

AD Summary

Dies ist eine Zusammenfassung für das Modul **Algorithmen und Datenstrukturen** von Eldar Omerovic.



Code wurde nicht getestet und ist nicht immer korrekt. Ich habe den Code direkt in diesem Sheet geschrieben. Es geht mehr ums Verständnis und Konzept von einzelnen Funktionen und Algorithmen!
Oft habe ich auch Ausdrücke verwendet, bei denen ich weiß, dass sie nicht funktionieren wie z.B. generische Arrays erstellt, da der Typ keine Rolle spielte im Beispiel!

Weitere meiner Zusammenfasung von Modulen ander Hochschule Luzern sind auf Github verfügbar.

<https://github.com/omeldar/hslu>

SW.01 - Einführung in Algorithmen und Datenstrukturen

Was sind Algorithmen und Datenstrukturen

Definition Algorithmus

Definition Datenstruktur

Algorithmen und Datenstrukturen

Gleichwertige Algorithmen

Komplexität eines Algorithmus

Definition Komplexität

Interpretation der O-Notation

Definition Big-O-Notation

Wichtige Ordnungsfunktionen

Andere Faktoren

Nachhaltiges Programmieren

SW.01 - Rekursion

Grundlegendes zur Rekursion

Iteration vs. Rekursion

Call Stack

Mächtigkeit der Rekursion

PROs und KONs der Rekursion

Rekursionstypen

SW.02 - Collections

Was sind Datenstrukturen

Verschiedene Arten von Datenstrukturen

Eigenschaften von Datenstrukturen

Collection Framework Übersicht

Speicherverwaltung

SW.02 - Arrays, Listen, Queue und Stack

Reihenfolge und Sortierung von Datenstrukturen

Operationen auf Datenstrukturen

Statische vs. Dynamische Datenstrukturen

Explizite vs. Implizite Beziehungen

Direkter vs. Indirekter Zugriff

Aufwand von Operationen

SW.02 - Equals, Hashcode, Comparable

Equals

Verwendung

Default Implementation

Equals-Contract

HashCode

Hashcode-Contract

Comparable<T> Interface

CompareTo-Contract

CompareTo Implementation

SW.03 - Bäume

Gerichtete und ungerichtete Bäume

Grundelemente eines Baumes

Ordnung eines Baumes

Grad eines Knoten

Pfad zu einem Knoten

Tiefe eines Knoten

Niveaus / Ebenen eines Baumes

Höhe eines Baumes

Gewicht eines Baumes

Füllgrad eines Baumes

SW.03 - Binäre Bäume

Traversieren von binären Bäumen

SW.03 - Performancemessungen

SW.04 - Tipps für die Java-Praxis

Veraltete Implementationen

Collections and synchronization

Concurrent vs Synchronized

Immutable bzw. Unmodifiable Collections

Handling von leeren Listen

Generics and Raw Types

Keine generischen Arrays

SW.05 - Threads

Vor- und Nachteile von Nebenläufigkeit

Mass für die Parallelisierung

Erzeugen und Starten von Threads

Beenden eines Threads

SW.05 - Synchronisation

Elementare Synchronisationsmechanismen

Thread Lebenszyklus

Synchronisation durch Monitor-Konzept

Deadlock

SW.06 - Thread Steuerung

Wartezustand

Semaphore

SW.07 Weiterführende Konzepte

Future und Callable

Callable, Future und ExecutorService

Exceptions

Atomic

Synchronisierte Collections

SW.11 Sortieren

Motivation fürs Sortieren

Voraussetzung fürs Sortieren

Soriteralgorithmen und deren Komplexität

Radix-Sortieralgorithmen

Stabiler vs. Instabiler Sortieralgorithmus

Internes vs. Externes Sortieren

SW.11 - Parallelisierungsframeworks

Parallelisierungsgrad

SW.12 - Automaten

Grundlagen

Sprachen

Endliche Automaten

Überblick zur Nebenläufigkeit

`start()` -Methode der `Thread` Klasse

`join()` -Methode der `Thread` Klasse

Schlüsselwort `synchronized`

`wait()` -Methode der Klasse `Object`

`notify()` und `notifyAll()` Methoden der `Object` Klasse

Semaphor Implementation

`Callable` Interface

`ExecutorService` & `Future<T>`

`Fork` & `Join`

Wichtige Klassen, Methoden & Interfaces

Fork-Join-Framework Klassen

Task-Aufteilung Methoden

`ForkJoinPool` -Framework Methoden

`ForkJoin` -Framework Methoden

Detaillierter Überblick der Datenstrukturen und Algorithmen

Grundlegende Datenstrukturen

Array

Stack

Listen

Fortgeschrittene Datenstrukturen

Heap

Queue (hier: CircularQueue)

Binary Search Tree (Binärer Suchbaum)

Hashbasierte Datenstrukturen

Hashtable

Grundlegende Sortieralgorithmen

Insertion Sort (Direktes Einfügen)

Selection Sort (Direktes Auswählen)

Bubble Sort

Shellsort

Höhere Sortieralgorithmen

Suchalgorithmen

Binary Search

Graphtheorie

Grundlagen

Gerichtete und ungerichtete Graphen

Grad

Markierte Graphen

Formale Beschreibung

Dicht vs Dünn besetzter Graph

Pfade und Zyklen

Bewertete Graphen

Baum

Problemlösung mittels Graphen

Grundtypen der Zielfindung / Zielfunktion

Adjazenzmatrix

Adjazentlisten

Traversieren

Breitensuche (Breadth First Search, BFS)

Tiefensuche (Depth First Search, DFS)

Transitive Hülle

Algorithmen

Algorithmus von Floyd

Alle kürzesten Pfade finden

Algorithmus von E.W. Dijkstra

Algorithmus von E.W. Dijkstra - Beispiel

Informationen zur MEP

SW.01 - Einführung in Algorithmen und Datenstrukturen

| Foliensatz: E11_IP_AlgorithmenDatenstrukturenKomplexität

Was sind Algorithmen und Datenstrukturen

Definition Algorithmus

Ein Algorithmus ist ein Verfahren zur Lösung eines Problems; genauer gesagt zur Lösung einer Problemklasse.

Etwas Informatiknäher:

Algorithmus = Lösungsverfahren (Rezept, Anleitung), welches einen Input zu einem Output verarbeitet.

Problemklassen, die mit Algorithmen gelöst werden können, nennt man **berechenbar**.

Eigenschaften eines Algorithmus sind:

- **Schrittweises** Verfahren
- **Endet** nach endlich vielen Schritten (→ **terminiert**).
- **Was** ein Schritt zu leisten hat, ist eindeutig.

Einige Beispiele

Berechnung des grössten gemeinsamen Teilers (ggT) für zwei beliebige natürliche Zahlen

→ Euklidischer Algorithmus

Zeichnen der Verbindungslien, welche zwei Punkte verbindet.

→ Bresenham Algorithmus

Sortierung von zufällig vorliegenden ganzen Zahlen

→ Mergesort Algorithmus

Definition Datenstruktur

Eine Datenstruktur ist ein Konzept zur Speicherung und zur Organisation von Daten. Es handelt sich um eine **Struktur**, weil die Daten in einer bestimmten Art und Weise angeordnet und verknüpft werden, um den Zugriff auf sie und ihre Verwaltung möglichst effizient zu ermöglichen. Datenstrukturen sind daher insbesondere auch durch die **Operationen** charakterisiert, welche Zugriff und Verwaltung realisieren.

Einige Beispiele

- Array: direkter Zugriff (+), fixe Grösse (-)
- Liste: flexible Grösse (+), sequentieller Zugriff (-)

Algorithmen und Datenstrukturen

- Algorithmen operieren typisch auf Datenstrukturen und Datenstrukturen bedingen spezifische Algorithmen. Beides ist eng miteinander verbunden.
- Bei vielen Algorithmen hängt der Ressourcenbedarf, d.h. die benötigte Laufzeit und Speicheraufwand, von der Verwendung geeigneter Datenstrukturen ab.

Die Implementation als Code eines Algorithmus, kann auf verschiedenste Weise gemacht werden. Es gibt nicht nur eine richtige Lösung. Oft können Algorithmen Iterativ und auch Rekursiv implementiert werden.

Wenn unterschiedliche Implementierungen alle den gleichen Input bekommen und damit auch immer den gleichen Output erzeugen, spricht man von **gleichwertigen** Implementationen. Auch wenn der Speicherbedarf oder der Ressourcenbedarf nicht gleich sind. Entscheidend ist, dass die Ausgabe aufgrund einer Eingabe die Selbe ist.

Gleichwertige Algorithmen

Die Gleichwertigkeit allgemein zu beweisen, ist ein **unlösbares Problem**. Das bedeutet, dass es kein Java-Programm gibt, das den Quellcode von zwei verschiedenen Implementationen einliest und feststellen kann, ob sie gleichwertig sind oder nicht!

Komplexität eines Algorithmus

Definition Komplexität

Komplexität, auch Aufwand oder Kosten, eines Algorithmus ist der notwendige Ressourcenbedarf, um den Algorithmus anzuwenden. Also die Ressourcen die benötigt werden, um die Eingabe mithilfe des Algorithmus zur Ausgabe umzuwandeln.

Ressourcenbedarf

Der Ressourcenbedarf kann in zwei Kategorien aufgeteilt werden.

- Rechenzeit → Zeitkomplexität
- Speicherbedarf → Specherkomplexität

Eingabedaten

Die Eingabedaten spielen hierbei auch eine Rolle. Einen höheren Integer-Wert zu speichern, benötigt mehr Speicher als einen niedrigeren zu speichern.

- Grösse der Datenmenge
(z.B. 10 vs 1'000'000'000 Elemente zu sortieren)
- Grösse des Datenwertes
(z.B. Fakultät von 10 vs. von 1'000'000'000 zu berechnen)

Bei der Ermittlung des Ressourcenbedarfs interessiert uns jedoch nicht der absolute Ressourcenbedarf für ein Beispiel, sondern es interessiert uns wie sich der Algorithmus verhält, wenn wir grössere oder einfacher mehr Daten speichern. Je nach Hardware wäre die Rechenzeit sowieso auf jedem Gerät eine Andere.

Dafür können wir die Big-O-Notation verwenden.

Interpretation der O-Notation

Die Rechenzeit einer Beispiels-Funktion verhält sich gemäss der Ordnung $O(n^2)$. Dies bedeutet:

- Verdoppelung von n
→ $2^2 = 4$ -fache Rechenzeit
- Verdreibachung von n
→ $3^2 = 9$ -fache Rechenzeit

- Verzehnfachung von n
 $\rightarrow 10^2 = 100$ -fache Rechenzeit

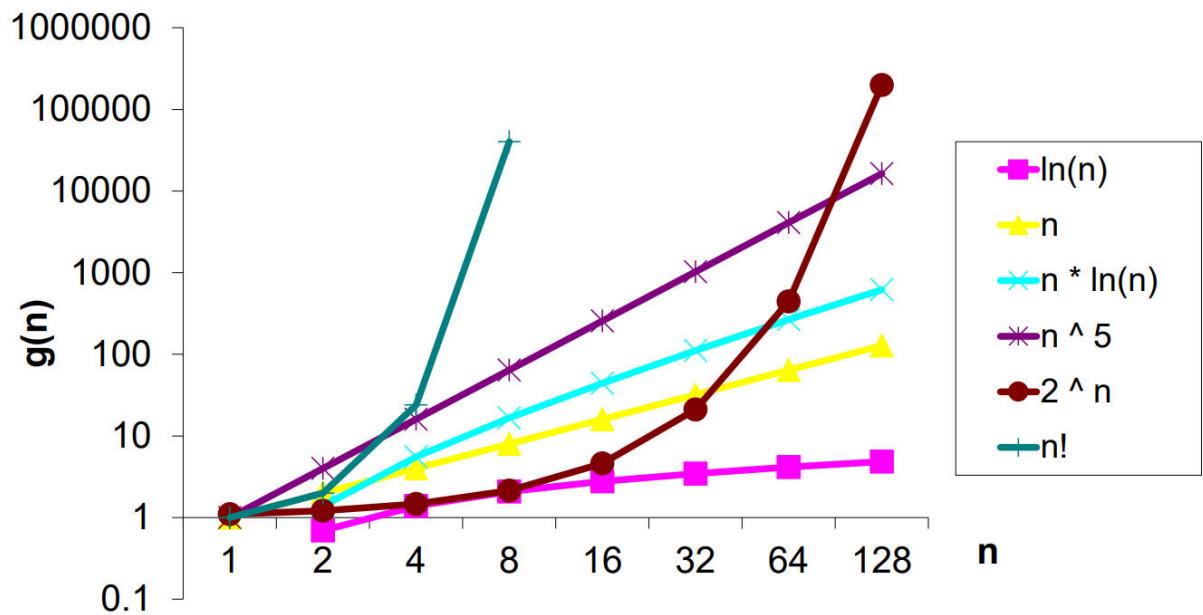
Definition Big-O-Notation

Die Big-O-Notation bringt zum Ausdruck, dass die Funktion $f(n)$ höchstens so schnell wächst, wie eine andere Funktion $g(n)$. $g(n)$ ist die obere Schranke für $f(n)$.

Wichtige Ordnungsfunktionen

Wachstum	Big-O-Notation	Verwendung
Konstant	$O(1)$	Hashing
Logarithmisch	$O(\ln(n))$	binäres Suchen
Linear	$O(n)$	Suchen in Texten
$n \log n$	$O(n * \ln(n))$	raffiniertes Sortieren
Polynomial	$O(n^m)$	simples Sortieren
Exponential	$O(d^n)$	Optimierungen
Fakultät	$O(n!)$	Permutationen, TSP

TSP = Travelling Salesman Problem. Mehr dazu in späteren Kapiteln.



Andere Faktoren

- Die Big-O-Notation (Ordnung) ignoriert konstante Faktoren. Diese können aber praktisch relevant sein!
- Die Ordnung gilt für grosse n . Im praktischen, wenn man einen Algorithmus für ein kleines n sucht, können Algorithmen mit höheren Ordnungen die bessere Wahl sein.
- Man differenziert oft:
 - Best Case
 - Worst Case
 - Average Case

Meistens ist der Average Case das bessere Mass als der Worst Case.

- Geht es um die konkrete Ausführungszeit so spielen viele andere Faktoren ebenfalls eine Rolle
 - vorliegende Eingabedaten
 - Hardware-Effekte (Cores, Branch Prediction, ...)

Nachhaltiges Programmieren

Algorithmen mit niedrigerer Komplexität sind nachhaltiger:

- Code, der nicht ausgeführt wird, verbraucht keine Energie
Berechnungen sollten nicht auf Vorrat durchgeführt werden
- Speziell das Verschieben von Daten kostet viel Energie
Effiziente Datenformate/Datenstrukturen, Caches und Buffer helfen Datenflüsse gering zu halten.

SW.01 - Rekursion

| *Foliensatz: E12_IP_Rekursion*

Grundlegendes zur Rekursion

Viele Algorithmen und Datenstrukturen sind von Natur aus selbstähnlich, bzw. selbstbezüglich:

- Der ggT von (21, 15) ist gleich dem ggT von (21-15, 15).
- Ein Ordner enthält Dateien und weitere Ordner
- Ein Teil einer Matrix, einer Liste, eines Baumes, eine Graphen ist selbst wieder eine Matrix, eine Liste, ein Baum, ein Graph

Rekursion bedeutet, dass sich etwas in sich selbst wiederholt.

Iteration vs. Rekursion

Bei einem iterativ implementierten Algorithmus wird ein Call auf die Funktion des Algorithmus gemacht. Dieser iteriert dann für eine gewisse Zeit, bis das Problem gelöst ist und die Ausgabe bereitsteht. Bei einem rekursiv implementierten Algorithmus ruft die Funktion sich selbst immer wieder mit neuen Eingabewerten auf. Es wird auch vor jedem weiteren Funktionsaufruf eine Condition (Bedingung) überprüft, um sicherzustellen, dass die Funktionsaufrufe irgendwann enden.

Beispiel: Fakultät berechnen - Iterativ

Die iterative Fakultätsberechnung ist ziemlich einfach.

$$n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \dots \cdot n$$

Pseudocode

```
int calcFactorial(of number)
    int result = 1

    for (as long as i <= number)
        result = result * i

    return result
```

Beispiel: Fakultät berechnen - Rekursiv

Vereinfacht dargestellt:

- Gesucht ist $5! = ?$

- Wird zurückgeführt auf $5 \cdot 4!$

- Wird zurückgeführt auf $5 \cdot (4 \cdot 3!)$

- ...

- Wird zurückgeführt auf
 $5 \cdot (4 \cdot (3 \cdot (2 \cdot 1!)))$

- Der Rekursionsbasis
entnehmen wir: $1! = 1$

Pseudocode

```
int calcFactorial(of number)
    if (number == 0 OR number == 1)
        return 1
    else
        return n * calcFactorial(number - 1)
```

Hier wird die Funktion `calcFactorial(of number)` von sich selber aufgerufen. Jedes mal wenn dies passiert definiert man also das man den jetzigen Wert mit der Fakultät der Zahl - 1 multipliziert. Dies macht man bis man bei 1 angekommen ist. Danach wird ganz simpel multipliziert und die Methodenaufrufe geben jeweils deren Werte zurück bis wir das Ergebnis vom ersten Aufruf zurück erhalten.

Call Stack

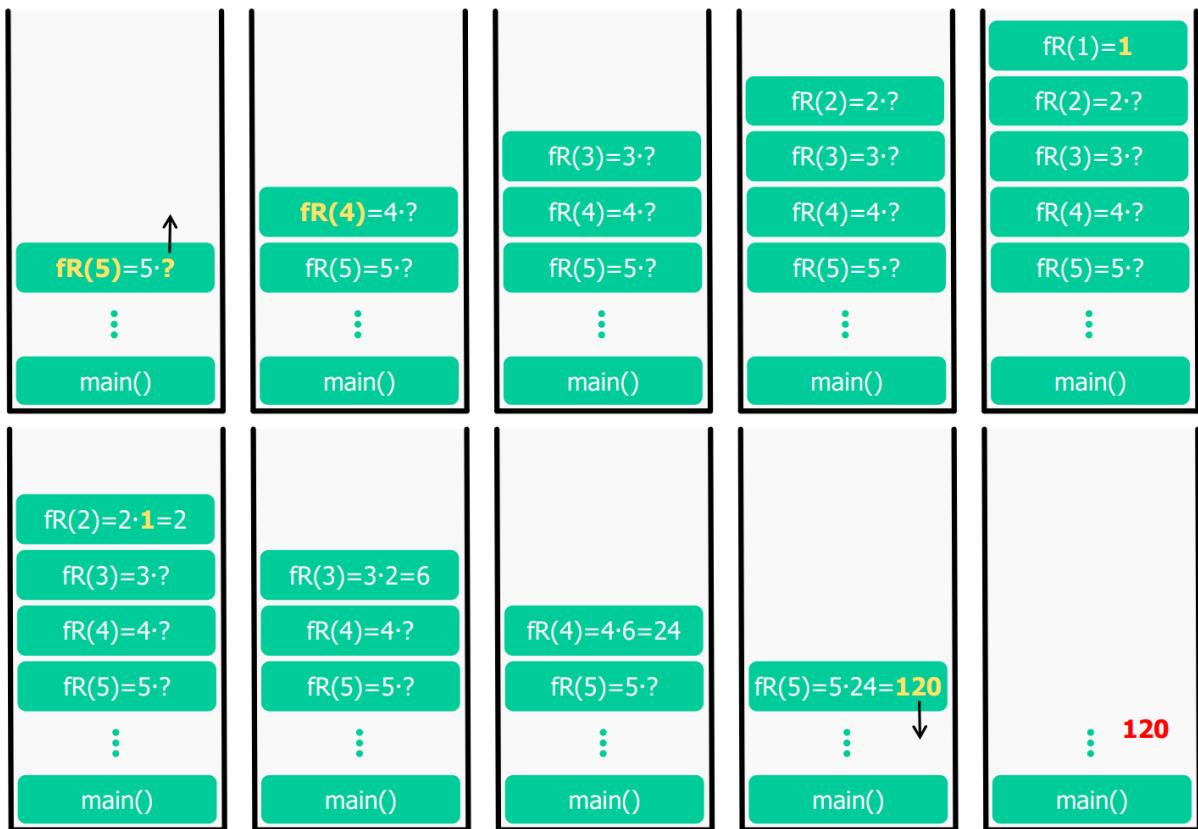
Für die Ausführung eines Programmes verwendet die JVM (Java Virtual Machine) zwei wichtige Speicher: Heap und Call Stack

Heap: In diesem Speicherbereich werden die Objekte gespeichert. D.h deren Instanzvariablen bzw. Zustände. Nicht mehr referenzierbare Objekte werden durch den Garbage Collector (GC) automatisch gelöscht.

Call Stack: Bei der Ausführung eines Java-Programms wird eine Kette von Methoden aufgerufen. Ursprung ist die `main()`-Methode. Jeder Methodenaufruf erfordert Speicher. Insbesondere für die aktuellen Parameter und lokalen Variablen. Dazu dient der Call Stack. Jeder Thread besitzt seinen Call Stack.

Ein neuer Methodenaufruf bewirkt, dass der Call Stack wächst bzw. darauf ein zusätzliches Stack Frame angelegt wird.

Der Call-Stack in Aktion (Beispiel der rekursiven Fakultätsberechnung)



Mächtigkeit der Rekursion

- Rekursion und Iteration sind praktisch gleich mächtig.
- Das heisst, die Menge der berechenbaren Problemstellungen bei Verwendung der Rekursion und Verwendung der Iteration ist gleich.
- Dies bedeutet wiederrum, dass eine rekursive Implementation auch immer iterativ und umgekehrt umgesetzt werden kann.

PROs und KONs der Rekursion

PROs

- häufig einfache und elegante Problemlösung
- (meistens) weniger Quellcode
- Korrektheit häufig einfacher zu zeigen
- Lisp und Prolog (Programmiersprachen) kennen nur die Rekursion

⇒ Generell, sollte immer eine iterative Lösung gegenüber einer rekursiven bevorzugt werden.

KONs

- grosser Speicherbedarf auf dem Call Stack
- Gefahr eines Stack Overflow
- tendenziell langsamere Programmausführung

Rekursionstypen

Lineare Rekursion

Die Ausführung einer Methode `m(...)` führt zu höchstens einem rekursiven Aufruf, d.h. primitiv rekursiv. z.B. Fakultät.

Nichtlineare Rekursion

Die Ausführung der Methode `m(...)` führt zu mehr als einem rekursiven Aufruf.

z.B. Fibonacci-Zahlen: $f_5 \rightarrow f_3, f_4$

Direkte Rekursion

Eine rekursive Methode ruft sich direkt selbst auf.

```
int functionA(b) {  
    return functionA(b)  
}
```

Indirekte Rekursion

Eine rekursive Methode ruft sich indirekt selbst auf.

```
int functionA(x) {  
    return functionB(x)  
}  
  
int functionB(y) {  
    return functionA(y)  
}
```

SW.02 - Collections

Was sind Datenstrukturen

Datenstrukturen werden verwendet, um eine Mengen von Daten bzw. Objekten effizient zu speichern und verarbeiten zu können. Unter Verarbeiten versteht man Funktionen, wie z.B. alle enthaltenen Daten/Objekte einzeln zu bearbeiten, nach bestimmten Objekten zu suchen, zu zählen, zu filtern oder nach unterschiedlichen Kriterien zu sortieren etc.

Verschiedene Arten von Datenstrukturen

Es gibt unterschiedliche Datenstrukturen:

- Array: indexierte Reihung
- Tree: hierarchisch geordnete Daten (Baumstruktur)
- List: einfache Reihung
- Map: Zuordnung zwischen Schlüssel und Wert(-paaren)
- Queue: Warteschlange, FIFO
- Set: Sammlung ohne Duplikate
- Stack: Stapel- oder Kellerspeicher, FILO

Java bietet für jede Art verschiedene Implementationen dieser Datenstrukturen.

Eigenschaften von Datenstrukturen

Datenstrukturen unterscheiden sich in vielerlei Hinsicht:

- Grösse: Dynamisch oder statisch
- Zugriff: Direktzugriff oder indirekt / sequenziell
- Sortierung: Sortiert oder unsortiert
- Suche: Beschleunigte Suche (binär oder über Hashwert)
- Geschwindigkeit: Grundlegende Operationen wie Suchen, Einfügen, Anhängen, Entfernen, Verschieben von einzelnen Elementen an verschiedenen Positionen

Collection Framework Übersicht

Interfaces: Abstrakte Datentypen, welche verschiedenartige Datenstrukturarten repräsentieren:

`List<E>` abstrahiert Listen, `Map<K, V>` abstrahiert `Map`, etc.

Implementationen: Konkrete, wiederverwendbare Implementationen

`LinkedList<E>` und `ArrayList<E>` sind Implementationen von `List<E>`

Algorithmen: Methoden zur Behandlung von Datenstrukturen

`iterator()` ermöglicht sequenziellen Zugriff, `sort(List<E>)` sortiert beliebige List-Implementationen

Speicherverwaltung

Stark vereinfacht erklärt!

Speicher wird in Blöcken angefordert. Um Speicher anzufordern, kann man die Methode `malloc(int)` verwenden:

- `malloc(4); x.write("HSLU");`
- `malloc(3); x.write("IST");`
- `malloc(5); x.write("SUPER");`

\$00	\$01	\$02	\$03	\$04	\$05	\$06	\$07	\$08	\$09	\$0A	\$0B	\$0C	\$0D	\$0E	\$0F
H	S	L	U	I	S	T	S	U	P	E	R				

Dieser Speicher kann auch freigegeben werden:

- `free(blue) ← kein Java`

\$00	\$01	\$02	\$03	\$04	\$05	\$06	\$07	\$08	\$09	\$0A	\$0B	\$0C	\$0D	\$0E	\$0F
H	S	L	U	I	S	T	S	U	P	E	R				

SW.02 - Arrays, Listen, Queue und Stack

Reihenfolge und Sortierung von Datenstrukturen

Die unterschiedlichen Datenstrukturen gehen unterschiedlich mit der Anordnung und Sortierung der gespeicherten Daten um. Folgende Möglichkeiten gibt es:

- Ungeordnet, Reihenfolge nicht deterministisch (\rightarrow Reine Sammlung)
- Behalten die Reihenfolge der Datenelemente (z.B. in der Folge des Einfügens) implizit bei
- Sortieren (typisch beim Einfügen) die Elemente in die Datenstruktur

Operationen auf Datenstrukturen

Es gibt elementare Methoden die auf Datenstrukturen angewendet werden können:

- Einfügen von Elementen
- Suchen von Elementen
- Entfernen von Elementen
- Ersetzen von Elementen

Es gibt auch Operationen welche in Abhängigkeit einer (optionalen) Reihenfolge oder Sortierung (natürlich oder speziell) sind:

- Nachfolger: Nachfolgendes Datenelement
- Vorgänger: Vorangehendes Datenelement
- Sortierung: Sortieren der Datenelemente nach Attributwerten
- Maxima und Minima: kleinstes und grösstes Datenelement

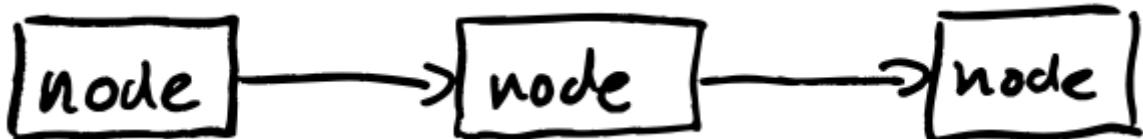
Statische vs. Dynamische Datenstrukturen

Statische Datenstrukturen verändern nach der Initialisierung ihre Grösse nicht mehr. Somit können sie nur eine beschränkte Anzahl Elemente aufnehmen.

Dynamische Datenstrukturen können ihre Grösse während der Lebensdauer verändern. Somit können sie eine beliebige Anzahl Elemente aufnehmen. Dies ist durch den Speicher limitiert.

Explizite vs. Implizite Beziehungen

Bei expliziten Datenstrukturen werden die Beziehungen zwischen den Daten von jedem Element selber explizit mit Referenzen festgehalten. z.B LinkedList



Bei impliziten Datenstrukturen werden die Beziehungen zwischen den Daten nicht von den Elementen selber festgehalten. z.B Array



Direkter vs. Indirekter Zugriff

Bei direktem Zugriff hat man auf jedes Element in der Datenstruktur direkten Zugriff. z.B. Array.

Implizite Beziehung → Direkter Zugriff

Bei indirektem Zugriff hat man keinen direkten Zugriff auf alle Datenelemente. Man kann jedoch sequenziell ein Element nach dem anderen erhalten. z.B. LinkedList

Explizite Beziehung → Indirekter Zugriff

Aufwand von Operationen

Der Aufwand (Rechen- und Speicherkomplexität) variiert sowohl für die verschiedenen Operationen als auch der enthaltenen Datenmenge.

Meistens interessiert uns die Ordnung, also wie sich der Aufwand in Abhängigkeit zur Anzahl Elemente verhält.

SW.02 - Equals, Hashcode, Comparable

Equals

Equals entscheidet über die Gleichheit von Objekten. Diese Methode ist auf der Basisklasse `Object` bereits definiert: `boolean equals(Object obj)`

Durch das **Überschreiben** dieser Methode können bzw. müssen wir für jede spezialisierte Klasse (Typ) die gewünschte Art von Gleichheit **selber bestimmen**.

Verwendung

Viele Collections setzen voraus, dass `equals()` richtig implementiert ist!

Wenn wir Objekte auf die Gleichheit prüfen möchten, verwenden wir immer die `equals()` Methode. Nur elementare Datentypen dürfen mit `==` verglichen werden.

Der Gleichheitsoperator `==` prüft immer auf Identität! z.B. mit Strings:

```
String msg1 = "HELLO";
String msg2 = "HELLO";
boolean never = (msg1 == msg2); // FALSCH!
boolean ever = (msg1.equals(msg2)); // RICHTIG!
```

Default Implementation

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

Equals-Contract

- **Reflexitt:** `x.equals(x)` liefert immer true
- **Symmetrie:** `x.equals(y)` liefert das gleiche wie `y.equals(x)`
- **Transitivitt:** `x.equals(y) == true` und `y.equals(z) == true` → dann muss `x.equals(z) == true` sein.
- **Konsistenz:** Bei wiederholten Vergleichen und unvernderten Objekten immer das gleiche Resultat.
- **Vergleich mit null:** `x.equals(null)` liefert immer false, keine `NullPointerException` !

HashCode

Die `hashCode()` Methode liefert im Einklang mit `equals()` die Adresse des Objekts als Hashwert. Somit liefern nur identische Objekte den selben `int`-Wert! Dies gilt jedoch nur für eine Laufzeit (Programmausführung).

Hashcode-Contract

- Wenn zwei Objekte im Sinne von `equals()` gleich sind, müssen auch die Hashwerte identisch sein.
- Sind zwei Objekte im Sinne von `equals()` ungleich, sollten sie (idealerweise) verschiedene Rückgabewerte liefern → ist keine Voraussetzung, würde perfekte Hashes verlangen).
- Wenn das Objekt nicht verändert wird, darf sich der Hashwert nicht verändern.

Comparable<T> Interface

Mit `Comparable` kann auf andere Faktoren als die Gleichheit zwischen Objekten geprüft werden. Das `Comparable` Interface erlaubt es beliebigen Klassen, sich mit anderen Objekten des selben Typs **vergleichbar** zu machen.

CompareTo-Contract

- `this < other` liefert einen negativen Wert (typisch -1)
- `this == other` liefert 0
- `this > other` liefert einen positiven Wert (+1)

CompareTo Implementation

1. Test auf Identität mit `==` (→ 0 bei Identität)
2. Schrittweiser Vergleich relevanter Attribute
 - a. Elementare Datentypen mit `==`, `>`, `<` aber besser mit `compare(...)` auf den jeweilige Wrapper-Typen
 - b. Klassentypen: Vergleich jeweils an deren eigene `compareTo()` Methode

SW.03 - Bäume

Bäume sind in der Informatik Datenstrukturen, die eine hierarchische Struktur haben. Ein Dateisystem mit Verzeichnissen und Dateien ist zum Beispiel hierarchisch aufgebaut.

Gerichtete und ungerichtete Bäume

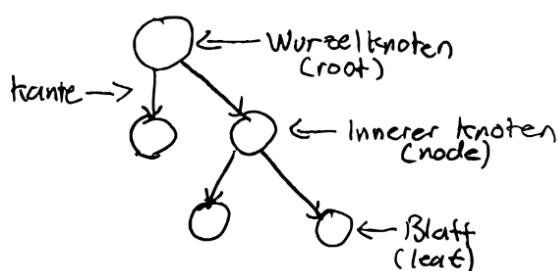
Ungerichtete Bäume in der Informatik sind Datenstrukturen, die aus Knoten und Kanten bestehen und keine spezifische Richtung haben. Jeder Knoten kann mit anderen Knoten über Kanten verbunden sein, wodurch ein Netzwerk von Beziehungen entsteht. Ungerichtete Bäume ermöglichen es, hierarchische Strukturen abzubilden, in denen die Beziehungen zwischen den Knoten symmetrisch sind und keine spezifische Hierarchie vorgibt.

Gerichtete Bäume können in zwei Richtungen gerichtet sein. Von der Wurzel aus nach unten zu den Blättern (**Out-Tree**) oder von den Blättern nach oben zur Wurzel (**In-Tree**). Der Out-Tree ist hier die weitaus häufiger verwendete Form.

Es gibt diverse Spezialformen von Bäumen:

- Binär-Baum - am einfachsten und häufigsten
- AVL-Baum - höhenbalancierter Binärbaum
- B-Baum - balancierter Baum, nicht zwingend binär
- B*-Baum - restriktivere Form des B-Baumes
- Binomial-Baum - speziell strukturierter Baum

Grundelemente eines Baumes



Ein Baum ist aufgebaut wie ein Graph. Er hat Knoten (Datenelemente) und Kanten, die die Relationen der Datenelemente zueinander aufzeigen.

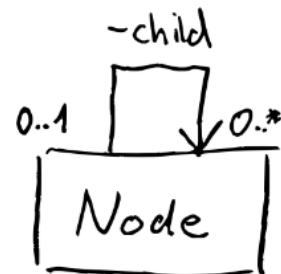
Wurzelknoten (root): Innerer Knoten ohne Parent

Innerer Knoten (node): Alle Knoten mit einem Parent und mind. einem Child

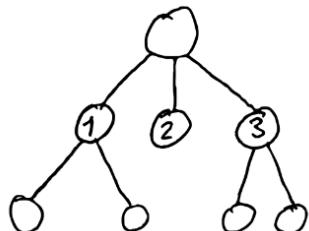
Blattknoten (leaf): Knoten ohne Child

Modelliert wird ein Baum wie folgt:

Ein Baum besteht aus Nodes. Das Wurzelement kennt die Referenzen zu seinen direkten Child-Nodes, diese wiederrum zu ihren direkten Child-Nodes, usw....



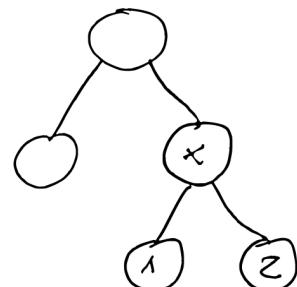
Ordnung eines Baumes



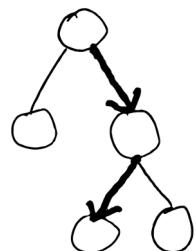
Die Ordnung eines Baumes beschreibt wie viele Children der Knoten mit den meisten Children hat. In diesem Beispiel hat der Wurzelknoten die meisten Child-Nodes (hier: 3). Somit ist die Ordnung des Baumes 3.

Grad eines Knoten

Der Grad eines Knoten beschreibt wie viele Kinder dieser Knoten hat. Im Beispiel rechts sehen wir, dass der Knoten x den Grad 2 hat.



Pfad zu einem Knoten

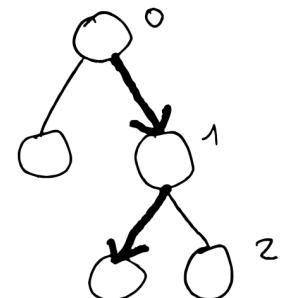


Ein Pfad im Baum beschreibt den Weg vom Wurzelknoten zu einem bestimmten Knoten. Hier sehen wir dass die Länge des Pfads 2 ist. Alle Kanten die von einem Knoten zu einem anderen führen, können als Pfad beschrieben werden.

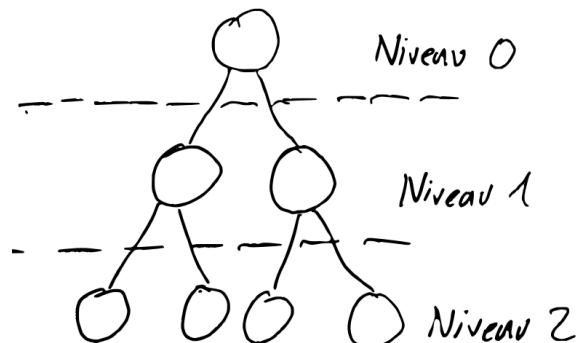
Tiefe eines Knoten

Die Tiefe eines Knoten beschreibt die Länge seines Pfades.

Wir sehen im Beispiel, dass die Tiefe des Knoten 2 ist. Es führen 2 Kanten (Pfadlänge 2) vom Wurzelknoten zum Blattknoten.



Niveaus / Ebenen eines Baumes

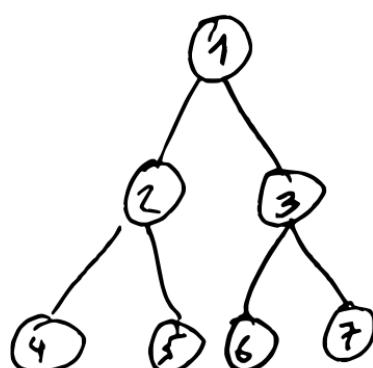


Als Niveau oder Ebene bezeichnet man die Menge aller Knoten, welche die gleiche Tiefe haben. Im Beispiel links sehen wir, dass der Baum 3 Niveaus / Ebenen hat.

Höhe eines Baumes

Die Höhe eines Baumes definiert sich aus der Tiefe des Knoten welcher am weitesten von der Wurzel entfernt ist. → Unterstes Niveau + 1

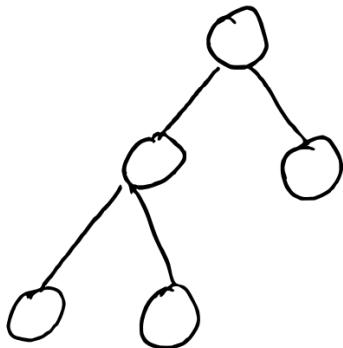
Gewicht eines Baumes



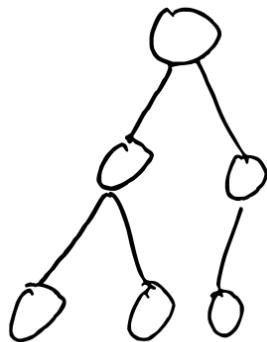
Das Gewicht eines Baumes wird durch die Anzahl Knoten bestimmt, die dieser trägt. In diesem Beispiel links, hat der Baum 7 Knoten und somit ein Gewicht von 7.

Füllgrad eines Baumes

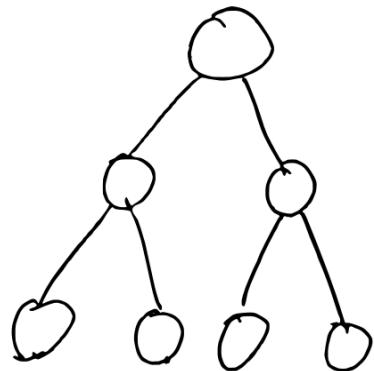
Ausgefüllter Baum



Voller Baum



Kompletter Baum /
Vollständiger Baum



Alle innere Knoten haben die maximale Anzahl an Kindern.

Alle Niveaus ausser das letzte haben die maximale Anzahl Knoten. Das letzte Niveau ist linksbündig (oder auch rechtsbündig) angeordnet.

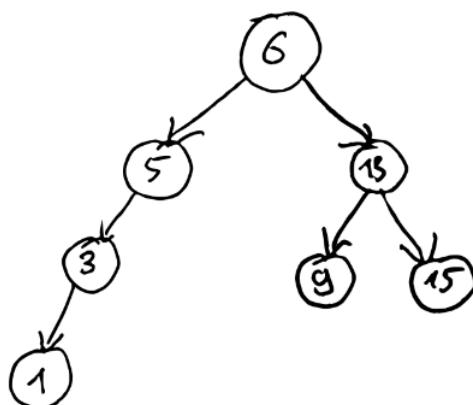
Jedes Niveau hat die maximale Anzahl Knoten. Er hat für sein Gewicht die minimale Anzahl Niveaus

SW.03 - Binäre Bäume

Ein binärer Baum ist als Baum mit einer Ordnung von 2 definiert. Jeder Knoten hat maximal zwei Kindknoten.

Traversieren von binären Bäumen

Aufgrund der spezifischen Eigenschaft von binären Bäumen (Ordnung 2), kann man diese auf drei unterschiedliche Arten traversieren. Dies wird hier anhand eines Beispiels erklärt. Der Beispiel-Baum:



Bezeichnung	travers-Methode (Pseudocode)	Reihenfolge der Verarbeitung (process)
Preorder	<pre>process(node); travers(node.left); travers(node.right);</pre>	6, 5, 3, 1, 13, 9, 15
Inorder	<pre>travers(node.left); process(node); travers(node.right);</pre>	1, 3, 5, 6, 9, 13, 15
Postorder	<pre>travers(node.left); travers(node.right); process(node);</pre>	1, 3, 5, 9, 15, 13, 6

SW.03 - Performancemessungen

Es gibt einige wichtige Punkte, die zu befolgen sind, wenn man akkurate Performancemessungen machen möchte:

- Die Messungen der ersten Ausführungen ignorieren
- Mehrere Ausführungen (mind. 3) messen und eine Durchschnittszeit berechnen (iterative Wiederholung)
- `System.currentTimeMillis()` kann dazu verwendet werden

SW.04 - Tipps für die Java-Praxis

Veraltete Implementationen

- `java.util.Stack`
- `java.util.Vector`
- `java.util.HashTable`

Diese Datenstrukturen existieren seit JDK 1.0, sind jedoch nützlich, da sie thread safe sind. Dies macht sie langsamer als andere Datenstrukturen. Wenn es keinen Grund gibt eine synchronisierte Datenstruktur zu verwenden, sollten diese nicht verwendet werden. Ansonsten könnten diese auch heute noch nützlich sein.

Collections and synchronization

Für jede Datenstruktur der Collections-Klasse gibt es eine synchronisierte Implementation. z.B eine synchronized ArrayList:

```
final List<T> syncList = Collections.synchronizedList(new ArrayList<>());
```

Concurrent vs Synchronized

Neben synchronisierten Datenstrukturen gibt es Collection-Implementationen, welche einen Concurrent-Prefix im Namen tragen. Beispiele:

- `java.util.concurrent.ConcurrentMap<K,V>`

- `java.util.concurrent.ConcurrentLinkedDeque<E>`
- `java.util.concurrent.ConcurrentSkipListSet<E>`

Diese Datenstrukturen sind so geschickt implementiert, dass sie mit atomaren Operationen parallele Operationen erlauben und ohne klassische Synchronisation trotzdem thread safe sind.

Immutable bzw. Unmodifiable Collections

Analog zu den synchronized*-Methoden, wird dabei für eine bestehende Liste ein Proxy erzeugt, das keine Modifikation mehr an der Liste zulässt.

```
List<String> demo = new ArrayList<String>();  
List<String> unmodifiableList = Collections.unmodifiableList(demo);  
unmodifiableList.add("Try"); // Runtime error*
```

So kann eine Liste von immutable Objects ebenfalls als immutable geschützt werden.

Handling von leeren Listen

Generell sollte kein null-Wert als Rückgabe auf eine leere Collection folgen. Nur weil eine Collection leer ist, sollte man diese nicht null setzen.

Es gibt einfache Wege wie man an leere Collections kommt, dementsprechend sollte immer eine leere Collection zurückgegeben werden. Über statische Hilfsmethoden der Klasse Collections können sogar direkt leere (und unmodifiable!) Collections generiert werden.

- `Collections.emptyList();`
- `Collections.emptySet();`
- `Collections.emptyMap();`

Generics and Raw Types

Mit Java 1.5 wurden Generics eingefügt. Diese ermöglichen es uns Klassen und Methoden zu erstellen, die mit unterschiedlichen Typen arbeiten können.

Raw Typen

Aus technischen Gründen (Backwards-Compatibility) sind raw-Implementationen immer noch in Java vorhanden. Diese raw-Types sollten jedoch nicht mehr verwendet werden.

```
List rawList = new ArrayList();
```

Selbst wenn man nur eine Liste von Objects erzeugen will, sollte man diese Liste explizit typisieren:

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

Warum Raw Types schlecht sind

Bei der Verwendung von raw-Types **verliert man deutlich mehr Typsicherheit** als man denkt. In Java gibt es explizite Regeln was bei Generics ein Subtype ist und was nicht:

Eine `List<String>` ist zwar ein Subtype von `List`, aber

Eine `List<String>` ist kein Subtype von `List<Object>`

Keine generischen Arrays

In Java gibt es keine generische Arrays! Keine der folgenden Ausdrücke ist zulässig:

```
... = new List<E>[];
... = new List<String>[];
... = new E[];
```

⇒ Es sind generell in Java Collections Arrays vorzuziehen! Arrays sind hauptsächlich für kleine Datenmengen zu verwenden!

SW.05 - Threads

Vor- und Nachteile von Nebenläufigkeit

Vorteile

- Steigerung der Performance
Bsp. kann auf mehreren CPUs das Sortieren eines grossen Arrays auf mehrere Threads verteilt werden.
- Zur Verfügung stehende Rechenleistung wird besser ausgenutzt

Nachteile

- Programmcode mit Multithreading ist oft schwer zu verstehen und mit hohem Aufwand zu warten
- Erschwertes Debugging, weil CPU-Zuteilung an die Threads nicht deterministisch ist

- Durch Auslagerung von blockierenden Tätigkeiten in separate Threads können Anwendungen reaktiv gehalten werden.
- Parallel ablaufende Threads müssen koordiniert werden. Vor allem wenn sie auf gemeinsame Daten zugreifen

Mass für die Parallelisierung

Masszahl für den Performance-Gewinn ist der Speedup (Beschleunigung bzw. Leistungssteigerung)

$$S = \frac{T_{seq}}{T_{par}}$$

Gesetz von Amdahl (https://de.wikipedia.org/wiki/Amdahlsches_Gesetz)

Hier wird vorausgesetzt, dass wir von einem fest vorgegebenen Programm und somit mit einer fixen Problemgrösse ausgehen.

$$S(N) = \frac{\text{Sequenzielle Laufzeit}}{\text{Parallele Laufzeit}} = \frac{1}{\frac{P}{N} + (1 - P)} = \frac{1}{(1 - P)}$$

P: prozentualer parallelisierbarer Anteil, **N:** Anzahl Prozessoren

Gesetz von Gustafson (https://de.wikipedia.org/wiki/Gustafsons_Gesetz)

Hier wird die Problemgrösse als variabel betrachtet. Die Problemgrösse vergrössert sich in einem festen Zeitfenster.

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

Erzeugen und Starten von Threads

Zur Ausführung des Programms startet die JVM den `main`-Thread. Die JVM selbst bildet einen Prozess des Betriebssystem. Wenn das Betriebssystem selbst Threads unterstützt, kann die JVM die Java-Threads auf sie abbilden. Es gibt generell zwei Arten wie ein Java Thread erzeugt werden kann.

- Ableitung der Klasse `Thread` → Erzeugen eines Thread Objekts
- Implementation des Interfaces `Runnable` → Erzeugen eines Objekts der `Runnable` Klasse oder erzeugen eines Thread Objekts der Klasse `Thread`

Das Thread Objekt muss mit der Methode `start()` gestartet werden. Gemäss IP Dokumentation, soll in den meisten Fällen das `Runnable` Interface verwendet werden, wenn man nur die `run()` Methode überschreiben möchte und keine Methoden der `Thread` Klasse.

Beenden eines Threads

Mit dem Ende der `run()` Methode soll auch der Thread beendet werden. Allgemein ist ein Thread beendet, wenn eine der folgenden Bedingungen zutrifft:

- Die `run()` Methode wird ohne Fehler beendet
- In der `run()` Methode tritt eine Exception auf, welche die `run()` Methode beendet
- Wenn ein Programm `System.exit()` aufgerufen wird
- Der Thread wird von aussen aktiv beendet

Erwartet wird, dass wenn die Beendigung eines Threads angeordnet wird, dieser zeitnahe beendet wird, nachdem er benutzte Objekte in einen konsistenten Zustand gebracht hat (z.B. gemeinsame Ressourcen bereinigt, kritische Bereiche freigegeben). Auch soll er die anstehende Aufgaben zu Ende führen, er soll jedoch keine neuen Aufgabe beginnen.

Aktives Beenden eines Threads

Es gibt Möglichkeiten für das Beenden eines Threads durch einen anderen Programmteil.

- erzwungen (forceful cancellation) → sofortiger Abbruch
- verzögert (deferred cancellation) → heißt Abbruch beim nächsten Abbruchpunkt
- kooperativ (cooperative cancellation) → heißt Threads wird gebeten zu beenden und muss sich dann selbst beenden, z.B. durch `return`

Bis zur Version 1.1 verwendete Java die Methode `stop()` für den erzwungenen Abbruch. Seit 1.2 ist diese deprecated und sollte nicht verwendet werden!

Beenden eines Threads durch Interrupt

Die Methode `interrupt()` setzt ein Interrupt-Flag des Threads, das eine Unterbrechungsanforderung signalisiert. Diese Methode löst keine Exception aus. Die Methode `isInterrupted()` liest dieses Flag. Sie gibt `true` zurück, wenn das Interrupted-Flag gesetzt wurde.

Befindet sich der Thread in einer blockierten Wartemethode, wird er durch `interrupt()` geweckt. Ist der Thread nicht im Wartemodus, stösst jedoch im weiteren Verlauf auf eine Wartemethode, wird diese gleich nach Betreten wieder verlassen.

In beiden Fällen wird eine `InterruptedException` geworfen. Folgende Wartemethoden lösen eine `InterruptedException` aus:

- `sleep()`
- `wait()`
- `join()`

Wichtig ist, dass das Interrupted-Flag wieder auf false gesetzt wird, wenn die Exception ausgelöst wird. Somit muss dieses wieder neu gesetzt werden:

```
public void run() {  
    try {  
        while (Thread.currentThread().isInterrupted() == false) {  
            // do thing  
        }  
    } catch (InterruptedException ex) {  
        // Thread was interrupted in a waiting method  
        Thread.currentThread().interrupt();  
    } ...  
}
```

Handling von InterruptedException

Wenn eine InterruptedException ausgelöst wird, sollte man üblicherweise diese abfangen und terminieren. Es ist üblich dann mindestens eine Exception Meldung auszugeben oder einen Log Eintrag zu machen. Im `catch`-Teil sollte die Beendigung des Threads eingeleitet werden. Auch sollte nochmals die `interrupt()` Methode aufgerufen werden, da das Interrupted-Flag zurückgesetzt wurde.

SW.05 - Synchronisation

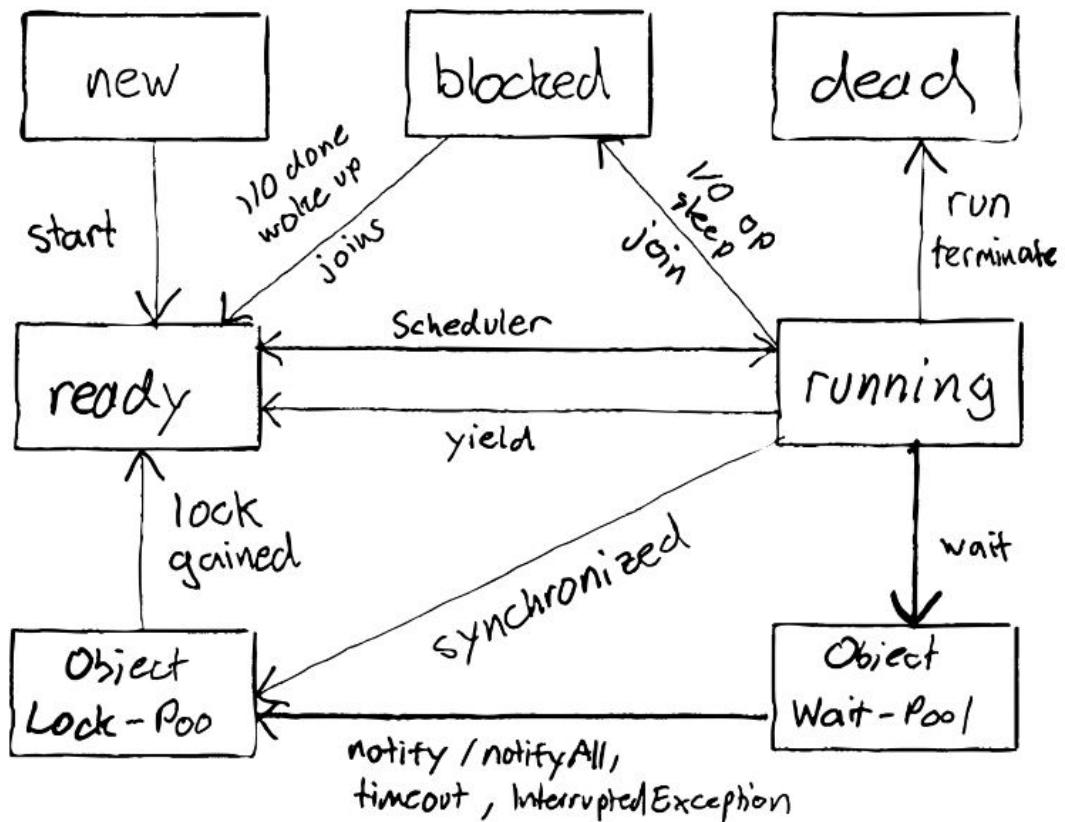
Elementare Synchronisationsmechanismen

Eine gute Lösung für den gegenseitigen Ausschluss muss vier Bedingungen erfüllen:

1. In einem kritischen Abschnitt darf sich zu jedem Zeitpunkt höchstens nur ein Thread befinden
2. Es dürfen keine Annahmen über die zugrunde liegende Hardware (Clock, CPU-Anzahl) gemacht werden

3. Ein Thread darf andere Threads nicht blockieren, ausser er ist in einem kritischen Bereich
4. Es muss sichergestellt sein, dass ein Thread nicht unendlich lange warten muss, bis er in den kritischen Bereich eintreten kann.

Thread Lebenszyklus



Synchronisation durch Monitor-Konzept

Monitore werden verwendet, um die Synchronisatzion von Zugriffen zu handhaben. Das Betreten von kritischen Abschnitten kann mit wieder betretenden Monitoren (Reentrant Monitor) oder geschachtelten Monitoren (Nested Monitor) gemacht werden.

Reentrant Monitor

Eine synchronisierte Methode ruft eine andere auf, wobei beide synchronisierte Code Abschnitte denselben lock-Pool verwenden. Hier wird auf beiden Methoden das gleiche Lock-Object verwendet:

```
public class ReentrantMonitorExample {  
    public synchronized void outerMethod() {  
        System.out.println("Outer Method");  
        innerMethod();  
    }  
  
    public synchronized void innerMethod() {  
        System.out.println("Inner Method");  
    }  
  
    public static void main(String[] args) {  
        ReentrantMonitorExample example = new ReentrantMonitorExample();  
        example.outerMethod();  
    }  
}
```

Nested Monitor

Eine synchronisierte Methode ruft eine andere auf, bei der beide synchronisierte Code Abschnitte verschiedene Objekt lock-Pools verwenden. Hier gibt es unterschiedliche Lock-Objects:

```
public class NestedMonitorExample {  
    private Object lock1 = new Object();  
    private Object lock2 = new Object();  
  
    public void method1() {  
        synchronized (lock1) {  
            System.out.println("Method 1: Sync block 1");  
            method2();  
        }  
    }  
  
    public void method2() {  
        synchronized (lock2) {  
            System.out.println("Method 2: Sync block 2");  
        }  
    }  
  
    public static void main(String[] args) {  
        NestedMonitorExample example = new NestedMonitorExample();  
        example.method1();  
    }  
}
```

Deadlock

Ein Deadlock tritt dann auf, wenn zwei Threads sich gegenseitig blockieren. Beide Threads warten jeweils auf die Beendigung des anderen.

```
private final Object lock1 = new Object();
private final Object lock2 = new Object();

public void method1() {
    synchronized (lock1) {
        synchronized (lock2) {
            // do something
        }
    }
}

public void method2() {
    synchronized (lock2) {
        synchronized (lock1) {
            // do something
        }
    }
}
```

Wenn hier `method1()` und `method2()` von verschiedenen Threads etwa gleichzeitig aufgerufen werden, erwirbt Thread 1 den `lock1`, während Thread 2 den `lock2` erwirbt. Jetzt benötigen beide Threads den jeweils anderen lock zusätzlich. Da aber beide locks vergeben sind, kommt kein Thread in der Ausführung weiter.

Man soll keine Nested Monitore verwenden, wenn man dies umgehen kann. Diese können zu Deadlocks führen.

SW.06 - Thread Steuerung

Wartezustand

Befindet sich ein Thread in einem Wartezustand, gibt es drei Wege, wie er diesen wieder verlassen kann:

- Ein anderer Thread signalisiert den Zustand Wechsel mittels `notify()` bzw. `notifyAll()`
- Die im Argument angegebene Zeit (timeout) ist abgelaufen
- Ein anderer Thread ruft die Methode `interrupt` des wartenden Threads auf

Wenn das Warten aufgrund einer Bedingung passiert, sollte immer die `wait()` Methode in einer Iteration (Schleife) aufgerufen werden und die Bedingung immer wieder neu überprüft werden. Es kann nämlich passieren, dass nach einem `notify`

ein anderer Thread etwas geändert hat und die Bedingung trotzdem noch nicht erfüllt ist.

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

Semaphore

Eine semaphore funktioniert ähnlich wie eine Ampel. Wenn sich ein Thread in einem Block befindet, wird die Ampel rot gesetzt.

Für ein Semaphor `s` sei

- `p` die Anzahl der abgeschlossenen P-Operationen auf `s`
- `v` die Anzahl der abgeschlossenen V-Operationen auf `s`
- `n` ein Initialwert, der `n` Passiersignale im Semaphore initialisiert

Dann gilt: $s \geq 0$ and $s = n + v - p$

SW.07 Weiterführende Konzepte

Future und Callable

Das Runnable Interface implementiert eine nebenläufig auszuführende Aufgabe und besitzt keinen Rückgabewert. Soll eine Aufgabe einen Wert zurück liefern, geht das nur über ein spezielles Rückgabe-Attribut.

Callable<T>

Ein `Callable` kann einen Rückgabewert besitzen. Das Interface `Callable` kennt nur eine `call()` Methode. Falls eine Exception während der Ausführung der `call()` Methode passiert, wird diese zurückgegeben. Falls eine Exception bei der Ausführung der `call()` Methode geworfen wird, wird diese zurückgegeben.

Das `Callable` Objekt selbst wird einem `FutureTask` als Argument übergeben.

Future<T>

`Future<T>` bietet auch `get(long timeout, TimeUnit unit)` an, bei welcher der Aufrufer eine maximale Wartezeit angeben kann. Wenn das Ergebnis nicht innerhalb der vorgegebenen Zeit verfügbar ist, wird eine `TimeOutException` geworfen.

Mit `isDone()` kann der Bearbeitungsstatus abgefragt werden.

Zum Abbrechen kann `cancel(boolean mayInterruptIfRunning)` benutzt werden.

Callable, Future und ExecutorService

Ein `Callable` wird einem `ExecutorService` über die Methode `submit()` zur Ausführung übergeben. Die Methode `submit()` liefert ein `FutureTask` Objekt zurück. Über das `FutureTask` Objekt kann die Rückgabe erfragt werden.

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

Tasks an einen ExecutorService übergeben

- `Future<?> submit(Runnable task)`
Das zurückgegebene Future Objekt wird verwendet, um `isDone()`, `cancel()` und `isCancelled()` aufzurufen. Der `get()` Aufruf liefert nur den Wert `null`.
- `Future<T> submit(Runnable task, T result)`
Hier liefert die `get()` Methode des Future Objekts das vorgegebene `result` Objekt als Ergebnis zurück.
- `Future<T> submit(Callable<T> task)`
In dieser Version wird ein `Future` Objekt zurückgeliefert, mit dem das Ergebnis der Berechnung abgeholt werden kann.

Exceptions

Beim Aufruf von `submit()` wird nicht direkt eine `Exception` geworfen, falls im Task eine `Exception` auftritt. Erst bei einem Zugriff mit dem `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);
}
```

Wenn eine Exception im Task abgefangen wird, muss diese trotzdem weitergegeben werden. Denn ansonsten wird diese im anderen Thread nie ausgelöst.

Atomic

Der lesende oder schreibende Zugriff auf Variablen eines primitiven Datentyps (32-bit) in Java ist atomar, d. h. nicht unterbrechbar. Ausnahmen sind 64bit Typen `long` und `double`. Zugriffe auf Referenzvariablen sind dagegen immer atomar. Unabhängig davon, ob es sich um eine 32- oder 64-bit JVM handelt. Werden Variablen mit `volatile` gekennzeichnet, ist der Zugriff garantiert immer atomar, unabhängig vom primitiven Datentypen.

concurrent.atomic

Seit Java 5 gibt es das Paket `java.util.concurrent.atomic` mit Kapselungen (Wrapper Klassen) für verschiedene Datentypen. Die gängigen Klassen sind `AtomicBoolean`, `AtomicInteger`, `AtomicLong` und `AtomicReference`.

Compare-and-Set

Zentral ist die Compare-and-Set Operation. Mit ihr kann eine Variable atomar gelesen und verändert werden. → `boolean compareAndSet(int expect, int update)`

Synchronisierte Collections

Java besitzt seit seiner ersten Version die standardisierten Container `Vector`, `Stack`, `HashTable` und `Dictionary`. Die öffentlichen Methoden dieser Container sind durch `synchronized` geschützt, was jedoch in Singlethreaded Anwendungen zu unnötigen Performance Verlusten führt. Bei den Containern des Collection-Frameworks hat man diesen Schutz weggelassen. Für jedes Collection Interface steht eine öffentliche Klassenmethode von Collections zur Verfügung, die eine entsprechend synchronisierte Containerfassade zurückliefert.

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

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

SW.11 Sortieren

Motivation fürs Sortieren

- Suchen bzw. Zugriff viel schneller
- Implementation von Algorithmen setzen oft sortierte Daten voraus

Voraussetzung fürs Sortieren

- Datenelemente müssen einen Key-Wert (Schlüssel) haben, nach dem sie sortiert werden können. Diese Keys müssen vergleichbar sein (Comparable).

Soriteralgorithmen und deren Komplexität

- Einfache Sortieralgorithmen $\rightarrow O(n^2)$
 - direktes Einfügen (Insertion Sort)
 - direktes Auswählen (Selection Sort)
 - direktes Austauschen (Bubble Sort)
- Höhere Sortieralgorithmen $\rightarrow O(n \cdot \log(n))$
 - Quicksort
 - Heapsort
 - Mergesort
- Sortiernetzwerke mit $O(n)$ Prozessoren/Cores $\rightarrow O(\log(n))$
 - parallelisierter Mergesort

Radix-Sortieralgorithmen

- Radix-Sortieralgorithmen bedingen spezielle Anforderungen an die Schlüssel und finden entsprechend selten Verwendung.
 - Ein eigentliches Vergleichen wird damit hinfällig
- Die Schlüsselwerte geben direkt die Position des Elements an, wodurch das Sortieren mit einer Komplexität (bestenfalls) mit $O(n)$ gelöst ist
- Beispiele: Counting-Sort, Radix-Sort, Bucket-Sort

Stabiler vs. Instabiler Sortieralgorithmus

Stabiler Sortieralgorithmus

Der Algorithmus **garantiert**, dass durch das Sortieren **die Reihenfolge unter gleichen Datenelementen nicht ändert.**

[C (1)], [A], [D], [C(2)], [B] → sortieren → [A], [B], [C (1)], [C (2)], [D]

Instabiler Sortieralgorithmus

Ein instabiler Sortieralgorithmus garantiert nicht, dass die Reihenfolge von gleichen Elementen gleich bleibt.

[C (1)], [A], [D], [C(2)], [B] → sortieren → [A], [B], [C (2)], [C (1)], [D]

Internes vs. Externes Sortieren

Internes Sortieren

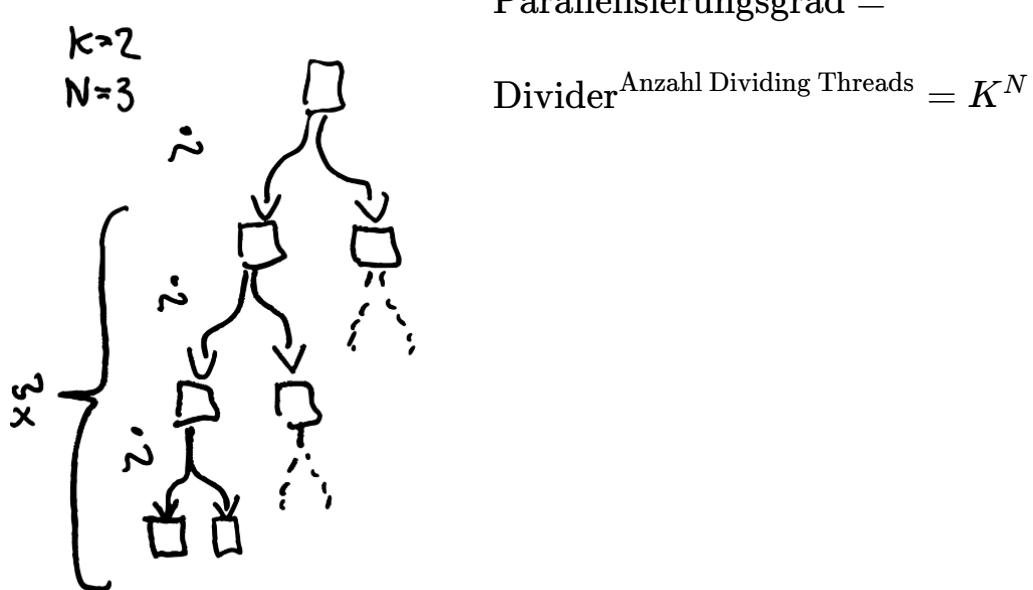
- Daten liegen im Arbeitsspeicher vor
- Direktes Vergleichen der Daten möglich
- Internes Sortieren ist primär von Bedeutung
- Beispiel: Sortieren von Arrays

Externes Sortieren

- Daten liegen in einem externen Massenspeicher vor
- NUR Lesen und Schreiben möglich, kein direktes Vergleichen!
- Interner Speicher ist für alle Daten zu klein
- Beispiel: Sortieren von sequentiellen Dateien / Files.

SW.11 - Parallelisierungsframeworks

Parallelisierungsgrad



SW.12 - Automaten

Grundlagen

Zustände: Z, Eingaben: E, Ausgaben: A

Mealy-Automat

Ein Zustandsübergang bewirkt auch eine Ausgabe

Moore-Automat

Zustand selbst ist auch mit einer Ausgabe verbunden

Sprachen

1. Typ 0: Unbeschränkte Grammatiken (Unrestricted Grammar)

- Unbeschränkte Grammatiken haben keine Einschränkungen in Bezug auf die Regeln oder Produktionsmuster.
- Sie können jede Art von formaler Sprache erzeugen.

- Diese Grammatiken sind äußerst mächtig, aber auch schwer zu analysieren und zu verarbeiten.

2. Typ 1: Kontextsensitive Grammatiken (Context-Sensitive Grammar)

- Kontextsensitive Grammatiken erlauben Regeln, bei denen eine Variable auf der linken Seite durch eine längere Zeichenkette ersetzt werden kann, aber nur in einem bestimmten Kontext.
- Die Produktionsregeln müssen die Form $\alpha \rightarrow \beta$ haben, wobei α und β Zeichenketten sind und $|\alpha| \leq |\beta|$.
- Diese Grammatiken werden verwendet, um viele natürliche und künstliche Sprachen zu beschreiben.

3. Typ 2: Kontextfreie Grammatiken (Context-Free Grammar)

- Kontextfreie Grammatiken haben Regeln, bei denen die Variable auf der linken Seite immer durch eine bestimmte Zeichenkette ersetzt werden kann, unabhängig vom Kontext.
- Die Produktionsregeln müssen die Form $A \rightarrow \beta$ haben, wobei A eine einzelne Variable und β eine Zeichenkette ist.
- Diese Grammatiken werden oft zur Definition von Programmiersprachen und syntaktischen Strukturen verwendet.

4. Typ 3: Reguläre Grammatiken (Regular Grammar)

- Reguläre Grammatiken sind die einfachsten Grammatiken in der Chomsky-Hierarchie.
- Die Regeln bestehen aus einer einzelnen Variable, gefolgt von einem Terminalsymbol oder der Variable selbst.
- Diese Grammatiken werden zur Beschreibung regulärer Ausdrücke und endlicher Automaten verwendet.
- Die von regulären Grammatiken erzeugten Sprachen sind die regulären Sprachen.

Endliche Automaten

- endliche Anzahl von Zuständen
- endliches Eingabealphabet
- DEA (deterministischer endlicher Automat) & NEA (nichtdeterministischer endlicher Automat)

DEA vs. NEA

Der Hauptunterschied zwischen einem deterministischen endlichen Automaten (DEA) und einem nichtdeterministischen endlichen Automaten (NEA) liegt in ihrer Art, Eingaben zu verarbeiten und Zustandsübergänge zu definieren:

Deterministischer endlicher Automat (DEA):

- Ein DEA hat für jeden Zustand und jedes Eingabesymbol genau einen definierten Zustandsübergang.
- Bei gegebener aktueller Zustand und Eingabe gibt es immer genau einen eindeutigen nächsten Zustand.
- DEA sind präzise und vorhersehbar, da sie eindeutige Entscheidungen treffen können.

Nichtdeterministischer endlicher Automat (NEA):

- Ein NEA kann für einen Zustand und ein Eingabesymbol mehrere mögliche Zustandsübergänge haben oder sogar ϵ (leeres Eingabesymbol) als Übergang verwenden.
- Bei gegebener aktueller Zustand und Eingabe kann es mehrere mögliche nächste Zustände geben oder auch keinen Übergang.
- NEA können mehrere alternative Pfade und Entscheidungen haben, was zu einem nicht eindeutigen Verhalten führt.

Zusammenfassend kann man sagen, dass DEAs streng deterministisch sind und für jede Eingabe einen eindeutigen nächsten Zustand haben, während NEAs nichtdeterministisch sind und für dieselbe Eingabe mehrere mögliche nächste Zustände haben können. NEAs sind oft flexibler in Bezug auf die akzeptierten Sprachen, aber ihre Ausführung erfordert möglicherweise zusätzliche Mechanismen wie das Backtracking, um alle möglichen Pfade zu durchsuchen. DEAs sind einfacher zu implementieren und zu verstehen, haben jedoch geringere Ausdrucksstärke als NEAs.

Überblick zur Nebenläufigkeit

`start()` -Methode der `Thread` Klasse

Um Parallelität zu erreichen, müssen wir neue Threads starten können.

```
final Thread thread = new Thread(myTask, "MyTask-Thread");
thread.start();
```

Mithilfe der `isAlive()` -Methode kann überprüft werden, ob ein Thread im **RUNNABLE** Zustand ist.

`join()` -Methode der `Thread` Klasse

Die Methode `join()` in Java wird verwendet, um den aktuellen Thread auf den Abschluss eines anderen Threads zu warten. Diese Methode ist blockierend. Bedeutet, dass der Thread, der auf einen anderen warten muss, blockiert wird.

```
main() {
    Thread thread = new Thread(() -> {
        Thread.sleep(2000);
    });

    thread.start();

    thread.join(); // main thread waits until other thread completed
}
```

Schlüsselwort `synchronized`

Alle Java-Objekte besitzen eine implizite Sperre, also einen Lock. Den Zugriff auf den Lock erhält man durch das Schlüsselwort `synchronized`. Während des Wartens auf den Lock kann der Thread nicht mit `interrupt` unterbrochen werden.

Diese beide Implementationen sind äquivalent.

```
private int count = 0;
public synchronized int increment() {
    count++;
    return count;
}
```

```
private int count = 0;
public int increment() {
    synchronized (this) {
        count++;
        return count;
    }
}
```

wait() -Methode der Klasse Object

Threads müssen den Lock eines Objektes besitzen, um `wait()` aufrufen zu können.

```
public final void wait() throws InterruptedException
public final void wait(long timeout) throws InterruptedException
```

Beim Aufruf von `wait()` wird der aufrufende Thread in einen Wartezustand versetzt. Gleichzeitig wird der Lock auf diesen Abschnitt freigegeben.

notify() und notifyAll() Methoden der Object Klasse

`notify()` weckt genau einen Thread im Warte-Zustand auf. Falls mehrere Threads warten, ist nicht vorhersehrbar oder bestimmbar, welcher Thread aufgeweckt wird. Der Thread wartet noch einmal, bis er den Lock für den synchronized Abschnitt erhält. Erst dann wird er wieder **ready** zur erneuten Ausführung.

`notifyAll()` hingegen weckt alle an diesem Objekt wartenden Threads auf. Generell wird immer empfohlen `notifyAll()` zu verwenden.

Semaphor Implementation

```
private int sema; // Semaphor counter

public Semaphore(final int init) {
    sema = init;
}

public synchronized void acquire() throws InterruptedException {
    while (sema == 0) {
        this.wait();
    }
    sema--;
}

public synchronized void release() {
    sema++;
    this.notifyAll();
}
```

Callable Interface

Hier übernimmt das nebelaufige Programm ein Callable. In diesem Beispiel wird ein Array sortiert und zurückgegeben → Rückgabewert.

```
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;
    }
}
```

ExecutorService & Future<T>

1. Arbeit an ExecutorService übergeben
2. Etwas anderes machen
3. Das Resultat abholen

```
new Random().nextBytes(array);
final Callable<byte[]> task = new ArraySortTask(array);
final ExecutorService executor = Executors.newCachedThreadPool();
final Future<byte[]> result = executor.submit(task);
//...do something else
final byte[] sortedArray = result.get();
```

Fork & Join

Fork wird von einem übergeordneten Thread aufgerufen, um einen neuen Thread zu erstellen.

Join wird von beiden Threads (Parent & Child) aufgerufen, wenn die Operation zu Ende ist. Der Parent-Thread wartet, bis der Sub-Thread die Operation beendet hat (joins) und fährt danach fort.

Threadpool ForkJoinPool

Methoden des ForkJoinPool

- `void execute(ForkJoinTask<?> task)`
Führt den übergebenen Task asynchron aus
- `<T> T invoke(ForkJoinTask<T> task)`
Startet die Ausführung des Tasks, wobei gewartet wird, bis der Task fertig ist (synchrone Ausführung). Zudem gibt es die Methode `invokeAll()`, wo mehrere Tasks ausgeführt werden und auf das Ende aller Tasks gewartet wird.
- `<T> ForkJoinTask<T> submit(ForkJoinTask<T> task)`
Führt den mitgegebenen Task asynchron aus und liefert ein `ForkJoinTask` Objekt zurück, das ein `Future` Objekt ist, mit dem man z.B. auf den Rückgabewert zugreifen kann.

Wichtige Klassen, Methoden & Interfaces

Fork-Join-Framework Klassen

`RecursiveAction` : Für Aufgaben ohne Rückgabe

`RecursiveTask` : Für Aufgaben mit Rückgabe

`CountedCompleter` : Es wird mitgezählt wie viele Subtasks laufen. Es kann frühzeitig abgebrochen werden. Wenn man in einem der Subtasks das Ziel erreicht hat, können alle abgebrochen werden.

Task-Aufteilung Methoden

`void run()` : Zu überschreibende Methode des Runnable-Interface

`Integer call()` : Zu überschreibende Methode des Callable-Frameworks

`Integer compute()` : Zu überschreibende Methode des Fork-Join Frameworks für

`RecursiveTask`

`void compute()` : Zu überschreibende Methode des Fork-Join Frameworks für `CountedCompleter` und `RecursiveAction`

`Future<T> submit(Callable<T>)` : Übergibt ein `Callable` oder `Runnable` einem ExecutorService zur Ausführung. Über das Future mithilfe der `get()` Methode kann die Rückgabe gelesen werden.

ForkJoinPool -Framework Methoden

`void execute()` : Führt einen Task asynchron aus, wartet nicht auf Fertigstellung → keine Rückgabe

`void invoke()` : Führt einen Task synchron aus, wartet auf Fertigstellung → Rückgabe

`<T> ForkJoinTask<T> submit()` : Führt einen Task asynchron aus, wartet nicht auf Fertigstellung, Rückgabe über `ForkJoinTask`, welcher ein `Future` ist → Rückgabe über `get()`

ForkJoin -Framework Methoden

`void fork()` : Methode um Tasks in 2 Threads auszuführen. Es wird nicht gewartet, bis diese abgeschlossen sind. Das Programm wird forgesetzt.

`void invokeAll()` , `void invoke()` : Methode des `ForkJoinPool` um Tasks in 2 Threads auszuführen. Hier wird jedoch gewartet, bis diese abgeschlossen sind.

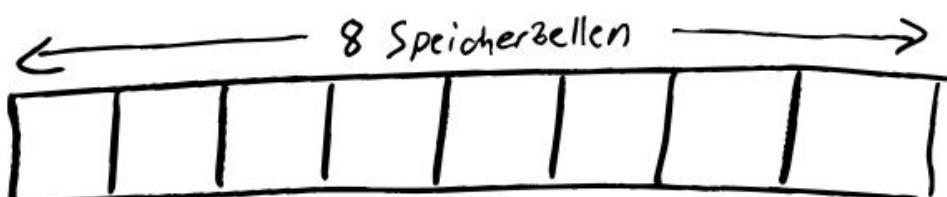
Detaillierter Überblick der Datenstrukturen und Algorithmen

Grundlegende Datenstrukturen

Array

$\Rightarrow O(1)$

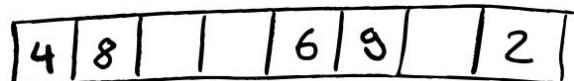
Das Array ist eine einfache Datenstruktur. Sie speichert einzelne Werte in einzelne Zellen. Die Arraygrösse wird bei der Initialisierung gesetzt. Es ist somit eine statische Datenstruktur.



Dieses Array ist leer. Der Speicher für die Daten jedoch wird bei der Erstellung des Arrays bereits reserviert.



→ Ein Array ist dann voll, wenn alle Positionen belegt sind.



→ Leere Plätze mitten im Array sind möglichst zu vermeiden

Eigenschaften

- Das Array ist eine statische Datenstruktur
 - Die Grösse wird bei der Initialisierung festgelegt
- Implizite Datenstruktur
 - Die einzelnen Elemente haben keine Beziehung untereinander
- Direkter Zugriff
 - Auf jedes Element kann über den Index direkt zugegriffen werden
- Reihenfolge
 - Der Array behält die Positionen der Datenelemente unverändert bei

Code

```
int[] array;  
int size;  
  
ArrayConstructor(int capacity) { //Constructor  
    array = new int[capacity];  
    size = 0;  
}
```



Achtung: In Java können keine generischen Arrays erzeugt werden. Arrays und Generics vertragen sich nicht wirklich!

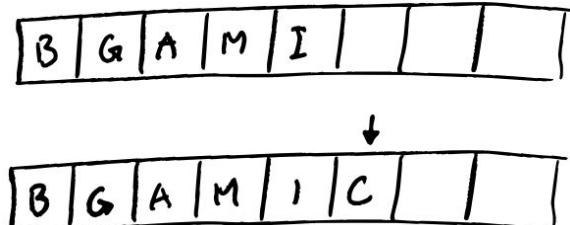
Operationen

Die Implementation der Operationen hängt davon ab, ob das Arrray sortiert oder unsortiert ist.

Einfügen in ein unsortiertes Array

Da das Array unsoritert ist, können wir das neue Item einfach im Array anhängen.

Hier beträgt der Aufwand: $O(1)$

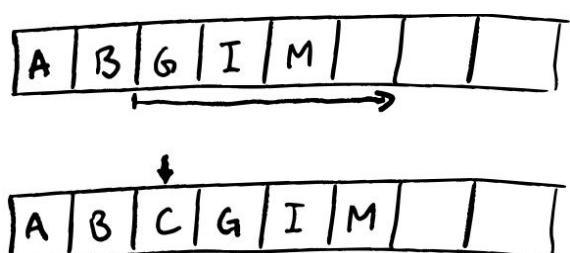


```
void add(char item) {
    ensureCapacity(); //ensure free space
    array[size] = item;
    size++;
}
```

Einfügen in ein sortiertes Array

Der Aufwand um die richtige Position zu suchen (binär): $O(\log_2(n))$, dann aber die restlichen Elemente nach rechts zu verschieben mit $O(n)$. Daraus resultiert ein Aufwand von

$$O(\log_2(n)) + O(n) \rightarrow O(n)$$



Das hinzufügen eines Datenelementes in ein sortiertes Array ist etwas komplizierter. Wir müssen zuerst die richtige Stelle zum einfügen des Elements finden und dann alle nachfolgenden Elemente um eine Stelle verschieben.

```
void add(char item) {
    ensureCapacity(); // ensure free space

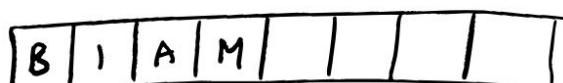
    // find the index where the item should be inserted
    int index = 0;
    while (index < size && array[index] < item) {
        index++;
    }

    // shift array elements to the right to make space for the new item
    for (int i = size; i > index; i--) {
        array[i] = array[i - 1];
    }

    array[index] = item;
    size++;
}
```

Entfernen aus einem unsortierten Array

Wenn wir ein Item aus einem unsortierten Array löschen, nehmen wir das letzte Item und stellen es an die Stelle des gelöschten. So füllen wir die Lücke möglichst effizient auf.



Das Durchsuchen hat einen Aufwand von $O(n)$, da nicht binär gesucht werden kann in einem unsortierten Array. Der Aufwand um die Lücke zu schliessen hat einen Aufwand von $O(n)$. Somit Aufwand: $O(n) + O(1) \rightarrow O(n)$

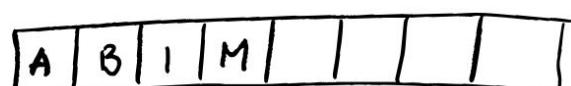
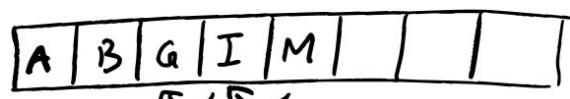
```
void remove(char item) {
    int index = -1;

    for (int i = 0; i < size; i++) {
        if (array[i] == item) {
            index = i;
            break;
        }
    }

    if (index != -1) {
        array[index] = array[size - 1];
        // assign null character
        array[size - 1] = '\0';
        size--;
    }
}
```

Entfernen aus einem sortierten Array

Wenn wir aus einem sortierten Array Elemente entfernen wollen, müssen wir alle Elemente rechts davon nach links verschieben, um die Elemente sortiert zu hinterlassen.



Das Durchsuchen dauert hier schneller, da binär gesucht werden kann $O(\log_2(n))$. Das Verschieben der Elemente jedoch nach links hat eine Komplexität von $O(n)$.

Somit Aufwand: $O(\log_2(n)) + O(n) \rightarrow O(n)$

```
void remove(char item) {
    int index = binarySearch(item); // what binary search does is explained in algos
    if (index != -1) {
        // shift elements to left
        for (int i = index; i < size - 1; i++) {
            array[i] = array[i + 1];
        }
        array[size] = '\0'; // set last cell to null character since it moved left
        size--;
    }
}
```

Stack

$\Rightarrow O(1)$

Ein Stack ist eine Last in first out (LIFO) Datenstruktur. Wie der Namen Stack schon beschreibt, handelt es sich hierbei um einen Stapel von Daten. Ein Stack kann auf unterschiedliche Weisen implementiert werden → [Stack Overflow](#)

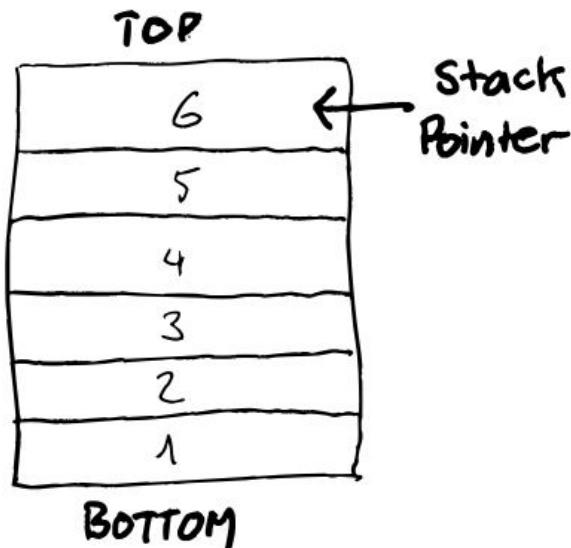
Stack basierend auf LinkedList

- `push()` und `pop()` haben beide Worst Case Complexity von $O(1)$
- Für jedes neue Item auf den Stack muss Memory allocated werden. Dies benötigt jedesmal Ressourcen.

Stack basierend auf dynamic Array (ArrayList)

- `push()` kann eine Worst Case Complexity von $O(n)$ haben, wenn das Array vergrössert und die Daten kopiert werden müssen. → Ansonsten auch $O(1)$

Beispiel, vereinfacht mithilfe Array implementiert



```

int[] stack = new int[capacity];
int pointer = -1; // -1 = empty

void push(int number) {
    pointer++;
    stack[pointer] = number;
}

int pop() {
    int temp = stack[pointer];
    stack[pointer] = 0;
    pointer--;
    return stack[pointer];
}

```



Beim `push()` sollte noch am Anfang gechecked werden, ob der Stack schon voll ist. Wenn dieser voll wäre, sollte eine `StackOverflowException` oder `IndexOutOfBoundsException` geworfen werden. Dieses Problem entsteht nicht, wenn man einen Stack mit dynamischer Grösse hat.

Listen

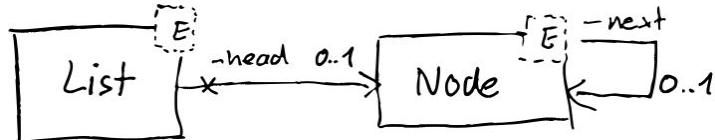
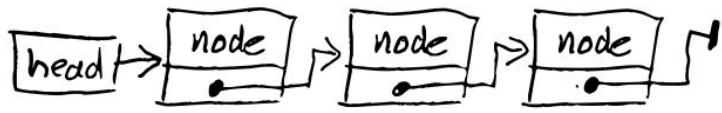
⇒ Entfernen aus Liste immer: $O(n)$

Einfügen	unsortiert	sortiert
Einfach Verkettet	$O(1)$ <code>head</code>	$O(n)$
Doppelt verkettet	$O(1)$ <code>head</code> , <code>tail</code>	$O(n)$

Listen sind explizite Datenstrukturen. Die Liste selber kennt nur den `head` (`head` und `tail` bei double-linked-lists). Die Elemente haben also Referenzen auf die nächsten (und vorherigen in double-linked-lists) Datenelemente.

Einfach verkettete Listen

Auf einfach verketteten Listen hat die Liste eine head-Referenz. Hier ist die Referenz zur ersten Node gespeichert. Jede Node hat dann eine Referenz auf die nächste Node. Die letzte Node hat keine Referenz, da kein Element folgt. Hier können Elemente einfach hinzugefügt werden.

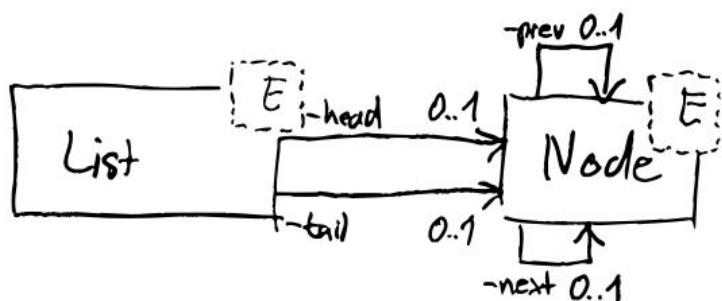
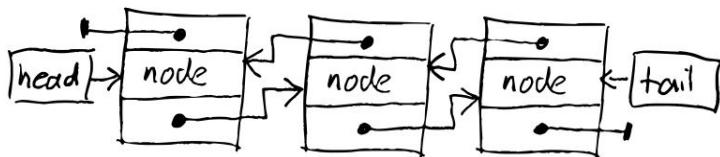


```
class Node{
    private E value;
    private Node next;
}
```

```
class LinkedList{
    private Node head;
}
```

Doppelt verkettete Listen

Auf doppelt verketteten Listen hat die Liste eine head-Referenz auf das erste Datenelement und eine tail-Referenz auf das letzte Element. Die Nodes (Datenelemente) selbst haben immer eine Referenz auf das vorangehende und nachfolgende Element (Vorgänger und Nachfolger). Das erste Datenelement hat keine Referenz auf einen Vorgänger, da keiner existiert. Das letzte Datenelement hat keine Referenz auf einen Nachfolger, da keiner existiert.



```
class Node{
    private E value;
    private Node next;
    private Node prev;
}
```

```
class LinkedList{
    private Node head;
    private Node tail;
}
```

Einfügen in Listen

Die einfachste Art ein Datenelemente einer Liste anzufügen ist es, dieses an den Schluss zu setzen. Dazu erstellt man die entsprechende Node und referenziert die neue Node als Nachgänger auf der zuvor letzten Node.

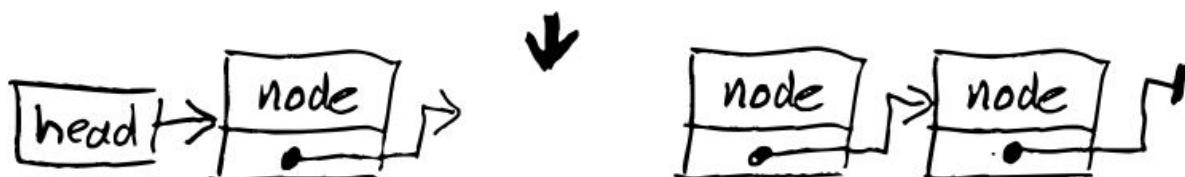
```
public add(E data) {
    Node newNode = new Node(data);

    if (head == null) { // if no head node, add as head node (1. node)
        head = newNode;
    } else {
        Node current = head;

        while (current.next != null) {
            current = current.next; // search end of list
        }
        current.next = newNode; // add new item to end of list
    }
}
```

Wenn wir dies auf eine double-linked list machen würden, müssten wir natürlich, wenn wir das Element hinzufügen, auch dessen `prev`-Referenz auf das vorangehende Element setzen. Auch müssten wir die `tail`-Referenz auf der Liste selbst anpassen.

Wenn wir das Element an einer bestimmten Stelle hinzufügen wollen, weil wir z.B. mit einer sortierten Liste arbeiten, müssen wir einige Referenzen ebenfalls anpassen. Natürlich wäre die Liste jetzt eine andere. z.B. `SortedLinkedList`



```
void add(E item) {
    int index = binarySearch(item);
    Node newNode = new Node(item);

    if (index == 0) { // if new item should be first in list
        newNode.next = head; // reference current head to this new item as next
        head = newNode; // add new item to start and set as head
    } else {
        Node current = head;
        int count = 0;
        while (count < index - 1) { // move to correct position in list
            current = current.next;
            count++;
        }
    }
}
```

```

    newNode.next = current.next; // add next reference of current item to new node,
                                // since it takes its position
    current.next = newNode; // set new node as next reference of current
}
}

```

Bei einer double-linked-list wäre zu beachten, dass nicht nur die `next`-Referenz geändert werden muss, sondern auch die `prev`-Referenz. Auch müsste, wenn das Element zuletzt eingefügt wird, die `tail`-Referenz auf der Liste angepasst werden.

Löschen aus Listen

Beim Löschen aus Listen macht man das gleiche wie beim Einfügen. Jenachdem ob diese sortiert ist oder nicht, ob es eine double-linked-list oder eine einfache linked-list ist, müssen unterschiedliche Referenzen neu gesetzt werden. Das Prinzip bleibt das Gleiche.

Fortgeschrittene Datenstrukturen

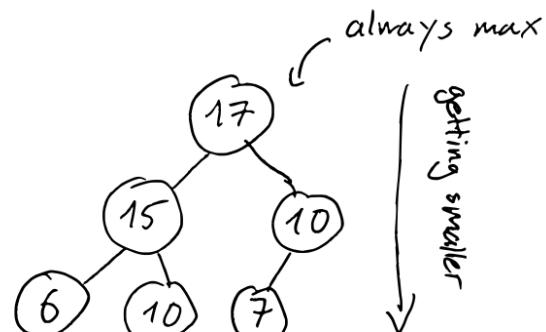
Heap

Der Heap ist generell wie ein Binary Tree aufgebaut. Dieser Binary Tree ist *weakly sorted*. Dabei gibt es zwei unterschiedliche Arten die Daten anzugeben. Einmal als **Max Heap** → Größtes Element zu oberst und als **Min Heap** → Kleinstes Element zu oberst.

Im Max Heap werden Elemente von Oben nach Unten immer kleiner.

Im Min Heap werden Elemente immer von links nach rechts eingefügt. Somit ist der Heap immer ein voller Binary Tree.

Heaps werden als Arrays gespeichert. Wenn ein Heap voll wird, wird das gleiche gemacht wie bei einer ArrayList. Es wird ein Array mit einer doppelten Größe erstellt und die Daten werden in das neue Array kopiert. Danach wird das neue Array verwendet um weitere Daten einzufügen zu können.



Max Heap Example

```

int size = 0;
int capacity = 10;
int[] heapArray = new int[capacity];

```

```

void swap(int indexOne, int indexTwo) {
    int temp = items[indexOne];
    heapArray[indexOne] = heapArray[indexTwo];
    heapArray[indexTwo] = heapArray[indexOne];
}

void ensureCapacity() {
    if (size == capacity) {
        heapArray = Arrays.copyOf(items, capacity * 2);
        capacity *= 2;
    }
}

```



Heapsort Algorithmus: Daten werden in den Heap geladen und es wird so lange `poll()` ausgeführt bis der Heap leer ist. Da `poll()` immer das erste (grösste) Element ausliest, hat man am Ende sortierte Daten vorliegen.

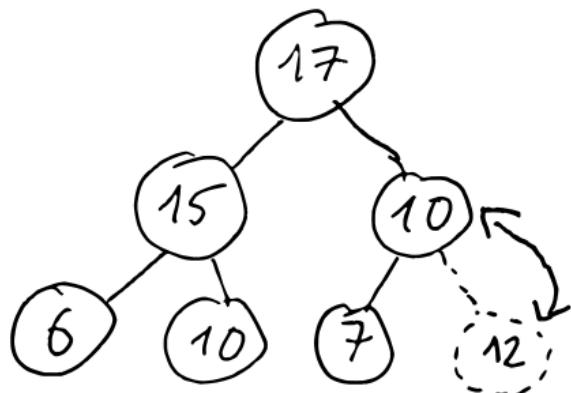
Usecase PriorityQueue:

Der wichtigste Prozess ist immer zuoberst auf dem Heap. Wenn der Scheduler entscheidet, wann welcher Prozess CPU Ressourcen zugewiesen bekommt, kann ein `poll()` gemacht werden, da der wichtigste Prozess immer zuoberst liegt.

Insert

$\Rightarrow O(\log_2(n))$

Beim Einfügen von Elementen wird der Baum von links nach rechts aufgefüllt. Wenn wir also in diesem Beispiel den Wert `12` in den Heap speichern, wird eine neue Node *unten rechts* angehängt. Da nun die Anordnung der Elemente nicht mehr stimmt, muss man die Plätze einiger Elemente tauschen. Hierbei wird überprüft, ob die Parent-Node kleiner als die eingefügte Node ist. Wenn dies zutrifft, werden die Nodes getauscht. Dies macht man so lange, bis die Anordnung stimmt.



```
void add(item) {
    ensureCapacity();
    heapArray[size] = item;
    size++;
    moveUpToRightPos();
}
```

```
void moveUpToRightPos() {
    int index = size - 1; // index of newly added item
    while (hasParent(index) && parent(index) < heapArray[index]) {
        swap(getParentIndex(index), index);
        index = getParentIndex(index); // move index to swapped position
    }
}
```

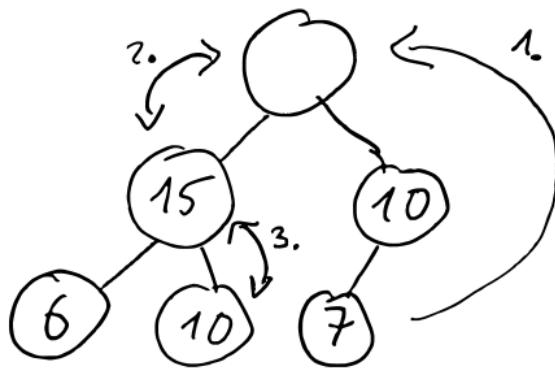
Die Methode `moveUpToRightPos()` tauscht das neu eingefügte Item solange dieses Item eine Parent-Node hat und diese einen kleineren Wert hat, als die eingefügte Node.

Poll

$\Rightarrow O(\log_2(n))$

Poll entfernt das oberste Item in einem Heap. In einer Heap Struktur kann nur das oberste Item entfernt werden. Danach muss der Tree natürlich angepasst werden.

In diesem Beispiel entfernen wir das Item `17` im Heap. Wir nehmen nun das letzte Item im Heap die `7` und setzen diese an die Spitze. Nun stimmt die Anordnung nicht mehr. Dazu tauschen wir nach unten herab das Item immer mit dem grösseren (wenn gleich gross mit dem linken) Child. Dies machen wir, bis das Item die richtige Position erreicht hat.



```
int poll() {
    if (size == 0)
        throw new IllegalStateException();
    int item = heapArray[0];
    heapArray[0] = heapArray[size - 1];
    size--;
    moveDownToRightPos();
    return item;
}
```

```
void moveDownToRightPos() {
    int index = 0;
    while (hasLeftChild(index)) { // works because its a full tree (filled left -> right)
        int smallerChildIndex = getLeftChildIndex(index);
        if (hasRightChild(index) && rightChild(index) > leftChild(index)){
            biggerChildIndex = getRightChildIndex(index);
        }

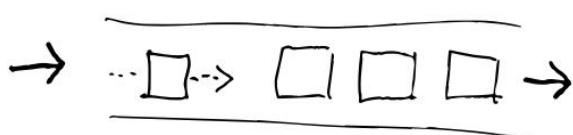
        if (heapArray[index] > heapArray[biggerChildIndex]) {
            break;
        } else {
            swap(index, biggerChildIndex);
        }
        index = biggerChildIndex; // move index to swapped position
    }
}
```

Queue (hier: CircularQueue)

$\Rightarrow O(1)$



Oft als `CircularQueue` implementiert. Im Beispiel wird ebenfalls die Queue als `CircularQueue` mithilfe eines Ringbuffers implementiert.



Eine Queue verfolgt das einfache Prinzip von FIFO (First In First Out). Bedeutet, dass es funktioniert wie eine simple Warteschlange.

CircularQueue Implementation mit einer Liste

Der Vorteil bei der Implementation mit der Liste ist, dass der Ringbuffer einfach wachsen kann. Es wäre hier eine doppelt verkettete Liste notwendig.

CircularQueue Implementation mit einem Array

Bei der Implementation der Queue mit einem Ringbuffer, werden die Daten eigentlich in einem Array gespeichert. Man merkt sich in diesem Array zwei Werte. Einmal den Start der Queue (Front) und das Ende der Queue (Rear). Wenn man ein weiteres Element hinzufügt, wird der Pointer der auf das Wartschenlangen-Ende zeigt um eine Stelle verschoben. Wenn man ein Element aus der Warteschlange holt, wird der Platz in dieser Array-Zelle frei und der Front Pointer verschiebt sich auf das nächste Datenelement im Array.

```
private T[] buffer;
private int front;
private int rear;
private int size;
private int capacity;

// here constructor
public RingBufferQueue(int capacity) {
    this.capacity = capacity;
    buffer = (T[]) new Object[capacity];
    front = 0;
    rear = -1;
    size = 0;
}
```

Helper-Functions

```
boolean isFull() {
    return size == capacity;
}

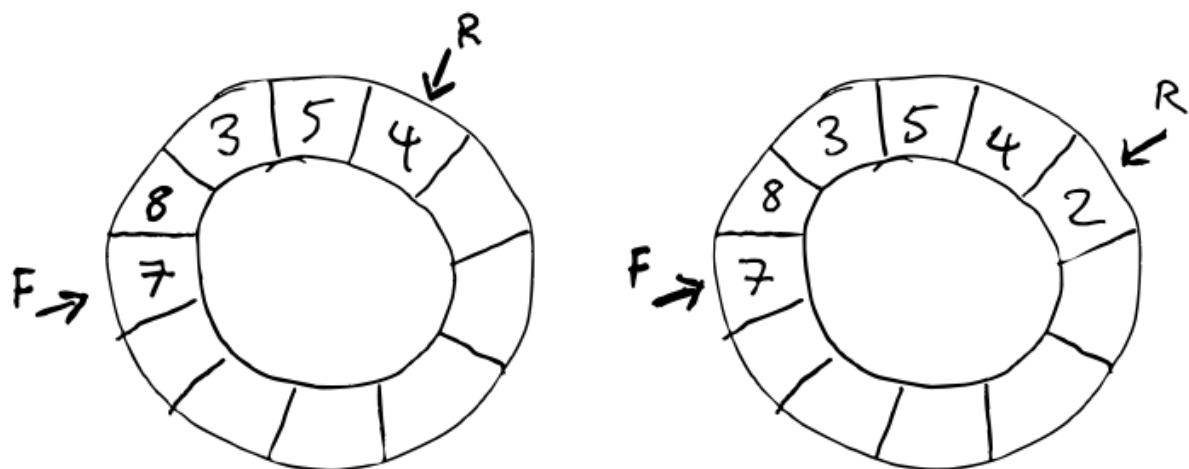
boolean isEmpty() {
    return size != capacity;
}
```

Items der Queue zufügen

Die Stelle in die ein neues Element gespeichert wird, muss errechnet werden. Da das Array als Ringbuffer genutzt wird und somit einen geschlossen Kreis ergeben soll.

Wenn also der Pointer am Array-Ende ist, soll er wieder auf den Array Start springen.

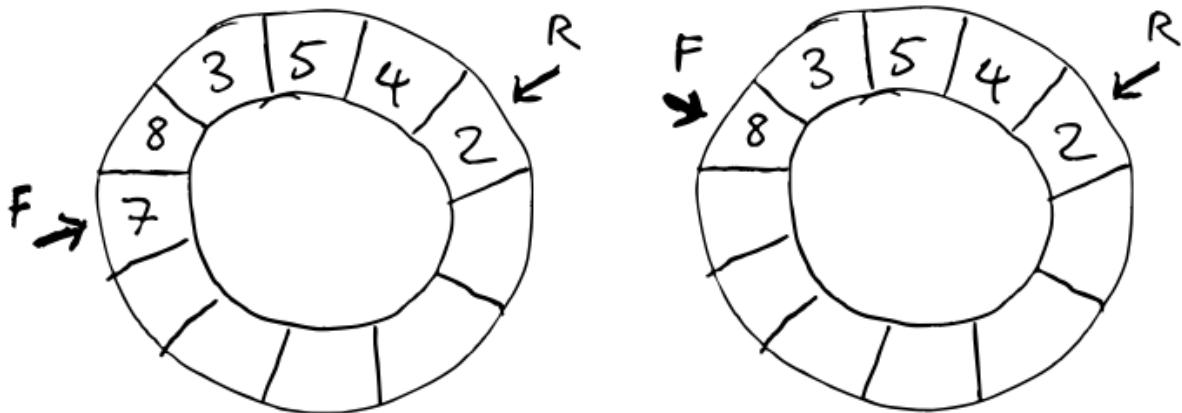
Um dies einfach zu errechnen kann man folgende Formel nutzen: $(\text{rear} + 1) \% \text{arraylength} \rightarrow \text{new rear value}$



```
void enqueue(T item) {
    if (isFull()) {
        throw new RuntimeException("Queue is full. Unable to enqueue");
    }
    rear = (rear + 1) % capacity;
    buffer[rear] = item;
    size++;
}
```

Items aus der Queue entfernen

Der Front Pointer zeigt immer auf die Stelle, von der beim nächsten dequeue das Element gelesen und entfernt wird. Danach muss der Pointer auf die nächste Stelle im Array zeigen. Auch hier muss diese Stelle errechnet werden, da das Array ja einen RingBuffer ergeben soll. Dies macht man wie folgt: $(\text{front} + 1) \% \text{arraylength} \rightarrow \text{new front value}$



```

T dequeue() {
    if (isEmpty()) {
        throw new RuntimeException("Queue is empty. Nothing to dequeue");
    }
    T item = buffer[front];
    buffer[front] = null;
    front = (front + 1) % capacity;
    size--;
    return item;
}

```

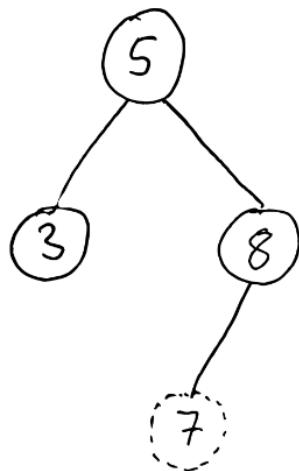
Binary Search Tree (Binärer Suchbaum)

Der binäre Baum ist ein Baum mit der Ordnung 2. Jeder Knoten kann also maximal 2 Kinder haben.

Der binäre Suchbaum enthält als Erweiterung zu den eigentlichen Datenelementen einen Schlüsselwert, nach welchem die Datenelemente im Baum geordnet und gesucht werden können. z.B wenn wir einen Baum mit Studenten füllen, könnte man als Schlüsselwert deren Matrikelnummer verwenden. Nach dieser kann man effizient suchen und ordnen.

Im Prinzip werden im binären Suchbaum die Werte der Schlüssel von links nach rechts immer grösser. Wenn man den binären Suchbaum Inorder durch traversiert, erhält man eine geordnete Liste der Daten.

Einfügen in den Binary Search Tree



Um Datenelemente in den Binary Search Tree einzufügen, kann die gleiche Vorgehensweise wie bei der Binary Search verwendet werden.

1. Element mit der Wurzel vergleichen, wenn als Wurzel dann nach rechts traversieren, wenn als Wurzel nach links
2. Schritt 1 solange wiederholen bis wir an ein Leaf kommen
3. Element jetzt am Leaf anhängen. Ließ wird zu einem inneren Knoten und neues Element wird zum Leaf.

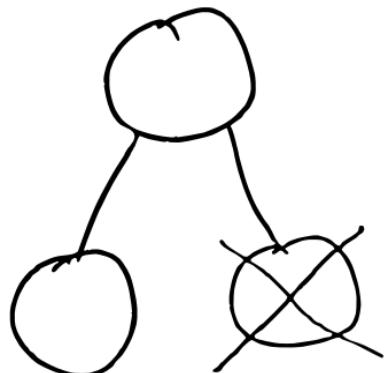
```
Node add(Node currentNode, T value) {  
    if (currentNode == null) {  
        return new Node(value); // return new item on leaf or empty tree  
    }  
  
    if (item.equals(currentNode))  
        return currentNode; // dont change tree if values already in tree  
  
    if (value < currentNode.value) {  
        currentNode.left = addNode(currentNode.left, value);  
    } else if (value > currentNode.value) {  
        currentNode.right = addNode(currentNode.right, value);  
    }  
  
    return currentNode;  
}
```

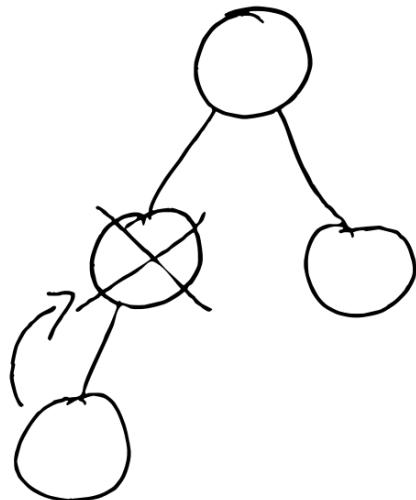
Entfernen aus dem Binary Search Tree

Es gibt drei unterschiedliche Szenarien beim Entfernen von Datenelementen aus einem Binary Search Tree.

I. Leaf entfernen

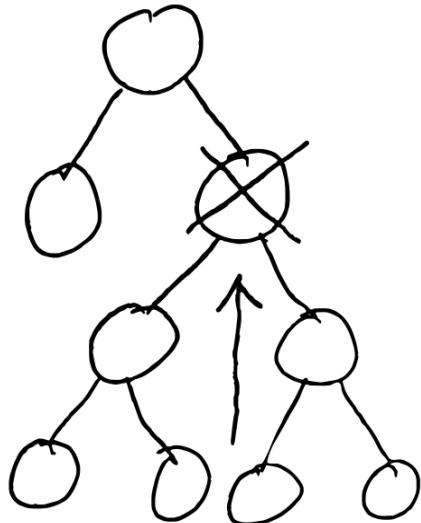
Das erste Szenario ist das einfachste. Ein Leaf soll aus dem Tree entfernt werden. Dazu löscht man die Referenz im Parent. Mehr muss nicht gemacht werden.





II. Knoten mit einem Kind entfernen

Im zweiten Szenario soll man einen Knoten mit einem Kind entfernen. Auch dies ist einfach. Wir referenzieren das Kind vom zu lösrenden Node als das Kind vom Parent Node. Das Kind rutscht also eine Ebene nach oben und wird zum alleinigen Kind vom Vater der zu löschenen Node.



III. Knoten mit zwei Kindern entfernen

Wenn ein innerer Knoten mit zwei Kindern entfernt wird, tauschen wir diesen mit dem *linksten* Knoten aus dem rechten Subtree des zu lösrenden Knoten aus. Dieses Element ist dem zu löschenen Element am nächsten.

Wir nehmen also das kleinste Element, welches grösser ist als das zu löschenen und tauschen die Plätze. So muss der Rest des Baumes nicht verändert werden, da die Anordnung der Datenelemente wieder stimmt.

Hashbasierte Datenstrukturen

Hashtable

Die Grundidee hinter hasbasierten Datenstrukturen ist es, die Hashwerte der Datenelemente als Index zu verwenden. So kann schnell gesucht, hinzugefügt und gelöscht werden.

Beispiel

Wir wollen Personen, die in einen Raum kommen in eine Liste aufnehmen. Diese Personen haben Namen und sie haben IDs. Diese betreten wie folgt den Raum:

Laura (42), Liam (38), Julia (13), Milan (80), Emilia (12), Jonas (17)

Die Datenstruktur danach sieht so aus:

0	1	2	3	4	5	6	7	8	9
80		42	13	12			17	38	

Berechnung des Index

Der Index wird mithilfe des Hashwerts und der Länge des Arrays berechnet

$$\text{index} = \text{ID \% Länge des Arrays}$$

Für Laura im obigen Beispiel: $\text{Index} = 42 \% 10 = 2$

Sondierung

Hash-Kollisionen treten dann auf, wenn zwei Datenelemente den gleichen Hashwert erhalten. Hash-Kollisionen sind jedoch nicht das einzige Problem, beim nutzen von Hashtabellen. Auch das Berechnen der Index mit zwei unterschiedlichen Hashwerten, kann den selben Index ergeben. In diesen Fällen, müssen wir sondieren.

Bei der Sondierung füllen wir die Zellen hinter dem berechneten Speicherort (Array-Zelle) ein, sofern die berechnete Stelle bereits besetzt ist.

Um diesen Wert wieder zu finden (beim Suchen), müssen wir sicherstellen, dass wir den berechneten Speicherort prüfen, ob das gesuchte Datenelement sich dort befindet. Wenn dies nicht der Fall wäre, durchsuchen wir alle nachfolgenden Zellen,

bis wir eine leere finden. Denn solange wir keine leere Zelle vorfinden, könnte es sein, dass der Wert sondiert wurde und nach hinten verschoben wurde. Deswegen kann auch Elemente aus einer Sondierungskette nicht so einfach löschen. Denn dann fänden wir die Werte hinter diesem nicht mehr.

Elemente aus Sondierungsketten entfernen

Wenn Elemente in einer Hashtable entfernt werden, muss ein Tombstone (Grabstein) hinterlegt werden. Denn wenn sich dieses Element in einer Sondierungskette befindet und wir dieses einfach entfernen würden, könnten alle Elemente in dieser Sondierungskette nach dem entfernten nicht mehr gefunden werden, da die Suche bei der ersten leeren Speicherzelle beendet wird. Der Tombstone erlaubt es neue Werte in diese Zelle zu speichern. Gleichzeitig aber ist kein Wert mehr gespeichert.

0	1	2	3	4	5	6	7	8	9
80		42	T	12			17	38	

Grundlegende Sortieralgorithmen

Algorithmus	Komplexität	Stabil / Instabil
Insertion Sort	$O(n^2)$	stabil
Selection Sort	$O(n^2)$	instabil

Insertion Sort (Direktes Einfügen)

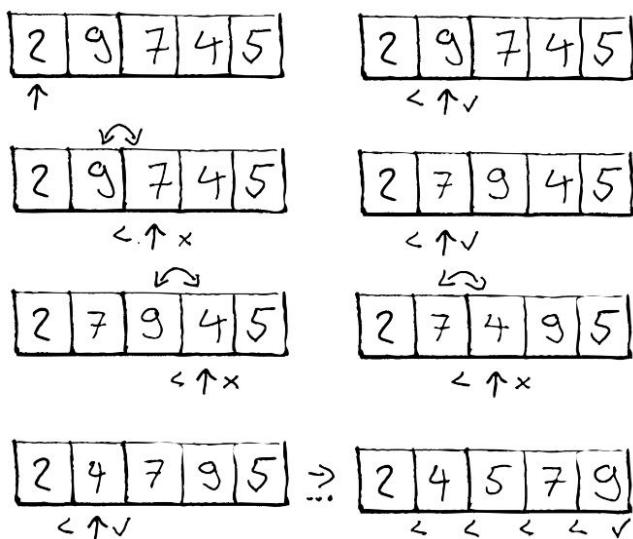
⇒ Best Case: $O(n)$

⇒ Worst Case: $O(n^2)$

⇒ Average Case: $O(n^2)$

Beim Insertionsort gehen wir von links nach rechts und vergleichen einen Wert mit seinem linken (Vorgänger). Wenn der Vorgänger grösser ist, müssen wir die Elemente austauschen. Wir vergleichen so lange nach links, bis wir die richtige Stelle gefunden haben. Die restlichen Werte werden um einen Platz nach rechts geschoben.

Beispiel



1. Wir vergleichen ob $2 \leq 9$, dies ist korrekt, also muss nichts geändert werden.

2. Wir vergleichen ob $9 \leq 7$, dies ist falsch. Wir tauschen die Werte und vergleichen nun ob $2 \leq 7$. Dies ist korrekt, wir können fortfahren.

3. Wir vergleichen ob $9 \leq 4$, dies ist falsch. Wir tauschen und vergleichen ob $7 \leq 4$, was wieder falsch ist, also müssen wir nochmals tauschen. $2 \leq 4$ ist korret. Die 4 ist jetzt an der richtigen Stelle.

4. Dies wiederholen wir für die 5 und haben dann ein sortiertes Array.

```

public static void insertionSort(int[] arr) {
    int length = arr.length;

    for (int i = 1; i < length; i++) {
        int currentElement = arr[i];
        int leftElementIndex = i - 1;

        // shift elements to right
        while (leftElementIndex >= 0 && arr[leftElementIndex] > currentElement) {
            arr[leftElementIndex + 1] = arr[leftElementIndex];
            leftElementIndex--;
        }

        arr[leftElementIndex + 1] = currentElement;
    }
}

```

Selection Sort (Direktes Auswählen)

$$\Rightarrow O(n^2)$$

Beim Selection Sort wird zuerst das kleinste Datenelement gesucht. Dieses wird an den Start gelegt. Nun wird in den restlichen Daten wieder das kleinste Datenelement gesucht und an die 2. Stelle gelegt. Dies wird gemacht, bis die Daten sortiert vorliegen.

2	9	7	4	5
↑				

2	9	7	4	5
	↑			

2	4	9	7	5
	↑			

2	4	5	9	7
		↑		

2	4	5	7	9
			↑	

1. Kleinstes Datenelement im Array ist die 2. Hier ist sie schon an der richtigen Stelle.

2. Im restlichen Array ist das kleinste Datenelement die 4. Sie wird an die 2. Stelle gesetzt

3. Im restlichen Array ist das kleinste Datenelement die 5. Sie wird an die 3. Stelle gelegt

4. Dies wird gemacht bis das Array sortiert ist.

```

public static void selectionSort(int[] arr) {
    int length = arr.length;

    for (int i = 0; i < length - 1; i++) {
        int minIndex = i;

        // Find lowest value
        for (int j = i + 1; j < length; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }

        // Swap lowest value with value at current position to insert
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}

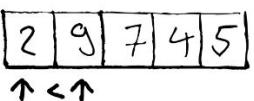
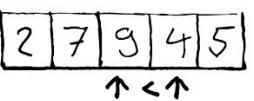
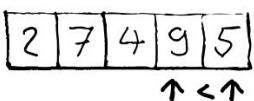
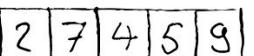
```

Bubble Sort

$\Rightarrow O(n^2)$

Beim Bubblesort vergleicht man immer zwei Elemente nebeneinander und tauscht diese, wenn notwendig. Dies macht man einmal von links nach rechts durch. Danach sind jedoch die Elemente noch nicht ganz sortiert. Dies muss so lange wiederholt werden, bis alle Elemente sortiert sind.

Bubble Phase 1

		2 < 9 → korrekt, nicht tauschen
		9 < 7 → falsch, tauschen 9 < 4 → falsch, tauschen 9 < 5 → falsch, tauschen
		

Bubble Phase 2

2	7	4	5	9
↑ < ↑				

2	7	4	5	9
↑ < ↑				

2	4	7	5	9
↑ < ↑				

...

2 < 7 → korrekt, nicht tauschen

7 < 4 → falsch, tauschen

7 < 5 → falsch, tauschen

...

Bubble Phase X - End

2	4	5	7	9
↑ < ↑				

Am Ende aller Bubble Phasen ist das Array sortiert.

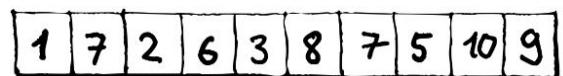
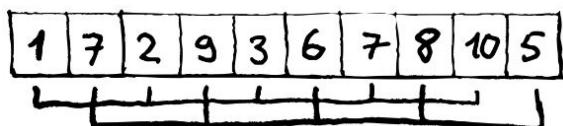
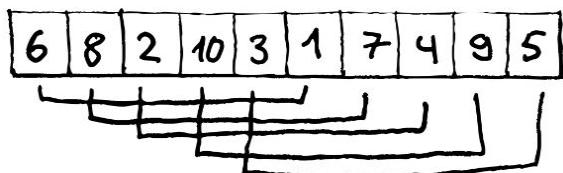
```
public static void bubbleSort(int[] arr) {
    int length = arr.length;

    for (int i = 0; i < length - 1; i++) {
        for (int j = 0; j < length - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Exchange items if order not correct
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Shellsort

$$\Rightarrow O(n^{1.5})$$

Beim Shellsort, sortiert man zuerst die Datenelemente grob durch, indem man Daten-Prächen erzeugt. Anfangs ist der Datenpaar-Abstand = arraylength/2 Durch dieses Verfahren, muss viel weniger getauscht werden, als bei dem Insertion-Sort.



1. Paare im 5er Abstand werden verglichen und vertauscht wenn notwendig
2. Abstand wird nochmals halbiert, jetzt vergleicht man im Abstand 2 alle Paare und vertauscht diese, wenn notwendig
3. Im letzten Schritt führt man einen normale Insertion-Sort aus, da der Abstand = 1 wird.



Code nicht relevant für MEP

Höhere Sortieralgorithmen

Algorithmus	Ordnung	stabil / instabil	Worst Case O
Mergesort	$O(n \cdot \log n)$	stabil	\leftarrow
Heapsort	$O(n \cdot \log n)$	instabil	\leftarrow
Quicksort	$O(n \cdot \log n)$	instabil	$O(n^2)$

Suchalgorithmen

Binary Search

$\Rightarrow O(\log_2(n))$ oder auch $O(\text{ld}(n))$

ld steht für Logarithmus Dualis

Binary Search ist ein effizienter Suchalgorithmus für sortierte Listen oder Arrays. Es vergleicht den gesuchten Wert mit dem mittleren Element und schließt die Hälfte der Liste aus, in der sich der Wert nicht befinden kann. Der Algorithmus wiederholt diesen Prozess, bis der Wert gefunden wird oder festgestellt wird, dass er nicht vorhanden ist.

Das Vorgehen ist folgendermassen:

1. Trennelement in der Mitte der sortierten Daten ermitteln
2. Mit dem gesuchten Wert vergleichen, zuerst $=$, dann $>$ oder $<$
3. In der verbleibenden Hälfte Prozess wiederholen

Beispiel: Wir suchen den Wert 3

1	2	3	4	5	6	7
---	---	---	---	---	---	---

$$\text{mid} = 0 + \frac{6-0}{2} = 3, \text{ Index: } 3$$

Wir nehmen den mittleren Wert in der Datenstruktur, die 4. Wir vergleichen zuerst ob die 4 unserem gesuchten Wert entspricht. Dies scheint nicht der Fall zu sein. Die 4 ist grösser als der gesuchte Wert, somit können wir die rechte Hälfte der Daten ignorieren (da diese ja alle auch grösser sind).

1	2	3	4	5	6	7
---	---	---	---	---	---	---

$$\text{mid} = 0 + \frac{2-0}{2} = 1, \text{ Index: } 1$$

Nun wiederholen wir den Prozess.
Mittleres Element bestimmen → 2. Die 2 ist nicht der gesuchte Wert. Die 2 ist kleiner als der gesuchte Wert → rechte verbleibende Hälfte durchsuchen.

1	2	3	4	5	6	7
---	---	---	---	---	---	---

Wir nehmen wieder den mittleren Wert (dieses mal gibt es nur einen) → 3. Die 3 ist der gesuchte Wert. Wir haben den Wert gefunden.

$$\text{mid} = 2 + \frac{2-2}{2} = 2, \text{Index: } 2$$

Code

In diesem Codebeispiel arbeiten wir mit einem Array!

```
static int binarySearch(int[] array, int target) {  
    int left = 0; // set left init boundary  
    int right = array.length -1; // set right init boundary  
  
    while (left <= right) { // when right > left boundary: all elements searched  
        int mid = left + (right - left) / 2;  
        if (array[mid] == target) {  
            return mid; //Wert gefunden  
        } else if (array[mid] < target) {  
            left = mid + 1;  
        } else {  
            right = mid - 1;  
        }  
    }  
    return -1; //Zielwert nicht gefunden  
}
```

Graphtheorie



Auf PDF werden Diagramme als Mermaid Code angezeigt. Auf Github bei diesem PDF können die Graphen angeschaut werden.

Grundlagen

Ein Graph beschreibt die Beziehung zwischen Objekten. Die Objekte heissen Knoten, die Beziehungen werden mithilfe von Kanten dargestellt.

```
graph RL  
K1(( Knoten 1 )) ---|Kante| K2(( Knoten 0))
```

Zum Beispiel kann man ein Transportnetz als Graph darstellen. Die Knoten hierbei wären die Städte, und die Kanten die Verbindungen zwischen den Städten.

```
graph LR  
A((Atlantis)) --- B((Coralville))  
A --- C((Seashell Bay))  
B --- C
```



Code nicht relevant für MEP

Gerichtete und ungerichtete Graphen

Gerichtete Graphen

Bei einem gerichteten Graphen kann man an einer Kante nur in eine Richtung gehen

```
graph LR  
Knoten1((1))--> Knoten2((2))
```

Bei gerichteten Graphen heissen 2 Knoten **stark verbunden**, wenn eine

Ungerichtete Graphen

Bei einem ungerichteten Graphen kann man an einer Kante in beide Richtungen gehen

```
graph LR  
Knoten1((1))--- Knoten2((2))
```

Ein ungerichteter Graph kann immer als gerichteter Graph dargestellt werden,

Verbindung in beide Richtungen zwischen den Knoten existiert:

```
graph LR  
Knoten1((1))--> Knoten2((2))  
Knoten2 --> Knoten1
```

da man eine Kante mit 2 Pfeilen ersetzen kann.

Grad

Jeder Knoten in einem Graphen hat einen Grad. Dieser Grad bestimmt wie viele Kanten von dem Knoten ausgehen.

Für gerichtete Graphen:

```
graph LR
A((A)) --- B((B)) --- C((C)) --- D((D))
```

Knoten	Grad
A	1
B	2
C	2
D	1

Für ungerichtete Graphen:

```
graph LR
A((A)) --> B((B)) --> C((C)) --> D((D))
```

Knoten	Eingangsgrad	Ausgangsgrad
A	0	1
B	1	1
C	1	1
D	1	0

Markierte Graphen

In markierten Graphen tragen die Kanten zusätzliche Informationen.

```
graph RL
K1(( C)) ---|4| K2(( A))
K2(( B)) ---|8| K3(( A))
K3 --- |6| K1
```

Ein Beispiel bei einem Transportnetz wäre hierbei die Reisedauer zwischen den Städten.

Formale Beschreibung

Graph

- Als 2-Tupel: $G = (V, E)$
- Knotenmenge V (Knoten = Vertex)
 $V = \{a, b, c, d, e\}, |V| = 5$
- Kantenmenge E (Kante = Edge), $E \subseteq V \times V$
 $E = \{(a,b), (b,a), (b,c), (a,c), (c,c), (c,d), (a,d)\}, |E| = 7$

Darstellung von G :

```
graph LR
E((E))
A((A)) --> B((B))
A --> D
A --> C
B --> A
B --> C((C))
C --> C
C --> D((D))
```

Teilgraph

- Als 2-Tupel: $G' = (V', E')$
- Knotenmenge $V' \subseteq V$
 $V' = \{a, b, c\}$
 $V' \subseteq V = \{a, b, c, d, e\}$
- Kantenmenge $E' \subseteq E$ und $E' \subseteq V' \times V'$
 $E' = \{(a, b), (b, c), (a, c)\}$
 $E' \subseteq E = \{(a, b), (b, a), (b, c), (a, c), (c, c), (c, d), (a, d)\}$

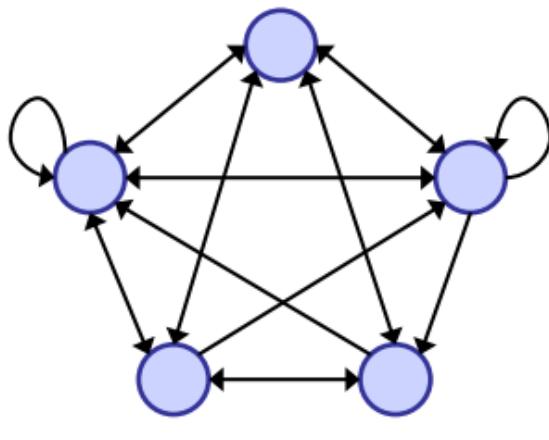
Darstellung von G' :

```
graph LR
A((A)) --> B((B))
A --> C
B --> C((C))
```

Dicht vs Dünn besetzter Graph

Dicht

$$|E| \approx |V|^2$$

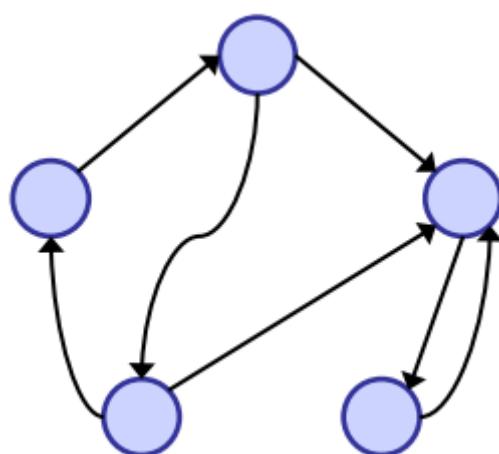


$$|E| = 19 \text{ und } |V| = 5$$

$$19 \approx 25$$

Dünn

$$|E| \ll |V|^2$$



$$|E| = 7 \text{ und } |V| = 5$$

$$7 \ll 25$$

Eine andere Möglichkeit dies einzuschätzen ist es die tatsächlichen Kanten durch die möglichen Kanten zu Teilen.

$$Q = \frac{\text{TatsächlicheKanten}}{\text{MöglicheKanten}}$$

Wenn dies ≥ 0.5 ist spricht man von einem dichten Graphen, da mindestens 50% der möglichen Kanten tatsächlich existieren.



Letztere Methode ist nicht offizieller Inhalt des Moduls Algorithmen und Datenstrukturen

Pfade und Zyklen

Falls ein Graph eine Verbindung (Kante) zwischen $A \rightarrow B \rightarrow C \rightarrow D$ hat, existiert ein Pfad/Weg von A nach D. Dieser hat dann die Länge 3.

Ein Pfad von Knoten x nach Knoten x heisst Zyklus.

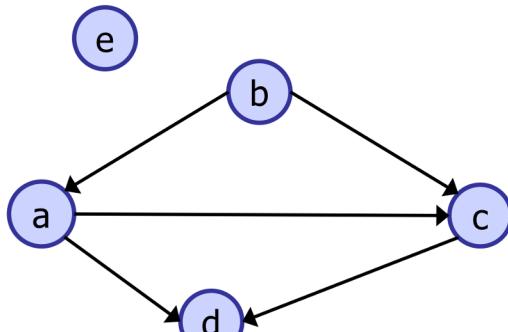
```

graph LR
    A((A)) --> A
  
```

Gerichtete Graphen

- sind zyklenfrei oder azyklisch, wenn sie keine Zyklen enthalten.

```
graph LR
    A((A)) --> B((B))
```



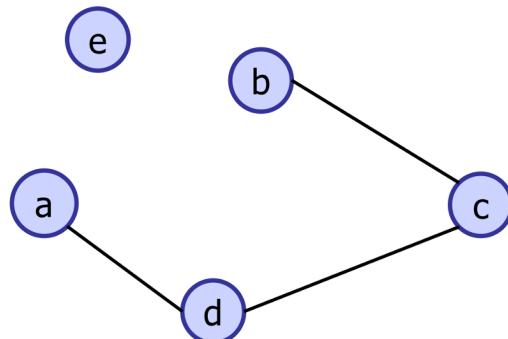
Gerichteter Zyklusfreier Graph

Ungerichtete Graphen

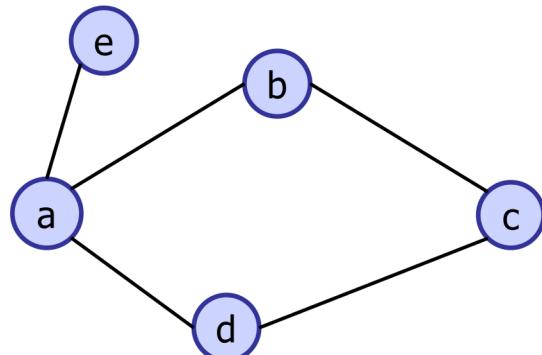
- sind zyklenfrei oder azyklisch, wenn es zwischen zwei beliebigen Knoten höchstens einen Pfad gibt (ohne triviale Zyklen)
- Triviale Zyklen: (a,b,a), (a,a)

```
graph LR
    A((A)) --- B((B))
    C((C)) --- C
```

Bei n Knoten und mindestens n Kanten, gibt es einen Zyklus



Ungerichteter Zyklusfreier Graph

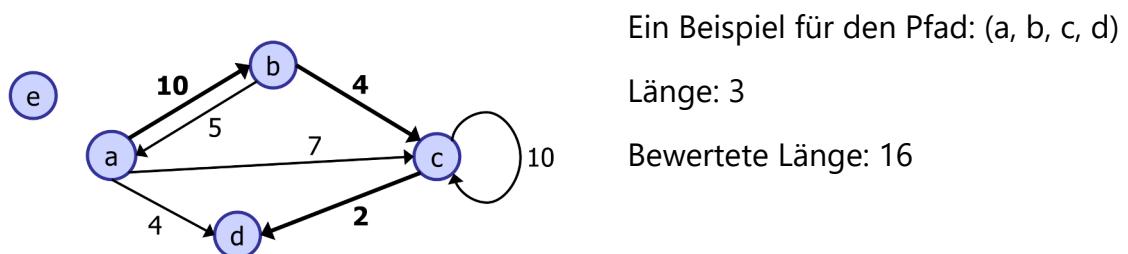


Ungerichteter Graph mit mind. Zyklus

Bewertete Graphen

Formale Beschreibung:

- Als 3-Tupel: $G = (V, E, f(E))$
- Die Gewichtsfunktion $f(E)$, ordnet jeder Kante ein Gewicht (einen Wert) zu.
- Es resultieren bei einem bewerteten Graphen 2 Dinge:
 - Eine **Länge**: Anzahl Kanten zwischen zwei Knoten
 - Eine **bewertete Länge**: Summe der Bewertungen zwischen zwei Knoten, auch als Entfernung gekennzeichnet

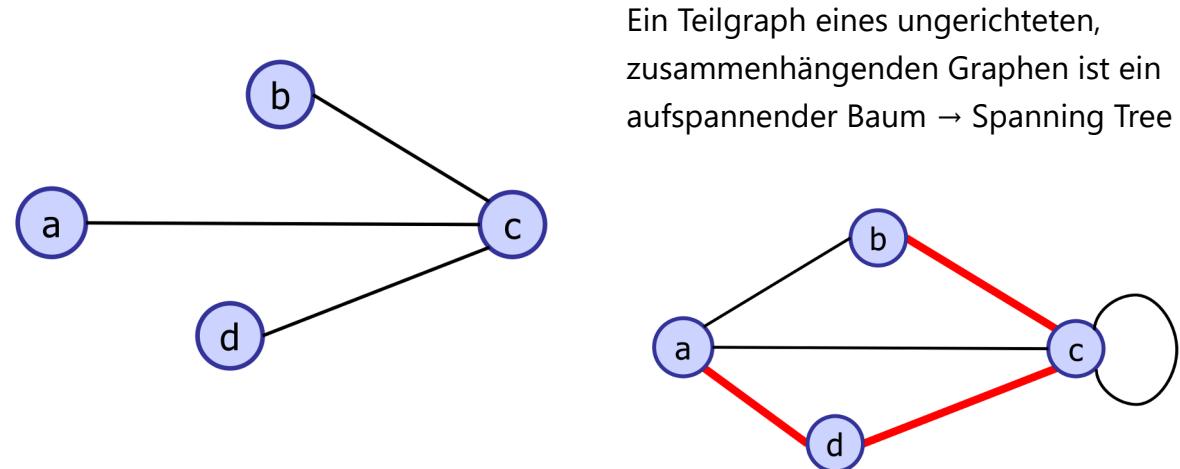


Baum

Ein ungerichteter, zusammenhängender und zyklenfreier Graph ist ein Baum.

- ungerichtet: Kanten sind symmetrisch.
- zusammenhängend: Graph zerfällt nicht in Komponenten.
- zyklenfrei: Zwischen je zwei Knoten gibt es genau einen Pfad.

Ein Baum mit n Knoten hat $n-1$ Kanten



Folgende Graphen-Algorithmen liefern mitunter einen Spanning-Tree für einen ungerichteten Graphen:

- Breitensuche
- Tiefensuche
- Algorithmus von Dijkstra

Problemlösung mittels Graphen

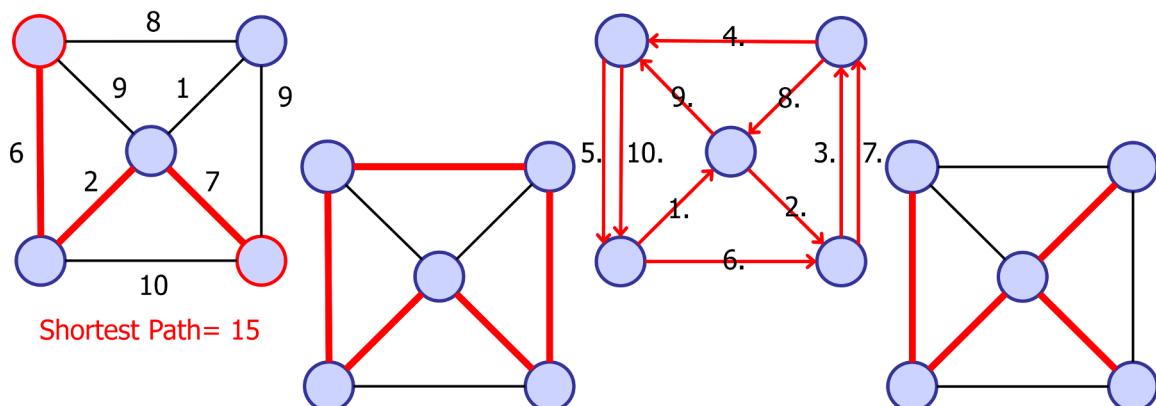
Typische Vorgehensweise:

1. Problem als Graphen modellieren.
2. Zielfunktion als Eigenschaft des Graphen formulieren.
3. Mit Hilfe eines Graphen-Algorithmus das Problem lösen.

Grundtypen der Zielfindung / Zielfunktion

Pfad, d.h. Kanten finden:

- Kürzesten Weg finden
- Rundweg via alle Knoten finden (kürzester Rundweg → Travelling Salesman Problem, TSP)
- Jede Kante besuchen (Chinese Postman Problem)
- Alle Knoten aufspannen (Spanning-Tree)



Kanten Werte zuordnen:

- Max. Flow, d.h. max. «Durchfluss» bei gegebenen Kapazitäten finden.

- Min. Cost Flow, d.h. «Angebot (+), Nachfrage (-)» befriedigen bei min. Kosten.

Knoten Werte zuordnen:

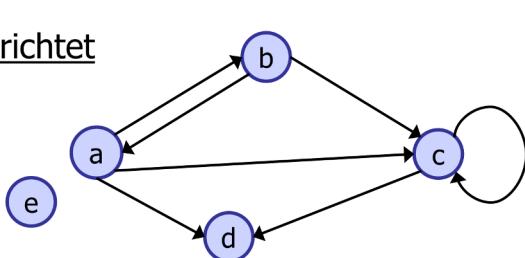
- Coloring, d.h. alle verbundenen Kontenpaare besitzen unterschiedliche Farben.
- Scheduling, d.h. mögliche Abfolge finden, trotz Abhängigkeiten/Restriktionen.

Adjazenzmatrix

- Speicherkomplexität $\rightarrow O(|V|^2)$

Gerichtet, unbewertet

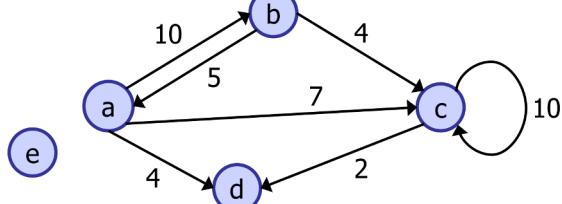
gerichtet



-	a	b	c	d	e
a	0	1	1	1	0
b	1	0	1	0	0
c	0	0	1	1	0
d	0	0	0	0	0
e	0	0	0	0	0

Für gerichtete, unbewertete Graphen zeigt die Matrix ob eine Verbindung von Knoten x zu Knoten y existiert (True, False).

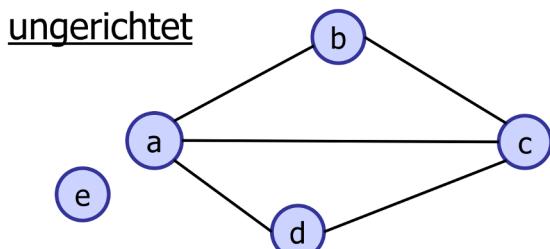
gerichtet, bewertet



-	a	b	c	d	e
a	∞	10	7	4	∞
b	5	∞	4	∞	∞
c	∞	∞	10	2	∞
d	∞	∞	∞	∞	∞
e	∞	∞	∞	∞	∞

Für gerichtete, bewertete Graphen zeigt die Matrix die Bewertung von Knoten x zu y.

ungerichtet, unbewertet

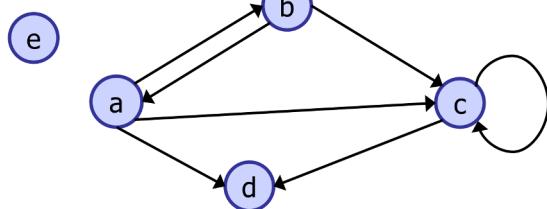


-	a	b	c	d	e
a	0	1	1	1	0
b	1	0	1	0	0
c	1	1	0	1	0
d	1	0	1	0	0
e	0	0	0	0	0

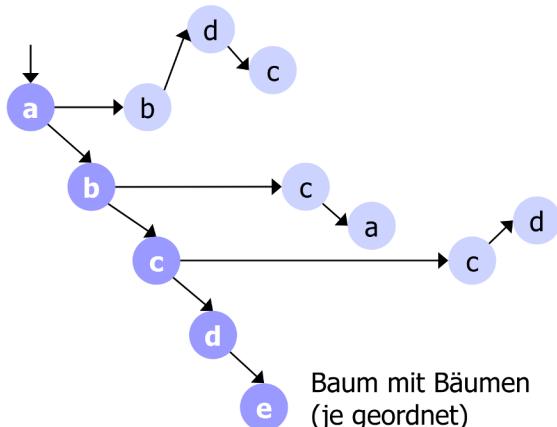
Für ungerichtete, unbewertete Graphen zeigt die Matrix ob eine Verbindung existiert. Hierbei sehen wir auch, dass sich die Matrix in der Diagonale spiegelt.

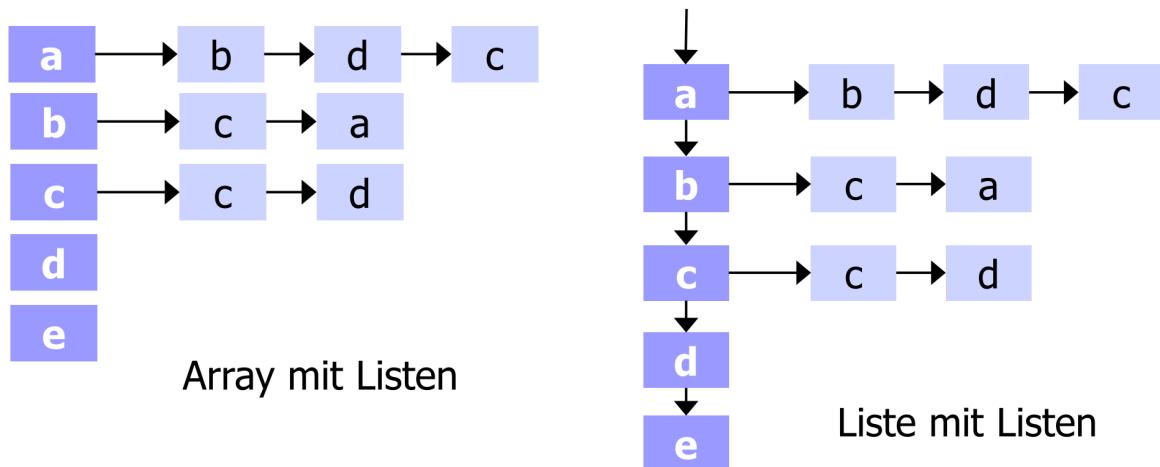
Adjazenzlisten

- Speicherkomplexität $\rightarrow O(|V|+|E|)$



Es gibt unterschiedliche Möglichkeiten Adjazenzlisten umzusetzen. Hier sind drei davon:





Traversieren

- Bei vielen Graphalgorithmen geht man durch alle Knoten über alle Kanten durch.
- Um bei Graphen mit Zyklen nicht in Schleifen zu kommen muss man diese markieren
- Man unterscheidet zwischen folgenden Strategien
 - Breitensuche
 - Tiefensuche

Breitensuche (Breadth First Search, BFS)

Die Breitensuche ist vor allem dann gut, wenn man den Shortest Path in einem unmarkierten Graphen finden möchte.

Bei der Breitensuche werden zuerst die angrenzenden Knoten angeschaut, bevor in einer tieferen Ebene gesucht wird.

Die Zeitkomplexität beträgt → **O(|V| + |E|)**

Funktioniert nach dem **FIFO** Prinzip. First in - First out. → Wie eine **Queue**.

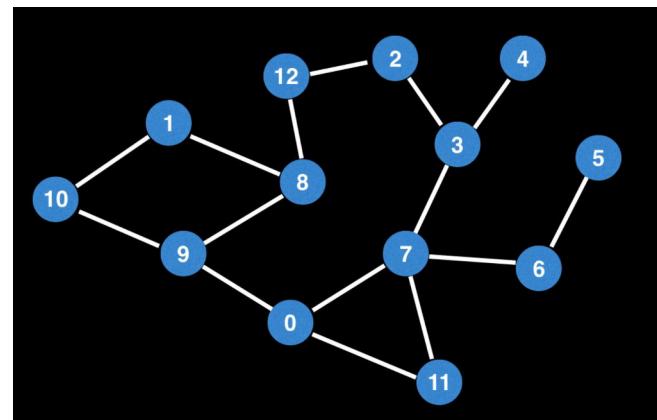
Breadth First Search Algorithm | Shortest Path | Graph Theory

Breadth First Search (BFS) algorithm explanation video with shortest path code

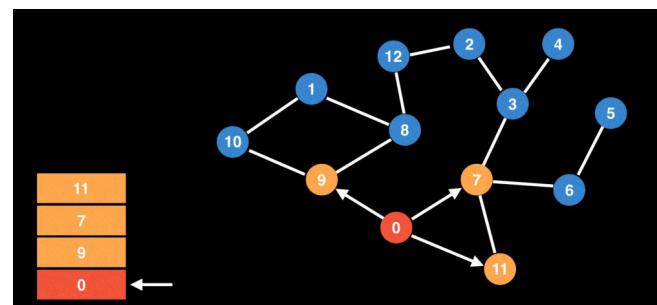
<https://www.youtube.com/watch?v=oDqjPvD54Ss>

Breadth First Search Algorithm

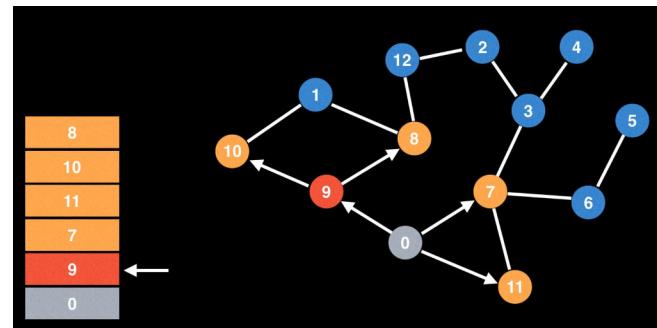
Startzustand



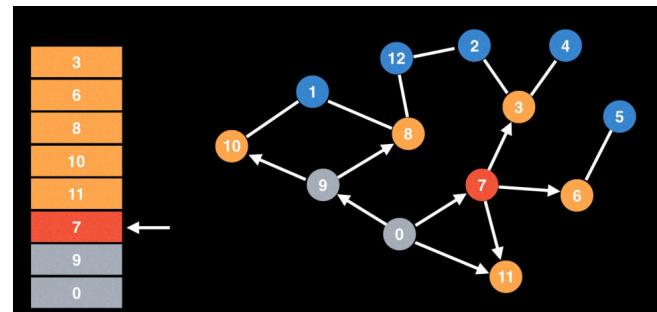
So geht man zuerst zur Start-Node und sucht deren direkt anhängenden Nodes ab. Danach geht man von einer dieser Nodes zu deren direkt anhängenden Nodes und macht dort das selbe.



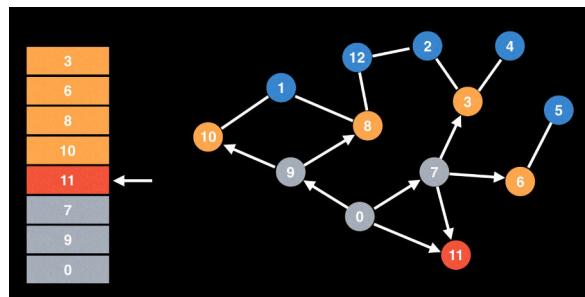
Rechts auf den Bildern sieht man die jeweils gerade anhängenden Nodes. In dieser Reihenfolge werden weitere Nodes abgesucht.



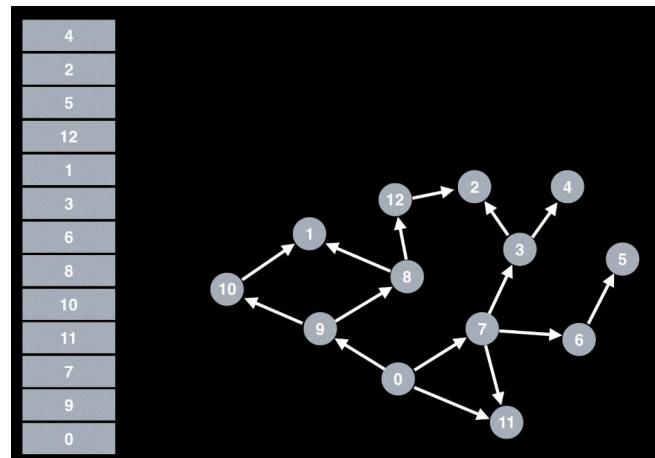
Wenn wir auf eine Zahl treffen, auf der wir keine neuen Nodes entdecken. Gehen wir einfach weiter zur nächsten in der **Queue**.



Dies wird weiter so gemacht bis schlussendlich alle Nodes bearbeitet wurden.



Am Schluss sind alle Nodes durchsucht. Der Algorithmus endet.



Tiefensuche (Depth First Search, DFS)

Die Zeitkomplexität beträgt → $O(|V|+|E|)$.

Funktioniert nach dem **LIFO** Prinzip. Last in - First out. → Wie ein **Stack**.

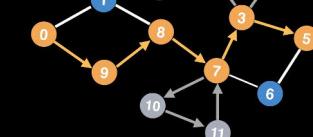
Depth First Search Algorithm | Graph Theory

Depth First Search (DFS) algorithm explanation

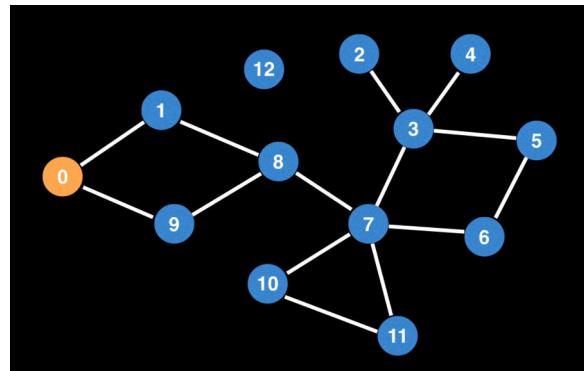
Source code:

▶ <https://www.youtube.com/watch?v=7fujbpJ0LB4>

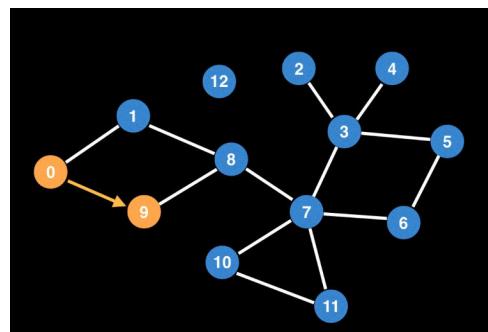
Depth First Search Algorithm



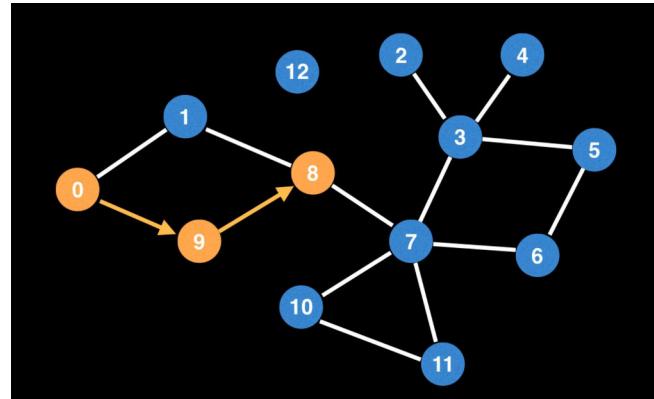
Startzustand



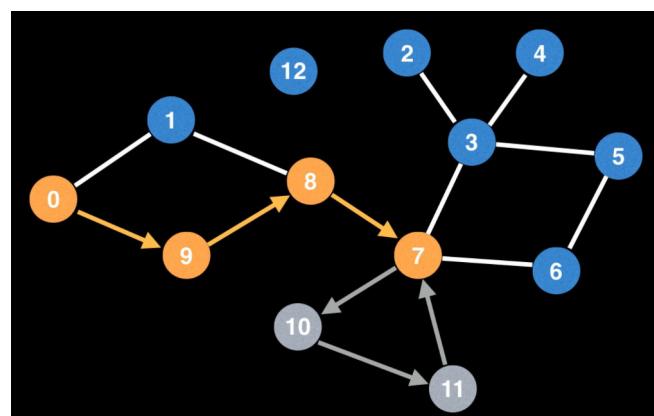
Bei der Tiefensuche geht man von einem Knoten direkt zum nächsten. Dabei ist es egal welchen Knoten man zuerst durchsucht. Dies macht man bis man irgendwann nicht weiterkommt.



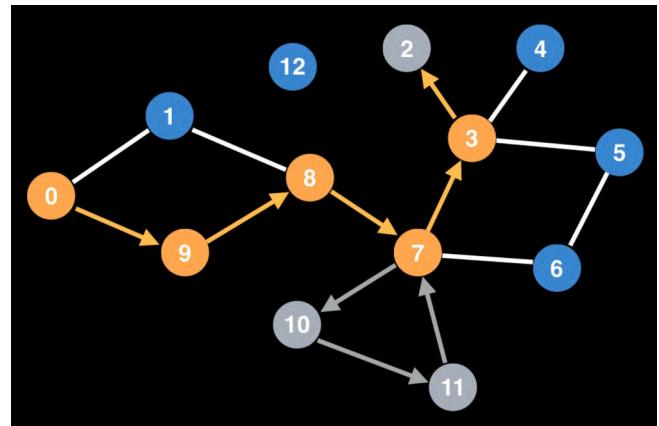
Nehmen wir das Beispiel auf Bild 4 auf dem Knoten mit der Nummer 7. Wir kommen nachdem wir von 7 → 10 → 11 gehen wir zurück auf die 7. Da wir die hier aber nun auch wo anders weiterkommen, können wir direkt von hier fortfahren.



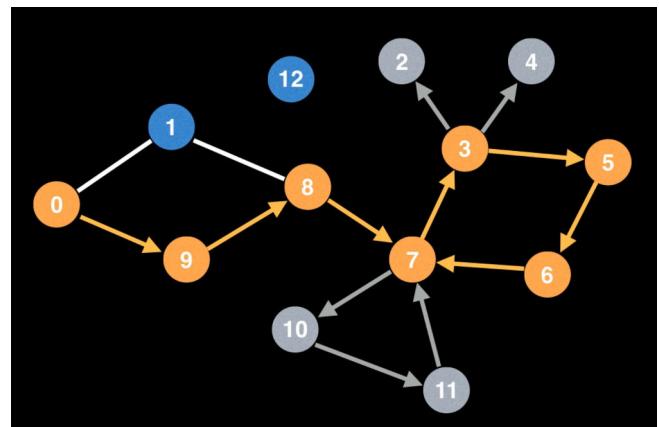
Nachdem wir von 7 → 3 → 2 gekommen sind, stellen wir fest, dass wir hier nicht mehr weiterkommen. Wenn wir also gar nicht mehr weiterkommen, können wir zurück zum letzten Knoten. Von hier aus versuchen wir nun wieder weiterzukommen.



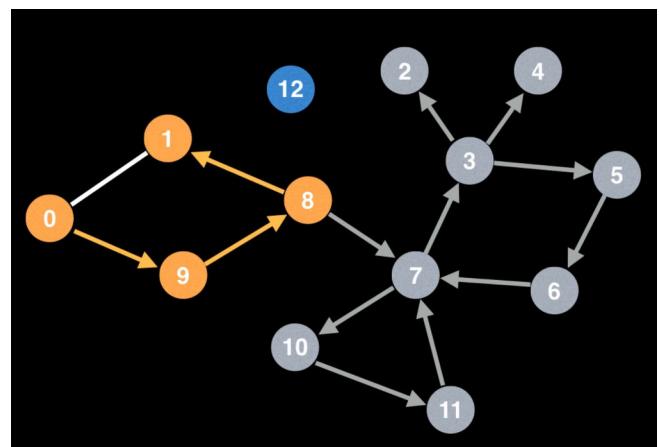
Nehmen wir an, wir haben nun alle Knoten durchsucht und landen wieder bei der 7. Von der 7 an, haben wir jedoch ebenfalls keine neuen Knoten. Was nun passiert ist, dass wir Schritt für Schritt zurück gehen und bei jedem Knoten, ob wir neue Knoten wählen können.



So kommen wir zurück über die 8 zum Knoten mit der 1 und so auch wieder ans Ende. Wenn wir am Ende (Start-Knoten) angekommen sind, sind wir fertig.



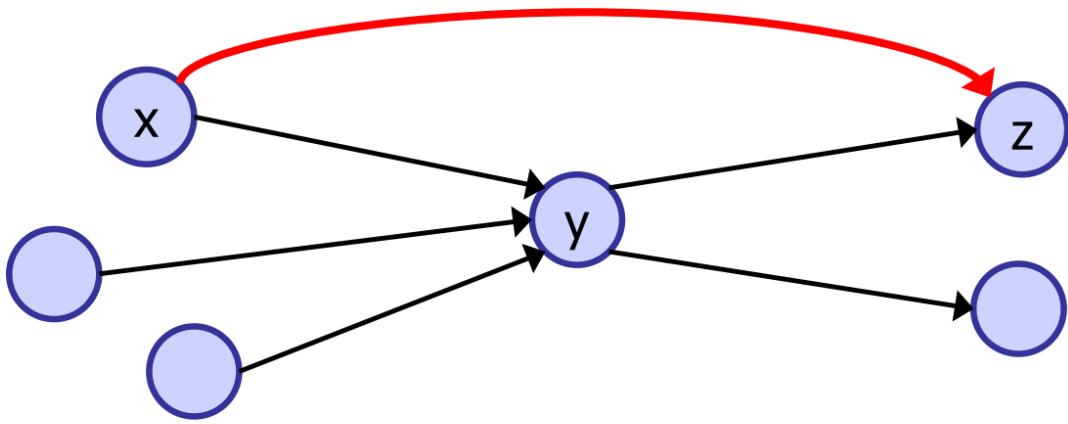
Der Algorithmus ist nach diesem letzten Schritt nun beendet.



Transitive Hülle

Transitive Hüllen erhält man, wenn zwei Kanten (a,b) und (b,c) indirekt also auch $a \rightarrow c$ ergeben.

Mit einer Kante (x, y) und einer Kante (y, z) folgt auch (x, z), d.h. der Knoten z ist vom Knoten x aus erreichbar



Algorithmen

	BFS	Dijkstra's	Bellman Ford	Floyd Warshall
Complexity	$O(V+E)$	$O((V+E)\log V)$	$O(VE)$	$O(V^3)$
Recommended graph size	Large	Large/ Medium	Medium/ Small	Small
Good for APSP?	Only works on unweighted graphs	Ok	Bad	Yes
Can detect negative cycles?	No	No	Yes	Yes
SP on graph with weighted edges	Incorrect SP answer	Best algorithm	Works	Bad in general
SP on graph with unweighted edges	Best algorithm	Ok	Bad	Bad in general

Reference: Competitive Programming 3, P. 161, Steven & Felix Halim

Algorithmus von Floyd

Mit dem Floyd Warshall Algorithm versucht man alle kürzesten Wege zwischen allen Knoten zu finden. → Generierte table am Schluss für mich nicht schlüssig, was die Werte darstellen sollen.



Nicht verstanden und keine Zeit gehabt anzuschauen

Alle kürzesten Pfade finden

Nehmen wir also an, dass wir einen bewerteten Graphen haben und nun von a nach b den kürzesten Weg finden möchten. Dazu können wir gleich die Kante (a,b) mit der Bewertung 11

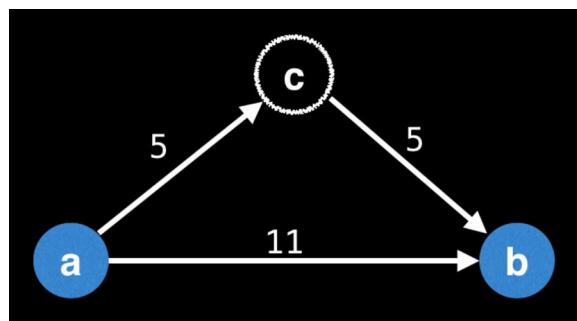


nehmen und uns merken, dass die Entfernung von a nach b = 11 ist.

Wenn also $(a,c) + (c,b) < (a,b)$, ist der kürzeste Pfad (a,c,b) , hier = 10.

Wenn wir die Adjazenzmatrix als Array abbilden können wir uns also merken, dass $m[a][c] + m[c][b] < m[a][b]$.

Wenn wir nun einen dritten Punkt c hinzufügen, sehen wir, dass wir nun noch einen anderen Pfad von a nach b haben.



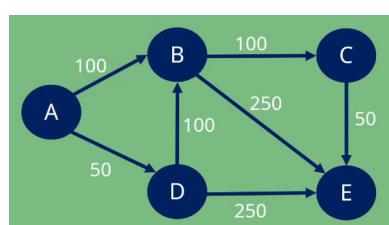
Algorithmus von E.W. Dijkstra

Der Dijkstra Algorithmus ist ein Greedy Algorithmus.

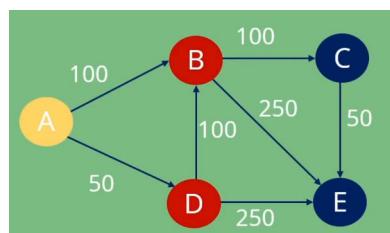
Beim Dijkstra Algorithmus dürfen die Kantenbewertungen nicht negativ sein.

Sollten negative Werte auftreten könnte man den Bellman-Ford-Algorithmus verwenden.

Algorithmus von E.W. Dijkstra - Beispiel



Iteration	A	B	C	D	E	
Kosten	0	∞	∞	∞	∞	Warteschlange: A
Vorgänger	-	-	-	-	-	Erledigt: —



Iteration		A	B	C	D	E	
1	Kosten	0	100	∞	50	∞	Warteschlange: B, D
	Vorgänger	-	A	-	A	-	Erledigt: —

Iteration		A	B	C	D	E	
2	Kosten	0	100	∞	50	300	Warteschlange: B, E
	Vorgänger	-	A	-	A	D	Erledigt: A
							Ausgewählt: D (kürzer)

Iteration		A	B	C	D	E	
3	Kosten	0	100	∞	50	300	Warteschlange: B, E
	Vorgänger	-	A	-	A	D	Erledigt: A, D
							Ausgewählt: C (kürzer)

Informationen zur MEP

Bei uns war es so, dass die Probeprüfung zum Thema Nebenläufigkeit hauptsächlich multiple-choice war, während an der Prüfung viel programmieren von Hand auf Papier gefragt war. Nicht zu vergleichen mit der Probe-MEP!

Die zwei anderen Teile der Prüfung waren vergleichbar wie in der Probeprüfung.