

# ZUSAMMENFASSUNG

## Lernziele

- Komplexe Algorithmen und Datenstrukturen erklären und an praktischen Beispielen anwenden
- Ein vorgegebener Algorithmus auf seine Komplexität analysieren und mit der O-Notation beschreiben und klassifizieren
- Vorgegebene Algorithmen und Datenstrukturen implementieren
- Eigene Abstrakte Datentypen definieren und mithilfe eigener Algorithmen und Datenstrukturen implementieren

## Unterlagen / Bücher

- Vorlesungsfolien von Herr Letsch
- Data Structures and Algorithms in Java, 6th Edition, 2014, Goodrich

## Erlaubte Hilfsmittel an der Prüfung

- Open Book Prüfung
  - o Alle Unterlagen erlaubt, ausser alte Prüfungen
-

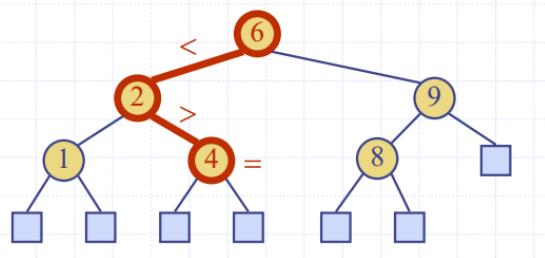
<b>BINÄRE SUCH-BÄUME .....</b>	<b>6</b>
MULTIMAPS (WIEDERHOHLUNG) .....	6
GEORDNETE MULTIMAPS .....	6
BINÄRE SUCHE .....	6
SUCHTABELLE .....	7
<i>Performance</i> .....	7
BINÄRER SUCH-BAUM (11.1) .....	7
<i>Suche</i> .....	7
<i>Einfügen</i> .....	8
<i>Löschen</i> .....	8
<i>Performance</i> .....	10
<i>Implementierung</i> .....	10
<b>AVL BÄUME .....</b>	<b>12</b>
HÖHE EINES AVL BAUMS .....	12
AVL – BÄUME .....	12
EINFÜGEN IN EINEN AVL BAUM .....	12
TRINODE UMSTRUKTURIERUNG .....	13
UMSTRUKTURIERUNG ALS EINZEL-ROTATIONEN .....	14
UMSTRUKTURIERUNG ALS DOPPEL-ROTATIONEN .....	14
CUT/LINK RESTRUKTURIERUNG-ALGORITHMUS .....	15
LÖSCHEN AUS EINEM AVL BAUM .....	17
BALANCIERUNG NACH DEM LÖSCHEN .....	17
LÖSCHEN EINES EXTERNEN KNOTENS .....	18
IMPLEMENTIERUNG .....	19
LAUFEZEITEN VON AVL BÄUMEN .....	19
<b>SPLAY BÄUME .....</b>	<b>20</b>
SUCHEN IN EINEM SPLAY BAUM .....	20
SPLAYING .....	21
VISUALISIERUNG DER SPLAYING FÄLLE .....	21
BEISPIEL .....	22
WEITERES BEISPIEL .....	22
BEISPIEL FÜR LÖSCHEN .....	22
SPLAY BÄUME UND MULTI-/MAPS .....	22
<b>MERGE SORT (SORTIERUNG DURCH MISCHEN) .....</b>	<b>23</b>
DIVIDE-AND-CONQUER .....	23
MERGE-SORT .....	23
MISCHEN ZWEIER SORTIERTER SEQUENZEN .....	23
MERGE-SORT BAUM .....	24
AUSFÜHRUNGS-BEISPIEL .....	24
ANALYSE VON MERGE-SORT .....	25
ZUSAMMENFASSUNG DER SORTIER-ALGORITHMEN .....	25
IMPLEMENTIERUNG .....	25
<b>QUICK SORT .....</b>	<b>26</b>
PARTITIONIERUNG .....	26
QUICK-SORT TREE .....	26

AUSFÜHRUNGS-BEISPIEL .....	27
WORST-CASE LAUFZEITVERHALTEN.....	27
ERWARTETE LAUFZEIT .....	28
IN-PLACE QUICK-SORT .....	28
IN-PLACE PARTITIONIERUNG .....	28
JAVA IMPLEMENTATION (FÜR UNGLEICHÉ ÉLÉMЕНTE).....	29
ZUSAMMENFASSUNG DER SORTIER-ALGORITHMEN .....	29
<b>SORTING LOWER BOUND .....</b>	<b>30</b>
VERGLEICHSBASIERTE SORTIERUNG.....	30
ZÄHLEN DER VERGLEICHE.....	30
HÖHE DES ENTSCHEIDUNGS-BAUMES.....	30
DIE UNTERE GRENZE (LOWER BOUND).....	30
<b>BUCKET-SORT.....</b>	<b>31</b>
BEISPIEL.....	31
EIGENSCHAFTEN UND ERWEITERUNGEN .....	31
LEXIKOGRAPHISCHE ORDNUNG .....	32
LEXIKOGRAPHISCHE SORTIERUNG .....	32
<b>RADIX SORT.....</b>	<b>33</b>
RADIX-SORT FÜR BINÄRE ZAHLEN .....	33
BEISPIEL.....	33
<b>PATTERN MATCHING.....</b>	<b>34</b>
STRINGS.....	34
PATTERN MACHTING ALGORITHMEN .....	34
<i>Brute-Force Algorithmus</i> .....	34
<i>Boyer-Moore Algorithmus</i> .....	35
<i>Knuth-Morris-Pratt Algorithmus</i> .....	38
<b>TRIES .....</b>	<b>41</b>
PREPROCESSING STRINGS .....	41
STANDARD TRIES .....	41
<i>Analyse des Standard-Tries</i> .....	41
SUCHE IN EINEM TRIE .....	42
KOMPROMIERTE TRIES.....	42
KOMPAKTE REPRÄSENTATION.....	42
SUFFIX TRIE .....	43
<i>Analyse des Suffix-Tries.</i> .....	43
<b>DYNAMISCHE PROGRAMMIERUNG.....</b>	<b>44</b>
RUCKSACKPROBLEM .....	44
<i>Versuch durch Aufzählung - Brute Force</i> .....	44
<i>Versuch mit jeweils Grösstem - Greedy Algorithmus</i> .....	44
<i>Versuch mit Subproblemen - Dynamische Programmierung</i> .....	44
TECHNIK DER DYNAMISCHEN PROGRAMMIERUNG .....	45
SUBSEQUENZEN.....	45
LÄNGE GEMEINSAME SUBSEQUENZ (LCS, LONGEST COMMON SUBSEQUENCE) .....	46
<i>Ein schwacher Versuch für das LCS Problem</i> .....	46
<i>Versuch mit dynamischer Programmierung</i> .....	46

DER LCS ALGORITHMUS.....	46
<i>Visualisierung des LCS Algorithmus.....</i>	47
<i>Analyse des LCS Algorithmus .....</i>	47
<i>Auslesen der LCS.....</i>	47
<b>GRAPHEN.....</b>	<b>48</b>
KANTENTYPEN .....	48
<i>Gerichtete Kanten.....</i>	48
<i>Ungerichtete Kanten.....</i>	48
<i>Gerichteter Graph .....</i>	48
<i>Ungerichteter Graph .....</i>	48
ANWENDUNGEN .....	48
TERMINOLOGIE .....	49
EIGENSCHAFTEN .....	49
<i>Eigenschaft 1.....</i>	49
<i>Eigenschaft 2.....</i>	49
HAUPTMETHODEN DES GRAPH-ADT's.....	50
KANTEN-LISTEN STRUKTUR.....	50
ADJAZENZ-LISTEN STRUKTUR .....	50
ADJAZENZ-MATRIX STRUKTUR .....	51
PERFORMANCE .....	51
SUBGRAPHEN .....	51
<i>Connectivity.....</i>	52
<i>Bäume und Wälder .....</i>	52
<i>Spanning Trees und Wälder .....</i>	52
<b>DEPTH-FIRST SEARCH .....</b>	<b>53</b>
DFS ALGORITHMUS .....	53
BEISPIEL.....	53
DFS UND LABYRINTH .....	54
EIGENSCHAFTEN VON DFS .....	54
<i>Eigenschaft 1.....</i>	54
<i>Eigenschaft 2.....</i>	54
ANALYSE VON DFS.....	54
PFADE FINDEN .....	54
ZYKLEN FINDEN.....	55
<b>BREATH-FIRST SEARCH / BREITENSUCHE.....</b>	<b>55</b>
BFS ALGORITHMUS.....	55
BEISPIEL.....	56
EIGENSCHAFTEN .....	56
ANALYSIS .....	57
APPLIKATIONEN .....	57
DFS vs. BFS .....	57
<b>DIRECTED GRAPHS / GERICHTETE GRAPHEN.....</b>	<b>58</b>
EIGENSCHAFTEN .....	58
ANWENDUNGEN .....	58
GERICHTETE TIEFENSUCHE .....	58
ERREICHBARKEIT .....	58
STRING CONNECTIVITY .....	59

Algorithmus.....	59
Streng verbundene Komponenten.....	59
TRANSITIVER ABSCHLUSS .....	60
Berechnung.....	60
FLOYD-WARSHALL-ALGORITHMUS.....	60
Grundidee.....	60
Pseudocode.....	61
Beispiel.....	61
DAG'S UND TOPOLOGISCHE ORDNUNG.....	62
TOPOLOGISCHE SORTIERUNG.....	63
Algorithmus.....	63
Algorithmus mit Tiefensuche.....	63
Beispiel.....	64
<b>SHORTEST PATHS TREES / KÜRZESTE PFADE BÄUME .....</b>	<b>66</b>
GEWICHTETE GRAPHEN .....	66
KÜRZESTER PFAD .....	66
Eigenschaften.....	66
KANTEN RELAXATION (ENTSPANNUNG).....	67
BEISPIEL.....	67
DIJKSTRA'S ALGORITHMUS .....	68
Analyse.....	68
KÜRZESTER PFAD BÄUME .....	69
WIESO DIJKSTRA'S ALGORITHMUS FUNKTIONIERT.....	69
WIESO ES FÜR GEWICHTE KLEINER NULL NICHT FUNKTIONIERT.....	69
BELLMAN-FORD ALGORITHMUS .....	69
Beispiel.....	70
DAG-BASIERTER ALGORITHMUS.....	70
Beispiel.....	70
<b>MINIMUM SPANNING TREES (MINIMAL AUFPANNENDE BÄUME) .....</b>	<b>71</b>
SCHLAUFEN-EIGENSCHAFT .....	71
AUFTeilungs-EIGENSCHAFT .....	71
KRUSKAL'S ALGORITHMUS .....	72
Datenstruktur.....	72
Repräsentation einer Partition .....	72
Partition-Basierte Implementation.....	73
Beispiel.....	73
PRIM-JARNIK'S ALGORITHMUS .....	76
Beispiel.....	76
Analyse.....	77
BORUVKA'S ALGORITHMUS .....	77
Beispiel.....	78

# Binäre Such-Bäume



Die Binären Suchbäume sind in BST Bäume, AVL Bäume (selbstbalancierend) und Splay Bäume (selbstbalancierend, in der Suchmaschinen Suche, oft sehr schnell aber nicht immer) aufgeteilt.

## Multimaps (Wiederholung)

Die Multimaps sind ungeordnet und verwenden folgende Methoden.

<b>Find(K)</b> Liefert Entry zum Schlüssel k oder null	<b>findAll(k)</b> liefert einen iterierbare Collection mit allen Entries zum Schlüssel k
<b>Insert(k,o)</b> neue Entry zum Schlüssel k und Wert o	<b>Remove(e)</b> Entfernt die Entry e (und Rückgabe von e)

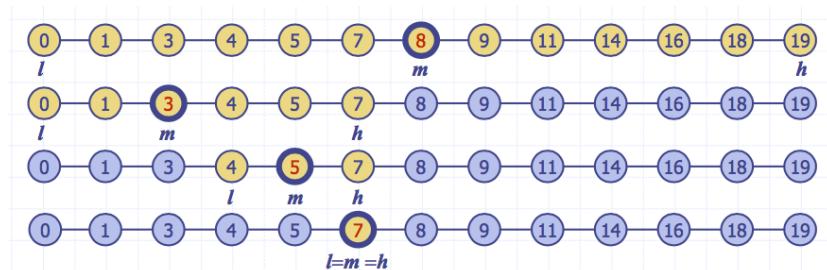
## Geordnete Multimaps

Die Keys folgen einer vollständigen Ordnungsrelation ( $k$  tief  $m \leq k$  tief  $n$ )

<b>First()</b> liefert erste Entry in der Multimap-Ordnung	<b>Successors(k)</b> liefert Iterator über Entries mit Schlüssel grösser oder gleich k, nicht abnehmende Ordnung
<b>Last()</b> liefert letzte Entry in der Multimap-Ordnung	<b>Predecessors(k)</b> liefert Iterator über Entries mit Schlüssel kleiner oder gleich k, nicht zunehmende Ordnung.

## Binäre Suche

Bei einer Multimap, realisiert als eine array-basierte Sequenz, sortiert nach dem key, so gilt für die Operation **find (k)**: Die Anzahl der Kandidaten wird bei jedem Schritt halbiert und terminiert nach  $O(\log n)$  Schritten). Beispiel für **find(7)**.



## Suchtabelle

Eine Suchtabelle ist eine Multimap, welche mithilfe einer sortierten Sequenz implementiert wird. Die Entries der Multimap werden in einer Array-basierten Sequenz abgespeichert, sortiert nach dem Schlüssel.

Die Suchtabelle sind nur dann effektiv, wenn die Multimap klein ist und vor allem Suchoperationen ausgeführt werden.

## Performance

**Find** benötigt  $O(\log n)$ , falls eine Binärsuche eingesetzt wird.

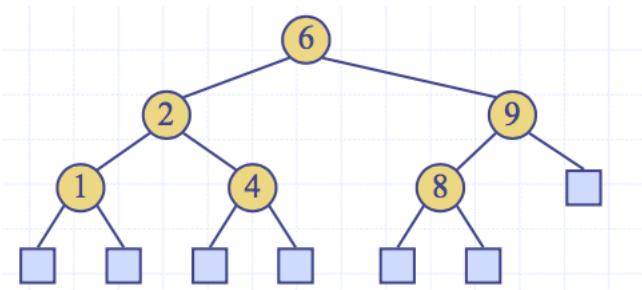
**Insert** benötigt  $O(n)$  Zeit, da wir im schlimmsten Fall die alten Entries um einen Platz nach hinten schieben müsse, um Platz für einen neuen Entry zu schaffen.

**Remove** benötigt  $O(n)$  Zeit, aus demselben Grund wie der insert().

## Binärer Such-Baum (11.1)

Ein binärer Such-Baum ist ein binärer Baum, welcher Keys (oder Key-Value-Entries) in seinen internen Knoten speichert und folgende Bedingung erfüllt:

Gegeben sind die drei Knoten **u**, **v** und **w**. **u** ist im linken Teilbaum von **v** und **w** ist im rechten Teilbaum von **v**. So gilt **key(u) < key(v) < key(w)**. Die externen Knoten (Blatt-Knoten) speichern keine Daten. Die Indorder-Traversierung eines binären Such-Baumes besucht die Keys in nicht absteigender Folge.



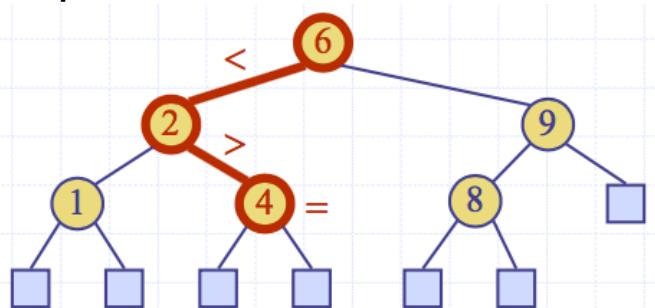
## Suche

Bei einer Suche nach dem Key **k** beginnen wir beim Root. Welches der nächste Knoten ist, hängt vom Resultat des Vergleiches von **k** mit dem Key des aktuellen Knotens ab. Im Falle, dass wir ein Blatt erreichen, wurde der Key nicht gefunden und wir geben null zurück.

```

Algorithm TreeSearch(k, v)
  if T.isExternal (v)
    return null          // Beispiel: T.left(8)
  if k < key(v)
    return TreeSearch(k, T.left(v)) // Beispiel: key(6)
  else if k = key(v)      // Beispiel: key(4)
    return v
  else { k > key(v) } // Beispiel: key(2)
    return TreeSearch(k, T.right(v))
  
```

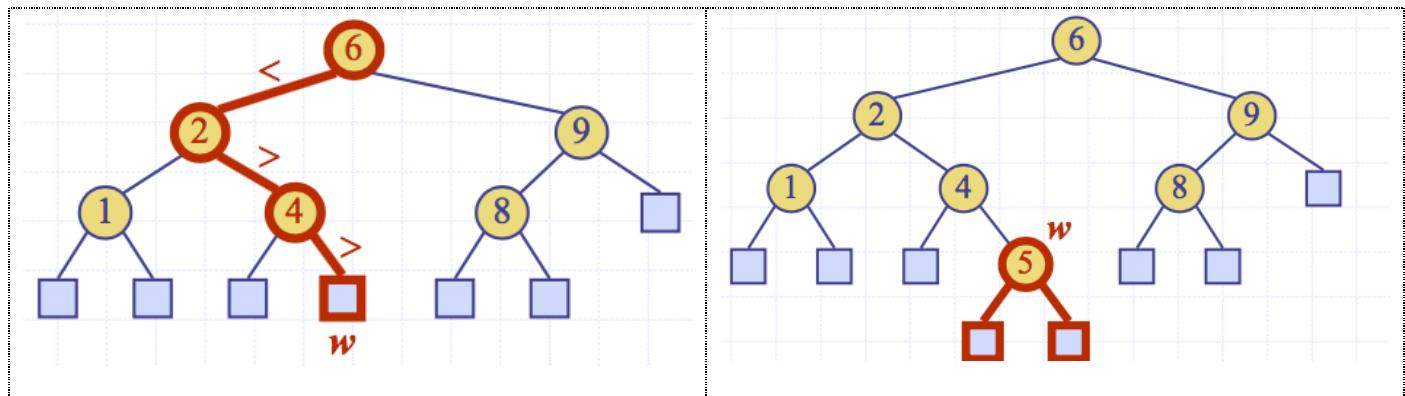
**Beispiel für find(4) → TreeSearch(4, root)**



Bei der Operation `insert(k, o)` suchen wir zuerst den Key k (mittels TreeSearch).

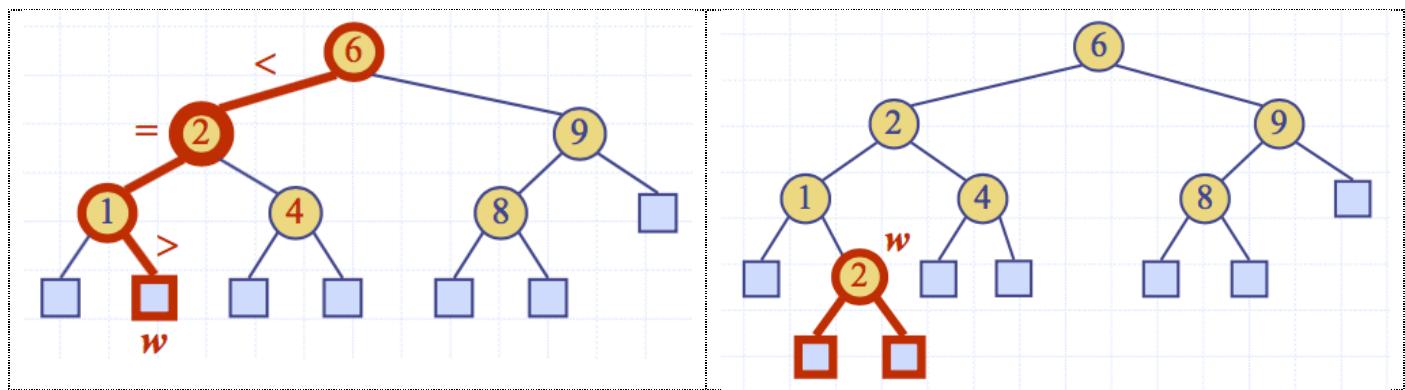
### Beispiel, insert(5)

Annahme, dass k noch nicht im Baum vorhanden ist und w wäre das Blatt, welches mit der Suche gefunden wird. In diesem Falle fügen wir k beim Knoten w ein und expandieren w in einen internen Knoten.



### Beispiel, insert(2)

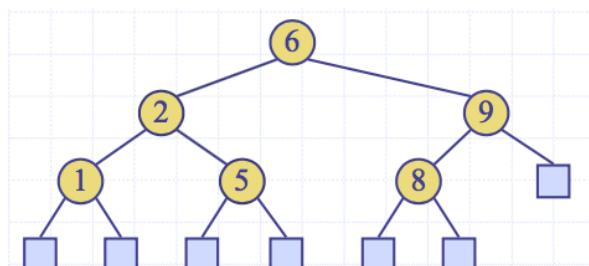
Annahme, dass der Baum eine Multimap ist und k bereits vorhanden. Im jeweils linken Teilbaum von k wird weitergesucht bis man auf ein Blattknoten w stösst. Wir fügen daher k beim Knoten w ein und expandieren w in einen internen Knoten.



### Löschen

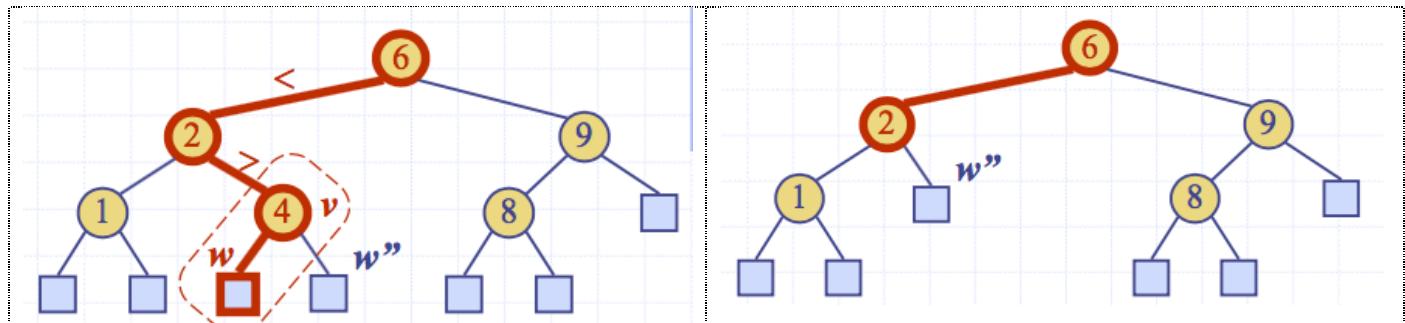
Bei einer Operation `remove(k)` suchen wir zuerst den Key k. Dann gibt es drei Fälle zu unterscheiden. Der Knoten v mit Schlüssel k ist

- ein Knoten mit zwei Blatt-Kinder (**Beispiel:** Knoten 5)
- ein Knoten mit einem Blatt-Kind (**Beispiel:** Knoten 9)
- ein Knoten ohne Blatt-Kinder (**Beispiel:** Knoten 2)

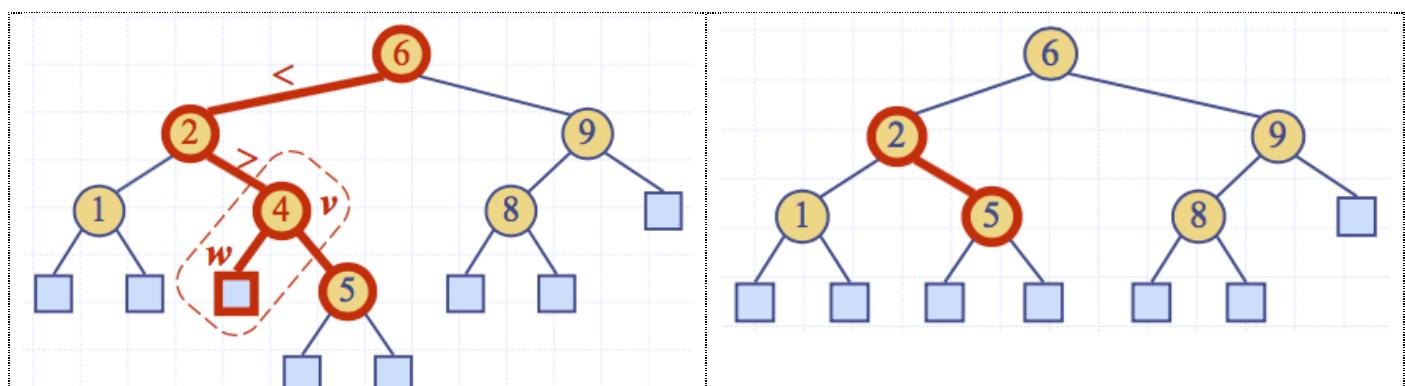


**Beispiel, remove(4)**

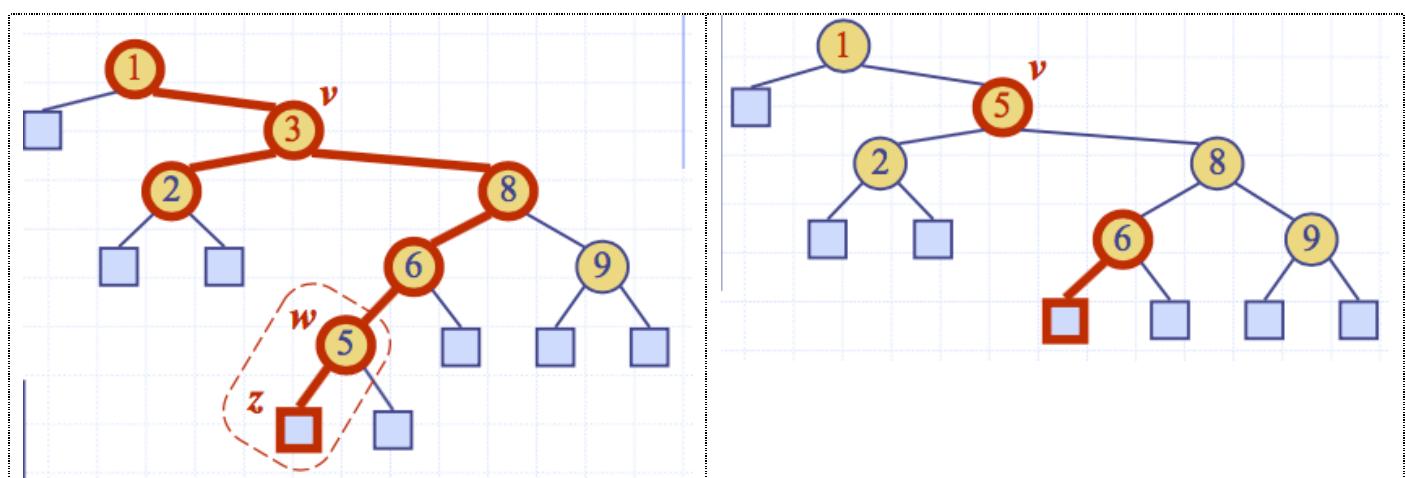
Annahme, dass k im Baum ist und v ein entsprechender Knoten. Wenn der Knoten v zwei Blatt-Kinder w und w" hat, wird v und w vom Baum gelöscht mit der Operation removeExternal(w), welche w und seine Eltern-Knoten löscht und den Eltern-Knoten durch w" (seine Geschwister-Knoten) ersetzt.

**Weiteres Beispiel, remove(4)**

Wenn der Knoten v ein Blatt-Kind w hat, wird v und w vom Baum gelöscht mit der Operation removeExternal(w), welche w und seine Eltern-Knoten löscht und den Eltern-Knoten durch den Geschwister-Knoten w ersetzt.

**Beispiel, remove(3) ohne Blattkinder**

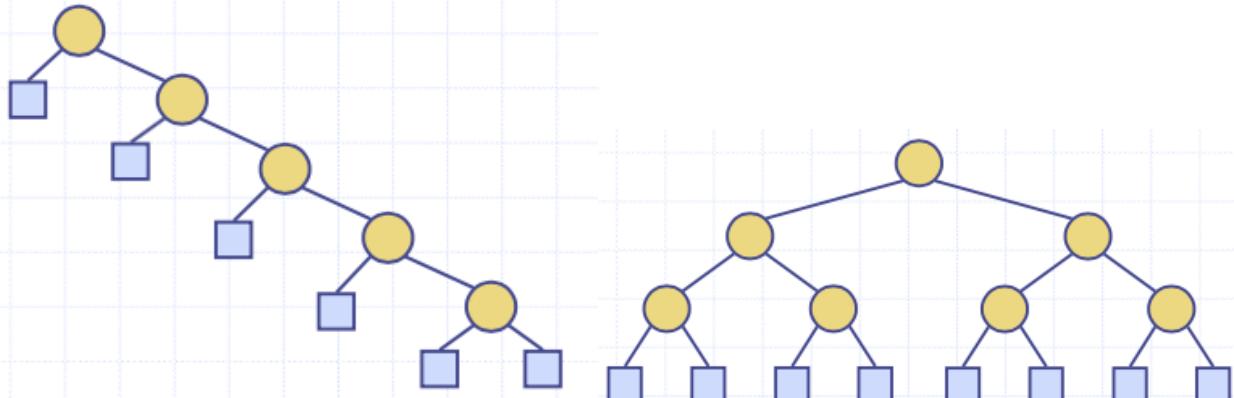
Wir betrachten den Fall, bei welchem der zu löschenende Key k in einem Knoten v gespeichert ist, der keine Blattkinder hat. Zuerst gilt es den internen Knoten w zu finden, welcher v in der Inorder-Traversierung folgt. Kopiere nun key(w) in den Knoten v. Zuletzt lösche den Knoten w und sein linkes Kind z (welches ein Blatt sein muss) mit der Operation removeExternal(z).



Betrachte eine Map mit n Entries, welcher mit einem binären Suchbaum implementiert ist mit der Höhe h.

- Nötiger Speicher ist  $O(n)$
- Methoden find, insert und remove benötigen  $O(h)$

Die Höhe h ist  $O(n)$  im schlechtesten Fall und  $O(\log n)$  im besten Fall



## Implementierung

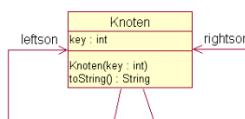
### Der Knoten:

```
public class Knoten {
    Knoten leftson, rightson;
    int key;

    // Hier könnte ein Infotype kommen
    // mit dem ein ValueObjekt definiert wird

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

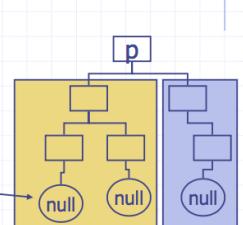
    public String toString() {
        return String.valueOf(key);
    }
}
```



### Der Binärbaum (schematisch)

```
public class Baum {
    Knoten wurzel;
    Baum() {
        wurzel = null;
    }

    Knoten suchen(Knoten p, int x) {
        if (p == null) return null;
        if (x < p.key)
            return suchen(p.leftson,x);
        else
            if (x > p.key)
                return suchen(p.rightson,x);
        return p;
    }
}
```



## Ebenfalls schematisch

```
public class Baum {
    ...
    public void einfuegen(int k) {
        wurzel = einfuegen(wurzel,k); //nicht public: friendly
    }

    Knoten einfuegen(Knoten p, int k) {
        if (p == null)
            return new Knoten(k);
        else if (k < p.key)
            p.leftson = einfuegen(p.leftson,k);
        else if (k > p.key)
            p.rightson = einfuegen(p.rightson,k);
        return p;
    }
}
```

```
public void entfernen(int k) {
    wurzel = entfernen(wurzel,k);
}

Knoten entfernen(Knoten p, int k) {
    if (p == null) return p;
    if (k < p.key)
        p.leftson = entfernen(p.leftson,k);
    else
        if (k > p.key)
            p.rightson = entfernen(p.rightson,k);
        else {
            if (p.leftson == null)
                return p.rightson;
            if (p.rightson == null)
                return p.leftson;
            Knoten q = vatersymmach(p);
            if (q == p) {
                p.key = p.rightson.key;
                q.rightson = p.rightson.rightson;
            }
            else {
                p.key = q.leftson.key;
                q.leftson = q.leftson.rightson;
            }
        }
    return p;
}
```

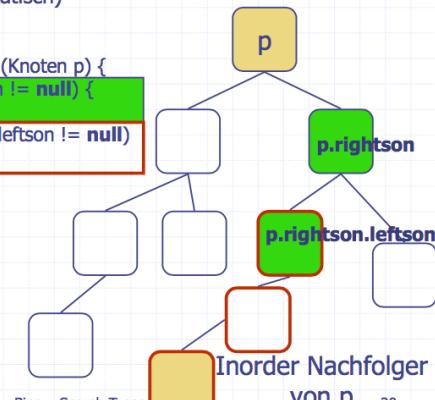
Ersetze den Knoten durch den Nachfolger gemäss Inorder / vatersymmach().

Dies garantiert, dass die Inorder-Reihenfolge eingehalten wird.

## Algorithmen und Datenstrukturen 2

Der Binärbaum (schematisch)

```
public class Baum {
    ...
    Knoten vatersymnach (Knoten p) {
        if (p.rightson.leftson != null) {
            p = p.rightson;
            while (p.leftson.leftson != null)
                p = p.leftson;
        }
        return p;
    }
}
```



Goodrich, Tamassia

Binary Search Trees

Inorder Nachfolger

VON p

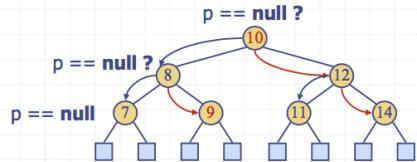
20

```
public class BaumTest {
    public static void main(String args[]) {
        Baum B = new Baum();
        B.einfügen(17);
        B.einfügen(11);
        B.einfügen(7);
        B.einfügen(14);
        B.einfügen(12);
        B.einfügen(22);
        System.out.println(B);

        System.out.println(B.suchen(14));
        System.out.println(B.suchen(8));
        B.entfernen(17);
        B.entfernen(7);
        B.entfernen(12);
        B.entfernen(14);
        B.entfernen(22);
        B.entfernen(11);
        System.out.println(B);
    }
}
```

Inorder()

```
String inorder(Knoten p) {
    if (p == null)
        return "";
    String str =
        inorder(p.leftson) + "(" + p.toString() + ")" + inorder(p.rightson);
    return str;
}
```



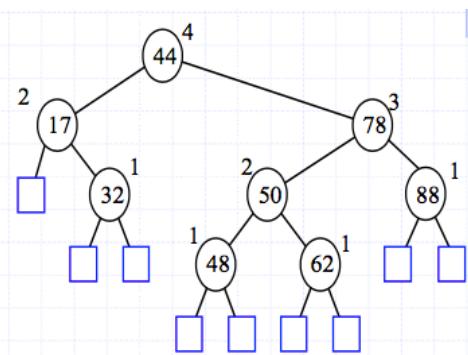
Preorder : (17)(11)(7)(14)(12)(22)  
Inorder : (7)(11)(12)(14)(17)(22)  
Postorder: (7)(12)(14)(11)(22)(17)

14

null

Preorder :  
Inorder :  
Postorder:

# AVL Bäume



Ein AVL Baum ist ein binärer Suchbaum, bei dem für jeden internen Knoten  $v$  von  $T$  gilt: die Höhe der Kinder von  $v$  unterscheiden sich höchstens um 1. AVL Bäume sind grundsätzlich balanciert.

## Höhe eines AVL Baums

- ◆ **Behauptung:** Die **Höhe** eines AVL Baumes  $T$ , der  $n$  Keys speichert, ist  $O(\log(n))$ .
- ◆ **Plausibel machen:** Als erstes müssen wir die **Anzahl Knoten** als Funktion der Höhe finden:  $n(h)$   
Die **minimale Anzahl interner Knoten** eines AVL Baums der Höhe  $h$ :
  - Sicher gilt  $n(1) = 1$  und  $n(2) = 2$
  - Für  $h \geq 3$  umfasst ein AVL Baum der Höhe  $h$  die Wurzel, einen AVL-Unterbau der Höhe  $h-1$  und den zweiten AVL-Unterbau der Höhe  $h-2$ .
  - Also gilt rekursiv:  $n(h) = 1 + n(h-1) + n(h-2)$

- ◆ Ausgangslage:  $n(h) = 1 + n(h-1) + n(h-2)$   
 $n(1) = 1$  und  $n(2) = 2$
- ◆ Sicher gilt  $n(h-1) > n(h-2)$ , also  $n(h) > 2n(h-2)$   
 $n(h) > 2n(h-2) = 2(1+n(h-3)+n(h-4)) > 2(1+2n(h-4))$   
 $n(h) > 4n(h-4)$   
...
- ◆  $n(h) > 2^{h-2}$
- Das geht so weiter bis  $h-2i=2$  ist oder  $i=(h-2)/2=h/2-1$ ;  
(dann ist  $n(2)=2$ )
- ◆ insgesamt:  $n(h) > 2^{h/2-1}$
- ◆ logarithmieren:  $h < 2\log(n(h)) + 2$
- ◆ Somit gilt für die Höhe eines AVL Baumes  $O(\log(n))$

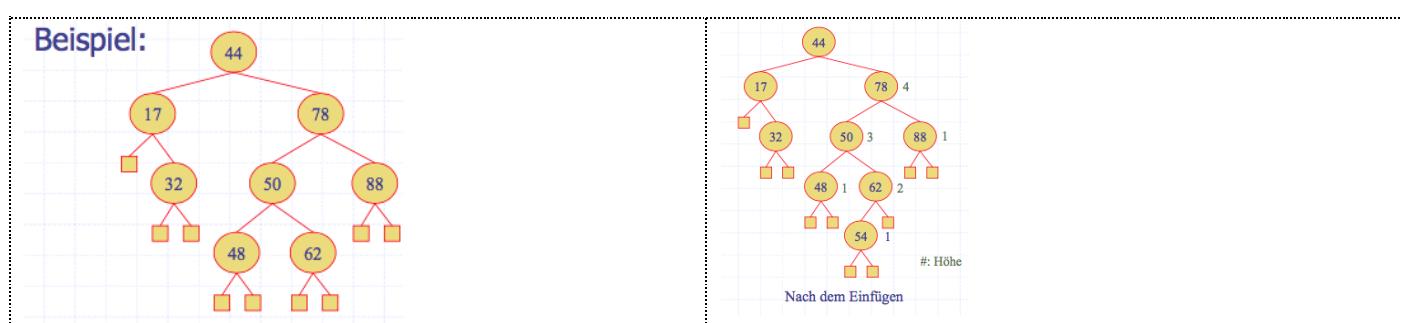
## AVL – Bäume

Die Balance ist mit  $b(K) = \text{Höhe (links)} - \text{Höhe (rechts)}$  definiert. In den AVL Bäumen gilt  $b(k)$  ist  $-1, 0$  und  $1$ . Nachdem Einfügen eines neuen Knoten kann gelten  $-2 \leq b(k) \leq 2$ . Oft beschriftet man die Knoten gleich mit diesem  $b(k)$ . In anderen Quellen wird manchmal auch  $\text{Höhe (rechts)} - \text{Höhe (links)}$  angeben. Grundsätzlich ist beides richtig. Einfach immer das gleiche.

## Einfügen in einen AVL Baum

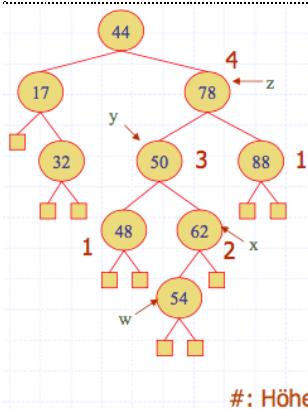
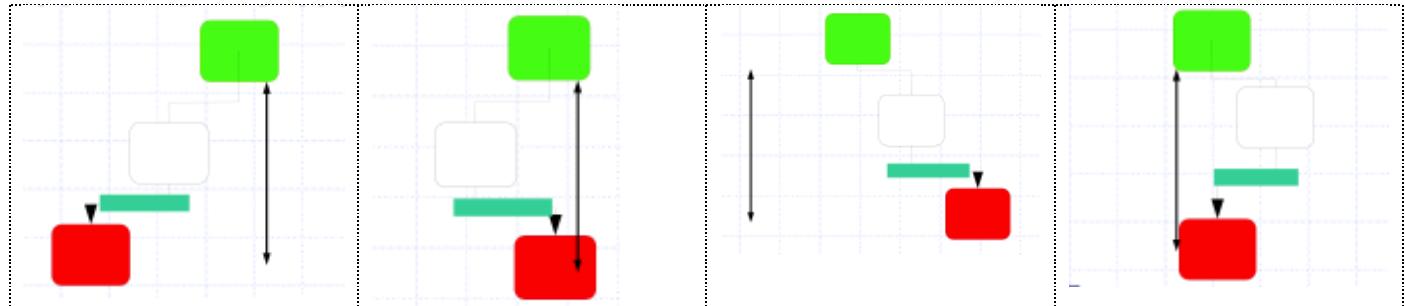
Ein Einfügen erfolgt wie bei einem binären Suchbaum. Somit wird immer ein externer Knoten expandiert.

## Beispiel



## Verletzungen des AVL-Merkals können beispielsweise auftreten beim:

- 1. Einfügen eines Knotens in den linken Teilbaum des linken Sohnes
- 2. Einfügen eines Knotens in den rechten Teilbaum des linken Sohnes
- 3. Einfügen eines Knotens in den rechten Teilbaum des rechten Sohnes
- 4. Einfügen eines Knotens in den linken Teilbaum des rechten Sohnes

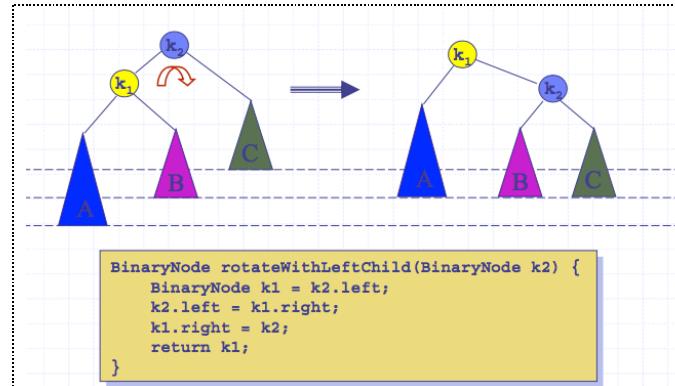
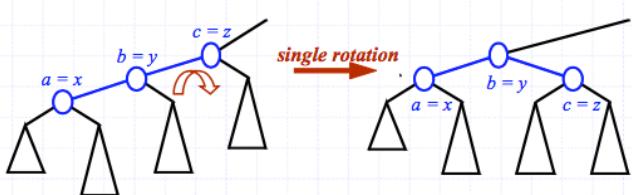
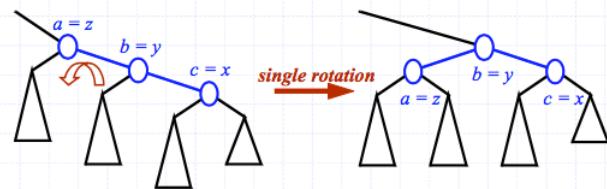


Falls das Einfügen dazu führt, dass T unbalanciert ist, müssen wir den Baum ausbalancieren. In folgendem Fall wandern wir im neuen Baum aufwärts bis wir den ersten Knoten x finden, dessen Grosseltern z ein unbalancierter Knoten ist.

## Trinode Umstrukturierung

Sei (a,b,c) ein Inorder-Listing von x, y und z. Das Ziel ist die Durchführung der nötigen Rotationen, damit b zum obersten Knoten des Baumes wird.

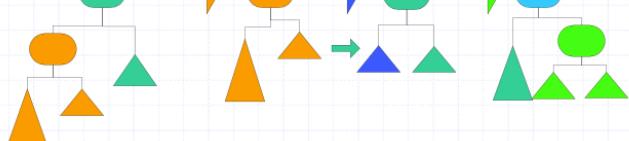
## Umstrukturierung als Einzel-Rotationen



```

BinaryNode rotateWithLeftChild( BinaryNode k2 ) {
    BinaryNode k1 = k2.left; // Hilfsknoten
    k2.left = k1.right;
    k1.right = k2;
    return k1;
}

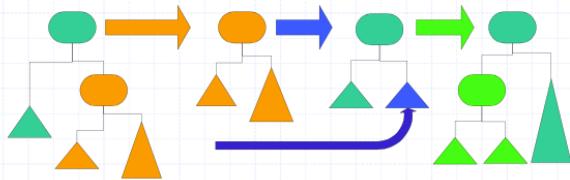
```



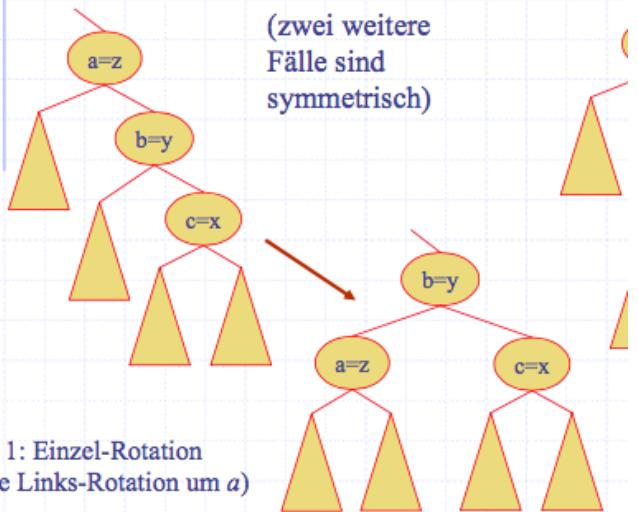
```

BinaryNode rotateWithRightChild( BinaryNode k1 ) {
    BinaryNode k2 = k1.right; // Hilfsknoten
    k1.right = k2.left;
    k2.left = k1;
    return k2;
}

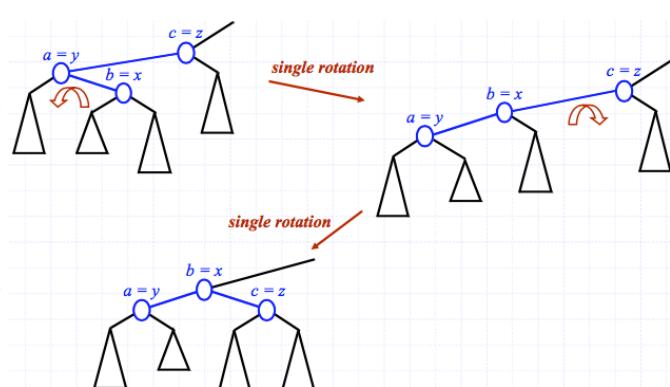
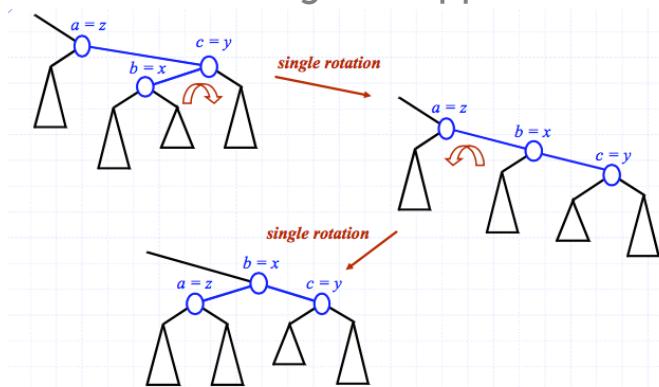
```

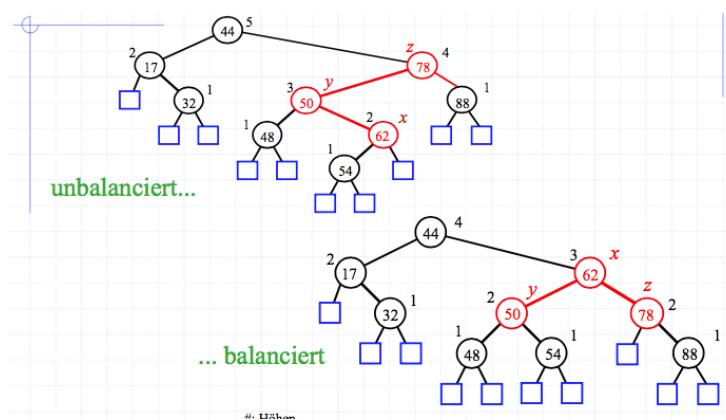


(zwei weitere Fälle sind symmetrisch)

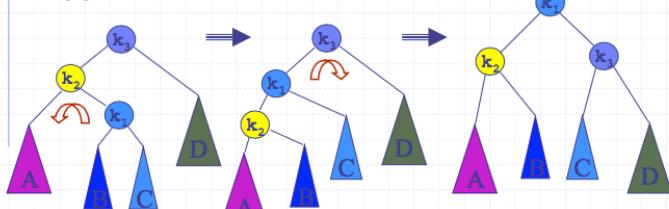


## Umstrukturierung als Doppel-Rotationen



**Ein Beispiel vom Einfügen** **Doppelrotation**

Die mit dem Rightchild funktioniert genau gleich.



```
BinaryNode doubleRotateWithLeftChild(BinaryNode k3) {
    k3.left = rotateWithRightChild(k3.left);
    return rotateWithLeftChild(k3);
}
```

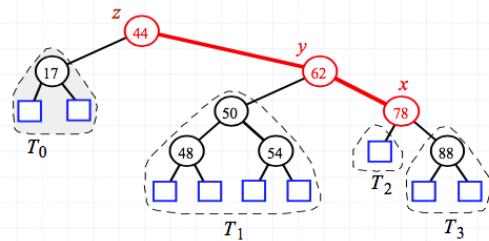
**Cut/Link Restrukturierung-Algorithmus**

Entspricht dem `restructure()` Algorithmus.

 **Algorithmus `restructure(x)`:**

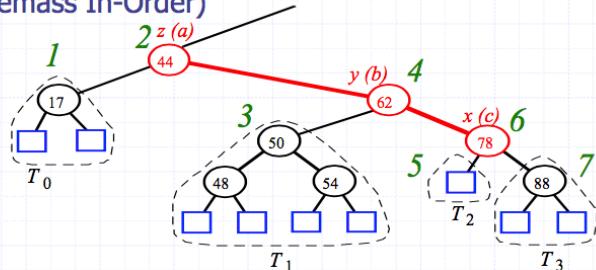
- **Input:** Ein Knoten  $x$  eines binären Suchbaumes  $T$ , welcher einen Eltern-Knoten  $y$  und einen Grosseltern-Knoten  $z$  besitzt.
- **Output:** restrukturierter Baum  $T$ , mit rotierten Knoten (entweder Single oder Double)  $x$ ,  $y$  und  $z$ .
- 1: Sei  $(a, b, c)$  die (Inorder) geordnete Liste der Knoten  $x$ ,  $y$  und  $z$ , und sei  $(T_0, T_1, T_2, T_3)$  die Liste der vier Unterbäume von  $x$ ,  $y$  und  $z$  (an den nicht rotierten Knoten  $x$ ,  $y$  oder  $z$ ).
- 2: Ersetze den Unterbaum mit Root  $z$  durch den Unterbaum mit Root  $b$ .
  - ◆ 2a: Setze  $a$  als linkes Kind von  $b$  und  $T_0$  als den linken resp. rechten Unterbaum von  $a$ .
  - ◆ 2b: Setze  $c$  als rechtes Kind von  $b$  und  $T_2$ ,  $T_3$  als den linken resp. rechten Unterbaum von  $c$ .

- ◆ Schauen wir uns den Algorithmus etwas genauer an...
- ◆ Jeden Baum, den man ausbalancieren muss, kann man in 7 Teile aufteilen: x, y, z und die 4 Bäume, mit Wurzeln direkt unterhalb x, y und z (T0-T3)

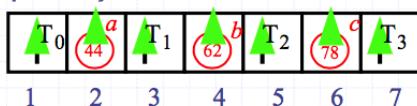


- ◆ Zuerst nummerieren wir die sieben Teile (In-Order Traversierung)

- ◆ Dann nennen wir die Knoten um (a, b, c: gemäss In-Order)

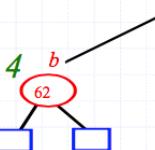


- ◆ Jetzt stellen wir x,y und z (child,parent, grandparent) in In-Order in das Array.

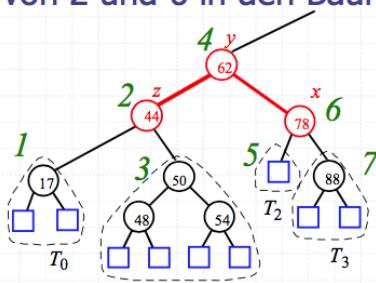


- ◆ nun bauen wir den Baum schrittweise wieder auf:

- wir beginnen mit b



- ◆ Am Schluss setzen wir 1,3,5 und 7 als Kinder von 2 und 6 in den Baum ein:

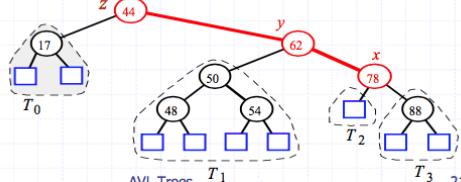


- ◆ Das Ergebnis ist ein balancierter Baum!

- ◆ Nun bauen wir einen neuen Baum, welcher balanciert ist.

- Dazu ordnen wir die Knoten so um, dass die Ordnung bei der In-Order Traversierung erhalten bleibt.
  - Dies funktioniert immer, ob nun der Originalbaum balanciert ist oder nicht.

- ◆ Schauen wir uns das Ganze an einem Beispiel an:

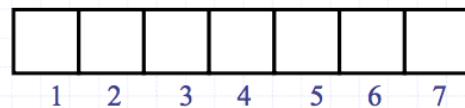


015 Goodrich, Tamassia

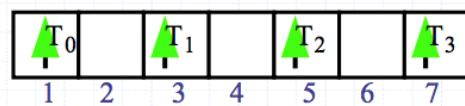
AVL Trees

21

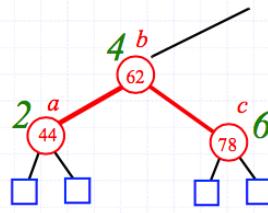
- ◆ Wir kreieren ein Array mit 7 Elementen: 1 bis 7



- ◆ nun schneiden wir die 4 Bäume ab und plazieren sie in das Array (In-Order)



- .. und setzen Element 2 (a) und Element 6 (c) als Kinder von 4 im Baum ein.



- ◆ Dieser Algorithmus bewirkt das selbe, wie die vier Rotationen, die wir vorher besprochen haben.

- ◆ Vorteil:

- keine Fallunterscheidung
- eleganter

- ◆ Nachteil:

- komplexerer Programmcode

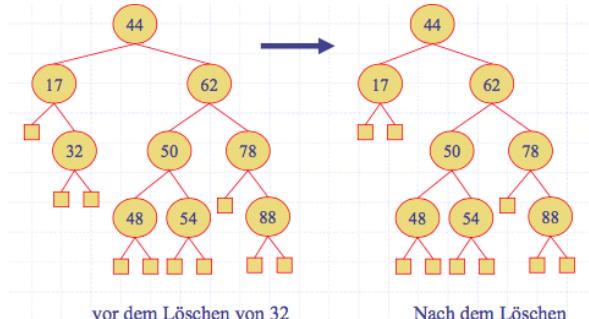
- ◆ Komplexität

- beide Algorithmen haben die selbe Zeitkomplexität

## Löschen aus einem AVL Baum

Das Löschen beginnt wie im binären Suchbaum. D.h. der gelöschte Knoten wird ein leerer externer Knoten. Sein Eltern-Knoten, w, kann jetzt die Balance aus dem Gleichgewicht bringen.

### Beispiel



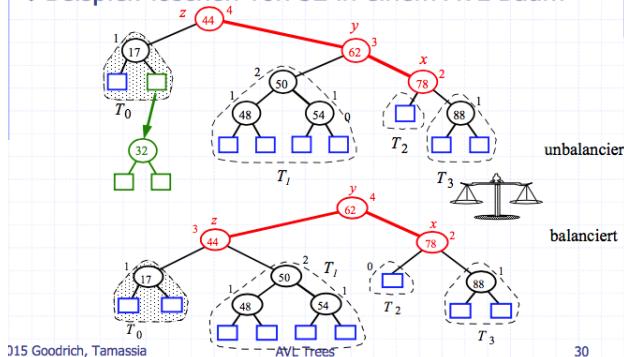
### Balancierung nach dem Löschen

Sei z der erste unbalancierte Knoten, während man w den Baum nach oben traversiert. Sei y das Kind von z mit der grösseren Höhe und x das Kind von y mit der grösseren Höhe.

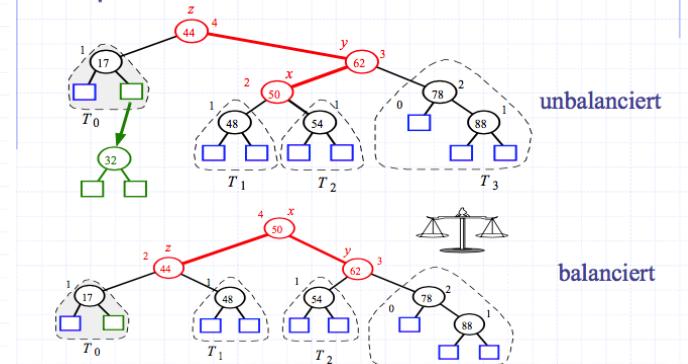
Dann sollte der Aufruf von `restructure(x)` folgen um die Balance von z herzustellen.

Die Umstrukturierung kann eine neue Unbalance hervorruhen bei Knoten höher im Baum. Somit muss die Balance weiter geprüft werden bis die Wurzel von T erreicht ist.

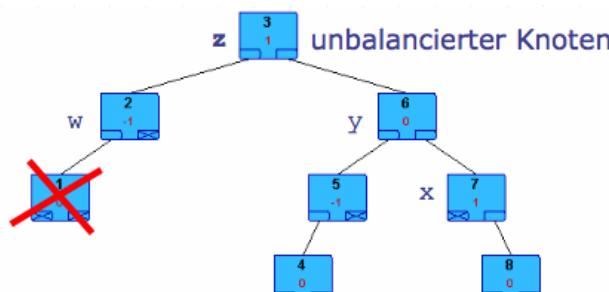
#### Beispiel: löschen von 32 in einem AVL Baum



#### Beispiel: löschen von 32 in einem AVL Baum

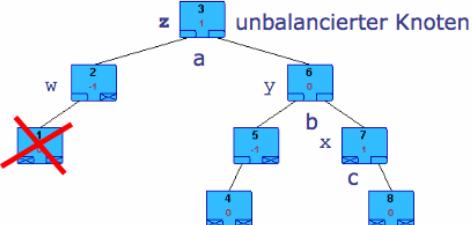
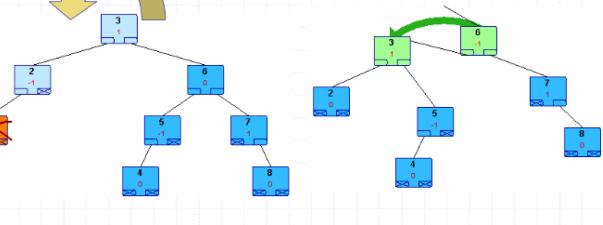
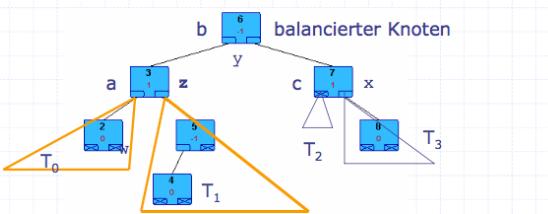
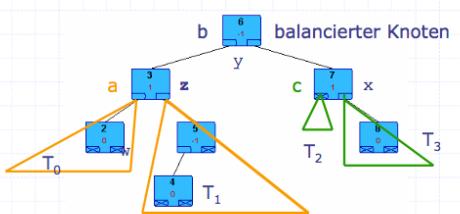


## Löschen eines externen Knotens

**Beispiel delete(1)**

Sei  $z$  der erste unbalancierte Knoten, auf den man bei der Traversierung von  $w$  aus den Baum aufwärts trifft. Sei  $y$  jenes Kind von  $z$  mit der grösseren Höhe (evtl. mehrdeutig) und sei  $x$  das Kind von  $y$  mit der grösseren Höhe (evtl. mehrdeutig).

Beachte bei diesem Beispiel ist Balance = Höhe (rechts) – Höhe (links).

<p><b>Beispiel: delete(1)</b></p>  <p>unbalancierter Knoten</p>	<p><b>Beispiel: delete(1)</b></p>  <p>Linksrotation</p>
<p><b>Beispiel: delete(1)</b></p>  <p>balancierter Knoten</p> <p>2: Ersetze den (rechten) Unterbaum beim rotierten <math>z=3</math> mit dem Unterbaum beim rotierten <math>b=6</math>.</p> <p>2a: Setze <math>a</math> als linkes Kind von <math>b</math> und <math>T_0, T_1</math> als linken und rechten Unterbaum von <math>a</math>.</p>	<p><b>Beispiel: delete(1)</b></p>  <p>balancierter Knoten</p> <p>2: Ersetze den (rechten) Unterbaum beim rotierten <math>z=3</math> mit dem Unterbaum beim rotierten <math>b=6</math>.</p> <p>2b: Setze <math>c</math> als rechtes Kind von <math>b</math> und <math>T_2, T_3</math> als linken und rechten Unterbaum von <math>c</math>.</p>

## Implementierung

Gemäss Goodrich und Tamassia. In der Übung wurde eine andere Implementierung angestrebt. Dies ist einem separaten Dokument zu entnehmen.

```

public class AVLItem extends Item {
    int height;
    AVLItem(Object k, Object e, int h) {
        super(k, e);
        height = h;
    }
    public int height () {
        return height;
    }
    public int setHeight(int h) {
        int oldHeight = height;
        height = h;
        return oldHeight;
    }
}

private boolean isBalanced(Position p) {
    // test whether node p has balance factor
    // between -1 and 1
    int bf=height(T.leftChild(p))- height(T.rightChild(p));
    return ((-1 <= bf) && (bf <= 1));
}

private Position tallerChild(Position p) {
    // return a child of p with height no
    // smaller than that of the other child
    if(height(T.leftChild(p)) >= height(T.rightChild(p)));
        return T.leftChild(p);
    else
        return T.rightChild(p);
}

public void insertItem(Object key, Object element) throws
    InvalidKeyException {
    super.insertItem(key, element); // may throw an
                                    // InvalidKeyException
    Position zPos = actionPos; // start at the insertion position
    T.replace(zPos, new AVLItem(key, element, 1));
    rebalance(zPos);
}

public Object removeElement(Object key) throws
    InvalidKeyException {
    Object toReturn = super.remove(key);
    // may throw an InvalidKeyException
    if(toReturn != NO_SUCH_KEY) {
        Position zPos = actionPos; //start at removal position
        rebalance(zPos);
    }
    return toReturn;
}

```

```

public class AVLTree extends BinarySearchTree
    implements Multimap {
    public AVLTree(Comparator c) {
        super(c);
        T = new RestructurableNodeBinaryTree();
    }
    private int height(Position p) {
        if (T.isExternal(p))
            return 0;
        else
            return ((AVLItem) p.element()).height();
    }
    private void setHeight(Position p) {
        // called only if p is internal
        ((AVLItem) p.element()).setHeight(
            1+Math.max(height(T.leftChild(p)),
            height(T.rightChild(p))));
    }

    private void rebalance(Position zPos) {
        //traverse the path of T from zPos to the root;
        //for each node encountered
        //recompute its height and
        //perform a rotation if it is unbalanced
        while (! T.isRoot(zPos)) {
            zPos = T.parent(zPos);
            setHeight(zPos);
            if (!isBalanced(zPos)) {
                Position xPos = tallerChild(tallerChild(zPos));
                zPos = ((RestructurableNodeBinaryTree)T)
                    .restructure(xPos);
                setHeight(T.leftChild(zPos));
                setHeight(T.rightChild(zPos));
                setHeight(zPos);
            }
        }
    }
}

```

In der Übung prüfen wir ob wir bereits darüber hinaus sind, also entfällt dort der Schritt mit T.parent und wird nach das if geschoben.

## Laufzeiten von AVL Bäumen

Eine einzelne **Restrukturierung** ist O(1) unter Benutzung eines verlinkten binären Baumes.

**Find** ist O(log n), da die Höhe eines Baumes O(log n) ist. In diesem Fall ist keine Restrukturierung nötig.

**Insert** ist O(log n). Das find am Anfang ist O(log n) und die Umstrukturierung Baumaufwärts (Sicherstellung der Höhen) ist ebenfalls O(log n).

**Delete** ist ebenfalls O(log n). Aus den gleichen Gründen wie bei insert.

# Splay Bäume

Splay Bäume sind wie der AVL Baum auch binäre Suchbäume. Bei einem Splay Baum wird nach einem Zugriff auf einen Knoten dieser zur Root bewegt (nach Update und Suchen!). Der tiefste interne zugegriffene Knoten wird zur Root (Spalying).

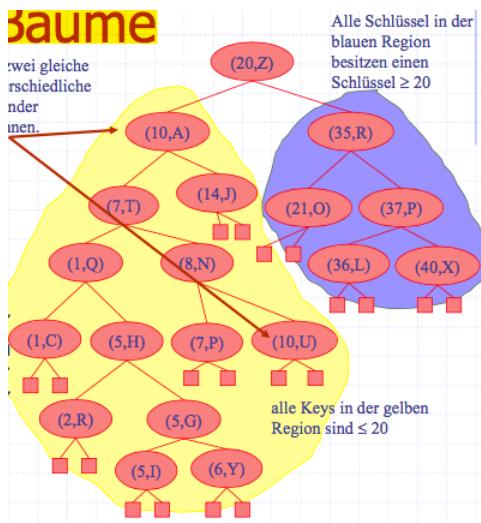
## Splaying-Costs O(h)

**Durchschnittlich:**  $O(\log n)$ , für oft besuchte Knoten sogar wesentlich schneller, da diese immernäher an die Root rücken.

**Worst-Case** Höhe eines Baumes ist  $n$ , somit  $O(n)$   $O(h)$  Rotationen, jede mit  $O(1)$

## Regeln BSB (Binären Suchbaum)

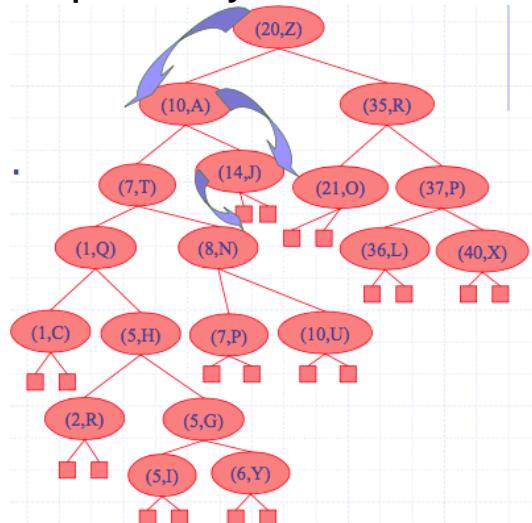
- Entries sind nur in internen Knoten gespeichert.
- Keys gespeichert im linken Teilbaum von  $v$  sind kleiner oder gleich wie der Key von  $v$ .
- Keys gespeichert im rechten Teilbaum von  $v$  sind grösser oder gleich wie der Key in  $v$
- Eine Inorder-Traversierung retourniert die Keys in geordneter Folge.



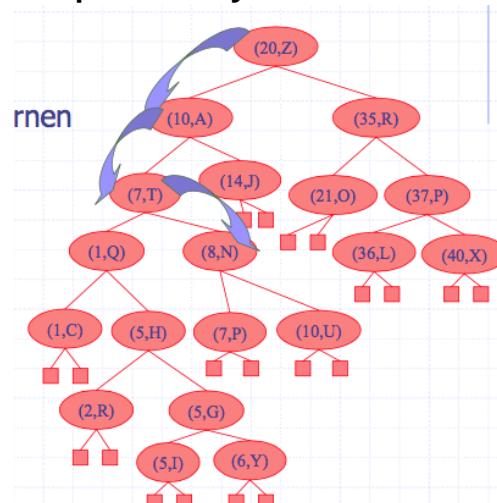
## Suchen in einem Splay Baum

Beim Suchen geht er den Baum abwärts bis zum gesuchten Entry oder einem externen Knoten.

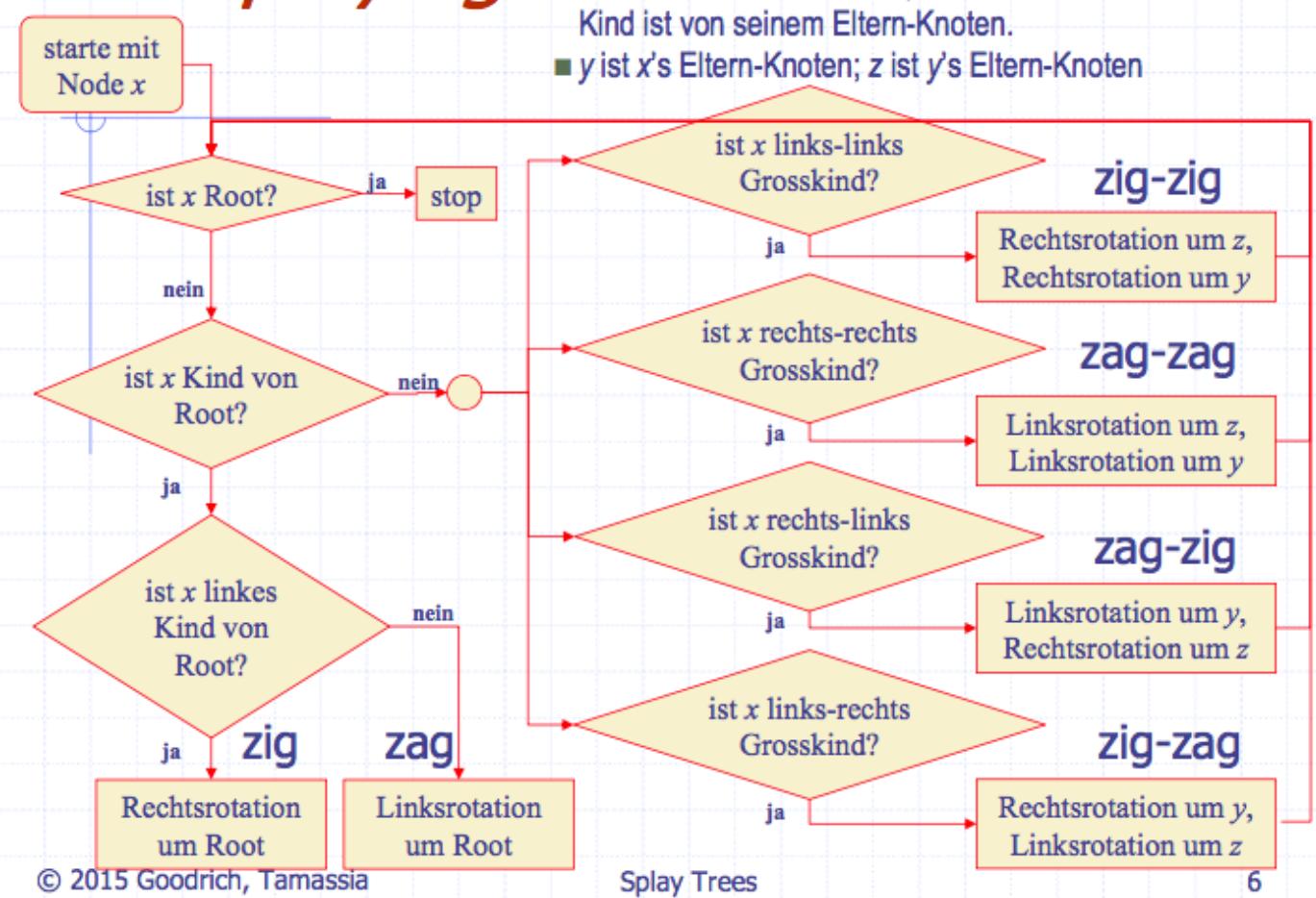
### Beispiel mit Key 11



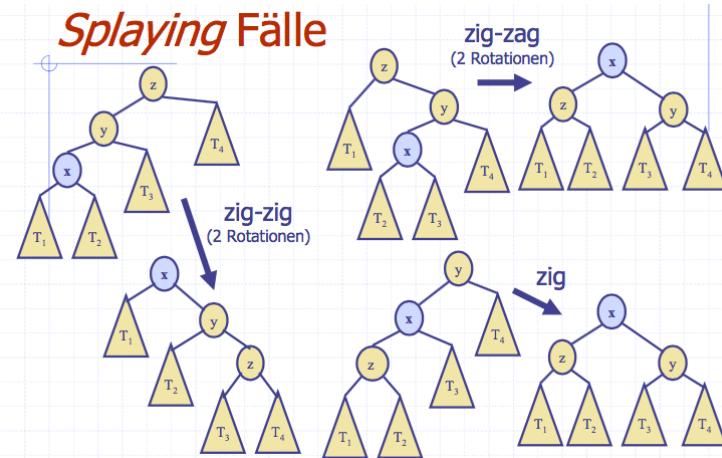
### Beispiel mit Key 8



# Splaying:



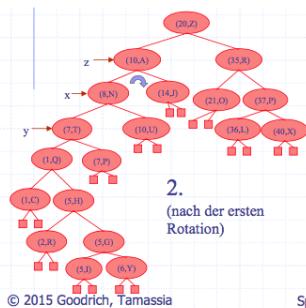
## Visualisierung der Splaying Fälle



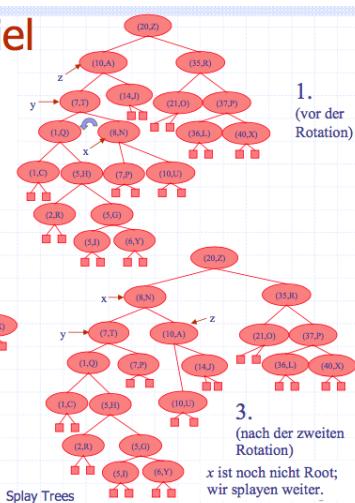
## Beispiel

### Splaying Beispiel

- sei  $x = (8, N)$ 
  - $x$  ist rechtes Kind; der Elternknoten ist linkes Kind des Grosseltern-Knoten
  - Linkssrotation um  $y$ , dann Rechtsrotation um  $z$



© 2015 Goodrich, Tamassia



1.  
(vor der Rotation)

2.  
(nach der ersten Rotation)

3.  
(nach der zweiten Rotation)

$x$  ist noch nicht Root;  
wir splayen weiter.

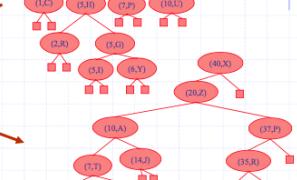
## Weiteres Beispiel

### Weiteres Beispiel

- Z.B. splay (40,X)

Nach erstem splay()  
(mit zwei Rotationen)

Beginn



10

© 2015 Goodrich, Tamassia

Splay Trees

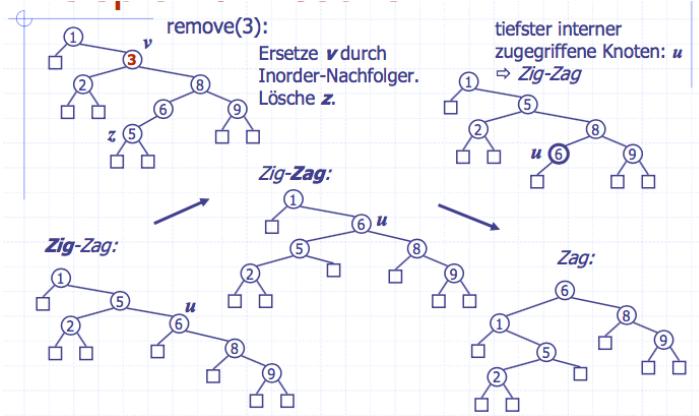
- Nun ist  $x$  linkes Kind von Root
- Rechtsrotation um Root

1.  
(vor der Rotation)

2.  
(nach der Rotation)

$x$  ist Root: STOP

## Beispiel für Löschen

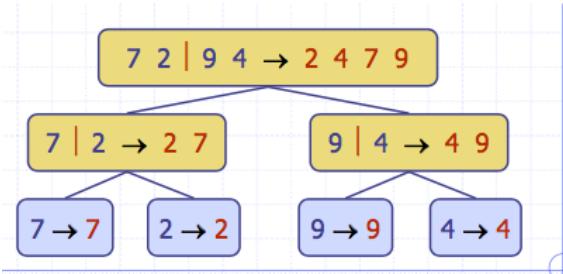


## Splay Bäume und Multi-/Maps

Welcher Knoten wird „splayed“ nach jeder Operation.

Methode	Splay Knoten
find(k)	wenn Key gefunden, benutze diesen Knoten wenn Key nicht gefunden, benutze den Eltern-Knoten des externen Knoten am Ende
insert(k,v)	Benutze den neuen Knoten bei welchem der Entry eingefügt/ersetzt wurde
remove(k)	Benutze den Eltern-Knoten des internen Knotens welcher gelöscht wurde

## Merge Sort (Sortierung durch Mischen)



Merge Sort basiert auf dem Pattern von Divide-and-Conquer. Daher wird dies zu Beginn nochmals kurz erklärt.

### Divide-and-Conquer

Divide-and-Conquer (Teile-und-Herrsche) ist ein generelles Paradigma beim Design von Algorithmen.

**Divide:** Input-Daten S in zwei getrennte Teilmengen S1 und S2 aufteilen.

**Recur (Wiederhole):** die Teilprobleme S1 und S2 rekursiv lösen.

**Conquer:** mischen der Lösungen von S1 und S2 in die Lösung S

Die Verankerung (Base Case) der Rekursion sind Teilprobleme der Grösse 0 oder 1 (Frage der Verankerung).

**Merge-Sort** ist ein Sortier-Algorithmus basierend auf dem Teile-und-Herrsche Paradigma. Wie beim Heap-Sort wird ein Comparator benutzt und er hat eine Laufzeit von  $O(n \log n)$ . Im Gegensatz aber nutzt er keine Priority-Queue und der Zugriff der Daten erfolgt sequentiell (geeignet zum Sortieren von Daten auf der Disk).

### Merge-Sort

```

Algorithm mergeSort(S, C)
  Input sequence S with n elements, comparator C
  Output sequence S sorted according to C   C = Comparator
  if S.size() > 1
    (S1, S2) ← partition(S, n/2)   aufteilung
    S1 ← mergeSort(S1, C)           trennen
    S2 ← mergeSort(S2, C)           bearbeiten
    S ← merge(S1, S2)             zusammenfügen
  return S
  
```

Immer wieder ein Rekursiver Aufruf der Methode

Ein Merge-Sort auf einer Input-Sequenz S mit n Elementen besteht aus drei Schritten:

**Divide:** S in zwei Sequenzen S1 und S2 von je  $n/2$  Elementen aufteilen

**Recur:** rekursiv S1 und S2 sortieren

**Conquer:** S1 und S2 in eine sortierte Sequenz mischen

### Mischen zweier sortierter Sequenzen

```

Algorithm merge(A, B)
  Input sequences A and B with n/2 elements each
  Output sorted sequence of A ∪ B
  S ← empty sequence
  while ¬A.isEmpty() ∧ ¬B.isEmpty()
    if A.first().element() < B.first().element()
      S.insertLast(A.remove(A.first()))
    else
      S.insertLast(B.remove(B.first()))
  while ¬A.isEmpty()
    S.insertLast(A.remove(A.first()))
  while ¬B.isEmpty()
    S.insertLast(B.remove(B.first()))
  return S
  
```

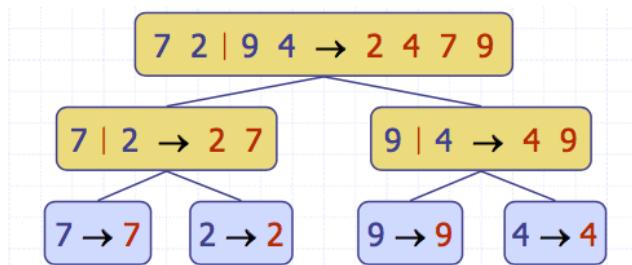
**Conquer:** Mischen von zwei sortierten Sequenzen A und B in die sortierte Sequenz S, enthaltend die Vereinigung der Elemente von A und B.

Mischen zweier sortierter Sequenzen vo je  $n/2$  Elementen mit double-linked Listen hat eine Laufzeit von  $O(n)$

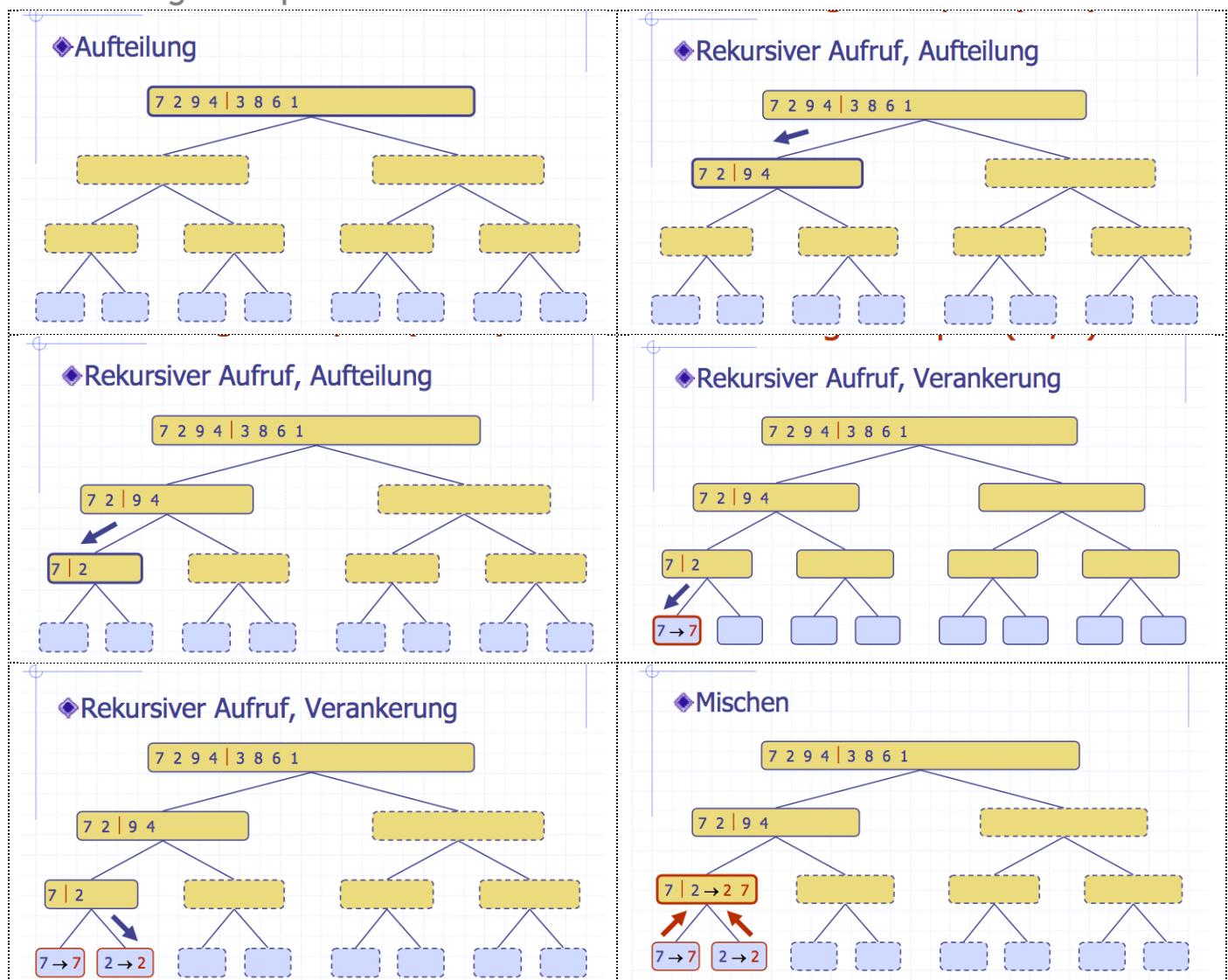
## Merge-Sort Baum

Die Ausführung eines Merge-Sort kann als binärer Baum dargestellt werden.

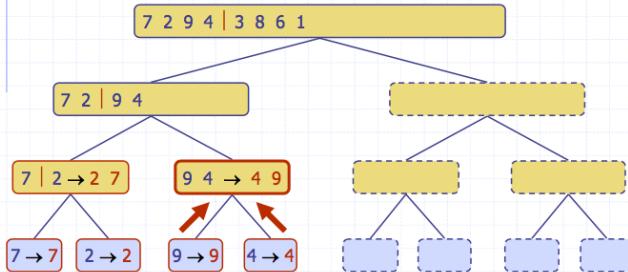
- Jeder Knoten repräsentiert einen rekursiven Aufruf des Merge-Sort und enthält
  - o Unsortierte Sequenz vor der Ausführung und der Aufteilung
  - o Sortierte Sequenz nach dem Ende der Ausführung
- Die Wurzel entspricht dem initialen Aufruf
- Die Blätter sind Aufrufe auf Teilsequenzen der Grösse 0 oder 1



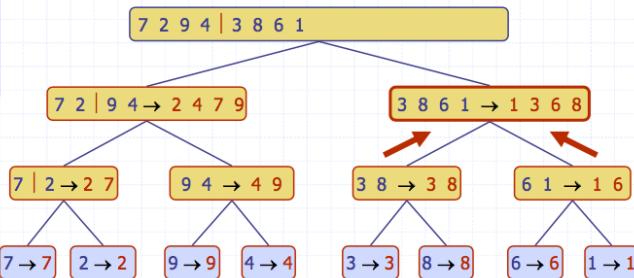
### Ausführungs-Beispiel



### Rekursiver Aufruf, ..., Verankerung, Mischen

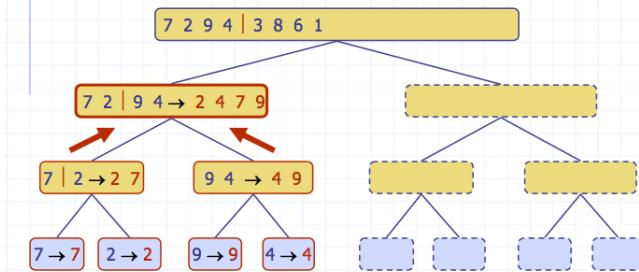


### Rekursiver Aufruf, ..., Mischen, Mischen

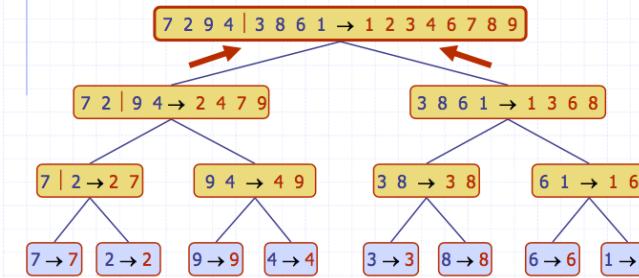


## Ausführungs-Beispiel (VIII/X)

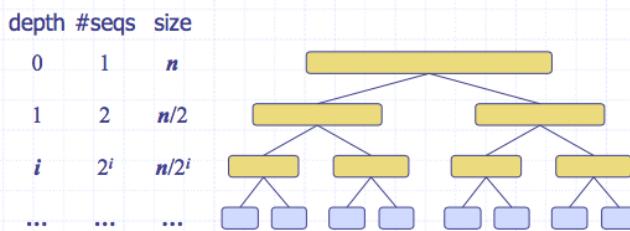
### Mischen



### Mischen



## Analyse von Merge-Sort



Die **Höhe** des Merge-Sort Baumes ist  **$O(\log n)$**  (bei jedem rekursiven Aufruf findet eine Aufteilung in zwei Hälften statt). Der **Gesamt-Aufwand aller Knoten einer Tiefe  $i$**  ist  **$O(n)$**  (Aufteilung und Mischen von  $2^i$  Sequenzen der Grösse  $n/2^i$ , daher  $2^{i+1}$  rekursive Aufrufe).

Somit ist die **totale Laufzeit** des Merge-Sort  **$O(n \log n)$**

## Zusammenfassung der Sortier-Algorithmen

Algorithmus	Zeitverhalten	Bemerkungen
selection-sort	$O(n^2)$	<ul style="list-style-type: none"> <li>langsam</li> <li>in-place</li> <li>für kleine Data Sets (&lt; 1K)</li> </ul>
insertion-sort	$O(n^2)$	<ul style="list-style-type: none"> <li>langsam</li> <li>in-place</li> <li>für kleine Data Sets (&lt; 1K)</li> </ul>
heap-sort	$O(n \log n)$	<ul style="list-style-type: none"> <li>schnell</li> <li>in-place</li> <li>für grosse Data Sets(1K – 1M)</li> </ul>
merge-sort	$O(n \log n)$	<ul style="list-style-type: none"> <li>schnell</li> <li>sequentieller Datenzugriff</li> <li>für riesige Data Sets(&gt; 1M)</li> </ul>

## Implementierung

zusammenfassen von 2, 4, 8 ... Elementen

```

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
    Object[] out = new Object[in.length]; // output array
    Object[] temp; // temp array reference used for swapping.
    int n = in.length;
    for (int i=1; i < n; i*=2) {
        // each iteration sorts all length-2*i runs
        for (int j=0; j < n; j+=2*i)
            // each iteration merges two length-i pair
            merge(in,out,c,j);
        // merge from in to out two length-i runs at j
        temp = in; in = out; out = temp;
        // swap arrays for next iteration
    }
    // the "in" array contains the sorted array, so re-copy it
    System.arraycopy(in,0,orig,0,in.length);
}

```

Zusammenfügen zweier Arrays zu einem

```

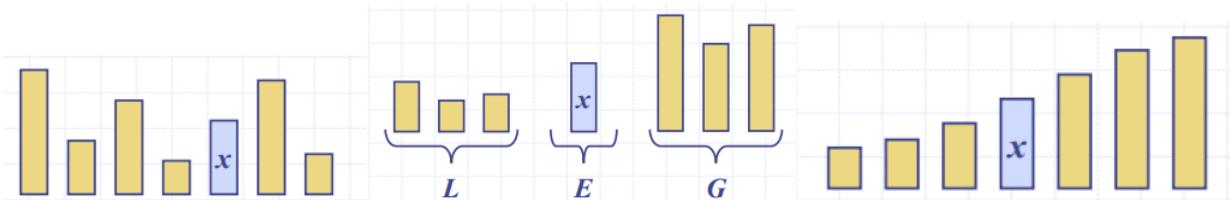
protected static void merge(Object[] in, Object[] out,
    Comparator c, int start, int inc) {
    // merge in[start,start+inc-1] and
    // in[start+inc,start+2*inc-1]
    int x = start; // index into run #1
    int end1 = Math.min(start+inc, in.length);
    // boundary for run #1
    int end2 = Math.min(start+2*inc, 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++];
    if (x < end1) // first run didn't finish
        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);
}

```

## Quick Sort

Quick-Sort ist ein Sortier-Algorithmus basierend auf dem Divide-and-Conquer Paradigma.

- Divide** Auswahl eines zufälligen Elementes  $x$  (genannt Pivot) und Aufteilung von  $S$  in  $L$   
Elemente kleiner als  $x$ ,  $E$  Elemente gleich  $x$  und  $G$  Elemente größer als  $x$ .
- Recur** sortiere  $L$  und  $G$
- Conquer** Vereine  $L$ ,  $E$  und  $G$ .



### Partitionierung

```

Algorithm partition(S, p) 
Input sequence  $S$ , position  $p$  of pivot
Output subsequences  $L, E, G$  of the
elements of  $S$  less than, equal to,
or greater than the pivot, resp.
 $L, E, G \leftarrow$  empty sequences
 $x \leftarrow S.\text{remove}(p)$ 
 $E.\text{insertLast}(x)$ 
while  $\neg S.\text{isEmpty}()$ 
 $y \leftarrow S.\text{remove}(S.\text{first}())$ 
if  $y < x$ 
 $L.\text{insertLast}(y)$ 
else if  $y = x$ 
 $E.\text{insertLast}(y)$ 
else {  $y > x$  }
 $G.\text{insertLast}(y)$ 
return  $L, E, G$ 

```

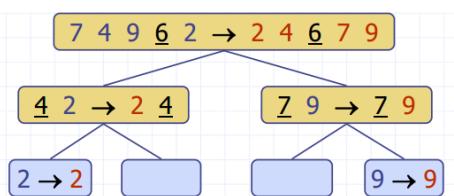
Dabei findet eine Aufteilung der Input-Sequenz statt. Es wird jeweils ein Element  $y$  von  $S$  entfernt und abhängig vom Vergleich mit dem Pivot  $x$  entweder in  $L$ ,  $E$  oder  $G$  eingefügt.

Jedes Einfügen und Entfernen findet jeweils am Anfang oder am Ende der Sequenz statt und hat eine Laufzeit von  $O(1)$ .

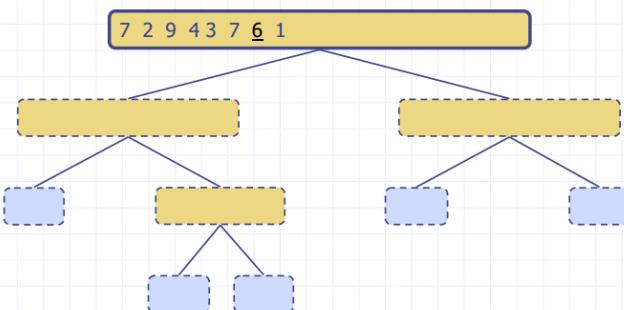
Die Aufteilung des Quick-Sort benötigt daher  $O(n)$ .

### Quick-Sort Tree

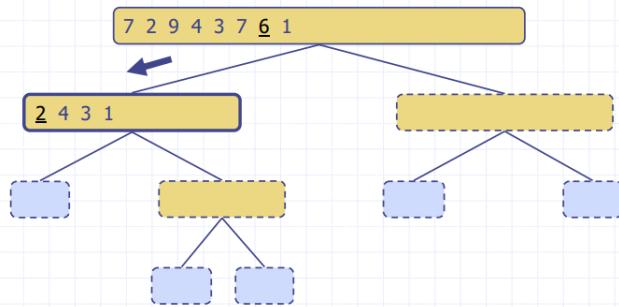
Die Ausführung eines Quick-Sort kann als binärer Baum dargestellt werden. Jeder Knoten repräsentiert einen rekursiven Aufruf des Quick-Sort und enthält eine unsortierte Sequenz vor der Ausführung und sein Pivot, sowie eine sortierte Sequenz und sein Pivot nach dem Ende der Ausführung. Die Wurzel entspricht dem initialen Aufruf. Die Blätter sind Aufrufe auf Teilsequenzen der Grösse 0 oder 1.



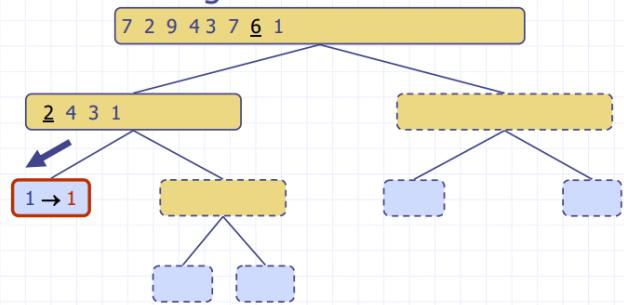
## ◆ Pivot-Selektion



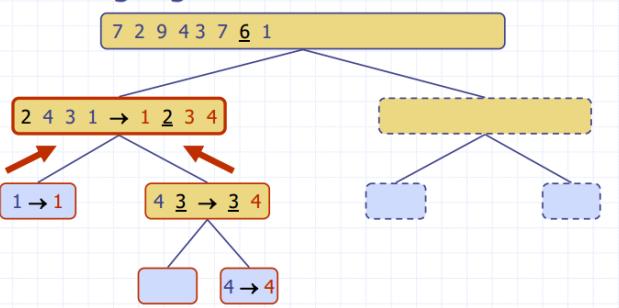
## ◆ Aufteilung, Rekursiver Aufruf, Pivot-Selektion



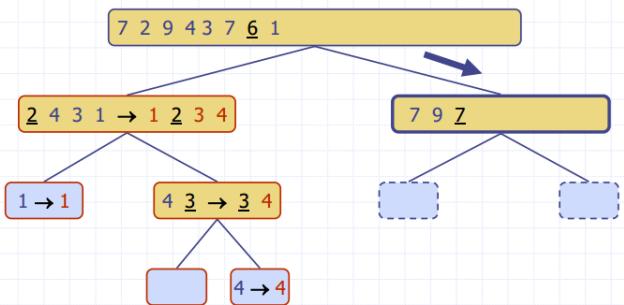
## ◆ Aufteilung, Rekursiver Aufruf, Verankerung



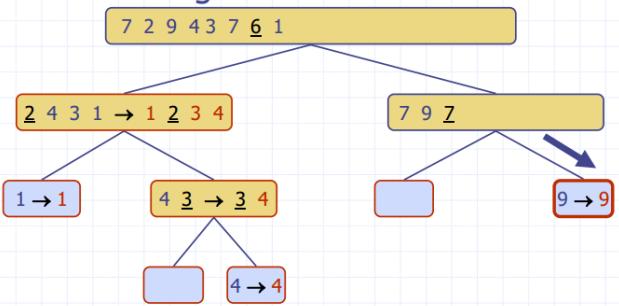
## ◆ Rekursiver Aufruf, ..., Verankerung, Vereinigung



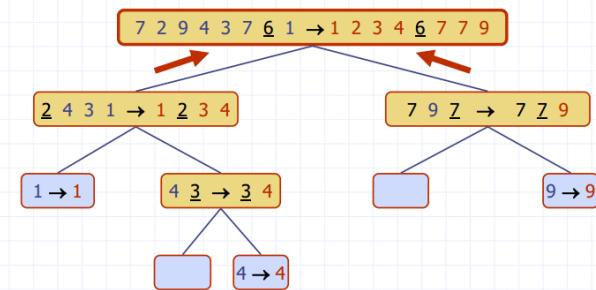
## ◆ Rekursiver Aufruf, Pivot-Selektion



## ◆ Aufteilung, ..., Rekursiver Aufruf, Verankerung



## ◆ Vereinigung, Vereinigung



## Worst-Case Laufzeitverhalten

Der Worst-Case des Quick-Sort tritt dann auf, wenn das Pivot das Minimum- oder Maximum-Element ist. Eines von L oder G hat dann in diesem Fall die Länge  $n - 1$ , das andere 0 (wenn alle Werte ungleich). Die Laufzeit ist somit proportional zu der Summe:

I der Summe.  
 $n + (n - 1) + \dots + 2 + 1 : \sum_{i=0}^n i = \frac{n^2 + n}{2}$

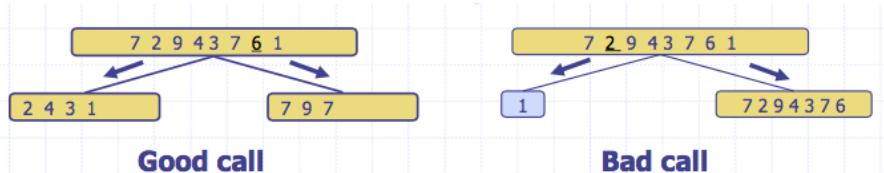
Quick Sort ist somit O(n²)

Der Worst-Case Laufzeit des Quick-Sort ist somit  $O(n^2)$ . Die Höhe ist in diesem Fall linear abhängig zu der Anzahl von Elementen.

## Erwartete Laufzeit

Als Beispiel ein rekursiver Aufruf von Quick-Sort auf einer Sequenz der Länge s:

- Good call: die Länge von L und G sind beide kleiner als  $3s/4$
- Bad call: entweder L oder G ist länger als  $3s/4$



Ein Aufruf von Good ist mit einer Wahrscheinlichkeit von  $\frac{1}{2}$  zu erreichen. Ein Pivot im ersten und im letzten Viertel sind schlecht, während jene in der Mitte gut sind.

Bei einem Knoten in der Tiefe  $i$  wird erwartet dass  $i/2$  Elternknoten „good calls“ sind und die Länge der Input-Sequenz des aktuellen Calls höchstens  $(3/4)^{i/2} * n$  ist.

Somit ist für einen Knoten der Tiefe  $2\log tief(4/3) * n$  die erwartete Input-Länge 1. Die erwartete Höhe des Quick-Sort Baumes ist  $O(\log n)$ . Der Gesamtaufwand für alle Knoten einer Tiefe ist  $O(n)$ . Somit ergibt sich eine totale Laufzeit von  $O(n \log n)$ .

## In-Place Quick-Sort

### Algorithm *inPlaceQuickSort(S, l, r)*

**Input** sequence  $S$ , ranks  $l$  and  $r$

**Output** sequence  $S$  with the elements of rank between  $l$  and  $r$  rearranged in increasing order

```

if  $l \geq r$ 
    return
i ← a random integer between  $l$  and  $r$ 
x ←  $S.\text{elemAtRank}(i)$ 
( $h, k$ ) ← inPlacePartition(x)
inPlaceQuickSort(S, l, h - 1)
inPlaceQuickSort(S, k + 1, r)

```

Quick-Sort kann auch „In-Place“ implementiert werden. Im Partitionierungs-Schritt werden die Elemente der Input-Sequenz derart umgeordnet, dass:

- Die Elemente kleiner als das Pivot einen Index kleiner als  $h$  haben
- Die Elemente gleich dem Pivot einen Index zwischen  $h$  und  $k$  haben.
- Die Elemente grösser als das Pivot einen Index grösser als  $k$  haben.

Die rekursiven Calls betrachten die Elemente mit dem Index kleiner als  $h$  und grösser als  $k$ .

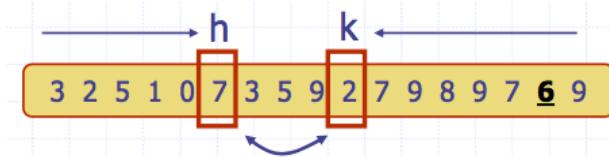
## In-Place Partitionierung

Bei der Partitionierung mit zweier Indices um  $S$  in  $L$  und  $E & G$  aufzuteilen. Eine ähnliche Methode kann auch  $E & G$  in  $E$  und  $G$  aufteilen).



Folgendes wiederholen bis  $h$  und  $k$  sich kreuzen:

- $H$  nach rechts bis zu einem Element  $\geq x$  schieben
- $K$  nach links bis zu einem Element  $< x$  schieben.
- Wenn sich  $h$  und  $k$  noch nicht gekreuzt haben, werden die Elemente mit den Indizes  $h$  und  $k$  vertauscht.



## Java Implementation (für ungleiche Elemente)

Gemäss Goodrich und Tamassia.

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

\*1

## Zusammenfassung der Sortier-Algorithmen

Algorithmus	Zeitverhalten	Bemerkungen
selection-sort	$O(n^2)$	♦ in-place ♦ langsam (gut für kleine Inputs)
insertion-sort	$O(n^2)$	♦ in-place ♦ langsam (gut für kleine Inputs)
quick-sort	$O(n \log n)$ expected	♦ in-place ♦ schnellster (gut für grosse Inputs)
heap-sort	$O(n \log n)$	♦ in-place ♦ schnell (gut für grosse Inputs)
merge-sort	$O(n \log n)$	♦ sequentieller Datenzugriff ♦ schnell (gut für riesige Inputs)

# Sorting Lower Bound

## Vergleichsbasierte Sortierung

Viele Sortier-Algorithmen sind vergleichsbasiert. Diese sortieren durch Vergleiche zwischen Paaren von Objekten. Beispiele dafür sind Bubble-Sort, Selection-Sort, Insertion-Sort, Heap-Sort, Quick-Sort...

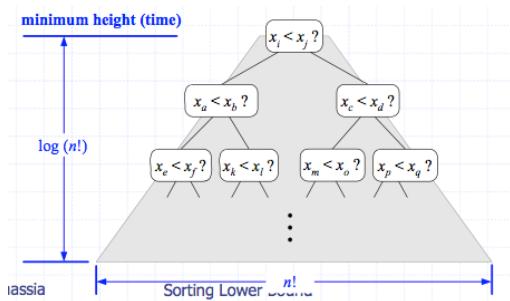
Es soll hier eine untere Grenze (Lower Bound) der Laufzeit hergeleitet werden für alle Algorithmen, welche Vergleiche benutzen um  $n$  Elemente  $x_1, x_2, \dots$  zu sortieren.

## Zählen der Vergleiche

Wir zählen dazu die Anzahl Vergleiche. Jeder mögliche Durchgang des Algorithmus korrespondiert mit einem Wurzel-zu-Blatt Pfad in einem Entscheidungs-Baum.

## Höhe des Entscheidungs-Baumes

Die Höhe des Entscheidungs-Baumes entspricht der unteren Grenze der Laufzeit. Jede mögliche Input-Permutation führt zu einem anderen Pfad. Wenn dies nicht so wäre, hätte ein Input ...4...5 die selbe Ausgangs-Ordnung wie ...5...4, was falsch wäre. Da  $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot m$  Blätter vorhanden sind, beträgt die Höhe mindestens  $\log(n!)$ .



## Die untere Grenze (Lower Bound)

Jeder Vergleichs-basierte Sortier-Algorithmus hat daher eine minimale Laufzeit von  $\log(n!)$ . Mit Umformungen lässt sich darauf schliessen dass er eine Laufzeit von  $O(n \log n)$  hat.

$$\log(n!) \sim \log\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) = \frac{n}{2} \log\left(\frac{n}{2}\right) \quad n! \geq \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \rightarrow O(n \log n)$$

## Bucket-Sort

```

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

```

Sei  $S$  eine Sequenz von  $n$  (Key,Element) Items mit Keys im Bereich von  $[0, N-1]$ . Bucket Sort benutzt die Keys als Index in einem Hilfs-Array  $B$  von Sequenzen (Buckets=Eimer).

### Phase 1

Sequenz  $S$  leeren durch verschieben jedes Items  $(k,o)$  in einen Bucket  $B[k]$

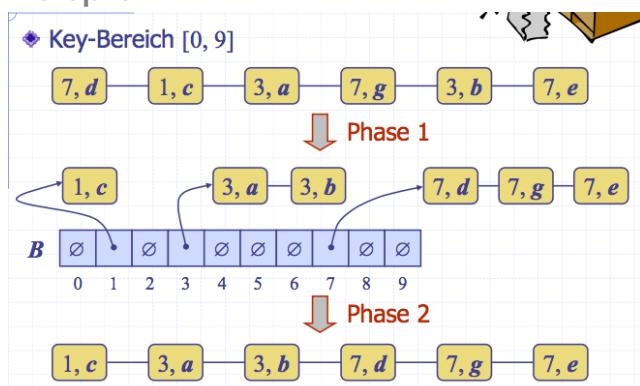
### Phase 2

Für  $i = 0, \dots, N-1$  verschiebe die Items des Buckets  $B[i]$  an das Ende der Sequenz  $s$ .

## Analyse

Phase 1 benötigt  $O(n)$  Zeit und die Phase 2 benötigt  $O(n+N)$  Zeit. Somit benötigt der gesamte Bucket-Sort  $O(n+N)$  Zeit.

## Beispiel



Das Beispiel verdeutlicht noch einmal dass der Wertebereich bekannt sein muss. Zudem ist es nur mit ganzen Zahlen mögliche, welche zudem noch positiv sind. Alles andere geht nicht.

Aber zu sortieren von Zahlen (z.B. Postleitzahlen) ist dieser extrem schnell.

## Eigenschaften und Erweiterungen

### Key Eigenschaften

Die Keys werden als Indices in einem Array benutzt und können somit wie bereits erwähnt nicht beliebige Objekte sein. Wie aufgefallen ist kein externer Comparator nötig.

### Stabile Sort Eigenschaft

Ein stabiles Sortierverfahren ist ein Sortieralgorithmus, der die Reihenfolge der Datensätze, deren Sortierschlüssel gleich sind, bewahrt. Also bei einem instabilen Sortieralgorithmus können zwei Datensätze mit identischen Schlüssel vertauscht werden.

### Erweiterungen

String-Keys aus einem Set  $D$  von möglichen Strings, wobei  $D$  eine konstante Größe hat (z.B. die Namen der 50 U.S. Staaten)

- sortiere  $D$  und berechne den Index  $r(k)$  jedes Strings  $k$  von  $D$  in der sortierten Sequenz
- füge Item  $(k, o)$  in Bucket  $B[r(k)]$  ein

### Integer Keys im Bereich $[a, b]$

- stecke Item  $(k, o)$  in Bucket  $B[k-a]$

## Lexikographische Ordnung

Ein d-Tupel ist eine Sequenz von d Keys ( $k_1, k_2, k_3, \dots, k_d$ ), wobei Key  $k_i$  als die i-te Dimension des Tupel bezeichnet wird.

**Beispiel:** die kartesischen Koordinaten eines Punktes im Raum sind ein 3-Tupel

Die lexikographische Ordnung von zwei d-Tupels ist rekursiv definiert als:

$$\begin{aligned} (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_d) &< (\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_d) \\ \Leftrightarrow \\ \mathbf{x}_1 &< \mathbf{y}_1 \vee \mathbf{x}_1 = \mathbf{y}_1 \wedge (\mathbf{x}_2, \dots, \mathbf{x}_d) < (\mathbf{y}_2, \dots, \mathbf{y}_d) \end{aligned}$$

Daher, die Tupel werden der Dimension nach verglichen. (Zuerst die erste Dimension, dann die Zweite, etc....).

## Lexikographische Sortierung

**Algorithm** *lexicographicSort(S)*

**Input** sequence  $S$  of  $d$ -tuples

**Output** sequence  $S$  sorted in  
lexicographic order

**for**  $i \leftarrow d$  **downto** 1  
**stableSort**( $S, C_i$ )

Sei  $C_i$  der Comparator welcher zwei Tupel nach ihrer i-ten Dimension vergleicht.

Sei  $\text{stableSort}(S, C)$  ein stabiler Sortier-Algorithmus welcher den Comparator  $C$  benutzt.

Die Lexikographische-Sortierung sortiert eine Sequenz von  $d$ -Tupeln in lexikographischer Ordnung, indem  $d$ -mal  $\text{stableSort}$ , je einmal pro Dimension, durchgeführt wird. Die Lexikographische-Sortierung läuft in  $O(d*T(n))$  Zeit, wobei  $T(N)$  die Laufzeit von  $\text{stableSort}$  ist.

### Beispiel

(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)

## Radix Sort

### Algorithm *radixSort(S, N)*

```

Input sequence S of d-tuples such
that  $(0, \dots, 0) \leq (x_1, \dots, x_d)$  and
 $(x_1, \dots, x_d) \leq (N - 1, \dots, N - 1)$ 
for each tuple  $(x_1, \dots, x_d)$  in S
Output sequence S sorted in
lexicographic order
for i  $\leftarrow$  d downto 1
  bucketSort(S, N)
  
```

Radix-Sort ist eine Spezialisierung des lexikographischen-Sort, welcher Bucket-Sort als stabilen Sortier-Algorithmus für jede Dimension benutzt.

Es ist anwendbar für Tupel mit Integer-Keys im Bereich [0,N-1] in jeder Dimension i. Radix-Sort läuft in  $O(d(n+N))$  Zeit.

## Radix-Sort für binäre Zahlen

### Algorithm *binaryRadixSort(S)*

```

Input sequence S of b-bit
integers
Output sequence S sorted
replace each element x
of S with the item  $(0, x)$ 
for i  $\leftarrow$  0 to b - 1
  replace the key k of
  each item  $(k, x)$  of S
  with bit  $x_i$  of x
  bucketSort(S, 2)
  
```

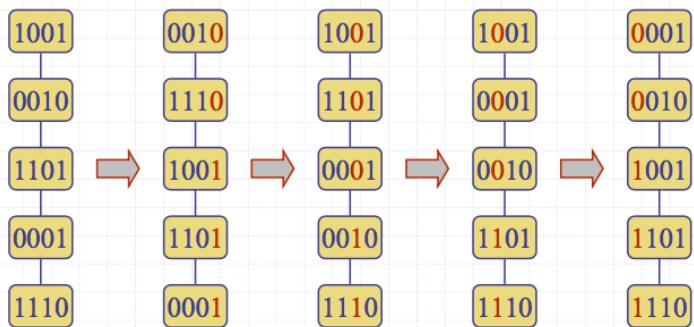
Gegeben sei eine Sequenz von *b*-Bit Integers.  $X = x_{b,1} \dots x_1 x_0$ . Jedes Element wird als *b*-Tupel von Integern im Bereich {0,1} dargestellt. Darauf wendet man den Radix-Sort mit  $N = 2$  an.

Diese Anwendung des Radix-Sort-Algorithmus läuft in  $O(b*n)$  Zeit.

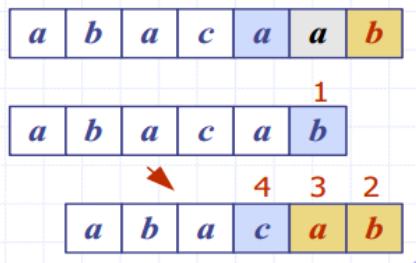
Beispielsweise kann eine Sequenz von 32-Bit Integers in linearer Zeit sortiert werden.

## Beispiel

Sortierung einer Sequenz von 4-Bit Integers.



## Pattern Matching



### Strings

Ein String (Zeichenkette) ist eine Sequenz von Characters (Zeichen). Beispiele dafür sind: ein Java Programm, ein HTML Dokument, eine DNA Sequenz oder ein digitales Bild.

#### Definition

Ein Alphabet  $\Sigma$  ist ein Set von möglichen Zeichen für eine Familie von Strings. Beispiele für Alphabete: ASCII, Unicode, {0,1} oder {A,C,G,T} (DNA Grundbausteine).

Sei  $P$  ein String der Länge  $m$ . Ein Substring  $P[i..j]$  von  $P$  ist die Subsequenz von  $P$ , bestehend aus den Zeichen mit Index zwischen und inklusiv  $i$  und  $j$ . Ein Präfix von  $P$  ist ein Substring vom Typ  $P[0..i]$ . Ein Suffix von  $P$  ist ein Substring vom Typ  $P[i..m-1]$ .

### Pattern Matching Algorithmen

Gegeben sind Strings  $T$  (Text) und  $P$  (Pattern, Muster). Das Pattern Matching Problem besteht darin, einen Substring in  $T$  zu finden, der mit  $P$  übereinstimmt.

Dies findet Anwendung in Texteditoren, Suchmaschinen und der biologischen Forschung.

#### Brute-Force Algorithmus

##### Algorithm BruteForceMatch( $T, P$ )

**Input** Text  $T$  der Länge  $n$  und Pattern  $P$  der Länge  $m$

**Output** Startindex eines Substrings von  $T$ , welcher mit  $P$  übereinstimmt, oder  $-1$  falls kein solcher Substring existiert.

```
for i ← 0 to n - m
  { testen der i'ten Verschiebung des Patterns }
  j ← 0
  while j < m ∧ T[i + j] = P[j]
    j ← j + 1
  if j = m
    return i { match bei i }
  return -1 { kein match !}
```

Der Brute-Force Pattern Matching Algorithmus vergleicht das Pattern  $P$  mit dem Text  $T$  für jede mögliche Position von  $P$  relativ zu  $T$ , bis entweder eine Übereinstimmung gefunden wurde oder alle möglichen Platzierungen des Patterns ausprobiert wurden.

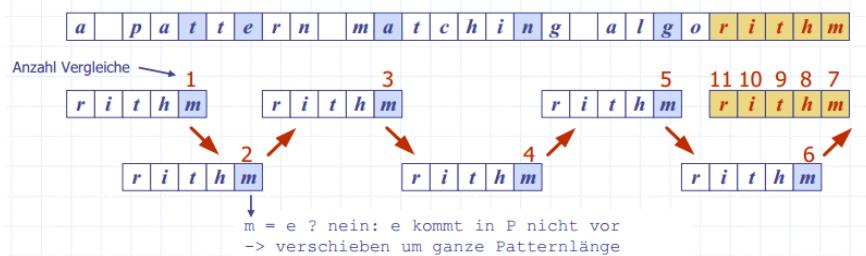
Das Brute-Force Pattern Matching benötigt  $O(nm)$  Zeit.

#### Worst Case Beispiel

$T = aaa \dots ah$

$P = aaah$

Worst Cases können in Bild-Analysen und DNA-Sequenzanalysen eintreten. In sprachlichen Texten eher nicht (sofern das Suchmuster irgendwie sinnvoll ist).



Der Boyer-Moore Pattern Matching Algorithmus basiert auf zwei Heuristiken.

### «Looking-Glass» Heuristik

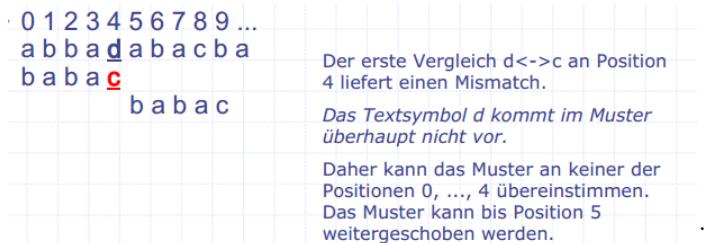
Vergleiche P mit einer Subsequenz von T. Starte dabei am Ende des Patterns.

### «Character-Jump» Heuristik

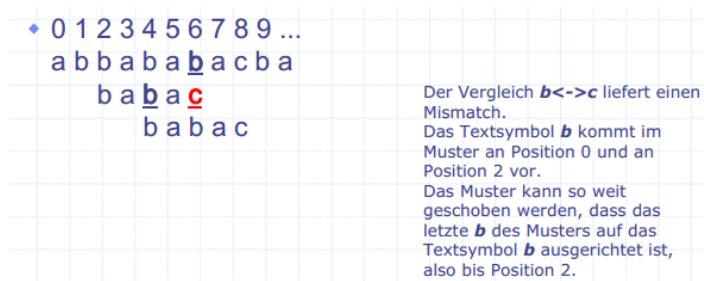
falls bei  $T[i] = c$  keine Übereinstimmung

- Falls P das Zeichen c enthält, verschiebe P bis das letzte Auftreten von c in P mit  $T[i]$  übereinstimmt.
- Sonst, verschiebe P bis  $P[0]$  mit  $T[i+1]$  übereinstimmt. («Schlechtes Zeichen-Strategie»)

1) Ist bereits das erste verglichene Textsymbol ein Symbol, das im Muster überhaupt nicht vorkommt, so kann das Muster um m Positionen hinter dieses Symbol weitergeschoben werden.



2) Falls eine Ungleichheit auftritt, das fehlerhafte Zeichen aber an anderer Stelle im Muster vorkommt, dann kann das Muster nur so weit geschoben werden, bis dieses Vorkommen («die andere Stelle») auf das Textsymbol ausgerichtet ist.



Nicht immer liefert die «Schlechtes Zeichen»-Strategie ein gutes Ergebnis.

In folgender Situation hat der Vergleich a-b einen Mismatch ergeben. Eine Ausrichtung des letzten Vorkommens des a im Muster auf das a im Text würde eine negative Verschiebung ergeben. Man behilft sich indem man stattdessen um 1 schiebt.

0	1	2	3	4	5	6	7	8	9	...
a	b	a	a	b	a	b	a	c	b	a
c	a	<b>b</b>	a	<b>b</b>						
c	a	b	a	b						
	c	a	b	a	b					

Besser ist es, in diesem Fall die grösstmögliche Schiebedistanz aus der Struktur des Musters abzuleiten. Die Schiebedistanz richtet sich danach, ob das Suffix, das übereinstimmt hat, noch an anderer Stelle im Muster vorkommt. Diese Vorgehensweise heisst «Gutes-Ende»- Strategie (good suffix heuristics).

### «Gutes Ende»-Strategie (good suffix heuristics)

Das Suffix ab hat bereits übereingestimmt. Das Muster kann so weit geschoben werden, bis das nächste Vorkommen von ab im Muster auf die Textsymbole ab ausgerichtet ist.

0	1	2	3	4	5	6	7	8	9	...
a	b	<u>a</u>	<u>a</u>	<u>b</u>	a	b	a	c	b	a
c	a	<u>b</u>	<u>a</u>	<u>b</u>						
c	<u>a</u>	<u>b</u>								

### Die «Last-Occurrence» Funktion

Boyer-Moore's Algorithmus analysiert zuerst das Pattern P und das Alphabet  $\Sigma$  um die «last-occurrence» Funktion L aufzubauen. Diese bildet  $\Sigma$  auf Integers ab, wobei  $L(c)$  wie folgt definiert ist.

$$L : \Sigma \rightarrow \mathbb{N}_0 \cup \{-1\}$$

$$L : c \rightarrow L(c) = \max_i \{ i \mid P[i] = c \} \text{ falls } c \text{ in } P \text{ vorkommt, sonst } -1$$

### Beispiel

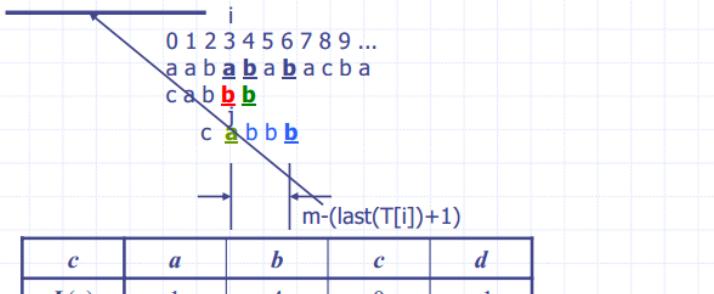
$\Sigma = \{a, b, c, d\}$	<table border="1"> <thead> <tr> <th>c</th><th>a</th><th>b</th><th>c</th><th>d</th></tr> </thead> <tbody> <tr> <td><math>L(c)</math></td><td>4</td><td>5</td><td>3</td><td>-1</td></tr> </tbody> </table>	c	a	b	c	d	$L(c)$	4	5	3	-1
c	a	b	c	d							
$L(c)$	4	5	3	-1							
$P = abacab$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

Die Funktion L(c) lässt sich darstellen als ein Array, dessen Indices durch numerische Werte des Alphabets gegeben sind. Die Funktion lässt sich in  $O(m+s)$  berechnen, wobei m die Länge von P und s die Anzahl Zeichen in  $\Sigma$  ist.

Zuerst betrachten wir den Fall, dass das Zeichen  $T[i]$  im Pattern auch vorkommt ( $\text{last}(T[i]) > -1$ ).

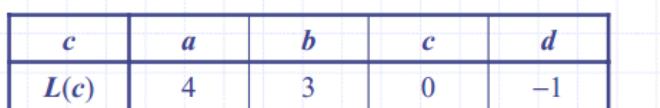
In diesem Fall müssen wir bis zum letzten Auftreten des Zeichens ( $T[i]$  im Pattern P) verschieben. Die Berechnung des neuen i's ergibt sich zu:

$$i = i + m - (\text{last}(T[i]) + 1) \quad m = 5 : i = 3 + 5 - (\text{last}(a) + 1) = 8 - 2 = 6 \text{ (neues } i\text{)}$$



Nun betrachten wir den Fall, dass das Zeichen  $T[i]$  im Pattern zwar vorkommt ( $\text{last}(T[i]) > -1$ ), aber bereits vorbei ist. In diesem Fall müssen wir das Pattern um eine Stelle nach vorne verschieben. Die Berechnung des neuen i's ergibt sich zu:

$$i = i + m - j \quad m = 5 : i = 3 + 5 - 2 = 8 - 2 = 6 \text{ (neues } i\text{)}$$



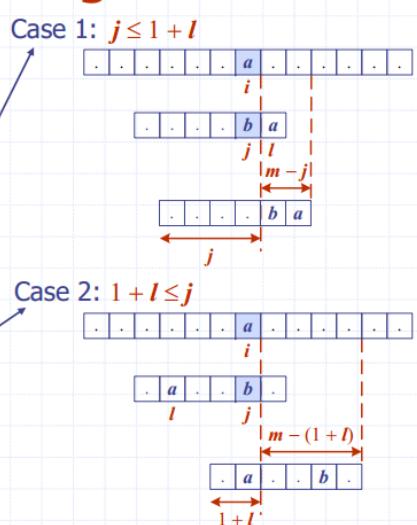
c	a	b	c	d
$L(c)$	4	3	0	-1

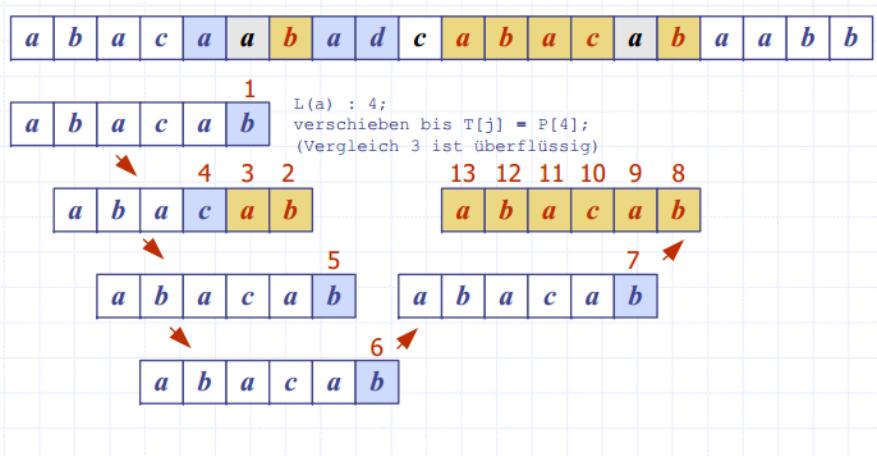
### Der Boyer-Moore Algorithmus

```

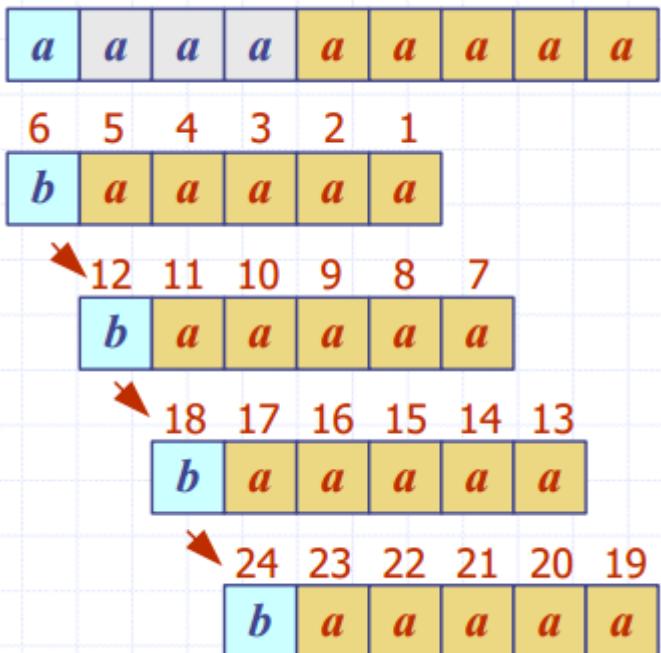
Algorithm BoyerMooreMatch( $T, P, \Sigma$ )
   $L \leftarrow \text{lastOccurrenceFunction}(P, \Sigma)$ 
   $i \leftarrow m - 1$  {actual T-index}
   $j \leftarrow m - 1$  {actual P-index}
  repeat
    if  $T[i] = P[j]$ 
      if  $j = 0$ 
        return  $i$  {match at  $i$ }
      else
         $i \leftarrow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

```





## Analyse



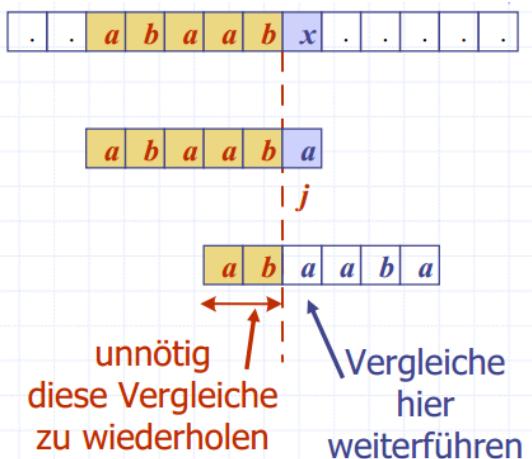
Der Boyer-Moore Algorithmus benötigt  $O(n*m + s)$  Zeit.

## Beispiel für einen Worst-Case

Der Worst-Caste kommt vor allem bei Bild- und DNA-Sequenzen vor und ist bei Text unwahrscheinlich.

Die Boyer-Moore Algorithmus ist signifikant schneller als der Brute-Force Algorithmus (angewandt auf Textanalyse).

## Knuth-Morris-Pratt Algorithmus



Der Knuth-Morris-Pratt Algorithmus vergleicht das Muster gegen den Text von links-nach-rechts, aber schiebt das Muster intelligenter als der Brute-Force Algorithmus.

Bei Nichtübereinstimmung: Was ist das Maximum um das Muster zu verschieben um redundante Vergleiche zu vermeiden.

Antwort: Der längste Präfix von P [0..j] der gleichzeitig Suffix von P [1..j] ist.

## KMP Fehl-Funktion

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

The diagram illustrates the step-by-step extraction of suffixes from the word "die".

- Step 1:** The word "die" is shown as a sequence of four boxes: **a**, **b**, **a**, **b**. A vertical dashed line is positioned after the second box.
- Step 2:** The suffix **'on** is extracted, leaving the prefix **r** and the suffix **fix**. The extracted suffix is shown in a separate row: **a**, **b**, **a**, **b**, **a**.
- Step 3:** The suffix **'rffix** is extracted, leaving the prefix **'ce**. The extracted suffix is shown in a separate row: **a**, **b**, **a**, **a**, **b**, **a**.
- Step 4:** The suffix **'ce** is extracted, leaving the prefix **j**. The extracted suffix is shown in a separate row: **a**, **b**, **a**, **a**, **b**, **a**.
- Step 5:** The final extracted suffix is **j**.

## Algorithmus im Pseudocode

**Algorithm *KMPMatch*( $T, P$ )**

```

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

```

In einer Vorlaufsphase (Preprocessing) sucht der Algorithmus Übereinstimmungen von Präfixes des Musters im Muster selbst.

Die Fehl-Funktion (failure function)  $F(j)$  ist definiert als die Grösse des längsten Präfixes von  $P[0..j]$ , so dass dieser auch Suffix von  $P [1..j]$  ist.

KMP modifiziert den Brute-Force Algorithmus so, dass bei einer Differenz  $P[j]$  nicht gleich  $T[i]$  der Index  $j$  gesetzt wird mit  $j \leftarrow F(j-1)$ .

Die «failure function» kann als Array dargestellt werden, in  $O(m)$  Zeit.

Bei jeder Iteration der while-Schleife wird entweder: i um eines erhtzt oder die Verschiebung  $i-j$  nimmt um mindestens 1 zu. ( $F(j-1) < j$ ).

Somit ergeben sich maximal  $2n$  Iterationen in der while-Schleife.

Der KMP Algorithmus benötigt  $O(m+n)$  Zeit.

## Fehlfunktion im Pseudocode

**Algorithm** *failureFunction(P)*

```

F[0] ← 0
i ← 1
j ← 0
while i < m
    if P[i] = P[j]
        {we have matched j + 1 chars}
        F[i] ← j + 1
        i ← i + 1
        j ← j + 1
    else if j > 0 then
        {use failure function to shift P}
        j ← F[j - 1]
    else
        F[i] ← 0 {no match}
        i ← i + 1

```

a	b	a	a	b	a
---	---	---	---	---	---

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

**Berechnung der Fehl-Funktion:**

- 1) Bestimmen aller Ränder aller Pattern-Substrings  
(Rand: längster Präfix, der gleichzeitig auch Suffix ist)
- 2) Bestimmen der maximalen Länge dieser Ränder
- 3) F(j) entspricht der Länge des längsten Randes des Substrings P[0..j]

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

**Beispiel**

a	b	a	c	a	a	b	a	c	c	a	b	a	b	b
1	2	3	4	5	6									
a	b	a	c	a	b									
						7								
a	b	a	c	a	b									
						8	9	10	11	12				
a	b	a	c	a	b									
						13								
a	b	a	c	a	b									
						14	15	16	17	18	19			
a	b	a	c	a	b									

**Java-Beispiel**

```
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)) {
            System.out.print(" match: " + pattern.charAt(j));
            if (j == m - 1) {
                System.out.print('\n');
                return i - m + 1; // match
            }
            i++;
            j++;
        } else if (j > 0) {
            System.out.print(" fail(" + (j-1) + "): " );
            System.out.print(j);
        } else {
            i++;
            System.out.print(" j == 0 -> i: " + i);
        }
    }
    return -1; // no match
}
```

a	b	a	a	b	a	a	b	a
j	0	1	2	3	4	5	6	7
P[j]	a	b	a	a	b	a	a	b

Ränder-Länge: 0 1 2 3 4 5 6 max.Länge

P[0]=a	{}	0
P[1]=ab	{ } b	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 aba	4
P[7]=abaabaab	{ } ab abaab	5
P[8]=abaababa	{ } a aba abaaba	6

**Java-Beispiel:**

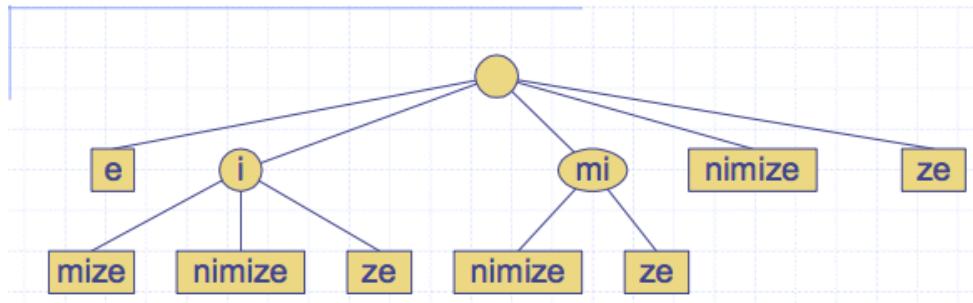
Aufruf von KMPMacht() mit:  
text: abacaabaccabacabaabb  
pattern: abacab

Resultat von KMPmatch(): 10

```
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;
}
```

/\* Session-Log:  
fail = 001012  
i = 0 j = 0 match: a  
i = 1 j = 1 match: b  
i = 2 j = 2 match: a  
i = 3 j = 3 match: c  
i = 4 j = 4 match: a  
i = 5 j = 5 fail(4): 1  
i = 5 j = 1 fail(0): 0  
i = 5 j = 0 match: a  
i = 6 j = 1 match: b  
i = 7 j = 2 match: a  
i = 8 j = 3 match: c  
i = 9 j = 4 fail(3): 0  
i = 9 j = 0 j == 0 -> i: 10  
i = 10 j = 0 match: a  
i = 11 j = 1 match: b  
i = 12 j = 2 match: a  
i = 13 j = 3 match: c  
i = 14 j = 4 match: a  
i = 15 j = 5 match: b

## Tries



### Preprocessing Strings

Durch Vorverarbeitung (Preprocessing) des Musters wird eine Geschwindigkeitsverbesserung beim Suchen erzielt. Nach Vorverarbeitung des Muster erzielt der KMP Algorithmus eine Geschwindigkeit, die proportional zur Text-Grösse ist. Ist der Text gross, unveränderlich und wird oft durchsucht, könnte anstelle des Musters der Text vorverarbeitet werden.

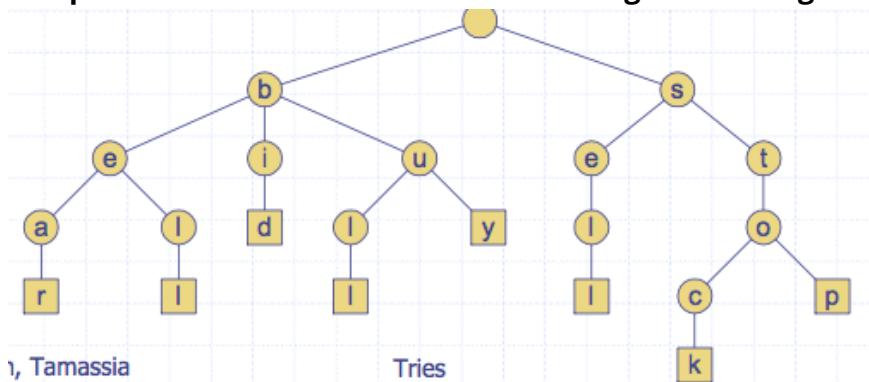
Ein Trie ist eine kompakte Datenstruktur für die Repräsentation einer Menge von Strings, wie z.B. alle Wörter eines Textes. Tries erlauben Pattern Matching mit einer Geschwindigkeit, welche proportional zur Grösse des Patterns ist.

### Standard Tries

Der Standard-Trie für eine Menge von Strings  $S$ , ist ein geordneter Baum, so dass:

- Jeder ausser dem Wurzel-Knoten ein Zeichen hat
- Die Kinder eines Knoten alphabetisch geordnet sind
- Die Pfade von den externen Knoten zur Wurzel die Strings von  $S$  beinhalten

### Beispiel eines Standard-Tries für eine Menge von Strings



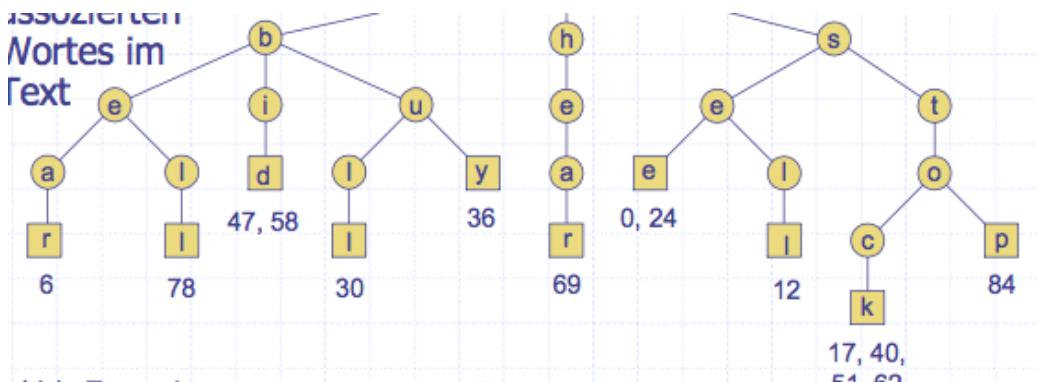
### Analyse des Standard-Tries

Ein Standard-Trie benötigt  $O(n)$  Speicher und unterstützt Suchen, Einfügen und Löschen in  $O(dm)$  Zeit wobei  $n$  = totale Länge der Strings in  $S$ ,  $m$  = Länge des String-Parameters der Operation und  $d$  = Grösse des Alphabets.

## Suche in einem Trie

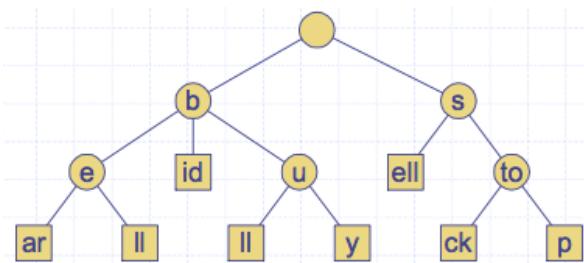
Einfügen der Wörter des Texts in einen Trie. Jedes Blatt speichert die Positionen des assoziierten Wortes im Text.

s	e	a	b	e	a	r	?	s	e	l	s	t	o	c	k	!							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
see	a	b	u	ll	?	bu	y	s	to	ck	!												
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	
b	i	d	st	ock	!	b	i	d	st	ock	!												
47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68		
h	e	a	ar	the	be	ll	?	s	to	p	!												
69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88				



## Kompromierte Tries

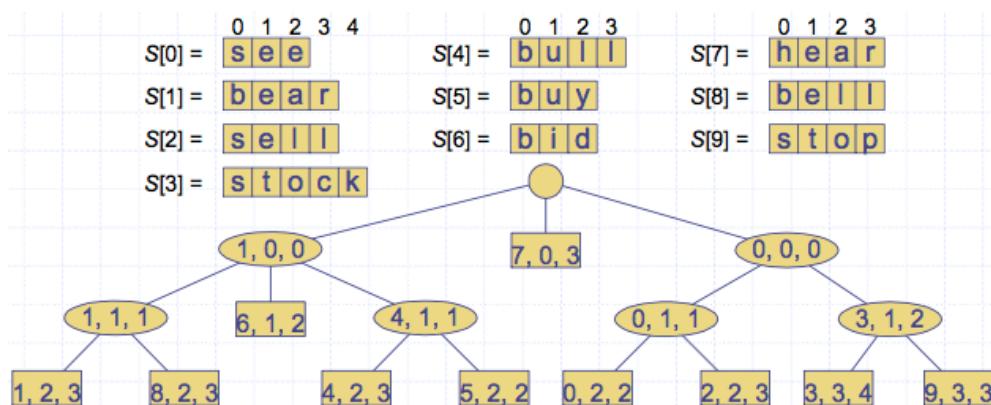
Ein komprimierter Trie (Compressed Trie) hat nur interne Knoten mit mindestens zwei Kindern. Hergleitet von einem Standard-Trie durch Komprimierung von Pfaden von redundanten Knoten.



## Kompackte Repräsentation

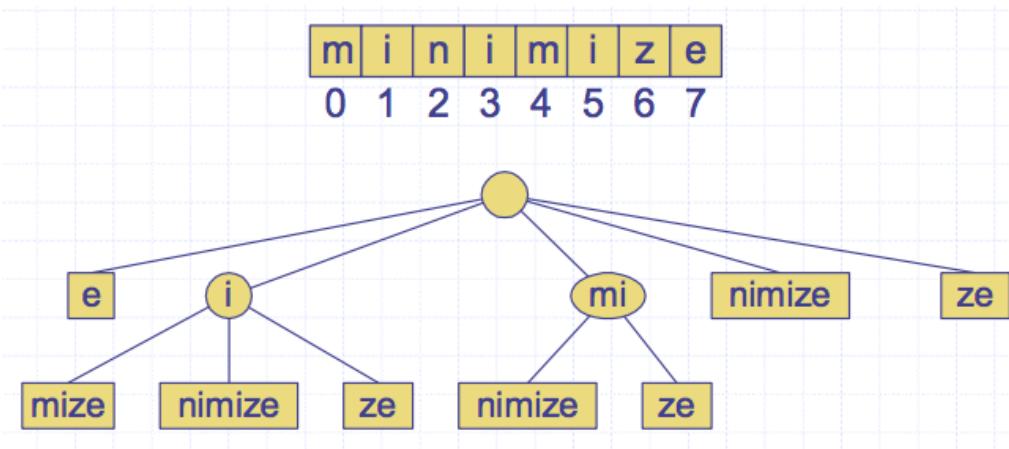
Eines komprimierten Tries für ein Array von Strings:

- Knoten speichert Indizes anstelle von Substrings
- Benötigt O(s) Speicher, wobei s die Anzahl Strings im Array ist
- Dienst als eine Hilfs-Index-Struktur

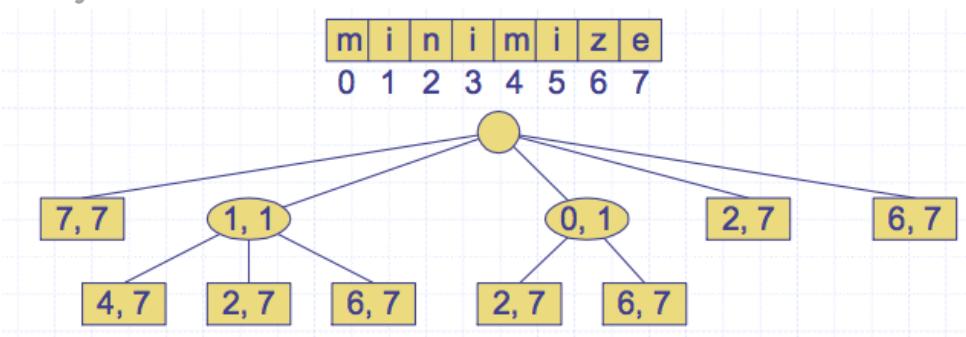


## Suffix Trie

Der Suffix-Trie eines Strings X ist der komprimierte Trie von allen Suffixen von X.



### Analyse des Suffix-Tries



Ein Suffix-Trie eines Strings X der Länge n über einem Alphabet der Grösse d benötigt  $O(n)$  Speicher. Es erlaubt Pattern Matching in X in  $O(dm)$  Zeit, wobei m die Länge des Pattern ist. Zudem kann es in  $O(n)$  Zeit erstellt werden.

# Dynamische Programmierung

## Rucksackproblem

Dynamische Programmierung ist ein generelles Algorithmen-Design-Paradigma. Anstatt der generellen Struktur schauen wir zuerst ein Motivations-Beispiel an. **Das Rucksack-Problem**

### Gegeben

N Gegenstände mit einem bestimmten Gewicht und Wert. Ein Rucksack mit einer bestimmten Gewichtskapazität.

### Gesucht

Füllung des Rucksacks, so dass der Wert der Gegenstände maximal ist.

### Versuch durch Aufzählung – Brute Force

#### Brute Force

Versuche alle möglichen Varianten. Betrachte nur jene Varianten, welche das maximale Gewicht nicht überschreiten. Nimm die beste Variante.

#### Laufzeit

Die Anzahl der möglichen Varianten ist exponentiell  $\rightarrow O(2^n)$ . Das ist ein sehr schlechter Algorithmus.

### Versuch mit jeweils Grösstem – Greedy Algorithmus

#### Idee

Nimm wiederholend den Gegenstand mit grössten Verhältnis von Wert und Gewicht.

#### Beispiel

4 Gegenstände

Wert	Gewicht	W/G
1	1	1.00
4	3	1.33
5	4	1.25
1	2	0.50

#### Resultat

$4(3\text{kg}) + 1(1\text{kg}) = 5 \rightarrow 4 \text{ kg mit einem totale Wert von } 5$ . Dies ist nicht optimal, besser wäre  $1(1\text{kg}) + 5(4\text{kg}) = 6!!$ .

### Versuch mit Subproblemen – Dynamische Programmierung

Wir konstruieren optimale Subprobleme „bottom-up“. Probleme der Länge 1 sind einfach, somit beginnen wir mit diesen. Dann Subprobleme der Länge 2,3, ... und so weiter.

#### Subprobleme bei Rucksack (\* Subproblem)

1...4 Gegenstände, 1..5 kg maximales Gewicht

kg \ Wert/kg	1	2	3	4	5
1/1	1	1	1	1	1
4/3	1	1	4	5	5
5/4	1	1	4	5	6
1/2	1	1	4	5	6

Was ist der grösstmögliche Wert für die ersten beiden Gegenstände 1/1 und 4/3 bei einer Gewichtslimite von 3 kg?

→ grösstmöglicher Wert für dieses Subproblem: 4 (mit Gegenstand 4/3).

Die Laufzeit für die ganze Tabelle beträgt O(Anzahl Gegenständen \* Anzahl Gewichte.).



kg \ Wert/kg	1	2	3	4	5
1/1	1	1	1	1	1
4/3	1	1	4	5	5
5/4	1	1	4	5	6
1/2	1	1	4	5	6

### ★ Nächstes Subproblem:

- nächster Gegenstand: 4/3
- nächstes Gewicht:  
aktueller Gewicht – Gewicht Gegenstand  
 $5 - 4 = 1$

Lösung für das ganze Problem:  
größtmöglicher Wert ist 6

Gegenstände für den Rucksack:

- ◆ beginne mit dem größtmöglichen Wert und gehe mit den Gegenständen zurück bis der Wert ändert: entsprechender Gegenstand gehört in Rucksack
- ◆ fahre mit nächstem Subproblem weiter:
  - ein Gegenstand weniger
  - unter Abzug des Gewichtes

**Lösung: 1/1 und 5/4 !**

## Technik der dynamischen Programmierung

Anwendbar auf Probleme welche anfänglich eine sehr grosse Laufzeit zu benötigen scheinen (möglicherweise sogar exponentiell). Voraussetzung:

### Einfache Subprobleme

Die Subprobleme können durch wenige Variablen ausgedrückt werden. z.B. j, k, l, m, etc.

### Subproblem-Optimierung

Das globale Optimum kann durch optimale Subprobleme ausgedrückt werden.

### Subprobleme überlappen

Die Subprobleme sind nicht unabhängig, sie überlappen (sollte somit bottom-up konstruiert werden).

### Subsequenzen

Eine Subsequenz eines Charakterstrings  $x_0x_1x_2x_3\dots x_{n-1}$  ist ein String der Form  $x_{i_1}x_{i_2}\dots$ , wo bei  $i_j < i_{j+1}$ . Es ist nicht dasselbe wie ein Substring. Beispiel für ABCDEFGHIJK

- Subsequenz: ACEGIJK
- Subsequenz: DFGHK
- Keine Subsequenz: DAGH

**Länge gemeinsame Subsequenz (LCS, Longest Common Subsequence)**

Gegeben sind die beiden Stirngs X und Y.

**Longest Common Subsequence (LCS)**

Finde die längste Subsequenz, welche in X sowie auch in Y enthalten ist.

**Beispiel**

ABCDEFHG und XZACKDFWGH haben ACDFG als längste gemeinsame Subsequenz.

**Anwendung**

zum Beispiel bei Vergleichen von DANN's (Alphabet ist A,C;G,T) oder Source-Files mit diff.

**Ein schwacher Versuch für das LCS Problem****Eine Brute-Force Lösung**

Aufzählung aller Subsequenzen von X. Testen, welche ebenfalls Subsequenzen von Y sind. Die längste Subsequenz wird als Resultat gewählt.

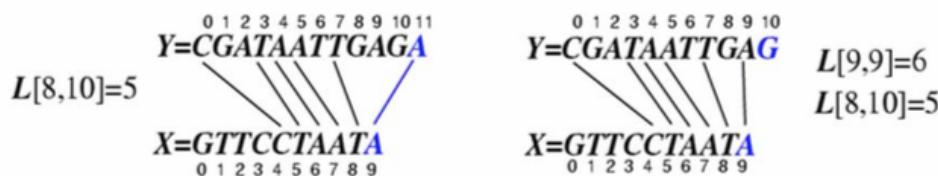
**Analysis**Wenn X die Länge n hat, dann ergeben sich  $2^n$  Subsequenzen. Das ist ein exponentieller Algorithmus.**Versuch mit dynamischer Programmierung**Definiere  $L[i,j]$  als Länge der längsten gemeinsamen Subsequenz von  $X[0..i]$  and  $Y[0..j]$ Erlaube -1 als Index, so dass  $L[-1,k] = 0$  and  $L[k,-1]=0$ 

Zum Signalisieren, dass der null Teil von X oder Y keine

Übereinstimmung hat mit dem Anderen

Definiere  $L[i,j]$  folgendermassen:

1. wenn  $x_i=y_j$ , dann  $L[i,j] = L[i-1,j-1] + 1$  (Übereinstimmung)
2. wenn  $x_i \neq y_j$ , dann  $L[i,j] = \max\{L[i-1,j], L[i,j-1]\}$  (keine Übereinstimmung)

**Beispiele:****Der LCS Algorithmus****Algorithm**  $\text{LCS}(X, Y)$ :**Input:** Strings  $X$  and  $Y$  with  $n$  and  $m$  elements, respectively**Output:** For  $i = 0, \dots, n-1$ ,  $j = 0, \dots, m-1$ , the length  $L[i, j]$  of a longest string that is a subsequence of both the string  $X[0..i] = x_0x_1\dots x_i$  and the string  $Y[0..j] = y_0y_1\dots y_j$ **for**  $i=1$  to  $n-1$  **do**   $L[i,-1] = 0$ **for**  $j=0$  to  $m-1$  **do**   $L[-1,j] = 0$ **for**  $i=0$  to  $n-1$  **do**  **for**  $j=0$  to  $m-1$  **do**    **if**  $x_i = y_j$  **then**       $L[i,j] = L[i-1,j-1] + 1$     **else**       $L[i,j] = \max\{L[i-1,j], L[i,j-1]\}$ **return** array  $L$

## Visualisierung des LCS Algorithmus

		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
T	1	0	0	1	1	2	2	2	2	2	2	2	2
T	2	0	0	1	1	2	2	2	3	3	3	3	3
C	3	0	1	1	1	2	2	2	3	3	3	3	3
C	4	0	1	1	1	2	2	2	3	3	3	3	3
T	5	0	1	1	1	2	2	2	3	4	4	4	4
A	6	0	1	1	2	2	3	3	3	4	4	5	5
A	7	0	1	1	2	2	3	4	4	4	4	5	5
T	8	0	1	1	2	3	3	4	5	5	5	5	6
A	9	0	1	1	2	3	4	4	5	5	5	6	6

## Analyse des LCS Algorithmus

Wir haben zwei verschachtelte Loops. Der äussere Loop iteriert n mal, der innere Loop iteriert m mal. Es ist ein konstanter Aufwand innerhalb jeder Iteration des inneren Loops vorhanden. Somit ergibt sich eine totale Laufzeit von **O(nm)**.

## Auslesen der LCS

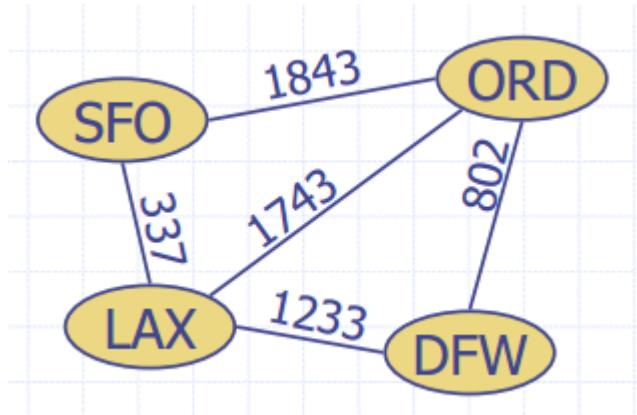
Die Lösung ist in L[n,m] enthalten. Die gesuchte Subsequenz kann von einer L-Tabelle ausgelesen werden. Beginne dazu am Ende mit i=m-1, j=n-1. Wenn das Zeichen für i und j gleich ist, füge L[i,j] in den LCS ein und wechsle die Position auf L[i-1,j-1]. Sonst folge der Kolonne oder der Reihe mit demselben Wert.

		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
T	1	0	0	1	1	2	2	2	2	2	2	2	2
T	2	0	0	1	1	2	2	2	3	3	3	3	3
C	3	0	1	1	1	2	2	2	3	3	3	3	3
C	4	0	1	1	1	2	2	2	3	3	3	3	3
T	5	0	1	1	1	2	2	2	3	4	4	4	4
A	6	0	1	1	2	2	3	3	4	4	4	5	5
A	7	0	1	1	2	2	3	4	4	4	4	5	5
T	8	0	1	1	2	3	3	4	5	5	5	5	6
A	9	0	1	1	2	3	4	4	5	5	5	6	6

Y=CGATAATTGAGA  
X=GTTCTAAATA  
0 1 2 3 4 5 6 7 8 9 10 11

★  $X_i = Y_j$

## Graphen

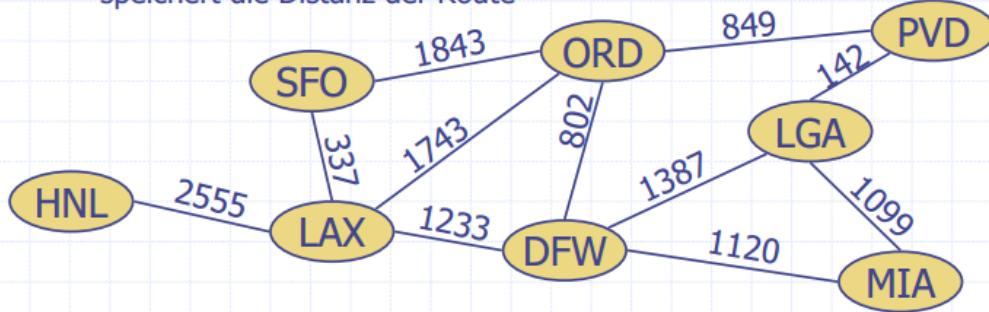


Ein Graph ist ein Paar  $(V, E)$ , wobei  $V$  ein Set von Vertizes (Knoten) ist und  $E$  eine Collection von Vertizes-Paaren, Kanten (Edge) genannt. Vertizes und Kanten sind Positionen und speichern Elemente.

### Beispiel

Ein Vertex repräsentiert ein Flughafen und speichert den Flughafen-Code. Eine Kante repräsentiert eine Flugroute zwischen zwei Flughäfen und speichert die Distanz der Route.

SPEICHERT DIE DISTANZ DER ROUTE



## Kantentypen

### Gerichtete Kanten



Geordnetes Paar von Vertizes  $(u,v)$ . Erster Vertex  $u$  entspricht dem Ursprung. Zweiter Vertex  $v$  entspricht dem Ziel. Ein Beispiel dafür ist ein Flug.

### Ungerichtete Kanten



Ungeordnetes Vertizes-Paar  $(u,v)$ . Zum Beispiel eine Flugroute.

## Gerichteter Graph

Alle Kanten sind gerichtet. Zum Beispiel ein Flugplan.

## Ungerichteter Graph

Alle Kanten sind ungerichtet. Zum Beispiel ein Flugrouten-Plan.

## Anwendungen

Elektronische Schaltungen (printed circuit board, integrated circuit), Transport-Netzwerke (Autobahnnetz, Flugnetz), Computer Netzwerke (LAN, Internet, Web) und Datenbanken (Entity-Relationship-Diagramm).

## Terminologie

**End-Vertizes** (oder Endpunkte) einer Kante

- **U** und **V** sind die Endpunkte der Kante **a**

Kanten sind **incident** (enden) an einem Vertex

- **a, d** und **b** sind **incident in V**

**Adjazente** (benachbarte) Vertizes

- **U** und **V** sind **adjacent**

**Grad (Degree)** eines Vertex: Anzahl incidenter Kanten

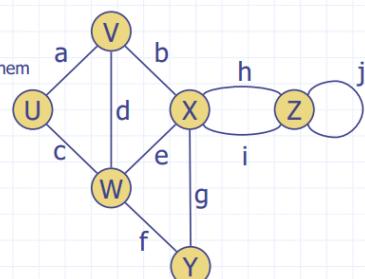
- **X** besitzt Grad 5

**Parallele Kanten**

- **h** und **i** sind parallele Kanten

**Schleife**

- **j** ist eine Schleife

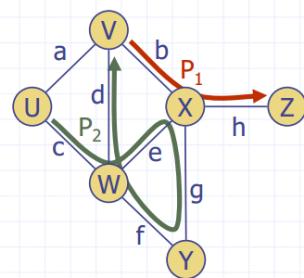


### Pfad

- Sequenz von alternierenden Vertizes und Kanten
- beginnt mit einem Vertex
- endet mit einem Vertex
- jede Kante beginnt und endet an einem ihrer Endpunkte

### Einfacher Pfad

- ein Pfad, so dass alle seine Vertizes und Kanten unterschiedlich sind
- Beispiele
- $P_1 = (V, b, X, h, Z)$  ist ein **einfacher Pfad**
  - $P_2 = (U, c, W, e, X, g, Y, f, W, d, V)$  ist **nicht einfacher**



## Zyklus

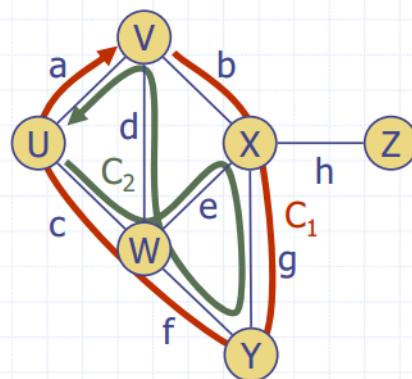
- zirkuläre Sequenz alternierender Vertizes und Kanten.

### Einfacher Zyklus

- ein Zyklus mit lauter verschiedenen Ecken/Vertizes und Kanten.

Beispiele

- $C_1 = (V, b, X, g, Y, f, W, c, U, a)$  ist ein **einfacher Zyklus**.
- $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a)$  ist ein **nicht einfacher Zyklus**.



## Eigenschaften

### Eigenschaft 1

$$\sum_v \deg(v) = 2m$$

**Beweis:** Jede Kante wird zweimal gezählt.

### Eigenschaft 2

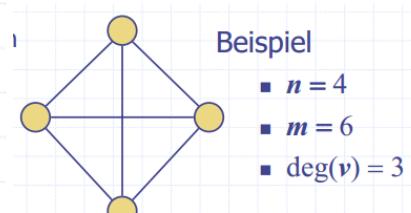
In einem ungerichteten Graphen ohne Schleifen und ohne Mehrfach-Kanten (parallele Kanten) gilt (dies entspricht einem ungerichteten, einfachen Graphen):

$$M \leq n(n-1) / 2$$

**Beweis:** Jeder Vertex besitzt einen Grad von höchstens  $(n-1)$ .

## Notation

- n** Anzahl Vertizes
- m** Anzahl Kanten
- deg(v)** Grad von Vertex **v**



# Hauptmethoden des Graph-ADT's

Vertizes und Kanten sind Positionen und speichern Elemente.

Zugriffs-Methoden	Update-Methoden	Iterator-Methoden
<ul style="list-style-type: none"> <li><b>endVertices(e)</b>: an array of the two endvertices of e</li> <li><b>opposite(v, e)</b>: the vertex opposite of v on e</li> <li><b>areAdjacent(v, w)</b>: true iff(*) v and w are adjacent</li> <li><b>replace(v, x)</b>: replace element at vertex v with x</li> <li><b>replace(e, x)</b>: replace element at edge e with x</li> </ul>	<ul style="list-style-type: none"> <li><b>insertVertex(o)</b>: insert a vertex storing element o</li> <li><b>insertEdge(v, w, o)</b>: insert an edge (v,w) storing element o</li> <li><b>removeVertex(v)</b>: remove vertex v (and its incident edges)</li> <li><b>removeEdge(e)</b>: remove edge e</li> </ul>	<ul style="list-style-type: none"> <li><b>incidentEdges(v)</b>: collection of edges incident to v</li> <li><b>vertices()</b>: collection of all vertices in the graph</li> <li><b>edges()</b>: collection of all edges in the graph</li> </ul>

## Kanten-Listen Struktur

### Vertex / Knoten Objekt

- Element
- Referenz auf eine Position in der Vertex-Sequenz

### Kanten Objekt

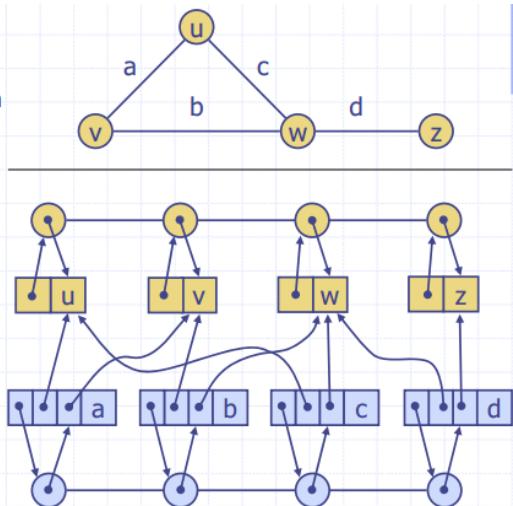
- Element
- Ursprungs-Vertex Object
- Ziel-Vertex Object
- Referenz auf die Position in der Kanten-Sequenz

### Vertex-Sequenz

- Sequenz der Vertex-Objekte

### Kanten-Sequenz

- Sequenz von Kanten-Objekten



## Adjazenz-Listen Struktur

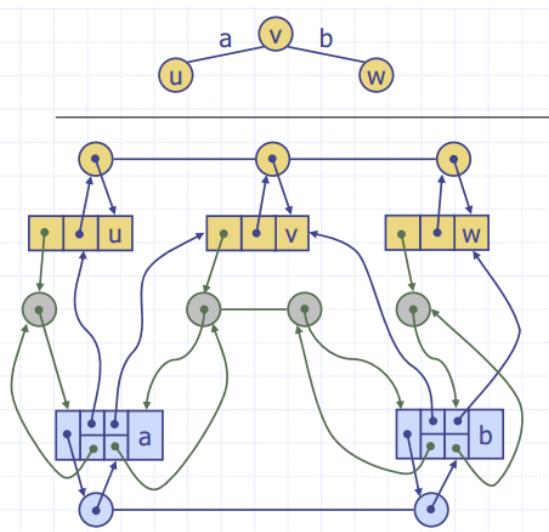
### Kantenlisten Struktur

#### Inzidenz-Sequenz für jeden Vertex

- Sequenz der Positionen auf Kantenobjekte der inzidenten Kanten

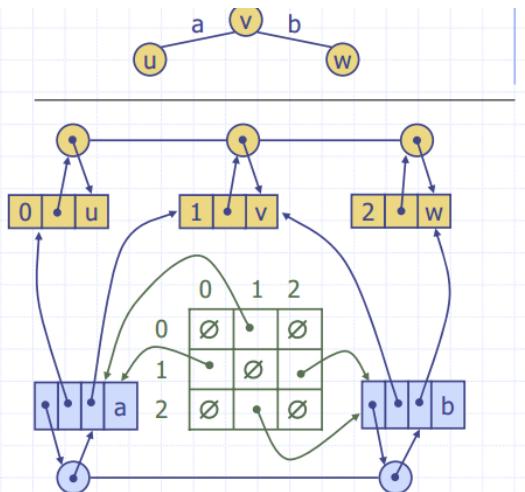
#### Erweiterte Kanten-Objekte

- Referenziert auf die assoziierten Positionen in der Inzidenzsequenz der Endvertizes

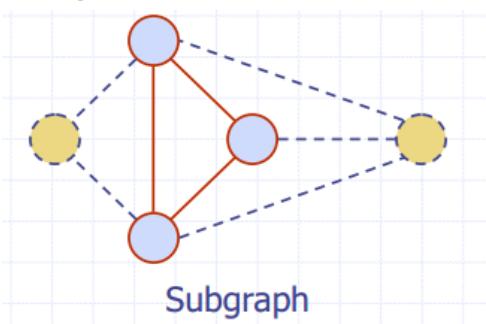


**Adjazenz-Matrix Struktur****Kantenlisten Struktur****erweiterte Vertex-Objekte**

- Integer Key (Index) assoziiert mit Vertex
- 2D-Array Adjazenz-Array
- Referenziert auf die Kantenobjekte für adjazente (benachbarte) Vertizes
  - **null** für nichtadjazente (nichtbenachbarte) Vertizes

**Performance**

<b>◆ <math>n</math> Vertizes, <math>m</math> Kanten ◆ keine parallelen Kanten ◆ keine Schleifen</b>	<b>Kanten Liste</b>	<b>Adjazenz Liste</b>	<b>Adjazenz Matrix</b>
<b>Space</b>	<b><math>n + m</math></b>	<b><math>n + m</math></b>	<b><math>n^2</math></b>
<b>incidentEdges(<math>v</math>)</b>	<b><math>m</math></b>	<b><math>\deg(v)</math></b>	<b><math>n</math></b>
<b>areAdjacent (<math>v, w</math>)</b>	<b><math>m</math></b>	<b><math>\min(\deg(v), \deg(w))</math></b>	<b>1</b>
<b>insertVertex(<math>o</math>)</b>	<b>1</b>	<b>1</b>	<b><math>n^2</math></b>
<b>insertEdge(<math>v, w, o</math>)</b>	<b>1</b>	<b>1</b>	<b>1</b>
<b>removeVertex(<math>v</math>)</b>	<b><math>m</math></b>	<b><math>\deg(v)</math></b>	<b><math>n^2</math></b>
<b>removeEdge(<math>e</math>)</b>	<b>1</b>	<b>1</b>	<b>1</b>

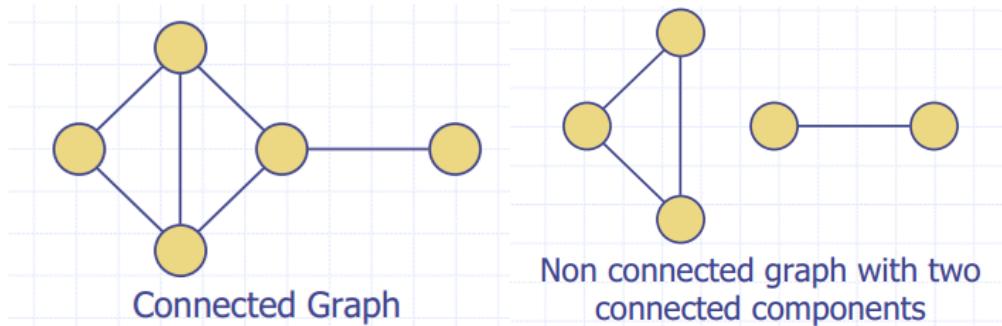
**Subgraphen**

Ein Subgraph S eines Graphen G ist ein Graph, so dass gilt:

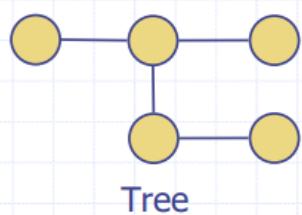
- Die Kanten von S sind eine Teilmenge der Kanten von G
- Die Vertizes von S sind eine Teilmenge der Vertizes von G

## Connectivity

Ein Graph heisst verbunden (connected), falls zwischen jedem Paar Vertizes ein Pfad existiert. Eine verbundene Komponente eines Graphen G ist ein verbundener Subgraph von G.

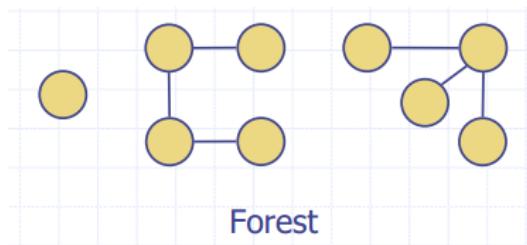


## Bäume und Wälder



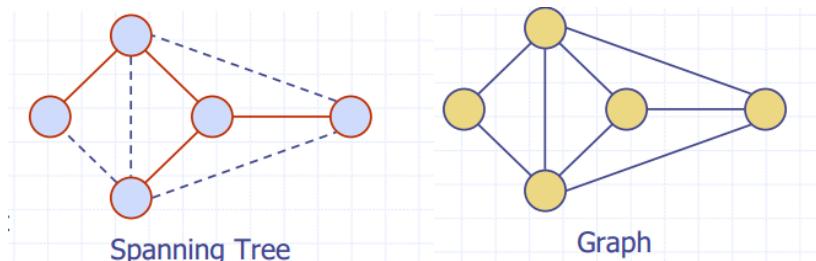
Ein (freier) Baum ist ein ungerichteter Graph T, so dass T verbunden ist und T keine Zykeln aufweist. Die Definition eines Baumes unterscheidet sich von jener eines Wurzelbaumes.

Ein Wald ist ein ungerichteter Graph ohne Zyklen. Die verbundenen Komponenten eines Waldes sind Bäume.



## Spanning Trees und Wälder

Ein aufspannender Baum (Spanning Tree) eines verbundenen Graphen ist ein aufspannender Subgraph, welcher auch ein Baum ist. Ein aufspannender Baum ist nicht eindeutig, ausser der Graph, von dem ausgegangen wird, ist ein Baum. Ausspannende Bäume werden beispielsweise in Kommunikationsnetzwerken eingesetzt. Ein Wald eines Graphen ist ein aufspannender Subgraph, welcher auch ein Wald ist.



## Depth-First Search

Depth-First Search (DFS) ist eine allgemeine Technik für die Traversierung eines Graphen. Eine DFS Traversierung eines Graphen  $G$  besucht alle Vertizes und Kanten von  $G$ , bestimmt ob  $G$  verbunden ist, berechnet/bestimmt die verbundenen Komponenten von  $G$  und berechnet einen aufspannenden Wald von  $G$ .

DFS auf einem Graphen mit  $n$  Vertizes und  $m$  Kanten benötigt  $O(n+m)$  Zeit. DFS kann man auch erweitern, um andere Graphenprobleme zu lösen.

- Finden und Ausgeben eines Pfades zwischen zwei gegebenen Vertizes
- Finden von Zyklen in Graphen

Depth-First Search entspricht in etwa der Euler-Tour bei binären Bäumen.

### DFS Algorithmus

Der Algorithmus benutzt einen Mechanismus, um «Labels» auf Kanten und Vertizes zu setzen.

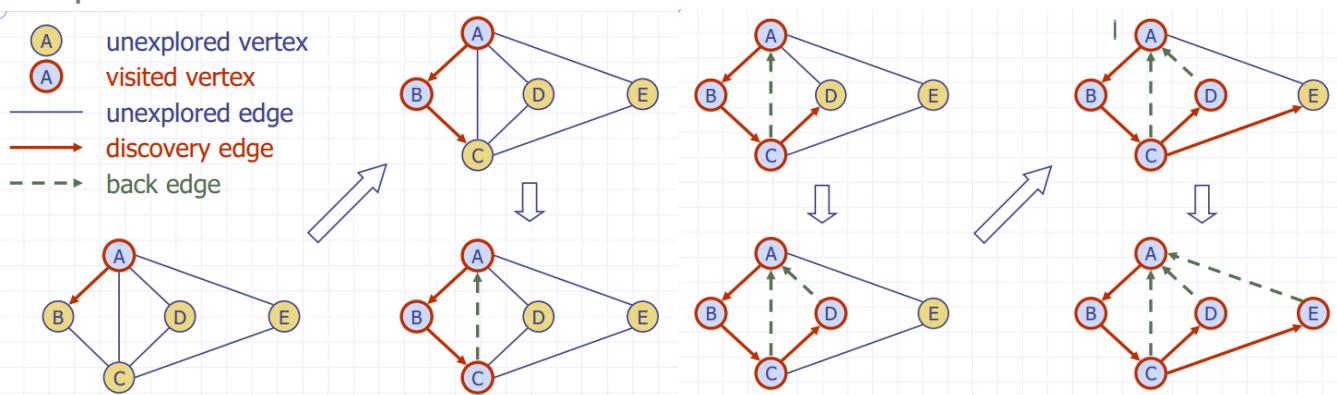
#### Algorithm $DFS(G, v)$

**Input** graph  $G$  and a start vertex  $v$  of  $G$   
**Output** labeling of the edges of  $G$  in the connected component of  $v$  as discovery edges and back edges  
 $setLabel(v, VISITED)$   
**for all**  $e \in G.incidentEdges(v)$   
  **if**  $getLabel(e) = UNEXPLORED$   
     $w \leftarrow opposite(v, e)$   
    **if**  $getLabel(w) = UNEXPLORED$   
       $setLabel(e, DISCOVERY)$   
       $DFS(G, w)$   
    **else**  
       $setLabel(e, BACK)$

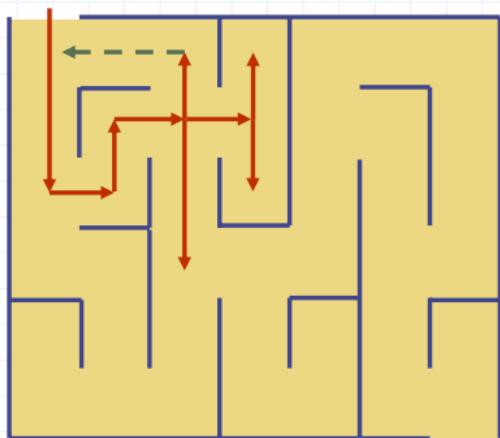
#### Algorithm $DFS(G)$

**Input** graph  $G$   
**Output** labeling of the edges of  $G$  as discovery edges and back edges  
**for all**  $u \in G.vertices()$   
   $setLabel(u, UNEXPLORED)$   
**for all**  $e \in G.edges()$   
   $setLabel(e, UNEXPLORED)$   
**for all**  $v \in G.vertices()$   
  **if**  $getLabel(v) = UNEXPLORED$   
     $DFS(G, v)$

### Beispiel



# DFS und Labyrinth

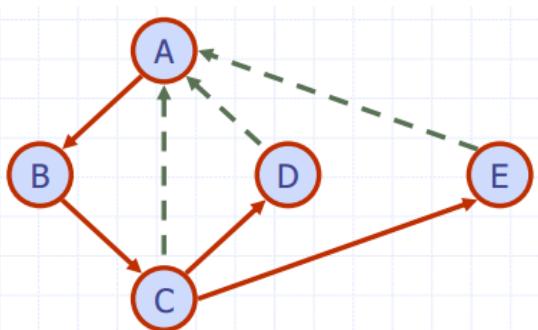


Der DFS Algorithmus ähnelt der klassischen Strategie zur Erkundung eines Labyrinths.

Wir markieren jede besuchte Kreuzung, Ecke und Sackgasse (Vertex). Wir markieren jeden besuchten Korridor (Kante).

Wir notieren den Rückweg zum Eingang (Start Vertex) mittels Rekursion auf dem Stack.

## Eigenschaften von DFS



## Eigenschaft 1

DFS ( $G, V$ ) besucht Vertizes und Kanten in der verbundenen Komponente von  $G$  beginnend bei  $v$ .

## Eigenschaft 2

Die von  $\text{DFS}(G, v)$  markierten, besuchten Kanten bilden einen aufspannenden Baum für die verbundene Komponente von  $G$  beginnend bei  $v$ .

## Analyse von DFS

Setzen/Lesen eines Vertex/Kanten-Labels benötigt O(1) Zeit. Jeder Vertex wird zweimal markiert. Zuerst als Unexplored und dann als VISITED. Jede Kante wird ebenfalls zweimal markiert. Zuerst als UNEXPLORED und dann als DISCOVERY oder BACK. Die Methode incidentEdges() wird pro Vertex einmal aufgerufen.

DFS benötigt  $O(n+m)$  Zeit, sofern der Graph mit Hilfe seiner Adjazenzlisten-Struktur dargestellt wird.  
Es gilt:  $\sum_v \deg(v) = 2m$

## Pfade finden

**Algorithm** *pathDFS(G, v, z)*

```

setLabel(v, VISITED)
S.push(v)
if v = z
    finish: result is S.elements()
for all e ∈ G.IncidentEdges(v)
    if getLabel(e) = UNEXPLORED
        w ← opposite(v, e)
        if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            S.push(e)
            pathDFS(G, w, z)
            S.pop()
        else
            setLabel(e, BACK)
    S.pop()

```

Wir können den DFS Algorithmus spezialisieren, um einen Pfad zwischen zwei gegebenen Vertizes u und z mit Hilfe des Template Methode Patterns zu finden.

Zuerst rufen wir  $\text{DFS}(G, u)$  mit  $u$  als Startvertex auf.

Mit Hilfe eines Stacks S merken wir uns den Pfad zwischen dem Startvertex und dem aktuellen Vertex.

Sobald wir den Zielvertex z gefunden haben, geben wir den Pfad mit Hilfe des Stacks aus.

## Zyklen finden

```

Algorithm cycleDFS(G, v)
  setLabel(v, VISITED)
  S.push(v)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      S.push(e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        cycleDFS(G, w)
      S.pop()
    else
      T ← new empty stack
      repeat
        o ← S.pop()
        T.push(o)
      until o = w
      finish: result is T.elements()
  S.pop()

```

Wir können den DFS Algorithmus spezialisieren um einfache Zyklen des Template Methode Patterns zu finden.

Mit Hilfe eines Stacks S merken wir uns den Pfad zwischen dem Startvertex und dem aktuellen Vertex.

Sobald wir auf eine Back-Edge(*v,w*) treffen, geben wir den Zyklus als Teil des Stacks aus: vom obersten Stackelement bis zum Vertex *w*.

## Breadth-Frist Search / Breitensuche

Breadth-First Search (BFS) ist eine generelle Technik für die Traversierung eines Graphen. Eine BFS Traversierung eines Graphen *G*

- Besucht alle Vertizes und alle Kanten von *G*
- Bestimmt, ob *G* verbunden ist
- Berechnet/bestimmt die verbundene Komponenten von *G*
- Berechnet einen aufspannenden Wald von *G*

BFS auf einem Graphen mit *n* Vertizes und *m* Kanten benötigt  $O(n+m)$  Zeit. BFS kann man auch erweitern, um andere Graphenprobleme zu lösen.

- Finden und Ausgeben eines Pfades mit einer minimalen Anzahl Kanten zwischen zwei Vertizes.
- Finden von einfachen Zyklen, falls es solche gibt.

## BFS Algorithmus

Der Algorithmus benutzt einen Mechanismus, um «Labels» auf Kanten und Vertizes zu setzen.

```

Algorithm BFS(G)
  Input graph G
  Output labeling of the edges
    and partition of the
    vertices of G
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      BFS(G, v)

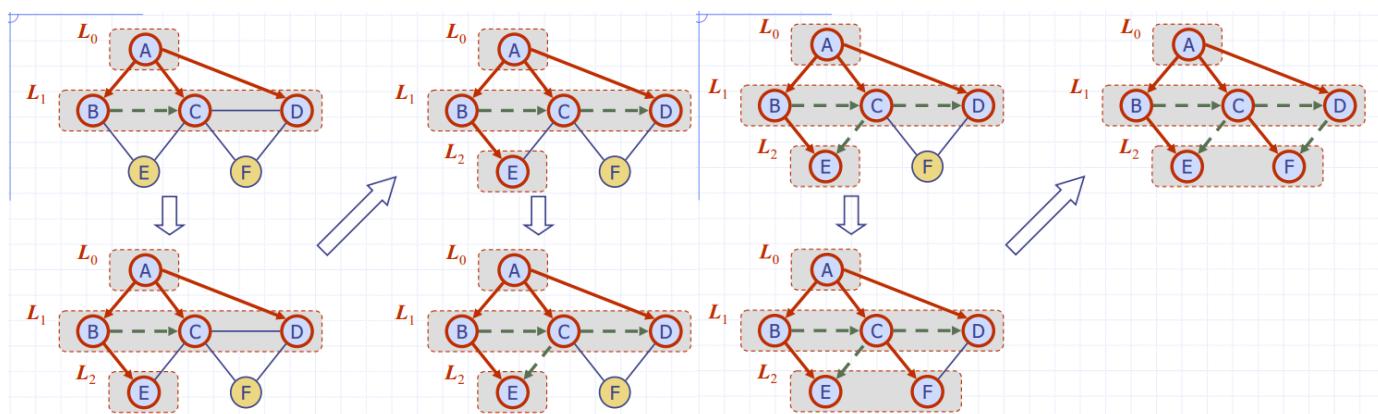
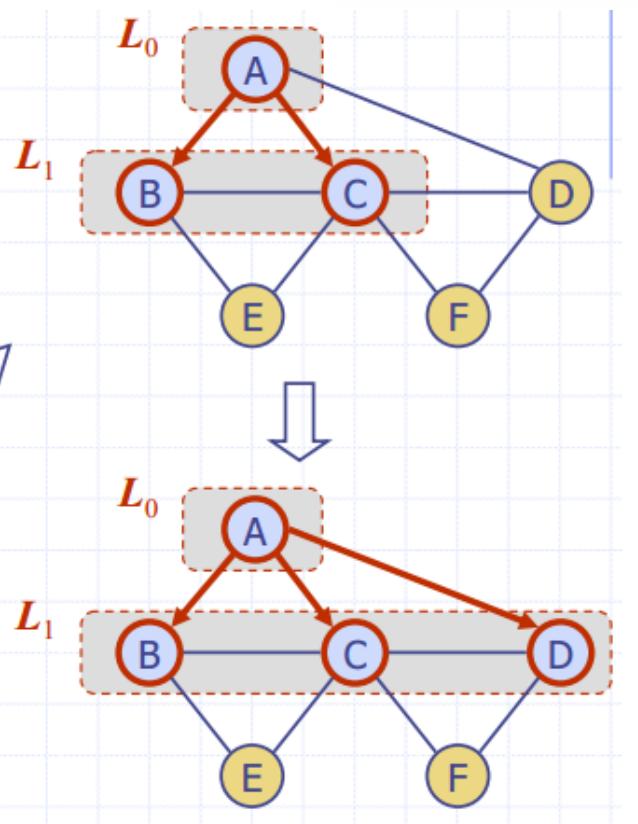
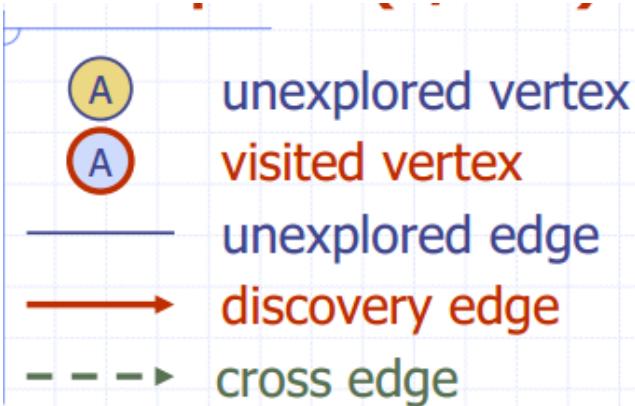
```

```

Algorithm BFS(G, s)
  L0 ← new empty sequence
  L0.insertLast(s)
  setLabel(s, VISITED)
  i ← 0
  while ¬ Li.isEmpty()
    Li+1 ← new empty sequence
    for all v ∈ Li.elements()
      for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
          w ← opposite(v,e)
          if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            setLabel(w, VISITED)
            Li+1.insertLast(w)
          else
            setLabel(e, CROSS)
    i ← i + 1

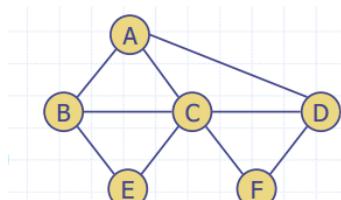
```

## Beispiel



## Eigenschaften

## Notation



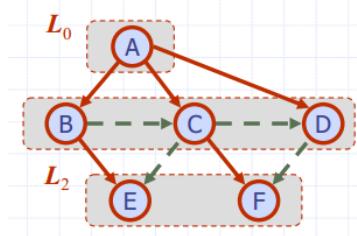
$G_s$ : verbundene Komponente von s (s: Start-Vertex)

## Eigenschaft 1

BFS ( $G, s$ ) besucht alle Vertizes und Kanten in  $G_s$

## Eigenschaft 2

Die Discovery-Kanten von  $\text{BFS}(G, s)$  bilden einen aufspannenden Baum  $T_s$  von  $G_s$ .



### Eigenschaft 3

für jeden Vertex  $v$  in  $L_1$  gilt:

- Der Pfad in  $T_s$  von  $s$  nach  $v$  besitzt  $i$  Kanten
  - Jeder Pfad von  $s$  nach  $v$  in  $G_s$  besitzt mindestens  $i$  Kanten

## Analysis

Setzen/Lesen eines Vertex-/Kanten-Labels benötigt  $O(1)$  Zeit. Jeder Vertex wird zweifach markiert, einmal als UNEXPLORED und einmal als VISITED. Jede Kante wird ebenfalls zweifach markiert. Einmal als UNEXPLORED und einmal als DISCOVERY oder CROSS.

Jeder Vertex wird einmal in die Sequenz  $L_i$  eingetragen. Die Methode `incidentEdges()` wird einmal pro Vertex aufgerufen. BFS benötigt  $O(n+m)$  Zeit, sofern der Graph mit Hilfe seiner Adjazenzliste dargestellt wird. Es gilt:  $\sum_v \deg(v) = 2m$

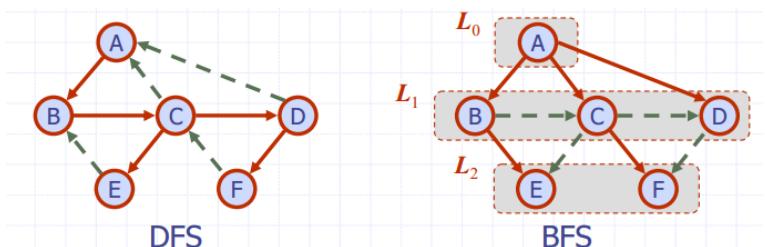
## Applikationen

Mit Hilfe des Template Method Pattern können wir die BFS Traversierung eines Graphen  $G$  benutzen, um folgende Probleme in  $O(n+m)$  Zeit zu lösen:

- Bestimmen der verbundenen Komponenten von  $G$
- Bestimmen eines aufspannenden Waldes von  $G$
- Bestimmen eines einfachen Zyklus in  $G$ , oder bestimmen, ob  $G$  ein Wald ist.
- Bei zwei gegebenen Vertizes von  $G$ : Finden eines Pfades in  $G$  zwischen den beiden Vertizes mit minimaler Anzahl Kanten oder bestimmen, ob ein solcher Pfad existiert.

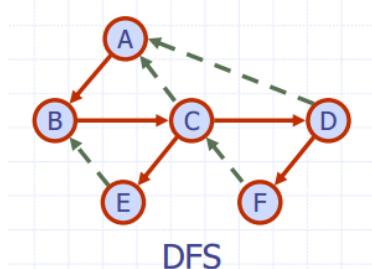
## DFS vs. BFS

Applikationen	DFS	BFS
Aufspannender Wald, Verbundene Komponenten, Pfade, Zyklen	✓	✓
Kürzester Pfad		✓
Biconnected Komponenten	✓	



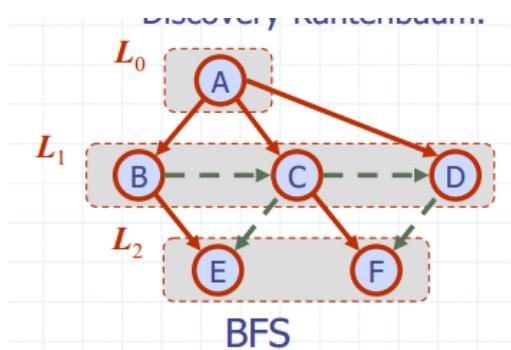
## Back edge (v,w) (Rückwärtskante)

w ist ein Vorfahre von v im Baum der Suchkanten.



## Cross edge (v,w) (Kreuzungskante)

w ist auf der selben Stufen wie v oder auf dem nächsten Level im Discovery-Kantenbaum.



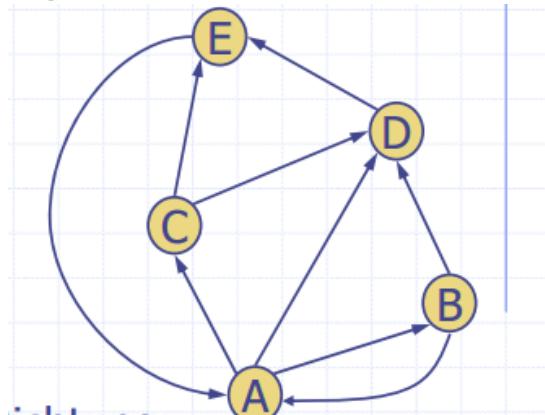
# Directed Graphs / Gerichtete Graphen

Ein gerichteter Graph (Directed Graph / Digraph) ist ein Graph, dessen Kanten alle gerichtet sind.

## Anwendungen

Einbahnstrassen, Flüge, Task Scheduling

## Eigenschaften



Ein Diagraph ist:  $G=(V,E)$  derart, dass jede Kante nur in eine Richtung geht. Kante  $(a,b)$  geht von  $a$  nach  $b$ , aber nicht von  $b$  nach  $a$ .

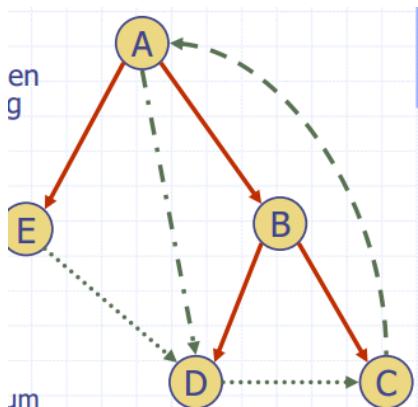
Wenn  $G$  einfach ist, dann  $m \leq n(n+1)$

Wenn In- und Out-Kanten separaten Adjazenz-Listen sind:  
Laufzeit für Zugriff auf In- und Out-Kanten proportional zur Grösse der Listen

## Anwendungen

Zum Beispiel, dass bereits erwähnte Scheduling. Kante  $(a,b)$  bedeutet, dass Task  $a$  terminieren muss, bevor Task  $b$  gestartet wird.

## Gerichtete Tiefensuche



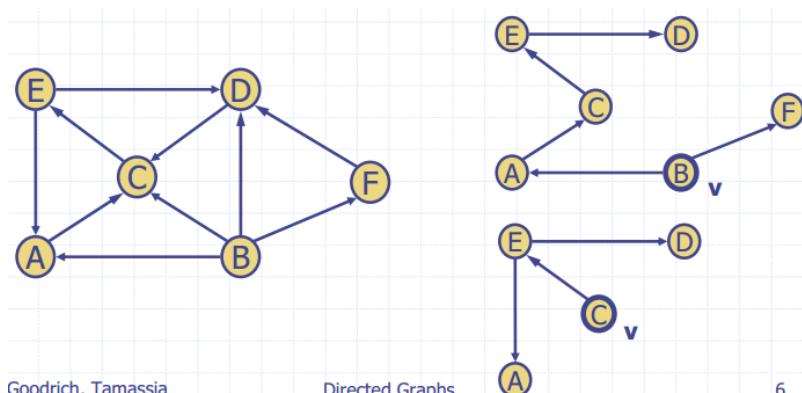
Wir können die Traversierungs-Algorithmen (DFS und BFS) für Diagraphen spezialisieren, indem Kanten nur entlang ihrer Richtung traversiert werden. Im gerichteten DFS-Algorithmus haben wir vier Typen von Kanten.

- Baumkanten (discovery), Kante des Waldes (fett)
- Rückkanten (back), Verbindung zu einem Vorgänger (gestrichelt)
- Vorwärtskanten (forward), Verbindung zu einem Nachfolger im Baum (- und ....)
- Kreuzungskanten (cross), alle übrigen Kanten (gepunktet)

Eine gerichtete Tiefensuche beginnt bei einem Vertex  $s$  und bestimmt die Vertizes, welche von  $s$  aus erreichbar sind.

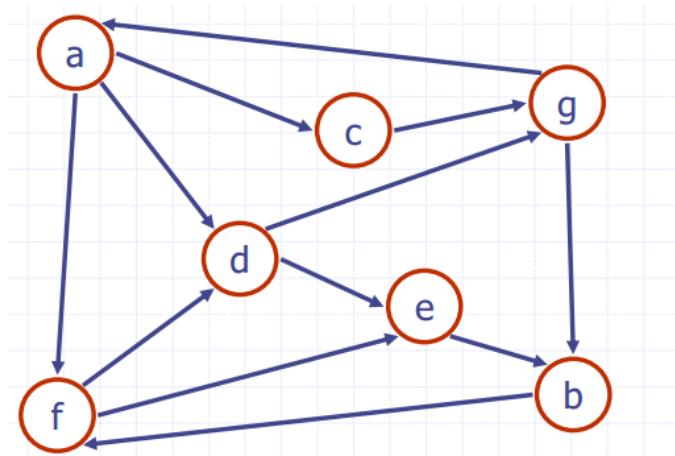
## Erreichbarkeit

DFS Baum mit Wurzel  $v$ : Vertizes erreichbar von  $v$  durch gerichtete Pfade.



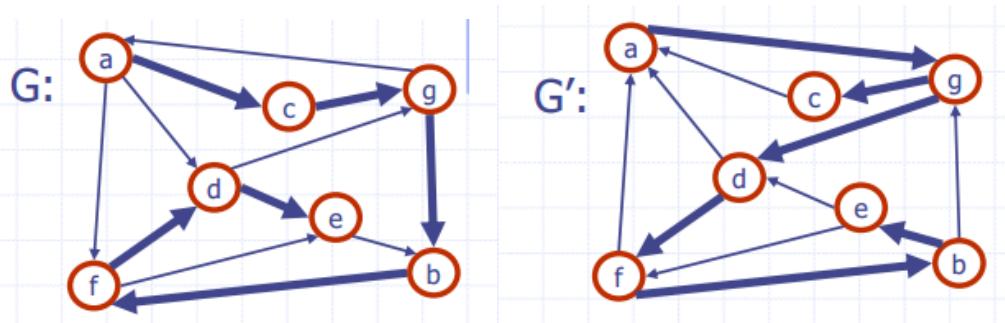
## String Connectivity

Davon spricht man, wenn jeder Vertex alle anderen Vertizes erreichen kann.



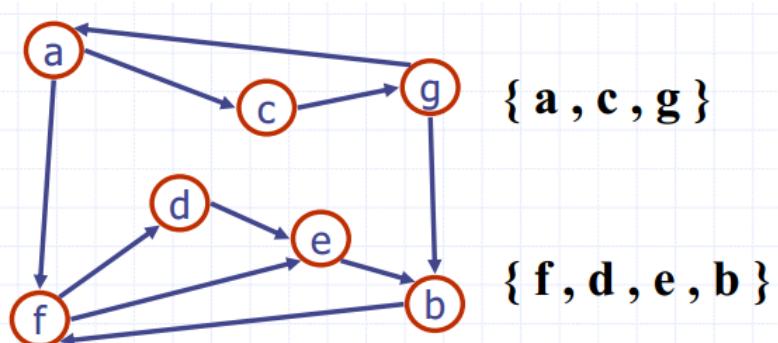
### Algorithmus

- Wähle einen Vertex v in G.
- Führe eine Tiefensuche von v in G durch.
  - o Wenn es einen nicht besuchten Vertex w gibt, dann return false // NOK.
- $G'$  sei G mit umgekehrten Kanten (Richtungen)
- Führe eine Tiefensuche durch von v in  $G'$ 
  - o Wenn es einen nicht besuchten Vertex w gibt, dann return false
  - o Sonst return true.
- Die Laufzeit des Algorithmus ist  $O(n+m)$ .



### Streng verbundene Komponenten

Maximaler Subgraph, sodass jeder Vertex alle anderen Vertizes im Subgraph erreichen kann. Die Laufzeit ist  $O(n+m)$  mit der Tiefensuche, diese ist aber komplizierter (ähnlich zu Binconnectivity).

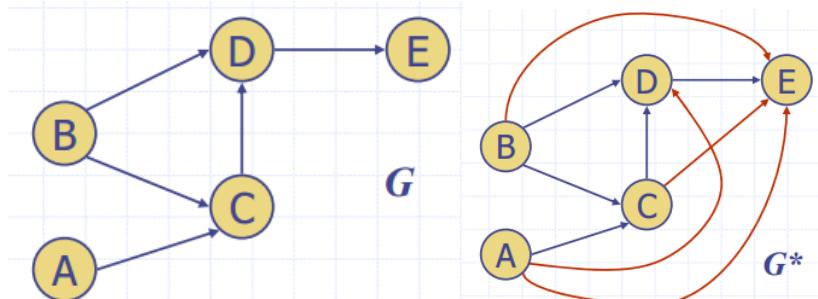


## Transitiver Abschluss

Gegeben ist ein Diograph G: der transitive Abschluss von G ist der Diograph  $G^*$ , sodass:

- $G^*$  hat die gleichen Vertizes wie G
- Wenn G einen gerichteten Pfad von u nach v ( $u \neq v$ ), dann hat  $G^*$  eine gerichtete Kante von u nach v.

Der transitive Abschluss stellt die gesamte Erreichbarkeitsinformation über einen Diographen zur Verfügung.



### Berechnung

Tiefensuche, beginnend bei jedem Vertex  $\rightarrow O(n(n+m))$ .

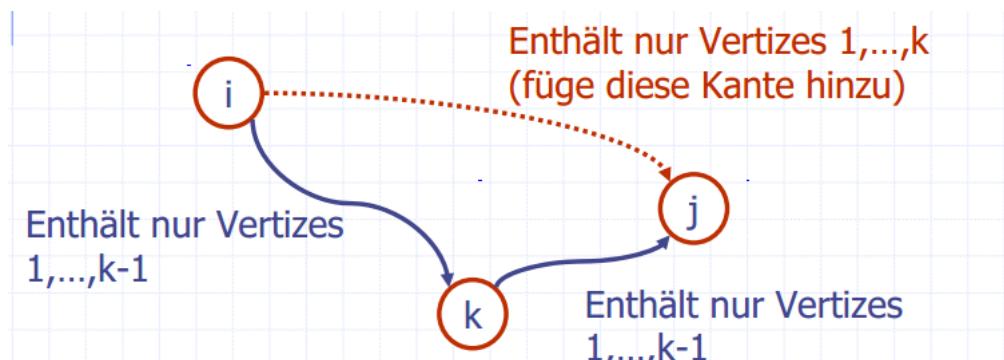
Als Alternative ist der Floyd-Warshall-Algorithmus aus der dynamischen Programmierung möglich.

### Floyd-Warshall-Algorithmus

#### Grundidee

**Idee 1** Nummerierung der Vertizes: 1,2,...,n.

**Idee 2** Beachte nur Pfade mit Vertizes 1,2,...,k als Zwischenvertex.



**Algorithm FloydWarshall( $G$ )**

```

Input digraph  $G$ 
Output transitive closure  $G^*$  of  $G$ 
 $i \leftarrow 1$ 
for all  $v \in G.vertices()$ 
    denote  $v$  as  $v_i$ 
     $i \leftarrow i + 1$ 
 $G_0 \leftarrow G$ 
for  $k \leftarrow 1$  to  $n$  do
     $G_k \leftarrow G_{k-1}$ 
    for  $i \leftarrow 1$  to  $n$  ( $i \neq k$ ) do
        for  $j \leftarrow 1$  to  $n$  ( $j \neq i, k$ ) do
            if  $G_{k-1}.areAdjacent(v_p, v_k) \wedge$ 
                 $G_{k-1}.areAdjacent(v_k, v_j)$ 
            if  $\neg G_k \text{ areAdjacent}(v_p, v_j)$ 
                 $G_k.insertDirectedEdge(v_p, v_j, k)$ 
    return  $G_n$ 

```

Floyd-Warshall's Algorithmus nummeriert die Vertizes von  $G$  als  $v_1, \dots, v_n$  und berechnet eine Serie von Diagrahen  $G_0, \dots, G_n$ .

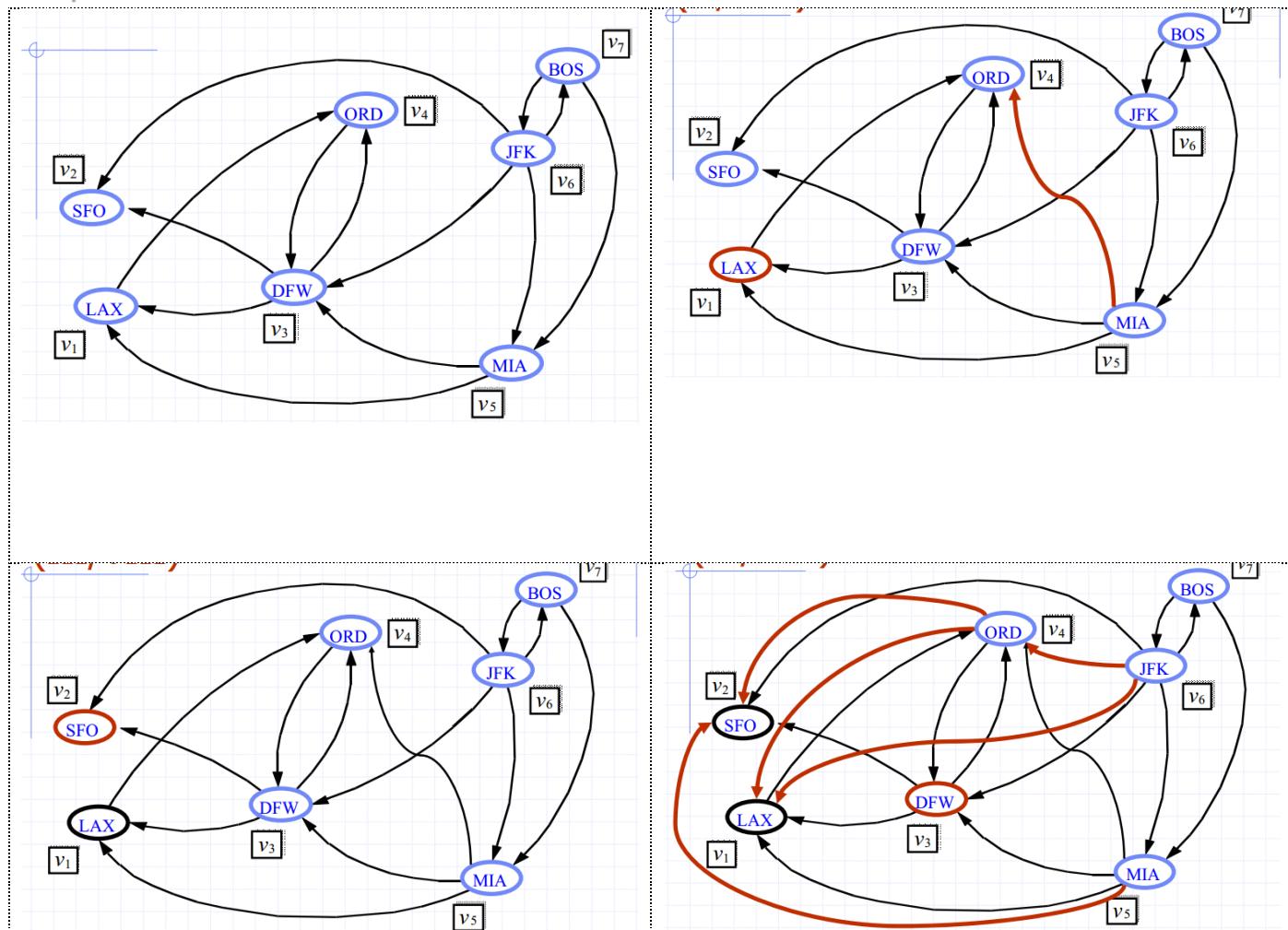
- $G_0 = G$
- $G_k$  hat eine gerichtete Kante  $(v_i, v_j)$ , falls  $G$  einen gerichteten Pfad von  $v_i$  nach  $v_j$  mit Zwischenvertex aus der Menge  $\{v_1, \dots, v_k\}$  hat.

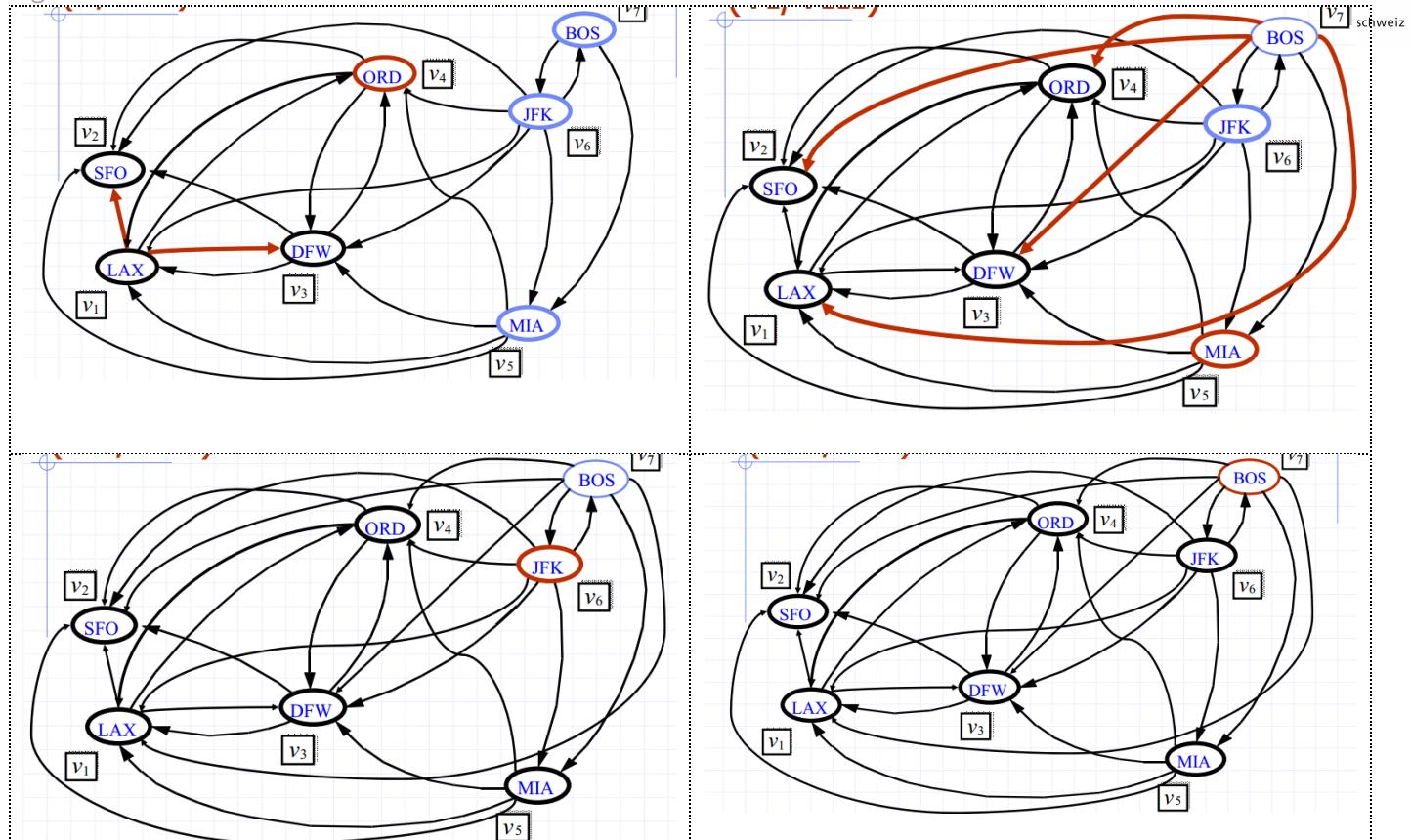
Es gilt:  $G_n = G^*$

In der Phase  $k$ , Diagraph  $G_k$  ist aus  $G_{k-1}$  berechnet.

Die Laufzeit beträgt  $O(n^3)$  unter der Annahme das  $\text{areAdjacent}(\text{from}, \text{to})$  mit  $O(1)$  abläuft.

Beispiel

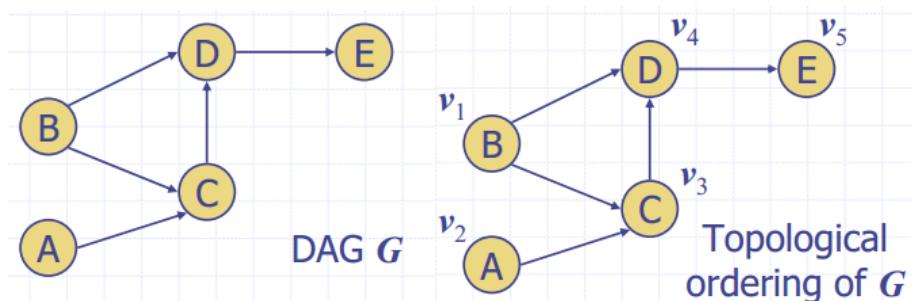




## DAG's und topologische Ordnung

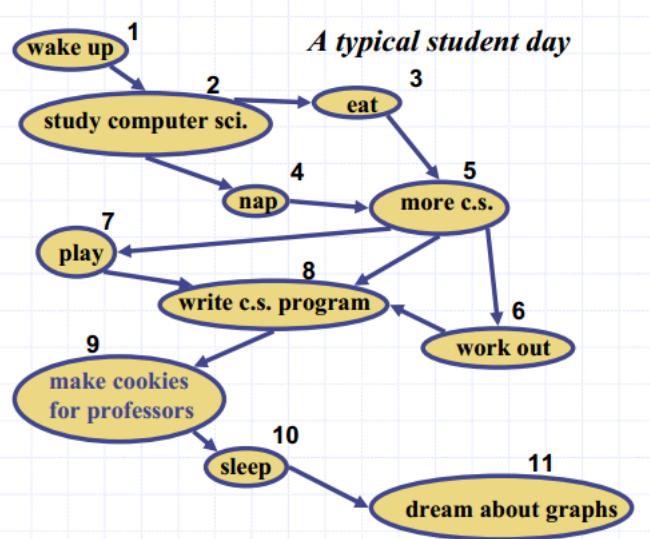
Ein gerichteter azyklischer Graph (Directed Acyclic Graph) ist ein Diagraph, der keine gerichtete Zyklen enthält. Eine topologische Ordnung eines Diagraphs ist definiert durch die Nummerierung  $v_1, \dots, v_n$  der Vertizes, sodass für jede Kante  $(v_i, v_j)$  gilt  $i < j$ .

Beispiel: In einem Task-Scheduling Diagraphen bestimmt die topologische Ordnung die Task-Sequenz mit Präzedenzbedingungen.



## Topologische Sortierung

Nummerierte Vertizes, sodass für  $(u,v)$  in  $E$  gilt:  $u < v$ .



### Algorithmus

Mit einer Laufzeit von  $O(n+m)$ .

**Algorithm** TopologicalSort( $G$ )

```

 $H \leftarrow G$  { Temporary copy of  $G$  }
 $n \leftarrow G.\text{numVertices}()$ 
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  $H$ 
  
```

### Algorithmus mit Tiefensuche

Ebenfalls in  $O(n+m)$  Zeit.

**Algorithm** topologicalDFS( $G$ )

```

Input dag  $G$ 
Output topological ordering of  $G$ 
 $n \leftarrow G.\text{numVertices}()$ 
for all  $u \in G.\text{vertices}()$ 
   $\text{setLabel}(u, \text{UNEXPLORED})$ 
for all  $e \in G.\text{edges}()$ 
   $\text{setLabel}(e, \text{UNEXPLORED})$ 
for all  $v \in G.\text{vertices}()$ 
  if  $\text{getLabel}(v) = \text{UNEXPLORED}$ 
     $\text{topologicalDFS}(G, v)$ 
  
```

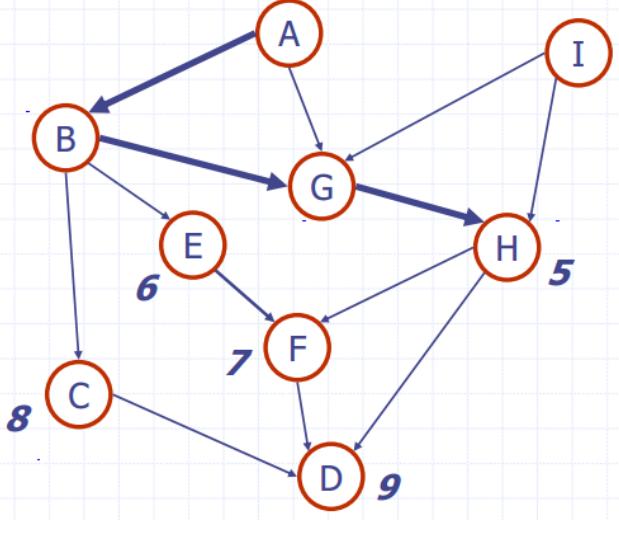
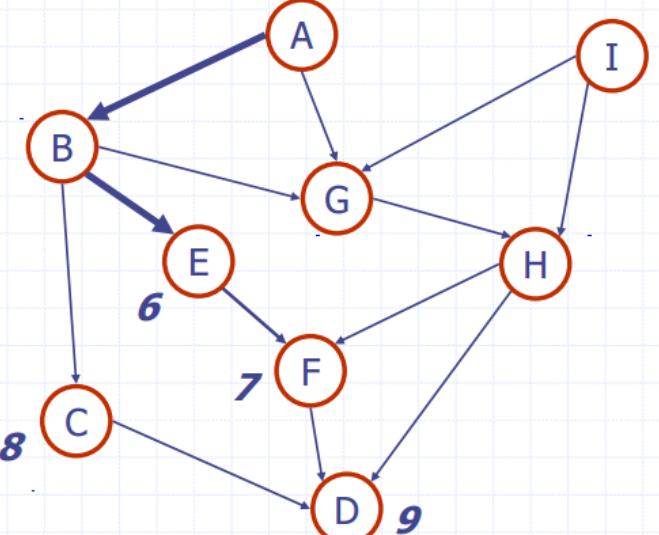
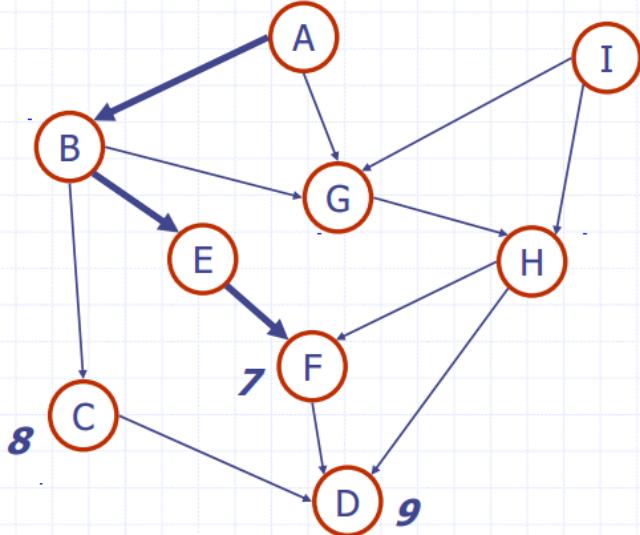
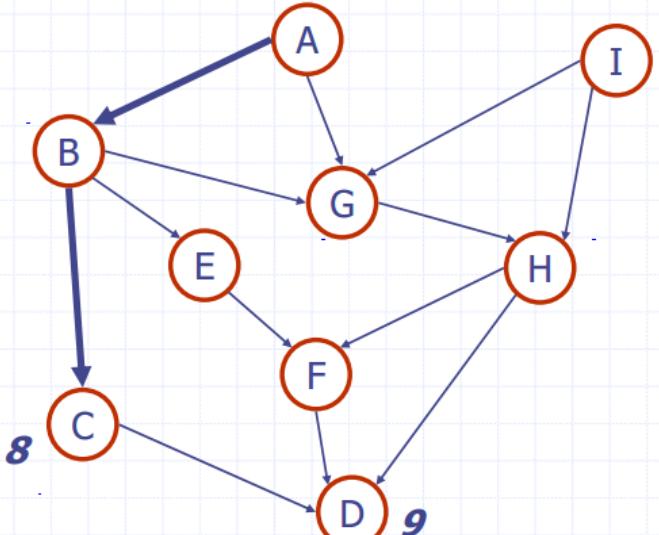
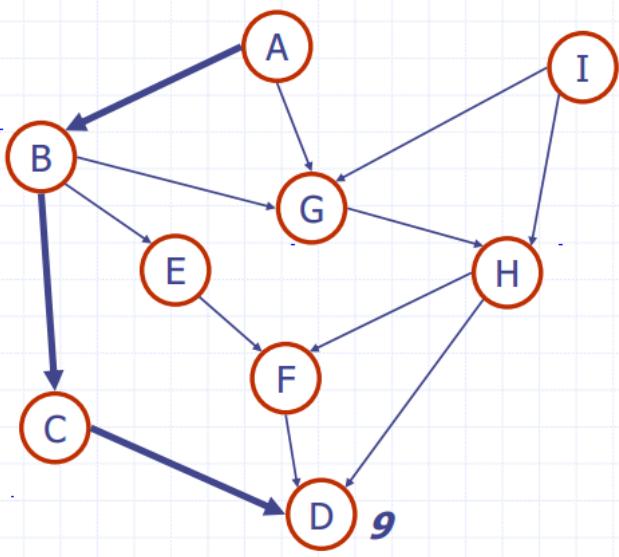
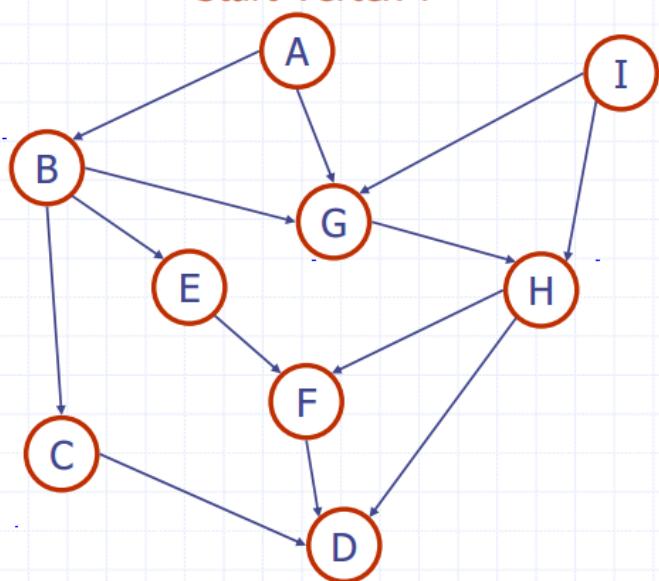
▲  $O(n+m)$  time

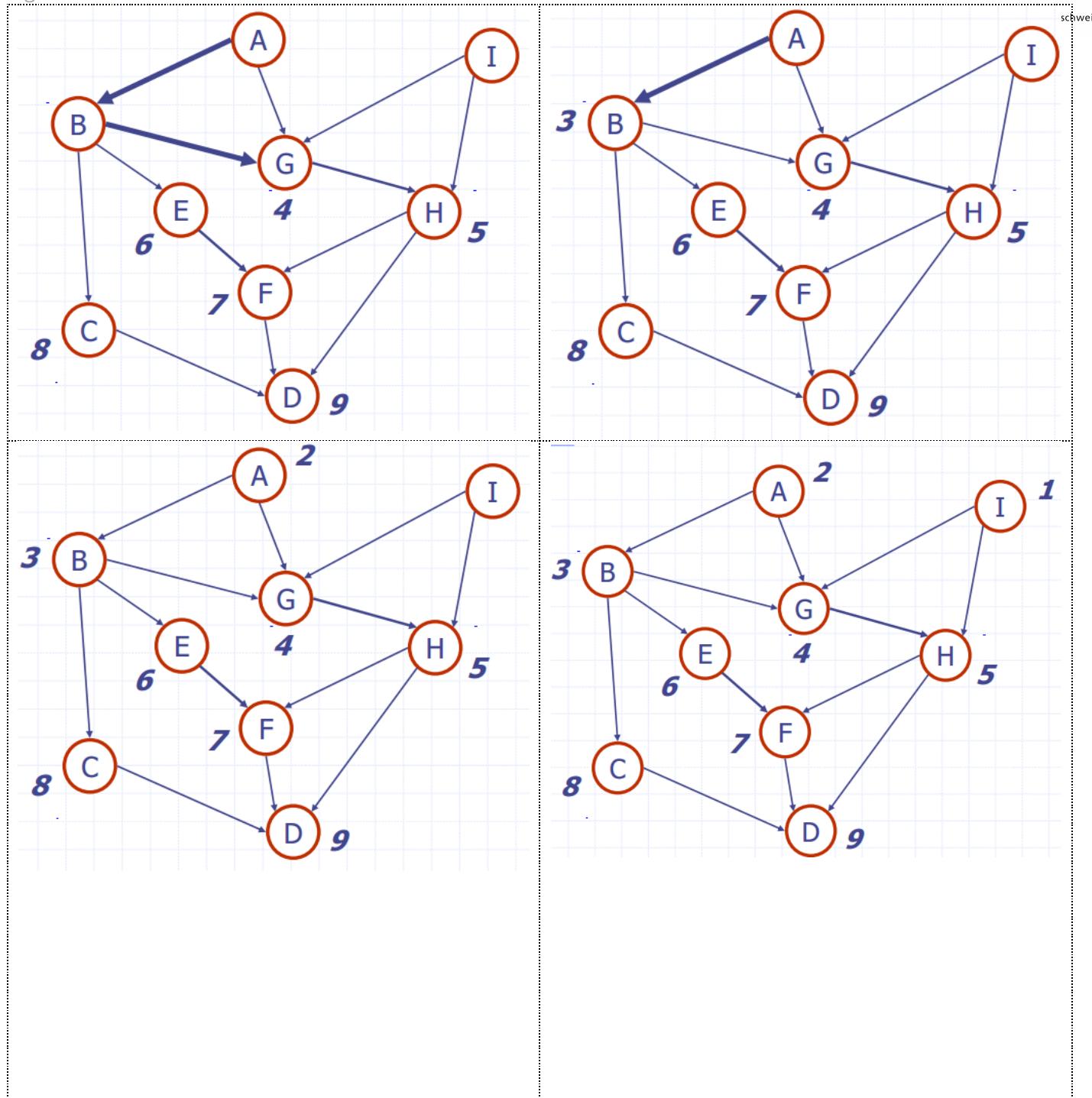
**Algorithm** topologicalDFS( $G, v$ )

```

Input graph  $G$  and a start vertex  $v$  of  $G$ 
Output labeling of the vertices of  $G$ 
  in the connected component of  $v$ 
   $\text{setLabel}(v, \text{VISITED})$ 
for all  $e \in G.\text{outgoingEdges}(v)$ 
  if  $\text{getLabel}(e) = \text{UNEXPLORED}$ 
     $w \leftarrow \text{opposite}(v, e)$ 
    if  $\text{getLabel}(w) = \text{UNEXPLORED}$ 
       $\text{setLabel}(e, \text{DISCOVERY})$ 
       $\text{topologicalDFS}(G, w)$ 
    else
      { $e$  is a forward or cross edge}
     $\text{Label } v \text{ with topological number } n$ 
     $n \leftarrow n - 1$ 
  
```

Start-Vertex  $v$





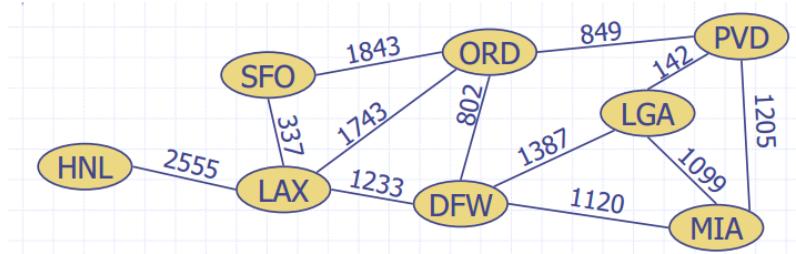
# Shortest Paths Trees / Kürzeste Pfade Bäume

## Gewichtete Graphen

In einem gewichteten Graphen hat jede Kante einen assoziierten nummerischen Wert, das sog. Gewicht der Kante. Die Kanten-Gewichte können z.B. Distanzen, Kosten, etc. repräsentieren.

## Beispiel

In einem Flug-Routen-Graphen repräsentieren die Gewichte der Kanten die Distanz zwischen den Flughäfen.



## Kürzester Pfad

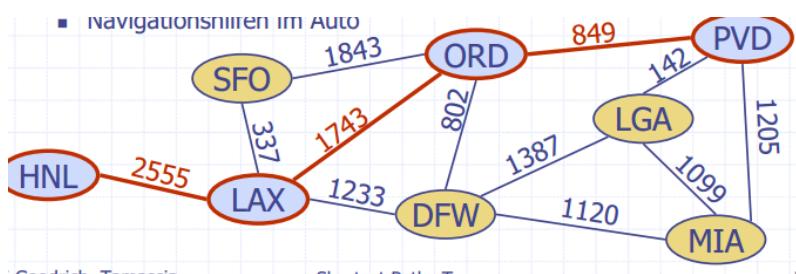
Gegeben sei ein gewichteter Graph mit zwei Vertizes u und v. Wir möchten den Weg mit dem kleinsten totalen Gewicht zwischen u und v finden. Die Länge des Pfades ist die Summe der Gewichte seiner Kanten.

## Beispiel

Kürzester Weg von Providence nach Honolulu

## Anwendungen

Routing im Internet, Flugreservierungen, Navigationshilfen im Auto.



## Eigenschaften

### Eigenschaft 1

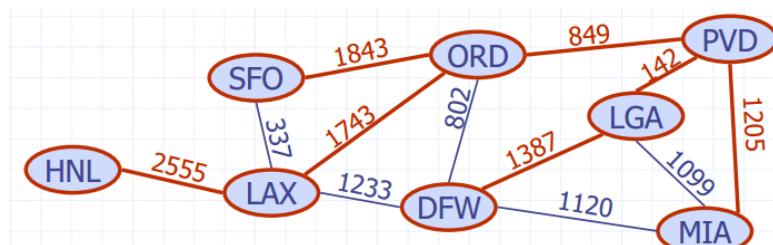
Ein Teilweg eines kürzesten Weges ist selbst auch ein kürzester Weg.

### Eigenschaft 2

Es existiert ein Baum von kürzesten Wegen von einem Start-Vertex zu allen anderen Vertizes.

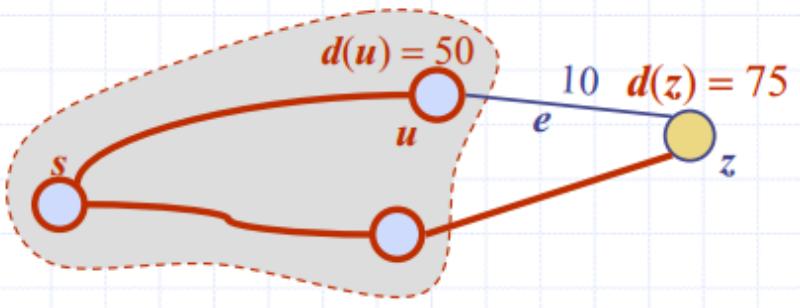
## Beispiel

Baum der kürzesten Wege von Providence



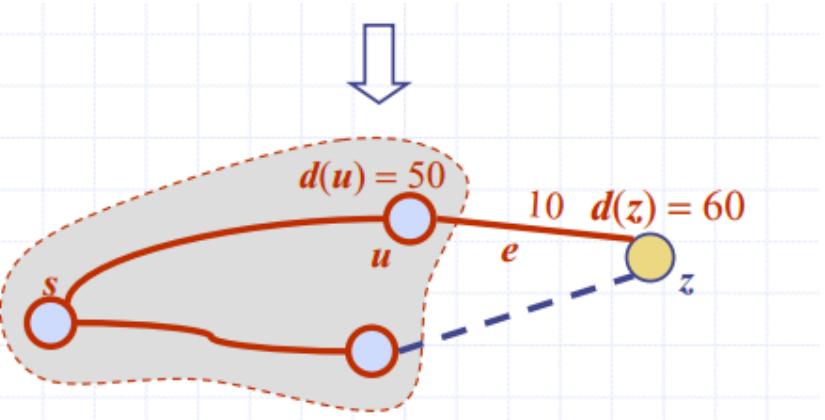
## Schauen wir eine Kante $e = (u, z)$ an, so dass

- $u$  der soeben der Wolke hinzugefügte Vertex ist
- $z$  nicht in der Wolke ist

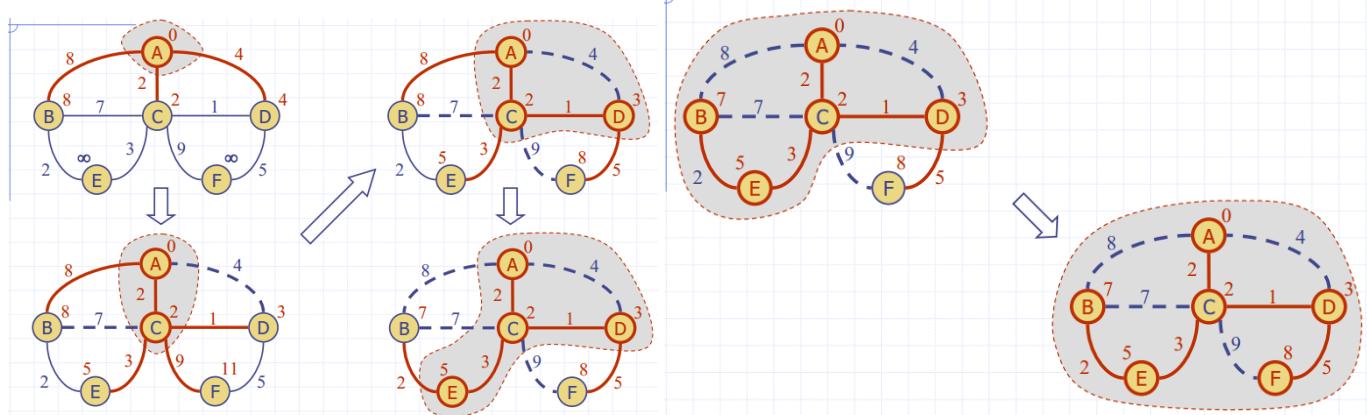


Die Relaxation einer Kante  $e$  aktualisiert die Distanz  $d(z)$  wie folgt:

$$d(z) \leftarrow \min(d(z), d(u) + \text{weight}(e))$$



### Beispiel



## Dijkstra's Algorithmus

Die Distanz eines Vertex v zu einem Vertex s ist die Länge des kürzesten Pfades zwischen s und v. Der Dijkstra Algorithmus berechnet die Distanzen zu allen Vertizes von einem Start-Vertex s aus.

### Annahmen

Der Graph ist verbunden. Die Kanten sind ungerichtete und es gibt keine negative Kantengewichte.

Wir bilden eine Wolke (Cloud) von Vertizes, beginnend mit s und fügen nach und nach alle Vertizes ein. Mit jedem Vertex v speichern wir eine Eigenschaft d(v) welche die Distanz von v zu s im Untergraph (bestehend aus der Wolke und den Nachbar-Vertizes) angibt.

Bei jedem Schritt fügen wir der Wolke den Vertex u hinzu, welcher ausserhalb der Wolke ist und die kleinste Distanz d(u) ausweist. Zudem aktualisieren wir die Distanzen von allen Nachbar-Vertizes von u.

Eine *Adaptierbare Priority Queue* speichert die Vertizes ausserhalb der Wolke

- Key: Distanz
- Element: Vertex

#### Locator-basierte Methoden

- *insert(k,e)* gibt einen Locator zurück
- *replaceKey(l,k)* ändert den Schlüssel eines Eintrags

Wir speichern zwei Eigenschaften (labels) mit jedem Vertex:

- Distanz d(v)
- Locator in der Priority Queue

```

Algorithm DijkstraDistances(G, s)
Q ← neue Heap-basierte Priority Queue
for all v ∈ G.vertices()
  if v = s
    setDistance(v, 0)
  else
    setDistance(v, ∞)
  l ← Q.insert(getDistance(v), v)
  setLocator(v,l)
while ¬Q.isEmpty()
  u ← Q.removeMin().getValue()
  for all e ∈ G.incidentEdges(u)
    { relax edge e }
    z ← G.opposite(u,e)
    r ← getDistance(u) + weight(e)
    if r < getDistance(z)
      setDistance(z,r)
      Q.replaceKey(getLocator(z),r)
```

## Analyse

- Graph Operationen
  - o Die Methode incidentEdges wird für jeden Vertex augerufen.
- Label Operationen
  - o Wir setzen/lesen das Distanz- und das Locator-Label eines Vertex u O(deg(z)) mal.
  - o Setzen/Lesen eines Labels braucht O(1) Zeit.
- Priority Queue Operationen
  - o Jeder Vertex wird einmal hinzugefügt und einmal gelöscht, wobei jedes Einfügen und Löschen O(log n) Zeit braucht.
  - o Der Schlüssel eines Vertex in der Prioirty Queue wird höchstens deg(w) mal geändert und jede Änderung braucht O(log n) Zeit.
- Der Dijkstra's Algorithmus läuft in O((n+m) log n) Zeit, gegeben dass der Graph mit einer Adjazenz Listen Struktur implementiert wird.
  - o Wir erinnern uns, dass  $\sum_v \deg(v) = 2m$
- Die Laufzeit kann auch als (m log n) angegeben werden, da der Graph verbunden ist.

**Algorithm DijkstrasShortestPathsTree( $G, s$ )**

```

...
for all  $v \in G.vertices()$ 
...
setParent( $v, \emptyset$ )
...

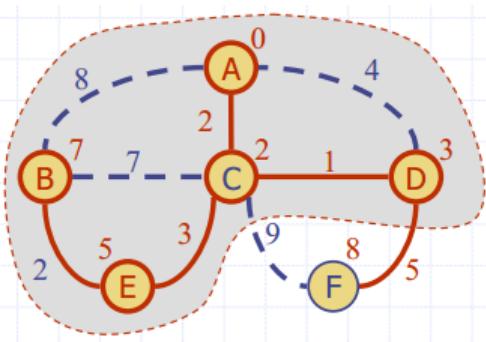
for all  $e \in G.incidentEdges(u)$ 
{ relax edge  $e$  }
 $z \leftarrow G.opposite(u,e)$ 
 $r \leftarrow getDistance(u) + weight(e)$ 
if  $r < getDistance(z)$ 
  setDistance( $z,r$ )
  setParent( $z,e$ )
  Q.replaceKey(getLocator( $z$ ),r)

```

Mit Hilfe des Template Method Patterns können wir Dijkstra's Algorithmus erweitern, damit er einen Baum von kürzesten Wegen vom Start-Vertex aus zu allen anderen Vertizes zurückgibt.

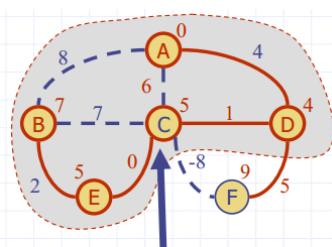
Mit jedem Vertex speichert wir ein drittes Label (Eltern-Kanten im kürzesten Weg). Im Relaxations-(Enspannungs) Schritt aktualisieren wir dieses Label.

## Wieso Dijkstra's Algorithmus funktioniert



Dijkstra's Algorithmus basiert auf der Greedy (gierig) Methode. Angenommen, es würden nicht alle kürzesten Distanzen gefunden. Sagen wir, F sei der erste falsche Vertex, welchen den Algorithmus verarbeitet. Wenn der vorhergehende Vertex D wirklich auf den kürzesten Weg ist, dann stimmt seine Distanz. Aber die Kante (D,F) wurde dann relaxed (entspannt)!. Also solange  $d(F) \geq d(D)$ , kann F's Distanz nicht falsch sein. Also existiert kein falscher Vertex.

## Wieso es für Gewichte kleiner Null nicht funktioniert



C's reale Distanz ist 1,  
aber es ist schon in der  
Menge mit  $d(C)=5$ !

Dijkstra's Algorithmus basiert auf der Greedy (gierig) Methode. Wenn ein Vertex mit einer Kante mit einem Gewicht  $< 0$  später der Wolke hinzugefügt wird, bringt dies die Distanzen für die Vertizes durcheinander.

## Bellman-Ford Algorithmus

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

```

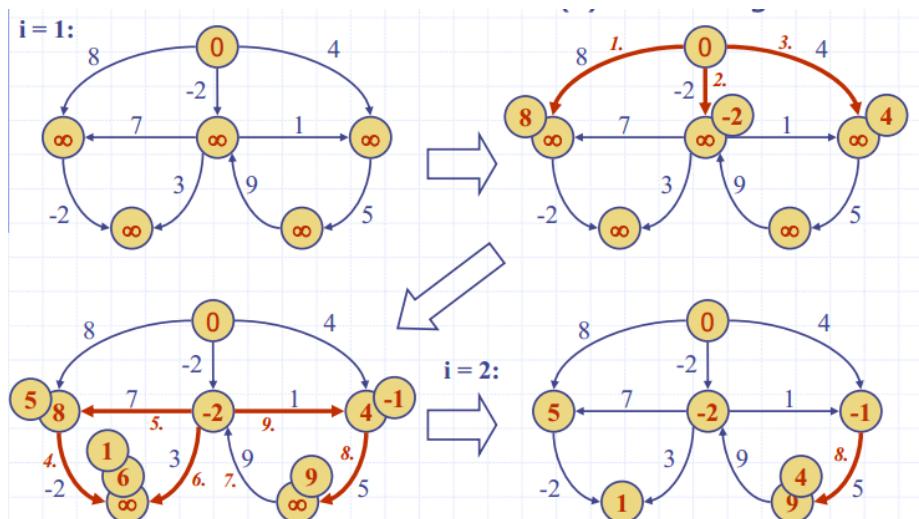
for all  $v \in G.vertices()$ 
  if  $v = s$ 
    setDistance( $v, 0$ )
  else
    setDistance( $v, \infty$ )
for  $i \leftarrow 1$  to  $n-1$  do
  for each  $e \in G.edges()$ 
    { relax edge  $e$  }
     $u \leftarrow G.origin(e)$ 
     $z \leftarrow G.opposite(u,e)$ 
     $r \leftarrow getDistance(u) + weight(e)$ 
    if  $r < getDistance(z)$ 
      setDistance( $z,r$ )

```

Funktioniert auch mit negativ-gewichteten Kanten. Voraussetzung sind aber gerichtete Kanten und keine negativ-gewichtete Schlaufen. Die Iteration i findet alle kürzesten Pfade welche i Kanten benutzt. Die Laufzeit ist  $O(nm)$ .

## Beispiel

Die Vertizes sind mit deren  $d(v)$  Werte angeschrieben.



## DAG-basierter Algorithmus

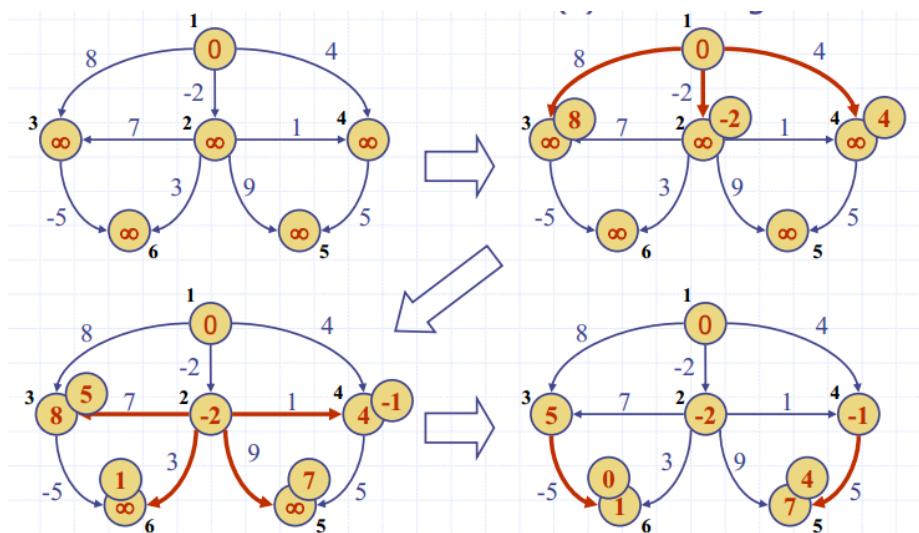
```

Algorithm DagDistances(G, s)
for all v in G.vertices()
    if v = s
        setDistance(v, 0)
    else
        setDistance(v,  $\infty$ )
    Perform a topological sort of the vertices
    for u  $\leftarrow$  1 to n do
        {in topologischer Reihenfolge}
        for each e in G.outEdges(u)
            { relax edge e }
            z  $\leftarrow$  G.opposite(u, e)
            r  $\leftarrow$  getDistance(u) + weight(e)
            if r < getDistance(z)
                setDistance(z, r)
    
```

Funktioniert auch mit negativ-gewichteten Kanten. Benutzt eine topologische Reihenfolge. Zudem benutzt es keine ausgefallenen Datenstrukturen. Der Algorithmus ist viel schneller als jener von Dijkstra. Die Laufzeit ist  $O(n+m)$ .

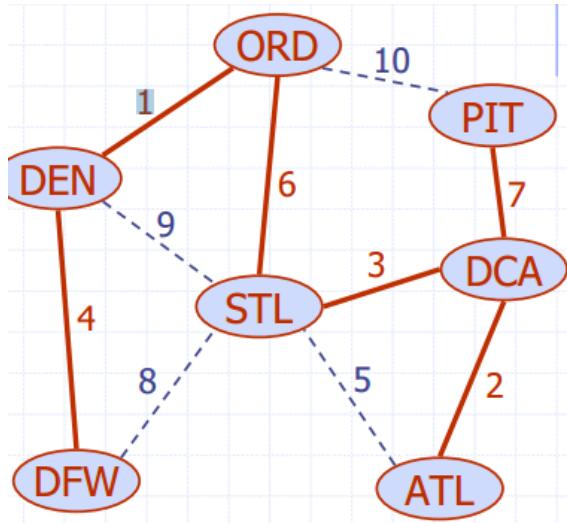
## Beispiel

Die Vertizes sind mit deren  $d(v)$  Werte angeschrieben.



# Minimum Spanning Trees

## (Minimal aufspannende Bäume)



Aufspannender Subgraph

Subgraph eines Graphen G beinhaltend alle Vertizes von G

Aufspannender Baum

Aufspannender Subgraph, der selbst ein (freier) Baum ist.

Minimal Aufspannender Baum (MST)

Aufspannender Baum eines gewichteten Graphen mit minimalem totalen Kantengewicht.

Anwendungen

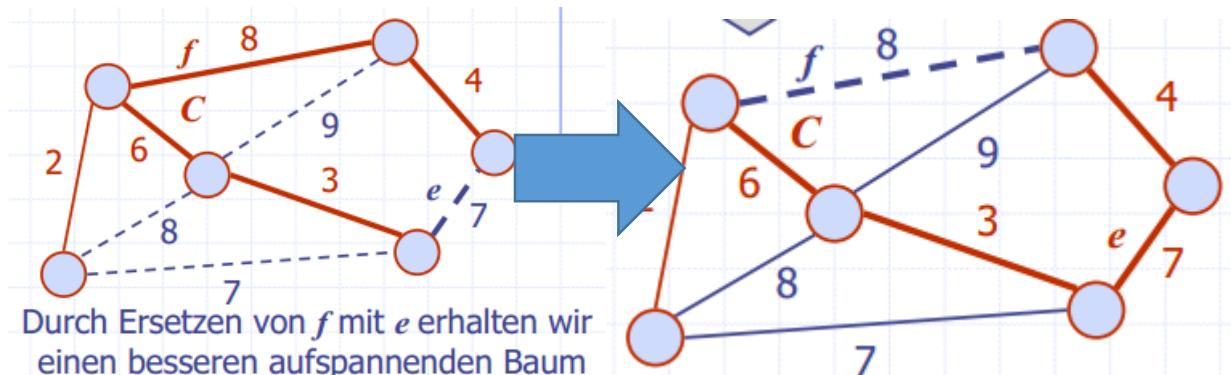
Kommunikationsnetzwerke, Transportnetzwerke

### Schlaufen-Eigenschaft

T sei ein minimal aufspannender Baum eines gewichteten Graphen G. e sei eine Kante von G, welche nicht in T ist und C sei die Schlaufe, die durch e mit T entsteht. Für jede Kante f von C  $\text{weight}(f) \leq \text{weight}(e)$ .

### Beweis

Per Widerspruch. Wenn  $\text{weight}(f) > \text{weight}(e)$  können wir einen besseren aufspannenden Baum erreichen indem wir e mit f ersetzen.

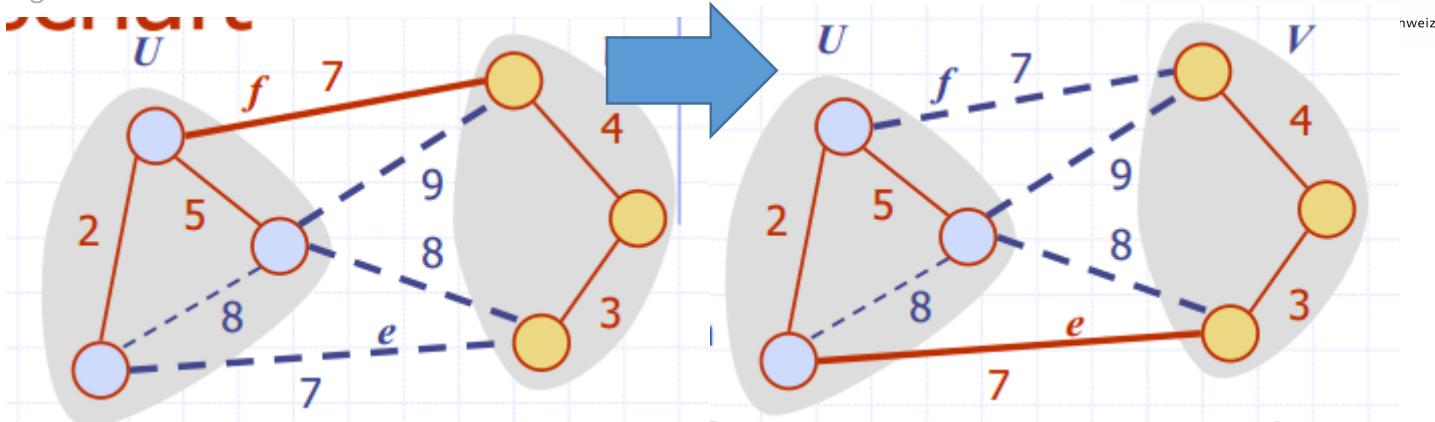


### Aufteilungs-Eigenschaft

Sei G aufgeteilt in zwei Vertex Teilmengen U und V. Sei e eine Kante mit dem kleinsten Gewicht zwischen den Partitionen. Es existiert ein minimal aufspannender Baum von G der die Kante e beinhaltete.

### Beweis

Sei T ein MST von G. Wenn T e nicht beinhaltet: nehmen wir an, es existiert eine Schleife C durch e mit T und sei f eine Kante von C zwischen den Partitionen. Durch die Schlaufen-Eigenschaft  $\text{weight}(f) \leq \text{weight}(e)$ . Also  $\text{weight}(f) = \text{weight}(e)$ . Wir erhalten wieder einen MST, wenn wir f mit e ersetzen.



## Kruskal's Algorithmus

```

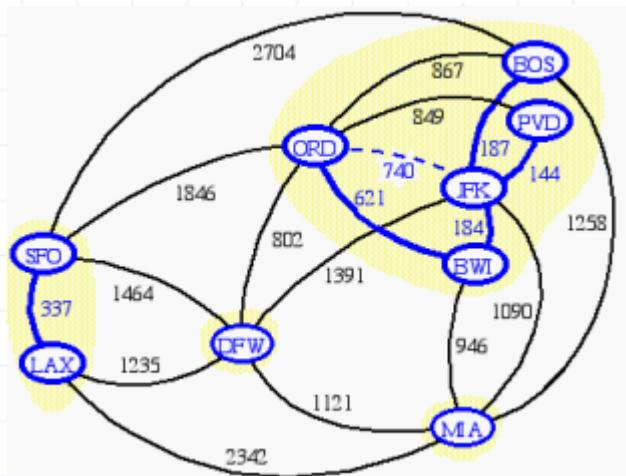
Algorithm KruskalMST( $G$ )
  for jeden Vertex  $V$  in  $G$  do
    definiere eine  $Cloud(v)$  of  $\leftarrow \{v\}$ 
   $Q$ : eine Priority Queue
  Alle Kanten in  $Q$  einfügen mit dem Gewicht als Key
   $T \leftarrow \emptyset$ 
  while  $T$  weniger als  $n-1$  Kanten hat do
    edge  $e = Q.removeMin()$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

```

Eine Priority Queue speichert die Kanten ausserhalb der Wolke. Key: Gewicht, Element: Kante.

Zum Schluss des Algorithmus haben wir eine Wolke welche den MST umfasst und ein Baum  $T$  welcher unser MST ist.

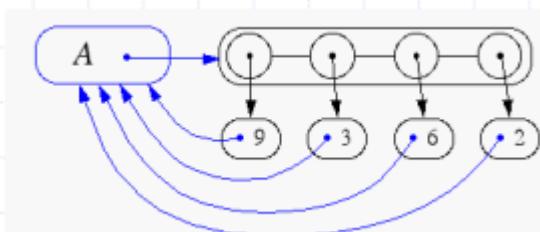
## Datenstruktur



Der Algorithmus behält einen Wald von Bäumen. Eine Kante ist akzeptiert, wenn Sie verschiedene Bäume verbindet. Wir brauchen eine Datenstruktur welche eine Partition verwaltet. Zum Beispiel eine Sammlung von disjunkten Sets mit folgenden Operationen:

- Find( $u$ ): Gibt ein Set  $U$  zurück enthaltet  $u$ .
- Union( $u, v$ ): ersetzt die Sets, welche  $u$  und  $v$  speichern mit deren Vereinigung.

## Repräsentation einer Partition



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

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

- In der Operation  $\text{union}(u,v)$  verschieben wir die Elemente des kleineren Sets in die Sequenz des grösseren Sets und aktualisieren deren Referenzen.
- Die Zeit für die Operation  $\text{union}(u,v)$  ist  $\min(n_u, n_v)$ , wobei  $n_u$  und  $n_v$  die Grössen der Sets sind , die  $u$  und  $v$  beinhalten.

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

### Partition-Basierte Implementation

Eine Partition-basierte Version von Kruskal's Algorithmus führt Wolken-Vereinigungen mit union's und Test mit find's aus.

#### Algorithm Kruskal( $G$ ):

**Input:** Ein gewichteter Graph  $G$ .

**Output:** Ein MST  $T$  für  $G$ .

$P$  sei eine Partition der Vertizes von  $G$ , wobei jeder Vertex ein Set für sich bildet

$Q$  sei eine Priority Queue, welche die Kanten von  $G$  nach Gewichtung sortiert

$T$  sei ein ursprünglich leerer Baum

**while**  $Q$  is nicht leer **do**

$(u,v) \leftarrow Q.\text{removeMinElement}()$

**if**  $P.\text{find}(u) \neq P.\text{find}(v)$  **then**

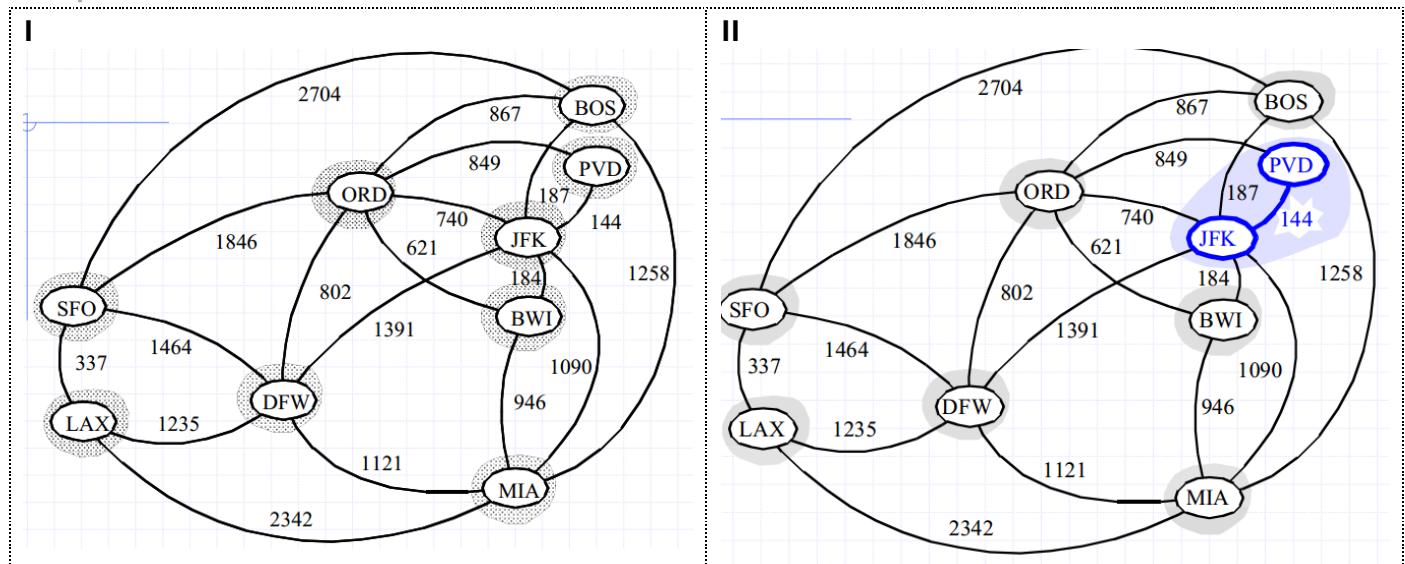
        Add  $(u,v)$  to  $T$

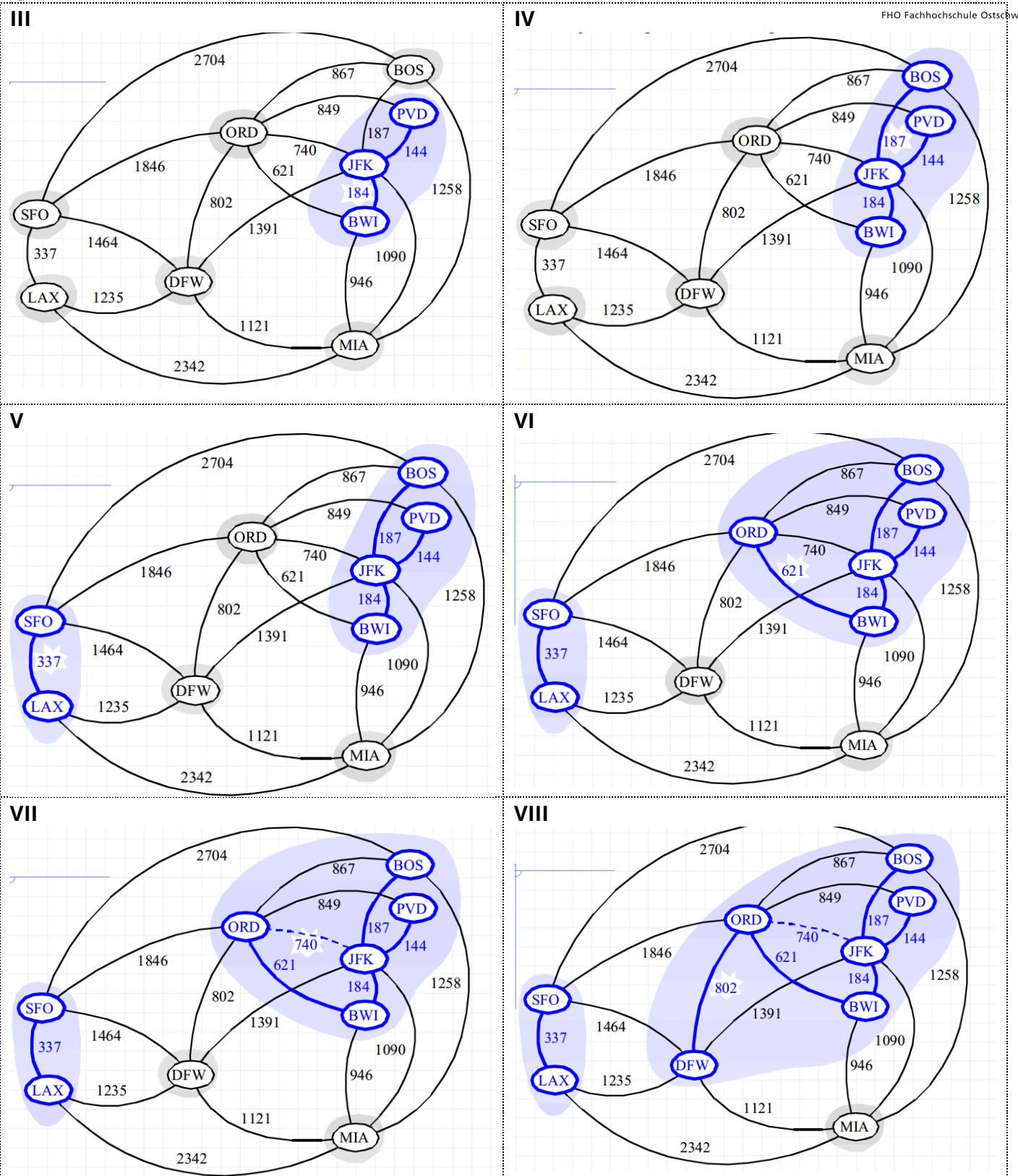
$P.\text{union}(u,v)$

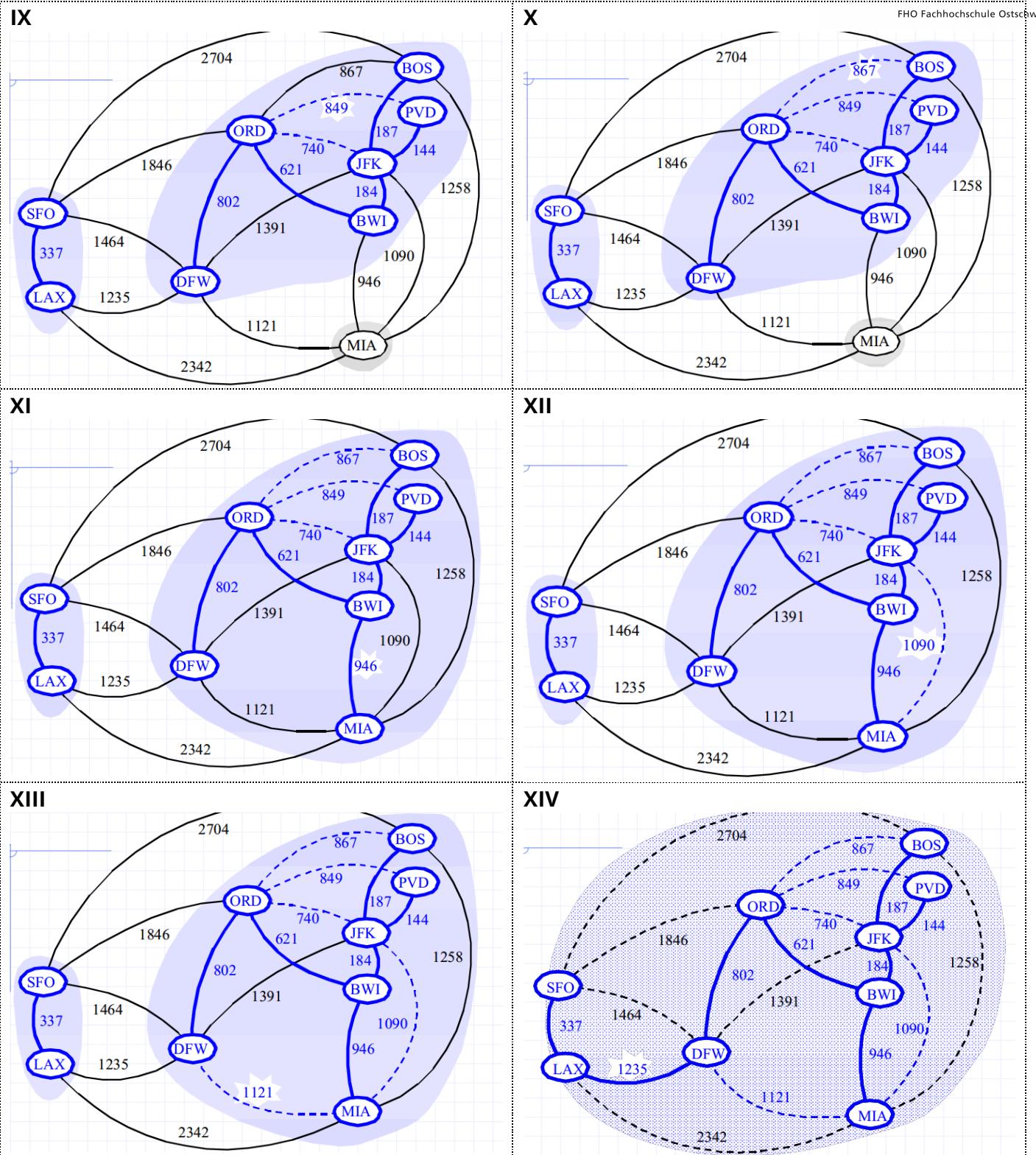
**return**  $T$

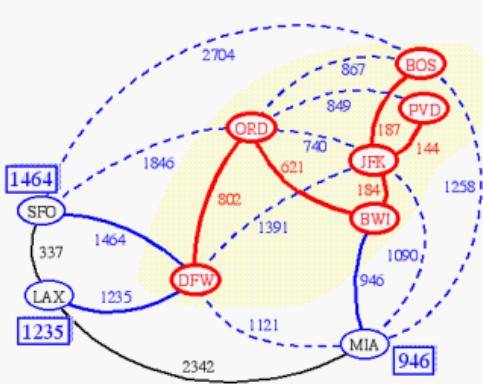
Laufzeit:  
 $O(m \log n)$

### Beispiel









Gleich wie Dijkstra's Algorithmus (für einen verbundenen Graphen). Wir nehmen einen beliebigen Vertex  $s$  und generieren den MST als eine Wolke von Vertizes von  $s$ . Wir speichern zu jedem Vertex ein Label  $d(v)$  = die kleinste Gewichtung einer Kante, welche  $v$  mit einem Vertex der Wolke verbindet.

Für jeden Schritt fügen wir der Wolke den Vertex  $u$  von ausserhalb der Wolke mit dem kleinsten Distanz-Label hinzu. Und wir aktualisieren die Labels der Nachbar-Vertizes von  $u$ .

```

Algorithm PrimJarnikMST( $G$ )
 $Q \leftarrow$  neue Heap basierte Priority Queue
 $s \leftarrow$  ein Vertext von  $G$ 
for all  $v \in G.\text{vertices}()$ 
  if  $v = s$ 
     $\text{setDistance}(v, 0)$ 
  else
     $\text{setDistance}(v, \infty)$ 
     $\text{setParent}(v, \emptyset)$ 
     $l \leftarrow Q.\text{insert}(\text{getDistance}(v), v)$ 
     $\text{setLocator}(v, l)$ 
while  $\neg Q.\text{isEmpty}()$ 
   $u \leftarrow Q.\text{removeMin}()$ 
  for all  $e \in G.\text{incidentEdges}(u)$ 
     $z \leftarrow G.\text{opposite}(u, e)$ 
    if  $Q.\text{contains}(z)$ 
       $r \leftarrow \text{weight}(e)$ 
      if  $r < \text{getDistance}(z)$ 
         $\text{setDistance}(z, r)$ 
         $\text{setParent}(z, e)$ 
         $Q.\text{replaceKey}(\text{getLocator}(z), r)$ 

```

Eine Priority Queue speichert die Vertizes ausserhalb der Wolke. Der Key ist die Distanz, das Element der Vertex.

Dazu gibt es Locator-basierte Methoden

- Insert( $k, e$ ) gibt einen Locator zurück
- replaceKey( $l, k$ ) ändert den Schlüssel eines Eintrags.

Wir speichern drei Labels zu jedem Vertext:

- Distanz
- Elternkante MST
- Locator in Priority Queue.

### Beispiel

