Übersicht der Algorithmen in AlgDat

Inhalt

Binäre Such-Bäume

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

- Gegeben sind die drei Knoten u, v, und w.
- u ist im linken Teilbaum von v. w ist im rechten Teilbaum von v
- So gilt: key(u) ≤ key (v) ≤ key(w)

Methoden

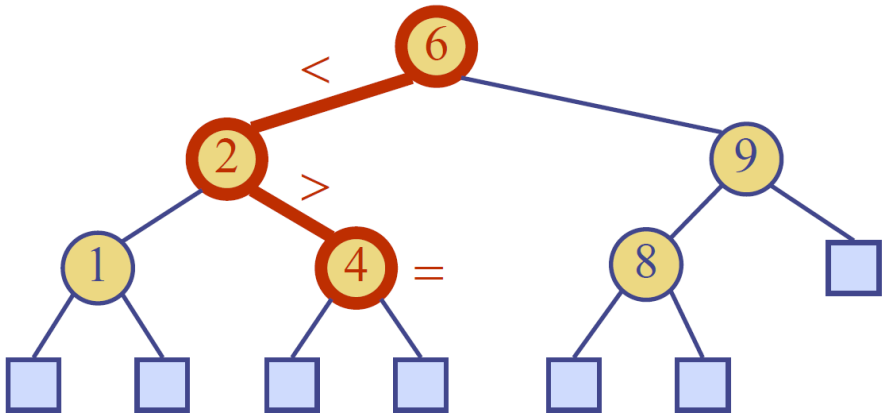
Suche

Gesucht wird nach dem Key *k*:

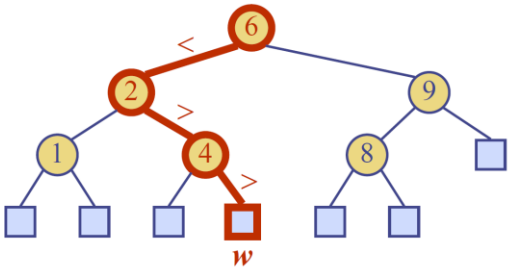
- Beginne bei Root
- Vergleich des Keys des aktuellen Knoten und *k*
- Je nach vergleich weiter zum rechten oder linken Kind des aktuellen Knoten und wieder vergleichen
- Wird ein Blatt erreicht, wurde der Key nicht gefunden

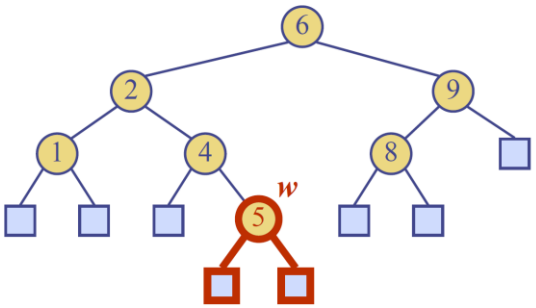
Algorithmus in Pseudo Code:

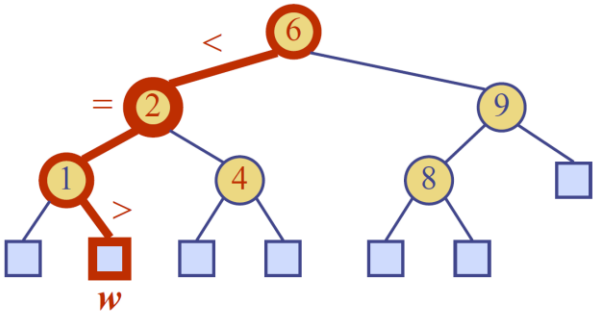
```
TreeSearch(k, v)
    if T.isExternal (v)
        return v
    if k < key(v)
        return TreeSearch(k, T.left(v))
    else if key = key(v)
        return v
    else //key > key(v)
        return TreeSearch(k, T.right(v))
```

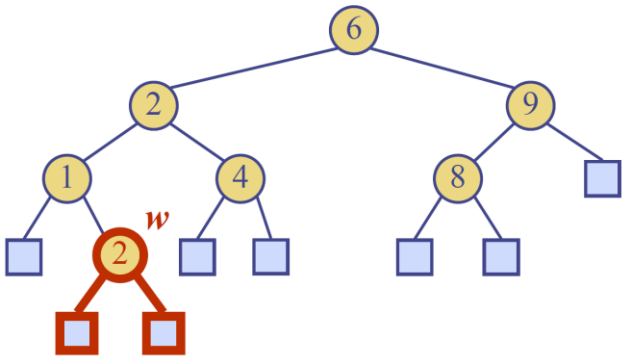


Einfügen

- Zuerst wird der Key *k* mit TreeSearch gesucht
- Annahme *k* ist noch nicht im Tree vorhanden und w ist das Blatt, welches mit der Suche gefunden wird
    - *k* beim Knoten *w* einfügen und *w* in einen internen Knoten expandieren
    - Bsp.: insert(5)
    - 

    - 

- Annahme Baum ist eine Multimap und *k* ist schon im Baum vorhanden
    - Im jeweils linken Teilbaum von *k* wird weitergesucht, bis man auf einen Blattknoten *w* stösst
    - *k* beim Knoten *w* einfügen und *w* in einen internen Knoten expandieren
    - Bsp.: insert(2):
    - 

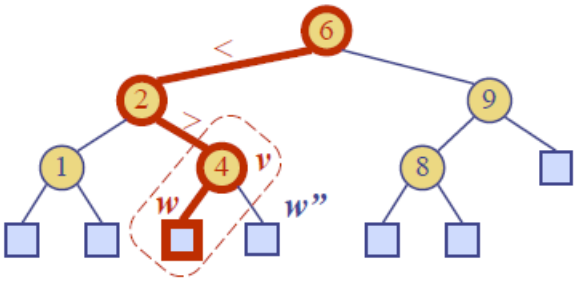
    - 

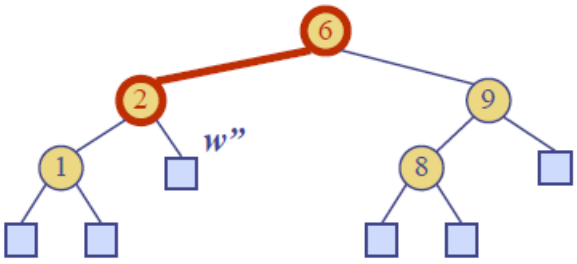
Löschen

Bei remove(k) wird zuerst der Key k gesucht

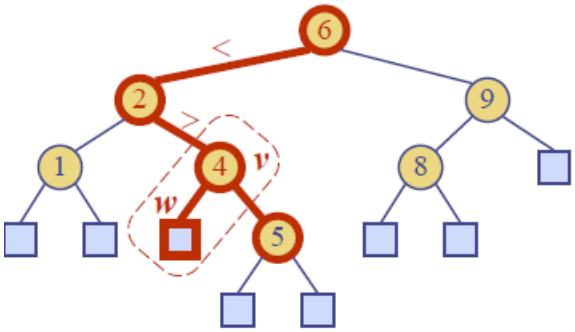
Anschliessend muss zwischen drei fällen unterschieden werden
Der Knoten v mit Schlüssel k ist:

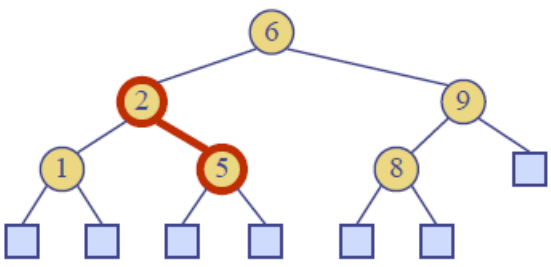
- Ein Knoten mit zwei Blatt-Kindern
  - Blatt Kinder w (links vom Knoten) und w'' (rechts vom Knoten)
  - v und w werden gelöscht
    - um w zu löschen, braucht es die Funktion removeExternal(w)
    - removeExternal(w) löscht w und seinen Eltern-Knoten und ersetzt diesen mit w'' (Geschwister-Knoten)
  - Bsp.: remove (4), ruf removeExternal(w) auf:

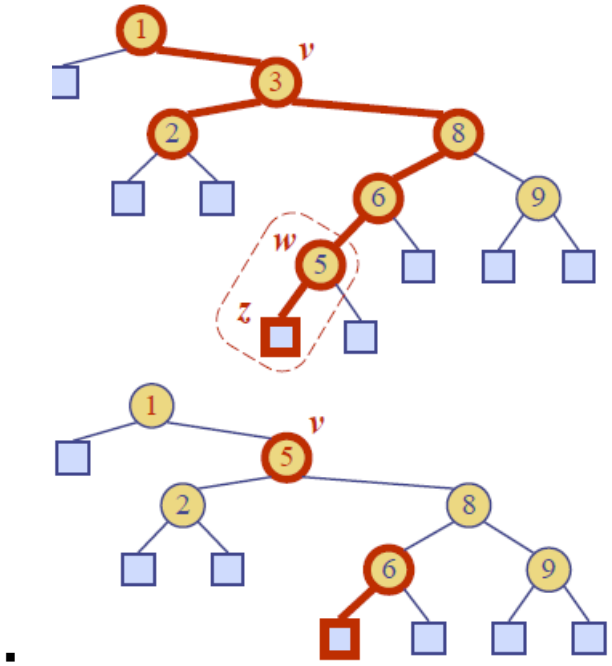




- Ein Knoten mit einem Blatt-Kind
  - genau wie bei zwei Blatt-Kindern wird die Funktion removeExternal(w) aufgerufen
    - löscht w und seinen Eltern-Knoten
    - ersetzt Eltern-Knoten durch Geschwister-Knoten von w:
  - Bsp.: remove (4), ruft removeExternal(w) auf:

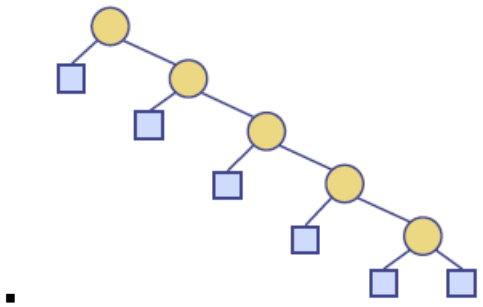
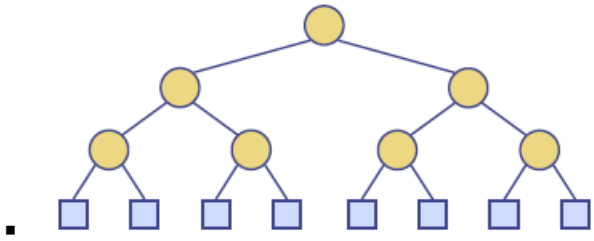




- Ein Knoten ohne Blatt-Kinder
    - Key: k, zu löschender Knoten: v
    - finde internen Knoten w, welcher v in der Inorder-Traversierung (immer left most node) folgt
    - kopiere key(w) in Knoten v
    - lösche Knoten w und sein linkes Kind z (welches Blatt sein muss) mit removeExternal(z)
    - Bsp.: remove(3)
    - 

Performance

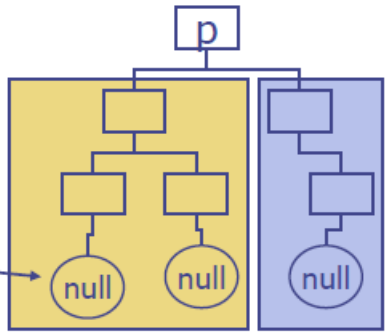
Map mit n Entries, Baum mit höhe h

- Nötiger Speicher ist O(n)
- Methoden find, insert und remove sind O(h)
    - Höhe h ist O(n) im schlechtesten Fall
    - 
    - O(log n) im besten Fall
    - 

Implementierung : Baum

- Der Binärbaum (schematisch)

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

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



Implementierung : einfügen()

- Der Binärbaum (schematisch)

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

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

Implementierung : entfernen()

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

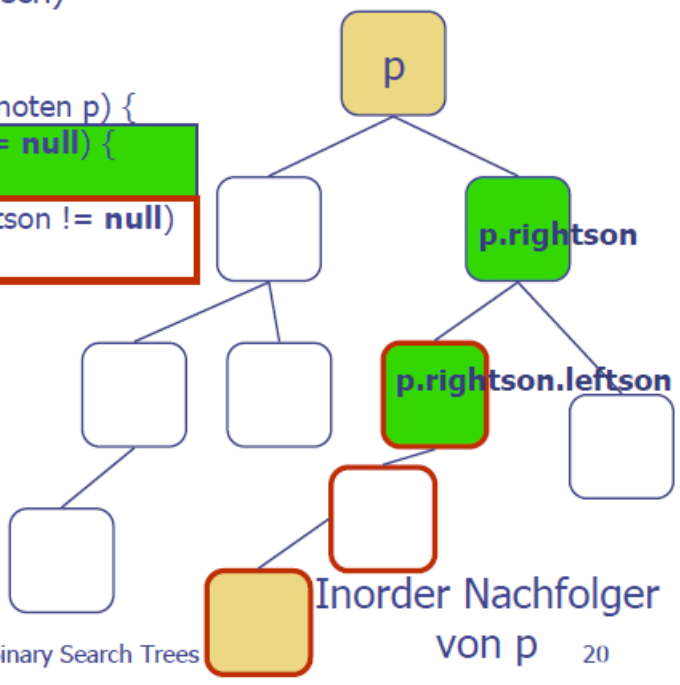
Ersetze den Knoten durch den Nachfolger gemäss Inorder / vatersymnach().

Dies garantiert, dass die Inorder-Reihenfolge eingehalten wird.

Implementierung : Hilfsmethode

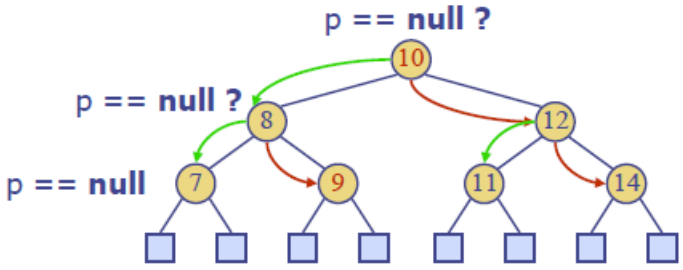
- Der Binärbaum (schematisch)

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



Implementierung : inorder()

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



AVL Tree

Binärer-Baum der immer Balanciert ist.
Für jeden internen Knoten v gilt:

- Die Höhe der Kinder von v unterscheiden sich höchstens um 1

Die Höhe eines AVL Baumes T, der n Keys speichert, ist O(log(n)).

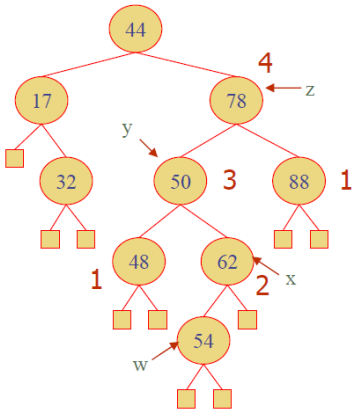
Methoden

Einfügen

Gleich wie beim binären Such-Baum, externen Knoten wird expandiert. Dabei kann es aber zu einer Verletzung des AVL-Merkmals (Höhenunterschied von max. 1) kommen:

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

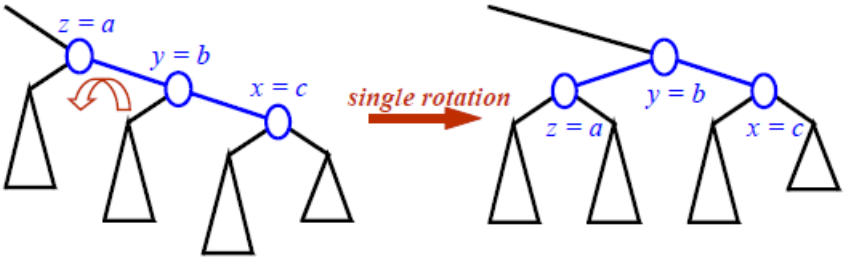
Ist das der Fall, muss der Baum wieder ausbalanciert werden. Dazu wandern wir vom neuen Knoten aufwärts, bis der erste Knote x gefunden wird, wessen Grosseltern z ein unbalancierter Knoten ist. Den Eltern-Knoten nennen wir y.



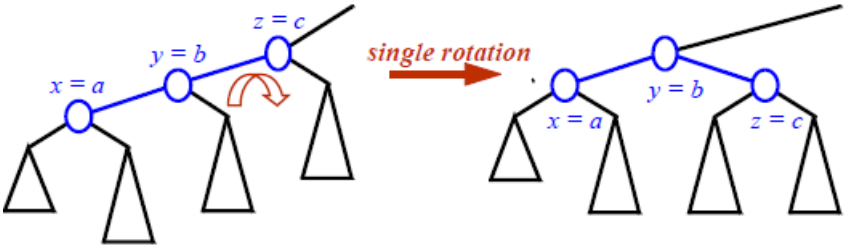
Einzel-Rotation

Ziel: y soll zum obersten Knoten des Teilbaums bestehend aus x, y und z werden.

Rotation nach links:
Linkes Kind von y, wird rechtes Kind von z



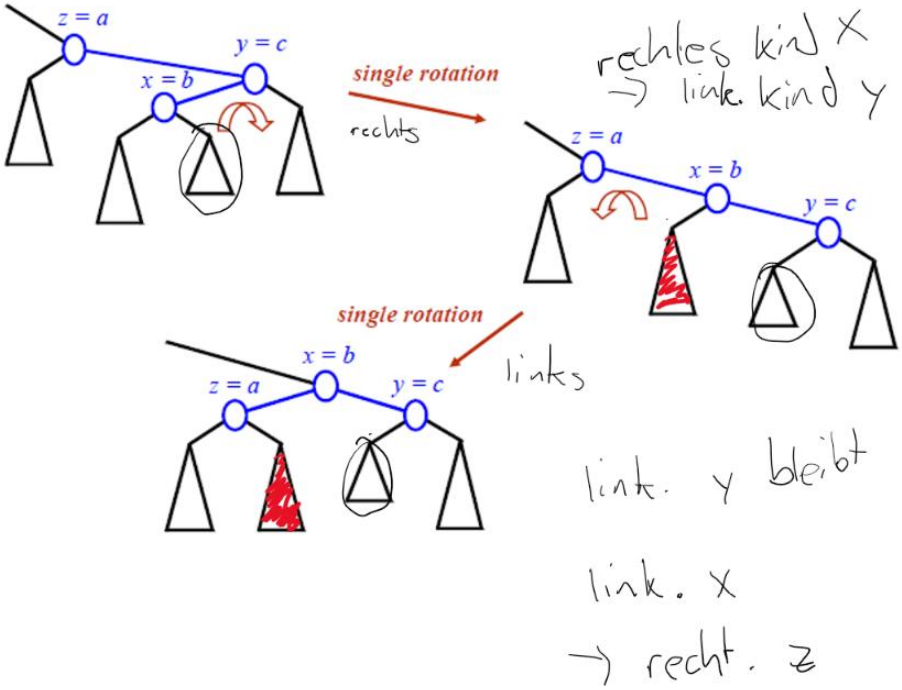
Rotation nach rechts:
Rechtes Kind von y, wird linkes Kind von z



Doppel-Rotation

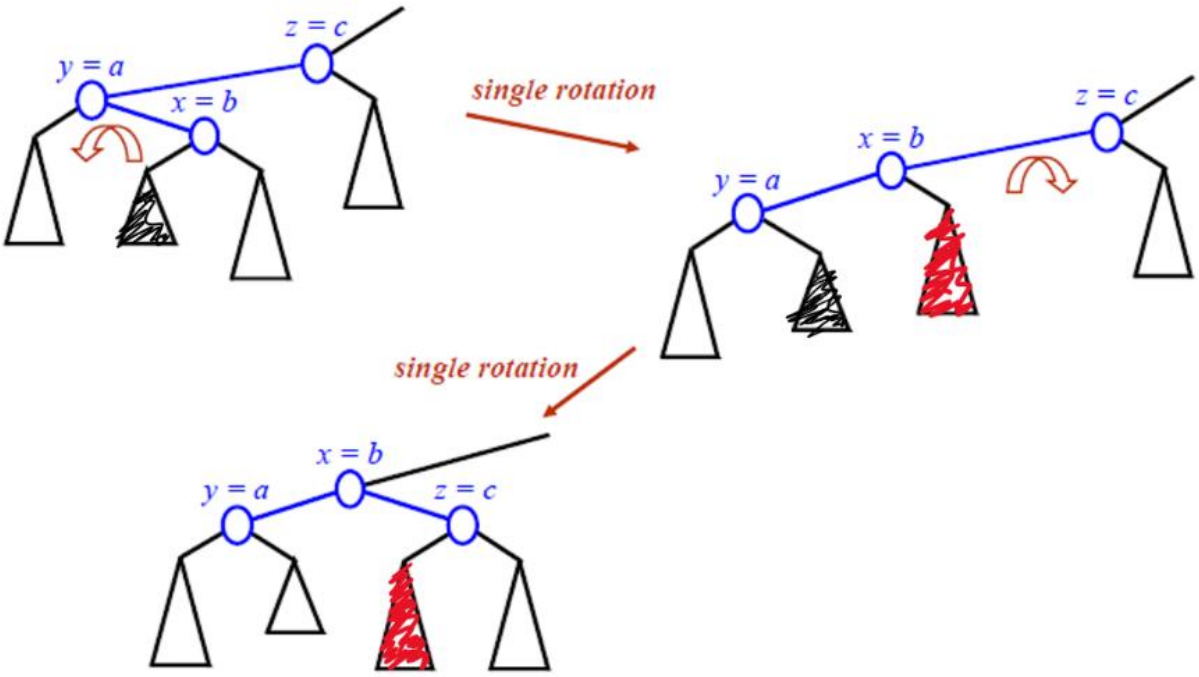
Findet Anwendung bei «Zick Zack», Ziel: x wird zum obersten Knoten in Teilbaum bestehend aus x, y und z.

Zuerst rechts dann links:
1. Rechtes Kind von x wird linkes Kind von y
2. Linkes Kind von y bleibt, linkes Kind von x wir rechtes Kind von z



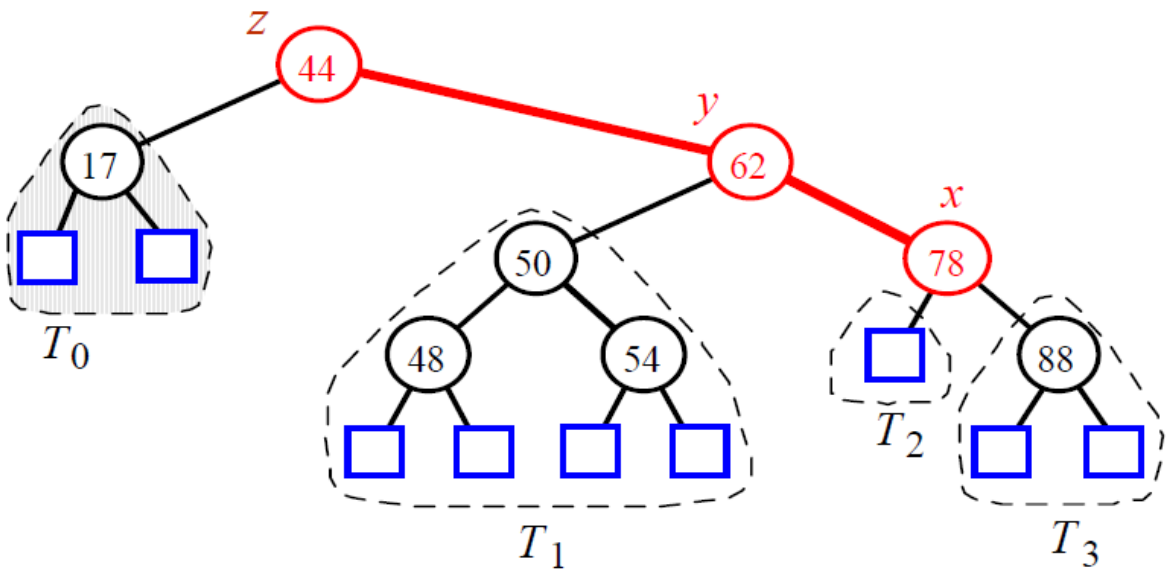
Zuerst links dann rechts:

1. Linkes Kind von x wird rechtes Kind von y
2. Rechtes Kind von x wird linkes Kind von z

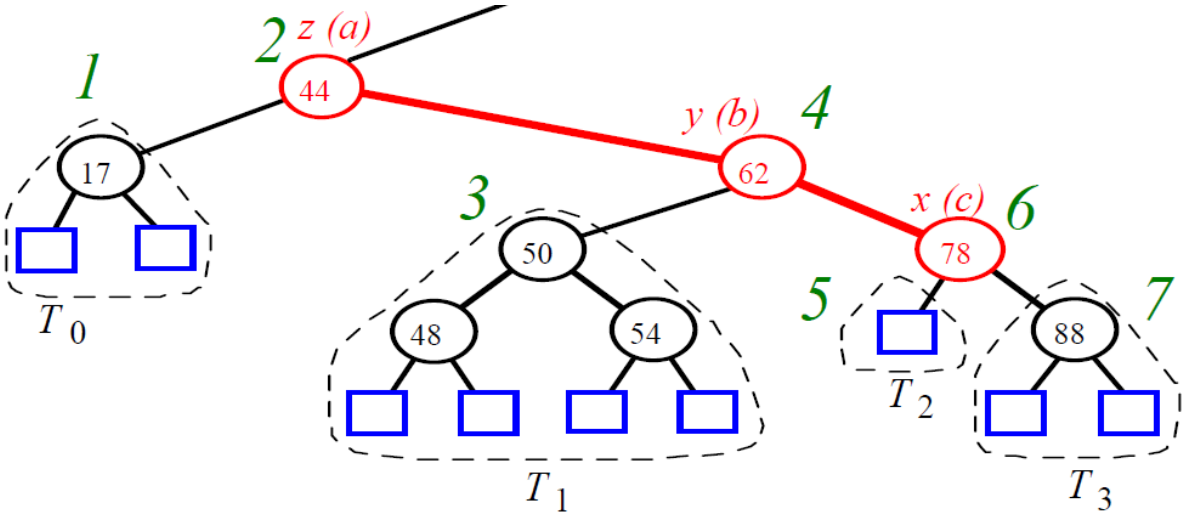


Cut-Link

Unterteilung eines Baums in x, y und z sowie 4 Bäume mit Wurzeln direkt unterhalb x, y, z (T0 – T3)

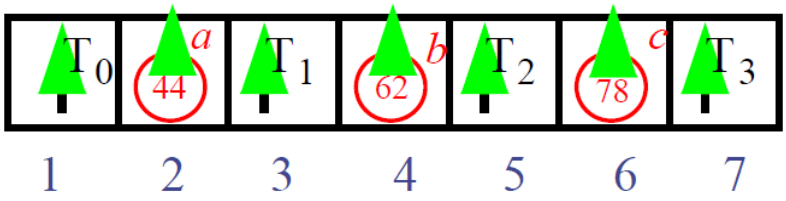


Die Knoten werden nun so umgeordnet, dass die Ordnung bei der In-Order Traversierung erhalten bleibt. Zuerst werden die sieben teile nach In-Order nummeriert und z, y, x werden in a, b, c umbenannt:

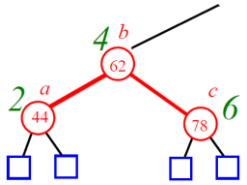
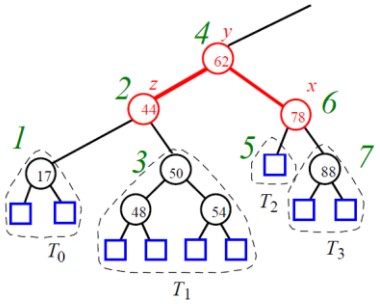


Nun wird ein Array mit 7 Elementen kreiert (1-7), die 4 Bäume werden im Array platziert:

x, y und z (c, b, a) werden ebenfalls In-Order ins Array gepackt:



Der Baum wird schrittweise wieder aufgebaut:

- Beginnend mit 4 (b)
- 2 (a) und 6 (c) werden Kinder von 4 (b)
  - 
- 1, 3, 5 und 7 (T0-T3) werden Kinder von 2 (a) und 6 (c)
  - 

Löschen

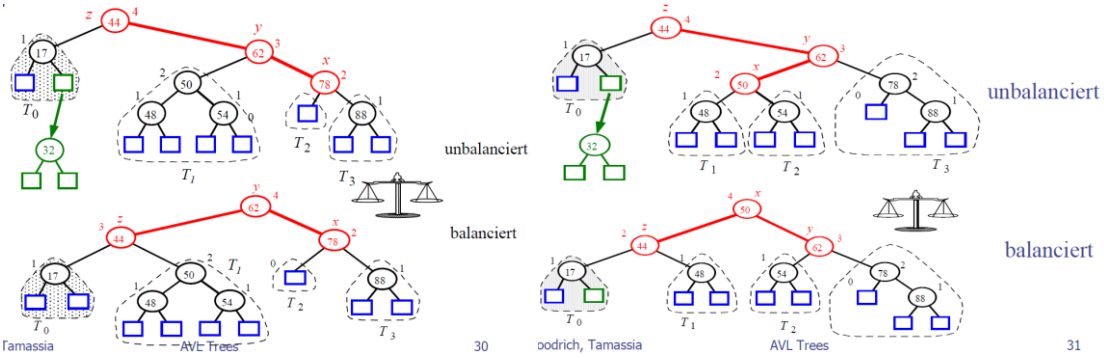
Beginnt wie im Binären Suchbaum, d.h der gelöschte Knoten wird ein leerer externer Knoten. Der Eltern-Knoten w, kann jetzt die Balance aus dem Gleichgewicht bringen.
Balancieren:

Sei z der erste unbalancierte Knoten, wenn man von w den Baum nach oben traversiert. Dabei ist y das Kind von z mit der grösseren höhe und x das Kind von y mit der grösseren höhe

Aufruf von restructure(x) um die Balance von z herzustellen.

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

Bsp. löschen von 32 (links: Einzelne-Rotation, rechts: Doppelte-Rotation)



Laufzeiten

- Restrukturierung ist O(1)
  - Bei Verwendung eines verlinkten binären Baumes
- Find ist O(log n)
  - Höhe des Baumes O(log n)
- Insert ist O(log n)
  - Find zu Beginn ist O(log n)
  - Evtl. Restrukturierung ist O(1)
- Delete ist O(log n)
  - Find zu Beginn ist O(log n)
  - Evtl. Restrukturierung ist O(1)

Implementierung (I/V)

Eine Java-basierte Implementierung eines AVL Baumes benötigt eine Knoten Klasse:

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


Implementierung (II/V)

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

Implementierung (III/V)

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

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

Implementierung (IV/V)

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

Implementierung (V/V)

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

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

Splay Tree

Ein Splay Baum ist ein binärer Such-Baum, bei welchem nach einen Zugriff auf einen Knoten dieser zur Root bewegt wird.

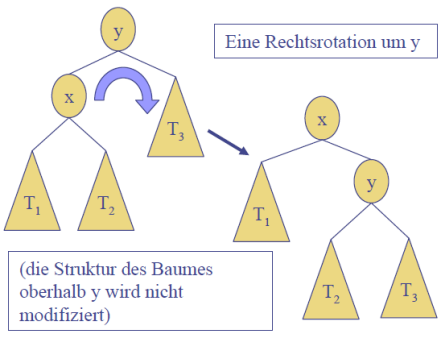
Methode	Splay Knoten
find(k)	wenn Key gefunden, benutze diesen Knoten wenn Key nicht gefunden, benutze den Eltern-Knoten des externen Knoten am Ende
insert(k,v)	Benutze den neuen Knoten bei welchem der Entry eingefügt/ersetzt wurde
remove(k)	Benutze den Eltern-Knoten des internen Knotens welcher gelöscht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

Splaying

Bewegen eines Knoten zur Root unter Benutzung von Rotationen

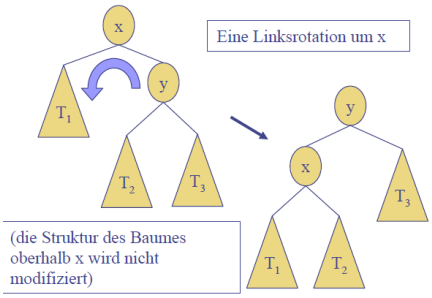
Rechts-Rotation:

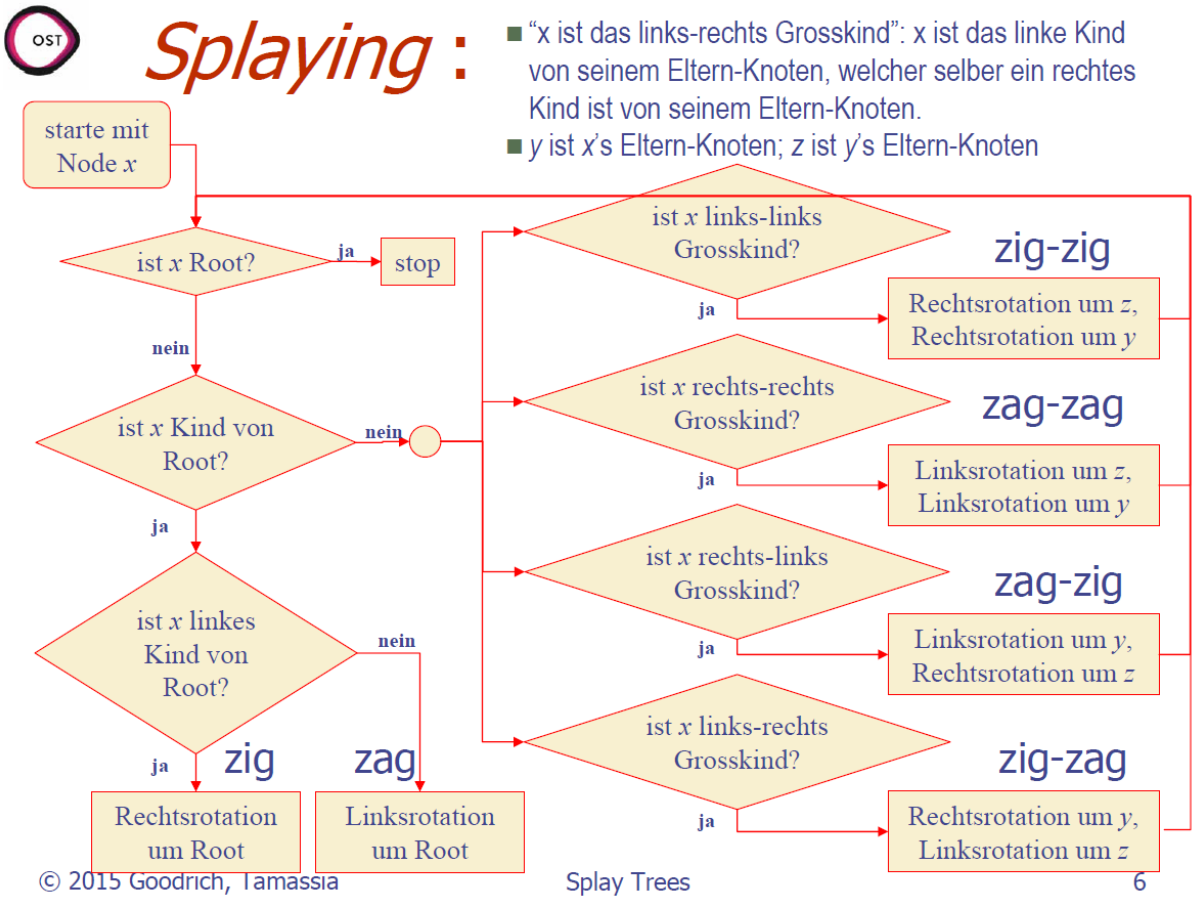
- Macht das linke Kind x des Knoten y zu y's Eltern-Knoten
- y wird zum rechten Kind von x



Links-Rotation:

- Macht das rechte Kind y des Knoten x zu x's Eltern-Knoten
- x wird zum linken Kind von y





Kosten:

- O(h), wobei h die Höhe des Baums ist
  - Durchschnittlich O(log n)
  - Je öfter ein Knoten besucht wird desto schneller, da näher an Root
  - Worst-Case ist die Höhe des Baumes n, somit O(n)
    - O(h) rotationen, jede mit O(1)

Merge Sort

Merge Sort basiert auf «Divide-and-Conquer»:

- Divide
  - Input wird in S1 und S2 geteilt
  - Je n/2 elemente
- Recur
  - Teilprobleme S1 und S2 werden rekursiv gelöst
  - Be Merge Sort wird rekursiv sortiert
- Conquer
  - Die Lösung von S1 und S2 werden in Lösung S gemischt
  - Lösung S ist sortierte Sequenz

Ein Base Case ist ein Teilproblem der Grösse 0 oder 1

Pseudo Code

MergeSort

```
mergeSort(S,C)
    Input sequences S with n elements, comparator C
    Output sequence S sorted according to C

    if S.size() > 1
        (S1, S2) = partition(S, n/2) //split S in half
        S1 = mergeSort(S1, C) //recursive call of mergeSort using partitioned
S, this is done until S1.size() = 1
        S2 = mergeSort(S2, C) //same but with S2
        S = merge(S1,S2)
    return S
```

Merge

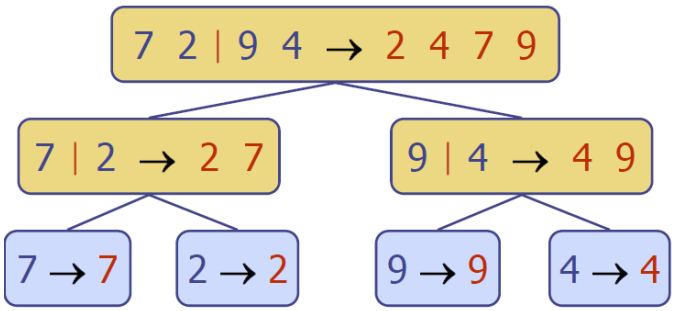
Mischen zweier sortierter Sequenzen von je n/2 Elemente mit double-linked Listen: O(n) Laufzeit

```
merge(A,B) // takes two in them selves sorted sequences and merges them
together into one sorted sequence
    Input sequences A and B with n/2 elements each
    Output sorted sequence of A and B

    S = empty sequence

    while !A.isEmpty() and !B.isEmpty()
        if A.first().element() < B.first().element()
            S.insertLast(A.remove(A.first())) // remove returns element it
removes from sequence
        else
            S.insertLast(B.remove(B.first()))
    while !A.isEmpty()
        S.insertLast(A.remove(A.first())) // append remaining elements from A
    while !B.isEmpty()
        S.insertLast(B.remove(B.first())) // append remaining elements from B
```

Merge-Sort Baum

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

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

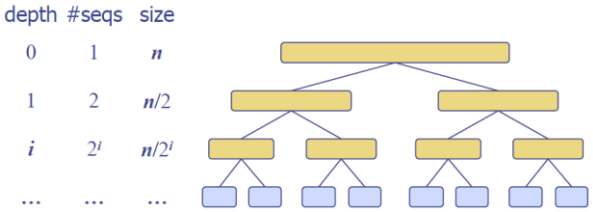


Ablauf:

- Von oben nach unten:
  - Aufsplitten der Sequenz bis länge = 1
- Von unten nach oben
  - Sortierte Rückführung der Teilsequenzen in Eltern-Knoten

Analyse

- Höhe h des Baumes ist O(log n)
  - Aufteilung in zwei Hälften bei jedem rekursivem Aufruf
- Gesamtaufwand aller Knoten einer Tiefe i ist O(n)
  - Aufteilen und Mischen 2^i Sequenzen der Grösse $n/2^i$
  - 2^{i+1} rekursive Aufrufe
- Totale Laufzeit des Merge-Sort ist O(n log n)



Nicht-Rekursiver Merge-Sort

zusammenfassen von 2, 4, 8 ... Elementen

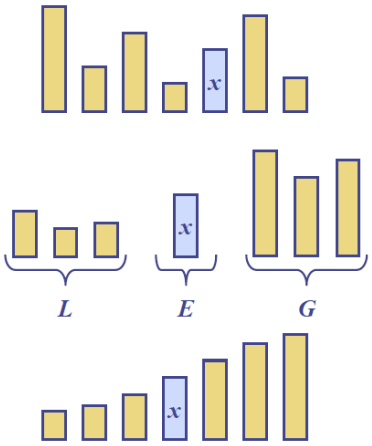
```
public static void mergeSort(Object[] orig, Comparator c) {
  Object[] in = new Object[orig.length];
  // make a new temporary array
  System.arraycopy(orig,0,in,0,in.length);
  // copy the input
  Object[] out = new Object[in.length]; // output array
  Object[] temp; // temp array reference used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

Zusammenfügen zweier Arrays zu einem

```
protected static void merge(Object[] in, Object[] out,
  Comparator c, int start, int inc) {
  // merge in[start..start+inc-1] and
  // in[start+inc..start+2*inc-1]
  int x = start; // index into run #1
  int end1 = Math.min(start+inc, in.length);
  // boundary for run #1
  int end2 = Math.min(start+2*inc, in.length);
  // boundary for run #2
  int y = start+inc;
  // index into run #2 (could be beyond array boundary)
  int z = start; // index into the out array
  while ((x < end1) && (y < end2))
    if (c.compare(in[x],in[y]) <= 0) out[z++] = in[x++];
    else out[z++] = in[y++];
  if (x < end1) // first run didn't finish
    System.arraycopy(in, x, out, z, end1 - x);
  else if (y < end2) // second run didn't finish
    System.arraycopy(in, y, out, z, end2 - y);
}
```

Quick-Sort

Auch hier wieder Divide-and-Conquer



- Divide
  - Auswahl eines Pivot-Elements x
  - Aufteilung:
    - L → Elemente kleiner als x
    - E → Elemente gleich x
    - G → Elemente grösser als x
- Recur
  - Sortiere L und G
- Conquer
  - Vereine L, E und G

Partitionierung

Ablauf:

- Entnahme eines Elements y aus Sequenz S
- Vergleich von y und Pivot Element x
- Je nach Resultat des Vergleichs y in entweder L, E oder G einfügen

Aufteilung des Quick-Sort hat eine Laufzeit von O(n)

- Jedes einfügen und entfernen jeweils am Anfang oder am Ende der Sequenz ist O(1)

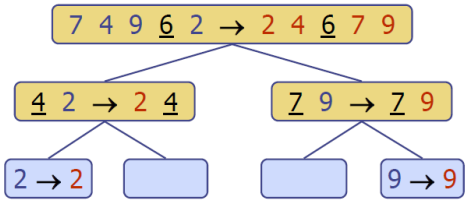
Pseudo-Code:

```
partition(S, p)
    Input sequence S, position p of pivot
    Output subsequences L,E,G of the elements of S

    L, E, G = empty sequences
    x = S.remove(p)
    E.instertLast(x)

    while !S.isEmpty()
        y = S.remove(S.first()) // remove function returns the removed element
        if y < x
            L.instertLast(y)
        else if y=x
            E.instertLast(y)
        else // y = x
            G.insertLast(y)
    return L, E, G
```

Quick-Sort Tree

Die Ausführung eines Quick-Sort kann als binärer Baum dargestellt werden

- Jeder Knoten repräsentiert einen rekursiven Aufruf des Quick-Sort und enthält:
  - unsortierte Sequenz vor der Ausführung und sein Pivot
  - sortierte Sequenz und sein Pivot nach dem Ende der Ausführung
- die Wurzel entspricht dem initialen Aufruf
- die Blätter sind Aufrufe auf Teilsequenzen der Grösse 0 or 1



Ablauf:

- Selektion eines Pivot Elements
- Aufteilung nach Pivot
- Rekursiver Aufruf
  - Selektion eine Pivot Elements in Subsequenz
  - Wird bis zur Verankerung wiederholt
- Verankerung
  - Subsequenz hat eine Länge von 1
- Vereinigung
  - Sortierte Zusammenführung der einzelnen Subsequenzen

In-Place Quick-Sort

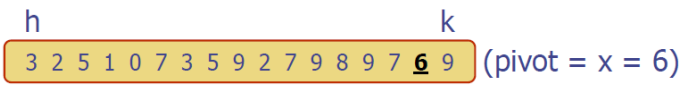
Quick-Sort kann auch In-Place (ohne kopieren von Daten in weitere Datenstruktur) implementiert werden

Bei der Partitionierung werden die Elemente derart umgeordnet, dass:

- Elemente ≤ Pivot Element, haben Index kleiner als h
- Elemente ≥ Pivot Element, haben Index grösser als k

Die rekursiven Aufrufe betrachten dann:

- Elemente mit Index kleiner als h
- Elemente mit Index grösser als k



3 h	2	5	1	0	7	3	5	9	2	7	9	8	9	7	6 Pivot	9 k

Solange sich h und k nicht kreuzen:

- Schiebe h nach rechts bis zu einem Element > Pivot
- Schiebe k nach links bis zu einem Element < Pivot
- Tausche Element von Index h mit Element von Index k

3	2	5	1	0	7 h	3	5	9	2 k	7	9	8	9	7	6 Pivot	9

h = 7 k=2, h und k werden getauscht

3	2	5	1	0	2 h	3	5	9	7 k	7	9	8	9	7	6 Pivot	9

Wiederholen bis sich h und k kreuzen

3	2	5	1	0	2	3	5 k	9 h	7	7	9	8	9	7	6 Pivot	9

Pivot Element mit Element an Index h tauschen

3	2	5	1	0	2	3	5	6	7	7	9	8	9	7	9	9

Wiederholen für L und G

L = Elemente Links von Pivot:

3	2	5	1	0	2	3	5

G = Elemente Rechts von Pivot:

7	7	9	8	9	7	9	9

Achtung!! Array wird nicht wirklich gesplittet, sonst wäre nicht In-Place

Pseudo-Code

```
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 >= r
        return
    i = random(l,r) // random int between l and r
    x = S.elemAtRank(i)
    (h,k) = inPlacePartition(x)
    inPlaceQuickSort(S,l, h-1) // move h one index to left
    inPlaceQuickSort(S,k+1,r) // move k one index to right
```

Java Implementation

```
public static void quickSort (Object[] S, Comparator c) {
   if (S.length < 2) return; // the array is already sorted in this case
   quickSortStep(S, c, 0, S.length-1); // recursive sort method
}
private static void quickSortStep (Object[] S, Comparator c,
                                   int leftBound, int rightBound ) {
   if (leftBound >= rightBound) return; // the indices have crossed
   Object temp;  // temp object used for swapping
   Object pivot = S[rightBound];
   int leftIndex = leftBound;     // will scan rightward
   int rightIndex = rightBound-1; // will scan leftward
   while (leftIndex <= rightIndex) { // scan right until larger than the pivot
     while ( (leftIndex <= rightIndex) && (c.compare(S[leftIndex], pivot)<=0) )
       leftIndex++;
     // scan leftward to find an element smaller than the pivot
     while ( (rightIndex >= leftIndex) && (c.compare(S[rightIndex], pivot)>=0))
       rightIndex--;
     if (leftIndex < rightIndex) { // both elements were found
       temp = S[rightIndex];
       S[rightIndex] = S[leftIndex]; // swap these elements
       S[leftIndex] = temp;
     }
   } // the loop continues until the indices cross
   temp = S[rightBound]; // swap pivot with the element at leftIndex
   S[rightBound] = S[leftIndex];
   S[leftIndex] = temp; // the pivot is now at leftIndex, so recurse
   quickSortStep(S, c, leftBound, leftIndex-1);
   quickSortStep(S, c, rightIndex+1, rightBound);
 }
```

Laufzeit

Worst-Case tritt dann auf, wenn das Pivot das Minimum oder Maximum-Element ist. L oder G hat dann die Länge n-1, das jeweilige andere die Länge 0. Die Laufzeit wäre damit Proportional zur Summe:

$$n + (n-1) + \ldots + 2 + 1: \sum_{i=0}^{n} i = \frac{n^2 + n}{2}$$

Worst-Case Laufzeit ist somit $O(n^2)$

Die erwartete Laufzeit beläuft sich aber auf O(n log n)

- Gleichmässiger Baum: O(log n)
- Gesamter Aufwand für alle Knoten einer Tiefe: O(n)

* siehe Analyse des Merge-Sort Baum zur genaueren Erklärung der Laufzeit

Sorting Lower Bound

Viele Sortier-Algoritmen sind vergleichsbasiert.

- Diese sortieren durch Vergleiche zwischen Paaren von Objekten

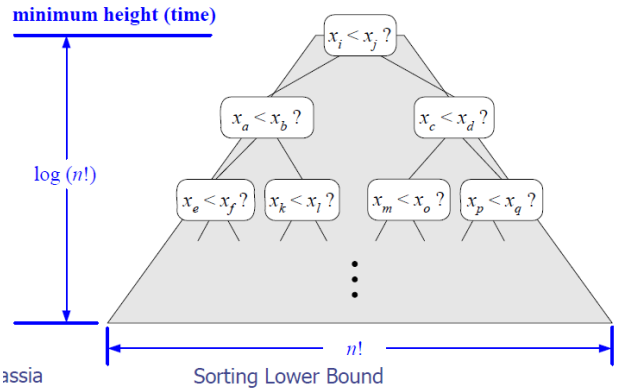
Es soll eine untere Grenze (Lower Bound) der Laufzeit hergeleitet werden für alle Algorithmen, welche Vergleiche benutzen um n Elemente x1, x2, …, xn zu sortieren.

Wir zählen die Anzahl der Vergleiche!
Jeder mögliche Durchgang des Algorithmus korrespondiert mit einem Wurzel-zu-Blatt Pfad ein einem Entscheidungs-Baum

Die Höhe des Entscheidungs-Baumes entspricht der unteren Grenze der Laufzeit.
Jede mögliche Input-Permutation führt zu einem anderen Pfad.

- wenn dies nicht so wäre, hätte ein Input …4…5… die selbe Ausgangs-Ordnung wie …5…4…(was falsch wäre).

Da n!=1*2*…*n Blätter vorhanden sind, beträgt die Höhe mindestens log (n!)



Die untere Grenze (Lower Bound)

- Jeder Vergleichs-basierte Sortier-Algorithmus hat eine minimale Laufzeit von: *log (n!)*
- Jeder solch Algorithmus hat somit eine Laufzeit von mindestens:

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

- Oder gem. Buch Anh. A Proposition 6: Stirling-Annäherung: $n! \geq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \rightarrow O(n \log n)$
- ⇨vergleichs-basierte Sortierung hat eine untere Grenze der Laufzeit von $\Omega(n \log(n))$.

Bucket-Sort

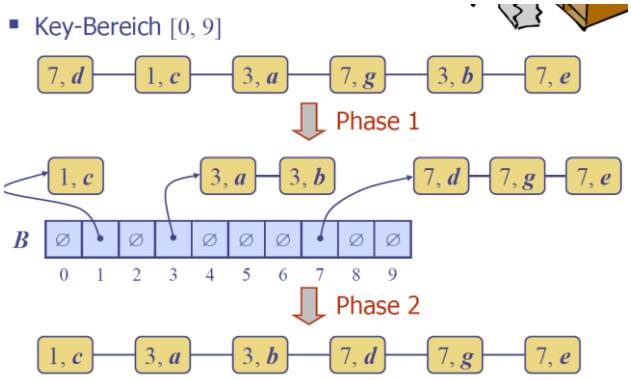
Verteilt Elemente eines Arrays in «Buckets» (oder Eimer), jeder Bucket wird dann einzeln sortiert.

Sei S eine Sequenz von n (Key, Element) Entries mit Keys im Bereich von [0, N − 1]
Ablauf:

1. Array mit leeren Buckets erstellen
2. Scatter: über Original-Array iterieren und jedes Objekt in einem Bucket platzieren
    a. Key werden dazu verwendet, um zu bestimmen in welchen Bucket das Element platziert wird (siehe Bsp. weiter unten).
3. Alle nicht leeren Buckets sortieren
    a. Entweder mit anderem Sortier-Algorithmus oder durch rekursivem Bucket-Sort
4. Gather: Buckets der Reihenfolge nach leeren und Elemente in ursprüngliches Array platzieren

Scatter hat eine Laufzeit von O(n)
Sortieren und Gather benötigen O(N+n)
Bucket-Sort benötigt O(n + N) Zeit

Bsp.:



Pseudo-Code

```
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 // B[][]

    while !S.isEmpty()
        f = S.first()
        (k,o) = S.remove(f) // remove returns object f
        B[k].insertLast((k,o)) // append object f in Bucket-Array
corresponding to key
    for i=0 to N-1 // go to all buckets
        while !B[i].isEmpty // if not empty loop through bucket
            f = B[i].first()
            (k,o) = B[i].remove(f)
            S.insertLast((k,o))
```

Eigenschaften und Erweiterungen

- Key Eigenschaften
  - Die Keys werden als Indices in einem Array benutzt und können somit nicht beliebige Objekte sein.
  - Es ist kein externer Comparator nötig.
- **Stabile** Sort Eigenschaft
  - Die relative Ordnung von zwei Entries mit dem selben Key werden durch den Algorithmus nicht verändert.

Erweiterungen

- Integer Keys im Bereich $[a, b]$
  - stecke Entry(k, o) in Bucket $B[k - a]$
- String-Keys aus einem Set D von möglichen Strings, wobei D eine konstante Grösse hat (z.B. die Namen der 50 U.S. Staaten)
  - sortiere D und berechne den Index $r(k)$ jedes Strings k von D in der sortierten Sequenz
  - füge Entry(k, o) in Bucket $B[r(k)]$ ein

Radix-Sort

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

Radix-Sort ist anwendbar für Tupel mit Integer-Keys im Bereich [0, N − 1] in jeder Dimension i.

Radix-Sort läuft in O(d(n + N)) Zeit

Lexikographische Ordnung

Lexikographische-Sortierung sortiert eine Sequenz von d-Tupeln in lexikographischer Ordnung, indem d-mal stableSort, je einmal pro Dimension, durchgeführt wird.

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)

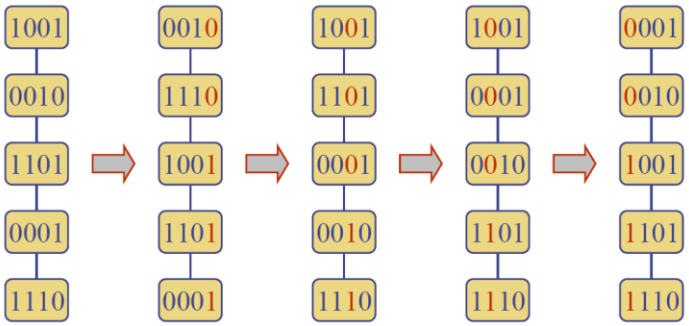
Lexikographische-Sortierung läuft in O(d*T(n)) Zeit, wobei T(n) die Laufzeit von stableSort ist.

Binary Radix-Sort

Gegeben sei eine Sequenz von n b-bit Integers (x = xb − 1 … x1x0). Jedes Element wird als b-Tupel von Integern im Bereich {0,1} dargestellt. Darauf wendet man den Radix-Sort mit N=2 (N ist Wertebereich Key, da Bit entweder 0 oder 1 gibt es nur 2 Möglichkeiten) an

Binary Radix-Sort läuft in O(b*n) Zeit.

Bsp.:



Laufzeitvergleich der Sortier-Algorithmen

Algorithmus	Zeitverhalten	Bemerkungen
selection-sort	$O(n^2)$	▪ langsam ▪ in-place ▪ für kleine Data Sets (< 1K)
insertion-sort	$O(n^2)$	▪ langsam ▪ in-place ▪ für kleine Data Sets (< 1K)
heap-sort	$O(n \log n)$	▪ schnell ▪ in-place ▪ für grosse Data Sets(1K — 1M)
merge-sort	$O(n \log n)$	▪ schnell ▪ sequentieller Datenzugriff ▪ für riesige Data Sets(> 1M)

Pattern Matching

Brute-Force

Der Brute-Force Pattern Matching Algorithmus vergleicht das Pattern P mit dem Text T für jede mögliche Position von P relativ zu T, bis

- Eine Übereinstimmung gefunden wurde
- Alle möglichen Platzierungen des Patterns ausprobiert wurden

Laufzeit

Brute Force benötigt O(nm) Zeit.
Worst Case:

- T = aaaa … ah
- P = aaah
- Treten in Bild- und DNA Analysen auf
- In sprachlichen Texten eher nicht

Pseudo-Code

```
BruteForceMatch(T,P)
    Input Text T der Länge n und Pattern P der Länge m
    Output Startindex eines Substrings von T, wechler mit P übereinstimmt,
oder -1 falls kein solcher Substring existiert

    n = T.length()
    m = P.length()

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

Boyer-Moore

Boyer-Moore basiert auf zwei Heuristiken:

- Looking-Glass
  - Vergleiche P mit einer Subsequenz von T
  - Starte dabei am Ende des Patterns
- Character-Jump
  - 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
    - Ansonsten verschiebe P bis P[0] mit T[i+1]

Boyer-Moore funktioniert ähnlich wie Brute-Force, passt aber die Verschiebung an.
Ist bereits das erste verglichene Symbol (nicht vergessen!! Boyer-Moore beginnt von hinten) ein Symbol das gar nicht im Pattern vorkommt, wird um die Länge des Patterns verschoben:

```
0 1 2 3 4 5 6 7 8 9 ...
a b b a d a b a c b a
b a b a c
        b a b a c
```

Sind die Symbole ungleich, das Textsymbol aber woanders im Pattern vorhanden, wird das Pattern geschoben, bis der erste Vergleich matched:

```
0 1 2 3 4 5 6 7 8 9 ...
a b b a b a b a c b a
    b a b a c
        b a b a c
```

Last Occurrence

Analyse des Patterns P und das Alphabet Σ um die «last-occurence» Funktion L aufzubauen. Diese bildet Σ auf Integers ab.

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

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

Bedeutet: Der Buchstabe im Alphabet bekommt die Zahl der Stelle an welcher er als letztes im Pattern vorkommt. Kommt er gar nicht vor, bekommt er -1

Bsp.:

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

$Pos: 012345$

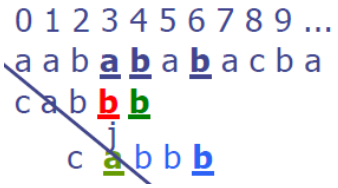
c	a	b	c	d
$L(c)$	4	5	3	-1

Die funktion lässt sich in O(m+s) berrechnen:

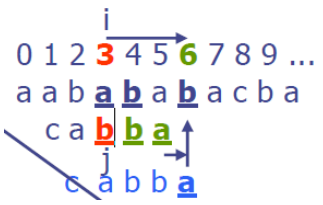
- m = Länge von P
- s = Anzahl Zeichen in Σ

Berechnung der Verschiebung

Fall, Zeichen T[i] kommt im Pattern vor:

- Verschieben bis zum letzten Auftreten des Zeichens T[i] in Pattern P
- i = i + m – (last(T[i]) +1)
- Bsp. m = 5: i = 3 + 5 - (last(a) + 1) = 8 - 2 = 6
c	a	b	c	d
L(c)	1	4	0	–1
  - 

Fall, Zeichen T[i] kommt im Pattern vor, ist aber bereits vorbei. last(x) > current index j (i = index im text, j = index im pattern)

- Pattern wird um eine Stelle nach vorne verschoben
- i = i + m – j
- Bsp. m = 5: i = 3 + 5 – 2= 8 -2 = 6
c	a	b	c	d
L(c)	4	3	0	–1
  - 

Pseudo-Code

```
BoyerMooreMatch(T,P,Σ)

    L = lastOccurenceFunction(P,Σ)
    n = T.length()
    m = P.length()
    i = m-1 //T index
    j = m-1 //P index

    do
        if T[i] = P[j]
            if j = 0
                return i //Match at i
            else
                i = i-1
                j = j-1
        else //character jump
            l = L[T[i]]
            i = i + m - min(j, 1+l)
            j = m - 1
    while i <= n-1 //überprüfung am schluss des loops
return -1 // no match
```

Analyse

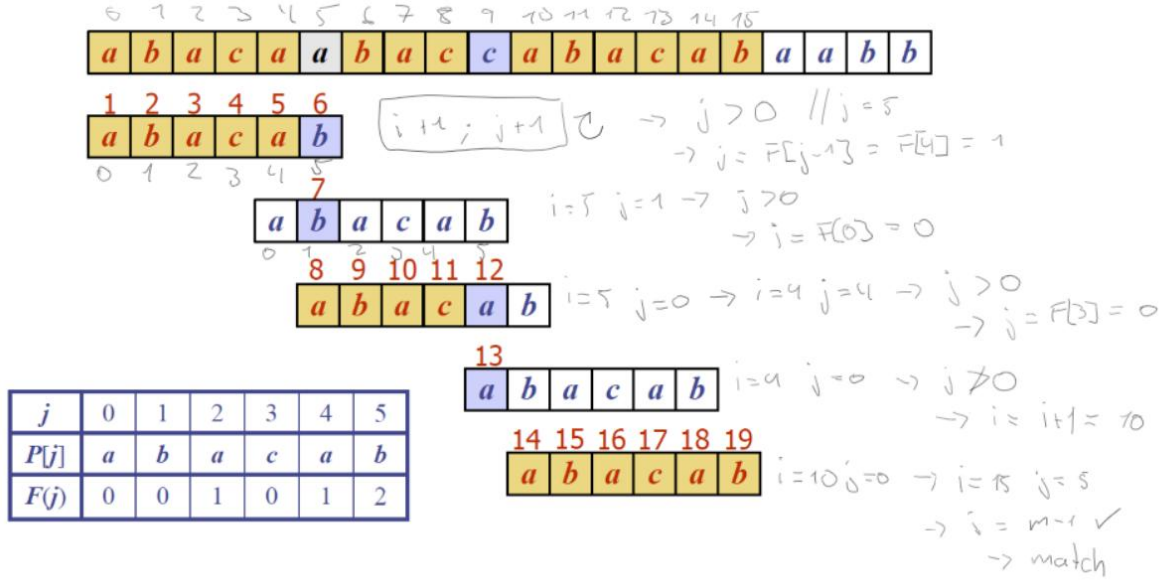
- O (n*m+s)
    - n = Länge des Textes
    - m = Länge des Patterns
    - s = Anzahl Zeichen im Alphabet Σ
- Worst Case:
    - T = aaaa … a
    - P = baaa
- Vor allem bei Bild- und DNA- Sequenzen (kleines Alphabet, viel Wiederholung)

Knuth-Morris-Pratt

Vergleich des Musters gegen Text von links-nach-rechts, Verbesserung bei der Verschiebung des Musters gegenüber Brute-Force.
Bei Nichtübereinstimmung ist das Maximum um welches das Muster verschoben werden kann der längste Präfix von P[0..j] der gleichzeitig Suffix von P[1..j] ist. Dadurch werden redundante vergleiche verhindert.

Pseudo-Code

```
KMPMatch(T,P)
    F = failureFunction(P)
    i = 0
    j = 0
    n = T.length()
    m = p.length()

    while i<n
        if T[i] = P[j]
            if j = m-1
                return i-m+1 //match found
            else //weiter durch Text und Pattern iterieren und vergleichen
                i = i+1
                j = j+1
        else
            if j > 0
                j = F[j-1]
            else
                i = i + 1
return -1 //no match
```



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

Bei jeder Iteration der while-Schleife wird:

- Entweder i um 1 erhöht
- Die Verschiebung nimmt um mindestens 1 zu (F(j-1) < j)

Preprocessing / Fehl-Funktion

F(j) ist definiert als die Grösse des längsten Präfixes von P[0..j] , so dass dieser auch Suffix von P[1..j] ist.

Berechnung

1. Auflistung der Ränder

   a.
   P[0]=a
   P[1]=ab
   P[2]=aba
   P[3]=abaa
   P[4]=abaab
   P[5]=abaaba

2. Aufsuchen der Präfixe, welche auch Suffixe sind

   a.

	Ränder-Länge: 0	1	2	3
P[0]=a	{}			
P[1]=ab	{}			
P[2]=aba	{}	a		
P[3]=abaa	{}	a		
P[4]=abaab	{}		ab	
P[5]=abaaba	{}	a		aba

3. Länge Notieren

   a.

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

Pseudo-Code

```
failureFunction(P)
    F[0] = 0 //erste Zeile der Tabelle immer 0
    i = 1
    j = 0
    m = P.length()
    while i < m
        if P[i] = P[j] //matched j+1 chars
            F[i] = j+1
            i = i+1
            j = j+1
        else if j > 0 // use failure function to shift P
            j = F[j-1]
        else
            F[i] = 0 //kein match
            i = i+1
```

Java-Beispiel:

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

Java-Beispiel:

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

Dynamische Programmierung

Aufteilung eines Problems in Subprobleme nach dem «bottom-up» Prinzip. Also vom kleinsten Problem bis zum Grössten. Vereinfacht die Lösung und verringert die Laufzeit des Problems

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 Problem könnte mit Brute Force gelöst werden, dabei ist die Anzahl der möglichen Varianten aber exponentiell, also 2^n → Laufzeit $O(2^n)$

Aufteilung in Subprobleme:

1. Erstellen der Tabelle, Rucksack mit 8kg Platz:

kg	1	2	3	4	5	6	7	8
wert/kg								
7/3								
4/2								
8/4								
9/5								
3/1								

2. Beginn mit erstem wert/kg für jedes Subproblem grösstmöglicher Wert eintragen
    a. Jedes «Objekt» darf nur einmal verwendet werden

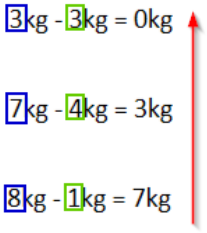
kg	1	2	3	4	5	6	7	8
wert/kg								
7/3	0	0	7	7	7	7	7	7
4/2								
8/4								
9/5								
3/1								

3. Wiederhole für alle Zeilen, mit Inbezugnahme vorheriger Werte

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

4. Auslesen des Resultats
    a. Beginn ganz unten rechts mit dem Grössten wert
    b. Spalte hochgehen, bis Wert sich ändert, letzter Wert kommt in den Sack, Kg von Gesamtgewicht abziehen
    c. Spalte nach links gehen zu verbleibendem Subproblem und wiederholen

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



Longest Common Subsequence

Gegeben sind die beiden Strings X und Y, finde die längste Subsequenz, welche in X sowohl als auch in Y enthalten ist.
Bsp.: ABCDEFG und XZACKDFWGH haben ACDFG als längste gemeinsame Subsequenz.

Auch hier wäre es theoretisch wieder mit Brute Force möglich. Wieder wäre die Laufzeit exponentiell, wenn X die Länge n hat, wäre die Laufzeit 2^n

Algorithmus

Definiere L[i][j] als Länge der längsten gemeinsamen Subsequenz von X[0..i] und Y[0..j].

Erlaube -1 als Index, so dass L[-1][k]= 0 und L[k,-1] = 0 (erste Zeile und Spalte in Matrix immer 0).

L[i][j] wird folgendermassen definiert:

- Wenn $x_i = y_j$ dann L[i][j] = L[i-1][j-1] + 1 (Übereinstimmung, Wert diagonal in Matrix wird um 1 erhöht)
- $x_i \neq y_j$, dann L[i][j] =max{ L[i-1,j], L[i,j-1} (keine Übereinstimmung, der grössere Wert von oberhalb oder links in der Matrix wird übernommen)

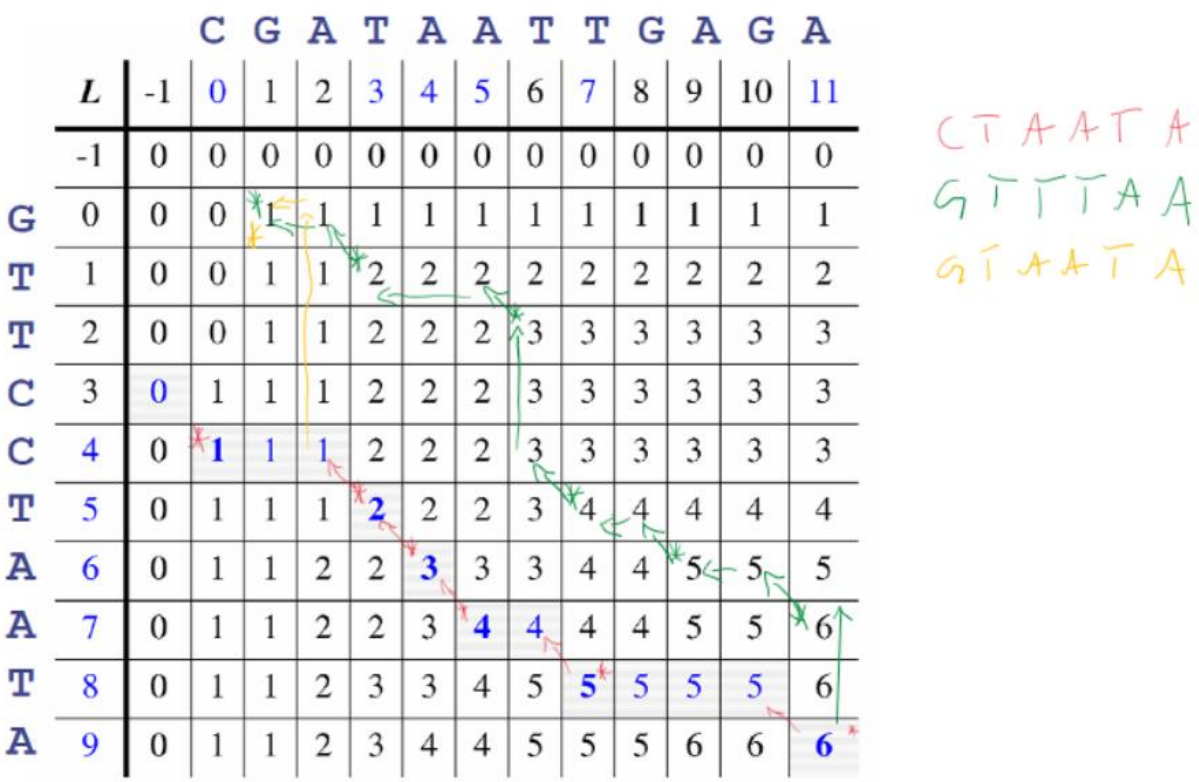
Analyse

Zwei verschachtelte Loops

- äusserer iteriert n mal
- innerer iteriert m mal
- konstanter Aufwand innerhalb jeder Iteration des inneren Loops

Totale Laufzeit: O(nm), Verbesserung gegenüber $O(2^n)$

Auslesen des Resultats



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

Es kann mehrere Lösungspfade geben. Generell gilt aber:

- Beginn ganz unten rechts
- Der Zeile links oder der Spalte nach oben folgen bis ein Match oder Wertänderung
- Bei einem Match diagonal nach oben Links springen
- Entweder nach oben oder nach Links bis wieder Match
- Usw.

Pseudo-Code

```
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] and Y[0..j]

    n = X.length()
    m = Y.length()

    for i = 1 to n-1 do //fill first row with 0
        L[i, -1] = 0
    for j = 0 to m-1 do //fill first column with 0
        L[-1, j] = 0
    for i = 1 to n-1 do
        for j = 0 to m-1 do
            if X[i] = Y[j] then
                L[i][j] = L[i-1][j-1] + 1 //upper-left value of current
position + 1
            else
                L[i][j] = max(L[i-1][j], L[i][j-1]) //take bigger value from
either above or to the left of current position
    return array L
```

Graphen

Ein Graph besteht aus:

- Einem Set Vertices V (Knoten)
- Einem Set Kanten E (Edges)
    - Eine Collection von Vertices-Paaren
- Vertices und Kanten sind Positionen und speichern Elemente

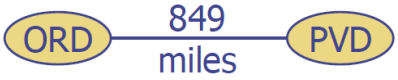
Kanten-Typen

Gerichtete Kanten (Kante mit Pfeil/Richtung)

- Geordnetes Paar von Vertices (u,v)
- Erster Vertex u entspricht dem Ursprung
- Zweiter Vertex v entspricht dem Ziel
- Bspw. ein Flug

Ungerichtete Kanten (ohne Pfeil/Richtung)

- Ungeordnetes Vertices-Paar (u,v)
- Bspw. Flugroute



Gerichteter Graph

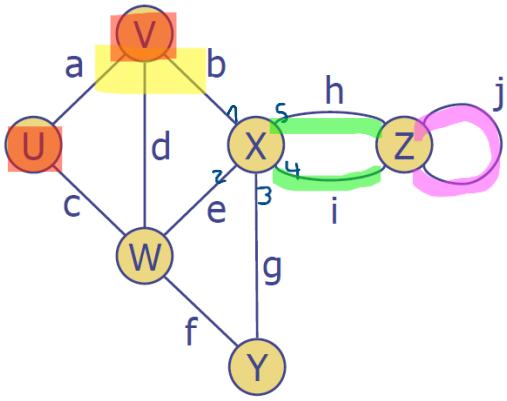
- Alle Kanten sind gerichtet
- Bspw. Flugplan

Ungerichteter Graph

- Alle Kanten sind ungerichtet
- Bspw. Flugrouten-Plan

Terminologie

Kanten



Kanten sind inzident (enden) an einem Vertex

- a, d und b sind inzident in V

Adjazente (benachbarte) Vertices

- U und V sin adjazent

Grad (Degree) eines Vertex: Anzahl inzidenter Kanten

- X besitzt grad 5

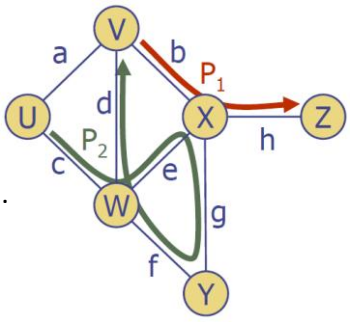
Parallele Kanten

- h und i sind parallele Kanten

Schleife

- j ist eine Schleife

Pfade

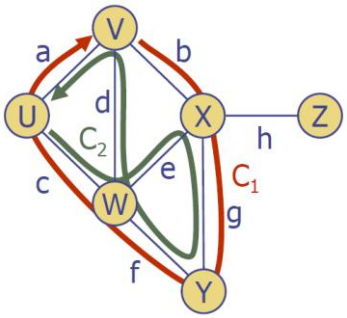


Sequenz von alternierenden Vertices und Kanten.
Beginnt und endet mit einem Vertex.
Jede Kante beginnt und endet an einem ihrer Endpunkte.

Einfacher Pfad:

- ein Pfad, so dass alle seine Vertices und Kanten unterschiedlich sind.
- Bsp:
  - P1 = (V,b,X,h,Z) ist einfach
  - P2 = (U,c,W,e,X,g,Y,f,W,d,V) ist nicht einfach

Zyklen



Sequenz von alternierenden Vertices und Kanten.
Beginnt mit einem Vertex und endet mit einer Kante.

Einfacher Zyklus:

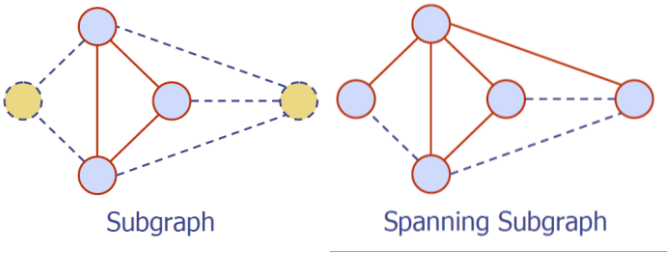
- Ein Zyklus, so dass alle seine Vertices und Kanten unterschiedlich sind
- Bsp.:
  - C1= (V,b,X,g,Y,f,W,c,U,a) ist ein einfacher Zyklus
  - C2= (U,c,W,e,X,g,Y,f,W,d,V,a) ist ein nicht einfacher Zyklus

Subgraphen

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

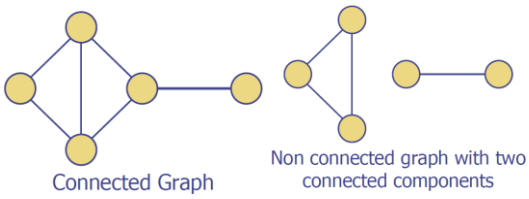
- Die Kanten Von sind eine Teilmenge der Kanten von G
- Die Vertices von S sind eine Teilmenge der Vertices von G

Ein spanning Subgraph A des Graphen G ist ein Subgraph , welcher alle Vertices von G enthält.



Connectivity

Ein Graph heisst verbunden falls zwischen jedem Paar von Vertices ein Pfad existiert.
Eine Verbundene Komponente eines Graphen G ist ein Verbundener Subgraph von G.



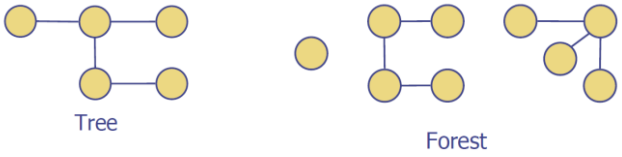
Bäume und Wälder

Ein (freier) Baum ist ein ungerichteter Graph T, so dass

- T verbunden ist
- T keine Zyklen aufweist

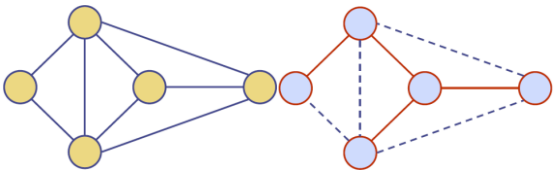
➔ Die Definition eines freien Baums unterscheidet sich von jener eines Wurzelbaums

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



Spanning Trees

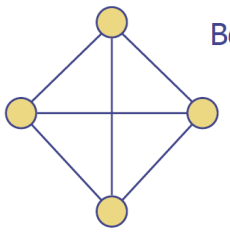
Ein aufspannender Baum eines verbundenen Graphen ist ein aufspannender Subgraph, welcher auch ein Baum ist.
Ein aufspannender Baum ist nicht eindeutig, ausser der Graph, von dem ausgegangen wird, ist ein Baum.
Aufspannende Bäume werden beispielsweise in Kommunikationsnetzwerken eingesetzt.
Ein aufspannender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



Eigenschaften

Notation:

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



Beispiel

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

Eigenschaft 1:

- $\sum_v \deg(v) = 2m$
- Jede Kante wird zweimal gezählt

Eigenschaft 2:

- In einem ungerichteten Graphen ohne Schleifen und ohne Mehrfach-Kanten (parallele Kanten) gilt (dies entspricht einem ungerichteten, einfachen Graphen):
  - $m \leq n\frac{n-1}{2}$
- Beweis:
  - Jeder Vertex besitzt einen Grad von höchstens (n-1)

Methoden

Vertices und Kanten sind Positionen und speichern Elemente

e = Kanten (Edges)
v = Vertex

Zugriffs-Methoden

endVertices(e): gibt ein Array der beiden Vertices an den Enden der Kante e zurück
opposite(v, e): gibt den Vertex gegenüber von v and der Kante e zurück
areAdjacent(v, w): gibt true zurück, wenn v und w benachbart sind
replaceInVertex(v, x): ersetze das Element an Vertex v mit x
replaceInEdge(e, x): ersetze das Element an Kante e mit x

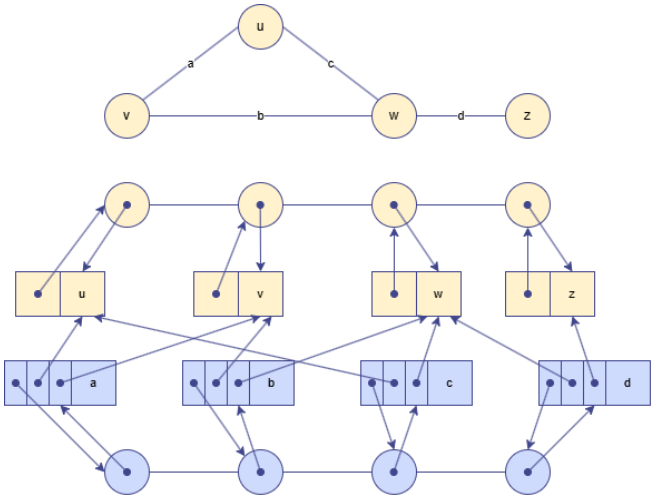
Update-Methoden

insertVertex(o): füge einen Vertex ein, welcher element o speichert; gibt den neuen Vertex zurück
insertEdge(v, w, o): füge eine Kante (v,w) ein, welche Element o speichert; gibt die neue Kante zurück
removeVertex(v): entferne Vertex v (zusammen mit seinen anliegenden Kanten)
removeEdge(e): entferne Kante e

Iterator-Methoden

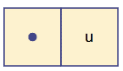
incidentEdges(v): Alle Kanten anliegen an Vertex v
vertices(): Alle Vertices im Graph
edges(): alle Kanten im Graph

Kanten-Listen Struktur



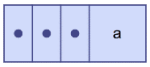
Vertex / Knoten Objekt

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



Kanten Objekt

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



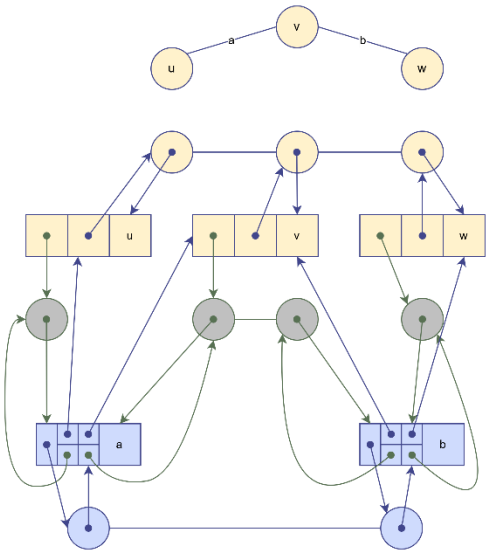
Vertex-Sequenz

Sequenz der Vertex-Objekte (Linked-List)

Kanten-Sequenz

Sequenz der Kanten-Objekte (Linked-List)

Adjazenz-Listen Struktur



Erweiterung der Kanten-Listen Struktur.

Inzidenz-Sequenz

Sequenz der Positionen auf Kantenobjekte der inzidenten Kanten, vereinfacht ausgedrückt:

- Eine Liste von Kantenreferenzen die zu genau einem Vertex gehören

Ein Vertex Objekt verweist auf eine «Linked-List» welche dann wiederum Verweise auf eine Kante enthält.

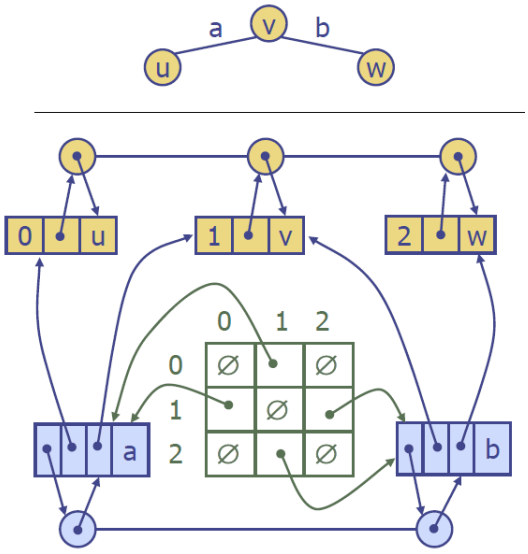
Erweiterte Kanten-Objekte

Referenziert auf die Assoziierten Positionen in der Inzidenzsequenz der Endvertices, vereinfacht:

- Zeigt auf die Stelle in der Inzidenz-Sequenz, auf welche auch die Vertices, welche der Kante anliegen zeigen.

Verweist auch direkt auf die verbundenen Vertices.

Adjazenz-Listen Struktur

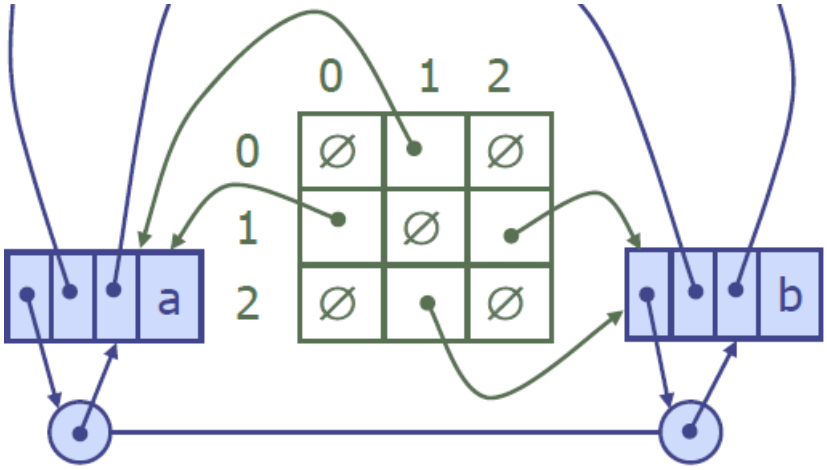


Erweiterung der Kantenlisten Struktur

Erweiterte Vertex-Objekte

Die Vertex-Objekte bekommen einen Index Key, dieser wird in der Matrix verwendet um den Vertex zu identifizieren.

2D-Array Adjazenz-Array



Adjazente benachbarte Vertices enthalten in der Matrix einen Eintrag, welcher auf das entsprechende Kantenobjekt zeigt. Im Bild sind 1 und 0 (u und v) adjazent, verbunden werden sie über die Kante a. Für nicht adjazente Vertices, wird *null* in die Matrix geschrieben.

Performance der Strukturen

▪ 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

Depth-First Search

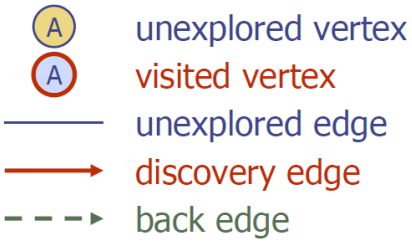
Eine DFS-Traversierung eines Graphen G

- besucht alle Vertices und Kanten von G
- bestimmt, ob G verbunden ist
- berechnet / bestimmt die verbundenen Komponenten von G
- berechnet einen aufspannenden Wald von G

Auf einem Graphen mit n Vertices und m Kanten benötigt ein DFS O(n+m) Zeit.

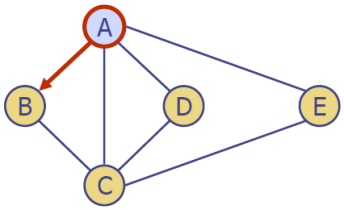
Algorithmus

Zuerst werden alle Vertices und Kanten auf «unexplored» gesetzt.



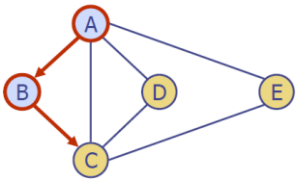
Beginne bei Startvertex (A):

- Setze Vertex auf «visited»
- Setze erste Kante der Funktion incidentEdges() auf «discovery»
    - incidentEdges() gibt Kanten in absteigender Reihenfolge, abhängig von gegenüber Zurück

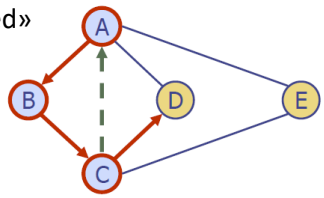


Rekursiver Aufruf der Funktion mit Vertex B:

- Kante AB bereits «discovery», wird daher nicht überprüft
- Weiter zu Kante BC
    - Hier wieder derselbe Ablauf: Setze Kante auf «discovery» und rufe die Funktion mit dem «opposite» auf (C)

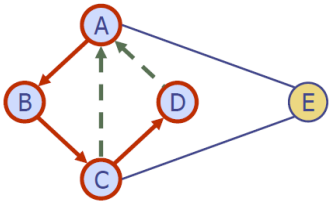


Aufruf bei Vertex C



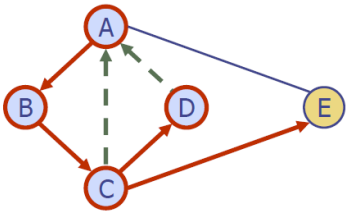
- Erste Kante CA, ist zwar «unexplored», «opposite» Vertex A ist aber «visited»
    - Setze Kante auf «back»
- Kante CB, bereits «discovery»
    - Skip
- Kante CD, weder «discovery» noch ist D «visited»
    - Setze Kante auf «discovery» und rufe die Funktion mit dem «opposite» auf (D)

Aufruf bei Vertex D



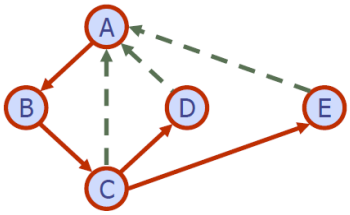
- Kante DA, «opposite» Vertex A ist «visited»
    - Setze Kante auf «back»
- Kante CD bereits «discovery»
    - Ignorieren
- Funktion Terminiert für D, zurück zu C

Aufruf bei Vertex C



- Nächste Kante ist CE, Kante sowie Vertex sind «unexplored»
    - Setze Kante auf «discovery»
    - Rekursiver Aufruf der funktion mit Vertex E

Aufruf bei Vertex E



- Selbe Situation wie bei D, entweder Kante «discovery» oder opposite «visited»
    - Ignoriere «discovery»
    - Setze Kante zu «visited» auf «back»

Algorithmus geht nun alle Vertices rekursiv noch durch

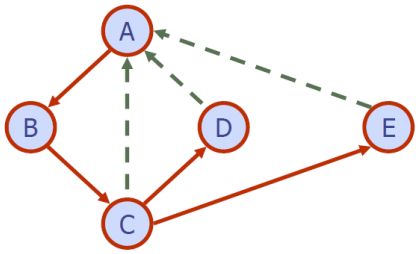
- Merkt alle Kanten entweder «discovery» oder «back»
    - Werden ignoriert
- Terminiert, nachdem alle Kanten von Vertex A «überprüft wurden»

Pseudo-Code

```
DFS (G)
    Input Graph G
    Output labeling of the edges of G as discovery edges and back edges

    for all u in G.vertices() //set all vertices to UNEXPLORED
        setLabel(u, UNEXPLORED)
    for all e in G.edges() //set all edges to UNEXPLORED
        setLabel(u, UNEXPLORED)
    for all v in G.vertices()
        if getLabel(v) = UNEXPLORED
            DFS(G,v)

DFS (G, v)
    Input graph G and start vertex v
    Output labeling og the edges of G in the connected component of v as discovery and back
edges

    setLabel(v, Visited) //set label of current vertex
    for all e in G.incidentEdges(v) //return all edges connected to vertex v in ascending
order according to opposite vertex
        if getLabel(e) = UNEXPLORED
            w = opposite(v, e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                DFS(G, w) //recursive function call with the "next" vertex as "startpoint"
            else
                setLabel(e, BACK) //if opposite vertex is already visited
```

Eigenschaften

- DFS(G, v) besucht alle Vertices 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



Analyse

- Setzen/Lesen eines Vertex-/Kanten-Labels benötigt O(1) Zeit
- Jeder Vertex wird zweimal markiert
  - Zuerst als Unexplored
  - Dann als Visited
- Jede Kante wird zweimal markiert:
  - Zuerst Unexplored
  - Dann Discovery oder Back
- incidentEdges() wird pro Vertex einmal aufgerufen
- DFS benötigt O(n+m) Zeit, sofern der Graph mithilfe seiner Adjazenzenlisten-Struktur dargestellt wird
  - Es gilt: $\sum_v \deg(v) = 2m$

Erweiterungen

DFS kann man auch erweitern, um andere Graphenprobleme zu lösen

Pfade finden

Mithilfe von DFS kann ein Pfad zwischen zwei gegebenen Vertices u und z gefunden werden.

- DFS(G, u) wird mit u als Startvertex aufgerufen
- Mithilfe des Stack S merkt man sich den Pfad zwischen Startvertex und dem aktuellen Vertex
- Sobald der Zielvertex Z gefunden wurde, wird der Pfad mithilfe des Stacks ausgegeben

```
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)  Pop Vertex
        pathDFS(G, w, z)
        S.pop()
      else
        setLabel(e, BACK)
  S.pop()  Pop Kante zu Vertex
```

Zyklen finden

Genau wie für Pfade kann DFS modifiziert werden um Zyklen zu finden.

- Auch wird sich der Pfad mit dem Stack gemerkt
- Sobald man auf eine Back-Edge (v, w) trifft, wird der Zyklus als teil des Stacks ausgegeben
    - Vom obersten Stack Element 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()
```

Breadth-First Search

Eine BFS-Traversierung eines Graphen G

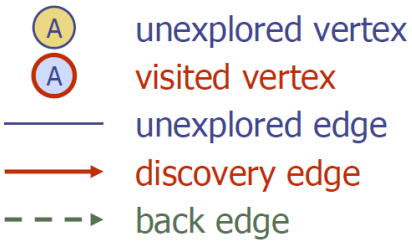
- besucht alle Vertices und Kanten von G
- bestimmt, ob G verbunden ist
- berechnet / bestimmt die verbundenen Komponenten von G
- berechnet einen aufspannenden Wald von G

Auf einem Graphen mit n Vertices und m Kanten benötigt ein DFS O(n+m) Zeit.

Algorithmus

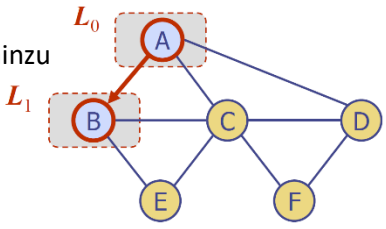
Der Algorithmus benutzt einen Mechanismus, um "Labels" auf Kanten und Vertizes zu setzen.

Zuerst werden alle Vertices und Kanten auf «unexplored» gesetzt.

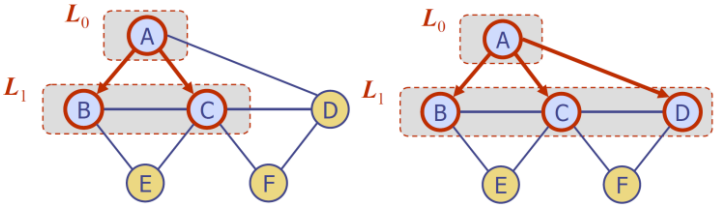


Beginne bei Startvertex (A):

- erstelle ein neues Level L_0
- füge Vertex dem neuen Level hinzu
- setze Vertex auf «visited»
- erstelle weiteres Level L_1
- führe für alle Elemente in L_0 incidentEdges() aus
    - füge den «opposite» des ersten Resultats dem Level L_1 hinzu
        - sofern label unexplored
    - setze Kante auf «discovery» und vertex auf «visited»

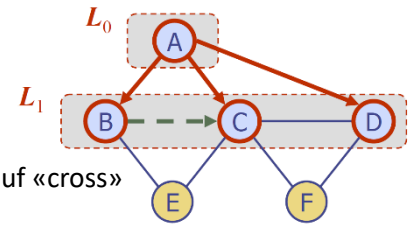


Wiederhole für alle Resultate der Funktion incidentEdges()

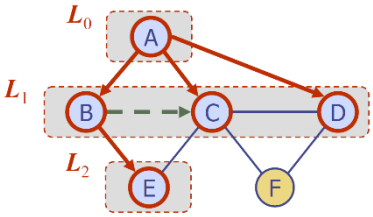


Gehe weiter zu L_1 und wiederhole

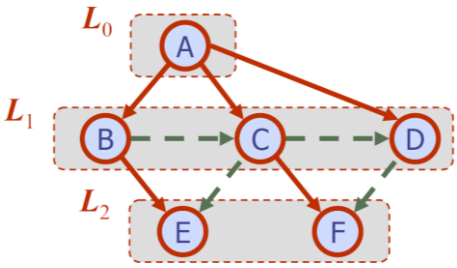
- erstelle L_2
- hole incidentEdges() für Vertices in L_1
- ist gegenüberliegender Vertex bereits «visited», setze Kante auf «cross»



- ist gegenüberliegender Vertex noch «unexplored»
  - setze Kante auf «discover»
  - setze Vertex auf «visited»
  - füge Vertex L2 hinzu



Wiederhole für weitere Level. Abbruch, sobald ein Level leer ist



Pseudo-Code

```
BFS(G)
    Input Graph G
    Output labeling of the edges and partition of the vertices of G

    for all u in G.vertices()
        setLabel(u, UNEXPLORED)
    for all e in G.edges()
        setLabel(e, UNEXPLORED)
    for all v in G.vertices()
        if getLabel(v) = UNEXPLORED
            BFS(G,v)

BFS(G,s)
    L0 = new empty sequence
    L0.insertLast(s)
    setLabel(s, VISITED)
    i = 0
    while Li.isEmpty()
        L(i+1) = new empty sequence
        for all v in 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)
                        L(i+1).insertLast(w)
                    else
                        setLabel(e, CROSS)
        i = i+1
```

Eigenschaften

1. BFS(G, s) besucht alle Vertices und Kanten in G
2. Die Discovery-Kanten von BFS(G, s) bilden einen aufspannenden Baum Ts von Gs
3. Für jeden Vertex v in Li gilt:
    a. der Pfad in Ts von s nach v besitzt i Kanten.
    b. jeder Pfad von s nach v in Gs besitzt mindestens i Kanten.

Analyse

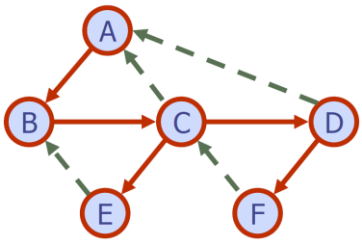
- Setzen/Lesen eines Vertex-/Kanten-Labels benötigt O(1) Zeit.
- Jeder Vertex wird zweifach markiert
    - Einmal als Unexplored
    - Einmal als Visited
- Jede Kante wird zweifach markiert
    - Einmal als Unexplored
    - Einmal als Discovery oder Cross
- Jeder Vertex wird einmal in die Sequenz L_i eingetragen
- incidentEdges() wird einmal pro Vertex aufgerufen
- BFS benötigt O(n+m) Zeit, sofern der Graph mithilfe seiner Adjazenzliste dargestellt wird
    - es gilt $\Sigma v \deg(v) = 2m$

Applikationen

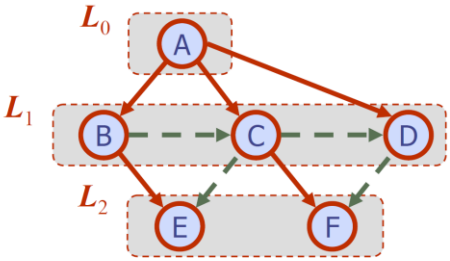
- Wir können die BFS Traversierung eines Graphen G benutzen, um folgende Probleme in O(n + m) Zeit zu lösen:
  - Bestimmen der verbundenen Komponenten von G
  - Bestimmen eines aufspannenden Waldes von G
  - Bestimmen eines einfachen Zyklus in G
    - Oder bestimmen, ob G ein Wald ist
  - Bei zwei gegebenen Vertices von G:
    - Finden eines Pfades in G zwischen den beiden Vertices mit minimaler Anzahl Kanten oder bestimmen, ob ein solcher Pfad existiert

DFS vs. BFS

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



DFS



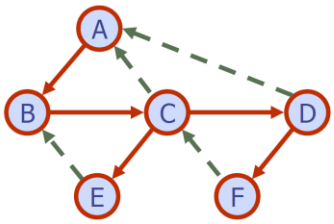
BFS

Back edge (v,w)
(Rückwärtskante)

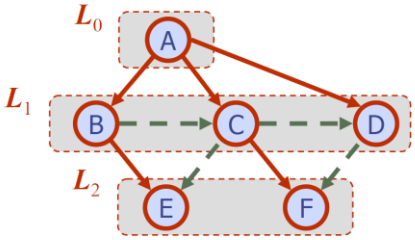
- w ist ein Vorfahre von v im Baum der Suchkanten

Cross edge (v,w)
(Kreuzungskante)

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



DFS

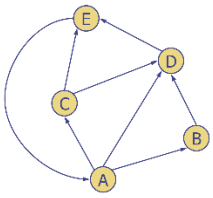


BFS

Gerichtete Graphen

Ein Graph, dessen Kanten alle gerichtet sind.
Anwendungen:

- Einbahnstrassen
- Flüge
- Task Scheduling



Eigenschaften

Ein Digraph ist: G = (V,E) derart, dass:

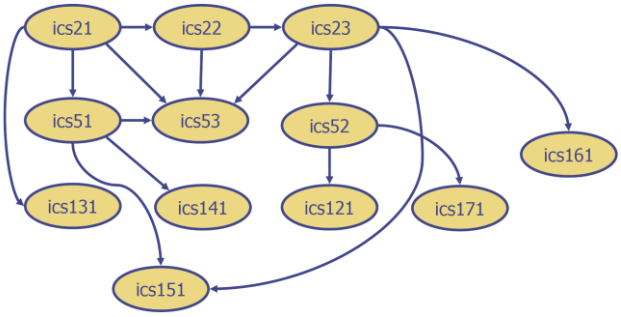
- Jede Kante nur in eine Richtung:
  - Kante (a,b):
    geht von a nach b
    nicht aber von b nach a

Wenn G einfach ist: $m \leq n(n-1)$
Wenn In- und Out-Kanten in separaten Adjazenz-Listen sind:

- Laufzeit für Zugriff auf In- und Out-Kanten proportional zur Grösse der Listen

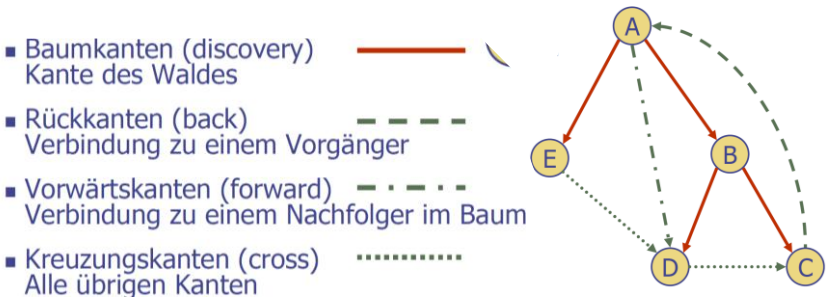
Anwendungen

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



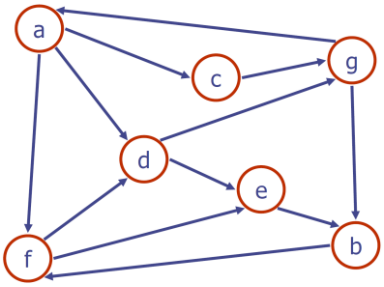
Gerichtete Tiefensuche

Bei der Tiefensuche für Digraphen, werden die Kanten nur entlang ihrer Richtung traversiert. Dabei gibt es folgende Kantentypen:



Connectivity

Strong Connectivity: Jeder Vertex kann alle anderen Vertices erreichen



Algorithmus

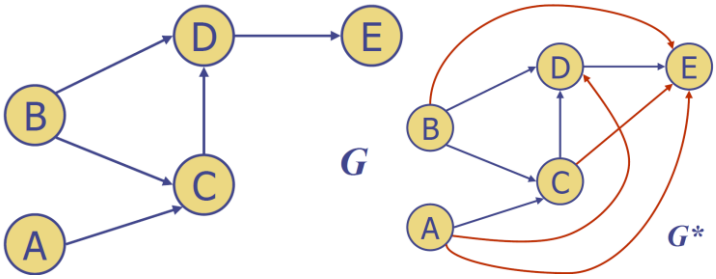
- Wähle einen Vertex v in G
- Führe Tiefensuche von v in G durch
    - Gibt es einen nicht besuchten Vertex w:
      return false //NOK
- G' sei G mit umgekehrten Kanten-Richtungen
- Führe Tiefensuche von v in G' aus
    - Gibt es einen nicht besuchten Vertex w:
      return false //NOK
    - Sonst:
      return true //OK
- Laufzeit: O(n+m)

Transitiver Abschluss

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

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

Der transitive Abschluss stellt die gesamte Erreichbarkeitsinformation über einen Digraph zur Verfügung



Floyd-Warshall

Nummeriert die Vertices von G als v, …, vn und berechnet eine Serie von Digraphen G0, …, Gn

- G0 = G
- Gk hat eine gerichtete Kante (vi, vj), falls G einen gerichteten Pfad von vi nach vj mit Zwischenvertex aus der Menge {v1, …,vk} hat

Es gilt: Gn = G*

In der Phase k, Digraph Gk ist aus Gk-1 berechnet

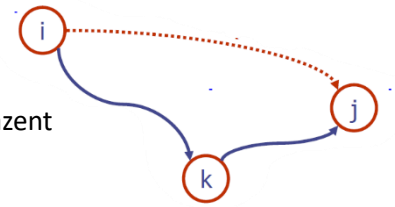
Laufzeit: $O(n^3)$, Annahme: areAdjacent(from, to) mit O(1) (d.h Adjazenz-Matrix)

Code:

```
FloydWarshall(G)
    Input digraph G
    Output transitive closure G* of G
    i = 1
    for all v = G.vertices()
        denote v as vi
        i = i+1
    G0 = G
    for k = 1 to n do
        Gk = G(k-1)
        for i = 1 to n (i != k)
            for j = 1 to n (j != i, k) do
                if G(k-1).areAdjacent(vi,vk) and G(k-1).areAdjacent(vk,vj)
                    if !Gk.areAdjacent(vi,vj)
                        Gk.insertDirectedEdge(vi,vj,k)
    return Gn
```

Ablauf:

- i → k → j
- setze k auf v1, i auf v2 und j auf v3
- überprüfe i (v2) und k(v1) adjazent AND k(v1) und j(v3) adjazent
    - nein: erhöhe j um 1
        - j auf v4
    - ja: überprüfe ob i (v2) und j (v3) bereits adjazent
        - nein: setze Kante zwischen i (v2) und j (v3)
        - ja: weiter im loop



Topologische Sortierung

Algorithm *topologicalDFS(G)*
  Input dag *G*
  Output topological ordering of *G*
  $n \leftarrow$ *G.numVertices*()
  for all $u \in$ *G.vertices*()
    setLabel(*u*, *UNEXPLORED*)
  for all $e \in$ *G.edges*()
    setLabel(*e*, *UNEXPLORED*)
  for all $v \in$ *G.vertices*()
    if *getLabel*(*v*) = *UNEXPLORED*
      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 $e \in$ *G.outgoingEdges*(*v*)
    if *getLabel*(*e*) = *UNEXPLORED*
      $w \leftarrow$ *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*
  $n \leftarrow n - 1$

- O(n+m) time.

Shortest Path Trees

Gewichtete Graphen haben Kanten mit assoziierten nummerischen Werten, das Gewicht oder Kosten. Ziel ist es in einem gewichteten Graphen, den kürzesten Pfad zu finden. Die länge/distanz des Pfades ist dabei die Summe der Gewichte aller Kanten.

Eigenschaften

Ein Teilweg eines Kürzesten Weges ist selbst auch ein kürzester Weg

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

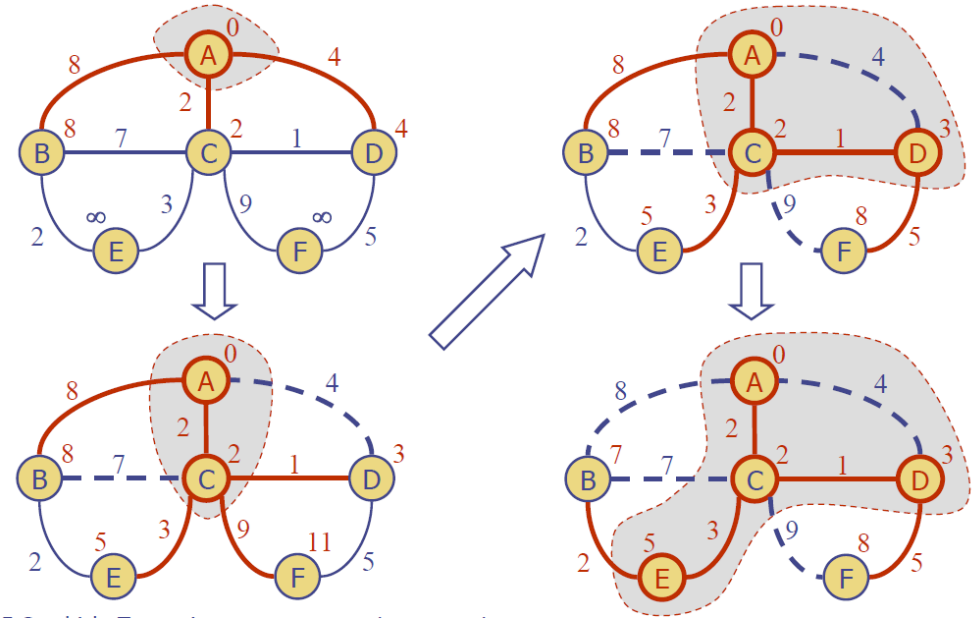
Dijkstra

Berechnet die Distanzen zu allen Vertices von einem Start-Vertex s aus. Dabei werden folgende Annahmen getroffen:

- Der Graph ist verbunden
- Die Kanten sind ungerichtet
- Die Kantengewichte sind nicht negativ

Ablauf:

- Vermerke alle Vertices mit ∞ / Integer.MAX_VALUE
- Beginne bei Startvertex
    - Erstelle eine Wolke und füge Startvertex hinzu
- Überprüfe alle Kanten von Startvertex ausgehend
    - Vermerke Kosten von Start zu Vertex
- Gehe zu opposite mit geringsten Kosten
    - Füge Vertex der Wolke hinzu
- Überprüfe von aktuellem Vertex wieder alle Kanten
    - Falls Wert des aktuellen Vertex + Kosten der Kante < Wert des adjazenten Vertex:
        - Relaxation: ersetze Wert des Adjazenten Vertex mit Rechnung von oben
- Füge Kante mit geringstem Wert der Wolke hinzu
    - Vertex muss adjazent zur Wolke sein, nicht aber zum zuletzt überprüfen Vertex
- Weiter zu neu hinzugefügtem Vertex und wiederhole



Pseudo-Code

- Adaptierbare Priority Queue speichert Vertices ausserhalb der Wolke (umgekehrt zur Theorie)
  - Key: Distanz
  - Element: Vertex
- Locator-basierte Methoden:
  - Insert(k,e) gibt einen Locator zurück
  - replaceKey(l,k) ändert den Schlüssel eines Eintrags
- Speichern zwei Eigenschaften mit jedem Vertex
  - Distanz d(v)
  - Locator der Priority Queue

```
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)
```

Analyse

incidentEdges() wird für jeden Vertex einmal aufgerufen

Distanz- und Locator-Label eines Vertex z wird O(deg(z) mal gesetzt oder gelesen.
Setzen/Lesen eines Labes: O(1)

Jeder Vertex wird einmal in die Priority Queue eingefügt und einmal gelöscht, wobei jedes Einfügen oder Löschen O(log n) Zeit benötigt

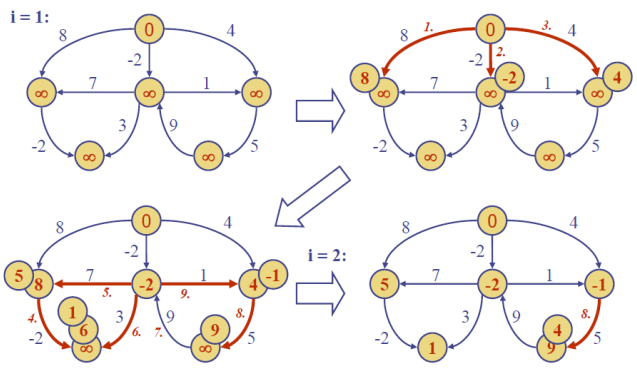
Der Schlüssel eines Vertex w in der Priority Queue wird maximal deg(w) mal geändert, wobei jede Änderung O(log n) Zeit benötigt

Dijkstra's Algorithmus läuft in O((n + m) log n) Zeit, vorausgesetzt dass der Graph mit einer Adjazenz-Listen-Struktur implementiert ist.
(Σ_v deg(v) = 2m)

Bellman-Ford

- Funktioniert auch mit negativ-gewichteten Kanten
- Voraussetzung: gerichtete Kanten und keine negativ-gewichtete Schlaufen
- Iteration *i* findet alle kürzesten Pfade welche *i* Kanten benutzen
- Laufzeit: O(nm)

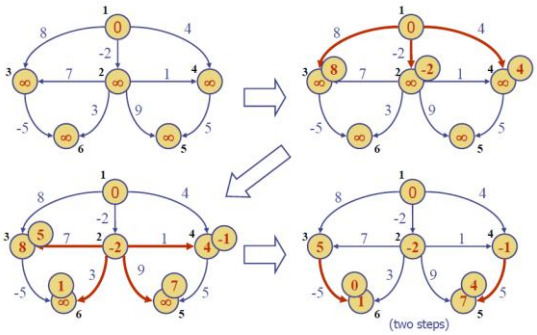
```
Algorithm BellmanFord(G, s)
  for all v ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
  for i ← 1 to n-1 do
    for each e ∈ G.edges()
      { relax edge e }
      u ← G.origin(e)
      z ← G.opposite(u, e)
      r ← getDistance(u) + weight(e)
      if r < getDistance(z)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



DAG

- Funktioniert auch mit negativ-gewichteten Kanten
- Benutzt eine topologische Reihenfolge
- Benutzt keine ausgefallenen Datenstrukturen
- Ist viel schneller als Dijkstra's Algorithmus
- 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)
```

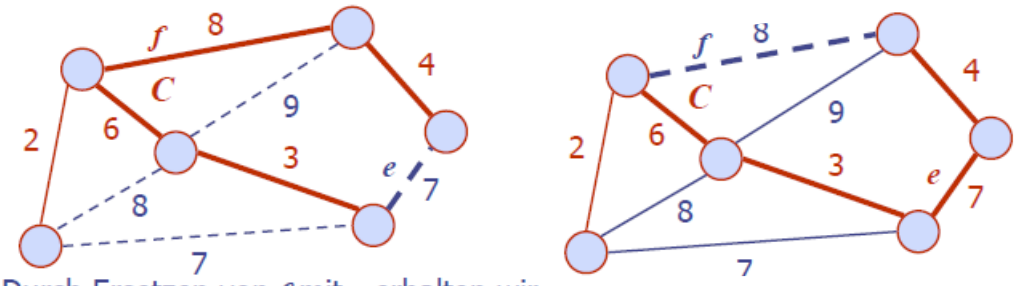


Minimum Spanning Trees

Ein minimum spanning tree ist ein aufspannender Baum eines Graphen mit minimalem totalen Kantengewicht, der alle Vertizes beinhaltet (Ein Graph, indem von einem Vertex jeder andere Vertex mit dem kleinstmöglichen Gewicht erreicht werden kann). Anwendung: Kommunikationsnetzwerke, Transportnetzwerke.

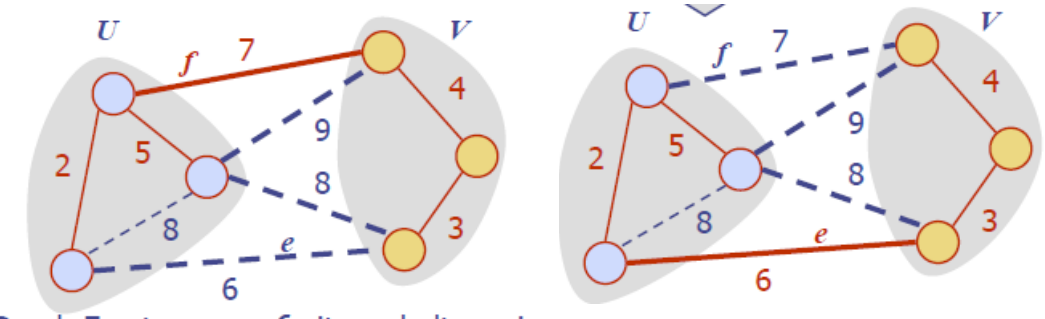
Eigenschaften

Schlaufen Eigenschaft: Sei T ein minimal aufspannender Baum eines gewichteten Graphen G. e sei eine Kante von G, welche nicht in T ist und C die Schlaufe, die durch e mit T entsteht.
Für jede Kante f von C: weigth(f) ≤ weight(e), sonst kann ein besserer aufspannender Baum erreicht werden in dem e mit f ersetzt wird.



Durch Ersetzen von *f* mit *e* erhalten wir einen besseren aufspannenden Baum

Aufteilungs-Eigenschaft: Sei G aufgeteilt in zwei Vertex Teilmengen U und V, e eine Kante mit dem kleinsten Gewicht zwischen den Partitionen.
Es existiert ein minimal aufspannender Baum von G der Kante e beinhaltet



Durch Ersetzen von *f* mit *e* erhalten wir einen besseren aufspannenden Baum

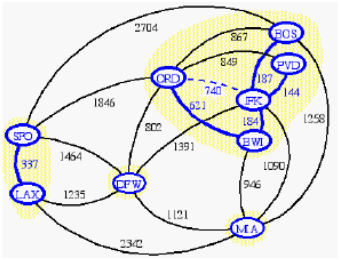
Kruskal Algorithmus

- Eine Priority Queue speichert die Kanten ausserhalb der Wolke
  - Key: Gewicht
  - Element: Kante
- Zum Schluss des Algorithmus
  - Wir haben eine Wolke welche den MST umfasst
  - Ein Baum T welcher unser MST ist

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*

Datenstruktur:

- Der Algorithmus behält einen Wald von Bäumen
- Eine Kante ist akzeptiert, wenn sie verschiedene Bäume verbindet
- Wir brauchen eine Datenstruktur welche eine **Partition** verwaltet.
  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



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. Jede Element wird also höchstens log n mal verarbeitet

- Eine partition-basierte Version von Kruskal's Algorithmus führt Wolken-Vereinigungen mit `union`'s und Tests mit `find`'s aus.

Algorithm Kruskal(*G*):
  Input: Ein gewichteter Graph *G*.
  Output: Ein MST *T* für *G*.
P sei eine Partition der Vertizes von *G*, wobei jeder Vertex ein Set für sich bildet
Q sei eine Priority Queue, welche die Kanten von *G* nach Gewichtung sortiert
T sei ein ursprünglich leerer Baum
while *Q* is nicht leer **do**
  (*u*,*v*) ← *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)

Ablauf:

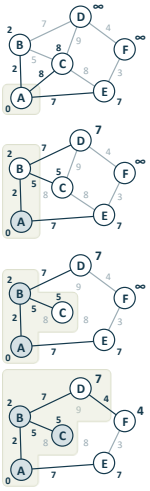
- Jeder Vertex bekommt seine eigene Wolke
- Starte bei Kante mit geringstem Wert
- Vereine beide Wolken der Vertices, welche mit der Kante verbunden sind.
- Gehe zur Kante mit nächst Grösseren Wert
    - Verbindet Sie zwei Wolken
        - Führe Wolken Zusammen
    - Sind Vertices bereits in einer Wolke, «streiche» Kante
- Wiederhole bis n-1 Kanten der Wolke hinzugefügt werden
    - n = Anzahl Vertices im gesamten Graph

Prim-Jarnik's Algorithmus

Gleich wie Dijkstra's Algorithmus. Jedoch wird nicht mehr die Distanz zum Startwert, sondern zur Wolke gespeichert. Zu jedem Vertex werden drei Eigenschaften gespeichert: Die Distanz, die Elternkante MST und den Locator der Priority Queue. Ein Hash-Set bildet die Wolke ab.

- 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)
```



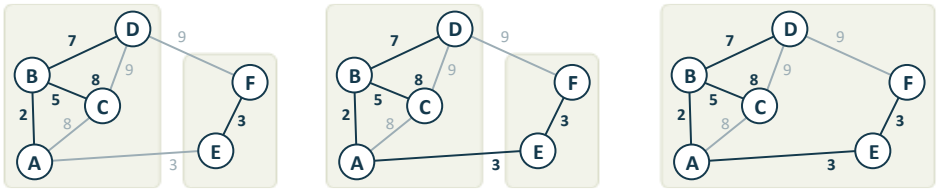
- Graph Operationen
    - Methode `incidentEdges` wird für jeden Vertex einmal aufgerufen
- Label Operationen
    - Wir setzen/lesen das Distanz-, Parent- und Locator-Label eines Vertex z insgesamt $O(\deg(z))$ mal
    - Setzen/Lesen eines Labels benötigt $O(1)$ Zeit
- Priority Queue Operationen
    - 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
- Prim-Jarnik's Algorithmus läuft in $O((n + m) \log n)$ Zeit, vorausgesetzt dass der Graph ist mit einer Adjazenz-Listen Struktur implementiert ist
    - Wir erinnern uns, dass $\sum_v \deg(v) = 2m$

Boruvka's Algorithmus

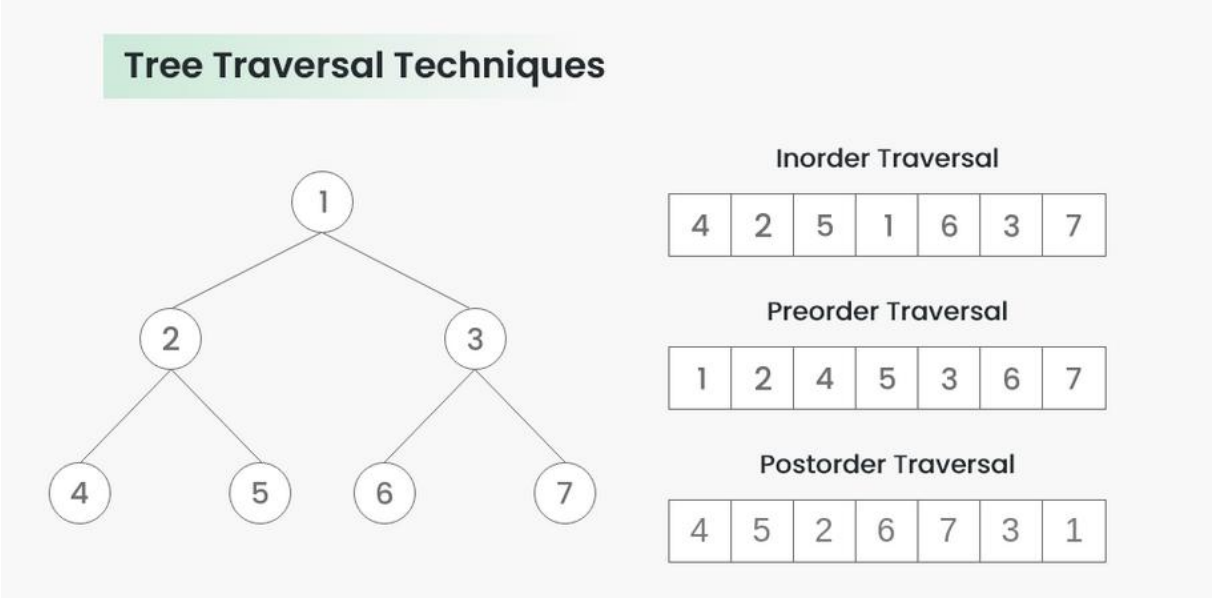
Bildet wie der Kruskal's Algorithmus viele Wolken aufs Mal. Jede Iteration der while-Schlaufe halbiert die Anzahl der verbundenen Komponenten in T. Pro Wolke gibt es eine Priority Queue.

Kleinste Kante jedes Vertex bestimmen, dann diese so verbundenen Vetizes in gemeinsame Wolke setzen.

```
Algorithm BoruvkaMST(G)
 T ← V {nur die Vertizes von G}
 while T < n-1 edges do
  for each verbundene Komponente C in T do
    if e is not in T then // Edge e is the smallest Edge in C to another
component in T
     add e to T
  return T
```



Traversierungsarten



xO-Notationsliste

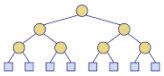
Multimap / Suchtabelle

find(k)	insert(k)	remove(k)
$O(\log n)$	$O(n)$	$O(n)$

Binary Search Tree

Speicherplatz	find(k)	insert(k)	remove(k)
$O(n)$	$O(h)$	$O(h)$	$O(h)$

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



AVL Tree

Restrukturierung	find(k)	insert(k)	delete(k)
$O(1)$	$O(\log n)$	$O(\log n)$ *wegen find()* *eventuelle Restrukturierung baumaufwärts ist* $O(1)$	$O(\log n)$ *wegen find()* *eventuelle Restrukturierung baumaufwärts sind* $O(log\ n)$

Splay Tree

$\boldsymbol{O(h)}$

Durchschnitt: $\mathbf{O(\log n)}$*, für oft besuchte Knoten schneller, da diese immer näher an die Root rücken. Worst-Case: Höhe des Baumes ist* $\mathbf{n}$*, somit* $\mathbf{O(n)}$ *Rotationen, jede mit* $\mathbf{O(1)}$

Merge Sort

Höhe des Trees	Gesamt-Aufwand aller Knoten einer Tiefe i	Totale Laufzeit
$O(\log n)$	$O(n)$ *Aufteilung und Mischen von* 2^i *Sequenzen der Grösse* $n/2^i$*,* 2^{i+1} *rekursive Aufrufe*	$O(n * log\ n)$

Quick Sort

Partitio-nierung	Höhe im Optimalfall	Erwartete Laufzeit (durchschnittlich)	Worst-Case Laufzeit
$O(n)$	$O(\log n)$ *weil Halbierung auf jeder Stufe*	$O(n * \log n)$ *Tiefe im Optimalfall:* $O(log\ n)$*, Gesamtaufwand für alle Knoten einer Tiefe:* $O(n)$	$O(n^2)$ *wenn das Pivot das Minimum oder Maximum-Element ist. L oder G hat dann die Länge* $n-1$*, das andere* 0*. Die Laufzeit ist proportional zur Summe.* $\sum_{i=0}^{n} i = \frac{n^2+n}{2}$ *Anzahl Vergleiche:* $\frac{(n-1)n}{2}$

Ein Pivot ist gut, wenn die Länge von L & G beide kleiner als ¾ der Inputlänge sind

Bucket Sort

Laufzeit einzelner Phasen	Erwartete Laufzeit	Worst-Case Laufzeit
$O(n)$ *Phase 1: Buckets füllen* $O(N+n)$ *Phase 2: Elemente aus allen Buckets herausnehmen*	$O(N+n)$ *n: Anzahl Elemente, N: Anzahl Buckets*	$O(n^2)$ *Wenn alle Elemente im selben Bucket landen*

Brute Force

$\boldsymbol{O(n * m)}$

n:Textlänge, m: Patternlänge

Boyer Moore

Laufzeit	Last Occurrence Funktion
$O(n * (m + s))$ *n-mal die Last-Occurence-Funktion*	$O(m + s)$ m ist die Länge von P und s die Anzahl Zeichen im Alphabet Σ

KMP

Laufzeit	Failure Function
$O(m + n)$ *Bei jeder Iteration der while-schleife wird entweder* i *um eines erhöht oder die Verschiebung* i − j *nimmt um mindestens* 1 *zu. Somit ergeben sich maximal* 2n *iterationen in der while-Schleife.*	$O(m)$ m ist die Länge von P Bei jeder Iteration der while-schleife wird entweder i um eines erhöht oder die Verschiebung $i - j$ nimmt um mindestens 1 zu. Somit ergeben sich maximal $2m$ Iterationen in der while-Schleife.

Standard-Trie

Speicherplatz	find(k)	insert(k)	remove(k)
$O(n)$	$O(dm)$	$O(dm)$	$O(dm)$

n = totale Länge der Strings in S, m = Länge des String-Parameters der Operation, d = Grösse des Alphabets

Trie kompakte Repräsentation (Array)

$O(s)$ Speicherplatz

s = Anzahl Strings im Array

Suffix-Trie

Speicherplatz	Pattern Matching	Erstellen
$O(n)$	$O(d * m)$	$O(n)$

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

Dijkstra

Laufzeit	Setzen / Lesen Labels	Einfügen / Löschen Priority Queue	Schlüssel ändern
$O((n + m) * \log n)$	$O(dez(z) * O(1)$ *Anzahl * Zeit*	$O(\log n)$	$\deg(w) * O(\log n)$

Bellman-Ford

$O(n * m)$

DAG-basierter Algorithmus

$O(n+m)$

Kruskal

Laufzeit	find	Union(u,v)	Max Anzahl Verarbeitungen pro Element
$O(m * \log n)$ *Partitionsbasiert*	$O(1)$	$\min(n_u, n_V)$ n_u *und* n_v *sind die Grössen der Sets, die* u *und* v *beinhalten*	$\log n$

Prim Jarnik

Laufzeit	Setzen / Lesen Labels	Einfügen / Löschen Priority Queue	Schlüssel ändern
$O((n+m) * \log n)$ *Mit Adjazenz-Listen Struktur*	$O(dez(z) * O(1)$ *Anzahl * Zeit*	$O(\log n)$	$\deg(w) * O(\log n)$

Das ABC für die Besitzer eines $O(n^2)$-Gehirns (aka die Autoren dieser Zusammenfassung)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z