablen eines primitiven Datentyps. + long + double

↪ der lesende & schreibende Zugriff darauf ist atomar (= nicht unterbrechbar)

einzige beide Ausnahmen aus 64-bit Typen

→ Zugriffe auf Referenzvariablen immer atomar, egal wieviel bit.

→ Oft besteht eine Operation aus mehreren Schritten, obwohl im Code dafür nur 1 Anweisung angegeben ist.

↪ bsp. an counter++

- ↪ 1) Laden des Inhalts (Lesen)
- 2) Inkrementieren des Inhalts.
- 3) Speichern des neuen Ergebnis (Schreiben)

## Wie kann ich eine Variable atomic lesen und verändern?

→ zentral ist die Compare-and-Set Operation. Sie benötigt hierzu:

- 1) Eine Speicherstelle
- 2) Den erwarteten, alten Wert
- 3) einen neuen Wert

signalisiert, ob Änderung stattgefunden hat.

angewandt auf den Wert des Objekts

↪ boolean compareAndSet (int expect, int update)

↪ wenn der Inhalt der Speicherzelle mit erwartetem, alten Wert übereinstimmt ⇒ neuer Wert an Speicherstelle geschrieben

↪ wenn der Inhalt der Speicherzelle mit erwartetem, alten Wert nicht übereinstimmt (weil z.B. zwischenzeitlich geändert) ⇒ findet keine Modifikation statt.

→ weitere Methoden von AtomicInteger:

↪ int addAndGet (int delta)

↪ Der Wert wird atomic um delta erhöht. Der neue Wert wird zurückgegeben.

## → int decrementAndGet()

↪ Der Inhalt wird atomar dekrementiert & der neue Wert wird zurückgegeben.

## → int incrementAndGet()

↪ Der Inhalt wird atomar inkrementiert & der neue Wert wird zurückgegeben.

## → int get(int newValue)

↪ Der Wert wird durch newValue ersetzt & der neue Wert wird zurückgegeben.

## → int get()

↪ liefert den aktuellen Wert.

→ alle Getter-Methoden gibt es auch in der getAnd-Version. (d.h. der Wert, welcher vor Operation war)

## Beispiel atomarer Zähler:

- Thread-sicherer Zähler, ohne das aufwendige synchronized und somit ein Performance Gewinn

```
public final class AtomicCounter {  
    Codeskizze  
  
    private final AtomicInteger counter = new AtomicInteger(0);  
  
    public void increment() {  
        counter.incrementAndGet();  
    }  
    public void decrement() {  
        counter.decrementAndGet();  
    }  
    public int get() {  
        return counter.get();  
    }  
}
```

Hier können auch getAnd-Methoden verwendet werden, weil der Rückgabewert nicht gelesen wird.

## Komplexere Änderung

- Wir wollen einen Maximalwert in einer AtomicLong Variablen speichern, der von verschiedenen Threads durch Aufruf einer update Methode geändert werden kann.

```
private static final AtomicLong VALUE = new AtomicLong();  
  
public static void update(final long newVal) {  
    long alt, neu;  
    do {  
        alt = VALUE.get();  
        neu = Math.max(alt, newVal);  
    } while (VALUE.compareAndSet(alt, neu) == false);  
}
```

update muss aus mehreren Schritten bestehen, um atomare Zugriffe zu garantieren.

## Änderung ab Java 8

- Um Iterationen zu vermeiden, stehen ab Java 8 in den Atomic Klassen die Methoden accumulateAndGet und getAndAccumulate sowie updateAndGet und getAndUpdate zur Verfügung.

```
private static final AtomicLong VALUE = new AtomicLong();  
  
public static void update(final long newVal) {  
    VALUE.accumulateAndGet(newVal, Math::max);  
} newVal wird in VALUE gespeichert, wenn Math::max wahr ist.  
  
mult gibt den alten Wert von VALUE zurück.  
public static long mult(final long operand) {  
    return VALUE.getAndUpdate((long a) -> operand * a);  
}  
operand wird mit dem Parameter a multipliziert und in VALUE gespeichert.  
Beim Aufruf ist VALUE das Argument von a.
```

Codeskizze

## Thread-sichere Container:

→ JAVA BEZIEHT SEIT SEINER ERSTEN VERSION DIE STANDARDISIERTEN CONTAINER **VECTOR, STACK, HASHTABLE, DICTIONARY**.

→ FÜR JEDES COLLECTION INTERFACE STEHT EINE ÖFFENTLICHE KLASSENMETHODE VON COLLECTIONS ZUR VERFÜGUNG, DIE EINE SYNCHRONISIERTE CONTAINER-FASSADE ZURÜCKLIEFERT (WEIL DAS COLLECTION FRAMEWORK KEINE SYNCHRONISIERTE WRAPPER-KLASSEN HAT.)

## Bsp: Synchronisierte Containerfassaden

```
List<BankAccount> list = new ArrayList<>();
List<BankAccount> syncList = Collections.synchronizedList(list);

Map<String, String> map = new HashMap<>();
Map<String, String> syncMap = Collections.synchronizedMap(map);
```

Codeskizze

- ↪ Konzept ist **nur geeignet für OS, auf die wenig darauf zugegriffen wird** (gelesen/geschrieben/etc.)
- ↪ Probleme:
  - alle Zugriffe sind nur mit einem Monitor geschützt (Synchronisations-Empässe)
  - Reines, paralleles Lesen nicht möglich

### Wirklich Thread-sichere Iteration

- Um Thread-Sicherheit wirklich zu gewährleisten, muss vor der Iteration der Container selbst geschützt werden.

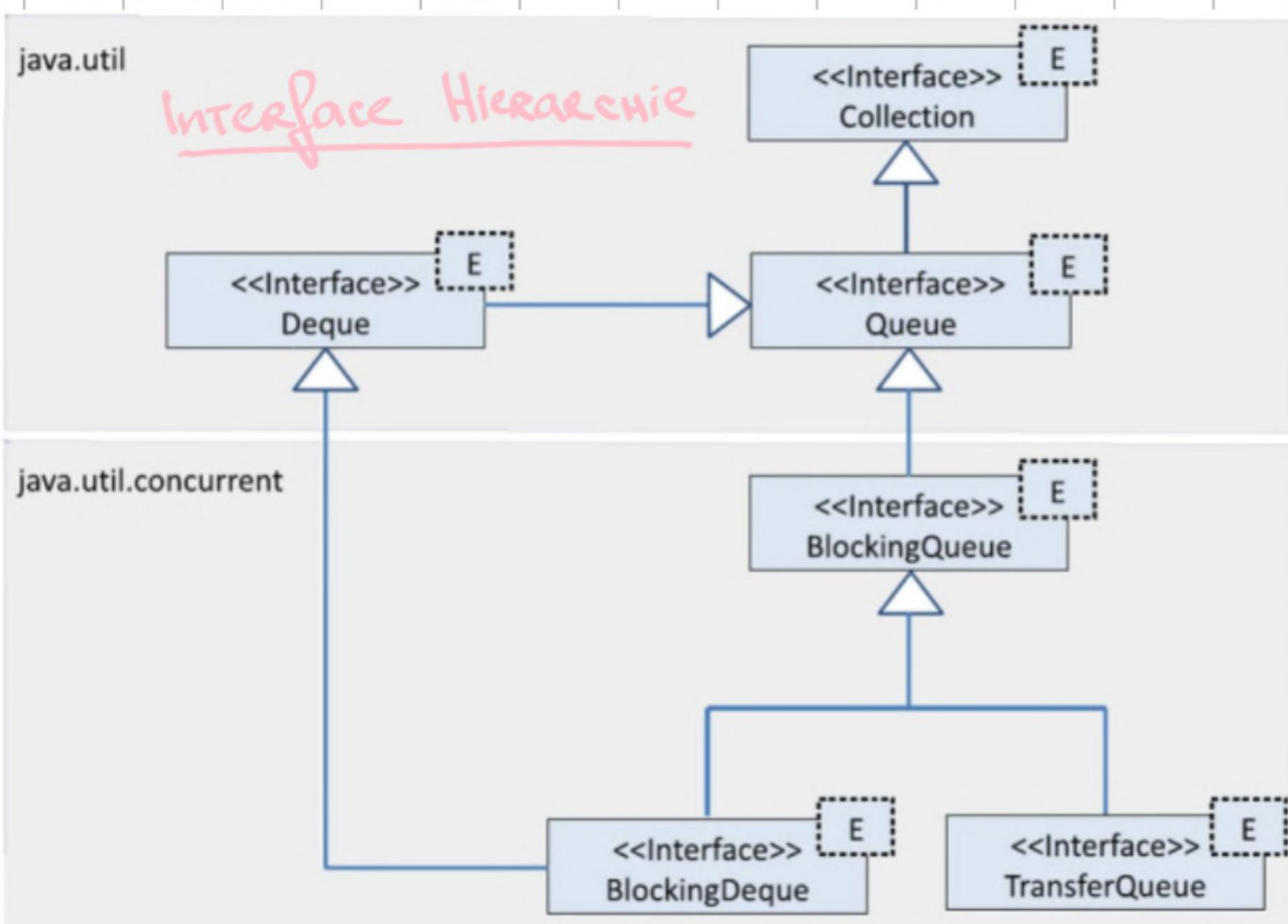
```
synchronized (syncList) {
    for (int i = 0; i < syncList.size(); i++) {
        // tut was mit syncList.get(i)
    }
}
```

Codeskizze

- Dadurch wird die Nebenläufigkeit sehr stark eingeschränkt, da der Container für alle anderen Zugriffe gesperrt wird.

Blocking Queues ↪ **bequeme Möglichkeit die Synchronisation zwischen Threads zu verwalten & sicherzustellen ohne aufwendige manuelle Synchronisationsmechanismen.**

- = DS, die als Warteschlange für das Austauschen von Elementen zwischen Threads verwendet wird.
- ↪ Spezielle Art der Queue. Unterrichtet **blockierende Operationen**.



↪ = Operationen zum **Einfügen/Entfernen von Elementen aus der Warteschlange** können blockiert werden, wenn bestimmte Bedingungen erfüllt.

↪ z.B.: Thread versucht Element einzufügen, aber Queue ist voll. Also: Thread wird blockiert bis Bedingung erfüllt.

↪ Wichtigste Methoden:

↪ **boolean offer(E e)**

↪ **boolean offer(E e, long timeout, TimeUnit unit)**

↪ was Operation erfolgreich (true)?  
gibt false zurück wenn angegebene Zeit abgelaufen ist.

↪ **E poll()**

↪ **E poll(long timeout, TimeUnit unit)**

↪ entnimmt ein Element vom Anfang der Queue. Liefert null, wenn kein Element vorhanden ist oder angegebene Zeit abgelaufen.

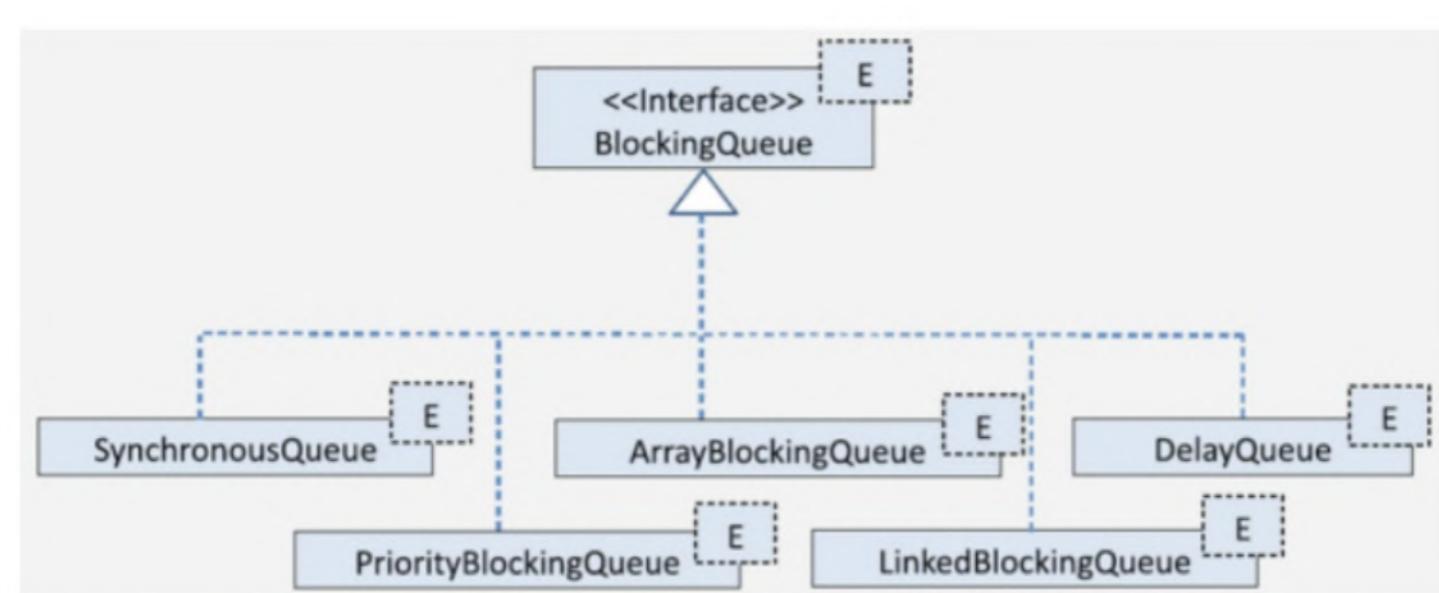
↪ **Void put(E e)**

↪ fügt ein Element am Ende der Queue ein & wartet (blockierend) ggf. bis ein Platz vorhanden ist.

↪ **E take()**

entnimmt Element am Anfang der Queue & wartet (blockierend) ggf. bis ein Element vorhanden ist.

## Wie implementiere ich eine Blocking Queue?



- Mit Ausnahme von PriorityBlockingQueue werden Elemente immer am Ende der Queue eingefügt und am Anfang der Queue entnommen (FIFO-Prinzip).
- LinkedBlockingQueue gibt es auch als LinkedBlockingDeque Variante.

### BlockingQueue - Konkrete Klassen

- **ArrayBlockingQueue<E>**
  - o ist eine Queue mit einer festen Größe (Kapazität).
- **LinkedBlockingQueue<E>**
  - o existiert sowohl als kapazitätsbeschränkte als auch als unbeschränkte Queue.
- **DelayQueue<E>**
  - o kann nur Objekte aufnehmen, deren Klasse das Interface Delayed implementiert. Für die interne Organisation werden die Methoden compareTo und getDelay verwendet.
- **PriorityBlockingQueue<E>**
  - o sortiert mit Hilfe der compareTo-Methode bzw. mit dem explizit angegebenen Comparator-Objekt ihre verwalteten Elemente.
- **SynchronousQueue<E>**
  - o ist eine blockierende Queue, bei der die beteiligten Threads aufeinander warten müssen. SynchronousQueue hat keine Kapazität.

# SEMESTERWOCHE 9

## Voraussetzungen für Sortieren

- unter den Datenelementen existiert eine totale Ordnung, bzw. lineare Ordnung:

↳ x ist kleiner y ODER  
 ↳ x ist y gleich ODER  
 ↳ x ist grösser y

Typischerweise verglichen über ihre Schlüssel / Keys

bspw. einziges Attribut

z.B. personID

lastName, firstName

- Sie müssen miteinander vergleichbar sein: z.B. Benzinauto vs. Benzinauto  
 Elektroauto vs. Elektroauto  
 ↳ Benzinauto vs. Elektroauto

- beim klassischen Sortieren vergleicht man alle Elemente miteinander & sortiert sie dann → vergleichsbasierte Algorithmen.

↳ z.B. seien  $n = 4$  Datenelemente:  $D_1, D_2, D_3, D_4$ .

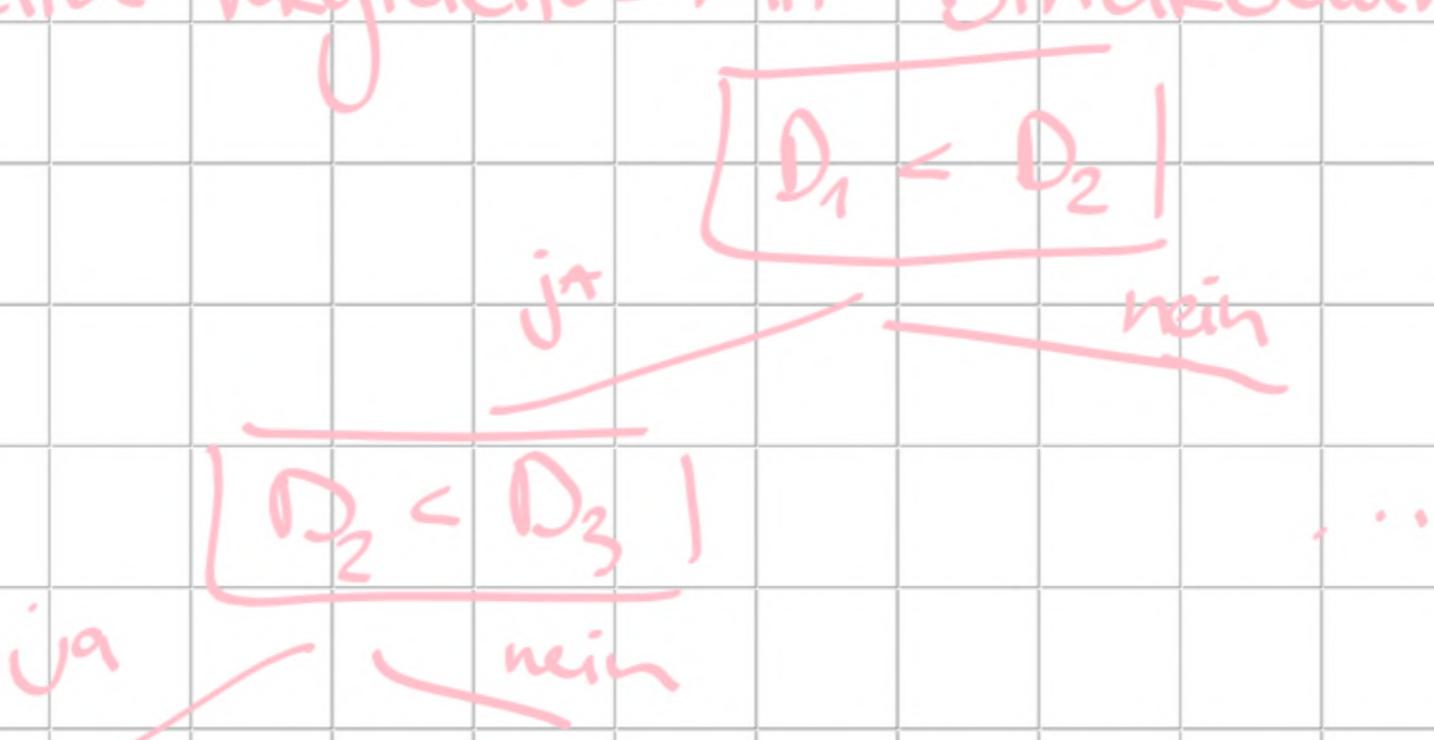
Auf allen Datenelementen ist eine totale Ordnung definiert, d.h. sie lassen sich untereinander vergleichen/bzw. sortieren.

↳ mit  $n = 4$  sind  $n! = 24$  verschiedene Ausgangssituationen für das Sortieren denkbar:

$D_1, D_2, D_3, D_4$   
 $D_1, D_2, D_4, D_3$   
 $D_1, D_4, D_2, D_3$   
 $D_4, D_3, D_2, D_1$   
 ;



bei solchen Vergleichen mit Binärbaum arbeiten!



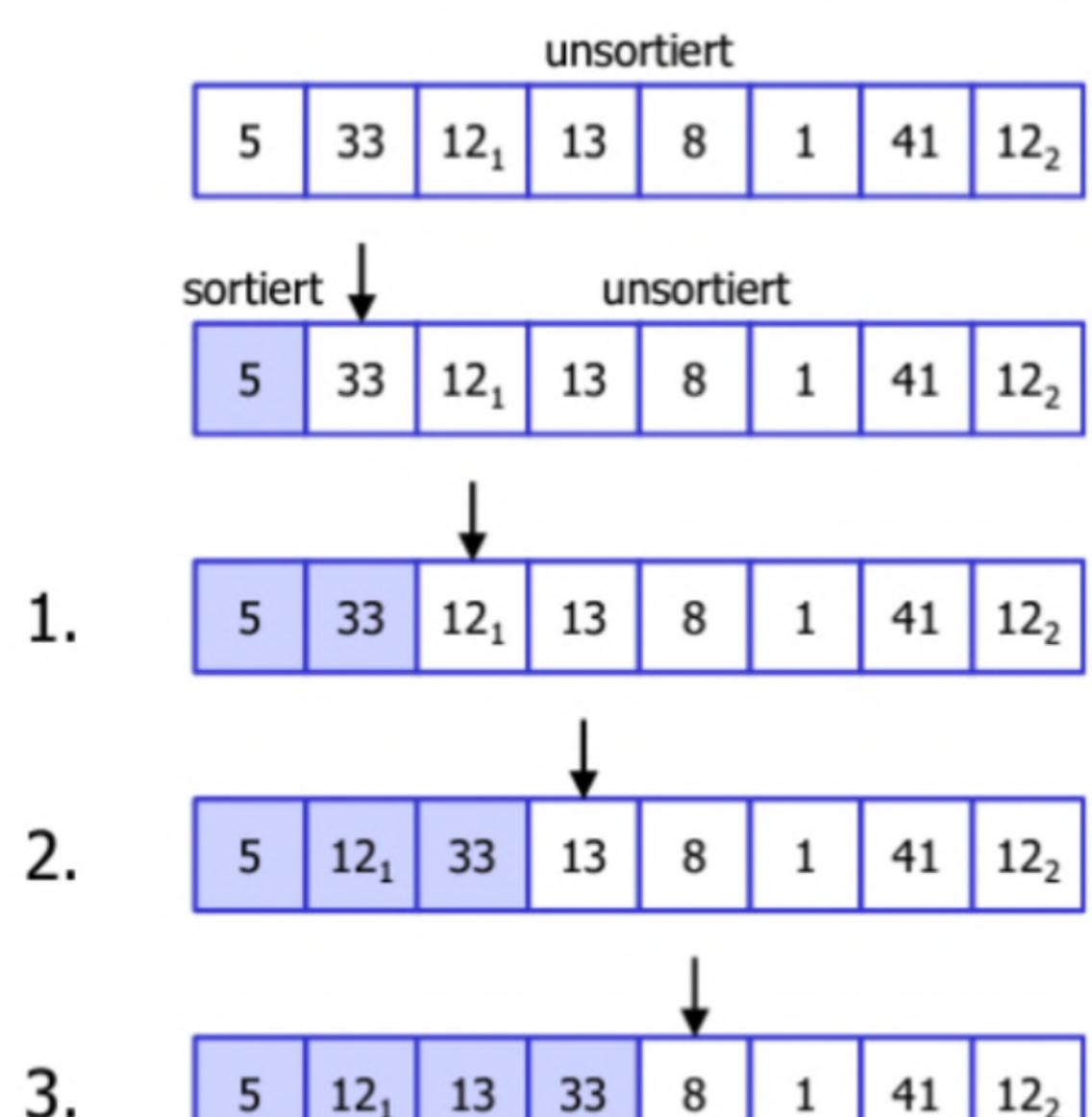
- damit man alle möglichen Ausgangslagen findet, sind gemäss der Baumtheorie (hier im Beispiel 6) mindestens  $(h-1)$  Vergleiche und "C" (compares) notwendig:  $C \geq (h-1)$ . Im Beispiel sind 5 Vergleiche notwendig.

Bei einem Binärbaum mit "B" Blättern:  
 $h \geq \log_2(B) + 1$

- Wenn Elemente zuerst nach der ersten Stelle sortiert werden, dann nach der zweiten etc., nennt man DAS → Radix-Sortieralgorithmen. ← oder auch Radix-Sort, Counting-Sort, Bucket-Sort.  
 ↳ sie arbeiten im besten Fall mit Zeitskomplexität  $O(n)$ !

↳ funktioniert wie Sortierung nach Einern: je nachdem wie viele es von einer Ziffer hat.

## Insertion Sort:



```
public static void insertionSort(final int[] a) {
    int elt;
    int j;
    for (int i = 1; i < a.length; i++) {
        elt = a[i]; // next elt for insert
        j = i; // a[0]..a[j-1] already sorted
        while ((j > 0) && (a[j - 1] > elt)) {
            a[j] = a[j - 1]; // shift right
            j--; // go further left
        }
        a[j] = elt; // insert elt
    } // a[0]...a[j] sorted
}
```

Best Case:  $O(n)$

↳ Array ist bereits sortiert.

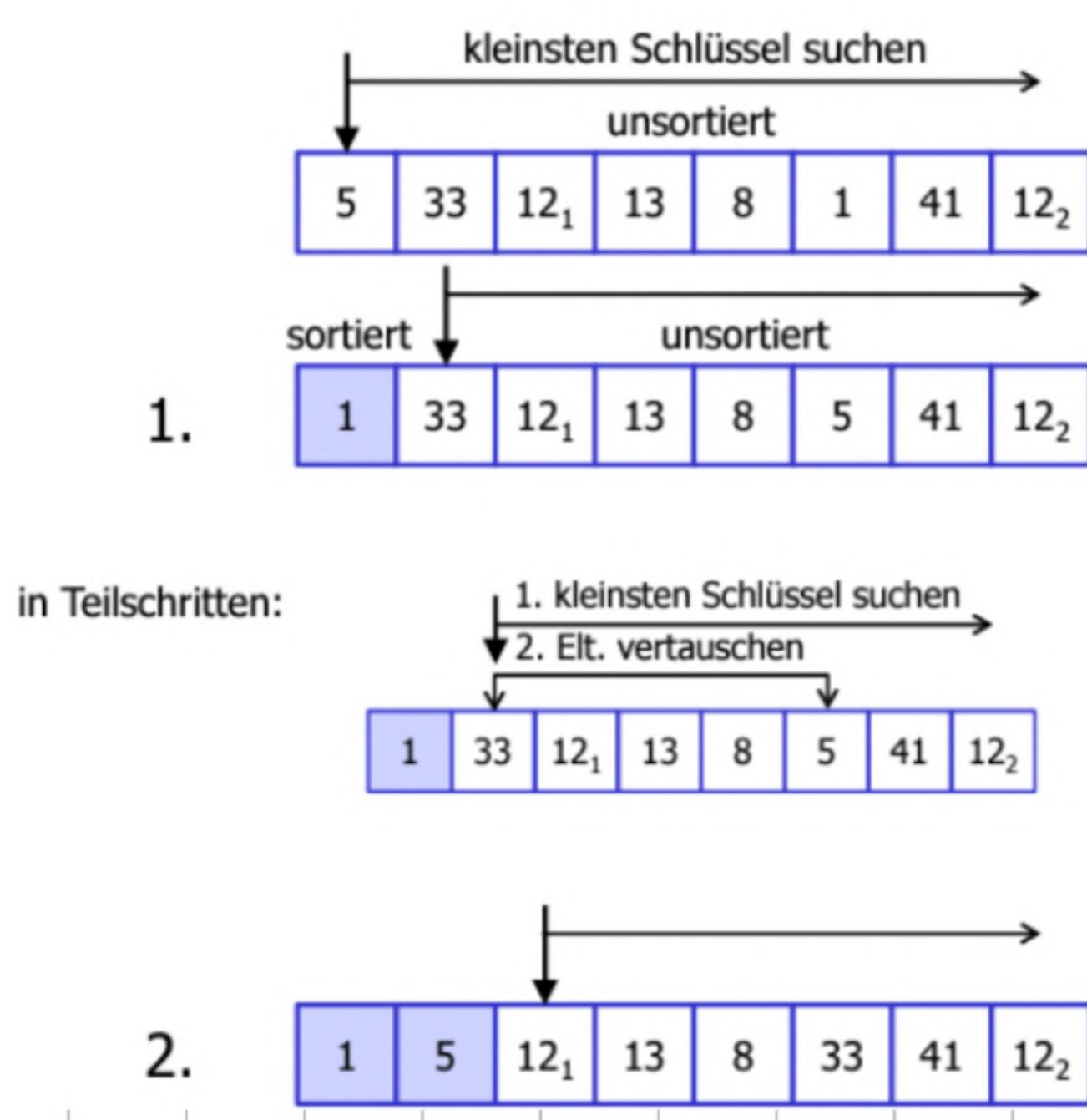
Avg. Case:  $O(n^2)$

↳ das einzufügende Element kommt in die Mitte des Arrays.

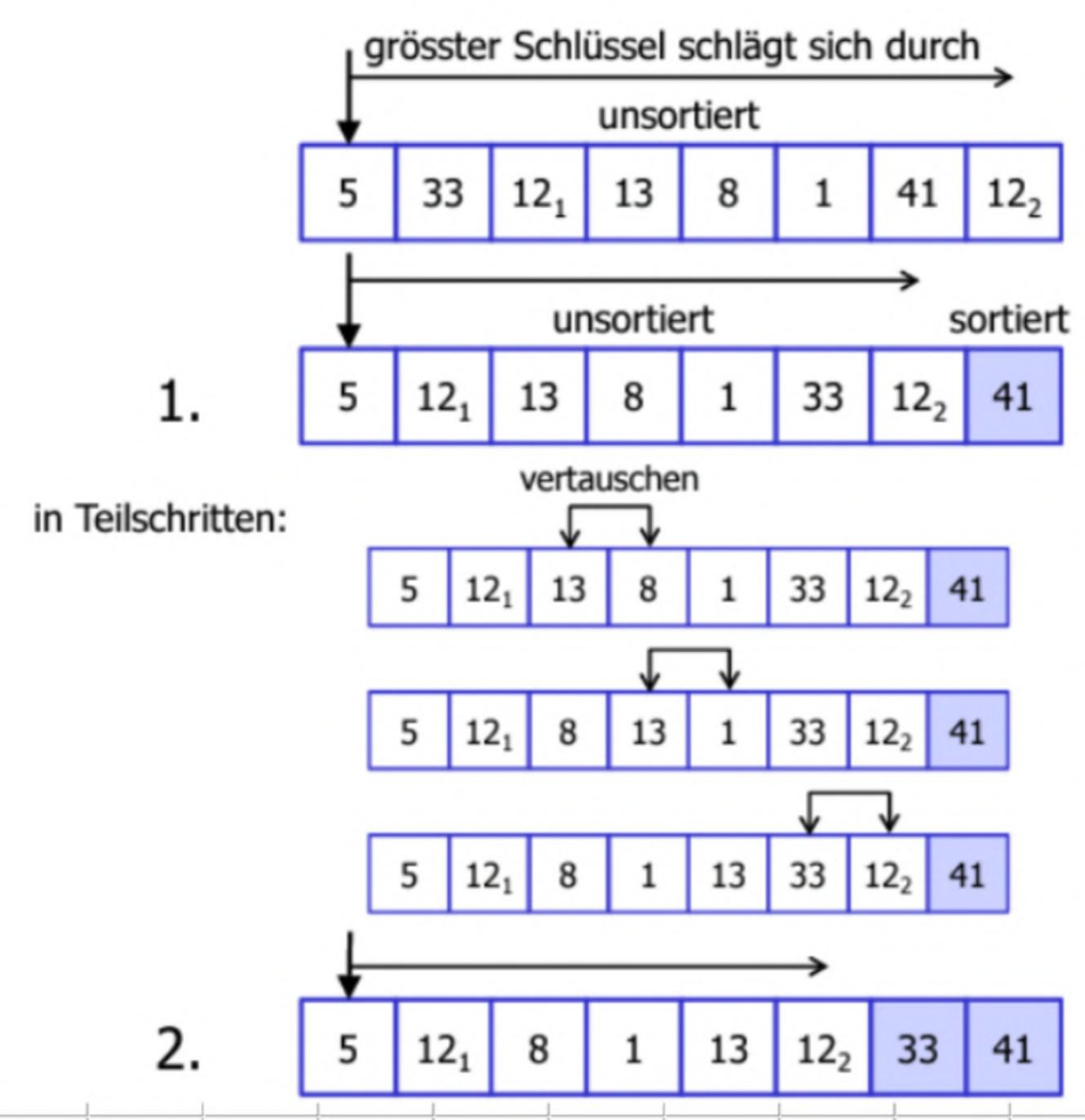
Worst Case:  $O(n^2)$

↳ Array ist umgedreht sortiert.

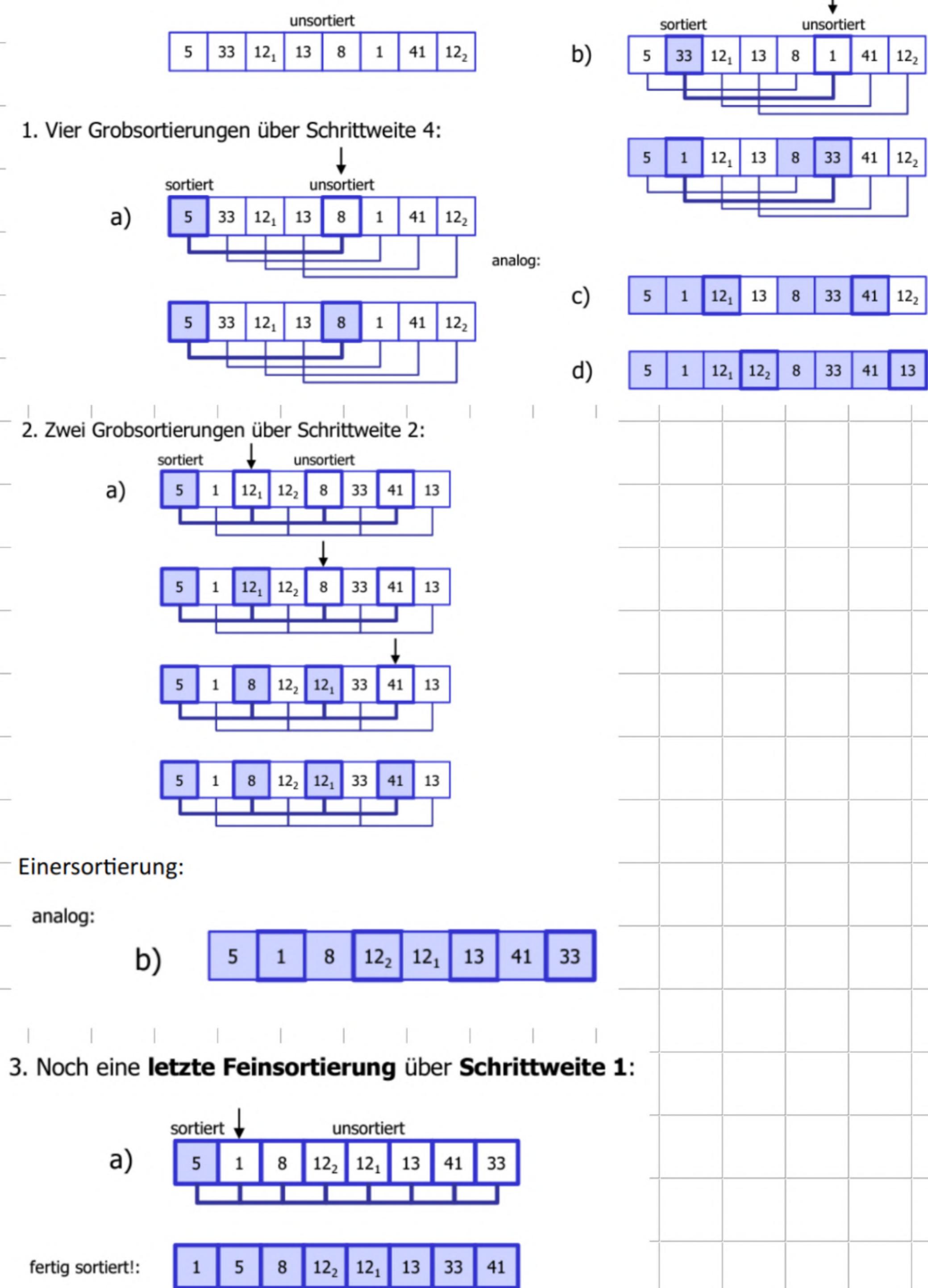
## Selection Sort:



## Bubble Sort:



## Shell Sort:



- einfache Sortieralgs besitzen meistens eine Zeithkomplexität von  $O(n^2)$
- Sortieralgs, die Daten über größere Distanzen verschieben, arbeiten schneller.
- es muss zwischen Best Case, Avg. Case & Worst Case unterschieden werden.
- Selection Sort & Insertion Sort bevorzugen!
- Shellsort ist eine Brücke zwischen einfachen & höheren Sortieralgs.  
 $\hookrightarrow O(n^{1.5})$

# SEMESTERWOCHE 10

- "Teile & Herrsche" → Lösungsprinzip von Quicksort => Problem in Teilprobleme zerlegen, lösen. Dann Teillösungen zur Gesamtlösung zusammensetzen.
- "Reduziere & Herrsche" → " " " Bubble Sort & Selection Sort

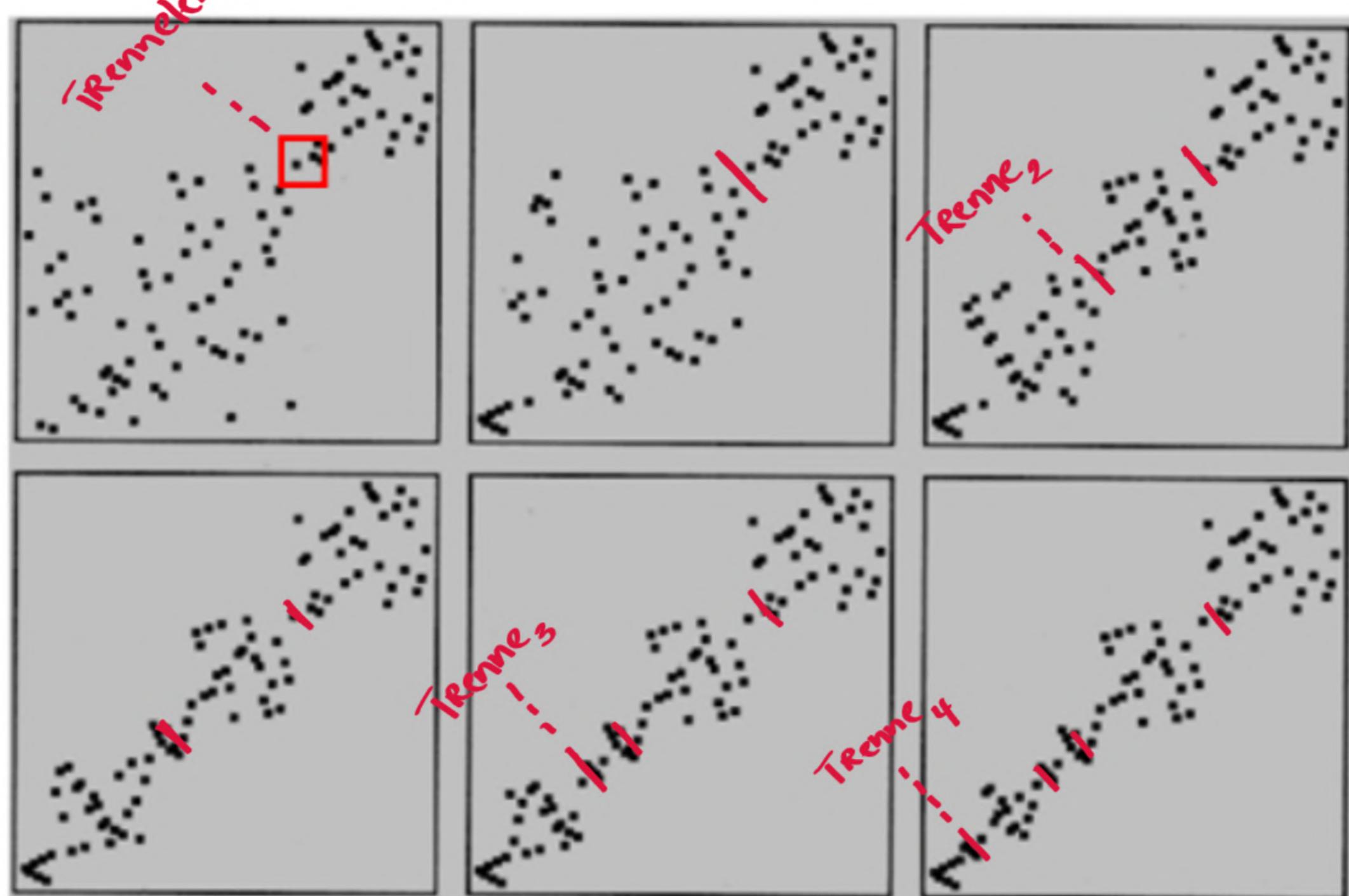
so erreicht man einen rekursiven Lösungsansatz.

sofern die Teilprobleme unabhängig voneinander lösbar sind, ist es sogar parallel.

Implementation:

```
/*
 * Vertauscht zwei bestimmte Zeichen im Array.
 *
 * @param a Zeichen-Array
 * @param firstIndex Index des ersten Zeichens
 * @param secondIndex Index des zweiten Zeichens
 */
private static final void exchange(final char[] a,
    final int firstIndex,
    final int secondIndex) {
    char tmp;
    tmp = a[firstIndex];
    a[firstIndex] = a[secondIndex];
    a[secondIndex] = tmp;
}
```

Quicksort Prinzip visualisiert:



```
public static final void quickSort(final char[] a, final int left, final int right) {
    int up = left; // linke Grenze
    int down = right - 1; // rechte Grenze (ohne Trennelement)
    char t = a[right]; // rechtes Element als Trennelement
    boolean allChecked = false;
    do {
        while (a[up] < t) {
            up++; // suche grösseres (>=) Element von links an
        }
        while ((a[down] >= t) && (down > up)) {
            down--; // suche echt kleineres (<) Element von rechts an
        }
        if (down > up) { // solange keine Überschneidung
            exchange(a, up, down);
            up++; down--;
            // linke und rechte Grenze verschieben
        } else {
            allChecked = true; // Austauschen beendet
        }
    } while (!allChecked);
    exchange(a, up, right); // Trennelement an endgültige Position (a[up])
    if (left < (up - 1)) quickSort(a, left, (up - 1)); // linke Hälfte
    if ((up + 1) < right) quickSort(a, (up + 1), right); // rechte Hälfte, ohne T'Elt.
}
```

Weiteres Beispiel anhand vom ABC:

```
quickSort(0/11) → quickSort(5/11) Orange: sortierter Teil.
ABCDEFFIKHEG  
ABCDEFEKHIG  
ABCDEFFEGHIK  
quickSort(5/7) Rot: Trennelement
ABCDEFFEGHIK  
ABCDEEFFGHIK  
quickSort(6/7) ABCDEEFFFGHIK
ABCDEEFFEGHIK  
quickSort(0/3) ABCDEEFFFGHIK
DACBEFFIKHEG  
ADCBEFFIKHEG  
ABCDEFFIKHEG  
quickSort(9/11) ABCDEEFFFGHIK
ABCDEEFFGHIK  
ABCDEEFFGHIK  
quickSort(2/3) quickSort(9/10)
ABCDEFFIKHEG ABCDEEFFFGHIK
ABCDEFFIKHEG ABCDEEFFGHIK
```

- Was wenn Elemente gleich sind wie das Trennelement? → wenn es viele sind, dann werden sie vor dem Trennelement geschrieben.

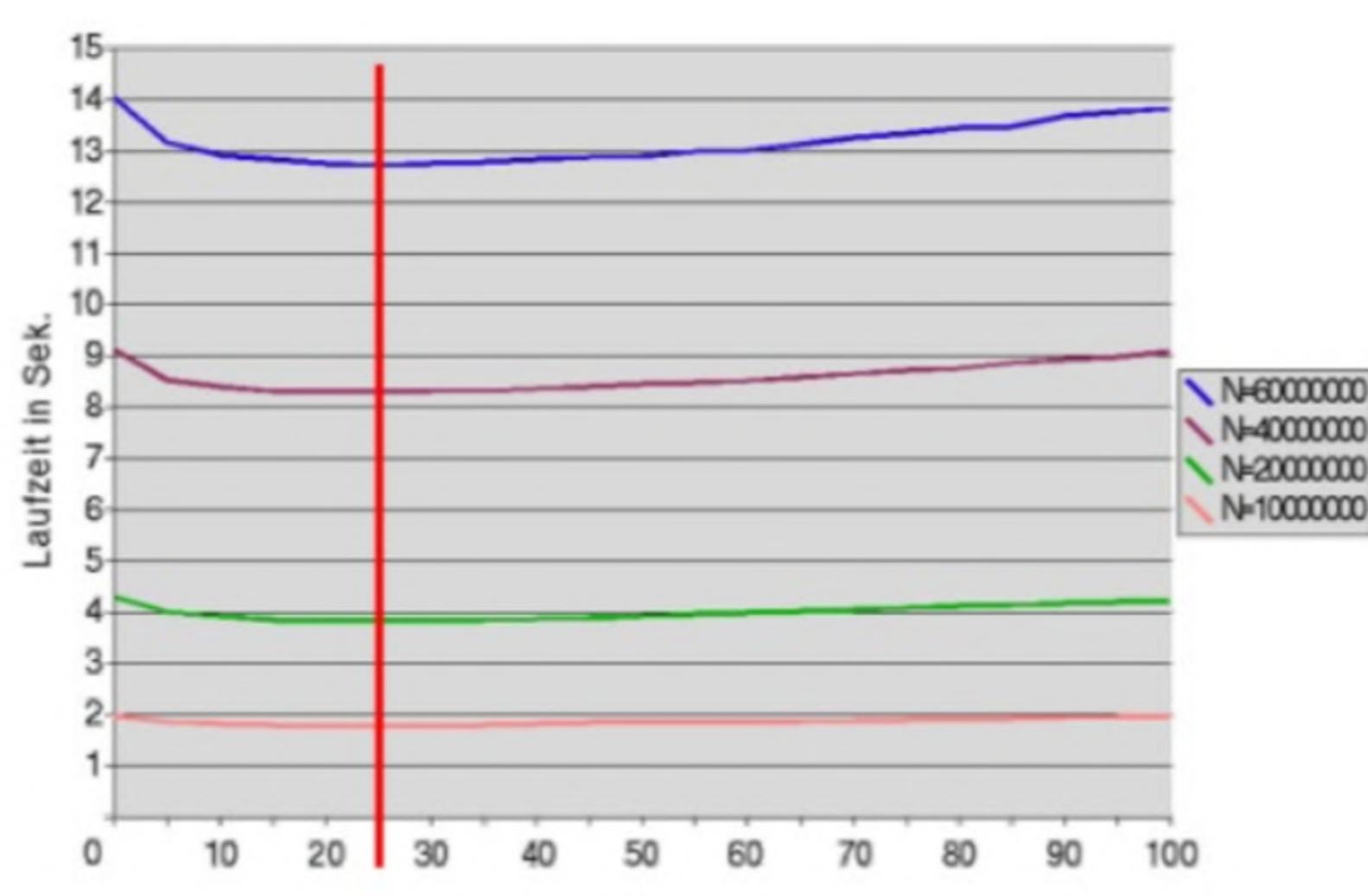
→ Es resultiert: T wird rechts geschoben.

→ damit es symmetrischer wieder wird, lässt man diese Elemente auch die Seite wechseln.

- Was wenn wir nur noch 2 Elemente zu sortieren haben? Rufen wir Quicksort dafür auf! Benötigt das viel Zeit & Speicher.

→ Lösung: Sortieralgorithmen kombinieren, z.B. mit insertion sort. Resultat: Laufzeitverbesserung um 20%.

Beispiel: Quick-Insertion-Sort: Für welches M ist die Laufzeit = f(M) minimal, und zwar bei N = 10, 20, 40 und 60 Mio. Elementen?



Den Messungen kann man entnehmen, dass M praktisch unabhängig von N ist und die kürzeste Laufzeit bei  $M \approx 25$  liegt.

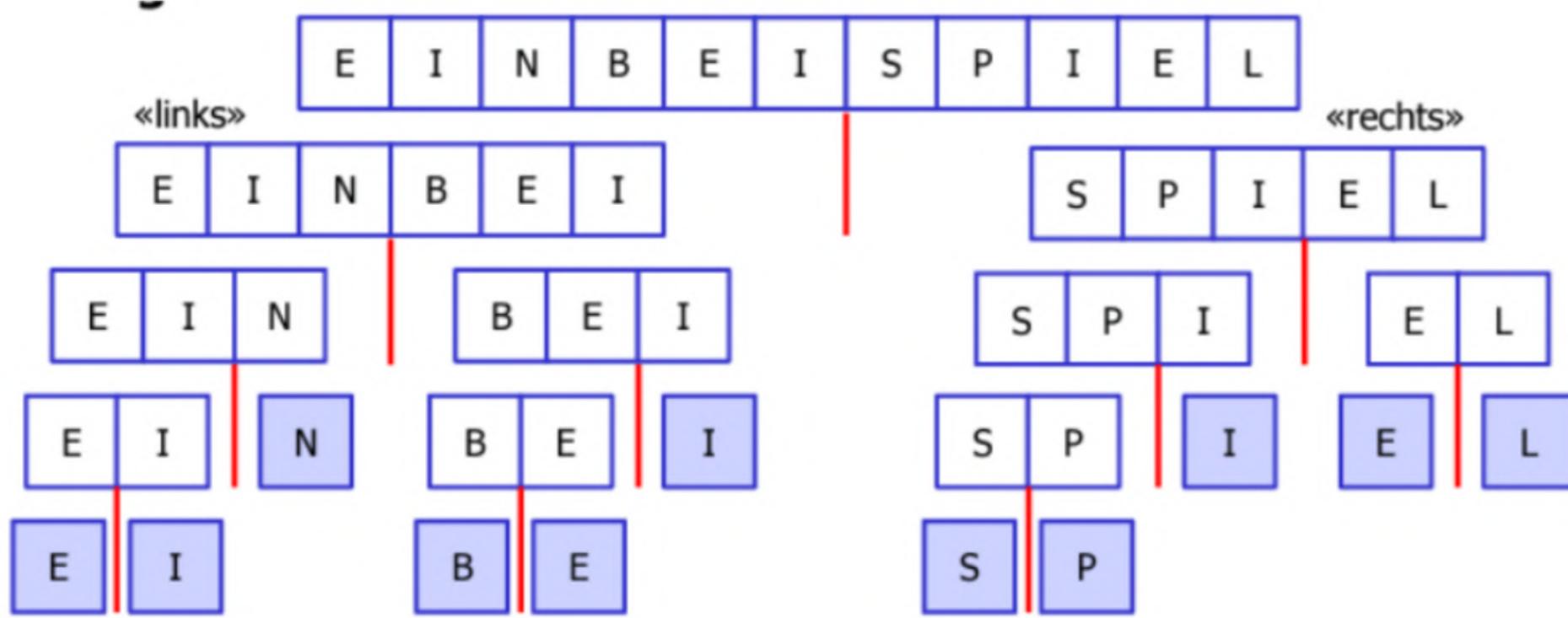
Wenn man sich  $M = 0$  vorstellt sortiert man mit reinem Quicksort zuende. Wenn man M beliebig gross macht, z.B. 60 Mio. Würde man nur noch nach direktem einfügen sortieren. Man muss also einen «Sweet-Spot» finden, um zum Insertion Sort zu wechseln.

Beispiel M = 5 mit ABC

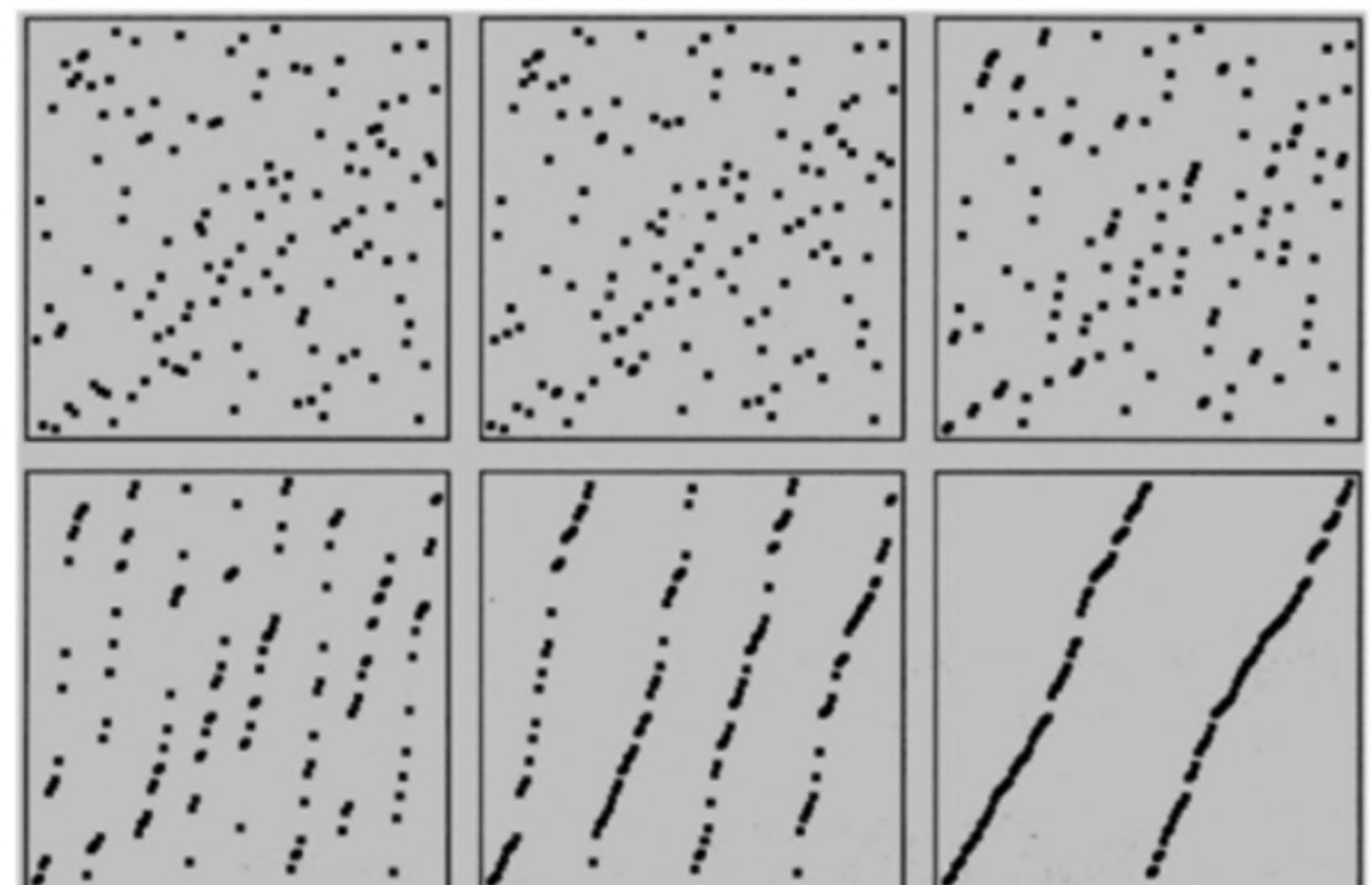
```
quickSort(0/11) → quickSort(5/11)
ABCDEFFIKHEG  
ABCDEFEKHIG  
ABCDEFFEGHIK  
insertionSort(5/7) ABCDEFFEGHIK
ABCDEEFFGHIK  
insertionSort(9/11) ABCDEEFFFGHIK
ABCDEEFFGHIK  
insertionSort(0/3) insertionSort(0/3)
DACBEFFIKHEG  
ADCBEFFIKHEG  
ABCDEFFIKHEG
```

# Mergesort

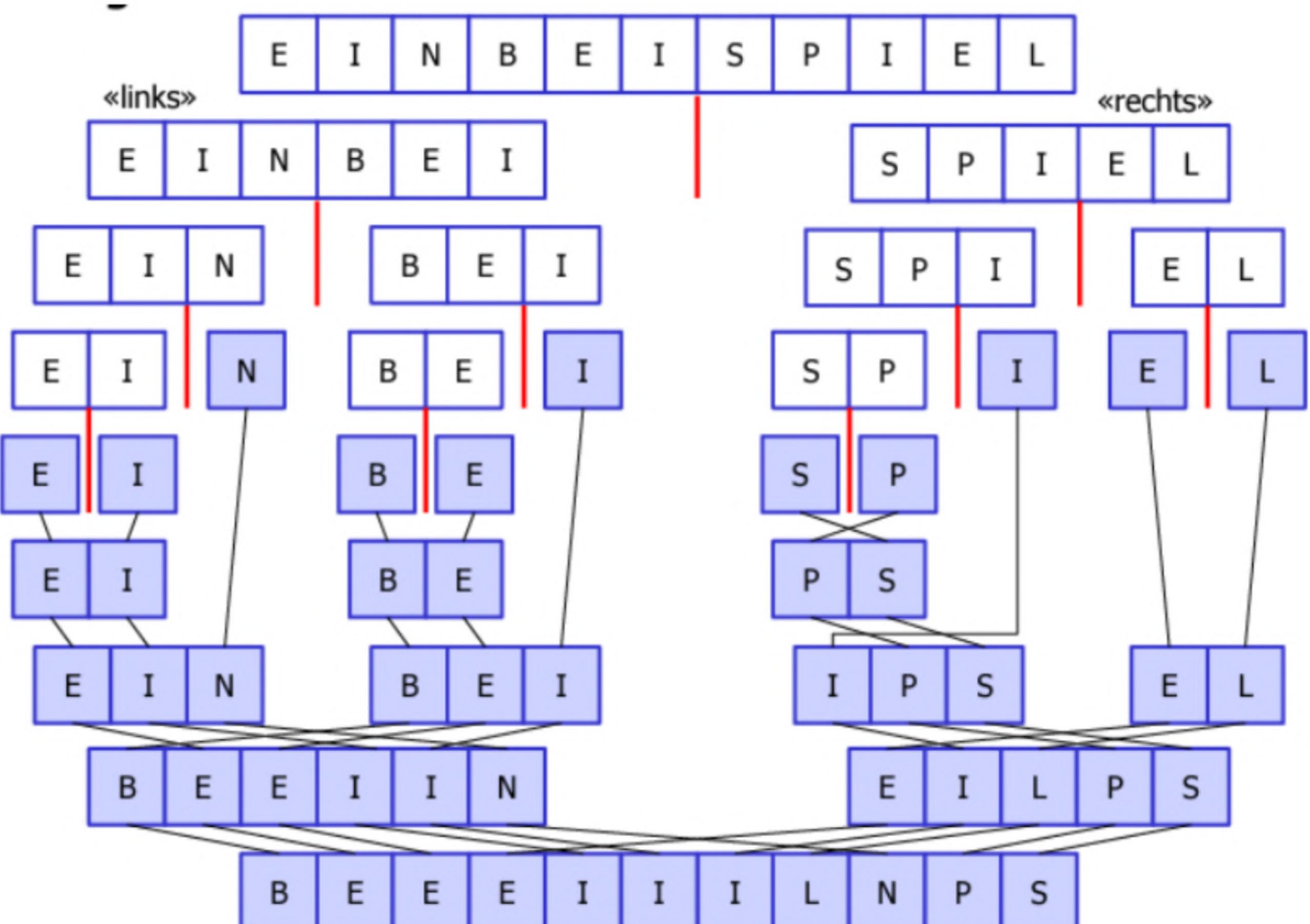
In der ersten Phase wird nur halbiert, bis es nichts mehr zu halbieren gibt.



Dann gehts rückwärts, das alles wird rückwärts zusammengemischt / gemerged und sortiert.



weil man immer  
halbierst sieht man  
hier gut am Schluss,  
dass die 2 Hälften  
nur noch gemerged  
werden müssen.



## Implementation:

```

private static char[] b;
/**
 * Sortiert ein Zeichen-Array mit dem Mergesort-Algorithmus.
 * @param a Zeichen-Array zum Sortieren
 */
public static void mergeSort(final char[] a) {
    b = new char[a.length]; // zusätzlicher Speicher fürs Mergen
    mergeSort(a, 0, a.length - 1);
}

/**
 * Rekursiver Mergesort-Algorithmus.
 * @param a Zeichen-Array zum Sortieren
 * @param left linke Grenze, zu Beginn 0
 * @param right rechte Grenze, zu Beginn a.length - 1
 */
private static void mergeSort(final char a[], final int left,
    final int right) {
    ...
}
private static void mergeSort(final char a[], final int left, final int right) {
    int i, j, k, m;
    if (right > left) {
        m = (right + left) / 2; // Mitte ermitteln
        mergeSort(a, left, m); // linke Hälfte sortieren
        mergeSort(a, m + 1, right); // rechte Hälfte sortieren
        // "Mergen"
        for (i = left; i <= m; i++) { // linke Hälfte in Hilfsarray kopieren
            b[i] = a[i];
        }
        for (j = m; j < right; j++) { // rechte Hälfte umgekehrt in Hilfsa. kopieren
            b[right + m - j] = a[j + 1];
        }
        i = left; j = right; // Index für linke und rechte Hälfte
        for (k = left; k <= right; k++) { // füge sortiert in a ein
            if (b[i] <= b[j]) {
                a[k] = b[i]; i++;
            } else {
                a[k] = b[j]; j--;
            }
        }
    }
}

```

Rekursive Methoden werden zuerst aufgerufen.