Algorithmen und Datenstrukturen

Zusammenfassung

1 CONTENTS

2 MULTIMAP

Eine Multimap ist eine Erweiterung einer Map, bei der mehrere Values einem bestimmten Key zugeordnet und für diesen zurückgegeben werden können.

Methoden:

- **find(k)**: liefert den ersten Value-Entry zum Schlüssel k
- **findAll(k)**: liefert eine iterierbare Collection mit allen Value-Entries zum Schlüssel k
- **insert(k,o)**: neuen Entry zum Schlüssel k mit Value o einfügen
- **remove(e)**: entfernt den Entry e von seinem Schlüssel (und Rückgabe von e)

2.1 GEORDNETE MULTIMAP

Bei dieser Art von Multimap sind die Key-Value Paare nach Key aufsteigend sortiert. ($k_m <= k_n$)

Sie verfügt zusätzlich zu den Multimap Methoden folgende Methoden:

- **first()**: liefert den ersten Key-Value-Entry in der Multimap-Ordnung
- **last()**: liefert den letzten Key-Value-Entry in der Multimap-Ordnung
- **successors(k)**: liefert einen Iterator über die Key-Value-Entries mit einem Schlüssel, der grösser oder gleich k ist (nicht abnehmende Ordnung)
- **predecessors(k)**: liefert einen Iterator über die Key-Value-Entries mit einem Schlüssel, der kleiner oder gleich k ist (nicht zunehmende Ordnung)

2.2 SUCHTABELLE

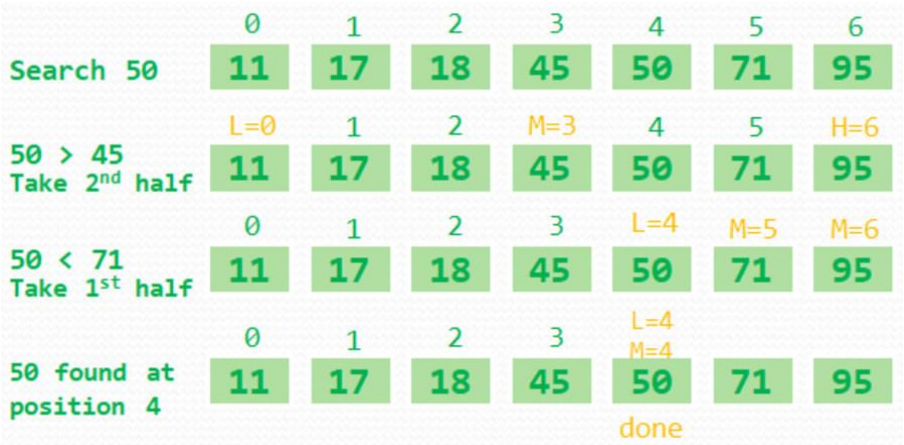
Eine Suchtabelle ist eine Multimap, welche mithilfe einer sortierten Sequenz implementiert wird. Die Key-Value-Entries werden in einer Array-basierten Sequenz abgespeichert, sortiert nach Schlüssel. Sie ist dann effektiv, wenn die Multimap klein ist und vor allem Such-Operationen ausgeführt werden.

Laufzeiten:

- **insert(k,o): O(n)**
  Im schlimmsten Fall wird an Position 0 des Arrays eingefügt, dann müssen alle Entries eins nach hinten gschoben werden.
- **remove(e): O(n)**
  Im schlimmsten Fall wird an Position 0 des Arrays gelöscht, dann müssen alle Entries eins nach vorne geschoben werden.

2.2.1 Binäre Suche

Beispiel für die Operation find(50) in einer Suchtabelle:

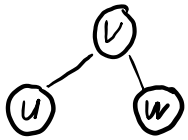


Bei jedem Schritt wird die Anzahl Kandidaten halbiert.
Laufzeit find(k): O(log(n))

3 Binärer Such-Baum

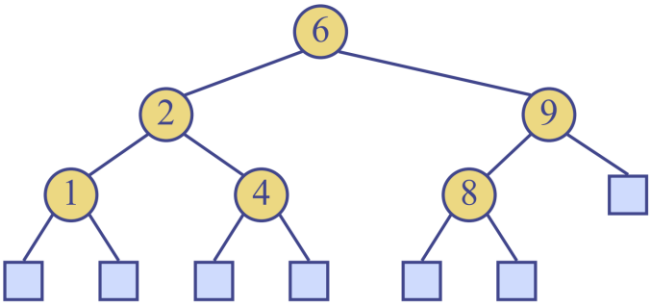
3.1 Eigenschaften

- Ist ein binärer Baum, das heisst Knoten haben höchstens zwei direkte Nachkommen, die sich eindeutig in ein linkes und ein rechtes Kind einteilen lassen.
- Speichert Key-Value-Entries in seinen internen Knoten
- u ist im linken Teilbaum und w im rechten Teilbaum von v
  key(u) ≤ key(v) ≤ key(w)



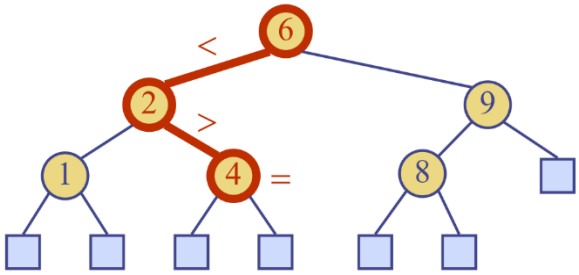
- Inorder-Traversierung (linker Teilbaum – Wurzel – rechter Teilbaum) besucht die Schlüssel in nicht absteigender Reihenfolge
- Externe Knoten (Blatt-Knoten) speichern keine Daten

3.2 Beispiel



3.3 Suche

Man beginnt bei der Wurzel (6), vergleicht den Schlüssel mit dem gesuchten Schlüssel und macht je nach Ergebnis links (Wurzel-Schlüssel zu gross) oder rechts (Wurzel-Schlüssel zu klein) weiter. Wenn ein Blatt erreicht wurde, wurde der Schlüssel nicht gefunden. Beispiel für find(4):



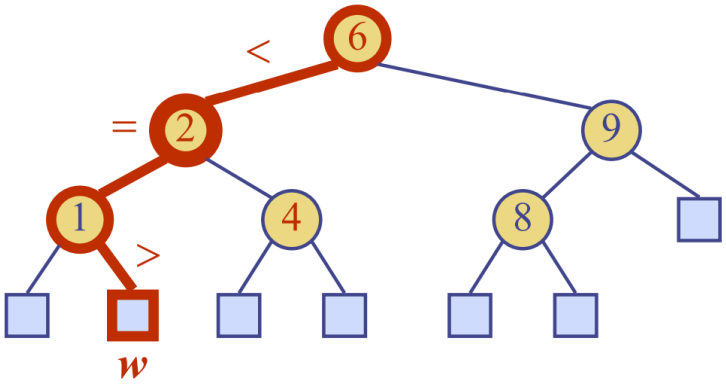
3.4 Einfügen

Zuerst wird nach dem Schlüssel gesucht. Nun gibt es zwei Optionen: Schlüssel wird gefunden oder nicht.

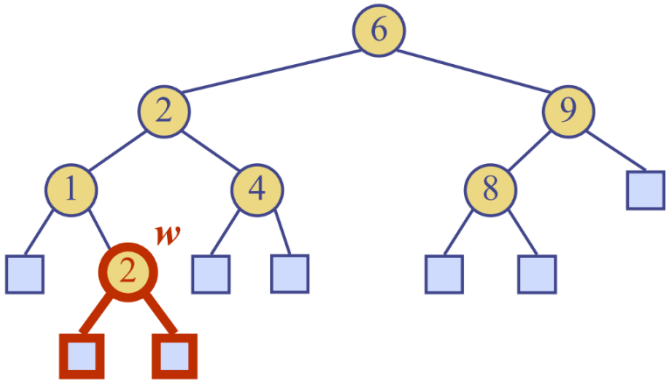
3.4.1 Schlüssel wird gefunden

Wenn wir den Schlüssel gefunden, suchen wir im linken Teilbaum des Treffers weiter, bis wir auf ein Blatt stossen. Dieses Blatt ist der Ort für den neuen Knoten. Der neue Knoten muss auch noch zu einem internen Knoten expandiert werden. Beispiel insert(2):

find(2):



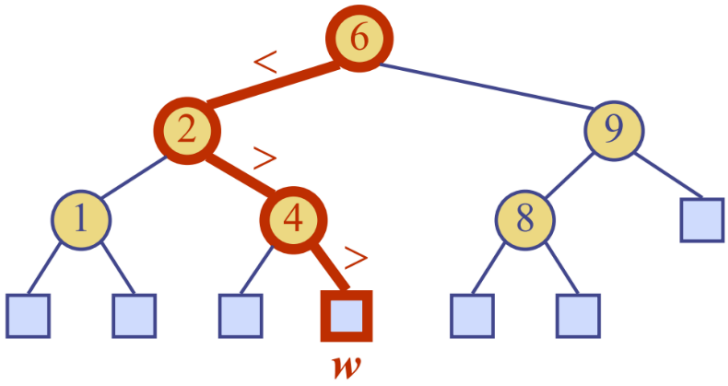
insert(2):



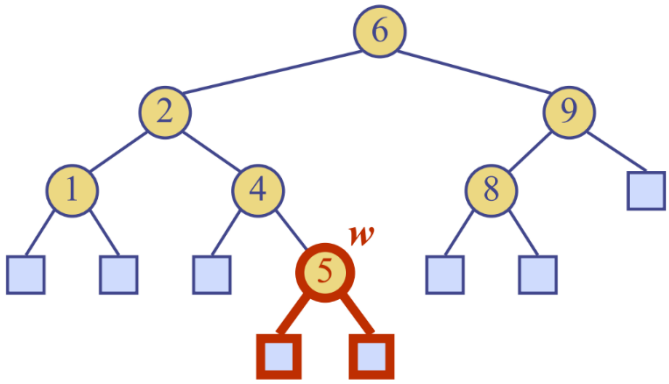
3.4.2 Schlüssel wird nicht gefunden

Wird der Schlüssel nicht gefunden, ist die Suche ja bei einem Blatt gelandet. Dieses Blatt ist der Ort für den neuen Knoten. Der neue Knoten muss auch noch zu einem internen Knoten expandiert werden. Beispiel insert(5):

find(5):



insert(5):

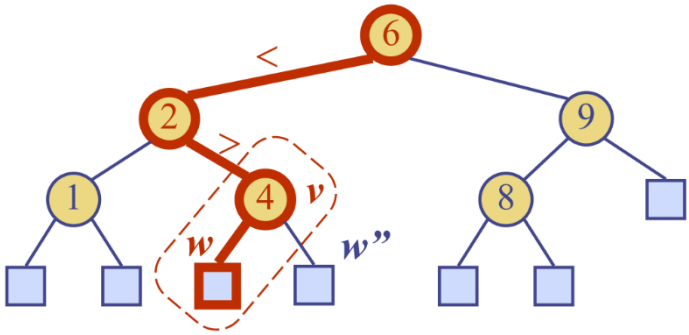


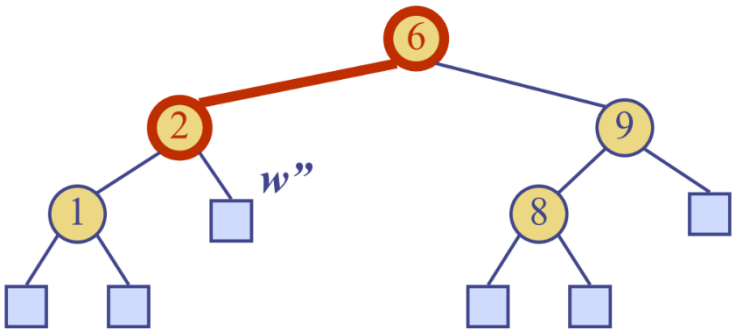
3.5 Löschen

Beim Löschen suchen wir zuerst den Schlüssel. Danach gibt es drei Fälle

3.5.1 Knoten hat zwei Blatt-Kinder

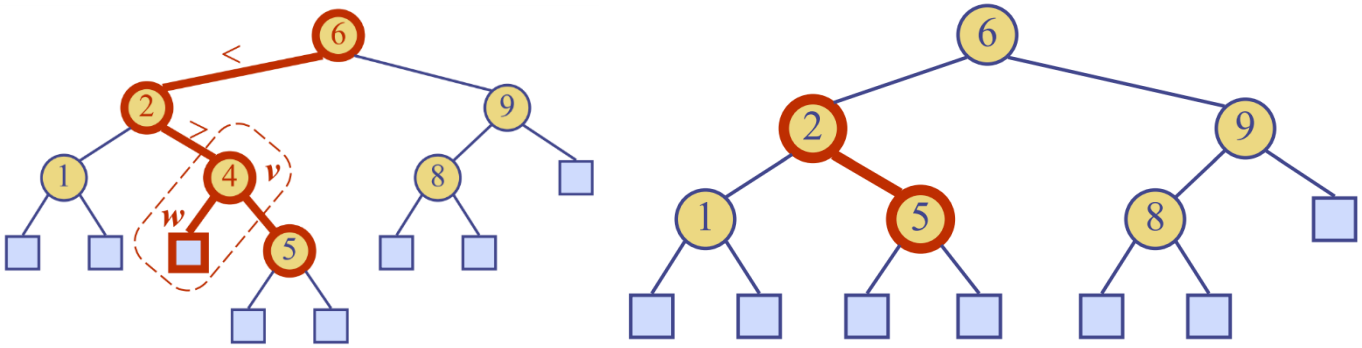
Das Dreierpaket linkes Blatt, zu löschender Knoten und rechtes Blatt wird durch nur noch das rechte Blatt ersetzt. Beispiel remove(4):





3.5.2 Knoten hat ein Blatt-Kind

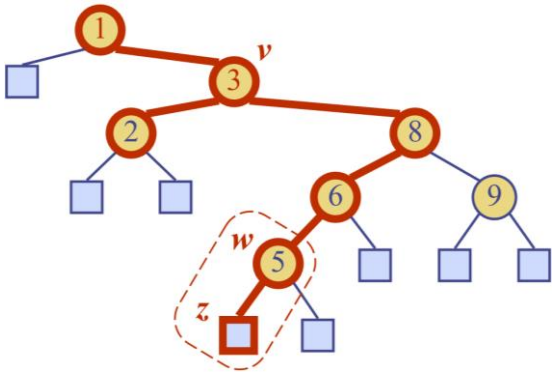
Das Dreierpaket linkes Blatt, zu löschender Knoten und rechtes Blatt wird durch den einzigen Nachkommen des zu löschendenden Knotens ersetzt. Beispiel remove(4):



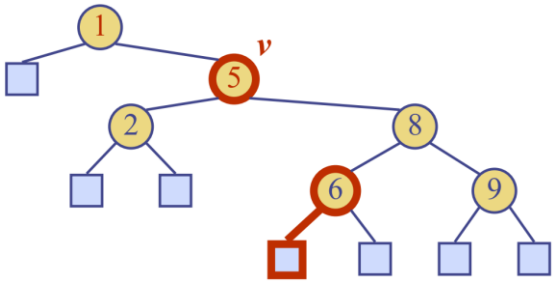
3.5.3 Knoten hat kein Blatt-Kind

Dies ist der komplizierteste Fall. Beispiel remove(3):

1. Den Nachfolger des zu löschenden Knotens in der Inorder-Traversierung finden (hier 5)



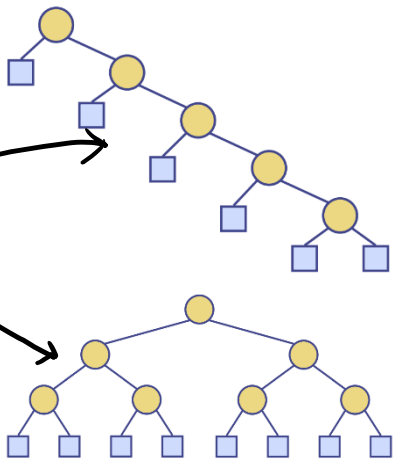
2. Den Nachfolger (w) an die Position des zu löschenden Knotens (v) kopieren
3. Den Nachfolger (w) und sein linkes Kind (z, welches ein Blatt sein muss) löschen



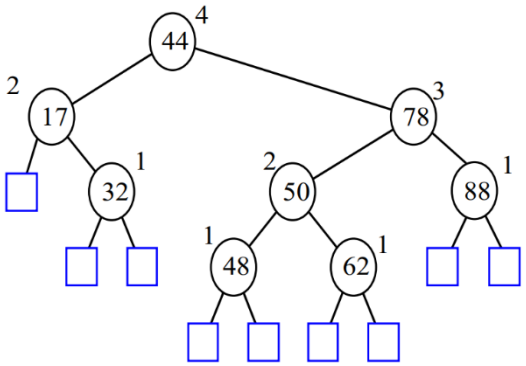
3.6 PERFORMANCE

n = Anzahl Key-Value-Entries, h = Höhe des Baumes

- Benötigter Speicher: O(n)
- Find, Insert & Remove: O(h)
- h = O(n) im schlechtesten Fall
- h = O(log(n)) im besten Fall



3.7 AVL Baum



Ein AVL-Baum ist ein binärer Suchbaum, bei dem für jeden internen Knoten v gilt: die Höhe der Kinder von v unterscheiden sich höchstens um 1. Sie sind also immer balanciert. Im Beispiel unten sind die Höhen neben den Knoten notiert.

- Für die Höhe eines AVL-Baums gilt: O(log(n))
- Balance eines Knotens: b(k) = Höhe(links) – Höhe(rechts)
- B(k) immer gleich -1, 0 oder 1
- Nach dem Einfügen eines neuen Knotens kann b(k) auch noch gleich -2 oder 2 sein

3.7.1 Einfügen

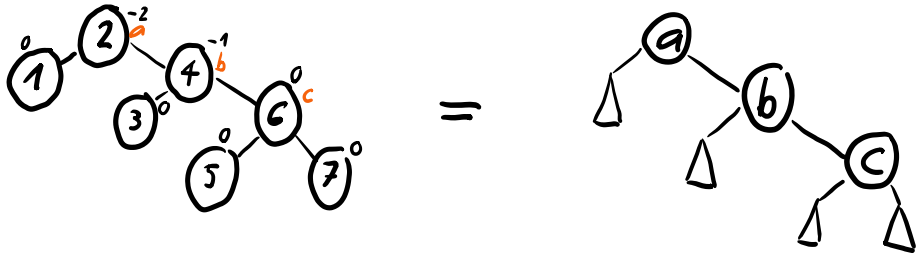
Das Einfügen funktioniert grundsätzlich gleich wie beim binären Suchbaum, der Schlüssel wird gesucht und dann externe Knoten expandiert. Es kann jedoch passieren, dass die AVL-Balance nicht mehr erfüllt ist, dann muss rotiert werden.

3.7.2 Trinode Umstrukturierung

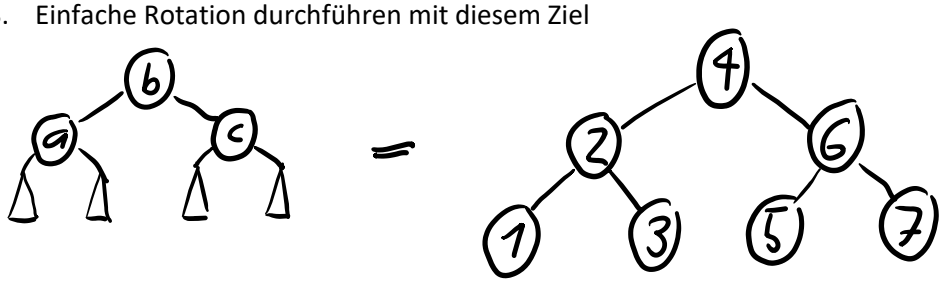
- Fall 1: Einfache Links-Rotation um a
- Fall 2: Einfache Rechts-Rotation um a
- Fall 3: Einfache Rechts-Rotation um c, dann eine Links-Rotation um a
- Fall 4: Einfache Links-Rotation um a, dann eine Rechts-Rotation um c

3.7.2.1 Einfache Rotation

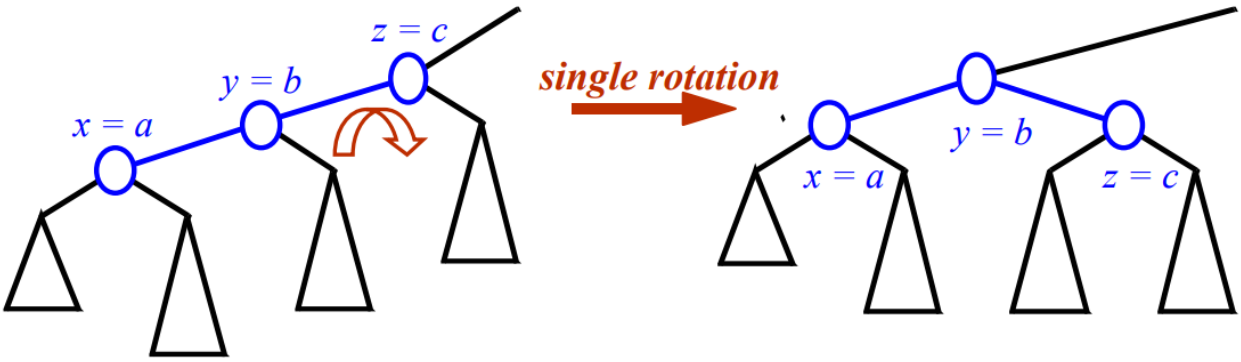
1. Ablauf gilt für Fall 1. Den Knoten bestimmen, wo b(k) = 2 oder -2 ist
2. Die beiden In-Order Nachfolger mit Kindern bestimmen, den Knoten in In-Order Reihenfolge a, b, c zuweisen



3. Einfache Rotation durchführen mit diesem Ziel

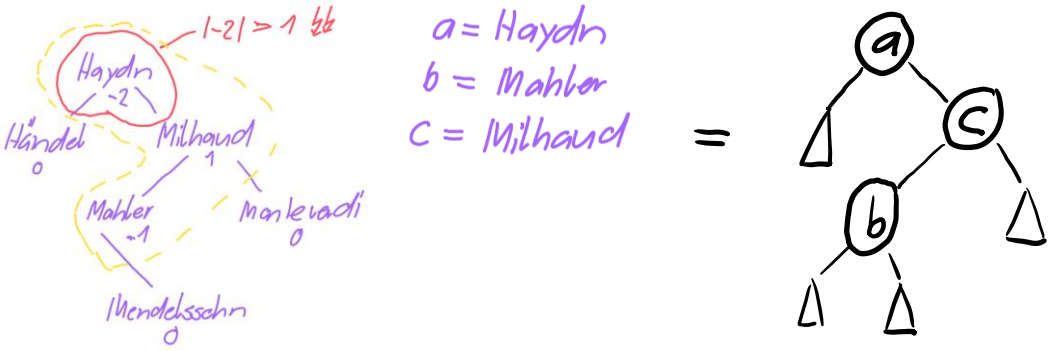


Es könnte auch so aussehen (Fall 2):

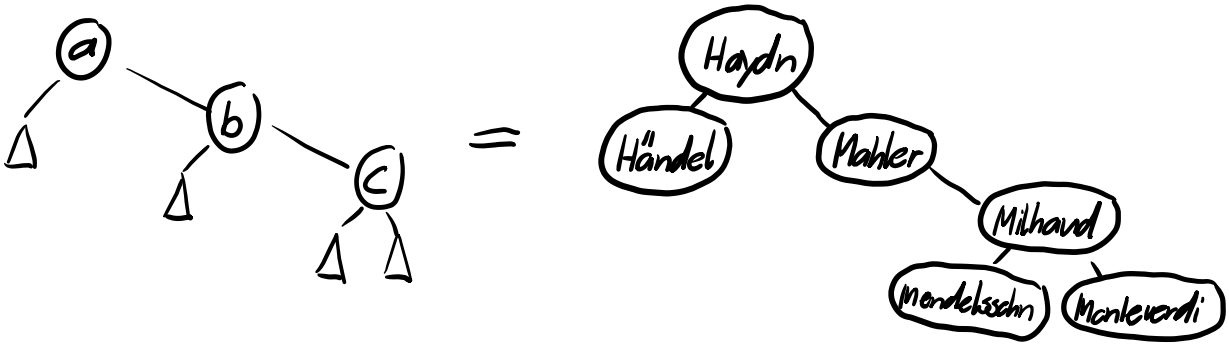


3.7.2.2 Doppelte Rotation

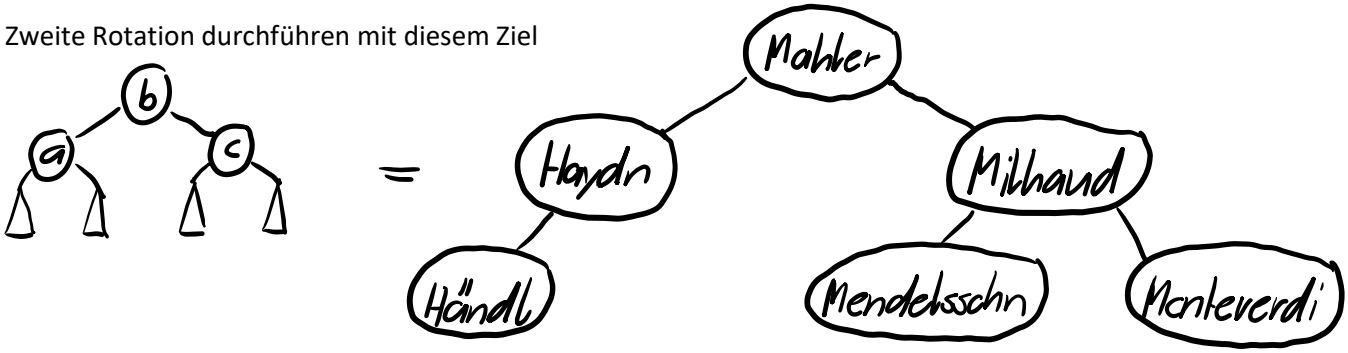
1. Ablauf gilt für Fall 3. Den Knoten bestimmen, wo b(k) = 2 oder -2 ist
2. Die beiden In-Order Nachfolger mit Kindern bestimmen, den Knoten in In-Order Reihenfolge a, b, c zuweisen



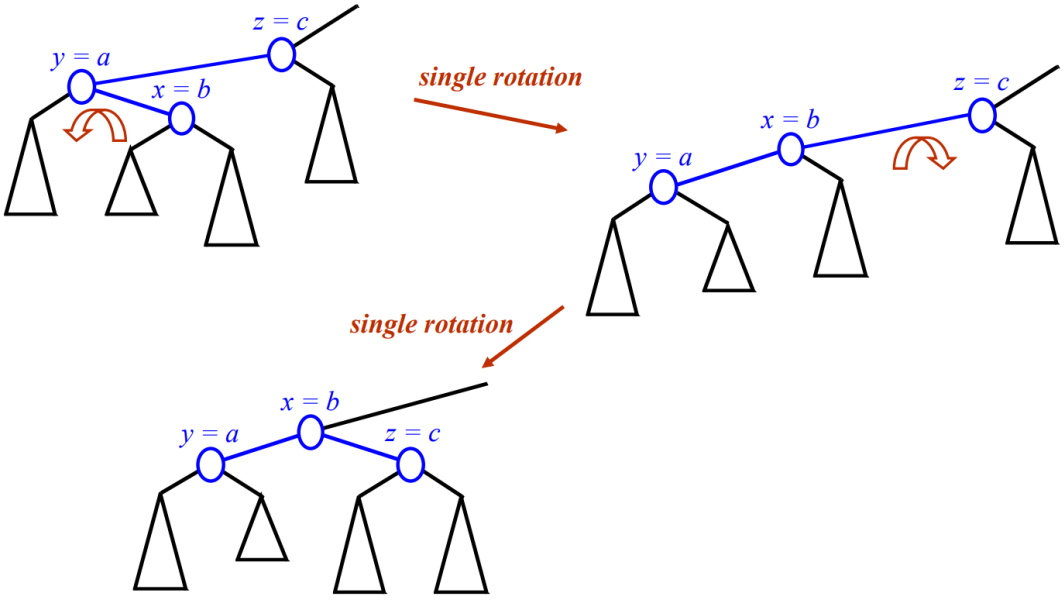
3. Erste Rotation durchführen



4. Zweite Rotation durchführen mit diesem Ziel



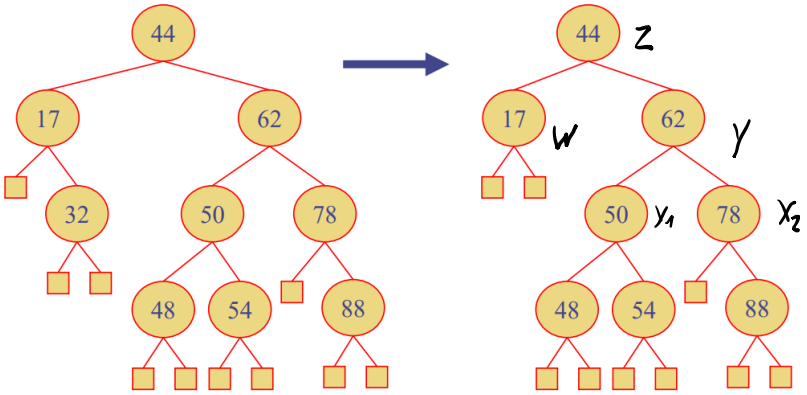
Es könnte auch so aussehen (Fall 4):



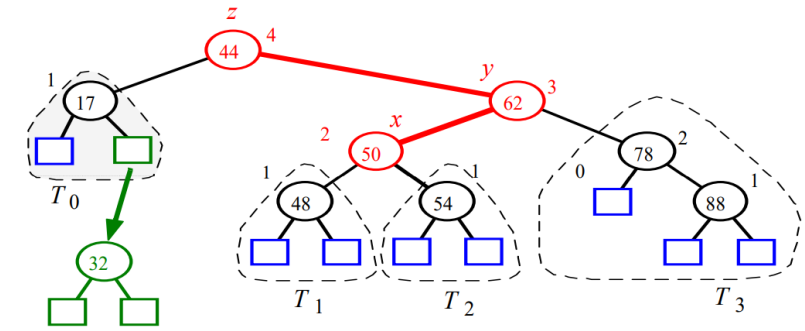
3.7.3 Löschen

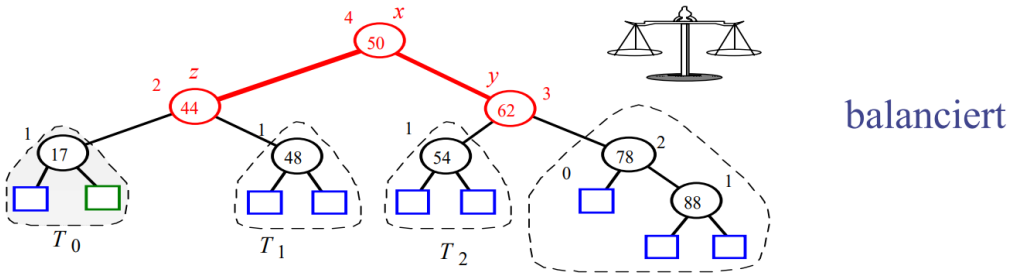
Das Löschen beginnt wie im binären Suchbaum, die Balance kann daher danach nicht mehr gegeben sein.

Beispiel Löschen von 32:

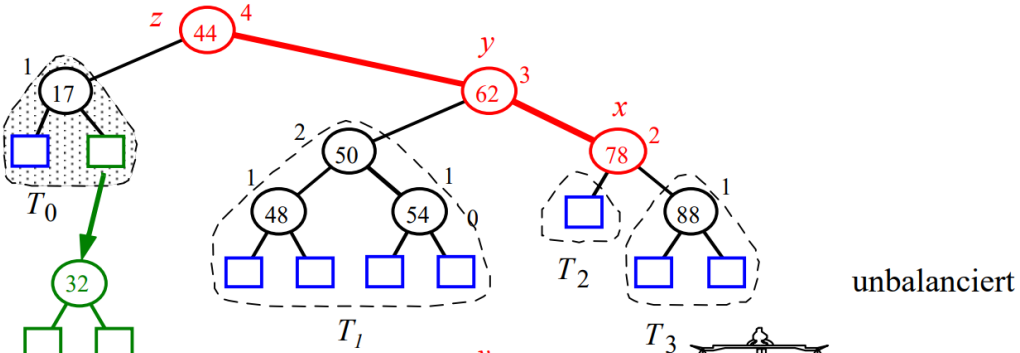


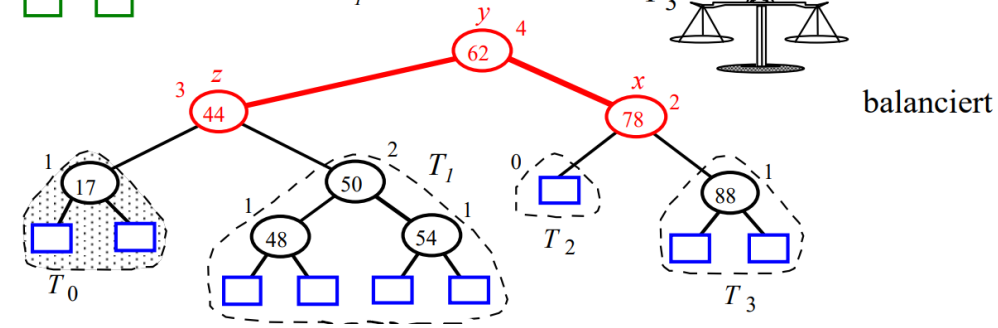
Cut/Link auf x_1:





Cut/Link auf x_2:





3.7.4 Cut/Link Restrukturierungs-Algorithmus

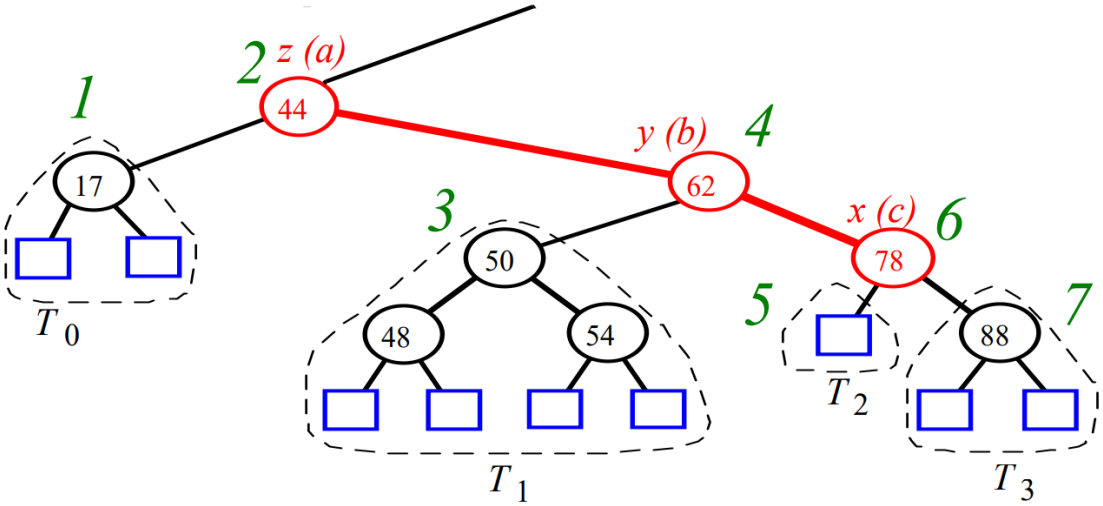
Input: Ein Knoten x eines binären Suchbaumes T, welcher einen Eltern-Knoten y und einen Grosseltern-Knoten z besitzt.

Output: restrukturierter Baum T, mit rotierten Knoten (entweder Single oder Double) x, y und z.

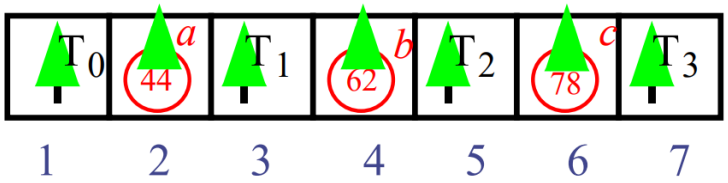
Vorteile gegenüber den vier Rotationen von vorher: keine Fallunterscheidung, eleganter, gleiche Zeitkomplexität

Nachteile gegenüber den vier Rotationen von vorher: komplexerer Programmcode

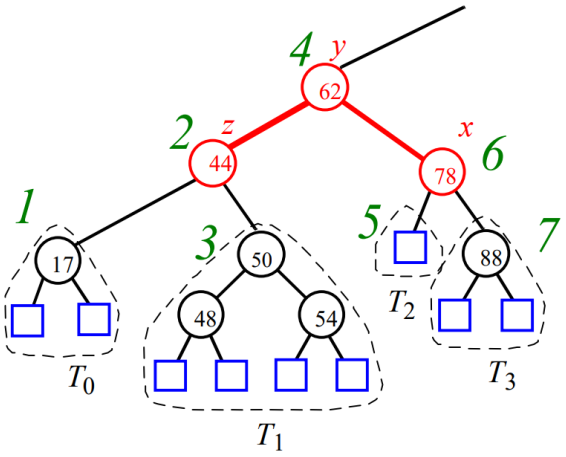
1. Jeden Baum, den man ausbalancieren muss, kann man in 7 Teile aufteilen: x, y, z und die 4 Bäume, mit Wurzeln direkt unterhalb x, y und z (T_0-T_3). Danach müssen die 7 Teile gemäss In-Order nummeriert und x,y,z gemäss In-Order a,b, c zugewiesen werden. Z bekommt der Knoten, dessen Balance <-1 bzw. >1 ist.



2. Danach werden die sieben Teil in ein Array eingefügt.



3. Dann wird der Baum wieder aufgebaut: 4 als Root, 2 als linkes Kind, 6 als rechtes Kind, 1/3/5/7 als Kinder von 2 und 6



3.7.5 Laufzeitanalyse

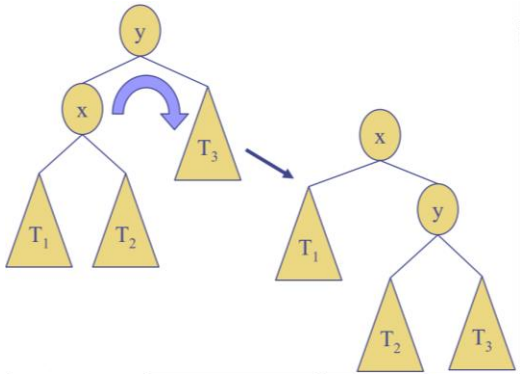
- Einzelne Restrukturierung unter Benutzung eines verlinkten binären Baumes: O(1)
- Find: O(log(n))
- Insert: O(log(n))
- Delete: O(log(n))

3.8 SPLAY BAUM

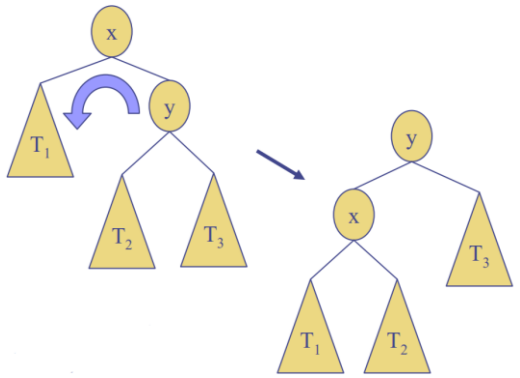
Splay Bäume sind binäre Suchbäume. In einem normalen binären Suchbaum ist es möglich, dass gleiche Schlüssel weit auseinander gespeichert sind. Beim Splay Baum ist das Ziel, dass der zuletzt verwendete Knoten die Wurzel ist. Beim Einfügen, Löschen und Suchen wird der Knoten gemäss Tabelle zu Root bewegt mit der Operation **splay**. Der tiefste zugegriffene Knoten wird somit Root.

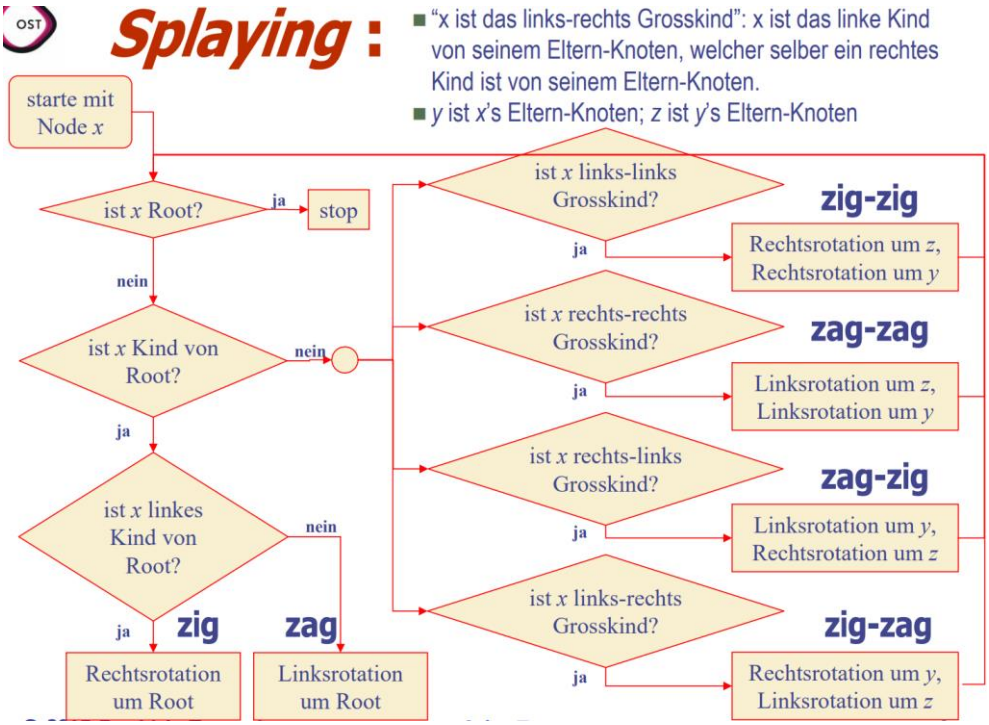
Methode	***Welcher Knoten wird splayed nach der Operation?***
find(k)	Schlüssel gefunden: Treffer benutzen Schlüssel nicht gefunden: Eltern-Knoten des Blattes am Ende
insert(k,v)	Der neue Knoten, bei welchem der Entry eingefügt/ersetzt wurde
remove(k)	Eltern-Knoten des internen Knotens, der gelöscht wurde

- Rechts-Rotation um y (Struktur oberhalb von y wird nicht modifiziert):



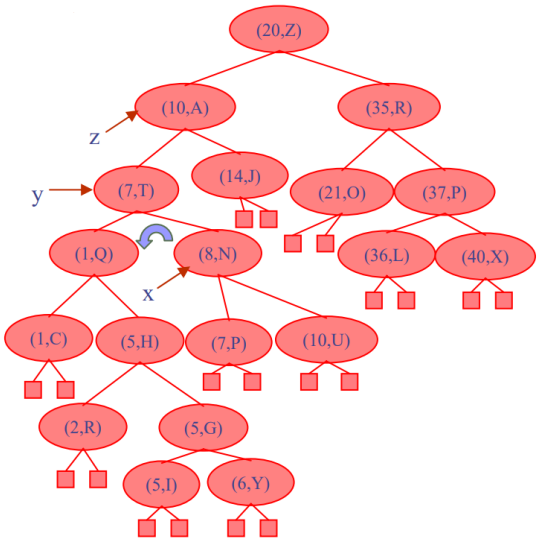
- Links-Rotation um x (Struktur oberhalb von y wird nicht modifiziert):



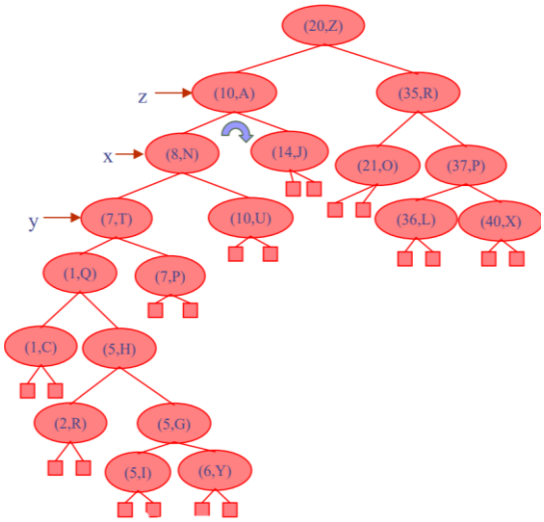


3.8.1 Beispiel

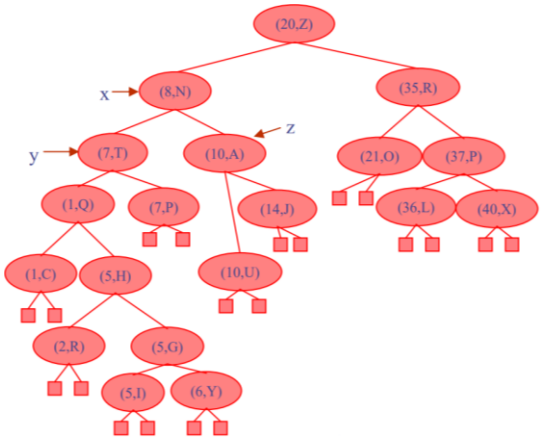
1. Ausgangslage



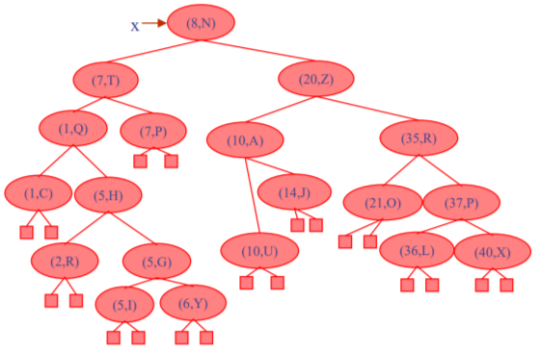
2. Erste Rotation: Linksrotation um y (x ist nicht Root, x ist nicht Kind von Root, x ist rechts-links Grosskind, daher zuerst Linksrotation um y, dann Rechtsrotation um z)



3. Zweite Rotation: Rechtsrotation um z

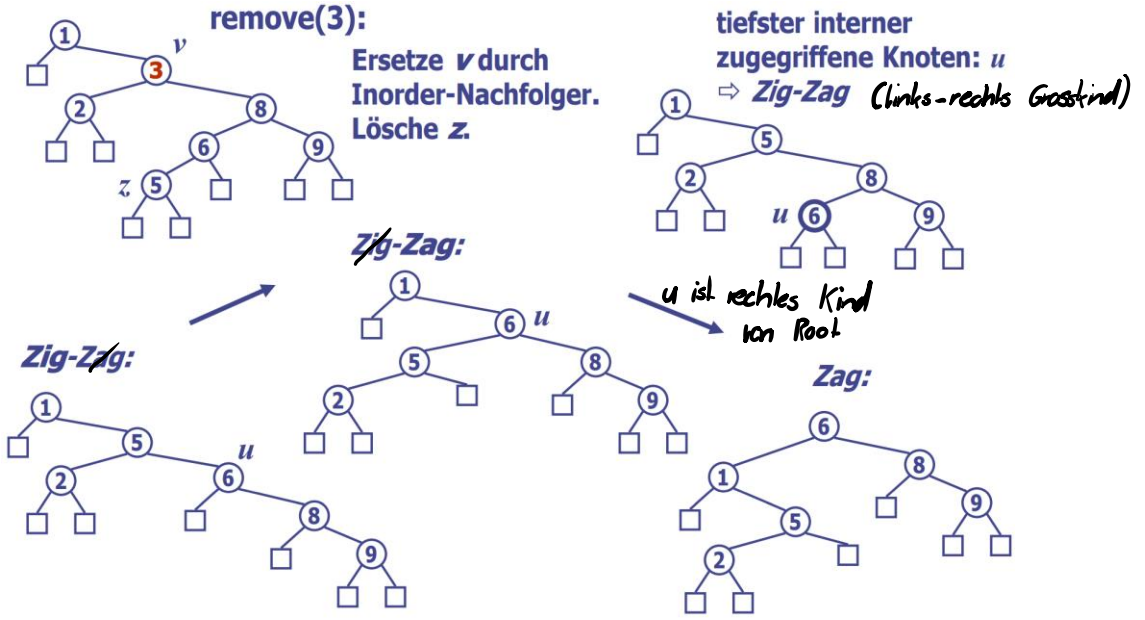


4. Dritte Rotation: Rechtsrotation um Root (x ist linkes Kind von Root)



3.8.2 Löschen

Das Löschen beginnt wie beim binären Suchbaum. Siehe 3.5.3 Knoten hat kein Blatt-Kind für dieses Beispiel. Der tiefste zugegriffene Knoten ist der Elternknoten des gelöschten Knotens, er wird somit zu Root rotiert.



3.8.3 Laufzeitanalyse

- Splaying Cost: O(h)
  h ist die Höhe des Baumes bis zum gesuchten Knoten
  - Durchschnittlich O(log(n))
    Für oft besuchte Knoten sogar wesentlich schneller, weil diese immer näher an Root rücken.
  - Worst Case: O(n)
    Wenn die Höhe des Baumes n ist. Dann braucht es O(n) = O(h) Rotationen.
- Einzelne Rotation unter Benutzung eines verlinkten binären Baumes: O(1)

4 SORTIERALGORITHMEN

Algorithmus	***Zeitverhalten***	***Bemerkungen***
Selection Sort	$O(n^2)$	• langsam • in-place • für kleine Datasets (< 1K)
Insertion Sort	$O(n^2)$	• langsam • in-place • für kleine Datasets (< 1K)
Quick Sort	O(n * log(n)) *erwartet*	• schnellster • in-place • für grosse Datasets (1K – 1M)
Heap Sort	O(n * log(n))	• schnell • in-place • für grosse Datasets (1K – 1M)
Merge Sort	O(n * log(n))	• schnell • sequenzieller Datenzugriff • für riesige Datasets (> 1M)

4.1 MERGE SORT

4.1.1 Divide-and-Conquer

Auf Deutsch Teile-und-Herrsche. Generelles Paradigma beim Design von Algorithmen:

- **Divide (Teilen):** Input-Daten S in zwei getrennte Teilmengen S_1 und S_2 aufteilen
- **Recur (Wiederholen):** Die Teilprobleme S_1 und S_2 rekursiv lösen
- **Conquer (Herrschen):** mischen der Lösungen von S_1 und S_2 in die Lösung von S

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

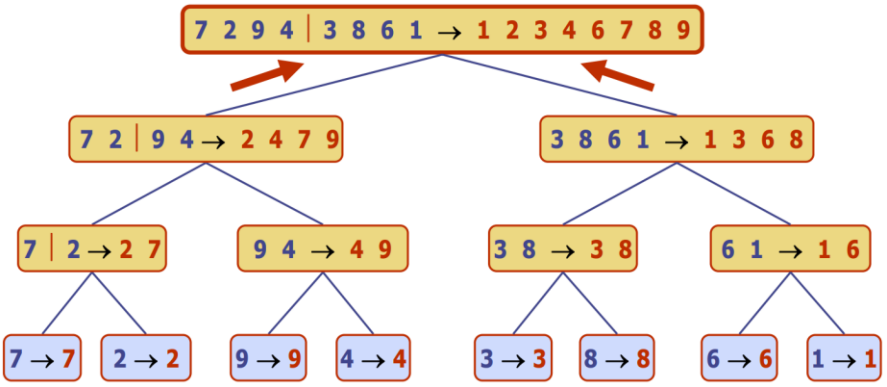
4.1.2 Merge Sort

Merge Sort verwendet das Divide-and-Conquer Paradigma. Zuerst wird die Sequenz in schlussendlich Teilsequenzen mit der Grösse 1 aufgeteilt. Danach werden die Teilsequenzen wieder zusammengefügt. Das schöne ist, dass wir jeweils nur das vorderste Element der beiden Teilsequenzen beim Mergen betrachten müssen, weil die Teilsequenzen bereits sortiert sind. Die Ausführung kann als binärer Baum dargestellt werden.

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

```
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
```



4.1.3 Laufzeitanalyse

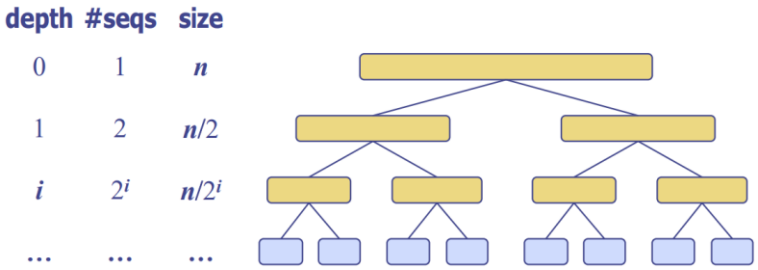
Aufteilen:

- O(1) fürs Aufteilen selber
- Passiert log(n) mal (Höhe des Merge-Sort-Baumes)

Zusammenfügen und sortieren:

- Jedes Element muss von der Teilsequenz in die neue Rückgabesequenz eingefügt werden, also O(n)
- Das erste Elment ansehen und vergleichen in O(1)
- Passiert log(n) mal

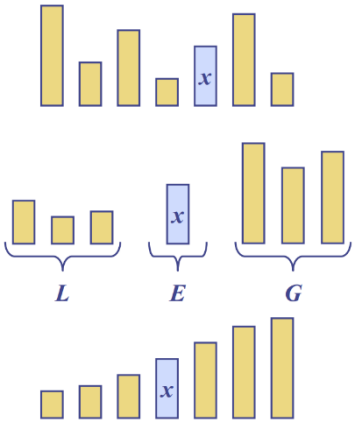
log(n) * O(1) + log(n) * O(n) * O(1) =
log(n) * (O(1) + O(n) + O(1)) =
O(n * log(n))



4.2 Quick Sort

Quick Sort basiert ebenfalls auf dem 4.1.1 Divide-and-Conquer Paradigma:

- **Divide (Teilen):** Auswahl eines zufälligen Elements x (genannt Pivot) und Aufteilung von S in:
  - **L** = Elemente kleiner als x
  - **E** = Elemente gleich x
  - **G** = Elemente grösser als x
- **Recur (Wiederholen):** L und G sortieren
- **Conquer (Herrschen):** L, E und G vereinen



4.2.1 Partitionierung (Divide)

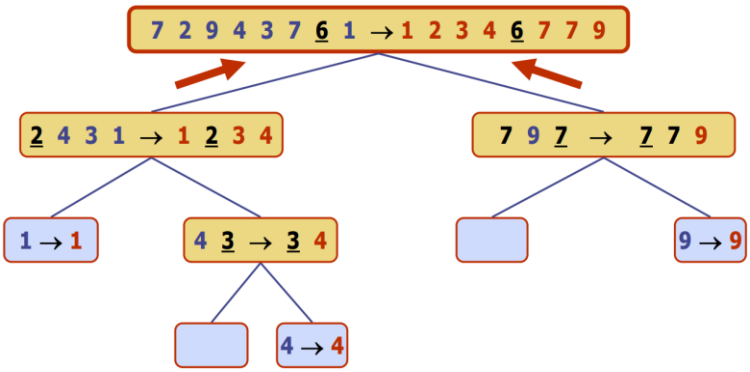
Ein Element y wird aus S entfernt und in L, E oder G eingefügt, abhängig vom Vergleich mit dem Pivot (x).

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

4.2.2 Ausführung Beispiel

Die Ausführung kann als binärer Baum dargestellt werden, jeder Knoten repräsentiert einen rekursiven Aufruf. Die Wurzel entspricht dem initialen Aufruf und die Blätter sind Aufrufe von Teilsequenzen der Grösse 0 oder 1.

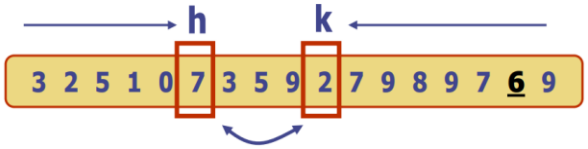
- Links vom Pfeil ist die unsortierte Sequenz vor der Ausführung
- Pivot ist schwarz und unterstrichen
- Rechts ist die sortierte Sequenz nach der Ausführung
- Das linke Kind entspricht jeweils L
- Das rechte Kind entspricht jeweils G



4.2.3 In-Place Implementierung

Bei der Partitionierung startet man links von der Sequenz mit der Variable h und rechts mit k. Die beiden Indexe wandern Richtung andere Seite, solange h und k sich nicht gekreuzt haben.

- H stoppt, wenn ein Element grösser als das Pivot kommt
- K stoppt, bis ein Element kleiner als das Pivot kommt
- Wenn sich h und k noch nicht gekreuzt haben, werden die beiden Elemente vertauscht, dann geht's weiter.



Sobald h und k sich gekreuzt haben, ist die Partitionierung abgeschlossen. Man hat nun die Teilsequenzen L (0 bis h-1 bzw. l) und G (k+1 bis Ende Array bzw. r).

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

4.2.4 Laufzeitanalyse

4.2.4.1 Worst Case

Der Worst-Case des Quick-Sort tritt dann auf, wenn das Pivot das Minimum- oder Maximum-Element ist. Entweder L oder G hat dann die Länge n-1, das andere 0 (wenn alle Elemente ungleich).

Dann ist die Laufzeit proportional zu der Summe: $n + (n-1) + \ldots + 2 + 1 : \quad \sum_{i=0}^{n} i = \frac{n^2+n}{2}$

Die Laufzeit ist also $O(n^2)$ im Worst Case.

4.2.4.2 *Erwartete Laufzeit*

Rekursiver Aufruf von Quick-Sort auf einer Sequenz der Länge s:

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

Ein Good call hat eine Wahrscheinlichkeit von 50%, die Hälfte der möglichen Pivots bewirken Good calls.

Bei einem Knoten in der Tiefe i wird erwartet, dass die Länge der Input-Sequenz des aktuellen Calls höchstens $(3/4)^{i/2} * n$ ist. Somit ist für einen Knoten der Tiefe $2 * \log_{4/3}(n)$ die erwartete Input-Länge 1. Und somit ist die erwartete Höhe des Quick-Sort-Baumes log(n).

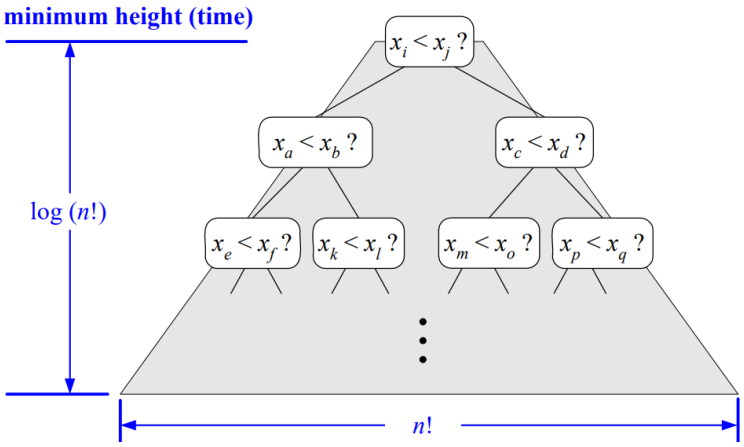
4.2.4.3 *Gesamte Laufzeit*

- Partitionierung
  - Entfernen eines Elements aus der Sequenz und Einfügen am Ende von L/E/G: O(1)
  - Das Ganze muss O(n) mal gemacht werden
- Anzahl erwartete Rekursionen: O(log(n))

Gesamt: O(n*log(n))

4.3 Lower Bound

Jeder vergleichsbasierte Sortier-Algorithmus hat eine minimale Laufzeit von O(log(n!)).



$$\log(n!) \geq \log\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) = \frac{n}{2}\log\left(\frac{n}{2}\right)$$

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

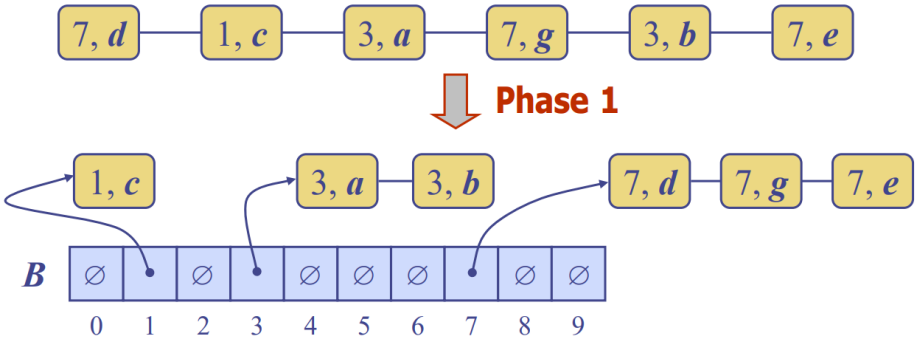
Stirling-Annäherung:

Somit ist die untere Grenze: Ω(n*log(n))

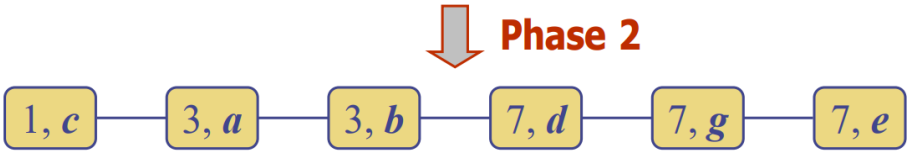
4.4 Bucket Sort

Wir haben eine Sequenz S mit n Key-Element-Entries mit Keys im Bereich von 0 bis N-1. Die Keys werden als Index im Hilfs-Array B verwendet und können somit nicht beliebige Objekte sein. Es ist kein externen Comparator nötig. Der Algorithmus ist stabil, das heisst die relative Ordnung von zwei Entries mit dem selben Key wird durch den Algorithmus nicht verändert.

In der Phase 1 wird jeder Entry (k, o) in seinen Bucket B[k] verschoben.



In der Phase 2 wird das Hilfs-Array B von links nach rechts (die Sequenzen innerhalb der Buckets in B ebenfalls von links nach rechts) zurück in S geschrieben.




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

4.4.1 Erweiterungen

- Wenn im Wertebereich [a, b] a nicht gleich 0 ist, macht es Sinn, den Entry (k, o) in den Bucket B[k-a] zu stecken.
- Wenn Strings als Keys verwendet werden, kann ein Set D von konstanter Grösse mit möglichen Strings verwendet werden (beispielsweise 26 Kantone). D wird dann sortiert und jedem String ein Index r(k) zugeordnet. Der Bucket des Strings ist dann B[r(k)].

4.4.2 Lexikographische Ordnung

Ein d-Tupel ist eine Sequenz von d Keys ($k_1, k_2, k_3, \ldots, k_d$), wobei Key k_i als die i-te Dimension des Tupels bezeichnet wird. Beispielsweise ist eine dreidimensionale Koordinate ein 3-Tupel.

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

$$(x_1, x_2, \ldots, x_d) < (y_1, y_2, \ldots, y_d)$$

$$\Leftrightarrow$$

$$x_1 < y_1 \vee x_1 = y_1 \wedge (x_2, \ldots, x_d) < (y_2, \ldots, y_d)$$

Das heisst, die Tupel werden der Dimension nach verglichen (zuerst die erste Dimension, dann die Zweite, usw.).

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)

C_i ist der Comparator, der zwei Tupel nach ihrer i-ten Dimension vergleicht. StableSort ist ein stabiler Sortier-Algorithmus, welcher C_i verwendet. Es wird zuerst nach Dimension d sortiert und dann die restlichen Dimensionen von rechts nach links bis 1.

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)

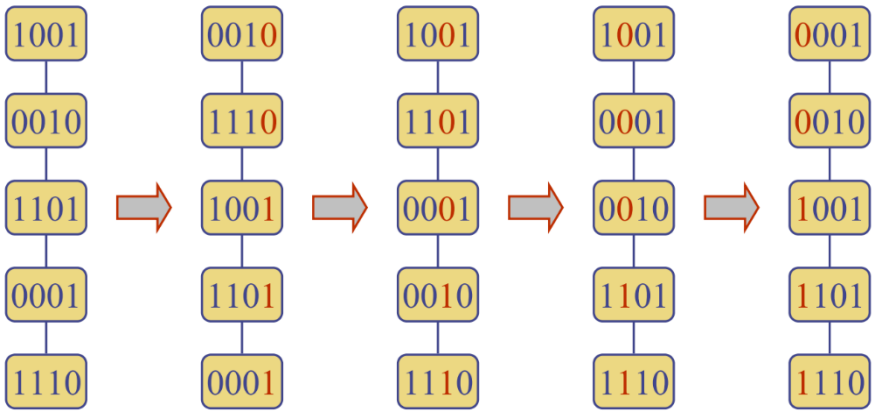
4.4.3 Radix Sort

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

Radix Sort ist eine Spezialisierung der lexikographischen Sortierung, welcher Bucket Sort als stabilen Sortieralgorithmus verwendet. Es ist anwendbar für Tupel mit Integer-Keys, die in allen Dimensionen im Bereich 0 bis N-1 liegen.

Es ist gut geeignet zur Sortierung von binären Tupels. Die Keys sind im Bereich von 0 bis 1, also ist N=2.

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



4.4.4 Laufzeitanalyse

4.4.4.1 Bucket Sort

Phase 1: O(n)
Phase 2: O(N+n)
Gesamt: O(N+n)

4.4.4.2 Lexikographische Sortierung

Gesamt: O(d * T(n))
T(n) ist die Laufzeit von stableSort

4.4.4.3 Radix Sort

Gesamt: O(d * (N+n))
Bei binären Zahlen: O(b*n)
b ist die Anzahl Bits pro Tupel

5 Pattern Matching

Ein String ist eine Sequenz von Characters. Ein Alphabet Σ ist ein Set von möglichen Zeichen für eine Familie von Strings. Beispiele für Alphabete: ASCII, Unicode, {0, 1}, {A, C, G, T} (DNA-Grundbausteine)

P ist ein String der Länge m. Ein Substring P[i..j] ist eine Subsequenz von P bestehend aus den Zeichen mit Index zwischen und inklusive i und j.

Präfix: P[0..i]
Suffix: P[i..m-1]

Beim Pattern-Matching Problem geht es darum, einen Substring im Text T zu finden, der mit dem Pattern P übereinstimmt.

5.1 Brute Force

Bei der Brute Force Methode vergleicht der Algorithmus das Pattern P mit dem Text T für jede mögliche Position von P relativ zu T, bis eine Übereinstimmung gefunden wurde oder alle möglichen Platzierungen ausprobiert wurden.

```
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 !}
```

i ist der aktuelle Index innerhalb von T und j der aktuelle Index innerhalb von P. Wenn die Zeichen an i+j und j übereinstimmen, wird j um 1 erhöht. Wenn j gleich der Länge von P ist, haben wir einen Match und i wird returned, welches der Startindex des Matches ist.

Sobald aber ein Mismatch vorkommt, wird j wieder auf 0 gesetzt. Wenn wir am Ende von T angekommen sind und es keinen Match gab, wird -1 returned.

Laufzeit: O(n*m)

5.2 Boyer-Moore

5.2.1 Last Occurence Funktion

Boyer-Moore analysiert zuerst das Pattern und das Alphabet Σ, um die Last Occurence Funktion L aufzubauen. Das Ziel ist, den höchsten Index jedes Characters des Alphabets in P zu speichern, bzw. -1 wenn der Character nicht in P vorkommt.

- $\Sigma = \{a, b, c, d\}$
- $P = abacab$
  $Pos: 012345$

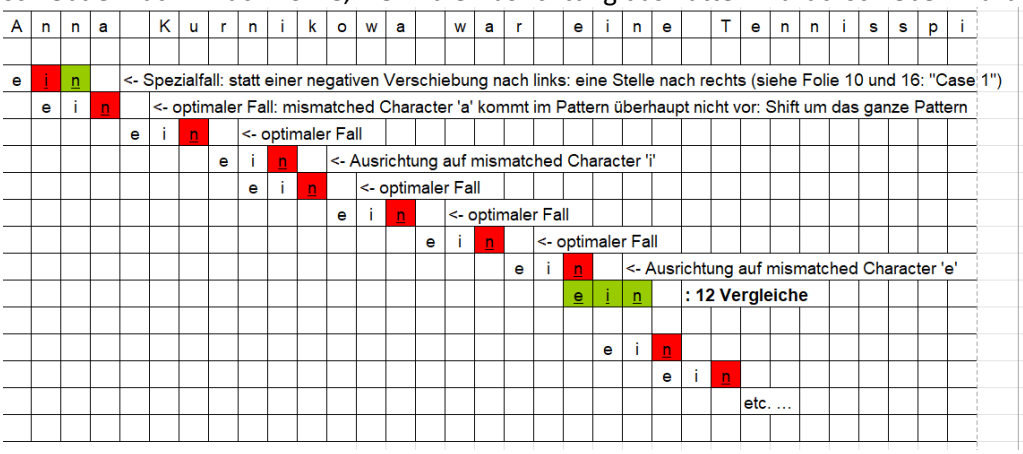
c	a	b	c	d
$L(c)$	4	5	3	−1

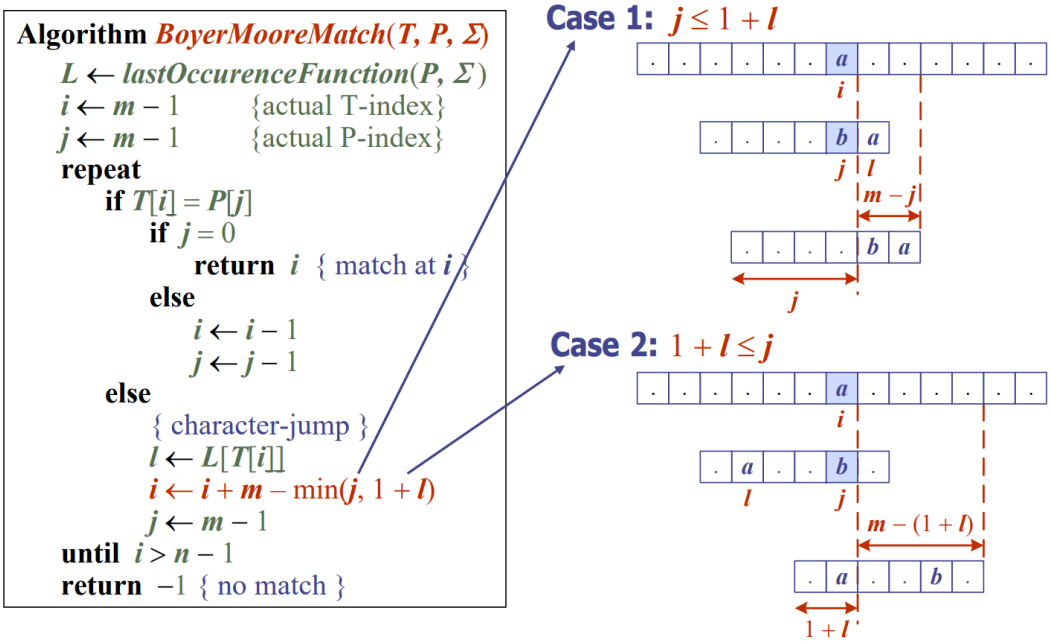
5.2.2 Algorithmus

Der Algorithmus basiert auf zwei Heuristiken:

- Looking-Glass Heuristik: P wird mit einer Subsequenz von T verglichen, dabei wird am Ende des Patterns gestartet
- 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

Man vergleicht jeweils den letzten Character von P mit dem zugehörigen Character in T. Ist es ein Match, geht man einen Character zurück, usw. Ist es jedoch ein Mismatch, richtet man das Pattern auf die Last Occurence aus oder schiebt einfach 1 nach vorne, wenn die Ausrichtung das Pattern zurückschieben würde.





5.2.3 Laufzeitanalyse

Laufzeit: O(n*m + s)

n = Länge Text, m = Länge Pattern, s = Anzahl Zeichen im Alphabet

5.3 KNUTH-MORRIS-PRATT

5.3.1 Preprocessing (Fehl-Funktion)

In der Vorlaufsphase sucht der Algorithmus Übereinstimmungen vom Präfix des Musters im Muster selbst.

Berechnung Fehl-Funktion:

1. Bestimmen aller Ränder aller Pattern-Substrings
   (Rand: längstes 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]

a	b	a	a	b	a	a	b	a

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

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

Algorithm *failureFunction(P)*

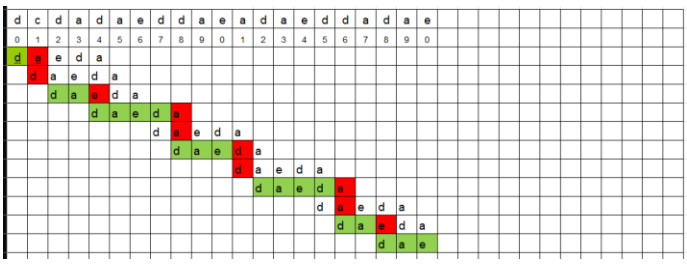
```
F[0] ← 0
i ← 1
j ← 0
while i < m
    if P[i] = P[j]
        {we have matched j + 1 chars}
        F[i] ← j + 1
        i ← i + 1
        j ← j + 1
    else if j > 0 then
        {use failure function to shift P}
        j ← F[j − 1]
    else
        F[i] ← 0 { no match }
        i ← i + 1
```

5.3.2 Algorithmus

Algorithm *KMPMatch(T, P)*

```
F ← failureFunction(P)
i ← 0
j ← 0
while i < n
    if T[i] = P[j]
        if j = m − 1
            return i - m + 1 { match }
        else
            i ← i + 1
            j ← j + 1
    else
        if j > 0
            j ← F[j − 1]
        else
            i ← i + 1
return −1 { no match }
```

Das Muster wird von links nach rechts mit dem Text verglichen, jedoch schiebt der Algorithmus das Muster intelligenter als Brute Force. Beim Preprocessing haben wir berechnet, um wie viel das Muster bei einem Mismatch geschoben werden kann.



Übereingestimmt hat "d", der Rand davon ist 0, der Index j von "a" ist 1, 1-0=+1, j kriegt den Wert vom Rand j=0
Erstes Zeichen hat nicht übereingestimmt, direkt +1
Übereingestimmt hat "da", der Rand davon ist 0, der Index j von "e" ist 2, 2-0=+2, j kriegt den Wert vom Rand j=0
Übereingestimmt hat "daed", der Rand davon ist 1, der Index j von "a" ist 4, 4-1=+3, j kriegt den Wert vom Rand j=1
Übereingestimmt hat "d", der Rand davon ist 0, der Index j von "a" ist 1, 1-0=+1, j kriegt den Wert vom Rand j=0
Übereingestimmt hat "dae", der Rand davon ist 0, der Index j von "d" ist 3, 3-0=+3, j kriegt den Wert vom Rand j=0
Erstes Zeichen hat nicht übereingestimmt, direkt +1
Übereingestimmt hat "daed", der Rand davon ist 1, der Index j von "a" ist 4, 4-1=+3, j kriegt den Wert vom Rand j=1
Übereingestimmt hat "d", der Rand davon ist 0, der Index j von "a" ist 1, 1-0=+1, j kriegt den Wert vom Rand j=0
Übereingestimmt hat "da", der Rand davon ist 0, der Index j von "e" ist 2, 2-0=+2, j kriegt den Wert vom Rand j=0
no match, 29 Vergleiche

5.3.3 Laufzeitanalyse

- Preprocessing: Max. O(2m)
  m = Länge des Patterns
- Algorithmus: Max. O(2n)
  n = Länge des Texts

Gesamt: O(2m + 2n) = O(m+n)

Nach Vorverarbeitung des Musters erzielt der KMP Algorithmus eine Geschwindigkeit, die proportional zur Text-Grösse ist.

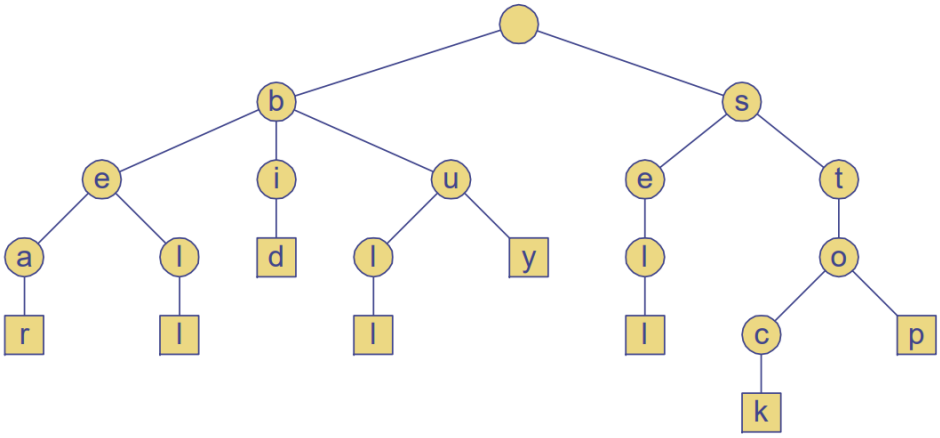
5.4 Tries

Wenn ein Text gross, unveränderlich und oft durchsucht wird, kann man anstelle des Musters (wie beim KMP-Algorithmus) auch den Text vorverarbeiten. 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.

Ein Standard-Trie ist ein geordneter Baum mit diesen Eigenschaften:

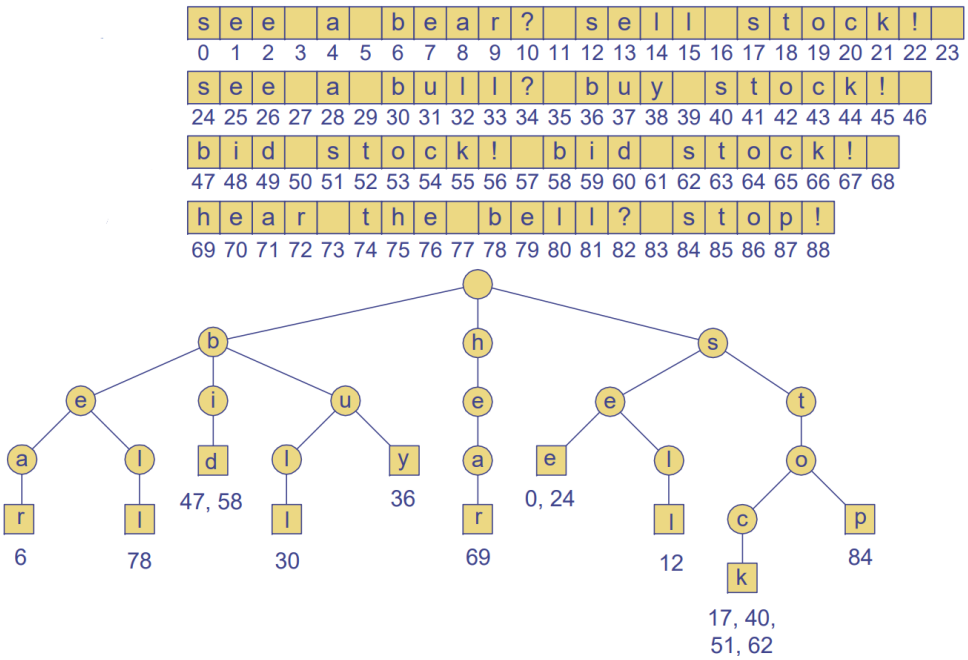
- Alle Knoten ausser der Wurzel-Knoten haben genau ein Zeichen.
- Die Pfade von den externen Knoten zur Wurzel beinhalten die Strings aus der Menge von Strings S.

S = { bear, bell, bid, bull, buy, sell, stock, stop }

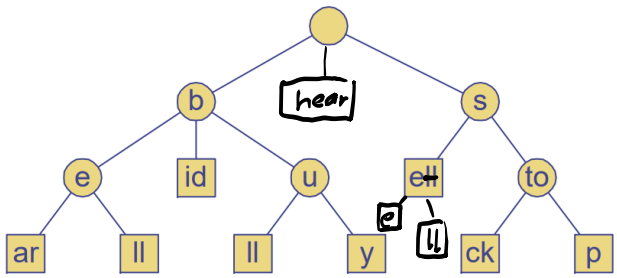


Speicherbedarf: O(n)
Suchen, Einfügen und Löschen: O(d*m)
n = totale Länge der Strings in S, m = Länge des String-Parameters der Operation, d = Grösse des Alphabets

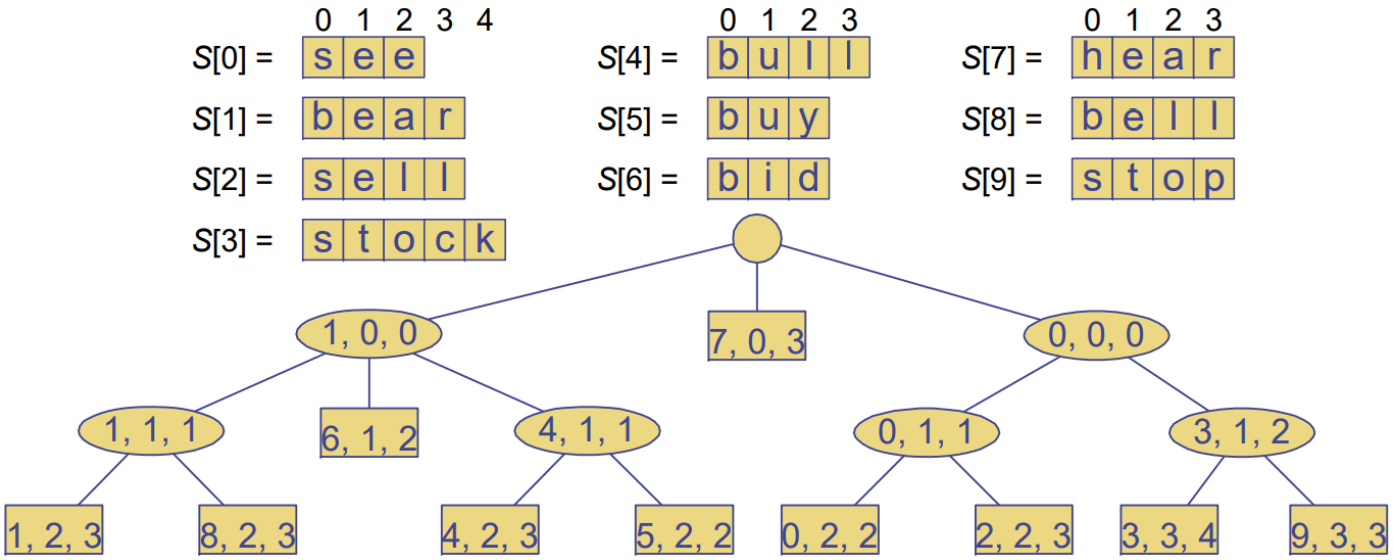
5.4.1 Suche in einem Trie



Komprimierte Version:

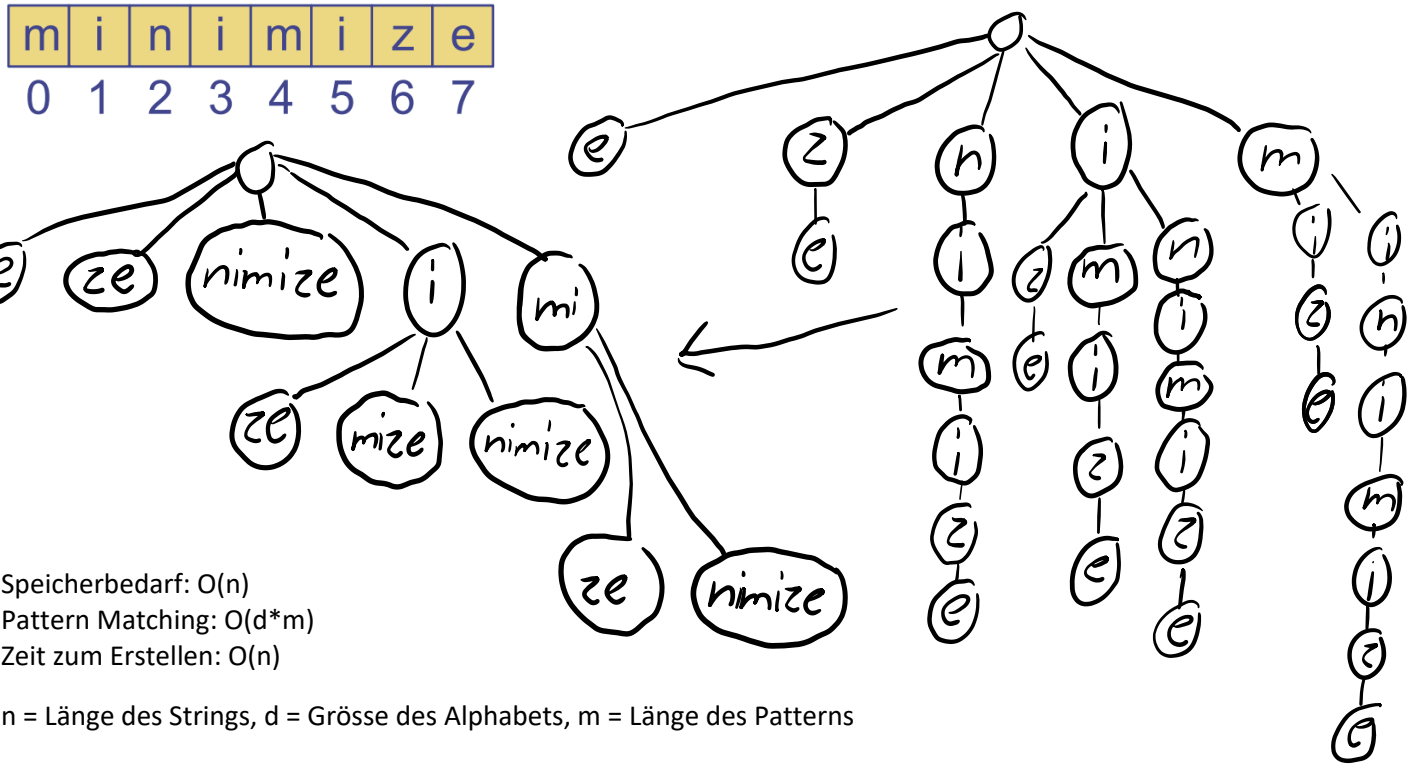


5.4.2 Kompakte Repräsentation



- Knoten speichert Indizes anstelle von Substrings
- Benötigt O(s) Speicher, wobei s die Anzahl Strings im Array ist
- Dient als Hilfs-Index Struktur
- Bedeutung Zahlen in den Knoten
    - Erste Zahl: Index in S, also Wort
    - Zweite Zahl: Index des Zeichens, wo der Substring beginnt innerhalb des Worts, das wir von der ersten Zahl wissen
    - Dritte Zahl: Index des Zeichens, wo der Substring aufhört

5.4.3 Suffix Trie



Speicherbedarf: O(n)
Pattern Matching: O(d*m)
Zeit zum Erstellen: O(n)

n = Länge des Strings, d = Grösse des Alphabets, m = Länge des Patterns

6 Dynamische Programmierung

Dynamische Programmierung ist anwendbar auf Probleme, welche anfänglich eine sehr grosse Laufzeit zu benötigen scheinen (möglicherweise sogar exponentiell). Voraussetzungen:

- **Einfache Subprobleme**: Die Subprobleme können durch wenige Variablen ausgedrückt werden.
- **Subproblem-Optimierung**: Das globale Optimum kann durch optimale Subprobleme ausgedrückt werden.
- **Subprobleme überlappen**: Die Subprobleme sind abhängig voneinander und der Algorithmus sollte somit bottom-up konstruiert werden.

6.1 Rucksack-Problem

Gegeben:

- n Gegenstände mit einem bestimmten Gewicht und Wert
- Ein Rucksack mit einer bestimmten Gewichts-Kapazität

Gesucht:

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

Das **Brute Force Verfahren**, um das Problem zu lösen, wäre alle möglichen Varianten zu probieren. Alle Varianten, die das maximale Gewicht nicht überschreiten, werden betrachtet und dann die beste Variante gewählt. Der Aufwand dafür ist jedoch riesig, nämlich $O(2^n)$.

Ein **Greedy Algorithmus** würde jeweils den Gegenstand mit dem grössten Gewicht, der noch reinpasst, in den Rucksack packen. Das gibt immer schnell eine Lösung, jedoch ist das möglicherweise nicht die optimale Lösung.

Bei der dynamischen Programmierung erstellen wir optimale Subprobleme, die einfacher zu lösen sind. Bei diesem Beispiel haben wir:

- Rucksack mit 7kg Kapazität
- Gegenstände:
  - 3kg mit 2$ Wert
  - 1kg mit 2$ Wert
  - 3kg mit 4$ Wert
  - 4kg mit 5$ Wert
  - 2kg mit 3$ Wert

Wir arbeiten Zeile für Zeile von links nach rechts ab.

1. Schauen, ob das Gewicht des Items der Reihe gleich oder kleiner als die Kapazität der Spalte ist.
   a. Wenn nein, Wert von Feld direkt obendran kopieren.
   b. Wenn ja, wählt man das grössere der beiden nachfolgenden Optionen aus:
      i. Wert des Felds direkt obendran (Item wird nicht hinzugefügt)
      ii. Wert des Items der Reihe +
      Wert des Felds obendran nach links verschoben um das Gewicht des Items der Reihe (Item wird hinzugefügt)

	0kg	***1kg***	***2kg***	***3kg***	***4kg***	***5kg***	***6kg***	***7kg***
Empty	0$	0$	0$	0$	0$	0$	0$	0$
3kg/2$	0$	0$	0$	2$	2$	2$	2$	2$
1kg/2$	0$	2$	2$	2$	4$	4$	4$	4$
3kg/4$	0$	2$	2$	4$	6$	6$	6$	8$
4kg/5$	0$	2$	2$	4$	6$	7$	7$	9$
2kg/3$	0$	2$	3$	5$	6$	7$	9$	10$

Die Lösung sind die Items, bei denen der Wert nicht von oben kopiert wurde: 2kg/3$, 4kg/5$, 1kg/2$

6.2 Longest Common Subsequence (LCS)

6.2.1 Subsequenzen

Eine Subsequenz eines Strings ist eine Sequenz, die durch Löschen einiger oder keiner Elemente aus dem ursprünglichen String ohne Änderung der Reihenfolge der verbleibenden Zeichen erhalten wird.

Zum Beispiel, wenn wir den String "ABC" haben, dann sind "A", "B", "C", "AB", "AC", "BC", und "ABC" Subsequenzen des Strings. Beachten Sie, dass auch der leere String und der ursprüngliche String selbst als Subsequenzen betrachtet werden.

Es ist wichtig zu beachten, dass eine Subsequenz nicht dasselbe ist wie ein Substring. Ein Substring ist eine zusammenhängende Folge von Zeichen innerhalb eines Strings. Im Gegensatz dazu muss eine Subsequenz nicht aus zusammenhängenden Zeichen bestehen. Zum Beispiel, in dem String "ABC", "AC" ist eine Subsequenz, aber kein Substring.

6.2.2 Algorithmus

Das Ziel von LCS ist es, die längste Subsequenz zu finden, die sowohl im String X als auch im String Y enthalten ist. Ein bekannter Anwendungsfall dafür ist Git.

Beispiel: ABCDEFG und XZACKDFWGH haben ACDFG als längste gemeinsame Subsequenz.

Die Brute Force Lösung wäre, alle Subsequenzen von X aufzulisten und zu testen, welche davon in Y vorhanden sind. Die längste Subsequenz, welche in Y enthalten ist, ist dann das Resultat. Die schlechte Laufzeit davon wäre $O(2^n)$.

Beispiel: X=abcde, Y=ace

Die Felder stellen die Subprobleme dar. Das Feld [a/a] beispielsweise stellt das Vergleichen von abcde und ace dar, wohingegen [d/c] nur das Vergleichen von de und ce darstellt.

1. Wir füllen die Zeile und Spalte bei Index -1 mit 0 auf
2. Wir beginnen oben links mit dem Ausfüllen, gehen die Zeile nach rechts durch und gehen dann eine Zeile runter, wo wir wieder links beginnen
3. Pro Feld gibt es zwei Optionen:
    - Die Characters stimmen überein: Wir addieren 1 zum Wert eins oben und eins links (diagonal↖) und notieren diesen Wert
    - Die Characters stimmen nicht überein: Wir nehmen den höheren Wert von entweder eins oben⬆ oder eins links⬅
4. Wenn die Tabelle fertig ausgefüllt ist, können wir die LCS auslesen. Wir beginnen dazu ganz unten rechts. Wenn die beiden Zeichen des Felds übereinstimmen, fügen wir es hinten in die LCS ein und wechseln wir in das Feld oben links (diagonal↖). Sonst folgen wir einfach der Zeile oder Spalte nach links oder oben, wo die Zahl gleichbleibt.

X/Y			***a***	***c***	***e***
	L	***-1***	***0***	***1***	***2***
	-1	0	0	0	0
a	***0***	0	1	1	1
b	***1***	0	1	1	1
c	***2***	0	1	2	2
d	***3***	0	1	2	2
e	***4***	0	1	2	3

LCS-Länge=3, LCS=ace

```
Algorithm LCS(X,Y):
Input: Strings X and Y with n and m elements, respectively
Output: For i = 0,...,n-1, j = 0,...,m-1, the length L[i, j] of a longest string
    that is a subsequence of both the string X[0..i] = x0x1x2...xi and the
    string Y[0.. j] = y0y1y2...yj
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 xi = yj 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
```

6.2.3 Laufzeitanalyse

- Äusserer For-Loop iteriert n mal
- Innerer For-Loop iteriert m mal

Gesamt: O(n*m)

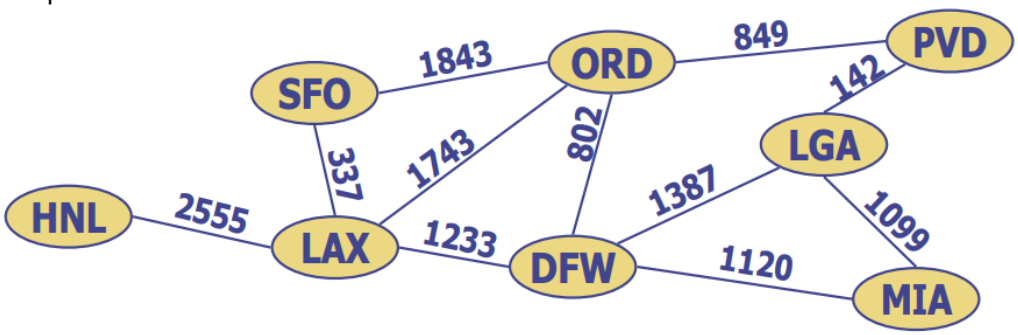
7 Graphs

7.1 Definitionen & Terminologien

Ein Graph ist ein Paar (V, E):

- V ist ein Set von Vertizes (Knoten)
- E ist eine Collection von Vertizes-Paaren (Kanten oder englisch Edges)
- V und E sind Positionen, die Elemente speichern

Beispiel:



Kanten können gerichtet oder ungerichtet sein:

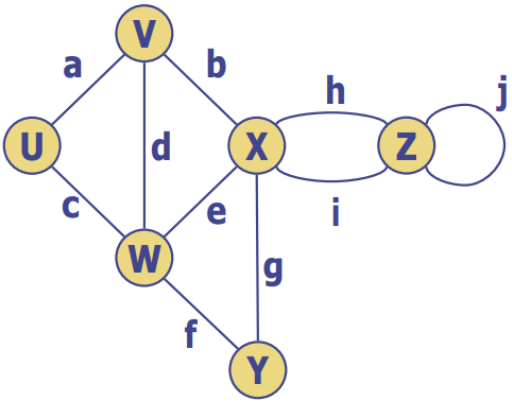
- Gerichtet: die beiden Vertizes sind geordnet (u, v)
  der erste Vertex u entspricht dem Ursprung und der zweite Vertex v dem Ziel

- Ungerichtet: die beiden Vertizes sind ungeordnet (u, v)

Auch Graphen können gerichtet oder ungerichtet sein:

- Gerichtet: alle Kanten sind gerichtet
- Ungerichtet: alle Kanten sind ungerichtet

- Kanten sind inzident (enden) an einem Vertex
  a, d, b sind inzident in V
- Adjazente (benachbarte) Vertizes
  V und X sind adjazent
- Der Grad (Degree) eines Vertex ist die Anzahl inzidenter Kanten
  V hat Grad 3
- h und i sind parallele Kanten
- j ist eine Schleife



Ein Pfad hat folgende Eigenschaften:

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

Ein einfacher Pfad ist ein Pfad, bei dem alle Vertizes und Kanten unterschiedlich sind.

Ein Zyklus in der Graphentheorie ist ein Kantenzug mit unterschiedlichen Kanten in einem Graphen, bei dem Start- und Endknoten gleich sind. Ein zyklischer Graph ist ein Graph mit mindestens einem Zyklus. Bei einem einfachen Zyklus sind alle Vertizes und Kanten unterschiedlich.

7.2 EIGENSCHAFTEN

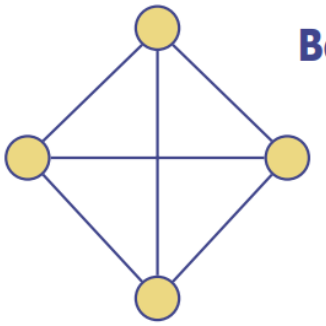
Die Summe der Degrees aller Vertizes ist 2m, weil jede Kante zweimal gezählt wird.

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

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

In einem einfachen (das heisst ohne Schleifen und ohne Mehrfach-Kanten (parallele Kanten)), ungerichteten Graphen gilt folgendes, weil jeder Vertex höchstens einen Grad von n-1 besitzt:

$$m \leq n\,(n-1)/2$$



Beispiel

- $n = 4$
- $m = 6$
- $\deg(v) = 3$

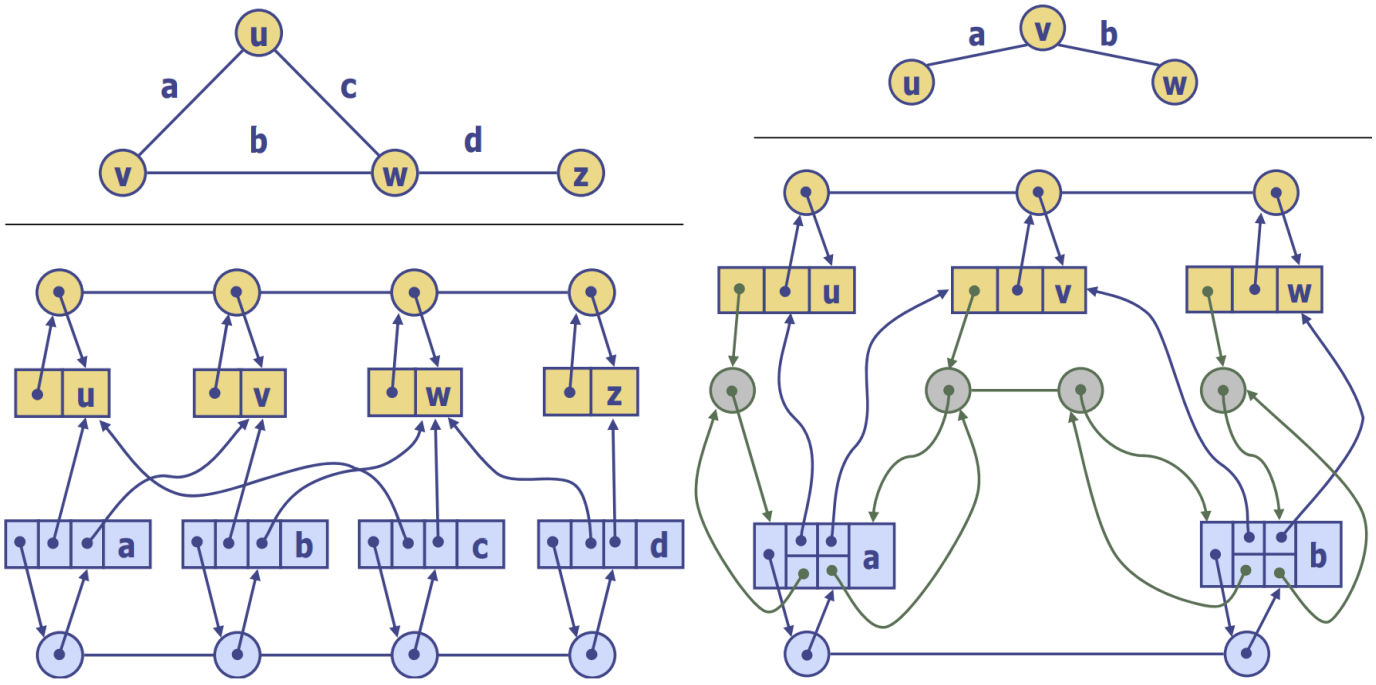
7.3 Methoden

- **Zugriffs-Methoden**
  - endVertices(e): an array of the two endvertices of e
  - opposite(v, e): the vertex opposite of v on e
  - areAdjacent(v, w): true iff(*) v and w are adjacent
  - replace(v, x): replace element at vertex v with x
  - replace(e, x): replace element at edge e with x

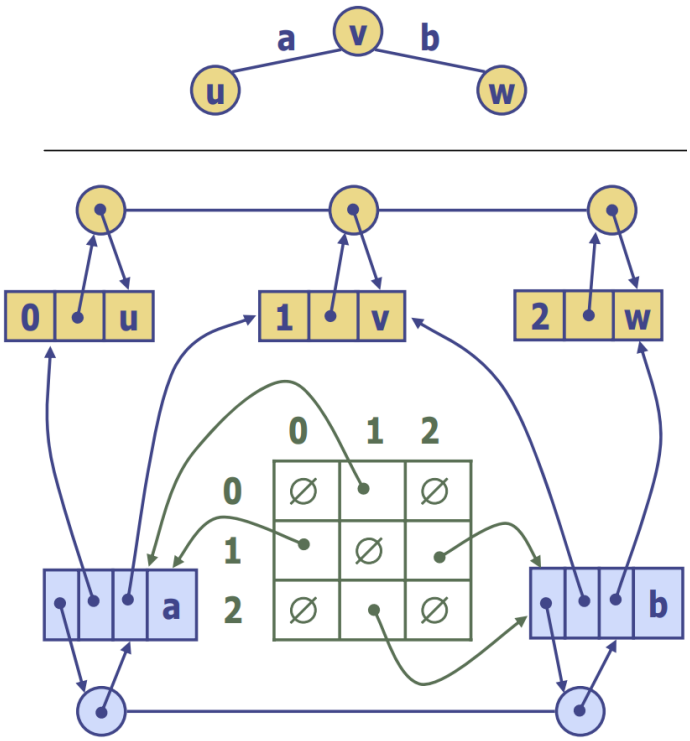
  (*) "iff": "if and only if"

- **Update-Methoden**
  - insertVertex(o): insert a vertex storing element o; return the new vertex
  - insertEdge(v, w, o): insert an edge (v,w) storing element o; return the new edge
  - removeVertex(v): remove vertex v (and its incident edges)
  - removeEdge(e): remove edge e
- **Iterator-Methoden**
  - incidentEdges(v): all edges incident to v
  - vertices(): all vertices in the graph
  - edges(): all edges in the graph

7.4 Kanten-Listen Struktur / Adjazenz-Listen Struktur



7.5 Adjazenz-Matrix Struktur



7.6 Performance

▪ n Vertizes, m Kanten ▪ keine parallelen Kanten ▪ keine Schleifen	Kanten Liste	Adjazenz Liste	Adjazenz Matrix
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	$\deg(v)$	n
areAdjacent (v, w)	m	$\min(\deg(v), \deg(w))$	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	$\deg(v)$	n^2
removeEdge(e)	1	1	1

7.7 Subgraphen

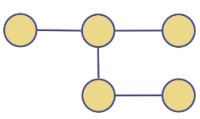
Ein Subgraph eines Graphen ist ein Graph, so dass gilt:

- die **Kanten** vom Subgraph sind eine Teilmenge der Kanten des Graphen
- die **Vertizes** vom Subgraph sind eine Teilmenge der Vertizes des Graphen

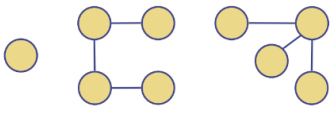
Ein **spanning (aufspannender) Subgraph** eines Graphen ist ein Subgraph, der alle Vertizes des Graphen enthält.

7.8 Connectivity

Verbunden (connected): es existiert ein Pfad zwischen jedem Paar von Vertizes (muss nicht direkt sein)


Tree

Eine verbundene Komponente eines Graphen ist ein verbundener Subgraph des Graphen


Forest

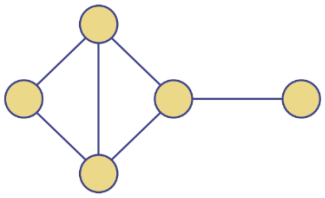
7.9 Bäume und Wälder

Ein **(freier) Baum** ist ein **ungerichteter** Graph T, so dass T **verbunden** ist und **keine Zyklen** aufweist.

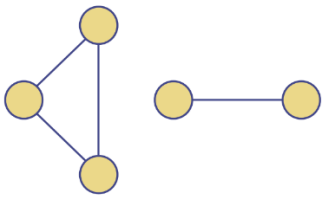
Ein **Wurzelbaum (auch gewurzelter Baum)** ist in der Graphentheorie ein Baum, der einen ausgezeichneten Knoten, die Wurzel, enthält. Die Definition unterscheidet sich von jener des (freien) Baumes. Von dieser Wurzel aus sind sämtliche anderen Knoten erreichbar (Out-Tree) oder sie kann ihrerseits von jedem anderen Knoten aus erreicht werden (In-Tree).

Ein **aufspannender Baum (Spanning Tree)** eines verbundenen Graphen ist ein aufspannender Subgraph, welcher auch ein Baum ist, er muss also alle Vertizes des Graphen enthalten, verbunden sein und keine Zyklen haben. Ein aufspannender Baum ist **nicht eindeutig**, ausser der Graph, von dem ausgegangen wird, ist ein Baum.

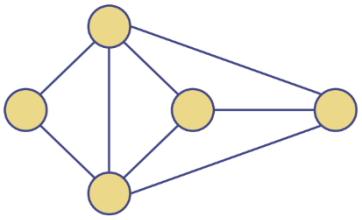
Ein **Wald** ist ein ungerichteter Graph ohne Zyklen. Die verbundenen **Komponenten eines Waldes sind Bäume**. Ein **aufspannender Wald** eines Graphen ist ein aufspannender Subgraph, welcher auch ein Wald ist



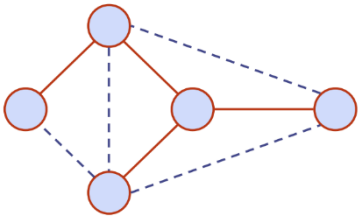
Connected Graph



Non connected graph with two connected components



Graph



Spanning Tree

7.10 Depth-First Search (DFS) / Tiefensuche

DFS ist eine allgemeine Technik zur Traversierung eines Graphen. Es entspricht in etwa der Euler-Tour bei binären Bäumen. Der Algorithmus benutzt einen Mechanismus, um "Labels" auf Kanten und Vertizes zu setzen. Eine DFS-Traversierung eines Graphen G:

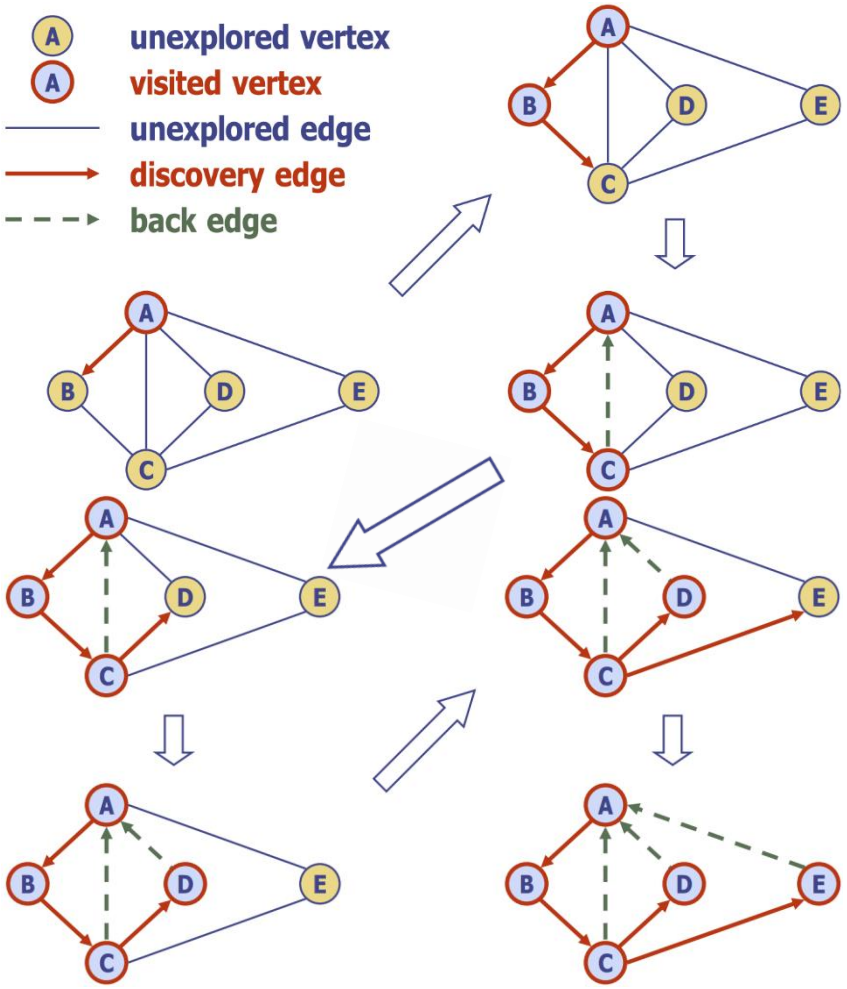
```
Algorithm DFS(G)
    Input graph G
    Output labeling of the edges of G
        as discovery edges and
        back edges
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      DFS(G,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

```
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 ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        DFS(G, w)
      else
        setLabel(e, BACK)
```

7.10.1 Eigenschaften

- $DFS\ (G, v)$ besucht alle Vertizes und Kanten in der verbundenen Komponente von G beginnend bei v
- die von $DFS\ (G, v)$ markierten, besuchten Kanten bilden einen aufspannenden Baum für die verbundene Komponente von G beginnend bei v

7.10.2 Beispiel



Man sieht, dass das Vorgehen der Strategie zum Lösen eines Labyrinths ähnelt. Wir arkieren jede besuchte Kreuzung, Ecke und Sackgasse (**Vertex**).Wir markieren jeden besuchten Korridor (**Kante**). Wir notieren den Rückweg zum Eingang (Start Vertex) (→ Rekursion auf dem Stack).

7.10.3 Laufzeitanalyse

- Graphen hat n Vertizes und m Kanten
- Setzen/Lesen eines Vertex-/Kanten-Labels benötigt $O(1)$ Zeit
- Jeder Vertex wird zweimal markiert: zuerst **UNEXPLORED**, dann **VISITED** → $O(2n)$
- Jede Kante wird zweimal markiert: zuerst **UNEXPLORED**, dann **DISCOVERY** oder **BACK** → $O(2m)$
- Die Methode incidentEdges() wird pro Vertex einmal aufgerufen. $O(deg(v))$ bei Adjazenzlisten-Strukturen, also $\sum_v \deg(v) = 2m$ → $O(2m)$
- $O(2n + 2m + 2m) = O(2n + 4m) = O(n + m)$

7.10.4 Erweiterung: Pfad zwischen zwei Vertices finden

Pfad zwischen u und z finden in G: DFS(G, u) aufrufen (DFS in G mit Start-Vertex u ausführen), mithilfe eines Stacks merken wir uns den Pfad zwischen u und dem aktuellen Vertex, sobald wir z gefunden haben, geben wir den Pfad mithilfe des Stacks aus

```
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()
```

7.10.5 Erweiterung: Einfache Zyklen finden

Mithilfe eines Stacks merken wir uns den Pfad zwischen dem Startvertex und dem aktuellen Vertex. Sobald wir auf eine Back-Kante (v, w) treffen, geben wir den Zyklus als Teil des Stacks aus: vom obersten Stackelement bis zum Vertex w.

```
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()
```

7.11 Breadth-First Search (BFS) / Breitensuche

BFS ist eine generelle Technik für die Traversierung eines Graphen. Der Algorithmus benutzt einen Mechanismus, um "Labels" auf Kanten und Vertizes zu setzen. Eine BFS-Traversierung eines Graphen G:

```
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
```

7.11.1 Eigenschaften

- $BFS\ (G, s)$ besucht alle Vertizes und Kanten in G_s (verbundene Komponente vom Start-Vertex s)
- Die Discovery-Kanten von $BFS\ (G, s)$ bilden einen aufspannenden Baum T_s von G_s
- Für jeden Vertex v in L_i 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

7.11.2 Applikationen

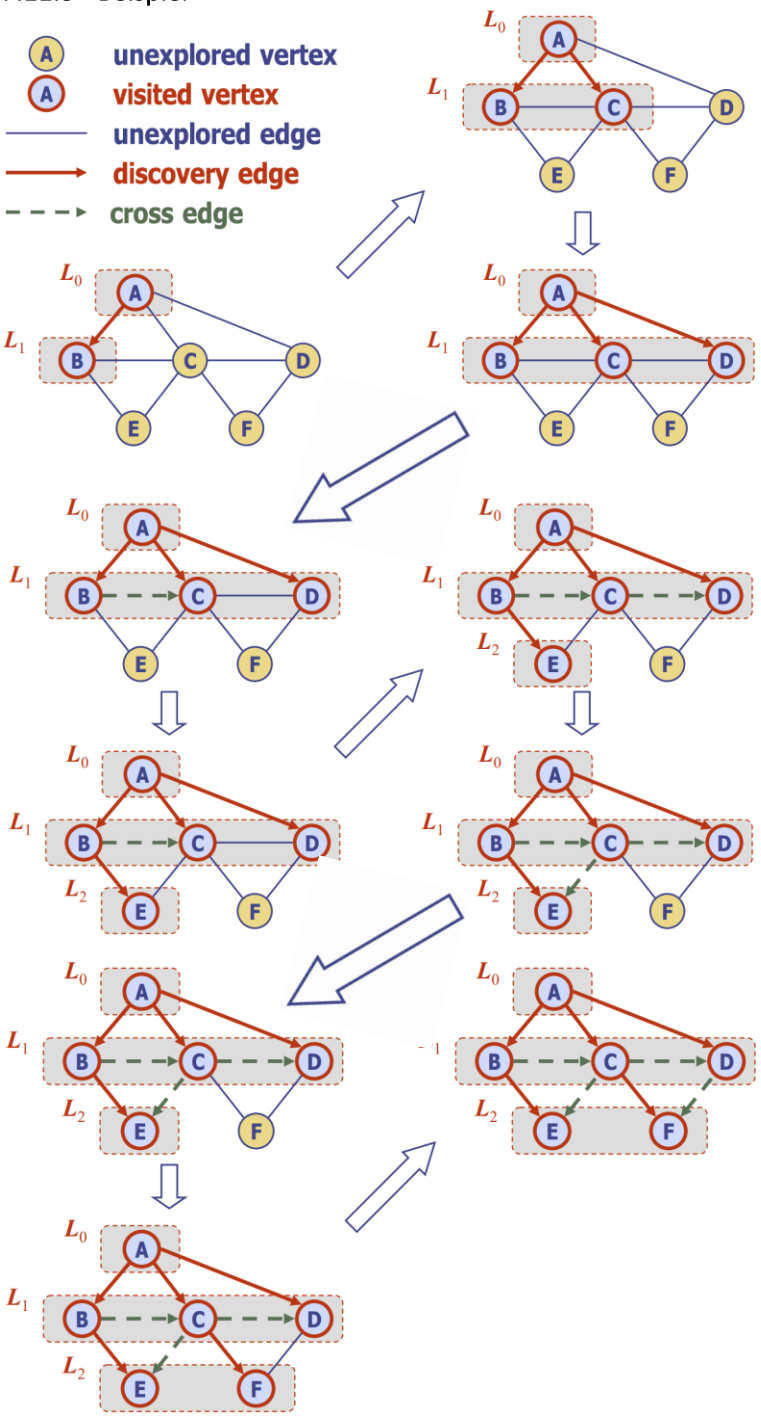
Wir können die BFS-Traversierung eines Graphen G benutzen, um folgende Probleme in $O(n + m)$ Zeit zu lösen:

- 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

BFS kann man auch erweitern, um andere Graphenprobleme zu lösen:

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

7.11.3 Beispiel



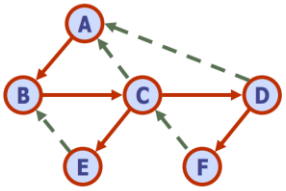
7.11.4 Laufzeitanalyse

- Graph hat n Vertizes und m Kanten
- Setzen/Lesen eines Vertex-/Kanten-Labels benötigt $O(1)$ Zeit
- Jeder Vertex wird zweimal markiert: zuerst **UNEXPLORED**, dann **VISITED** → $O(2n)$
- Jede Kante wird zweimal markiert: zuerst **UNEXPLORED**, dann **DISCOVERY** oder **CROSS** → $O(2m)$
- Die Methode incidentEdges() wird pro Vertex einmal aufgerufen. $O(deg(v))$ bei Adjazenzlisten-Strukturen, also $\sum_v \deg(v) = 2m$ → $O(2m)$
- $O(2n + 2m + 2m) = O(2n + 4m) = O(n + m)$

7.12 DFS vs. BFS

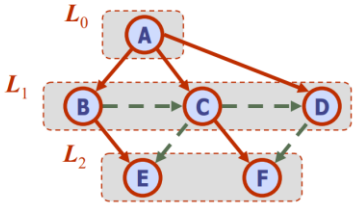
Applikationen	DFS	BFS
Aufspannender Wald, Verbundene Komponenten, Pfade, Zyklen	√	√
Kürzester Pfad		√
Biconnected Komponenten	√	

Back edge (v,w)
(Rückwärtskante)
- w ist ein Vorfahre von v im Baum der Suchkanten



DFS

Cross edge (v,w)
(Kreuzungskante)
- w ist auf der selben Stufe wie v oder auf dem nächsten Level im Discovery-Kantenbaum



BFS

7.13 Directed Graphs / Digraphs / Gerichtete Graphen

Graph, dessen Kanten alle gerichtet (jede Kante geht nur in eine Richtung) sind

7.13.1 Eigenschaften

- Graphen hat n Vertizes und m Kanten
- Wenn G **einfach** (das heisst ohne Schleifen und ohne Mehrfach-Kanten (parallele Kanten)) ist: $m \leq n(n-1)$
- Wenn **In-Kanten** (Kanten, die auf diesen Vertex zeigen) und **Out-Kanten** (Kanten, die von diesem Vertex ausgehen) in separaten Adjazenz-Listen sind: Laufzeit für Zugriff auf In- und Out-Kanten proportional zur Grösse der Listen

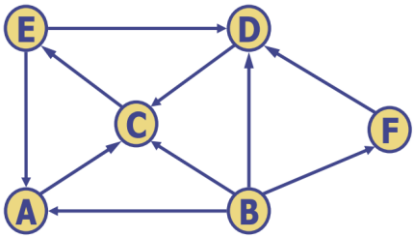
7.13.2 Applikationen

- **Scheduling**: $Kante\ (a,b)$ bedeutet, dass Task a terminieren muss bevor Task b gestartet wird
- **DFS**: kann spezialisiert werden, indem Kanten nur entlang ihrer Richtung traversiert werden, neu gibt es 4 statt 2 Typen von Kanten: DISCOVERY (Kante des Waldes), BACK (Verbindung zu einem Vorgänger), FORWARD (Verbindung zu einem Nachfolger im Baum), CROSS (Alle übrigen Kanten)
- **BFS**: kann spezialisiert werden, indem Kanten nur entlang ihrer Richtung traversiert werden

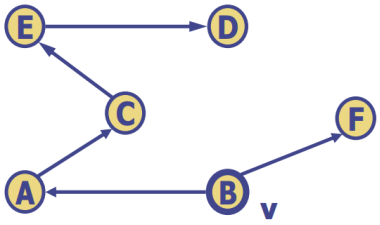
7.13.3 Erreichbarkeit

Bei einem gerichteten Graphen können möglicherweise nicht alle Vertices von allen Vertices erreicht werden, auch wenn der Graph verbunden ist

Von A aus sind beispielsweise nur C, D & E erreichbar:



Von B aus sind A, C, D, E & F erreichbar:



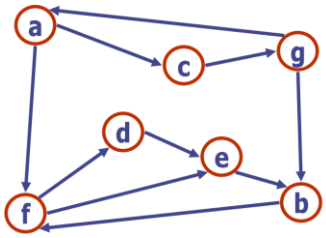
Strong Connectivity: Jeder Vertex kann alle anderen Vertizes erreichen, Brute Force Ansatz zur Überprüfung, ausführen für jeden Vertex v im Graph G:

1. DFS von v aus durchführen, wenn es einen nicht besuchten Vertex w gibt, $return\ false$
2. Die Richtungen aller Kanten in G umkehren

3. DFS von v aus durchführen, wenn es einen nicht besuchten Vertex w gibt, $return\ false$
4. Sonst $return\ true$

Laufzeit wie bei DFS: $O(n+m)$

Eine streng verbundene Komponente ist ein maximaler Subgraph, sodass jeder Vertex alle anderen Vertizes im Subgraph erreichen kann. Laufzeit: $O(n+m)$ mit Tiefensuche, ist aber komplizierter (ähnlich wie Biconnectivity)



{ a , c , g }

{ f , d , e , b }

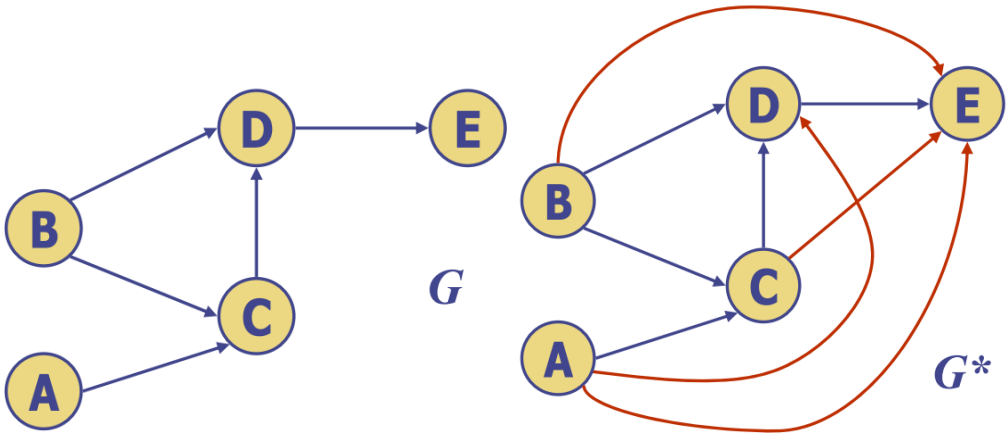
7.13.4 Transitiver Abschluss

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

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

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

Laufzeit: DFS beginnend bei jedem Vertex, also $O(n(n+m))$



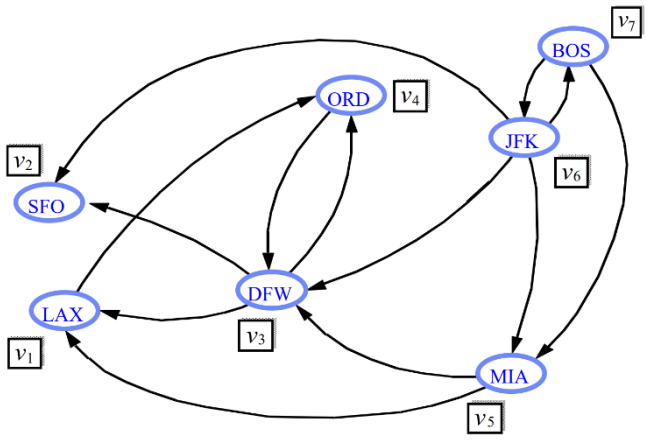
7.13.5 Floyd-Warshall Algorithmus

Alternative zum transitiven Abschluss, dynamische Programmierung

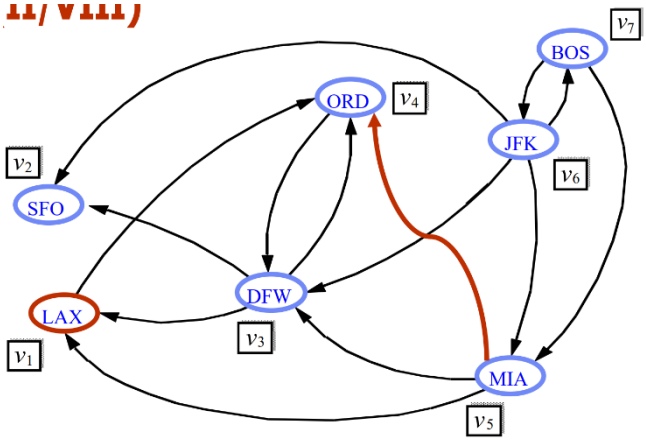
Wir haben folgenden Graphen:

$G = \{(1,4),(3,1),(3,2),(3,4),(4,3),(5,1),(5,3),(6,2),(6,3),(6,5),(6,7),(7,5),(7,6)\}$, Set $V = \{1,2,3,4,5,6,7\}$

Das Ganze grafisch & als Matrix (x aus (x,y) jeweils Reihennummer), 7 Schritte benötigt aufgrund der Set-Grösse



	1	2	3	4	5	6	7
1	1	0	0	1	0	0	0
2	0	1	0	0	0	0	0
3	1	1	1	1	0	0	0
4	0	0	1	1	0	0	0
5	1	0	1	0	1	0	0
6	0	1	1	0	1	1	1
7	0	0	0	0	1	1	1

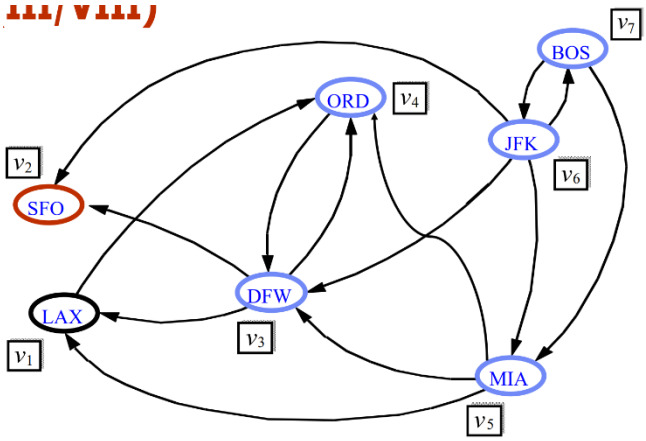


Schritt 1: 1. Spalte und 1. Reihe berücksichtigen, die Zahlen in {} sind die Felder, wo eine 1 steht

$S_1 = \{3,5\}$ $R_1 = \{4\}$ $S_1 \times R_1 = \{(3,4),(5,4)\}$

Neue Matrix:

	1	2	3	4	5	6	7
1	1	0	0	1	0	0	0
2	0	1	0	0	0	0	0
3	1	1	1	1	0	0	0
4	0	0	1	1	0	0	0
5	1	0	1	**1**	1	0	0
6	0	1	1	0	1	1	1
7	0	0	0	0	1	1	1

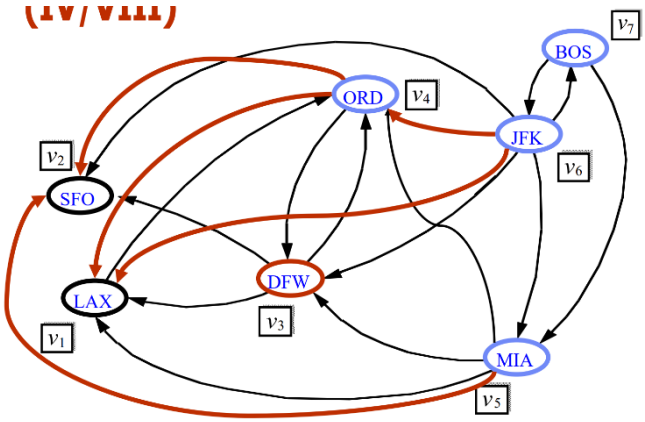


Schritt 2: 2. Spalte und 2. Reihe berücksichtigen

$S_2 = \{3,6\}$ $R_2 = \{\}$ $S_2 \times R_2 = \{\}$

Neue Matrix:

	1	2	3	4	5	6	7
1	1	0	0	1	0	0	0
2	0	1	0	0	0	0	0
3	1	1	1	1	0	0	0
4	0	0	1	1	0	0	0
5	1	0	1	1	1	0	0
6	0	1	1	0	1	1	1
7	0	0	0	0	1	1	1

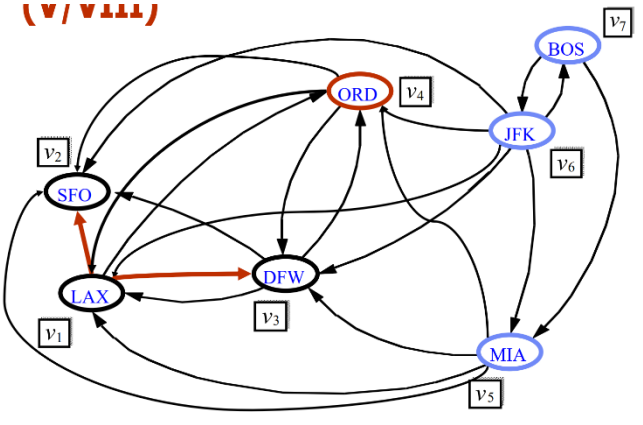


Schritt 3: 3. Spalte und 3. Reihe berücksichtigen

$S_3 = \{4,5,6\}$ $R_3 = \{1,2,4\}$ $S_3 \times R_3 = \{(4,1),(4,2),(4,4),(5,1),(5,2),(5,4),(6,1),(6,2),(6,4)\}$

Neue Matrix:

	1	2	3	4	5	6	7
1	1	0	0	1	0	0	0
2	0	1	0	0	0	0	0
3	1	1	1	1	0	0	0
4	**1**	**1**	1	1	0	0	0
5	1	**1**	1	1	1	0	0
6	**1**	1	1	**1**	1	1	1
7	0	0	0	0	1	1	1

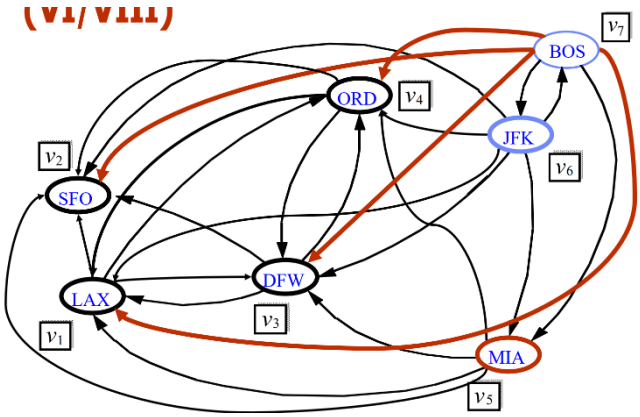


Schritt 4: 4. Spalte und 4. Reihe berücksichtigen

$S_4 = \{1,3,4,5,6\}$ $R_4 = \{1,2,3,4\}$ $S_4 \times R_4 = \{(1,1),(1,2),(1,3),(1,4),(3,1),(3,2),(3,3),(3,4),(4,1),(4,2),(4,3),(4,4),(5,1),(5,2),(5,3),(5,4),(6,1),(6,2),(6,3),(6,4)\}$

Neue Matrix:

	1	2	3	4	5	6	7
1	1	**1**	**1**	1	0	0	0
2	0	1	0	0	0	0	0
3	1	1	1	1	0	0	0
4	1	1	1	1	0	0	0
5	1	1	1	1	1	0	0
6	1	1	1	1	1	1	1
7	0	0	0	0	1	1	1

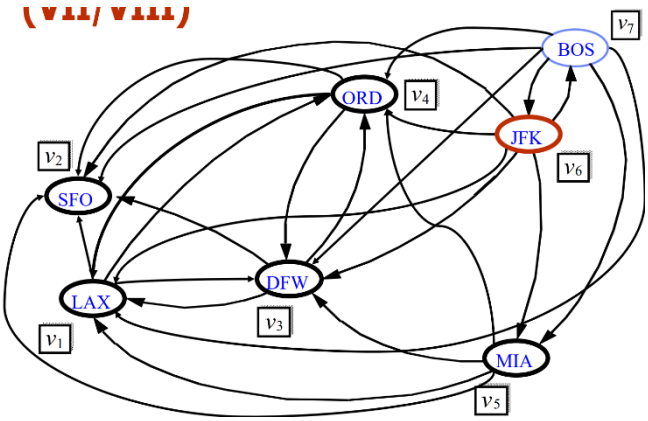


Schritt 5: 5. Spalte und 5. Reihe berücksichtigen

$S_5 = \{5,6,7\}$ $R_5 = \{1,2,3,4,5\}$ $S_5 \times R_5 = \{(5,1), (5,2), (5,3), (5,4), (5,5), (6,1), (6,2), (6,3), (6,4), (6,5), (7,1), (7,2), (7,3), (7,4), (7,5)\}$

Neue Matrix:

	1	2	3	4	5	6	7
1	1	1	1	1	0	0	0
2	0	1	0	0	0	0	0
3	1	1	1	1	0	0	0
4	1	1	1	1	0	0	0
5	1	1	1	1	1	0	0
6	1	1	1	1	1	1	1
7	1	1	1	1	1	1	1

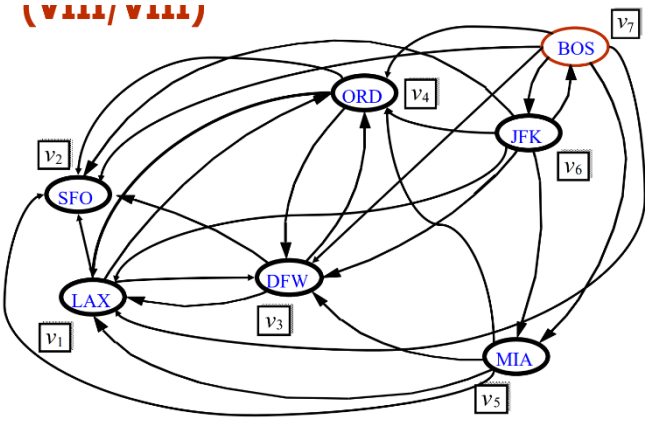


Schritt 6: 6. Spalte und 6. Reihe berücksichtigen

$S_6 = \{6,7\}$ $R_6 = \{1,2,3,4,5,6,7\}$ $S_6 \times R_6 = \{(6,1), (6,2), (6,3), (6,4), (6,5), (6,6), (6,7), (7,1), (7,2), (7,3), (7,4), (7,5), (7,6), (7,7)\}$

Neue Matrix:

	1	2	3	4	5	6	7
1	1	1	1	1	0	0	0
2	0	1	0	0	0	0	0
3	1	1	1	1	0	0	0
4	1	1	1	1	0	0	0
5	1	1	1	1	1	0	0
6	1	1	1	1	1	1	1
7	1	1	1	1	1	1	1



Schritt 7: 7. Spalte und 7. Reihe berücksichtigen

$S_7 = \{6,7\}$ $R_7 = \{1,2,3,4,5,6,7\}$ $S_7 \times R_7 = \{(6,1), (6,2), (6,3), (6,4), (6,5), (6,6), (6,7), (7,1), (7,2), (7,3), (7,4), (7,5), (7,6), (7,7)\}$

Neue Matrix:

	1	2	3	4	5	6	7
1	1	1	1	1	0	0	0
2	0	1	0	0	0	0	0
3	1	1	1	1	0	0	0
4	1	1	1	1	0	0	0
5	1	1	1	1	1	0	0
6	1	1	1	1	1	1	1
7	1	1	1	1	1	1	1

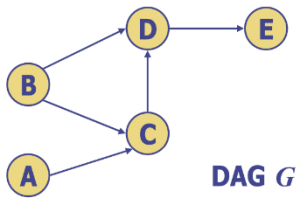
Das Ergebnis ist der transitive Abschluss des Graphen in Matrixform.

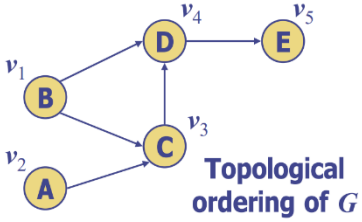
7.13.6 Directed Acyclic Graph (DAG) / topologische Ordnung

Ein DAG ist ein Digraph, der keine gerichteten Zyklen enthält.

Eine topologische Ordnung eines Digraphs ist definiert durch die Nummerierung:
$v_1, \ldots, v_n$ der Vertizes, sodass für jede Kante (v_i, v_j) gilt: $i < j$
In einem Task-Scheduling Digraphen bestimmt die Task-Sequenz mit Präzedenzbedinung die topologische Ordnung

Ein Digraph erlaubt nur dann eine topologische Ordnung, wenn es sich um einen DAG handelt



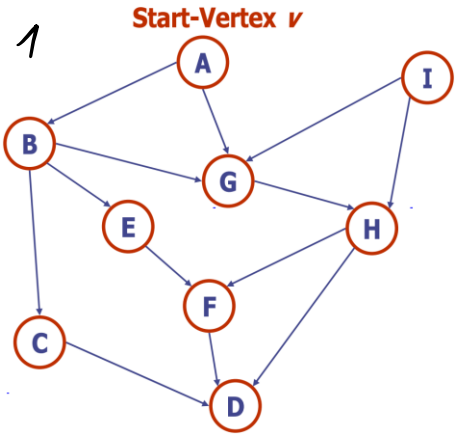


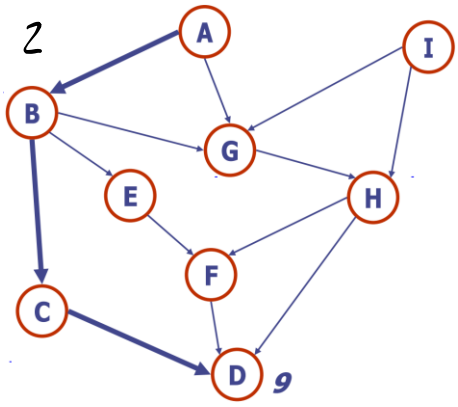
7.13.7 Topologische Sortierung mit DFS

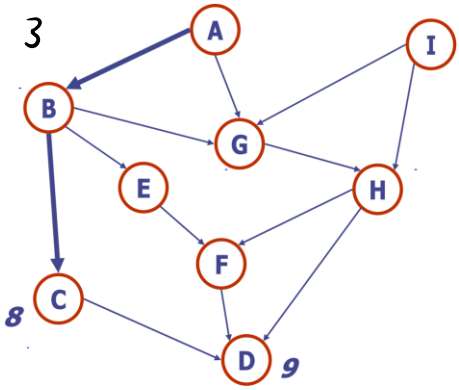
```
Algorithm topologicalDFS(G)
    Input dag G
    Output topological ordering of G
  n ← G.numVertices()
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      topologicalDFS(G, v)
```

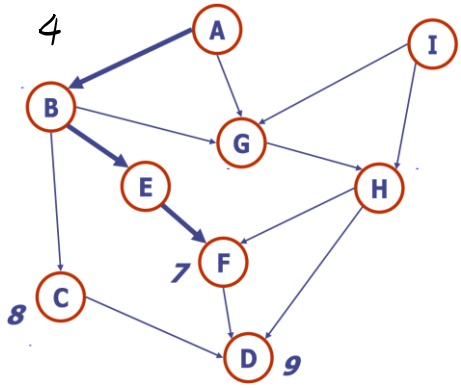
```
Algorithm topologicalDFS(G, v)
  Input graph G and a start vertex v of G
  Output labeling of the vertices of G
    in the connected component of v
  setLabel(v, VISITED)
  for all e ∈ G.outgoingEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
  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

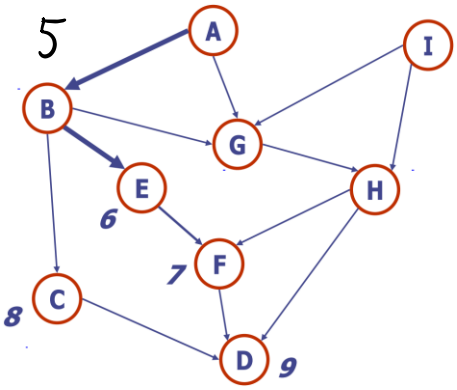
- **O(n+m) time.**

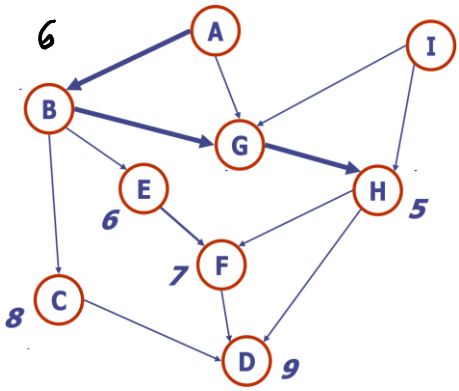


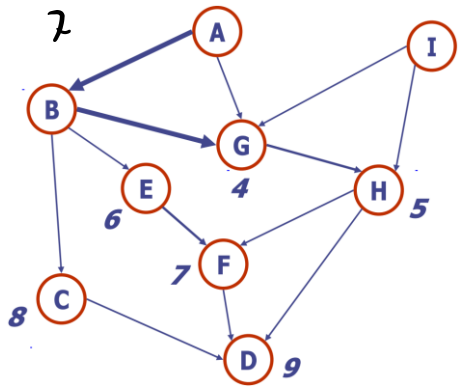


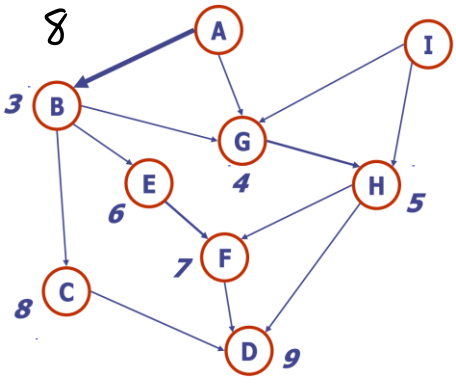


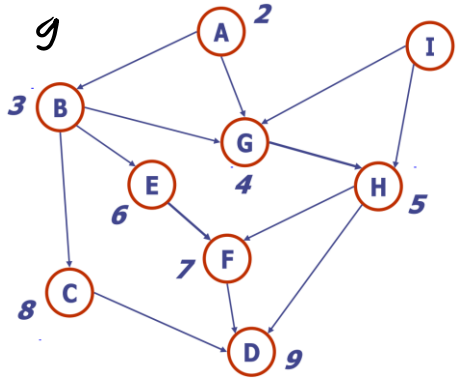


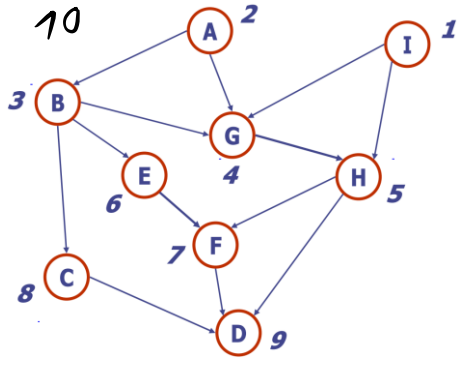




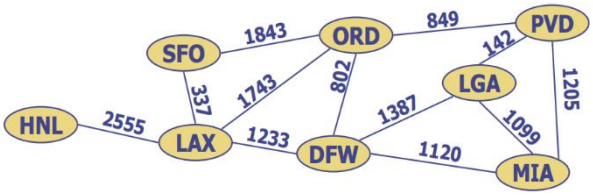








7.14 Shortest Paths Trees / Kürzeste Pfade Bäume

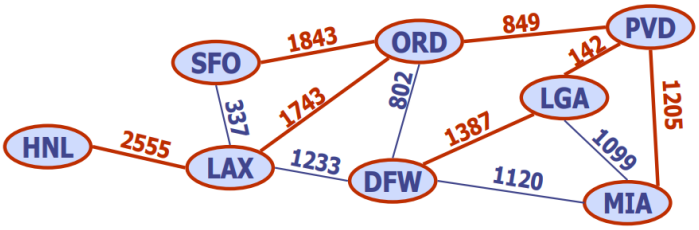


In einem **gewichteten Graphen** hat jede Kante einen assoziierten nummerischen Wert, das sogenannte **Gewicht** der Kante. Kanten-Gewichte können beispielsweise Distanzen oder Kosten repräsentieren. Wir möchten den kürzesten Pfad zwischen u und v finden, also der Pfad mit der niedrigsten Summe von Gewichten.

Eigenschaften des kürzesten Pfads:

1. Ein Teilweg eines kürzesten Weges ist selbst auch ein kürzester Weg
2. Es existiert ein Baum von kürzesten Wegen von einem Start-Vertex zu allen anderen Vertizes.

Beispiel: Baum der kürzesten Wege (rot) von PVD:



7.14.1 Dijkstra

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

```
Algorithm DijkstraDistances(G, s)
  Q ← new heap-based adaptable PQ
  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)
```

Annahmen:

- Der Graph ist verbunden
- Die Kanten sind ungerichtet
- Die Kantengewichte sind nicht negativ (basiert auf Greedy Methode, würde die Distanzen der Vertices durcheinanderbringen, wenn ein Vertex mit einer Kante mit einem negativen Gewicht später der Wolke hinzugefügt wird)

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.

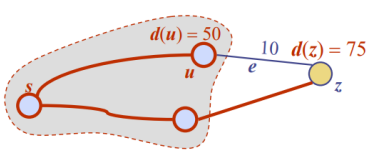
Bei jedem Schritt:

- Wir fügen der Wolke den Vertex u hinzu, welcher ausserhalb der Wolke ist und die kleinste Distanz $d(u)$ aufweist
- Wir aktualisieren 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 mit jedem Vertex: Distanz $d(v)$, Locator der Priority Queue

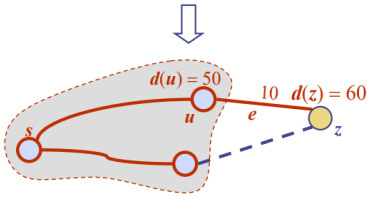
Kanten Relaxation (Entspannung):

- **Schauen wir eine Kante $e = (u,z)$ an, so dass**
  - **u der soeben der Wolke hinzugefügte Vertex ist**
  - **z nicht in der Wolke ist**

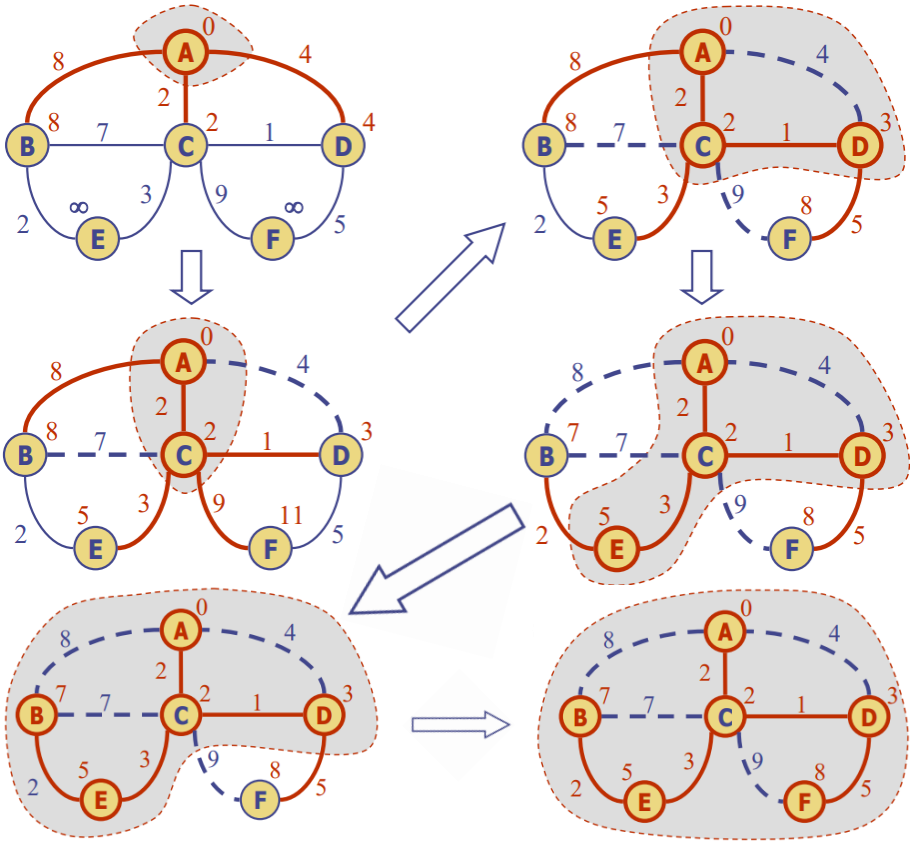


- **Die Relaxation einer Kante e aktualisiert die Distanz $d(z)$ wie folgt:**

$d(z) \leftarrow \min(\, d(z)\,,\; d(u) + weight(e)\,)$



7.14.1.1 *Beispiel*



7.14.1.2 *Laufzeitanalyse*

- Graph hat n Vertizes und m Kanten
- Methode $incidentEdges$ wird für jeden Vertex einmal aufgerufen $O(deg(v))$ bei Adjazenzlisten-Strukturen, also $\sum_v \deg(v) = 2m \rightarrow O(2m)$
- Wir setzen/lesen das Distanz- und das Locator-Label eines Vertex z insgesamt $O(deg(z))$ mal
- Setzen/Lesen eines Labels benötigt $O(1)$ Zeit
- Jeder Vertex wird einmal in die Priority Queue eingefügt und einmal gelöscht, wobei jedes Einfügen oder Löschen $O(log\ n)$ Zeit benötigt
- Der Schlüssel eines Vertex w in der Priority Queue wird maximal $deg(w)$ mal geändert, wobei jede Änderung $O(log\ n)$ Zeit benötigt
- $O((n + m) \cdot \log(n))$

7.14.1.3 *Kürzester Pfad Bäume*

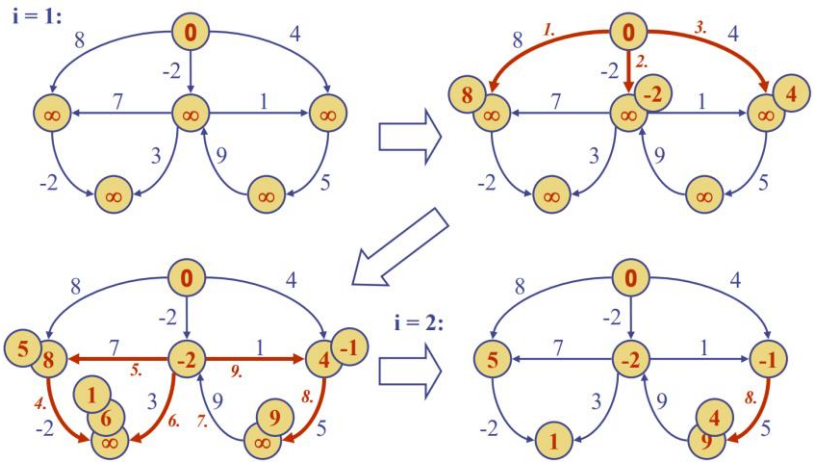
Dijkstra kann verwendet werden, um einen Baum von kürzesten Wegen vom Start-Vertex aus zu allen anderen Vertizes auszugeben. Mit jedem Vertex speichern wir ein drittes Label: Eltern-Kante im kürzesten Weg, im Relaxations-Schritt aktualisieren wir dieses Label

```
Algorithm DijkstraShortestPathsTree(G, s)
    …
  for all v ∈ G.vertices()
    …
    setParent(v, ∅)
    …
    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)
        setParent(z,e)
        Q.replaceKey(getLocator(z), r)
```

7.14.2 Bellman-Ford Algorithmus

Funktioniert auch mit negativ-gewichteten Kanten, Voraussetzung sind gerichtete Kanten und keine negativ-gewichteten Schlaufen, Iteration i findet alle kürzesten Pfade, welche i Kanten besitzen, Laufzeit $O(nm)$

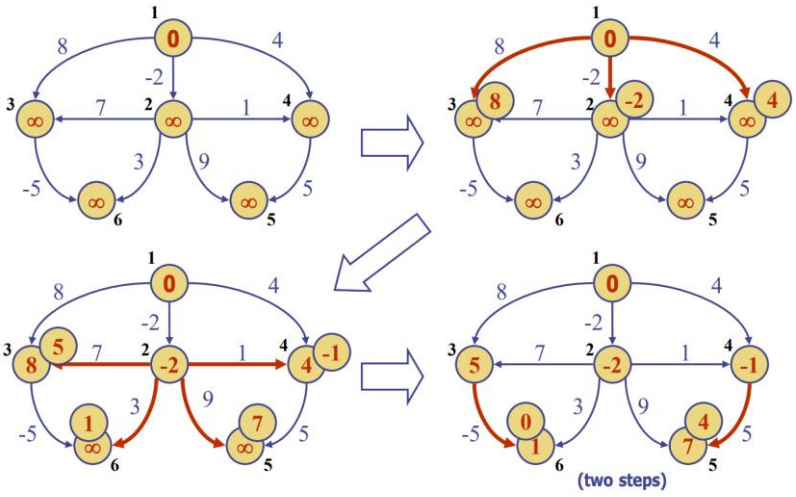
```
Algorithm BellmanFord(G, s)
  for all v ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
  for i ← 1 to n-1 do
    for each e ∈ G.edges()
      { relax edge e }
      u ← G.origin(e)
      z ← G.opposite(u, e)
      r ← getDistance(u) + weight(e)
      if r < getDistance(z)
        setDistance(z, r)
```



7.14.3 DAG-basierter Algorithmus

Funktioniert mit negativ-gewichteten Kanten, benutzt eine topologische Reihenfolge, benutzt keine ausgefallenen Datenstrukturen, ist viel schneller als Dijkstra, Laufzeit $O(n + m)$

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

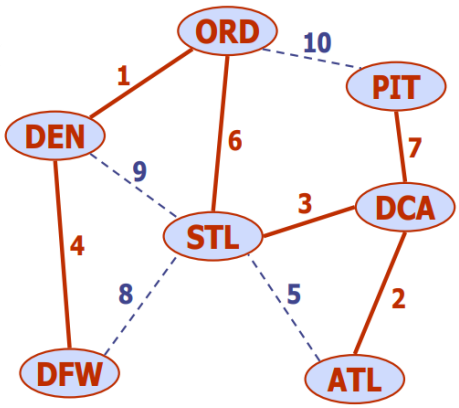


7.15 Minimum Spanning Trees / Minimale Aufspannende Bäume

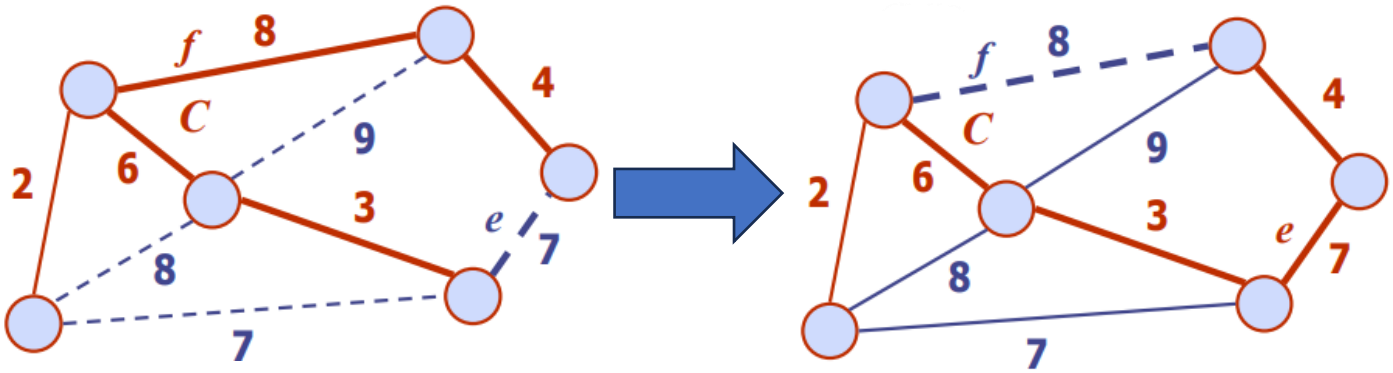
Aufspannender Subgraph: Subgraph eines Graphen G beinhaltend alle Vertizes von G

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

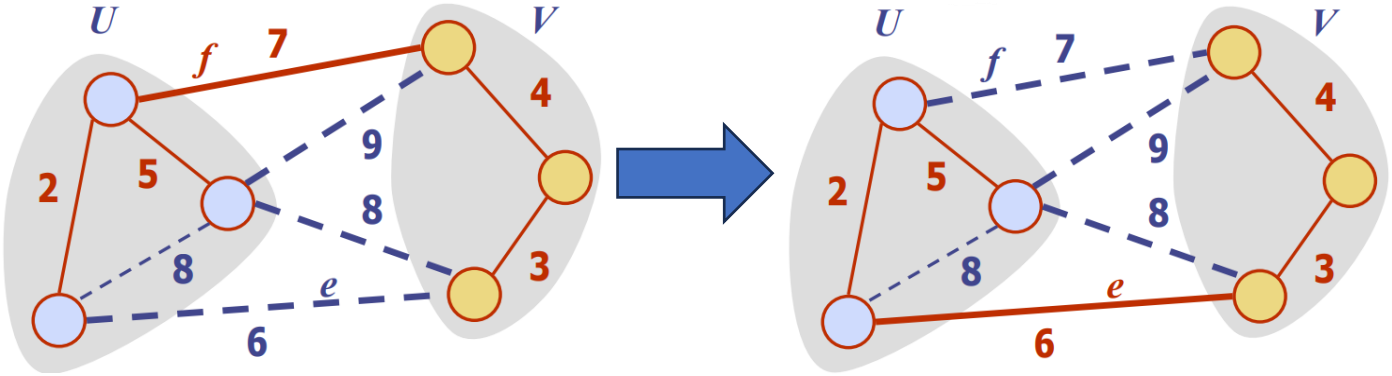
Minimal Aufspannender Baum (MST): Aufspannender Baum eines gewichteten Graphen mit minimalem totalen Kantengewicht



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 gilt: $weight(f) \leq weight(e)$, wenn die Bedingung nicht erfüllt ist, 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 beinhaltet, wenn die Teilmengen bereits durch f verbunden sind, welches ein höheres Gewicht als e hat, erhalten wir einen besseren MST, wenn wir f durch e ersetzen



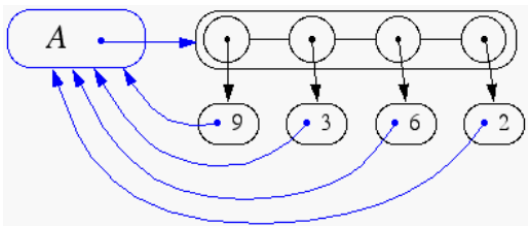
7.15.1 Kruskals Algorithmus

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 einen Baum T, welcher unser MST ist

Der Algorithmus behält einen Wald von Bäumen, eine Kante ist akzeptiert, wenn sie verschiedene Bäume verbindet, wir brauchen eine Datenstruktur, welche eine **Partition** verwaltet, z.B. eine Sammlung von disjunkten Sets mit folgenden Operationen: find(u) gibt ein Set U zurück enthaltend u, union(u,v) ersetzt die Sets, welche u und v speichern mit deren Vereinigung

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

Repräsentation einer Partition: jedes **Set** ist in einer **Sequenz** gespeichert, jedes **Element** hat eine **Referenz zurück auf das Set**, Operation $find(u)$ benötigt $O(1)$ Zeit und gibt das Set zurück, in dem u ist, in der Operation $union(u, v)$ verschieben wir die Elemente des kleineren Sets in die Sequenz des grösseren Sets und aktualisieren deren Referenzen, die Zeit für die Operation $union(u, v)$ ist $min\ (n_u, n_v)$, wobei n_u und n_v die Grössen der Sets sind, die u und v beinhalten, wenn ein Element verarbeitet wird, geht es in ein Set mit mindestens doppelter Grösse, jedes Element wird höchstens $log(n)$ mal verwendet

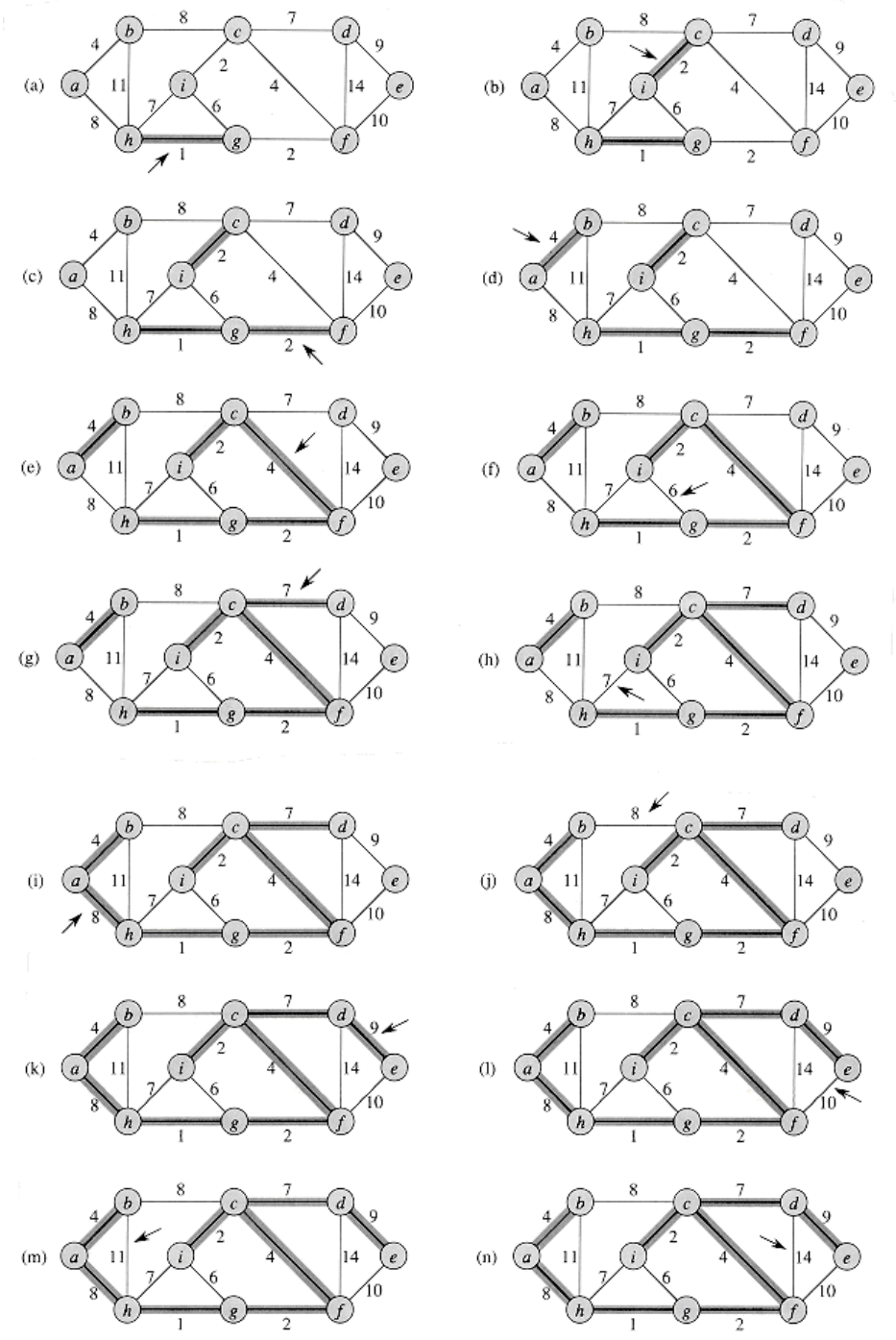


Partitions-Basierte Implementation:

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*) ← *Q*.removeMinElement().endVertices()
  if *P*.find(*u*) != *P*.find(*v*) **then**
    Add (*u*,*v*) to *T*
    P.union(*u*,*v*)
return *T*

Laufzeit: O(m log n)

Beispiel: In jedem Schritt fügen wir die Kante mit dem niedrigsten Gewicht ein, sofern sie keine Schlaufe generiert



7.15.2 Prim-Jarniks Algorithmus

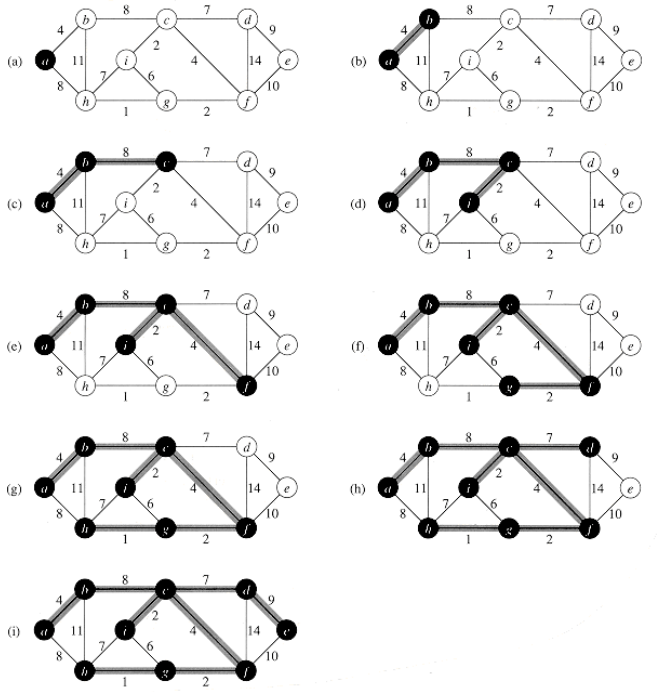
Wie Dijkstra (für verbundene Graphen), wir nehmen einen beliebigen Vertex s und generieren den MST als eine Wolke von Vertizes von s, wir speichern zu jedem Vertex ein Label $d(v)$ = die kleinste Gewichtung einer Kante, welche v mit einem Vertex der Wolke verbindet

Bei jedem Schritt fügen wir der Wolke den Vertex u von ausserhalb der Wolke mit dem kleinsten Distanz-Label hinzu

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 drei Eigenschaften zu jedem Vertex: Distanz $d(v)$, Elternkante MST, Locator der Priority Queue, ein Hash-Set bildet die Wolke ab.

```
Algorithm PrimJarnikMST(G)
  Q ← new heap-based adaptable PQ
  cloud ← new Hash-Set
  s ← a vertex of G
  for all v ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
    setParent(v, ∅)
    l ← Q.insert(getDistance(v), v)
    setLocator(v, l)
  while ¬Q.isEmpty()
    u ← Q.removeMin().getValue()
    cloud.add(u)
    for all e ∈ G.incidentEdges(u)
      z ← G.opposite(u, e)
      if ¬cloud.contains(z)
        r ← weight(e)
        if r < getDistance(z)
          setDistance(z, r)
          setParent(z, e)
          Q.replaceKey(getLocator(z), r)
```

Beispiel: Bei jedem Schritt wird der Vertex hinzugefügt, welcher mit der niedrigsten Cost von irgendeinem Vertex, das schon im MST ist, aus erreichbar ist, wir starten mit Vertex a



7.15.2.1 Laufzeitanalyse

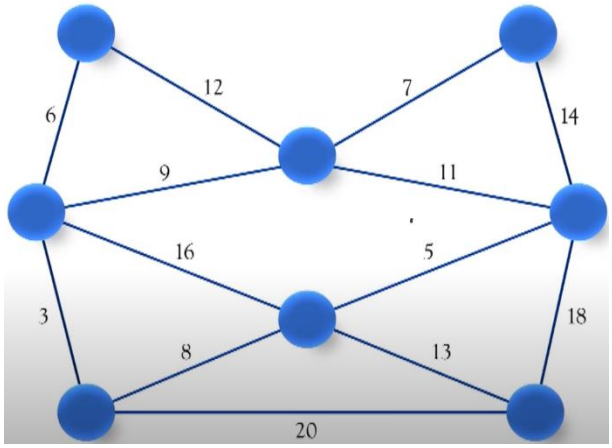
- Graph hat n Vertizes und m Kanten
- Methode $incidentEdges$ wird für jeden Vertex einmal aufgerufen $O(deg(v))$ bei Adjazenzlisten-Strukturen, also $\sum_v \deg(v) = 2m \rightarrow O(2m)$
- Wir setzen/lesen das Distanz-, Parent- und das Locator-Label eines Vertex z insgesamt $O(deg(z))$ mal
- Setzen/Lesen eines Labels benötigt $O(1)$ Zeit
- Jeder Vertex wird einmal in die Priority Queue eingefügt und einmal gelöscht, wobei jedes Einfügen oder Löschen $O(log\ n)$ Zeit benötigt
- Der Schlüssel eines Vertex w in der Priority Queue wird maximal $deg(w)$ mal geändert, wobei jede Änderung $O(log\ n)$ Zeit benötigt
- $O((n + m) \cdot \log(n))$ (sofern Adjazenz-Listen Struktur)

7.15.3 Boruvkas Algorithmus

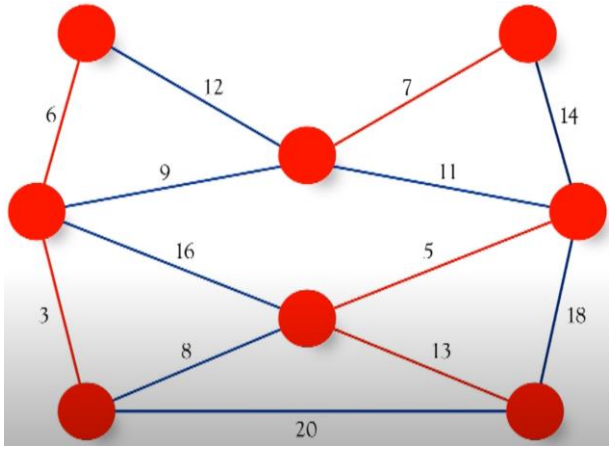
Wie Kruskals Algorithmus, bildet Boruvkas Algorithmus viele "Wolken" aufs Mal. Jede Iteration der while-Schlaufe halbiert die Anzahl der verbundenen Komponenten in T. Laufzeit $O(m \cdot \log(n))$

```
Algorithm BoruvkaMST(G)
    T ← V  {nur die Vertizes von G}
  while T weniger Kanten hat als n-1 do
    for each verbundene Komponente C in T do
      Kante e sei die kleinst gewichtete Kante von C zu einer anderen
      Komponente in T.
      if e ist nicht schon in T then
        Füge e T hinzu
  return T
```

Beispiel:



1. Durch jeden Vertex loopen und jeweils die angrenzende Kante mit dem geringsten Gewicht auswählen



2. Durch jede verbundene Komponente loopen und die Kante mit dem geringsten Gewicht auswählen, um jeweils zwei Komponenten zu verbinden
3. Wenn man noch keinen MST hat, Schritt 2 wiederholen