Script generated by TTT

Title: Täubig: GAD (03.06.2014)

Date: Tue Jun 03 13:53:18 CEST 2014

Duration: 122:14 min

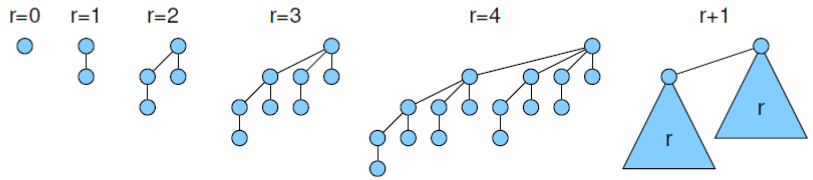
Pages: 45

Übersicht

Binomial-Bäume

Binomial Heaps bestehen aus Binomial-Bäumen

- Form-Invariante:



- Heap-Invariante:

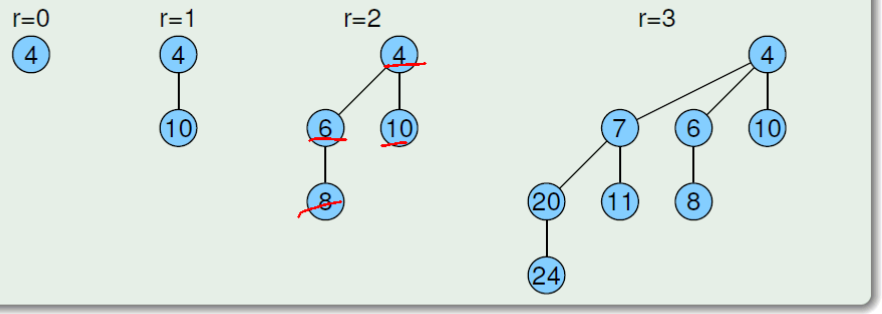
$$\text{prio(Vater)} \leq \text{prio(Kind)}$$

Elemente der Priority Queue werden in Heap Items gespeichert, die eine feste Adresse im Speicher haben und damit als Handles dienen können (im Gegensatz zu array-basierten Binärheaps)

Binomial-Bäume

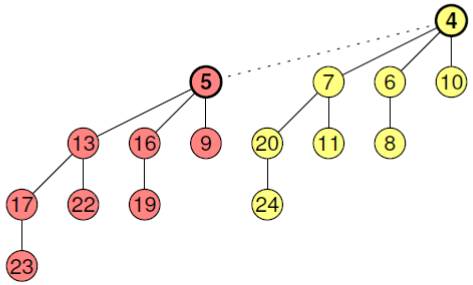
Beispiel

Korrekte Binomial-Bäume:

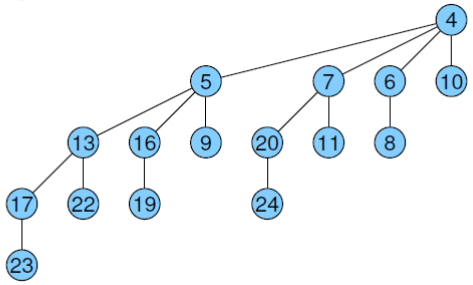


Binomial-Baum: Merge

Wurzel mit größerem Wert wird neues Kind der Wurzel mit kleinerem Wert! (Heap-Bedingung)

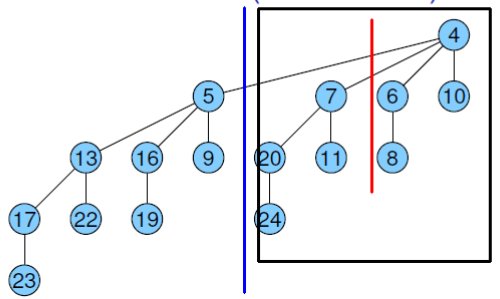


aus zwei B_{r-1} wird ein B_r

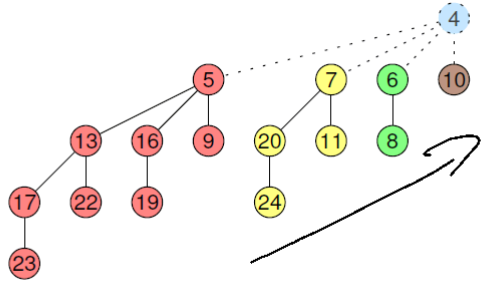


Binomial-Baum: Löschen der Wurzel (deleteMin)

aus einem B_r



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



Binomial-Baum: Knotenanzahl

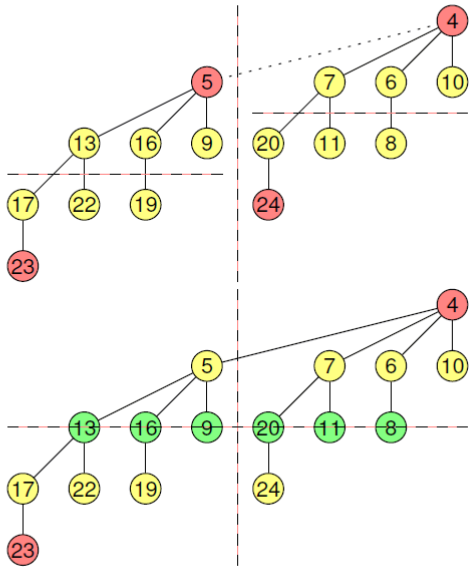
B_r hat auf Level $k \in \{0, \ldots, r\}$ genau $\binom{r}{k}$ Knoten

Warum?

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

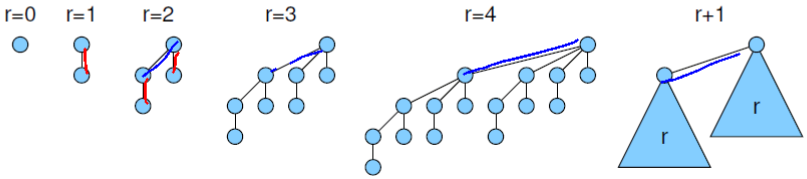
$$\binom{r}{k} = \binom{r-1}{k-1} + \binom{r-1}{k}$$

Insgesamt: B_r hat 2^r Knoten



Binomial-Bäume

Eigenschaften von Binomial-Bäumen:



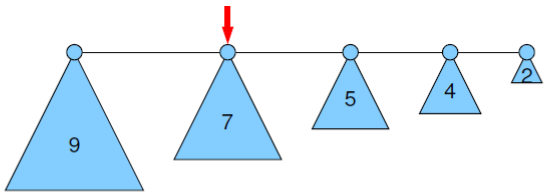
Binomial-Baum vom Rang r

- 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

Binomial Heap:

- verkettete Liste von Binomial-Bäumen
- pro Rang maximal 1 Binomial-Baum
- Zeiger auf Wurzel mit minimalem Prioritätswert

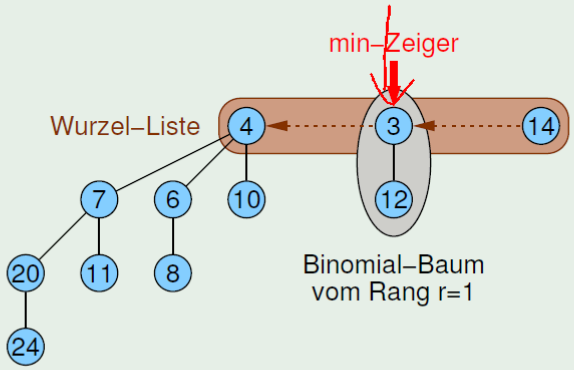


(Zahlen: Ränge)

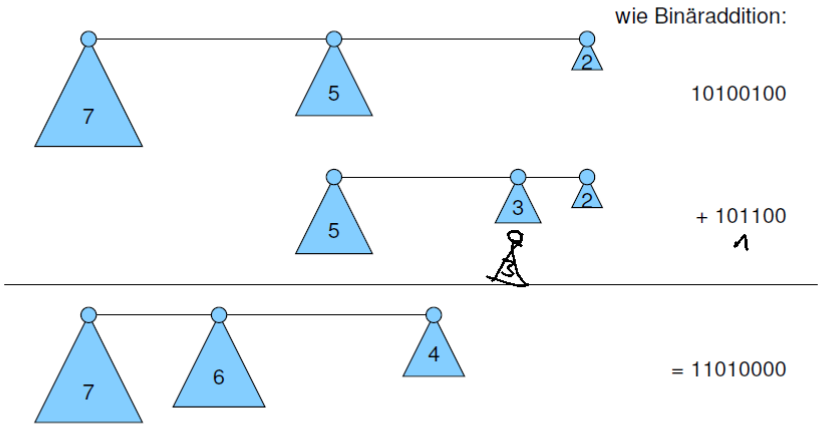
Binomial Heap

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

Vergleich Wörterbuch / Suchstruktur

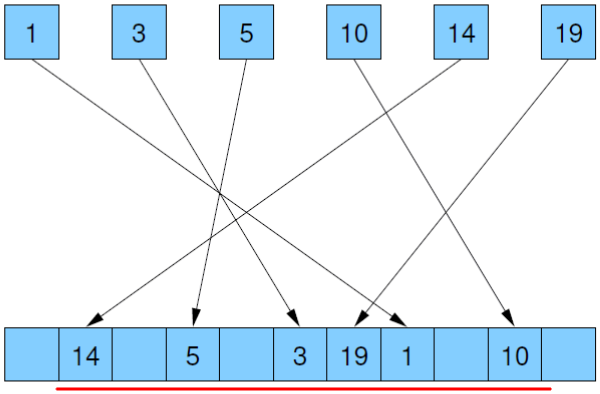
- S: Menge von Elementen
- Element e wird identifiziert über eindeutigen Schlüssel key(e)

Operationen:

- S.insert(Elem e): $S = S \cup \{e\}$
- S.remove(Key k): $S = S \setminus \{e\}$,
  wobei e das Element mit key(e) $== k$ ist
- S.find(Key k): (Wörterbuch)
  gibt das Element $e \in S$ mit key(e) $== k$ zurück, falls es existiert, sonst null
- S.locate(Key k): (Suchstruktur)
  gibt das Element $e \in S$ mit minimalem Schlüssel key(e) zurück, für das key(e) $\geq k$

Vergleich Wörterbuch / Suchstruktur

- Wörterbuch effizient über Hashing realisierbar



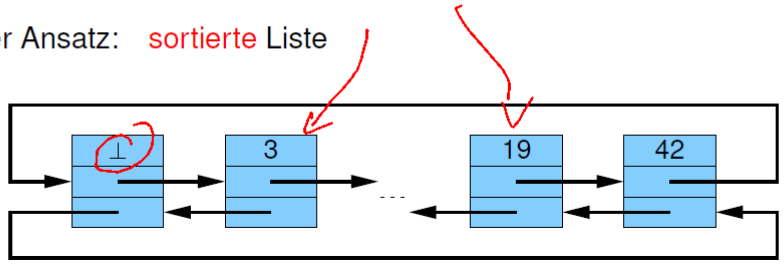
- Hashing zerstört die Ordnung auf den Elementen

$\Rightarrow$ keine effiziente locate-Operation

$\Rightarrow$ keine Intervallanfragen

Suchstruktur

Erster Ansatz: sortierte Liste



Problem:

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

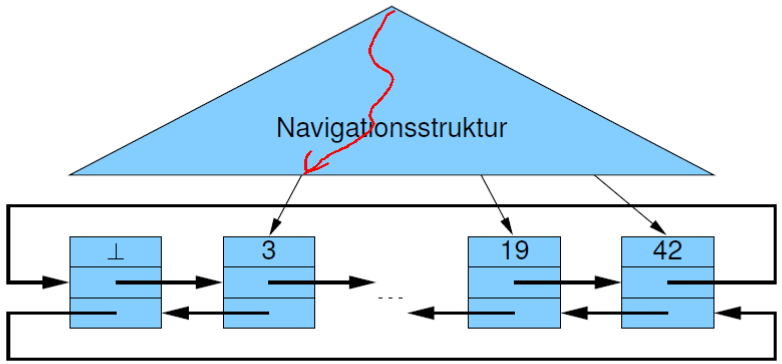
Einsicht:

- wenn locate effizient implementierbar, dann auch die anderen Operationen

Suchstruktur

Idee:

- füge Navigationsstruktur hinzu, die locate effizient macht

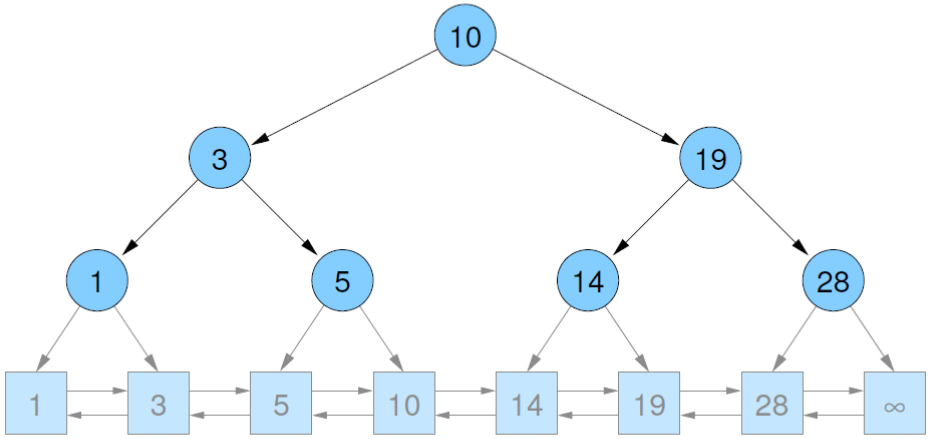


Suchbäume

extern Baumknoten enthalten nur Navigationsinformationen
Nutzdaten sind in den Blättern gespeichert.
(hier: mittels Zeiger auf Elemente einer sortierten Liste)

intern Nutzdaten sind schon an den inneren Knoten gespeichert

Binärer Suchbaum (ideal)

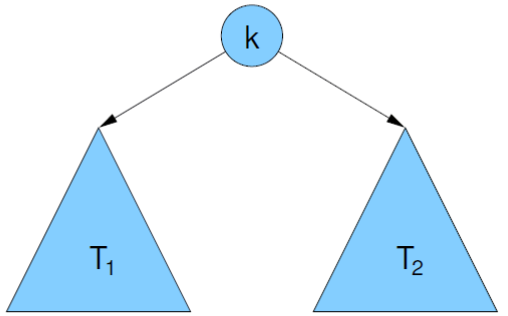


Binärer Suchbaum

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

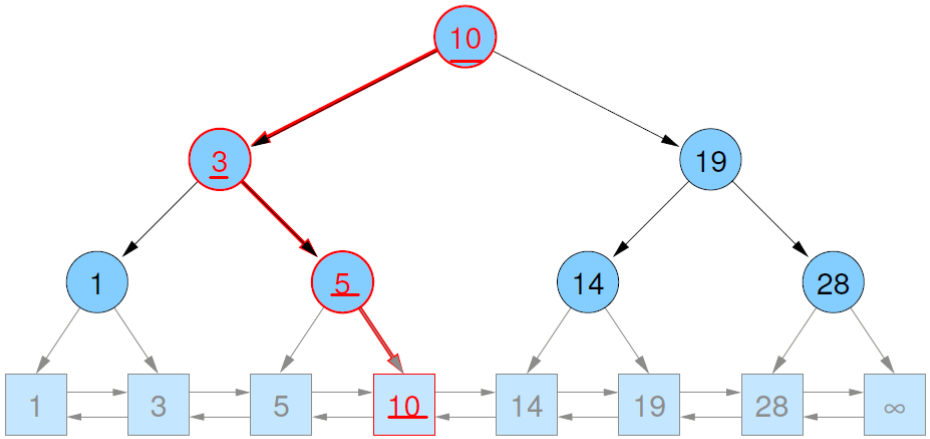
Formal: für einen Baumknoten v sei

- 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

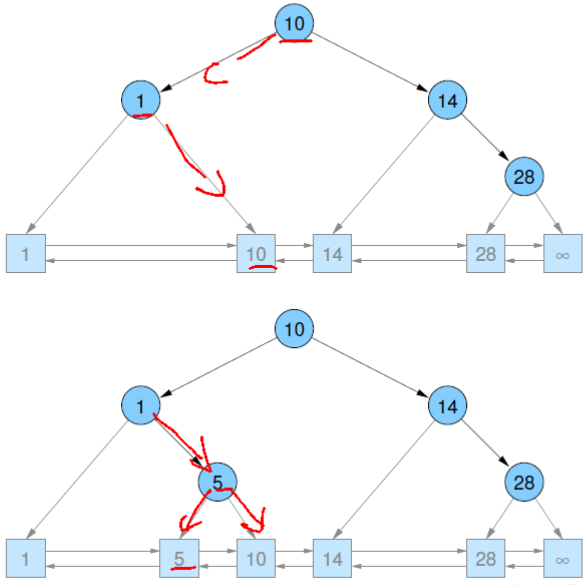
Binärer Suchbaum / insert, remove

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)

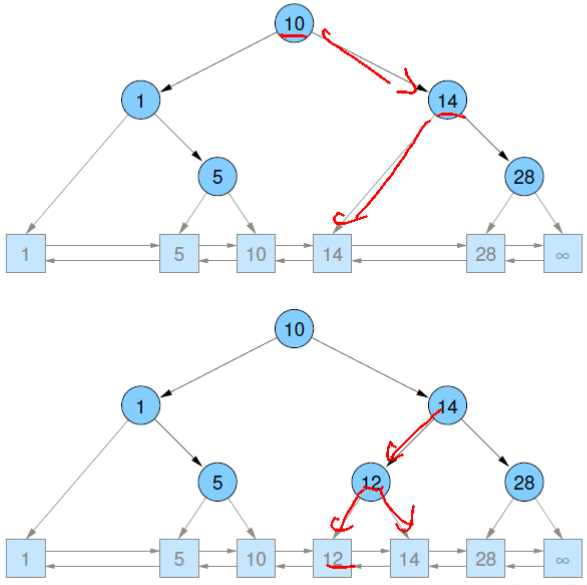
Binärer Suchbaum / insert, remove

insert(5)



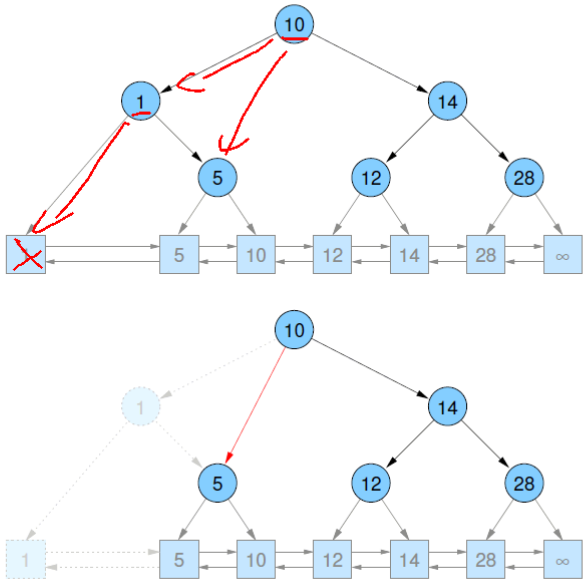
Binärer Suchbaum / insert, remove

insert(12)



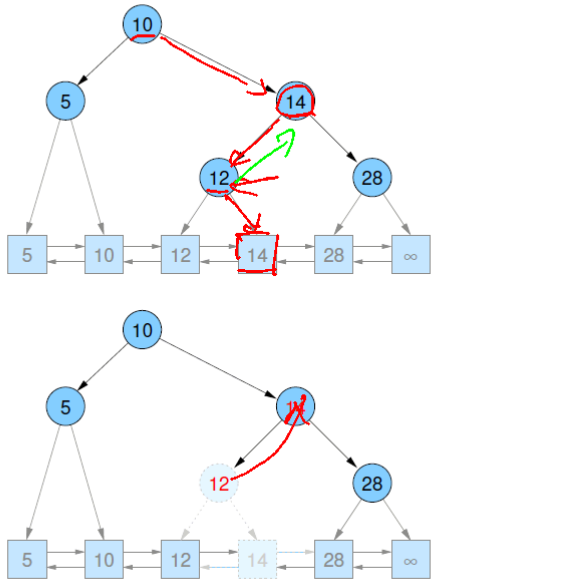
Binärer Suchbaum / insert, remove

remove(1)



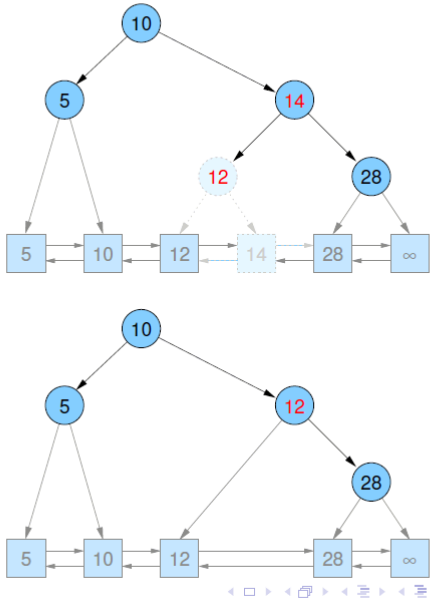
Binärer Suchbaum / insert, remove

remove(14)



Binärer Suchbaum / insert, remove

remove(14)



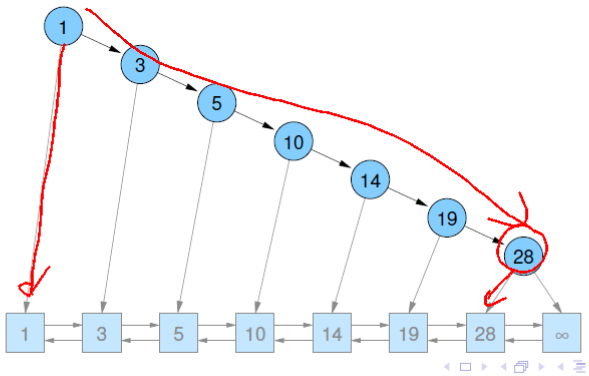
Binärer Suchbaum / worst case

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

Suchstrukturen AVL-Bäume

Übersicht

8 Suchstrukturen
- Allgemeines
- Binäre Suchbäume
- AVL-Bäume
- (a, b)-Bäume

H. Täubig (TUM) GAD SS'14 328

AVL-Bäume

Balancierte binäre Suchbäume

Strategie zur Lösung des Problems:

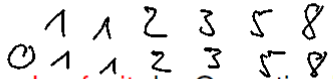
- Balancierung des Baums

Georgy M. **A**delson-**V**elsky & Evgenii 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: Worst Case / Fibonacci-Baum

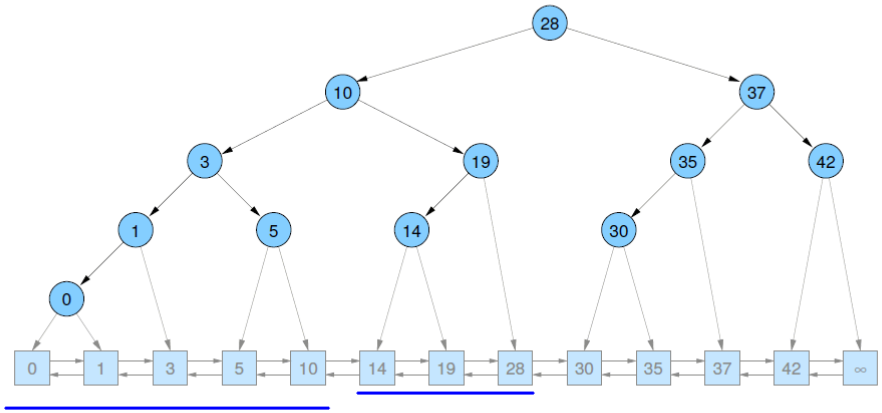


- Laufzeit der Operation hängt von der Baumhöhe ab
- Was ist die größte Höhe bei gegebener Anzahl von Elementen?
- bzw: Wieviel Elemente hat ein Baum mit Höhe h mindestens?
- Für mindestens ein Kind hat der Unterbaum Höhe $h - 1$.
  Worst case: Unterbaum am anderen Kind hat Höhe $h - 2$ (kleiner geht nicht wegen Höhendifferenzbeschränkung).

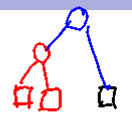
$\Rightarrow$ Anzahl der Blätter entspricht den Fibonacci-Zahlen:

$$F_k = F_{k-1} + F_{k-2}$$

AVL-Bäume: Worst Case / Fibonacci-Baum



AVL-Bäume: Worst Case / Fibonacci-Baum



- Fibonacci-Baum der Höhe 0: Baum bestehend aus einem Blatt
- Fibonacci-Baum der Höhe 1: ein innerer Knoten mit 2 Blättern
- Fibonacci-Baum der Höhe $h+1$ besteht aus einer Wurzel, deren Kinder Fibonacci-Bäume der Höhen h und $h-1$ sind

Explizite Darstellung der Fibonacci-Zahlen mit Binet-Formel:

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

- Baum der Höhe h hat F_{h+2} Blätter bzw. $F_{h+2}-1$ innere Knoten

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

AVL-Bäume: Operationen

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)

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

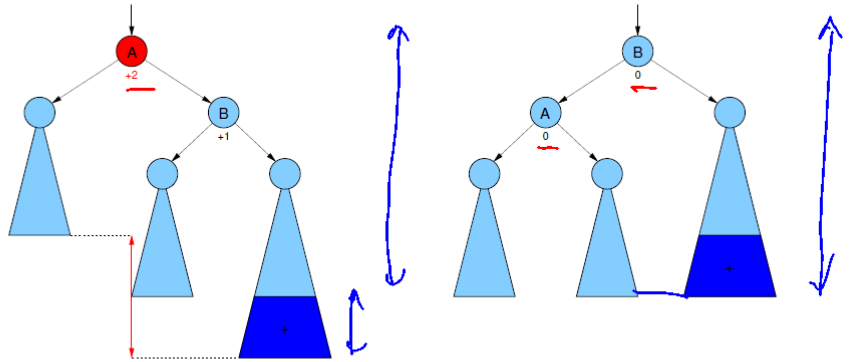
AVL-Bäume: insert

- Suche Knoten, an den das neue Blatt angehängt wird
- An diesem Knoten ändert sich die Höhendifferenz um ± 1 (linkes oder rechtes Blatt)
- gehe nun rückwärts zur Wurzel, aktualisiere die jeweilige Höhendifferenz und rebalanciere falls notwendig

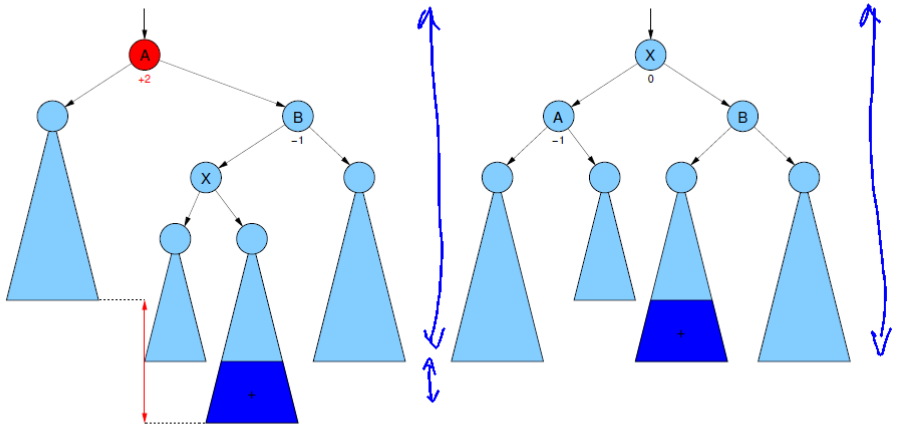
- Differenz 0: Wert war vorher ± 1, Höhe unverändert, also aufhören
- Differenz ± 1: Wert war vorher 0, Höhe ist jetzt um 1 größer, Höhendifferenz im Vaterknoten anpassen und dort weitermachen
- Differenz ± 2: Rebalancierung erforderlich, Einfach- oder Doppelrotation abhängig von Höhendifferenz an den Kindknoten

  danach Höhe wie zuvor, also aufhören

AVL-Bäume: Einfachrotation nach insert



AVL-Bäume: Doppelrotation nach insert



AVL-Bäume: remove

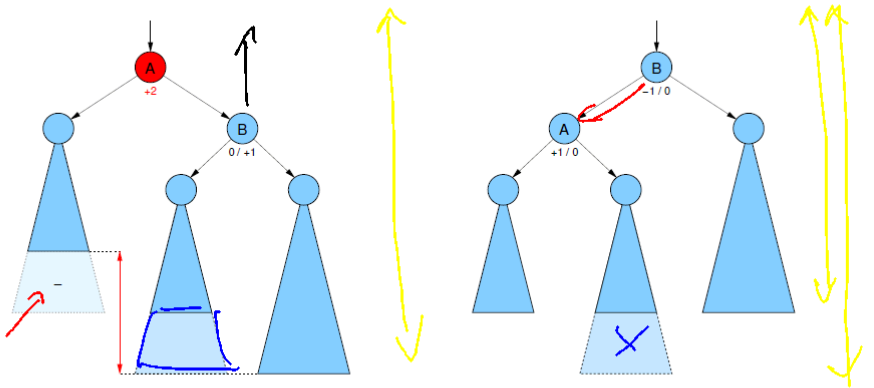
- Suche Knoten v, der entfernt werden soll
- Falls v ein Blatt ist oder genau 1 Kind hat, lösche v bzw. ersetze v durch sein Kind, aktualisiere Höhendifferenz des Vaterknotens und fahre dort fort.
- Falls v zwei Blätter hat, vertausche v mit dem rechtesten Knoten im linken Unterbaum (das nächstkleinere Element, das direkt vor v kommt) und lösche v dort

  v hat dort höchstens 1 (linkes) Kind, nun wie im ersten Fall

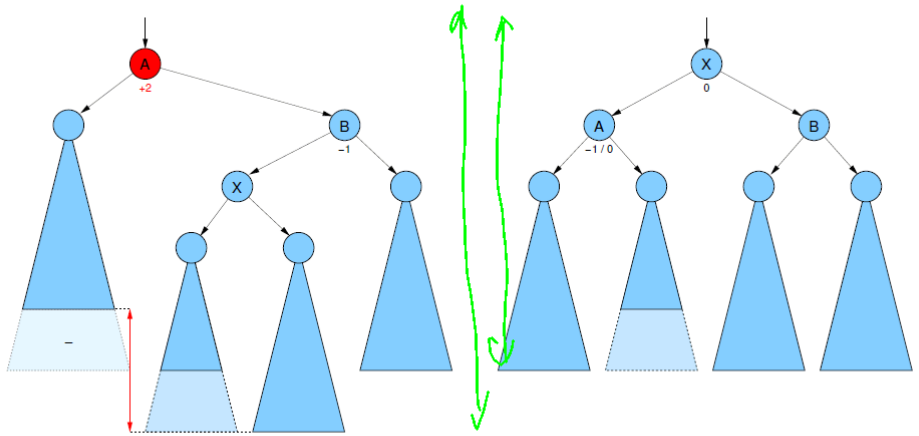
- Differenz 0: Wert war vorher ± 1, Höhe ist jetzt um 1 kleiner, Höhendifferenz im Vaterknoten anpassen und dort weitermachen
- Differenz ± 1: Wert war vorher 0, Höhe unverändert, also aufhören
- Differenz ± 2: Rebalancierung erforderlich, Einfach- oder Doppelrotation abhängig von Höhendifferenz an den Kindknoten

  falls notwendig Höhendifferenz im Vaterknoten anpassen und dort weitermachen

AVL-Bäume: Einfachrotation nach remove



AVL-Bäume: Doppelrotation nach remove



Übersicht