Zusammenfassung

Algorithmen und Datenstrukturen 2

Michael Wieland

Hochschule für Technik Rapperswil

13. August 2017

Mitmachen
Falls du an diesem Dokument mitarbeiten möchtest, kannst du es auf GitHub unter `https://github.com/michiwieland/hsr-zusammenfassungen` forken.

Inhaltsverzeichnis

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

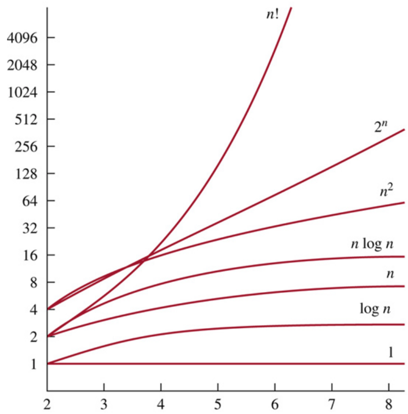


Abbildung 1: Laufzeiten

1.2. Summenformel

n Iterationen

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

1.3. n - 1 Iterationen

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

1.4. Stirling Formel

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

$$\lim_{n\to\infty} \log_2(n!) = n \cdot \log_2(n) \tag{3}$$

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 Operatione mit pos	$\mathcal{O}(1)$
Hash Table	Worst case (Kollisionen)	seach, 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 grösse 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.

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

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

```
public Collection<Entry<K, V>> inorder() {
   Collection<Entry<K, V>> inorderCollection = new LinkedList<>();
   inorder(root, inorderCollection);
   return inorderCollection;
}

protected void inorder(Node node, Collection<Entry<K, V>> inorderCollection) {
   if (node != null) {
      inorder(node.getLeftChild(), inorderCollection);
      inorderCollection.add(node.getEntry());
      inorder(node.getRightChild(), inorderCollection);
   }
}
```

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**
- 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.

5.3. Operationen

find(key) Liefert den Eintrag zum Schlüssel k oder null

Algorithm 1: TreeSearch(k,v)

if $T.isExternal(v)$ **then**
  return $null$
end if
if $k < key(v)$ **then**
  return $TreeSearch(k, T.left(v))$
else if $k = key(v)$ **then**
  return v
else
  return $TreeSearch(k, T.right(v))$
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.

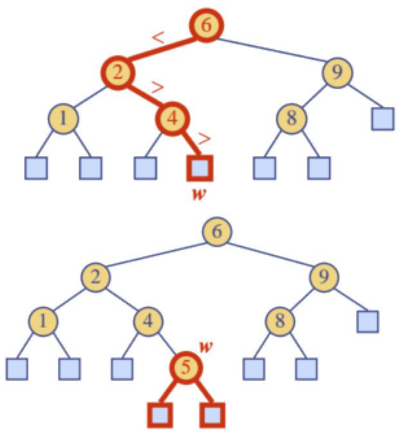


Abbildung 2: Einfügen wenn der Key 5 noch nicht vorhanden

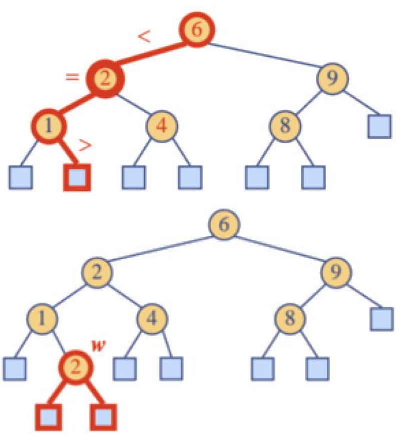


Abbildung 3: Einfügen wenn der Key 2 bereits vorhanden

Listing 2: Arraylist basierter Einsatz

```
public void add(int lower, int upper, int content) {
   int middle = (lower + upper) / 2;
   if (content < arrayList.get(middle).intValue()) {
      // go left
      if (middle == 0 || content > arrayList.get(middle - 1)) {
         arrayList.add(middle, content);
      } else {
         add(lower, middle - 1, content);
      }
   } else {
      // go right
      if (middle + 1 >= arrayList.size() || content < arrayList.get(middle + 1)) {
         arrayList.add(middle + 1, content);
      } else {
      add(middle + 1, upper, content);
      }
   }
}
```

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 löschender Knoten hat **zwei Blatt Kinder**: `removeExternal(w)` löscht den Blattknoten w und seinen Parent und ersetzt den Parent mit dem Geschwisterknoten von w
  - Zu löschender 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 löschender 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.

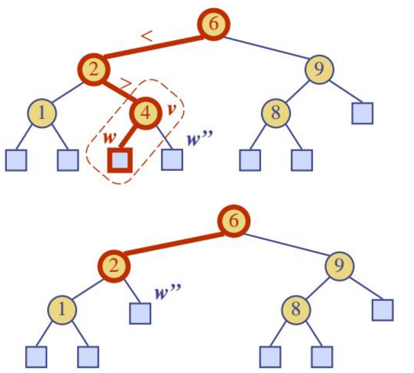


Abbildung 4: Zwei Blatt Kinder

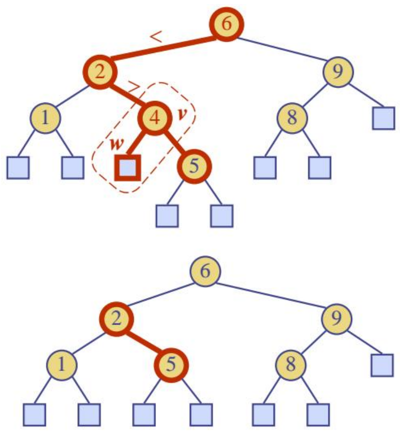


Abbildung 5: Ein Blatt Kind

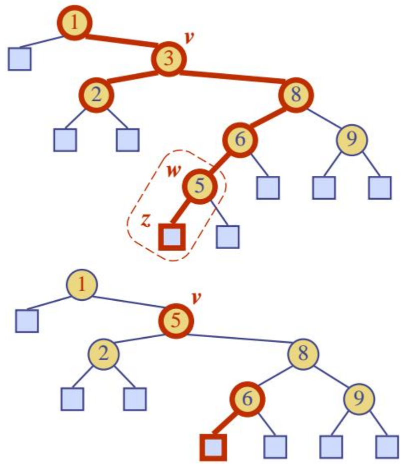


Abbildung 6: Keine Blatt Kinder

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.

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

```java
public class Node {
   private Entry<K, V> entry;
   private Node leftChild;
   private Node rightChild;

   Node(int key) {
      this.key = key;
   }

   public Entry<K, V> setEntry(Entry<K, V> entry) {
      Entry<K, V> oldEntry = entry;
      this.entry = entry;
      return oldEntry;
   }
}
```

Listing 4: BST Entry

```java
public static class Entry<K, V> {
   private K key;
   private V value;

   protected K setKey(K key) {
      K oldKey = this.key;
      this.key = key;
      return oldKey;
   }

   public V setValue(V value) {
      V oldValue = this.value;
      this.value = value;
      return oldValue;
   }
}
```

```java
protected class RemoveResult {
   private Node node;
   private Entry<K, V> entry;
}
```

```java
public class BinarySearchTree<K extends Comparable<? super K>, V> {
   // Root node
   Node root;
   public BinarySearchTree() {
      root = null;
   }
   protected Node newNode(Entry<K, V> entry) {
      return new Node(entry);
   }
   public Entry<K, V> insert(K key, V value) {
      Entry<K, V> newEntry = new Entry<K, V>(key, value);
      root = insert(root, newEntry);
      return newEntry;
   }
   protected Node insert(Node node, Entry<K, V> entry) {
      if (node == null) {
         return newNode(entry);
      } else if (entry.getKey().compareTo(node.getEntry().getKey()) <= 0) {
         node.leftChild = insert(node.leftChild, entry);
      } else if (entry.key > node.key) {
         node.rightChild = insert(node.rightChild, entry);
      }
      return node;
   }
   public Entry<K, V> find(K key) {
      Node result = find(root, key);
      if (result == null) {
         return null;
      } else {
         return result.getEntry();
      }
   }
   protected Node find(Node node, K key) {
      if (node == null) {
         return null;
      }
      if (key.compareTo(node.getEntry().getKey()) < 0) {
         return find(node.leftChild, key);
      }
      if (key.compareTo(node.getEntry().getKey()) > 0) {
         return find(node.rightChild, key);
      }
      return node;
   };
   public Collection<Entry<K, V>> findAll(K key) {
      Collection<Entry<K, V>> entries = new LinkedList<Entry<K, V>>();
      findAll(root, key, entries);
      return entries;
   }

   protected void findAll(Node node, K key, Collection<Entry<K, V>> entries) {
      if (node == null) {
         return;
      }
      if (key.compareTo(node.getEntry().getKey()) == 0) {
         entries.add(node.getEntry());
```

```
        }
        if (key.compareTo(node.getEntry().getKey()) <= 0) {
            findAll(node.leftChild, key, entries);
        }
        if (key.compareTo(node.getEntry().getKey()) >= 0) {
            findAll(node.rightChild, key, entries);
        }
    }

    public Collection<Entry<K, V>> inorder() {
        Collection<Entry<K, V>> coll = new LinkedList<>();
        inorder(root, coll);
        return coll;
    }

    public Collection<Entry<K, V>> inorder() {
        Collection<Entry<K, V>> coll = new LinkedList<>();
        inorder(root, coll);
        return coll;
    }

    protected void inorder(Node node, Collection<Entry<K, V>> coll) {
        if (node == null) {
            return;
        }
        inorder(node.getLeftChild(), coll);
        coll.add(node.getEntry());
        inorder(node.getRightChild(), coll);
    }

    public Entry<K, V> remove(Entry<K, V> entry) {
        if (entry == null) {
            return null;
        }
        RemoveResult result = remove(root, entry);
        root = result.node;
        return result.entry;
    }

    protected RemoveResult remove(final Node node, final Entry<K, V> entry) {
        RemoveResult result = null;
        if (node == null) {
            return new RemoveResult(null, null);
        }
        if (entry.getKey().compareTo(node.getEntry().getKey()) < 0) {
            result = remove(node.leftChild, entry);
            node.leftChild = result.node;
            return result.set(node);
        } else if (entry.getKey().compareTo(node.getEntry().getKey()) > 0) {
            result = remove(node.rightChild, entry);
            node.rightChild = result.node;
            return result.set(node);
        } else {
            // Key found: is this the correct entry?
            if (node.getEntry() != entry) {
                // Searching for next entry with this key
                result = remove(node.leftChild, entry);
                node.leftChild = result.node;
                if (result.entry == null) {
                    result = remove(node.rightChild, entry);
                    node.rightChild = result.node;
                }
```

```
            return result.set(node);
         }
         // We have reachted the correct node.
         if (node.leftChild == null) {
            return new RemoveResult(node.rightChild, node.getEntry());
         }
         if (node.rightChild == null) {
            return new RemoveResult(node.leftChild, node.getEntry());
         }
         Entry<K, V> entryRemoved = node.getEntry();
         Node q = getParentNext(node);
         if (q == node) {
            node.setEntry(node.rightChild.getEntry());
            q.rightChild = q.rightChild.rightChild;
         } else {
            node.setEntry(q.leftChild.getEntry());
            q.leftChild = q.leftChild.rightChild;
         }
         return new RemoveResult(node, entryRemoved);
      }
   }

   protected Node getParentNext(Node p) {
      if (p.rightChild.leftChild != null) {
         p = p.rightChild;
         while (p.leftChild.leftChild != null) {
            p = p.leftChild;
         }
      }
      return p;
   }

   public int getHeight() {
      return getHeight(root);
   }

   protected int getHeight(Node parent) {
      if (parent == null) {
         return -1;
      } else {
         int leftHeight = getHeight(parent.leftChild);
         int rightHeight = getHeight(parent.rightChild);
         return Integer.max(leftHeight, rightHeight);
      }
   }

   public int size() {
      return size(root);
   }

   protected int size(Node n) {
      if (n == null) {
         return 0;
      }
      return size(n.leftChild) + size(n.rightChild) + 1;
   }

   public boolean isEmpty() {
      return size() == 0;
   }
}
```

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.
- 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

1. x ist der gefundene Knoten
2. y ist der parent des gefundenen Knoten
3. z ist der grandparent des gefunden 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

```
protected AVLNode restructure(AVLNode xPos) {
   AVLNode yPos = xPos.getParent();
   AVLNode zPos = yPos.getParent();
   AVLNode newSubTreeRoot = null;
   if (yPos == zPos.getLeftChild()) {
      if (xPos == yPos.getLeftChild()) {
         newSubTreeRoot = rotateWithLeftChild(zPos);
      } else {
         newSubTreeRoot = doubleRotateWithLeftChild(zPos);
      }
   } else {
      if (xPos == yPos.getRightChild()) {
         newSubTreeRoot = rotateWithRightChild(zPos);
      } else {
         newSubTreeRoot = doubleRotateWithRightChild(zPos);
      }
   }
   return newSubTreeRoot;
}
```

6.3.1. Rechts rotieren (einfach)

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

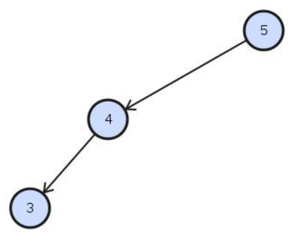


Abbildung 7: Rechts Rotation um c

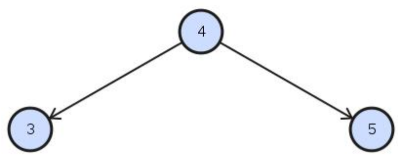


Abbildung 8: Nach der rechts Rotation

Listing 6: AVL Tree: Single right rotation

```
protected AVLNode rotateWithLeftChild(AVLNode k2) {
   AVLNode k1 = k2.getLeftChild();
   k2.setLeftChild(k1.getRightChild());
   k1.setRightChild(k2);

   if (k2.getLeftChild() != null) {
      k2.getLeftChild().setParent(k2);
   }
   adjustParents(k2, k1);

   return k1;
}
```

6.3.2. Links rotieren (einfach)

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

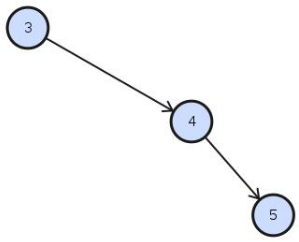


Abbildung 9: Links Rotation um a

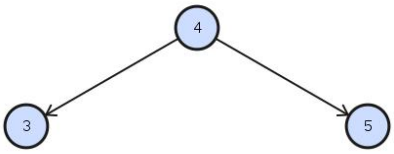


Abbildung 10: Nach der Links Rotation

Listing 7: AVL Tree: Single left rotation

```java
protected AVLNode rotateWithRightChild(AVLNode k1) {
   AVLNode k2 = k1.getRightChild();
   k1.setRightChild(k2.getLeftChild());
   k2.setLeftChild(k1);

   if (k1.getRightChild() != null) {
      k1.getRightChild().setParent(k1);
   }
   adjustParents(k1, k2);

   return k2;
}
```

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)

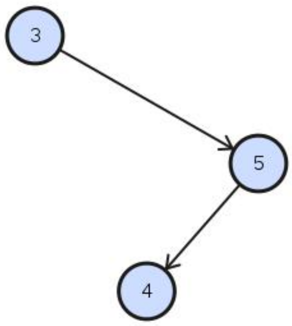


Abbildung 11: Rechts Rotation um b

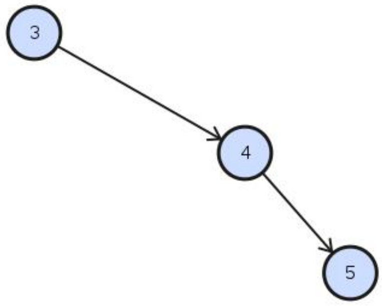


Abbildung 12: Links Rotation um a

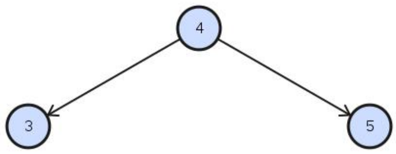


Abbildung 13: Nach Rechts/Links Rotation

Listing 8: AVL Tree: Right/Left Rotation

```
protected AVLNode doubleRotateWithLeftChild(AVLNode k3) {
   AVLNode rotatedRight = rotateWithRightChild(k3.getLeftChild());
   k3.setLeftChild(rotatedRight);
   return rotateWithLeftChild(k3);
}
```

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)

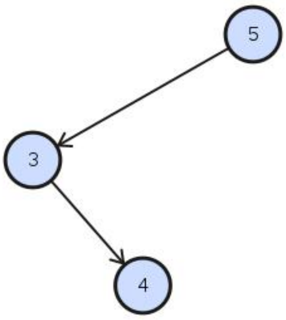


Abbildung 14: Links Rotation um a

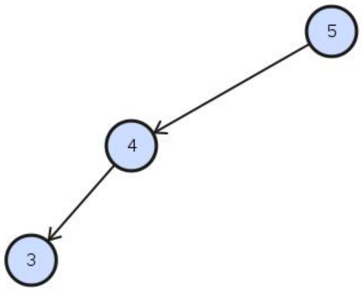


Abbildung 15: Rechts Rotation um c

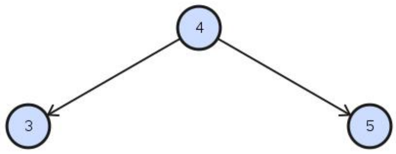


Abbildung 16: Nach Links/Rechts Rotation

Listing 9: AVL Tree: Left/Right Rotation

```java
protected AVLNode doubleRotateWithRightChild(AVLNode k3) {
   AVLNode rotatedLeft = rotateWithLeftChild(k3.getRightChild());
   k3.setRightChild(rotatedLeft);
   return rotateWithRightChild(k3);
}
```

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 Rotation 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 gefunden Knoten und der **Knoten der die AVL Eigenschaft verletzt**!
   d) Für x,y,z die Inorder Reihenfolge erstellen → (a,b,c)
2. Identifiiziere die Subtrees T0, T1, T2, T3 (Inorder Traversierung) von a,b und c
3. Aufschreiben des Inorder Arrays (1-7) → Inorder Reihenfolge

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 T0 als linken und T1 als rechten Unterbaum von a
7. Setze T2 als linken und T3 als rechten Unterbaum von c.

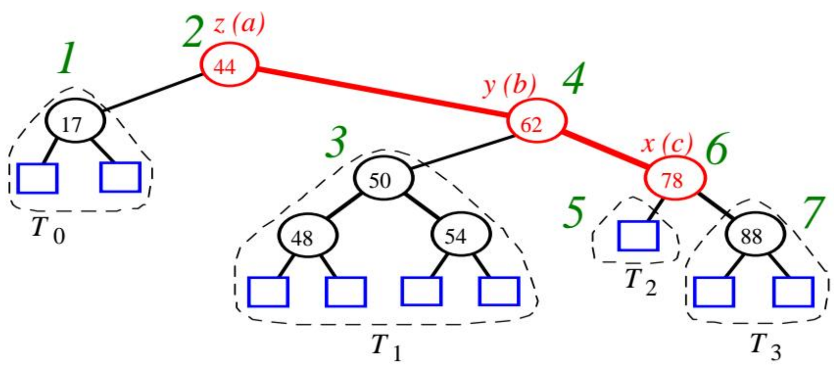


Abbildung 17: Cut/Link Restrukturierung

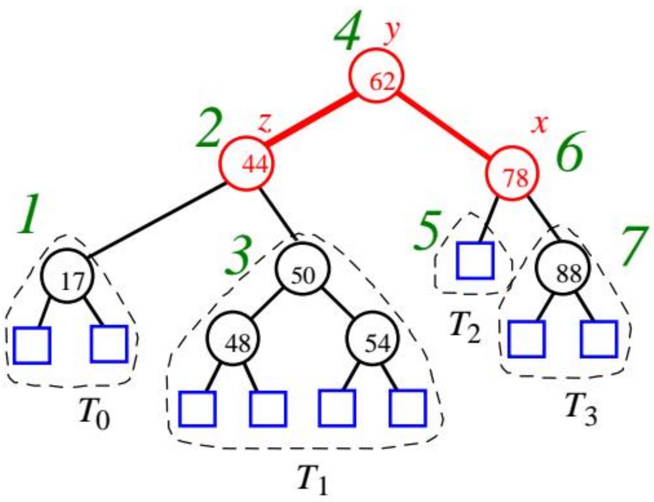


Abbildung 18: Balancierter Baum nach Cut/Link

6.5. Implementierung

Listing 10: AVL Tree Node

```java
protected class AVLNode extends BinarySearchTree<K, V>.Node {
   private int height;
   private Node parent;

   AVLNode(Entry<K, V> entry) {
      super(entry);
   }

   protected AVLNode setParent(AVLNode parent) {
      AVLNode old = avlNode(this.parent);
      this.parent = parent;
      return old;
   }

   protected int setHeight(int height) {
      int old = this.height;
      this.height = height;
      return old;
   }

   protected AVLNode getParent() { .. }
   protected int setHeight() { .. }


   @Override
   public AVLNode getLeftChild() {
      return avlNode(super.getLeftChild());
   }

   @Override
   public AVLNode getRightChild() {
      return avlNode(super.getRightChild());
   }
```

```
}
```

Listing 11: AVL Tree

```
class AVLTreeImpl<K extends Comparable<? super K>, V> extends BinarySearchTree<K, V> {

   protected AVLNode actionNode;

   protected AVLNode getRoot() {
      return avlNode(root);
   }

   protected AVLNode avlNode(Node node) {
      return (AVLNode) node;
   }

   public int getHeight() {
      return height(avlNode(root));
   }

   protected int height(AVLNode node) {
      return (node != null) ? node.getHeight() : -1;
   }

   public V put(K key, V value) {
      Entry<K, V> entry = super.find(key);
      if (entry != null) {
         // key already exists in the Tree
         return entry.setValue(value);
      } else {
         // key does not exist in the Tree yet
         super.insert(key, value);
         rebalance(actionNode);
         actionNode = null;
         return null;
      }
   }

   public V get(K key) {
      Entry<K, V> entry = super.find(key);
      if (entry != null) {
         return entry.getValue();
      } else {
         return null;
      }
   }

   @Override
   protected Node insert(Node node, Entry<K, V> entry) {
      if (node != null) {
         actionNode = avlNode(node);
      }
      // calling now the BST-insert() which will do the work:
      AVLNode result = avlNode(super.insert(node, entry));
      if (node == null) {
         // In this case: result of super.insert() is the new node!
         result.setParent(actionNode);
      }
      return result;
   }

   protected Node newNode(Entry<K, V> entry) {
```

```
      AVLNode avlNode = new AVLNode(entry);
      return avlNode;
   }

   public V remove(K key) {
      Entry<K, V> entry = super.find(key);

      Entry<K, V> toReturn = super.remove(entry);
      if (toReturn == null) {
         AVLNode zPos = actionNode;
         rebalance(zPos);
      }
      return toReturn.getValue();
   }

   protected boolean isBalanced(AVLNode node) {
      int leftHeight = height(node.getLeftChild());
      int rightHeight = height(node.getRightChild());

      int balance = leftHeight - rightHeight;
      return (balance >= -1) && (balance <= 1);
   }

   protected void rebalance(AVLNode node) {
      while (node != null) {
         setHeight(node);
         if (!isBalanced(node)) {
            AVLNode xPos = tallerChild(tallerChild(node));
            node = restructure(xPos);
            setHeight(node.getLeftChild());
            setHeight(node.getRightChild());
            setHeight(node);
         }
         node = node.getParent();
      }
   }

   protected AVLNode tallerChild(AVLNode node) {
      AVLNode leftChild = node.getLeftChild();
      AVLNode rightChild = node.getRightChild();
      if (height(leftChild) >= height(rightChild)) {
         return leftChild;
      } else {
         return rightChild;
      }
   }

   protected void setHeight(AVLNode node) {
      if (node == null) {
         return;
      }

      int heightLeftChild = -1;
      if (node.getLeftChild() != null) {
         heightLeftChild = node.getLeftChild().getHeight();
      }

      int heightRightChild = -1;
      if (node.getRightChild(); != null) {
         heightRightChild = node.getRightChild();.getHeight();
      }

```

```
        node.setHeight(1 + Math.max(heightLeftChild, heightRightChild));
    }

    protected void adjustParents(final AVLNode oldSubtreeRoot, final AVLNode newSubtreeRoot) {
        final AVLNode parentSubtree = oldSubtreeRoot.getParent();
        oldSubtreeRoot.setParent(newSubtreeRoot);
        if (oldSubtreeRoot == root) {
            newSubtreeRoot.setParent(null);
            root = newSubtreeRoot;
        } else {
            newSubtreeRoot.setParent(parentSubtree);
            if (oldSubtreeRoot == parentSubtree.getLeftChild()) {
                parentSubtree.setLeftChild(newSubtreeRoot);
            } else {
                parentSubtree.setRightChild(newSubtreeRoot);
            }
        }
    }

    protected void inorder(Collection<AVLNode> nodeList, AVLNode node) {
    if (node == null)
    return;
    inorder(nodeList, node.getLeftChild());
    nodeList.add(node);
    inorder(nodeList, node.getRightChild());
    }
}
```

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 = links-rechts Grosskind) → **Immer ausgehend von x!**
- Der Knoten x wird nach einem Zugriff zur Root bewegt (nach Update und Suchen)

7.1. Varianten

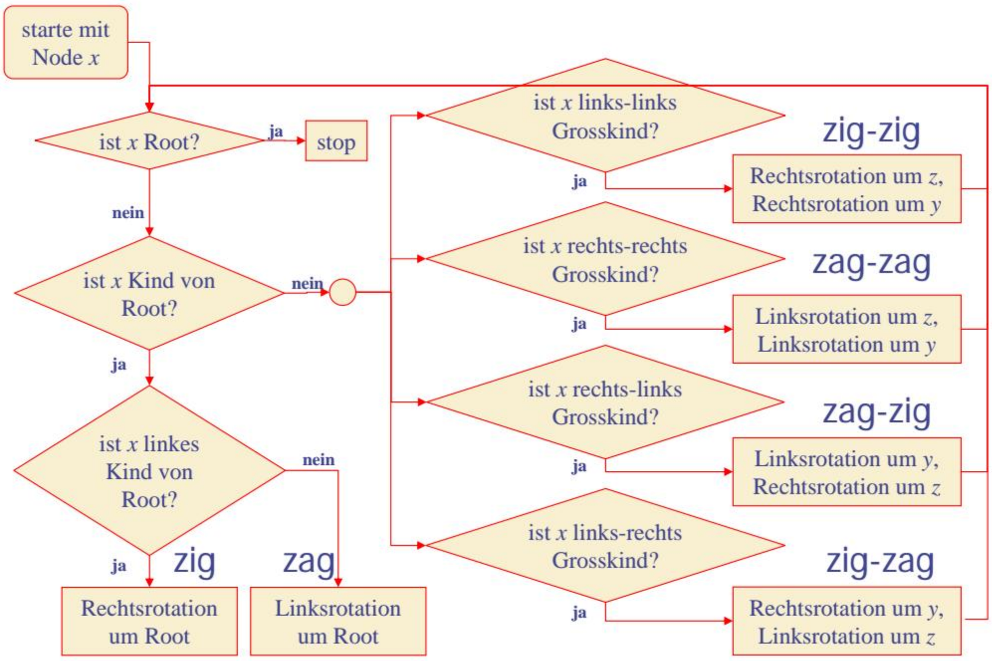


Abbildung 19: Splay Tree Flussdiagramm

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.

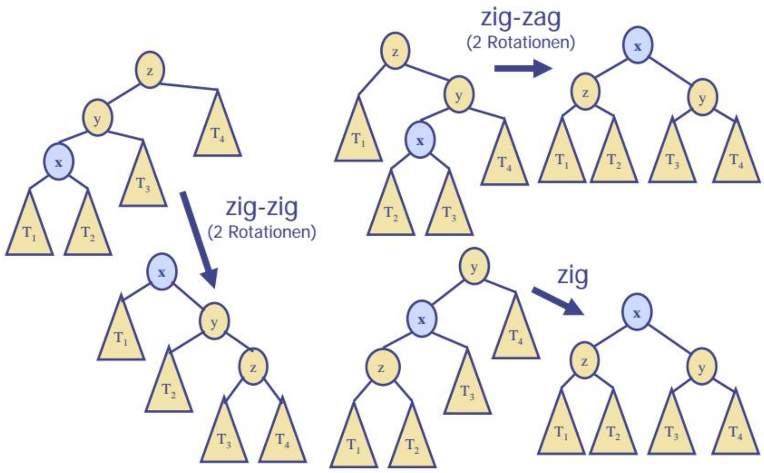


Abbildung 20: Splay Tree Beispiele

7.3. Remove

Beim Löschen wir der zu löschende Knoten mit dem nächsten Knoten in der **Inorder Reihenfolge** ersetzt.Man sieht am folgenden Beispiel, wobei der Wert 8 gelöscht wird.

1. Ersetze zu löschenden Knoten mit seinem Inorder Nachfolder. Dies ist nicht immer möglich, z.B wenn der zu löschende Knoten in externer Knoten ist. In diesem Fall kann dieser Schritt ausgelassen werden
2. Identifiziere x,y,z
   - x = Parent vom löschenden Knoten
   - y = Parent von x
   - z = Parent von z
3. Lösche den zu löschenden Knoten
4. Rotiere gemäss Zig-Zag Schema, damit x in die Root kommt.

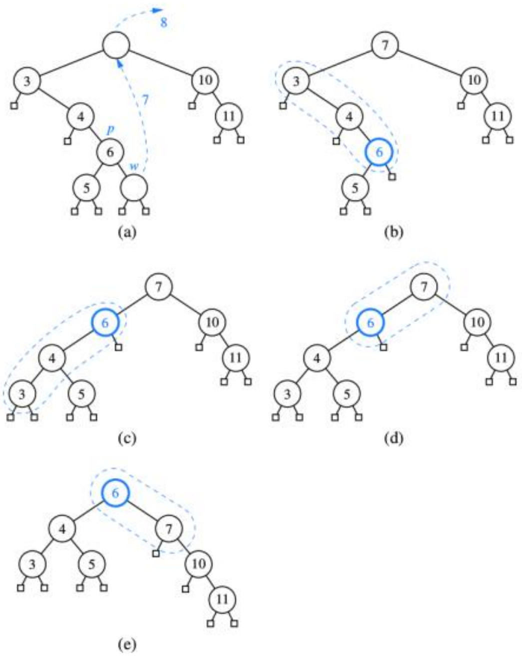


Abbildung 21: Splay Tree: Löschen des Wert 8

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()	$\mathcal{O}(h) \rightarrow \mathcal{O}(n)$	h=Height, Worst Case
	$\mathcal{O}(log(n))$	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 paarweisen 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, **in-place**, für kleine Daten Sets (< 1K)
Insertion Sort	$\mathcal{O}(n^2)$	langsam, **in-place**, **stable**, für kleine Daten Sets (< 1K)
Heap Sort	$\mathcal{O}(n \cdot log(n))$	schnell, **in-place**, für grosse Daten Sets (1K - 1M)
Merge Sort	$\mathcal{O}(n \cdot log(n))$	schnell, **stable** sequentieller Datenzugriff, für riesige Data Sets (> 1M)
Quick Sort	$\mathcal{O}(n \cdot log(n))$	schnellster, **in-place**, (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 $\mathcal{O}(n \cdot \mathcal{S}(n))$, wobei $\mathcal{S}(n)$ der Laufzeit des stabilen Sortieralgorithmus darstellt.

Algorithm 2: lexicographicSort(S)

Data: sequence S of d-Tuples
Result: sequence S sorted in lexicographic order

```
for i ← d downto 1 do
  | stableSort(S, C_i)
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 22: 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 Sequnez (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

```java
public static <T extends Comparable<? super T>> void bubbleSort2(T[] sequence) {
   // performance improvement -> last item is always sorted after iteration
   int max = sequence.length;
   boolean sorted = false;
   do {
      boolean swapped = false;
      for (int i = 1; i < sequence.length; i++) {
         if (sequence[i].compareTo(sequence[i - 1]) < 0) {
            T temp = sequence[i];
            sequence[i] = sequence[i - 1];
            sequence[i - 1] = temp;
            swapped = true;
         }
      }

      max--;

      if (!swapped) {
         sorted = true;
      }

   } while(!sorted);
}
```

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 grösse 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.

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

```java
public static void mergeSort(Object[] orig, Comparator c) {

   Object[] in = new Object[orig.length];

   // make a new temporary array
   System.arraycopy(orig,0,in,0,in.length);

   // copy the input
   Obj ect[] out = new Obj ect[in.length]; // output array
   Obj ect[] temp; // temp array reference used for swapping.
   int n = in.length;

   // each iteration sorts all length-2*i runs
   for (int i=1; i < n; i*=2) {

      // each iteration merges two length-i pair
      for (int j=0; j < n; j +=2*i) {}
         // merge from in to out two length-i runs at j
         merge(in,out,c,j ,i);
      }

      // swap arrays for next iteration
      temp = in;
      in = out;
      out = temp;
   }
   // the "in" array contains the sorted array, so re-copy it
   System.arraycopy(in,0,orig,0,in.length);
}

protected static void merge(Object[] in, Object[] out, Comparator c, int start, int
     inc) {
   // merge in[start..start+inc-1] and
   // in[start+inc..start+2*inc-1]
   int x = start; // index into run # 1
   int end1 = Math.min(start+inc, in.length);

   // boundary for run # 1
   int end2 = Math.min(start+2*inc, in.length);

   // boundary for run # 2
   int y = start+inc;

   // index into run # 2 (could be beyond array boundary)
   int z = start; // index into the out array
   while ((x < end1) && (y < end2)) {
      if (c.compare(in[x],in[y]) <= 0) {
         out[z++] = in[x++];
      } else {
         out[z++] = in[y++];
      }
   }

    // first run didn't finish
   if (x < end1) {
      System.arraycopy(in, x, out, z, end1 - x);
   } else if (y < end2) { // second run didn't finish
      System.arraycopy(in, y, out, z, end2 - y);
   }
}
```

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.

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	$\mathcal{O}(log(n))$		

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

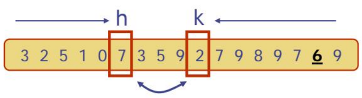


Abbildung 23: InPlace Quicksort

Listing 14: Inplace Quick Sort

```java
public static void quickSort (Object[] S, Comparator c) {
   if (S.length < 2) {
      return; // the array is already sorted in this case
   }
   quickSortStep(S, c, 0, S.length-1); // recursive sort method
}

private static void quickSortStep (Object[] S, Comparator c, int leftBound, int
    rightBound ) {

   if (leftBound >= rightBound) {
      return; // the indices have crossed
   }

   Object temp; // temp object used for swapping
   Object pivot = S[rightBound];
   int leftIndex = leftBound; // will scan rightward
   int rightIndex = rightBound-1; // will scan leftward
   while (leftIndex <= rightIndex) {
      // scan right until larger than the pivot
      while ( (leftIndex <= rightIndex) && (c.compare(S[leftIndex], pivot)<=0) ) {
         leftIndex++;
      }

      // scan leftward to find an element smaller than the pivot
      while ( (rightIndex >= leftIndex) && (c.compare(S[rightIndex], pivot)>=0)) {
         rightIndex--;
      }


      // swap
      if (leftIndex < rightIndex) { // both elements were found
         temp = S[rightIndex];
         S[rightIndex] = S[leftIndex]; // swap these elements
         S[leftIndex] = temp;
      }
   } // the loop continues until the indices cross

   temp = S[rightBound]; // swap pivot with the element at leftIndex
   S[rightBound] = S[leftIndex];
   S[leftIndex] = temp; // the pivot is now at leftIndex, so recurse

   // step over equal elements to speed up
   while ((leftIndex > leftBound) && (comp.compare(sequence[leftIndex],
       sequence[leftIndex - 1]) == 0)) {
      leftIndex--;
   }
   while ((rightIndex > 0) && (rightIndex < rightBound) &&
       (comp.compare(sequence[rightIndex], sequence[rightIndex + 1]) == 0)) {
      rightIndex++;
   }

   // recursive call
   quickSortStep(S, c, leftBound, leftIndex-1);
   quickSortStep(S, c, leftIndex+1, rightBound);
}
```

12. Bucket Sort

- Bucket Sort funktioniert nur mit positven 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

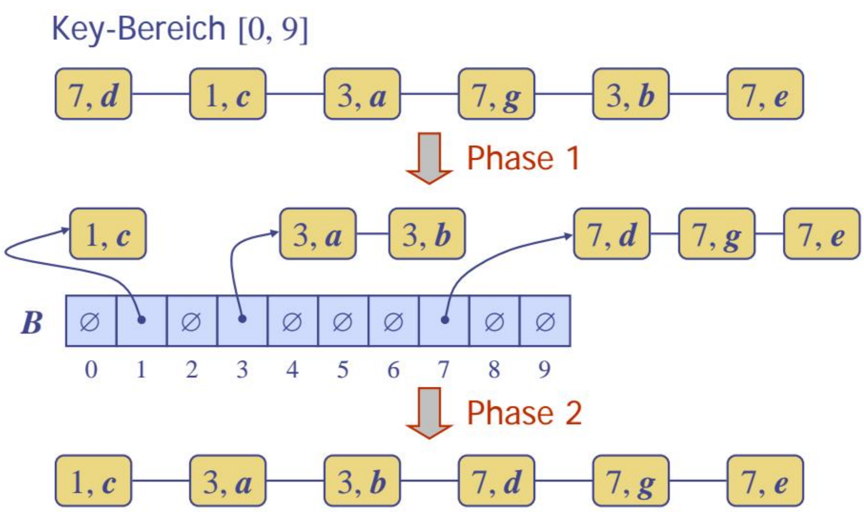


Abbildung 24: Bucket Sort

12.1. Laufzeiten

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

Tabelle 14: Big Oh Bucket Sort

12.2. Implementierung

```java
public static void sort(int[] a, int maxVal) {
   // fail fast
   if (maxVal <= 0) {
      throw new IllegalArgumentException();
   }
   if (a.length <= 1) {
      return a; //trivially sorted
   }

   int[] bucket= new int[maxVal + 1];

   for (int i=0; i<bucket.length; i++) {
      bucket[i]=0;
   }

   // Phase 1:
   for (int i=0; i<a.length; i++) {
      bucket[a[i]]++;
   }

   // Phase 2:
   int outPos=0;
   for (int i=0; i < bucket.length; i++) {
      for (int j=0; j<bucket[i]; j++) {
         a[outPos++]=i;
      }
   }
}
```

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. Beispiel

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

```
// 1. partition: order by last digit
|0| |1| |2| |3| |4| |5| |6| |7| |8| |9|
             |   |               |
           523 124             128
           483 584
           923
// 2. collect: 523, 483, 923, 124, 584, 128
// 3. partition: order by second digit
|0| |1| |2| |3| |4| |5| |6| |7| |8| |9|
         |                       |
        523                     483
        923                     584
        124
        128

// 4. collect: 523, 923, 124, 128, 483, 584
// 5. partition: order by first digit
|0| |1| |2| |3| |4| |5| |6| |7| |8| |9|
     |           |   |               |
     124         483 523            923
     128             584

// 6. collect:
124, 128, 483, 523, 584, 923
```

13.3. Implementierung

```java
public class RadixSort {
   // array of linked lists
   private final LinkedList<String>[] buckets;

   public RadixSort() {
      // create LinkedList for buckets
      buckets = (LinkedList<String>[]) new LinkedList<?>[1 + ('z' - 'a' + 1)];
      IntStream.range(0, buckets.length).forEach(
         i -> buckets[i] = new LinkedList<String>()
      );
   }

   public void radixSort(String[] data) {

      // find max index
      AtomicInteger maxLength = new AtomicInteger(-1);
      Arrays.stream(data).forEach(str -> {
         if (str.length() > maxLength.intValue()) {
            maxLength.set(str.length());
         }
      });

      // bucketsort from max index to first index
      for (int i = maxLength.get() - 1; i >= 0; i--) {
         bucketSort(data, i);
      }

   }

   protected void bucketSort(String[] data, int index) {

      // clear buckets
      Arrays.stream(buckets).forEach(list -> list.clear());

      // insert data elements to buckets
      Arrays.stream(data).forEach(str -> {
         if (str.length() <= index) {
            buckets[0].addLast(str);
         } else {
            buckets[str.charAt(index) - 'a' + 1].addLast(str);
         }
      });

      // shift bucket elements back into data array
      AtomicInteger i = new AtomicInteger(0);
      Arrays.stream(buckets).forEach(list -> list.forEach(str -> {
         data[i.getAndIncrement()] = str;
      }));
   }
}
```

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 Σ

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	$\mathcal{O}(n \cdot m)$	
Boyer-Moore Algorithmus	$\mathcal{O}(n \cdot m + s)$	Ist der schnellste (obschon schlechter 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

```
for i ← 0 to n - m do
    j ← 0
    while j < m ∧ T[i + j] = P[j] do
        j ← j + 1
    end
    if j = m then
        return i
    end
end
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 Occurence Funktion

Man erstellt eine Hashmap für alle Zeichen des Alphabets Σ 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 Missmatch 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)

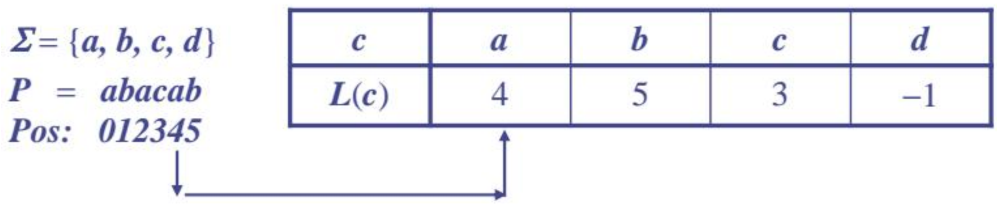


Abbildung 25: Boyer Moore Last Occurence

14.3.2. Vorgehen

1. Erstelle die Last Occurence Funktion
2. Starte am Ende des Patterns und Vergleiche das Pattern mit dem Text (**Rechts nach Links**)
3. Bei einem Missmatch, 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 um die Länge des Patttern.

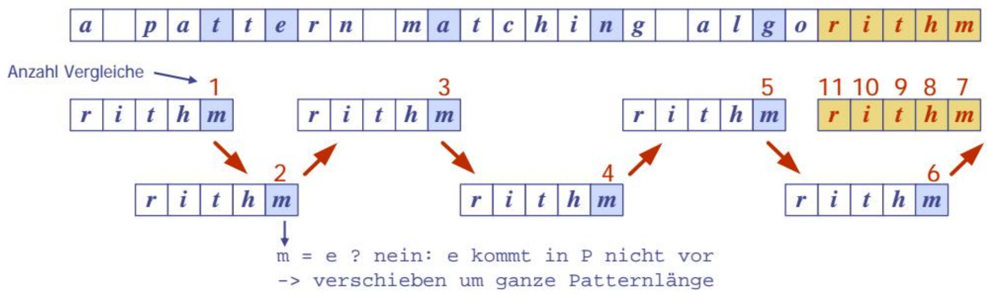


Abbildung 26: Boyer Moore Algorithmus

14.3.3. Implementierung

Gegeben ist der Text T, das Pattern P und das Alphabet.

Algorithm 4: BoyerMooreMatch(T, P, Σ)

```
L ← lastOccurenceFunction(P, Σ)
i ← m − 1 // Index in T
j ← m − 1 // Index in P
repeat
    if T[i] = P[j] then
        if j=0 then
            return i // match
        else
            i ← i − 1
            j ← j − 1
        end
    else
        l ← L[T[i]]
        i ← i + m − min(j, 1 + l)
        j ← m − 1
    end
until i > n -1;
return-1
```

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.

a	*b*	*a*	*a*	*b*	*a*	*a*	*b*	*a*

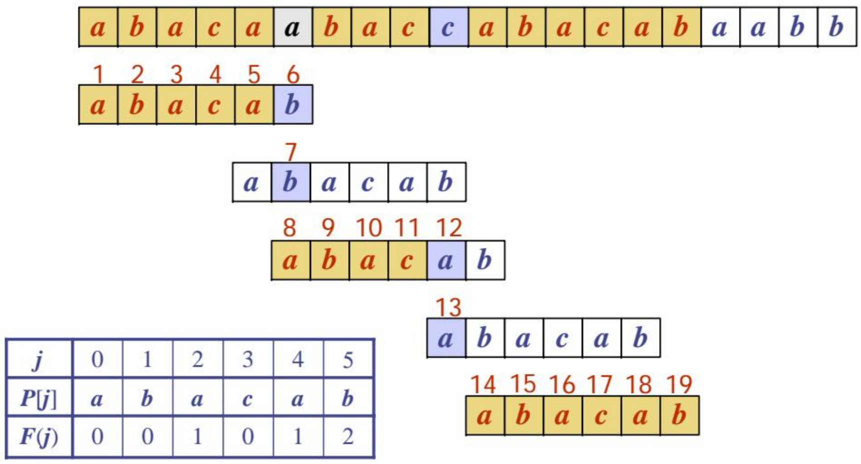
j	0	1	2	3	4	5	6	7	8
$P[j]$	*a*	*b*	*a*	*a*	*b*	*a*	*a*	*b*	*a*
$F(j)$	0	0	1	1	2	3	4	5	6

Ränder-Länge:	0	1	2	3	4	5	6	max.Länge
P[0]=a	{}							0
P[1]=ab	{}							0
P[2]=aba	{}	a						1
P[3]=abaa	{}	a						1
P[4]=abaab	{}		ab					2
P[5]=abaaba	{}	a		aba				3
P[6]=abaabaa	{}	a			abaa			4
P[7]=abaabaab	{}		ab			abaab		5
P[8]=abaabaaba	{}	a		aba			abaaba	6

Abbildung 27: 1. Fehlfunktion aufbauen

14.4.2. Vorgehen

1. Gehe von **Links nach Rechts**
2. Suche den ersten Mismatch (j=5)
3. Übergib die Stelle des **Zeichens davor** (j = (5 - 1) = 4) der Fehlfunktion F(4)
4. Der Rückgabewert der Fehlfunktion (=1) ist dann das Zeichen (j = 1 → zweites b), welches bis zur Position des Missmatch vorgeschoben werden kann.
5. Ist der Rückgabewert = 0 wird das Pattern einfach um 1 verschoben



j	0	1	2	3	4	5
P[*j*]	*a*	*b*	*a*	*c*	*a*	*b*
F(*j*)	0	0	1	0	1	2

Abbildung 28: 2. Knuth-Morris-Pratt Algorithm

14.4.3. Implementierung

Listing 15: Knuth-Morris-Pratt Algorithmus

```
public static int KMPmatch(String text, String pattern) {
   int n = text.length();
   int m = pattern.length();
   int[] fail = computeFailFunction(pattern);
   printFail(fail);
   int i = 0;
   int j = 0;
   while (i < n) {
   System.out.print("\ni = " + i + " j = " + j);
      if (pattern.charAt(j) == text.charAt(i)) {
         if (j == m - 1) {
               return i - m + 1; // match
         }
         i++;
         j++;
      } else if (j > 0) {
         j = fail[j - 1];
      } else {
         i++;
      }
   }
   return -1; // no match
}
```

Listing 16: Knuth-Morris-Pratt Algorithmus Fehlfunktion

```
public static int[] computeFailFunction(String pattern) {
   int[] fail = new int[pattern.length()];
   fail[0] = 0;
   int m = pattern.length();
   int j = 0;
   int i = 1;
   while (i < m) {
      if (pattern.charAt(j) == pattern.charAt(i)) {
         // j + 1 characters match
         fail[i] = j + 1;
         i++;
         j++;
      } else if (j > 0) {
         // j follows a matching prefix
         j = fail[j - 1];
      } else {
         // no match
         fail[i] = 0;
         i++;
      }
   }
   return fail;
}
```

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 zun 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 assozierten Wortes.

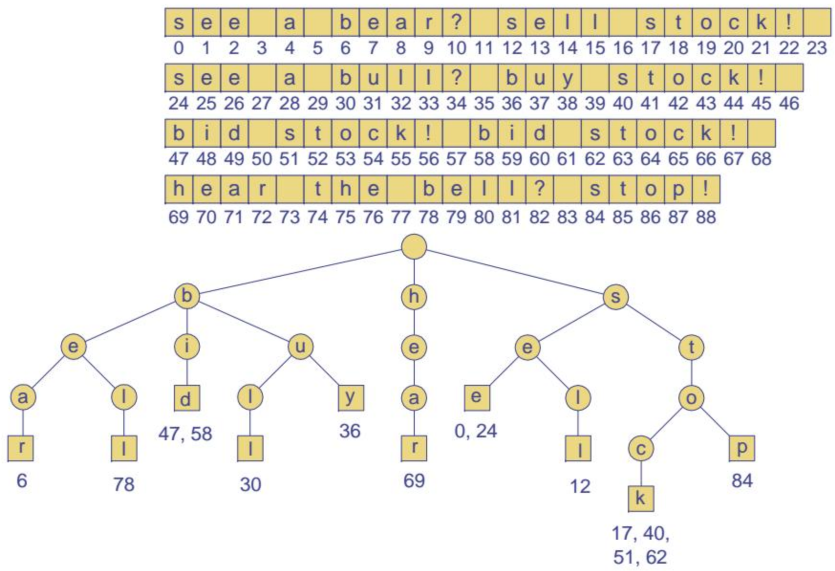


Abbildung 29: Trie Beispiel

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] = "stock"` → Knoten = (3,1,2) → `"to"`

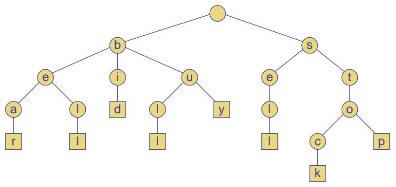


Abbildung 30: Trie Ausgangslage

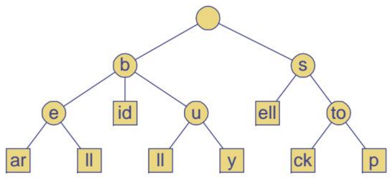


Abbildung 31: Trie nach Kompression

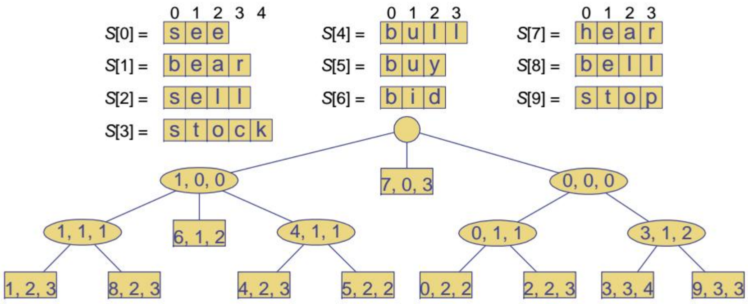


Abbildung 32: Kompakte Repräsentation eines komprimierten Tries

15.3. Suffix Trie

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

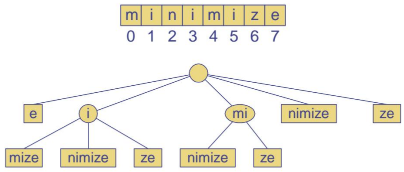


Abbildung 33: Suffix Trie

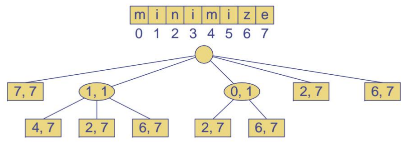


Abbildung 34: Suffix Trie with Index Representation

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

```java
public class TrieMultimap<V> implements Multimap<String, V> {

   private TrieNode<V> root;

   private enum Mutation {
      INSERT, REMOVE
   };

   public TrieMultimap() {
      this.root = new TrieNode<V>();
   }

   // Returns the first value for a given key. null if key is not found
   public V find(String key) {
      TrieNode<V> result = find(root, key);
      if (result != null)
      return result.getValues().get(0);
      else
      return null;
   }

   // return Iterator over all values. If key is not found: Iterator without next
   public Iterator<V> findAll(String key) {
      TrieNode<V> result = find(root, key);
      if (result != null)
```

```
            return result.getValues().iterator();
        else
            return new LinkedList<V>().iterator();
    }

    private TrieNode<V> find(TrieNode<V> node, String keySubstr) {
        if (keySubstr.length() == 0) {
            return node;
        }
        for (TrieNode<V> child : node.getChilds()) {
            if (keySubstr.startsWith(child.getKeySubstr())) {
                keySubstr = keySubstr.substring(child.getKeySubstr().length());
                return find(child, keySubstr);
            }
        }
        return null;
    }

    // Inserts a key/value pair into the multimap.
    public void insert(String key, V value) {
        TrieNode<V> result = find(root, key);
        if (result != null) {
            result.getValues().add(value);
        } else {
            mutate(Mutation.INSERT, root, key, 0, value);
        }
     }

     private boolean mutate(Mutation operation, TrieNode<V> node, String key, int
         keyIndex, V value) {
        if (key.length() == keyIndex) {
            // found the node!
            if (operation == Mutation.INSERT) {
                node.getValues().add(value);
            } else { // REMOVE
                node.getValues().clear();
            }
            return true;
        }
        for (TrieNode<V> child : node.getChilds()) {
            if (child.getKeySubstr().charAt(0) == key.charAt(keyIndex)) {
                if (child.getKeySubstr().length() > 1) { // a compressed node?
                    child = decompress(node, child);
                }
                boolean result = mutate(operation, child, key, ++keyIndex, value);
                compress(node, child);
                return result;
            }
         }

         // there is no corresponding child:
         if (operation == Mutation.INSERT) {
            TrieNode<V> newNode = new TrieNode<>();
            newNode.setKeySubstr(key.substring(keyIndex, keyIndex + 1));
            node.getChilds().add(newNode);
            mutate(Mutation.INSERT, newNode, key, ++keyIndex, value);
            compress(node, newNode);
         } else { // REMOVE
         return false; // not found
        }
        return false;
    }
```

```

    private TrieNode<V> decompress(TrieNode<V> node, TrieNode<V> child) {
       // insert an additional, single-char node (de-compressing):
       TrieNode<V> newChild = new TrieNode<>();
       newChild.setKeySubstr(child.getKeySubstr().substring(0, 1));
       child.setKeySubstr(child.getKeySubstr().substring(1));
       newChild.getChilds().add(child);
       node.getChilds().add(newChild);
       node.getChilds().remove(child);
       return newChild;
    }

    private void compress(TrieNode<V> node, TrieNode<V> child) {
       if ((child != root) && (child.getChilds().size() == 1)
       && (child.getValues().isEmpty())) {
          // compress:
          TrieNode<V> childOfChild = child.getChilds().get(0);
          child.setKeySubstr(child.getKeySubstr().concat(childOfChild.getKeySubstr()));
          child.getValues().addAll(childOfChild.getValues());
          child.getChilds().addAll(childOfChild.getChilds());
          child.getChilds().remove(childOfChild);
          return;
       }
       if (child.getChilds().isEmpty() && (child.getValues().isEmpty())) {
          // this is a removed node:
          node.getChilds().remove(child);
          return;
       }
    }

    // Removes all values for a given key.
    public void remove(String key) {
       TrieNode<V> result = find(root, key);
       if (result != null) {
          mutate(Mutation.REMOVE, root, key, 0, null);
       } else {
       return;
       }
    }

    public int size() {
       return size(root);
    }

    // return Number of values in this node and its child nodes.
    private int size(TrieNode<V> element) {
       int size = 0;
       for (TrieNode<V> child : element.getChilds()) {
          size += size(child);
       }
       size += element.getValues().size();
       return size;
    }
}
```

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 **hinzugefügt 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)

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 35: Dynamische Programmierung, Rucksackproblem

16.2. 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.

16.3. 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/links 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
	-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**★

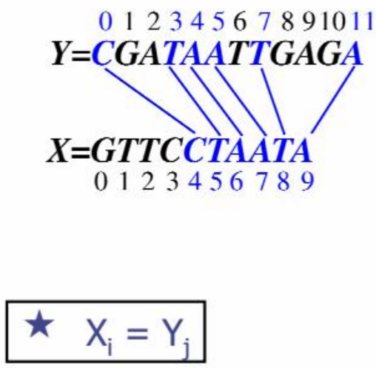


Abbildung 36: Longest Common Subsequence

16.3.1. Implementierung

```java
public class LCS {
   private int tableL[][]; // data array
   private String xStr;
   private String yStr;

   public int[][] calculateTable(final String xStr, final String yStr) {

      this.xStr = xStr;
      this.yStr = yStr;

      int n = xStr.length();
      int m = yStr.length();

      // +1 because of zero row/column
      tableL = new int[n + 1][m + 1];

      for (int i = 1; i <= n; i++) {
         for (int j = 1; j <= m; j++) {
            // - 1 because of the zero row/column
            if (xStr.charAt(i - 1) == yStr.charAt(j - 1)) {
               tableL[i][j] = tableL[i - 1][j - 1] + 1;
            } else {
               tableL[i][j] = Math.max(tableL[i - 1][j], tableL[i][j-1]);
            }
         }
      }

      return tableL;
   }

   public List<String> findAll() {
      List<String> list = new LinkedList<>();
      Deque<Character> stack = new LinkedList<>();
      find(xStr.length(), yStr.length(), stack, list);
      List<String> result = new LinkedList<>();
      list.stream().sorted().distinct().forEach(str -> result.add(str));
      return result;
   }

   private void find(final int xPos, final int yPos, Deque<Character> stack,
       List<String> stringList) {
      if ((xPos == 0) || (yPos == 0)) { // reached the end?
         stringList.add(stack.toString());
         return;
      } else {
         if (xStr.charAt(xPos - 1) == yStr.charAt(yPos - 1)) {
            stack.push(xStr.charAt(xPos - 1));
            find(xPos - 1, yPos - 1, stack, stringList);
            stack.pop();
         }
         if (tableL[xPos - 1][yPos] == tableL[xPos][yPos]) {
            find(xPos - 1, yPos, stack, stringList);
         }
         if (tableL[xPos][yPos - 1] == tableL[xPos][yPos]) {
            find(xPos, yPos - 1, stack, stringList);
         }
      }
   }
}
```

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 Ein Knoten des Edges ist der Ursprung und der andere das Ziel. Die Kante wird als Pfeil dargestelt

ungerichtete Kanten Ein ungeorneter Knoten Paar

gerichteter Graph Alle vorhandenen Edges sind gerichet

ungerichteter Graph Alle vorhandenen Edges sind ungerichtet

End-Vertizes Endpunkt einer Kante. Eine Kante ist `inzident` in einem Knoten (enden)

Adjazente Benachbarte Knoten sind adjazent

Grad eines Vertex Anzahl inzidenter Kanten. Wie viele Kanten mit einem anderen Knoten verbunden sind.

Parallele Kanten Zwei Kanten zwischen zwei Knoten, die eine Schleife bilden

Schleife Eine Kante mit dem gleichen Ursprung und Ziel

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

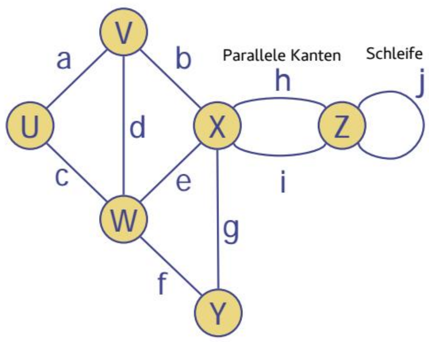


Abbildung 37: Parallele Kanten und Schleifen

17.1.1. Subgraphen

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

Aufspannender Subgraph Ein aufspannender Subgraph enthält alle Vertizes des Graphen, jedoch nicht alle Kanten.

17.1.2. Tree und Forest

Tree Ist ein Graph der connected ist und keine Zyklen aufweist

Forest Ist ein ungerichter 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)

Zyklus Ended mit einer Kante. Ein Zyklus verbindet implizit den letzten Vertex mit dem ersten. Ein einfacher Zyklus besucht nie zweimal den gleichen Vertex. (Einfacher Zyklus = rot, Nicht einfacher Zyklus = grün)

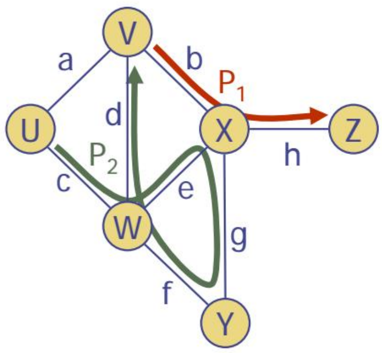


Abbildung 38: Pfad

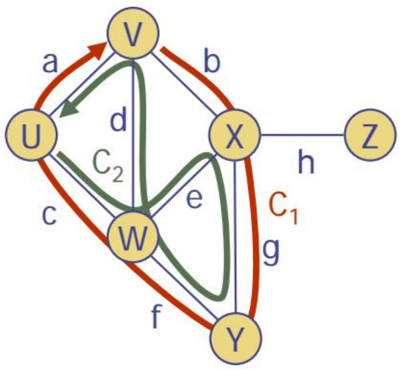


Abbildung 39: Zyklus

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 Liste von Kanten nach dem gesuchten Vertext durchsucht werden.
- Die Kanten Listen Sturktur **wird selten verwendet**

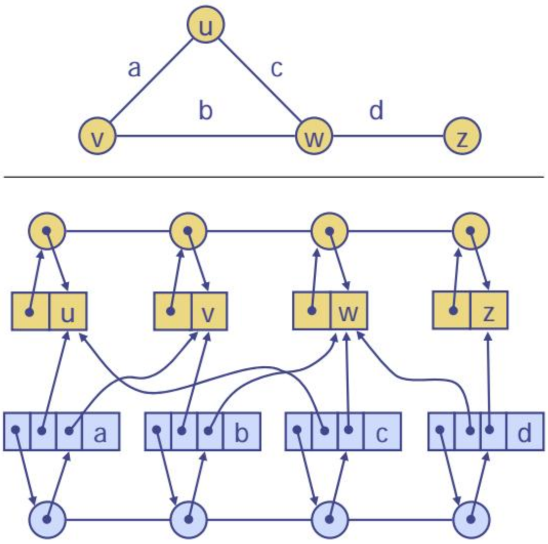


Abbildung 40: Kanten-Listen Struktur

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
- **Benötig sehr wenig Platz**, auch bei vielen Kanten/grossen Graphen.

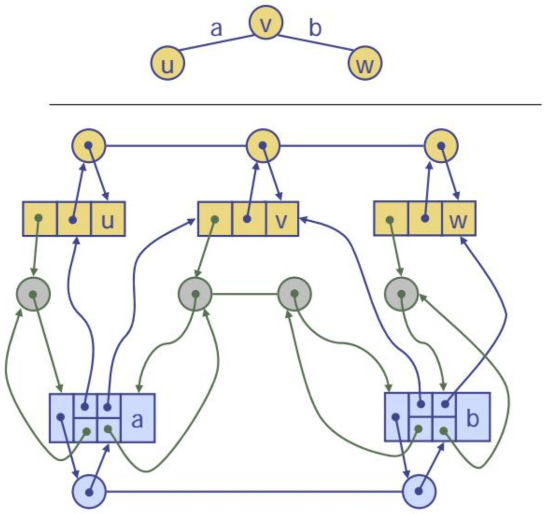


Abbildung 41: Adjazenz Listen Struktur

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

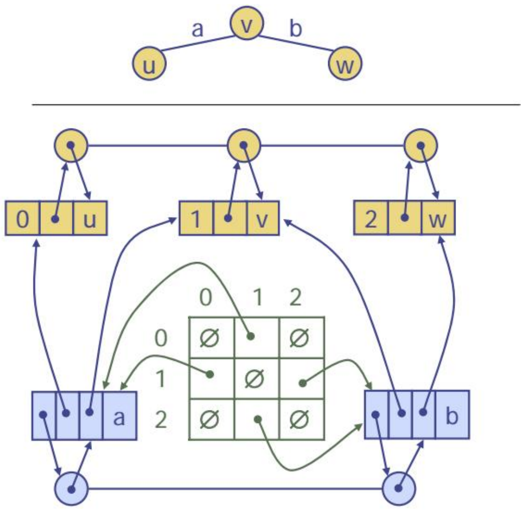


Abbildung 42: Adjazenz-Matrix Struktur

17.5. Laufzeiten

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

Operation	Kanten Liste	Adjazenz Liste	Adjazenz Matrix
Space	$\mathcal{O}(n+m)$	$\mathcal{O}(n+m)$	$\mathcal{O}(n^2)$
incidentEdges(v)	$\mathcal{O}(m)$	$\mathcal{O}(dev(v))$	$\mathcal{O}(n)$
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

```java
// Graph
public class Graph<T extends Comparable<T>> extends Observable {

   private ArrayList<Node<T>> nodes;
   private Stack<Node<T>> stack;

   public Graph() {
      nodes = new ArrayList<Node<T>>();
      stack = new Stack<Node<T>>();
   }

   public Node<T> getNode(int indx) {
      return nodes.get(indx);
   }

   public void addNode(Node<T> n) {
      if (nodes.contains(n)) {
         return; // list is a set !
      }
      nodes.add(n);
   }

   // DFS
   public ArrayList<Node<T>> depthFirstSearch(Node<T> from, Node<T> to) {
      from.setMark(true);
      stack.push(from);
      if (from == to) {
         return new ArrayList<Node<T>>(stack);
      }
      for (Node<T> n : from.getConnectedNodes()) {
         System.out.println(from.getObject() + ": " + n.getObject() + " ");
```

```
         if (!n.isMarked()) {
            ArrayList<Node<T>> path = depthFirstSearch(n, to);
            if (path != null) {
               return path;
            }
         }
      }
      stack.pop();
      return null;
   }

   // BFS
   public ArrayList<Node<T>> breadthFirstSearch(Node<T> from, Node<T> to) {
      ArrayList<IntermediatePath<T>> queue = new ArrayList<>();
      IntermediatePath<T> ip = new IntermediatePath<>(null, from);
      queue.add(ip);
      from.setMark(true);
      while (queue.size() > 0) {
         ip = queue.remove(0);
         if (ip.current == to) {
            ArrayList<Node<T>> path = new ArrayList<>();
            do {
               path.add(0, ip.current);
            } while ((ip = ip.previous) != null);
            return path;
         }
         for (Node<T> it : ip.current.getConnectedNodes()) {
            if (!it.isMarked()) {
               it.setMark(true);
               IntermediatePath<T> newIP = new IntermediatePath<T>(ip, it);
               queue.add(newIP);
               System.out.print(" previous: " + newIP.previous.current.getObject()
               + " current: " + newIP.current.getObject());
            }
         }
      }
      return null;
   }
}
```

```
// Node
public class Node<T extends Comparable<T>> {

   // Connected neighbour nodes
   private ArrayList<Node<T>> linked;
   private T obj;

   public Node(T obj) {
      this.obj = obj;
      linked = new ArrayList<Node<T>>();
   }

   public ArrayList<Node<T>> getConnectedNodes() {
      return linked;
   }

   public void connectTo(Node<T> n) {
      ListIterator<Node<T>> it = linked.listIterator();
      while(it.hasNext()) {
         Node<T> listNode = it.next();
         int compareResult = n.getObject().compareTo(listNode.getObject());
```

```
            if (compareResult == 0) {
                return; // list is a set!
            } else if (compareResult < 0) {
                it.previous();
                it.add(n);
                return;
            }
        }
        // Node has not yet been inserted
        linked.add(n);
    }

}
```

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

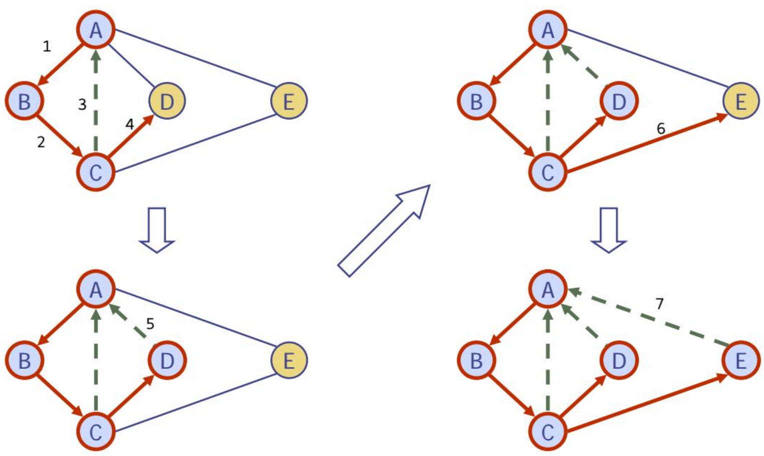


Abbildung 43: Depth First Search

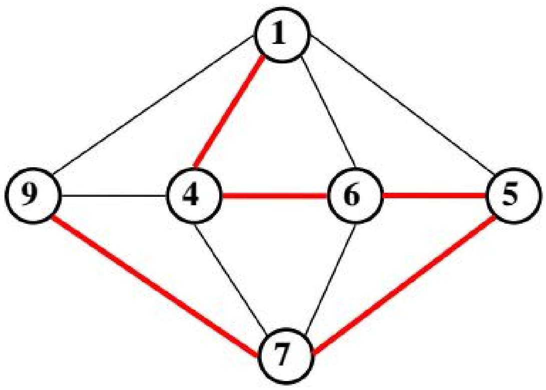


Abbildung 44: Tiefensuche

Algorithm 5: DFS(G)

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

forall *u in G.vertices()* **do**
  $setLabel(u, UNEXPLORED)$
end
forall *e in G.edges()* **do**
  $setLabel(e, UNEXPLORED)$
end
forall *v in G.vertices()* **do**
  if $getLabel(v) == UNEXPLORED$ **then**
    $DFS(G, v)$
  end
  else
    `// not conencted`
  end
end

Algorithm 6: DFS(G,v)

Data: Graph G and a Start vertex v of G

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

```
setLabel(v, VISITED)
forall e in G.incidentEdges(v) do
    if getLabel(e) == UNEXPLORED then
        w ← opposite(v, e)
        if getLabel(w) == UNEXPLORED then
            setLabel(e, DISCOVERY)
            DFS(G, W)
        end
        else
            setLabel(e, BACK)
        end
    end
end
```

18.2. 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 gezwungermassen 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**.

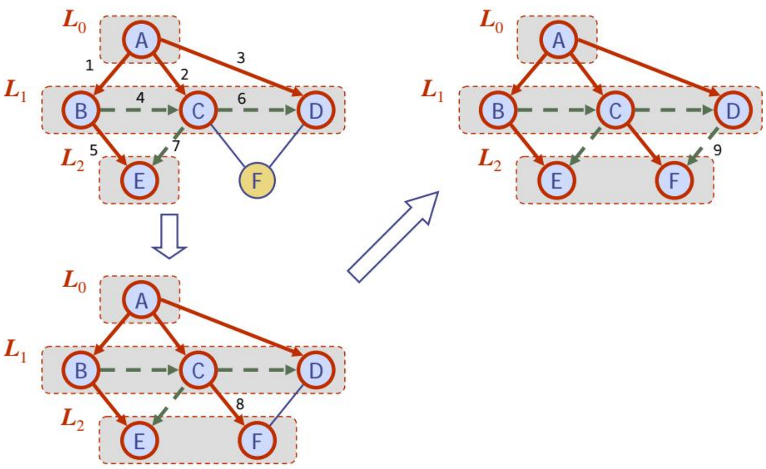


Abbildung 45: Breath First Search

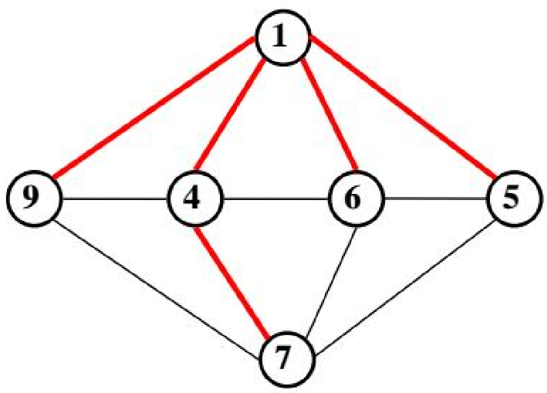


Abbildung 46: Breitensuche

Algorithm 7: BFS(G,s)

```
L_0 ← newemptysequence
L_0.insertLast(s)
setLabel(s, VISITED)
i ← 0
while ¬L_i.isEmpty do
    forall v in L_i.elements() do
        forall e in G.incidentEdges(v) do
            if getLabel(e) == UNEXPLORED then
                w ← opposite(v, e)
                if getLabel(w) == UNEXPLORED then
                    setLabel(e, DISCOVERY)
                    setLabel(w, VISITED)
                    L_{i+1}.insertLast(w)
                end
                else
                    setLabel(e, BACK)
                end
            end
        end
    end
end
```

18.3. 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

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

19.1. 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

19.2. 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 Algoritmus 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`.

19.3. 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)

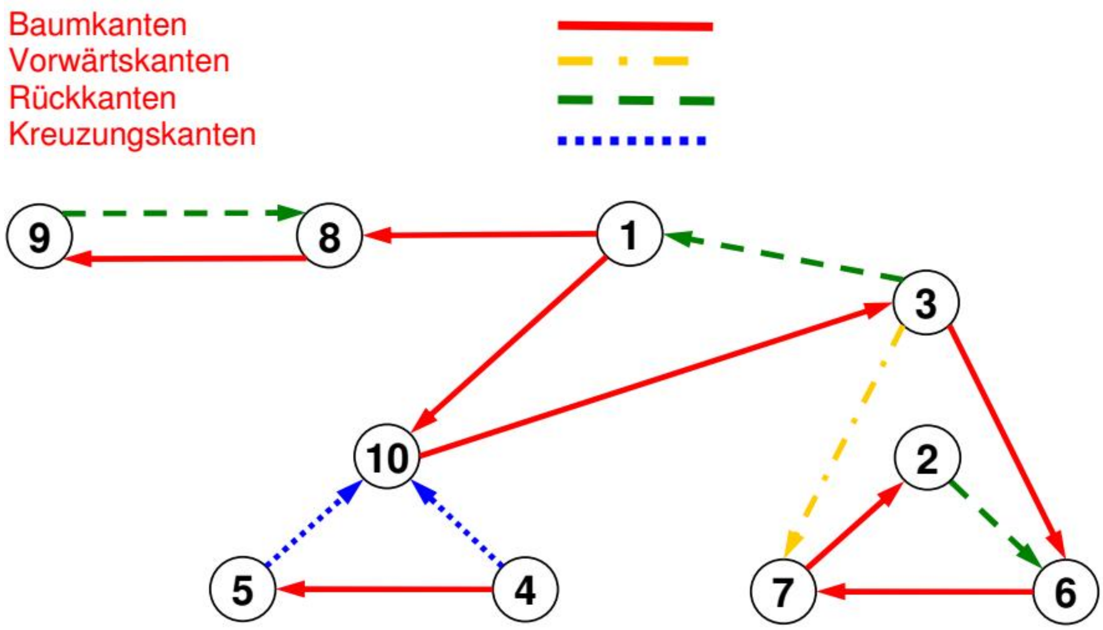


Abbildung 47: Digraph DFS

19.4. 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 Transitive Abschluss den direkten Weg zwischen zwei Vertex.

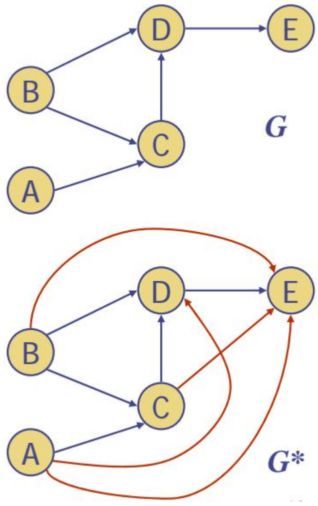


Abbildung 48: Transitive Abschluss

19.5. 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 49) In einem transitiven Abschluss sind alle Knoten direkt mit einander verbunden, welche so oder so über andere Knoten erreicht hätten werden können.

19.5.1. 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 $vi + 1$ und wiederholt die Prozedur.
4. Die neu gezeichneten Edges **bleiben für die folgenden Schritte bestehenden** und müssen ebenfalls beachtet werden (Dynamische Programmierung)

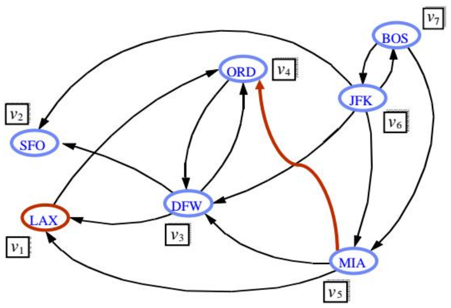


Abbildung 49: Schritt 1

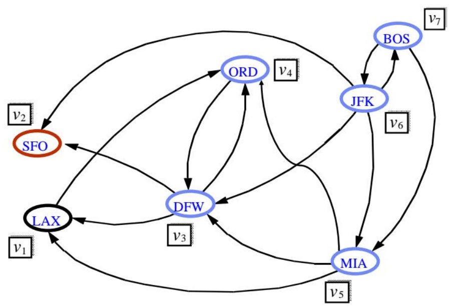


Abbildung 50: Schritt 2

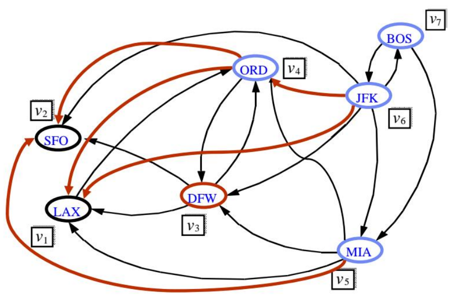


Abbildung 51: Schritt 3

Algorithm 8: FloydWarshall(G)

```
Data: digraph G
Result: transitive closure G* of G
i ← 1
forall v in G.vertices() do
    denote v as v_i
    i ← i + 1
end
G_0 ← G
for k ← 1 to n do
    G_k ← G_{k−1}
    for i ← 1 to n(i ≠ k) do
        for j ← 1 to n(j ≠ i, k) do
            if G_{k−1}.areAdjacent(v_i, v_k) ∧ G_{k−1}.areAdjacent(v_k, v_j) then
                if ¬G_k.areAdjacent(v_i, v_j) then
                    G_k.insertDirectedEdge(v_i, v_j, k)
                end
            end
        end
    end
end
return G_n
```

19.6. DAG: Directed Acyclic Graph

Ein DAG **enthält keine gerichteten Zyklen**.

19.7. Topolgische 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 9: TopologicalSort(G)

$H \leftarrow G$
$n \leftarrow G.numVertices()$
while $H is not empty$ **do**
  Let v be a vertex with no outgoing edges
  Label $v \leftarrow n$
  $n \leftarrow n - 1$
  Remove v from H
end

19.7.1. 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)

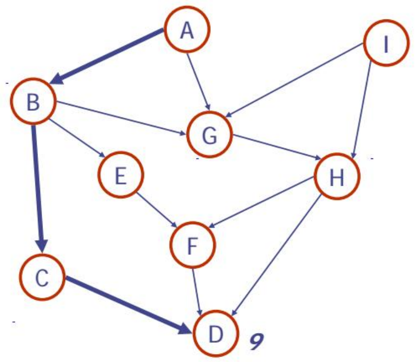


Abbildung 52: Schritt 1

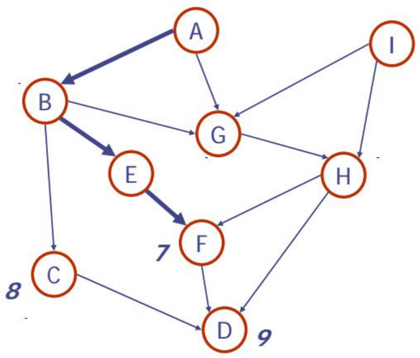


Abbildung 53: Nach dem ersten Backtracking

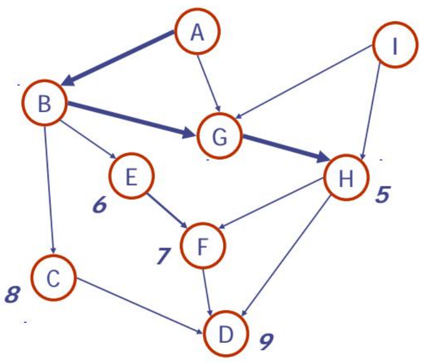


Abbildung 54: Nach dem zweiten Backtracking

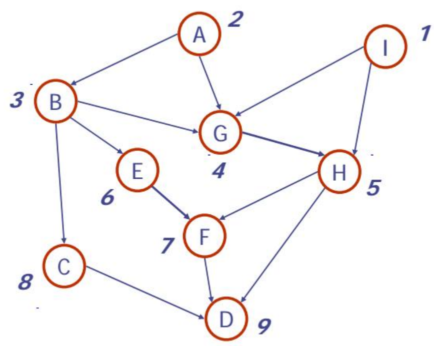


Abbildung 55: Topologische Sortierung

19.7.2. DAG Implementierung

Topologische Tiefensuche.

Algorithm 10: topologicalDFS(G)

```
Data: DAG G
Result: topological ordering of G
n ← G.numVertices()
forall u in G.vertices() do
    setLabel(u, UNEXPLORED)
end
forall e in G.edges() do
    setLabel(e, UNEXPLORED)
end
forall v in G.vertices() do
    if getLabel(v) = UNEXPLORED then
        topologicalDFS(G, v)
    end
end
```

Algorithm 11: 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
setLabel(v, VISITED)
forall e in G.outgoingEdges(v) do
    if getLabel(e) = UNEXPLORED then
        w ← opposite(v, e)
        if getLabel(w) = UNEXPLORED then
            setLabel(e, DISCOVERY)
            topologicalDFS(G, w)
        end
        else
            e is a forward or cross edge
        end
    end
end
Label v with topological number n
n ← n − 1
```

Listing 17: Directed DFS in Java

```java
public void directedDFS() {
   vertices.forEach(v -> vertexLabeling.put(v, VertexState.UNEXPLORED));
   edges.forEach(e -> edgeLabeling.put(e, EdgeState.UNEXPLORED));

   vertices.forEach(v -> {
      if (vertexLabeling.get(v) == VertexState.UNEXPLORED) {
         directedDFS(v);
      }
   });
}

public void directedDFS(Vertex vertex) {
   vertexLabeling.put(vertex, VertexState.VISITED);

   outgoingEdges(vertex).forEach(e -> {

      if (edgeLabeling.get(e) == EdgeState.UNEXPLORED) {
         Vertex opposite = opposite(vertex, e);
         if (vertexLabeling.get(opposite) == VertexState.UNEXPLORED) {
            edgeLabeling.put(e, EdgeState.DISCOVERY);
            displayOnGVS();
            directedDFS(opposite);
         } else {
            setKindOfEdge(vertex, e);
            displayOnGVS();
         }
      }
   });
}

private void setKindOfEdge(Vertex startVertex, Edge e) {
   Vertex endVertex = opposite(startVertex, e);
   if (path.contains(endVertex)) {
      // BACK
   } else if (subtreeNodes.get(startVertex).contains(endVertex)) {
      // FORWARD
   } else {
      // CROSS
   }
}
```

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 shiften. 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 ∞. 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

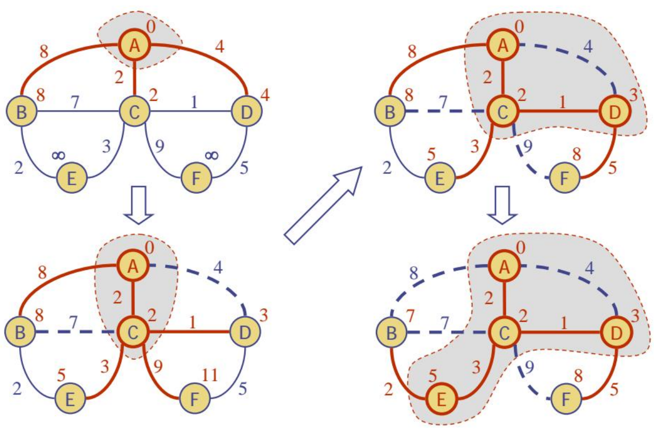


Abbildung 56: Dijkstra Algorithmus

20.2.2. 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 18: Dijkstra Algorithmus

```java
public void distances(AdjacencyListGraph<V, E> graph, Vertex<V> s) {
   AdaptablePriorityQueue<Integer, Vertex<V>> apq =
         new HeapAdaptablePriorityQueueGVS<Integer, Vertex<V>>();
   Map<Vertex<V>, Integer> distances = new LinkedHashMapGVS<Vertex<V>, Integer>();
   Map<Vertex<V>, Entry<Integer, Vertex<V>>> locators =
         new LinkedHashMap<Vertex<V>, Entry<Integer, Vertex<V>>>();
   Map<Vertex<V>, Edge<E>> parents = new LinkedHashMapGVS<Vertex<V>, Edge<E>>();
   gvs.set(apq, distances, parents);

   for (Vertex<V> v : graph.vertices()) {
      if (v == s) {
         distances.put(v, 0);
         // root node has no parents
         parents.put(v, null);
      } else {
         // set default distance to infinity
         distances.put(v, Integer.MAX_VALUE);
      }
      // add distance and vertex
      Entry<Integer, Vertex<V>> entry = apq.insert(distances.get(v), v);
      locators.put(v, entry);
   }

   while (!apq.isEmpty()) {
      // take next vertex out of the queue (removeMin) -> Kehrwert = cloud
      AdjacencyListGraph<V, E>.MyVertex<V> cloudVertex =
            (AdjacencyListGraph<V, E>.MyVertex<V>) (apq.removeMin().getValue());

      for (Edge<E> incidentEdge : cloudVertex.incidentEdges()) {
         Vertex<V> oppositVertex = graph.opposite(cloudVertex, incidentEdge);
         // calculate new weight: last vertex + edge weight
         int newWeight = distances.get(cloudVertex) + (Integer)
             incidentEdge.get(WEIGHT);
         if (newWeight < distances.get(oppositVertex)) {
            // relaxion
            distances.put(oppositVertex, newWeight);
            parents.put(oppositVertex, incidentEdge);
            apq.replaceKey(locators.get(oppositVertex), newWeight);
         }
      }
   }
}
```

20.3. 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.

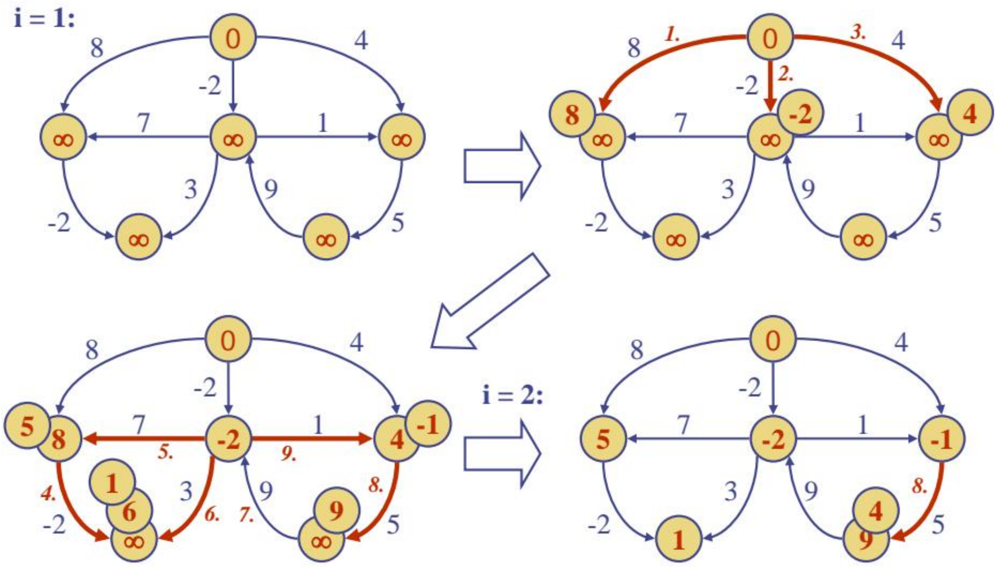


Abbildung 57: Bellman-Ford Algorithmus

20.3.1. Implementierung

Algorithm 12: BellmanFord(G,s)

```
forall v in G.vertices() do
    if v = s then
        setDistance(v, 0)
    end
    else
        setDistance(v, ∞)
    end
    for i ← 1 to n − 1 do
        u ← G.origin(e)
        z ← G.opposite(u, e)
        r ← getDistance(u) + weight(e)
        if r < getDistance(z) then
            setDistance(z, r)
        end
    end
end
```

20.4. 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: $\mathcal{O}(n + m)$

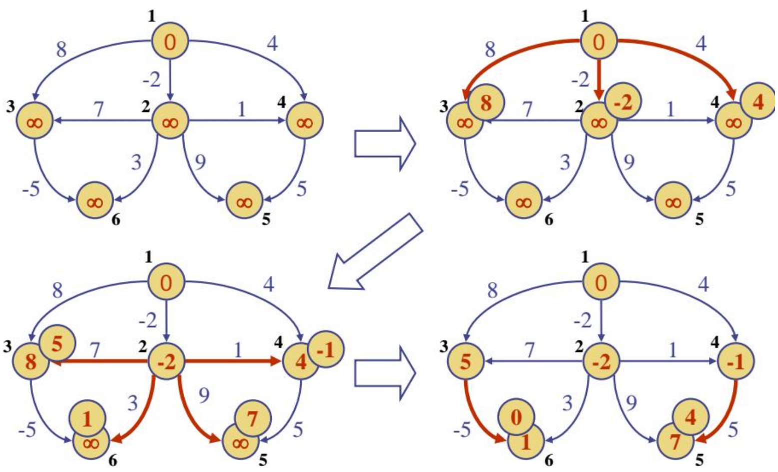


Abbildung 58: DAG Shortest Path

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.

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).

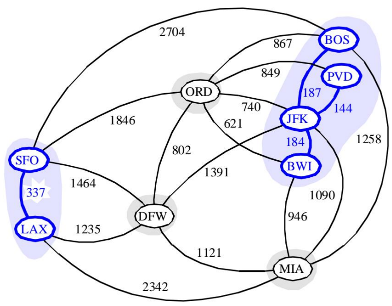


Abbildung 59: Step 1

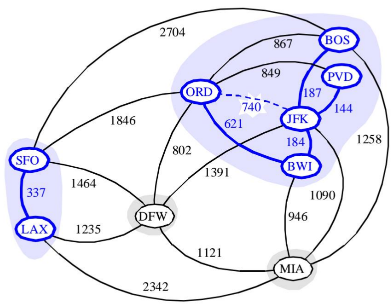


Abbildung 60: Step 2

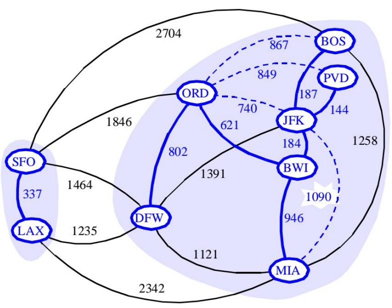


Abbildung 61: Step 3

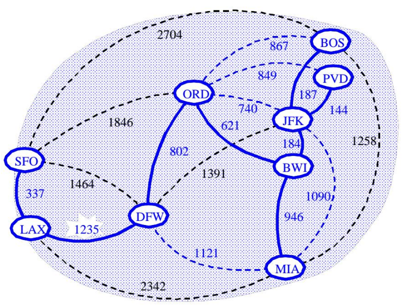


Abbildung 62: Step 4

21.2. 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) = kleinste Gewichtung einer Kante, welche v mit einem Vertex der Wolke verbindet. Bei jedem Schritt werden folgende Dinge durchgeführt:

21.2.1. Vorgehen

1. Die Standard Gewichte der Knoten ist ∞. 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

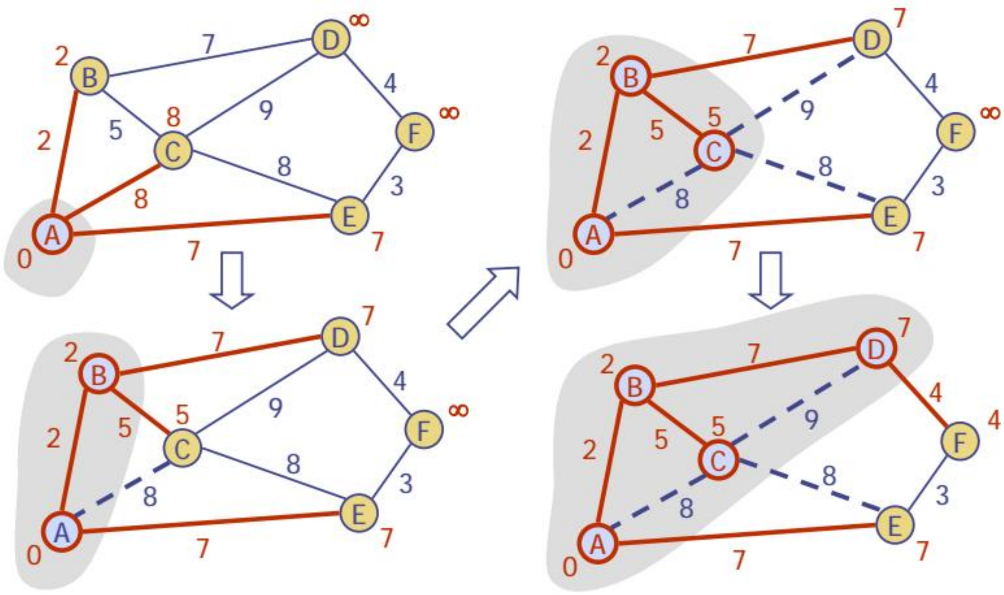


Abbildung 63: Prim-Jarnik's Algorithmus

21.3. 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**.

21.4. 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 22: Laufzeiten von Graph Operationen

A. Listings

B. Abbildungsverzeichnis

C. Tabellenverzeichnis