

## Wörterbuchimplementationen

# 10. Wiederholung Natürliche Suchbäume und Heaps

[Ottman/Widmayer, Kap. 2.3, 5.1, Cormen et al, Kap. 6, 12.1 - 12.3]

Hashing: Implementierung von Wörterbüchern mit erwartet sehr schnellen Zugriffszeiten.

Nachteile von Hashing: im schlechtesten Fall lineare Zugriffszeit.

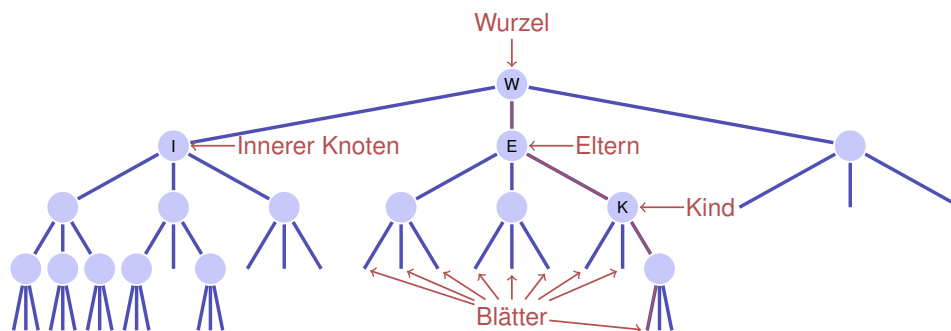
Manche Operationen gar nicht unterstützt:

- Aufzählen von Schlüssel in aufsteigender Anordnung
- Nächst kleinerer Schlüssel zu gegebenem Schlüssel

199

200

## Nomenklatur



- Ordnung des Baumes: Maximale Anzahl Kindknoten, hier: 3
- Höhe des Baumes: maximale Pfadlänge Wurzel – Blatt (hier: 4)

201

## Binäre Bäume

Ein binärer Baum ist entweder

- ein Blatt, d.h. ein leerer Baum, oder
- ein innerer Knoten mit zwei Bäumen  $T_l$  (linker Teilbaum) und  $T_r$  (rechter Teilbaum) als linken und rechten Nachfolger.

In jedem Knoten  $v$  speichern wir

key	
left	right

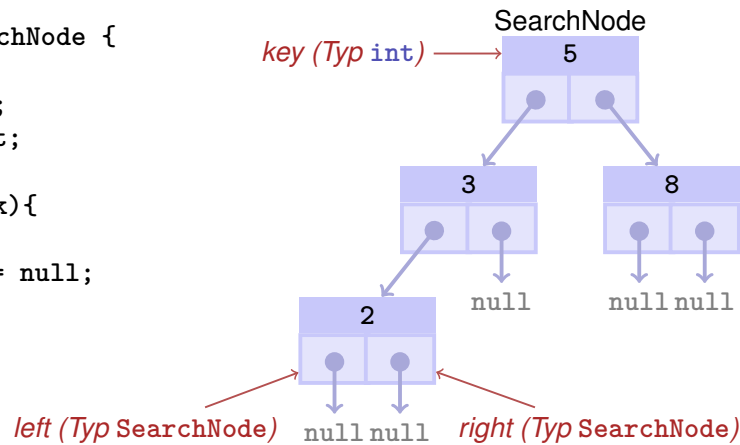
- einen Schlüssel  $v.key$  und
- zwei Zeiger  $v.left$  und  $v.right$  auf die Wurzeln der linken und rechten Teilbäume.
- Ein Blatt wird durch den **null**-Zeiger repräsentiert

202

## Baumknoten in Java

```
public class SearchNode {
    int key;
    SearchNode left;
    SearchNode right;

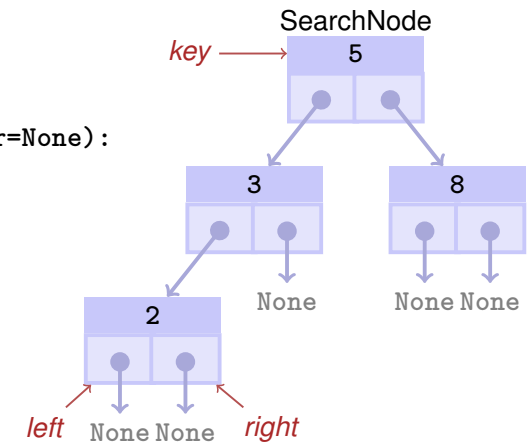
    SearchNode(int k){
        key = k;
        left = right = null;
    }
}
```



203

## Baumknoten in Python

```
class SearchNode:
    def __init__(self, k, l=None, r=None):
        self.key = k
        self.left, self.right = l, r
        self.flagged = False
```

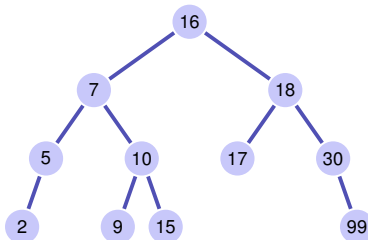


204

## Binärer Suchbaum

Ein binärer Suchbaum ist ein binärer Baum, der die *Suchbaumeigenschaft* erfüllt:

- Jeder Knoten  $v$  speichert einen Schlüssel
- Schlüssel im linken Teilbaum  $v.left$  kleiner als  $v.key$
- Schlüssel im rechten Teilbaum  $v.right$  grösser als  $v.key$



205

## Suchen

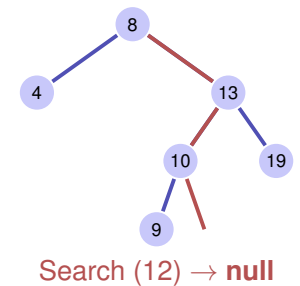
**Input:** Binärer Suchbaum mit Wurzel  $r$ , Schlüssel  $k$

**Output:** Knoten  $v$  mit  $v.key = k$  oder **null**

$v \leftarrow r$

```
while  $v \neq \text{null}$  do
    if  $k = v.key$  then
        return  $v$ 
    else if  $k < v.key$  then
         $v \leftarrow v.left$ 
    else
         $v \leftarrow v.right$ 
```

return null



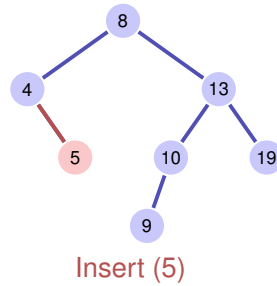
Search (12)  $\rightarrow$  null

206

## Einfügen eines Schlüssels

Einfügen des Schlüssels  $k$

- Suche nach  $k$ .
- Wenn erfolgreich:  
Fehlerausgabe
- Wenn erfolglos: Einfügen des Schlüssels am erreichten Blatt.

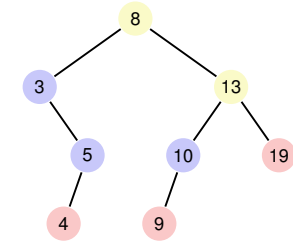


## Knoten entfernen

Drei Fälle möglich

- Knoten hat keine Kinder
- Knoten hat ein Kind
- Knoten hat zwei Kinder

[Blätter zählen hier nicht]



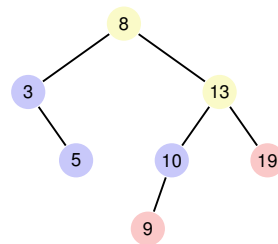
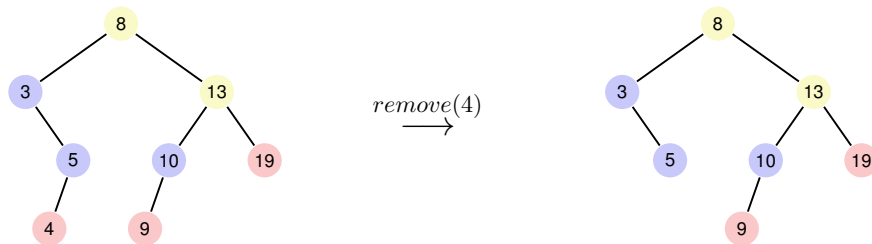
207

208

## Knoten entfernen

Knoten hat keine Kinder

Einfacher Fall: Knoten durch Blatt ersetzen.

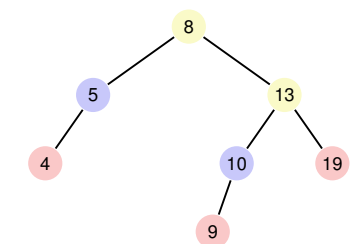
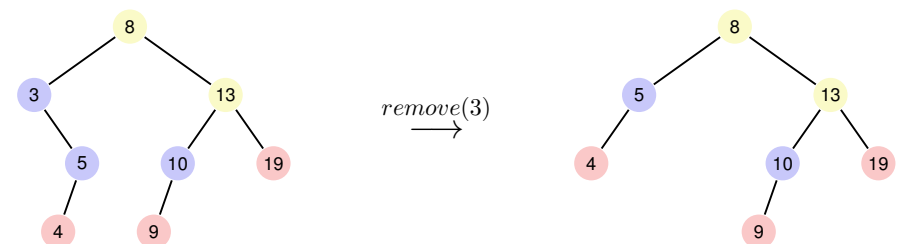


209

## Knoten entfernen

Knoten hat ein Kind

Auch einfach: Knoten durch das einzige Kind ersetzen.



210

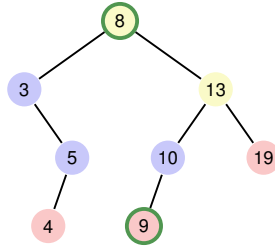
## Knoten entfernen

Knoten  $v$  hat zwei Kinder

Beobachtung: Der kleinste Schlüssel im rechten Teilbaum  $v.right$  (der *symmetrische Nachfolger* von  $v$ )

- ist kleiner als alle Schlüssel in  $v.right$
- ist grösser als alle Schlüssel in  $v.left$
- und hat kein linkes Kind.

Lösung: ersetze  $v$  durch seinen symmetrischen Nachfolger

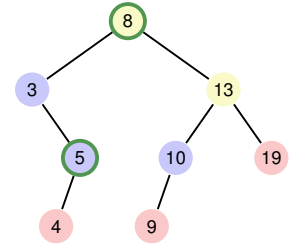


211

## Aus Symmetriegründen...

Knoten  $v$  hat zwei Kinder

Auch möglich: ersetze  $v$  durch seinen symmetrischen Vorgänger



212

## Algorithmus SymmetricSuccessor( $v$ )

**Input:** Knoten  $v$  eines binären Suchbaumes

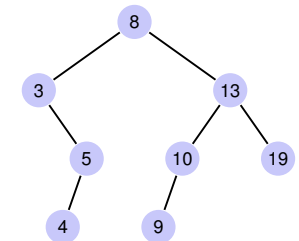
**Output:** Symmetrischer Nachfolger von  $v$

```
w ← v.right
x ← w.left
while x ≠ null do
  w ← x
  x ← x.left
return w
```

213

## Traversierungsarten

- Hauptreihenfolge (preorder):  $v$ , dann  $T_{left}(v)$ , dann  $T_{right}(v)$ .  
8, 3, 5, 4, 13, 10, 9, 19
- Nebenreihenfolge (postorder):  $T_{left}(v)$ , dann  $T_{right}(v)$ , dann  $v$ .  
4, 5, 3, 9, 10, 19, 13, 8
- Symmetrische Reihenfolge (inorder):  $T_{left}(v)$ , dann  $v$ , dann  $T_{right}(v)$ .  
3, 4, 5, 8, 9, 10, 13, 19



214

## Höhe eines Baumes

Die Höhe  $h(T)$  eines Baumes  $T$  mit Wurzel  $r$  ist gegeben als

$$h(r) = \begin{cases} 0 & \text{falls } r = \mathbf{null} \\ 1 + \max\{h(r.\text{left}), h(r.\text{right})\} & \text{sonst.} \end{cases}$$

Die Laufzeit der Suche ist somit im schlechtesten Fall  $\mathcal{O}(h(T))$

## Analyse

Suchen, Einfügen und Löschen eines Elementes  $v$  aus einem Baum  $T$  benötigt im schlechtesten Fall  $\mathcal{O}(h(T))$  Elementarschritte.

215

216

## Mögliche Höhen

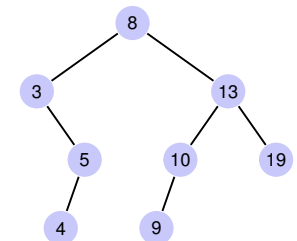
- 1 Die maximale Höhe  $h_n$  eines Baumes mit  $n$  inneren Knoten ergibt sich durch  $h_1 = 1$  und  $h_{n+1} \leq 1 + h_n$  als  $h_n \geq n$ .
- 2 Die minimale Höhe  $h_n$  eines (ideal balancierten) Baumes mit  $n$  inneren Knoten erfüllt  $n \leq \sum_{i=0}^{h-1} 2^i = 2^h - 1$ .

Somit

$$\lceil \log_2(n+1) \rceil \leq h \leq n$$

## Weitere unterstützte Operationen

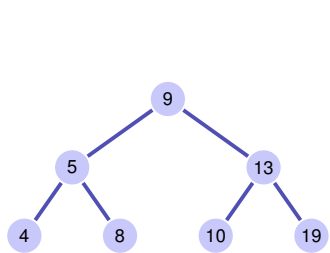
- $\text{Min}(T)$ : Auslesen des Minimums in  $\mathcal{O}(h)$
- $\text{ExtractMin}(T)$ : Auslesen und Entfernen des Minimums in  $\mathcal{O}(h)$
- $\text{List}(T)$ : Ausgeben einer sortierten Liste der Elemente von  $T$
- $\text{Join}(T_1, T_2)$ : Zusammenfügen zweier Bäume mit  $\max(T_1) < \min(T_2)$  in  $\mathcal{O}(n)$ .



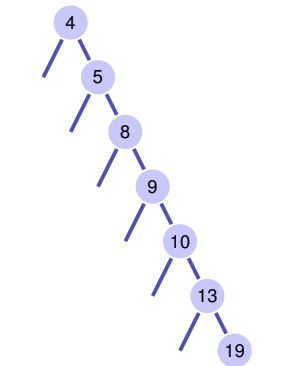
217

218

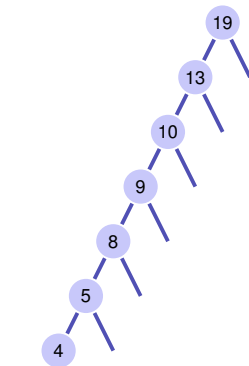
## Degenerierte Suchbäume



Insert 9,5,13,4,8,10,19  
bestmöglich  
balanciert



Insert 4,5,8,9,10,13,19  
Lineare Liste



Insert 19,13,10,9,8,5,4  
Lineare Liste

219

## [Probabilistisch]

Ein Suchbaum, welcher aus einer zufälligen Sequenz von Zahlen erstellt wird hat erwartete Pfadlänge von  $\mathcal{O}(\log n)$ .

Achtung: das gilt nur für Einfügeoperation. Wird der Baum zufällig durch Einfügen und Entfernen gebildet, ist die erwartete Pfadlänge  $\mathcal{O}(\sqrt{n})$ .

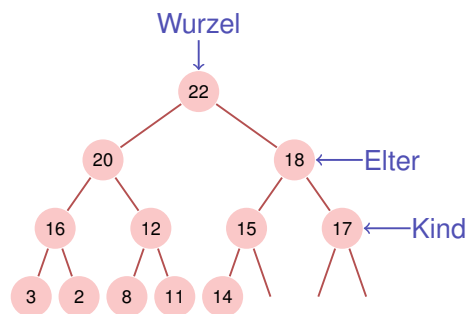
*Balancierte* Bäume stellen beim Einfügen und Entfernen (z.B. durch *Rotationen*) sicher, dass der Baum balanciert bleibt und liefern eine  $\mathcal{O}(\log n)$  Worst-Case-Garantie.

(not shown in class) 220

## [Max-]Heap<sup>8</sup>

Binärer Baum mit folgenden Eigenschaften

- 1 vollständig, bis auf die letzte Ebene
- 2 Lücken des Baumes in der letzten Ebene höchstens rechts.
- 3 **Heap-Bedingung:** Max-(Min-)Heap: Schlüssel eines Kindes kleiner (größer) als der des Elternknotens

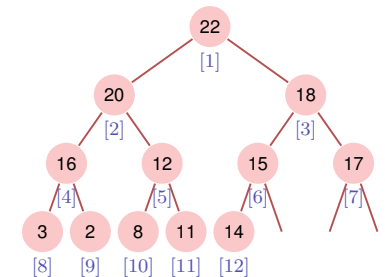
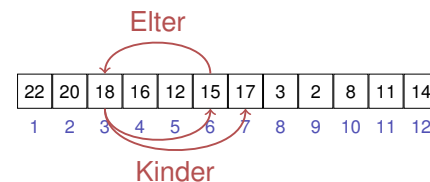


<sup>8</sup>Heap (Datenstruktur), nicht: wie in "Heap und Stack" (Speicherlokation)

## Heap und Array

Baum  $\rightarrow$  Array:

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



Abhängig von Startindex!<sup>9</sup>

<sup>9</sup>Für Arrays, die bei 0 beginnen:  $\{2i, 2i + 1\} \rightarrow \{2i + 1, 2i + 2\}$ ,  $\lfloor i/2 \rfloor \rightarrow \lfloor (i - 1)/2 \rfloor$

221

222

## Höhe eines Heaps

Vollständiger binärer Baum der Höhe<sup>10</sup>  $h$  hat

$$1 + 2 + 4 + 8 + \dots + 2^{h-1} = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

Knoten. Somit gilt für einen Heap der Höhe  $h$ :

$$2^{h-1} - 1 < n \leq 2^h - 1$$

$$\Leftrightarrow 2^{h-1} < n + 1 \leq 2^h$$

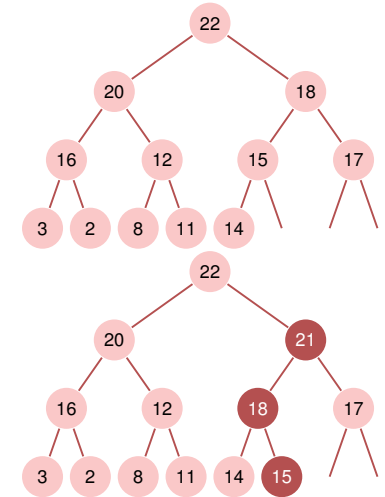
Also insbesondere  $h(n) = \lceil \log_2(n + 1) \rceil$  und  $h(n) \in \Theta(\log n)$ .

<sup>10</sup>Hier: Anzahl Kanten von der Wurzel zu einem Blatt

223

## Einfügen

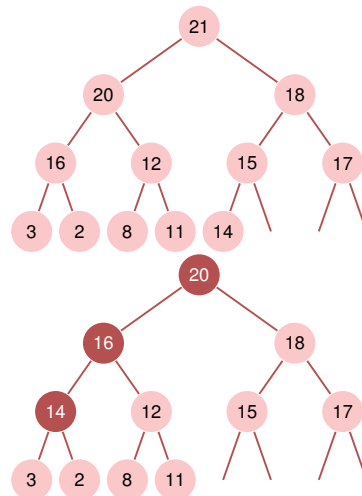
- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.
- Anzahl Operationen im schlechtesten Fall:  $\mathcal{O}(\log n)$



224

## Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).
- Anzahl Operationen im schlechtesten Fall:  $\mathcal{O}(\log n)$



225

## Algorithmus Versickern( $A, i, m$ )

**Input:** Array  $A$  mit Heapstruktur für die Kinder von  $i$ . Letztes Element  $m$ .

**Output:** Array  $A$  mit Heapstruktur für  $i$  mit letztem Element  $m$ .

```

while  $2i \leq m$  do
     $j \leftarrow 2i$ ; //  $j$  linkes Kind
    if  $j < m$  and  $A[j] < A[j + 1]$  then
         $j \leftarrow j + 1$ ; //  $j$  rechtes Kind mit grösserem Schlüssel
    if  $A[i] < A[j]$  then
        swap( $A[i], A[j]$ )
         $i \leftarrow j$ ; // weiter versickern
    else
         $i \leftarrow m$ ; // versickern beendet
    
```

226

## Heap Sortieren

$A[1, \dots, n]$  ist Heap.  
Solange  $n > 1$

- $\text{swap}(A[1], A[n])$
- $\text{Versickere}(A, 1, n - 1)$ ;
- $n \leftarrow n - 1$

		7	6	4	5	1	2
Tauschen	⇒	2	6	4	5	1	7
Versickern	⇒	6	5	4	2	1	7
Tauschen	⇒	1	5	4	2	6	7
Versickern	⇒	5	4	2	1	6	7
Tauschen	⇒	1	4	2	5	6	7
Versickern	⇒	4	1	2	5	6	7
Tauschen	⇒	2	1	4	5	6	7
Versickern	⇒	2	1	4	5	6	7
Tauschen	⇒	1	2	4	5	6	7

227

## Heap erstellen

**Beobachtung:** Jedes Blatt eines Heaps ist für sich schon ein korrekter Heap.

**Folgerung:** Induktion von unten!

228

## Algorithmus HeapSort( $A, n$ )

**Input:** Array  $A$  der Länge  $n$ .

**Output:**  $A$  sortiert.

// Heap Bauen.

**for**  $i \leftarrow n/2$  **downto** 1 **do**

└  $\text{Versickere}(A, i, n)$ ;

// Nun ist  $A$  ein Heap.

**for**  $i \leftarrow n$  **downto** 2 **do**

└  $\text{swap}(A[1], A[i])$   
└  $\text{Versickere}(A, 1, i - 1)$

// Nun ist  $A$  sortiert.

229

## Analyse: Sortieren eines Heaps

Versickere durchläuft maximal  $\log n$  Knoten. An jedem Knoten 2 Schlüsselvergleiche.  $\Rightarrow$  Heap Sortieren kostet im schlechtesten Fall  $2n \log n$  Vergleiche.

Anzahl der Bewegungen vom Heap Sortieren auch  $\mathcal{O}(n \log n)$ .

230



## [Analyse: Heap bauen]

Aufrufe an Versickern:  $n/2$ . Also Anzahl Vergleiche und Bewegungen  $v(n) \in \mathcal{O}(n \log n)$ .

Versickerpfade sind aber im Mittel viel kürzer:

$$v(n) = \sum_{l=0}^{\lfloor \log n \rfloor} \underbrace{2^l}_{\text{Anzahl Heaps auf Level } l} \cdot \underbrace{(\lfloor \log n \rfloor - l)}_{\text{Höhe Heaps auf Level } l} = \sum_{k=0}^{\lfloor \log n \rfloor} 2^{\lfloor \log n \rfloor - k} \cdot k$$

$$\leq \sum_{k=0}^{\lfloor \log n \rfloor} \frac{n}{2^k} \cdot k = n \cdot \sum_{k=0}^{\lfloor \log n \rfloor} \frac{k}{2^k} \in \mathcal{O}(n)$$

mit  $s(x) := \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$  ( $0 < x < 1$ )<sup>11</sup> und  $s(\frac{1}{2}) = 2$

<sup>11</sup>  $f(x) = \frac{1}{1-x} = 1 + x + x^2 + \dots \Rightarrow f'(x) = \frac{1}{(1-x)^2} = 1 + 2x + \dots$

(not shown in class) 231

232

## 11. AVL Bäume

Balancierte Bäume [Ottman/Widmayer, Kap. 5.2-5.2.1, Cormen et al, Kap. Problem 13-3]

### Ziel

Suchen, Einfügen und Entfernen eines Schlüssels in Baum mit  $n$  Schlüssel, welche in zufälliger Reihenfolge eingefügt wurden im Mittel in  $\mathcal{O}(\log_2 n)$  Schritten.

Schlechtester Fall jedoch:  $\Theta(n)$  (degenerierter Baum).

**Ziel:** Verhinderung der Degenerierung. Künstliches, bei jeder Update-Operation erfolgtes Balancieren eines Baumes

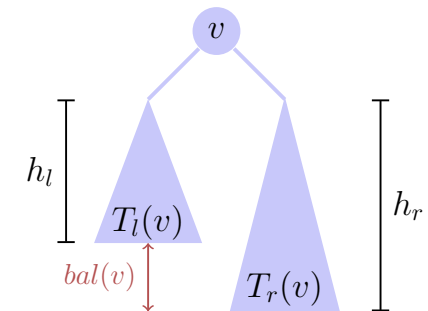
Balancierung: garantiere, dass ein Baum mit  $n$  Knoten stets eine Höhe von  $\mathcal{O}(\log n)$  hat.

**Adelson-Venskii und Landis (1962): AVL-Bäume**

### Balance eines Knotens

Die **Balance** eines Knotens  $v$  ist definiert als die Höhendifferenz seiner beiden Teilbäume  $T_l(v)$  und  $T_r(v)$

$$\text{bal}(v) := h(T_r(v)) - h(T_l(v))$$

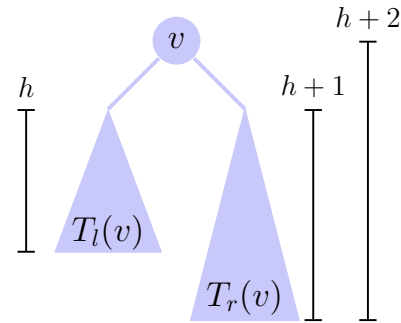


233

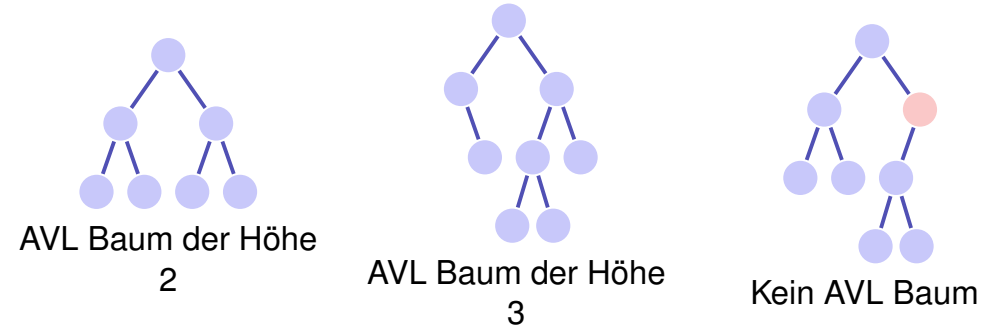
234

## AVL Bedingung

AVL Bedingung: für jeden Knoten  $v$  eines Baumes gilt  $\text{bal}(v) \in \{-1, 0, 1\}$



## (Gegen-)Beispiele



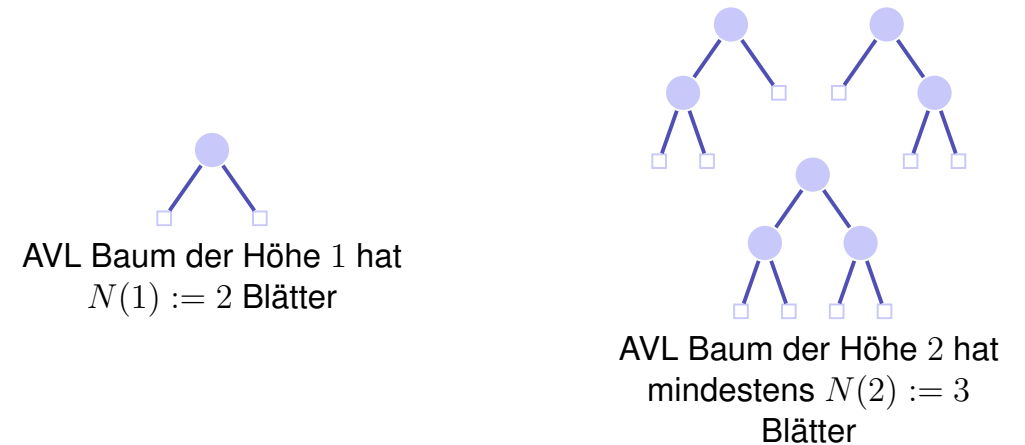
235

236

## Anzahl Blätter

- 1. Beobachtung: Ein Suchbaum mit  $n$  Schlüsseln hat genau  $n + 1$  Blätter. Einfaches Induktionsargument.
  - Der Suchbaum mit  $n = 0$  Schlüsseln hat  $m = 1$  Blätter
  - Wird ein Schlüssel (Knoten) hinzugefügt ( $n \rightarrow n + 1$ ), so ersetzt er ein Blatt und fügt zwei Blätter hinzu ( $m \rightarrow m - 1 + 2 = m + 1$ ).
- 2. Beobachtung: untere Grenze für Anzahl Blätter eines Suchbaums zu gegebener Höhe erlaubt Abschätzung der maximalen Höhe eines Suchbaums zu gegebener Anzahl Schlüssel.

## Untere Grenze Blätter



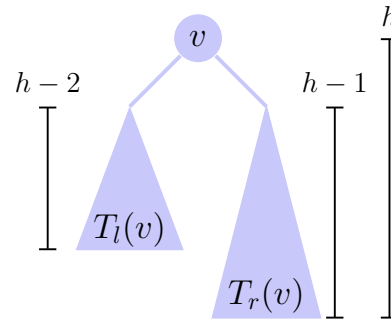
237

238

## Untere Grenze Blätter für $h > 2$

- Höhe eines Teilbaums  $\geq h - 1$ .
  - Höhe des anderen Teilbaums  $\geq h - 2$ .
- Minimale Anzahl Blätter  $N(h)$  ist

$$N(h) = N(h - 1) + N(h - 2)$$



Insgesamt gilt  $N(h) = F_{h+2}$  mit **Fibonacci-Zahlen**  $F_0 := 0, F_1 := 1, F_n := F_{n-1} + F_{n-2}$  für  $n > 1$ .

239

## Fibonacci Zahlen, geschlossene Form

Es gilt

$$F_i = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)$$

mit den Wurzeln  $\phi, \hat{\phi}$  der Gleichung vom goldenen Schnitt  $x^2 - x - 1 = 0$ :

$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1.618$$

$$\hat{\phi} = \frac{1 - \sqrt{5}}{2} \approx -0.618$$

240

## [Fibonacci Zahlen, Induktiver Beweis]

$$F_i \stackrel{!}{=} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i) \quad [*] \quad \left( \phi = \frac{1+\sqrt{5}}{2}, \hat{\phi} = \frac{1-\sqrt{5}}{2} \right).$$

- 1 Klar für  $i = 0, i = 1$ .
- 2 Sei  $i > 2$  und Behauptung [\*] wahr für alle  $F_j, j < i$ .

$$\begin{aligned} F_i &\stackrel{def}{=} F_{i-1} + F_{i-2} \stackrel{[*]}{=} \frac{1}{\sqrt{5}}(\phi^{i-1} - \hat{\phi}^{i-1}) + \frac{1}{\sqrt{5}}(\phi^{i-2} - \hat{\phi}^{i-2}) \\ &= \frac{1}{\sqrt{5}}(\phi^{i-1} + \phi^{i-2}) - \frac{1}{\sqrt{5}}(\hat{\phi}^{i-1} + \hat{\phi}^{i-2}) = \frac{1}{\sqrt{5}}\phi^{i-2}(\phi + 1) - \frac{1}{\sqrt{5}}\hat{\phi}^{i-2}(\hat{\phi} + 1) \end{aligned}$$

( $\phi, \hat{\phi}$  erfüllen  $x + 1 = x^2$ )

$$= \frac{1}{\sqrt{5}}\phi^{i-2}(\phi^2) - \frac{1}{\sqrt{5}}\hat{\phi}^{i-2}(\hat{\phi}^2) = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i).$$

(not shown in class) 241

## Baumhöhe

Da  $|\hat{\phi}| < 1$ , gilt insgesamt

$$N(h) \in \Theta \left( \left( \frac{1 + \sqrt{5}}{2} \right)^h \right) \subseteq \Omega(1.618^h)$$

und somit

$$\begin{aligned} N(h) &\geq c \cdot 1.618^h \\ \Rightarrow h &\leq 1.44 \log_2 n + c'. \end{aligned}$$

Ein AVL Baum ist asymptotisch nicht mehr als 44% höher als ein **perfekt balancierter Baum**.<sup>12</sup>

<sup>12</sup>Ein perfekt balancierter Baum hat Höhe  $\lceil \log_2 n + 1 \rceil$

242

## Einfügen

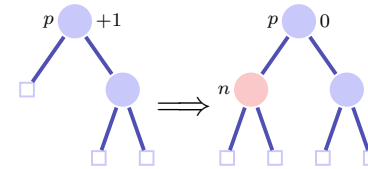
Balancieren

- Speichern der Balance für jeden Knoten
- Baum rebalancieren bei jeder Update-Operation

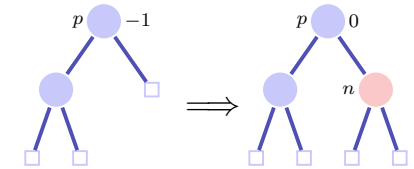
Neuer Knoten  $n$  wird eingefügt:

- Zuerst einfügen wie bei Suchbaum.
- Prüfe die Balance-Bedingung für alle Knoten aufsteigend von  $n$  zur Wurzel.

## Balance am Einfügeort



Fall 1:  $\text{bal}(p) = +1$



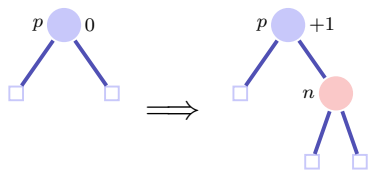
Fall 2:  $\text{bal}(p) = -1$

Fertig in beiden Fällen, denn der Teilbaum ist nicht gewachsen.

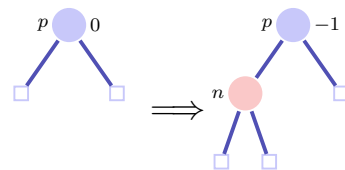
243

244

## Balance am Einfügeort



Fall 3.1:  $\text{bal}(p) = 0$  rechts



Fall 3.2:  $\text{bal}(p) = 0$ , links

In beiden Fällen noch nicht fertig. Aufruf von `upin(p)`.

## upin(p) - Invariante

Beim Aufruf von `upin(p)` gilt, dass

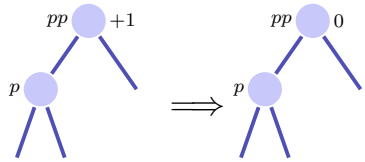
- der Teilbaum ab  $p$  gewachsen ist und
- $\text{bal}(p) \in \{-1, +1\}$

245

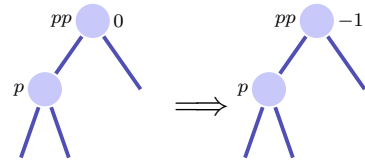
246

## upin(p)

Annahme:  $p$  ist linker Sohn von  $pp$ <sup>13</sup>



Fall 1:  $\text{bal}(pp) = +1$ , fertig.



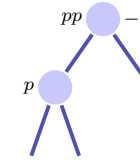
Fall 2:  $\text{bal}(pp) = 0$ , **upin(pp)**

In beiden Fällen gilt nach der Operation die AVL-Bedingung für den Teilbaum ab  $pp$

<sup>13</sup>Ist  $p$  rechter Sohn: symmetrische Fälle unter Vertauschung von  $+1$  und  $-1$

## upin(p)

Annahme:  $p$  ist linker Sohn von  $pp$



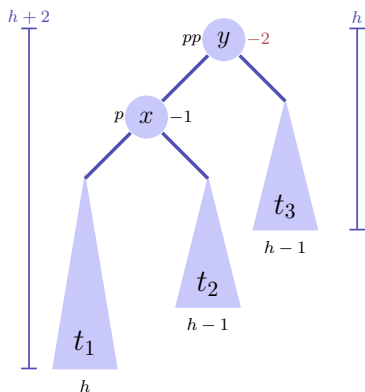
Fall 3:  $\text{bal}(pp) = -1$ ,

Dieser Fall ist problematisch: das Hinzufügen von  $n$  im Teilbaum ab  $pp$  hat die AVL-Bedingung verletzt. Rebalancieren!

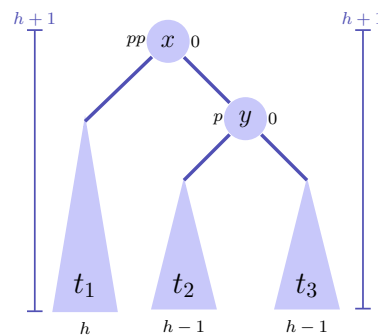
Zwei Fälle  $\text{bal}(p) = -1$ ,  $\text{bal}(p) = +1$

## Rotationen

Fall 1.1  $\text{bal}(p) = -1$ .<sup>14</sup>



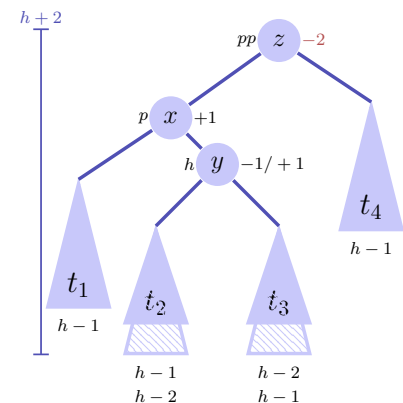
⇒  
Rotation  
nach  
rechts



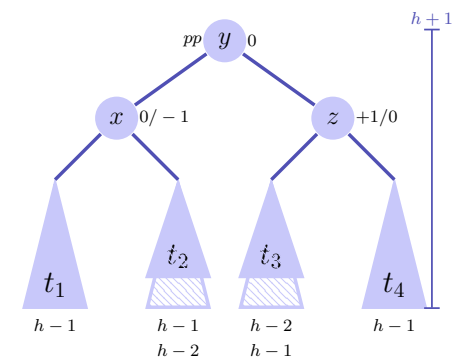
<sup>14</sup> $p$  rechter Sohn  $\Rightarrow \text{bal}(pp) = \text{bal}(p) = +1$ , Linksrotation

## Rotationen

Fall 1.2  $\text{bal}(p) = +1$ .<sup>15</sup>



⇒  
Doppel-  
rotation  
links-  
rechts



<sup>15</sup> $p$  rechter Sohn  $\Rightarrow \text{bal}(pp) = +1$ ,  $\text{bal}(p) = -1$ , Doppelrotation rechts links

## Analyse

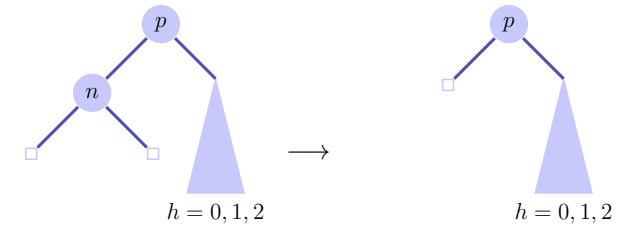
- Höhe des Baumes:  $\mathcal{O}(\log n)$ .
- Einfügen wie beim binären Suchbaum.
- Balancieren durch Rekursion vom Knoten zur Wurzel. Maximale Pfadlänge  $\mathcal{O}(\log n)$ .

Das Einfügen im AVL-Baum hat Laufzeitkosten von  $\mathcal{O}(\log n)$ .

## Löschen

Fall 1: Knoten  $n$  hat zwei Blätter als Kinder Sei  $p$  Elternknoten von  $n$ .  
⇒ Anderer Teilbaum hat Höhe  $h' = 0, 1$  oder  $2$

- $h' = 1$ :  $\text{bal}(p)$  anpassen.
- $h' = 0$ :  $\text{bal}(p)$  anpassen. Aufruf  $\text{upout}(p)$ .
- $h' = 2$ : Rebalancieren des Teilbaumes. Aufruf  $\text{upout}(p)$ .



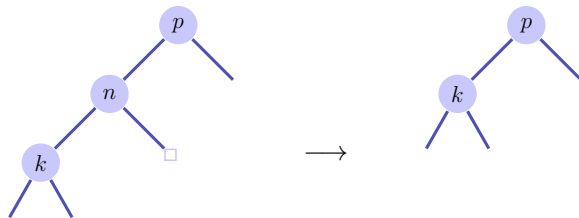
251

252

## Löschen

Fall 2: Knoten  $n$  hat einen inneren Knoten  $k$  als Kind

- Ersetze  $n$  durch  $k$ .  $\text{upout}(k)$



253

254

## Löschen

Fall 3: Knoten  $n$  hat zwei inneren Knoten als Kinder

- Ersetze  $n$  durch symmetrischen Nachfolger.  $\text{upout}(k)$
- Löschen des symmetrischen Nachfolgers wie in Fall 1 oder 2.

## upout (p)

Sei  $pp$  der Elternknoten von  $p$

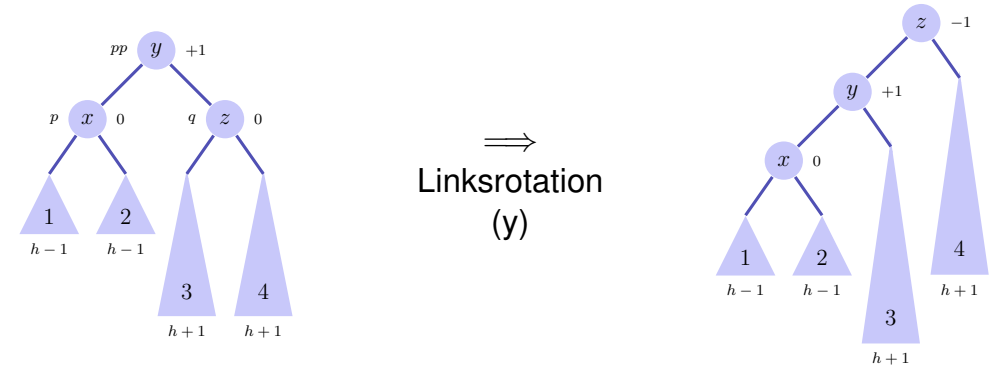
(a)  $p$  linkes Kind von  $pp$

- 1  $\text{bal}(pp) = -1 \Rightarrow \text{bal}(pp) \leftarrow 0$ . **upout (pp)**
- 2  $\text{bal}(pp) = 0 \Rightarrow \text{bal}(pp) \leftarrow +1$ .
- 3  $\text{bal}(pp) = +1 \Rightarrow$  nächste Folien.

(b)  $p$  rechtes Kind von  $pp$ : Symmetrische Fälle unter Vertauschung von  $+1$  und  $-1$ .

## upout (p)

Fall (a).3:  $\text{bal}(pp) = +1$ . Sei  $q$  Bruder von  $p$   
 (a).3.1:  $\text{bal}(q) = 0$ .<sup>16</sup>



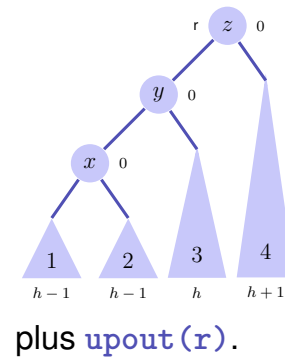
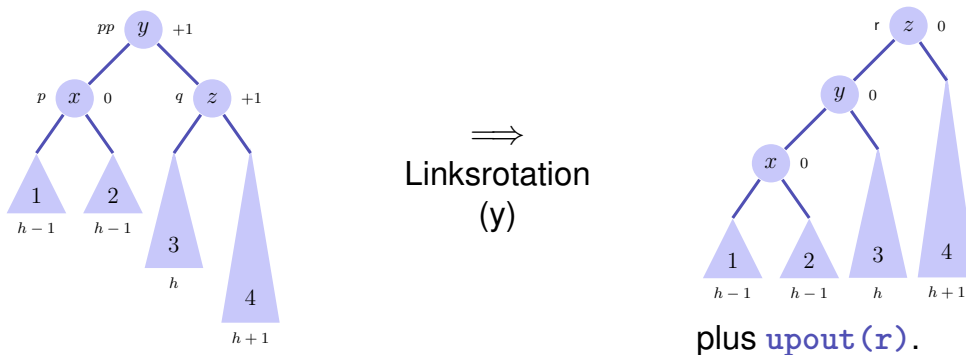
255

<sup>16</sup>(b).3.1:  $\text{bal}(pp) = -1, \text{bal}(q) = -1$ , Rechtsrotation.

256

## upout (p)

Fall (a).3:  $\text{bal}(pp) = +1$ . (a).3.2:  $\text{bal}(q) = +1$ .<sup>17</sup>

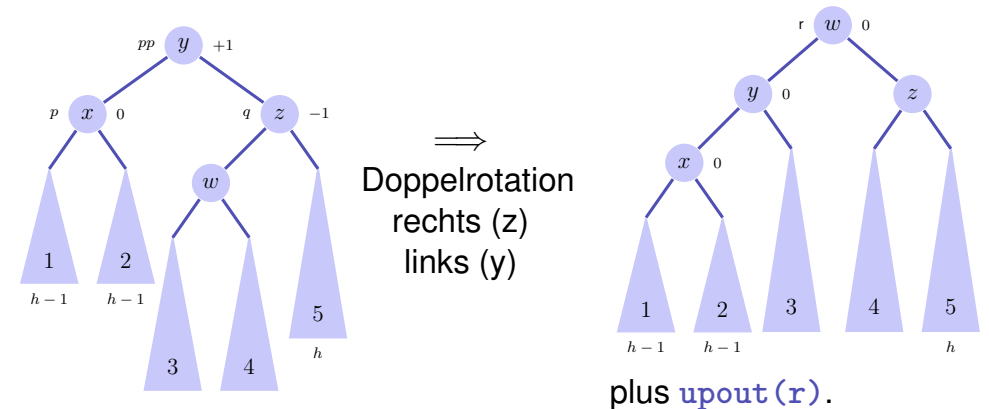


<sup>17</sup>(b).3.2:  $\text{bal}(pp) = -1, \text{bal}(q) = +1$ , Rechtsrotation+upout

257

## upout (p)

Fall (a).3:  $\text{bal}(pp) = +1$ . (a).3.3:  $\text{bal}(q) = -1$ .<sup>18</sup>



plus **upout (r)**.

<sup>18</sup>(b).3.3:  $\text{bal}(pp) = -1, \text{bal}(q) = -1$ , Links-Rechts-Rotation + upout

258

## Zusammenfassung

- AVL-Bäume haben asymptotische Laufzeit von  $\mathcal{O}(\log n)$  (schlechtester Fall) für das Suchen, Einfügen und Löschen von Schlüsseln
- Einfügen und Löschen ist verhältnismässig aufwändig und für kleine Probleme relativ langsam.