Eigenschaften von Algorithmen:

- **Determiniertheit**: Identische Eingaben führen stets zu identischen Ergebnissen.
- **Determinismus**: Ablauf des Verfahrens ist an jedem Punkt fest vorgeschrieben (keine Wahlfreiheit).
- **Terminierung**: Für jede Eingabe liegt das Ergebnis nach endlich vielen Schritten vor.
- **Effizienz**: «Wirtschaftlichkeit» des Aufwands relativ zu einem vorgegebenen Massstab (z.B. Laufzeit, Speicherplatzverbrauch).

GGT (Euklid)

ggT(a, b) =

1. a = b → a, resp. b, ist der ggT
2. a > b → a = a - b
3. a < b → b = b – a

```
while (a != b) {
   if (a > b) a = a - b;
   else b = b - a;
}
return a;
```

Fakultät

Programm mit Rekursion:

```
int fak(int n) {
   if (n == 0) return 1;
   else
      return n * fak(n-1);
}
```

$$n! = \begin{cases} 1 & \text{falls } n=0 \\ n*(n-1)! & \text{sonst} \end{cases}$$

Fakultätsberechnung mit Rekursion:

```
int fak(int n) {
     if (n == 0) return 1;
     else return n * fak(n-1);
}
```

Verankerung

Rekursiver Aufruf

Programm mit Iteration:

```
int fak(int n) {
   if (n == 0) return 1;
   else {
      int res = n;
      while (n > 1) {
         n--; res = n* res;
      }
      return res;
   }
}
```

Türme von Hanoi O(2^n)

```
void hanoi (int n, char from, char to, char help) {
   if (n > 0) {
      // bewege Stapel n-1 von from auf help
      hanoi(n-1, from, help, to);
      // bewege von from nach to
      System.out.println("bewege " + from + " nach " + to);
      // bewege Stapel n-1 von help auf to
      hanoi(n-1, help, to, from);
   }
}
```

```
main {
   hanoi (3, 'A', 'B', 'C');
}
```

```
bewege A nach B
bewege A nach C
bewege B nach C
bewege A nach B
bewege C nach A
bewege C nach B
bewege A nach B
```

Fibonacci-Zahlen O(2^n)

$$fib(n) = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ fib(n-1) + fib(n-2) & \text{sonst} \end{cases}$$

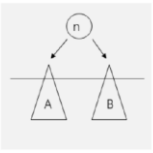
n	0	1	2	3	4	5	6	7	8	9	10	11	12	..
fn	0	1	1	2	3	5	8	13	21	34	55	89	144	..

```
public int fib(int n) {
   if (n == 0) return 0;
   else if (n == 1) return 1;
   else return fib(n-1) + fib(n-2);
}
```

Binärbäume

Die möglichen Arten von Traversierung (beim Binärbaum) sind:

1. Preorder: n, A, B
2. Inorder: A, n, B
3. Postorder: A, B, n
4. Levelorder: n, a_0, b_0, a_1, a_2, b_1, b_2, ...



Levelorder: von oben nach unten und links nach rechts

Preorder

1. Besuche die Wurzel und verarbeite die Daten.
2. Traversiere den linken Teilbaum (in Preorder).
3. Traversiere den rechten Teilbaum (in Preorder).

```
class TreeTraversal<T> implements Traversal<T> {
   private void preorder(TreeNode<T> node, Visitor<T> visitor) {
      if (node != null) {
         visitor.visit(node.element);
         preorder(node.left, visitor);
         preorder(node.right, visitor);
      }
   }
}
```

```
...
treeTraversal.preorder(ordersTree.root, myVisitor);
...
```

mit Stack:
pop(), visit(), push(right), push(left)

Postorder

1. Traversiere den linken Teilbaum (in Postorder).
2. Traversiere den rechten Teilbaum (in Postorder).
3. Besuche die Wurzel und verarbeite die Daten.

```
class TreeTraversal<T> implements Traversal<T> {
   private void postorder(TreeNode<T> node, Visitor<T> visitor) {
      if (node != null) {
         postorder(node.left, visitor);
         postorder(node.right, visitor);
         visitor.visit(node.element);
      }
   }
}
```

- Zuerst werden die Nachfolger abgearbeitet und dann der Knoten selbst (von unten nach oben).

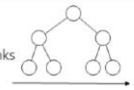
Mit Queue:
dequeue(), visit(), enqueue(left), enqueue(right)

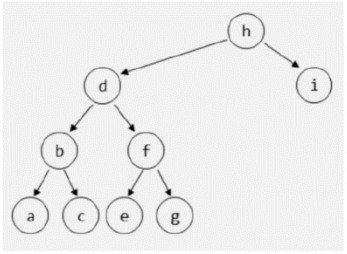
Inorder

1. Traversiere den linken Teilbaum (in Inorder).
2. Besuche die Wurzel und verarbeite die Daten.
3. Traversiere den rechten Teilbaum (in Inorder).

```
class TreeTraversal<T> implements Traversal<T> {
   private void inorder(TreeNode<T> node, Visitor<T> visitor) {
      if (node != null) {
         inorder(node.left, visitor);
         visitor.visit(node.element);
         inorder(node.right, visitor);
      }
   }
}
```

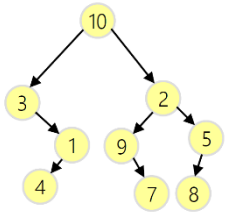
Der Baum wird quasi von links nach rechts abgearbeitet:





1. Preorder-Traversierung: 10, 3, 1, 4, 2, 9, 7, 5, 8
2. Inorder-Traversierung: 3, 4, 1, 10, 9, 7, 2, 8, 5

Preorder: h, d, b, a, c, f, e, g, i
Inorder: a, b, c, d, e, f, g, h, i
Postorder: a, c, b, e, g, f, d, i, h
Levelorder: h, d, i, b, f, a, c, e, g



Via Preorder "Hauptknoten" definieren,
Via Inorder Children rechts oder links davon

3 durch Preorder links von 10,
4,1 nach 3 rechts, da Inorder,
wo die 1 steht in Inorder. Links davon steht die 4. Somit ist die 4 linker Child

Dann prüfen was kommt in Preorder, zuerst 1 dann prüfen

Sortierte Binärbäume

suchen

```
public Object search(TreeNode<T> node, T  x) {
   if (node == null) return node;
   else if (x.compareTo(node.element) == 0)
      return node;
   else if (x.compareTo(node.element) <= 0)
      return search(node.left,x);
   else
      return search(node.right,x);
}
```

Einfügen

```
private TreeNode<T> insertAt(TreeNode<T> node, T x) {
   if (node == null)
      return new TreeNode(x);
   else if (x.compareTo(element) <= 0)
      node.left = insertAt(node.left, x);
   else
      node.right = insertAt(node.right, x);
    return node;
}
```

Löschen

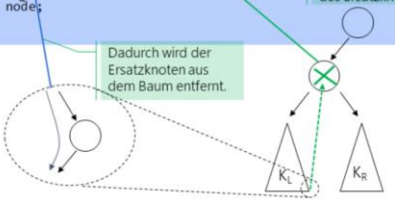
```
private TreeNode<T> removeAt(TreeNode<T> node, T x) {
   if (x.compareTo(node.element) == 0) { // delete this node
      if (node.left == null) {
         node = node.right; // no left subtree -> case 1 or 2
      } else if (node.right == null) {
         node = node.left; // no right subtree -> case 2
      } else {
         // two subtrees -> case 3
         node.left = findRepAt(node.left, node); // node.left is root of left subtree
      }
   } else if (x.compareTo(node.element) < 0) {
      node.left = removeAt(node.left, x);  // search in left subtree
   } else {
      node.right = removeAt(node.right, x); // search in right subtree
   }
   return node;
}
```

Übernächste Folie

```
private TreeNode<T> findRepAt(TreeNode<T> node, TreeNode<T> rep) {
   if (node.right == null) {
      // node is the rightmost node, the node that should be replaced gets its element
      rep.element = node.element;
      // remove rightmost node of left subtree (return value is the 'new' node)
      node = node.left;
   } else {
      // more nodes on the right side of left subtree
      node.right = findRepAt(node.right, rep);
   }
   return node;
}
```

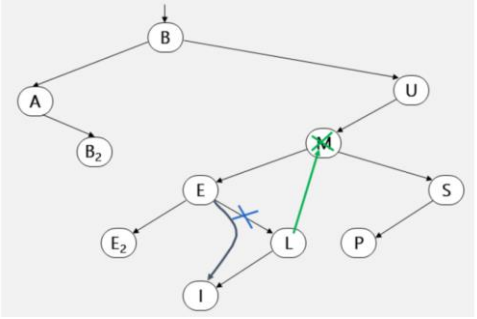
Der zu löschende Knoten erhält den Wert des Ersatzknotens.

Dadurch wird der Ersatzknoten aus dem Baum entfernt.



Sucht den Ersatzknoten und 'löscht' den zu löschenden Knoten.

47 von 51



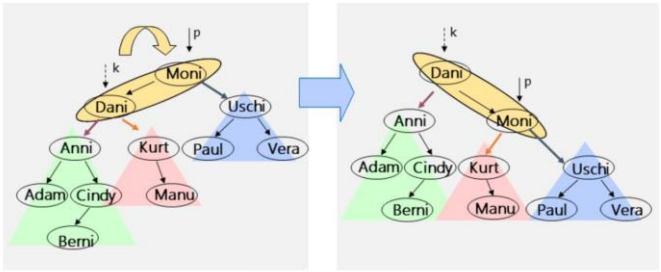
Ablauf:

1. Inhalt des Knotens M mit Inhalt des Knotens von L ersetzen.
2. `right` von E mit `left` von L ersetzen.

40 von 51

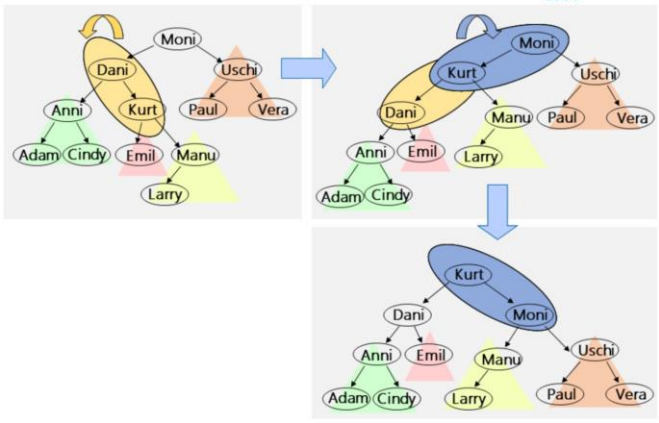
AVL-Bäume

Einzelrotation

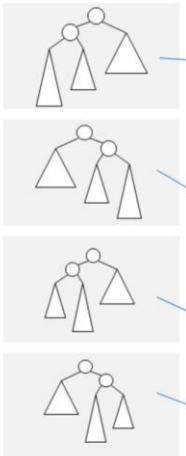



```
Node rotateR(Node p) {
    Node k = p.l;
    p.l = k.r;
    k.r = p;
    return k;
}
```

Doppelrotation



Rotation Methoden



```
private static Node rotateR(Node k2) {
    Node k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    k2.height = Math.max(height(k2.left), height(k2.right)) + 1;
    k1.height = Math.max(height(k1.left), k2.height) + 1;
    return k1;
}

private static Node rotateL(Node k1) {
    Node k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;
    k1.height = Math.max(height(k1.left), height(k1.right)) + 1;
    k2.height = Math.max(height(k2.right), k1.height) + 1;
    return k2;
}

private static Node rotateLR(Node k3) {
    k3.left = rotateL(k3.left);
    return rotateR(k3);
}

private static Node rotateRL(Node k1) {
    k1.right = rotateR(k1.right);
    return rotateL(k1);
}
```

Linksrotation linker Teilbaum.

Rechtsrotation rechter Teilbaum.

Einfügen:

```
private TreeNode insertAt(TreeNode p, T element) {
    if (p == null) {
        p new TreeNode<T>(element);
        p.height = 1;
        return p;
    } else {
        int c = element.compareTo((T) p.getValue());
        if (c == 0) {
            p.values.add(element);
        } else if (c < 0) {
            p.left = insertAt(p.left, element);
        } else if (c > 0) {
            p.right = insertAt(p.right, element);
        }
    }
    return balance(p);
}
```

Gleiche Werte dürfen nicht einfach eingefügt werden.

Wird bei jedem «Rücksprung» ausgeführt.

Balancieren:

```
private TreeNode<T> balance(TreeNode<T> p) {
    if (p == null) return null;
    if (height(p.left) - height(p.right) == 2) {
        if (height(p.left.left) > height(p.left.right)) {
            p = rotateR(p);
        } else {
            p = rotateLR(p);
        }
    } else if (height(p.right) - height(p.left) == 2) {
        if (height(p.right.right) > height(p.right.left)) {
            p = rotateL(p);
        } else {
            p = rotateRL(p);
        }
    }
    p.height = Math.max(height(p.left), height(p.right)) + 1;
    return p;
}
```

Passt die Höhe wieder an.

23 von 41

Adjazenz-Liste (Für Dijkstra und A-*)

Interface!

```
public interface Graph<N, E> {
   // füge Knoten mit Namen name hinzu, tue nichts, falls Knoten schon existiert
   public N addNode (String name);

   // finde den Knoten anhand seines Namens
   public N findNode(String name);

   // Iterator über alle Knoten des Graphen
   public Iterable<N> getNodes();

   // füge gerichtete und gewichtete Kante hinzu
   public void addEdge(String source, String dest, double weight) throws Throwable;
}
```

Generisches Interface, welches Knoten vom Typ N und Kanten vom Typ E verwaltet.

Generische Klasse des Knotens, welche Kanten vom Typ E verwaltet.

```
public class GraphNode<E> {
   protected String name;    // Name des Knoten
   protected List<E> edges;  // Kanten (allenfalls mit Attributen)

   public GraphNode() { edges  = new LinkedList<E>( ); }
   public GraphNode(String name) { this(); this.name = name; }
   public String getName() { return name; }
   public void setName(String name) { this.name = name; }
   public Iterable<E> getEdges() { return edges; }
   public void addEdge(E edge) { edges.add(edge); }
}
```

Iterable, nicht Iterator.

Generische Klasse der Kante, welche Knoten vom Typ N verwendet.

```
public class Edge<N> {
   protected N dest;  // Zielknoten der Kante
   protected double weight;  // Kantengewicht

   public Edge(N dest, double weight) { this.dest = dest; this.weight = weight; }
   public void setDest(N node) { this.dest = node; }
   public N getDest() {return dest; }
   public void setWeight(double w) { this.weight = w; }
   double getWeight() { return weight; }
}
```

```
public class AdjListGraph<N extends Node, E extends Edge> implements Graph<N, E> {
   private final List<N> nodes = new LinkedList<N>();
   private final Class nodeClazz; private final Class edgeClazz;

   public AdjListGraph(Class nodeClazz, Class edgeClazz) {
      this.nodeClazz = nodeClazz;
      this.edgeClazz = edgeClazz;
   }
   // füge Knoten hinzu, gebe alten zurück falls Knoten schon existiert
   public N addNode(String name) {
      N node = findNode(name);
      if (node == null) {
         node = (N) nodeClazz.getConstructor(new Class[]{}).newInstance();
         node.setName(name);
         nodes.add(node);
      }
      return node;
   }
...
```

Klassen der generischen Typen

Erzeuge Instanz

newInstance wird hier gebraucht, weil der Typ der Klasse erst zur Laufzeit zugewiesen wird. Somit kann nicht mit new gearbeitet werden.

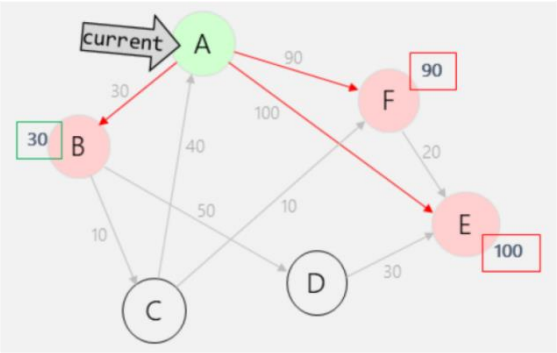
Das Beispiel zeigt nicht die vollständige Implementation des Interfaces

Graph:

Tiefensuche	Breitensuche

`void depthFirstSearch()`
`  s = new Stack();`
`  mark startNode;`
`  s.push(startNode)`
`  while (!s.empty()) {`
`  currentNode = s.pop()`
`  print currentNode`
`  for all nodes n adjacent to currentNode {`
`  if (!(marked(n)) {`
`  mark n`
`  s.push(n)`
`  }`
`  }`
`  }`
`}`
	`void breadthFirstSearch()`
	`  q = new Queue()`
	`  mark startNode`
	`  q.enqueue(startNode)`
	`  while (!q.empty()) {`
	`  currentNode = q.dequeue()`
	`  print currentNode`
	`  for all nodes n adjacent to currentNode {`
	`  if (!(marked(n)) {`
	`  mark n`
	`  q.enqueue(n)`
	`  }`
	`  }`
	`  }`
	`}`

Dijkstra



Teilt die Knoten in 3 Gruppen auf

- besuchte Knoten (kleinste Distanz bekannt)
- benachbart zu allen bereits besuchten Knoten
- unbesehene Knoten (der Rest)

Solange nicht alle Knoten besucht wurden (grün sind):

1. Berechne für alle benachbarten Knoten des aktuell besuchten Knotens (current) die neuen Gewichte (rote Pfeile).
2. Suche unter allen benachbarten Knoten (nicht nur jene des aktuellen Knotens) denjenigen, dessen Pfad zum Startknoten das kleinste Gewicht (= kürzeste Distanz) hat (grüner Rahmen um Zahl).
3. Besuche diesen (neuer aktueller Knoten und neuer Pfad zu diesem Knoten).

Soll nur der minimale Pfad für einen bestimmten Knoten gesucht werden, kann die Suche abgebrochen werden, sobald dieser «current» wird.

Umsetzten mittel PriorityQueue

```
for all nodes n in G {
   n.mark = black; // Alle Knoten noch unbesehen
   n.dist = inf;   // Distanz zu Beginn unendlich
   n.prev = null;  // Bester Vorgängerknoten in Richtung Start
}
start.dist = 0; start.mark = red;
current = start;
```

Z.B. mittels Priority-Queue implementieren

```
for all nodes in RED {
   current = findNodeWithSmallestDist(); // suche den besten roten Knoten
   if (current == goal) return; // kürzester Pfad gefunden -> Abbruch
   current.mark = green;
   for all neighbour in successors(current) {
      if (neighbour.mark != green) { // dieser Knoten wurde nicht besucht
         neighbour.mark = red;
         dist = current.dist + edge(current, neighbour);
         if (dist < neighbour.dist) {
            neighbour.dist = dist;
            neighbour.prev = current;
         }
      }
   }
}
```

Gierige Algorithmen (Greedy)

- Sie zeichnen sich dadurch aus, dass sie einen Folgezustand auswählen, der zum Zeitpunkt der Wahl den grössten Gewinn bzw. das beste Ergebnis verspricht, berechnet durch eine (lokale) Bewertungsfunktion.
- sind oft schnell: z.B. Dijkstra Algorithmus
- können aber in lokalen Maxima/Minima stecken bleiben. Lösung: z.B. stochastische Suchverfahren wie z.B. Simulated Annealing (später)

Topologische Sortierung

```
for all nodes n in G {
   for all s in successors(n) {
      s.incoming += 1;
   }
}
while G not empty {
   for all n in G {
      if (n.incoming == 0) {
         println(n.name);
         G -= n;
         for all s in successors(n) {
            s.incoming -= 1;
         }
      }
   }
}
```

Traveling Salesman Problem

- Es ist relativ einfach eine Lösung im Beispiel zu finden. Ist das der kürzeste Weg?
- Aber: manchmal gibt es überhaupt keine Lösung. Beispiel: Wenn mehr als eine Stadt nur über einen Weg erreichbar ist.
- Ob es der kürzeste Weg ist, lässt sich nur durch Bestimmen sämtlicher möglicher Wege zeigen → O(n!). Der Aufwand wächst faktoriell (pro Memoria: 50! > 3·1064).

Das folgende Greedy-Algorithmus führt zu einer Näherungslösung:

1. Die Kanten werden nach ihren Kosten sortiert.
2. Wähle billigste Kante unter folgenden Bedingungen (ungültige Kanten aus Liste entfernen):
   - Es darf kein Zyklus entstehen (eventuell am Ende/Rundreise)
   - Kein Knoten darf mit mehr als zwei Kanten verbunden sein
- Laufzeit liegt bei O(n2 log n2) – wir verzichten auf die Herleitung.
- Das Verfahren führt nicht immer zu einer optimalen Lösung. Trotzdem wird es in der Praxis erfolgreich eingesetzt.

Backtracking

```
boolean search (Node currentNode) {
   mark currentNode;
   if  currentNode == goal return true;
   else {
      for all nodes n adjacent to currentNode {
         if (!(marked(n)) {
            if (search(n)) return true;
         }
      }
   }
   unmark currentNode;
   return false;
}
```

Falls das Labyrinth Zyklen enthält, wird damit ein «im Kreis gehen» verhindert.

Dt. anschliessend

Jeden anschliessenden Knoten ein Mal überprüfen.

Beim Zurückgehen müssen die Knoten wieder «freigegeben» werden.

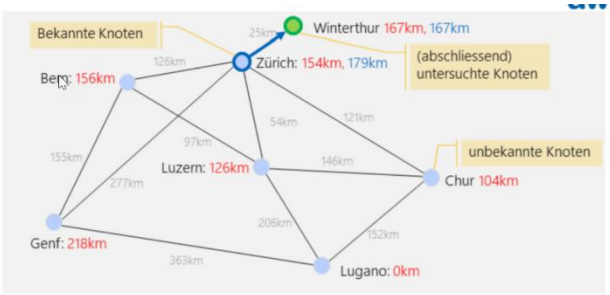
Rucksack-Problem)(2^n)

```
void teste (Gegenstand k) {
   teste (k + 1) // ohne Gegenstand k
   falls Gegenstand k noch Platz
      füge Element k zu der Menge hinzu
      falls neues Maximum speichere das
      teste (k + 1) // mit Gegenstand k
      nehme Element k aus der Menge weg
   }
}
```

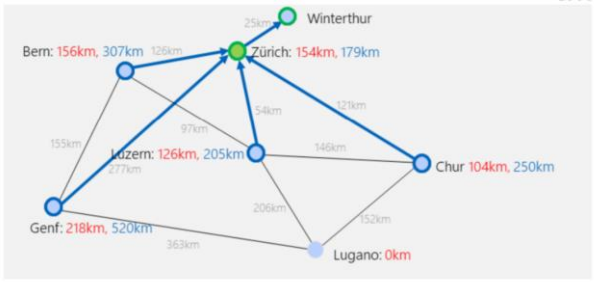
Teste alle Varianten ohne Gegenstand k.

Teste alle Varianten mit Gegenstand k.

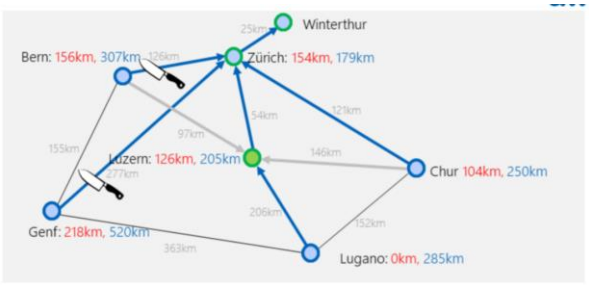
A*-Algorithmus



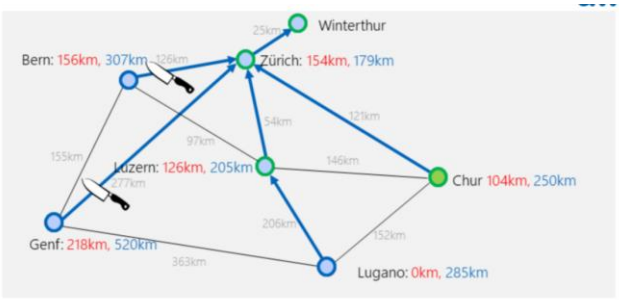
- Wir betrachten zunächst alle nicht untersuchten Nachbarnoten von Winterthur: das ist nur Zürich.
- Für Zürich berechnen wir die minimal erwarteten Kosten: 25km + 154km = 179km, das ist der f-Wert. Da es nur ein Knoten ist, bleiben diese Kosten aber ohne weitere Bedeutung.



- Wir betrachten nun alle nicht untersuchten Nachbarknoten von Zürich und berechnen deren f-Werte (Bern, Genf, Luzern und Chur, wobei z.B. Chur: 25 + 121 + 104 = 250).
- Wir wählen den vielversprechendsten, nicht untersuchten Knoten aus (f-Werte): Luzern mit 205km und untersuchen dessen nicht untersuchten Nachbarknoten Bern, Chur und Lugano.



- Chur wird nicht angepasst, da der neue Weg (f-Wert) länger ist: 25 + 54 + 146 + 104 = 329 > 285.
- Auch für Bern keine Anpassung: 25 + 54 + 97 + 156 = 332 > 307
- Lugano erhält den f-Wert von 25 + 54 + 206 = 285km, bisher der kürzeste Weg.
- Der Weg von Zürich via Bern und Genf kann mittels Pruning entfernt werden, da diese bereits schlechter sind als die bisherige Lösung (520 > 307 > 285).



- Im nächsten Schritt wird nun Chur als bester Kandidat untersucht. Dafür müssen die noch nicht untersuchten Nachfolgeknoten betrachtet werden: nur Lugano.
- Es ergibt sich kein besser f-Wert für Lugano 25 + 121 + 152 = 298 > 285, es muss nichts angepasst werden.
- Der nächstbessere Kandidat ist nun Lugano → die Suche ist abgeschlossen

Binäres Suchen im sortierten Array O(log(n))

```
static int binary(int[] a, int s) {
   int l = -1;
   int r = a.length;
   int m = (l + r) / 2;
   // Invariante && l == -1 && r == a.length
   while (l != r && a[m] != s) {
      if (a[m] < s) l = m;
      else r = m;
      m = (l + r) / 2;
   }
   // Invariante && (l == r || a[m] == s)
   return (a[m] == s)?m:-1;
}
```

l und r n
ausserha
daher is
zu Begin

Suchen in zwei Listen

```
static int indexOf(String[] a, String[] b) {
   int i = 0, j = 0;
   // Invariante && i == 0 && j == 0
   while (!a[i].equals(b[j]) && (i < a.length-1 || j < b.length-1)) {
      int c = a[i].compareTo(b[j]);
      if (c < 0 || j == b.length-1 ) i++;
      else if (c > 0 || i == a.length-1) j++;
   }
   // Invariante && (i == a.length-1 && j == b.length-1) || (a[i] == b[j])
   if (a[i].equals(b[j])) return i; else return -1;
}
```

Schleife wird verlassen wenn diese Bedingung nicht mehr → Negation der Bedingung g am Schluss.

Hashing

```
public class Hashtable {
   final int MAX = 97;
   final int INVALID = Integer.MINVALUE;
   int[] keys = new int[MAX];
   int[] vals = new int[MAX];

   private void init()
   {
      for (int i = 0; i < MAX; i++) {
         key[i] = INVALID;
      }
   }

   private int h(int key) {
      return key % MAX;
   }
```

Hash-Funktion

```
   public void put(int key, int val) {
      int h = h(key);
      if (keys[h] == INVALID) {
         keys[h] = key;
         vals[h] = val;
      }
      else {/* COLLISION */}
   }

   public int get(int key) {
      int h = h(key);
      if (keys[h] == key) {
         return vals[h];
      }
      else return INVALID;
   }
}
```

Überprüfe ob Feld frei.
Speichere Wert.
Schlüssel vorhanden?
Hole Wert.

Lineares Sondieren

```
int find(Object x ) {
    int currentPos = hash(x);

    while (array[ currentPos ] != null &&
       !array[currentPos].element.equals( x ) ) {
       currentPos = (currentPos  + 1) % array.length;
    }
    return currentPos;
}
```

Die Suche muss so lange fortgesetzt werden, bis das Element gefunden wurde oder die Zelle leer ist.

Quadratisches Sondieren

```
int findPos( Object x ) {
    int collisionNum = 0;
    int currentPos = hash(x);

    while (array[currentPos] != null &&
       !array[currentPos].element.equals(x))
    {
       currentPos += 2 * ++collisionNum - 1;
       currentPos = currentPos % array.length;
    }
    return currentPos;
}
```

Löschen!

Achtung HashTable muss reorganisiert werden ->
Lücken auffüllen, sonst funktioniert Sondieren nicht

Knuth-Morris-Pratt

```
public static int[] buildNextTab(String pattern) {
   int lenOfPattern = pattern.length();
   int lenOfSubpattern = 0;
   int posToCompare = 1;
   int[] next = new int[lenOfPattern - 1];

   while (posToCompare < lenOfPattern - 1) {
      if (pattern.charAt(posToCompare) == pattern.charAt(lenOfSubpattern)){
         next[posToCompare] = lenOfSubpattern + 1;
         lenOfSubpattern++;
         posToCompare++;
      }
      else {
         if (lenOfSubpattern != 0) {
            lenOfSubpattern = next[lenOfSubpattern - 1];
         }
         else {
            next[posToCompare] = 0;
            posToCompare++;
         }
      }
   }
   return next;
}
```

Die Zeichen stimmen überein, versuche Präfix und Suffix zu verlängern.

Schwierigster Teil des Algorithmus. Bei einer Abweichung darf die Subpatternlänge nicht einfach auf 0 gesetzt werden, da allenfalls bereits ein anderes Subpattern erkannt wurde. Es muss daher zunächst das nächst kleinere, gefundene Subpattern ausprobiert werden → siehe nächste Folie.

Das erste Zeichen des Subpattern und das aktuelle Zeichen weichen voneinander ab → das ist nicht der Beginn eines Subpattern.

```
public static void KMP(String textToSearch, String pattern){
   int lenOfText = textToSearch.length();
   int lenOfPattern = pattern.length();
   int[] next = buildNextTab(pattern);
   next = int posOfText = 0, posOfPattern = 0;

   while ((posOfText < lenOfText) && (posOfPattern < lenOfPattern)) {
      if (textToSearch.charAt(posOfText) == pattern.charAt(posOfPattern)) {
         posOfText++;
         posOfPattern++;
      }
      else {
         if (posOfPattern != 0) {
            posOfPattern = next[posOfPattern - 1];
         }
         else {
            posOfText++;
         }
      }
   }
   if (posOfPattern == lenOfPattern) {
      println("Position: " + Integer.toString(posOfText - lenOfPattern));
   }
}
```

Die Zeichen stimmen überein, versuche Pattern und Text zu verlängern.

Starte den nächsten Vergleich mit dem Subpattern gemäss der Verschiebung in der next-Tabelle.

Das erste Zeichen des Pattern und das aktuelle Zeichen des Texts weichen voneinander ab → das ist nicht der Beginn des Pattern.

Levenshtein Distanz

```
private static int minimum(int a, int b, int c) {
   return Math.min(Math.min(a, b), c);
}
```

Minimum dreier Werte bestimmen

```
public static int computeLevenshteinDistance(String str1,String str2) {
   int[][] distance = new int[str1.len() + 1][str2.len() + 1];

   for (int i = 0; i <= str1.len(); i++) distance[i][0] = i;
   for (int j = 1; j <= str2.len(); j++) distance[0][j] = j;

   for (int i = 1; i <= str1.len(); i++) {
      for (int j = 1; j <= str2.len(); j++) {
         int minEd = (str1.charAt(i - 1) == str2.charAt(j - 1)) ? 0 : 1;
         distance[i][j] = minimum(distance[i - 1][j] + 1,
            distance[i][j - 1] + 1, distance[i - 1][j - 1]+ minEd);
      }
   }
   return distance[str1.len()][str2.len()];
}
```

Initialisierung der Spalte und Zeile 0 (alle löschen / einfügen)

Zeile für Zeile und Spalte für Spalte Distanz berechnen.

Muss Update oder TakeOver ausgeführt werden?

Kürzeste Distanz ist in Zelle rechts unten

BubbleSort O(n^2) stable

Nach dem 1. Durchgang hat man die folgende Situation:

- Das grösste Element ist ganz rechts.
- Alle anderen Elemente sind zwar zum Teil an besseren Positionen (also näher an der endgültigen Position), im Allgemeinen aber noch unsortiert.

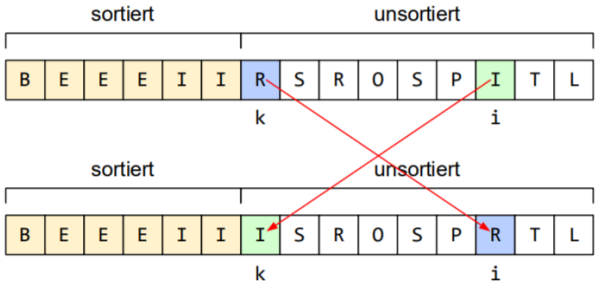
```
static <T extends Comparable> void BubbleSortG(T[] a) {
   for (int k = a.length-1; k > 0; k--){
      // bubbleUp
      for (int i = 0; i < k; i++) {
         if (a[i].compareTo(a[i+1]) > 0) swap (a, i, i+1);
      }
   }
}
```

```
static void BubbleSort1(char[] a) {
   for (int k = a.length-1; k > 0; k--) {
      boolean noSwap = true;
      for (int i = 0; i < k; i++) {
         if ( a[i] > a[i + 1])  {
            swap (a, i, i + 1);
            noSwap = false;
         }
      }
      if (noSwap) break;
   }
}
```

SelectionSort O(n^2) unstable

Suche jeweils das kleinste der verbleibenden Elemente und ordne es am Ende der bereits sortierten Elemente ein.

- Vorteil: Deutlich weniger Swap-Aufrufen als Bubble Sort.
- Nachteil: «Vorsortiertheit» kann nicht ausgenutzt werden.




```
static void SelectionSort(char[] a) {
   for (int k = 0; k < a.length; k++) {
      int min = k;
      for (int i = k + 1; i < a.length; i ++) {
         if (a[i] < a[min]) min = i;
      }
      if (min != k) swap (a, min, k);
   }
}
```

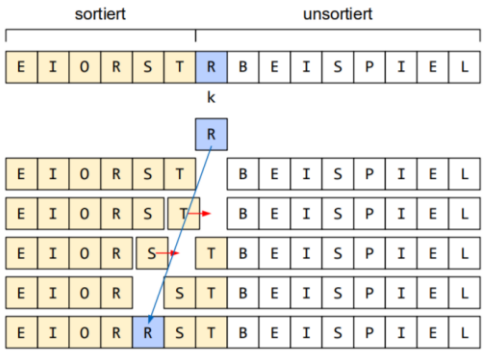
Obere Grenze des sortierten Bereichs.

Finde das kleinste Element.

Falls das kleinstes Element nicht schon am richtigen Platz: vertauschen.

InsertionSort O(n^2) stable

Vorgehen: Aus dem noch unsortierten Teil entnehmen wir das Element ganz links und ordnen es in den sortierten Teil an der richtigen Stelle ein.




```
static void InsertionSort(char[] a) {
   for (int k = 1; k < a.length; k++) {
      char x = a[k];
      int i;
      for (i = k; ((i > 0) && (a[i-1] > x)); i--)
         a[i] = a[i-1];
      a[i] = x;
   }
}
```

Element das eingeordnet werden soll.

Lücke verschieben.

Muss die Lücke noch verschoben werden?

Element einfügen.

QuickSort O(n*log(n)) unstable

Pivot wählen und in zwei Partitionen teilen, solange bis nur noch 1 Element übrigbleibt

- Teile den Bereich in zwei Bereiche «alle kleiner» und «alle grösser» als Pivot.
- Gebe Index der Grenze zurück.

```
static int partition (int[] arr, int left, int right) {
   int pivot = arr[(left + right) / 2];
   while (left <= right) {
      while (arr[left] < pivot) { left++; }
      while (arr[right] > pivot) { right--; }
      if (left <= right) {
         swap(arr,left,right);
         left++;
         right--;
      }
   }
   return left;
}
```

Hier wird direkt das Element in der Mitte als Pivot verwendet.

Finde linkes Element ≥ Pivot.

Finde rechtes Element ≤ Pivot.

l + r schon gekreuzt? Dann sind wir fertig.

Noch nicht fertig → Element austauschen und vorfahren.

Startposition der rechten Partition.

```
static void quickSort(int[] a){
   quickSort(a, 0, a.length-1);
}
```

```
static void quickSort(int[] arr, int left, int right) {
   if (left < right) {
      int mid = partition (arr, left, right);
      quickSort(arr, left, mid - 1);
      quickSort(arr, mid, right);
   }
}
```

Rekursiver Aufruf.

DistributionSort

Vorteile:

- Schneller geht's nicht.
- Linearer Algorithmus: die Komplexität ist also O(n).

Nachteile:

- Verfahren muss an den jeweiligen Sortierschlüssel angepasst werden.
- Geht nur bei Schlüsseln, die einen kleinen Wertebereich haben, oder auf einen solchen abgebildet werden können, ohne dass die Ordnung verloren geht.
- Allgemeines Hashing funktioniert nicht: Wieso?

Distribution-Sort ist mit Abstand der schnellste Algorithmus zum Sortieren.

- Es handelt sich aber nicht um ein allgemein anwendbares Sortierverfahren.

MergeSort O(n*log(n)) stable

```
Methode Mergesort (A) {
   if (A.size()) <= 1 return A;
   else {
      halbiere A in A1 und A2;
      A1 = Mergesort(A1);
      A2 = Mergesort(A2);
      return Merge(A1, A2)
   }
}
```

Liste

Zusammenführen der sortierten Listen (nächste Folie).

```
Methode Merge(linkeListe, rechteListe); {
   neueListe;
   while (!(linkeListe.isEmpty() OR rechteListe.isEmpty)) {
      if (linkeListe(0) <= rechteList(0) {
         neueListe.add(linkeListe(0)); linkeListe.remove(0);
      }
      else {
         neueListe.add(rechteListe(0)); rechteListe.remove(0);
      }
   }
   while (!linkeListe.isEmpty()) {
      neueListe.add(linkeListe(0)); linkeListe.remove(0);
   }
   while (!rechteListe.isEmpty()) {
      neueListe.add(rechteListe(0)); rechteListe.remove(0);
   }
   return neueListe;
}
```