Script generated by TTT

Title: TÃ¤ubig: GAD (18.06.2013)

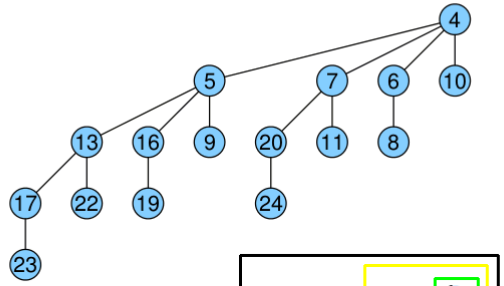
Date: Tue Jun 18 14:31:20 CEST 2013

Duration: 72:41 min

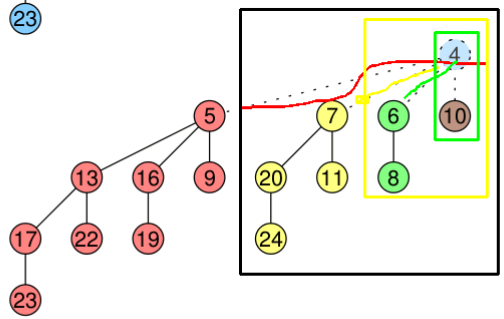
Pages: 35

Binomial-Baum: Löschen der Wurzel (deleteMin)

aus einem B_r



werden $B_{r-1}, \ldots, B_0$



Binomial-Baum: Knotenanzahl

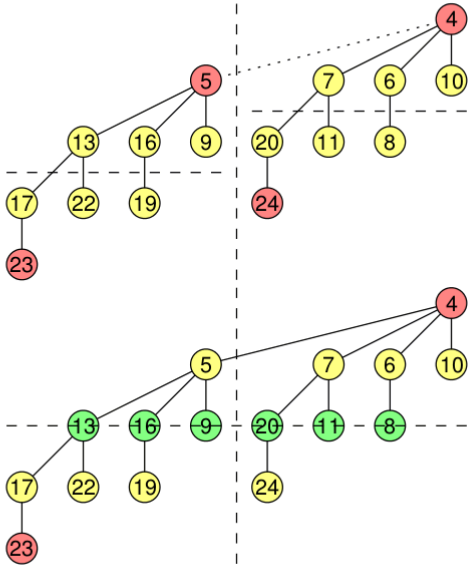
B_r hat auf Level $k \in \{0, \ldots, r\}$ genau $\binom{r}{k}$ Knoten

Warum?

Bei Bau des B_r aus 2 B_{r-1} gilt:

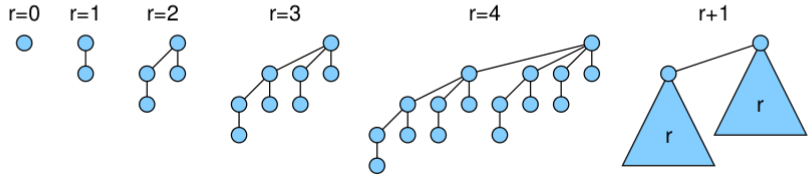
$$\binom{r}{k} = \binom{r-1}{k-1} + \binom{r-1}{k}$$

Insgesamt: B_r hat 2^r Knoten



Binomial-Bäume

Eigenschaften von Binomial-Bäumen:



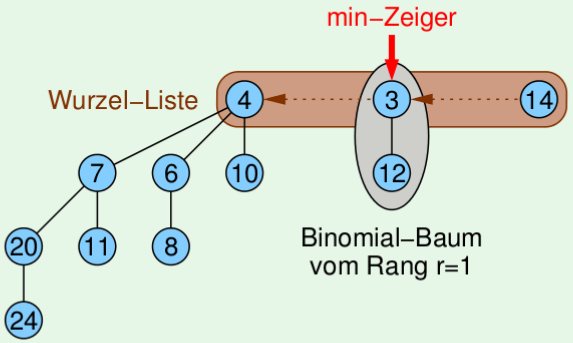
Binomial-Baum vom Rang r

- hat Höhe r (gemessen in Kanten)
- hat maximalen Grad r (Wurzel)
- hat auf Level $\ell \in \{0, \ldots, r\}$ genau $\binom{r}{\ell}$ Knoten
- hat $\sum_{\ell=0}^{r} \binom{r}{\ell} = 2^r$ Knoten
- zerfällt bei Entfernen der Wurzel in r Binomial-Bäume von Rang 0 bis $r-1$

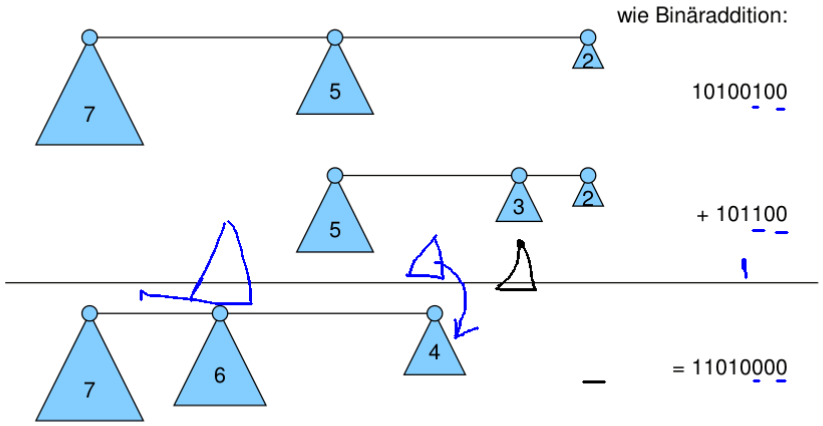
Binomial Heap

Beispiel

Korrekter Binomial Heap:



Merge von zwei Binomial Heaps



Aufwand für Merge: $O(\log n)$

Binomial Heaps

B_i: Binomial-Baum mit Rang i

Operationen:

- merge: $O(\log n)$
- insert(e): Merge mit B_0, Zeit $O(\log n)$
- min(): spezieller Zeiger, Zeit $O(1)$
- deleteMin():
  sei das Minimum in B_i,
  durch Löschen der Wurzel zerfällt der Binomialbaum in $B_0, \ldots, B_{i-1}$
  Merge mit dem restlichen Binomial Heap kostet $O(\log n)$

Binomial Heaps

Weitere Operationen:

- decreaseKey(h, k): siftUp-Operation in Binomial-Baum für das Element, auf das h zeigt,
  dann ggf. noch min-Zeiger aktualisieren

  Zeit: $O(\log n)$

- remove(h): Sei e das Element, auf das h zeigt.
  Setze prio(e) $= -\infty$ und wende siftUp-Operation auf e an bis e in der Wurzel, dann weiter wie bei deleteMin

  Zeit: $O(\log n)$

Bessere Laufzeit mit Fibonacci-Heaps

Fibonacci-Heaps

Verbesserung von Binomial Heaps mit folgenden Kosten:

- min, insert, merge: $O(1)$ (worst case)
- decreaseKey: $O(1)$ (amortisiert)
- deleteMin, remove: $O(\log n)$ (amortisiert)

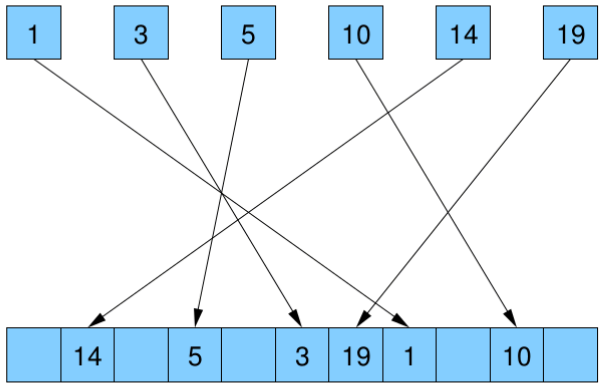
Wir werden darauf bei den Graph-Algorithmen zurückgreifen.

Übersicht

Übersicht

Vergleich Wörterbuch / Suchstruktur

- Wörterbuch effizient über Hashing realisierbar



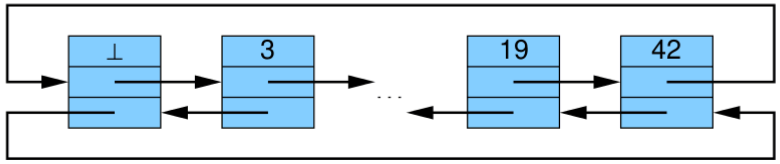
- Hashing zerstört die Ordnung auf den Elementen

$\Rightarrow$ keine effiziente locate-Operation

$\Rightarrow$ keine Intervallanfragen

Suchstruktur

Erster Ansatz: sortierte Liste



Problem:

- insert, remove, locate kosten im worst case $\Theta(n)$ Zeit

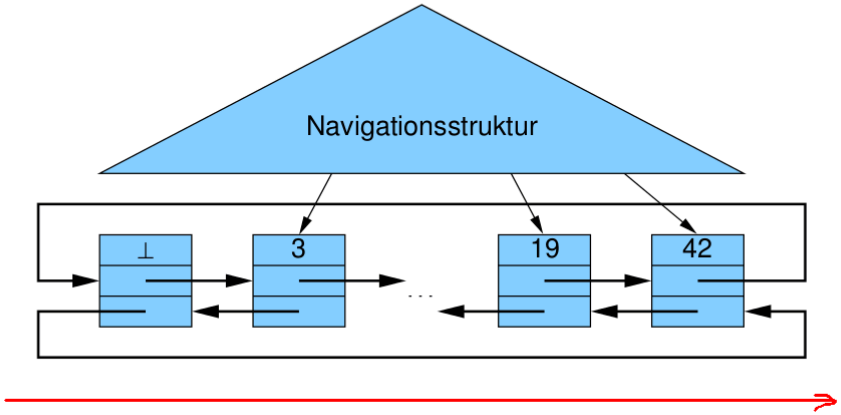
Einsicht:

- wenn locate effizient implementierbar, dann auch die anderen Operationen

Suchstruktur

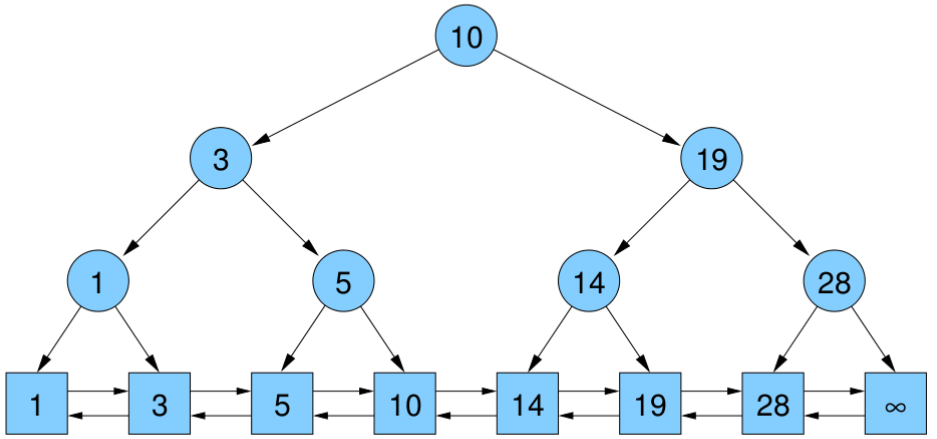
Idee:

- füge Navigationsstruktur hinzu, die locate effizient macht



Übersicht

Binärer Suchbaum (ideal)

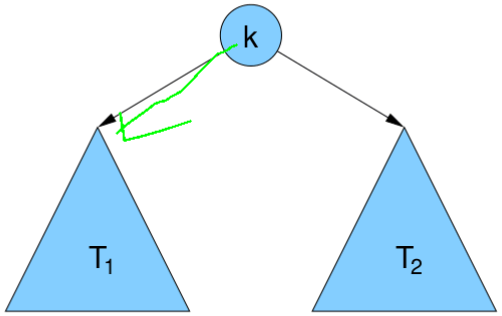


Binärer Suchbaum

Suchbaum-Regel:

Für alle Schlüssel
k_1 in T_1 und k_2 in T_2:

$$k_1 \leq k < k_2$$



locate-Strategie:

- Starte in Wurzel des Suchbaums
- Für jeden erreichten Knoten v:

  Falls key(v) $\geq k_{\text{gesucht}}$, gehe zum linken Kind von v,
  sonst gehe zum rechten Kind

Binärer Suchbaum

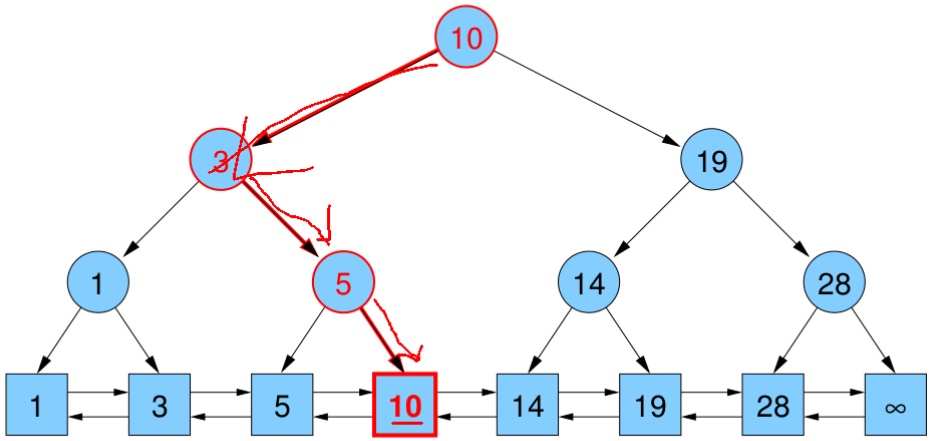
Formal: für einen Baumknoten v sei

- key(v) der Schlüssel von v
- $d(v)$ der Ausgangsgrad (Anzahl Kinder) von v

- Suchbaum-Invariante: $k_1 \leq k < k_2$
  (Sortierung der linken und rechten Nachfahren)
- Grad-Invariante: $d(v) \leq 2$
  (alle Baumknoten haben höchstens 2 Kinder)
- Schlüssel-Invariante:
  (Für jedes Element e in der Liste gibt es *genau einen*
  Baumknoten v mit key(v) == key(e))

Binärer Suchbaum / locate

locate(9)



Binärer Suchbaum / insert, remove

Strategie:

- insert(e):
  - erst wie locate(key(e)) bis Element e' in Liste erreicht
  - falls key(e') > key(e):
    füge e vor e' ein, sowie ein neues Suchbaumblatt für e und e' mit
    key(e) als Splitter Key, so dass Suchbaum-Regel erfüllt

Binärer Suchbaum / insert, remove

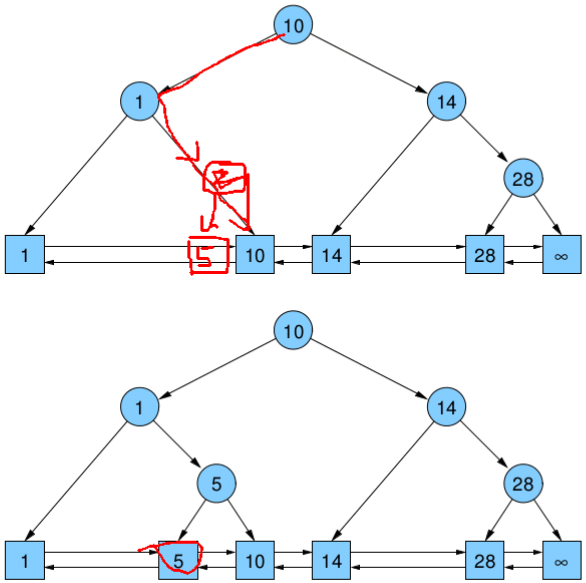
Strategie:

- insert(e):
  - erst wie locate(key(e)) bis Element e' in Liste erreicht
  - falls key(e') > key(e):
    füge e vor e' ein, sowie ein neues Suchbaumblatt für e und e' mit key(e) als Splitter Key, so dass Suchbaum-Regel erfüllt

- remove(k):
  - erst wie locate(k) bis Element e in Liste erreicht
  - falls key(e) $= k$, lösche e aus Liste und Vater v von e aus Suchbaum und
  - setze in dem Baumknoten w mit key(w) $= k$ den neuen Wert key(w) = key(v)

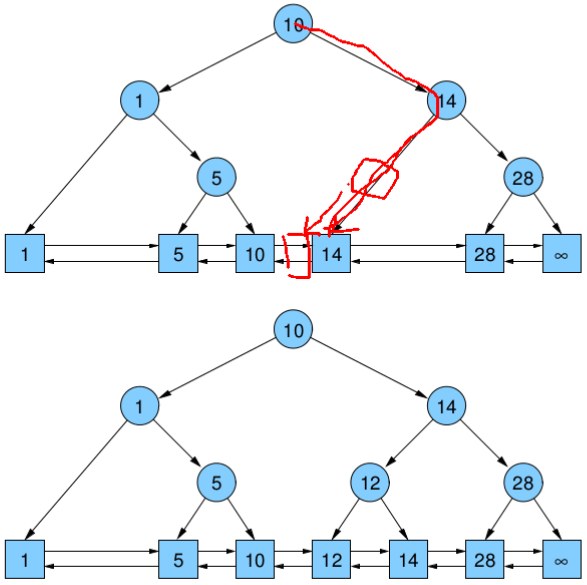
Binärer Suchbaum / insert, remove

insert(5)



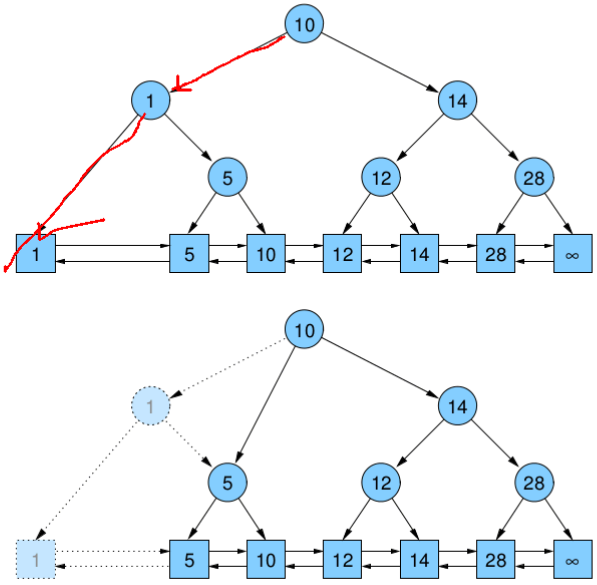
Binärer Suchbaum / insert, remove

insert(12)



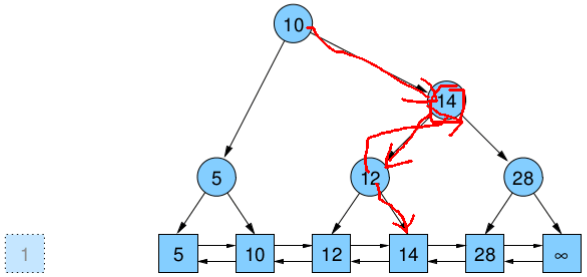
Binärer Suchbaum / insert, remove

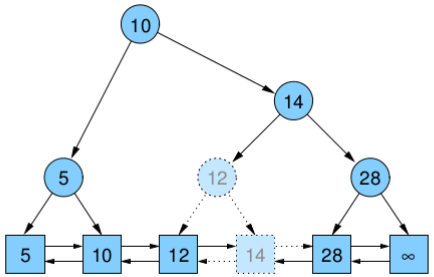
remove(1)



Binärer Suchbaum / insert, remove

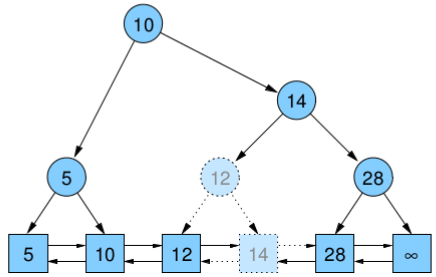
remove(14)

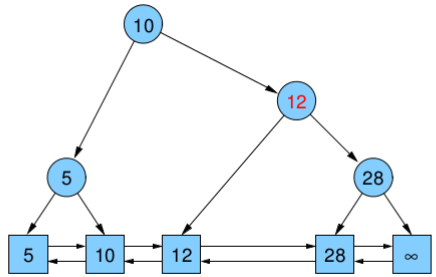




Binärer Suchbaum / insert, remove

remove(14)





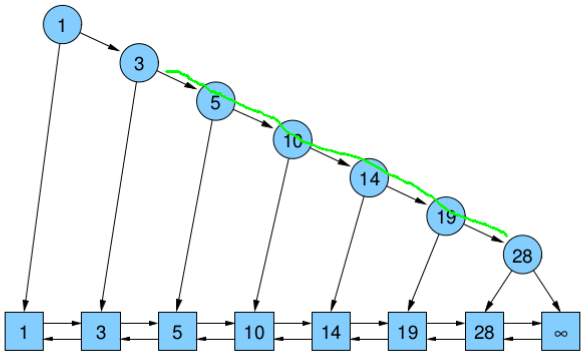
Binärer Suchbaum / worst case

Problem:

- Baumstruktur kann zur Liste entarten
- Höhe des Baums kann linear in der Anzahl der Elemente werden

$\Rightarrow$ locate kann im worst case Zeitaufwand $\Theta(n)$ verursachen

Beispiel: Zahlen werden in sortierter Reihenfolge eingefügt



Übersicht

AVL-Bäume

Balancierte binäre Suchbäume

Strategie zur Lösung des Problems:

- Balancierung des Baums

G. M. **A**delson-**V**elsky & E. M. **L**andis (1962):

- Beschränkung der Höhenunterschiede für Teilbäume auf $[-1, 0, +1]$

$\Rightarrow$ führt nicht unbedingt zu einem idealen unvollständigen Binärbaum (wie wir ihn von array-basierten Heaps kennen), aber zu einem hinreichenden Gleichgewicht

AVL-Bäume

Balancierte binäre Suchbäume

Worst Case: Fibonacci-Baum

- Laufzeit der Operation hängt von der Baumhöhe ab
- Was ist die schlimmste Höhe bei gegebener Anzahl von Elementen?
- bzw: Wieviel Elemente hat ein Baum bei gegebener Höhe h mindestens?

$\Rightarrow$ Für mindestens ein Kind hat der Unterbaum Höhe $h - 1$. Im worst case hat das andere Kind Höhe $h - 2$ (kleiner geht nicht wegen Höhendifferenzbeschränkung)

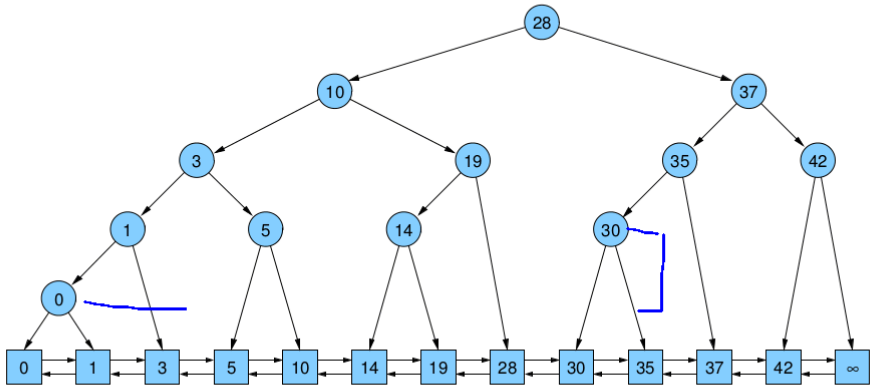
$\Rightarrow$ Anzahl der (inneren) Knoten entspricht den Fibonacci-Zahlen:

$$F_n = F_{n-1} + F_{n-2}$$

AVL-Bäume

Balancierte binäre Suchbäume

Worst Case: Fibonacci-Baum



AVL-Bäume

Balancierte binäre Suchbäume

Worst Case: Fibonacci-Baum

- Fibonacci-Baum der Stufe 0 ist der leere Baum
- Fibonacci-Baum der Stufe 1 ist ein einzelner Knoten
- Fibonacci-Baum der Stufe $h + 1$ besteht aus einer Wurzel, deren Kinder Fibonacci-Bäume der Stufen h und $h - 1$ sind

Explizite Formel:

$$F_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right]$$

$\Rightarrow$ Die Anzahl der Elemente ist exponentiell in der Höhe bzw. die Höhe ist logarithmisch in der Anzahl der Elemente.

AVL-Bäume

Balancierte binäre Suchbäume

Operationen auf einem AVL-Baum:

- insert und remove können zunächst zu Binärbäumen führen, die die Balance-Bedingung für die Höhendifferenz der Teilbäume verletzen

$\Rightarrow$ Teilbäume müssen umgeordnet werden, um das Kriterium für AVL-Bäume wieder zu erfüllen (Rebalancierung / Rotation)

- Dazu wird an jedem Knoten die Höhendifferenz der beiden Unterbäume vermerkt ($-1, 0, +1$, mit 2 Bit / Knoten)

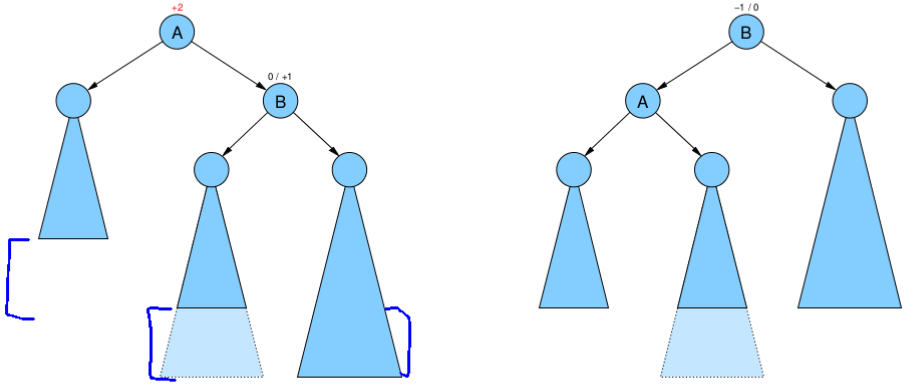
(Hausaufgabe . . .)

- Operationen locate, insert und remove haben Laufzeit $O(\log n)$

AVL-Bäume

Balancierte binäre Suchbäume

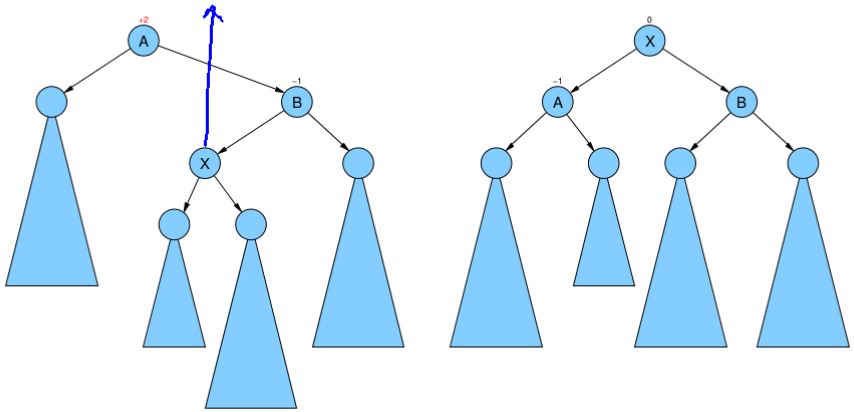
Einfachrotation:



AVL-Bäume

Balancierte binäre Suchbäume

Doppelrotation:



Übersicht