

RECAP Einführung

Was ist ein Algorithmus? z.B. euklidischer Algo, Bresenham, Mergesort, Dijkstra, Sieb von Eratosthenes, etc.

→ Lösungsverfahren für eine Problemklasse.

→ Verarbeitet durch eine Abfolge von Befehlen Input → Output für ein Problem (oder mehrere Probleme).

Algos haben immer ein Ende, d.h. terminiert!

Weitere Eigenschaften:

- SCHITTWEISES VERFAHREN
- WAS ein Schritt zu leisten hat, ist eindeutig
- ↳ z.B. liefere mir eine Zufallszahl. IST aber bei jedem Durchlauf anders.

Was sind gleichwertige Algos?

→ Algos können auf versch. Arten implementiert werden

↳ z.B. REKURSIVE ART (wenn sie sich selbst aufrufen)

↳ z.B. ITERATIVE ART (durch Schleifen)

→ Können den gleichen Output haben, d.h. gleichwertig sein, ABER sie sind ≠ gleich!

Wieso hängen Datenstrukturen mit Algos zusammen?
= Konzept zur Speicherung & Organisation von Daten. Insbesondere durch Operationen charakterisiert, also versch. Array-Arten

→ Algos arbeiten auf Datenstrukturen.

→ Datenstrukturen werden durch Algos verwaltet.

Was heißt Komplexität?

→ = Aufwand in Abhängigkeit des Inputs

→ Ressourcenbedarf ist vor allem

↳ Zeitkomplexität = Rechenzeit

↳ Speicherkomplexität = Speicherbedarf

, ist aber eig. mehr z.B. auch Java Garbage Collection, Compilation, Anzahl Cores, etc.

→ über Big O Notationen beschrieben

$$\text{ressourcenbedarf} = f(\text{Eingabedaten})$$

akademische Notation

Einfache Funktionen deren Ordnung bestimmen

→ Rules: 1) Konstanten weglassen

2) Vom "worst Case" ausgeln

3) Der am schnellst wachsende Faktor ist ausschlaggebend

4) Basis von log egal

→ Ordnungsfunktionen im Vergleich: $O(1) < O(\log(n)) < O(n) < O(n \cdot \log(n)) < O(n^m) < O(d^n) < O(n!)$

Einfache Funktionen deren Ordnung interpretieren

→ exakte Analyse unmöglich, da Konstanten in Realität doch eine grosse Rolle spielen.

→ Analyse von Algos nur für grosse n

→ Bei kleinen n → Algo mit hohem Ordnungsfaktor evtl. effizienter

RECAP REKURSION

= "Ähnlichkeit"

WAS HABEN Algos und DATENSTR. mit SELBSTÄHNLICHKEIT und SELBSTBEZUG zu tun?

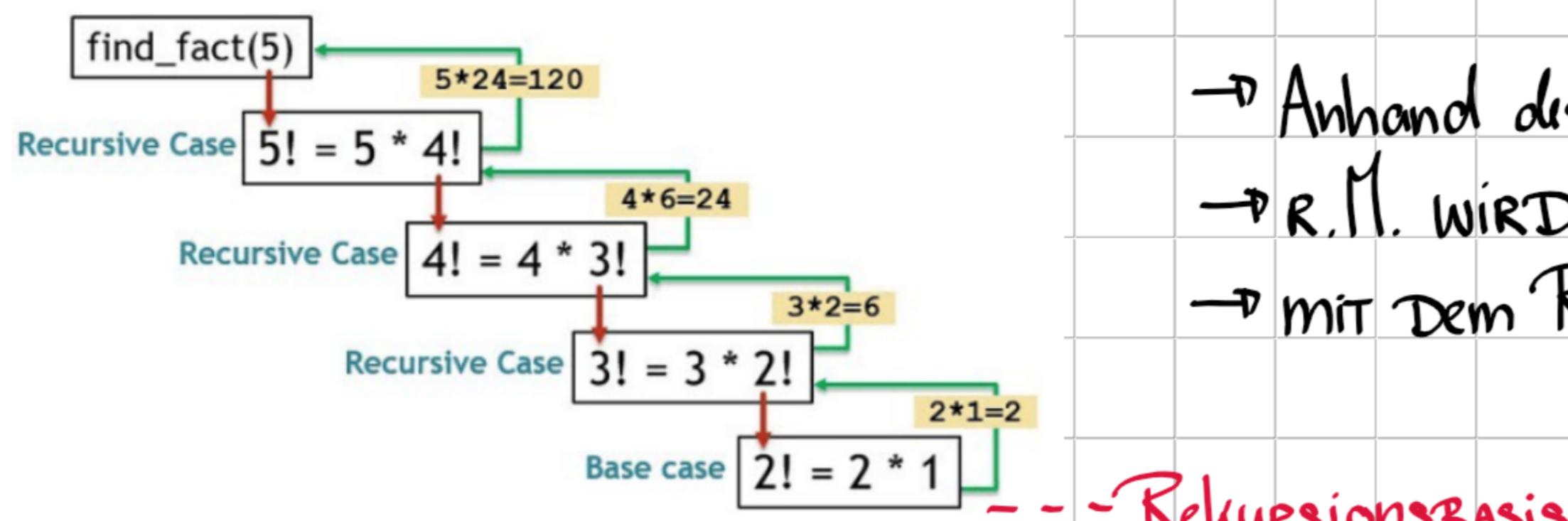
- meistens sind sie REKURSIV, d.h. ähnlich.
- VERGLEICHBAR mit ORDNERN auf Windows: 

| JEDE iterative Lösung kann auch REKURSIV gemacht werden & umgekehrt. |

WAS IST Bei einer REKURSIVEN METHODE REKURSIONSBASIS & REKURSIONSVORSCHRIFT?

- R.M. bestehen immer daraus.
- BASIS = WANN DER Algo TERMINIERT WIRD.
- VORSCHRIFT = MATH. FORMEL/REGEL um eine Folge von ZAHLEN/OBJEKTEN zu definieren.
 - beschreibt, wie jedes Element in der Folge basierend auf seinen VORHERIGEN Elementen berechnet wird.

Wie WIRD eine REKURSIVE METHODE abgearbeitet?



→ Anhand des Bsp. gut erkennbar.

→ R.M. WIRD ZUERST 1x GANZ DURCHGEARBEITET BIS ZUR REKURSIONSBASIS. → ZUR TIEFSTEN EBENE.

→ MIT DEM RESULTAT WIRD DANN RÜCKWÄRTS GEРЕЧНЕТ & ALLES AUFGELOST.

Was ist HEAP und CALL STACK?

- 2 SPEICHER, welche JVM braucht um Programm laufen lassen.
- HEAP: hier werden Objekte + Zustände (Variablen) gespeichert. & DATENSTRUKTUREN (z.B. ARRAYS).
 - WIRD DURCH GARBAGE COLLECTOR verwaltet.
- CALL STACK: hier werden alle Methodenaufrufe (FIRST: main() - METHOD) gespeichert.
 - FIFO-PRINZIP: erstes hinzugefügtes Element als LETZTES BEARBEITET
 - JEDER Methodenaufruf bewirkt: Stack Frame wird angelegt. Sobald Methode abgeschlossen - Stack entfernt.
 - enthält: Methodenname, Parameter, lokale Variablen, etc.
- CALL STACK HAT FIXE GRÖSSE. Bei Aufruf von zu viel Methoden (oder fehlerhafter Rekursion) → STACK OVERFLOW → Programm STÜRZT AB.

Wo ist der Unterschied zwischen REKURSION vs. ITERATION?

| JEDE iterative Lösung kann auch REKURSIV gemacht werden & umgekehrt. |

→ Wir sprechen von LINEARER REKURSION! (Es gibt auch: NICHT LINEARE R., NICHT GE SCHACHTELTE R., NICHT LINEAR GE SCHACHTELTE R.)

LINEARE REKURSION

- mehr als 1 neuer Methodenaufruf
- VERWENDET Funktionsaufrufe um Abläufe zu wiederholen
- MEHR Speicherplatz im CALL STACK weil mehr Methoden
- kann leichter in unendliche Schleife geraten, wenn Abbruchbedingung schlecht formuliert
- Risiko von STACK-Overflow

ITERATION

- nur 1 Methodenaufruf
- VERWENDET Schleifen Abläufe zu wiederholen
- weniger Speicherplatz, da nur Schleife
- klarere Anweisungen für Termination der Schleife

- kein Risiko von Stack-Overflow

RECAP COLLECTIONS

Was sind DATENSTRUKTUREN (DS)? ↗ z.B. Array, List, Tree, Map, etc.

→ Werden verwendet, um Menge von Daten/Objekten effizient zu speichern & zu verarbeiten.

Eigenschaften von DATENSTRUKTUREN?

→ z.B. Größe, Zugriff, Sortierung, Suche, Geschwindigkeit.

Java Collection Framework:

→ Stellt eine Auswahl von DS zur Verfügung.

→ besteht aus 3 Hauptteilen: Interfaces, Schnittstellen, Algorithmen.

→ Interfaces: Repräsentieren DS (z.B. List, Map.)

→ Schnittstellen: Können konkret im Projekt verwendet werden. (z.B. LinkedList, ArrayList.)

→ Algorithmen: vordefinierte Methoden für die Bearbeitung der DS. (z.B. iterator(), sort().)

Welche Arten von DS gibt es? ↗ die eingesetzte Wahrheitssatz gibt es nicht. Jede DS hat \oplus & \ominus .

→ **ARRAY**: indexierte Reihung.

→ **LIST**: einfache Reihung.

→ **Tree**: Hierarchisch geordnete Daten.

→ **Map**: Zuordnung Schlüssel - Wert.

→ **Queue**: Schlange FIFO oder Dequeue.

→ **Set**: Sammlung ohne Duplikate.

→ **Stack**: Stapel- oder Kellerspeicher FILO.

RECAP DATENSTRUKTUREN

Was sind die **Eigenschaften** von **Datenstrukturen**?

→ Unterscheiden sich z.B. in **Sortierung, Operationen, statisch/dynamisch, explizite/implizite BZ, direkter/indirekter Zugriff, Aufwand**.

Wie ist die **Komplexität** von Operationen auf untersch. Datenstrukturen?

→ Stellt sich aus versch. Operationen zusammen.

→ z.B. Bei einem **sortierten Array**. Darin möchte man ein Element einfügen.

↳ zuerst muss das Array an der richtigen Stelle durchsucht werden, dann Element eingefügt.
Verschiebt dann Elemente nach rechts.

↳ All diese Operationen haben ihre eigene **Komplexität** → man rechnet alle zusammen & nimmt grösste Komplexität als RICHTWERT.

Aufbau, Eigenschaften, Funktionsweise ausgewählter Datenstrukturen.

→ ARRAY:

- statisch
- implizite BZ zwischen Elementen
- direkter Zugriff auf Daten
- schnell

→ LIST:

- dynamisch
- sequenzieller Zugriff
- explizite BZ zwischen Elementen
- konstant

Wie implementiere ich DS?

→ Z.B. **ARRAYSTACK**:
a) Stack als Interface definieren
b) Klasse ARRAYSTACK implementieren
c) dazugehörige Methoden überschreiben.

Welche DS nehme ich?

→ **ARRAY** → statische, kleine Datenmengen.

LIST → dynamische, grosse Daten

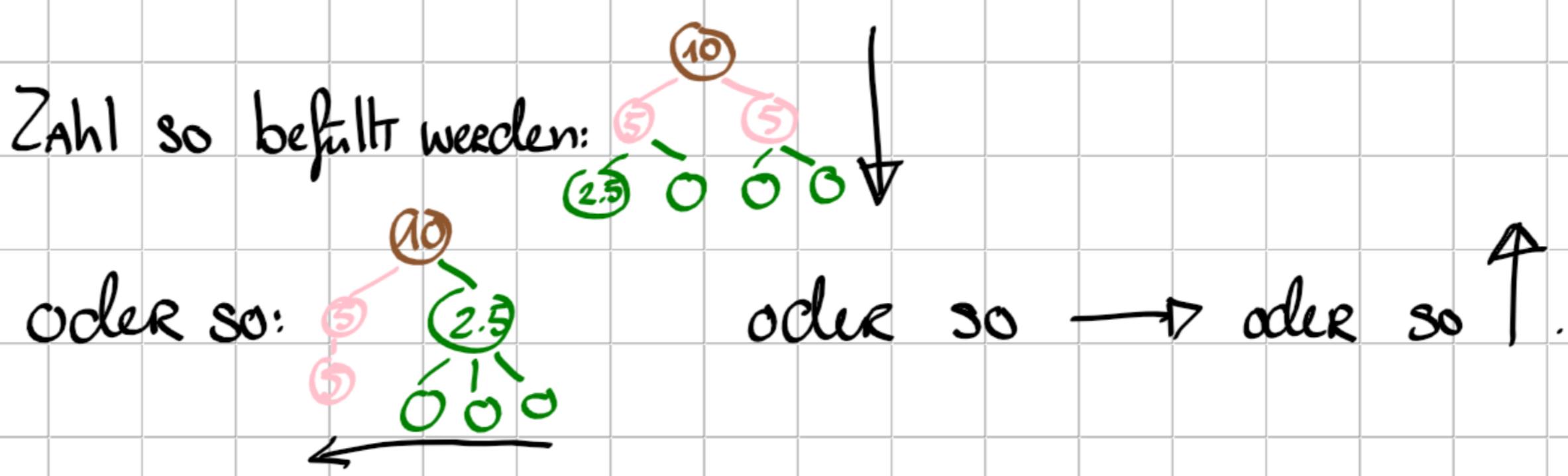
Queue → für Warteschlangen

Stack → Tiefenspeicher für Call Stack JVM, Tasks die aufeinander aufbauen & nacheinander in derselben Reihenfolge wieder abgearbeitet.

RECAP DS - BÄUME

Wie ist eine baumartige Datenstruktur (DS) aufgebaut?

- Wie ein hängender Baum.
- Es gibt eine Wurzel, den Root. An der Wurzel sind Äste (Kanten), Blätter (Knoten) und Blätter an Blättern (Leaf).
- Die Implementation erfolgt rekursiv.
- Kann bsp. mit grösster Zahl bis kleinster Zahl so gefüllt werden:



Welche Bäume gibt es?

- **Binärbaum**: Knoten haben immer nur max. 2 Children! z.B für Sortier- und Suchalgs.
- **AVL-Baum**: Binärbaum^①. Höhe der linken Hälfte des Baumes & Höhe der rechten Hälfte dürfen sich hier nur um max. 1 unterscheiden.
- **B-Baum**: Balancierter Baum. z.B für Organisation grosser Datensets.
- **Binomial-Baum**: Baumstruktur aus Sammlung von Bäumen, die nach bestimmten Regeln kombiniert werden.

Wurzel, Knoten, Blatt, Kanten - Baum ABC

- Siehe Notizen & Bild für besseres Verständnis.

Kenngrößen eines Baumes

→ **Ordnung**:

→ **Grad**:

→ **Pfad**:

→ **Tiefe**:

→ **Niveau**:

→ **Höhe**:

→ **Gewicht**:

→ **Füllgrad**:

- **Balanciert**: Gleichmässige Verteilung der Knoten auf versch. Ebenen. Führt zu effizienten Suche & Einfügung von Elementen.
- **Symmetrisch**: Baum hat spiegelbildliche & gleichmässige Eigenschaften. Links von der Wurzel sieht gleich aus, wie rechts.

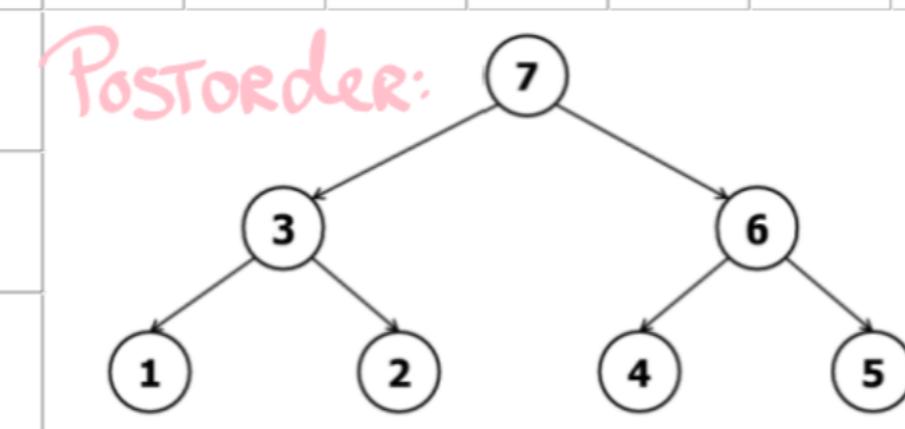
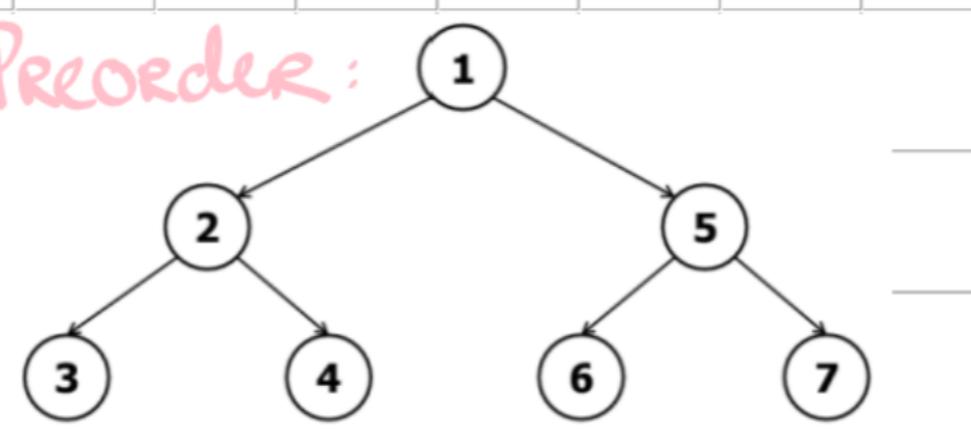
RECAP BINÄRE BÄUME

WAS SIND binäre Bäume?

- besitzen die **Ordnung 2**
- Höhendifferenz der Niveaus darf **max 1** betragen. → d.h. muss ausgeglichen & balanciert sein.
 - ↪ Wenn nicht der Fall, muss Baum neu organisiert werden. → über Rotationen.
- schnell in der Suche mit Aufwand $O(\log(n))$.

Wie TRAVERSiere ich BINÄRE BÄUME?

- 3 versch. Algos dafür

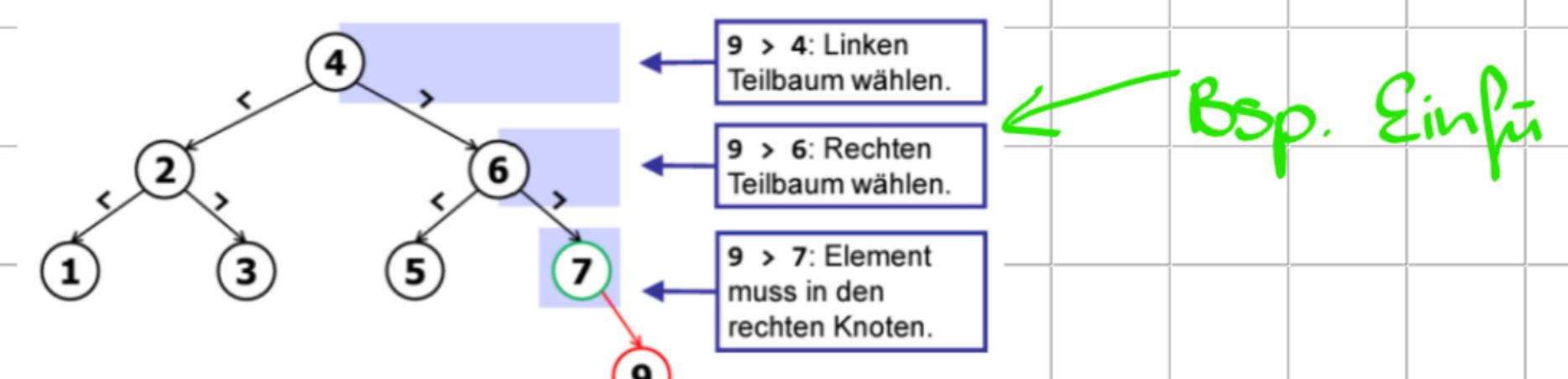


- **PreOrder** (Hauptreihenfolge): Wurzelknoten wird zuerst besucht, dann auf den linken Teilbaum, dann auf Rechten. → **Wurzel-Links-Rechts**
- **PostOrder** (Nebenreihenfolge): Zuerst linke Teilbaum traversiert, dann Rechte, dann Wurzelknoten → **Links-Rechts-Wurzel**.
- **InOrder** (Symm. Reihenfolge): Zuerst linke Teilbaum, dann Wurzelknoten, dann rechte Teilbaum → **Links-Wurzel-Rechts**.

WAS SIND die speziellen Eigenschaften von Binären SuchBäumen?

- **SORTIERUNGSREGEL**: Die Keys in den Knoten sind so angeordnet, dass alle Schlüssel im linken Teilbaum eines Knotens kleiner sind als der Key des Parents und alle Schlüssel im rechten Teilbaum größer sind.
- **EFFIZIENTE SUCHE**: Durch Sortierungsregel kann eine binäre Suche effizient durchgeführt werden. Baum wird rekursiv durchgesucht, beginnend vom Wurzelknoten & dann linker oder rechter Teilbaum weitergemacht.
- **EINFÜGEN & LÖSCHEN**: muss unter Einhaltung der Sortierungsregel erfolgen.
- **BALANCIERT**: für optimale Leistung werden oft Techniken wie der AVL-Baum oder Rot-Schwarz-Baum verwendet. Dabei wird Erhaltung einer begrenzten Höhendifferenz sichergestellt.

SUCHEN, EINFÜGEN, ENTFERNNEN von Knoten in binären Suchbäumen



- **SUCHE**: Suchmenge wird sukzessive halbiert bis das entsprechende Element gefunden wird oder eine leere Menge zurückgegeben wird. Aufwand: $O(\log(n))$.
- **EINFÜGEN**: Zuerst gesucht, wo das Element theoretisch im Baum existieren würde. An dieser Stelle wird neues Element eingefügt.
- **ENTFERNEN**: Szenario 1: Knoten ist ein Blatt. Dann kann Knoten einfach entfernt werden.
Szenario 2: Knoten hat ein Child. Dann wird Child zu Parent.
Szenario 3: Knoten hat 2 Children. Hier wird gelösches Element mit demjenigen Knoten aus dem rechten Teilbaum ersetzt, der dort am weitesten links steht & somit kleinsten Wert im rechten Teilbaum hat.
Oder umgekehrt: man sucht im linken Teilbaum das Element mit dem größten Wert.

⚠️ In diesem Szenario immer kontrollieren, ob Baum immer noch ausgeglichen & balanciert ist!

WAS IST ein ausgeglichener Baum und wie kriege ich den Zustand hin?

- Der "gewünschte Zustand" in einem Binären Baum.
- Höhendifferenz zwischen am weitesten auseinanderliegenden Niveaus: nicht mehr als 1 sein.
 - ↪ sonst über Rotation herstellen, wenn nicht der Fall!
 - ↪ evtl. ist zweite Rotation notwendig. Nennt sich Doppelrotation.

RECAP ZUSATZINPUT PERFORMANCEMESSUNG

WAS sind grundsÄTZLICHE Herausforderungen bei DER Performancemessung?

- möglichst mit Daten, die auch eigentlich gebraucht werden würden (+ auch ihre Menge!)
- Faustregel: 3 Messungen machen.
just-in-time-compiling
- Es spielen viele Faktoren in Zeitmessung hinein (z.B "KALTSTART", JIT, Garbage Collector, instanziieren von Klassen, etc.)

Wie geht man mit Nebeneffekten um?

- Eine Möglichkeit ist es, die Methoden iterativ aufzurufen, die Zeit zu messen & dann durch die Anzahl der Iterationen zu teilen. Damit mittelt man die Werte, aber man muss sich bewusst sein, dass die iterierende Lösung die Resultate verzerrt kann.
- Ein Messwert kann nie als fester Wert genommen werden!

Wie mache ich einfache, objektive Zeitmessungen in Java?

- Zeit messen vor dem Aufruf der Methode & danach. Dann $(\text{Zeit Davor}) - (\text{Zeit Danach})$ = Laufzeit der Methode.
- über `System.currentTimeMillis()`: Misst die Zeit seit 1970, 1. Januar.
`System.nanoTime()`: Misst die Zeit wann JVM läuft.
↑ ist aber auf jedem Rechner anders! Muss also auf Zielserver/Rechner ausführen
- Mit `System.currentTimeMillis()` ist der Trick: Den zu messenden Ablauf in einer Iteration wiederholen, dessen Gesamtdauer messen & dann diese durch die Anzahl Iterationen dividieren.

RECAP DS - HASHTABELEN

Wie funktionieren Hashtabellen?

- DURCH ein HASHWERT kann ein Objekt eindeutig identifizierbar sein.
- Es gibt keine perfekten Hashes.
- Mit Hash-Werten kann man sehr schnell suchen.
- ermöglichen effizienten Zugriff auf Key-Value-Paare. Schlüssel wird dabei in Index umgewandelt.

Wie implementiere ich Hashtabellen?

1. Hashfunktion definieren: Eine Hashfunktion nimmt einen Schlüssel als Eingabe & berechnet einen Index oder Position in der Hashtabelle an der der Wert gespeichert wird.
2. Speicherplatz reservieren: Eine Hashtabelle besteht aus einem Array oder einer anderen DS. Der Speicherplatz für die "Buckets" wird entsprechend der erwarteten Anzahl von Schlüssel-Wert-Paaren reserviert.
3. Einfügen: Um einen Wert in der Hashtabelle einzufügen, wird der Schlüssel mit Hashfunktion gehasht, um Index zu bestimmen. Wenn Bucket an Position leer → Wert dort gespeichert. Wenn Bucket an Position bereits Wert hat → Kollision. Lösung: verketzte Listen/um mehrere Werte im selben Bucket zu speichern.
4. Suchen: Um Wert für Schlüssel abzurufen, wird Schlüssel erneut mit Hashfunktion gehasht → Index wird erhalten. Wert im Bucket wird dann zurückgegeben. Bei Kollisionen müssen alle Werte im Bucket überprüft werden um gewünschten Wert zu finden.
5. Löschen: Schlüssel mit Hashfunktion gehasht, um Index zu bestimmen. Wert im entsprechenden Bucket wird entfernt. Bei Kollisionen müssen alle Werte im Bucket überprüft werden.

Die Wichtigkeit von guten Hashwerten?

- FUNDAMENTAL für gute & effiziente Implementation Hashtabellen.
- Gute Hashfunktion verteilt Schlüssel gleichmäßig über DS. → bei Schlechten passiert es, dass sich alle Werte am Index 0 sammeln.
- Vermeidung Kollisionen.
- final Values nehmen um Hashwert zu generieren, ansonsten gehen Objekte im Nirvana verloren.

Varianten zur Kollisionsbehandlung bei Hashtabellen?

- mit Tombstones arbeiten. Sie sagen: "Hier war mal was, jetzt ist es leer." → Sondierungsketten nicht unterbrochen.
- eleganter: Linkedlist verwenden bei Hashtkollisionen.

Grundlegende Operationen auf Hashtabellen - Aufwand & Ablauf

- Einfügen: Ablauf: Schlüssel wird gehasht, um Index in Hashtabelle zu bestimmen. Wenn Bucket an Position leer ist, wird Schlüssel-Paar dort eingefügt.

Aufwand: $O(1)$

Zugriff auf Bucket wird durch Hashwert bestimmt → deshalb Insertion effizient. Bei Kollisionen kann lineare Suche erforderlich sein, um leeren Bucket zu finden → erhöht Aufwand.

- SEARCH: Ablauf: Schlüssel wird gehasht, um Index in Hashtabelle zu bestimmen. Wert im entsprechenden Bucket dann geprüft. Bei Kollisionen müssen alle Werte im Bucket geprüft werden, um gewünschten Wert zu finden.

Aufwand: $O(1)$

- Löschen: Ablauf: Schlüssel wird gehasht, um Index in Hashtabelle zu bestimmen. Wert wird im entsprechenden Bucket entfernt.

Aufwand: $O(1)$

$O(1)$ setzt gute & effiziente Hashfunktionen voraus! + gleichmäßige Verteilung Schlüssel.

RECAP NEBENÄUFIGKEIT & THREADS

Vorteile & Nachteile der Nebenläufigkeit?

- Performance kann erhöht werden → man ist effizienter.
- Rechenleistung kann so voll ausgenutzt werden.
- Durch das Auslagern von blockierenden Tasks in Threads, kann das Programm reaktiv gehalten werden.
- Code ist komplexer
- Schwierig zu Debuggen
- Parallel laufende Threads müssen koordiniert werden, vor allem bei Zugriff auf gleiche Ressourcen.

Was ist das Mass für Parallelisierung?

$$\rightarrow \text{Der Speedup. Berechenbar via Formel} \quad S = \frac{T_{seq}}{T_{par}}$$

Was sind die Sichtweisen von Amdahl und Gustafson?

→ **Amdahl:** DASS ein Programm im Ganzen betrachtet werden muss. Der Speedup wird begrenzt vom nicht parallelisierbaren Teil.

$$S(N) = \frac{1}{\frac{P}{N} + (1 - P)} = \frac{1}{(1 - P) + \frac{1}{N}}$$

sequentielle Laufzeit geteilt durch parallele Laufzeit.
↑ Anzahl Prozesse
↑ prozentuale, parallelisierbare Anteil

→ **Gustafson:** DASS die Vergrößerung des zu bearbeitenden Problems im Wesentlichen nur auf den parallelisierbaren Programmteil auswirkt. Die Anwendung ist skalierbar.

$$S(N) = (1 - P) + N \cdot P$$

Lebenszyklus von THREADS

→ Ein THREAD kennt während seiner Verwendung versch. Zustände:

- 1) **NEW** (isAlive == false)
- 2) **RUNNABLE** (isAlive == true). WÄHREND ER IM RUNNABLE-ZUSTAND IST, KANN EIN THREAD AUCH **BLOCKED**, **WAITING** & **TIMED-WAITING** SEIN. AUS WAITING UND TIMED-WAITING KANN MAN IHN DURCH ENTWEDER **SLEEP**, **WAIT** ODER **JOIN** HERAUSHOLEN. AUS BLOCKED NICHT.
- 3) **TERMINATED** (isAlive == false)

2 Arten, einen Java Thread zu implementieren:

1) Ableitung der Klasse Thread:

- ↪ erzeugen eines Objektes "Thread" der abgeleiteten Klasse.
- ↪ **DIESE ART IST ZU VERMEIDEN!**

2) Implementation des Runnable Interface

- ↪ erzeugen eines Objektes der Runnable Klasse.
- ↪ erzeugen eines Thread-Objektes der Klasse Thread.

Wie wird ein Thread in Java beendet?

Allgemein ist ein Thread beendet, wenn...

- Run Methode ist ohne Fehler durchgelaufen
- Run Methode wirft eine Exception
- System.exit wird aufgerufen
- Thread wird aktiv von außen beendet

So beendet man einen Thread aktiv von außen:

- Erzwungen (sollte nicht verwendet werden, ist veraltet)
- Verzögert (ist bei Java irrelevant)
- Kooperativ (heißt Thread wird gebeten zu beenden & muss sich dann selbst beenden mit **interrupt**)
↳ bedarfsgerechtes Nachfragen (if `isAlive == false..`) → mehr Programmieraufwand

Wie beende ich den Thread kooperativ & mit InterruptedException?

Mit interrupt kann man einen Thread kooperativ beenden. Dabei wird ein Interrupted-Flag des Threads gesetzt welche eine Unterbrechungsanforderung signalisiert. Diese Methode löst keine Exception aus. Man kann mit der Methode `isInterrupted` das Flag auslesen und gibt true zurück, falls es gesetzt wurde. Die Methode `interrupted` stellt den Wert des Interrupted-Flag beim aktuellen Thread fest, sie setzt nach dem Aufruf das Interrupted-Flag immer auf false zurück! Da muss man aufpassen.

Eine Interrupted Exception wird geworfen wenn sich der Thread in einer blockierenden Wartemethode befindet und durch interrupt geweckt wird, oder wenn er nicht im Wartemodus ist, aber im weiteren Verlauf auf eine Wartemethode stösst. Diese wird gleich nach dem Betreten wieder verlassen. Diese drei Wartemethoden lösen eine InterruptedException aus:

- Sleep
- Wait
- Join

Wie geht man also mit der Interrupted Exception um? Entweder:

- Exception abfangen und terminieren. Im Catch Teil die Beendung des Threads einleiten, durch `return` oder besser nochmals die Methode `interrupt` aufrufen (denn das Interrupted-Flag wurde zurückgesetzt)
- Oder:
- Exception abfangen und eventuell nicht terminieren, wenn eine abgefangene Arbeitsphase beendet werden soll. Im catch Teil den Thread aufgrund einer Entscheidung beenden.

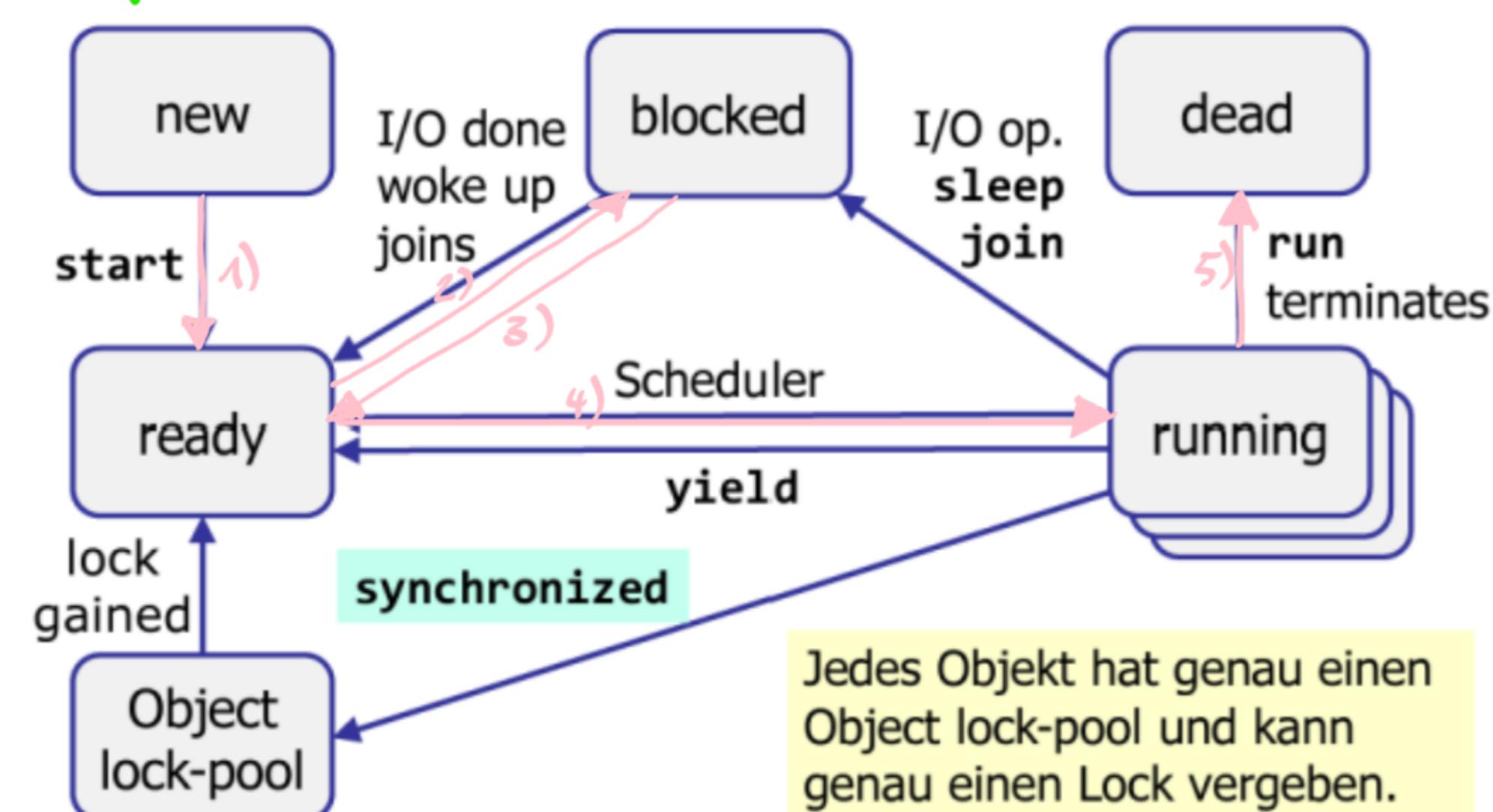
RECAP SYNCHRONISATION

WAS sind die Konzepte auf Zugriff gemeinsame Ressourcen?

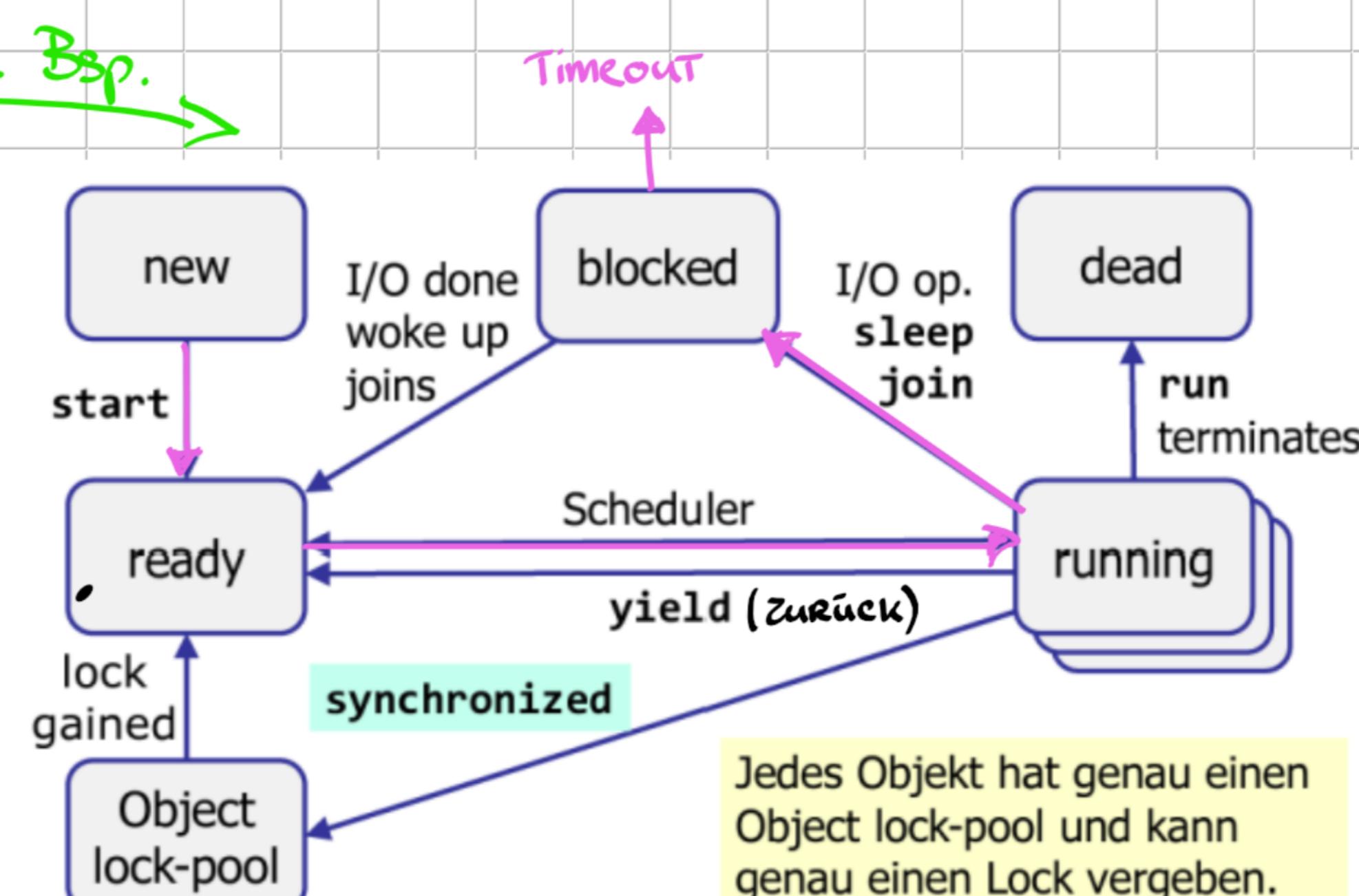
- Es dürfen keine Annahmen über die zugrunde liegende Hardware gemacht werden.
- In einem kritischen Abschnitt darf zu jedem Zeitpunkt nur 1 Thread befinden.
- Thread darf andere Threads nicht blockieren, außer er ist in einem kritischen Bereich.
- Ein Thread darf nicht unendlich warten, bis er in den kritischen Bereich eintreten kann.

WAS IST DER ERWEITERTE Thread Lebenszyklus?

Bsp. 1:



similar Bsp.

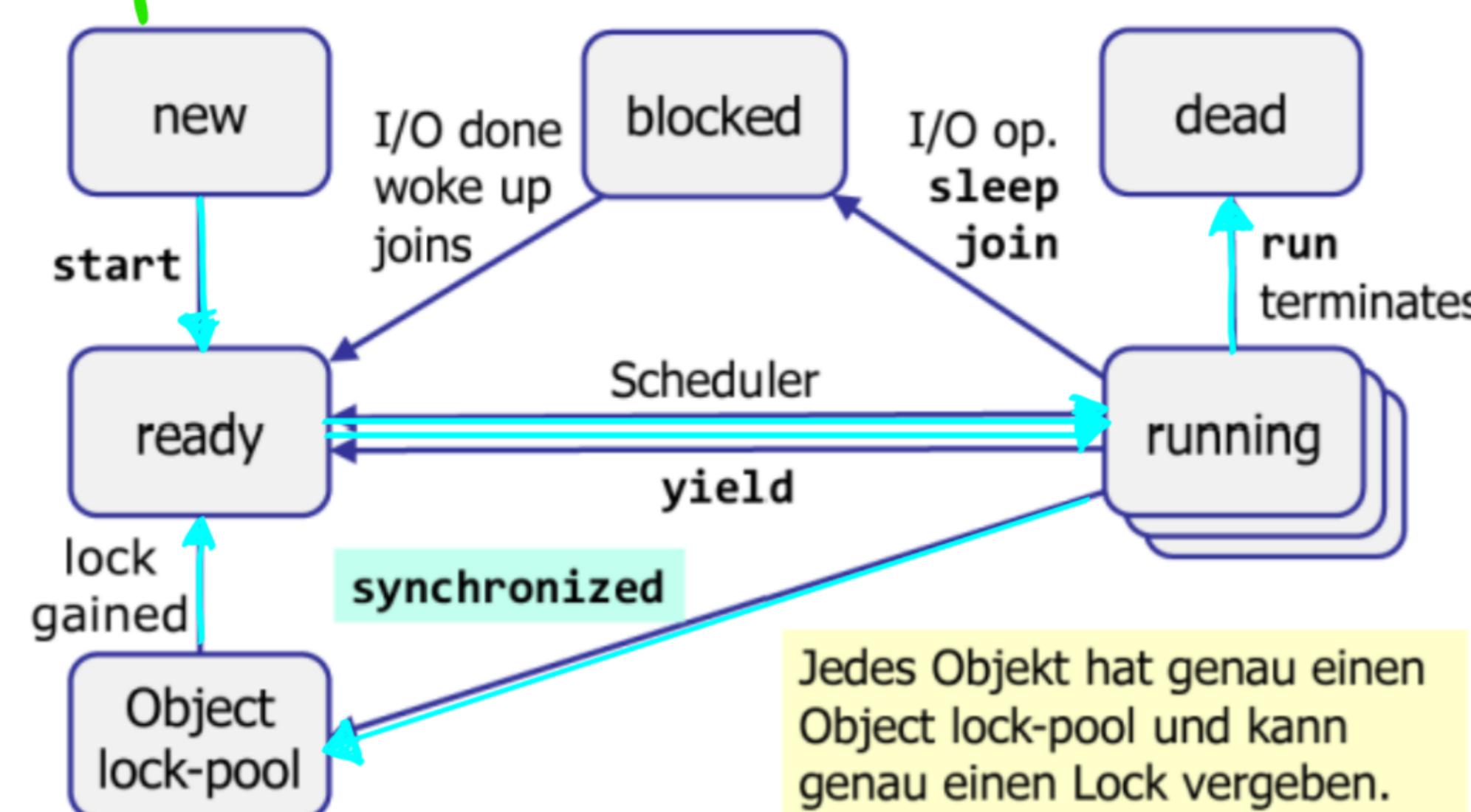


- 1) Thread wird gestartet (**new**)
- 2) Ist im **ready**-Zustand.
- 3) Scheduler kann Thread in **running**-Zustand versetzen.
- 4)

→ 2 Tasks sind ready, starten gemeinsam. Ein Task muss zw. machen, d.h. die Andere muss warten. 2 wird blockiert. Sobald er done ist, können beide weiterlaufen & gemeinsam sterben.

→ almost same Scenario, nur hat hier der Task vergessen, was er tun muss. D.h. Task 2 wartet für immer. Das ist jedoch kein Deadlock!

Bsp. 2:



→ Szenario: 2 Tasks wollen ein Glace kaufen. Jetzt haben sie den Task (**new**) & sind bereit (**ready**) & gehen los (**running**). Sie kommen beim Gefrierfach an. Beide wollen ein Glace, stehen hintereinander an (**Object lock-pool**). Task 1 nimmt ein Glace heraus, macht Kühlung wieder zu (**lock gained**). Dann Task 2 dasselbe, macht Kühlung wieder zu. Beide sind wieder **ready** und dann **running**.

Hier kann ein Deadlock passieren, wenn:

→ Tasks nicht wissen, welches Glace sie wollen. - so stehen sie vor dem Gefrierfach für immer & schauen sich an, weil die eine sich erst entscheiden kann, wenn die Andere sich entschieden hat & umgekehrt. Beide machen nichts für immer.

= Thread kann beliebig viele Locks ergriffen.

WAS IST DAS Monitor-Konzept & was ist Reentrant-Monitore & Nested-Monitore?

→ Monitor-Konzept: wie Raum mit einer Tür & Schlüssel. Nur 1 Thread kann zu einem bestimmten Zeitpunkt den Raum betreten & auf Ressourcen drinnen zugreifen.

→ Reentrant-Monitor: man kann kritischen Raum so oft betreten wie man möchte, solange man den Lock besitzt. Z.B. wenn ein Thread eine Methode betritt, dessen Lock er bereits besitzt, kann er sofort eintreten, ohne den Lock-pool zu durchlaufen.

Wie funktioniert das Monitor-Konzept in JAVA?

- Möglichst wenig Arbeit in synchronisierten Bereichen durchführen. Übermäßige Synchronisierung vermeiden.
- Nested Monitore vermeiden
- Offene Aufrufe verhindern Deadlocks. (Fremde Methoden, außerhalb eines synchronisierten Bereichs)
- Keine verschachtelten Blöcke machen. Dort können einfach Deadlocks entstehen.

RECAP THREADSTEUERUNG

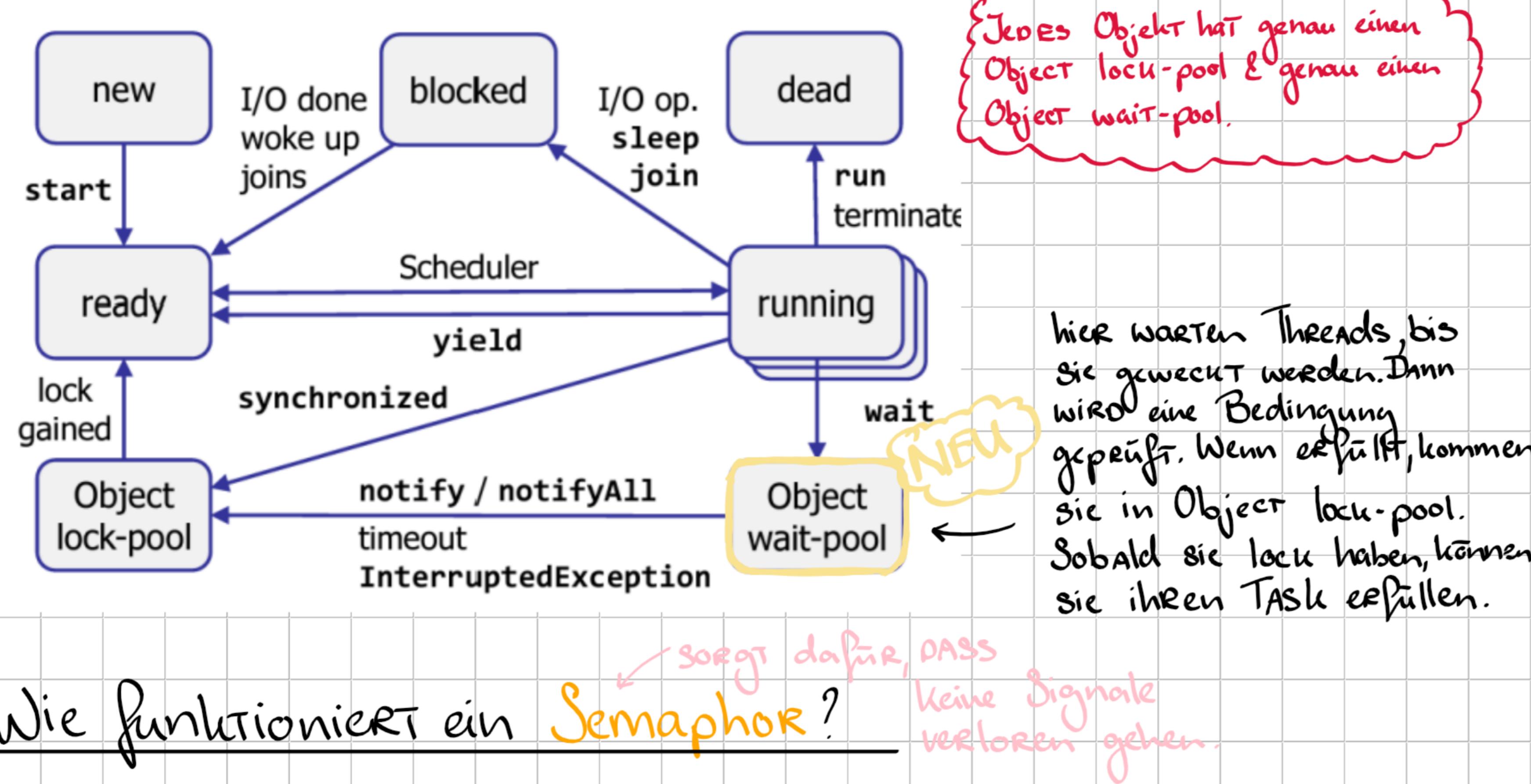
WAS Sind GUARDED Blocks?

- Zugang zu einem kritischen Abschnitt hängt von Bedingung oder Zustand ab.
- Konkret: Threads warten an einem Monitorobjekt auf Signal, dass er Thread(s) freischalten darf.
- GUARDED Block prüft Bedingung (muss true sein) bevor der Thread im Block fortfahren kann.

Wie funktioniert das nebenläufige WARTEN auf Bedingungen in Java?

- Bedingungen mit Iteration (`while()`) implementieren.
- Kann sein, dass Thread die Bedingung zuerst nicht erfüllt, aber dies nebenläufig ändert, wenn Thread erneut aufgerufen.

WAS IST DER VOLLSTÄNDIGE Lebenszyklus für JAVA Threads?



Wie funktioniert ein SEMAPHOR?

- kommt von der Bahn: ein mechanischer "Passier-Zähler". Kennt 2 Werte: P (passieren) und V (erhöhen).
- In Programmierung ist er ein Zähler & gibt an wieviele Ressourcen gerade verfügbar sind.
- Bsp: Thread benötigt eine Ressource. Er ruft `'acquire'` (passieren)-Operation auf. Wenn Anzahl Ressourcen > 0, dann wird Thread eine Ressource zugewiesen. Der Zähler wird um 1 verringert. Wenn 0, dann wird Thread blockiert & wartet, bis Ressource freigegeben wird. Wenn Thread Ressource nicht mehr benötigt, ruft er `"release"`-Operation auf. Zähler wird um 1 erhöht & Ressource steht anderen Threads zur Verfügung.

WAS SIND DIE REGELN FÜR WAIT, NOTIFY, NOTIFY ALL?

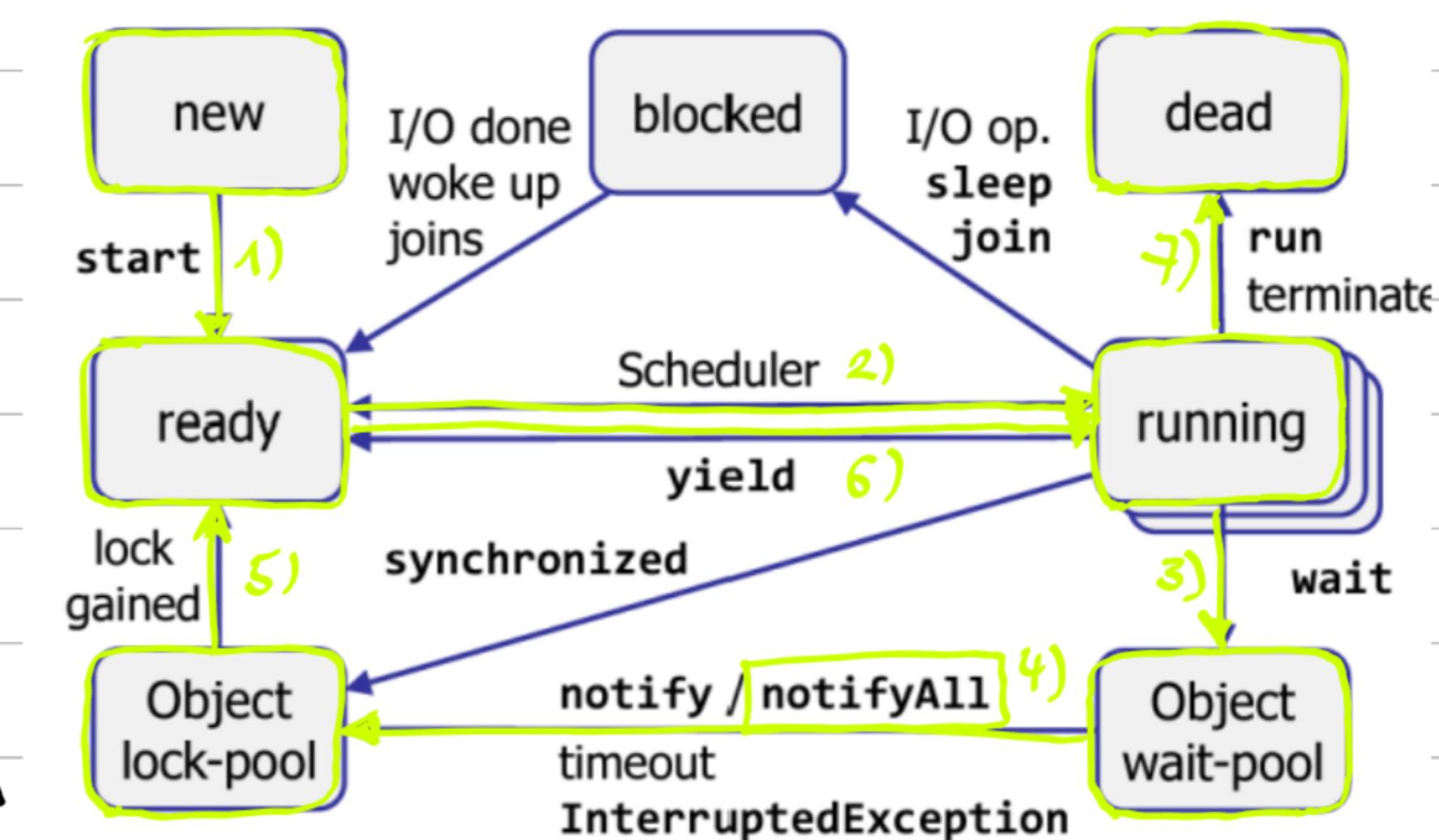
→ WAIT

- Aufruf sollte in `while()`-Schleife gesetzten, die eine bestimmte Bedingung prüft.

→ NOTIFY

- WECUT immer nur einen, random ausgewählten Thread aus dem Object-Wait-pool. Wenn Thread nicht auf Bedingungen passt, dann passiert nichts ⇒ Deadlock!
- auch das Notify-Signal wird nicht gespeichert (ist kein Thread da, geht der Aufruf verloren)

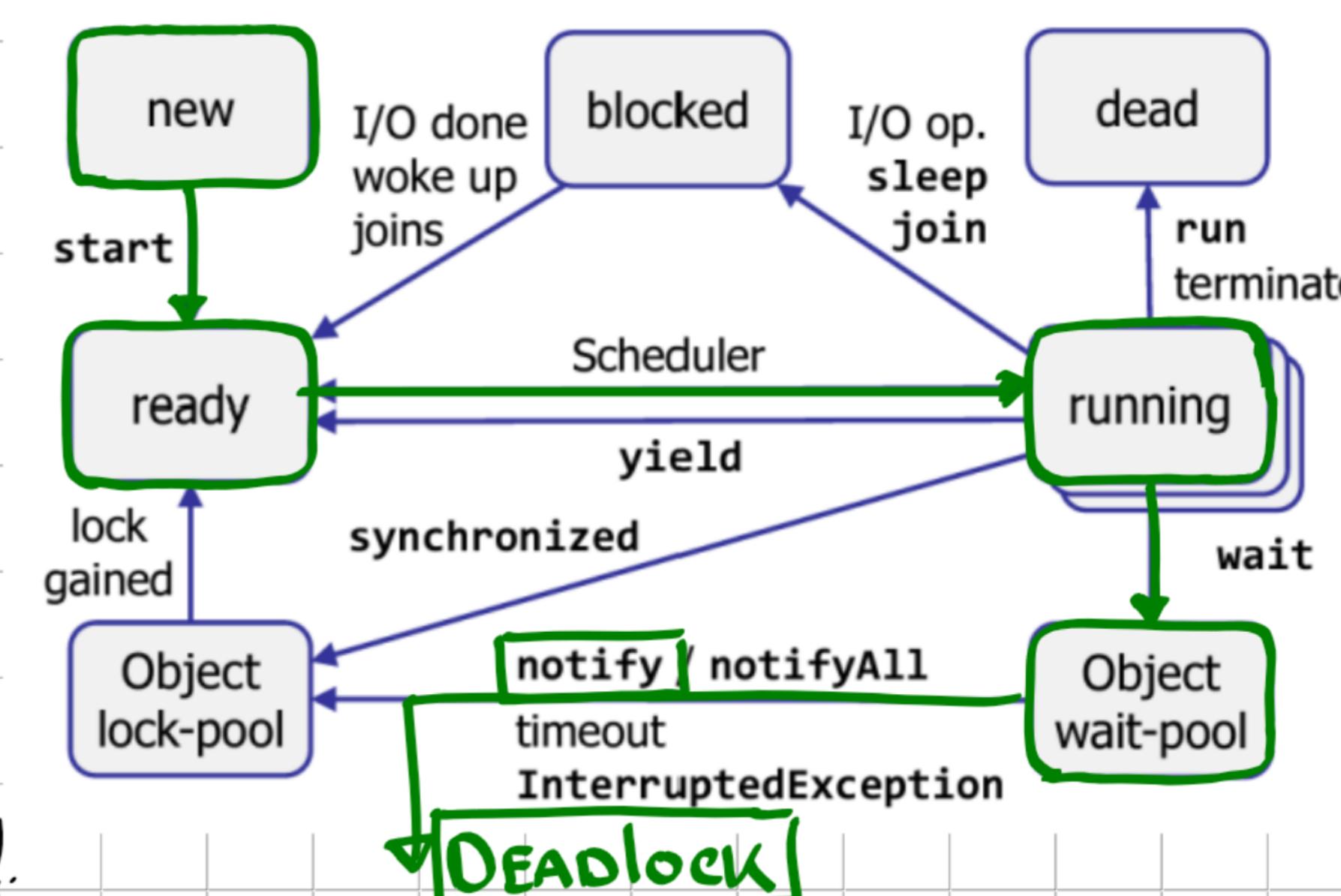
Bsp. 1:



Bsp. 1:

Wir haben wieder das Szenario mit Glace holen aus der Cafeteria. Wir haben gehen also zur Kühltruhe und stehen da vorne dran. (Object wait-pool) Leider können wir nicht direkt ein Glace holen, der Mensch hinter der Theke muss zuerst noch Glace in die Truhe legen. Zuerst die Wasserglace, er fragt in die Runde ob jemand eines haben möchte. Ich möchte ein Rahmglace also gehe ich zur Truhe und stelle mich vorne dran hin (Object lock-pool). Ich kann die Truhe nun öffnen und eines herausnehmen. Erst dann füllt der Glacemann die Rahmglace in den Kühler und er fragt erneut: «wer möchte ein Rahmglace» (notifyAll). Jetzt meldet sich meine Freundin, stellt sich vor den Kühler (Object lock-pool) und nimmt eines heraus. Wir haben beide ein Eis, gehen zurück und sterben.

Bsp. 2:



Bsp. 2:

Gleiches Beispiel, doch diesmal füllt der Glacemann zuerst die Rahmglace in den Kühler. Er fragt aber nicht in die Runde, ob jemand ein Rahmglace will sondern er fragt per Zufall einen Schüler, sagen wir das wäre ich (notify!). Aber ich möchte ein Wasserglace, die «Condition» passt also nicht und wir befinden uns in einem deadlock. Ich möchte kein Rahmglace aber er hat mich gefragt. Meine Kollegin die ein Rahmglace möchte steht genau wie ich den ganzen Tag vor der Truhe und wir machen gar nichts mehr. Sie wartet darauf, dass er fragt wer ein Rahmglace will, der Glacemensch wird nicht mehr fragen und ich bin enttäuscht dass ich mein Wasserglace nicht erhalten habe.

→ notifyAll

- weckt alle Threads im Object-wait-pool.
- Signal wird nicht gespeichert (ist kein Thread da, geht Aufruf verloren)
- empfohlene Methode, auch wenn Performance einen Einbruch geben kann.

RECAP THREAD POOLS

Bei welcher Art von Threads starte ich neu / verwende ich wieder?

- In Java kann man gestorbenen Thread nicht wieder neu starten...
- Bei mehreren Threads kann ein Threadpool helfen.

WAS sind **Nachteile** wenn man Threads einzeln STARTET?

- NICHT praxisgerecht: man arbeitet eher mit Thread Pools & selten mit Thread Objekten.
- viele, einzelne Threads belasten das Betriebssystem, da sie eher nicht nebenläufig ausgeführt werden können.

Wie trennt man das Konzept der Nebenläufigkeit von der technischen Realisation?

- Aufgabe bspw. man muss 1000 Aufgaben nebenläufig lösen.
 - ↪ THEORETISCH: 1000 Threads machen, fertig.
 - ↪ BESSER: Thread-Pool zur Hilfe nehmen. Dann ist es zwar nicht mehr 100% nebenläufig, aber korrekt.

WAS ist ein **Thread Pool**?

- Konzept: Threads in viele weitere kleine Tasks aufteilen & dann einzelne Tasks nebenläufig (= zur gleichen Zeit) erledigt.
 - ↪ massenhaft, chunky Threads, die ein nach dem anderen abgearbeitet werden müssen, belasten das OS.
- Es macht Sinn, die Menge der erzeugten Threads zu beschränken.
- Es gibt einen "Task-Submitter" Der Tasks erzeugt. Diese Tasks werden dann mit Executoren in eine Task Queue ausgetilgt. Am anderen Ende der Queue warten Threads in einem Pool mit Einstellung: "gibt es einen Task für mich?". Wenn ein Task kommt, dann schnappt sich Thread den ersten Task, führt ihn aus & holt sich nächsten Task.

Wie setze ich die **EXECUTORS-KLASSE** zur Erzeugung von Thread Pools adequat ein?

- Man könnte eigene Threadpools machen, muss man aber nicht.
- **EXECUTORS-CLASS** bietet Factory-Methoden an zur Erzeugung diverser Threadpools.
 - ↪ **ThreadPoolExecutor** (flexibler Pool zur parallelen Ausführung von Aufgaben)
 - ↪ ermöglicht Wiederverwendung von Threads (spart Zeit → keine Erstellung neuer Threads)
 - ↪ verwaltet Anzahl Threads im Pool & deren Lebenszyklen
 - ↪ führt Aufgaben in Queues aus
 - ↪ **ScheduledThreadPoolExecutor** (erweiterte Variante)
 - ↪ ermöglicht Ausführung von Aufgaben in bestimmten Zeitpunkten / Intervallen.
 - ↪ Aufgaben lassen sich mit Verzögerung STARTEN oder periodisch wiederholen.
 - ↪ **ForkJoinPool** (speziell für Verarbeitung rekursiver Aufgaben)
 - ↪ für Aufgaben, die in kleinere Teilaufgaben (forks) aufgeteilt sind und anschließend wieder zusammengeführt (joins) werden können.
 - ↪ ermöglicht effiziente Nutzung verfügbare Prozessorkerne (indem Arbeit zwischen Threads aufgeteilt & Ergebnisse wieder zusammengeführt.)

Wie schätzt man die Größe eines Thread Pools?

$$N_{\text{threads}} = N_{\text{cpu}} \cdot U_{\text{cpu}} \cdot \left(1 + \frac{W}{C}\right)$$

Auslastung CPU, $0 < U_{\text{cpu}} < 1$

Anzahl der zur Verfügung stehende Kerne
↳ `Runtime.getRuntime().availableProcessors() + 1`

Verhältnis zwischen Warte & Rechenzeit

Faustregel

⚠ Für rechenintensive Tasks & Vollauslastung diese Formel wählen

$$N_{\text{threads}} = N_{\text{cpu}} + 1$$

RECAP WEITERFÜHRENDE KONZEPTE

WAS sind ExecutorServices, Callable<V> und Future<V> + ihre Problematiken?

ExecutorService: - **Was:** Schnittstelle für Ausführung & Verwaltung von Threads
↳ stellt einen Pool aus Threads bereit. Somit muss man selbst keine Threads mehr erstellen/verwalten.
↳ Aufgaben & Runnable-Objekte werden ausgeführt

- **Tasks:** | 1) Automatische Erstellung & Verwaltung von Threads
2) Einreichen von Aufgaben: es HAT Methoden submit() & execute(). Aufgabe kann Runnable-Objekt sein
3) Gleichzeitige Ausführung von Aufgaben: mehr Performance durch Parallelität
4) Rückgabe von Future-Objekten. Bietet: Ergebnis abrufen, auf Ergebnis warten, Status der Aufgabe abfragen.
5) Thread-Pool Verwaltung: verwaltet, erstellt, entfernt den Thread-Pool. Kann die Anzahl Threads dynamisch anpassen.

Callable<T>

- **Was:** Substitut zum Runnable Interface. Callable<V> ist ein Interface, das Rückgabewert HAT & falls etwas in der Methode call() schief geht, liefert es eine Exception.

- **Tasks:** | 1) T call(): einzige Methode im Interface. Verantwortlich für Ausführung der Aufgabe & gibt Rückgabewert vom Typ T zurück. Bei Fehler: Exception.

Future<T>

- **Was:** Interface, das Handhabung mit asynchronen Berechnungen vereinheitlicht & vereinfacht. Bei Aufruf einer Methode die zum Zeitpunkt noch kein Ergebnis HAT, dann kann mit Future gearbeitet werden.

- **Tasks:** | 1) boolean isDone(): überprüft ob Aufgabe abgeschlossen & Resultat verfügbar.
2) boolean isCancelled(): ist Aufgabe abgebrochen?
3) boolean cancel(boolean mayInterruptIfRunning): bringt Aufgabe ab, wenn noch nicht abgeschlossen.
4) T get(): blockiert aufrufenden Thread, bis Ergebnis verfügbar ist & gibt Ergebnis zurück. Wenn Aufgabe abgebrochen → CancelledException ausgelöst.
5) T get(long timeout, TimeUnit unit): blockiert aufrufenden Thread, bis Ergebnis verfügbar ist & gibt Ergebnis zurück. Wenn Ergebnis innerhalb angegebenen Zeitraum nicht hier → TimeoutException

Wie handelt man Exceptions aus nebenläufig ausgeführten Tasks?

Bei execute: Um eine ArithmeticException zu werfen bei Division durch 0, senden wir den Task so an den Pool:

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

Bei submit/get: Bei submit wird nicht direkt ein Fehler geworfen. Erst bei get wird die Exception auf das zurückgegebene Future Objekt ausgelöst.

```
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

Wie erstelle & modifiziere ich Thread-sichere Zugriffe auf Atomic Variablen von elementaren Datentypen?

Atomic Variablen sind spezielle Variablen, welche nebenläufig verwendet werden. ermöglichen Variablen in mehreren Threads zu verwenden, ohne Angst, dass andere Threads den Wert zur gleichen Zeit ändern können.

In Java gibt es `java.util.concurrent.atomic.Atomic` dafür. Sie bietet folgende Implementierungen: `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, `AtomicReference`.

↳ z.B. um eine atomare Variable zu erstellen: `AtomicInteger counter = new AtomicInteger(0);`

Folgende Methoden lassen sich auf atomic Variablen anwenden:

↳ `int addAndGet(int delta)`

↳ Wert wird um Delta erhöht & Neues zurückgegeben.

↳ `int decrementAndGet()`

↳ Wert wird dekrementiert & Neues zurückgegeben.

↳ `int incrementAndGet()`

↳ Wert wird inkrementiert & Neues zurückgegeben.

↳ `int get(int newValue)`

↳ Wert wird durch newValue ersetzt & zurückgegeben.

↳ `int get()`

↳ Gettermethode

Was sind Wrapper-Klassen?

Sind da, um Datenstrukturen Thread-sicher zu gestalten. Für jedes Collection Interface steht eine öffentliche Klassenmethode von Collections zur Verfügung, die eine entsprechende synchronisierte Containerpassade zurückliefert. Konkreter so verwenden:

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

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

Hintereinander ausgeführte geschützte Methoden auf solchen Containern sind nicht fehlerfrei, da bei jedem Aufruf ein Taskwechsel passiert. → Iterationen über DS sind nicht atomic! Deshalb muss vor der Iteration der Container selbst geschützt werden. Dafür gibt es 2 Ansätze:

ITERATOR-SCHNITTSTELLE: Anstatt eine for-Schleife mit KeySet oder Map zu verwenden, verwendet man einen Iterator der von Map zurückgegeben wird. Der Iterator ist in der Lage, Änderungen während der Iteration zu erkennen & reagiert entsprechend.

```
Iterator<BankAccount> iterator = syncList.iterator();
while (iterator.hasNext()) {
    // tut was mit iterator.next()
}
```

Hier ist, bei nebenläufige Veränderung, eine Exception möglich.

THREAD-SICHERE ITERATION: Vor der Iteration den Container schützen. Dadurch wird jedoch die Nebentäufigkeit stark eingeschränkt, da der Container für alle anderen Zugriffe gesperrt wird.

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

Was ist eine Blocking Queue?

↳ `ArrayBlockingQueue<E>`
↳ `LinkedBlockingQueue<E>`
↳ `DelayQueue<E>`
....

Eine DS, die zum Austausch von Daten zwischen Threads verwendet wird. Elemente können normal eingefügt & entfernt werden. Der Unterschied zu einer normalen Queue besteht darin, dass die Operationen auf dieser DS blockieren, wenn die Queue voll oder leer ist.

Beispiel:

Nehmen wir ein Alltagsbeispiel: Es stehen während Corona Leute vor dem Einkaufsladen in einer Schlange. Die Queue (Schlange) hat eine begrenzte Kapazität, es können nur 5 Leute hintereinander stehen, sonst würde sie bis auf die Straße reichen. Die Menschen stellen dabei die Elemente in der Queue dar. Wenn die Schlange voll ist, blockieren weitere Menschen, die sich anstellen möchten und warten, bis ein Platz frei wird. Wenn ein Mensch den Laden betritt, rücken die Menschen hinter dieser Person nach vorne, um den Platz zu füllen und hinten kann sich wieder jemand neues anstellen. Wenn die Schlange leer ist, bittet der Türsteher auch keine Leute hinein, er ist blockiert. Bzw. Der Aufruf `queue.take()` blockiert, wenn die Queue leer ist bis ein Element verfügbar ist.

- `ArrayBlockingQueue<E>`
 - o Queue mit fester Größe
- `LinkedBlockingQueue<E>`
 - o Existiert als gröszenbeschränkt aber auch unbeschränkt.
- `DelayQueue<E>`
 - o Kann nur Objekte aufnehmen die das Interface Delay implementieren. Die Werte müssen «reifen» bevor sie entfernt werden können, sie werden erst nach einer bestimmten Verzögerungszeit entnommen
- `PriorityBlockingQueue<E>`
 - o Sortiert mit Hilfe von `compareTo` mit dem explizit angegebenen Comparator Objekt die Elemente
- `SynchronousQueue<E>`
 - o Blockierende Queue bei welcher die beteiligten Threads aufeinander warten müssen. Sie hat keine Kapazität bzw. sie dient als Punkt der Synchronisation zwischen Produzenten und Konsumenten, wobei ein Produzent ein Element direkt an den Konsumenten (Thread) übergibt.

Methoden auf der Blocking Queue:

• `boolean offer(E e);`

↳ versucht das Element am Ende der Queue einzufügen. True, wenn erfolgreich. False, wenn kein Platz.

↳ wenn `boolean offer(E e, long timeout, TimeUnit unit)`; → gibt false zurück, wenn Zeit abgelaufen

• `E poll();`

↳ entfernt & gibt das erste Element aus der Queue zurück. Wenn leer ist: gibt 0 zurück.

↳ auch mit Timeout verfügbar.

• `void put(E e);`

↳ fügt Element am Ende der Queue hinzu. Wenn Queue voll ist, blockiert Methode bis Platz frei wird.

• `E take();`

↳ entfernt & gibt erste Element aus Queue zurück. Wenn Q. leer ist, blockiert Methode bis Element da ist.

RECAP GRUNDLAGEN SORTIEREN

Welche Voraussetzungen braucht es, um Datenelemente zu sortieren?

Um 2 Elemente miteinander vergleichen zu können, muss man diese Interfaces implementieren:

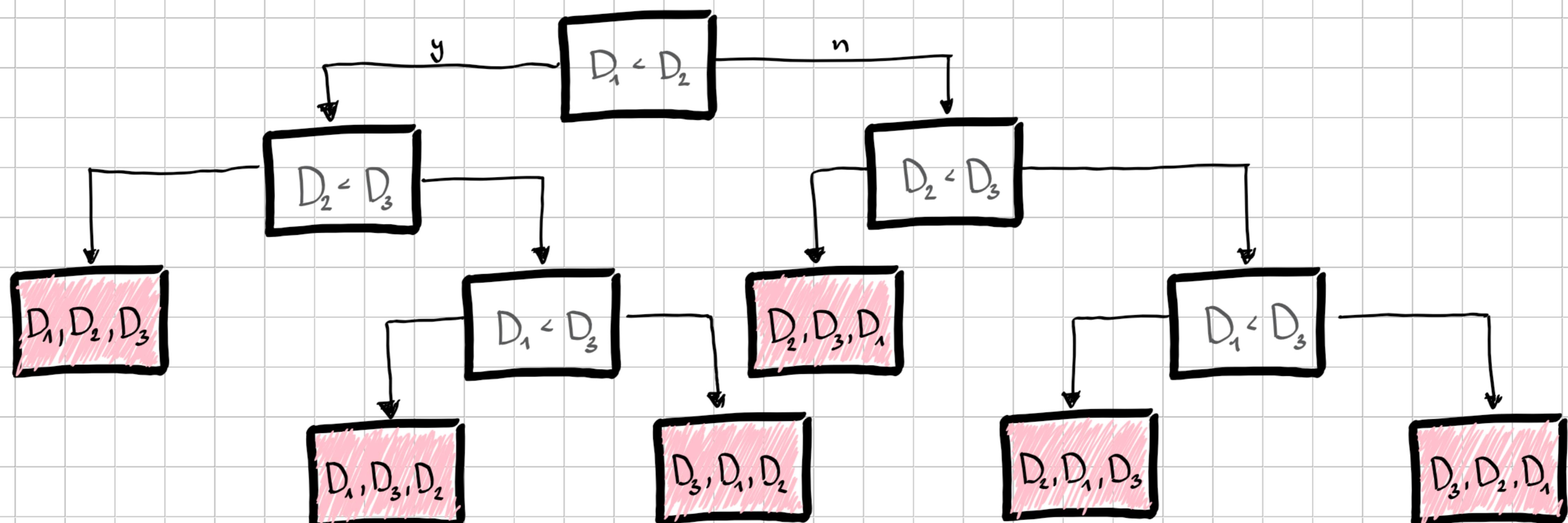
- Comparable <T> mit int compareTo (T o) für die natürliche Ordnung. Damit wird auf einen einzigartigen Wert sortiert. z.B. bei der Klasse "Person" die AHN-Nr., für eine DS mit Integers wäre es eine Zahl.
 - Comparator <T> mit int compare (T o1, T o2) für die spezielle Ordnung. Man kann dann 2 Werte sortieren lassen, z.B. bei der Klasse "Person" nach AHN-Nr. oder Schuhgrösse.
- + nicht vergessen hashCode zu überschreiben! Damit Objekte sortiert werden können, müssen sie vergleichbar sein.

Welche Zeitkomplexität haben Vergleichsbasierte & Radix-Sortierverfahren?

Radix-Sortierverfahren sind nicht-vergleichsbasiert. Sie sortieren Elemente basierend auf Ziffern/Zieichen. → gut geeignet für Zeichenketten und ganze Zahlen. Die Zeitkomplexität liegt bei $O(k \cdot n)$, wobei k = Anzahl Stellen in der längsten Zahl/Zeichenkette
n = Anzahl Elemente in der Liste.

Vergleichsbasierte Sortierverfahren vergleichen Elemente paarweise & sortieren in der richtigen Reihenfolge. → gut geeignet für jedes Element, dass mit einem Vergleichsoperator sortiert werden kann. Die Zeitkomplexität ist zwischen $O(n \cdot \log(n))$ und $O(n^2)$.

Wie sieht ein einfacher Entscheidungsbaum aus? ← (bei 3 Elementen)



Was ist der Unterschied zwischen internem und externem Sortieren?

Internes Sortieren = Daten liegen direkt im Arbeitsspeicher. Manche Algorithmen können nur intern sortieren → z.B. sortieren von Array.

Externes Sortieren = Daten liegen in einem externen Massenspeicher. Es ist nur Lesen & Schreiben möglich, kein direktes Vergleichen. (internes Speicher wäre für Daten zu klein) → z.B. sortieren von sequentiellen Files/Dateien.

Wie differenziert man Zeitkomplexität praktisch?

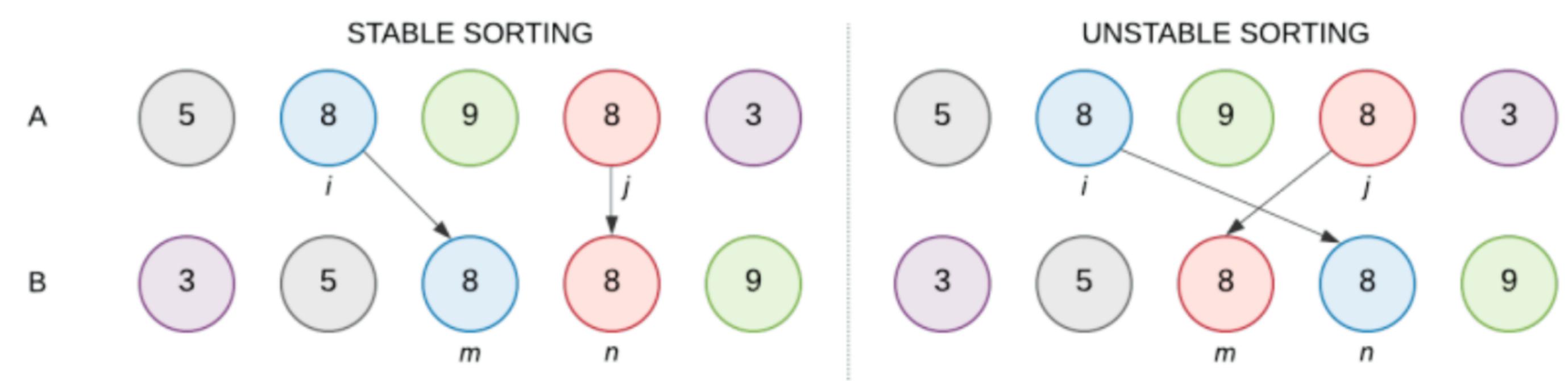
Nach Best-, Average- und Worst-Case (letzteres von besonderer Bedeutung, weil man wissen möchte, wie lange es dauert, wenn das System tatsächlich einmal ausgelastet ist mit "grossen n").

WAS GARANTIERT ein stabiler Sortieralgorithmus?

→ er behält die relevante Reihenfolge von Daten bei. Beispiel: Array mit 5 Werten wird sortiert & die Zahl 8 kommt 2x vor:

| STABILE ALGORITHMEN: |

- Merge SORT
- Insertion SORT
- Bubble SORT
- Counting SORT
- RADIX SORT



| INSTABILE ALGORITHMEN: |

- Selection SORT
- Quick SORT
- Heap SORT
- Shell SORT

RECAP EINFACHE SORTIERALGORITHMEN

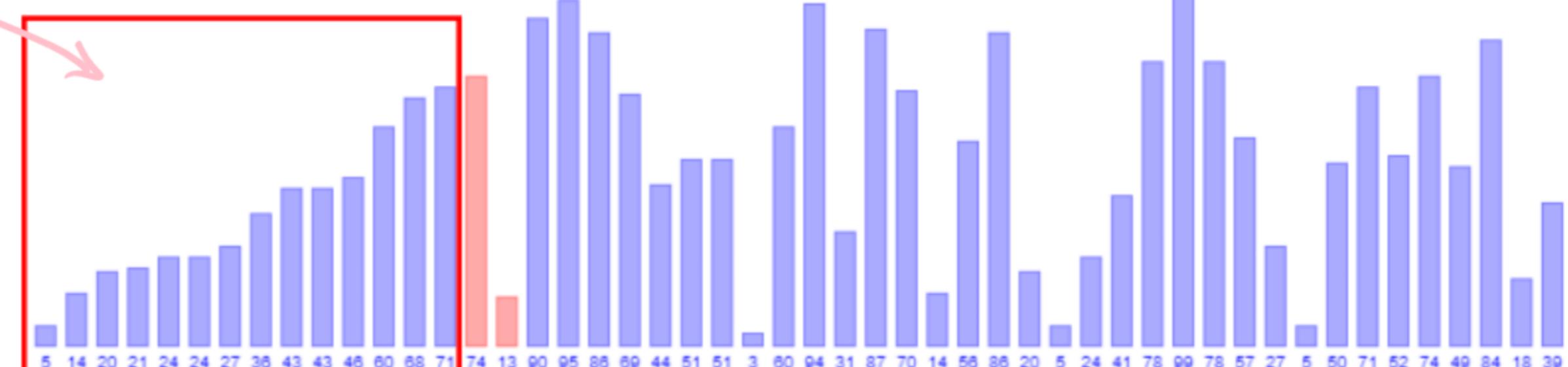
Wie zeige ich einfache Sortieralgorithmen für kleine DM auf Papier?

Insertion Sort:

- 1) Zum Anfang des Arrays gehen.
- 2) Den nächsten, nachfolgenden Wert ansetzen. Ist er kleiner? → vorne einfügen.
Ist er grösser? → hinten einfügen.

Stabilität: stabil
Komplexität: $O(n^2)$
Eigenschaften: vergleichsbasiert
für kleine Listen oder
(bereits) teilweise sortierte Listen

hier gilt der Rote Teil als sortiertes Array.
Alle nachfolgenden Zahlen werden darin sortiert.



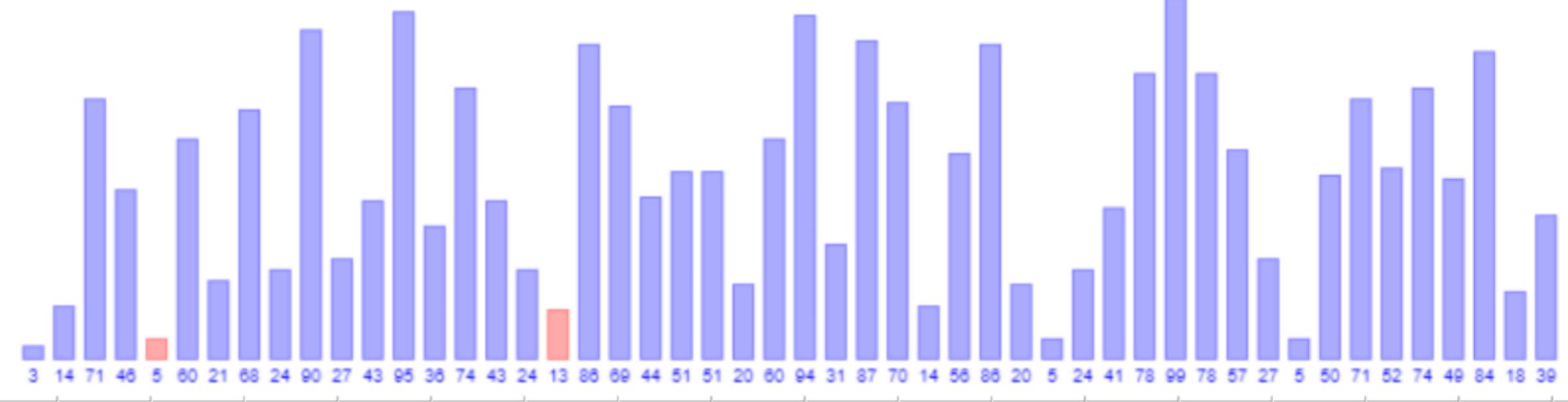
Selection Sort:

- 1) Beginn beim Anfang des Arrays. Dies ist der Wert 0.
- 2) Vergleiche nachfolgende Werte mit Wert auf 0. Ist er kleiner, so wird er an den Anfang gesetzt & ist die neue 0.

Stabilität: instabil

Komplexität: $O(n^2)$

Eigenschaften: vergleichsbasiert, einfache Implementierung
ineffizient für grosse Listen



Bubble Sort:

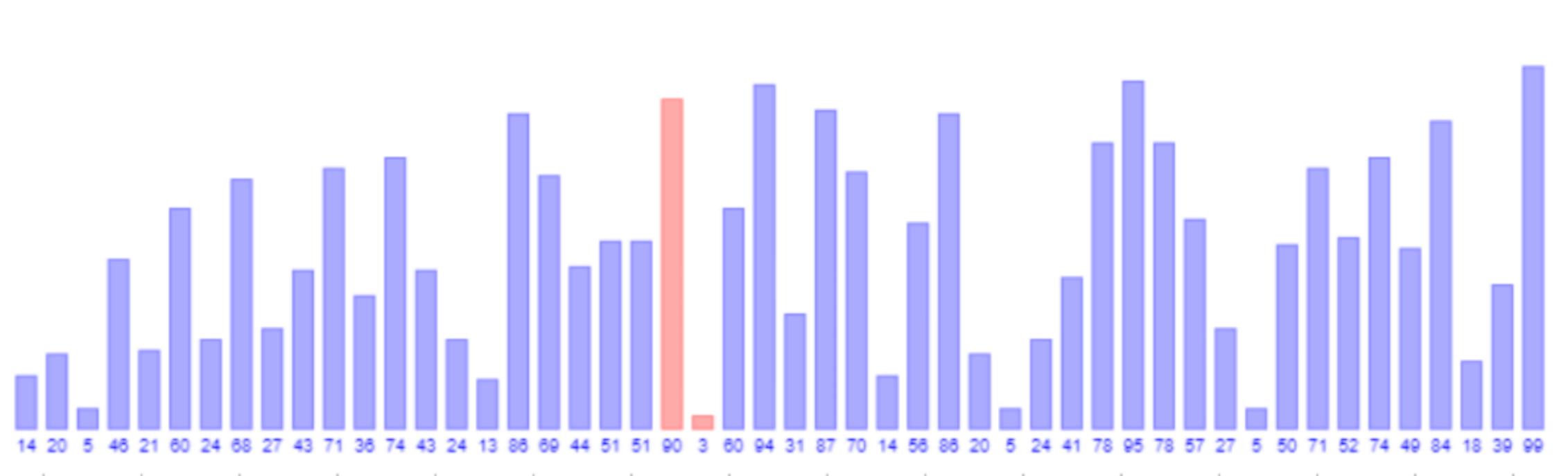
- 1) Zum Anfang des Arrays gehen & Wert auf 0 und 1 vergleichen (es werden immer 2 benachbarter Werte miteinander verglichen).
- 2) Ist der linke Wert > rechte Wert, werden Plätze getauscht. Sonst nicht.
- 3) Es geht weiter mit nächstem Wertepaar. Die höchsten Werte "blubben" am Ende des Arrays.

Stabilität: stabil

Komplexität: $O(n^2)$

Eigenschaften: vergleichsbasiert, einfache Implementierung
ineffizient für grosse Listen

Insertion Sort in smaller fragments



Shell Sort:

- 1) Array (virtuell) in kleinere Arrays aufteilen & innerhalb sortieren.
- 2) Ganzes Array wieder in weniger kleine Arrays aufteilen. Erneut sortieren.
- 3) Zum Schluss Werte in 1er Schritten sortiert.

Stabilität: instabil

Komplexität: $O(n^2)$ oder besser je nach Sequenz

Eigenschaften: vergleichsbasiert, verbessert Laufzeit von Insertion Sort.

RECAP HÖHERE SORTIERALGORITHMEN

Wie spiele ich konkret einen höheren Sortieralgo an einem Beispiel durch?

- 1) Trennelement wählen. Hier der letzte Eintrag im Array.
- 2) Array von vorne nach hinten durchtesten. Welche Elemente sind grösser als das Trennelement, welche kleiner?
- 3) Trennelement in die Struktur einordnen → sein finaler Platz.
- 4) 2 neue Trennelemente bestimmen, wiederholen.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	12	10	52_1	9	77	23	18	52_2	11	25	8	5	17				Trennelement	
2																	kleiner	
3	12	10	5	9	8	11	18	52_2	23	25	77	52_1	17				grösser	
4																	neutral	
5	12	10	5	9	8	11	17	18	52_2	23	25	77	52_1					
6																		
7	12	10	5	9	8	11	17	18	52_2	23	25	77	52_1					
8																		
9	8	10	5	9	12	11	17	18	25	23	52_2	77	52_1					
10																		
11	8	10	5	9	11	12	17	18	25	23	52_1	52_2	77					
12																		
13	8	10	5	9	11	12	17	18	25	23	52_1	52_2	77					
14																		
15	8	5	10	9	11	12	17	18	25	23	52_1	52_2	77					
16																		
17	8	5	9	10	11	12	17	18	23	25	52_1	52_2	77					
18																		
19	8	5		9	10	11	12	17	18	23	25	52_1	52_2	77				
20																		
21	5	8		9	10	11	12	17	18	23	25	52_1	52_2	77				
22																		
23	5	8	9	10	11	12	17	18	23	25	52_1	52_2	77				sortiertes Array	

Was sind die Laufzeitkomplexitäten der höheren Sortieralgs?

Anzahl sortierende Elemente in der Liste

Quicksort:

→ Average CASE: $O(n \cdot \log(n))$ ⇒ weil instabil
Worst CASE: $O(n^2)$

↳ tritt auf, wenn das Trennelement in jeder Teilung immer das Größte/Kleinste ist.

Quicksort-optimiert:

→ "

Mergesort:

→ Average CASE: $O(n \cdot \log(n))$ ⇒ weil stabil

Worst CASE:

Speicherkomplexität: $O(n)$

Heapsort:

→ Average CASE: $O(n \cdot \log(n))$ ⇒ weil stabil

Worst CASE:

Speicherkomplexität:

→ getMAX(): $O(n)$ ⇒ weil größtes Element ist die Wurzel
↳ Sinkprozess nach getMAX(): $O(\log_2(n))$

→ insert(): einfügen $O(n)$ aber Steigprozess ist $O(\log_2(n))$, also ist Komplexität $O(\log_2(n))$.

Was sind die Merkmale der behandelten höheren Sortieralgs?

Alle 3 Algorithmen (Quicksort, Mergesort, Heapsort) arbeiten generell mit einer Zeitkomplexität von $O(n \cdot \log(n))$.

Quicksort arbeitet instabil.

Mergesort & Heapsort arbeiten stabil.

"REDUZIERE und Herrsche" und "Teile und Herrsche" an Sortieralgs als Lösungsprinzipien?

"Teile und Herrsche" ist ein Prinzip für Quicksort, indem ein komplexes Problem in mehrere Teilprobleme zerlegt wird.

"Reduziere und Herrsche" erzeugt hingegen durch wiederholtes Anwenden auf kleinere Probleme eine Lösung.

Wie implementiere ich eine Heap-Datenstruktur?

Man muss ein paar wichtige Eigenschaften berücksichtigen. Ein Heap ist ein binärer Baum, der:

- > voll ist UND (strukturelle Bedingung)
- > jeder innere Knoten ist größer oder gleich gross wie seine Kinder (inhaltliche Bedingung)
- > kann in einem Array abgespeichert werden.

Was bietet die Java Klassenbibliothek betreffend Sortieren?

-> Arrays sortieren: Klasse `java.util.Arrays`

- ↳ `STATIC void sort(int[] a)`
- ↳ `STATIC void sort(Object[] a)`
- ↳ `STATIC <T> void sort(T[] a Comparator<? super T>c)`
- ↳ `STATIC void parallelSort(int[] a)`

-> Listen sortieren: Klasse `java.util.Collections`

- ↳ `STATIC <T extends Comparable<? super T>> void sort(List<T> list)`
- ↳ `STATIC <T> void sort(List<T> list, Comparator<? super T>c)`
- ↳ `STATIC void reverse(List<?> list)`

RECAP PARALLELISIERUNGSFRAWORKS

WAS IST DAS GRUNDPRINZIP DES FORK-JOIN-KONZEPTS?

Es ist das Aufrufen (Fork) & Zusammenführen (Join) von Aufgaben in einem Thread. Ein Parent-Thread ruft dabei ein Child-Thread auf, welches eine Aufgabe des Parent übernimmt & löst. Der Parent führt auch weiter fort. Sobald der Child-Thread seine Aufgabe gelöst hat, gibt er die Resultate an den Parent zurück.

EINIGE WICHTIGE PUNKTE ZUM FORK-JOIN FRAMEWORK IN JAVA?

- Das Framework basiert auf "forking" & "joining", d.h. eine grosse Aufgabe wird in kleinere Aufgaben aufgeteilt, die dann wieder parallel ausgeführt werden.
- ENTHALT die Klasse "ForkJoinPool", die eine Gruppe von Threads darstellt, die Aufgaben ausführen. Die Größe des Threadpool kann während der Laufzeit angepasst werden um alle verfügbaren Prozessoren zu nutzen.
- Aufgaben werden durch die abstrakte Klasse "RecursiveTask" oder "RecursiveAction" repräsentiert.
 - ↳ für Aufgaben ohne Ergebnis.
 - ↳ für Aufgaben mit Ergebnis.
- Aufgaben werden rekursiv aufgerufen & parallel ausgeführt. Wenn die Größe einer Aufgabe zu klein ist (Schwellwert entscheidet), dann wird sie nicht mehr aufgeteilt, sondern sequenziell ausgeführt.

Von welcher abstrakten Klasse des Fork-Join-Frameworks muss ich ableiten, für...

Aufgaben ohne Rückgabe?

- Klasse RecursiveAction
- Einsatz: Beispiel am Mergesort, da es das Lösungsprinzip "Teile und Herrsche" verwendet & damit rekursiv beschrieben werden kann.
Rekursionsbasis: eine zu sortierende Folge von einem Element ist sortiert.
- Rekursionsvorschrift:
 - 1) Die zu sortierende Folge von mehreren Elementen in 2 Hälften teilen.
 - 2) Sortieren der linken & rechten Hälfte.
 - 3) Zusammenführen der Hälften zur sortierten Folge via Reissverschlussverfahren, bzw. "To merge".

Aufgaben mit Rückgabe?

- Klasse RecursiveTask
- Einsatz: z.B. ein Check der Arraysortierung oder wenn durch die parallel Bearbeitung ein Ergebnis ermittelt wird.
Rekursionsbasis: der aktuelle Bereich berechnet & als Ergebnis zurückgegeben.
- Rekursionsvorschrift:
 - 1) Beide Hälften mit den Elementen werden in 2 geteilt.
 - 2) Beide Hälften werden geprüft.
 - 3) Die Resultate werden logisch addiert. ()

Aufgaben mit Warten am Ende der parallelen Bearbeitung?

- Klasse CounterCompleted (abstrakt)
- Einsatz: Bei abgeschlossenen Aufgaben, die warten müssen auf andere Aufgaben BEVOR das endgültige Ergebnis zurückgegeben werden kann.

WAS IST DAS Work-Stealing Verfahren des ForkJoinPools?

Konzept, bei dem inaktive Threads (Threads ohne Aufgaben momentan) Aufgaben von Nachbarsthreads klauen.

Wie verwende ich den ForkJoinPool?

- Größe des Pools anpassen auf die verfügbare Hardware. Damit wird die CPU-Auslastung maximiert. Mit `Runtime.availableProcessors()` initialisiert.
- unnötige Parallelisierung vermeiden. Wenn Aufgaben zu klein, lieber sequenziell lösen anstatt auf Teilen.
- Abhängigkeiten berücksichtigen: mit `CountedCompleter` & `join()` arbeiten!
- Verwenden vom Work-Stealing-Verfahren.

RECAP AUTOMATEN

WAS sind einige Anwendungsbeispiele für Automaten?

- Ampeln
- Getränkeautomaten
- Turingmaschine
- Prozessor

WAS sind die Drei Grundtypen von Automaten?

Ein Automat im Allgemeinen besteht aus einer Menge von Zuständen & einer Menge von Übergängen zwischen diesen Zuständen. Er wechselt basierend auf den Eingaben & dem aktuellen Zustand seinen Zustand.

Der Mealy-Automat ist ein spezieller Typ von Automat. Die Ausgabe hängt sowohl vom aktuellen Zustand als auch von der Eingabe ab. Das bedeutet, dass beim Durchlaufen des Automaten nicht nur der Zustand gewechselt wird, sondern auch eine Ausgabe erzeugt wird.

Der Moore-Automat ist ein weiterer Typ von Automat. Die Ausgabe hängt ausschließlich vom aktuellen Zustand ab, nicht von der Eingabe.

Wie stelle ich einen Automaten als Graphen ODER als Tabelle DAR?

Graph Tutorial : 1) Identifizieren der Zustände des Automaten. Jeder Zustand wird zu einem Knoten im Graphen.

2) Zeichnen der Zustände als Knoten & verbinden mit Kanten um die Übergänge zwischen den Zuständen darzustellen.

3) Kanten beschriften mit den entsprechenden Eingaben um deutlich zu machen, welche Eingaben zu welchem Zustandwechsel führen.

4) Falls erforderlich, den Startzustand des Automaten markieren. (Pfeilmarkierung / spezieller Anfangsknoten)

5) Endzustände des Automaten markieren, die den erfolgreichen Abschluss einer Berechnung oder das Erreichen eines bestimmten Ziels anzeigen

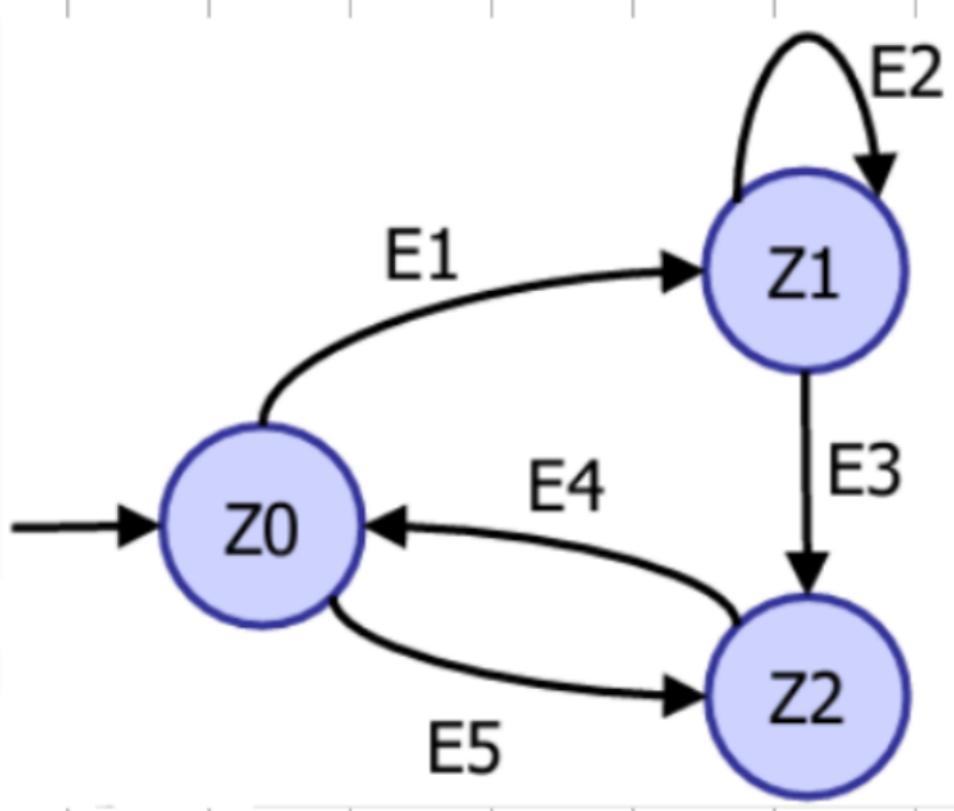


Tabelle : 1) Tabelle mit Spalten für die Eingaben, die aktuellen Zustände & resultierende Zustände erstellen.

2) Alle möglichen Eingaben in einer Spalte auflisten.

3) Für jeden Zustand des Automaten eine Liste erstellen.

4) Tabelle ausfüllen, indem man für jeden Zustand & jede Eingabe den resultierenden Zustand angibt.

Zustand	Eingabe	Folgezustand
Z0	E1	Z1
Z1	E2	Z1
Z1	E3	Z2
Z2	E4	Z0
Z0	E5	Z2

Wie sieht ein Automat als Code aus?

Die gängigste Methode ist der **Switch-Case**. Sie ermöglicht die einfache Überprüfung des aktuellen Zustands & die Ausführung von entsprechender Aktionen basierend auf Eingaben.

```
public Zustand vollzieheZustandsuebergang(final Zustand aktuellerZustand,  
    final Eingabe eingabe) {  
    switch (aktuellerZustand) {  
        case Z1:  
            switch (eingabe) {  
                case E1:  
                    methodA1();  
                    return Z2;  
                case E2:  
                    methodA2();  
                    return Z3;  
                // ...  
            }  
        case Z2:  
            switch (eingabe) {  
                // ...  
            }  
    }  
}
```