

# Praktische Datenstrukturen in Java

## Ein Überblick

Algorithmen und Datenstrukturen  
VU 186.866, 5.5h, 8 ECTS, 2023S  
Letzte Änderung: 2. Mai 2023

Vorlesungsfolien

**ac**  ALGORITHMS AND  
COMPLEXITY GROUP

 Informatics

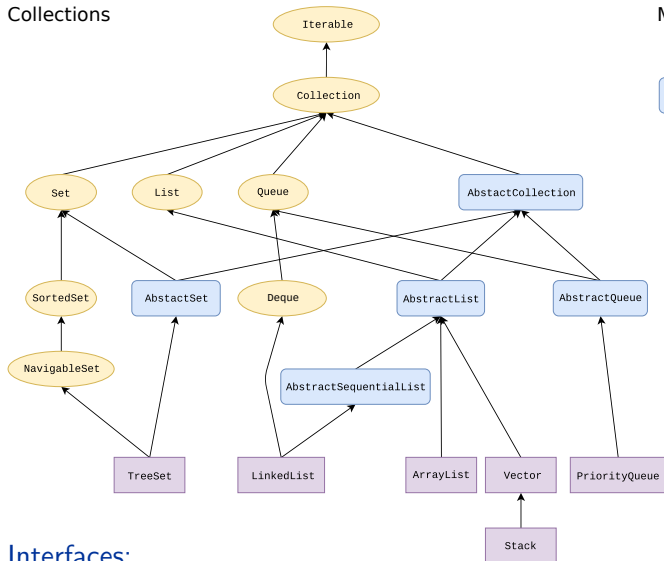
# Themen

- Java Collections-Framework
- Ausgewählte Klassen
- Algorithmen im Collections-Framework
- Weitere Beispiele für Bibliotheken

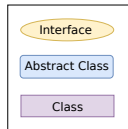
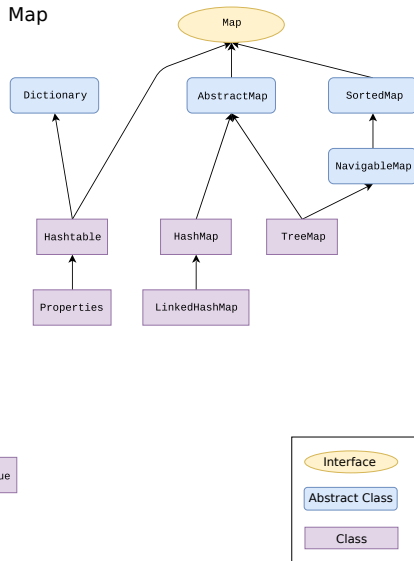
# Java Collections-Framework

# Grundlagen

## Collections



## Map



## Interfaces:

- Für unterschiedliche Datenstrukturen.
- Schnittstellen für den Zugriff auf die Datenstrukturen.
- Konkrete Implementierungen realisieren diese Interfaces.

# Interfaces

## Collection:

- Eine Collection verwaltet Objekte (Elemente).
- Interface enthält generelle Methoden (für alle Collection-Typen).
- Es gibt keine direkte Implementierung dieses Interfaces.

## Set:

- Eine Collection, die keine Duplikate enthält.

## List:

- Eine geordnete Collection (die Duplikate enthalten kann).
- Elemente können über einen Index angesprochen werden (nicht immer effizient).

## Queue/Deque:

- Verwalten von Warteschlangen.
- FIFO, Prioritätswarteschlangen.
- Einfügen und Löschen an beiden Enden bei Deque („double ended queue“).

# Interfaces

## Map:

- Maps verwalten Schlüssel mit dazugehörigen Werten.

## SortedSet und SortedMap:

- Sind spezielle Versionen von Set und Map, bei denen die Elemente (Schlüssel) in aufsteigender Reihenfolge verwaltet werden.

## Generelle Implementierungen (Beispiele)

| Konzept                  | Interface |            |               |            |         |
|--------------------------|-----------|------------|---------------|------------|---------|
|                          | Set       | List       | Queue         | Deque      | Map     |
| <b>Arrays</b>            | -         | ArrayList  | -             | ArrayDeque | -       |
| <b>Bäume</b>             | TreeSet   | -          | -             | -          | TreeMap |
| <b>Hashtabellen</b>      | HashSet   | -          | -             | -          | HashMap |
| <b>Heap</b>              | -         | -          | PriorityQueue | -          | -       |
| <b>Verkettete Listen</b> | -         | LinkedList | LinkedList    | LinkedList | -       |

**Anmerkung:** Interfaces SortedSet und SortedMap werden von TreeSet bzw. TreeMap zusätzlich implementiert.

## Ausgewählte Klassen



# Listenimplementierungen

## ArrayList:

- Indexzugriff auf Elemente ist überall gleich schnell ( $O(1)$ ).
- Einfügen und Löschen ist am Listenende schnell und wird mit wachsender Entfernung vom Listenende langsamer ( $O(n)$ ).

## LinkedList:

- Indexzugriff auf Elemente ist an den Enden schnell und wird mit der Entfernung von den Enden langsamer ( $O(n)$ ).
- Einfügen und Löschen ohne Indexzugriff ist überall gleich schnell ( $O(1)$ ).

# Queue-Implementierungen

`LinkedList` ist auch eine Implementierungen von Queue.

## PriorityQueue

- Ist als Min-Heap implementiert.
- Einfügen eines Elements und Löschen des ersten Elements in einer Queue der Größe  $n$  sind in  $O(\log n)$ .
- Löschen eines beliebigen Elements aus einer Queue der Größe  $n$  ist in  $O(n)$ .
- Lesen des ersten Elements in einer Queue ist in konstanter Zeit möglich ( $O(1)$ ).

# Beispiele zu ArrayList, LinkedList und PriorityQueue

---

```
import java.util.*;

public class Example {
    public static void main(String[] args) {
        List<Integer> myAL = new ArrayList<>();
        myAL.add(42); myAL.add(17);
        System.out.println (myAL); // [42, 17]

        List<Integer> myLL = new LinkedList<>();
        myLL.add(42); myLL.add(17);
        System.out.println (myLL); // [42, 17]

        PriorityQueue<Integer> myPQ = new PriorityQueue<>();
        myPQ.add(750);
        myPQ.add(500);
        myPQ.add(900);
        myPQ.add(100);

        while (!myPQ.isEmpty()) {
            System.out.print (myPQ.remove() + " ");
        } // 100 500 750 900
    }
}
```

# Set-Implementierungen: TreeSet

## TreeSet:

- Implementiert als Rot-Schwarz-Baum.
- null-Elemente sind nicht erlaubt.
- Die Laufzeit von Einfügen, Suchen und Löschen eines Elements liegt bei einem Baum mit  $n$  Elementen in  $O(\log n)$ .
- Auf die Elemente eines TreeSets muss eine Ordnung definiert sein (müssen vergleichbar sein, d.h. das Interface Comparable implementieren).

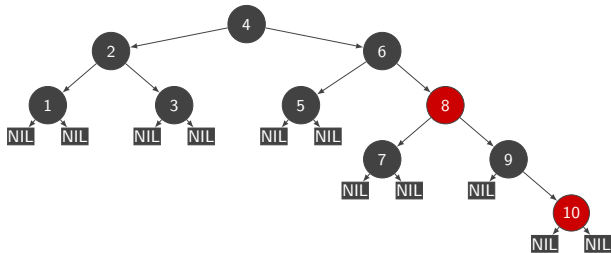
## Rot-Schwarz-Baum:

- Variante eines balancierten Suchbaums.
- Kann als Spezialfall eines B-Baums der Ordnung 4 betrachtet werden.
- Einfügen und Löschen ist effizienter als in B-Bäumen solange die Datenmenge nicht zu groß ist.

# Rot-Schwarz-Baum

- Selbstbalanzierender binärer Suchbaum: jeder Knoten entweder rot oder schwarz.
- Die Wurzel ist schwarz. Jeder Blattknoten (NIL) ist schwarz.
- Kindknoten von einem roten Knoten sind schwarz.
- Für jeden Knoten gilt, dass alle Pfade von diesem Knoten zu nachfolgenden Blattknoten die gleiche Anzahl an schwarzen Knoten beinhalten.
- Einfügen und Löschen können eine Rebalancierung erfordern.

Beispiel:



# Set-Implementierungen: HashSet

## HashSet:

- null-Elemente sind zulässig.
- Einfügen, Suchen und Löschen sind in konstanter Zeit möglich (abhängig von der Verteilung der Einträge in der Hashtabelle).
- Aber: Rehashing kann zu Performance-Problemen führen (siehe auch API-Beschreibung).

## Implementierung:

- Hashing mit Verkettung der Überläufer. Falls die Liste zu lange wird, wird sie in eine TreeMap umgewandelt.
- Maximaler Belegungsfaktor = 0.75.
- Bei Überschreitung des Belegungsfaktors verdoppelt sich die Größe. (Größe der Hashtabelle ist immer eine Zweierpotenz).

# Beispiele zu TreeSet und HashSet

---

```
import java.util.*;
```

```
public class Example {  
    public static void main(String[] args) {  
        Set<String> myTS = new TreeSet<>();  
        myTS.add("apples"); myTS.add("bananas"); myTS.add("strawberries"); myTS.add("grapes");  
        myTS.add("strawberries"); // adding duplicate elements will be ignored  
        System.out.println (myTS); // [apples, bananas, grapes, strawberries ]  
  
        Set<String> myHS = new HashSet<>();  
        myHS.add("apples"); myHS.add("bananas"); myHS.add("strawberries"); myHS.add("grapes");  
        myHS.add("strawberries"); // adding duplicate elements will be ignored  
        System.out.println (myHS); // [strawberries , bananas, apples , grapes ]  
    }  
}
```

---

# Verschiedene Implementierungen von Hashing

Beispiele für Verkettung der Überläufer:

|      | Belegungsfaktor | Wachstumsfaktor | Tabellengröße  |
|------|-----------------|-----------------|----------------|
| Java | 0.75            | 2               | Zweierpotenzen |
| C++  | 1               | >2              | Primzahlen     |
| Go   | 6.5             | 2               | Zweierpotenzen |

Beispiele für offenes Hashing:

|        | Belegungsfaktor | Wachstumsfaktor | Tabellengröße  | Sondierung  |
|--------|-----------------|-----------------|----------------|-------------|
| Python | 0.66            | 2               | Zweierpotenzen | Quadratisch |
| Ruby   | 0.5             | 2               | Zweierpotenzen | Quadratisch |

**Anmerkung:** Die Sondierungsfunktion verwendet in beiden Fällen zusätzlich eine sogenannte Perturbation.



# Maps

- Maps sind eine Verallgemeinerung von Arrays mit einem beliebigen Indextyp (nicht nur int).
- Eine Map ist eine Menge von Schlüssel-Werte Paaren.
  - Schlüssel müssen innerhalb einer Map eindeutig sein.
  - Werte müssen nicht eindeutig sein.

**Arten von Maps:** Wie bei Sets gibt es grundsätzlich zwei Versionen.

- HashMap: Implementierung wie HashSet.
- TreeMap: Implementierung wie TreeSet.

**Anmerkung:** Genau genommen sind Tree- und HashMap die primitiven Implementierungen die von Tree- und HashSet verwendet werden.

**Operationen auf Maps:** Bestimmte Methoden wie z.B. put, get, containsKey, containsValue, remove etc. werden angeboten.

# Beispiele zu HashMap und TreeMap

---

```
import java.util.*;
public class Example { public static void main(String[] args) {
    Map<String, Integer> myHM = new HashMap<>();
    myHM.put("one", 1); myHM.put("two", 2); myHM.put("three", 3); myHM.put("four", 4);
    myHM.putIfAbsent("five", 5); // only add if key does not exist or is mapped to `null`

    System.out.println (myHM); // {one=1, two=2, three=3, four=4, five=5}

    String id = "three";
    if (myHM.containsKey(id)) {
        System.out.println ("Found key " +id+" . Value: " +myHM.get(id)); // Found key three. Value: 3
    }

    Map<String, Integer> myTM = new TreeMap<>();
    myTM.put("one", 1); myTM.put("two", 2); myTM.put("three", 3); myTM.put("four", 4);
    myTM.putIfAbsent("four", 444); // only add if key does not exist or is mapped to `null`

    System.out.println (myTM); // {four=4, one=1, three=3, two=2}

    id = "four";
    if (myTM.containsKey(id)) {
        System.out.println ("Found key " + id + ". Value: " + myTM.get(id)); // Found key four. Value: 4
    }
}}
```

# Abstrakte Klassen

- Unterstützen die Entwicklung neuer Klassen im Framework.
- Implementieren einen Teil eines Interfaces und lassen bestimmte Teile noch offen.

## Neue Klasse implementieren:

- Auswählen einer geeigneten abstrakten Klasse von der geerbt wird.
- Implementierung aller abstrakten Methoden.
- Sollte die Collection modifizierbar sein, dann müssen auch einige konkrete Methoden überschrieben werden (siehe API).
- Testen der neuen Klasse (inklusive Performance).

# Abstrakte Klassen

## Beispiele:

- `AbstractCollection`
- `AbstractSet`
- `AbstractList` (basierend auf `Array`)
- `AbstractSequentialList` (basierend auf verketteter Liste)
- `AbstractQueue`
- `AbstractMap`

**API:** API-Dokumentation jeder abstrakten Klasse beschreibt genau, wie man eine Klasse ableiten muss:

- Grundlegende Implementierung.
- Welche Methoden müssen implementiert werden.
- Welche Methoden müssen überschrieben werden, wenn man Modifikationen zulassen möchte.

## Algorithmen im Collections-Framework

# Algorithmen im Collections-Framework

## Algorithmen:

- Das Collections-Framework bietet auch Algorithmen für die Verarbeitung von Container-Klassen an.
- Diese Algorithmen werden als statische Methoden (polymorphe Methoden) in der Hilfsklasse Collections gesammelt.

## Beispiele:

- **sort** sortiert die Elemente einer generischen Liste nach aufsteigender Größe.
- **binarySearch** sucht ein Element in der sortierten Liste (Voraussetzung) und liefert einen Index zurück, wenn das Element gefunden wurde (ansonsten eine negative Zahl).
- **max** liefert das größte Element einer Collection.
- **shuffle** mischt die Elemente einer generischen Liste zufällig.

# Beispiele zu Algorithmen im Collections-Framework

---

```
import java.util.*;
public class Example {
    public static void main(String[] args) {
        List<Integer> myAL = new ArrayList<>(Arrays.asList(5, 3, 1, 4));
        System.out.println (myAL); // [5, 3, 1, 4]
        Collections . sort (myAL);
        System.out.println (myAL); // [1, 3, 4, 5]

        System.out.println ( Collections .max(myAL)); // 5

        System.out.println ( Collections .binarySearch(myAL, 5)); // 3
        System.out.println ( Collections .binarySearch(myAL, 42)); // -5

        Collections . shuffle (myAL);
        System.out.println (myAL); // z.B.: [1, 4, 3, 5]
    }
}
```

---

# Algorithmen im Collections-Framework: Beispiel sort

## Sortieren von Collections:

- Zum Sortieren von Collections wird TimSort verwendet.
- TimSort ist eine Kombination von Mergesort und Insertionsort.
- TimSort wurde 2002 von Tim Peters für die Verwendung in Python entwickelt.
- Heute wird TimSort unter anderem in Python, Java, Android und Google Chrome verwendet.



# TimSort

**Grundlegende Idee:** Finde schon sortierte Stücke (Runs) des Inputs  $S$ , die dann paarweise verschmelzt werden (Merging).

```
TimSort( $S$ )  
runs  $\leftarrow$  partitioniere  $S$  in Runs  
 $\mathcal{R} \leftarrow$  leerer Stack  
while runs  $\neq \emptyset$   
    entferne Run  $r$  von runs and pushe  $r$  auf den Stack  $\mathcal{R}$   
    merge_collapse( $\mathcal{R}$ ) // ausbalanciertes Verschmelzen  
while height( $\mathcal{R}$ )  $\neq 1$   
    verschmelze die beiden oberen Runs am Stack
```

# TimSort: Berechnen von Runs

## Min\_Run:

- Runs sind mindestens Min\_Run lang.
- Min\_Run wird so gewählt, dass  $\frac{\text{Größe des Arrays}}{\text{Min_Run}}$  eine Zweierpotenz oder etwas kleiner als eine Zweierpotenz ist.
- Experimente zeigen, dass ein Wert zwischen 32 und 64 für Min\_Run optimal ist.

## Finden von Runs:

- Startend bei dem ersten Element, das noch nicht in einem Run ist, füge so lange Elemente zum Run hinzu, wie die Sequenz entweder aufsteigend oder *strikt* absteigend ist.
- Falls die Sequenz strikt absteigend war, invertiere die Reihenfolge der Elemente.
- Falls die Länge der Sequenz größer Min\_Run ist, ist damit ein Run gefunden.
- Andernfalls füge solange nachfolgende Elemente hinzu bis Min\_Run erreicht ist. Sortiere den entstandenen Run mittels (binärem) Insertionsort.

## Runs mit Min\_Run = 4:

1 3 4 5 8 7

1 5 4 6 8 9

8 5 4 3 2 1

⇒ Insertionsort

⇒ Invertieren

## TimSort: Verschmelzen der Runs (Merging)

- Nachdem das Array in Runs aufgeteilt ist und diese sortiert sind, werden diese wie bei Mergesort verschmolzen.
- Merging ist am effizientesten, wenn die Anzahl der Runs eine Zweierpotenz ist und die Runs möglichst gleich groß sind.
- Da Runs im Allgemeinen sehr unterschiedliche Größen haben können, wird versucht mithilfe von Invarianten eine sinnvolle Merge-Reihenfolge zu finden (Beachte: Nur „benachbarte“ Runs werden verschmolzen).
- Zusätzlich wird beim Merging eine Strategie namens Galloping verwendet, um möglicherweise vorhandene Strukturen in den Runs zu nutzen.

## TimSort: Invarianten und merge\_collapse

Stack  $\mathcal{R}$  von Runs mit Aufbau  $\mathcal{R}[1], \dots, \mathcal{R}[\text{height}(\mathcal{R})]$ , wobei  $\mathcal{R}[1]$  top-of-stack ist. Die Länge von Run  $\mathcal{R}[i]$  wird mit  $r_i$  bezeichnet.

```
merge_collapse( $\mathcal{R}$ )
while height( $\mathcal{R}$ ) > 1
    if height( $\mathcal{R}$ ) > 2 und  $r_3 \leq r_2 + r_1$ 
        if  $r_3 < r_1$ 
            verschmelze  $\mathcal{R}[2]$  und  $\mathcal{R}[3]$  am Stack
        else
            verschmelze  $\mathcal{R}[1]$  und  $\mathcal{R}[2]$  am Stack
    elseif height( $\mathcal{R}$ ) > 1 und  $r_2 \leq r_1$ 
        verschmelze  $\mathcal{R}[1]$  und  $\mathcal{R}[2]$  am Stack
    else break
```

Nach Ausführung von  $\text{merge\_collapse}(\mathcal{R})$  gelten folgende Invarianten:

- (1)  $r_3 > r_2 + r_1$
- (2)  $r_2 > r_1$

## TimSort: Beispiel zu Invarianten

Wiederholung Invarianten: (1)  $r_3 > r_2 + r_1$ , (2)  $r_2 > r_1$

Länge der Runs am Stack werden in der Reihenfolge  $r_1, r_2, r_3, \dots$  angegeben.

- 15, 67, 138. Invarianten erfüllt.
- Nächster Run hat Größe 16. Invariante (2) nicht erfüllt! (16, 15, 67, 138)
- Mergen ergibt: 31, 67, 138. Invarianten erfüllt.
- Nächster Run hat Größe 17. Invarianten erfüllt! (17, 31, 67, 138)
- Nächster Run hat Größe 15. Invariante (1) nicht erfüllt! (15, 17, 31, 67, 138)
- Mergen ergibt: 32, 31, 67, 138. Invariante (2) nicht erfüllt.
- Mergen ergibt: 63, 67, 138. Invarianten erfüllt.

# TimSort: Galloping

## Lineares Merging:

- Arrays  $A$  und  $B$  sollen verschmelzt (gemerget) werden.
- Vergleiche  $A[0]$  und  $B[0]$  und verschiebe den kleineren Wert in das Merge-Array.
- Ineffizient falls immer das selbe Array gewinnt.
- TimSort speichert, wie oft dasselbe Array gewinnt und wechselt in den Galloping-Mode falls dasselbe Array mehr als `Min_Galloping` mal gewinnt.

$A =$  1 3 4 5 7 8

$B =$  11 15 16 18 21 23

## Galloping:

- Falls  $A$  `Min_Galloping` mal gewonnen hat, suche das erste Element  $A[i]$  in  $A$  das größer als  $B[0]$  ist.
- Kopiere  $A[0]$  bis  $A[i - 1]$  in das Merge-Array.
- Suche nach  $A[i]$  mittels galoppierender binärer Suche, d.h. vergleiche  $B[0]$  mit  $A[1], A[3], A[7], \dots, A[2^k - 1]$  bis  $B[0]$  kleiner ist, dann binäre Suche.

# Vorteile und Nachteile von TimSort

## Vorteile:

- Extrem schnell auf Arrays, die bereits viel Struktur haben (Best-Case:  $O(n)$ ).
- Average- und Worst-Case  $O(n \log n)$  wie Mergesort.
- Im Durchschnitt weniger Objekt-Vergleiche als Quicksort.
- Stabil.

## Nachteile:

- Zusätzlicher Speicher: Average- und Worst-Case:  $O(n)$ . (Optimierter Quicksort:  $O(\log n)$ )
- Im Durchschnitt langsamer als ein optimierter Quicksort, wenn das Vergleichen von Elementen sehr effizient möglich ist

**Kompromisslösung:** TimSort wird nur auf Objekt-Arrays verwendet. Für das Sortieren von Arrays mit primitiven Datentypen (z.B. int) ist eine spezielle Quicksort Variante als `sort` in der Hilfsklasse `Arrays` vorgesehen.

# Sortieren: Stabilität (Wiederholung)

**Stabiles Sortierverfahren:** Ein stabiles Sortierverfahren ist ein Sortieralgorithmus, der die Reihenfolge der Datensätze, deren Sortierschlüssel gleich sind, bewahrt.

**Beispiel:**

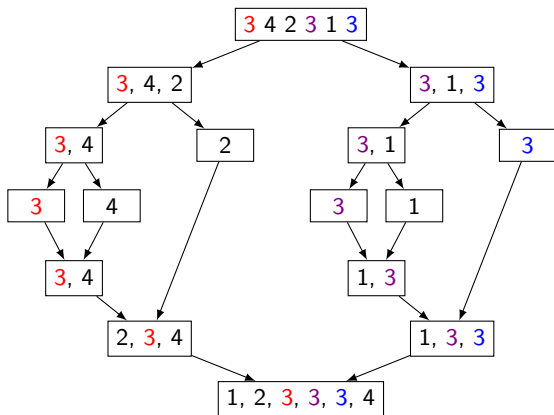
- Liste von Personendaten mit Abteilungsnummer (innerhalb der Abteilung sortiert):  
3 - Daniel, 4 - Maria, 2 - Paul, 3 - Erika, 1 - Anton, 3 - Sarah
- Sortierung nach Abteilungsnummer mit stabilem Sortierverfahren:  
1 - Anton, 2 - Paul, 3 - Daniel, 3 - Erika, 3 - Sarah, 4 - Maria



# Stabilität: Mergesort

Mergesort ist stabil.

Beispiel zur Illustration der Idee mit vereinfachter Ausgangssequenz 3 4 2 3 1 3



## Stabilität: Selectionsort

Selectionsort ist nicht stabil.

Beispiel mit vereinfachter Ausgangssequenz 3 4 2 3 1 3

3 4 2 3 1 3

1 4 2 3 3 3

1 2 4 3 3 3

1 2 3 4 3 3

1 2 3 3 4 3

1 2 3 3 3 4

# Stabilität: Algorithmen in dieser Vorlesung

Beispiele für stabile Sortierverfahren:

- Insertionsort
- Mergesort
- TimSort

Hinweis: Korrekte Implementierung erforderlich!

Beispiele für instabile Sortierverfahren:

- Selectionsort
- Quicksort

## Weitere Bibliotheken

# Weitere Beispiele für Bibliotheken

## Eclipse Collections

- <https://www.eclipse.org/collections/>
- Optimierte Sets und Maps, Immutable Collections, Collections für primitive Datentypen, Multimaps, Bimaps, Verschiedene Iterationsstile

## Google Guava

- <https://github.com/google/guava>
- Unterstützt z.B. Multisets, Multimaps, Bimaps

## Apache Commons Collections

- <https://commons.apache.org/proper/commons-collections/>

## Brownies Collections

- <http://www.magicwerk.org/page-collections-overview.html>
- z.B. High Performance Listen (GapList, BigList)

## JGraphT

- <https://github.com/jgrapht/jgrapht>
- Algorithmen und Datenstrukturen für Graphen