

## Zusammenfassung

# Algorithmen und Datenstrukturen 2

Julian Klaiber und Severin Dellsperger

Hochschule für Technik Rapperswil

17. Januar 2019

### Lizenz

"THE BEER-WARE LICENSE" (Revision 42): Julian Klaiber and Severin Dellsperger wrote this file. As long as you retain this notice you can do whatever you want with this stuff. If we meet some day, and you think this stuff is worth it, you can buy us a beer in return.

## Inhaltsverzeichnis

<b>1. Big Oh Laufzeitverhalten</b>	<b>6</b>
1.1. Laufzeiten . . . . .	6
1.2. Summenformel . . . . .	6
1.3. $n - 1$ Iterationen . . . . .	6
1.4. Stirling Formel . . . . .	6
<b>2. Übersicht Datenstrukturen</b>	<b>7</b>
2.1. Sortier- und Suchalgorithmen . . . . .	7
<b>3. Multimaps</b>	<b>8</b>
3.1. Operationen . . . . .	8
3.2. Geordnete Multimap . . . . .	8
<b>4. Bäume</b>	<b>9</b>
4.1. Terminologie . . . . .	9
4.2. Traversierung . . . . .	9
4.2.1. Laufzeit . . . . .	10
4.2.2. Implementierungen . . . . .	10
<b>5. BST: Binäre Such Bäume</b>	<b>11</b>
5.1. Speicherplatz, Laufzeiten . . . . .	11
5.2. Binäre Suche . . . . .	12
5.3. Operationen . . . . .	13
5.4. Binäre Sortierung . . . . .	16
5.5. Speicherverbrauch . . . . .	16
5.6. Implementierungen . . . . .	17
<b>6. AVL Tree</b>	<b>21</b>
6.1. Laufzeiten . . . . .	21
6.2. Operationen . . . . .	22
6.3. Rotationen / Trinode Umstrukturierung . . . . .	25
6.3.1. Rechts rotieren (einfach) . . . . .	25
6.3.2. Links rotieren (einfach) . . . . .	26
6.3.3. Rechts/Links Doppelrotation . . . . .	27
6.3.4. Links/Rechts Doppelrotation . . . . .	28
6.4. Cut/Link Restrukturierung . . . . .	29
6.5. Implementierung . . . . .	30
<b>7. Splay Tree</b>	<b>34</b>
7.1. Varianten . . . . .	34
7.2. Vorgehen . . . . .	35
7.3. Remove . . . . .	35
7.4. Splaying . . . . .	36
7.5. Laufzeiten . . . . .	36
<b>8. Sortieralgorithmen</b>	<b>37</b>
8.1. Eigenschaften . . . . .	37
8.2. Varianten . . . . .	37

8.3. Laufzeiten . . . . .	37
8.4. Lexikographische Sortierung . . . . .	38
<b>9. Bubble Sort</b>	<b>39</b>
9.1. Laufzeiten . . . . .	39
<b>10. Merge Sort</b>	<b>40</b>
10.1. Laufzeiten . . . . .	40
<b>11. Quick Sort</b>	<b>43</b>
11.1. Laufzeiten . . . . .	44
11.2. In Place Implementierung . . . . .	44
<b>12. Bucket Sort</b>	<b>47</b>
12.1. Laufzeiten . . . . .	48
12.2. Implementierung . . . . .	48
<b>13. Radix Sort</b>	<b>49</b>
13.1. Laufzeiten . . . . .	49
13.2. Radix Sort Algorithmen . . . . .	49
13.3. Beispiel . . . . .	50
13.4. Implementierung . . . . .	51
<b>14. Pattern Matching</b>	<b>52</b>
14.1. Laufzeiten . . . . .	52
14.2. Brute Force Algorithmus . . . . .	52
14.3. Boyer-Moore Algorithmus . . . . .	53
14.3.1. Last Occurrence Funktion . . . . .	53
14.3.2. Vorgehen . . . . .	54
14.3.3. Algorithmus . . . . .	55
14.3.4. Implementierung . . . . .	56
14.4. KMP: Knuth-Morris-Pratt Algorithmus . . . . .	57
14.4.1. Fehl-Funktion . . . . .	57
14.4.2. Vorgehen . . . . .	58
14.4.3. Algorithmus . . . . .	59
14.4.4. Implementierung . . . . .	60
<b>15. Tries</b>	<b>61</b>
15.1. Standard Trie . . . . .	61
15.1.1. Vorgehen . . . . .	61
15.2. Komprimierter Trie . . . . .	62
15.3. Suffix Trie . . . . .	62
15.4. Laufzeitverhalten / Speicherplatz . . . . .	63
15.5. Implementierung . . . . .	63
<b>16. Dynamische Programmierung</b>	<b>66</b>
16.1. Rucksack Problem . . . . .	66
16.2. Voraussetzung Subprobleme . . . . .	66
16.3. Subsequenzen . . . . .	67
16.4. LCS: Longest Common Subsequence . . . . .	67

16.5. Vorgehen . . . . .	68
16.5.1. Implementierung . . . . .	69
<b>17. Graphen</b>	<b>70</b>
17.1. Terminologie . . . . .	70
17.1.1. Subgraphen . . . . .	71
17.1.2. Tree und Forest . . . . .	71
17.1.3. Pfad und Zyklen . . . . .	71
17.2. Kanten-Listen Struktur . . . . .	72
17.2.1. Kanten-Listen Struktur Implementierung . . . . .	73
17.3. Adjazenz-Listen Struktur . . . . .	74
17.4. Adjazenz-Matrix Struktur . . . . .	74
17.5. Laufzeiten . . . . .	74
17.6. Implementierung . . . . .	75
<b>18. DFS und BFS</b>	<b>79</b>
18.1. DFS: Depth First Search . . . . .	79
18.2. Implementierung DFS . . . . .	81
18.3. BFS: Breadth First Search . . . . .	82
18.4. DFS vs. BFS . . . . .	84
<b>19. Gerichtete Graphen (Directed Graphs, Digraph)</b>	<b>85</b>
19.1. Scheduling . . . . .	85
19.2. Laufzeiten . . . . .	85
19.3. Strong Connectivity . . . . .	85
19.4. DFS und BFS . . . . .	86
19.5. Transitiver Abschluss . . . . .	87
19.6. Floyd-Warshalls Algorithmus . . . . .	88
19.6.1. Vorgehen . . . . .	88
19.6.2. Beispielaufgabe Floyd-Warshall . . . . .	90
19.7. DAG: Directed Acyclic Graph . . . . .	91
19.8. Topologische Sortierung . . . . .	91
19.8.1. Vorgehen . . . . .	92
19.8.2. DAG Implementierung . . . . .	93
<b>20. Shortest Path Trees</b>	<b>95</b>
20.1. Laufzeiten . . . . .	95
20.2. Dijkstra Algorithmus . . . . .	96
20.2.1. Vorgehen . . . . .	96
20.3. Dijkstra Algorithmus . . . . .	97
20.3.1. Dijkstra Distance Beispiel Übung . . . . .	98
20.3.2. Implementierung . . . . .	99
20.3.3. Implementierung Path Finder . . . . .	100
20.4. Bellman-Ford . . . . .	101
20.4.1. Dijkstra vs. Bellman-Ford . . . . .	101
20.4.2. Implementierung . . . . .	102
20.5. DAG basierter Algorithmus . . . . .	102

<b>21. Minimum Spanning Tree</b>	<b>103</b>
21.1. Kruskal Algorithmus . . . . .	104
21.1.1. Kruskal Implementierung . . . . .	105
21.1.2. Repräsentation einer Partition . . . . .	106
21.2. SPT und MST . . . . .	106
21.3. Prim-Jarnik's Algorithmus . . . . .	107
21.3.1. Vorgehen . . . . .	107
21.4. Borůvka's Algorithmus . . . . .	108
21.5. Laufzeit . . . . .	108
<b>A. Listings</b>	<b>109</b>
<b>B. Abbildungsverzeichnis</b>	<b>110</b>
<b>C. Tabellenverzeichnis</b>	<b>112</b>

## 1. Big Oh Laufzeitverhalten

### 1.1. Laufzeiten

1.  $\mathcal{O}(1)$  = konstant
2.  $\mathcal{O}(\log(n))$  = logarithmisch
3.  $\mathcal{O}(n)$  = linear
4.  $\mathcal{O}(n \cdot \log(n))$  = n-Log-n
5.  $\mathcal{O}(n^2)$  = Quadratisch
6.  $\mathcal{O}(n^3)$  = Qubisch
7.  $\mathcal{O}(2^n)$  = Exponentiell
8.  $\mathcal{O}(n!)$  = Fakultät

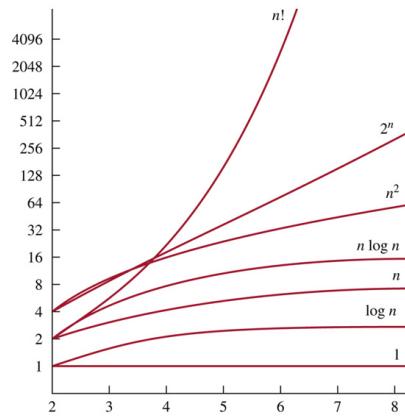


Abbildung 1: Laufzeiten

### 1.2. Summenformel

#### n Iterationen

$$\frac{n \cdot (n + 1)}{2} \quad (1)$$

#### 1.3. n - 1 Iterationen

$$\frac{n \cdot (n - 1)}{2} \quad (2)$$

#### 1.4. Stirling Formel

Die Stirling Formel kann für das Laufzeitverhalten von binären Suchbäumen verwendet werden.

$$\lim_{n \rightarrow \infty} \log_2(n!) = n \cdot \log_2(n) \quad (3)$$

## 2. Übersicht Datenstrukturen

Datenstruktur	Operationen	Laufzeitverhalten
Array	get, set,	$\mathcal{O}(1)$
Array	add, remove	$\mathcal{O}(n)$
List	addFirst, addLast, remove	$\mathcal{O}(1)$
List	get, set, add	$\mathcal{O}(n)$
Heap	Insert, remove, Up/Down-heap	$\mathcal{O}(\log(n))$
Heap	size, isEmpty, min, removeMin	$\mathcal{O}(1)$
Map	put, get, remove	$\mathcal{O}(n)$
Map	Sentinel-Trick	halb so viele Abfragen
Stack (Array-basiert)	alle Operationen	$\mathcal{O}(1)$
Stack (Array-basiert)	Speicherplatz	$\mathcal{O}(n)$
Deque	alle	$\mathcal{O}(1)$
Priority Queue	mit sortierter Liste	insert $\mathcal{O}(1)$ , removeMin/min $\mathcal{O}(n)$
Priority Queue	mit sortierter Liste	insert $\mathcal{O}(n)$ , removeMin/min $\mathcal{O}(1)$
Positional List mit doubly Linked List	suche nach pos	$\mathcal{O}(n)$
Positional List mit doubly Linked List	alle Operationen mit pos	$\mathcal{O}(1)$
Hash Table	Worst case (Kollisionen)	search, insert, remove $\mathcal{O}(n)$
Hash Table	Gute Streuung	alle $\mathcal{O}(1)$
Skip List	search, remove, insert, height	$\mathcal{O}(\log(n))$
Skip List	Speicherplatz	$\mathcal{O}(n)$

Tabelle 1: Laufzeitverhalten von Datenstrukturen

### 2.1. Sortier- und Suchalgorithmen

Weitere Sortieralgorithmen sind in Abschnitt 8 zu finden:

Algorithmus	Datenstruktur	Best Case	Worst Case
Insertion Sort	Array	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Insertion Sort	Doubly-Linked List	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Insertion Sort	Priority Queue	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Selection Sort	Array	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Selection Sort	Doubly-Linked List	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Selection Sort	Priority Queue	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Heap Sort	Heap	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$
Linear Search		$\mathcal{O}(n)$	$\mathcal{O}(2n)$
Binary Search		$\mathcal{O}(n)$	$\mathcal{O}(n + \log(n))$

Tabelle 2: Laufzeitverhalten von Sortier- und Suchalgorithmen

### 3. Multimaps

- Multimaps sind immer ungeordnet
- Multimaps können mehrere der gleichen Elemente erhalten.

#### 3.1. Operationen

**find(key)** Liefert den Wert für den Key oder null

**findAll(key)** Liefert eine iterierbare Collection mit allen Werten zum Key

**insert(key, value)** Fügt einen neuen Wert zum Schlüssel k ein

**remove(node)** Entfernt den kompletten Knoten

#### 3.2. Geordnete Multimap

Bei der geordneten Multimap sind die Keys der grössze nach geordnet. Sie besitzt folgende zusätzlichen Operationen.

**first()** Liefert den ersten Eintrag

**last()** Liefert den letzten Eintrag

**successor(key)** Liefert einen Iterator mit allen Knoten deren Key grösser oder gleich dem gegebenen Key ist. Aufsteigende Ordnung.

**predecessor(key)** Liefert einen Iterator mit allen Knoten deren Key kleiner oder gleich dem gegebenen Key ist. Absteigende Ordnung.

## 4. Bäume

### 4.1. Terminologie

**k-Baum** Ein Baum mit k Kindknoten pro Node

**Wurzel / Root** Elternknoten

**Interner Knoten** Knoten mit min. einem Child

**Externer Knoten / Blattknoten** Knoten ohne Childs:

**Vorgängerknoten** Parent

**Tiefe** Anzahl Vorgänger (nach oben) (Der Wurzel Knoten hat die Tiefe = 0)

**Höhe** Anzahl Ebenen der Nachfolger (nach unten). Die Höhe gibt die Anzahl Ebenen des Baumes an. (Externe Knoten haben die Höhe 0)

**Subtree** Baum aus einem Knoten und seinen Nachfolger

**Siblings** Zwillingsknoten

### 4.2. Traversierung

Gestartet wird immer beim Root, aufgeschrieben wird aber nur gemäss der Euler Tour Traversierung. Dabei zeichnet man startend links vom Parent Node eine Umrandung um den ganzen Tree und zieht bei jedem Knoten einen Strich in eine bestimmte Richtung:

#### Preorder / Strich nach Links

Ein Node wird vor seinen Nachfolgern besucht, wobei zuerst der linke Node und danach der rechte Node abgearbeitet wird. ( $ParentNode \Rightarrow LeftNode \Rightarrow RightNode$ )

#### Postorder / Strich nach Rechts

Ein Node wird nach seinen Nachfolgern besucht ( $LeftNode \Rightarrow RightNode \Rightarrow ParentNode$ ).

#### Inorder / Strich nach Unten

Ein Knoten wird **nach** seinem linken Subtree und **vor** seinem rechten Subtree besucht. ( $LeftNode \Rightarrow ParentNode \Rightarrow RightNode$ )

#### Breath First / Level Traversierung

Es werden zuerst alle Nodes einer Ebenen besucht bevor man zu einer tieferen Ebenen voranschreitet. Die Nodes werden dabei von links nach rechts abgearbeitet.

### 4.2.1. Laufzeit

Alle Traversierungen laufen mit  $\Theta(n)$  (Theta)

### 4.2.2. Implementierungen

Listing 1: Inorder Traversal

---

```
1 public Collection<Entry<K, V>> inorder() {
2     Collection<Entry<K, V>> inorderCollection = new LinkedList<>();
3     inorder(root, inorderCollection);
4     return inorderCollection;
5 }
6
7 protected void inorder(Node node, Collection<Entry<K, V>> inorderCollection) {
8     if (node != null) {
9         inorder(node.getLeftChild(), inorderCollection);
10        inorderCollection.add(node.getEntry());
11        inorder(node.getRightChild(), inorderCollection);
12    }
13 }
```

---

## 5. BST: Binäre Such Bäume

- Im Gegensatz zu herkömmlichen Datenstrukturen (Array, Linked-Lists) mit linearer Laufzeit, erlauben Binäre Suchbäume **logarithmische Laufzeiten**
- Die **Inorder Traversal** retourniert die Keys in **geordneter Folge**
- Die **Inorder Traversal besucht** die Keys in **nicht absteigender Folge**
- Ein BST kann mit einer Map oder Multimap implementiert sein. Im Falle einer Map wird die Value einfach überschrieben.
- Gegeben sind immer drei Knoten **u, v, w** wobei:
  - v die Root für den Teilbaum ist
  - u im linken Teilbaum von v ist (kleinerer oder gleicher Wert)
  - w im rechten Teilbaum von v ist (grösserer Wert)
  - und  $key(u) \leq key(v) \leq key(w)$
- Der binäre Such Baum ist ein binärer Baum, der die Keys in seinen internen Knoten speichert
- Die externen Knoten speichern keine Daten, nur die internen
- Man erkennt am einfachsten, ob ein BST nullterminiert ist oder über externe abschliessende Nodes verfügt, wenn man im Konstruktor die Initialisierung der beiden Variablen `leftson` und `rightson` überprüft. (=**null**?)

### 5.1. Speicherplatz, Laufzeiten

Find, Insert und Remove benötigen  $\mathcal{O}(h)$  (Höhe), wobei die Höhe h im Worst Case  $\mathcal{O}(n)$  und im Best Case  $\mathcal{O}(\log(n))$  ist. Der Worst Case ist wenn die Elemente vorsortiert eingefügt werden

Methode	Best Case	Worst Case
Speicherplatz	$\mathcal{O}(n)$	$\mathcal{O}(n)$
find()	$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$
insert()	$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$
remove()	$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$
sort()	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n^2)$

Tabelle 3: Laufzeitverhalten von Suchtabellen

## 5.2. Binäre Suche

Binäre Suche benötigt immer random access, damit in die Mitte des Payloads springen kann und von dort aus die Suche starten kann. (Divide and Conquer) Des Weiteren müssen die Daten sortiert sein. Die binäre Suche ist eine Suche nach "Einschachtelung", da die linke und rechte Grenze immer enger zusammengezogen wird, bis man auf das Resultat stößt. Ein Suchpfad ist dann invalid, wenn eine der Grenzen überschritten wird. **Terminiert nach  $O(\log n)$**

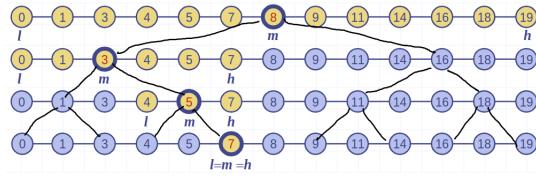


Abbildung 2:  $\text{find}(7)$

### 5.3. Operationen

**find(key)**

Liefert den Eintrag zum Schlüssel k oder null

---

**Algorithm 1:** TreeSearch(k,v)

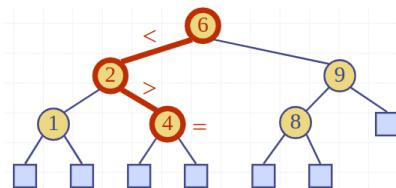
---

```

1: if  $T.isExternal(v)$  then
2:   return  $v$ 
3: end if
4: if  $k < key(v)$  then
5:   return  $TreeSearch(k, T.left(v))$ 
6: else if  $k = key(v)$  then
7:   return  $v$ 
8: else
9:   return  $TreeSearch(k, T.right(v))$ 
10: end if

```

---



**insert(key, object)**

- Wenn der Key noch nicht vorhanden ist, wird gemäss Binary Search Algorithm nach einem passenden Blatt Knoten gesucht. Wurde der Blatt Knoten gefunden, wird der neue Key eingefügt und in einen internen Knoten expandiert.
- Wenn der Key bereits vorhanden ist, wird ausgehend von dem ersten Treffer des existierenden Key im linken Teilbaum weitergesucht bis man auf einen passenden Blatt Knoten trifft. Wurde der Blatt Knoten gefunden, wird der neue Key eingefügt und in einen internen Knoten expandiert.
- Wenn man eine Multimap verwendet, werden mehrfache Knoten immer im linken Teilbaum eingefügt.

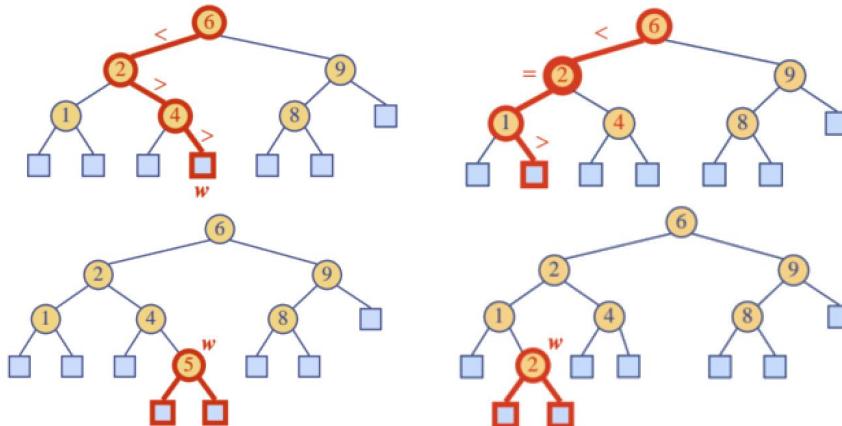


Abbildung 3: Einfügen wenn der Key Abbildung 4: Einfügen wenn der Key  
5 noch nicht vorhanden 2 bereits vorhanden

Listing 2: ArrayList basierter Einsatz

---

```

1  public void add(int lower, int upper, int content) {
2      int middle = (lower + upper) / 2;
3      if (content < arrayList.get(middle).intValue()) {
4          // go left
5          if (middle == 0 || content > arrayList.get(middle - 1)) {
6              arrayList.add(middle, content);
7          } else {
8              add(lower, middle - 1, content);
9          }
10     } else {
11         // go right
12         if (middle + 1 >= arrayList.size() || content < arrayList.get(middle +
13             1).intValue()) {
14             arrayList.add(middle + 1, content);
15         } else {
16             add(middle + 1, upper, content);
17         }
18     }
}

```

---

**remove(key)**

- Beim Löschen muss die Inorder Traversierung erhalten bleiben
- Beim Remove kann es vorkommen, dass gleiche Keys im linken und rechten Teilbaum der Root zu liegen kommen. Dies muss dann bei der Suche beachtet werden.
- Es wird zwischen drei Varianten unterschieden:
  - Zu löschernder Knoten hat **zwei Blatt Kinder**: `removeExternal(w)` löscht den Blattknoten w und seinen Parent und ersetzt den Parent mit dem Geschwisterknoten von w
  - Zu löschernder Knoten hat **ein Blatt Kind**: Genau gleich wie beider Vorgehensweise mit zwei Blatt Kinder, jedoch mit dem Unterschied, dass der neue "Parent" kein Blattknoten ist.
  - Zu löschernder Knoten hat **keine Blatt Kinder**: Man nimmt den nächsten Knoten in der Inorder Traversierung und ersetzt den zu löschen Key mit diesem. Der Key der kopiert wurde wird dann gelöscht.

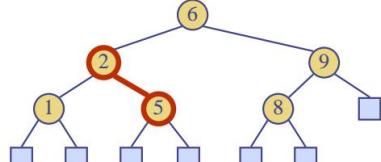
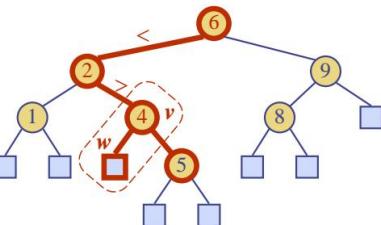
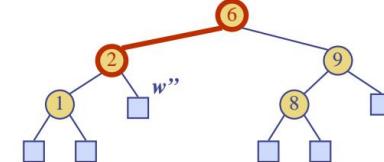
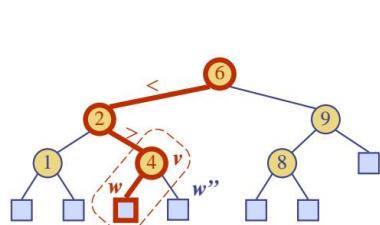


Abbildung 5: Zwei Blatt Kinder.

Abbildung 6: Ein Blatt Kind.

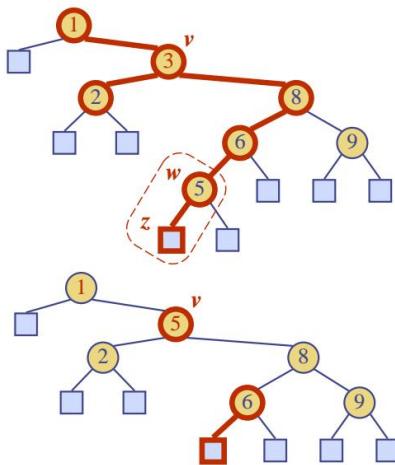


Abbildung 7: Keine Blatt Kinder

## 5.4. Binäre Sortierung

Eine Menge von n Zahlen kann sortiert werden, indem man diese zunächst in einen binären Suchbaum einfügt und dann durch das Inorder Traversal in sortierter Reihenfolge wieder ausgeben lässt.

**Worst Case** Im Worst-Case werden die Elemente sortiert eingefügt z.B. 1,2,3,4,5,6,...,n.

Dies führt dazu, dass für das Element x auch x Schritte nötig sind um es einzufügen. Für alle Elemente 1...n also:  $\sum_{i=1}^n i = \frac{n(n+1)}{2}$  Dies entspricht der Laufzeit  $\mathcal{O}(n^2)$

**Best Case**

Der Best-Case ist, wenn Elemente so eingefügt werden, dass ein vollständiger Binärbaum entsteht. Dies führt dazu, dass für das Element x nur  $\log_2(x)$  Schritte nötig sind um es einzufügen. Für alle Elemente 1..n also:  $\sum_{i=1}^n [\log_2(i)] \leq \sum_{i=1}^n \log_2(i)$  Die Laufzeit entspricht also  $(n \log_2(n))$

**Total**

Nicht abhängig vom Auslesen.

## 5.5. Speicherverbrauch

Binäre Suchbäume haben folgenden Speicherverbrauch

Beschreibung	Big Oh
Speicherverbrauch	$\mathcal{O}(n)$
Höhe	$\mathcal{O}(\log(n))$ im besten Fall $\mathcal{O}(n)$ im schlechtesten Fall

Tabelle 4: Speicherverbrauch von Binären Suchbäumen

## 5.6. Implementierungen

Listing 3: BST Node

```

1  public class Node {
2      private Entry<K, V> entry;
3      private Node leftChild;
4      private Node rightChild;
5
6      Node(int key) {
7          this.key = key;
8      }
9
10     public Entry<K, V> setEntry(Entry<K, V> entry) {
11         Entry<K, V> oldEntry = entry;
12         this.entry = entry;
13         return oldEntry;
14     }
15 }
```

Listing 4: BST Entry

```

1  public static class Entry<K, V> {
2      private K key;
3      private V value;
4
5      protected K setKey(K key) {
6          K oldKey = this.key;
7          this.key = key;
8          return oldKey;
9      }
10
11     public V setValue(V value) {
12         V oldValue = this.value;
13         this.value = value;
14         return oldValue;
15     }
16 }
```

---

```

1  protected class RemoveResult {
2      private Node node;
3      private Entry<K, V> entry;
4  }
```

---

```

1  public class BinarySearchTree<K extends Comparable<? super K>, V> {
2      // Root node
3      Node root;
4      public BinarySearchTree() {
5          root = null;
6      }
7      protected Node newNode(Entry<K, V> entry) {
8          return new Node(entry);
9      }
10     public Entry<K, V> insert(K key, V value) {
11         Entry<K, V> newEntry = new Entry<K, V>(key, value);
12         root = insert(root, newEntry);
13         return newEntry;
14     }
15     protected Node insert(Node node, Entry<K, V> entry) {
16         if (node == null) {
```

```

17         return newNode(entry);
18     } else if (entry.getKey().compareTo(node.getEntry().getKey()) <= 0) {
19         node.leftChild = insert(node.leftChild, entry);
20     } else if (entry.key > node.key) {
21         node.rightChild = insert(node.rightChild, entry);
22     }
23     return node;
24 }
25 public Entry<K, V> find(K key) {
26     Node result = find(root, key);
27     if (result == null) {
28         return null;
29     } else {
30         return result.getEntry();
31     }
32 }
33 protected Node find(Node node, K key) {
34     if (node == null) {
35         return null;
36     }
37     if (key.compareTo(node.getEntry().getKey()) < 0) {
38         return find(node.leftChild, key);
39     }
40     if (key.compareTo(node.getEntry().getKey()) > 0) {
41         return find(node.rightChild, key);
42     }
43     return node;
44 };
45 public Collection<Entry<K, V>> findAll(K key) {
46     Collection<Entry<K, V>> entries = new LinkedList<Entry<K, V>>();
47     findAll(root, key, entries);
48     return entries;
49 }
50
51 protected void findAll(Node node, K key, Collection<Entry<K, V>> entries) {
52     if (node == null) {
53         return;
54     }
55     if (key.compareTo(node.getEntry().getKey()) == 0) {
56         entries.add(node.getEntry());
57     }
58     if (key.compareTo(node.getEntry().getKey()) <= 0) {
59         findAll(node.leftChild, key, entries);
60     }
61     if (key.compareTo(node.getEntry().getKey()) >= 0) {
62         findAll(node.rightChild, key, entries);
63     }
64 }
65
66 public Collection<Entry<K, V>> inorder() {
67     Collection<Entry<K, V>> coll = new LinkedList<>();
68     inorder(root, coll);
69     return coll;
70 }
71
72 public Collection<Entry<K, V>> inorder() {
73     Collection<Entry<K, V>> coll = new LinkedList<>();
74     inorder(root, coll);
75     return coll;
76 }
77
78 protected void inorder(Node node, Collection<Entry<K, V>> coll) {

```

```

79     if (node == null) {
80         return;
81     }
82     inorder(node.getLeftChild(), coll);
83     coll.add(node.getEntry());
84     inorder(node.getRightChild(), coll);
85 }
86
87 public Entry<K, V> remove(Entry<K, V> entry) {
88     if (entry == null) {
89         return null;
90     }
91     RemoveResult result = remove(root, entry);
92     root = result.node;
93     return result.entry;
94 }
95
96 protected RemoveResult remove(final Node node, final Entry<K, V> entry) {
97     RemoveResult result = null;
98     if (node == null) {
99         return new RemoveResult(null, null);
100    }
101    if (entry.getKey().compareTo(node.getEntry().getKey()) < 0) {
102        result = remove(node.leftChild, entry);
103        node.leftChild = result.node;
104        return result.set(node);
105    } else if (entry.getKey().compareTo(node.getEntry().getKey()) > 0) {
106        result = remove(node.rightChild, entry);
107        node.rightChild = result.node;
108        return result.set(node);
109    } else {
110        // Key found: is this the correct entry?
111        if (node.getEntry() != entry) {
112            // Searching for next entry with this key
113            result = remove(node.leftChild, entry);
114            node.leftChild = result.node;
115            if (result.entry == null) {
116                result = remove(node.rightChild, entry);
117                node.rightChild = result.node;
118            }
119            return result.set(node);
120        }
121        // We have reached the correct node.
122        if (node.leftChild == null) {
123            return new RemoveResult(node.rightChild, node.getEntry());
124        }
125        if (node.rightChild == null) {
126            return new RemoveResult(node.leftChild, node.getEntry());
127        }
128        Entry<K, V> entryRemoved = node.getEntry();
129        Node q = getParentNext(node);
130        if (q == node) {
131            node.setEntry(node.rightChild.getEntry());
132            q.rightChild = q.rightChild.rightChild;
133        } else {
134            node.setEntry(q.leftChild.getEntry());
135            q.leftChild = q.leftChild.rightChild;
136        }
137        return new RemoveResult(node, entryRemoved);
138    }
139 }
140

```

```
141     protected Node getParentNext(Node p) {
142         if (p.rightChild.leftChild != null) {
143             p = p.rightChild;
144             while (p.leftChild.leftChild != null) {
145                 p = p.leftChild;
146             }
147         }
148         return p;
149     }
150
151     public int getHeight() {
152         return getHeight(root);
153     }
154
155     protected int getHeight(Node parent) {
156         if (parent == null) {
157             return -1;
158         } else {
159             int leftHeight = getHeight(parent.leftChild);
160             int rightHeight = getHeight(parent.rightChild);
161             return Integer.max(leftHeight, rightHeight);
162         }
163     }
164
165     public int size() {
166         return size(root);
167     }
168
169     protected int size(Node n) {
170         if (n == null) {
171             return 0;
172         }
173         return size(n.leftChild) + size(n.rightChild) + 1;
174     }
175
176     public boolean isEmpty() {
177         return size() == 0;
178     }
179 }
```

---

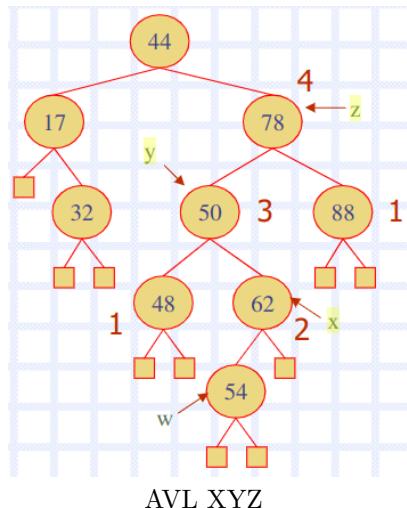
## 6. AVL Tree

- AVL Bäume sind BST die immer balanciert sind! (selbstbalanciert)
- Für jeden internen Knoten: Die Höhe darf sich bei beiden Kind Teilbäume um **höchstens um 1 unterscheiden**. (AVL Eigenschaft)
- Man spricht von der **Balance = Höhe(links) - Höhe(rechts)**. Die Balance darf somit -1, 0 oder 1 sein. **=Balancierungsfaktor**
- Die Höhe ist immer  $\mathcal{O}(\log(n))$
- Die maximale (und mittlere) Anzahl Vergleiche, die nötig sind, um einen Schlüssel zu finden, hängt direkt mit der Höhe zusammen
- Er wurden von Georgi Maximowitsch Adelson-Velski und Jewgeni Michailowitsch Landis (AVL) erfunden

### 6.1. Laufzeiten

Methode	Best und Worst Case
find()	$\mathcal{O}(\log(n))$
insert()	$\mathcal{O}(\log(n))$
remove()	$\mathcal{O}(\log(n))$
rebalance()	$\mathcal{O}(\log(n))$

Tabelle 5: Laufzeitverhalten von AVL Trees

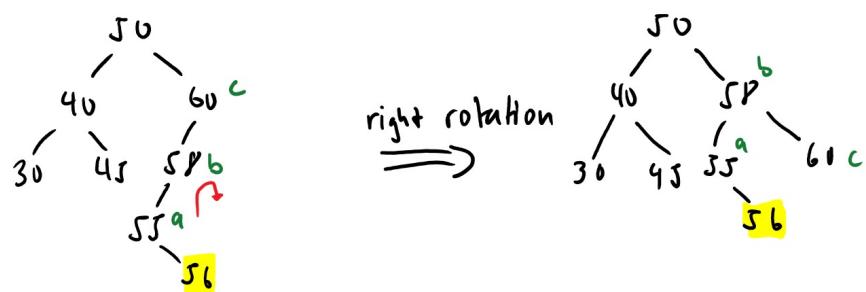
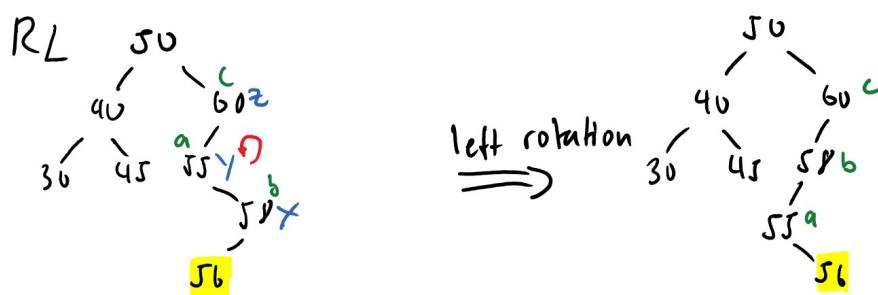
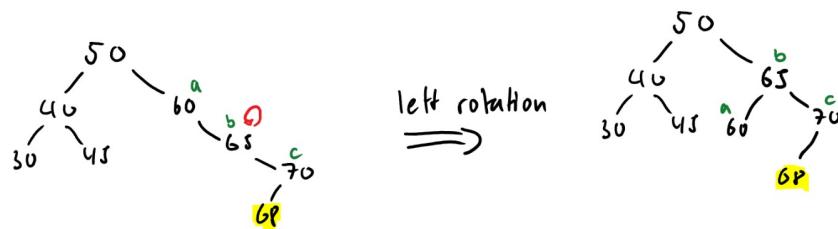
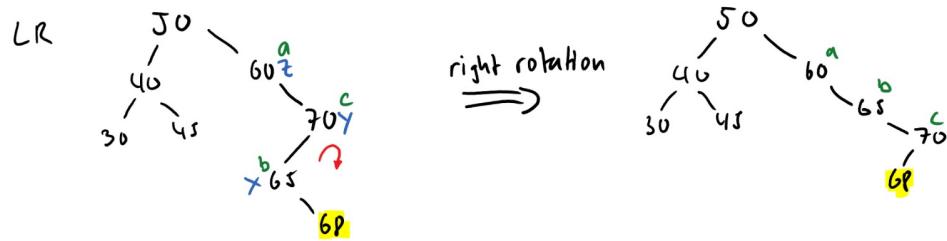
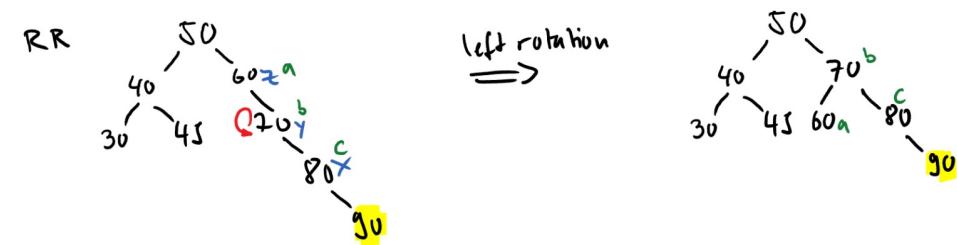


## 6.2. Operationen

**insert()** Wenn der AVL Tree nach dem Einfügen unbalanciert ist, sucht man aufwärts in Richtung root, bis zum dem Knoten x, dessen Grosseltern (2 Ebenen höher) ein unbalancierten Knoten ist. Durch die **Trinode Umstrukturierung** (Siehe weiter unten) kann der AVL Tree wieder ausbalanciert werden. Ausgehend vom eingefügten Knoten ist x = Parent, y = Grandparent, z = Grandgrandparent.

1. x ist der gefundene Knoten
2. y ist der parent des gefundenen Knoten
3. z ist der grandparent des gefundenen Knoten und der **Knoten der die AVL Eigenschaft verletzt!**
4. Für x,y,z die Inorder Reihenfolge erstellen → (a,b,c)
5. Baum **rotieren** gemäss a,b,c Anordnung
  - LL: a zuunterst (schlägt nach links aus) → Rechts rotieren (b wird parent)
  - RR: c zuunterst (schlägt nach rechts aus) → Links rotieren (b wird parent)
  - LR: b zuunterst (hat Richtungswechsel links) → Doppelrotation rechts, links (b wird parent)
  - RL: b zuunterst (hat Richtungswechsel rechts) → Doppelrotation links, rechts (b wird parent)





**remove()**

- Das Löschen funktioniert wie beim BST. (Siehe 5.3) Ist der AVL Tree nach dem Löschen unbalanciert, muss er wieder in die Balance gebracht werden. Dazu müssen zuerst die Knoten identifiziert werden:
  - z ist der erste unbalancierter Knoten
  - y ist das Kind von z mit der **grösseren Höhe** als sein Sibling
  - x ist das Kind von y mit der **grösseren Höhe** als sein Sibling
- Die Umstrukturierung kann eine neue Unbalance hervorrufen bei höheren Knoten im Baum. Somit muss die Balance weiter geprüft werden bis die Wurzel erreicht ist.

### 6.3. Rotationen / Trinode Umstrukturierung

Es gibt vier Varianten von Rotationen. Nach jeder Rotation muss die Inorder Traversal gleich bleiben!

Listing 5: AVL Tree Rotations

---

```

1 protected AVLNode restructure(AVLNode xPos) {
2     AVLNode yPos = xPos.getParent();
3     AVLNode zPos = yPos.getParent();
4     AVLNode newSubTreeRoot = null;
5     if (yPos == zPos.getLeftChild()) {
6         if (xPos == yPos.getLeftChild()) {
7             newSubTreeRoot = rotateWithLeftChild(zPos);
8         } else {
9             newSubTreeRoot = doubleRotateWithLeftChild(zPos);
10        }
11    } else {
12        if (xPos == yPos.getRightChild()) {
13            newSubTreeRoot = rotateWithRightChild(zPos);
14        } else {
15            newSubTreeRoot = doubleRotateWithRightChild(zPos);
16        }
17    }
18    return newSubTreeRoot;
19 }
```

---

#### 6.3.1. Rechts rotieren (einfach)

Man rotiert auf die rechte Seite, wenn der Baum nach links schlägt (a zuunterst).

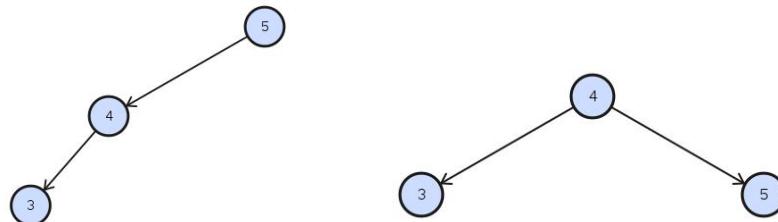


Abbildung 8: Rechts Rotation um c   Abbildung 9: Nach der rechts Rotation

Listing 6: AVL Tree: Single right rotation

---

```

1 protected AVLNode rotateWithLeftChild(AVLNode k2) {
2     AVLNode k1 = k2.getLeftChild();
3     k2.setLeftChild(k1.getRightChild());
4     k1.setRightChild(k2);
5
6     if (k2.getLeftChild() != null) {
7         k2.getLeftChild().setParent(k2);
8     }
9     adjustParents(k2, k1);
10
11    return k1;
12 }
```

---

### 6.3.2. Links rotieren (einfach)

Man rotiert auf die linke Seite, wenn der Baum nach rechts schlägt (c zuunterst).

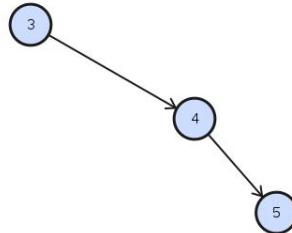


Abbildung 10: Links Rotation um a

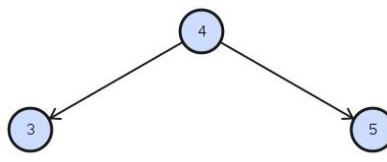


Abbildung 11: Nach der Links Rotation

Listing 7: AVL Tree: Single left rotation

---

```

1 protected AVLNode rotateWithRightChild(AVLNode k1) {
2     AVLNode k2 = k1.getRightChild();
3     k1.setRightChild(k2.getLeftChild());
4     k2.setLeftChild(k1);
5
6     if (k1.getRightChild() != null) {
7         k1.getRightChild().setParent(k1);
8     }
9     adjustParents(k1, k2);
10
11    return k2;
12 }
```

---

### 6.3.3. Rechts/Links Doppelrotation

Man rotiert zuerst nach rechts und dann nach links, wenn man einen Knick auf die linke Seite hat. (b zuunterst)

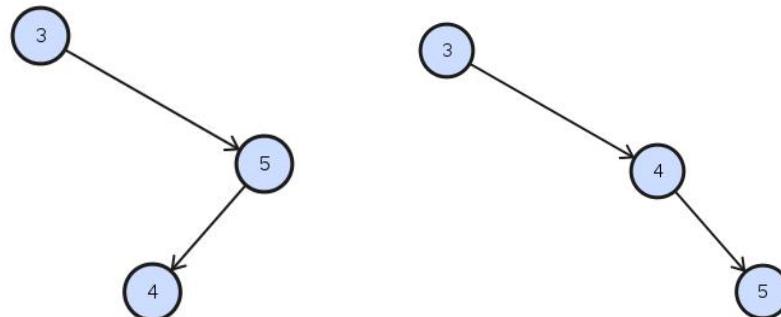


Abbildung 12: Rechts Rotation um b   Abbildung 13: Links Rotation um a

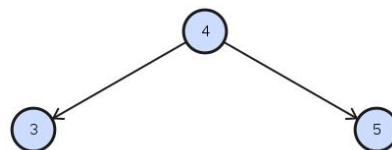


Abbildung 14: Nach Rechts/Links Ro-tation

Listing 8: AVL Tree: Right/Left Rotation

---

```

1 protected AVLNode doubleRotateWithLeftChild(AVLNode k3) {
2     AVLNode rotatedRight = rotateWithRightChild(k3.getLeftChild());
3     k3.setLeftChild(rotatedRight);
4     return rotateWithLeftChild(k3);
5 }
```

---

### 6.3.4. Links/Rechts Doppelrotation

Man rotiert zuerst nach links und dann nach rechts, wenn man einen Knick auf die rechte Seite hat. (b zuunterst)

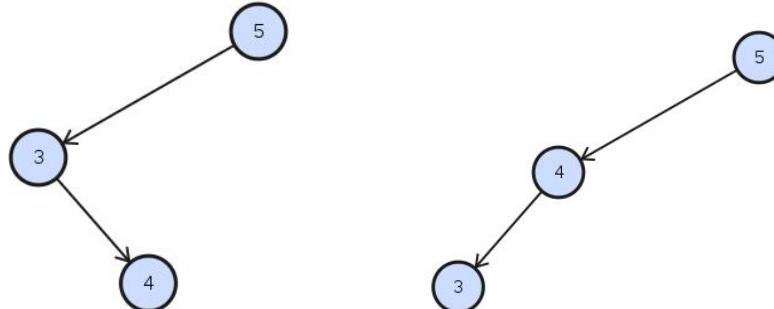


Abbildung 15: Links Rotation um a    Abbildung 16: Rechts Rotation um c

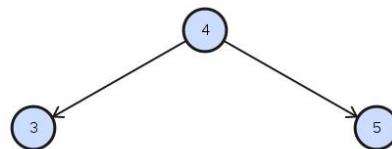


Abbildung 17: Nach Links/Rechts Rotation

Listing 9: AVL Tree: Left/Right Rotation

---

```

1 protected AVLNode doubleRotateWithRightChild(AVLNode k3) {
2     AVLNode rotatedLeft = rotateWithLeftChild(k3.getRightChild());
3     k3.setRightChild(rotatedLeft);
4     return rotateWithRightChild(k3);
5 }
```

---

## 6.4. Cut/Link Restrukturierung

Die Cut/Link Restrukturierung bewirkt das selbe wie die Rotationen. Er ist zwar komplexer, dafür eleganter, da man keine Fallunterscheidung machen muss. Die Laufzeit ist aber gleich wie bei den Rotationen.

1. Wie bei den Rotationen muss zuerst x,y,z und a,b,c identifiziert werden
  - a) x ist der gefundene Knoten
  - b) y ist der parent des gefundenen Knoten
  - c) z ist der grandparent des gefundenen Knoten und der **Knoten der die AVL Eigenschaft verletzt!**
  - d) Wenn nicht klar Inorder Reihenfolge für x,y,z verwenden
  - e) Für x,y,z die Inorder Reihenfolge erstellen → (a,b,c)
2. Identifizierte die Subtrees  $T_0, T_1, T_2, T_3$  (Inorder Traversierung) von a,b und c
3. Aufschreiben des Inorder Arrays (1-7) → Inorder Reihenfolge (b immer in der Mitte)
 

$T_0$	a	$T_1$	b	$T_2$	c	$T_3$
1	2	3	4	5	6	7

Tabelle 6: Inorder Array für Cut/Link Restrukturierung

4. Setze a als linkes Kind von b
5. Setze c als rechtes Kind von b
6. Setze  $T_0$  als linken und  $T_1$  als rechten Unterbaum von a
7. Setze  $T_2$  als linken und  $T_3$  als rechten Unterbaum von c.

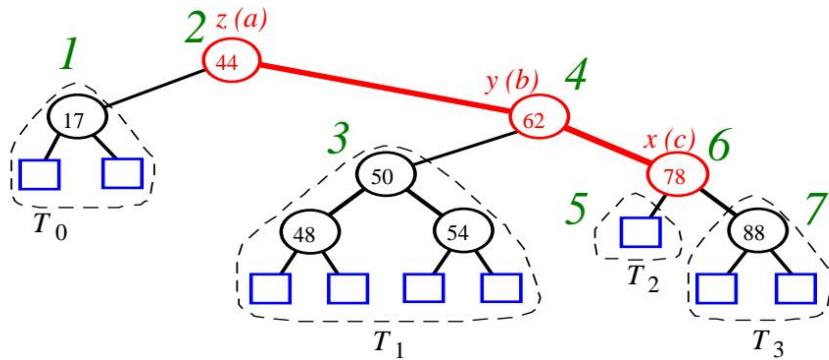


Abbildung 18: Cut/Link Restrukturierung

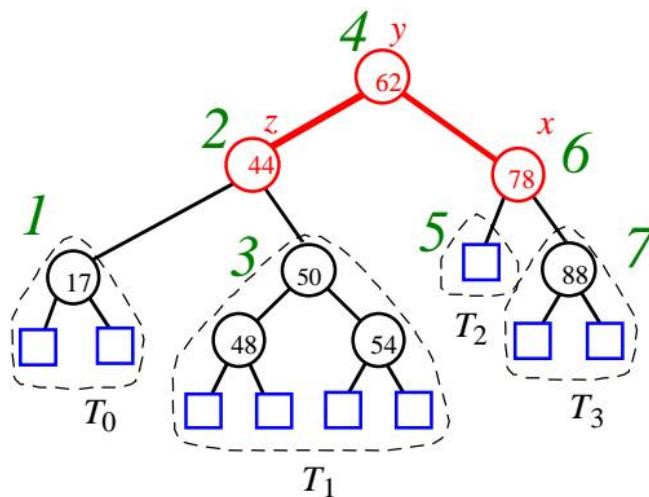


Abbildung 19: Balancierter Baum nach Cut/Link

## 6.5. Implementierung

Listing 10: AVL Tree Node

```

1  protected class AVLNode extends BinarySearchTree<K, V>.Node {
2      private int height;
3      private Node parent;
4
5      AVLNode(Entry<K, V> entry) {
6          super(entry);
7      }
8
9      protected AVLNode setParent(AVLNode parent) {
10         AVLNode old = avlNode(this.parent);
11         this.parent = parent;
12         return old;
13     }
14
15     protected int setHeight(int height) {
16         int old = this.height;
17         this.height = height;
18         return old;
19     }
20
21     protected AVLNode getParent() { .. }
22     protected int setHeight() { .. }
23
24
25     @Override
26     public AVLNode getLeftChild() {
27         return avlNode(super.getLeftChild());
28     }
29
30     @Override
31     public AVLNode getRightChild() {
32         return avlNode(super.getRightChild());
33     }

```

34 }

Listing 11: AVL Tree

```

1  class AVLTreeImpl<K extends Comparable<? super K>, V> extends BinarySearchTree<K, V> {
2
3      protected AVLNode actionNode;
4
5      protected AVLNode getRoot() {
6          return avlNode(root);
7      }
8
9      protected AVLNode avlNode(Node node) {
10         return (AVLNode) node;
11     }
12
13     public int getHeight() {
14         return height(avlNode(root));
15     }
16
17     protected int height(AVLNode node) {
18         return (node != null) ? node.getHeight() : -1;
19     }
20
21     public V put(K key, V value) {
22         Entry<K, V> entry = super.find(key);
23         if (entry != null) {
24             // key already exists in the Tree
25             return entry.setValue(value);
26         } else {
27             // key does not exist in the Tree yet
28             super.insert(key, value);
29             rebalance(actionNode);
30             actionNode = null;
31             return null;
32         }
33     }
34
35     public V get(K key) {
36         Entry<K, V> entry = super.find(key);
37         if (entry != null) {
38             return entry.getValue();
39         } else {
40             return null;
41         }
42     }
43
44     @Override
45     protected Node insert(Node node, Entry<K, V> entry) {
46         if (node != null) {
47             actionNode = avlNode(node);
48         }
49         // calling now the BST-insert() which will do the work:
50         AVLNode result = avlNode(super.insert(node, entry));
51         if (node == null) {
52             // In this case: result of super.insert() is the new node!
53             result.setParent(actionNode);
54         }
55         return result;
56     }
57
58     protected Node newNode(Entry<K, V> entry) {

```

```

59         AVLNode avlNode = new AVLNode(entry);
60         return avlNode;
61     }
62
63     public V remove(K key) {
64         Entry<K, V> entry = super.find(key);
65
66         Entry<K, V> toReturn = super.remove(entry);
67         if (toReturn == null) {
68             AVLNode zPos = actionNode;
69             rebalance(zPos);
70         }
71         return toReturn.getValue();
72     }
73
74     protected boolean isBalanced(AVLNode node) {
75         int leftHeight = height(node.getLeftChild());
76         int rightHeight = height(node.getRightChild());
77
78         int balance = leftHeight - rightHeight;
79         return (balance >= -1) && (balance <= 1);
80     }
81
82     protected void rebalance(AVLNode node) {
83         while (node != null) {
84             setHeight(node);
85             if (!isBalanced(node)) {
86                 AVLNode xPos = tallerChild(tallerChild(node));
87                 node = restructure(xPos);
88                 setHeight(node.getLeftChild());
89                 setHeight(node.getRightChild());
90                 setHeight(node);
91             }
92             node = node.getParent();
93         }
94     }
95
96     protected AVLNode tallerChild(AVLNode node) {
97         AVLNode leftChild = node.getLeftChild();
98         AVLNode rightChild = node.getRightChild();
99         if (height(leftChild) >= height(rightChild)) {
100             return leftChild;
101         } else {
102             return rightChild;
103         }
104     }
105
106     protected void setHeight(AVLNode node) {
107         if (node == null) {
108             return;
109         }
110
111         int heightLeftChild = -1;
112         if (node.getLeftChild() != null) {
113             heightLeftChild = node.getLeftChild().getHeight();
114         }
115
116         int heightRightChild = -1;
117         if (node.getRightChild() != null) {
118             heightRightChild = node.getRightChild().getHeight();
119         }
120

```

```
121     node.setHeight(1 + Math.max(heightLeftChild, heightRightChild));
122 }
123
124 protected void adjustParents(final AVLNode oldSubtreeRoot, final AVLNode
125     newSubtreeRoot) {
126     final AVLNode parentSubtree = oldSubtreeRoot.getParent();
127     oldSubtreeRoot.setParent(newSubtreeRoot);
128     if (oldSubtreeRoot == root) {
129         newSubtreeRoot.setParent(null);
130         root = newSubtreeRoot;
131     } else {
132         newSubtreeRoot.setParent(parentSubtree);
133         if (oldSubtreeRoot == parentSubtree.getLeftChild()) {
134             parentSubtree.setLeftChild(newSubtreeRoot);
135         } else {
136             parentSubtree.setRightChild(newSubtreeRoot);
137         }
138     }
139
140 protected void inorder(Collection<AVLNode> nodeList, AVLNode node) {
141     if (node == null)
142         return;
143     inorder(nodeList, node.getLeftChild());
144     nodeList.add(node);
145     inorder(nodeList, node.getRightChild());
146 }
147 }
```

---

## 7. Splay Tree

- Splay Trees sind auch binäre Suchbäume mit der zusätzlichen Operation `splay()`. Dies verschiebt den gefundenen Knoten in die Root.
- Das spezielle an Splay Trees ist es, dass sie nach jeder Operation (auch bei der Suche) dem Baum rotiert wird. Die Rotation ist dieselbe wie bei AVL Trees.
- Oft verwendete Elemente sind somit immer nahe beim Root und können schnell zugegriffen werden! (z.B bei Suchmaschinen, Caching, Garbage Collection)
- Das Bewegen eines Knoten zur Root unter Benutzung von Rotationen nennt man Splaying
  - **Zig:** linkes Kind (rechtsrotation)
  - **Zag:** rechtes Kind (linksrotation)
- Die Rotation hängt davon ab, welcher Typ von Kind x ist (links-rechts, rechts-rechts, etc.) z.B Ist x das linke Kind von seinem Parent, welcher selber ein rechtes Kind von seinem Parent ist ( $x = \text{links-rechts Grosskind}$ ) → **Immer ausgehend von } x!**
- Der Knoten x wird nach einem Zugriff zur Root bewegt (nach Update und Suchen)

### 7.1. Varianten

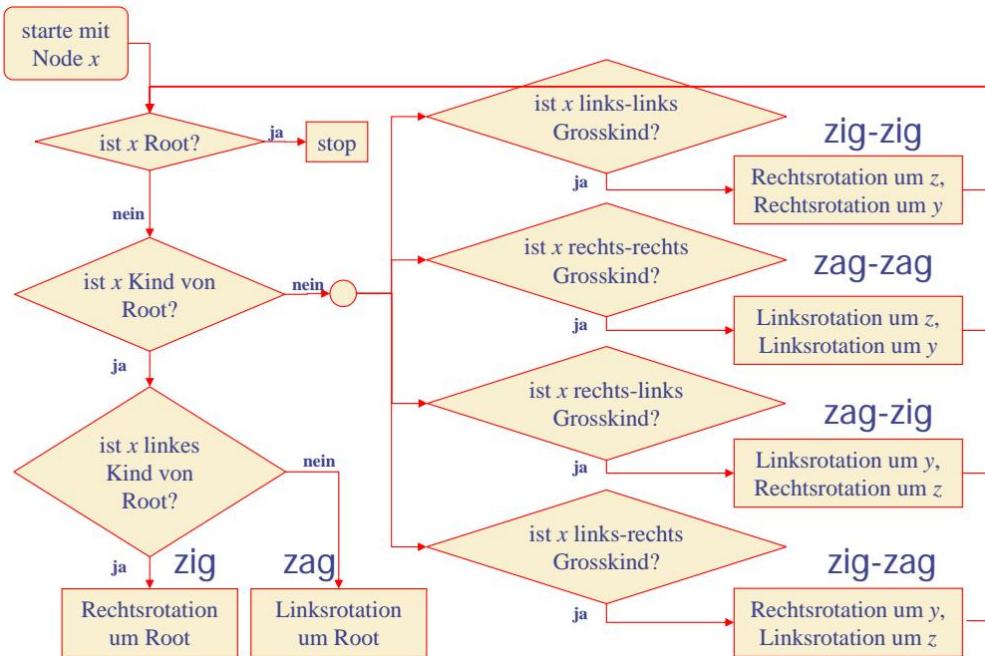


Abbildung 20: Splay Tree Flussdiagramm

## 7.2. Vorgehen

1. Knoten identifizieren
  - a) x ist der gesuchte/eingefügte Knoten (Spezialfall löschen)
  - b) y ist der Parent von x
  - c) z ist der Parent von y
2. Gemäss Flussdiagramm korrekte Fall auswählen und Rotation durchführen.

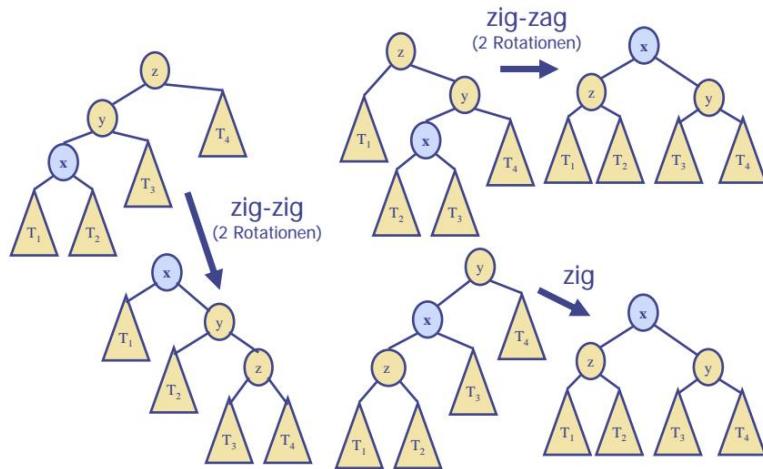


Abbildung 21: Splay Tree Beispiele

## 7.3. Remove

1. Ersetze den zu löschen Knoten mit dem **Inorder Nachfolger**
2. Splay des Inorder Nachfolgers des ersetzen Knotens
3. Rotation gemäss Zig-Zag Schema bis dieser in Root wird

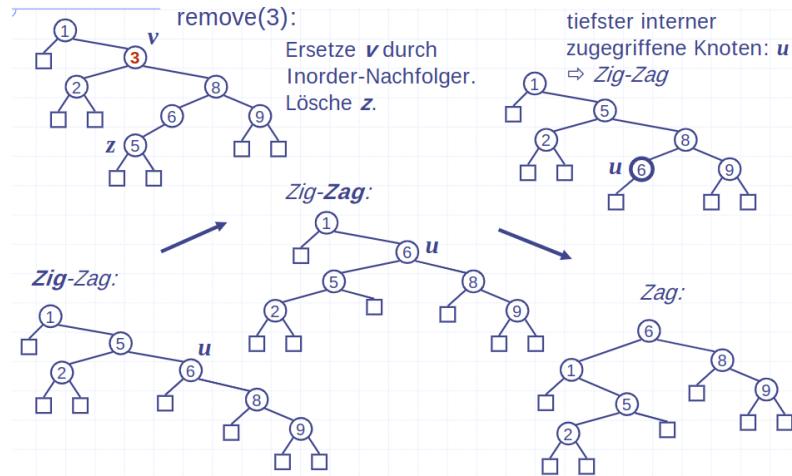


Abbildung 22: Splay Tree: Löschen des Wert 3

## 7.4. Splaying

Bei den Operationen werden jeweils andere Knoten gesplayed. Ziel jeder Operation ist es, dass der betroffenen Knoten immer als Root gesetzt wird. Dabei wird der zu findende/löschende Knoten  $x$  genannt.

Methode	Knoten zum Splayen
find( $k$ )	wenn der Key gefunden, benutze diesen Knoten wenn Key nicht gefunden, benutzen den Parent des externen Knoten am Ende
insert( $k, v$ )	Benutze den neuen Knoten bei welchem der Entry eingefügt/ersetzt wurde
remove( $k$ )	Benutze den Parent des internen Knotens welcher gelöscht wurde.

Tabelle 7: Laufzeitverhalten von Splay Trees

## 7.5. Laufzeiten

Methode	Laufzeitverhalten	Beschreibung
splay()	$\mathcal{O}(h) \rightarrow \mathcal{O}(n)$ $\mathcal{O}(\log(n))$	$h = \text{Height}$ , Worst Case Durchschnitt

Tabelle 8: Laufzeitverhalten von Splay Trees

## 8. Sortieralgorithmen

### 8.1. Eigenschaften

#### inplace

Ein Algorithmus arbeitet inplace, wenn er nur den Speicherplatz für die Input Daten benötigt und zusätzlich nur konstanten (**vom Input unabhängige** Menge an Speicher) verwendet. Der Algorithmus überschreibt also die Eingabedaten mit den Ausgabedaten.

#### stabil

Die relative Ordnung von zwei Items mit dem selben Key werden durch den Algorithmus nicht verändert. Die Ordnung bleibt in der Zielsequenz erhalten. (z.B wichtig bei Bestellungen vom selben Kunden. Die Reihenfolge muss erhalten bleiben)

### 8.2. Varianten

**Vergleichsbasierte Sortieralgorithmen** Vergleichsbasierte Algorithmen basieren auf dem paarweise Vergleich der zu sortierenden Elemente. Bei der Komplexitätsanalyse wird davon ausgegangen, dass der Aufwand zum Vergleich zweier Elemente konstant ist.

**Nicht vergleichsbasierte Sortieralgorithmen** Bei Sortierverfahren, die nicht auf Vergleichen beruhen, kann ein linearer Anstieg der benötigten Zeit mit der Anzahl der zu sortierenden Elemente erreicht werden. Bei großen Anzahlen zu sortierender Datensätze sind diese Algorithmen **den vergleichsbasierten Verfahren überlegen**, sofern sie (wegen des **zusätzlichen Speicherbedarfs**) angewendet werden können. Zudem können sie allerdings nur für numerische Datentypen verwendet werden.

### 8.3. Laufzeiten

Die untere Grenze aller folgenden Algorithmen kann mit der Stirling Formel (1.4) hergeleitet werden. Vergleichsbasierte Algorithmen wie Bubble Sort, Selection Sort, Insertion Sort, Heap Sort, Merge Sort, Quick Sort haben eine minimale Laufzeit von  $\Omega(n \cdot \log(n))$  (**Untere Grenze**)

Algorithmus	Big Oh	Bemerkung
Selection Sort	$\mathcal{O}(n^2)$	langsam, <b>in-place</b> , für kleine Daten Sets (< 1K)
Insertion Sort	$\mathcal{O}(n^2)$	langsam, <b>in-place, stable</b> , für kleine Daten Sets (< 1K)
Heap Sort	$\mathcal{O}(n \cdot \log(n))$	schnell, <b>in-place</b> , für grosse Daten Sets (1K - 1M)
Merge Sort	$\mathcal{O}(n \cdot \log(n))$	schnell, <b>stable</b> sequentieller Datenzugriff, für riesige Data Sets (> 1M)
Quick Sort	$\mathcal{O}(n \cdot \log(n))$	schnellster, <b>in-place</b> , (typischweise nicht stable)

Tabelle 9: Laufzeitverhalten von vergleichbasierten Sortieralgorithmen

Algorithmus	Big Oh	Bemerkung
Bucket Sort	$\mathcal{O}(n + N)$	Nur für positive Ganzzahlen
Radix Sort	$\mathcal{O}(d \cdot (n + N))$	d = Anzahl Tupel, N = Max Key Bereich

Tabelle 10: Laufzeitverhalten von nicht vergleichbasierten Sortieralgorithmen

### 8.4. Lexikographische Sortierung

- Ein Tupel ist ein Satz von Werten der i-ten Ordnung (3 Tupel = kartesische Koordinaten im Raum)
- Für die Lexikographische Sortierung ist ein Comparator nötig, der zwei Tupel nach ihrer i-ten Dimension vergleicht
- Für Lexikographische Sortierung ist ebenfalls ein stabiler Sortieralgorithmus nötig
- Lexikografische Sortierung läuft mit  $\mathcal{O}(n \cdot S(n))$ , wobei  $S(n)$  der Laufzeit des stabilen Sortieralgorithmus darstellt.

---

**Algorithm 2:** lexicographicSort(S)

---

**Data:** sequence S of d-Tuples

**Result:** sequence S sorted in lexicographic order

```

1 for i ← d downto 1 do
2   | stableSort(S, Ci)
3 end

```

---

(7,4,6) (5,1,5) (2,4,6) (2, 1, 4) (3, 2, 4)

i=3 (2, 1, 4) (3, 2, 4) (5,1,5) (7,4,6) (2,4,6)

i=2 (2, 1, 4) (5,1,5) (3, 2, 4) (7,4,6) (2,4,6)

i=1 (2, 1, 4) (2,4,6) (3, 2, 4) (5,1,5) (7,4,6)

Abbildung 23: Lexikographische Sortierung

## 9. Bubble Sort

- Der Bubble Sort ist ein sehr trivialer Sortieralgorithmus
- Er loopt über eine Sequenz und vertauscht ein Item mit dem Nächsten, falls dieses grösser als das Nächste ist. Die wird solange wiederholt, bis keine Vertauschungen stattgefunden haben.
- Ist stabil

### 9.1. Laufzeiten

Big Oh	Beschreibung	Bemerkung
Best Case	$\mathcal{O}(n)$	Aufsteigend sortierte Sequenz (1 Iteration, n Vergleiche)
Worst Case	$\mathcal{O}(n^2)$	Absteigend sortierte Sequenz (n Iterationen, n Vergleiche)
Worst Case mit Optimierung	$\mathcal{O}(n^2)$	Absteigend sortierte Sequenz (n - 1 Iterationen, n - i Vergleiche)

Tabelle 11: Big Oh Merge Sort

Listing 12: Bubble Sort

```

1  public static <T extends Comparable<? super T>> void bubbleSort2(T[] sequence) {
2      // performance improvement -> last item is always sorted after iteration
3      int max = sequence.length;
4      boolean sorted = false;
5      do {
6          boolean swapped = false;
7          for (int i = 1; i < sequence.length; i++) {
8              if (sequence[i].compareTo(sequence[i - 1]) < 0) {
9                  T temp = sequence[i];
10                 sequence[i] = sequence[i - 1];
11                 sequence[i - 1] = temp;
12                 swapped = true;
13             }
14         }
15         max--;
16         if (!swapped) {
17             sorted = true;
18         }
19     } while(!sorted);
20 }
```

## 10. Merge Sort

- Merge Sort basiert auf **Divide and Conquer**
- Läuft mit  $\mathcal{O}(n \cdot \log(n))$  (gleich wie der Heap Sort)
- Die Verankerung der Rekursion ist immer ein Teilproblem der grössse 0 oder 1
- Merge Sort wird von Java für die Sortierung verwendet. (Für die Sortierung von primitven Typen wird QuickSort verwendet.)
- Im Gegensatz zum Quicksort finden die Vergleiche beim Merge Sort während dem Rücklauf der Rekursion ab.
- Ist **stable** aber nicht in-place
- Ist meist langsamer wie Quick Sort.

```
Algorithm mergeSort(S, C)
  Input sequence S with n
    elements, comparator C
  Output sequence S sorted
    according to C
  if S.size() > 1
    (S1, S2)  $\leftarrow$  partition(S, n/2)
    S1  $\leftarrow$  mergeSort(S1, C)
    S2  $\leftarrow$  mergeSort(S2, C)
    S  $\leftarrow$  merge(S1, S2)
  return S
```

```
Algorithm merge(A, B)
  Input sequences A and B with
    n/2 elements each
  Output sorted sequence of A  $\cup$  B
  S  $\leftarrow$  empty sequence
  while  $\neg A.isEmpty() \wedge \neg B.isEmpty()$ 
    if A.first().element() < B.first().element()
      S.insertLast(A.remove(A.first()))
    else
      S.insertLast(B.remove(B.first()))
  while  $\neg A.isEmpty()$ 
    S.insertLast(A.remove(A.first()))
  while  $\neg B.isEmpty()$ 
    S.insertLast(B.remove(B.first()))
  return S
```

Abbildung 24: Merge Sort Algorithmus  
Abbildung 25: Sequenzen zusammenfügen

### 10.1. Laufzeiten

Big Oh	Beschreibung
Höhe	$\mathcal{O}(\log(n))$
Laufzeit Sortieren	$\mathcal{O}(n \cdot \log(n))$

Tabelle 12: Big Oh Merge Sort

Listing 13: Nicht Rekursiver Merge Sort

---

```

1  public static void mergeSort(Object[] orig, Comparator c) {
2
3      Object[] in = new Object[orig.length];
4
5      // make a new temporary array
6      System.arraycopy(orig,0,in,0,in.length);
7
8      // copy the input
9      Object[] out = new Object[in.length]; // output array
10     Object[] temp; // temp array reference used for swapping.
11     int n = in.length;
12
13     // each iteration sorts all length-2*i runs
14     for (int i=1; i < n; i*=2) {
15
16         // each iteration merges two length-i pair
17         for (int j=0; j < n; j +=2*i) {}
18             // merge from in to out two length-i runs at j
19             merge(in,out,c,j ,i);
20         }
21
22         // swap arrays for next iteration
23         temp = in;
24         in = out;
25         out = temp;
26     }
27     // the "in" array contains the sorted array, so re-copy it
28     System.arraycopy(in,0,orig,0,in.length);
29 }
30
31 protected static void merge(Object[] in, Object[] out, Comparator c, int start, int
32     inc) {
33     // merge in[start..start+inc-1] and
34     // in[start+inc..start+2*inc-1]
35     int x = start; // index into run # 1
36     int end1 = Math.min(start+inc, in.length);
37
38     // boundary for run # 1
39     int end2 = Math.min(start+2*inc, in.length);
40
41     // boundary for run # 2
42     int y = start+inc;
43
44     // index into run # 2 (could be beyond array boundary)
45     int z = start; // index into the out array
46     while ((x < end1) && (y < end2)) {
47         if (c.compare(in[x],in[y]) <= 0) {
48             out[z++] = in[x++];
49         } else {
50             out[z++] = in[y++];
51         }
52     }
53
54     // first run didn't finish
55     if (x < end1) {
56         System.arraycopy(in, x, out, z, end1 - x);
57     } else if (y < end2) { // second run didn't finish
58         System.arraycopy(in, y, out, z, end2 - y);
59     }

```

---

Listing 14: Rekursiver Merge Sort

---

```

1  public class MergeSort {
2
3      public static <T extends Comparable<? super T>> T[] mergeSort(T[] s) {
4
5          if (s.length > 1) {
6              int half = s.length / 2;
7              T[] s1 = newInstance(s, half);
8              System.arraycopy(s, 0, s1, 0, half);
9              T[] s2 = newInstance(s, half);
10             System.arraycopy(s, half, s2, 0, half);
11             s1 = mergeSort(s1);
12             s2 = mergeSort(s2);
13             s = merge(s1, s2);
14         }
15
16         return s;
17     }
18
19     static <T extends Comparable<? super T>> T[] merge(T[] a, T[] b) {
20         T[] s = newInstance(a, a.length * 2);
21         int ai = 0; // First Element in 'Sequence' A
22         int bi = 0; // First Element in 'Sequence' B
23         int si = 0; // First Element in 'Sequence' S
24
25         while (ai < a.length && bi < b.length) {
26             if (a[ai].compareTo(b[bi]) < 0) {
27                 s[si++] = a[ai++];
28             } else {
29                 s[si++] = b[bi++];
30             }
31         }
32         while (ai < a.length) {
33             s[si++] = a[ai++];
34         }
35         while (bi < b.length) {
36             s[si++] = b[bi++];
37         }
38         return s;
39     }
40 }
```

---

## 11. Quick Sort

- Beim Quicksort wird die Menge in **drei Teile unterteilt (Divide)**
  1. **L**: Less: Alle Elemente kleiner x
  2. **E**: Pivot: Alle Elemente gleich x
  3. **G**: Greater: Alle Elemente grösser x
- **Recur**: Sortiere L und G
- **Conquer**: vereine L, E und G
- Es gibt verschiedenen Möglichkeiten um das **Pivot** zu wählen. (Oft zufällig, am besten der Median)
- Die Laufzeit hängt stark von der Wahl des Pivots ab. Er ist deshalb nicht geeignet für Realtime Applikationen. Wird das Pivot jedoch gut gewählt, ist der Algorithmus sehr schnell.
- Der Quick Sort Algorithmus ist typischerweise nicht stabil, da er mit Swap Operationen arbeitet. Es gibt stabile Varianten.
- **Good Call** L und G sind beide kleiner als  $3s/4$
- **Bad Call** Entweder L oder G ist länger als  $3s/4$

**Algorithm *partition(S, p)***

**Input** sequence *S*, position *p* of pivot  
**Output** subsequences *L, E, G* of the elements of *S* less than, equal to, or greater than the pivot, resp.

```

L, E, G ← empty sequences
x ← S.remove(p)
E.insertLast(x)
while  $\neg S.isEmpty()$ 
    y ← S.remove(S.first())
    if y < x
        L.insertLast(y)
    else if y = x
        E.insertLast(y)
    else { y > x }
        G.insertLast(y)
return L, E, G

```

Abbildung 26: Aufteilung in der Inputsequenz.  $O(n)$

## 11.1. Laufzeiten

Big Oh	Beschreibung	Bemerkung	Anzahl Vergleiche
Worst Case Laufzeit	$\mathcal{O}(n^2)$	Wenn Pivot das Minimum oder Maximum ist	
Best Case Laufzeit	$\mathcal{O}(n \cdot \log(n))$	Wen das Pivot der Median	$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$
Höhe	$\mathcal{O}(\log(n))$	an	

Tabelle 13: Big Oh Quick Sort

## 11.2. In Place Implementierung

1. Pivot  $x = 6$
2. Wiederholung bis  $h$  und  $k$  sich kreuzen
  - a)  $h$  nach rechts bis zu einem Element  $\geq$  Pivot  $x$
  - b)  $k$  nach links bis zu einem Element  $<$  Pivot  $1x$
  - c) Wenn  $h$  und  $k$  noch nicht gekreuzt: Elemente mit Indizes  $h$  und  $k$  vertauschen

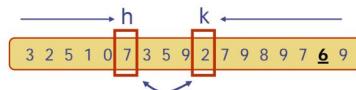


Abbildung 27: InPlace Quicksort

```

Algorithm inPlaceQuickSort(S, l, r)
Input sequence S, ranks l and r
Output sequence S with the
elements of rank between l and r
rearranged in increasing order
if l  $\geq$  r
    return
i  $\leftarrow$  a random integer between l and r
x  $\leftarrow$  S.elemAtRank(i)
(h, k)  $\leftarrow$  inPlacePartition(x)
inPlaceQuickSort(S, l, h - 1)
inPlaceQuickSort(S, k + 1, r)

```

Abbildung 28: In Place Quick Sort Algorithmus

Listing 15: Quick Sort ohne Comparator

```

1  public class MyQuickSort {
2
3      private int array[];
4      private int length;
5
6      public void sort(int[] inputArr) {
7
8          if (inputArr == null || inputArr.length == 0) {
9              return;
10         }
11         this.array = inputArr;
12         length = inputArr.length;
13         quickSort(0, length - 1);
14     }
15
16     private void quickSort(int lowerIndex, int higherIndex) {
17
18         int i = lowerIndex;
19         int j = higherIndex;
20         // calculate pivot number, I am taking pivot as middle index number
21         int pivot = array[lowerIndex+(higherIndex-lowerIndex)/2];
22         // Divide into two arrays
23         while (i <= j) {
24             while (array[i] < pivot) {
25                 i++;
26             }
27             while (array[j] > pivot) {
28                 j--;
29             }
30             if (i <= j) {
31                 exchangeNumbers(i, j);
32                 //move index to next position on both sides
33                 i++;
34                 j--;
35             }
36         }
37         // call quickSort() method recursively
38         if (lowerIndex < j)
39             quickSort(lowerIndex, j);
40         if (i < higherIndex)
41             quickSort(i, higherIndex);
42     }
43
44     private void exchangeNumbers(int i, int j) {
45         int temp = array[i];
46         array[i] = array[j];
47         array[j] = temp;
48     }
49
50     public static void main(String a[]){
51
52         MyQuickSort sorter = new MyQuickSort();
53         int[] input = {24,2,45,20,56,75,2,56,99,53,12};
54         sorter.sort(input);
55         for(int i:input){
56             System.out.print(i);
57             System.out.print(" ");
58         }
59     }
60 }
```

Listing 16: Inplace Quick Sort

---

```

1  public static void quickSort (Object[] S, Comparator c) {
2      if (S.length < 2) {
3          return; // the array is already sorted in this case
4      }
5      quickSortStep(S, c, 0, S.length-1); // recursive sort method
6  }
7
8  private static void quickSortStep (Object[] S, Comparator c, int leftBound, int
9      rightBound ) {
10
11     if (leftBound >= rightBound) {
12         return; // the indices have crossed
13     }
14
15     Object temp; // temp object used for swapping
16     Object pivot = S[rightBound];
17     int leftIndex = leftBound; // will scan rightward
18     int rightIndex = rightBound-1; // will scan leftward
19     while (leftIndex <= rightIndex) {
20         // scan right until larger than the pivot
21         while ( (leftIndex <= rightIndex) && (c.compare(S[leftIndex], pivot)<=0) ) {
22             leftIndex++;
23         }
24
25         // scan leftward to find an element smaller than the pivot
26         while ( (rightIndex >= leftIndex) && (c.compare(S[rightIndex], pivot)>=0)) {
27             rightIndex--;
28         }
29
30         // swap
31         if (leftIndex < rightIndex) { // both elements were found
32             temp = S[rightIndex];
33             S[rightIndex] = S[leftIndex]; // swap these elements
34             S[leftIndex] = temp;
35         }
36     } // the loop continues until the indices cross
37
38     temp = S[rightBound]; // swap pivot with the element at leftIndex
39     S[rightBound] = S[leftIndex];
40     S[leftIndex] = temp; // the pivot is now at leftIndex, so recurse
41
42     // step over equal elements to speed up
43     while ((leftIndex > leftBound) && (comp.compare(sequence[leftIndex],
44             sequence[leftIndex - 1]) == 0)) {
45         leftIndex--;
46     }
47     while ((rightIndex > 0) && (rightIndex < rightBound) &&
48             (comp.compare(sequence[rightIndex], sequence[rightIndex + 1]) == 0)) {
49         rightIndex++;
50     }
51
52     // recursive call
53     quickSortStep(S, c, leftBound, leftIndex-1);
54     quickSortStep(S, c, leftIndex+1, rightBound);
55 }
```

---

## 12. Bucket Sort

- Bucket Sort funktioniert nur mit positiven Ganzzahlen in einem Bereich
- Bucket Sort ist stabil
- S: Sequenz von n Items
- N: Maximaler Key
- n: Anzahl Items (Key, Value), mit Key im Bereich von [0, N-1]
- Der Bucket Sort benutzt die Keys als Index in einem Hilfs-Array B von Sequenzen
- Der Algorithmus ist in **drei Phasen eingeteilt:**
  1. Partitionierung: Verteilung der Elemente auf die Buckets B[k]
  2. Jeder Bucket wird mit einem weiteren Sortierverfahren wie beispielsweise Insertionsort sortiert
  3. Der Inhalt der sortierten Buckets wird konkateniert

```
Algorithm bucketSort(S, N)
Input sequence S of (key, element)
      items with keys in the range
      [0, N - 1]
Output sequence S sorted by
      increasing keys
B ← array of N empty sequences
while →S.isEmpty()
    f ← S.first()
    (k, o) ← S.remove(f)
    B[k].insertLast((k, o))
for i ← 0 to N - 1
    while →B[i].isEmpty()
        f ← B[i].first()
        (k, o) ← B[i].remove(f)
        S.insertLast((k, o))
```

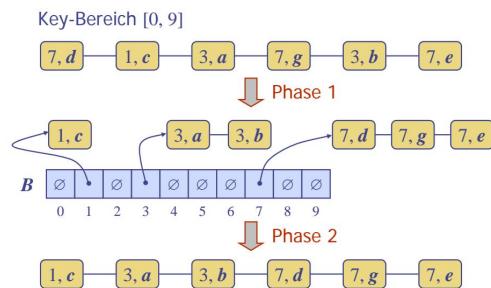


Abbildung 29: Bucket Sort Algorithmus

Abbildung 30: Bucket Sort

## 12.1. Laufzeiten

Big Oh	Bemerkung
$\mathcal{O}(n + N)$	Nur für positive Ganzzahlen

Tabelle 14: Big Oh Bucket Sort

## 12.2. Implementierung

---

```

1  public static void sort(int[] a, int maxVal) {
2      // fail fast
3      if (maxVal <= 0) {
4          throw new IllegalArgumentException();
5      }
6      if (a.length <= 1) {
7          return a; //trivially sorted
8      }
9
10     int[] bucket= new int[maxVal + 1];
11
12     for (int i=0; i<bucket.length; i++) {
13         bucket[i]=0;
14     }
15
16     // Phase 1:
17     for (int i=0; i<a.length; i++) {
18         bucket[a[i]]++;
19     }
20
21     // Phase 2:
22     int outPos=0;
23     for (int i=0; i < bucket.length; i++) {
24         for (int j=0; j<bucket[i]; j++) {
25             a[outPos++]=i;
26         }
27     }
28 }
```

---

## 13. Radix Sort

- Der Radix Sort ist eine Spezialisierung der lexikographischen Sortierung, welcher Bucket Sort als Sortieralgorithmus verwendet
- Er ist ebenfalls kein vergleichsbasierter Sortieralgorithmus wie z.B QuickSort, etc.
- Ist **stabil**
- Vorausgesetzt, dass die maximale Länge der zu sortierenden Schlüssel im vornherein bekannt sind, läuft Radix Sort mit linearer Laufzeit.
- Er besteht aus zwei Phasen:
  1. **Partitionierungsphase:** In dieser Phase werden die Daten in die vorhandenen Fächer aufgeteilt, wobei für jedes Zeichen des zugrundeliegenden Alphabets ein Fach zur Verfügung steht
  2. **Sammelphase:** Nach der Aufteilung der Daten in Fächer in Phase 1 werden die Daten wieder eingesammelt und auf einen Stapel gelegt

### 13.1. Laufzeiten

Big Oh	Bemerkung
$\mathcal{O}(d \cdot (n + N))$	d = Anzahl Tupel, N = Key Bereich Max

Tabelle 15: Big Oh Bucket Sort

### 13.2. Radix Sort Algorithmen

<b>Algorithm</b> <i>radixSort(S, N)</i> <p><b>Input</b> sequence <i>S</i> of <i>d</i>-tuples such that <math>(0, \dots, 0) \leq (x_1, \dots, x_d)</math> and <math>(x_1, \dots, x_d) \leq (N - 1, \dots, N - 1)</math> for each tuple <math>(x_1, \dots, x_d)</math> in <i>S</i></p> <p><b>Output</b> sequence <i>S</i> sorted in lexicographic order</p> <p><b>for</b> <math>i \leftarrow d</math> <b>downto</b> 1     <i>bucketSort(S, Ni)</i></p>	<b>Algorithm</b> <i>binaryRadixSort(S)</i> <p><b>Input</b> sequence <i>S</i> of <i>b</i>-bit integers</p> <p><b>Output</b> sequence <i>S</i> sorted replace each element <i>x</i> of <i>S</i> with the item <math>(0, x)</math></p> <p><b>for</b> <math>i \leftarrow 0</math> <b>to</b> <i>b</i> - 1     replace the key <i>k</i> of     each item <math>(k, x)</math> of <i>S</i>     with bit <math>x_i</math> of <i>x</i></p> <p><b>bucketSort(S, 2)</b></p>
--	---

Abbildung 31: Radix Sort Algorithmus

Abbildung 32: Radix Sort Binär

### 13.3. Beispiel

Die Sequenz 124, 523, 483, 128, 923, 584 soll sortiert werden.

```
1 // 1. partition: order by last digit
2 |0| |1| |2| |3| |4| |5| |6| |7| |8| |9|
3 |           | |           | |           |
4 |      523 124           128
5 |      483 584
6 |      923
7 // 2. collect: 523, 483, 923, 124, 584, 128
8 // 3. partition: order by second digit
9 |0| |1| |2| |3| |4| |5| |6| |7| |8| |9|
10 |           | |           |
11 |      523           483
12 |      923           584
13 |      124
14 |      128
15
16 // 4. collect: 523, 923, 124, 128, 483, 584
17 // 5. partition: order by first digit
18 |0| |1| |2| |3| |4| |5| |6| |7| |8| |9|
19 |           | |           | |           |
20 |      124           483 523           923
21 |      128           584
22
23 // 6. collect:
24 124, 128, 483, 523, 584, 923
```

---

### 13.4. Implementierung

---

```

1  public class RadixSort {
2      // array of linked lists
3      private final LinkedList<String>[] buckets;
4
5      public RadixSort() {
6          // create LinkedList for buckets
7          buckets = (LinkedList<String>[]) new LinkedList<?>[1 + ('z' - 'a' + 1)];
8          IntStream.range(0, buckets.length).forEach(
9              i -> buckets[i] = new LinkedList<String>()
10         );
11     }
12
13     public void radixSort(String[] data) {
14
15         // find max index
16         AtomicInteger maxLength = new AtomicInteger(-1);
17         Arrays.stream(data).forEach(str -> {
18             if (str.length() > maxLength.intValue()) {
19                 maxLength.set(str.length());
20             }
21         });
22
23         // bucket sort from max index to first index
24         for (int i = maxLength.get() - 1; i >= 0; i--) {
25             bucketSort(data, i);
26         }
27     }
28
29     protected void bucketSort(String[] data, int index) {
30
31         // clear buckets
32         Arrays.stream(buckets).forEach(list -> list.clear());
33
34         // insert data elements to buckets
35         Arrays.stream(data).forEach(str -> {
36             if (str.length() <= index) {
37                 buckets[0].addLast(str);
38             } else {
39                 buckets[str.charAt(index) - 'a' + 1].addLast(str);
40             }
41         });
42
43         // shift bucket elements back into data array
44         AtomicInteger i = new AtomicInteger(0);
45         Arrays.stream(buckets).forEach(list -> list.forEach(str -> {
46             data[i.getAndIncrement()] = str;
47         }));
48     }
49 }

```

---

## 14. Pattern Matching

**T** Der Text in dem gesucht werden soll

**P** Ein String (Pattern)

**m** Die Länge des Strings P

**s** Länge des Alphabets  $\Sigma$

**i** Index im Text

**j** Index im Pattern

**Präfix** Substring vom Typ  $P[0 .. i]$

**Suffix** Substring vom Typ  $P[i .. (m-1)]$

### 14.1. Laufzeiten

Algorithmus	Big Oh	Bemerkung
Brute Force Algorithmus	$\mathcal{O}(n \cdot m)$	Schlecht wenn Pattern am Ende unterschiedlich
Boyer-Moore Algorithmus	$\mathcal{O}(n \cdot m + s)$	Ist der schnellste (obwohl schlechterer Worst Case: Sogar schlechter wie Brute Force)
Knuth-Morris-Pratt Algorithmus	$\mathcal{O}(n + m)$ $\mathcal{O}(n + m)$ (Fehlfunktion)	

Tabelle 16: Big Pattern Matching Boyer-Moore und KMP

### 14.2. Brute Force Algorithmus

Der Brute Force Algorithmus vergleicht das Pattern P mit dem Text T für jede mögliche Position.

**Algorithm 3:** BruteForceMatch(T, P)

**Data:** Text T der Länge n und Pattern P der Länge m

**Result:** Startindex eines Substrings von T, welcher mit P übereinstimmt, oder -1 falls  
kein solcher Substring existiert. i aktueller Index im Text. j aktueller Index im  
Pattern

```

1 for i ← 0 to n - m do
2   | j ← 0
3   | while j < m ∧ T[i + j] = P[j] do
4   |   | j ← j + 1
5   | end
6   | if j = m then
7   |   | return i
8   | end
9 end
10 return -1

```

### 14.3. Boyer-Moore Algorithmus

- Benötigt  $\mathcal{O}(n \cdot m + s)$
- Der Algorithmus startet mit den Vergleichen am **Ende des Pattern**
- Verwendet **Character Jump Heuristik**: Falls keine Übereinstimmung → Richte das Pattern gemäss dem letzten Mismatch Zeichen im Pattern aus

#### 14.3.1. Last Occurrence Funktion

Man erstellt eine Hashmap für alle Zeichen des Alphabets  $\Sigma$  mit dem letzten Auftreten des Zeichens  $c$  im Pattern. Das neue  $i$  berechnet sich wie folgt:

$$i = i + m - \min(j, (last(c) + 1))$$

wobei

$i$  = Index des Mismatch Character in Text T

$m$  = Länge des Patterns

$c$  = Alle möglichen Zeichen des Text Alphabets

$L(c)$  = Letztes Auftreten des Zeichens  $c$  im Pattern (Start bei Index 0) (-1, falls nicht vorhanden)

$\Sigma = \{a, b, c, d\}$	<table border="1"> <tr> <td><math>c</math></td><td><math>a</math></td><td><math>b</math></td><td><math>c</math></td><td><math>d</math></td></tr> <tr> <td><math>L(c)</math></td><td>4</td><td>5</td><td>3</td><td>-1</td></tr> </table>	$c$	$a$	$b$	$c$	$d$	$L(c)$	4	5	3	-1
$c$	$a$	$b$	$c$	$d$							
$L(c)$	4	5	3	-1							
$P = abacab$											
Pos: 012345											

Abbildung 33: Boyer Moore Last Occurrence

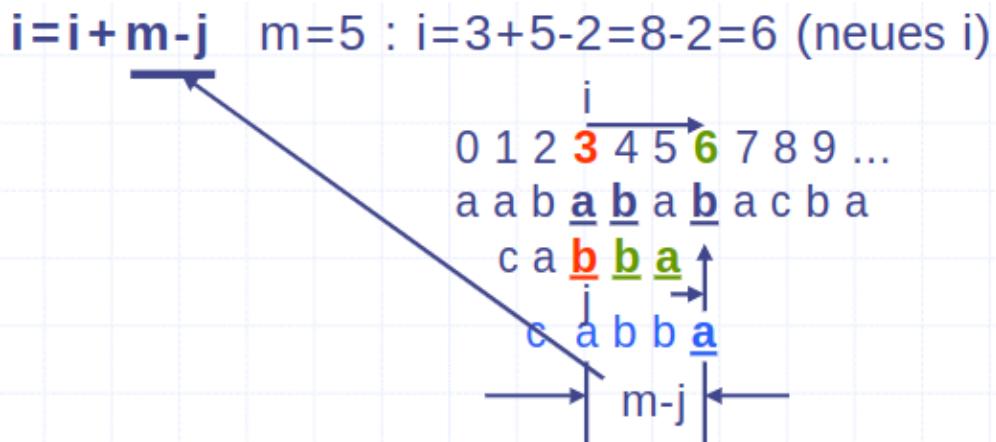


Abbildung 34: Match bereits vorbei

Berechnung Zeichen kommt in Pattern vor:  $i = i + m - (last(T[i]) + 1)$

### 14.3.2. Vorgehen

1. Erstelle die Last Occurrence Funktion
2. Starte am Ende des Patterns und Vergleiche das Pattern mit dem Text (**Rechts nach Links**)
3. Bei einem Mismatch, unterscheide folgende Fälle:
  - a) Wenn das Zeichen c **im Text** auch im Pattern vorkommt, verschiebe das Pattern bis zu dieser Stelle. (Siehe Last Funktion, falls Zeichen mehrmals vorhanden)
  - b) Wenn das Zeichen c **im Text** auch im Pattern vorkommt, aber die Last Funktion einen Index zurückliefert, der der **grösser ist** wie der Index des Mismatches, verschiebe einfach um 1.
  - c) Wenn das Zeichen c im Text nicht im Pattern vorkommt, verschiebe das Pattern hinter mismatched Character.

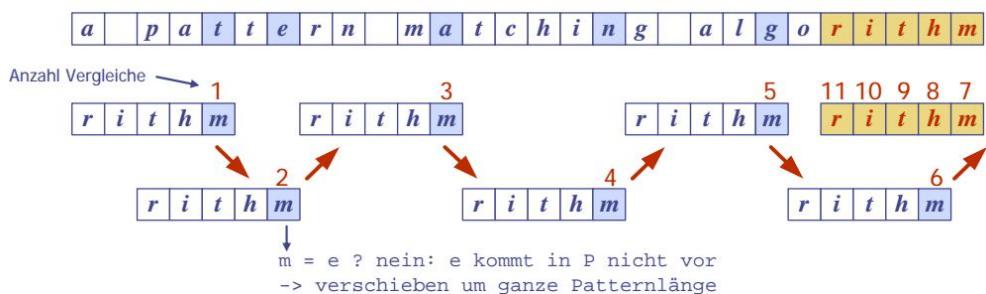
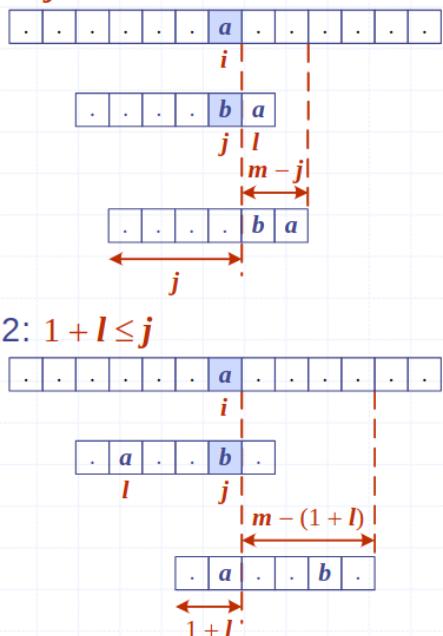


Abbildung 35: Boyer Moore Algorithmus

### 14.3.3. Algorithmus

```
Algorithm BoyerMooreMatch( $T, P, \Sigma$ )
   $L \leftarrow \text{lastOccurrenceFunction}(P, \Sigma)$ 
   $i \leftarrow m - 1$  {actual T-index}
   $j \leftarrow m - 1$  {actual P-index}
  repeat
    if  $T[i] = P[j]$ 
      if  $j = 0$ 
        return  $i$  {match at  $i$ }
      else
         $i \leftarrow i - 1$ 
         $j \leftarrow j - 1$ 
    else
      {character-jump}
       $l \leftarrow L[T[i]]$ 
       $i \leftarrow i + m - \min(j, 1 + l)$ 
       $j \leftarrow m - 1$ 
  until  $i > n - 1$ 
  return -1 {no match}
```

Case 1:  $j \leq 1 + l$



Case 2:  $1 + l \leq j$

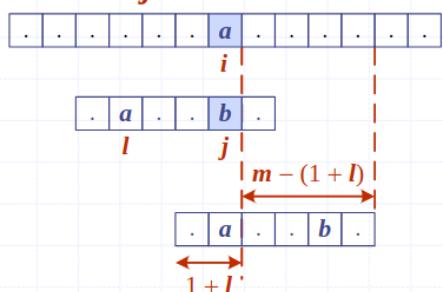
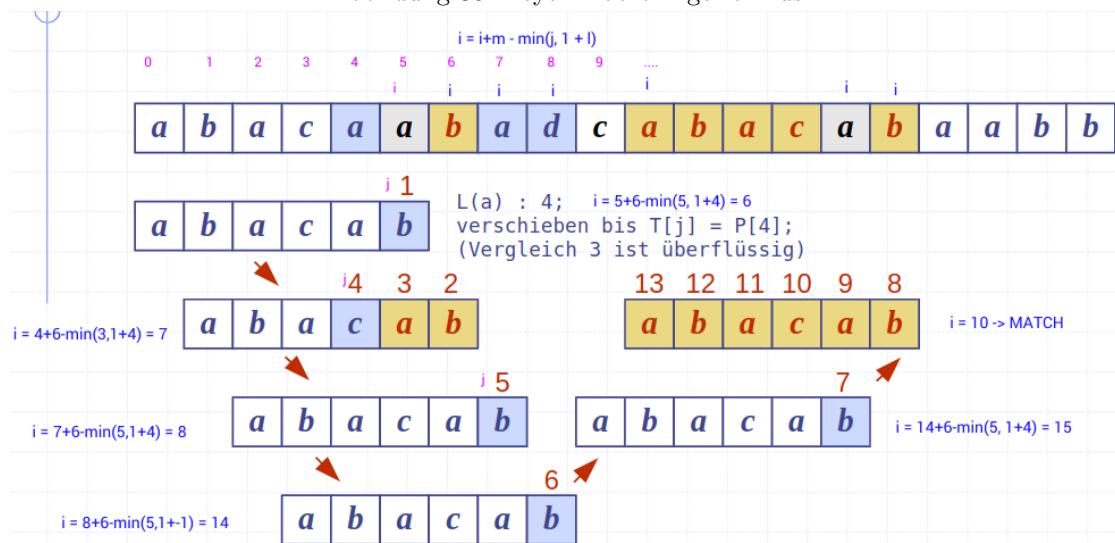


Abbildung 36: Boyer Moore Algorithmus



#### 14.3.4. Implementierung

```
1 public class KnuthMorrisPratt {
2
3     private static int totCount = 0;
4     private static int task = 0;
5
6     public static int kmpMatch(String t, int startIndex, int[] fail, String p) {
7         int count = 0;
8         int n = t.length();
9         int m = p.length();
10        int i = startIndex;
11        int j = 0;
12        while (i < n) {
13            count++;
14            if (task == 2) {
15                System.out.format("i=%d j=%d chars i=%c j=%c\n", i, j, t.charAt(i),
16                               p.charAt(j));
17            }
18            if (t.charAt(i) == p.charAt(j)) {
19                if (j == m - 1) {
20                    System.out.println("Number of comparison: " + count);
21                    totCount += count;
22                    return i - m + 1;
23                } else {
24                    i++;
25                    j++;
26                }
27            } else {
28                if (j > 0) {
29                    j = fail[j - 1];
30                } else {
31                    i++;
32                }
33            }
34        }
35        System.out.println("Number of comparison: " + count);
36        totCount += count;
37        return -1;
38    }
39}
```

---

## 14.4. KMP: Knuth-Morris-Pratt Algorithmus

Durch das Preprocessing beim KMP Algorithmus erreicht man eine Geschwindigkeit die **proportional zur Textlänge** ist.

- Benötigt  $\mathcal{O}(n + m)$
- Der Knut-Morris-Pratt Algorithmus vergleicht das Muster wie der Bruteforce von links nach rechts, aber schiebt das Muster intelligenter als dieser.
- Es wird um so viele Zeichen verschoben, sodass der **Präfix gleichzeitig auch Suffix** ist. Dies wird wie beim Boyer-Moore Algorithmus in einer Vorlaufsphase aufgebaut.

### 14.4.1. Fehl-Funktion

Die Fehlfunktion ist definiert als die Grösse des längsten Präfixes, sodass dieser auch Suffix des Patterns ist. Man betrachtet dabei immer einen Substring. Es sind auch Überlappungen (siehe Beispiel Index 6) möglich. Die Fehlfunktion läuft mit  $\mathcal{O}(m)$

1. Füge das Pattern in der zweiten Reihe der Tabelle ein
2. Betrachte das Pattern Schritt für Schritt, wobei man immer mehr Zeichen anschaut. (bis zur maximalen Länge). Der Präfix startet immer ganz Links und der Suffix ended immer ganz rechts! Leserichtung ist immer von links nach rechts.
3. Suche die maximale Länge für ein Pattern, das zugleich Präfix und Suffix ist. Es können auch Überschneidungen auftreten.

<i>a</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>a</i>
<i>P[j]</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>b</i>
<i>F(j)</i>	0	0	1	1	2	3	4	5

Ränder-Länge: 0 1 2 3 4 5 6 max.Länge

$P[0]=a$  {} 0  
 $P[1]=ab$  {} 0  
 $P[2]=\underline{ab}$  {} a 1  
 $P[3]=\underline{aba}$  {} a 1  
 $P[4]=\underline{baab}$  {} ab 2  
 $P[5]=\underline{baaba}$  {} a ba 3  
 $P[6]=\underline{baabaa}$  {} a baa 4  
 $P[7]=\underline{baabaa}$  {} ab baab 5  
 $P[8]=\underline{baabaaa}$  {} a aba baabaa 6

Abbildung 38: 1. Fehlfunktion aufbauen

### 14.4.2. Vorgehen

1. Gehe von **Links nach Rechts**
2. Suche den ersten Mismatch ( $j=5$ )
3. Übergebe den **Index des Zeichens davor** ( $j = (5 - 1) = 4$ ) der Fehlfunktion  $F(4)$
4. Der Rückgabewert der Fehlfunktion (=1) ist dann der Index 1 ( $j = 1 \rightarrow$  zweites b). Somit wird das Pattern an den Index 1 (Index welche  $F(4)$  herausgegeben hat) geschoben.
5. **Bemerkung:** Ist der Rückgabewert der Fehlfunktion = 0, wird der erste Buchstaben des Patterns auf Missmatch Position geschrieben
6. **Missmatch beim ersten Character des Pattern:** verschieben des Patterns um 1

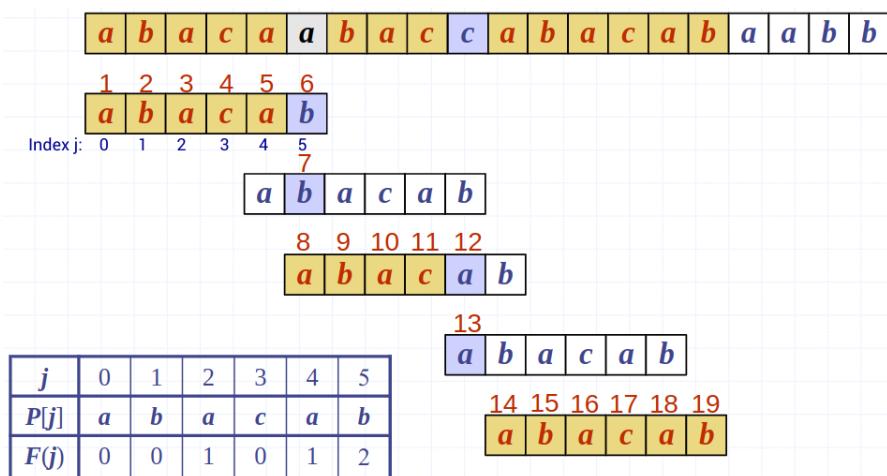


Abbildung 39: 2. Knuth-Morris-Pratt Algorithm

### 14.4.3. Algorithmus

```
Algorithm KMPMatch(T, P)
  F  $\leftarrow$  failureFunction(P)
  i  $\leftarrow$  0
  j  $\leftarrow$  0
  while i < n
    if T[i] = P[j]
      if j = m - 1
        return i - m + 1 { match }
      else
        i  $\leftarrow$  i + 1
        j  $\leftarrow$  j + 1
    else
      if j > 0
        j  $\leftarrow$  F[j - 1]
      else
        i  $\leftarrow$  i + 1
  return -1 { no match }
```

Abbildung 40: KMP Algorithmus

```
Algorithm failureFunction(P)
  F[0]  $\leftarrow$  0
  i  $\leftarrow$  1
  j  $\leftarrow$  0
  while i < m
    if P[i] = P[j]
      {we have matched j + 1 chars}
      F[i]  $\leftarrow$  j + 1
      i  $\leftarrow$  i + 1
      j  $\leftarrow$  j + 1
    else if j > 0 then
      {use failure function to shift P}
      j  $\leftarrow$  F[j - 1]
    else
      F[i]  $\leftarrow$  0 { no match }
      i  $\leftarrow$  i + 1
```

Abbildung 41: KMP Failure Algorithm

#### 14.4.4. Implementierung

Listing 17: Knuth-Morris-Pratt Algorithmus

---

```

1 public static int KMPmatch(String text, String pattern) {
2     int n = text.length();
3     int m = pattern.length();
4     int[] fail = computeFailFunction(pattern);
5     printFail(fail);
6     int i = 0;
7     int j = 0;
8     while (i < n) {
9         System.out.print("\ni = " + i + " j = " + j);
10        if (pattern.charAt(j) == text.charAt(i)) {
11            if (j == m - 1) {
12                return i - m + 1; // match
13            }
14            i++;
15            j++;
16        } else if (j > 0) {
17            /*Verschiebe Pattern an Index j, welches die Failfunction zurckgegeben hat*/
18            j = fail[j - 1];
19        } else {
20            i++;
21        }
22    }
23    return -1; // no match
24 }
```

---

Listing 18: Knuth-Morris-Pratt Algorithmus Fehlfunktion

---

```

1 public static int[] computeFailFunction(String pattern) {
2     int[] fail = new int[pattern.length()];
3     fail[0] = 0;
4     int m = pattern.length();
5     int j = 0;
6     int i = 1;
7     while (i < m) {
8         if (pattern.charAt(j) == pattern.charAt(i)) {
9             // j + 1 characters match
10            fail[i] = j + 1;
11            i++;
12            j++;
13        } else if (j > 0) {
14            // j follows a matching prefix
15            j = fail[j - 1];
16        } else {
17            // no match
18            fail[i] = 0;
19            i++;
20        }
21    }
22    return fail;
23 }
```

---

## 15. Tries

- Mit der Trie Datenstruktur ist ein Pattern Matching möglich, das **proportional zur Grösse des Pattern** ist. (im Vergleich zum KNP, der proportional zum Text läuft)
- Bei Tries gibt es ein Preprocessing des Textes so, dass die Wörter im Baum und die Position als Leaf gespeichert sind.
- **Gross/Kleinschreibung** muss beachtet werden, wobei grosse Buchstaben vor kleinen aufgelistet wird.

### 15.1. Standard Trie

- Der Standard Trie ist ein geordneter Baum für eine Menge von Strings ( $S$ ), so dass:
  - jeder äussere Knoten ausser der Root hat die Kinder alphabetisch geordnet
  - die Pfade von der Root zu den externen Knoten beinhalten die Wörter/Strings

#### 15.1.1. Vorgehen

1. Root zeichnen (leerer Knoten)
2. Root Childs für alle Anfangsbuchstaben erstellen und diese alphabetisch ordnen  
(ACHTUNG: **Gross/Kleinschreibung beachten**: GROSS vor klein)
3. Vorheriger Schritt wiederholen, bis alle Zeichen im Trie abgelegt sind.
4. Jeder Blattknoten speichert die Positionen des assoziierten Wortes.

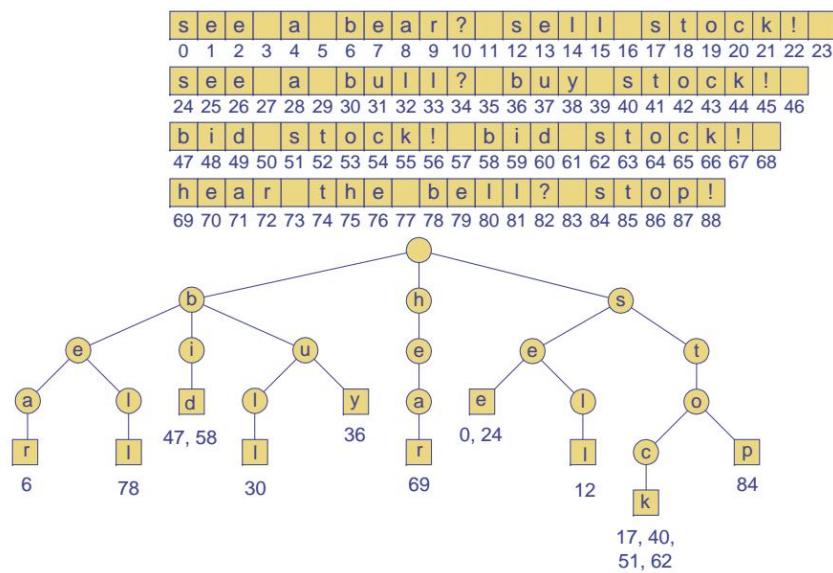


Abbildung 42: Trie Beispiel

## 15.2. Komprimierter Trie

- Beim komprimierten Trie haben alle **internen Knoten mindestens 2 Kinder** und nach Möglichkeit mehrere Buchstaben pro Node.
- Die kompakte Representation eines komprimierten Tries ist ein Array aller Strings
  - Jeder Knoten speichert dann nur noch die Indizes in dem Array anstatt den Strings
  - [index im array], [start zeichen innerhalb des array item] [end zeichen innerhalb des array item]
  - z.B  $S[3] = \text{"stock"}$  → Knoten = (3,1,2) → "to"

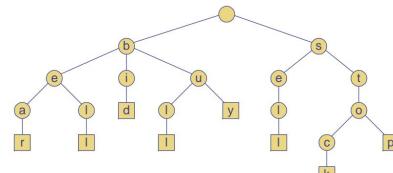


Abbildung 43: Trie Ausgangslage

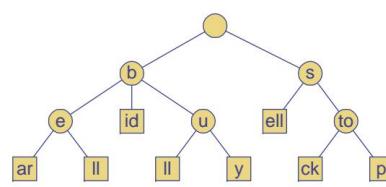


Abbildung 44: Trie nach Kompression

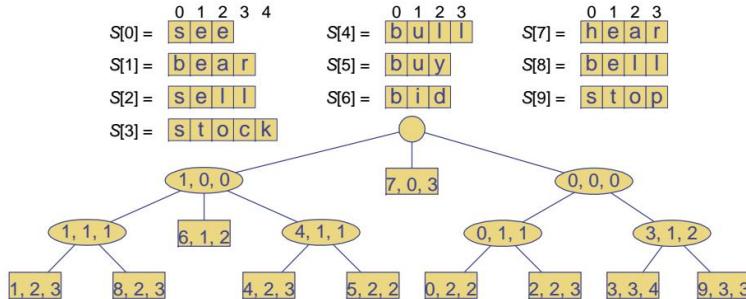


Abbildung 45: Kompakte Repräsentation eines komprimierten Tries

## 15.3. Suffix Trie

- Der Suffix Trie eines Strings ist der komprimierte Trie von allen Suffixen des Strings
- Mit einer Suffix kann alles gefunden werden (nicht nur am Wortanfang)
  - Das komplette Wort
  - Suffix
  - Prefix
  - Substrings

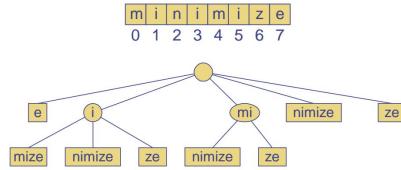


Abbildung 46: Suffix Trie

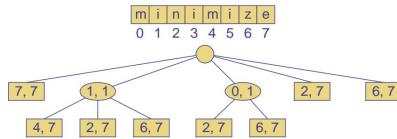


Abbildung 47: Suffix Trie with Index Representation

## 15.4. Laufzeitverhalten / Speicherplatz

- n** totale Länge der Strings in S
- m** Länge des Pattern
- d** Grösse des Alphabets

Beschreibung	Big Oh
Speicherverbrauch	$\mathcal{O}(n)$
Suchen, Einfügen, Löschen	$\mathcal{O}(dm)$
Erstellen des Trie	$\mathcal{O}(n)$

Tabelle 17: Big Oh Tries

## 15.5. Implementierung

```

1  public class TrieMultimap<V> implements Multimap<String, V> {
2
3      private TrieNode<V> root;
4
5      private enum Mutation {
6          INSERT, REMOVE
7      };
8
9      public TrieMultimap() {
10         this.root = new TrieNode<V>();
11     }
12
13     // Returns the first value for a given key. null if key is not found
14     public V find(String key) {
15         TrieNode<V> result = find(root, key);
16         if (result != null)
17             return result.getValues().get(0);
18         else
19             return null;
20     }
21
22     // return Iterator over all values. If key is not found: Iterator without next
23     public Iterator<V> findAll(String key) {
24         TrieNode<V> result = find(root, key);
25         if (result != null)
26             return result.getValues().iterator();

```

```

27     else
28         return new LinkedList<V>().iterator();
29     }
30
31     private TrieNode<V> find(TrieNode<V> node, String keySubstr) {
32         if (keySubstr.length() == 0) {
33             return node;
34         }
35         for (TrieNode<V> child : node.getChilds()) {
36             if (keySubstr.startsWith(child.getKeySubstr())) {
37                 keySubstr = keySubstr.substring(child.getKeySubstr().length());
38                 return find(child, keySubstr);
39             }
40         }
41         return null;
42     }
43
44     // Inserts a key/value pair into the multimap.
45     public void insert(String key, V value) {
46         TrieNode<V> result = find(root, key);
47         if (result != null) {
48             result.getValues().add(value);
49         } else {
50             mutate(Mutation.INSERT, root, key, 0, value);
51         }
52     }
53
54     private boolean mutate(Mutation operation, TrieNode<V> node, String key, int
55         keyIndex, V value) {
56         if (key.length() == keyIndex) {
57             // found the node!
58             if (operation == Mutation.INSERT) {
59                 node.getValues().add(value);
60             } else { // REMOVE
61                 node.getValues().clear();
62             }
63             return true;
64         }
65         for (TrieNode<V> child : node.getChilds()) {
66             if (child.getKeySubstr().charAt(0) == key.charAt(keyIndex)) {
67                 if (child.getKeySubstr().length() > 1) { // a compressed node?
68                     child = decompress(node, child);
69                 }
70                 boolean result = mutate(operation, child, key, ++keyIndex, value);
71                 compress(node, child);
72                 return result;
73             }
74         }
75         // there is no corresponding child:
76         if (operation == Mutation.INSERT) {
77             TrieNode<V> newNode = new TrieNode<V>();
78             newNode.setKeySubstr(key.substring(keyIndex, keyIndex + 1));
79             node.getChilds().add(newNode);
80             mutate(Mutation.INSERT, newNode, key, ++keyIndex, value);
81             compress(node, newNode);
82         } else { // REMOVE
83             return false; // not found
84         }
85         return false;
86     }
87

```

```

88     private TrieNode<V> decompress(TrieNode<V> node, TrieNode<V> child) {
89         // insert an additional, single-char node (de-compressing):
90         TrieNode<V> newChild = new TrieNode<>();
91         newChild.setKeySubstr(child.getKeySubstr().substring(0, 1));
92         child.setKeySubstr(child.getKeySubstr().substring(1));
93         newChild.getChilds().add(child);
94         node.getChilds().add(newChild);
95         node.getChilds().remove(child);
96         return newChild;
97     }
98
99     private void compress(TrieNode<V> node, TrieNode<V> child) {
100        if ((child != root) && (child.getChilds().size() == 1)
101        && (child.getValues().isEmpty())) {
102            // compress:
103            TrieNode<V> childOfChild = child.getChilds().get(0);
104            child.setKeySubstr(child.getKeySubstr().concat(childOfChild.getKeySubstr()));
105            child.getValues().addAll(childOfChild.getValues());
106            child.getChilds().addAll(childOfChild.getChilds());
107            child.getChilds().remove(childOfChild);
108            return;
109        }
110        if (child.getChilds().isEmpty() && (child.getValues().isEmpty())) {
111            // this is a removed node:
112            node.getChilds().remove(child);
113            return;
114        }
115    }
116
117    // Removes all values for a given key.
118    public void remove(String key) {
119        TrieNode<V> result = find(root, key);
120        if (result != null) {
121            mutate(Mutation.REMOVE, root, key, 0, null);
122        } else {
123            return;
124        }
125    }
126
127    public int size() {
128        return size(root);
129    }
130
131    // return Number of values in this node and its child nodes.
132    private int size(TrieNode<V> element) {
133        int size = 0;
134        for (TrieNode<V> child : element.getChilds()) {
135            size += size(child);
136        }
137        size += element.getValues().size();
138        return size;
139    }
140}

```

---

## 16. Dynamische Programmierung

Bei der dynamischen Programmierung geht es darum, auf die Lösungen von Subproblemen (die während dem Lösen entstehen) aufzubauen, um das ganze Problem zu lösen. Dynamische Programmierung kann dann erfolgreich eingesetzt werden, wenn das Optimierungsproblem aus vielen gleichartigen Teilproblemen besteht und eine optimale Lösung des Problems sich aus optimalen Lösungen der Teilprobleme zusammensetzt

### 16.1. Rucksack Problem

1. Das Rucksackproblem (NP Vollständig), lässt sich nur so schnell lösen, weil wir ganze Zahlen haben
2. Annahme: die Objekte sind genau einmal vorhanden. Der Rucksack bietet Platz für 8kg.
3. Man geht spaltenweise **von links nach rechts** und prüft welcher maximale Wert für ein Gewicht möglich ist. Achtung: Es können auch mehrere Werte zusammengezählt werden. Das Gewicht auf der X-Achse darf aber nie überschritten werden.
  - Bsp. Gewicht = 6
  - $6 = 7/3 + 4/2 + 3/1 \rightarrow$  maximal 14
4. In einem ersten Schritt werden die grösstmöglichen Werte in die Tabelle abgefüllt.
5. Solange kein grösserer Wert gefunden wird, wird der **maximale Wert pro Spalte beibehalten**. In der nächsten Spalte wird aber wieder von vorne begonnen.
6. Ist die komplette Tabelle ausgefüllt, steht der maximal mögliche Wert ganz unten rechts.
7. Wenn da nächste Subproblem keine Verbesserung bringt, geht man wieder zurück zum Optimum des letzten Subproblems und übernimmt diesen Wert. Dabei geht man so vor, dass man in einer **Spalte von unten nach oben geht, bis sich der Wert ändert**. Dieser Wert wird ebenfalls in den Rucksack gelegt.
8. Das nächste Subproblem wird wie folgt bestimmt:
  - Ausgehend vom gerade hinzugefügten Objekt, wird das **hinzugefügtes Gewicht vom Gewicht auf der X-Achse abgezogen**. Man geht in der Tabelle also um x Spalten nach links.  
(z.B. Gewicht = 7  $\Rightarrow$  8/4 konnte hinzugefügt werden  $\rightarrow$  Weiter bei Gewicht = 3)
  - Solange sich das Gewicht in einer Spalte nicht ändert, wird innerhalb der Spalte nach oben verschoben. (Der Wert/kg hat ja keine Verbesserung gebracht)

### 16.2. Voraussetzung Subprobleme

1. **Einfache Subprobleme:** Subprobleme können durch wenige Variablen ausgedrückt werden.
2. **Subproblem-Optimierung:** Das globale Optimum kann durch optimale Subprobleme ausgedrückt werden.
3. **Subprobleme überlappen:** Die Subprobleme sind nicht unabhängig, sie überlappen (sollte somit bottom-up konstruiert werden).

kg Wert / kg \	1	2	3	4	5	6	7	8
7 / 3	0	0	7	7	7	7	7	7
4 / 2	0	4	7	7	11	11	11	11
8 / 4	0	4	7	8	11	12	15	15
9 / 5	0	4	7	8	11	12	15	16
3 / 1	3	4	7	10	11	14	15	18

Abbildung 48: Dynamische Programmierung, Rucksackproblem

### 16.3. Subsequenzen

1. Beispiel: ABCDEFGHIJK

Subsequenz: ACEGIJK

Subsequenz: DFGHK

Nicht Subsequenz: DAGH  $\rightarrow$  D darf nicht vor A kommen

### 16.4. LCS: Longest Common Subsequence

- Finde die längste Subsequenz die in zwei Sequenzen enthalten ist, wobei die Subsequenz nicht an einem Stück sein muss (eher **Submenge**, kein Substring!). Die Reihenfolge der auftreten Zeichen muss aber der Reihenfolge im Text entsprechen.
- Der BruteForce Algorithmus läuft exponentiell mit  $\mathcal{O}(2^n)$
- Beim Ansatz mit dynamischer Programmierung hat man eine Laufzeit von  $\mathcal{O}(n \cdot m)$
- LCS mit dynamischer Programmierung wird z.B bei Versionsverwaltungstools verwendet
- Es gibt eine zusätzliche Reihe/Spalte mit dem Index -1, damit es möglich ist, KEINE Übereinstimmung abzubilden.
- **Beispiel LCS:** ABCDEFG und XZACKDFWGH haben ACDFG als längste gemeinsame Subsequenz

## 16.5. Vorgehen

### Tabelle Aufbauen

1. -1 Zeile und Spalte Zeichen und Felder mit 0 initialisieren
2. Tabelle aufbauen: Für alle Felder zeilenweise von **links nach rechts und oben nach unten**
  - **Match:** Wenn die beiden Zeichen gleich sind: Zähle 1 zum Wert oben links (diagonal) hinzu
  - **No Match:** Nimm ansonsten den maximalen Wert zwischen dem Wert links der aktuellen Position, oder oben von der aktuellen Position.

### Lösungen finden

Die LCS wird von rechts nach links aufgebaut. Es kann verschiedene Lösungen geben, da man von nach jedem Match entweder der Reihe oder Kollonne folgen kann.

1. Beginne unten rechts und prüfe ob die Zeichen gleich sind.
2. Wenn die Zeichen gleich sind, nimm das Zeichen in die LCS (von rechts nach links) und verschiebe diagonal nach **links/oben**.
3. Wenn die Zeichen nicht gleich sind folge der **Kolonne oder Reihe**, bis die beiden Zeichen wieder gleich sind oder sich die Zahlen ändern.
4. Wiederhole diese Schritte bis man ganz oben/lefts angekommen ist.

		C	G	A	T	A	A	T	T	G	A	G	A		
		L	-1	0	1	2	3	4	5	6	7	8	9	10	11
L		-1	0	0	0	0	0	0	0	0	0	0	0	0	0
G		0	0	0	1	1	1	1	1	1	1	1	1	1	1
T		1	0	0	1	1	2	2	2	2	2	2	2	2	2
T		2	0	0	1	1	2	2	2	3	3	3	3	3	3
C		3	0	1	1	1	2	2	2	3	3	3	3	3	3
C		4	0	1*	1	1	2	2	2	3	3	3	3	3	3
T		5	0	1	1	1	2*	2	2	3	4	4	4	4	4
A		6	0	1	1	2	2	3*	3	3	4	4	5	5	5
A		7	0	1	1	2	2	3	4*	4	4	4	5	5	6
T		8	0	1	1	2	3	3	4	5	5*	5	5	5	6
A		9	0	1	1	2	3	4	4	5	5	5	6	6	6*

$Y = CGATAATTGAGA$

$X = GTTCCTAATA$

$0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11$

$\star \ X_i = Y_j$

Abbildung 49: Longest Common Subsequence

### 16.5.1. Implementierung

---

```

1  public class LCS {
2      private int tableL[][]; // data array
3      private String xStr;
4      private String yStr;
5
6      public int[][] calculateTable(final String xStr, final String yStr) {
7
8          this.xStr = xStr;
9          this.yStr = yStr;
10
11         int n = xStr.length();
12         int m = yStr.length();
13
14         // +1 because of zero row/column
15         tableL = new int[n + 1][m + 1];
16
17         for (int i = 1; i <= n; i++) {
18             for (int j = 1; j <= m; j++) {
19                 // - 1 because of the zero row/column
20                 if (xStr.charAt(i - 1) == yStr.charAt(j - 1)) {
21                     tableL[i][j] = tableL[i - 1][j - 1] + 1;
22                 } else {
23                     tableL[i][j] = Math.max(tableL[i - 1][j], tableL[i][j-1]);
24                 }
25             }
26         }
27
28         return tableL;
29     }
30
31     public List<String> findAll() {
32         List<String> list = new LinkedList<>();
33         Deque<Character> stack = new LinkedList<>();
34         find(xStr.length(), yStr.length(), stack, list);
35         List<String> result = new LinkedList<>();
36         list.stream().sorted().distinct().forEach(str -> result.add(str));
37         return result;
38     }
39
40     private void find(final int xPos, final int yPos, Deque<Character> stack,
41                       List<String> stringList) {
42         if ((xPos == 0) || (yPos == 0)) { // reached the end?
43             stringList.add(stack.toString());
44             return;
45         } else {
46             if (xStr.charAt(xPos - 1) == yStr.charAt(yPos - 1)) {
47                 stack.push(xStr.charAt(xPos - 1));
48                 find(xPos - 1, yPos - 1, stack, stringList);
49                 stack.pop();
50             }
51             if (tableL[xPos - 1][yPos] == tableL[xPos][yPos]) {
52                 find(xPos - 1, yPos, stack, stringList);
53             }
54             if (tableL[xPos][yPos - 1] == tableL[xPos][yPos]) {
55                 find(xPos, yPos - 1, stack, stringList);
56             }
57         }
58     }
}

```

---

## 17. Graphen

### 17.1. Terminologie

**G: Graph** Ein Paar  $(V, E)$ . Besteht aus einem Set von Knoten und einer Collection von Kanten.

**V: Vertizes** Ein Knoten

**E: Edges** Eine Kante, enthält ein Paar von Vertizes

**n** Anzahl Vertizes

**m** Anzahl Kanten (**min:**  $m = n - 1$  (Liste), **max:**  $m = \frac{n \cdot (n-1)}{2}$  (vollvermascht)). Mit  $n - 1$  Kanten lässt sich der ganze Graph verbinden (Ring)

**gerichtete Kanten** Erster Knoten des Edges ist der Ursprung und der andere das Ziel. Die Kante wird als Pfeil dargestellt

**ungerichtete Kanten** Ein ungeordneter Knoten Paar

**gerichteter Graph** Alle vorhandenen Edges sind gerichtet

**ungerichteter Graph** Alle vorhandenen Edges sind ungerichtet

**End-Vertizes** Endpunkt einer Kante. Eine Kante ist **inzident** (enden) an einem Knoten

**Adjazente** Benachbarte Knoten sind adjazent. U und V sind adjazent.

**Inzident** Kanten enden an einem Vertex. a, d und b sind inzident in V

**Grad eines Vertex** Anzahl inziderter Kanten. Wie viele Kanten mit einem anderen Knoten verbunden sind. X besitzt Grad 5.  $\sum_v \deg(v) = 2m$

**Parallele Kanten** Zwei Kanten zwischen zwei Knoten, die eine Schleife bilden. Ziel und Ursprung sind gleich. h und i sind parallele Kanten

**Schleife** Eine Kante mit dem gleichen Ursprung und Ziel. j ist eine Schleife

**Connected** Ein Graph ist connected, falls zwischen allen Vertizes ein Pfad existiert.

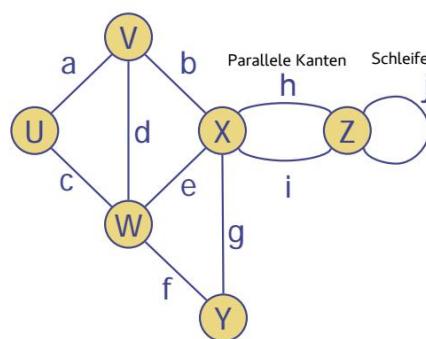


Abbildung 50: Parallel Kanten und Schleifen

### 17.1.1. Subgraphen

**Subgraph** Alle Kanten und Vertizes des Subgraphen sind eine Teilmenge des Graphen

**Aufspannender Subgraph (Spanning)** Ein aufspannender Subgraph enthält alle Vertizes des Graphen, jedoch nicht alle Kanten.

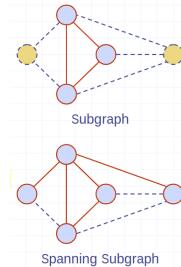


Abbildung 51: Subgraphen

### 17.1.2. Tree und Forest

**Tree** Ist ein Graph der connected ist und keine Zyklen aufweist

**Forest** Ist ein ungerichteter Graph ohne Zyklen der aus Trees besteht.

**Spanning Tree** Ist ein connected, nicht eindeutiger (Pfade können ändern), loopfreier Tree.

### 17.1.3. Pfad und Zyklen

**Pfad** Beginnt und Endet mit einem Vertex. (Einfacher Pfad = rot, Nicht einfacher Pfad = grün)

**Pfad** Vertexe können mehrmals vorkommen

**Einfacher Pfad** Jeder Vertex kommt nur einmal vor

**Zyklus** Endet mit einer Kante. Ein Zyklus verbindet implizit den letzten Vertex mit dem ersten

**Einfacher Zyklus** besucht nie zweimal den gleichen Vertex. (Einfacher Zyklus = rot, Nicht einfacher Zyklus = grün)

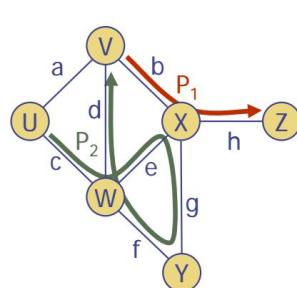


Abbildung 52: Pfad

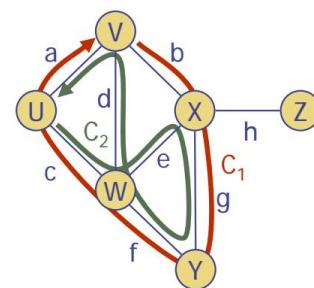


Abbildung 53: Zyklus

## 17.2. Kanten-Listen Struktur

- Grundsätzlich gibt es Objekte für die Vertices und die Edges
- Man hält sich je eine Sequenz für Vertices und eine für Edges
- Jedes Vertex Objekt hält eine Referenz auf die Position in der Vertex Sequenz
- Jedes Edge Objekt hält eine Referenz auf den Ursprungs- und Ziel Vertex, sowie eine Referenz auf seine Position in der Kanten-Struktur.
- Beim Einfügen kann der neue Vertex einfach am Ende der Liste angefügt werden.
- Beim Löschen muss die gesamte Liste von Kanten nach dem gesuchten Vertext durchsucht werden.
- Die Kanten Listen Sturktur **wird selten verwendet**

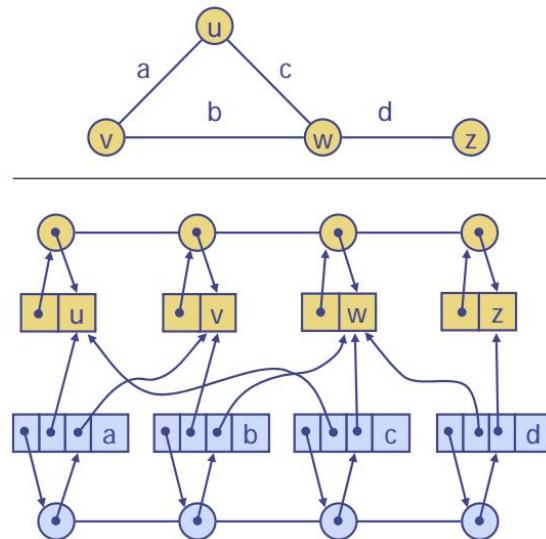


Abbildung 54: Kanten-Listen Struktur

### 17.2.1. Kanten-Listen Struktur Implementierung

---

```

1  public class KantenListenGraph {
2
3      private DoubleLinkedListPosition<Vertex> vSeqHead;
4      private DoubleLinkedListPosition<Edge> eSeqHead;
5
6      public Vertex insertVertex(Object o) {
7          vSeqHead = new DoubleLinkedListPosition<Vertex>(vSeqHead);
8          Vertex vertex = new Vertex(o, vSeqHead);
9          return vertex;
10     }
11
12    public Edge insertEdge(Vertex v, Vertex w, Object o) {
13        Vertex[] endVertices = {v, w};
14        eSeqHead = new DoubleLinkedListPosition<Edge>(eSeqHead);
15        Edge edge = new Edge(o, endVertices, eSeqHead);
16        return edge;
17    }
18    ...
19
20    public class Vertex {
21
22        private Object object;
23        protected DoubleLinkedListPosition<Vertex> position;
24
25        public Vertex(Object o, DoubleLinkedListPosition<Vertex> position) {
26            this.object = o;
27            this.position = position;
28            position.element = this;
29        }
30        ...
31
32        public class Edge {
33            private Object object;
34            private Vertex[] vertices;
35            protected DoubleLinkedListPosition<Edge> position;
36
37            public Edge(Object o, Vertex[] vertices, DoubleLinkedListPosition<Edge> position) {
38                this.object = o;
39                this.vertices = vertices;
40                this.position = position;
41                position.element = this;
42            }
43            ...
44
45        public class DoubleLinkedListPosition<E> {
46            protected DoubleLinkedListPosition<E> previous;
47            protected DoubleLinkedListPosition<E> next;
48            protected E element;
49
50            DoubleLinkedListPosition(DoubleLinkedListPosition<E> next) {
51                this.next = next;
52                if (next != null) {
53                    next.previous = this;
54                }
55            }
56            ...

```

---

### 17.3. Adjazenz-Listen Struktur

- Baut auf der Kanten-Listen Struktur auf, mit der Erweiterung einer Inzidenz-Sequenz für jeden Vertex. Diese enthält die Positionen auf die erweiterten Kantenobjekte der inzidenten Kanten.
- Wird immer verwendet wenn man Mutation in dem Graphen macht
- Jedes Kanten Objekt hält eine Referenz auf den Ursprungs- und Ziel Vertex
- **Benötigt sehr wenig Platz**, auch bei vielen Kanten/grossen Graphen.

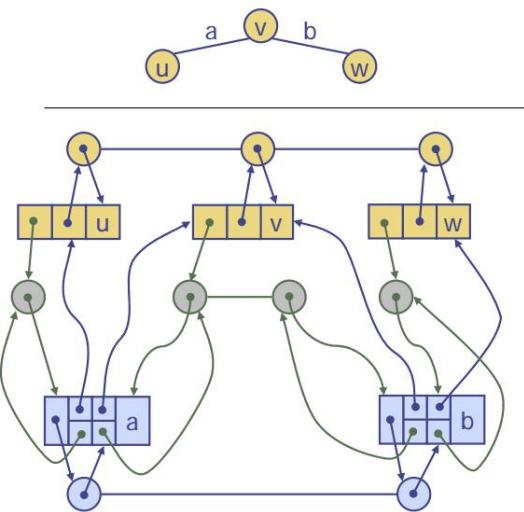


Abbildung 55: Adjazenz Listen Struktur

### 17.4. Adjazenz-Matrix Struktur

- Baut auf der Kanten-Listen Struktur auf, mit der Erweiterung, dass die Vertex Objekte einen Index in die Adjazenz Matrix halten.
- Ist besonders für den lesenden Zugriff geeignet
- Kann **Anfragen zur Nachbarschaft sehr schnell** beantworten, benötigt jedoch mehr Platz

### 17.5. Laufzeiten

- n Vertizes
- m Kanten
- keine parallelen Kanten
- keine Schleifen

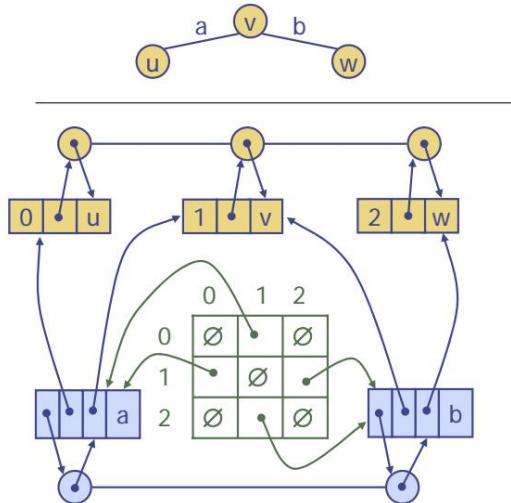


Abbildung 56: Adjazenz-Matrix Struktur

Operation	Kanten Liste	Adjazenz Liste	Adjazenz Matrix
Space	$\mathcal{O}(n + m)$	$\mathcal{O}(n + m)$	$\mathcal{O}(n^2)$
incidentEdges(v)	$\mathcal{O}(m)$	$\mathcal{O}(\deg(v))$	$\mathcal{O}(n)$
areAdjacent(v, w)	$\mathcal{O}(m)$	$\mathcal{O}(\min(\deg(v), \deg(w)))$	$\mathcal{O}(1)$
insertVertex(o)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n^2)$
insertEdge(v, w, o)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
removeVertex(v)	$\mathcal{O}(m)$	$\mathcal{O}(\deg(v))$	$\mathcal{O}(n^2)$
removeEdge(e)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Tabelle 18: Laufzeiten von Graph Operationen

## 17.6. Implementierung

```

1 // Graph
2 public class Graph<T extends Comparable<T>> extends Observable {
3
4     private ArrayList<Node<T>> nodes;
5     private Stack<Node<T>> stack;
6
7     public Graph() {
8         nodes = new ArrayList<Node<T>>();
9         stack = new Stack<Node<T>>();
10    }
11
12    public Node<T> getNode(int indx) {
13        return nodes.get(indx);
14    }
15
16    public void addNode(Node<T> n) {
17        if (nodes.contains(n)) {
18            return; // list is a set !
19        }
20        nodes.add(n);

```

```

21     }
22
23     // DFS
24     public ArrayList<Node<T>> depthFirstSearch(Node<T> from, Node<T> to) {
25         from.setMark(true);
26         stack.push(from);
27         if (from == to) {
28             return new ArrayList<Node<T>>(stack);
29         }
30         for (Node<T> n : from.getConnectedNodes()) {
31             System.out.println(from.getObject() + ": " + n.getObject() + " ");
32             if (!n.isMarked()) {
33                 ArrayList<Node<T>> path = depthFirstSearch(n, to);
34                 if (path != null) {
35                     return path;
36                 }
37             }
38         }
39         stack.pop();
40         return null;
41     }
42
43     // BFS
44     public ArrayList<Node<T>> breadthFirstSearch(Node<T> from, Node<T> to) {
45         ArrayList<IntermediatePath<T>> queue = new ArrayList<>();
46         IntermediatePath<T> ip = new IntermediatePath<>(null, from);
47         queue.add(ip);
48         from.setMark(true);
49         while (queue.size() > 0) {
50             ip = queue.remove(0);
51             if (ip.current == to) {
52                 ArrayList<Node<T>> path = new ArrayList<>();
53                 do {
54                     path.add(0, ip.current);
55                 } while ((ip = ip.previous) != null);
56                 return path;
57             }
58             for (Node<T> it : ip.current.getConnectedNodes()) {
59                 if (!it.isMarked()) {
60                     it.setMark(true);
61                     IntermediatePath<T> newIP = new IntermediatePath<T>(ip, it);
62                     queue.add(newIP);
63                     System.out.print(" previous: " + newIP.previous.current.getObject()
64                     + " current: " + newIP.current.getObject());
65                 }
66             }
67         }
68         return null;
69     }
70 }
```

---

```

1 // Node
2 public class Node<T extends Comparable<T>> {
3
4     // Connected neighbour nodes
5     private ArrayList<Node<T>> linked;
6     private T obj;
7
8     public Node(T obj) {
9         this.obj = obj;
10        linked = new ArrayList<Node<T>>();
```

```
11     }
12
13     public ArrayList<Node<T>> getConnectedNodes() {
14         return linked;
15     }
16
17     public void connectTo(Node<T> n) {
18         ListIterator<Node<T>> it = linked.listIterator();
19         while(it.hasNext()) {
20             Node<T> listNode = it.next();
21             int compareResult = n.getObject().compareTo(listNode.getObject());
22             if (compareResult == 0) {
23                 return; // list is a set!
24             } else if (compareResult < 0) {
25                 it.previous();
26                 it.add(n);
27                 return;
28             }
29         }
30         // Node has not yet been inserted
31         linked.add(n);
32     }
33
34 }
```

---

**Aufgabe 3**

$G$  sei ein einfach verbundener Graph mit  $n$  Knoten und  $m$  Kanten.  
Zeigen Sie, dass  $O(\log(m))$  gleich  $O(\log(n))$  ist.

**Lösung**

$$O(\log(m)) = O(\log(n)) \Leftrightarrow \log(m) \in O(\log(n)) \cap \log(n) \in O(\log(m))$$

Die Anzahl Kanten hängt direkt von der Anzahl Knoten ab. Es kann eine untere  $m$  sowie eine obere Schranke  $\bar{m}$  für  $m$  gefunden werden:

$$\underline{m} = n - 1 \quad (\text{Liste})$$

$$\bar{m} = \frac{n(n-1)}{2} \quad (\text{Voll vermaschter Graph})$$

Somit muss bewiesen werden, dass

$$\log(\bar{m}) \in O(\log(n)) \cap \log(n) \in O(\log(\underline{m}))$$

Beweis von  $\log\left(\frac{n(n-1)}{2}\right) \in O(\log(n))$ :

Konstanten und niederwertige Terme streichen:

$$\log(n^2) \in O(\log(n))$$

$$2\log(n) \in O(\log(n))$$

$$\log(n) \in O(\log(n))$$

Beweis von  $\log(n) \in O(\log(n-1))$ :

Konstanten streichen:

$$\log(n) \in O(\log(n))$$

Abbildung 57: Berechnungsaufgabe

## 18. DFS und BFS

### 18.1. DFS: Depth First Search

- Tiefensuche
- Mit dem DFS Algorithmus werden durch Graphen traversiert
- Die mit **DISCOVERY** markierten und besuchten Kanten bilden einen Spanning Tree
- Der DFS kann verwendet werden um einen **Pfad und Zyklung zu finden**. Ebenfalls kann eine Topologische Sortierung damit erstellt werden.
- In einem gerichteten Graph, geht der DFS Algorithmus so tief wie möglich und macht anschliessend ein Backtracking.
  1. Setzt in einem ersten Schritt alle Edges und Vertizes auf **UNEXPLORED**
  2. Führt rekursiv für alle Vertizes den DFS Algorithmus durch
  3. Die Incident Kanten sind aufsteigend sortiert!
  4. Wenn die Kante nicht bereits besucht wurde, geht man zum gegenüberliegenden Knoten und setzt ihn auf **DISCOVERY**.
  5. Geht man von einem Knoten aus zurück, fängt man den nächsten Iterationsschritt, bei dem Knoten an, der besucht wurde, bevor man zurück ging.
  6. Terminiert der rekursive DFS Algorithmus wird noch einmal jeder Vertex überprüft, ob er **UNEXPLORED** ist. Gibt es immer noch **UNEXPLORED** Edges, kann der Graph nicht connected sein, da der Algorithmus garantiert alle verbundenen Knoten besucht.
  7. Gibt es eine Kante mit dem **BACK** Label, hat man einen Zyklus

---

**Algorithm 4:** DFS( $G$ ) und BFS( $G$ )

---

**Data:** Graph  $G$   
**Result:** Labeling of the edges of  $G$  as discovery edges and back edges

```

1 forall  $u$  in  $G.vertices()$  do
2   | setLabel( $u$ , UNEXPLORED)
3 end
4 forall  $e$  in  $G.edges()$  do
5   | setLabel( $e$ , UNEXPLORED)
6 end
7 forall  $v$  in  $G.vertices()$  do
8   | if getLabel( $v$ ) == UNEXPLORED then
9     |   | DFS( $G$ ,  $v$ )
10    | end
11   | else
12     |   | // not conencted
13   | end
14 end

```

---

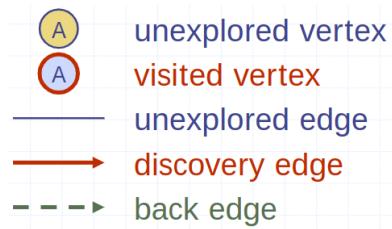


Abbildung 58: DFS Benennung

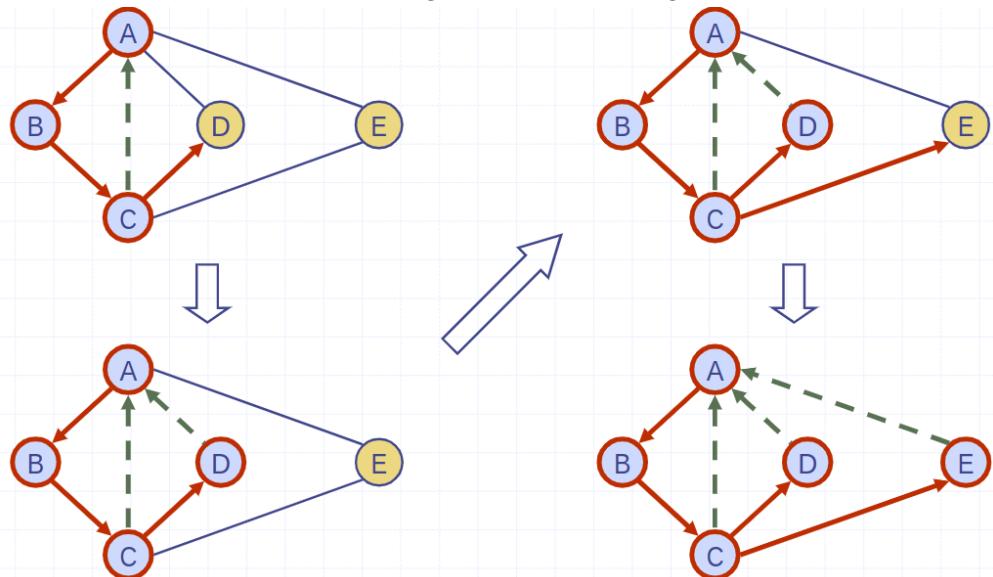


Abbildung 59: DFS Beispiel

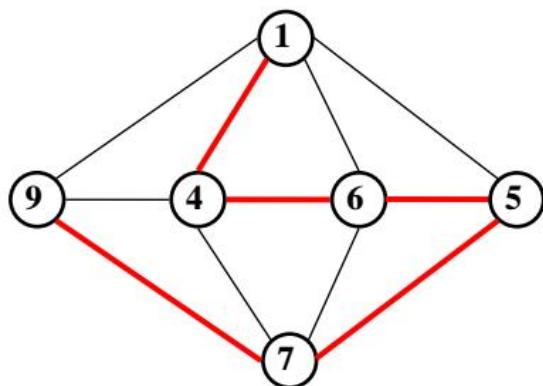


Abbildung 60: Tiefensuche

---

**Algorithm 5:** DFS( $G, v$ )

---

**Data:** Graph  $G$  and a Start vertex  $v$  of  $G$

**Result:** Labeling of the edges of  $G$  in the connected component of  $v$  as discovery edges and back edges

```

1 setLabel( $v, VISITED$ )
2 forall  $e$  in  $G.incidentEdges(v)$  do
3   if getLabel( $e$ ) == UNEXPLORED then
4      $w \leftarrow opposite(v, e)$ 
5     if getLabel( $w$ ) == UNEXPLORED then
6       setLabel( $e, DISCOVERY$ )
7       DFS( $G, W$ )
8     end
9     else
10    | setLabel( $e, BACK$ )
11   end
12 end
13 end

```

---

## 18.2. Implementierung DFS

```

1 public ArrayList<Node<T> depthFirstSearch(Node<T> from, Node<T> to) {
2   from.setMark(true);
3   stack.push(from);
4   if (from == to) {
5     return new ArrayList<Node<T>>(stack);
6   }
7   for (Node<T> n : from.getConnectedNodes()) {
8     System.out.println(from.getObject() + " : " + n.getObject() + " ");
9     if (!n.isMarked()) {
10       ArrayList<Node<T>> path = depthFirstSearch(n, to);
11       if (path != null) {
12         return path;
13       }
14     }
15   }
16   stack.pop();
17   return null;
18 }

```

---

### 18.3. BFS: Breadth First Search

- Breitensuche
- Der BFS Algorithmus findet im Gegensatz zum DFS den **direktestens Pfad** zu einem Knoten. Dies ist meist auch der kürzeste. Dies muss aber nicht gezwungenermassen sein, da z.B mehrere kleine Pfade schneller sind wie der direkte. (Beispiel SBB)
- Der BFS Algorithmus besucht jeden Vertex genau ein mal.
- Der BFS Algorithmus initialisiert die Knoten und Edges auf die selbe Weise wie der DFS.
- Der BFS Algorithmus ist im Gegensatz zum DFS **nicht rekursiv**.
- **BFS im Gegensatz zu DFS:** Finden und ausgeben eines Pfades mit einer minimalen Anzahl von Kanten.

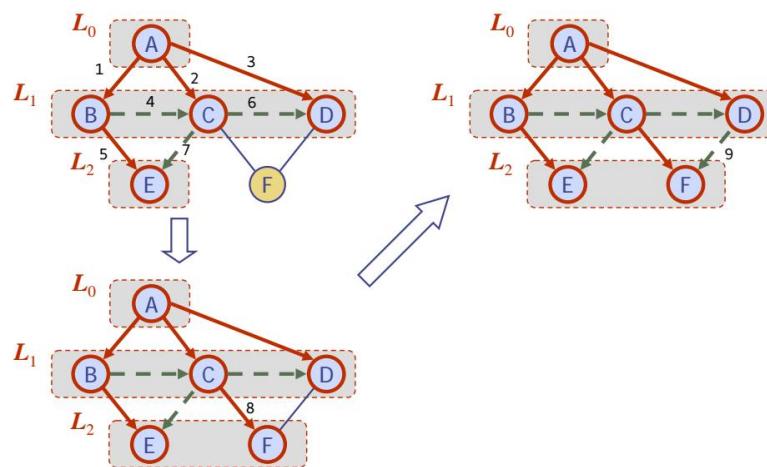


Abbildung 61: Breath First Search

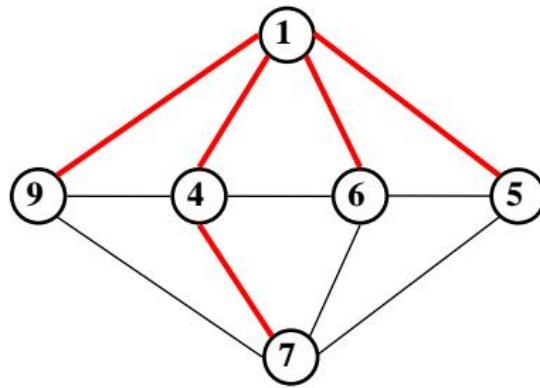


Abbildung 62: Breitensuche

**Algorithm 6:** BFS( $G, s$ )

---

```

1  $L_0 \leftarrow newEmptySequence$ 
2  $L_0.insertLast(s)$ 
3  $setLabel(s, VISITED)$ 
4  $i \leftarrow 0$ 
5 while  $\neg L_i.isEmpty$  do
6   forall  $v$  in  $L_i.elements()$  do
7     forall  $e$  in  $G.incidentEdges(v)$  do
8       if  $getLabel(e) == UNEXPLORED$  then
9          $w \leftarrow opposite(v, e)$ 
10        if  $getLabel(w) == UNEXPLORED$  then
11           $setLabel(e, DISCOVERY)$ 
12           $setLabel(w, VISITED)$ 
13           $L_{i+1}.insertLast(w)$ 
14        end
15        else
16           $setLabel(e, BACK)$ 
17        end
18      end
19    end
20  end
21 end

```

---

### 18.4. DFS vs. BFS

- n Vertizes
- m Kanten
- keine parallelen Kanten
- keine Schleifen

Beschreibung	Depth First Search	Breadth First Search
Laufzeit	$\mathcal{O}(n + m)$	$\mathcal{O}(n + m)$
Aufspannender Wald, Verbundene Komponenten, Pfade, Zyklen	✓	✓
Kürzester Pfad		✓
Biconnected Komponenten	✓	

Tabelle 19: Laufzeiten von Graph Operationen

## 19. Gerichtete Graphen (Directed Graphs, Digraph)

Ein gerichteter Graph (Digraph, Directed Graph) ist ein Graph, dessen Kanten alle gerichtet sind. Das bedeutet, dass die Kanten nur **unidirektional** begehbar sind. Die In und Out Katen werden in separaten Adjazenz Listen geführt.

### 19.1. Scheduling

Kante (a,b) bedeutet, dass Task a terminieren muss bevor Task b gestartet wird.

### 19.2. Laufzeiten

Wenn die In und Out Kanten in separaten Adjazenz Listen geführt werden, verläuft die Laufzeit proportional zur Grösse der Liste.

Beschreibung	Laufzeiten
Strong Connectivity Algorithmus	$\mathcal{O}(n + m)$
Transitiver Abschluss	$\mathcal{O}(n(n + m))$
Floyd-Warshalls Algorihtmus	$\mathcal{O}(n^3)$
Topologische Sortierung	$\mathcal{O}(n + m)$
Topologische Sortiereung mit DFS	$\mathcal{O}(n + m)$

Tabelle 20: Laufzeiten von Graph Operationen

### 19.3. Strong Connectivity

Bei gerichteten Graphen ist es nicht garantiert, dass alle Vertizes erreichbar sind. Bei einem Graphen der streng verbunden, kann jedoch **jeder Vertex alle anderen Vertizes erreichen**. Mit dem Strong Connectivy Algorihtmus kann mit nur **zwei Tiefensuchen** herausgefunden werden, ob ein Graph streng verbunden ist.

1. Wähle einen Vertex v in G und führe eine Tiefensuche durch. Wenn es einen nicht besuchten Vertex gibt, gib **false** zurück
2. Erstelle eine Kopie von G mit umgekehrten Kanten
3. Wähle den gleichen Vertex v in G' und führe eine Tiefensuche durch. Wenn es einen nicht besuchten Vertex gibt, gib **false** zurück. Ansonsten **true**.

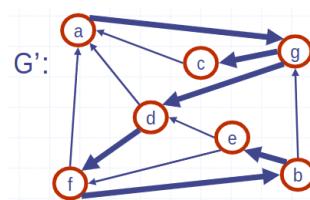
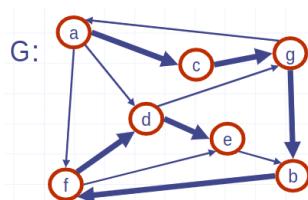


Abbildung 63: Strong Connectivity 1 Abbildung 64: Strong Connectivity 2

## 19.4. DFS und BFS

Sowohl die Tiefensuche als auch die Breitensuche kann für gerichtete Graphen angepasst werden.

**discovery** Baumkanten/Discovery sind Kanten des Pfades

**back** Rückkanten sind Verbindungen zu einem Vorgänger des selben Astes (der bereits auf DISCOVERY gesetzt ist)  
 (ACHTUNG gäbe es eine Verbindung von 2 nach 8, wäre es keine back-Kante sondern eine Cross Kante → Anderer Ast)

**forward** Vorwärtskanten sind Verbindungen zu einem Nachfolger im Baum (der bereits auf DISCOVERY gesetzt ist)

**cross** Kreuzkanten sind alle übrigen Kanten (z.B andere Pfad)

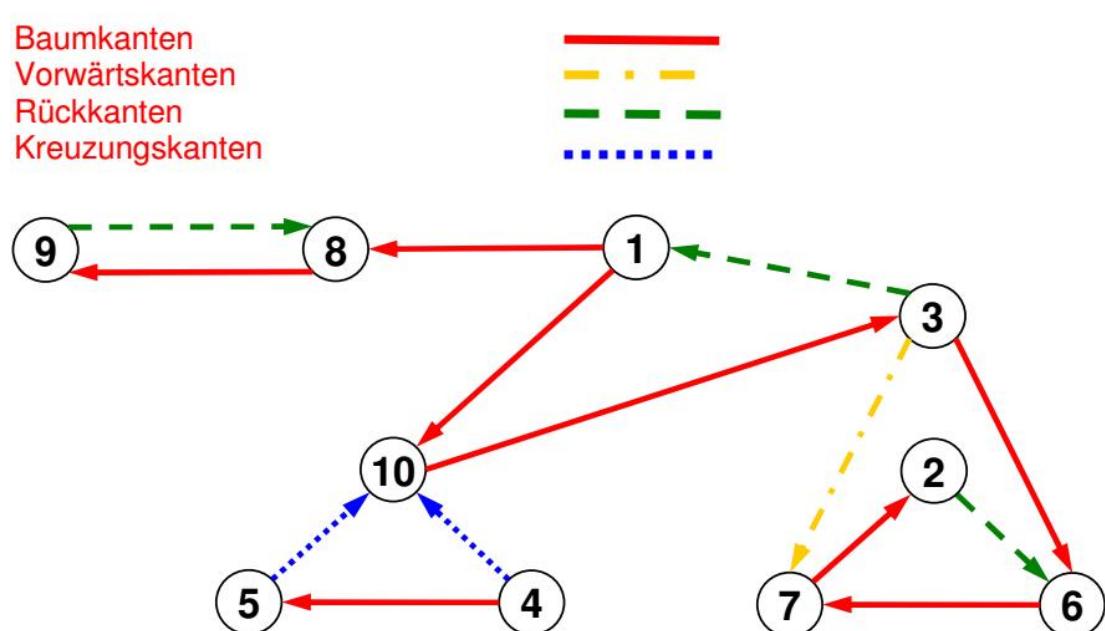


Abbildung 65: Digraph DFS

### 19.5. Transitiver Abschluss

Der Transitive Abschluss erweitert einen bestehenden Graphen um "Abkürzungen", sofern sowieso ein Pfad von einem Vertex zum anderen Vertex besteht. Wenn dies der Fall ist, bietet der Transitiven Abschluss den direkten Weg zwischen zwei Vertex.

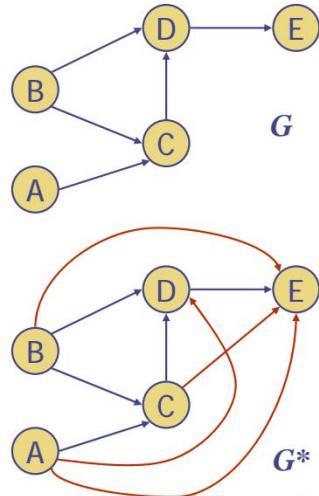


Abbildung 66: Transitiver Abschluss

## 19.6. Floyd-Warshalls Algorithmus

Der Floyd Warshalls Algorithmus erstellt einen Transitiven Abschluss für einen Graphen G. Er basiert auf dynamischer Programmierung. Das bedeutet, dass man auf die Lösung des vorangegangenen Problems aufsetzt. (z.B Nutzen des roten Pfeils, siehe Abbildung 67) In einem transitiven Abschluss sind alle Knoten direkt mit einander verbunden, welche so oder so über andere Knoten erreicht hätten werden können.

### 19.6.1. Vorgehen

1. Nummeriere alle Vertices der Reihe nach ( $v_1$  bis  $v_i$ )
2. Ausgehend vom ersten Vertex  $v_i$ 
  - a) Folge allen ausgehenden Edges zum nächsten Vertex und merke diesen.
  - b) Folge allen eingehenden Edges und verbinde diese Vertices jeweils → mit den Vertices im vorherigen Schritt, sofern diese nicht bereits verbunden sind.
3. Sind alle Vertices in diesem Schritt verbunden, geht man zum nächsten Vertex  $v_i + 1$  und wiederholt die Prozedur.
4. Die neu gezeichneten Edges **bleiben für die folgenden Schritte bestehend** und müssen ebenfalls beachtet werden (Dynamische Programmierung)

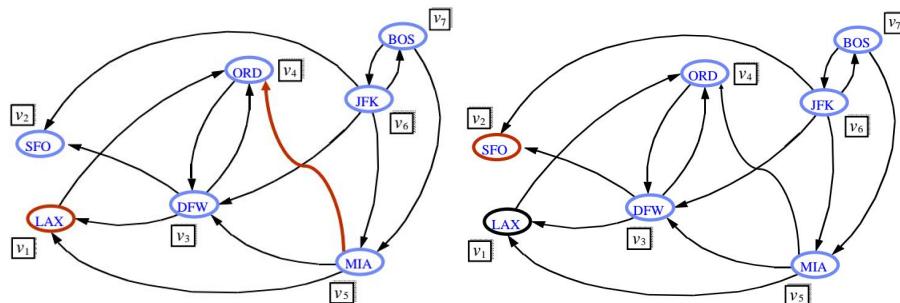


Abbildung 67: Schritt 1

Abbildung 68: Schritt 2

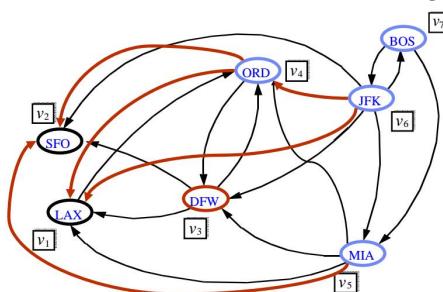


Abbildung 69: Schritt 3

---

**Algorithm 7:** FloydWarshall(G)

---

**Data:** digraph G  
**Result:** transitive closure  $G^*$  of G

```

1  $i \leftarrow 1$ 
2 forall  $v$  in  $G.vertices()$  do
3   | denote  $v$  as  $v_i$ 
4   |  $i \leftarrow i + 1$ 
5 end
6  $G_0 \leftarrow G$ 
7 for  $k \leftarrow 1$  to  $n$  do
8   |  $G_k \leftarrow G_{k-1}$ 
9   | for  $i \leftarrow 1$  to  $n(i \neq k)$  do
10  |   | for  $j \leftarrow 1$  to  $n(j \neq i, k)$  do
11  |   |   | if  $G_{k-1}.areAdjacent(v_i, v_k) \wedge G_{k-1}.areAdjacent(v_k, v_j)$  then
12  |   |   |   | if  $\neg G_k.areAdjacent(v_i, v_j)$  then
13  |   |   |   |   |  $G_k.insertDirectedEdge(v_i, v_j, k)$ 
14  |   |   |   | end
15  |   |   | end
16  |   | end
17  | end
18 end
19 return  $G_n$ 
```

---

### 19.6.2. Beispielaufgabe Floyd-Warshall

Aktueller Knoten ist grau markiert

- Immer die angrenzenden Knoten beachten
- Vorherig eingezeichnete Edges **müssen** beachtet werden
- **Fragen:**  
Welche angrenzenden Knoten (welche auf mich zeigen, **abkürzende eingeschlossen**) können über mich abkürzen?

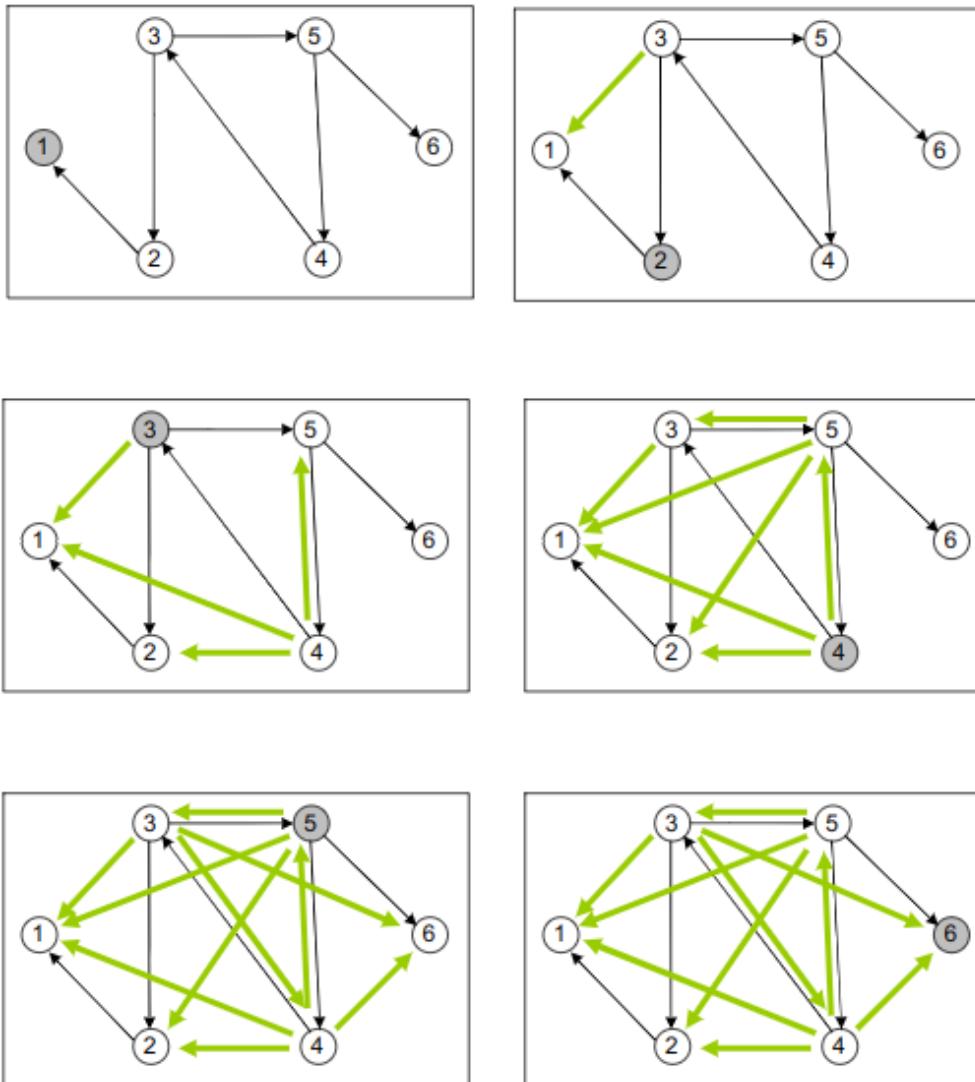


Abbildung 70: Floyd-Warshall Beispiel

## 19.7. DAG: Directed Acyclic Graph

Ein DAG enthält keine gerichteten Zyklen.

## 19.8. Topologische Sortierung

Man spricht von einer Topologischen Ordnung, wenn die Vertizes so **nummeriert** werden, dass die gerichteten Kanten immer auf grössere Vertizes zeigen. Eine topologische Ordnung kann nur in DAG's (keine Zyklen) erstellt werden.

- Eine Topologische Sortierung wird zum Beispiel beim kompilieren von C++ Objekt Files benötigt. So kann garantiert werden, dass keine unaufgelöste Referenzen existieren.
- Gleiches gilt bei Package Manager mit Dependencies.
- Der Algorithmus für die Topologische Sortierung läuft mit  $\mathcal{O}(n + m)$

---

**Algorithm 8:** TopologicalSort( $G$ )

---

```

1  $H \leftarrow G$ 
2  $n \leftarrow G.\text{numVertices}()$ 
3 while  $H$  is not empty do
4   Let  $v$  be a vertex with no outgoing edges
5   Label  $v \leftarrow n$ 
6    $n \leftarrow n - 1$ 
7   Remove  $v$  from  $H$ 
8 end

```

---

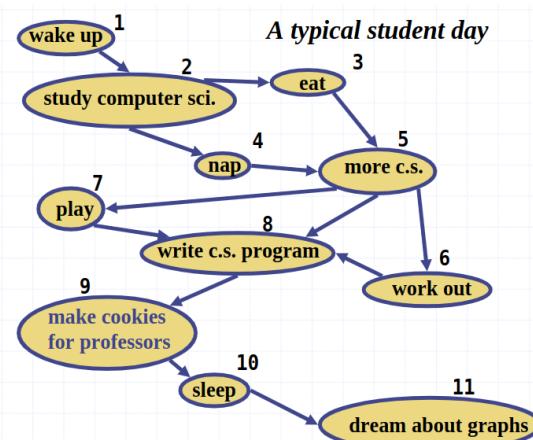


Abbildung 71: Topological Ordering

### 19.8.1. Vorgehen

Gibt es Zyklen, gibt es keine Topologische Sortierung und es ist kein DAG.

- Nimm den ersten Vertex, gemäss der Sortierung von `G.vertices()`.
- Mache eine Tiefensuche:
  1. Besuche die Vertices (gemäss gegebener Sortierung von `v.outgoingEdges()`), solange es einen gerichteten Edge hat und der Zielvertex noch nicht besucht wurde.
  2. Wenn es nicht mehr weitergeht, mache ein Backtracking, bis es einen neuen Weg gibt.
  3. Beim Backtracking werden die Nummern gesetzt. Angefangen beim Maximum (Anzahl Vertices)

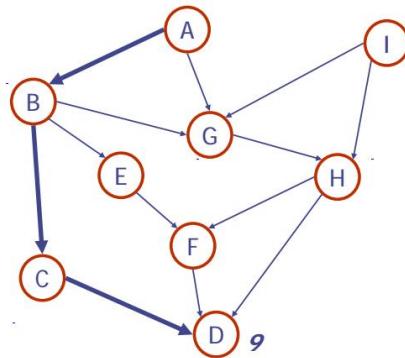


Abbildung 72: Schritt 1

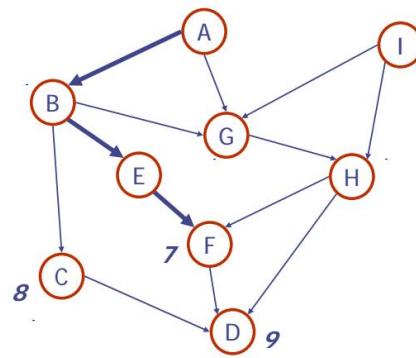


Abbildung 73: Nach dem ersten Backtracking

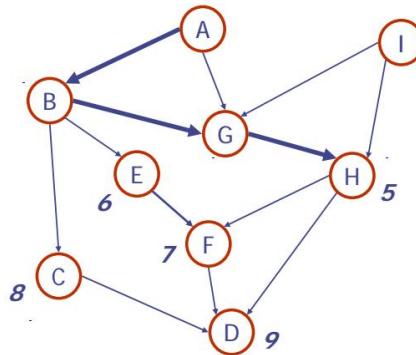


Abbildung 74: Nach dem zweiten Backtracking

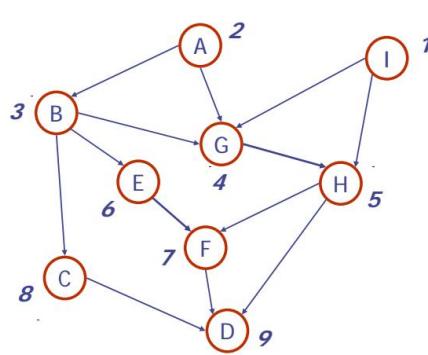


Abbildung 75: Topologische Sortierung

### 19.8.2. DAG Implementierung

Topologische Tiefensuche.

---

**Algorithm 9:** topologicalDFS(G)

---

**Data:** DAG G  
**Result:** topological ordering of G

```

1  $n \leftarrow G.\text{numVertices}()$ 
2 forall  $u$  in  $G.\text{vertices}()$  do
3   |  $\text{setLabel}(u, \text{UNEXPLORED})$ 
4 end
5 forall  $e$  in  $G.\text{edges}()$  do
6   |  $\text{setLabel}(e, \text{UNEXPLORED})$ 
7 end
8 forall  $v$  in  $G.\text{vertices}()$  do
9   | if  $\text{getLabel}(v) = \text{UNEXPLORED}$  then
10  |   |  $\text{topologicalDFS}(G, v)$ 
11  | end
12 end
```

---

**Algorithm 10:** topologicalDFS(G,v)

---

**Data:** graph G and a start vertex v of G  
**Result:** labeling of the vertices of G in the connected component of v

```

1  $\text{setLabel}(v, \text{VISITED})$ 
2 forall  $e$  in  $G.\text{outgoingEdges}(v)$  do
3   | if  $\text{getLabel}(e) = \text{UNEXPLORED}$  then
4     |   |  $w \leftarrow \text{opposite}(v, e)$ 
5     |   | if  $\text{getLabel}(w) = \text{UNEXPLORED}$  then
6     |   |   |  $\text{setLabel}(e, \text{DISCOVERY})$ 
7     |   |   |  $\text{topologicalDFS}(G, w)$ 
8     |   | end
9     |   | else
10    |   |   |  $e$  is a forward or cross edge
11    |   | end
12    | end
13 end
14 Label  $v$  with topological number  $n$ 
15  $n \leftarrow n - 1$ 
```

---

Listing 19: Directed DFS in Java

---

```

1  public void directedDFS() {
2      vertices.forEach(v -> vertexLabeling.put(v, VertexState.UNEXPLORED));
3      edges.forEach(e -> edgeLabeling.put(e, EdgeState.UNEXPLORED));
4
5      vertices.forEach(v -> {
6          if (vertexLabeling.get(v) == VertexState.UNEXPLORED) {
7              directedDFS(v);
8          }
9      });
10 }
11
12 public void directedDFS(Vertex vertex) {
13     vertexLabeling.put(vertex, VertexState.VISITED);
14
15     outgoingEdges(vertex).forEach(e -> {
16
17         if (edgeLabeling.get(e) == EdgeState.UNEXPLORED) {
18             Vertex opposite = opposite(vertex, e);
19             if (vertexLabeling.get(opposite) == VertexState.UNEXPLORED) {
20                 edgeLabeling.put(e, EdgeState.DISCOVERY);
21                 displayOnGVS();
22                 directedDFS(opposite);
23             } else {
24                 setKindOfEdge(vertex, e);
25                 displayOnGVS();
26             }
27         }
28     });
29 }
30
31 private void setKindOfEdge(Vertex startVertex, Edge e) {
32     Vertex endVertex = opposite(startVertex, e);
33     if (path.contains(endVertex)) {
34         // BACK
35     } else if (subtreeNodes.get(startVertex).contains(endVertex)) {
36         // FORWARD
37     } else {
38         // CROSS
39     }
40 }
```

---

## 20. Shortest Path Trees

- Der SPT Algorithmus benötigt einen gewichteten Graphen
- In einem gewichteten Graphen hat jede Kante einen assoziierten numerischen Wert, das sogenannte Gewicht
- Typische Anwendungsfälle sind Routing, Verkehr oder Navigation im Auto
- Ein kürzester Pfad hat zwei Eigenschaften
  1. Ein Teilweg eines kürzesten Weges ist selbst auch ein kürzester Weg
  2. Es existiert ein Baum von kürzesten Wegen von einem Start Vertex zu allen anderen Vertizes

### 20.1. Laufzeiten

Beschreibung	Laufzeiten
Dijkstra Algorithmus mit Adjazenz Listen Struktur	$\mathcal{O}((n + m) \cdot \log(n))$
Bellman Ford	$\mathcal{O}(n \cdot m)$
DAG basierter Ansatz	$\mathcal{O}(n + m)$

Tabelle 21: Laufzeiten von Graph Operationen

## 20.2. Dijkstra Algorithmus

Der Dijkstra Algorithmus berechnet die Distanzen zu allen Vertizes von einem Start Vertex aus. Dazu müssen **drei Annahmen** getroffen werden:

1. Der Graph ist verbunden
2. Die Kanten sind ungerichtet
3. Die Kantengewichte sind **nicht negativ**

Der Dijkstra Algorithmus ist ein Greedy Algorithmus, der immer den Vertex mit der kleinsten Distanz der Cloud hinzufügt. Um trotzdem mit negativen Gewichten umgehen zu können, könnte man die Gewichte einfach um das grösste negative Gewicht verschieben. Dabei muss aber beachtet werden, dass man die Wertebereiche der Datentypen nicht überschreitet.

**Relaxation(Entspannung)** Wenn ein besserer Pfad gefunden wurde, werden die umliegenden Vertizes aktualisiert.

### 20.2.1. Vorgehen

1. Die Standard Gewichte der Knoten ist  $\infty$ . Ausnahme ist der Start Vertex, dieser hat das Gewicht von 0.
2. Aktualisiere alle Gewichte der umliegenden Knoten, falls es nun einen kürzeren Pfad zu einem Knoten gibt. (Immer **aufaddieren**: Knoten Gewicht + Kanten Gewicht)
3. Der Wolke wir jener Vertex hinzugefügt, welcher noch nicht in der Wolke ist und den kleinsten Wert aufweist. Er muss aber von der Wolke erreichbar sein.
4. Fahre fort mit dem Knoten der in die Wolke hinzugefügt wurde
5. Wiederhole diese Schritte, bis alle Vertex in der Wolke sind.
6. Der Shortest Path ist nun der rote Pfad

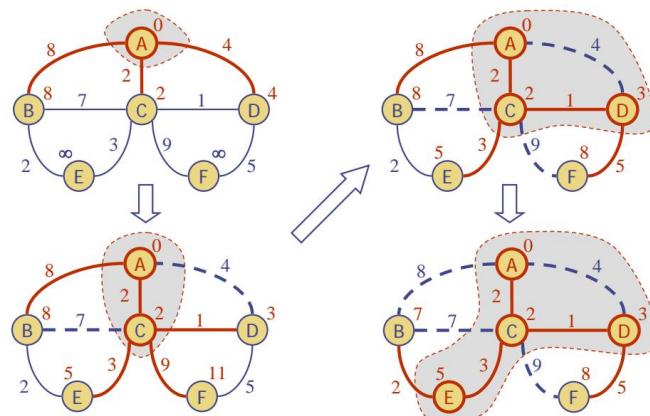


Abbildung 76: Dijkstra Algorithmus

### 20.3. Dijkstra Algorithmus

1. Eine Adaptierbare Priority Queue speichert die Vertizes ausserhalb der Wolke.

Key: Distanz

Element: Vertex

**Wegen replaceKey() brauchen wir eine Adaptable Priority Queue**

2. Locator-basierte Methoden

insert(k,e) gibt einen Locator zurück

replayeKey(l,k) ändert den Schlüssel eines Eintrags

3. Wir speichern zwei Eigenschaften (labels) mit jedem Vertex:

Distanz  $d(V)$

Locator in der Priority Queue

```
Algorithm DijkstraDistances( $G, s$ )
   $Q \leftarrow$  new heap-based adaptable PQ
  for all  $v \in G.vertices()$ 
    if  $v = s$ 
      setDistance( $v, 0$ )
    else
      setDistance( $v, \infty$ )
     $l \leftarrow Q.insert(getDistance(v), v)$ 
    setLocator( $v, l$ )
  while  $\neg Q.isEmpty()$ 
     $u \leftarrow Q.removeMin().getValue()$ 
    for all  $e \in G.incidentEdges(u)$ 
      { relax edge  $e$  }
       $z \leftarrow G.opposite(u, e)$ 
       $r \leftarrow getDistance(u) + weight(e)$ 
      if  $r < getDistance(z)$ 
        setDistance( $z, r$ )
       $Q.replaceKey(getLocator(z), r)$ 
```

```
Algorithm DijkstraShortestPathsTree( $G, s$ )
  ...
  for all  $v \in G.vertices()$ 
  ...
  setParent( $v, \emptyset$ )
  ...

  for all  $e \in G.incidentEdges(u)$ 
    { relax edge  $e$  }
     $z \leftarrow G.opposite(u, e)$ 
     $r \leftarrow getDistance(u) + weight(e)$ 
    if  $r < getDistance(z)$ 
      setDistance( $z, r$ )
      setParent( $z, e$ )
       $Q.replaceKey(getLocator(z), r)$ 
```

Abbildung 77: Dijkstra Distance

Abbildung 78: Dijkstra Shortest Path

### 20.3.1. Dijkstras Distance Beispiel Übung

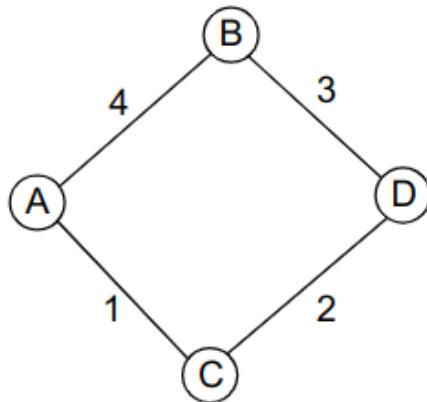


Abbildung 79: Tree Vorgabe

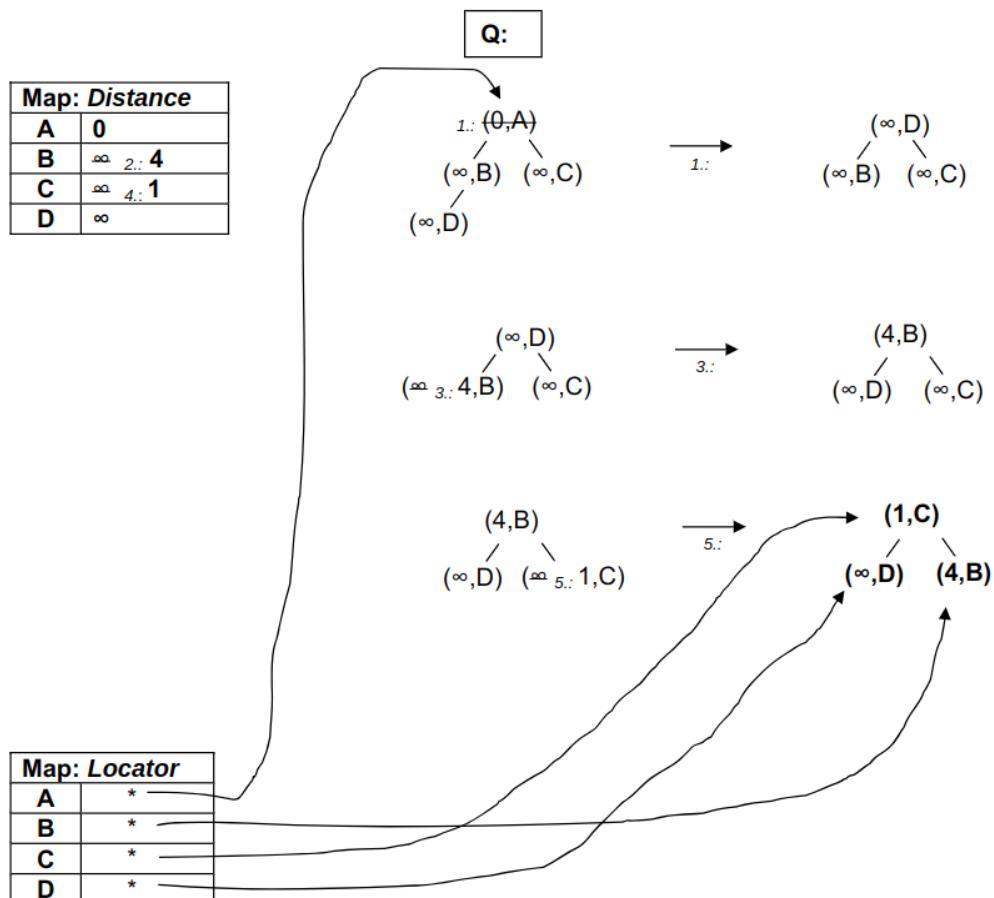


Abbildung 80: Lösung Queue und Map

### 20.3.2. Implementierung

Der Dijkstra Algorithmus verwendet eine adaptierbare Priority Queue, wobei der Key die Distanz und das Value der Vertex ist. Die Eigenschaft der APQ ist es, dass man die Keys verändern kann. (Im Gegensatz zur Priority Queue)

Listing 20: Dijkstra Algorithmus

---

```

1  public void distances(AdjacencyListGraph<V, E> graph, Vertex<V> s) {
2      AdaptablePriorityQueue<Integer, Vertex<V>> apq =
3          new HeapAdaptablePriorityQueueGVS<Integer, Vertex<V>>();
4      Map<Vertex<V>, Integer> distances = new LinkedHashMapGVS<Vertex<V>, Integer>();
5      Map<Vertex<V>, Entry<Integer, Vertex<V>>> locators =
6          new LinkedHashMap<Vertex<V>, Entry<Integer, Vertex<V>>>();
7      Map<Vertex<V>, Edge<E>> parents = new LinkedHashMapGVS<Vertex<V>, Edge<E>>();
8      gvs.set(apq, distances, parents);
9
10     for (Vertex<V> v : graph.vertices()) {
11         if (v == s) {
12             distances.put(v, 0);
13             // root node has no parents
14             parents.put(v, null);
15         } else {
16             // set default distance to infinity
17             distances.put(v, Integer.MAX_VALUE);
18         }
19         // add distance and vertex
20         Entry<Integer, Vertex<V>> entry = apq.insert(distances.get(v), v);
21         locators.put(v, entry);
22     }
23
24     while (!apq.isEmpty()) {
25         // take next vertex out of the queue (removeMin) -> Kehrwert = cloud
26         AdjacencyListGraph<V, E>.MyVertex<V> cloudVertex =
27             (AdjacencyListGraph<V, E>.MyVertex<V>) (apq.removeMin().getValue());
28
29         for (Edge<E> incidentEdge : cloudVertex.incidentEdges()) {
30             Vertex<V> oppositVertex = graph.opposite(cloudVertex, incidentEdge);
31             // calculate new weight: last vertex + edge weight
32             int newWeight = distances.get(cloudVertex) + (Integer)
33                 incidentEdge.get(WEIGHT);
34             if (newWeight < distances.get(oppositVertex)) {
35                 // relaxion
36                 distances.put(oppositVertex, newWeight);
37                 parents.put(oppositVertex, incidentEdge);
38                 apq.replaceKey(locators.get(oppositVertex), newWeight);
39             }
40         }
41     }

```

---

### 20.3.3. Implementierung Path Finder

---

```

1  public ArrayList<Point> findPath(int startx, int starty, int endx, int endy) {
2      for (int x = 0; x < LEN_X; x++) {
3          for (int y = 0; y < LEN_Y; y++) {
4              if (x == startx && y == starty) {
5                  distances[x][y] = 0.0;
6              } else {
7                  distances[x][y] = Double.POSITIVE_INFINITY;
8              }
9              VertexPos v = new VertexPos(x, y);
10             Entry<Double, VertexPos> l = q.insert(distances[x][y], v);
11             locators[x][y] = l;
12         }
13     }
14     while (!q.isEmpty()) {
15         Entry<Double, VertexPos> l = q.removeMin();
16         int ux = l.getValue().x;
17         int uy = l.getValue().y;
18         if ((ux == endx) && (uy == endy)) { // found :-)
19             generatePath(ux, uy);
20             break;
21         }
22         int zx = 0;
23         int zy = 0;
24         for (Direction direction : Direction.values()) {
25             zx = Math.max(Math.min(direction.nextX(ux), LEN_X - 1), 0);
26             zy = Math.max(Math.min(direction.nextY(uy), LEN_Y - 1), 0);
27             double r = distances[ux][uy] + map.calcWeight(ux, uy, zx, zy);
28             if (r < distances[zx][zy]) {
29                 distances[zx][zy] = r;
30                 parents[zx][zy] = direction.getOpposite();
31                 q.replaceKey(locators[zx][zy], r);
32
33                 // showing current position on map:
34                 currentX = zx;
35                 currentY = zy;
36                 try {
37                     Thread.sleep(1); // 0: much faster; >1 : slower, but more details
38                 } catch (InterruptedException e) {}
39             }
40         }
41     }
42     setChanged();
43     notifyObservers();
44     return path;
45 }
46 private void generatePath(int endX, int endY) {
47     path.add(0, new Point(endX, endY));
48     int x = endX;
49     int y = endY;
50     while (parents[x][y] != null) {
51         int newX = parents[x][y].nextX(x);
52         int newY = parents[x][y].nextY(y);
53         path.add(0, new Point(newX, newY));
54         x = newX;
55         y = newY;
56     }
57 }
58 }
```

---

## 20.4. Bellman-Ford

- Im Gegensatz zum Dijkstra Algorithmus, funktioniert der BF Algorithmus **auch mit negativen Gewichten**
- Es gibt zwei voraussetzungen
  - gerichtete Kanten
  - keine negativ-gewichtete Schlaufen!
- Die Laufzeit ist jedoch deutlich schlechter:  $\mathcal{O}(n \cdot m)$
- Der BF Algorithmus **iteriert über alle Kanten** des Graphen und nicht nur um die umliegenden Kanten.

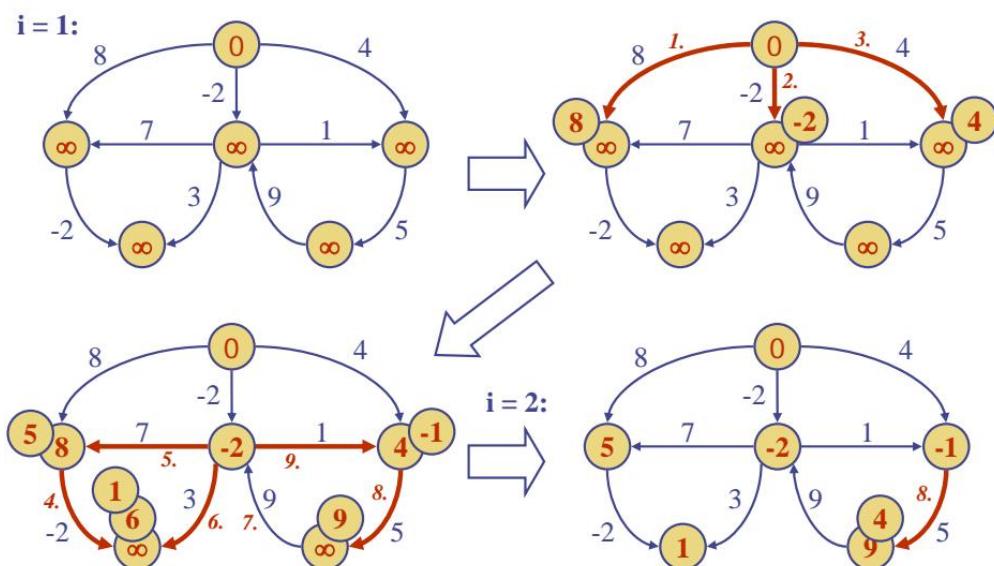


Abbildung 81: Bellman-Ford Algorithmus

### 20.4.1. Dijkstra vs. Bellman-Ford

Der zentrale Unterschied zu Dijkstra ist, dass Bellman-Ford bei beiden Loops immer über alle Kanten und Vertices geht.

### 20.4.2. Implementierung

**Algorithm 11:** BellmanFord( $G, s$ )

---

```

1 forall  $v$  in  $G.vertices()$  do
2   | if  $v = s$  then
3     |   | setDistance( $v, 0$ )
4   | end
5   | else
6     |   | setDistance( $v, \infty$ )
7   | end
8   | for  $i \leftarrow 1$  to  $n - 1$  do
9     |   |  $u \leftarrow G.origin(e)$ 
10    |   |  $z \leftarrow G.opposite(u, e)$ 
11    |   |  $r \leftarrow getDistance(u) + weight(e)$ 
12    |   | if  $r < getDistance(z)$  then
13      |   |   | setDistance( $z, r$ )
14    |   | end
15  | end
16 end

```

---

### 20.5. DAG basierter Algorithmus

- Funktioniert wie der BF Algorithmus mit negativ-gewichteten Kanten
- Ein DAG ist ein gerichteter Graph **ohne Zyklen**
- Benutzt eine topologische Reihenfolge
- Ist sehr schnell:  $\mathcal{O}(n + m)$
- Kontrolliert bei jedem Knoten die **Outgoing Edges**

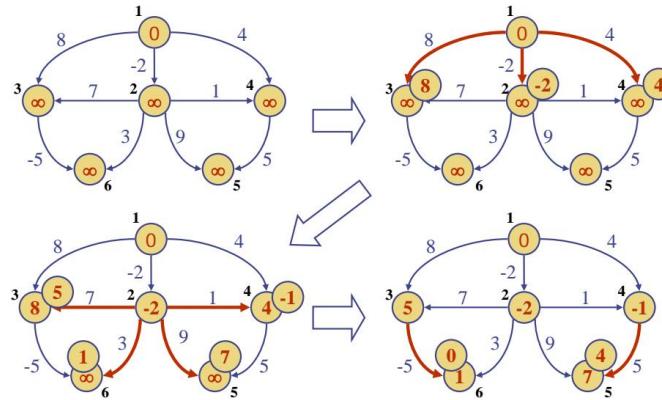


Abbildung 82: DAG Shortest Path

## 21. Minimum Spanning Tree

- Anwendungsfälle sind Kommunikationsnetzwerke und Transportnetzwerke
- Ein minimaler Spanning Tree ist ein Subset von Kanten in einem ungerichteten, bidirektionalen, gewichteten Graphen der alle Knoten ohne Zyklen und mit den kleinsten Kosten verbindet.
- **Aufspannender Subgraph** Subgraph eines Graphen  $G$  beinhaltet alle Vertizes von  $G$
- **Aufspannender Baum** Aufspannender Subgraph, der selbst ein (freier) Baum ist
- **Minimal Aufspannender Baum (MST)** Aufspannender Baum eines gewichteten Graphen mit minimalem totalen Kantengewicht
- **Anwendungen:** Kommunikationsnetzwerke, Transportnetzwerke

### Schlaufen Eigenschaft

Gibt es eine Kante  $e$  die noch nicht zum MST gehört und ein tieferes Gewicht hat, wie mindestens eine Kante im MST, ersetzt sie die Kanten mit dem höheren Gewicht, sofern sie den MST zu einer Schleife formt.

**Aufteilungseigenschaft** Die Kante mit dem **kleinsten Gewicht** muss Teil des Pfades sein

### 21.1. Kruskal Algorithmus

- Der Kruskal Algorithmus merkt sich einen Forest von Trees.
- Eine Kante ist akzeptiert, wenn sie zwei Trees verbindet.
- Eine Priority Queue speichert die Kanten ausserhalb der Wolke (Key: Gewicht, Value: Kante).

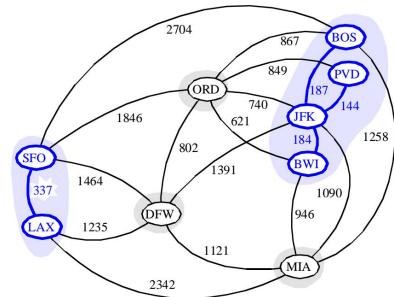


Abbildung 83: Step 1

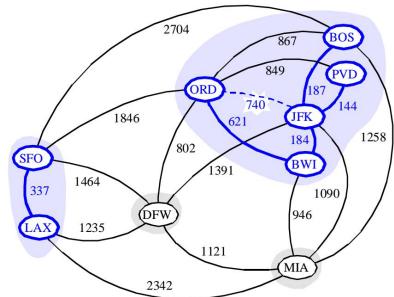


Abbildung 84: Step 2

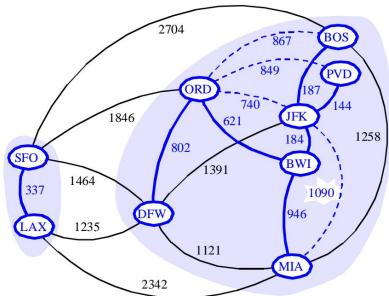


Abbildung 85: Step 3

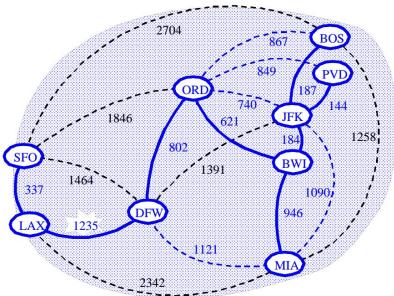


Abbildung 86: Step 4

### 21.1.1. Kruskal Implementierung

```

Algorithm Kruskal( $G$ ):
  Input: Ein gewichteter Graph  $G$ .
  Output: Ein MST  $T$  für  $G$ .
   $P$  sei eine Partition der Vertizes von  $G$ , wobei jeder Vertex ein Set für sich bildet
   $Q$  sei eine Priority Queue, welche die Kanten von  $G$  nach Gewichtung sortiert
   $T$  sei ein ursprünglich leerer Baum
  while  $Q$  ist nicht leer do
     $(u,v) \leftarrow Q.\text{removeMinElement}()$ 
    if  $P.\text{find}(u) \neq P.\text{find}(v)$  then
      Add  $(u,v)$  to  $T$ 
       $P.\text{union}(u,v)$ 
  return  $T$ 

```

Laufzeit:  
 $O(m \log n)$

Abbildung 87: Partition-basierte Implementation

```

Algorithm KruskalMST( $G$ )
  for jeden Vertex  $V$  in  $G$  do
    definiere eine  $\text{Cloud}(v)$  of  $\leftarrow \{v\}$ 
   $Q$ : eine Priority Queue
  Alle Kanten in  $Q$  einfügen mit dem
  Gewicht als Key
   $T \leftarrow \emptyset$ 
  while  $T$  weniger als  $n-1$  Kanten hat do
    edge  $e = Q.\text{removeMin}()$ 
     $u, v$ : Endpunkte von  $e$ 
    if  $\text{Cloud}(v) \neq \text{Cloud}(u)$  then
      Füge Kante  $e$   $T$  hinzu
      Merge  $\text{Cloud}(v)$  und  $\text{Cloud}(u)$ 
  return  $T$ 

```

Abbildung 88:  $n-1$  Kanten alle Vertizes im MST

### 21.1.2. Repräsentation einer Partition

- Jedes Set ist in einer Sequenz gespeichert
- Jedes Element hat eine Referenz zurück auf das Set

Operation **find(u)** benötigt **O(1)** Zeit und gibt das Set zurück, in dem u ist

In der Operation **union(u,v)** verschieben wir die Elemente des kleineren Sets in die Sequenz des grösseren Sets und aktualisieren deren Referenz

Die Zeit für die Operation **union(u,v)** ist  $\min(n_u, n_v)$ , wobei  $n_u$  und  $n_v$ , die Grössen der Sets sind, die u und v beinhalten

- Wenn ein Element verarbeitet wird, geht es in ein Set mit mindestens doppelter Grösse. Jedes Element wird also höchstens **log n** mal verarbeitet

## 21.2. SPT und MST

**Aufgabe:** Gewicht vom Minimal Spanning Tree und Shortest Path Tree berechnen

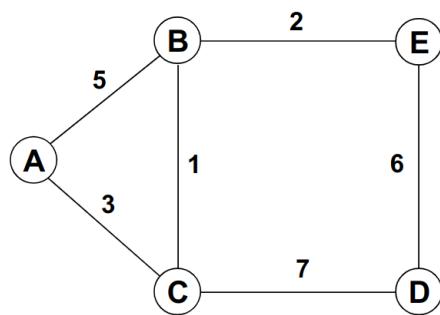


Abbildung 89: Spanning Tree Vorgabe

SPT	MST
A-C: 3	B-C: 1
C-B: 1	B-E: 2
B-E: 2	A-C: 3
C-D: 7	E-D: 6
Total=13	Total=12

Tabelle 22: Berechnung

### 21.3. Prim-Jarnik's Algorithmus

Ist ein modifizierter Dijkstra Algorithmus. Wir nehmen einen beliebigen Vertex  $s$  und generieren den minimalen Spanning Tree als Wolke von Vertizes von  $s$ . Danach speichern wir zu jedem Vertex ein label  $d(v) = \text{kleinste Gewichtung einer Kante, welche } v \text{ mit einem Vertex der Wolke verbindet}$ . Bei jedem Schritt werden folgende Dinge durchgeführt:

#### 21.3.1. Vorgehen

1. Die Standard Gewichte der Knoten ist  $\infty$ . Ausnahme ist der Start Vertex, dieser hat das Gewicht von 0.
2. Aktualisiere alle Gewichte der umliegenden Knoten mit den **Gewichten der Kante**. Hier wird nichts aufaddiert.
3. Der Wolke wir jener Vertex hinzugefügt, welcher noch nicht in der Wolke ist und den kleinsten Wert aufweist. Er muss aber von der Wolke erreichbar sein.
4. Fahre fort mit dem Knoten der in die Wolke hinzugefügt wurde
5. Wiederhole diese Schritte, bis alle Vertex in der Wolke sind.
6. Der Minimal Spanning Tree ist nun der rote Pfad

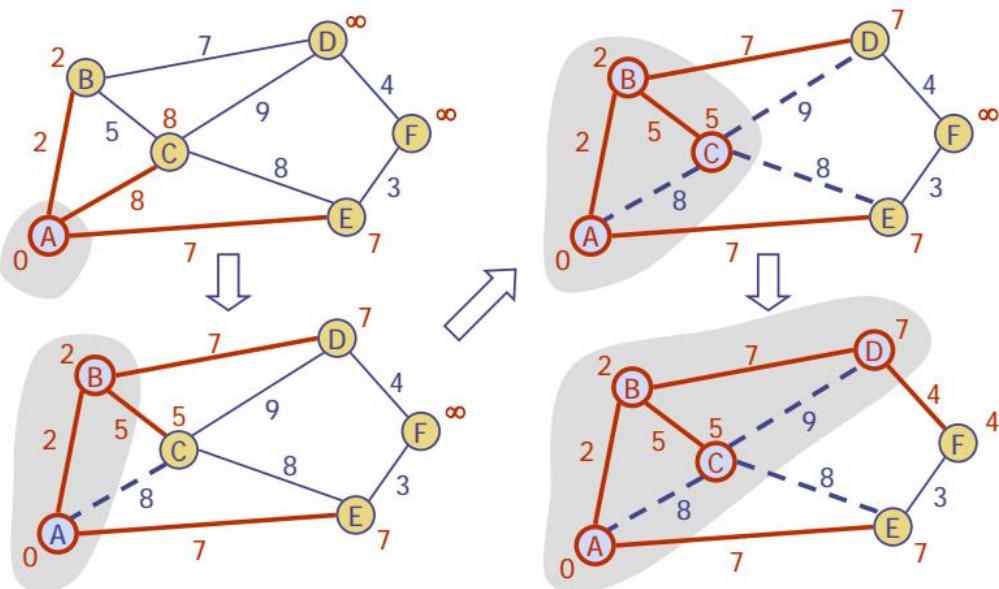


Abbildung 90: Prim-Jarnik's Algorithmus

```

Algorithm PrimJarnikMST( $G$ )
   $Q \leftarrow$  new heap-based adaptable PQ
   $cloud \leftarrow$  new Hash-Set
   $s \leftarrow$  a vertex of  $G$ 
  for all  $v \in G.vertices()$ 
    if  $v = s$ 
       $setDistance(v, 0)$ 
    else
       $setDistance(v, \infty)$ 
       $setParent(v, \emptyset)$ 
     $l \leftarrow Q.insert(getDistance(v), v)$ 
     $setLocator(v,l)$ 
  while  $\neg Q.isEmpty()$ 
     $u \leftarrow Q.removeMin().getValue()$ 
     $cloud.add(u)$ 
    for all  $e \in G.incidentEdges(u)$ 
       $z \leftarrow G.opposite(u,e)$ 
      if  $\neg cloud.contains(z)$ 
         $r \leftarrow weight(e)$ 
        if  $r < getDistance(z)$ 
           $setDistance(z,r)$ 
           $setParent(z,e)$ 
           $Q.replaceKey(getLocator(z),r)$ 

```

Abbildung 91: Prim-Jarnik Algorithmus

## 21.4. Borůvka's Algorithmus

Der Borůvka's Algorithmus arbeitet wie der Kruskal Algorithmus, mit einem Unterschied: Hier gibt es **für jede Cloud eine Priority Queue**. Dieser Algorithmus wird selten verwendet und ist **nur historisch relevant**.

## 21.5. Laufzeit

Beschreibung	Laufzeiten
Partition-based Kruskal	$\mathcal{O}(m \cdot \log(n))$
Prim-Jarnik's	$\mathcal{O}(m \cdot \log(n))$
Borůvka's	$\mathcal{O}(m \cdot \log(n))$

Tabelle 23: Laufzeiten von Graph Operationen

## A. Listings

1.	Inorder Traversal . . . . .	10
2.	ArrayList basierter Einsatz . . . . .	14
3.	BST Node . . . . .	17
4.	BST Entry . . . . .	17
5.	AVL Tree Rotations . . . . .	25
6.	AVL Tree: Single right rotation . . . . .	25
7.	AVL Tree: Single left rotation . . . . .	26
8.	AVL Tree: Right/Left Rotation . . . . .	27
9.	AVL Tree: Left/Right Rotation . . . . .	28
10.	AVL Tree Node . . . . .	30
11.	AVL Tree . . . . .	31
12.	Bubble Sort . . . . .	39
13.	Nicht Rekursiver Merge Sort . . . . .	41
14.	Rekursiver Merge Sort . . . . .	42
15.	Quick Sort ohne Comparator . . . . .	45
16.	Inplace Quick Sort . . . . .	46
17.	Knuth-Morris-Pratt Algorithmus . . . . .	60
18.	Knuth-Morris-Pratt Algorithmus Fehlfunktion . . . . .	60
19.	Directed DFS in Java . . . . .	94
20.	Dijkstra Algorithmus . . . . .	99

## B. Abbildungsverzeichnis

1.	Laufzeiten . . . . .	6
2.	find(7) . . . . .	12
3.	Einfügen wenn der Key 5 noch nicht vorhanden . . . . .	14
4.	Einfügen wenn der Key 2 bereits vorhanden . . . . .	14
5.	Zwei Blatt Kinder. . . . .	15
6.	Ein Blatt Kind. . . . .	15
7.	Keine Blatt Kinder . . . . .	15
8.	Rechts Rotation um c . . . . .	25
9.	Nach der rechts Rotation . . . . .	25
10.	Links Rotation um a . . . . .	26
11.	Nach der Links Rotation . . . . .	26
12.	Rechts Rotation um b . . . . .	27
13.	Links Rotation um a . . . . .	27
14.	Nach Rechts/Links Rotation . . . . .	27
15.	Links Rotation um a . . . . .	28
16.	Rechts Rotation um c . . . . .	28
17.	Nach Links/Rechts Rotation . . . . .	28
18.	Cut/Link Restrukturierung . . . . .	29
19.	Balancierter Baum nach Cut/Link . . . . .	30
20.	Splay Tree Flussdiagramm . . . . .	34
21.	Splay Tree Beispiele . . . . .	35
22.	Splay Tree: Löschen des Wert 3 . . . . .	36
23.	Lexikographische Sortierung . . . . .	38
24.	Merge Sort Algorithmus . . . . .	40
25.	Sequenzen zusammenfügen . . . . .	40
26.	Aufteilung in der Inputsequenz. $O(n)$ . . . . .	43
27.	InPlace Quicksort . . . . .	44
28.	In Place Quick Sort Algorithmus . . . . .	44
29.	Bucket Sort Algorithmus . . . . .	47
30.	Bucket Sort . . . . .	47
31.	Radix Sort Algorithmus . . . . .	49
32.	Radix Sort Binär . . . . .	49
33.	Boyer Moore Last Occurrence . . . . .	53
34.	Match bereits vorbei . . . . .	53
35.	Boyer Moore Algorithmus . . . . .	54
36.	Boyer Moore Algorithmus . . . . .	55
37.	Boyer Moore Beispiel . . . . .	55
38.	1. Fehlfunktion aufbauen . . . . .	57
39.	2. Knuth-Morris-Pratt Algorithm . . . . .	58
40.	KMP Algorithmus . . . . .	59
41.	KMP Failure Algorithm . . . . .	59
42.	Trie Beispiel . . . . .	61
43.	Trie Ausgangslage . . . . .	62
44.	Trie nach Kompression . . . . .	62
45.	Kompakte Repräsentation eines komprimierten Tries . . . . .	62
46.	Suffix Trie . . . . .	63
47.	Suffix Trie with Index Representation . . . . .	63

48.	Dynamische Programmierung, Rucksackproblem . . . . .	67
49.	Longest Common Subsequence . . . . .	68
50.	Parallele Kanten und Schleifen . . . . .	70
51.	Subgraphen . . . . .	71
52.	Pfad . . . . .	71
53.	Zyklus . . . . .	71
54.	Kanten-Listen Struktur . . . . .	72
55.	Adjazenz Listen Struktur . . . . .	74
56.	Adjazenz-Matrix Struktur . . . . .	75
57.	Berechnungsaufgabe . . . . .	78
58.	DFS Benennung . . . . .	80
59.	DFS Beispiel . . . . .	80
60.	Tiefensuche . . . . .	80
61.	Breath First Search . . . . .	82
62.	Breitensuche . . . . .	83
63.	Strong Connectivity 1 . . . . .	85
64.	Strong Connectivity 2 . . . . .	85
65.	Digraph DFS . . . . .	86
66.	Transitive Abschluss . . . . .	87
67.	Schritt 1 . . . . .	88
68.	Schritt 2 . . . . .	88
69.	Schritt 3 . . . . .	88
70.	Floyd-Warshall Beispiel . . . . .	90
71.	Topological Ordering . . . . .	91
72.	Schritt 1 . . . . .	92
73.	Nach dem ersten Backtracking . . . . .	92
74.	Nach dem zweiten Backtracking . . . . .	92
75.	Topologische Sortierung . . . . .	92
76.	Dijkstra Algorithmus . . . . .	96
77.	Dijkstra Distance . . . . .	97
78.	Dijkstra Shortest Path . . . . .	97
79.	Tree Vorgabe . . . . .	98
80.	Lösung Queue und Map . . . . .	98
81.	Bellman-Ford Algorithmus . . . . .	101
82.	DAG Shortest Path . . . . .	102
83.	Step 1 . . . . .	104
84.	Step 2 . . . . .	104
85.	Step 3 . . . . .	104
86.	Step 4 . . . . .	104
87.	Partition-basierte Implementation . . . . .	105
88.	n-1 Kanten alle Vertizes im MST . . . . .	105
89.	Spanning Tree Vorgabe . . . . .	106
90.	Prim-Jarnik's Algorithmus . . . . .	107
91.	Prim-Jarnik Algorithmus . . . . .	108

## C. Tabellenverzeichnis

1.	Laufzeitverhalten von Datenstrukturen . . . . .	7
2.	Laufzeitverhalten von Sortier- und Suchalgorithmen . . . . .	7
3.	Laufzeitverhalten von Suchtabellen . . . . .	11
4.	Speicherverbrauch von Binären Suchbäumen . . . . .	16
5.	Laufzeitverhalten von AVL Trees . . . . .	21
6.	Inorder Array für Cut/Link Restrukturierung . . . . .	29
7.	Laufzeitverhalten von Splay Trees . . . . .	36
8.	Laufzeitverhalten von Splay Trees . . . . .	36
9.	Laufzeitverhalten von vergleichbasierten Sortieralgorithmen . . . . .	37
10.	Laufzeitverhalten von nicht vergleichbasierten Sortieralgorithmen . . . . .	37
11.	Big Oh Merge Sort . . . . .	39
12.	Big Oh Merge Sort . . . . .	40
13.	Big Oh Quick Sort . . . . .	44
14.	Big Oh Bucket Sort . . . . .	48
15.	Big Oh Bucket Sort . . . . .	49
16.	Big Pattern Matching Boyer-Moore und KMP . . . . .	52
17.	Big Oh Tries . . . . .	63
18.	Laufzeiten von Graph Operationen . . . . .	75
19.	Laufzeiten von Graph Operationen . . . . .	84
20.	Laufzeiten von Graph Operationen . . . . .	85
21.	Laufzeiten von Graph Operationen . . . . .	95
22.	Berechnung . . . . .	106
23.	Laufzeiten von Graph Operationen . . . . .	108