

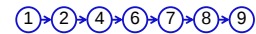
Bäume, Binäre Suchbäume, Balancierte Bäume, Binärer Heap, Effizienter Online-Median, Graphen und Fallstudie Kürzeste Wege (Dijkstra)

7. DYNAMISCHE DATENSTRUKTUREN II

220

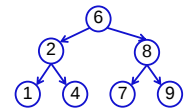
Bäume – Motivation

- Erinnerung an die Fallstudie Online-Statistik
 - Median konnte nicht effizient berechnet werden
 - Verlinkte Listen schaffen Abhilfe bzgl. effizienterem sortierten Einfügen von Elementen. Laufzeit für die Suche eines Elementes wird jedoch nicht verringert



- Suchbäume können verwendet werden, um indizierte Elemente effizient zu suchen

- Was nützt uns das für den Median?
- Antwort etwas später



Literatur: Ottmann, Widmayer, Algorithmen und Datenstrukturen, Kapitel 5

221

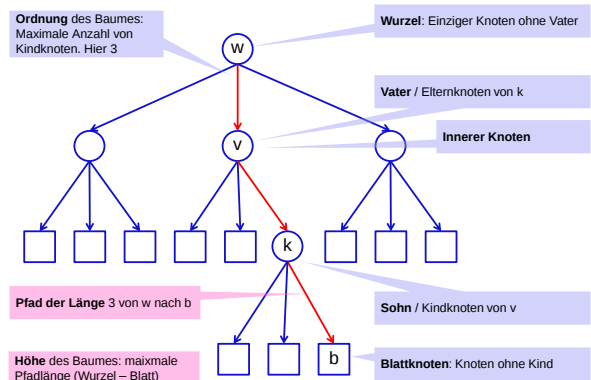
Bäume – Motivation

- Bäume sind
 - Verallgemeinerte Listen: Knoten können mehrere Nachfolger haben
 - Spezielle Graphen: Graphen bestehen aus Knoten und Kanten. Ein Baum ist ein zusammenhängender, gerichteter*, azyklischer Graph.
- Verwendung
 - Entscheidungsbäume: Hierarchische Darstellung von Entscheidungsregeln
 - Syntaxbäume: Parsen und Traversieren von Ausdrücken, z.B. in einem Compiler
 - Codebäume: Darstellung eines Codes, z.B. Morsealphabet, Huffman Code
 - Suchbäume: ermöglichen effizientes Suchen eines Elementes

*manche Autoren betrachten auch ungerichtete Bäume (wir nicht)

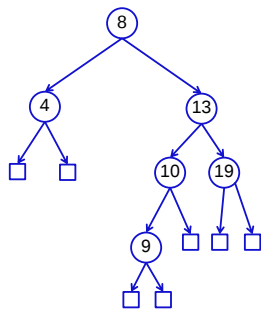
222

Bäume: Nomenklatur



223

Binärer Suchbaum



$(-\infty, 4), (4, 8), (8, 9), (9, 10), (10, 13), (13, 19), (19, \infty)$

- Baum der Ordnung 2
- Knoten beherbergen paarweise verschiedene Schlüssel (und potentiell andere Nutzdaten)
- Schlüssel im linken Teilbaum kleiner als am Knoten
- Schlüssel im rechten Teilbaum grösser als am Knoten
- (Leere*) Blätter repräsentieren Schlüsselintervalle

*ausser bei Blattsuchbäumen, betrachten wir hier nicht

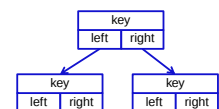
224

Datenstruktur Suchknoten

Ähnlich wie bei den Listen besteht ein Baum aus verketteten / verflochtenen Instanzen einer Datenstruktur Knoten (SearchNode)

```
public class SearchNode {
    int key; // Schlüssel
    SearchNode left; // Linker Teilbaum
    SearchNode right; // rechter Teilbaum

    // Konstruktor: Knoten ohne Nachfolger
    SearchNode(int k) {
        key = k;
        left = right = null;
    }
}
```



225

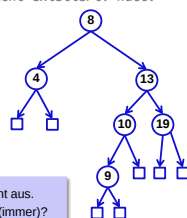
Datenstruktur Suchbaum

```
public class SearchTree {
    SearchNode root; // Wurzelknoten

    // Konstruktor: Leerer Suchbaum
    SearchTree() {
        root = null;
    }

    // Gibt Knoten mit Schlüssel k zurück. Wenn nicht existiert: null.
    public SearchNode Search(int k) {
        Node n = root;
        while (n != null && n.key != k) {
            if (k < n.key) n = n.left;
            else n = n.right;
        }
        return n;
    }

    ... // Einfügen, Löschen
}
```



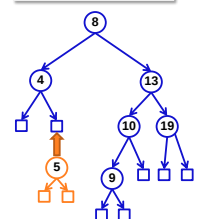
Sieht effizient aus. Stimmt das (immer)?

226

Knoten Einfügen

```
// Fügt Knoten mit Schlüssel k ein. Gibt erzeugten Knoten zurück.
// null, wenn Knoten mit Schlüssel k bereits existiert
public SearchNode Insert(int k) {
    if (root == null)
        return root = new SearchNode(k);
    SearchNode t = root;
    while (true) {
        if (k == t.key)
            return null;
        if (k < t.key) {
            if (t.left == null)
                return t.left = new SearchNode(k);
            else
                t = t.left;
        } else { // k > t.key
            if (t.right == null)
                return t.right = new SearchNode(k);
            else
                t = t.right;
        }
    }
}
```

Traversiere Baum bis zum passenden Intervall-Blatt, ersetze Intervallblatt mit Knoten



227

Knoten Löschen

Drei Fälle

1. Knoten hat **keine Kinder**
 - Knoten entfernen
2. Knoten hat **nur ein Kind**
 - Knoten durch Kind ersetzen
3. Knoten hat **zwei Kinder**
 - Knoten durch einen *symmetrischen Nachfolger* ersetzen
 - Symmetrischer Nachfolger: Knoten im rechten (*linken*) Teilbaum, welcher am weitesten links (*rechts*) steht.
 - Korrespondiert mit dem kleinsten (*grössten*) Schlüssel, welcher gerade noch grösser (*kleiner*) als der Schlüssel des zu entfernenden Knotens ist
 - Symmetrischer Nachfolger hat maximal ein Kind

```
graph TD; 8((8)) --> 3((3)); 8 --> 12((12)); 3 --> empty1[ ]; 3 --> 5((5)); 5 --> 4((4)); 5 --> empty2[ ]; 4 --> empty3[ ]; 4 --> empty4[ ]; 12 --> 10((10)); 12 --> 19((19)); 10 --> 9((9)); 10 --> empty5[ ]; 9 --> empty6[ ]; 9 --> empty7[ ]; 19 --> empty8[ ]; 19 --> empty9[ ];
```

- # Symmetrischer Nachfolger
- ```
public SearchNode SymmetricDesc(SearchNode node)
{
 if (node.left == null)
 return node.right;
 if (node.right == null)
 return node.left;
 SearchNode n = node;
 SearchNode parent = null;
 n = n.right; // guaranteed to be non-null
 while (n.left != null) {
 parent = n;
 n = n.left;
 }
 if (parent != null) { // implies that n != node.right
 parent.left = n.right; // delete n from parent
 n.left = node.left; // set left child of n
 n.right = node.right; // set right child of n
 }
 else // parent == null <-> n = node.right
 n.left = node.left;
 return n;
}
```
- 
- Diagram illustrating the SymmetricDesc algorithm. The tree structure shows nodes 8 (root, green), 3 (left child of 8, blue), 13 (right child of 8, green), 5 (right child of 3, blue), 4 (left child of 5, red), 10 (left child of 13, blue), 19 (right child of 13, red), and 9 (left child of 19, red). Blue squares represent null children. Arrows indicate pointer updates: one arrow points from 'node' to 8, another from 'n' to 3, and a third from 'parent' to 3.
- 230

```

graph TD
 8((8)) --> 3((3))
 8 --> 13((13))
 3 --> 5((5))
 3 --> S1[]
 5 --> 4((4))
 5 --> S2[]
 4 --> S3[]
 4 --> S4[]
 13 --> 10((10))
 13 --> 19((19))
 10 --> 9((9))
 10 --> S5[]
 9 --> S6[]
 9 --> S7[]
 19 --> S8[]
 19 --> S9[]

```

e.right

# Knoten Löschen

```

public boolean Delete (int k) {
 SearchNode n = root;
 if (n != null && n.key == k) {
 root = SymmetricDesc(root);
 return true;
 }
 else {
 while (n != null)
 {
 if (n.left != null && k == n.left.key) {
 n.left = SymmetricDesc(n.left);
 return true;
 }
 else if (n.right != null && k == n.right.key) {
 n.right = SymmetricDesc(n.right);
 return true;
 }
 else if (k < n.key)
 n = n.left;
 else
 n = n.right;
 }
 return false;
 }
}

```

229

```

graph TD
 8((8)) --- 3((3))
 8 --- 13((13))
 3 --- empty1[]
 3 --- 5((5))
 5 --- 4((4))
 5 --- empty2[]
 4 --- empty3[]
 4 --- empty4[]
 13 --- 10((10))
 13 --- 19((19))
 10 --- 9((9))
 10 --- empty5[]
 9 --- empty6[]
 9 --- empty7[]
 19 --- empty8[]
 19 --- empty9[]

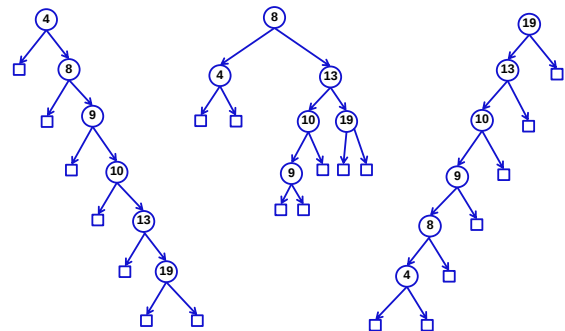
```

== n.left.key {  
 rsc(n.left);

# Degenerierte Bäume

Binäre Bäume können, je nach Updateoperationen, zu linearen Listen degenerieren! Folgerung?

231



# Balancierte Bäume

Komplexität von Suchen, Einfügen und Löschen eines Knoten in binären Suchbäumen *im Mittel*  $O(\log_2 n)$

**Worst case:**  $O(n)$  bei Degeneration des Baumes

Verhinderung der Degeneration: künstliches, bei jeder Update-Operation erfolgtes Balancieren eines Baumes: worst case auch  $O(\log_2 n)$

Balancieren: garantiere, dass ein Baum mit  $n$  Knoten stets eine Höhe von  $O(\log n)$  hat.

232

# AVL Bäume

Adelson-Velskij und Landis (1962): für jeden Knoten  $p$  im höhenbalancierten Baum gilt folgende **Invariante**: **Unterschied Höhe linker Teilbaum und Höhe rechter Teilbaum von  $p$  maximal 1**.

Es folgt (ohne Beweis): Höhe AVL Baum  $\leq 1 + 1.44 \log_2 n$ .

AVL Baum der Höhe 2

AVL Baum der Höhe 2

AVL Baum der Höhe 3

Kein AVL Baum

233

A diagram of a tree structure. The root node is at the top. It has two children. The left child is at height  $h+1$  (indicated by a downward arrow). The right child is at height  $h$  (indicated by a downward arrow). The root node is at height  $h+2$  (indicated by an upward arrow).

The image shows four binary tree diagrams. The first tree, labeled 'BFS Baum', is a complete binary tree with root 'a', children 'b' and 'c', and grandchildren 'd', 'e', 'f', and 'g'. The second tree, labeled 'AVL Baum', is a balanced binary tree with root 'a', left child 'b' (with children 'd' and 'e'), and right child 'c' (with child 'f'). The third tree, labeled 'DFS Baum', is a binary tree with root 'a', left child 'b' (with children 'd' and 'e'), and right child 'c' (with children 'f' and 'g'). The fourth tree is a more complex binary tree with root 'a', left child 'b' (with children 'd' and 'e'), and right child 'c' (with children 'f' and 'g'). The tree under 'c' has further children: 'f' has children 'h' and 'i', 'g' has children 'j' and 'k', 'h' has children 'l' and 'm', 'i' has children 'n' and 'o', 'j' has children 'p' and 'q', and 'k' has children 'r' and 's'.

Kein AVL Baum

# Balancefaktor

Man muss glücklicherweise nicht für jeden Teilbaum die Höhe mitführen. Es genügt der Balancefaktor

$bal(p) = \text{Höhe Rechter Teilbaum} - \text{Höhe Linker Teilbaum}$

AVL-Baum:  $bal(p) \in \{-1, 0, 1\}$

Balancefaktoren für untere Knoten:

Diagram illustrating four possible balance factor configurations for a node and its children in an AVL tree:

- Root node balance factor 0: Root has two leaf children.
- Root node balance factor -1: Root has a left child (internal node) and a right child (leaf). The left child has two leaf children.
- Root node balance factor +1: Root has a left child (leaf) and a right child (internal node). The right child has two leaf children.
- Root node balance factor 0: Root has two internal children. Each child has two leaf children.

234

# Einfügen

- Suche nach Einfügeort (leeres Blatt!) wie bei binärem Suchbaum. Sei  $p$  der *Elternknoten* beim Einfügeort. Drei Fälle:
  - $bal(p) = +1 \Rightarrow bal(p) = 0$   
fertig
 
  - $bal(p) = -1 \Rightarrow bal(p) = 0$   
fertig
 
  - $bal(p) = 0 \Rightarrow bal(p) = \pm 1$   
komplizierter:  
Höhe des Teilbaumes wächst  
Aufruf der Funktion **upin(p)**


235

- 

- 

- st
- 
- z.B.

## Balancierung bei Höhenzunahme

### Funktion upin(p)

Sei pp der Elternknoten von p

3a) p ist linker Sohn von pp

3a.1)  $bal(pp) = +1 \Rightarrow bal(pp) = 0$  fertig.

3a.2)  $bal(pp) = 0 \Rightarrow bal(pp) = -1$ , Aufruf upin(pp)

3a.3)  $bal(pp) = -1 \Rightarrow$  nächste Folie

3b) p ist rechter Sohn von pp

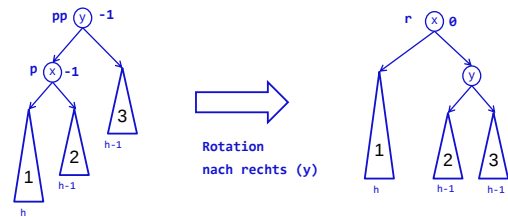
Symmetrische Fälle 3b.1) – 3b.3) wie bei 3a) unter Vertauschung von +1 und -1

236

## Rotationen

Fall 3a.3  $bal(pp) = -1$

3a.3.1:  $bal(p) = -1$



[ Fall 3b.3.1:  $bal(pp) = +1, bal(p) = +1$ : Rotation nach links ]

237

## Rotationen – Quellcode

AVLNode RotateRight(AVLNode p)

```
{
 AVLNode left = p.left;
 p.left = left.right;
 left.right = p;
 return left;
}
```

AVLNode RotateLeft(AVLNode p)

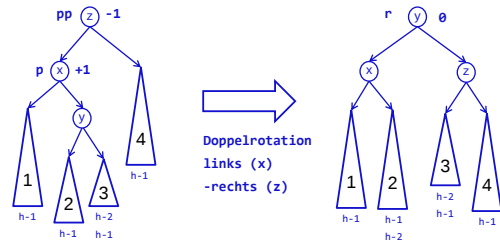
```
{
 AVLNode right = p.right;
 p.right = right.left;
 right.left = p;
 return right;
}
```

238

## Rotationen

Fall 3a.3  $bal(pp) = -1$

3a.3.2:  $bal(p) = +1$



[ Fall 3b.3.2:  $bal(pp) = +1, bal(p) = -1$ : Doppelrotation rechts-links ]

239

## Ausschnitt upin Quellcode

```
else // parent.bal == -1
{
 if (p.bal == -1)
 {
 parent.bal = 0;
 p.bal = 0;
 Replace(parent, RotateRight(parent));
 }
 else if (p.bal == 1)
 {
 int oldBalance = p.right.bal;
 AVLNode r = RotateLeft(p);
 if (oldBalance <= 0)
 {
 p.bal = 0;
 }
 else
 {
 p.bal = -1;
 parent.left = r;
 r = RotateRight(parent);
 p.bal = 0;
 }
 if (oldBalance >= 0)
 {
 parent.bal = 0;
 }
 else
 {
 parent.bal = 1;
 Replace(parent, r);
 }
 }
}
```

Zur Illustration, wie es in etwa aussieht. Der ganze Quellcode wie immer auf der Homepage.

AVL Bäume sind nicht schwierig zu implementieren, wenn man die hier vorgestellten Regeln genau folgt.

Flüchtigkeitsfehler schleichen sich aber leicht ein und dann muss man mit **konsequentem Einflechten von Checks der Invarianten arbeiten**.

## Defensives Programmieren !

```
int GetDepth(AVLNode node) {
 if (node == null) return 0;
 int left = GetDepth(node.left);
 int right = GetDepth(node.right);
 if (left > right) return left+1;
 else return right+1;
}

boolean CheckBal(AVLNode node) {
 int left = GetDepth(node.left);
 int right = GetDepth(node.right);
 if (left < right)
 return (node.bal == 1);
 else if (left > right)
 return (node.bal == -1);
 else
 return (node.bal == 0);
}

boolean Check(AVLNode node) {
 if (node == null) return true;
 return CheckBal(node) && Check(node.left) && Check(node.right);
}
```

- Einflechten von **assert(Check(node))** nach / in jeder elementaren Operation (Einfügen / Löschen): prüft die Invariante.
- Frühe Fehlererkennung !
- An-/ abschaltbar ( $\rightarrow$  Performance).

241

## Löschen eines Knotens\*

- Fall 1: Knoten hat nur (leere) Blätter als Kinder

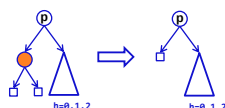
- Sei p der Vater des Knotens

- Dann hat anderer Teilbaum Höhe 0,1 oder 2

- Höhe 1  $\rightarrow$  Anpassen Balance (p)

- Höhe 0  $\rightarrow$  Balance(p)=0, upout(p)

- Höhe 2  $\rightarrow$  Balancieren durch Rotation, upout(p)  
[alternativ upout(blatt(p)), enthält Balancierung]



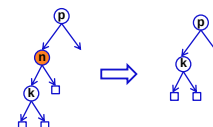
242

## Löschen eines Knotens\*

- Fall 2: Knoten hat einen inneren Knoten als Kind

- Ersetze Knoten n durch sein Kind k

- Upout(k)



- Fall 3: Knoten hat zwei innere Knoten als Kind

- Ersetze Knoten durch seinen symmetrischen Nachfolger.

- Löschen des symmetrischen Nachfolgers ist auf Fall 1 oder Fall 2 zurückzuführen

243

## Balancierung bei Höhenabnahme\*

### Funktion upout(p)

Sei pp der Elternknoten von p

a) p ist linker Sohn von pp

a.1)  $bal(pp) = -1 \Rightarrow bal(pp) = 0$  upout(pp).

a.2)  $bal(pp) = 0 \Rightarrow bal(pp) = 1$ , fertig

a.3)  $bal(pp) = +1 \Rightarrow$  nächste Folie

3b) p ist rechter Sohn von pp

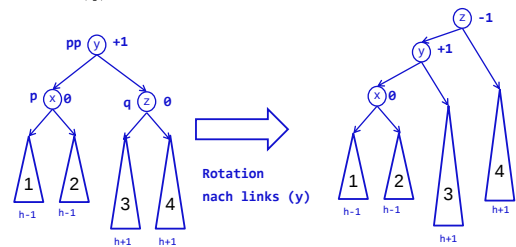
Symmetrische Fälle b.1) – b.3) wie bei a) unter Vertauschung von +1 und -1

244

## Rotationen\*

Fall a.3  $bal(pp) = +1$ , sei q Bruder von p

a.3.1:  $bal(q) = 0$



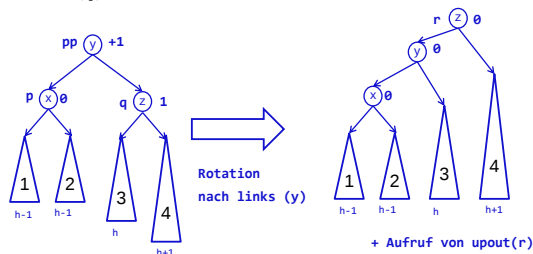
[ Fall b.3.1:  $bal(pp) = +1, bal(q) = -1$ : Rotation nach rechts]

245

## Rotationen\*

Fall a.3  $bal(pp) = +1$

a.3.2:  $bal(q) = +1$



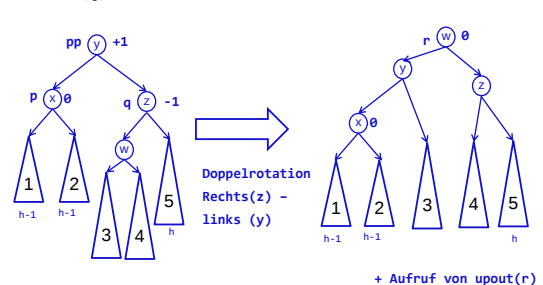
[ Fall 3b.3.2:  $bal(pp) = +1, bal(q) = +1$ : Rotation nach rechts, upout]

246

## Rotationen\*

Fall a.3  $bal(pp) = +1$

a.3.3:  $bal(q) = -1$



[ Fall 3b.3.2:  $bal(pp) = +1, bal(q) = -1$ : Rotation Links-Rechts, upout]

247

## Balancierte Bäume – Zusammenfassung

Vorteil:

AVL-Bäume haben worst-case Komplexität  $O(\log n)$  für Finden, Einfügen und Löschen von Knoten.

Nachteil:

Einfügen und Löschen kompliziert, schwierig zu implementieren und für kleine Probleme relativ langsam.

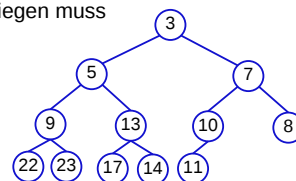
Heaps schaffen Abhilfe, wenn der Zielparameter von Algorithmen nur das Minimum oder Maximum der Daten ist.

248

## Heaps

Ein (Min-)Heap ist ein Binärbaum, welcher

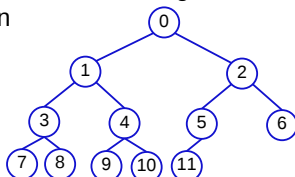
- die (Min-)Heap-Eigenschaft hat: Schlüssel eines Kindes ist immer grösser als der des Vaters. [Max-Heap: Kinder-Schlüssel kleiner als Vater-Schlüssel]
- bis auf die letzte Ebene vollständig ist
- höchstens eine Lücke in der letzten Ebene hat, welche rechts liegen muss



249

## Heaps und Arrays

Ein Heap lässt sich sehr gut in einem Array speichern



Es gilt

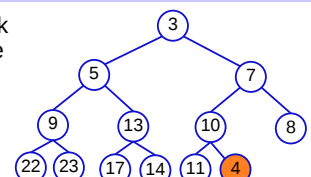
$$\text{Vater}(i) = \lfloor (i-1)/2 \rfloor, \text{Kinder}(i) = \{2i, 2i+1\}$$

|   |   |   |   |   |   |   |   |   |   |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|----|----|

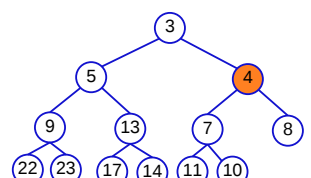
250

## Einfügen

- Füge ein neues Element k an der ersten freien Stelle ein. Verletzt Heap-Eigenschaft potentiell.



- Stelle Heap-Eigenschaft wieder her durch sukzessives Aufsteigen von k.



Worst-Case Komplexität  $O(\log n)$

251

## Datenstruktur ArrayHeap

```
public class ArrayHeap {

 float[] data; // Array zum Speichern der Daten
 int used; // Anzahl belegte Knoten

 ArrayHeap () {
 data = new float[16];
 used = 0;
 }

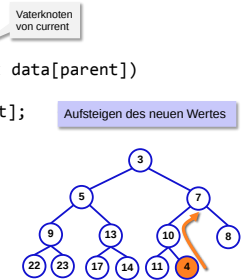
 void Grow(){ ... } // Binäres Vergrößern von data

 ...
}
```

252

## Insert

```
public void Insert(double value)
{
 System.out.println("Insert "+value);
 if (used == data.length)
 Grow();
 int current = used;
 int parent = (current-1)/2;
 while (current > 0 && value < data[parent])
 {
 data[current] = data[parent];
 current = parent;
 parent = (current-1)/2;
 }
 data[current] = value;
 used++;
 Check(0);
}
```



253

## GetMin

Das kleinste Element ist immer an der Wurzel im Baum. Somit kann es sehr schnell ausgelesen werden ( $O(1)$ ).

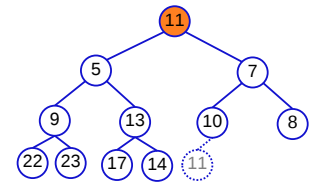
Wie verhält es sich aber mit Auslesen *und* Entfernen?

Wiederholtes Entfernen der Wurzel ergibt Schlüssel in aufsteigender Reihenfolge: das kann z.B. auch zum *Sortieren* verwendet werden.

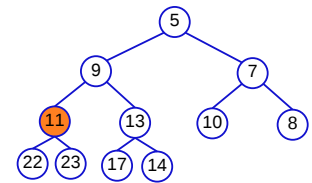
254

## Minimum entfernen

- Ersetze die Wurzel durch den letzten Knoten



- Lass die Wurzel nach unten sinken, um die Invariante wiederherzustellen

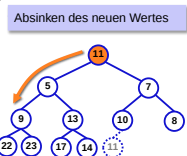


Worst-Case Komplexität  $O(\log n)$

255

## Wurzel Extrahieren

```
public float ExtractRoot() {
 assert(used > 0);
 float min = data[0];
 float value = data[used];
 int i = 0;
 int j;
 while(true)
 {
 if (2*i+2 >= used || data[2*i+1] < data[2*i+2])
 j=2*i+1; // Linker Knoten
 else
 j=2*i+2; // Rechter Knoten
 if (j >= used || value < data[j]) // fertig
 break;
 data[i] = data[j];
 i=j;
 }
 data[i] = value;
 used--;
 return min;
}
```



256

## Sortieren

Als Seiteneffekt fällt noch ein Sortieralgorithmus mit worst-case Komplexität  $O(n \log n)$  ab

Folgender Code sortiert (absteigend) direkt im Datenarray und gibt das Array zurück.

```
public double[] Sort()
{
 while (used > 0)
 {
 int pos = used-1;
 data[pos] = ExtractRoot();
 }
 return data;
}
```

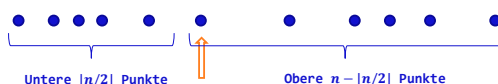
Möchte man aufsteigend sortieren, verwendet man einen Max-Heap, d.h. einen Min-Heap mit geänderter Vergleichsoperation: aus  $a < b$  wird  $a > b$ .

257

## Verwendungsbeispiel Heap

Können wir das schnelle Auslesen, Extrahieren und Einfügen von Minimum (bzw. Maximum) im Heap für einen online-Algorithmus des Median nutzen?

Beobachtung: Der Median bildet sich aus Minimum der oberen Hälfte der Daten und / oder Maximum der unteren Hälfte der Daten.



258

## Online Median

Verwende Max-Heap  $H_{max}$  und Min-Heap  $H_{min}$ . Bezeichne Anzahl Elemente mit  $|H_{max}|$  und  $|H_{min}|$

- Einfügen neuen Wertes  $v$  in
  - $H_{max}$ , wenn  $v \leq \max(H_{max})$
  - $H_{min}$ , sonst
- Rebalancieren der beiden Heaps
  - Falls  $|H_{max}| > \lfloor n/2 \rfloor$ , dann extrahiere Wurzel von  $H_{max}$  und füge den Wert bei  $H_{min}$  ein.
  - Falls  $|H_{min}| < \lfloor n/2 \rfloor$ , dann extrahiere Wurzel von  $H_{min}$  und füge den Wert bei  $H_{max}$  ein.
- Gesamt worst-case Komplexität  $O(\log n)$

259

## Berechnung Median

### Berechnung Median

- Wenn  $n$  ungerade, dann  

$$\text{median} = \min(H_{\min})$$
  - Wenn  $n$  gerade, dann  

$$\text{median} = \frac{\max(H_{\max}) + \min(H_{\min})}{2}$$
- worst-case Komplexität  $O(1)$

260

## Online Median

```
public class OnlineMedian {
 ArrayHeap minHeap;
 ArrayHeap maxHeap;
 int n;

 OnlineMedian(){
 n = 0;
 minHeap = new ArrayHeap(true);
 maxHeap = new ArrayHeap(false);
 }

 public void Insert(double value){...}

 public double Get() {...}
}
```

261

## Online Median

```
public void Insert(double value){
 if (maxHeap.NumberElements()==0 || value < maxHeap.GetRoot())
 maxHeap.Insert(value);
 else
 minHeap.Insert(value);
 ++n;
 if (maxHeap.NumberElements() < n/2)
 maxHeap.Insert(minHeap.ExtractRoot());
 else if (maxHeap.NumberElements() > n/2)
 minHeap.Insert(maxHeap.ExtractRoot());
}

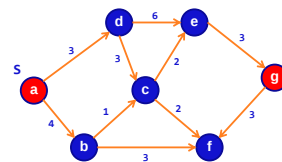
public double Get() {
 if (n%2 == 0)
 return (minHeap.GetRoot()+maxHeap.GetRoot())/2;
 else
 return maxHeap.GetRoot();
}
```

262

## Fallstudie Dijkstra's Shortest Path

Gegeben: Gerichteter Graph  $(V, E)$  mit Knotenmenge  $V$  und Kantenmenge  $E$ , bei dem jeder Kante  $e \in E$  eine Länge  $l(e) \geq 0$  zugeordnet ist.

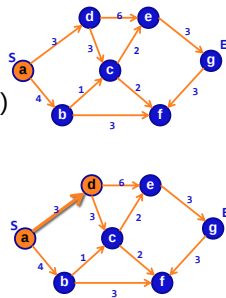
Problem: Finde zu Startpunkt  $S \in V$  und Endpunkt  $E \in V$  den kürzesten Pfad entlang der Kanten im Graph.



263

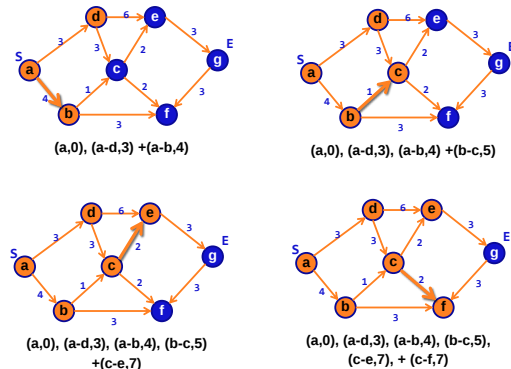
## Algorithmus Idee

- Durchprobieren aller Pfade zu ineffizient
- Dijkstra's Idee: Aufbau der kürzesten Pfade bis Ziel gefunden
- Starte bei S, (Knoten, Pfadlänge) : (a,0)
- Kürzester Weg von (a,0)  
 Zusätzliche Kante  
 +(a-d,3)



264

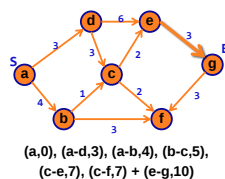
## Algorithmus Idee



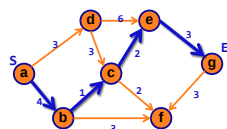
265

## Algorithmus Idee

- Algorithmus terminiert, wenn Ziel erreicht



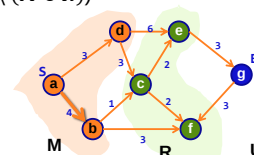
Weg finden: über Vorgänger zurücklaufen



266

## Implementation

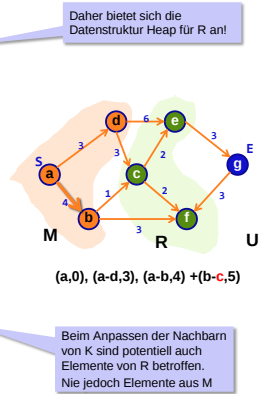
- Grundsätzlich wird die Menge der Knoten unterteilt in
  - Knoten, die schon als Teil eines minimalen Pfades erkannt wurden ( $M$ )
  - Knoten, die nicht in  $M$  enthalten sind, jedoch von  $M$  aus direkt (über eine Kante) erreichbar sind ( $R$ ) und
  - Knoten die noch nie berücksichtigt wurden ( $U := V \setminus (M \cup R)$ )



267

## Algorithmus

- Ein Knoten K aus R mit minimaler Pfadlänge in R kann nicht mit kürzerer Pfadlänge über einen anderen Knoten in R erreicht werden.
- Daher kann er zu M hinzugenommen werden.
- Dabei vergrößert sich R potentiell um die Nachbarschaft von K und die Pfadlänge aller von K aus direkt erreichbaren Knoten muss angepasst werden.



268

## Algorithmus

- Setze Pfadlänge(S)=0;
- Starte mit  $M = \{S\}$ ; Setze  $K := S$ ;
- Solange ein neuer Knoten K hinzukommt und dieser nicht der Zielknoten ist
  - Für jeden Nachbarknoten N von K:
    - Passe die Pfadlänge alle Nachbarknoten N von K an, sofern sie kürzer wird
    - War der Knoten noch nicht in R, dann nehme ihn hinzu
    - War der Knoten schon in R, so passe die Datenstruktur R an den neuen Wert der Pfadlänge von N an
- Wähle als neuen Knoten den Knoten mit kleinster Pfadlänge in R

269

## Implementation Knoten

```
public class Node {
 // Gerichtete Kanten zu anderen Knoten
 LinkedList<Edge> out;
 String name;
 // Variablen für den Algorithmus
 int pathLen; // kürzeste Pfadlänge an diesem Knoten
 Node pathParent; // vorheriger Knoten im kürzesten Pfad

 Node(String n) {
 out = new LinkedList<Edge>();
 pathParent = null;
 pathLen = Integer.MAX_VALUE;
 name = n;
 }

 public void AddOutEdge(Edge e){
 out.add(e);
 }

 // Getters und Setters für pathLen und pathParent weggelassen
}
```

270

## Implementation Kanten

```
public class Edge {
 // Zielknoten und Länge
 Node to;
 int length;

 Edge (Node t, int l) {
 to = t; length = l;
 }

 public int GetLength() {
 return length;
 }

 public Node GetDestination() {
 return to;
 }
}
```

271

## Implementation: Graph

```
public class Graph {
 LinkedList<Node> nodes;

 Graph(){
 nodes = new LinkedList<Node>();
 }

 public void AddNode(Node n){
 nodes.add(n);
 }

 ...
}
```

272

## Implementation Algorithmus

```
LinkedList<Node> ShortestPath(Node S, Node E)
{
 // Initialisierung
 LinkedList<Node> path = new LinkedList<Node>();

 Heap R=new Heap();
 for (Node node: nodes)
 {
 node.SetPathLen(Integer.MAX_VALUE);
 node.SetPathParent(null);
 }
 S.SetPathLen(0);
 Node newNode = S;

 ...
}
```

273

## Implementation Algorithmus

```
...
// Kernstück des Algorithmus
while (newNode != null && newNode != E){
 for (Edge edge: newNode.out){
 int newLength = newNode.GetPathLen() + edge.length;
 Node dest = edge.to;
 int prevLength = dest.GetPathLen();
 if (newLength < prevLength){
 dest.SetPathLen(newLength);
 dest.SetPathParent(newNode);
 if (prevLength == Integer.MAX_VALUE) // not in R
 R.Insert(dest);
 else
 R.DecreaseKey(dest);
 }
 }
 newNode = R.ExtractRoot();
}
...
}
```

274

## Implementation Algorithmus

```
...
// Rückwärtstraversieren
while (newNode != null)
{
 path.push(newNode);
 newNode = newNode.GetPathParent();
}

return path;
}
```

275

# Heap: DecreaseKey

```
public class Heap {
 ...
 public void DecreaseKey(Node n)
 {
 int current;
 for (current=0; current<used && data[current] != n; ++current);
 assert(current < used);
 int parent = (current-1)/2;
 while (current > 0 && Smaller(n, data[parent]))
 {
 data[current] = data[parent];
 current = parent;
 parent = (current-1)/2;
 }
 data[current] = n;
 }
}
```

Das ist leider  $O(n)$  im Heap.  
Abhilfe schafft Verwendung eines  
balancierten Baumes oder Hashing  
(→ Hashing: nächstes Kapitel).