Übersicht

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

Übersicht

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

Vergleich Wörterbuch / Suchstruktur

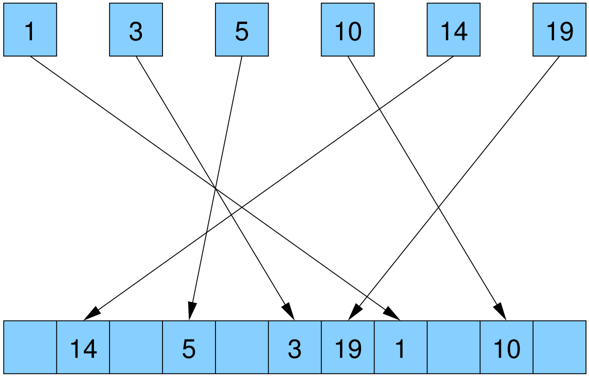
- S: Menge von Elementen
- Element e wird identifiziert über eindeutigen Schlüssel key(e)

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



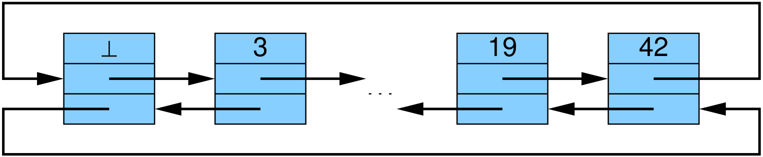
- Hashing zerstört die Ordnung auf den Elementen

$\Rightarrow$ keine effiziente locate-Operation

$\Rightarrow$ keine Intervallanfragen

Suchstruktur

Erster Ansatz: sortierte Liste



Problem:

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

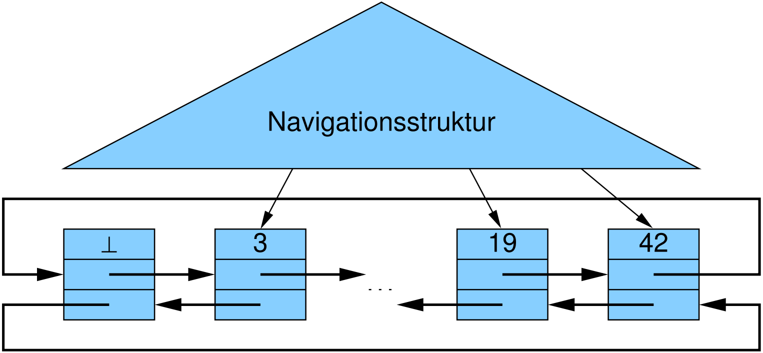
Einsicht:

- wenn locate effizient implementierbar, dann auch die anderen Operationen

Suchstruktur

Idee:

- füge Navigationsstruktur hinzu, die locate effizient macht



Suchbäume

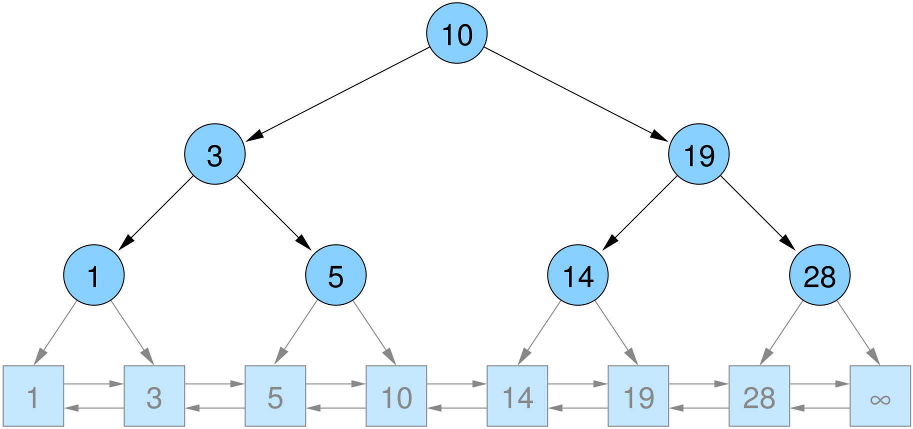
extern Baumknoten enthalten nur Navigationsinformationen
Nutzdaten sind in den Blättern gespeichert.
(hier: mittels Zeiger auf Elemente einer sortierten Liste)

intern Nutzdaten sind schon an den inneren Knoten gespeichert

Übersicht

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

Binärer Suchbaum (ideal)

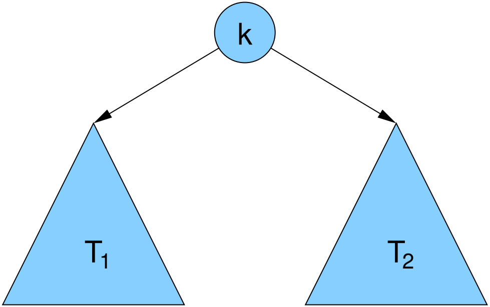


Binärer Suchbaum

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

Binärer Suchbaum

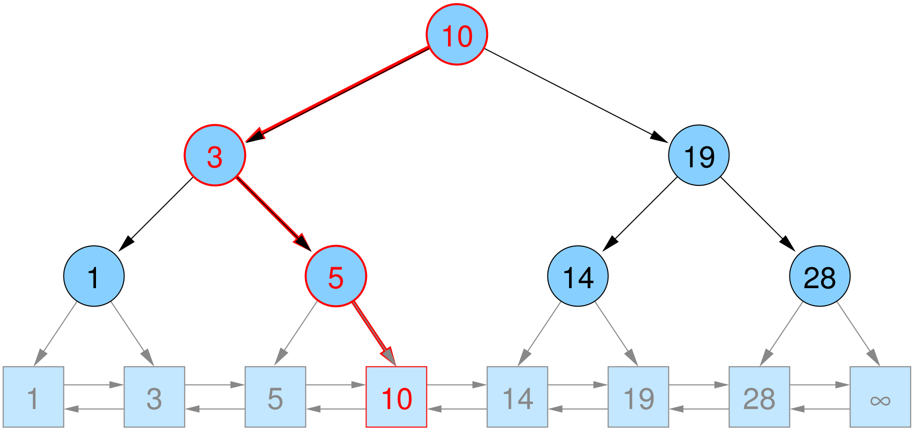
Formal: für einen Baumknoten v sei

- 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)



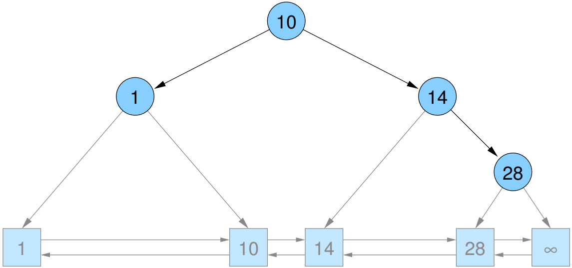
Binärer Suchbaum / insert, remove

Strategie:

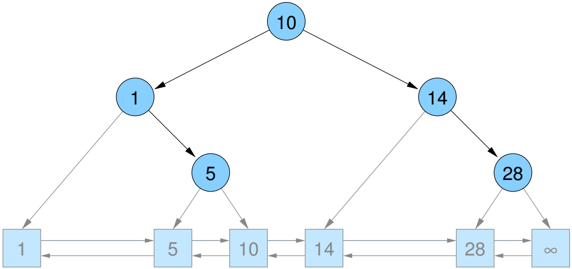
- insert(e):
  - erst wie locate(key(e)) bis Element e' in Liste erreicht
  - falls key(e') $>$ key(e):
    füge e vor e' ein, sowie ein neues Suchbaumblatt für e und e' mit key(e) als Splitter Key, so dass Suchbaum-Regel erfüllt

- 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)

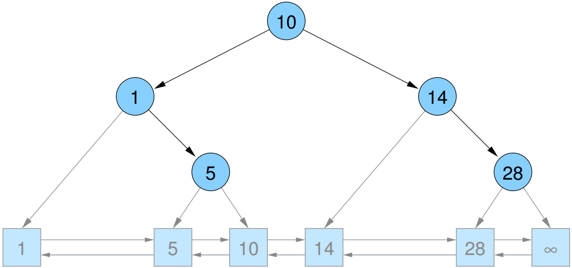
Binärer Suchbaum / insert, remove



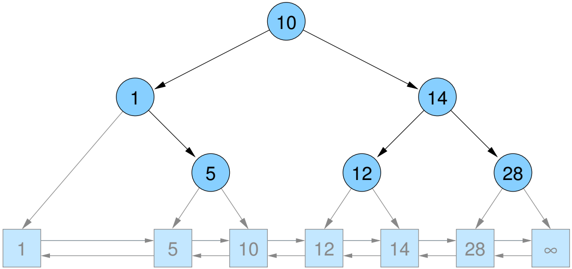
insert(5)



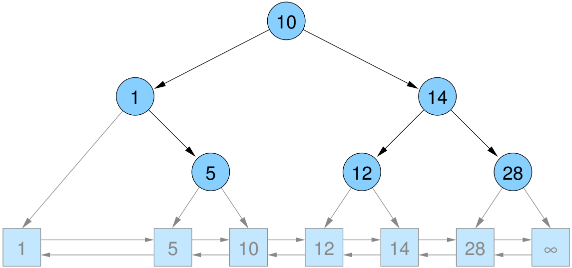
Binärer Suchbaum / insert, remove



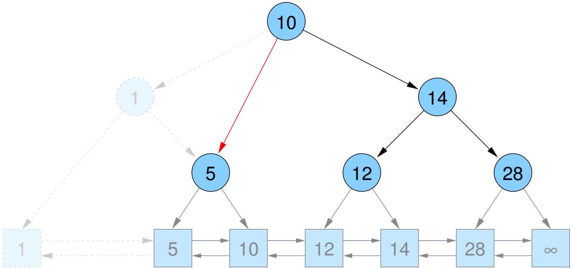
insert(12)



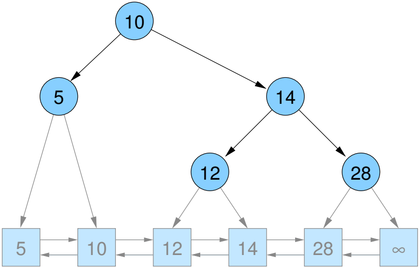
Binärer Suchbaum / insert, remove



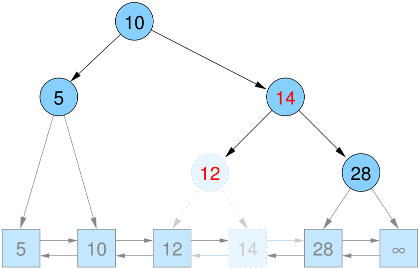
remove(1)



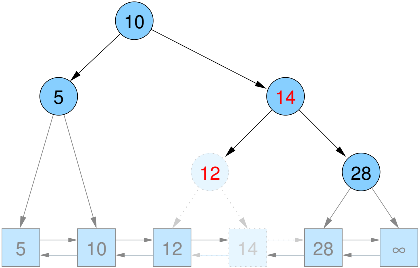
Binärer Suchbaum / insert, remove



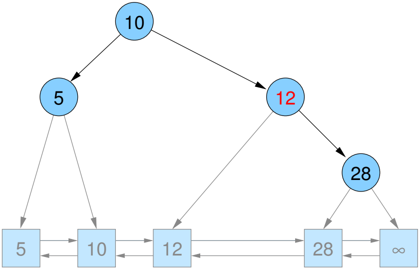
remove(14)



Binärer Suchbaum / insert, remove



remove(14)



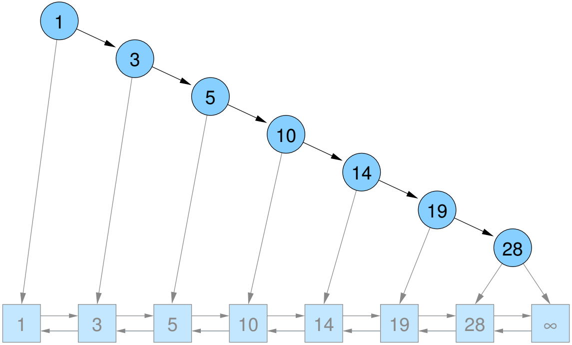
Binärer Suchbaum / worst case

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

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

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

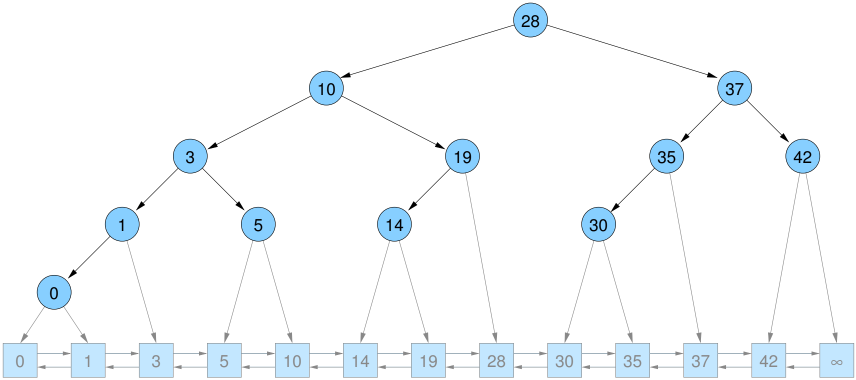
- Was ist die größte Höhe bei gegebener Anzahl von Elementen?
- bzw: Wieviel Elemente hat ein Baum mit Höhe h mindestens?

- 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

⇒ 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 $\mathcal{O}(\log n)$

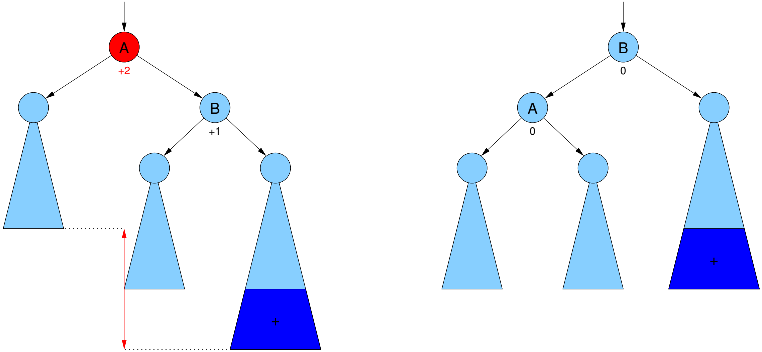
AVL-Bäume: insert

- 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

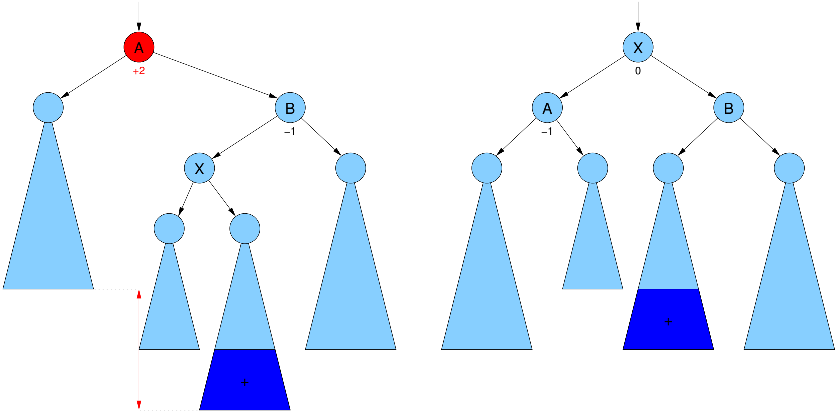
- Differenz 0: Wert war vorher ± 1, Höhe unverändert, also aufhören
- Differenz ± 1: Wert war vorher 0, Höhe ist jetzt um 1 größer, Höhendifferenz im Vaterknoten anpassen und dort weitermachen
- Differenz ± 2: Rebalancierung erforderlich, Einfach- oder Doppelrotation abhängig von Höhendifferenz an den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

  danach Höhe wie zuvor, also aufhören

AVL-Bäume: Einfachrotation nach insert



AVL-Bäume: Doppelrotation nach insert



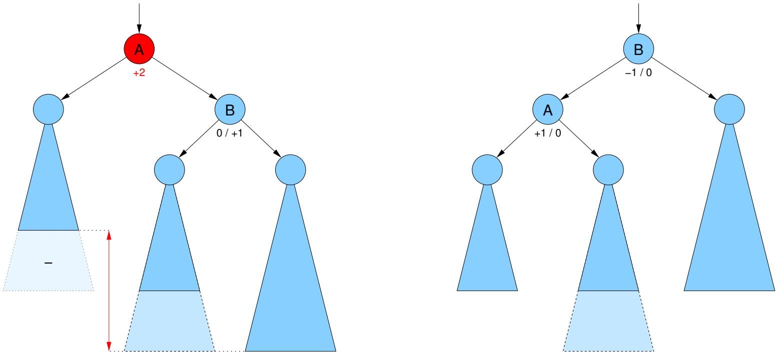
AVL-Bäume: remove

- 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 2 Kinder hat, vertausche v mit dem rechtesten Knoten im linken Unterbaum (nächstkleineres Element direkt vor v) und lösche v dort.
  v hat dort höchstens 1 (linkes) Kind, nun wie im ersten Fall

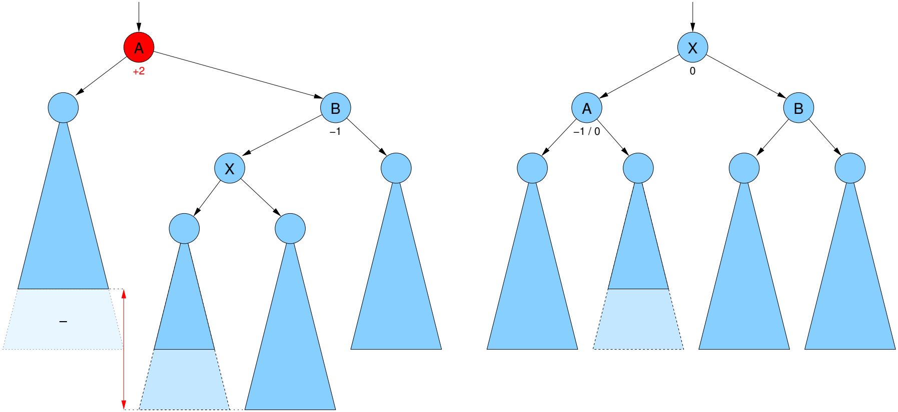
- Differenz 0: Wert war vorher ± 1, Höhe ist jetzt um 1 kleiner, Höhendifferenz im Vaterknoten anpassen und dort weitermachen
- Differenz ± 1: Wert war vorher 0, Höhe unverändert, also aufhören
- Differenz ± 2: Rebalancierung erforderlich, Einfach- oder Doppelrotation abhängig von Höhendifferenz an den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

  falls notwendig Höhendifferenz im Vaterknoten anpassen und dort weitermachen

AVL-Bäume: Einfachrotation nach remove



AVL-Bäume: Doppelrotation nach remove



Übersicht

8 Suchstrukturen

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

(a, b)-Baum

Andere Lösung für das Problem bei binären Suchbäumen, dass die Baumstruktur zur Liste entarten kann

Idee:

- $d(v)$: Ausgangsgrad (Anzahl Kinder) von Knoten v
- $t(v)$: Tiefe (in Kanten) von Knoten v
- Form-Invariante:
  alle Blätter in derselben Tiefe: $t(v) = t(w)$ für Blätter v, w
- Grad-Invariante:
  Für alle internen Knoten v (außer Wurzel) gilt:

$$a \leq d(v) \leq b \qquad \text{(wobei } a \geq 2 \text{ und } b \geq 2a - 1\text{)}$$

Für Wurzel r: $2 \leq d(r) \leq b$ (außer wenn nur 1 Blatt im Baum)

(a, b)-Baum

Lemma

Ein (a, b)-Baum für $n \geq 1$ Elemente hat Tiefe $\leq 1 + \left\lfloor \log_a \frac{n+1}{2} \right\rfloor$.

Beweis.

- Baum hat $n + 1$ Blätter $\quad$ ($+1$ wegen ∞-Dummy)
- Im Fall $n \geq 1$ hat die Wurzel Grad ≥ 2,
  die anderen inneren Knoten haben Grad $\geq a$.

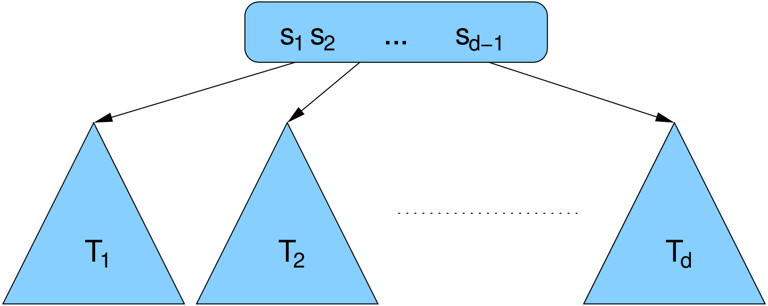
$\Rightarrow$ Bei Tiefe t gibt es $\geq 2a^{t-1}$ Blätter

- $n + 1 \geq 2a^{t-1} \quad \Leftrightarrow \quad t \leq 1 + \log_a \frac{n+1}{2}$
- Da t eine ganze Zahl ist, gilt $\quad t \leq 1 + \left\lfloor \log_a \frac{n+1}{2} \right\rfloor$.

□

(a, b)-Baum: Split-Schlüssel

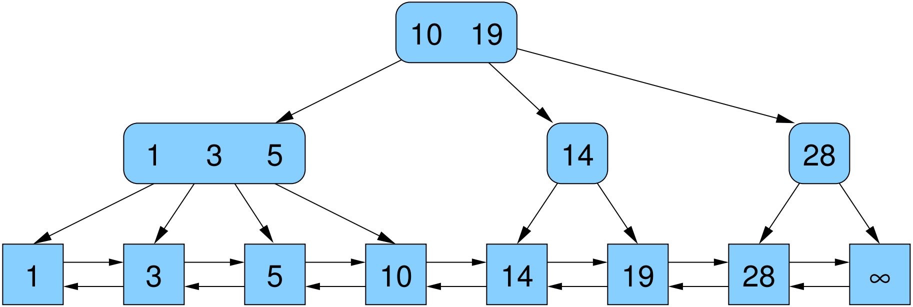
- Jeder Knoten v enthält ein sortiertes Array von $d(v) - 1$ Split-Schlüsseln $s_1, \ldots, s_{d(v)-1}$



- (a, b)-Suchbaum-Regel:
  Für alle Schlüssel k in T_i und k' in T_{i+1} gilt:
  $k \leq s_i < k'$ bzw. $s_{i-1} < k \leq s_i$ $(s_0 = -\infty,\ s_d = \infty)$

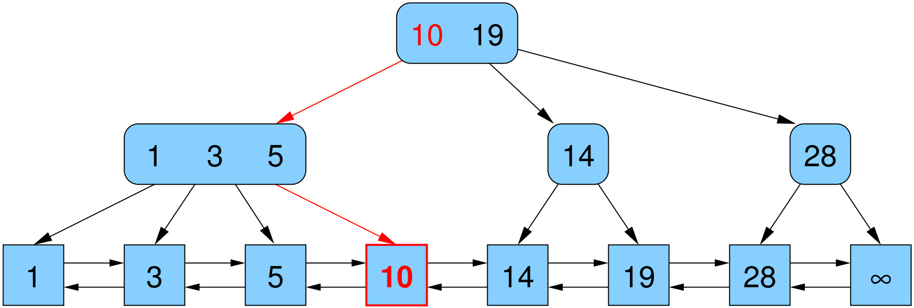
(a, b)-Baum

Beispiel: $(2, 4)$-Baum



(a, b)-Baum

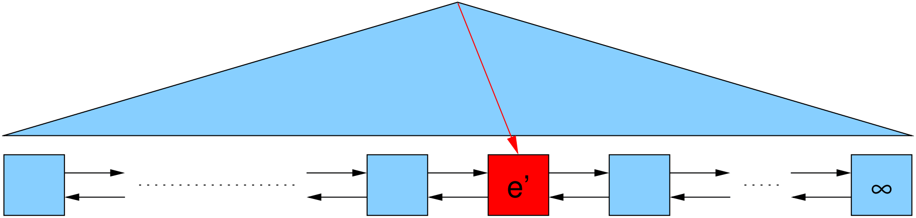
locate(9)



(a, b)-Baum

insert(e)

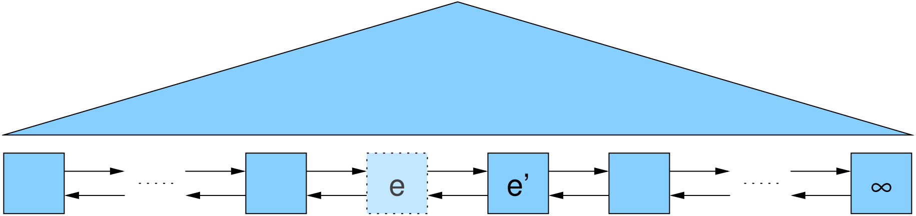
- Abstieg wie bei locate(key(e)) bis Element e' in Liste erreicht
- falls key(e)<key(e'), füge e vor e' ein



(a, b)-Baum

insert(e)

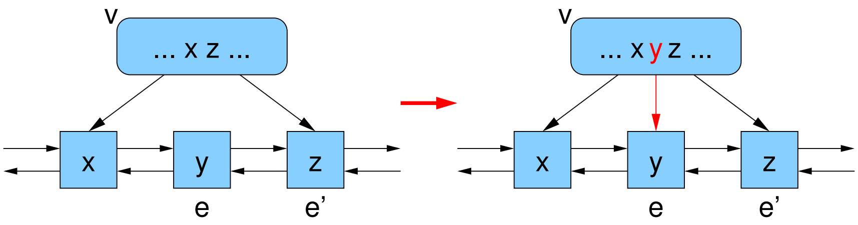
- Abstieg wie bei locate(key(e)) bis Element e' in Liste erreicht
- falls key(e)<key(e'), füge e vor e' ein



(a, b)-Baum

insert(e)

- füge key(e) und Handle auf e in Baumknoten v über e ein
- falls $d(v) \leq b$, dann fertig

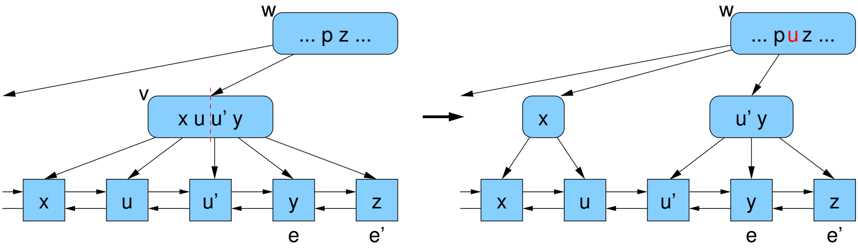


(a, b)-Baum

insert(e)

- füge key(e) und Handle auf e in Baumknoten v über e ein
- falls $d(v) > b$, dann teile v in zwei Knoten auf und
- verschiebe den Splitter (größter Key im linken Teil) in den Vaterknoten

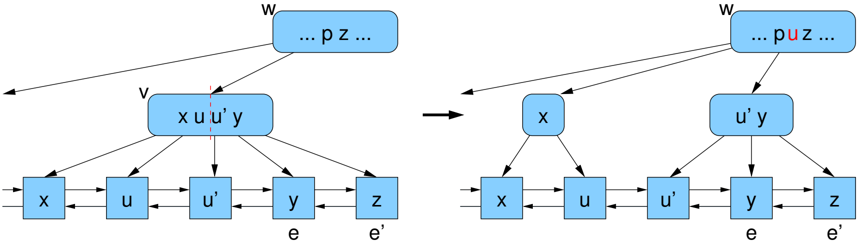
Beispiel: $(2, 4)$-Baum



(a, b)-Baum

insert(e)

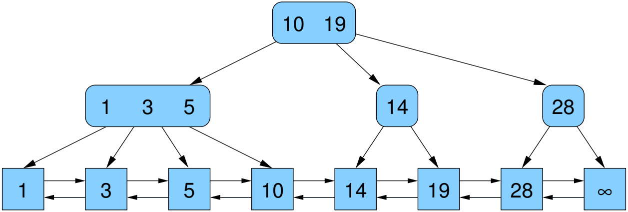
- falls $d(w) > b$, dann teile w in zwei Knoten auf usw.
  bis Grad $\leq b$
  oder Wurzel aufgeteilt wurde



(a, b)-Baum / insert

$a = 2$, $b = 4$

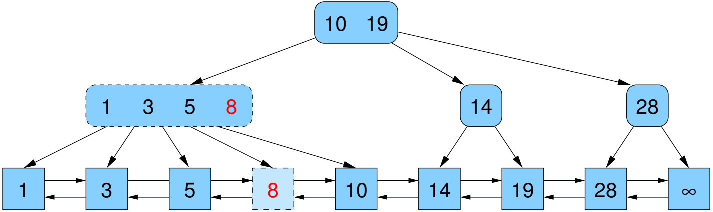
insert(8)



(a, b)-Baum / insert

$a = 2$, $b = 4$

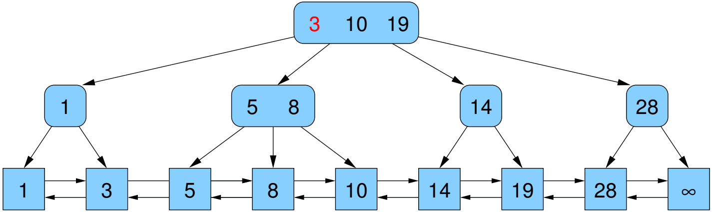
insert(8)



(a, b)-Baum / insert

$a = 2$, $b = 4$

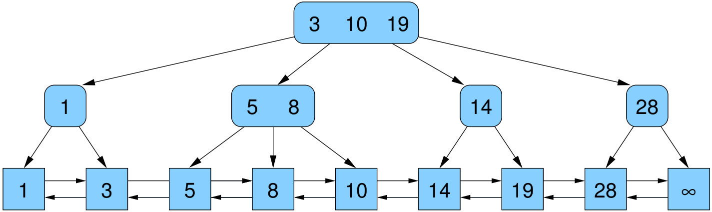
insert(8)



(a, b)-Baum / insert

$a = 2$, $b = 4$

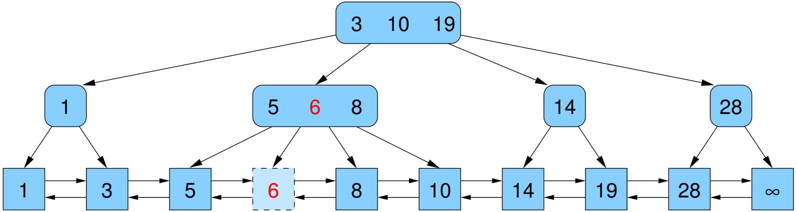
insert(6)



(a, b)-Baum / insert

$a = 2$, $b = 4$

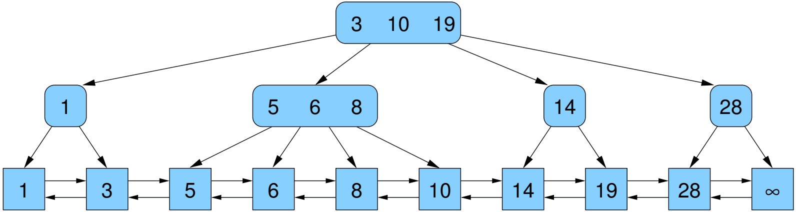
insert(6)



(a, b)-Baum / insert

$a = 2$, $b = 4$

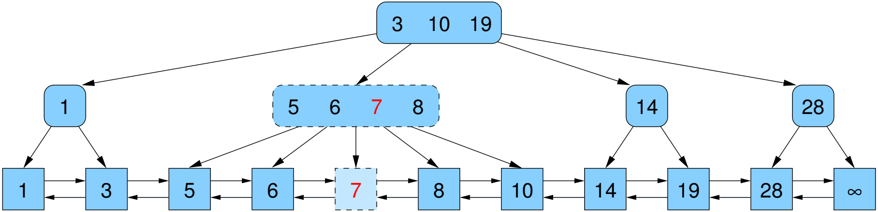
insert(7)



(a, b)-Baum / insert

$a = 2$, $b = 4$

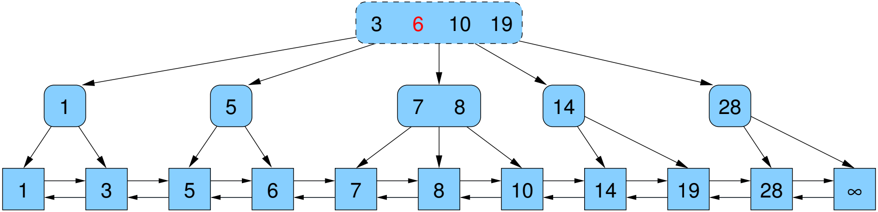
insert(7)



(a, b)-Baum / insert

$a = 2$, $b = 4$

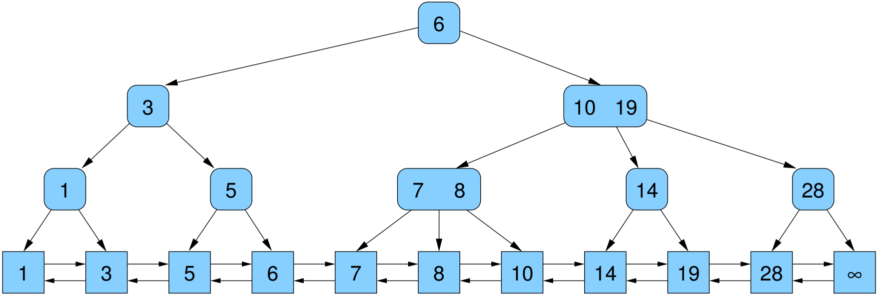
insert(7)



(a, b)-Baum / insert

$a = 2$, $b = 4$

insert(7)



(a, b)-Baum / insert

Form-Invariante

- alle Blätter haben dieselbe Tiefe, denn
  neues Blatt wird auf der Ebene der anderen eingefügt und
  im Fall einer neuen Wurzel erhöht sich die Tiefe aller Blätter um 1

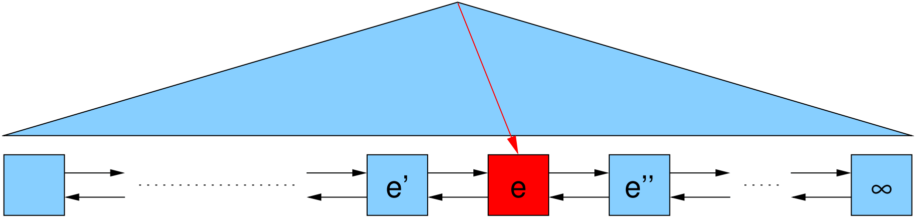
Grad-Invariante

- insert splittet Knoten mit Grad $b + 1$ in zwei Knoten
  mit Grad $\lfloor (b + 1)/2 \rfloor$ und $\lceil (b + 1)/2 \rceil$
- wenn $b \geq 2a - 1$, dann sind beide Werte $\geq a$
- wenn Wurzel Grad $b + 1$ erreicht und gespalten wird,
  wird neue Wurzel mit Grad 2 erzeugt

(a, b)-Baum

remove(k)

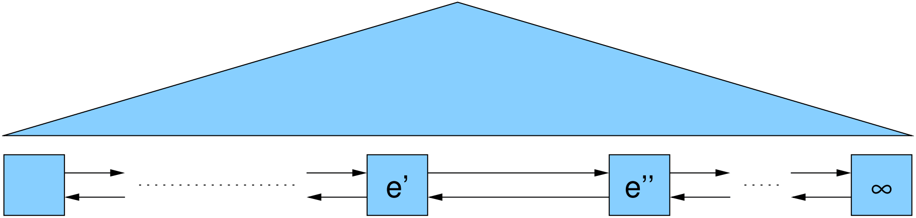
- Abstieg wie bei locate(k) bis Element e in Liste erreicht
- falls key(e) $= k$, entferne e aus Liste (sonst return)



(a, b)-Baum

remove(k)

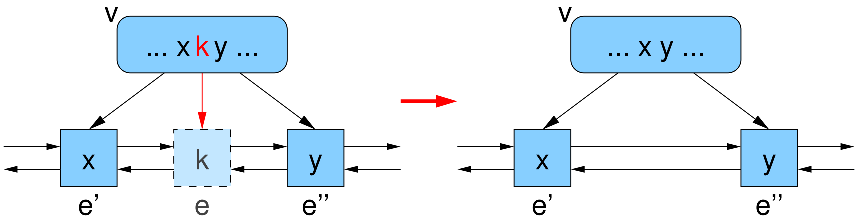
- Abstieg wie bei locate(k) bis Element e in Liste erreicht
- falls key(e) $= k$, entferne e aus Liste (sonst return)



(a, b)-Baum

remove(k)

- entferne Handle auf e und Schlüssel k vom Baumknoten v über e (wenn e rechtestes Kind: Schlüsselvertauschung wie bei binärem Suchbaum)
- falls $d(v) \geq a$, dann fertig

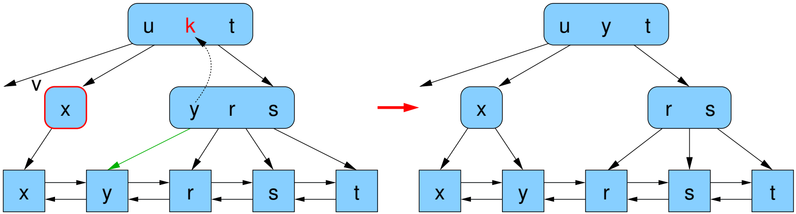


(a, b)-Baum

remove(k)

- falls $d(v) < a$ und ein direkter Nachbar v' von v hat Grad $> a$, nimm Kante von v'

Beispiel: $(2, 4)$-Baum

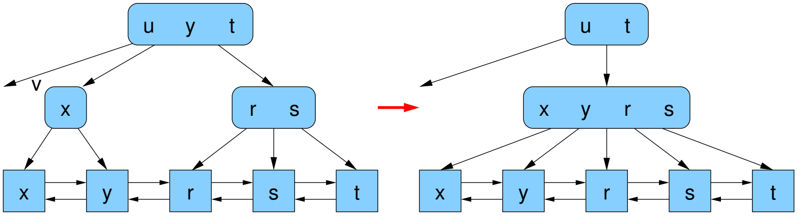


(a, b)-Baum

remove(k)

- falls $d(v) < a$ und kein direkter Nachbar von v hat Grad $> a$, merge v mit Nachbarn

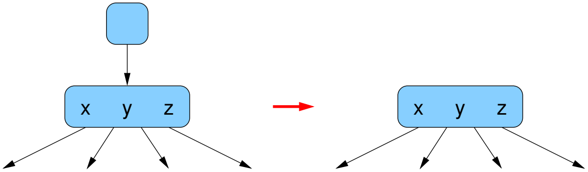
Beispiel: $(3, 5)$-Baum



(a, b)-Baum

remove(k)

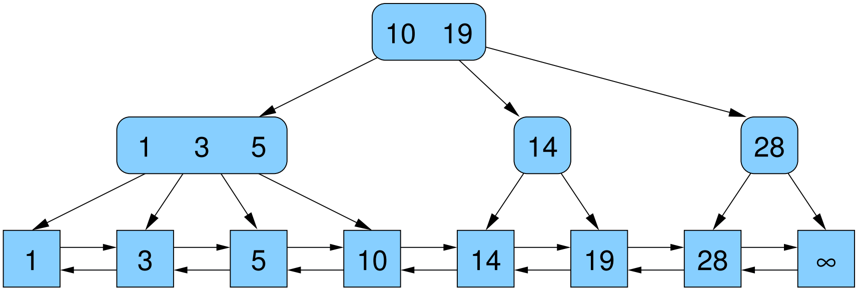
- Verschmelzungen können sich nach oben fortsetzen, ggf. bis zur Wurzel
- falls Grad der Wurzel < 2: entferne Wurzel
  neue Wurzel wird das einzige Kind der alten Wurzel



(a, b)-Baum / remove

$a = 2$, $b = 4$

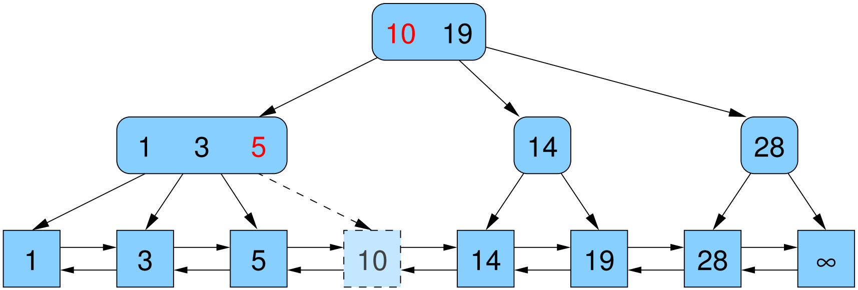
remove(10)



(a, b)-Baum / remove

$a = 2$, $b = 4$

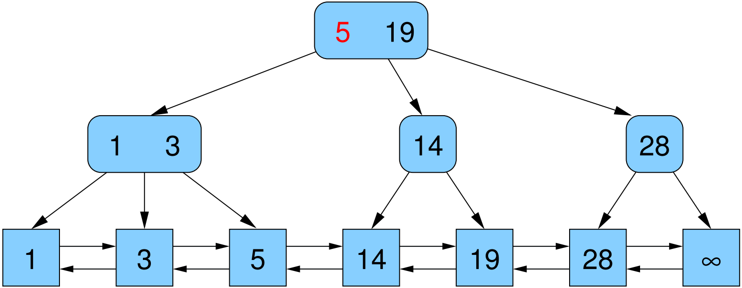
remove(10)



(a, b)-Baum / remove

$a = 2$, $b = 4$

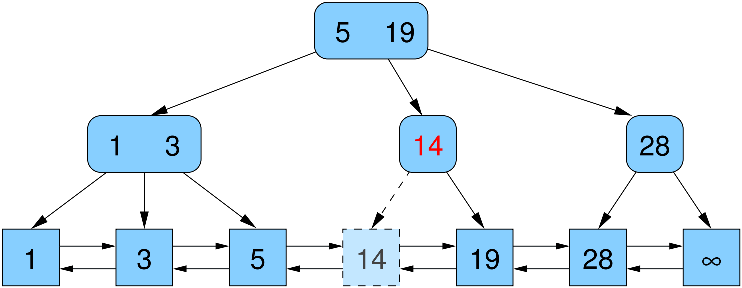
remove(10)



(a, b)-Baum / remove

$a = 2$, $b = 4$

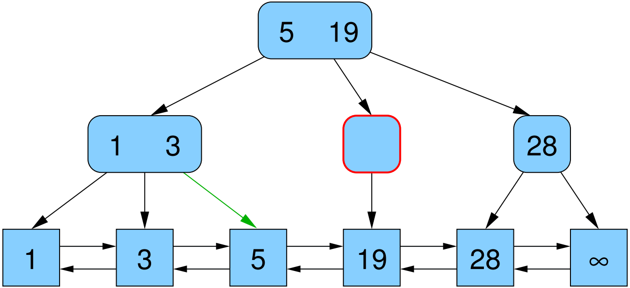
remove(14)



(a, b)-Baum / remove

$a = 2$, $b = 4$

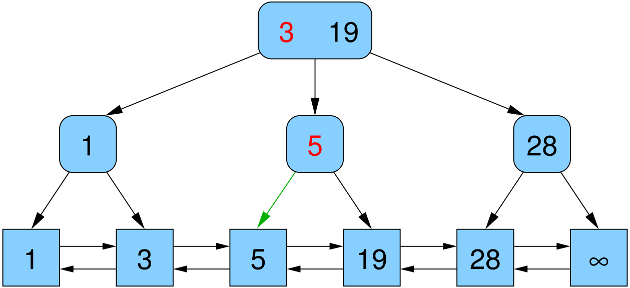
remove(14)



(a, b)-Baum / remove

$a = 2$, $b = 4$

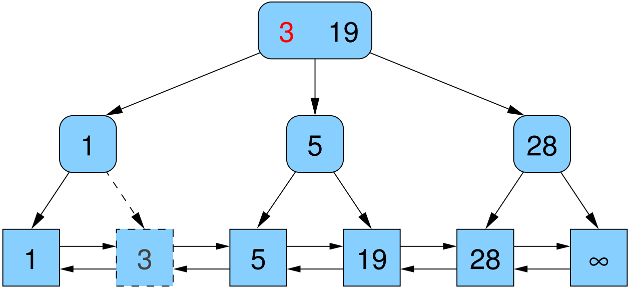
remove(14)



(a, b)-Baum / remove

$a = 2$, $b = 4$

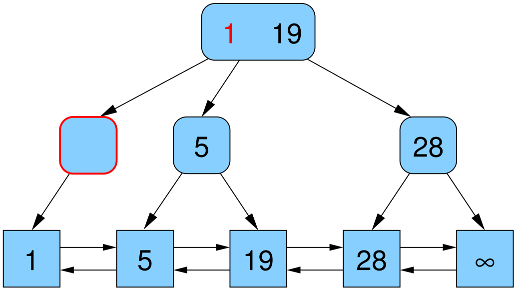
remove(3)



(a, b)-Baum / remove

$a = 2$, $b = 4$

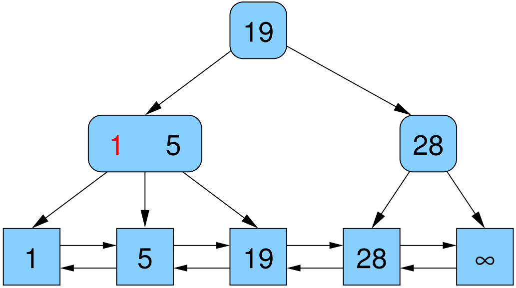
remove(3)



(a, b)-Baum / remove

$a = 2$, $b = 4$

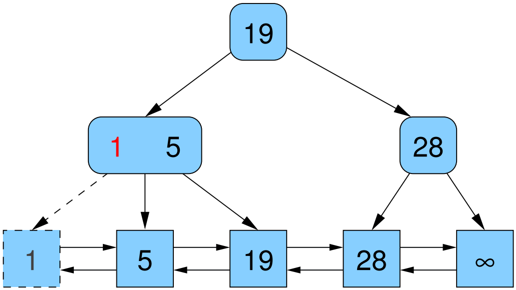
remove(3)



(a, b)-Baum / remove

$a = 2$, $b = 4$

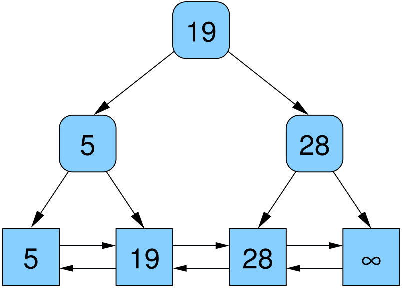
remove(1)



(a, b)-Baum / remove

$a = 2$, $b = 4$

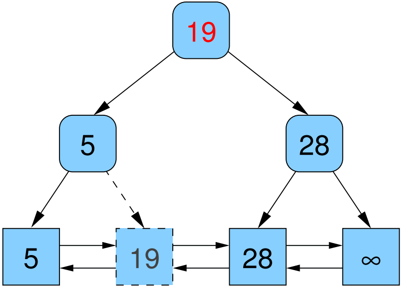
remove(1)



(a, b)-Baum / remove

$a = 2$, $b = 4$

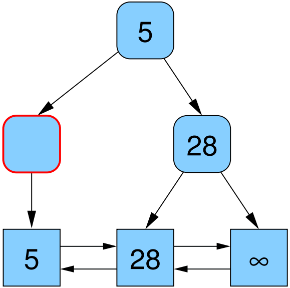
remove(19)



(a, b)-Baum / remove

$a = 2$, $b = 4$

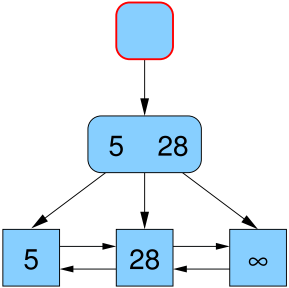
remove(19)



(a, b)-Baum / remove

$a = 2$, $b = 4$

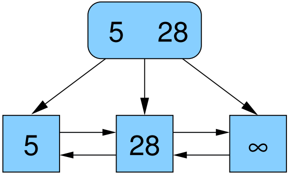
remove(19)



(a, b)-Baum / remove

$a = 2$, $b = 4$

remove(19)



(a, b)-Baum / remove

Form-Invariante

- alle Blätter behalten dieselbe Tiefe
- falls alte Wurzel entfernt wird, verringert sich die Tiefe aller Blätter

Grad-Invariante

- remove verschmilzt Knoten, die Grad $a - 1$ und a haben
- wenn $b \geq 2a - 1$, dann ist der resultierende Grad $\leq b$
- remove verschiebt eine Kante von Knoten mit Grad $> a$ zu Knoten mit Grad $a - 1$, danach sind beide Grade in $[a, b]$
- wenn Wurzel gelöscht, wurden vorher die Kinder verschmolzen, Grad vom letzten Kind ist also $\geq a$ (und $\leq b$)

Weitere Operationen im (a, b)-Baum

- min / max-Operation

  verwende first / last-Methode der Liste, um das kleinste bzw. größte Element auszugeben

  Zeit: $\mathcal{O}(1)$

- Range queries (Bereichsanfragen)

  suche alle Elemente im Bereich $[x, y]$:

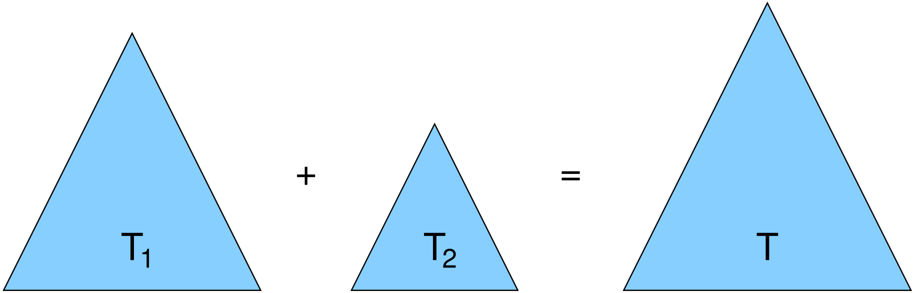
  - führe locate(x) aus und
  - durchlaufe die Liste, bis Element $> y$ gefunden wird

  Zeit: $\mathcal{O}(\log n + \text{Ausgabegröße})$

- Konkatenation / Splitting

Konkatenation von (a, b)-Bäumen

- verknüpfe zwei (a, b)-Bäume T_1 und T_2 mit s_1 bzw. s_2 Elementen und Höhe h_1 bzw. h_2 zu (a, b)-Baum T
- Bedingung: Schlüssel in $T_1 \leq$ Schlüssel in T_2



Konkatenation von (a, b)-Bäumen

- lösche in T_1 das ∞-Dummy-Element
- wenn danach dessen Vater-Knoten $< a$ Kinder hat, dann behandle dies wie bei remove

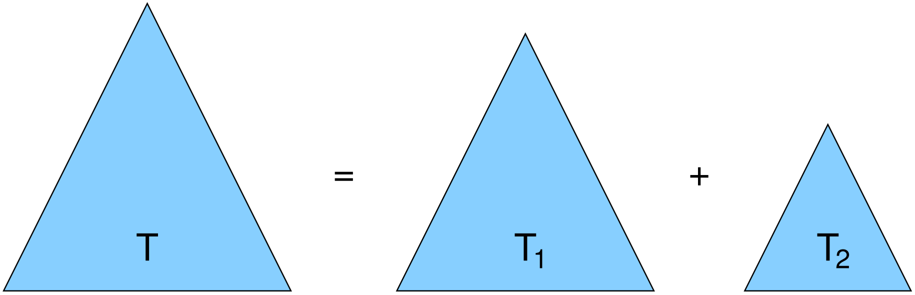
- verschmelze die Wurzel des niedrigeren Baums mit dem entsprechenden äußersten Knoten des anderen Baums, der sich auf dem gleichen Level befindet
- wenn dieser Knoten danach $> b$ Kinder hat, dann behandle dies wie bei insert

$\Rightarrow$ falls Höhe der Bäume explizit gespeichert: Zeit $\mathcal{O}(1 + |h_1 - h_2|)$

ansonsten (mit Höhenbestimmung): Zeit $\mathcal{O}(1 + \max\{h_1, h_2\}) \subseteq \mathcal{O}(1 + \log(\max\{s_1, s_2\}))$

Aufspaltung eines (a, b)-Baums

- spalte (a, b)-Baum T bei Schlüssel k in zwei (a, b)-Bäume T_1 und T_2 auf



Aufspaltung eines (a, b)-Baums

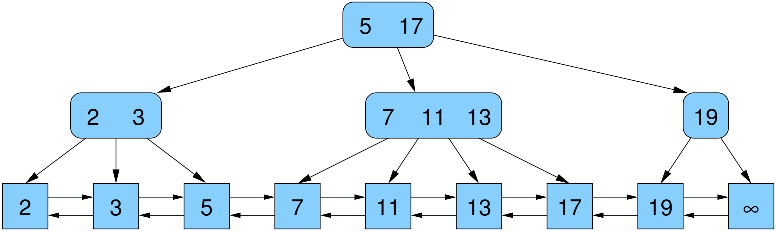
- Sequenz $q = \langle w, \ldots, x, y, \ldots, z \rangle$ soll bei Schlüssel y in Teile $q_1 = \langle w, \ldots, x \rangle$ und $q_2 = \langle y, \ldots, z \rangle$ aufgespalten werden
- betrachte Pfad von Wurzel zum Blatt y
- spalte auf diesem Pfad jeden Knoten v in zwei Knoten v_ℓ und v_r
- v_ℓ bekommt Kinder links vom Pfad,
  v_r bekommt Kinder rechts vom Pfad
  (evt. gibt es Knoten ohne Kinder)
- Knoten mit Kind(ern) werden als Wurzeln von (a, b)-Bäumen interpretiert
- Konkatenation der linken Bäume zusammen mit einem neuen ∞-Dummy ergibt einen Baum für die Elemente bis x
- Konkatenation von $\langle y \rangle$ zusammen mit den rechten Bäumen ergibt einen Baum für die Elemente ab y

Aufspaltung eines (a, b)-Baums

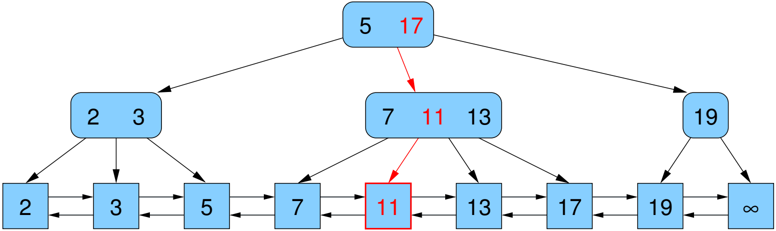
- diese $\mathcal{O}(\log n)$ Konkatenationen können in Gesamtzeit $\mathcal{O}(\log n)$ erledigt werden
- Grund: die linken Bäume haben echt monoton fallende, die rechten echt monoton wachsende Höhe
- Seien z.B. $r_1, r_2, \dots, r_k$ die Wurzeln der linken Bäume und $h_1 > h_2 > \dots > h_k$ deren Höhen
- verbinde zuerst r_{k-1} und r_k in Zeit $\mathcal{O}(1 + h_{k-1} - h_k)$,
  dann r_{k-2} mit dem Ergebnis in Zeit $\mathcal{O}(1 + h_{k-2} - h_{k-1})$,
  dann r_{k-3} mit dem Ergebnis in Zeit $\mathcal{O}(1 + h_{k-3} - h_{k-2})$ usw.
- Gesamtzeit:

$$\mathcal{O}\left(\sum_{1 \le i < k} (1 + h_i - h_{i+1})\right) = \mathcal{O}(k + h_1 - h_k) \in \mathcal{O}(\log n)$$

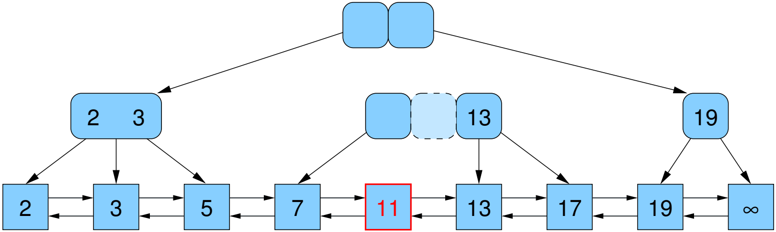
Aufspaltung eines (a, b)-Baums



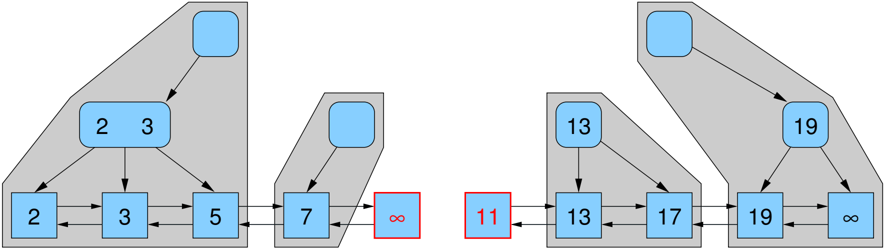
Aufspaltung eines (a, b)-Baums



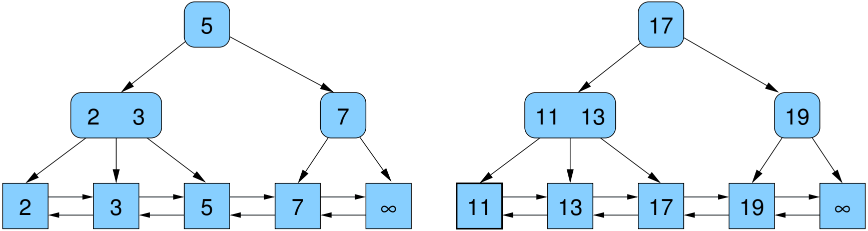
Aufspaltung eines (a, b)-Baums



Aufspaltung eines (a, b)-Baums



Aufspaltung eines (a, b)-Baums



Effizienz von insert / remove-Folgen

Satz

Es gibt eine Folge von n insert- und remove-Operationen auf einem anfangs leeren $(2, 3)$-Baum, so dass die Gesamtanzahl der Knotenaufspaltungen und -verschmelzungen in $\Omega(n \log n)$ ist.

Beweis: *siehe Übung*

Effizienz von insert / remove-Folgen

Satz

Für (a, b)-Bäume, die die erweiterte Bedingung $b \geq 2a$ erfüllen, gilt:

Für jede Folge von n insert- und remove-Operationen auf einem anfangs leeren (a, b)-Baum ist die Gesamtanzahl der Knotenaufspaltungen und -verschmelzungen in $\mathcal{O}(n)$.

Beweis: amortisierte Analyse, nicht in dieser Vorlesung