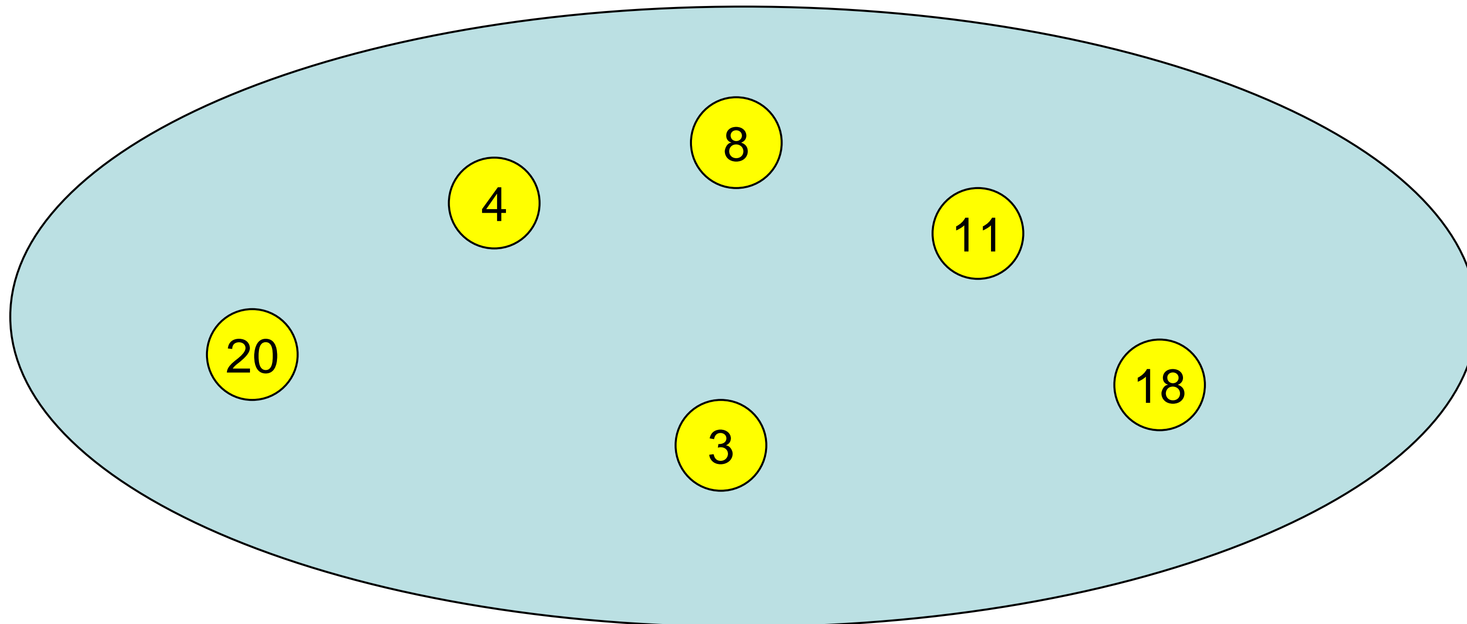
Effiziente Algorithmen und Datenstrukturen I
Kapitel 3: Suchstrukturen

Christian Scheideler

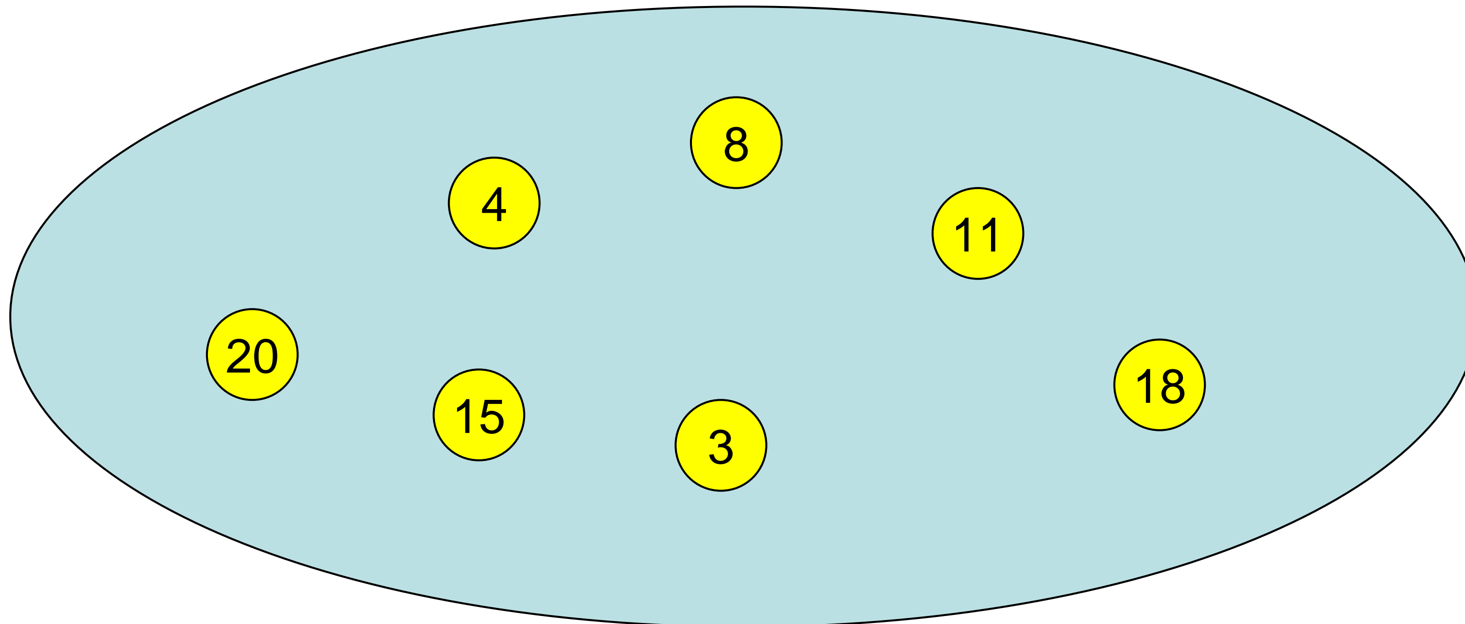
WS 2008

Suchstruktur



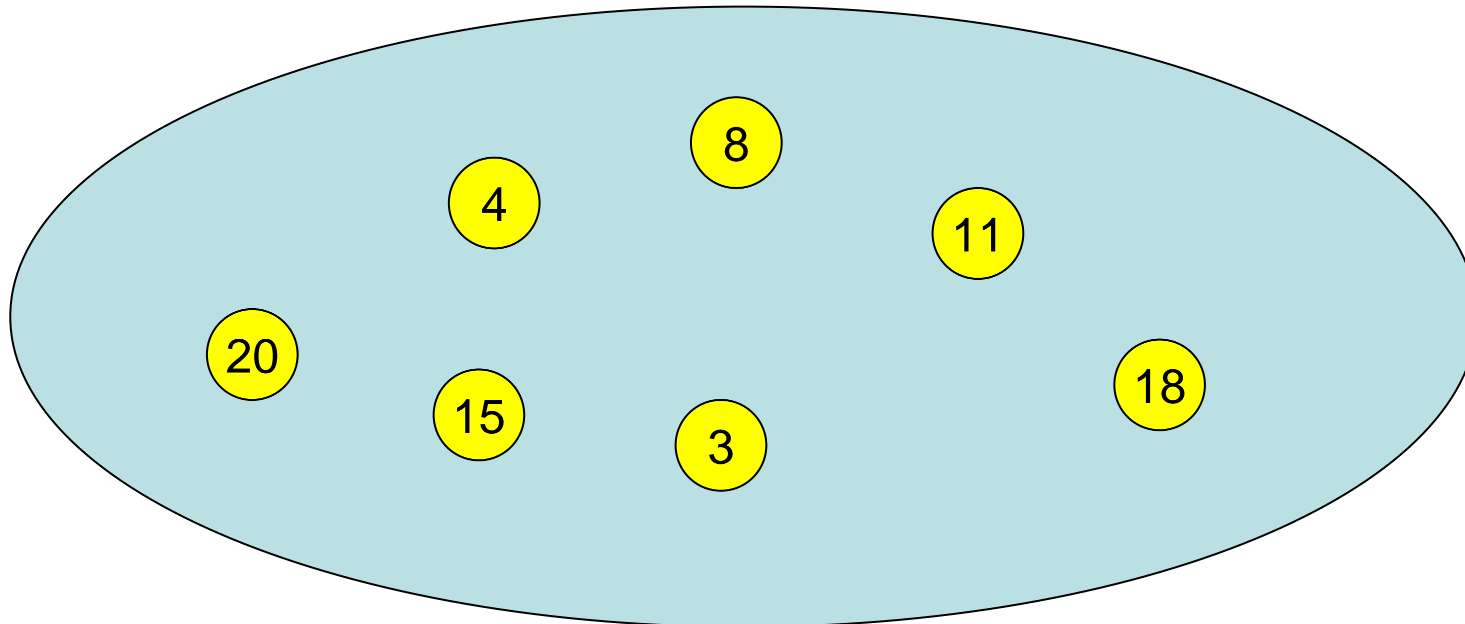
Suchstruktur

insert(15)



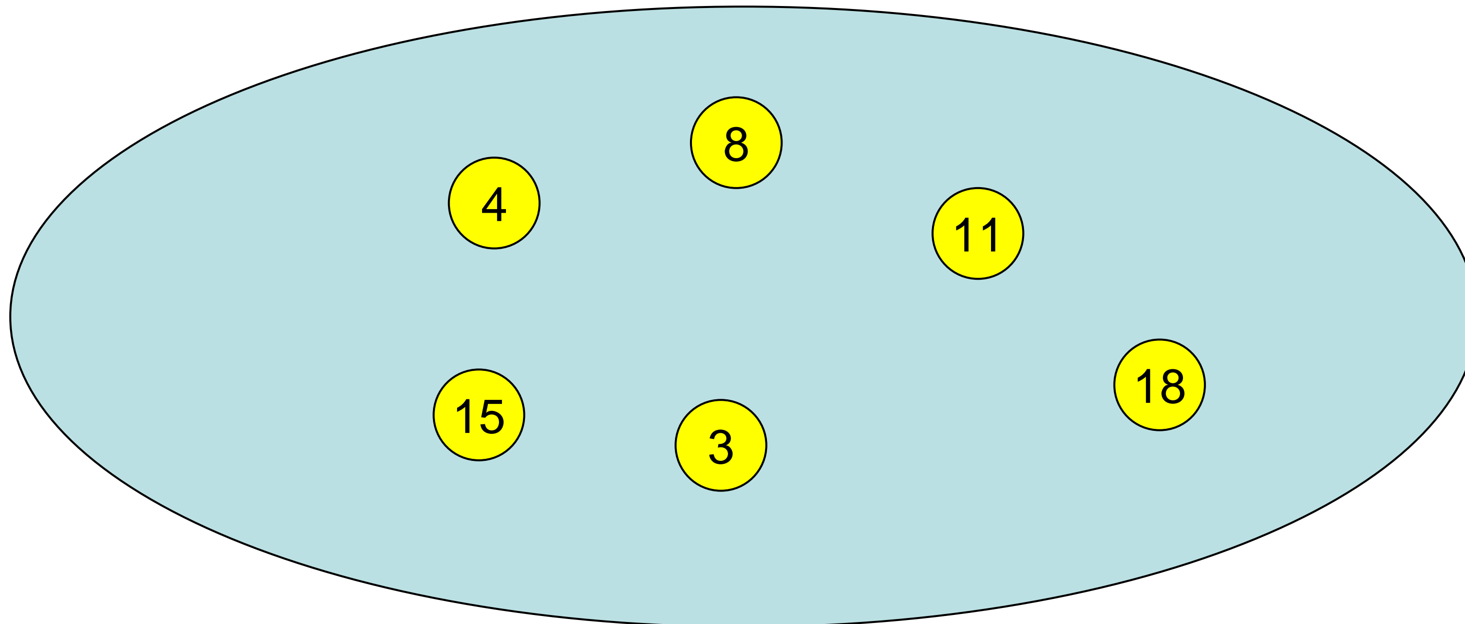
Suchstruktur

delete(20)



Suchstruktur

search(7) ergibt 8 (nächsten Nachfolger)



Suchstruktur

S: Menge von Elementen

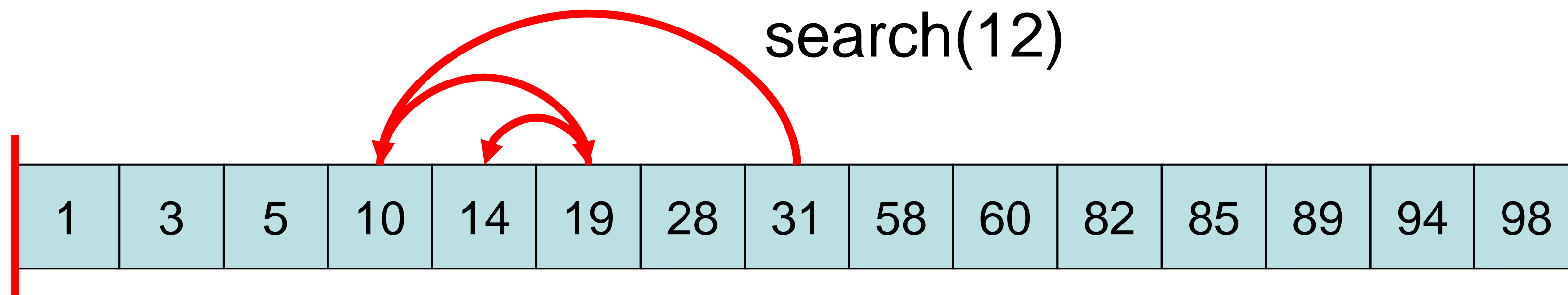
Jedes Element e identifiziert über key(e).

Operationen:

- S.insert(e: Element): $S:=S \cup \{e\}$
- S.delete(k: Key): $S:=S \setminus \{e\}$, wobei e das Element ist mit key(e)=k
- S.search(k: Key): gib $e \in S$ aus mit minimalem key(e) so dass key(e)>=k

Statische Suchstruktur

1. Speichere Elemente in sortiertem Feld.



search: über binäre Suche (O(log n) Zeit)

Binäre Suche

Eingabe: Zahl x und ein sortiertes Feld A[1],…,A[n]

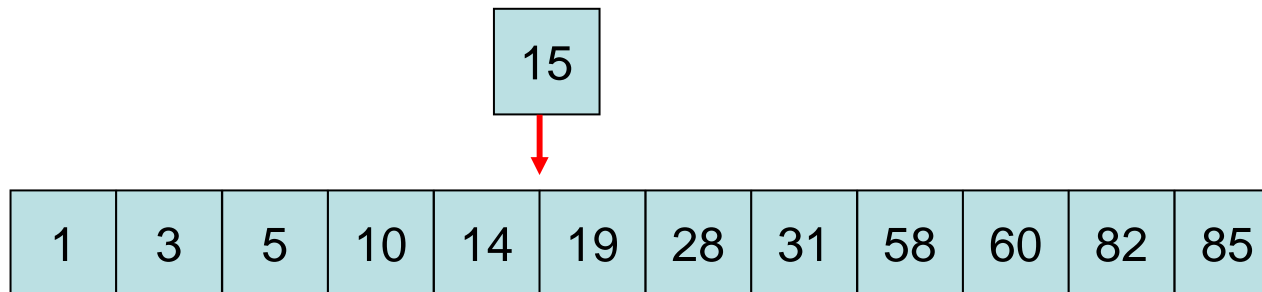
Binäre Suche Algorithmus:

```
l:=1; r:=n
while l < r do
    m:=(r+l) div 2
    if A[m] = x then return m
    if A[m] < x then l:=m+1
                 else r:=m
return l
```

Dynamische Suchstruktur

insert und delete Operationen:

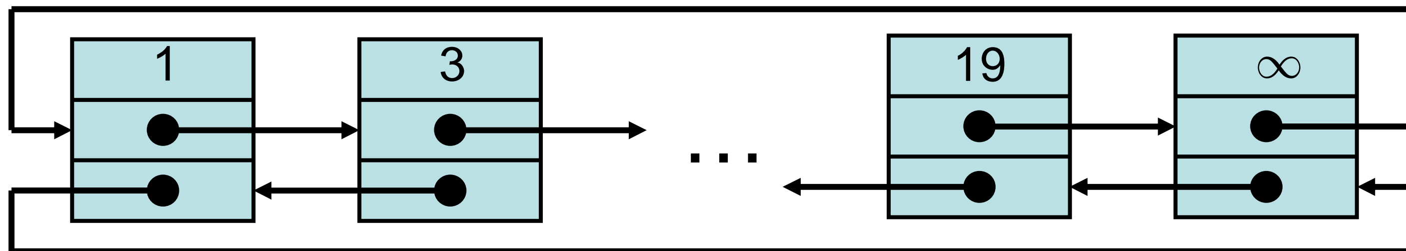
Sortiertes Feld schwierig zu aktualisieren!



Worst case: $\Theta(n)$ Zeit

Suchstruktur

2. Sortierte Liste (mit ∞-Element)

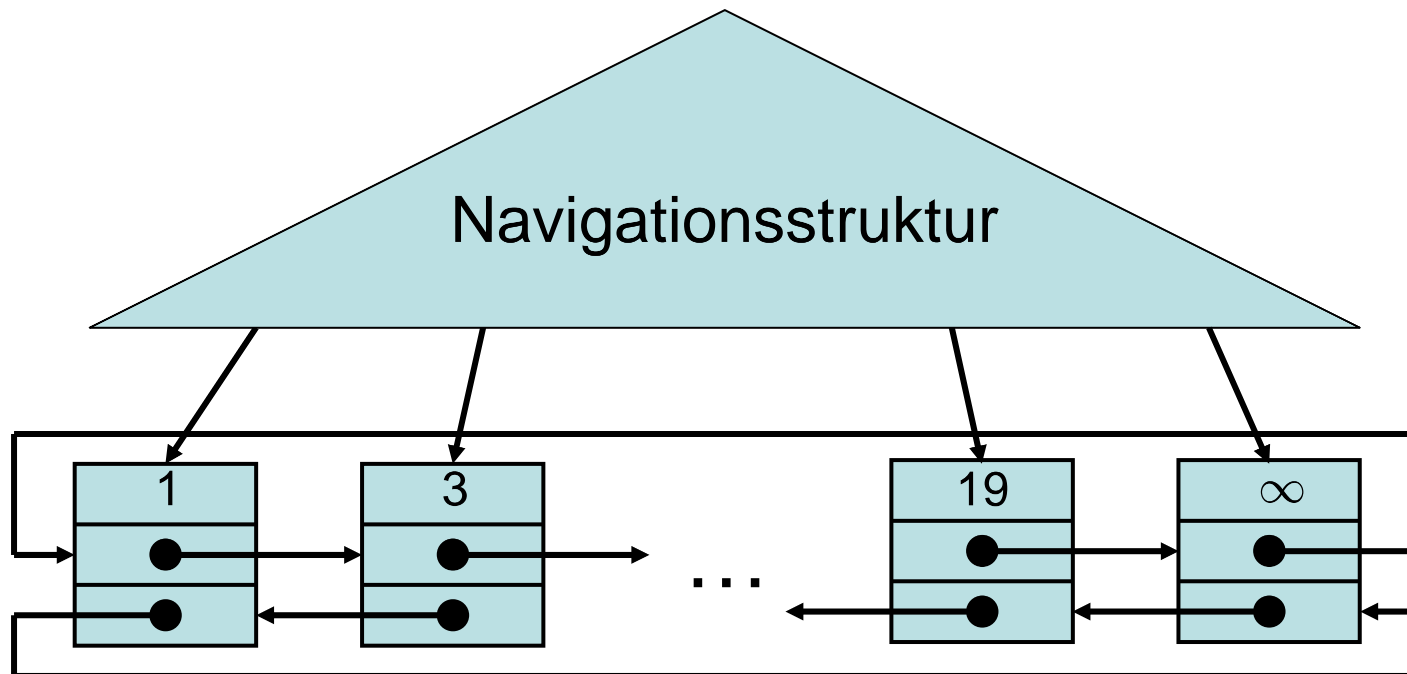


Problem: insert, delete und search kosten im worst case $\Theta(n)$ Zeit

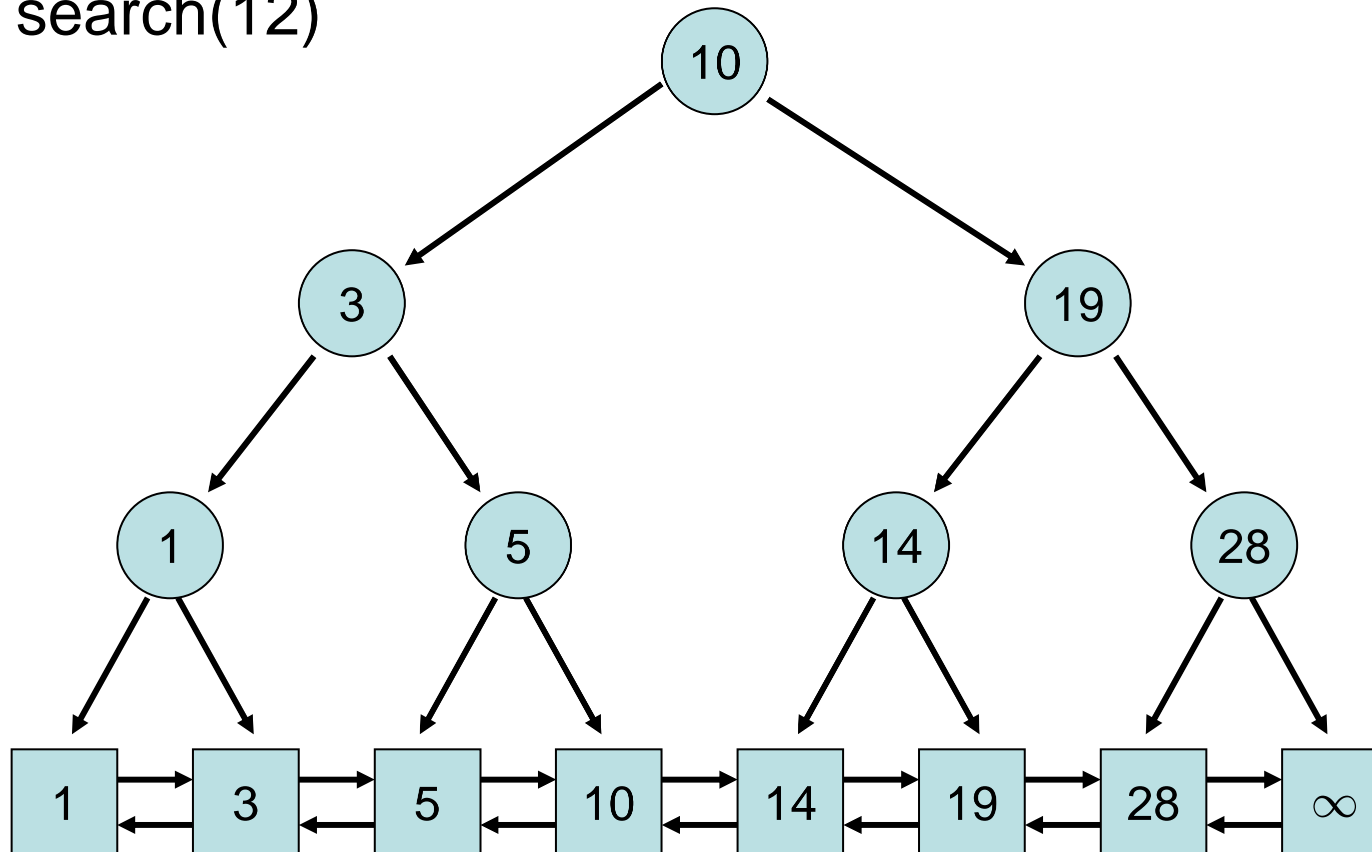
Einsicht: Wenn search effizient zu implementieren wäre, dann auch alle anderen Operationen

Suchstruktur

Idee: füge Navigationsstruktur hinzu, die search effizient macht

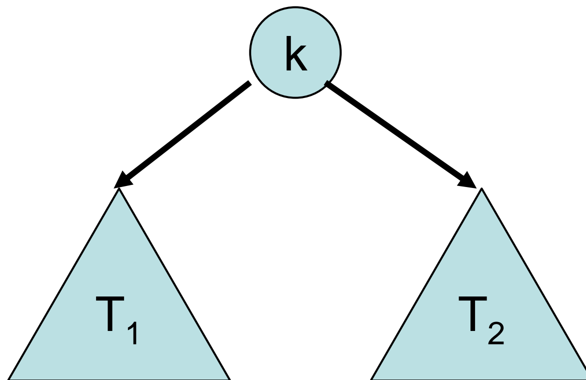


Binärer Suchbaum (ideal)



Binärer Suchbaum

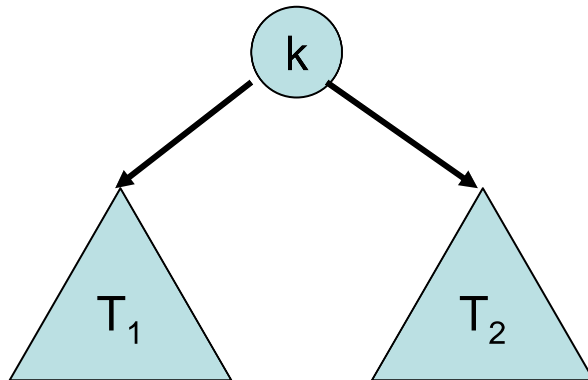
Suchbaum-Regel:



Für alle Schlüssel k' in T_1 und k'' in T_2: $k' \leq k < k''$

Damit search Operation einfach zu imple-mentieren.

search(k) Operation



Für alle Schlüssel k' in T_1 und k'' in T_2: $k' \leq k < k''$

Suchstrategie:

- Starte in Wurzel des Suchbaums
- Für jeden erreichten Knoten v:
  - Falls $key(v) \geq k$, gehe zum linken Kind von v, sonst gehe zum rechten Kind

Binärer Suchbaum

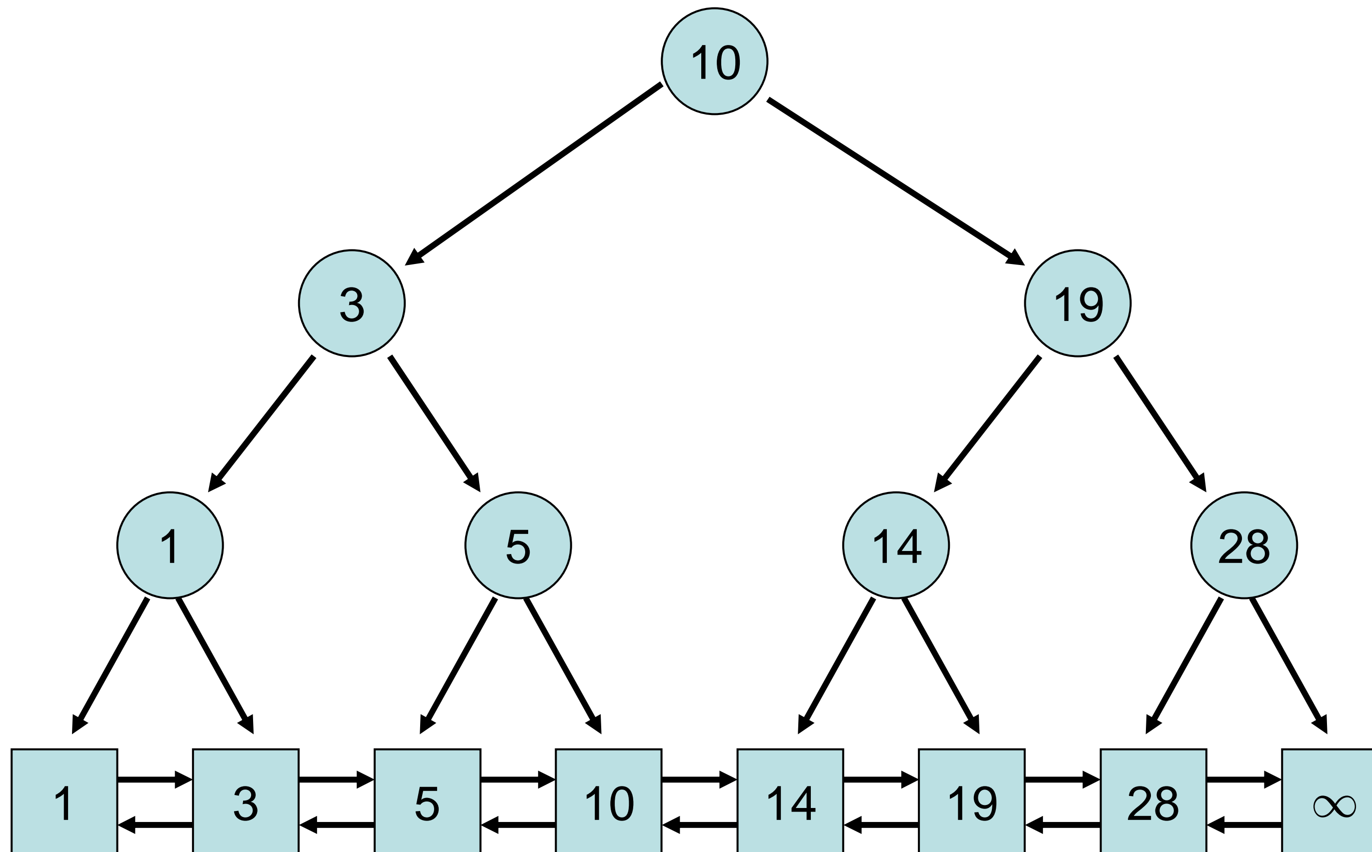
Formell: für einen Baumknoten v sei
- key(v) der Schlüssel in v
- d(v) die Anzahl Kinder von v

- Suchbaum-Regel: (s.o.)

- Grad-Regel:
  Alle Baumknoten haben zwei Kinder
  (sofern #Elemente >1)

- Schlüssel-Regel:
  Für jedes Element e in der Liste gibt es genau einen Baumknoten v mit key(v)=key(e).

Search(9)

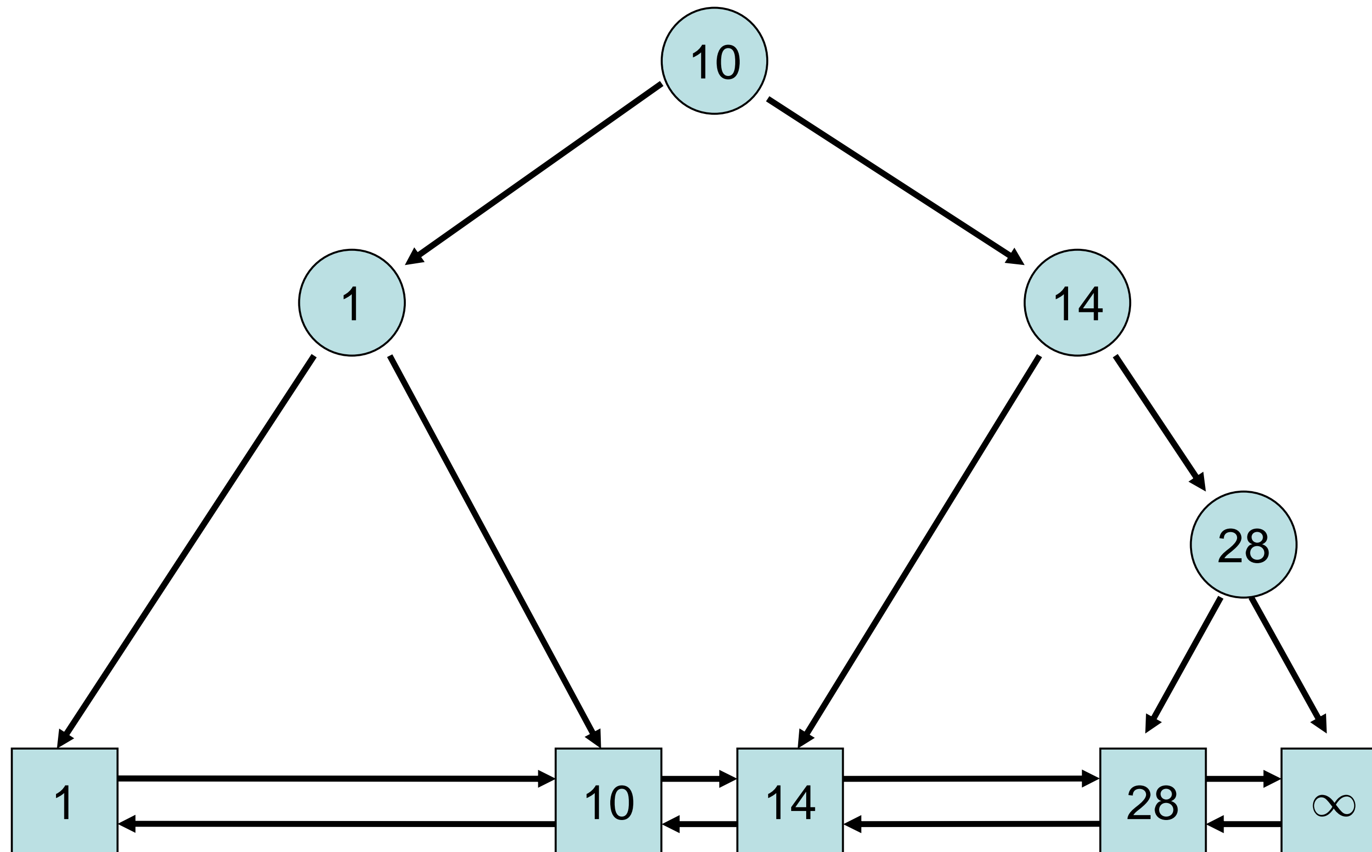


Insert und Delete Operationen

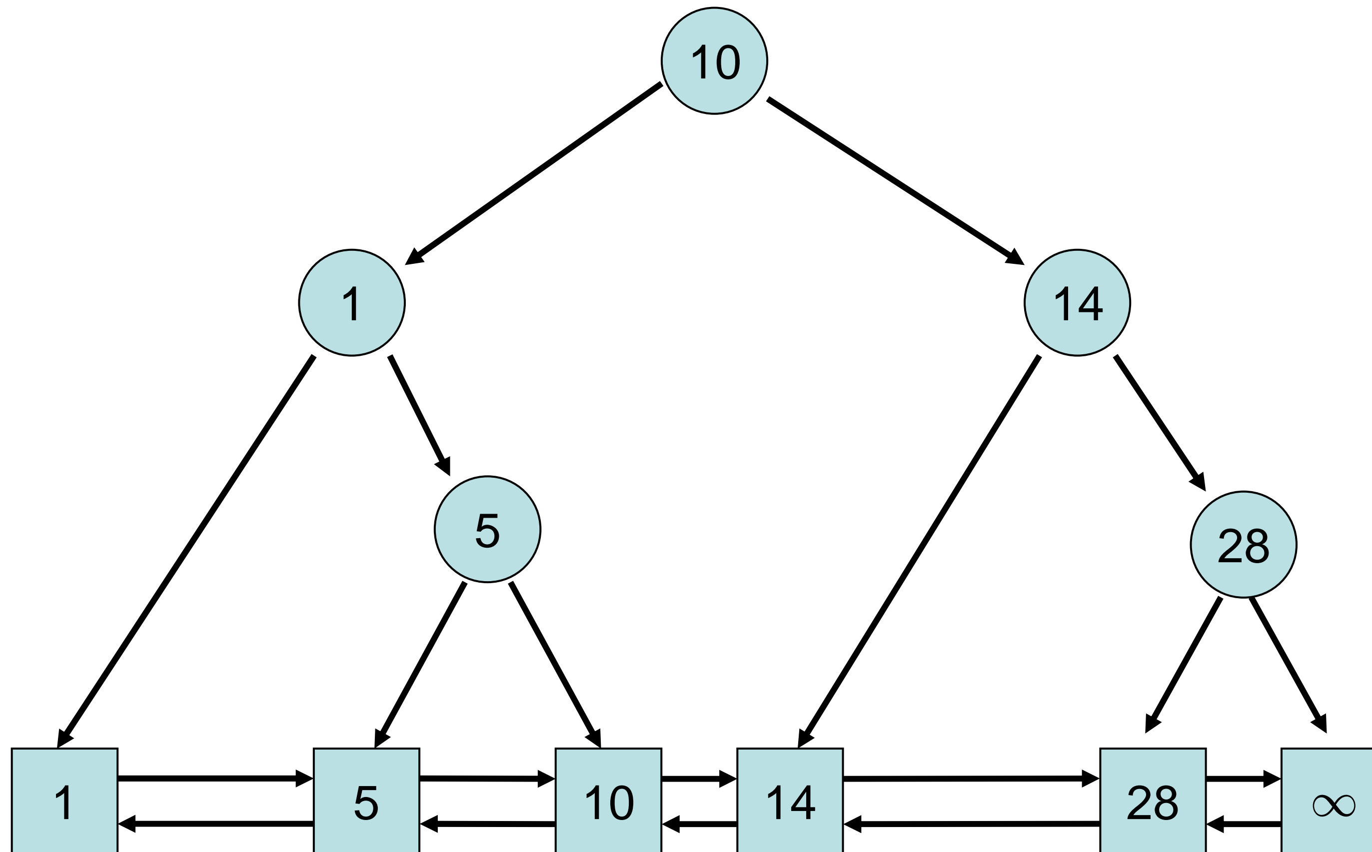
Strategie:

- insert(e):
  Erst search(key(e)) bis Element e´ in Liste erreicht. Falls key(e´)>key(e), füge e vor e´ ein und ein neues Suchbaumblatt für e und e´ mit key(e), so dass Suchbaum-Regel erfüllt.
- delete(k):
  Erst search(k) bis ein 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: key(w):=key(v)

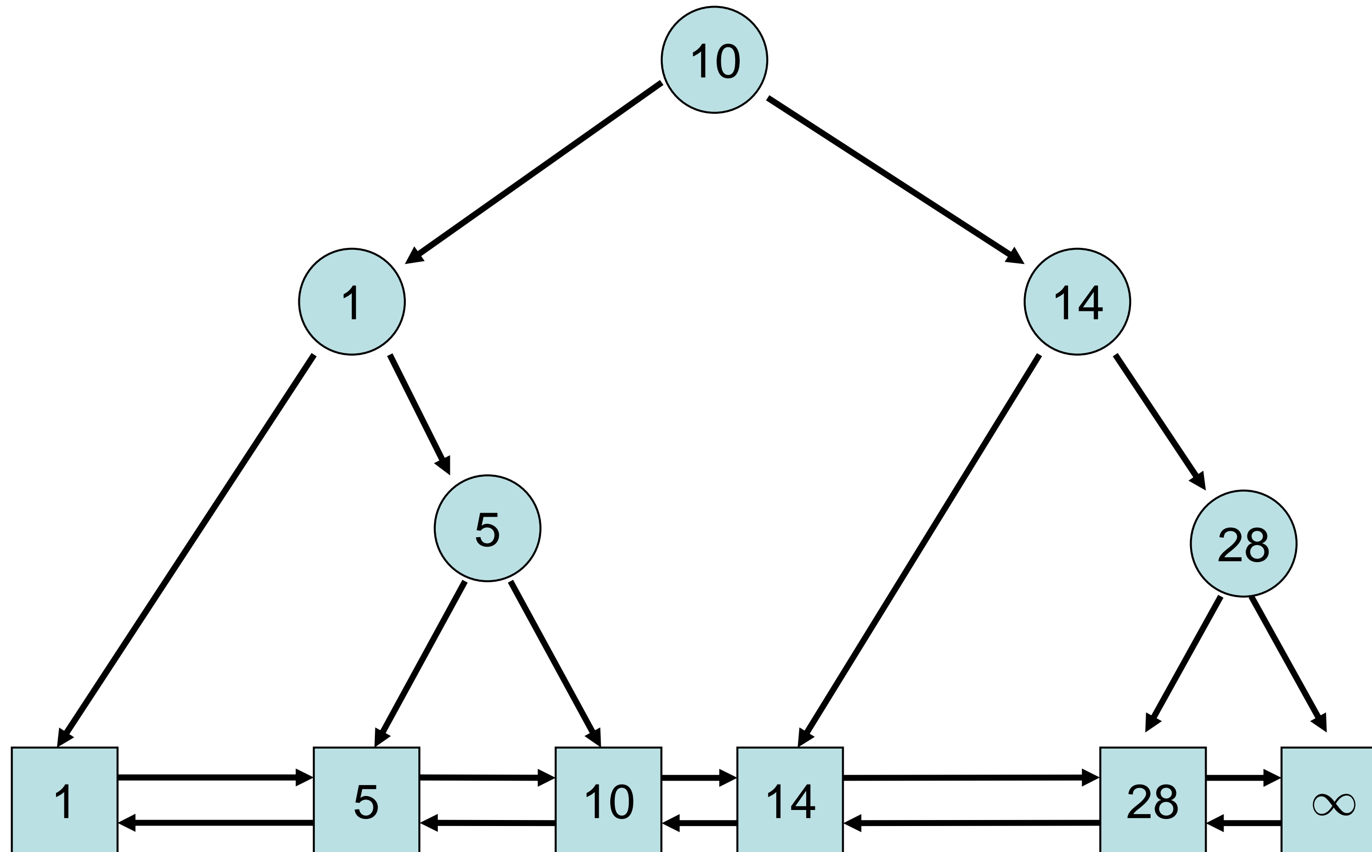
Insert(5)



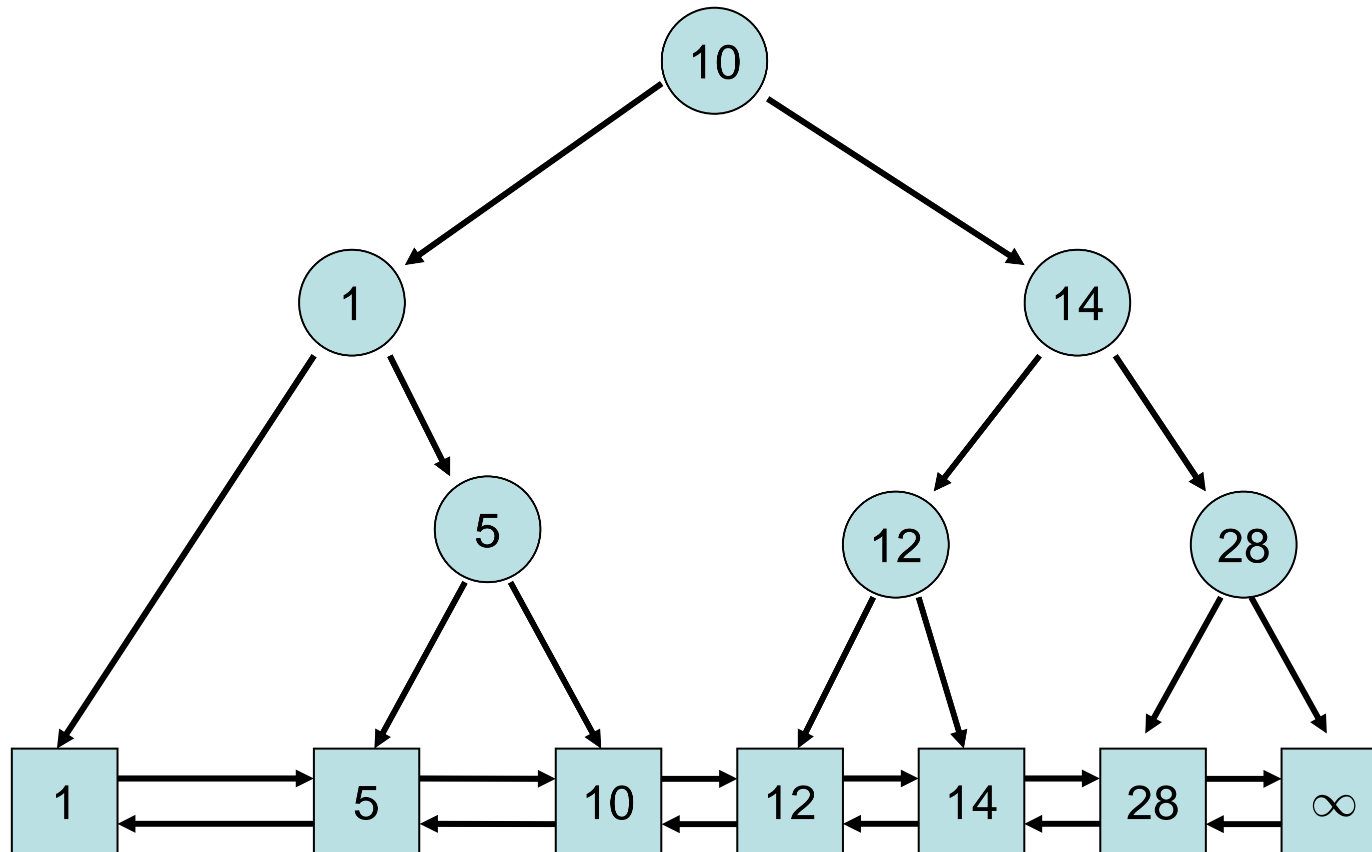
Insert(5)



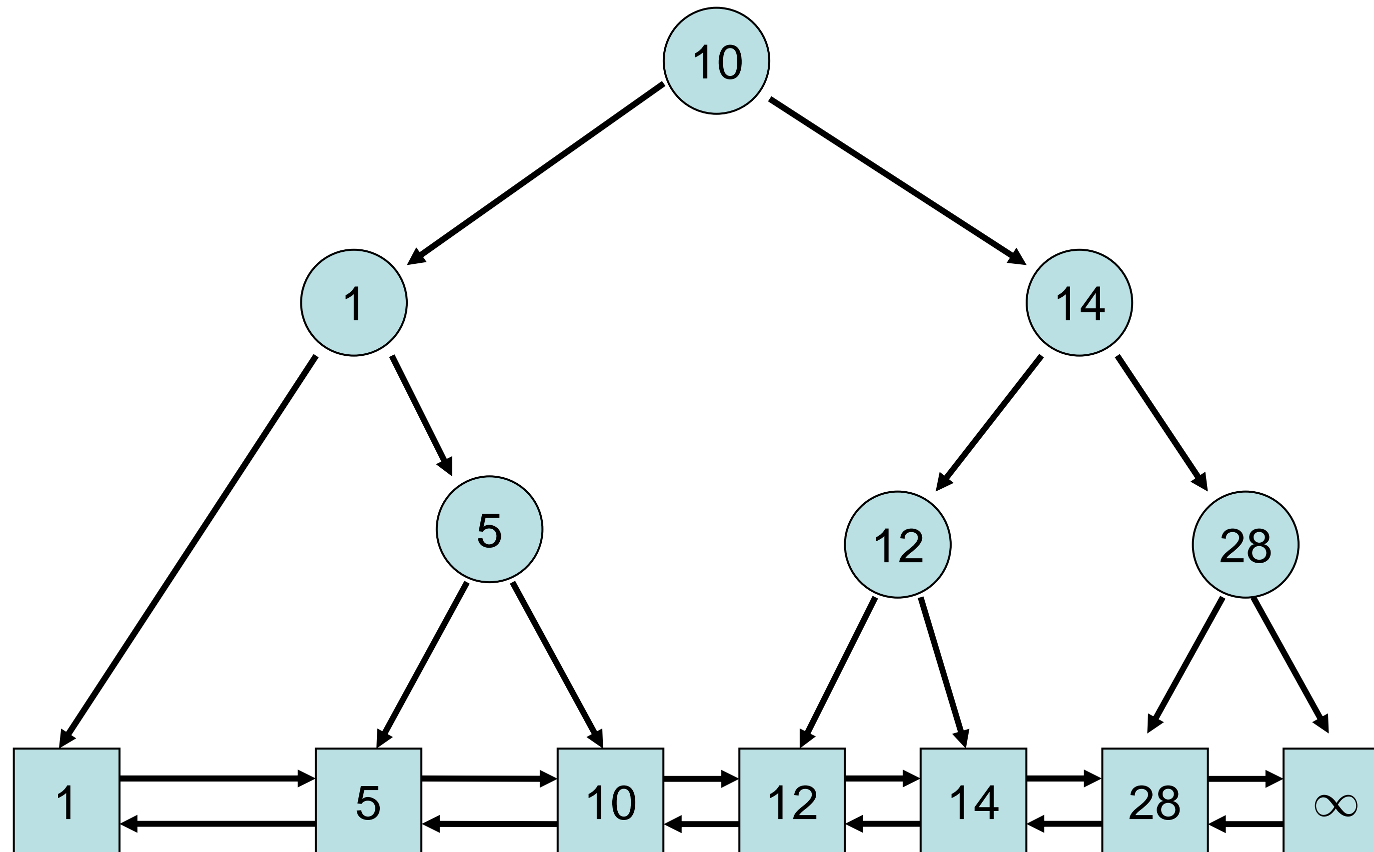
Insert(12)



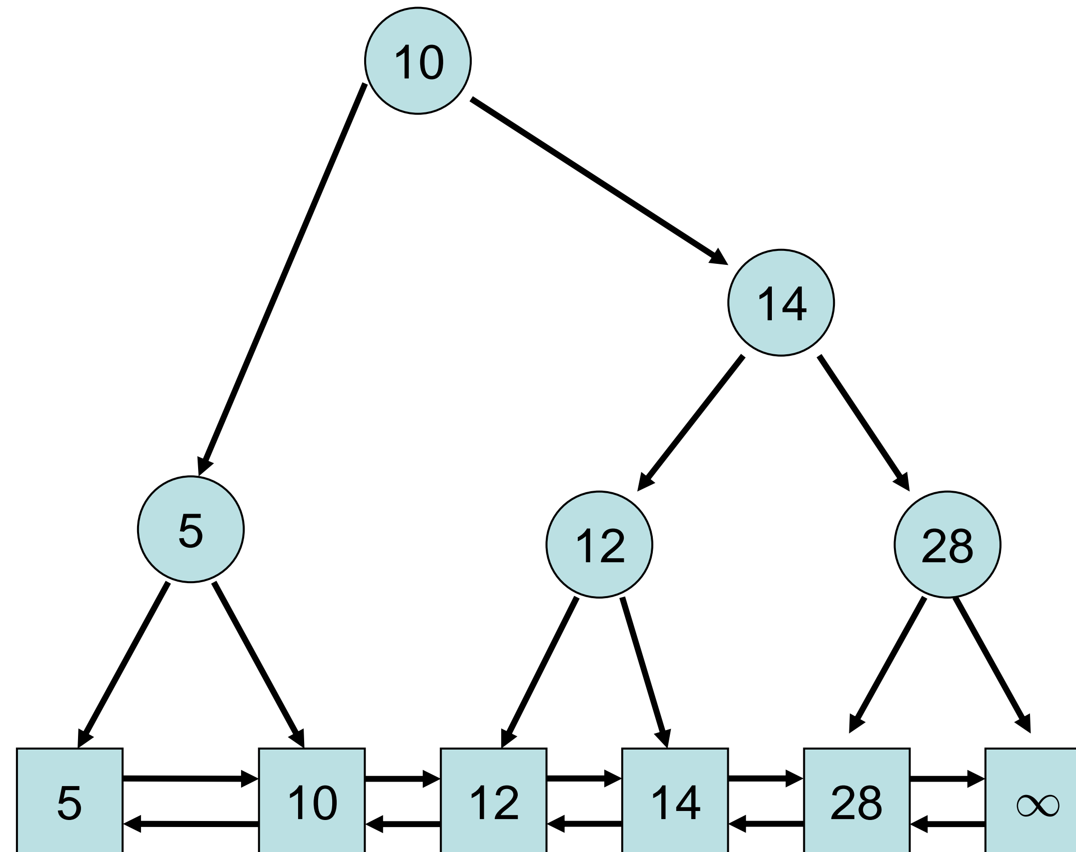
Insert(12)



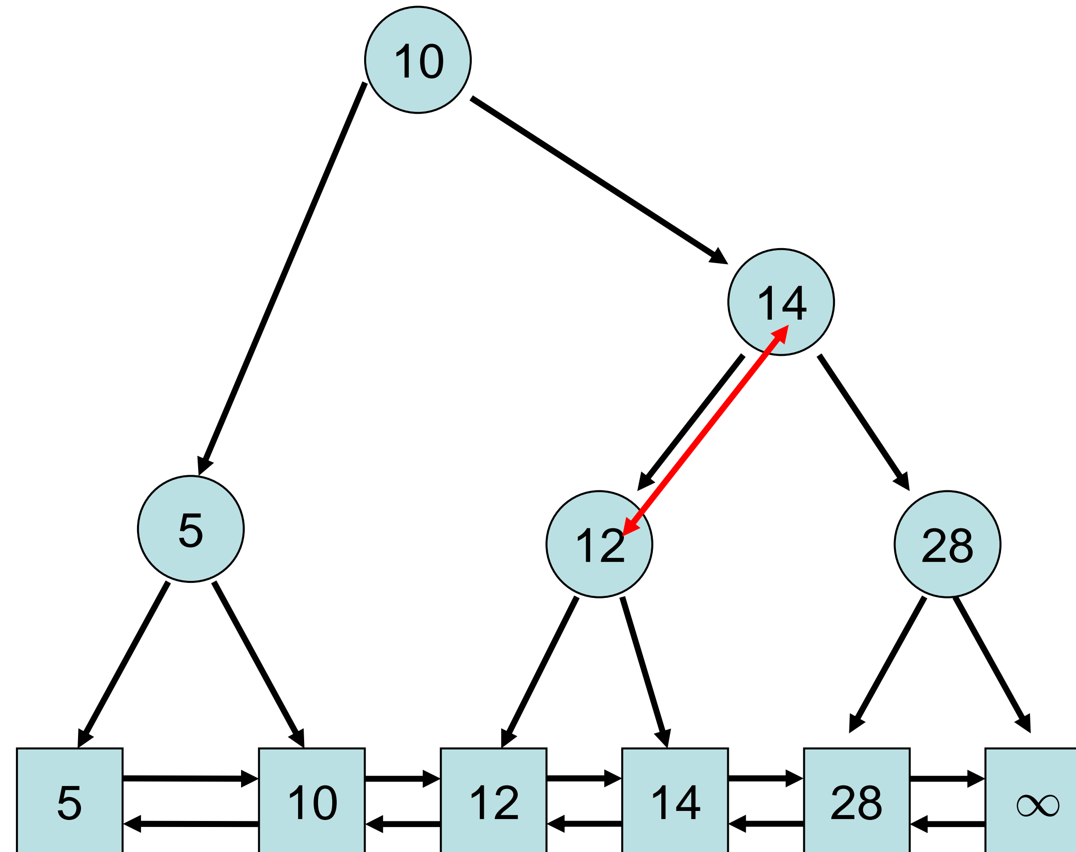
Delete(1)



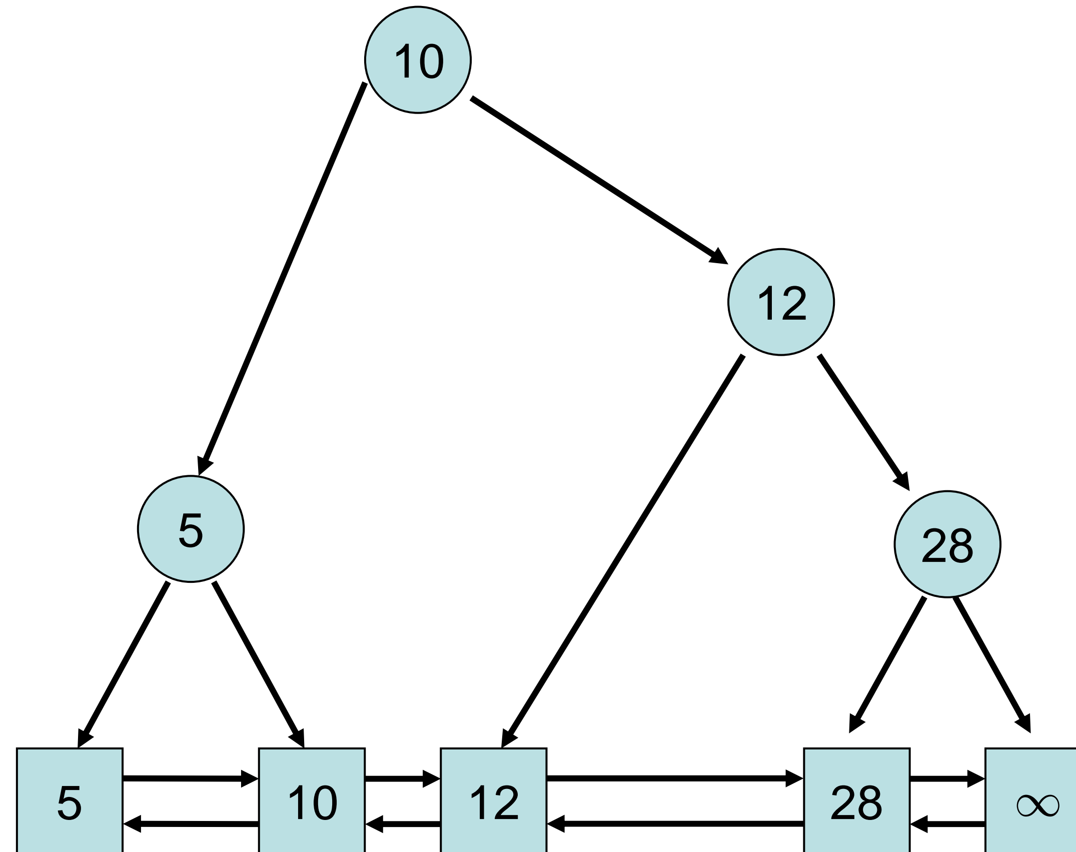
Delete(1)



Delete(14)



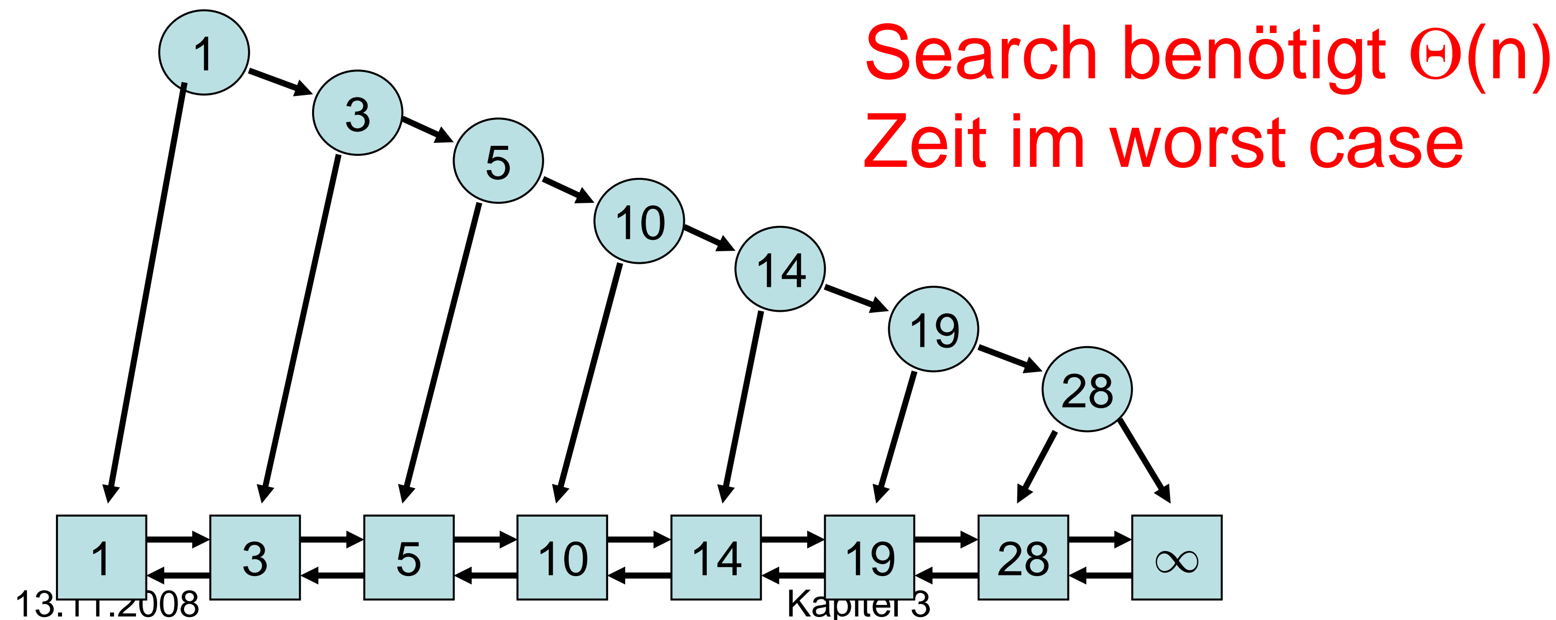
Delete(14)



Binärbaum

Problem: Binärbaum kann entarten!

Beispiel: Zahlen werden in sortierter Folge eingefügt



Binärbaum

Problem: Binärbaum kann entarten!

Lösungen:

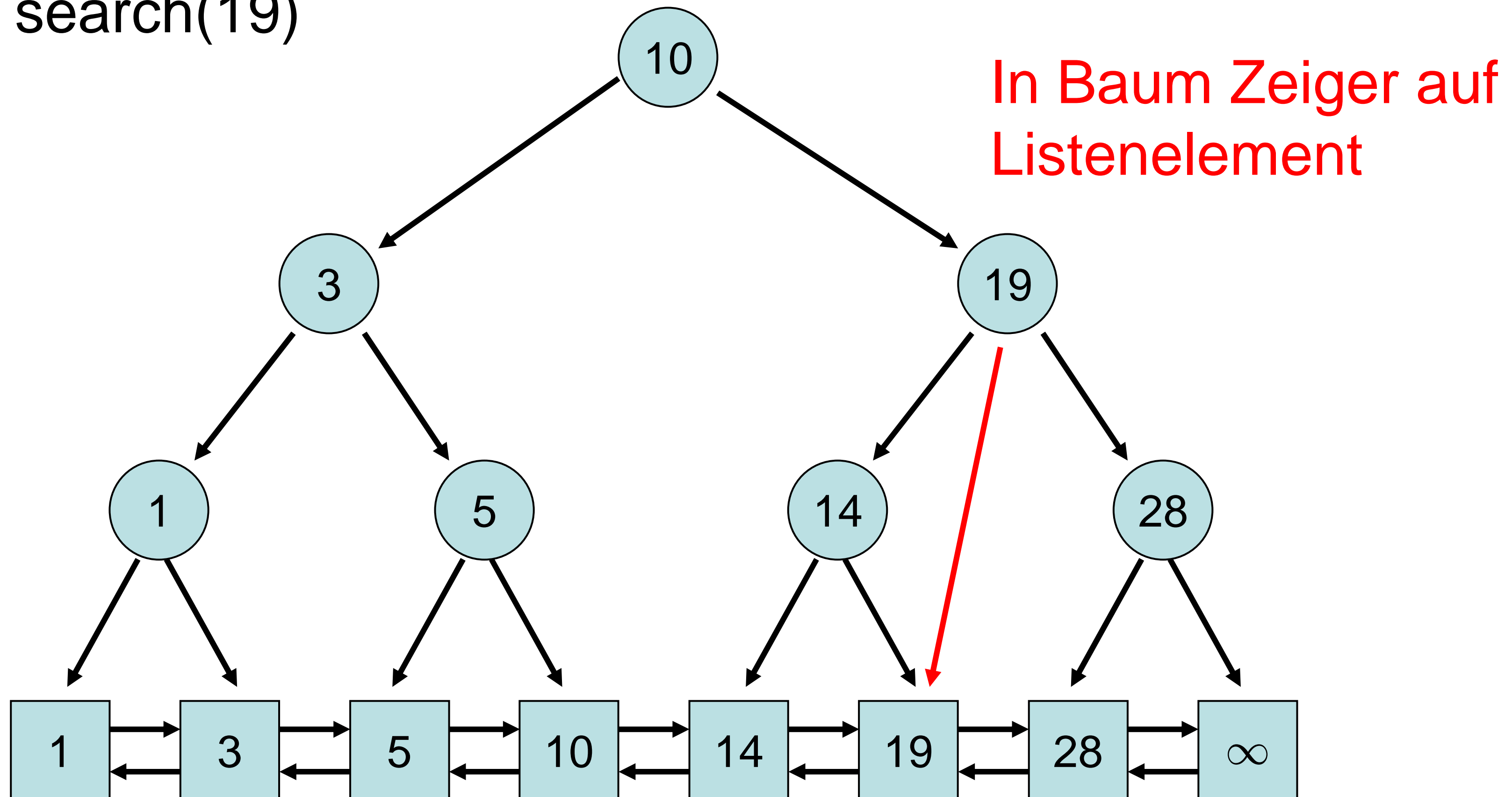
- Splay-Baum
  (sehr effektive Heuristik)
- Treaps
  (mit hoher Wkeit gut balanciert)
- (a,b)-Baum
  (garantiert gut balanciert)
- Rot-Schwarz-Baum
  (konstanter Reorganisationsaufwand)
- Gewichtsbalancierter Baum
  (kompakt einbettbar in Feld)

Splay-Baum

Üblicherweise: Implementierung als interner Suchbaum (d.h. Elemente direkt integriert in Baum und nicht in extra Liste)

Hier: Implementierung als externer Such-baum (wie beim Binärbaum oben)

Splay-Baum



Splay-Baum

Ideen:

1. Im Baum Zeiger auf Listenelemente
2. Bewege Schlüssel von zugegriffenem Element immer zur Wurzel

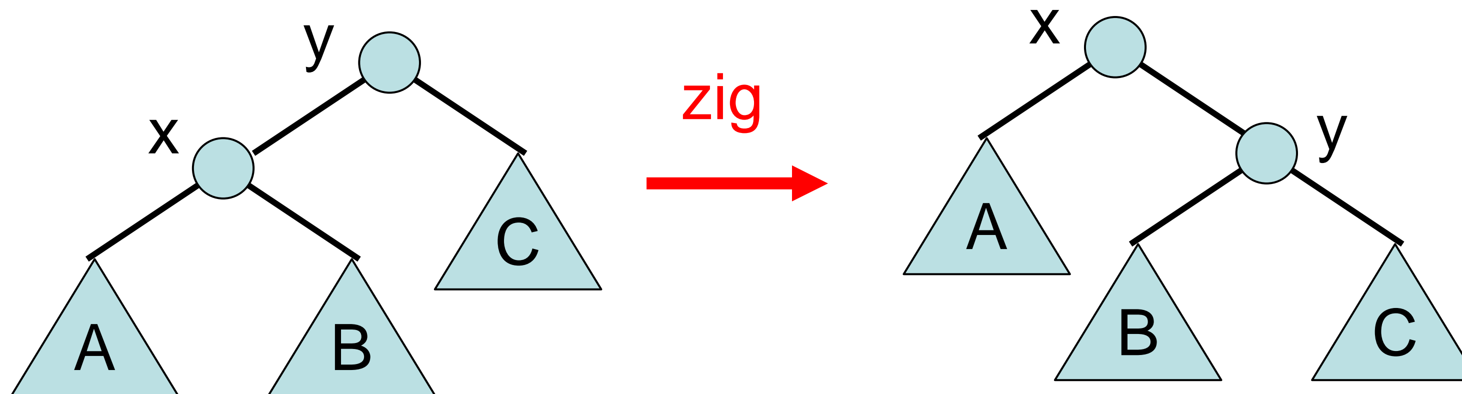
2. Idee: über Splay-Operation

Splay-Operation

Bewegung von Schlüssel x nach oben:

Wir unterscheiden zwischen 3 Fällen.

1a. x ist Kind der Wurzel:

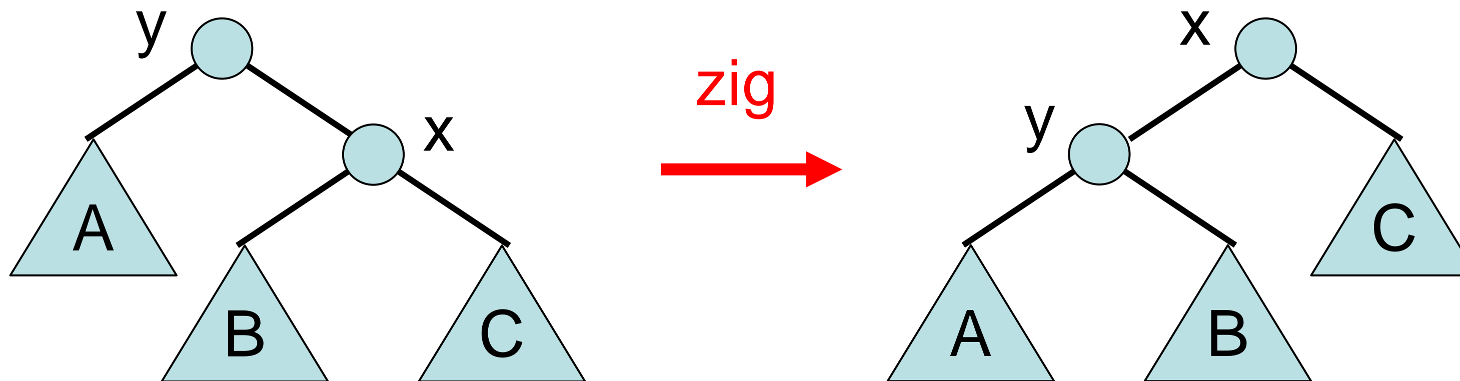


Splay-Operation

Bewegung von Schlüssel x nach oben:

Wir unterscheiden zwischen 3 Fällen.

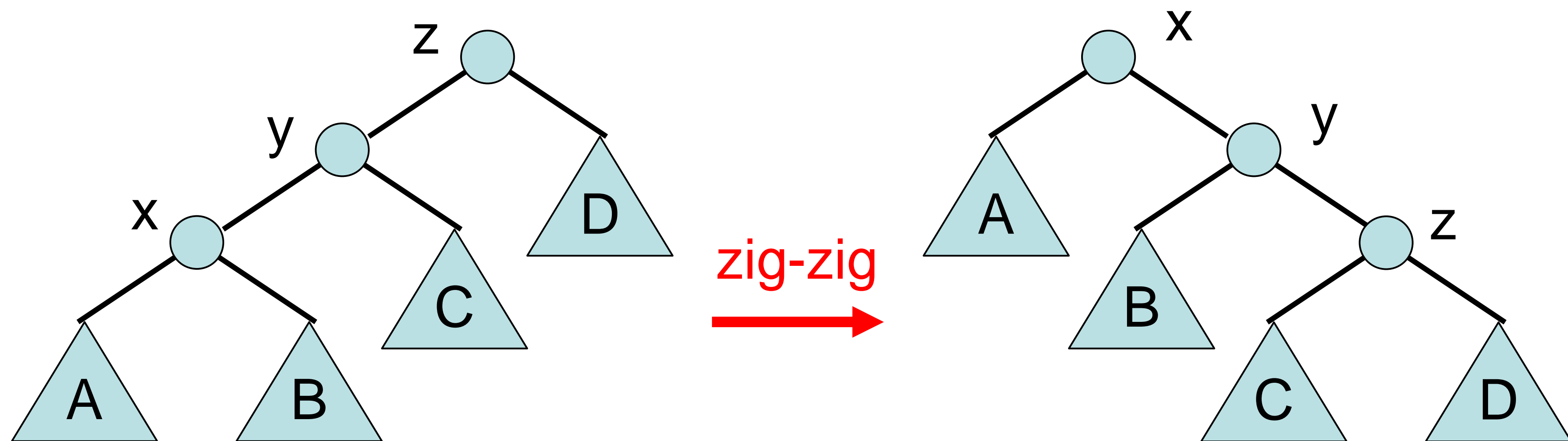
1b. x ist Kind der Wurzel:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

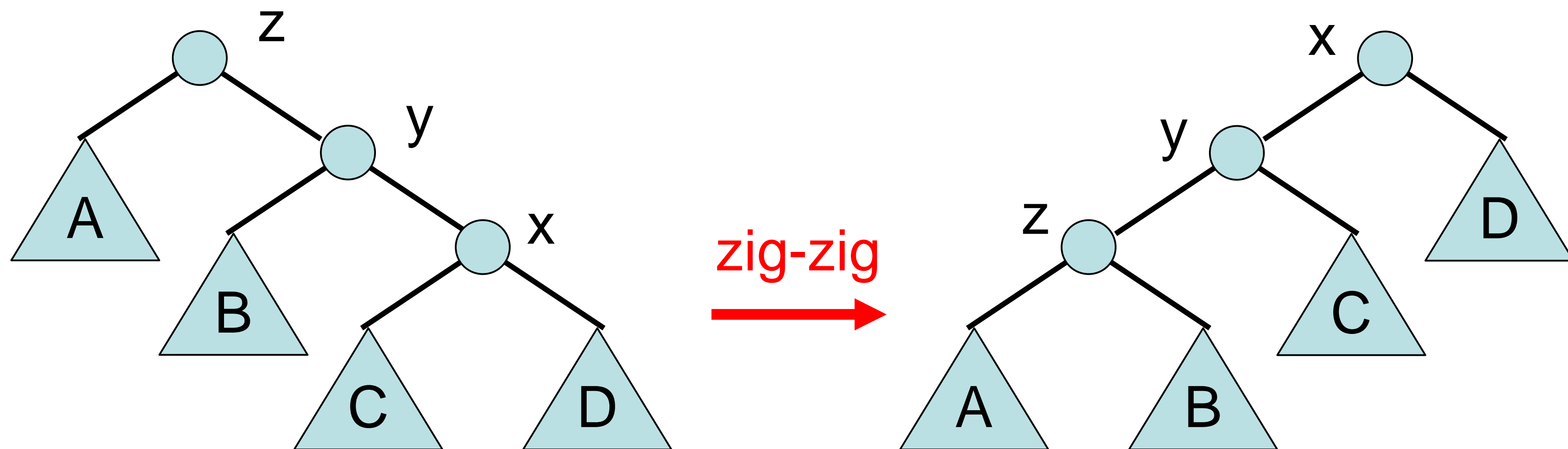
2a. x hat Vater und Großvater rechts:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

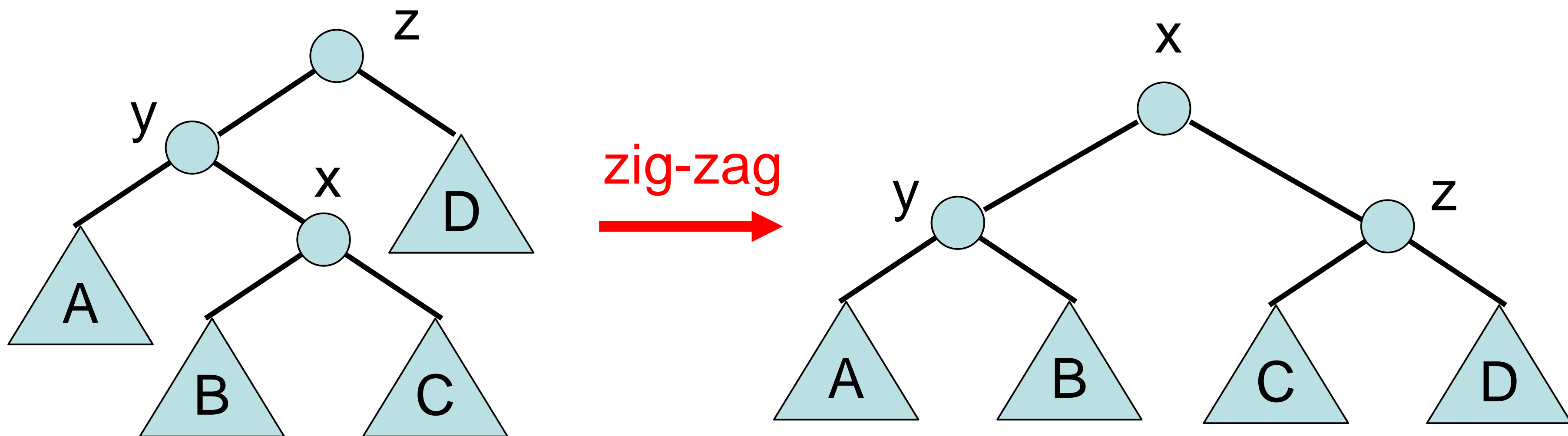
2b. x hat Vater und Großvater links:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

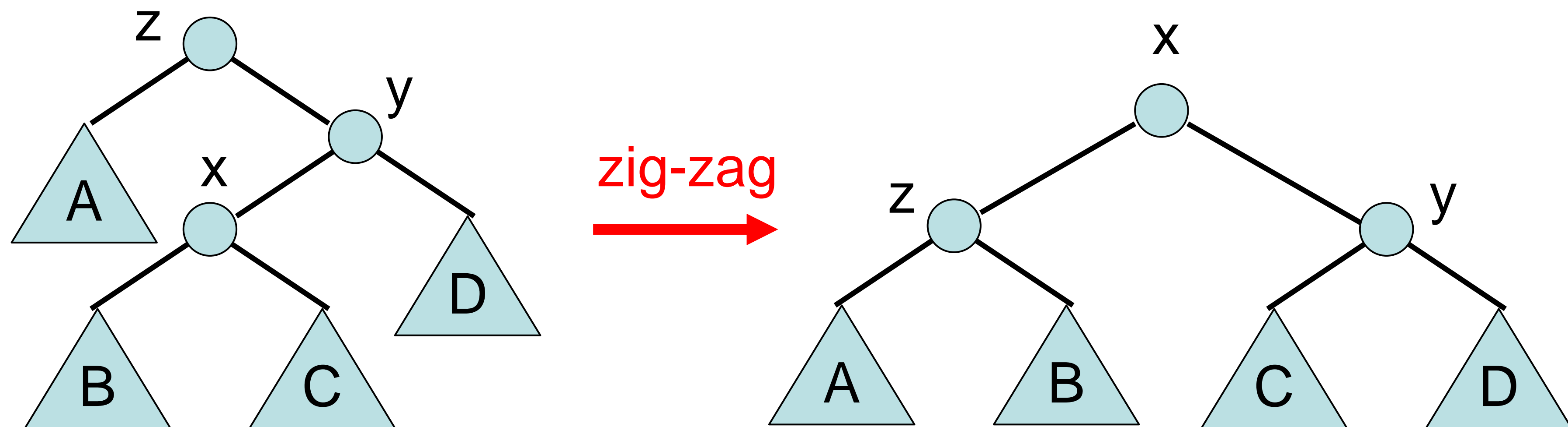
3a. x hat Vater links, Großvater rechts:



Splay-Operation

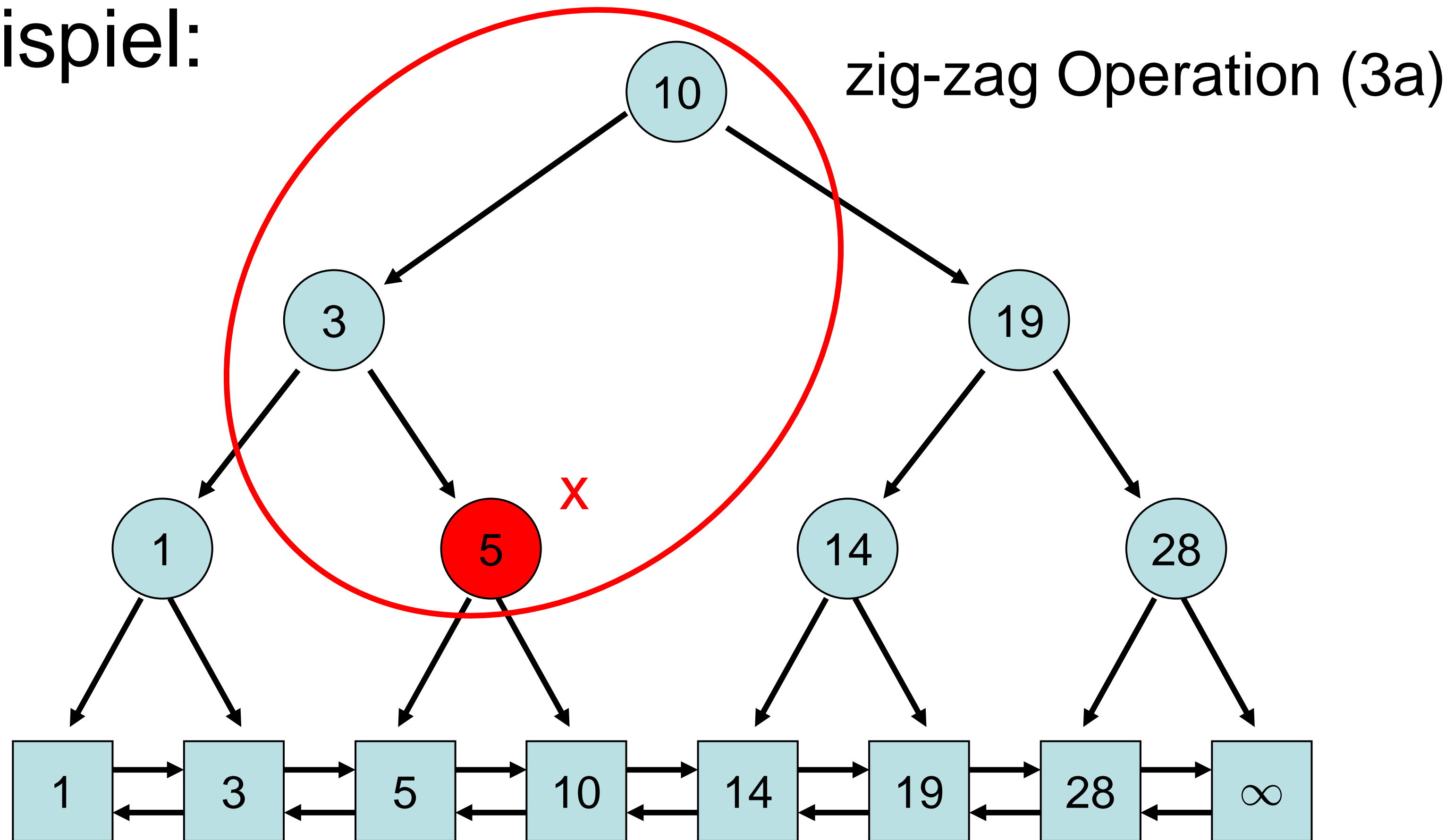
Wir unterscheiden zwischen 3 Fällen.

3b. x hat Vater rechts, Großvater links:

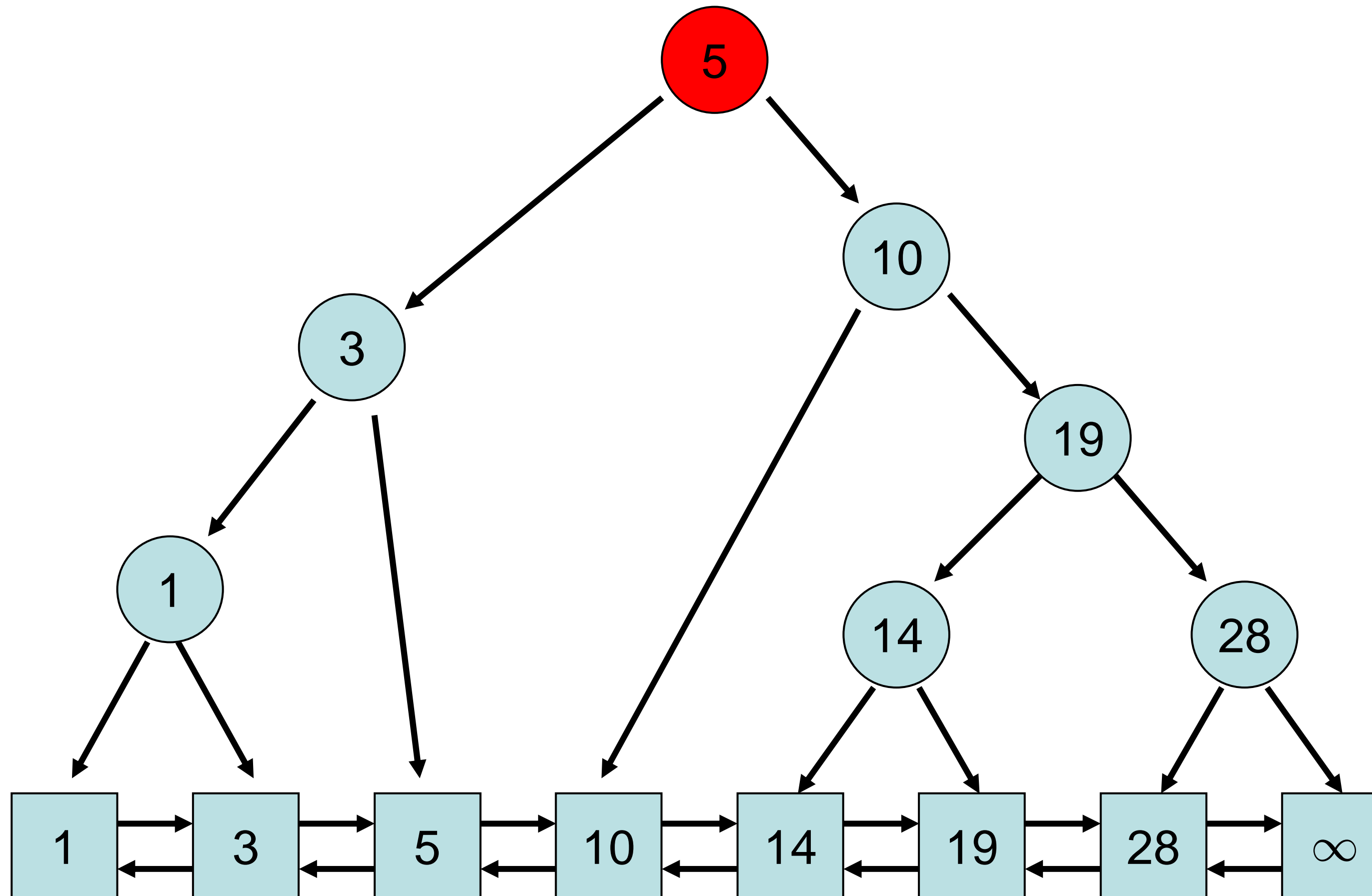


Splay-Operation

Beispiel:

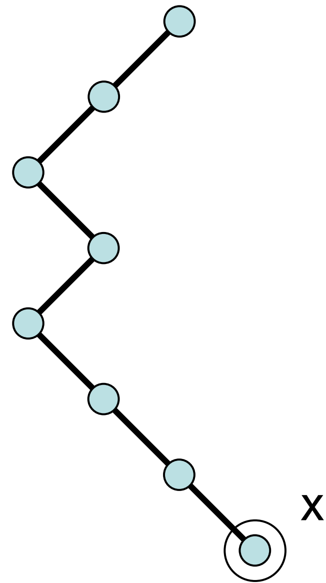


Splay-Operation

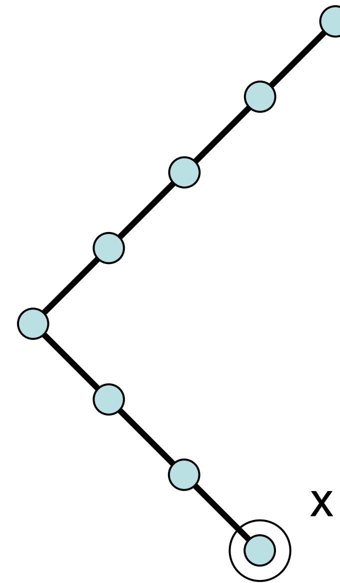


Splay-Operation

Beispiele:



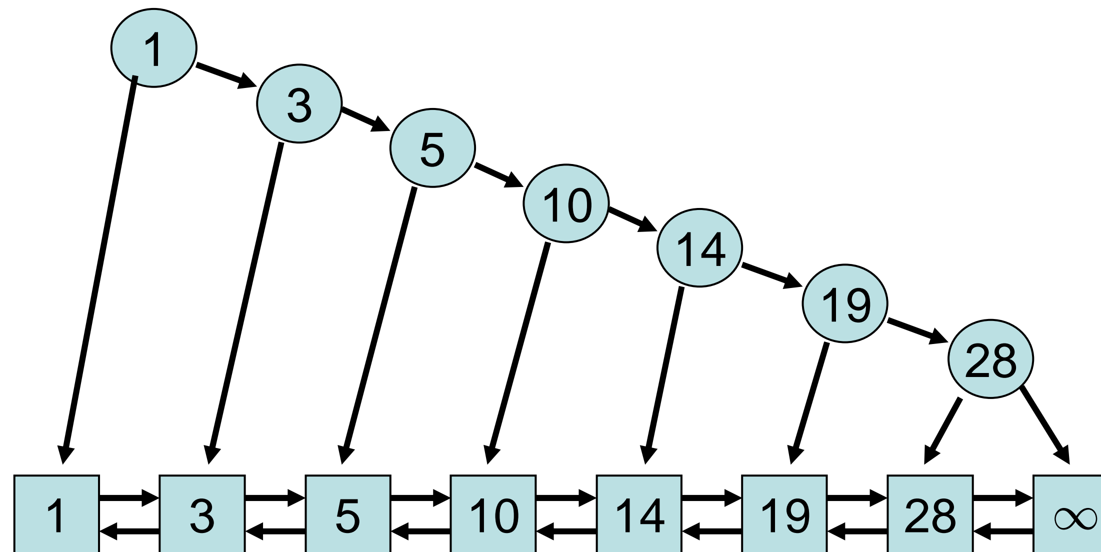
zig-zig, zig-zag, zig-zag, zig



zig-zig, zig-zag, zig-zig, zig

Splay-Operation

Baum kann im worst-case immer noch sehr unbalanciert werden! Aber amortisierte Kosten sind sehr niedrig.



Splay-Operation

search(k)-Operation: (exakte Suche)

- Laufe von Wurzel startend nach unten, bis k im Baumknoten gefunden (Abkürzung zur Liste) oder bei Liste angekommen
- k in Baum: rufe splay(k) auf

Amortisierte Analyse:

m Splay-Operationen auf beliebigem Anfangsbaum mit n Elementen (m>n)

Splay-Operation

- Gewicht von Knoten x: $w(x)$
- Baumgewicht von Baum T mit Wurzel x: $tw(x)= \sum_{y \in T} w(y)$
- Rang von Knoten x: $r(x) = \log(tw(x))$
- Potential von Baum T: $\phi(T) = \sum_{x \in T} r(x)$

Lemma 3.1: Sei T ein Splay-Baum mit Wurzel u und x ein Knoten in T. Die amortisierten Kosten für splay(x,T) sind max. $1+3(r(u)-r(x))$.

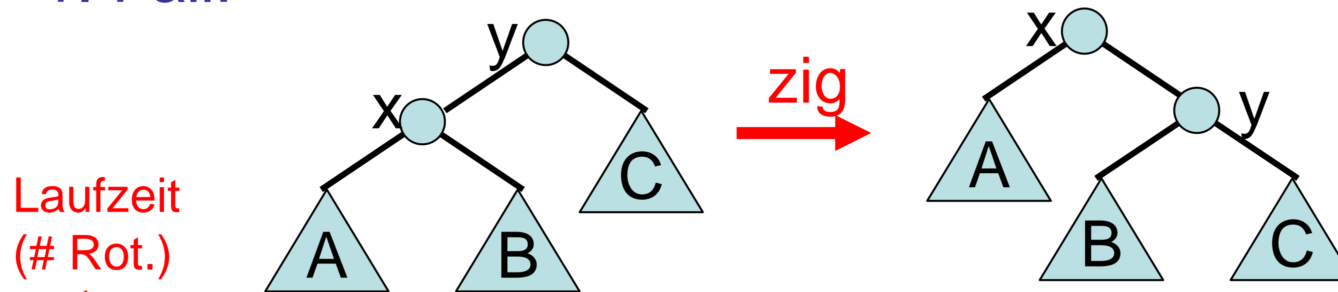
Splay-Operation

Beweis von Lemma 3.1:

Induktion über die Folge der Rotationen.

- r und tw : Rang und Gewicht vor Rotation
- r' und tw': Rang und Gewicht nach Rotation

1. Fall:



Amortisierte Kosten:

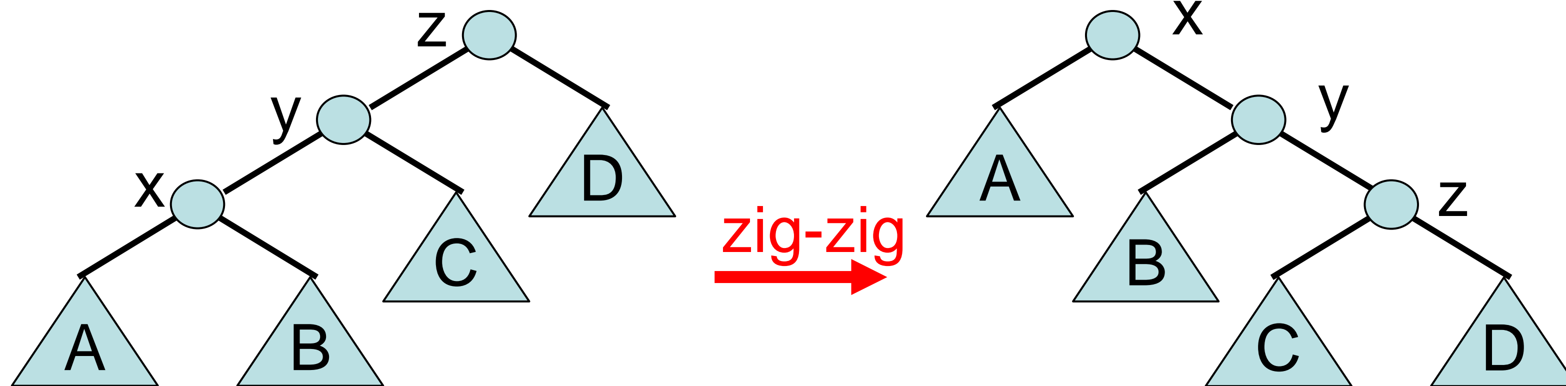
$\leq 1 + r'(x) + r'(y) - r(x) - r(y) \leq 1 + r'(x) - r(x)$ da $r'(y) \leq r(y)$

$\leq 1 + 3(r'(x) - r(x))$ da $r'(x) \geq r(x)$

Änderung von ϕ

Splay-Operation

2. Fall:



Amortisierte Kosten:

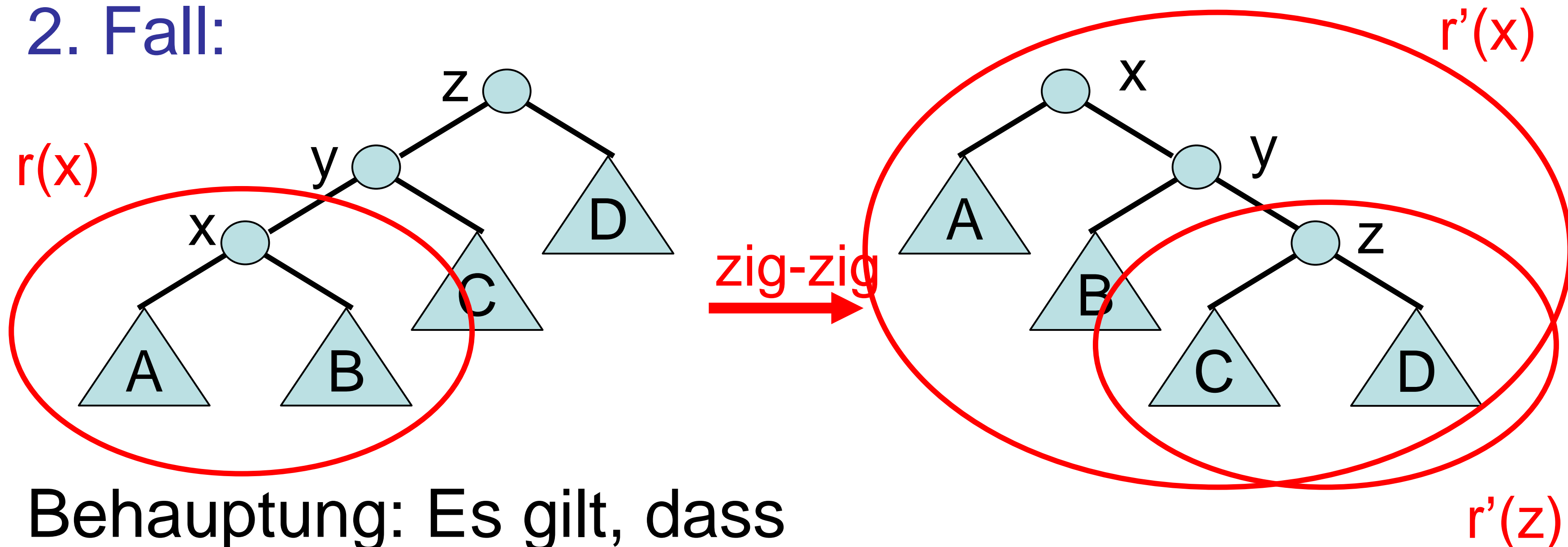
$\leq 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$

$= 2 + r'(y) + r'(z) - r(x) - r(y)$ da $r'(x) = r(z)$

$\leq 2 + r'(x) + r'(z) - 2r(x)$ da $r'(x) \geq r'(y)$ und $r(y) \geq r(x)$

Splay-Operation

2. Fall:



Behauptung: Es gilt, dass

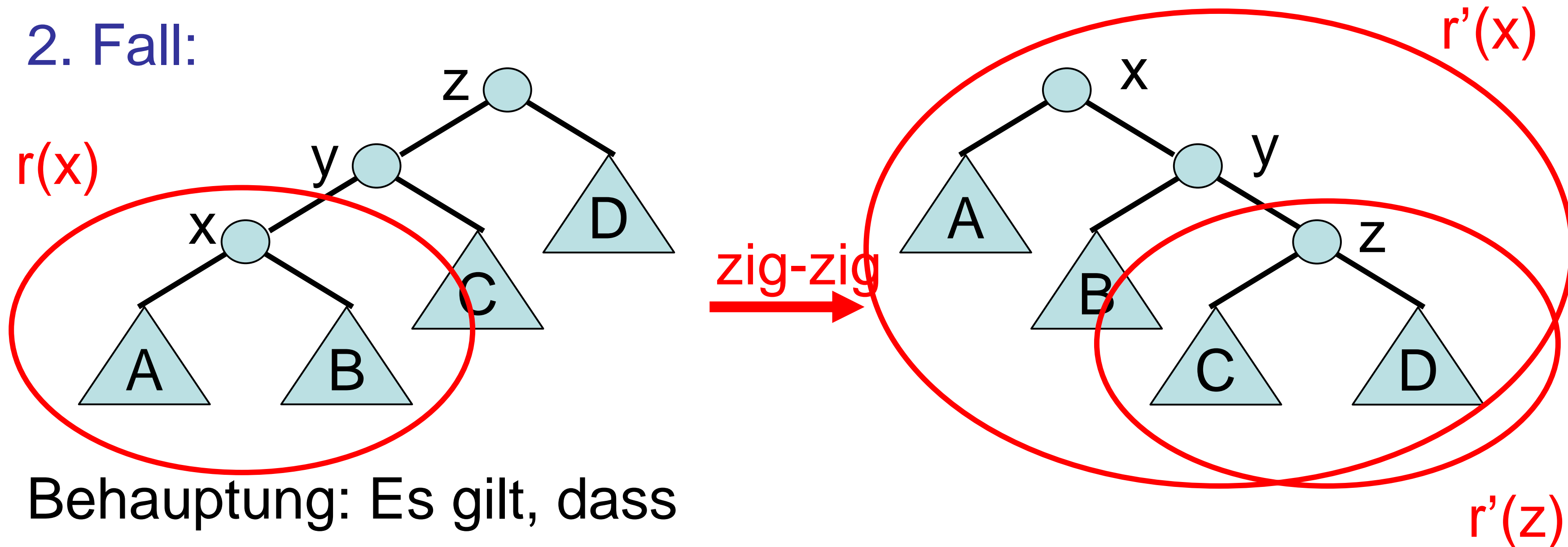
$$2+r'(x)+r'(z)-2r(x) <= 3(r'(x)-r(x))$$

d.h.

$$r(x)+r'(z) <= 2(r'(x)-1)$$

Splay-Operation

2. Fall:



Behauptung: Es gilt, dass

$$r(x)+r'(z) <= 2(r'(x)-1)$$

Ersetzungen: $r(x) \to \log x$, $r'(z) \to \log y$, $r'(x) \to \log 1$.

Betrachte die Funktion $f(x,y)=\log x + \log y$.

Zu zeigen: $f(x,y) <= -2$ für alle $x,y>0$ mit $x+y<1$.

Splay-Operation

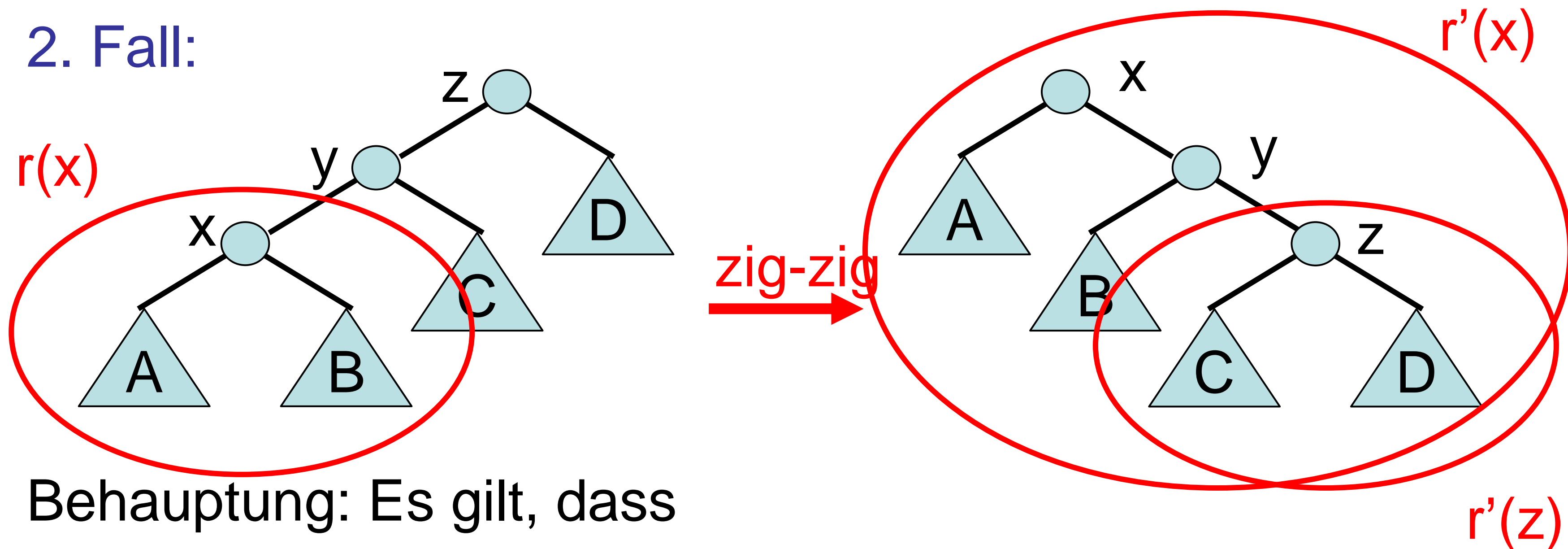
Lemma 3.2: Die Funktion $f(x,y)=\log x + \log y$ hat in dem Bereich $x,y>0$ mit $x+y<=1$ im Punkt $(½,½)$ ihr Maximum.

Beweis:

- Da die Funktion $\log x$ streng monoton wachsend ist, kann sich das Maximum nur auf dem Geradensegment $x+y=1$, $x,y>0$, befinden.
- Neues Maximierungsproblem: betrachte $g(x) = \log x + \log (1-x)$
- Einzige Nullstelle von $g'(x) = 1/x - 1/(1-x)$ ist $x=1/2$.
- Für $g''(x)= -(1/x^2 + 1/(1-x)^2))$ gilt $g''(1/2) < 0$.
- Also hat Funktion f im Punkt $(½,½)$ ihr Maximum.

Splay-Operation

2. Fall:



Behauptung: Es gilt, dass

$$r(x)+r'(z) <= 2(r'(x)-1)$$

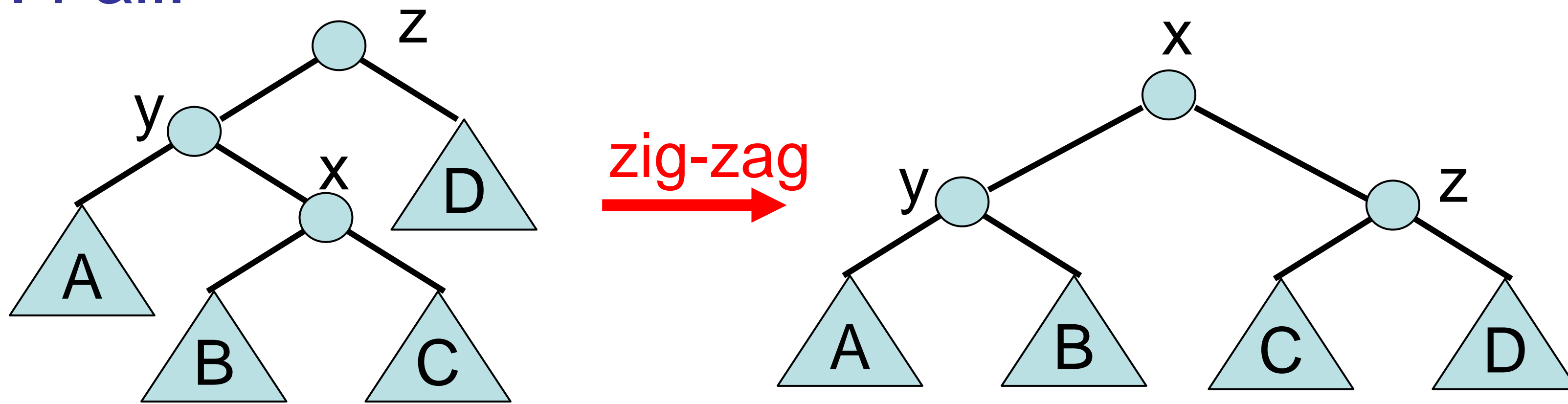
Ersetzungen: $r(x) \to \log x$, $r'(z) \to \log y$, $r'(x) \to \log 1$.

Es folgt: $f(x,y)=\log x + \log y < -2$ für alle $x,y>0$ mit $x+y<1$.

Die Behauptung ist also korrekt.

Splay-Operation

3. Fall:



Amortisierte Kosten:

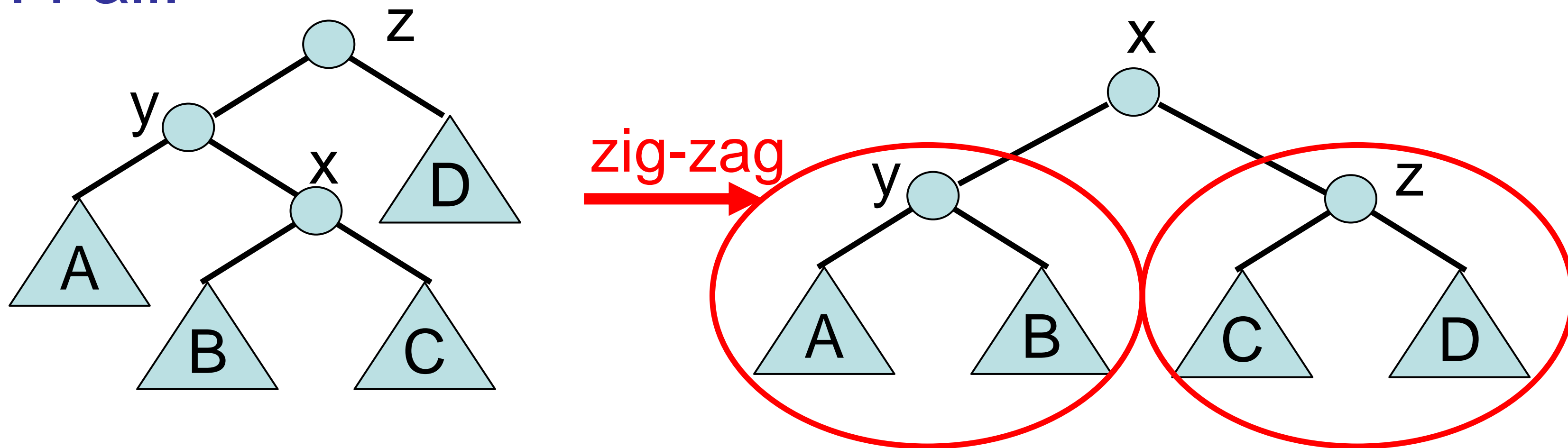
$\le 2+r'(x)+r'(y)+r'(z)-r(x)-r(y)-r(z)$

$\le 2+r'(y)+r'(z)-2r(x)$ da $r'(x)=r(z)$ und $r(x)\le r(y)$

$\le 2(r'(x)-r(x))$ denn…

Splay-Operation

3. Fall:



…es gilt:

$$2+r'(y)+r'(z)-2r(x) <= 2(r'(x)-r(x))$$

$$\Leftrightarrow \quad 2r'(x)-r'(y)-r'(z) >= 2$$

$$\Leftrightarrow \quad r'(y)+r'(z) <= 2(r'(x)-1)$$ analog zu Fall 2

Splay-Operation

Beweis von Lemma 3.1: (Fortsetzung)

Induktion über die Folge der Rotationen.

- r und tw : Rang und Gewicht vor Rotation
- r' und tw': Rang und Gewicht nach Rotation
- Für jede Rotation ergeben sich amortisierte Kosten von max. $1+3(r'(x)-r(x))$ (Fall 1) bzw. $3(r'(x)-r(x))$ (Fälle 2 und 3)
- Aufsummierung der Kosten ergibt max. $1 + \sum_{Rot.} 3(r'(x)-r(x)) = 1+3(r(u)-r(x))$

Splay-Operation

- Baumgewicht von Baum T mit Wurzel x:
  $tw(x)= \sum_{y \in T} w(y)$
- Rang von Knoten x: $r(x) = \log(tw(x))$
- Potential von Baum T: $\phi(T) = \sum_{x \in T} r(x)$

Lemma 3.1: Sei T ein Splay-Baum mit Wurzel u und x ein Knoten in T. Die amortisierten Kosten für $splay(x,T)$ sind max. $1+3(r(u)-r(x)) = 1+3 \cdot \log (tw(u)/tw(x))$.

Korollar 3.3; Sei $W=\sum_x w(x)$ und w_i das Gewicht von k_i in i-tem search. Für m search-Operationen sind die amortisierten Kosten $O(m+3\sum_{i=1}^m \log (W/w_i))$.

Splay-Baum

Theorem 3.4: Die Laufzeit für m search Operationen in einem n-elementigen Splay-Baum T ist höchstens

$$O(m+(m+n)\log n).$$

Beweis:

- Sei $w(x) = 1$ für alle Schlüssel x in T.
- Dann ist $W=n$ und $r(x) <= \log W = \log n$ für alle x in T.
- Erinnerung: für eine Operationsfolge F ist die Laufzeit $T(F) <= A(F) + \phi(s_0)$ für amortisierte Kosten A und Anfangszustand s_0
- $\phi(s_0) = \sum_{x \in T} r_0(x) <= n \log n$
- Aus Korollar 3.3 ergibt sich Theorem 3.4.

Splay-Baum

Angenommen, wir haben eine Wahrscheinlichkeitsverteilung für die Suchanfragen.

- $p(x)$: Wahrscheinlichkeit für Schlüssel x
- $H(p) = \sum_x p(x) \cdot \log(1/p(x))$: Entropie von p

Theorem 3.5: Die Laufzeit für m search Operationen in einem n-elementigen Splay-Baum T ist höchstens

$$O(m \cdot H(p) + n \cdot \log n).$$

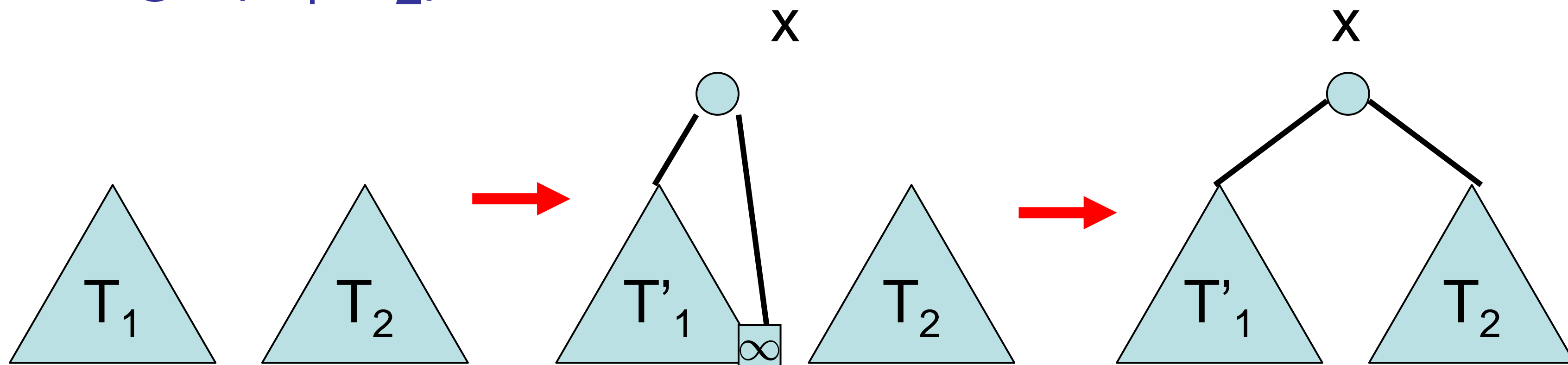
Beweis:
Folgt aus Theorem 3.4 mit $w(x) = n \cdot p(x)$ für alle x.

Laufzeit $\Omega(m \cdot H(p))$ für jeden statischen Suchbaum!

Splay-Baum Operationen

Annahme: zwei Splay-Bäume T_1 und T_2 mit $key(x)<key(y)$ für alle $x \in T_1$ und $y \in T_2$.

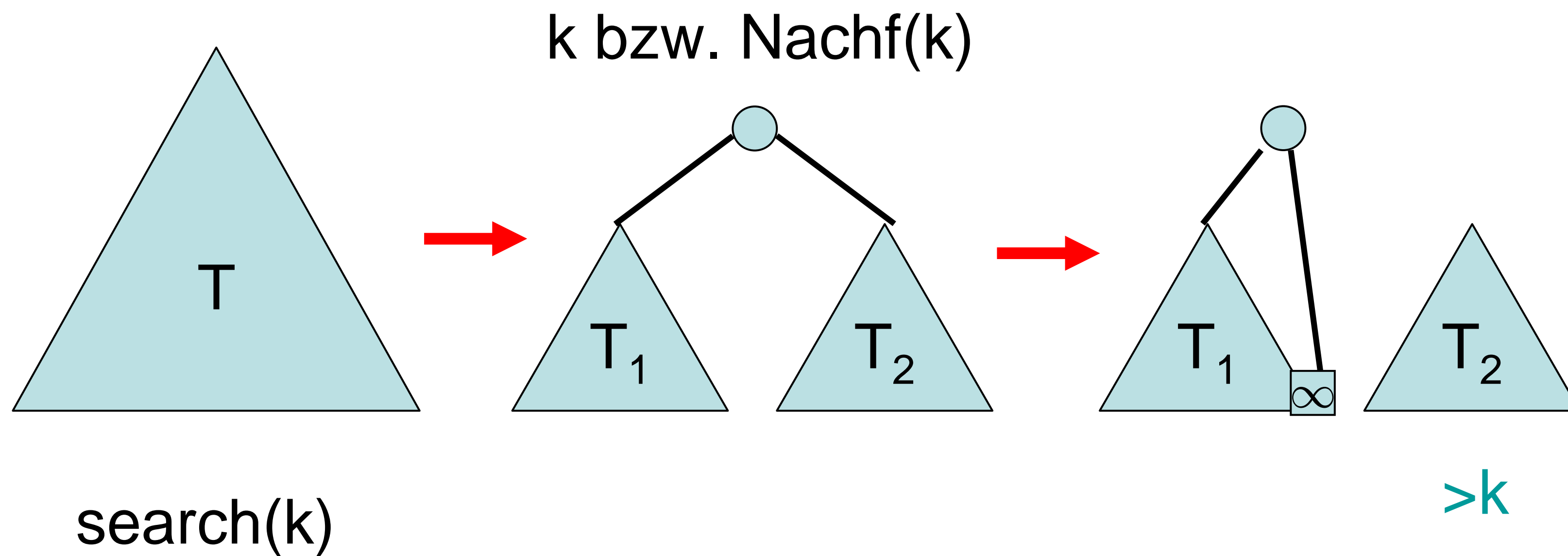
merge(T_1,T_2):



search(x), $x<\infty$ max. in T_1

Splay-Baum Operationen

split(k,T):



Splay-Baum Operationen

insert(e):

- insert wie im Binärbaum
- splay-Operation, um key(e) in Wurzel zu verschieben

delete(k):

- führe search(k) aus (bringt k in die Wurzel)
- entferne Wurzel und führe merge(T_1,T_2) der beiden Teilbäume durch

Splay-Operationen

- k_-: größter Schlüssel in T kleiner als k
- k_+: kleinster Schlüssel in T größer gleich k

Theorem 3.6: Die amortisierten Kosten der Operationen im Splay-Baum sind:

- search(k): $O(1+\log(W/w(k_+)))$
- split(k): $O(1+\log(W/w(k_+)))$
- merge(T_1,T_2): $O(1+\log(W/w(k_{max}(T_1))))$
- insert(e): $O(1+\log(W/w(key(e))))$
- delete(k): $O(1+\log(W/w(k_+))+\log(W/w(k_-)))$

Binärbaum

Problem: Binärbaum kann entarten!

Lösungen:

- Splay-Baum
  (sehr effektive Heuristik)
- Treaps
  (mit hoher Wkeit gut balanciert)
- (a,b)-Baum
  (garantiert gut balanciert)
- Rot-Schwarz-Baum
  (konstanter Reorganisationsaufwand)
- Gewichtsbalancierter Baum
  (kompakt einbettbar in Feld)

Treaps

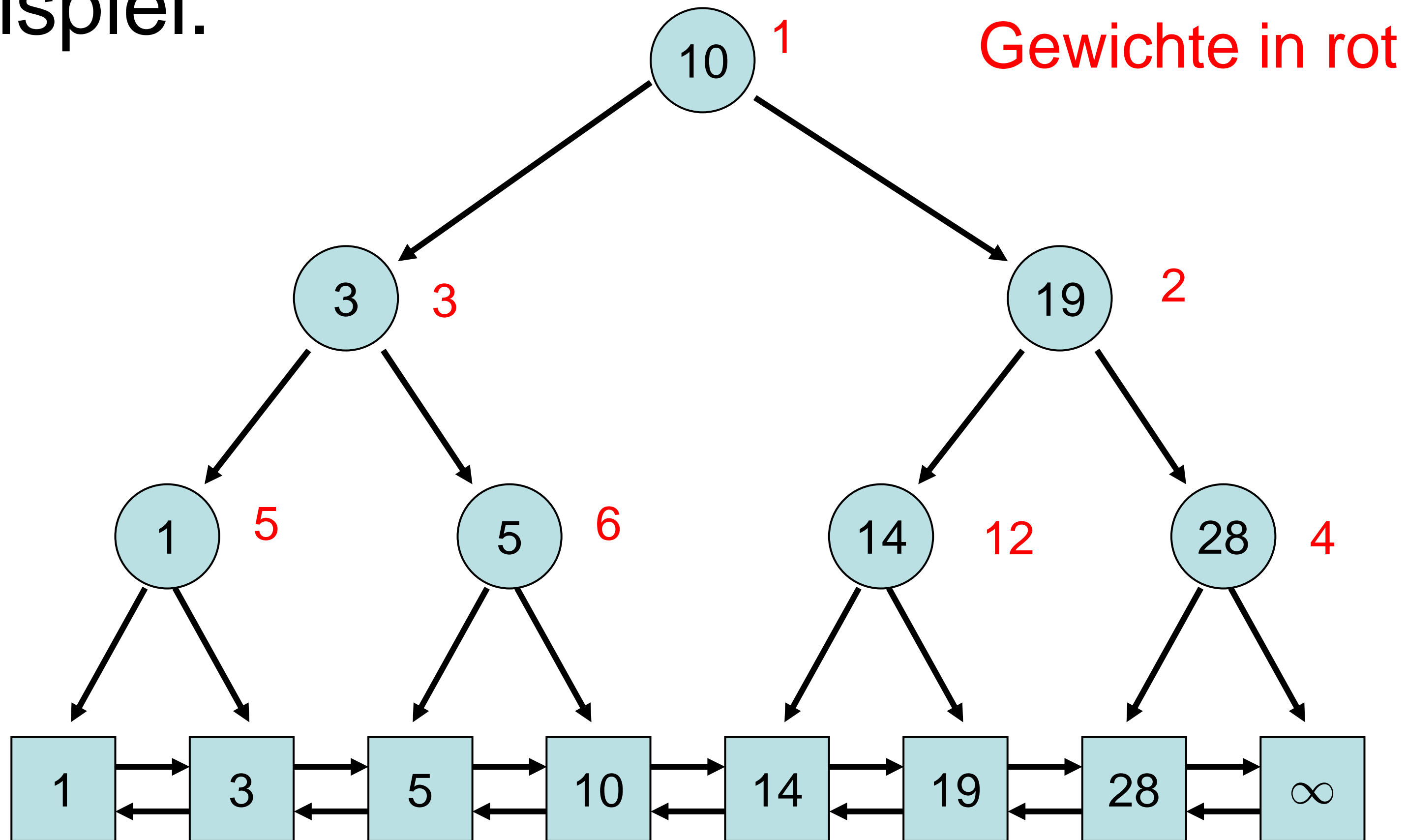
- $K=\{k_1,\ldots,k_n\}$: Menge von Schlüsseln
- $w(k_i)$: Gewicht von Schlüssel k_i

Ein Baum T zu K heißt Treap:

- T ist ein binärer Suchbaum für K
- T ist ein Heap bzgl. der Gewichte von K

Treaps

Beispiel:



Treaps

Search(k)-Operation:

- wie im binären Suchbaum

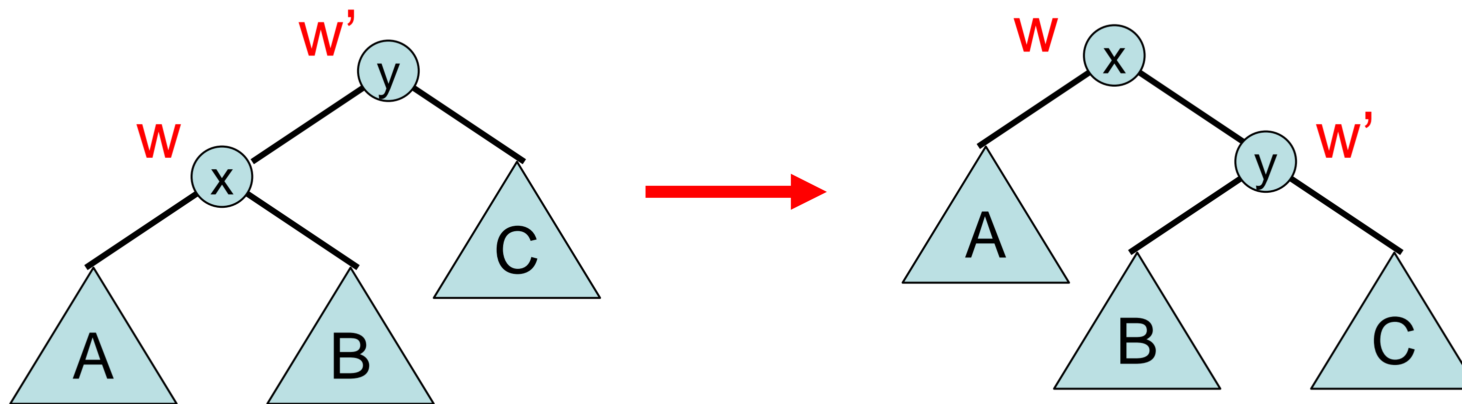
Insert(e)-Operation:

- führe insert(e) wie im binären Suchbaum aus
- siftup-Operation, um Heap-Eigenschaft sicherzustellen

Treaps

Siftup-Operation für x: wird über Rotationen durchgeführt

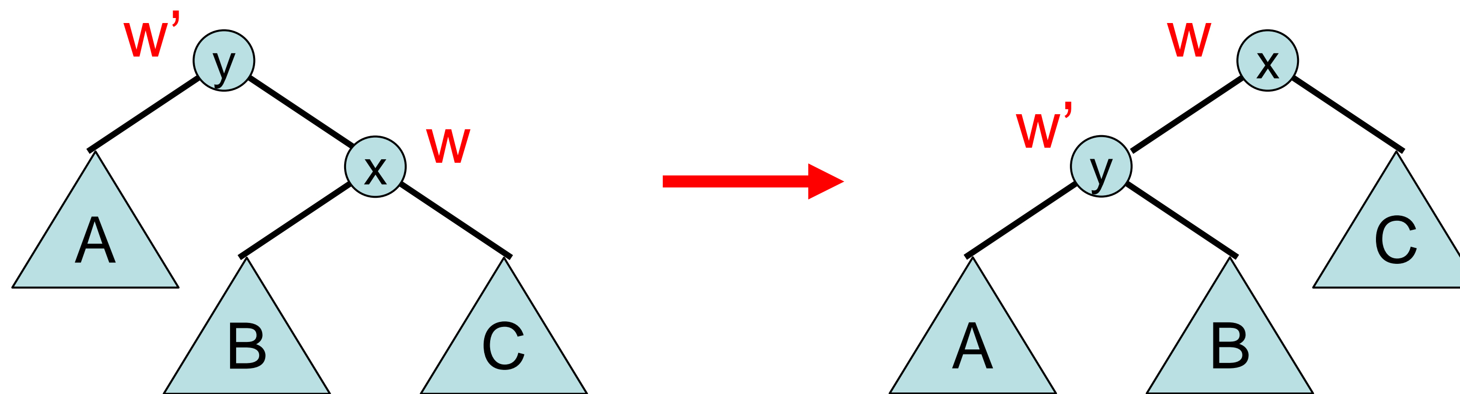
Fall 1: (w'>w)



Treaps

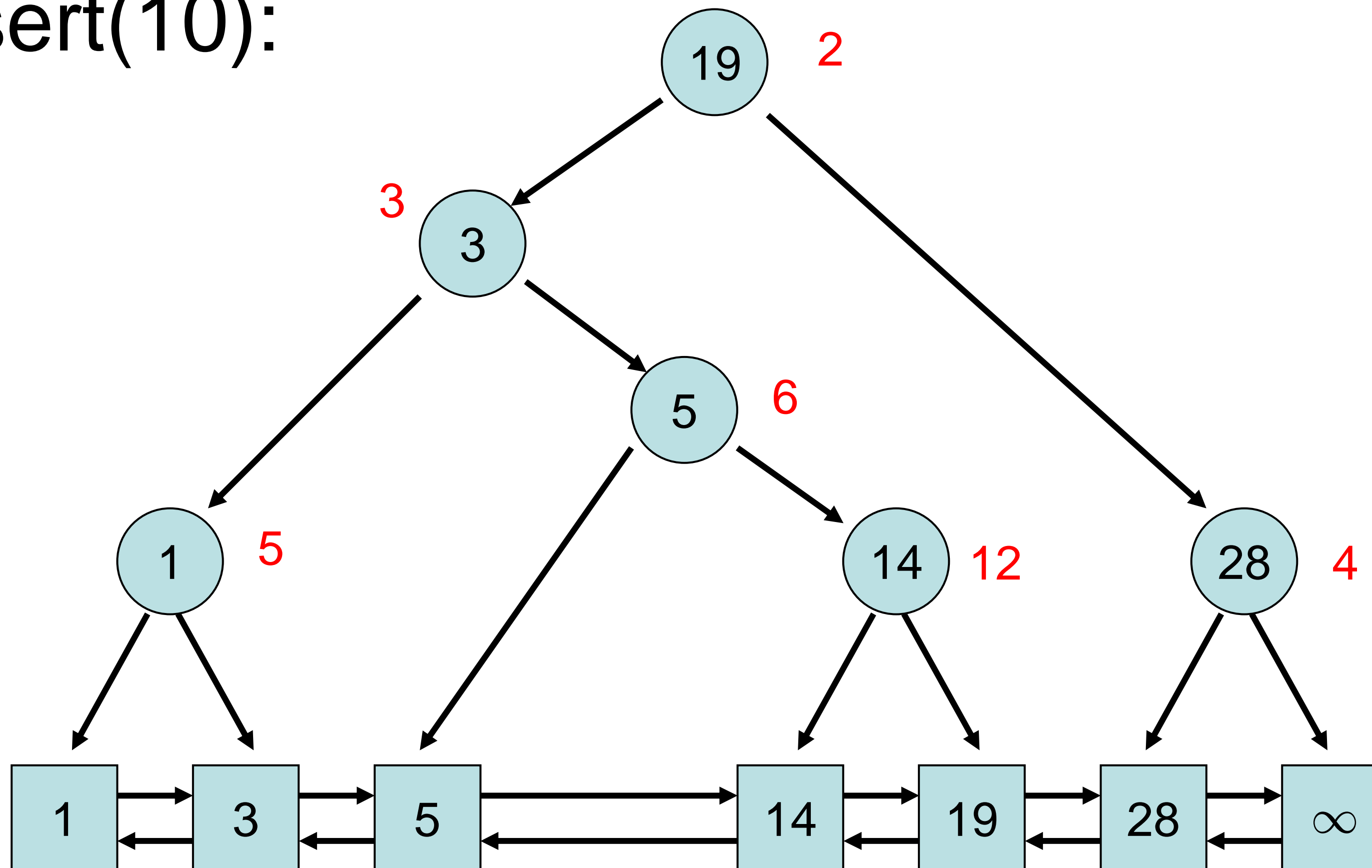
Siftup-Operation für x: wird über Rotationen durchgeführt

Fall 2: (w'>w)



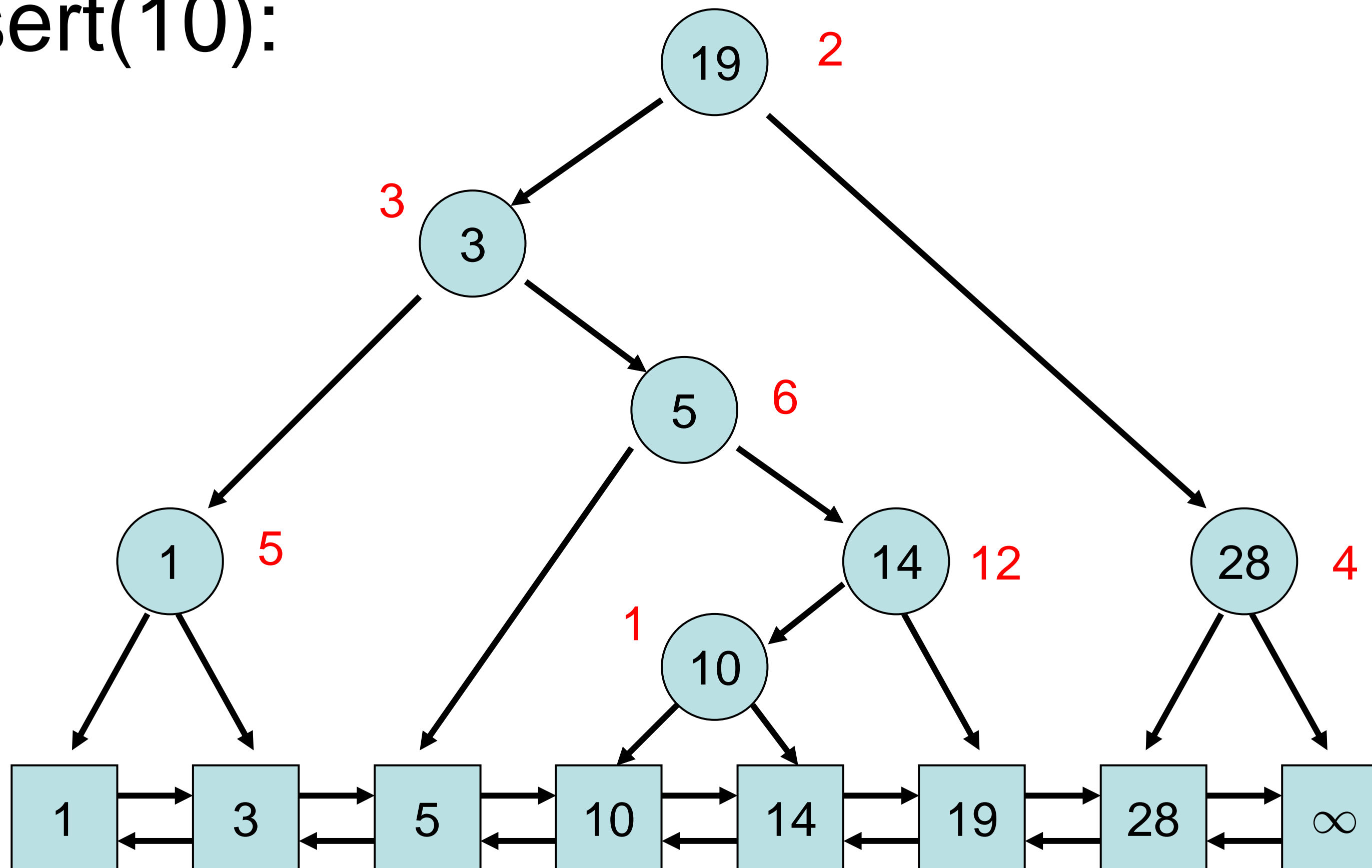
Treaps

Insert(10):



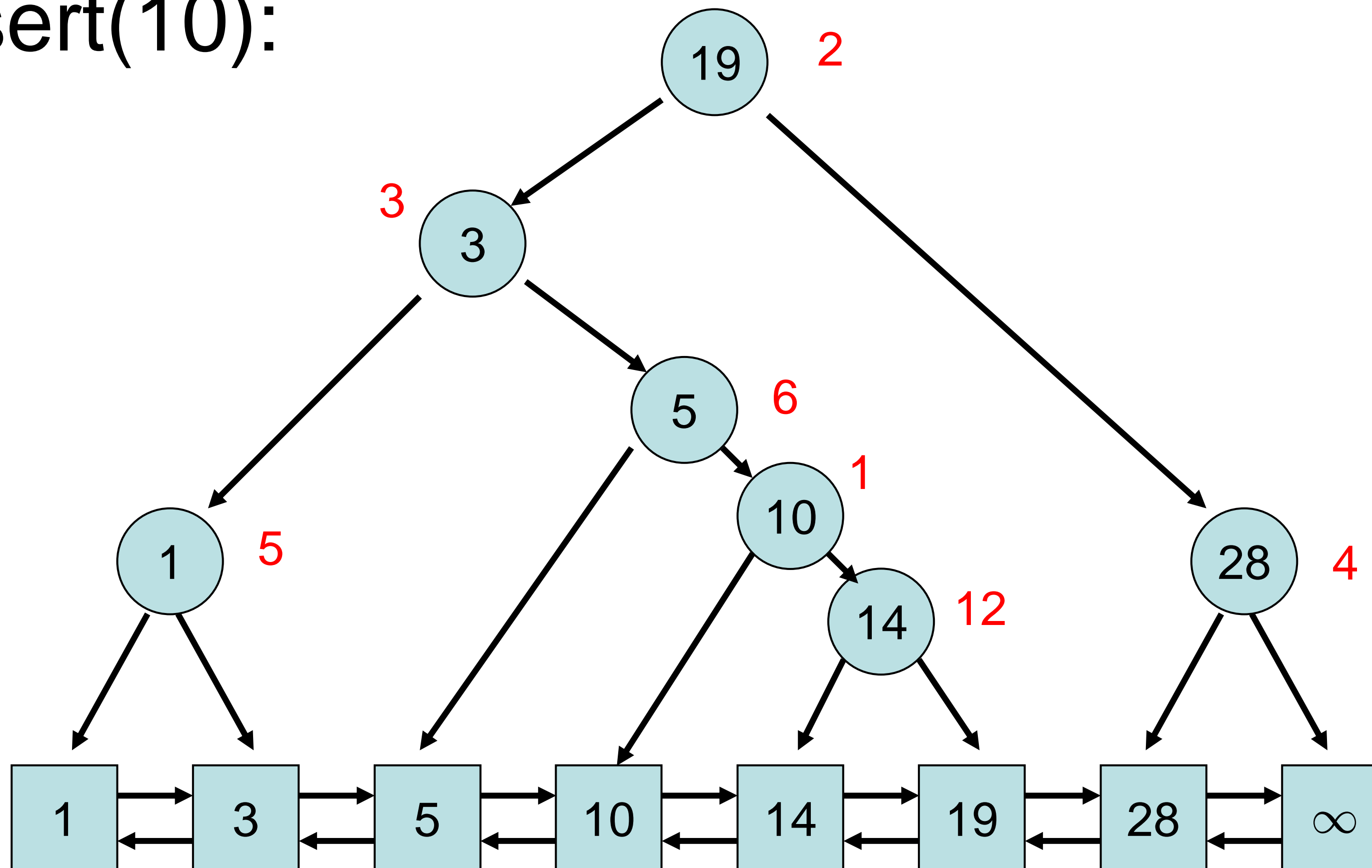
Treaps

Insert(10):



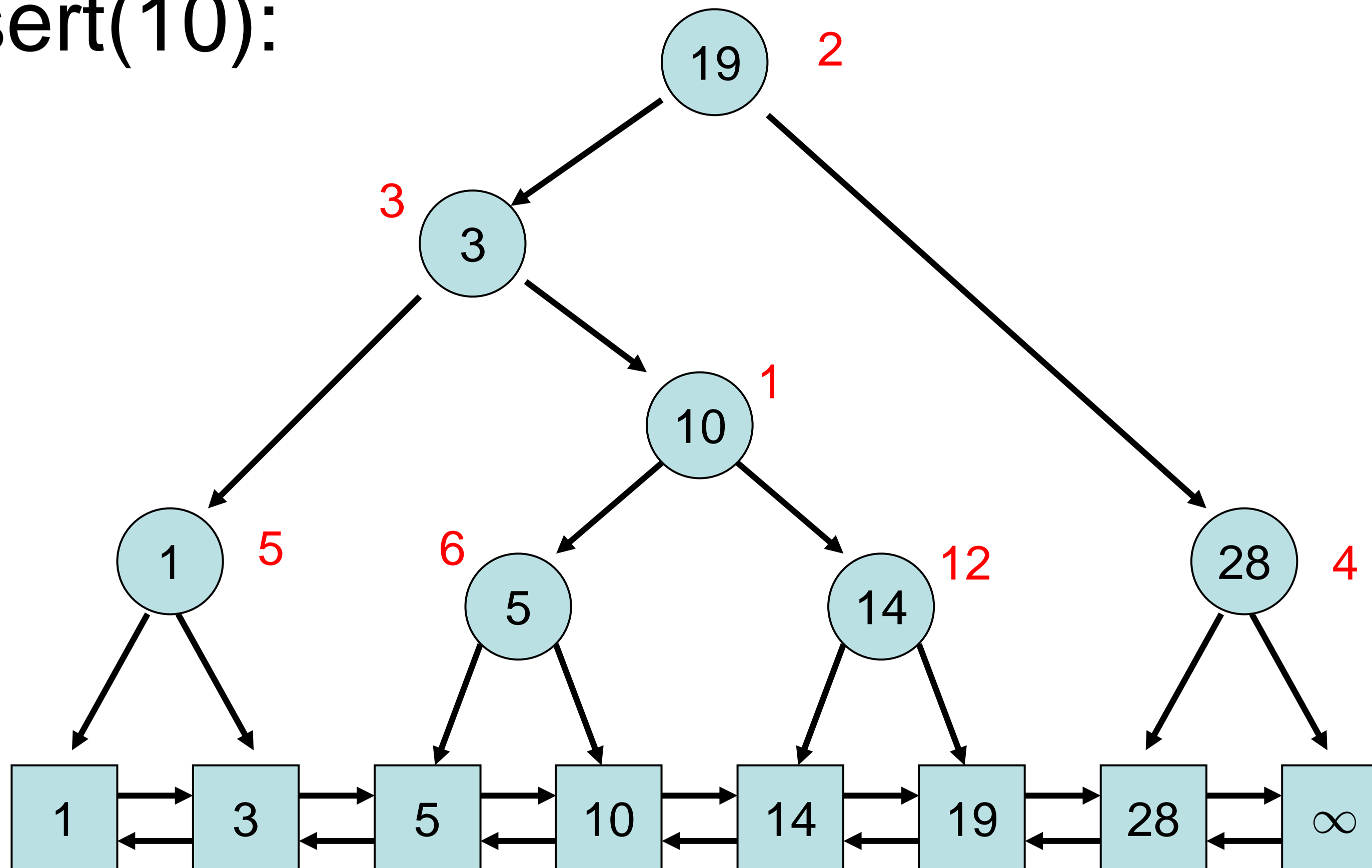
Treaps

Insert(10):



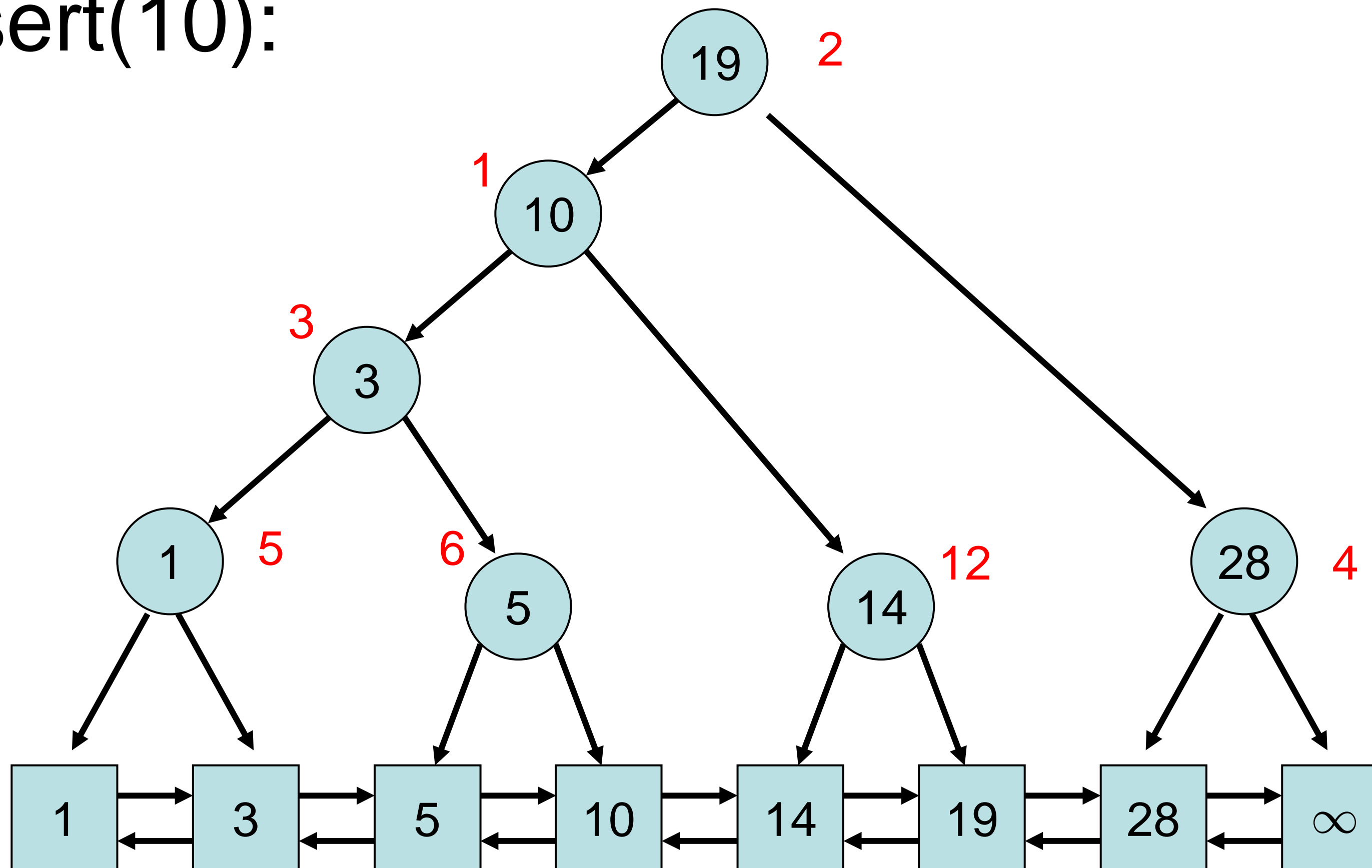
Treaps

Insert(10):



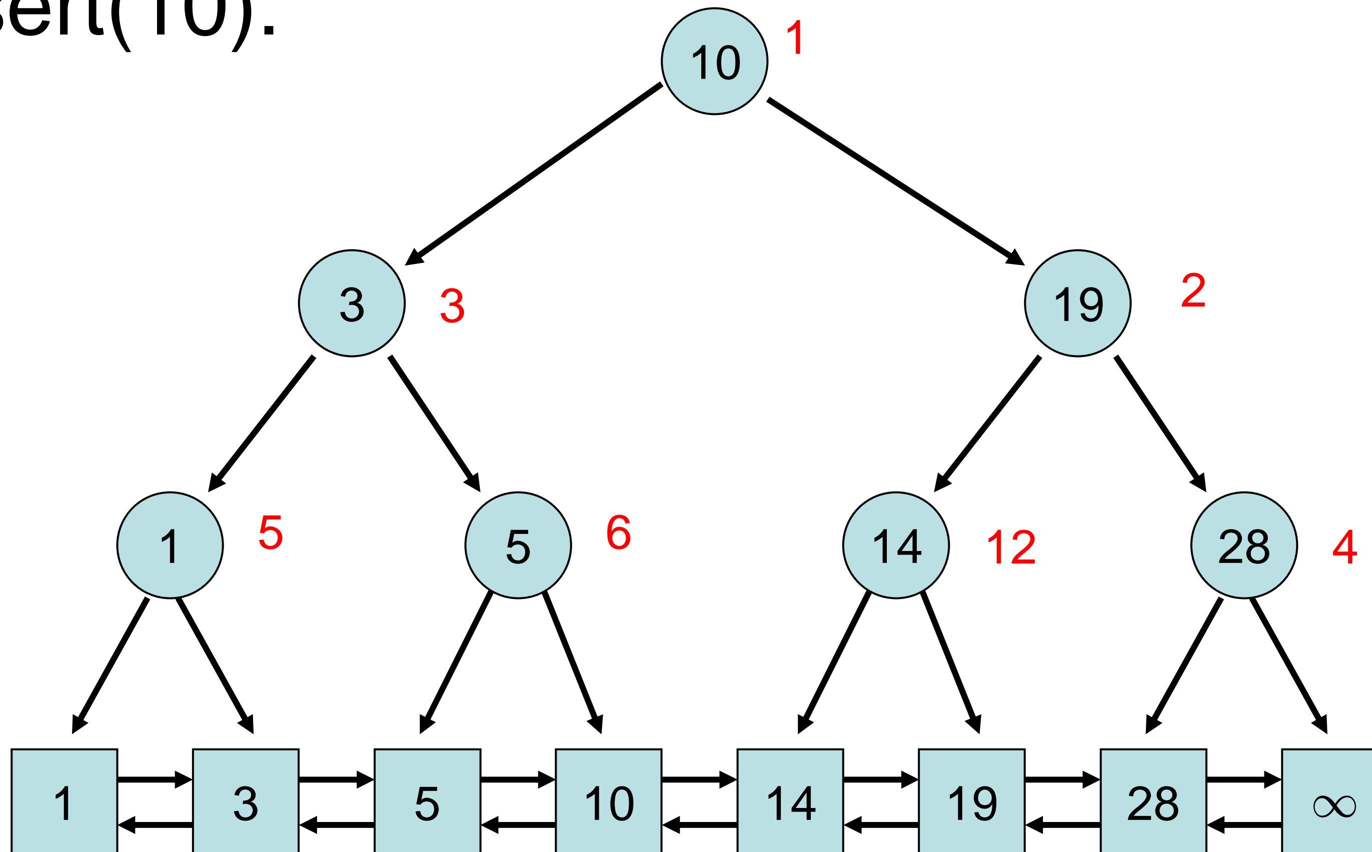
Treaps

Insert(10):



Treaps

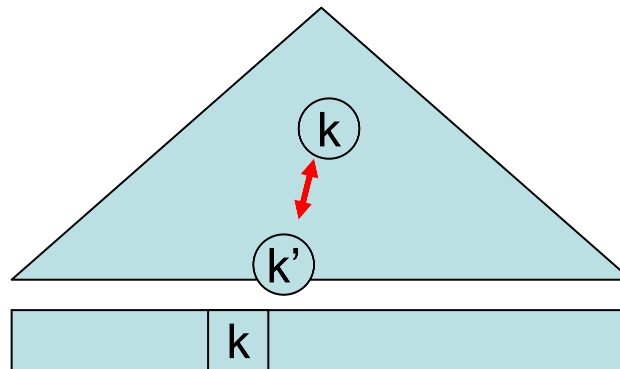
Insert(10):



Treaps

Delete(k)-Operation:

- Führe delete(k) wie im binären Suchbaum aus

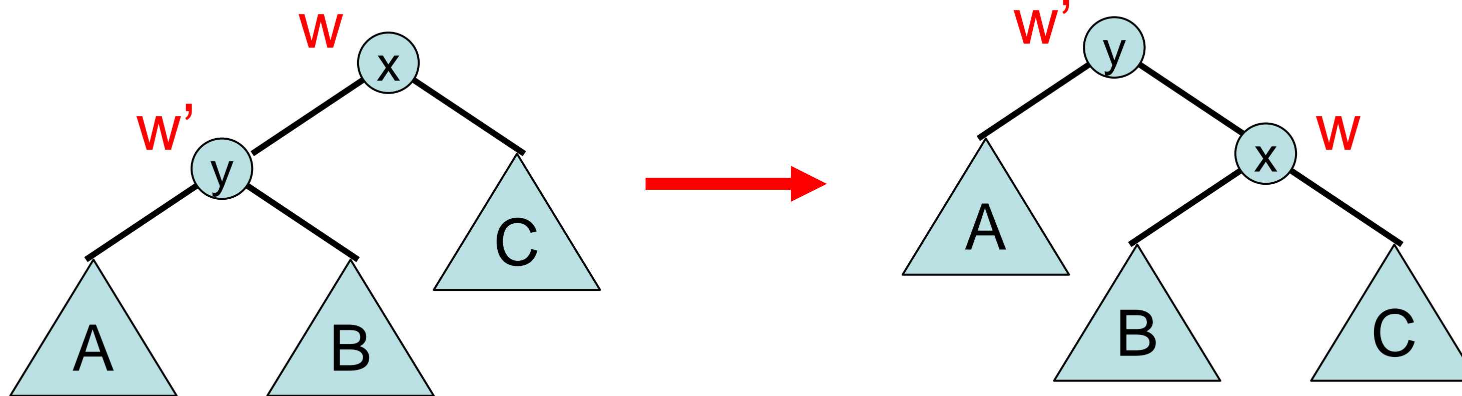


- siftdown-Operation, um Heap-Eigenschaft für k' zu reparieren (k' im Teilbaum von k!)

Treaps

Siftdown-Operation für x: wird über Rotationen durchgeführt

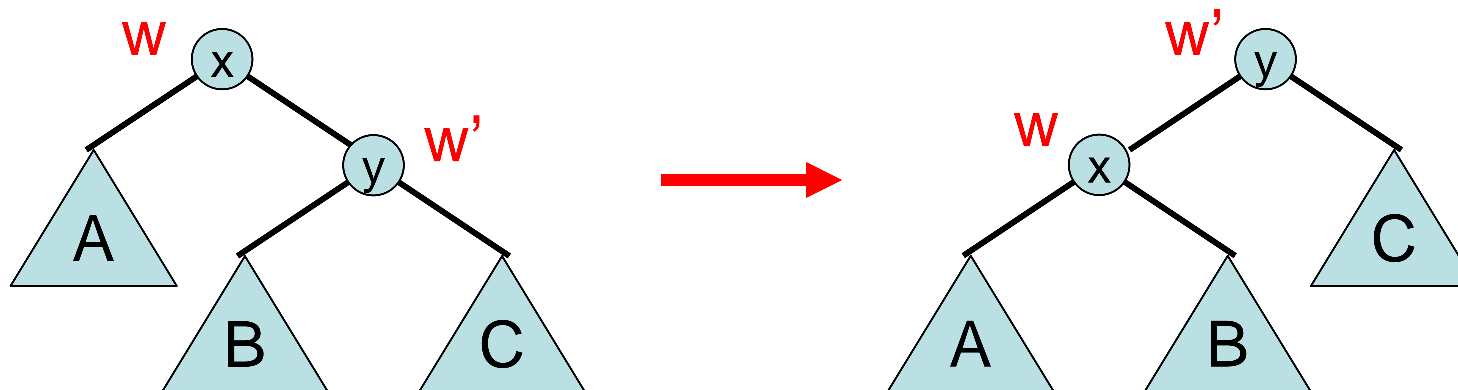
Fall 1: (w'<w , w' minimal für Kinder von x)



Treaps

Siftdown-Operation für x: wird über Rotationen durchgeführt

Fall 2: (w'<w, w' minimal für Kinder von x)



Treaps

Lemma 3.7: Sei $K=\{k_1,\ldots,k_n\}$ eine geordnete Schlüsselmenge mit paarweise verschiedenen Gewichten $w(k_i)$. Dann ist im Treap T für K das Element k_j genau dann Vorgänger von k_i wenn gilt:

$$w(k_j) = \min \{w(k) \mid k \in K_{i,j}\}$$

wobei $K_{i,j} = \{k_i,\ldots,k_j\}$.

Beweis: Übung.

Treaps

Lemma 3.8: Angenommen, die Gewichte seien eine zufällige Permutation π über $\{1,\ldots,n\}$ definiert mit $w(k_i)=\pi(i)$ für alle i. Dann gilt

$$\Pr[k_j \text{ ist Vorgänger von } k_i] = \frac{1}{|j - i| + 1}$$

Beweis:

- Aus Symmetriegründen ist für jedes $k \in \{i,\ldots,j\}$ $\Pr[\pi(k) \text{ ist minimal in } \{\pi(i),\ldots,\pi(j)\}]$ gleich
- Also ist nach Lemma 3.7

$$\Pr[k_j \text{ ist Vorgänger von } k_i] = 1/(|j-i|+1)$$

Treaps

- Sei $X_{i,j} \in \{0,1\}$ eine binäre Zufallsvariable, die 1 ist genau dann, wenn k_j Vorgänger von k_i ist.
- Sei L_i die Tiefe von Schlüssel k_i im Treap T (Tiefe der Wurzel: 0)
- Dann ist $L_i = \sum_j X_{i,j}$

Aus Lemma 3.7 folgt:

$$E[L_i] = \sum_j E[X_{i,j}]$$
$$= \sum_j \Pr[k_j \text{ ist Vorgänger von } k_i \text{ in } T]$$
$$= \sum_j 1/(|j-i|+1) < 2 \ln n$$

Treaps

Aus der Rechnung folgt:

Theorem 3.9: Wenn die Schlüssel uniform zufällig Gewichte aus einem genügend großen Bereich W wählen, so dass die Wkeit gleicher Gewichte vernachlässigbar ist, dann ist für jeden Schlüssel k im Treap T die erwartete Tiefe $O(\log n)$.

Diese Aussage gilt auch mit hoher Wahrscheinlichkeit (mindestens $1-1/n^c$ für jede Konstante $c>1$), d.h. Treap T ist m.h.W. balanciert.

Treaps

Laufzeiten der Operationen:

- search(k): $O(\log n)$ m.h.W.
- insert(e): $O(\log n)$ m.h.W.
- delete(k): $O(\log n)$ m.h.W.

(m.h.W.: mit hoher Wahrscheinlichkeit)

Binärbaum

Problem: Binärbaum kann entarten!

Lösungen:

- Splay-Baum
  (sehr effektive Heuristik)
- Treaps
  (mit hoher Wkeit gut balanciert)
- (a,b)-Baum
  (garantiert gut balanciert)
- Rot-Schwarz-Baum
  (konstanter Reorganisationsaufwand)
- Gewichtsbalancierter Baum
  (kompakt einbettbar in Feld)

(a,b)-Bäume

Problem: Binärbaum kann entarten!

Lösung: (a,b)-Baum

Idee:

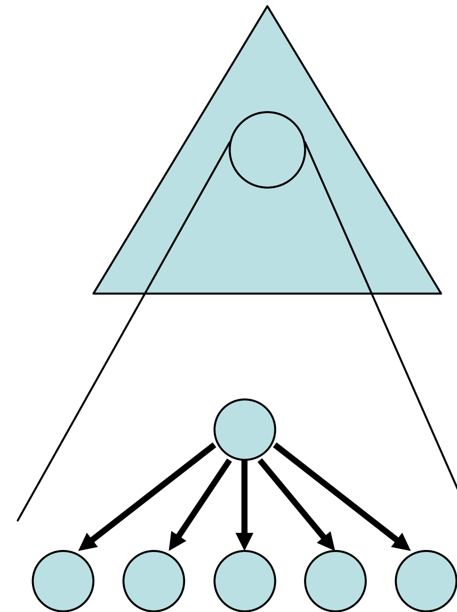
- Alle Knoten v (außer Wurzel) haben Grad d(v) mit a<=d(v)<=b, wobei a>=2 und b>=2a-1 ist
- Alle Blätter in derselben Ebene

(a,b)-Bäume

Formell: für einen Baumknoten v sei

- $d(v)$ die Anzahl der Kinder von v
- $t(v)$ die Tiefe von v (Wurzel hat Tiefe 0)

- Form-Regel:
  Für alle Blätter v,w: $t(v)=t(w)$
- Grad-Regel:
  Für alle inneren Knoten v außer Wurzel: $d(v) \in [a,b]$,
  für Wurzel r: $d(r) \in [2,b]$
  (sofern #Elemente >1)



(a,b)-Bäume

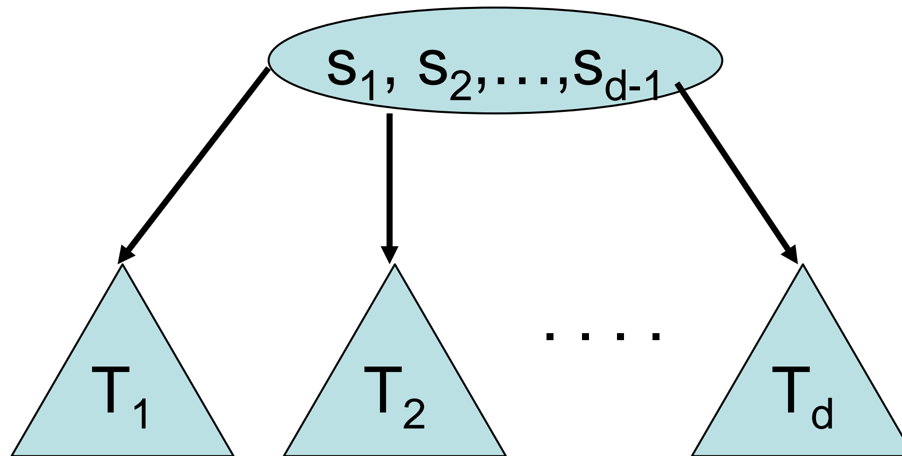
Lemma 3.10: Ein (a,b)-Baum für n Elemente hat Tiefe max. $1+\lfloor \log_a (n+1)/2 \rfloor$

Beweis:

- Die Wurzel hat Grad $>=2$ und jeder andere innere Knoten hat Grad $>=a$.
- Bei Tiefe t gibt es mindestens $2a^{t-1}$ Blätter
- $n+1>=2a^{t-1} \Leftrightarrow t<= 1+\lfloor \log_a (n+1)/2 \rfloor$

(a,b)-Bäume

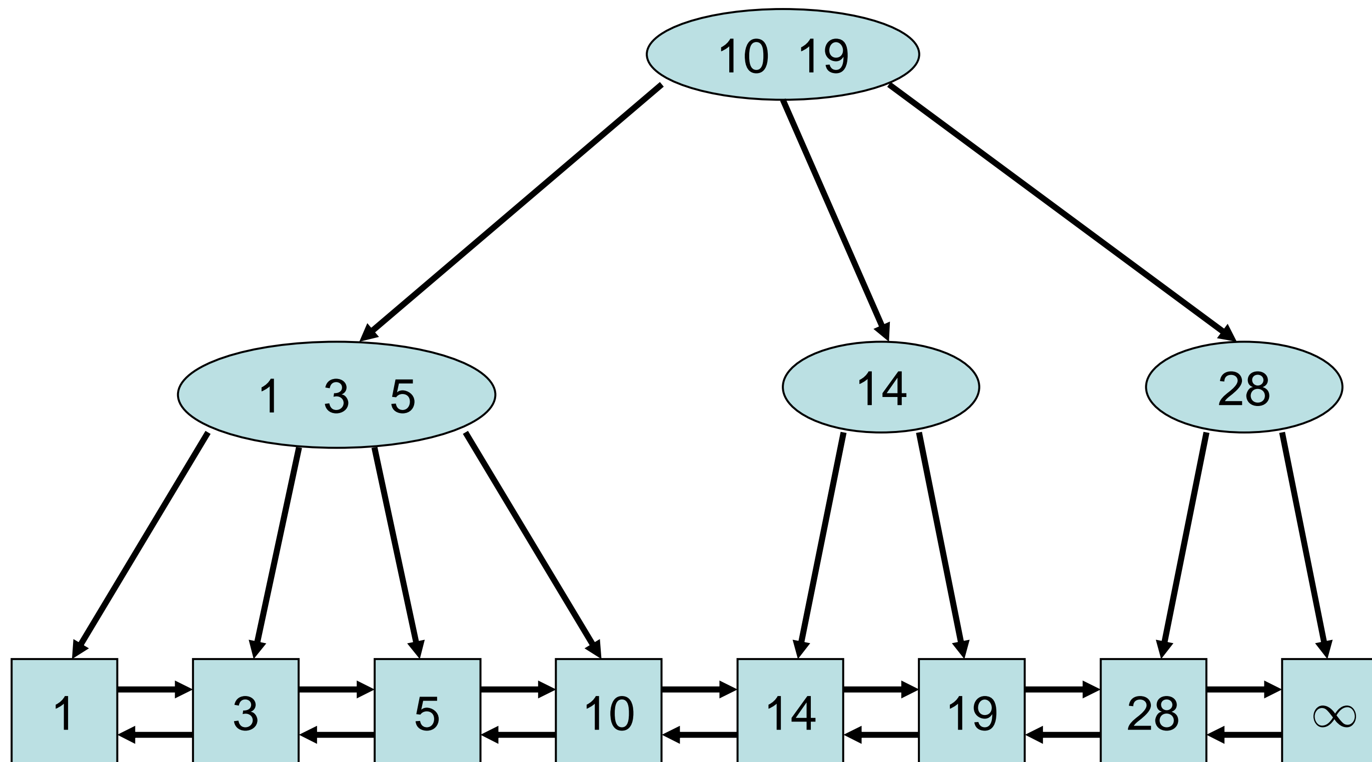
(a,b)-Suchbaum-Regel:



Für alle Schlüssel k in T_i und k' in T_{i+1}: $k <= s_i < k'$

Damit search Operation einfach zu implementieren.

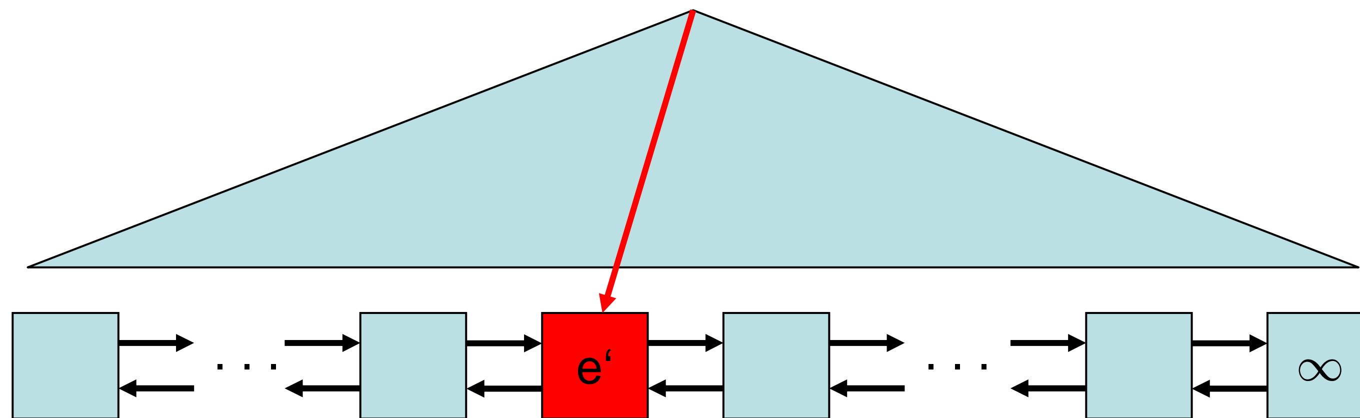
Search(9)



Insert(e) Operation

Strategie:

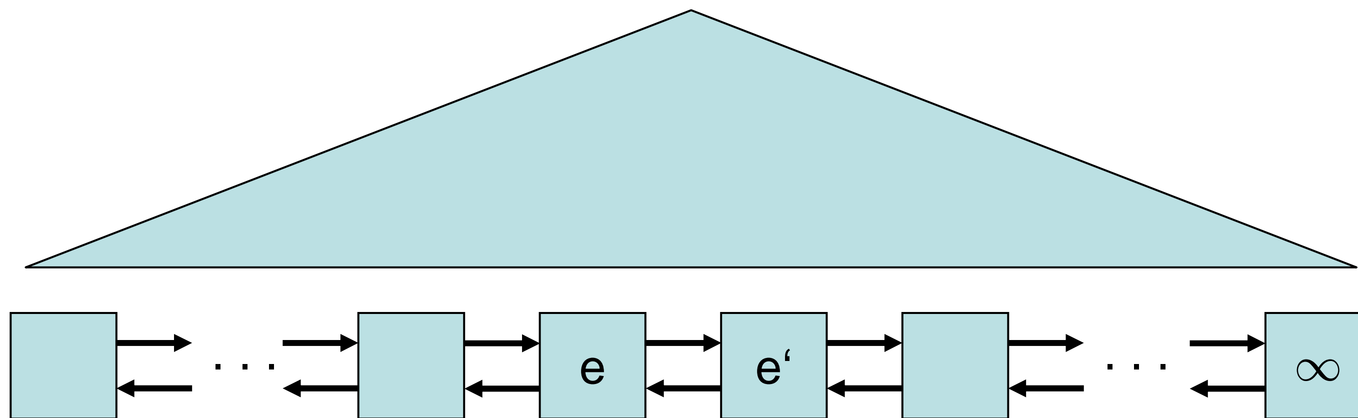
- Erst search(key(e)) bis Element e´ in Liste erreicht. Falls key(e´)>key(e), füge e vor e´ ein, ansonsten stop.



Insert(e) Operation

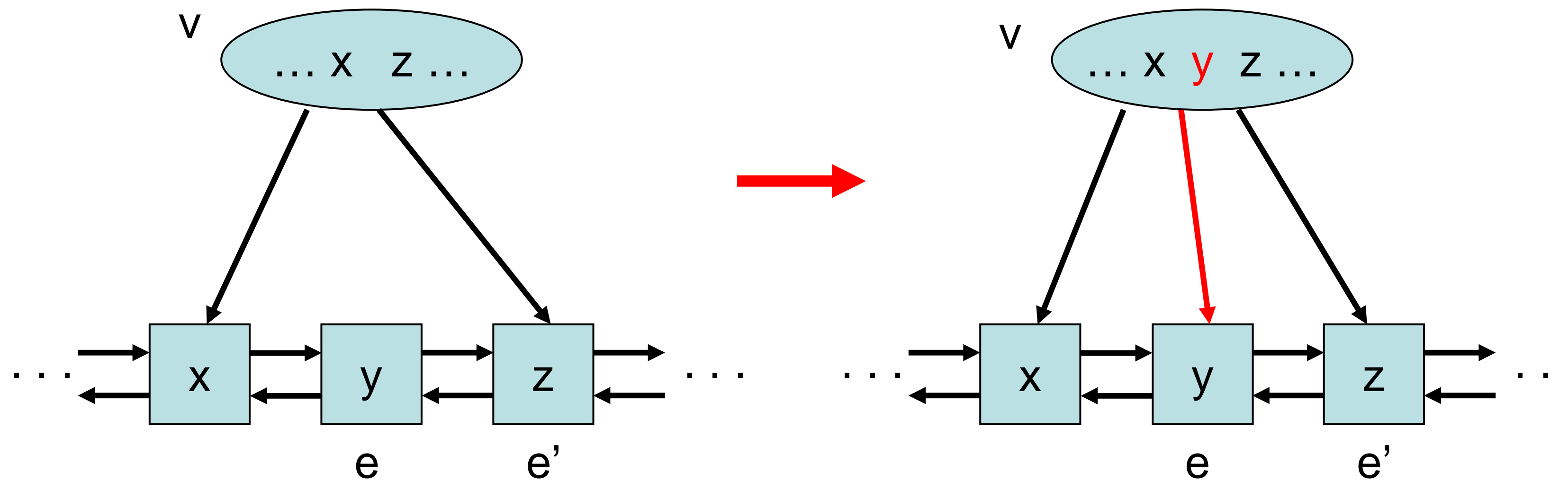
Strategie:

- Erst search(key(e)) bis Element e´ in Liste erreicht. Falls key(e´)>key(e), füge e vor e´ ein, ansonsten stop.



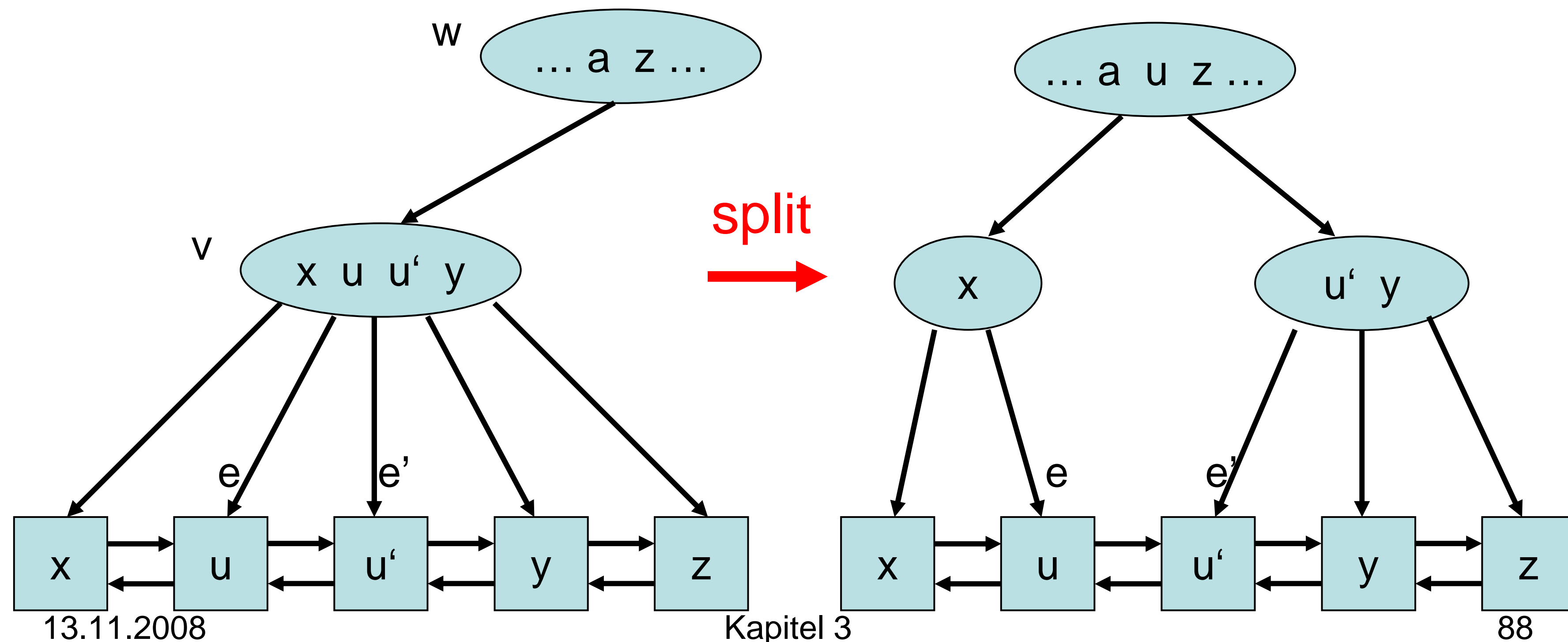
Insert(e) Operation

- Füge key(e) und Zeiger auf e in Baumknoten v über e´ ein. Falls Grad von v kleiner gleich b, dann fertig.



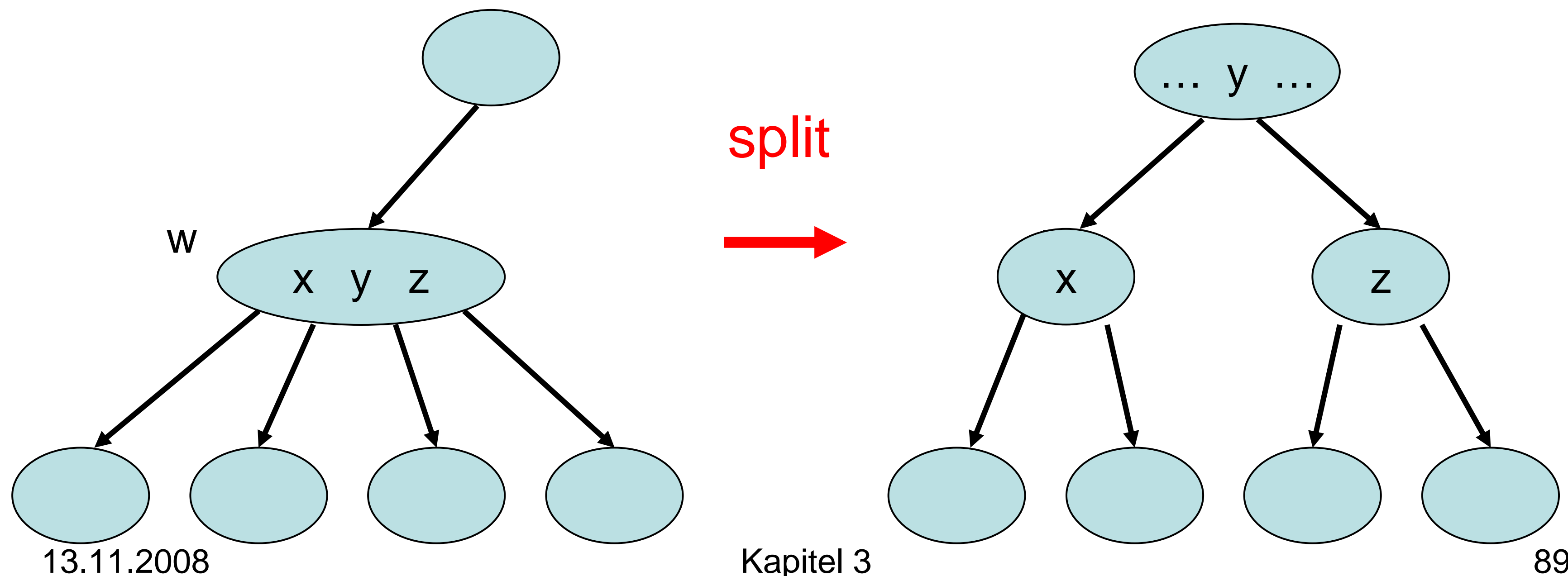
Insert(e) Operation

- Falls Grad von v größer als b, dann teile v in zwei Knoten auf. (Beispiel: a=2, b=4)

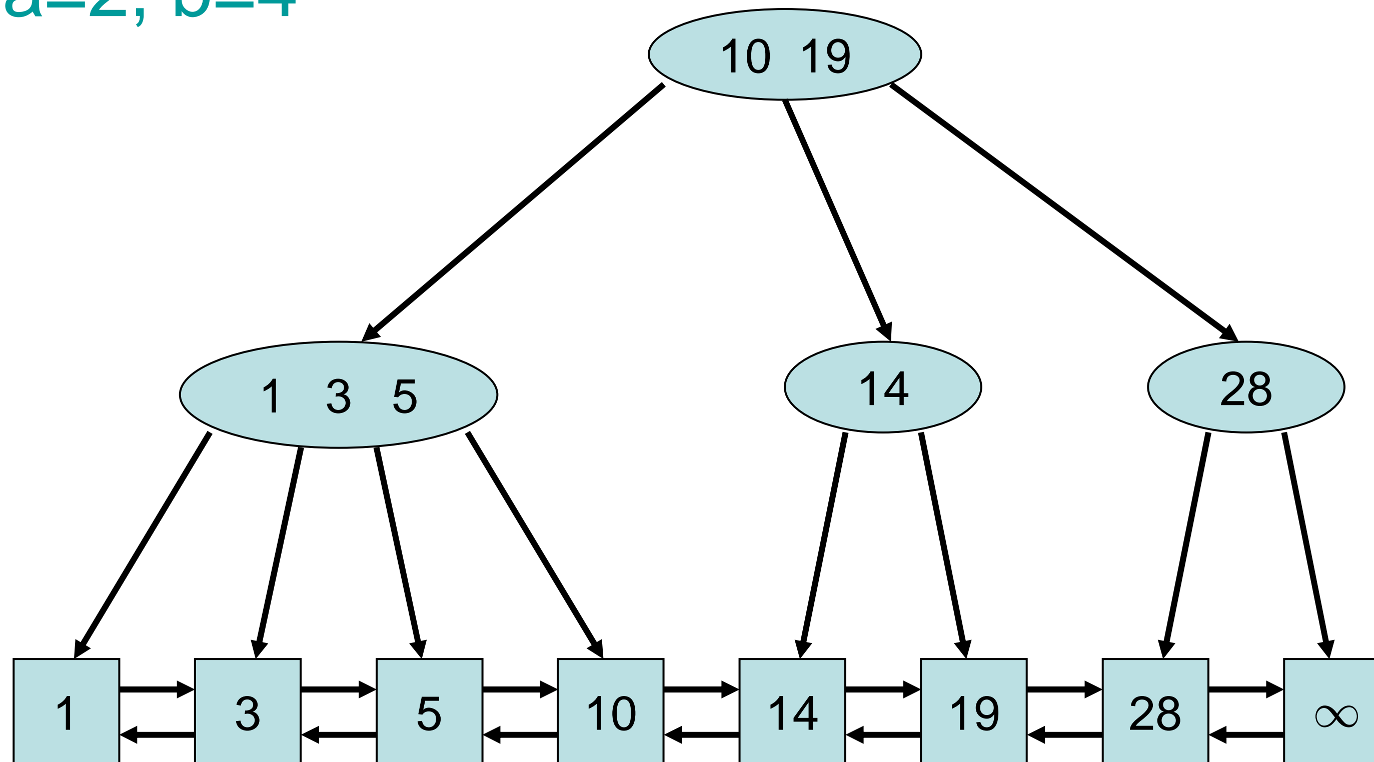


Insert(e) Operation

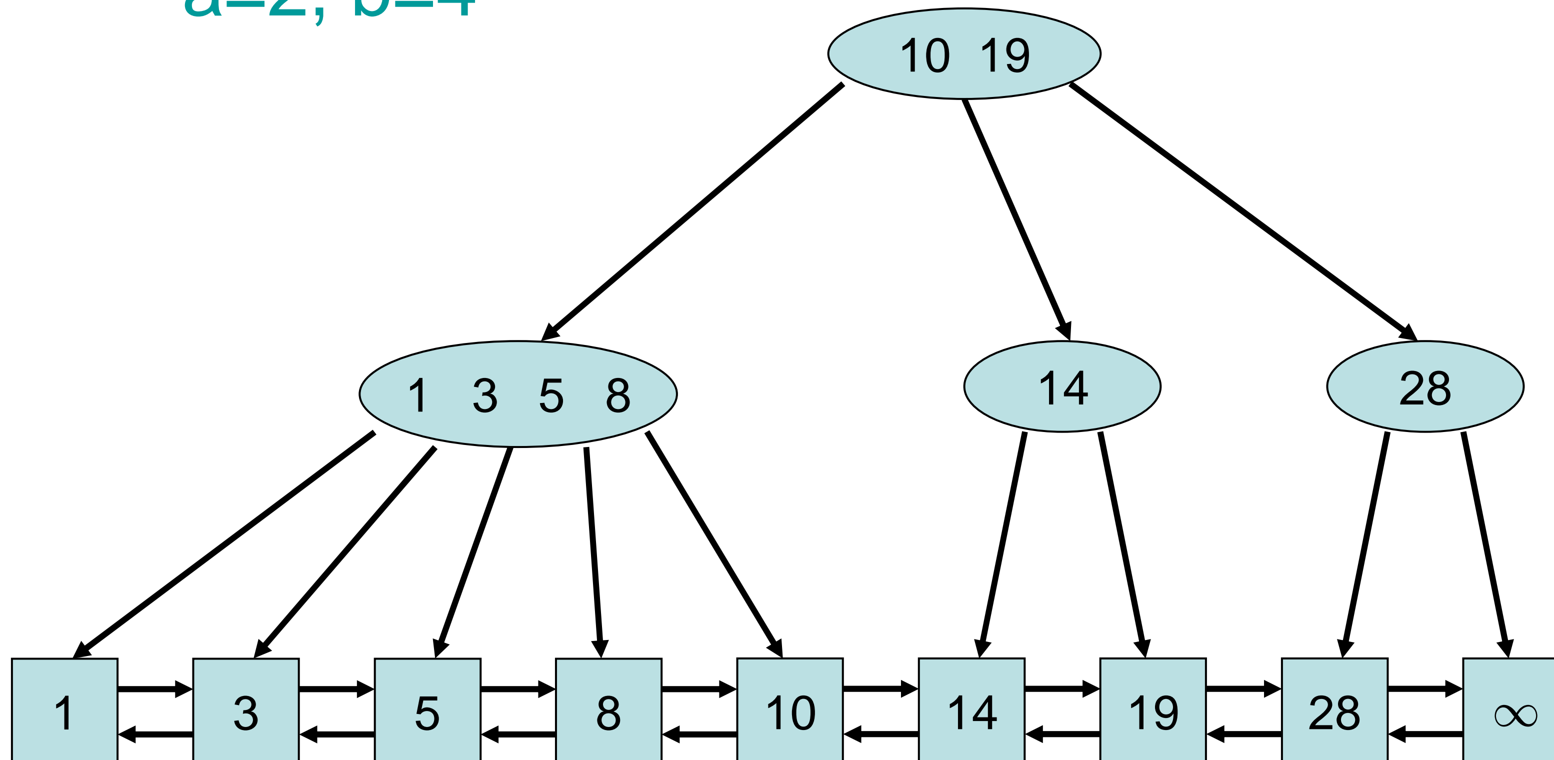
- Falls Grad von w größer als b, dann teile w in zwei Knoten auf (usw, bis Grad <=b oder Wurzel aufgeteilt wurde)



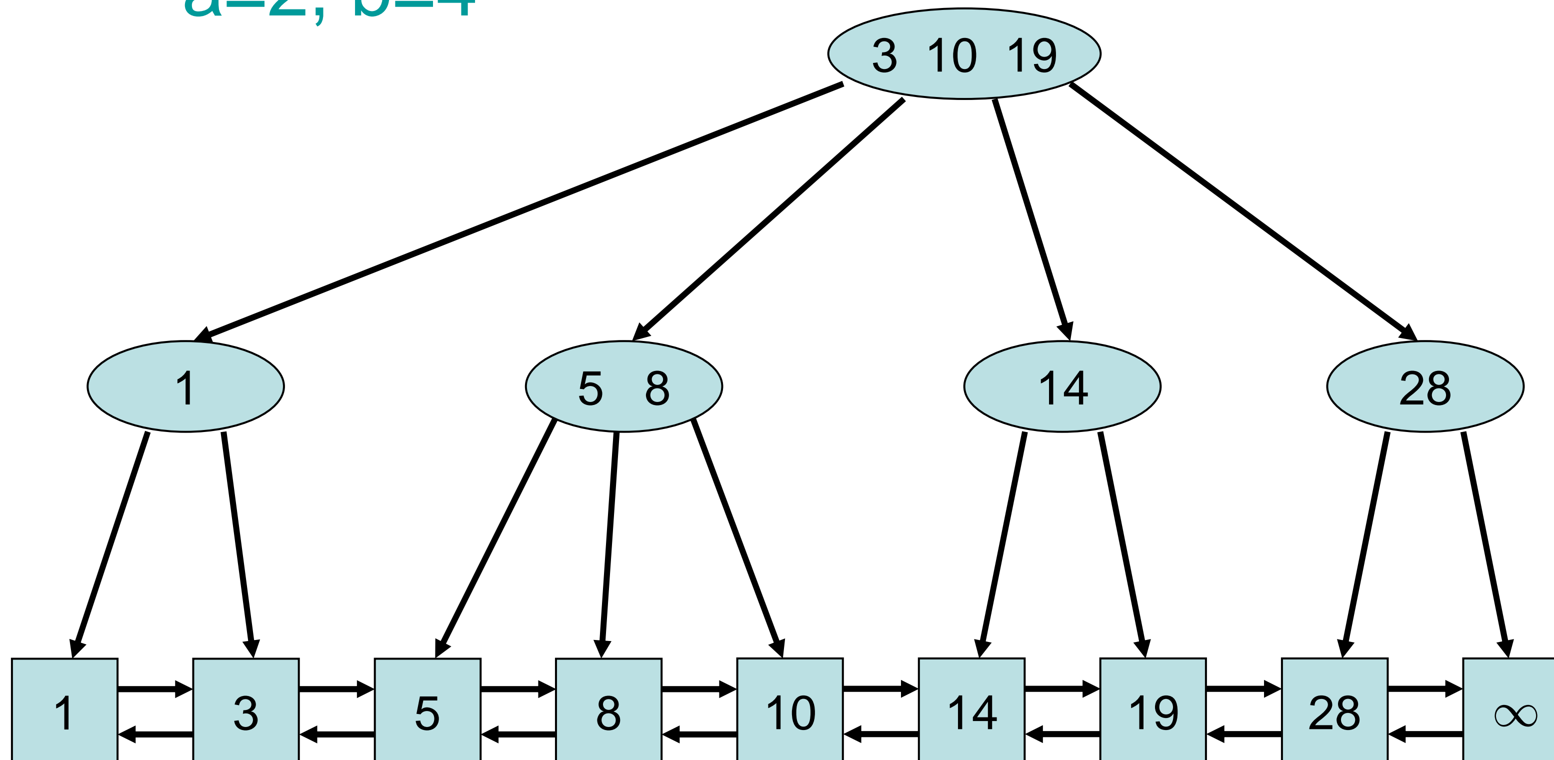
Insert(8)



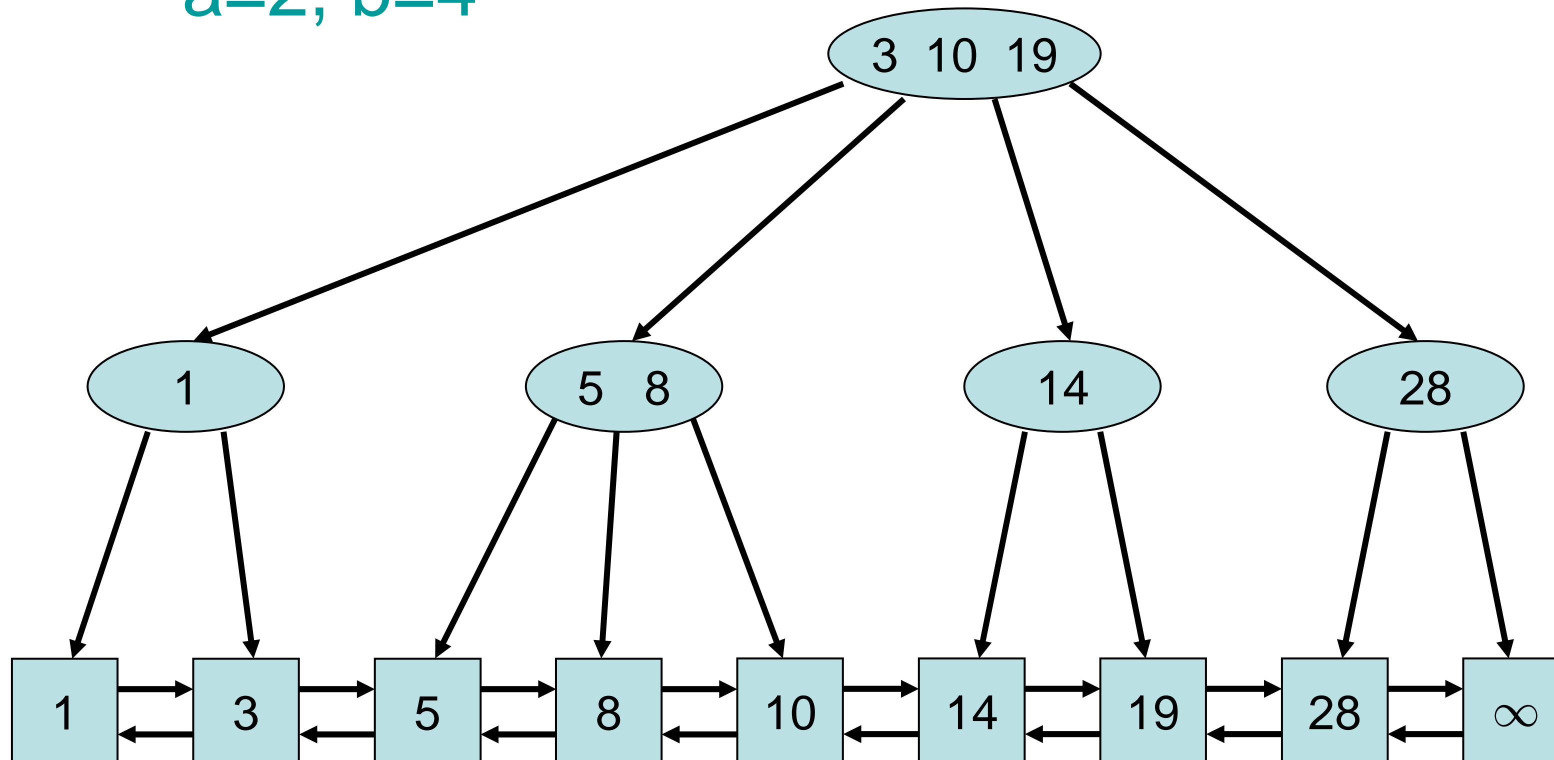
Insert(8)



Insert(8)

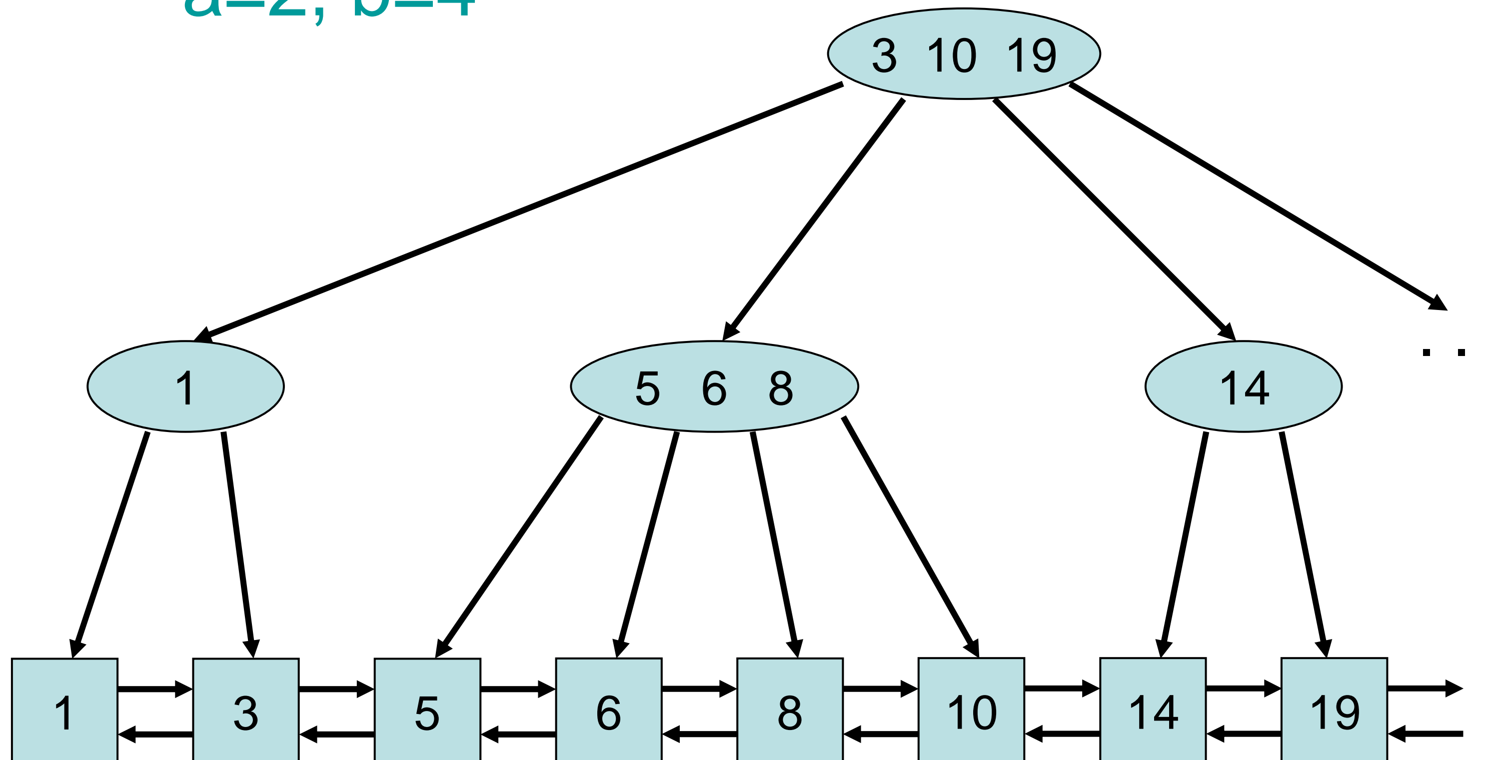


Insert(6)

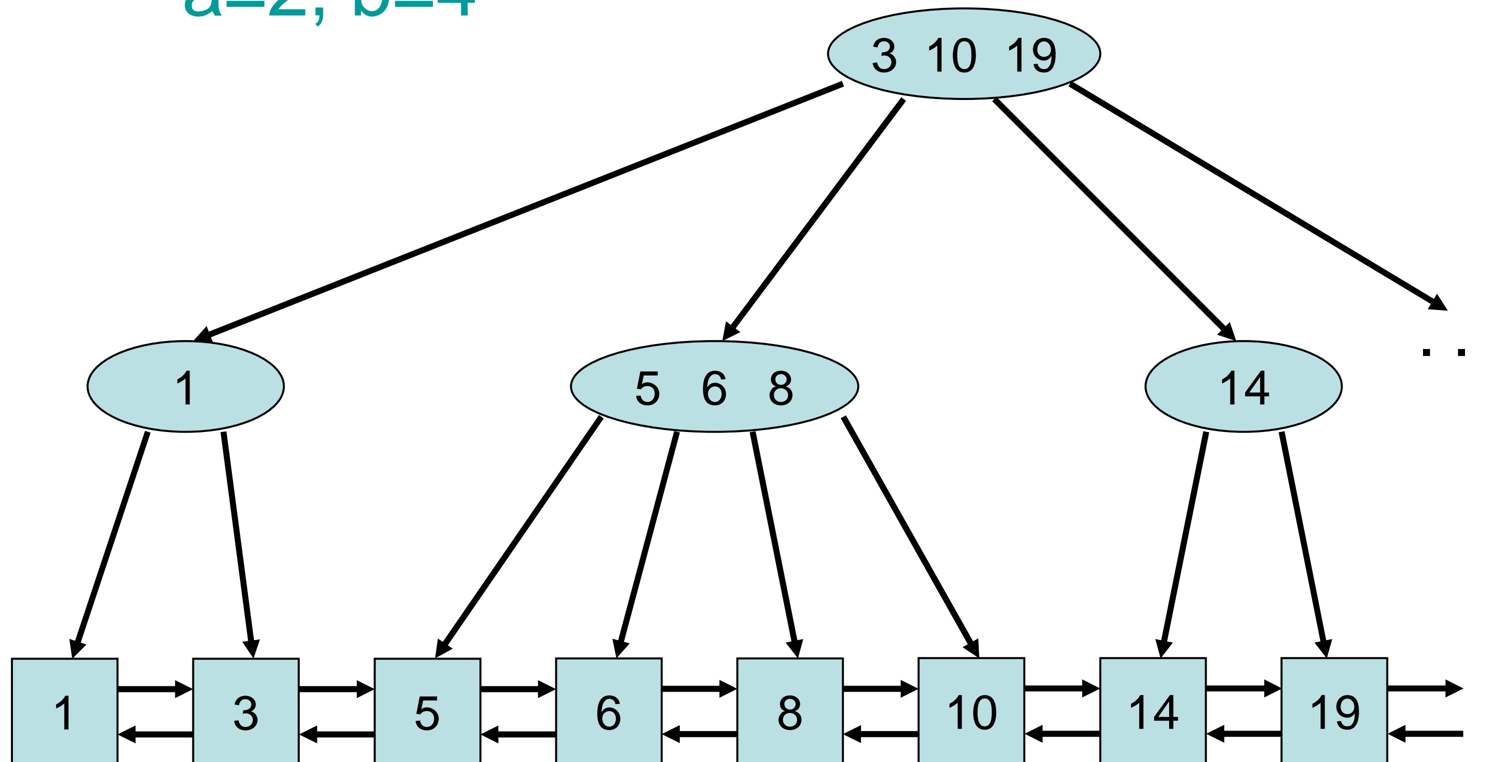


Insert(6)

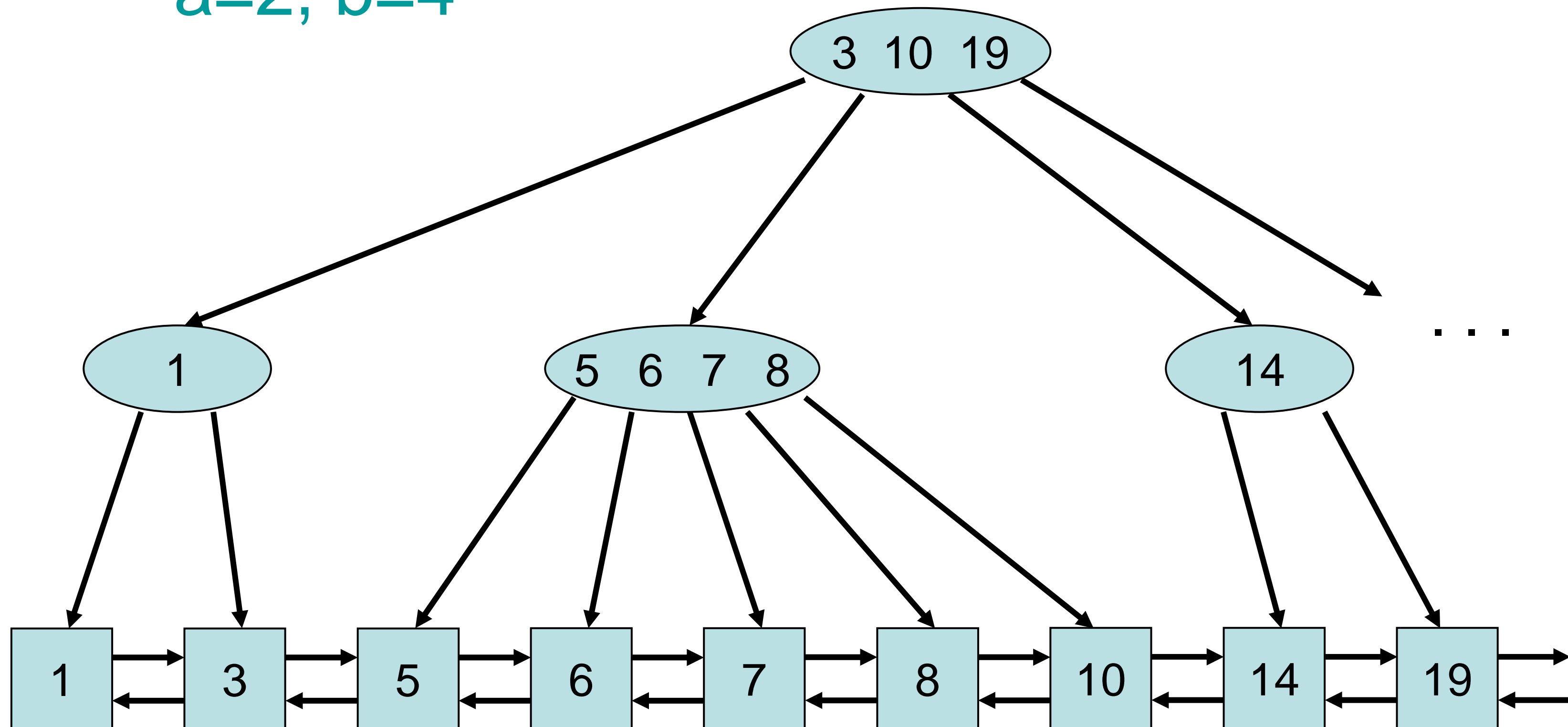
a=2, b=4



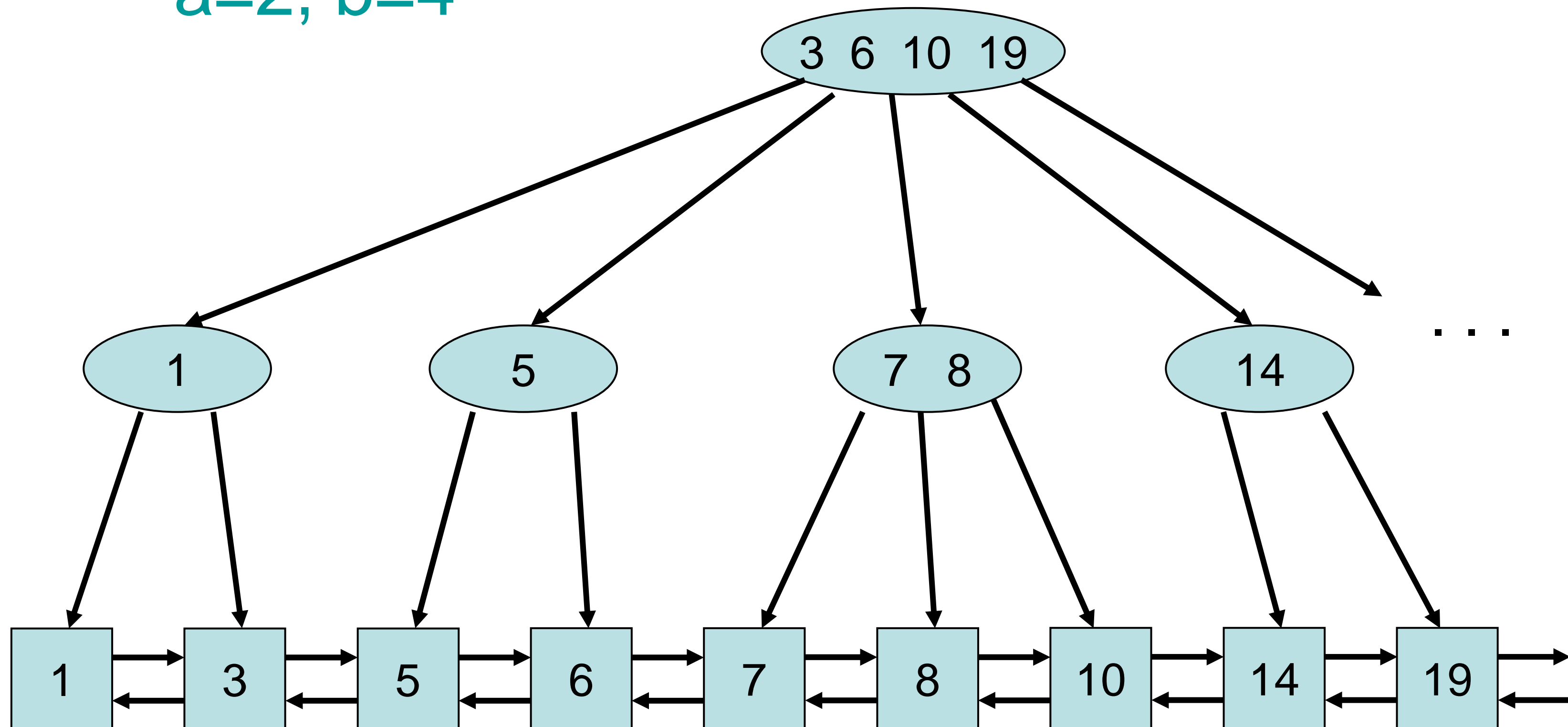
Insert(7)



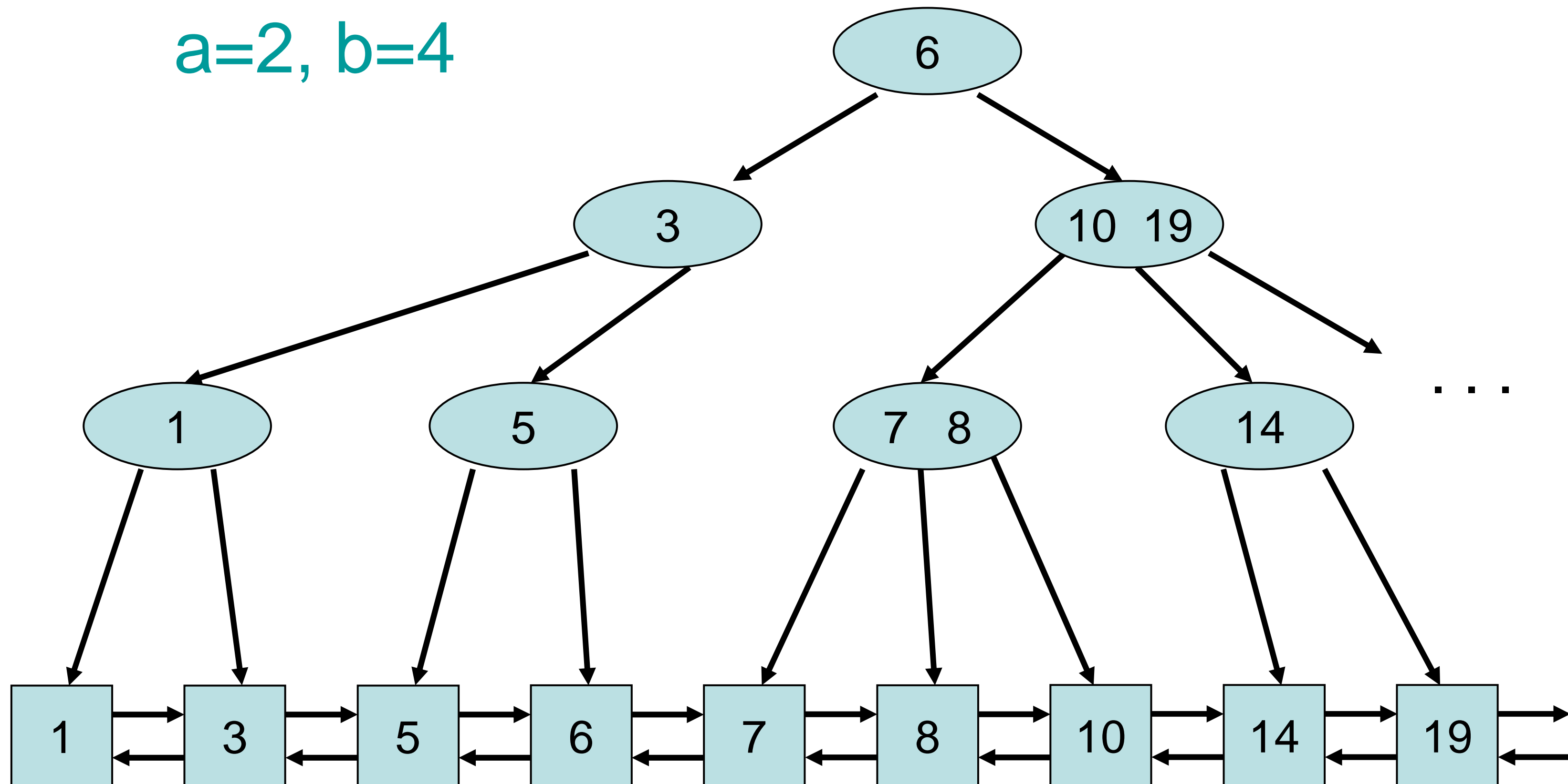
Insert(7)



Insert(7)



Insert(7)



Insert Operation

- Form-Regel:
  Für alle Blätter v,w: $t(v)=t(w)$
  Erfüllt durch Insert!

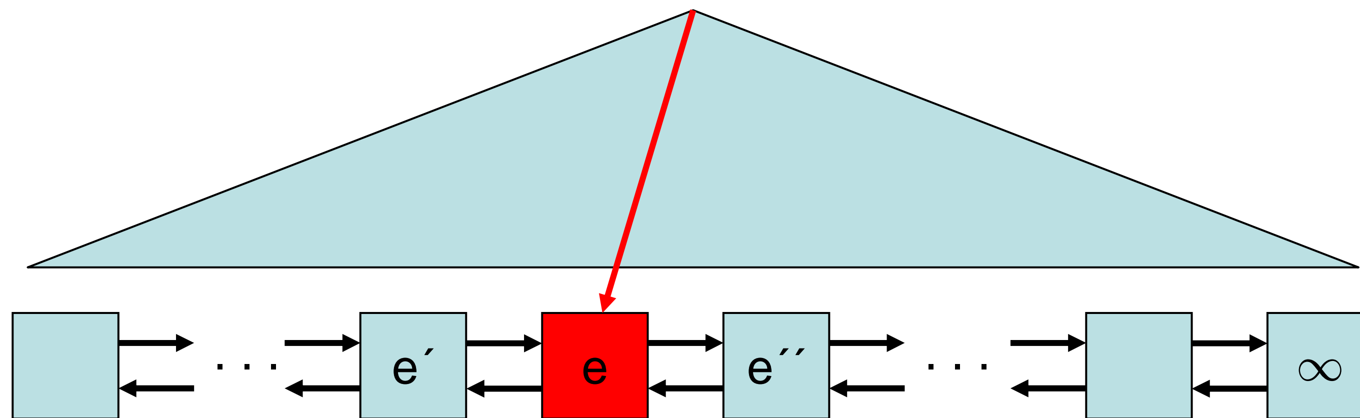
- Grad-Regel:
  Für alle inneren Knoten v außer Wurzel: $d(v) \in [a,b]$, für Wurzel r: $d(r) \in [2,b]$

  1) Insert splittet Knoten mit Grad $b+1$ in Knoten mit Grad $\lceil (b+1)/2 \rceil$ und $\lfloor (b+1)/2 \rfloor$. Wenn $b>=2a-1$, dann beide Werte $>=a$.
  2) Wenn Wurzel Grad $b+1$ erreicht, wird eine neue Wurzel mit Grad 2 erzeugt.

Delete(k) Operation

Strategie:

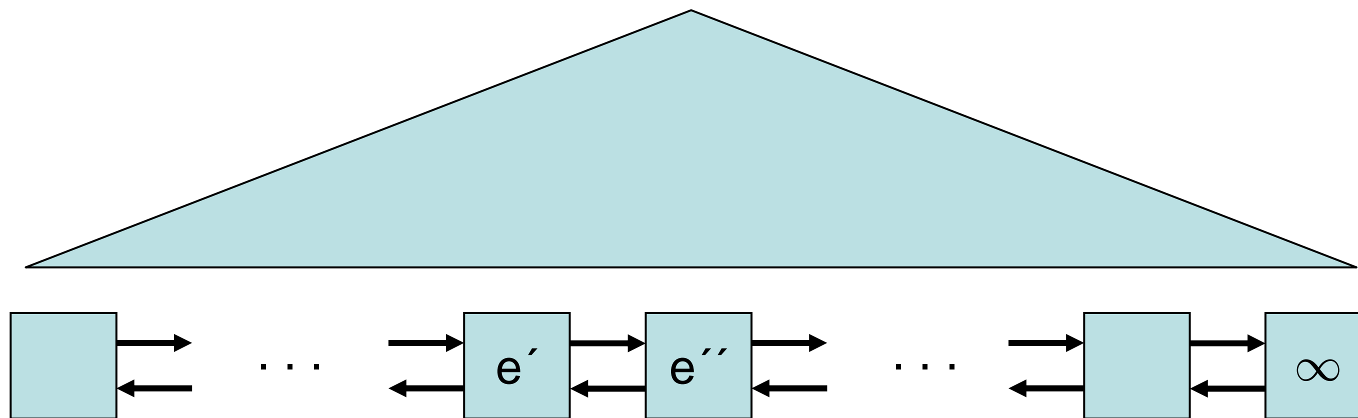
- Erst search(k) bis Element e in Liste erreicht. Falls key(e)=k, entferne e aus Liste, ansonsten stop.



Delete(k) Operation

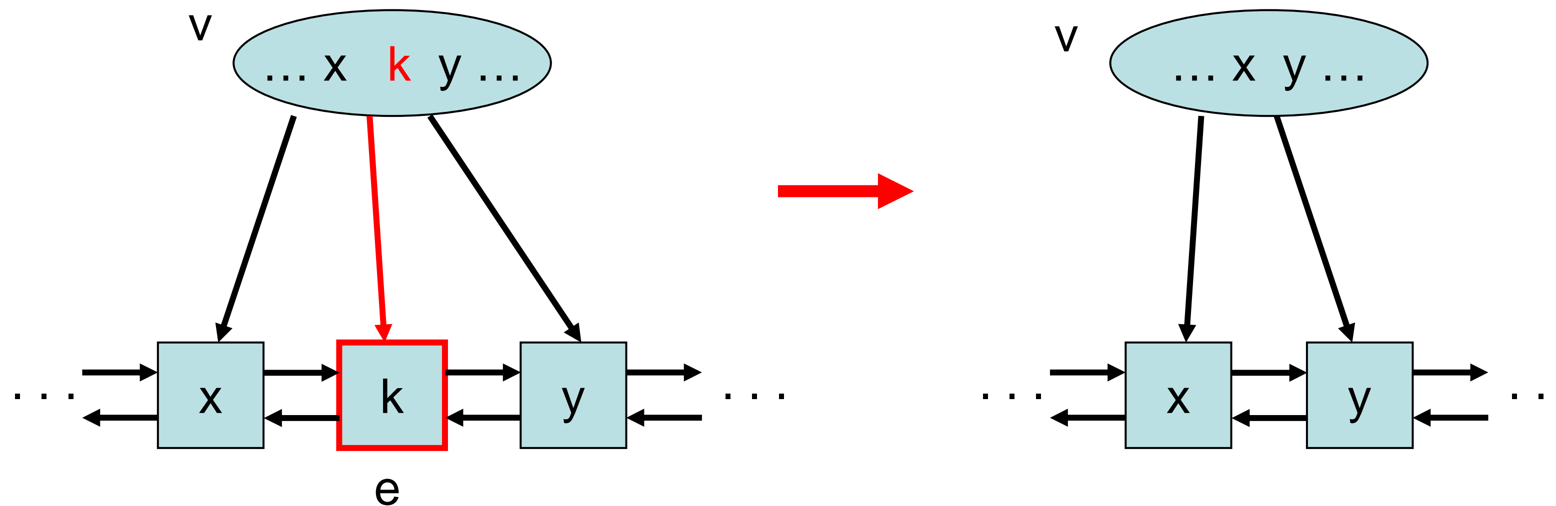
Strategie:

- Erst search(k) bis Element e in Liste erreicht. Falls key(e)=k, entferne e aus Liste, ansonsten stop.



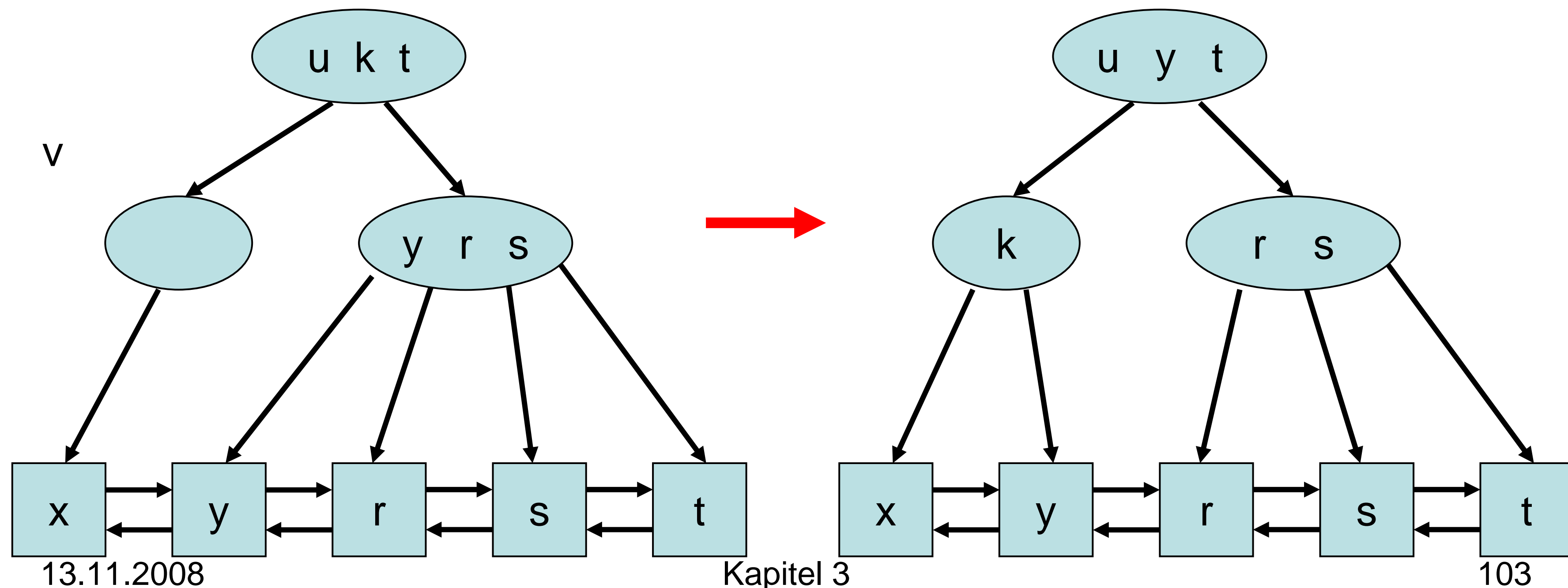
Delete(k) Operation

- Entferne Zeiger auf e und Schlüssel k vom Baumknoten v über e. (e rechtestes Kind: key Vertauschung wie bei Binärbaum!) Falls Grad von v größer gleich a, dann fertig.



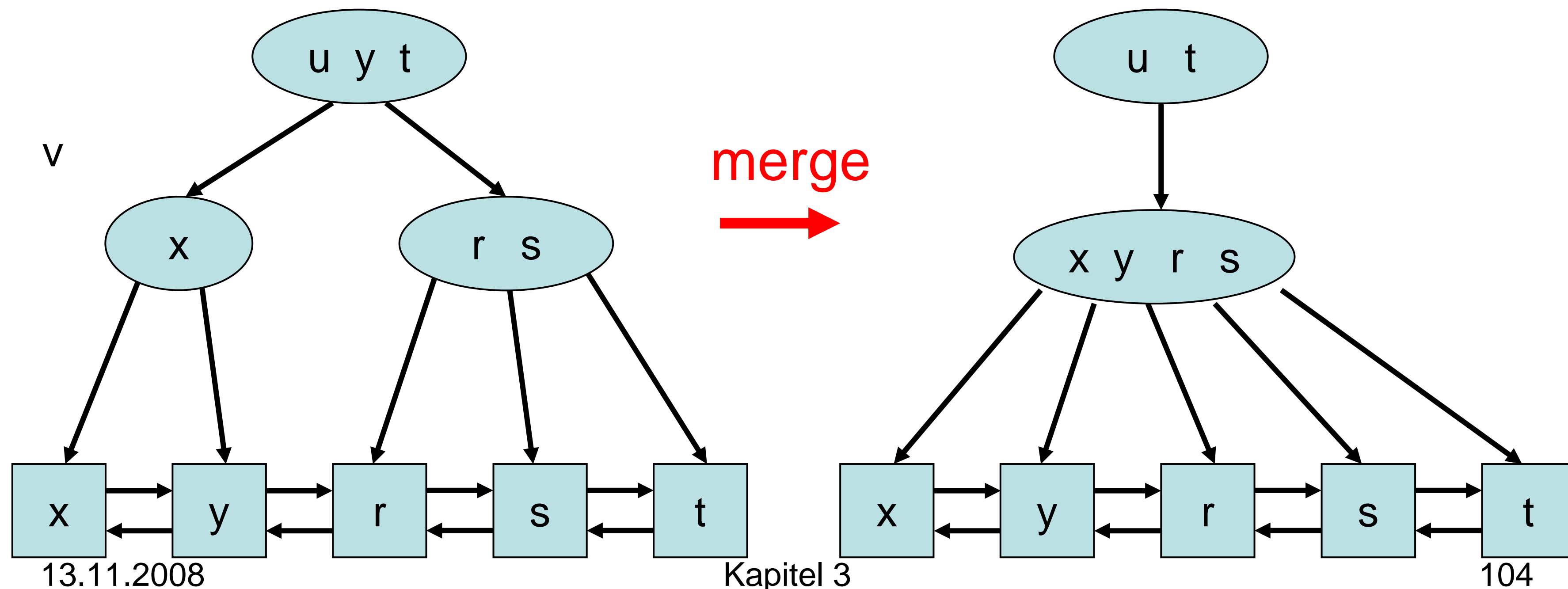
Delete(k) Operation

- Falls Grad von v kleiner als a und direkter Nachbar von v hat Grad >a, nimm Kante von diesem Nachbarn. (Beispiel: a=2, b=4)



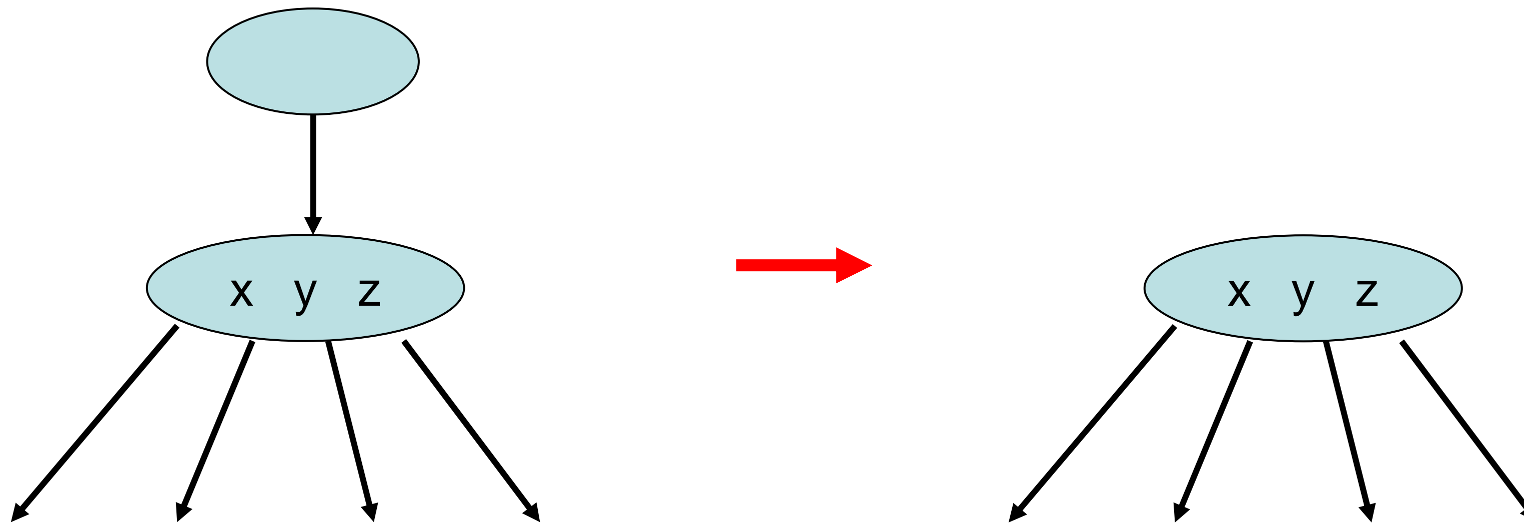
Delete(k) Operation

- Falls Grad von v kleiner als a und kein direkter Nachbar von v hat Grad >a, merge v mit Nachbarn. (Beispiel: a=3, b=5)

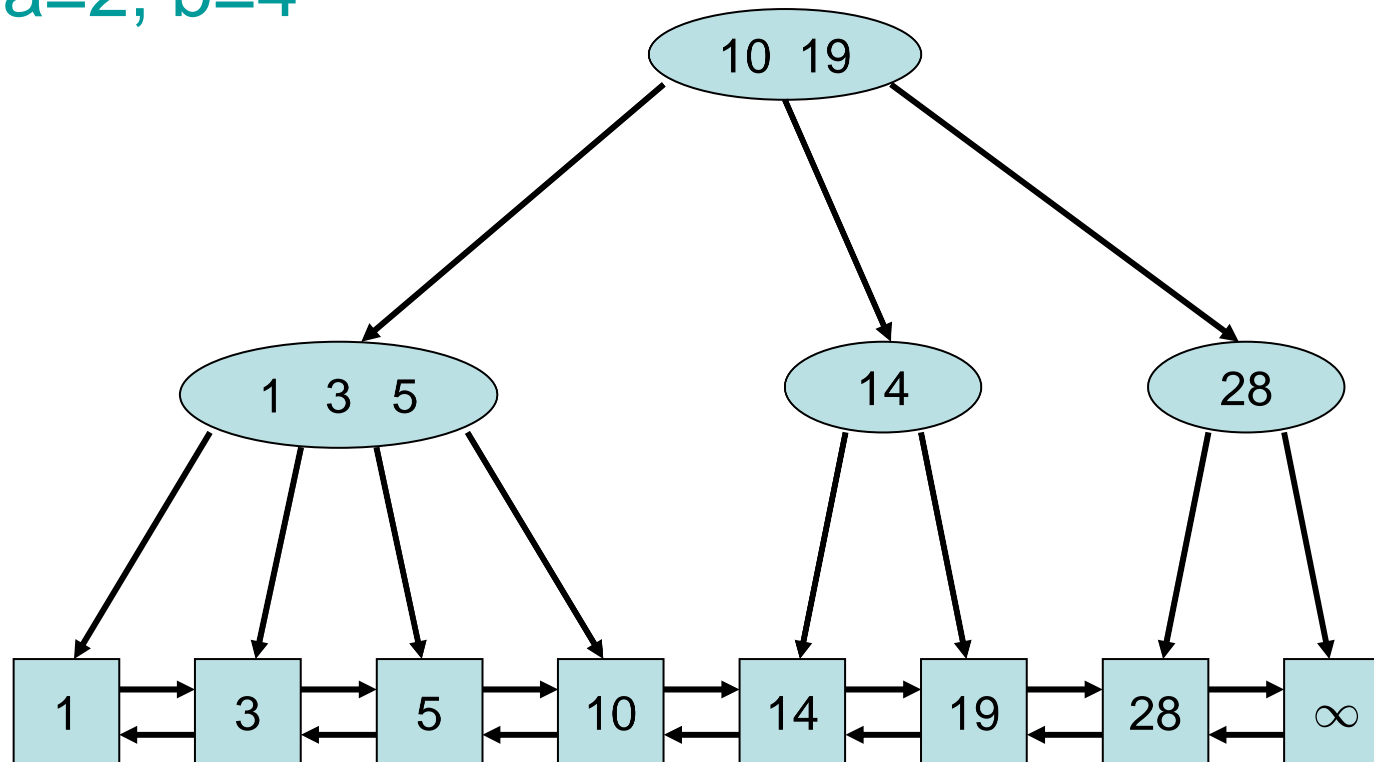


Delete(k) Operation

- Veränderungen hoch bis Wurzel, und Wurzel hat Grad <2: entferne Wurzel.

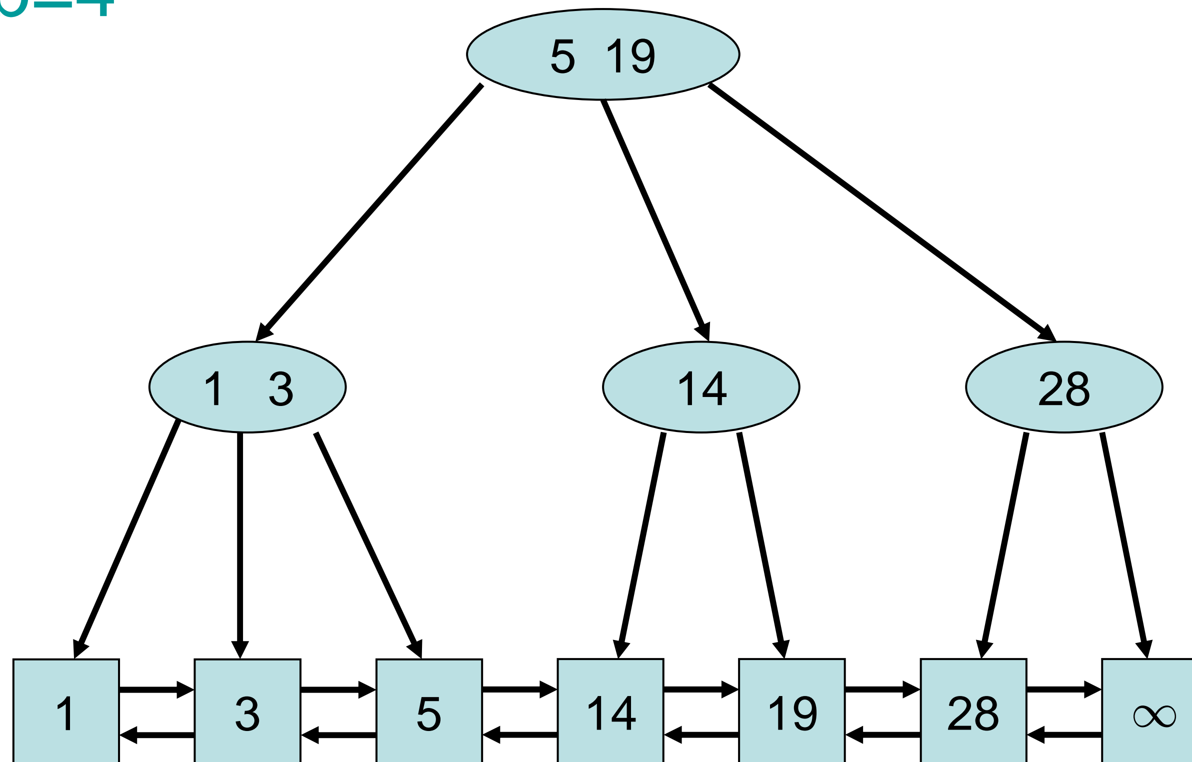


Delete(10)

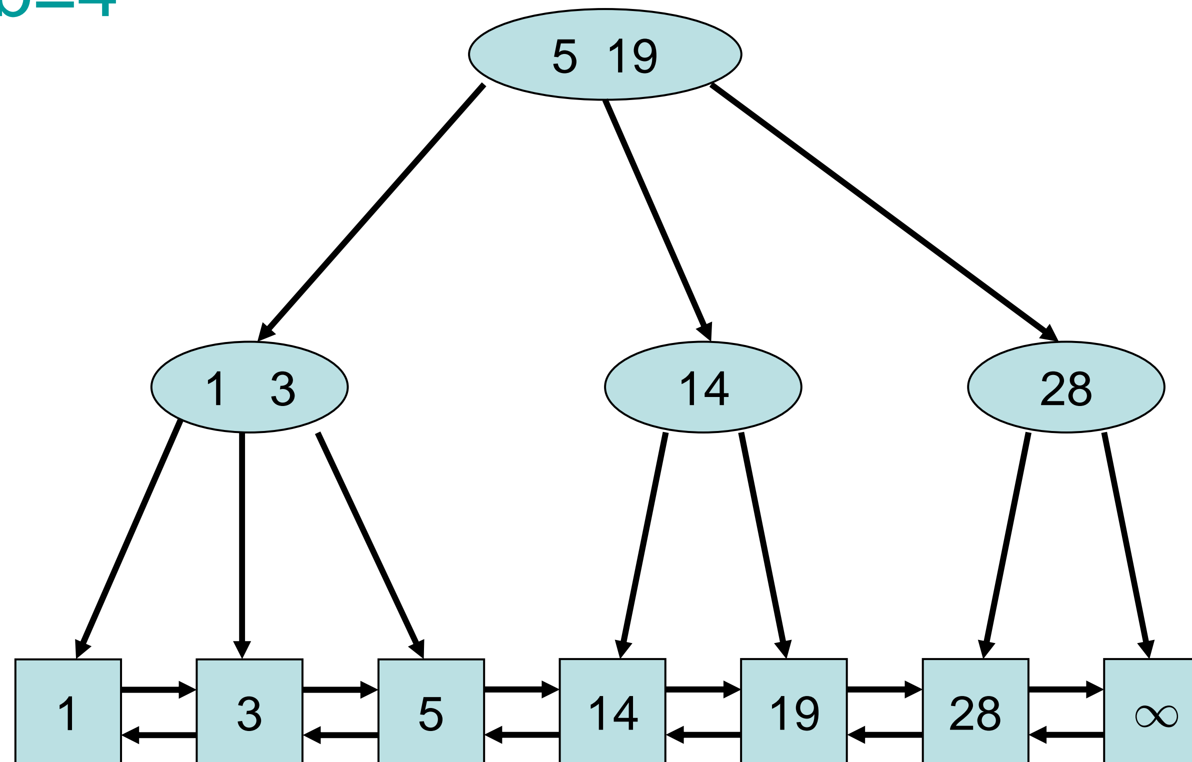


Delete(10)

a=2, b=4

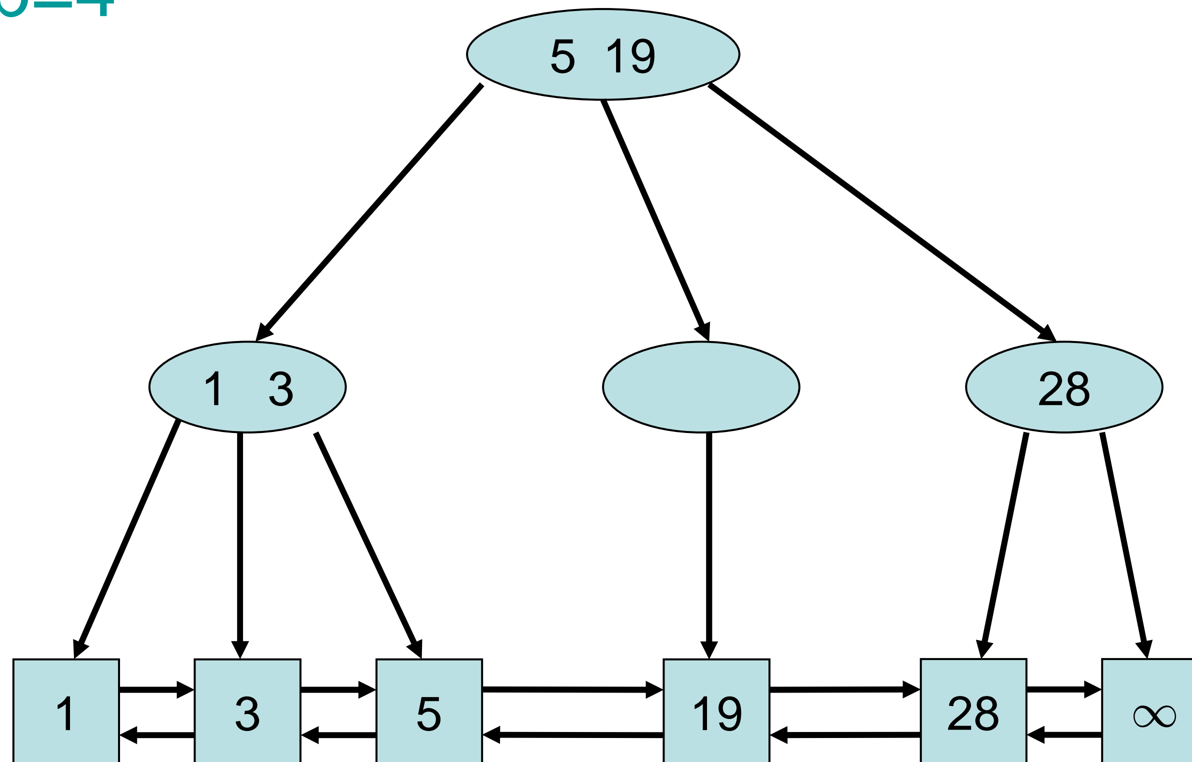


Delete(14)



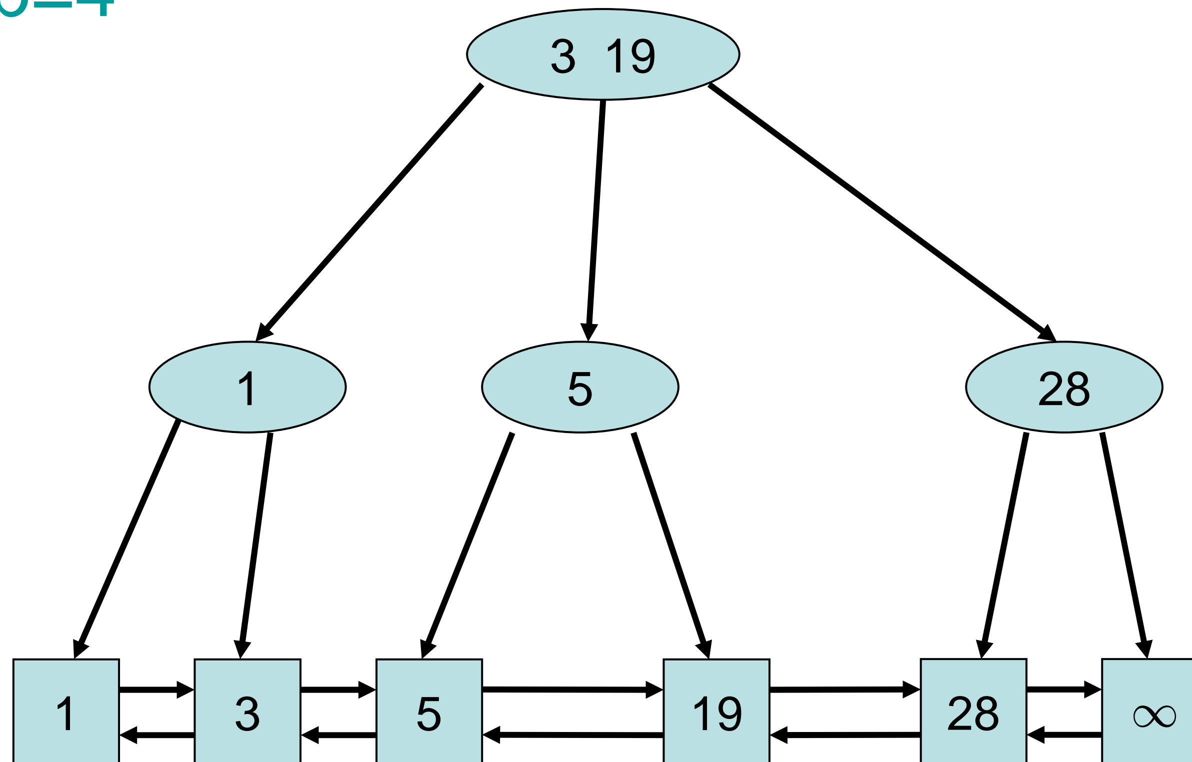
Delete(14)

a=2, b=4



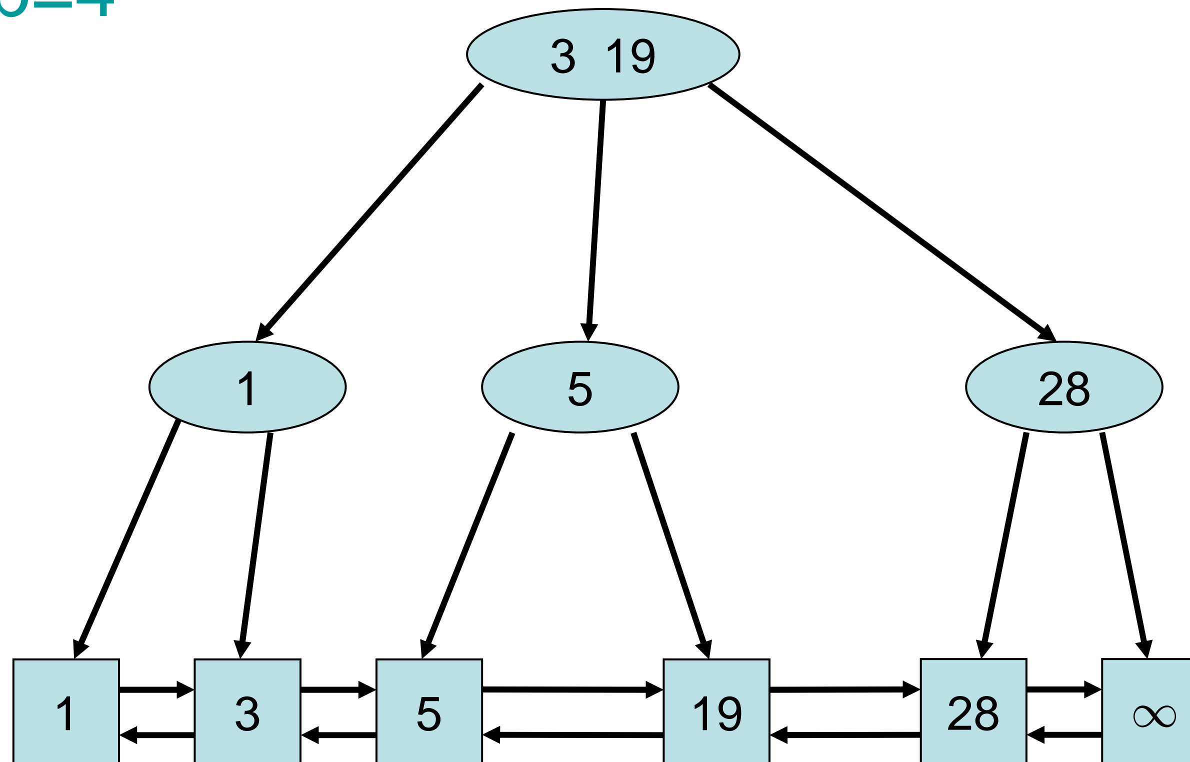
Delete(14)

a=2, b=4



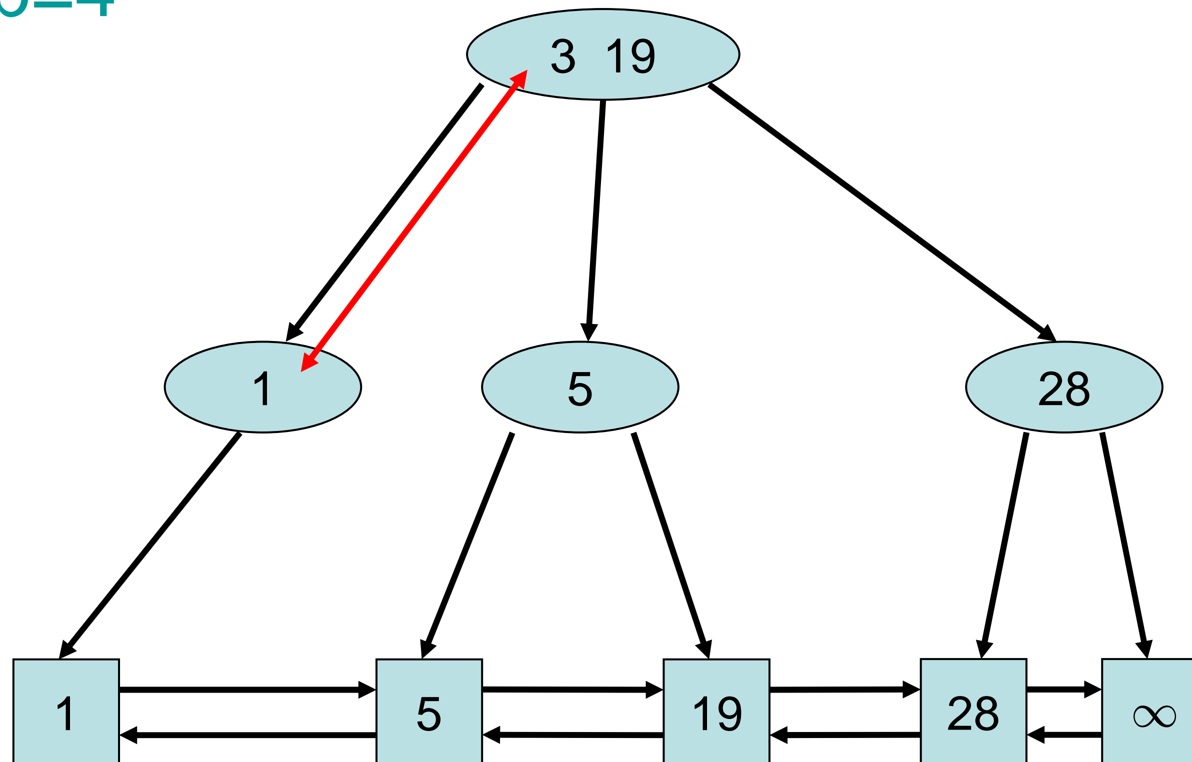
Delete(3)

a=2, b=4



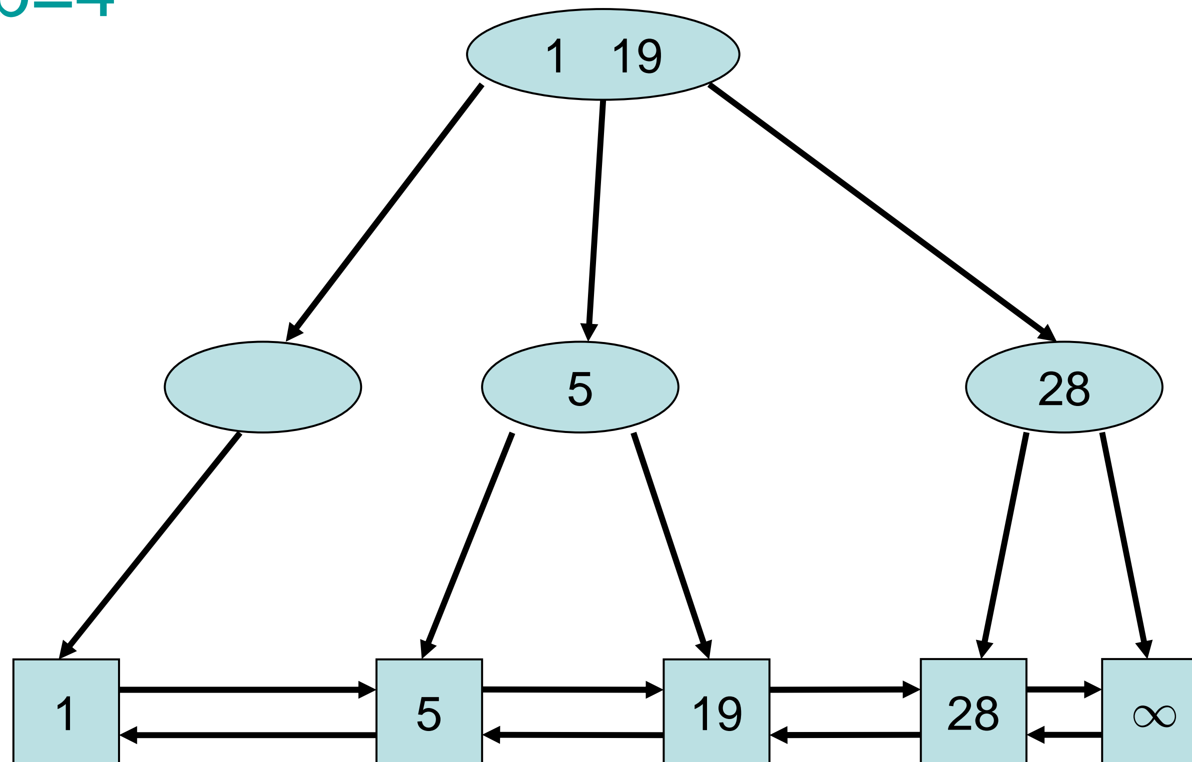
Delete(3)

a=2, b=4



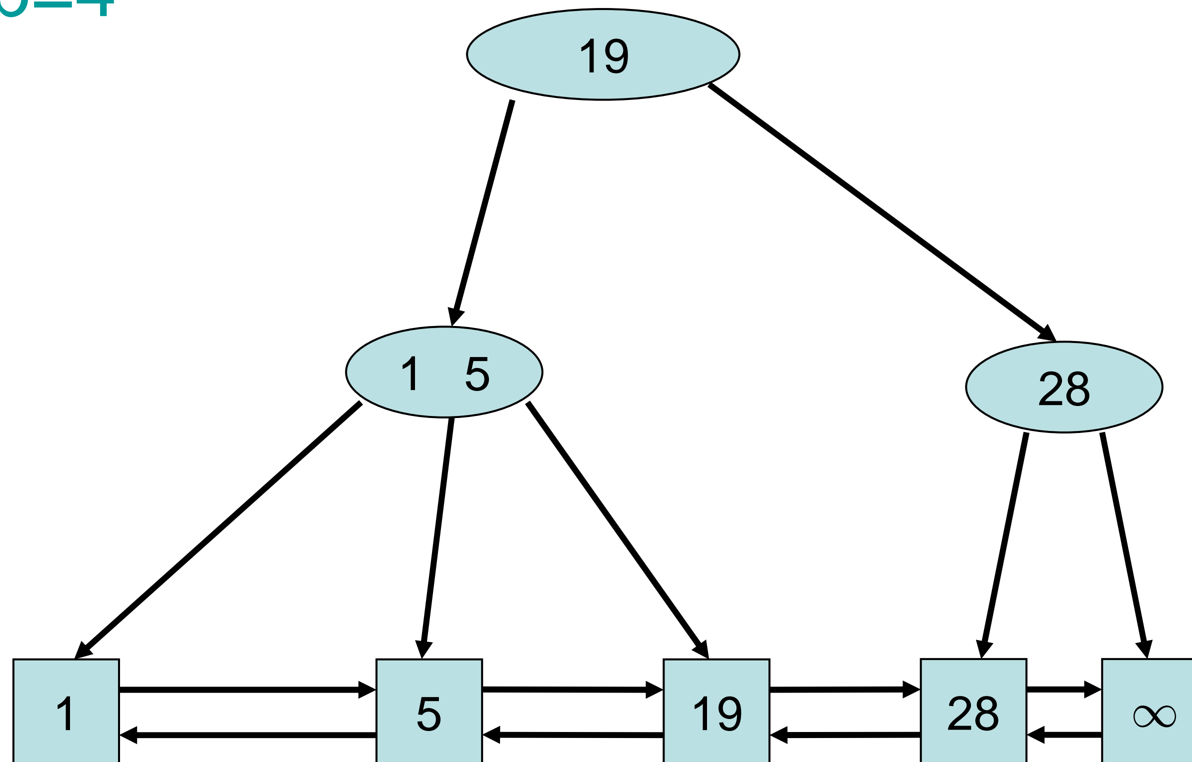
Delete(3)

a=2, b=4



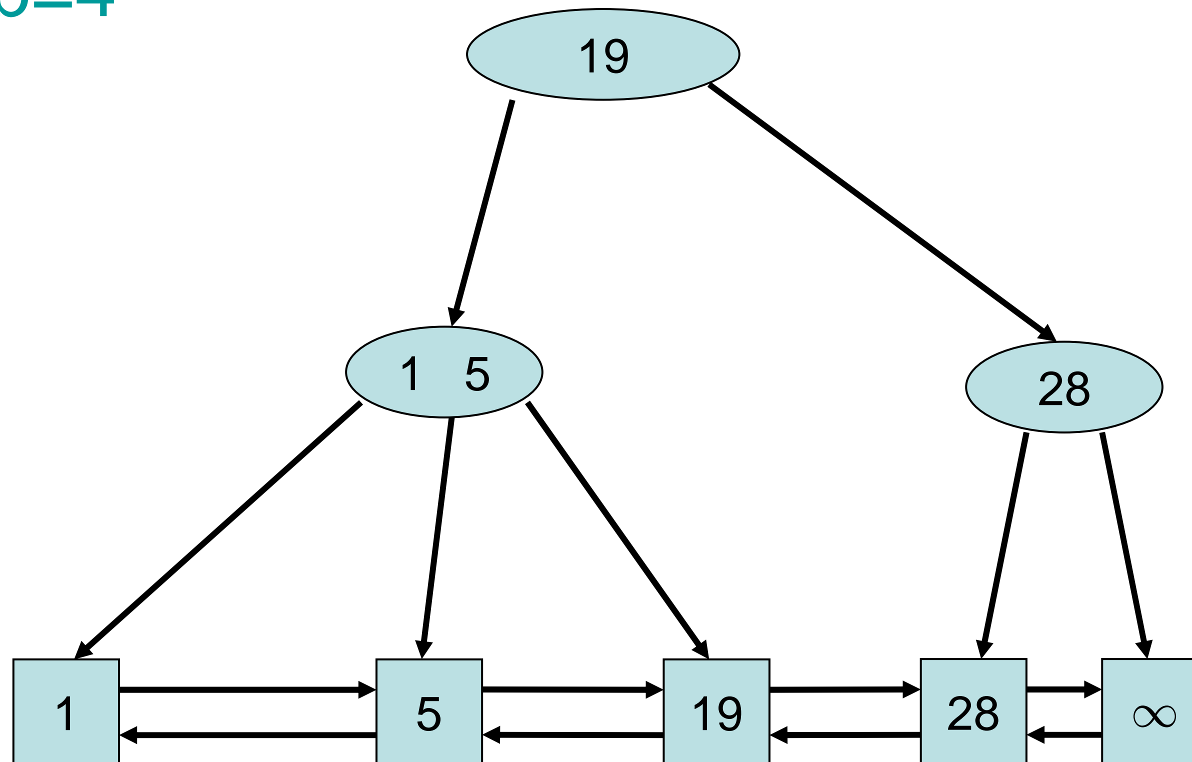
Delete(3)

a=2, b=4



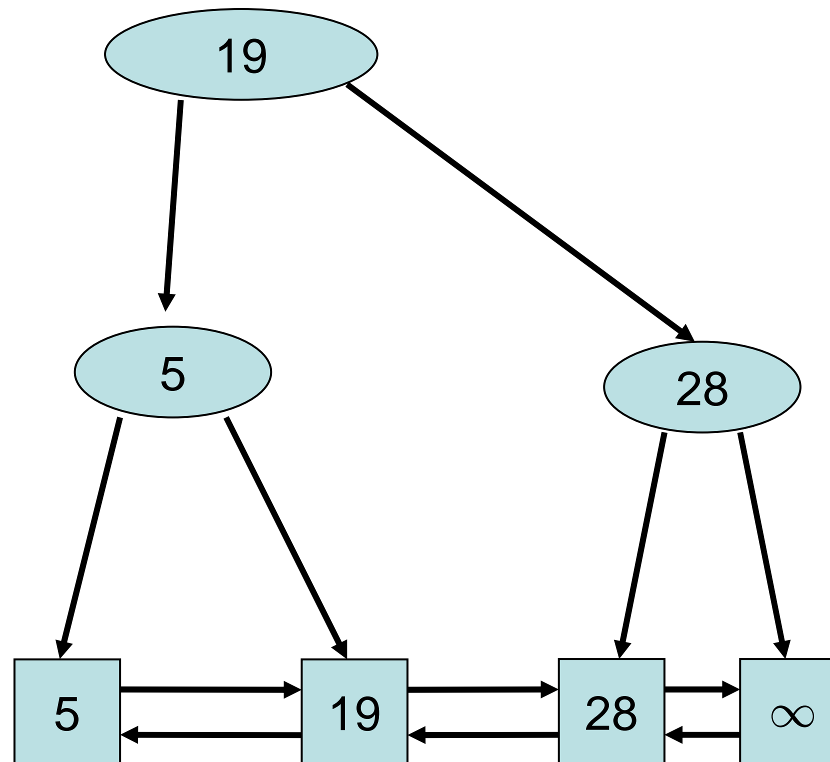
Delete(1)

a=2, b=4



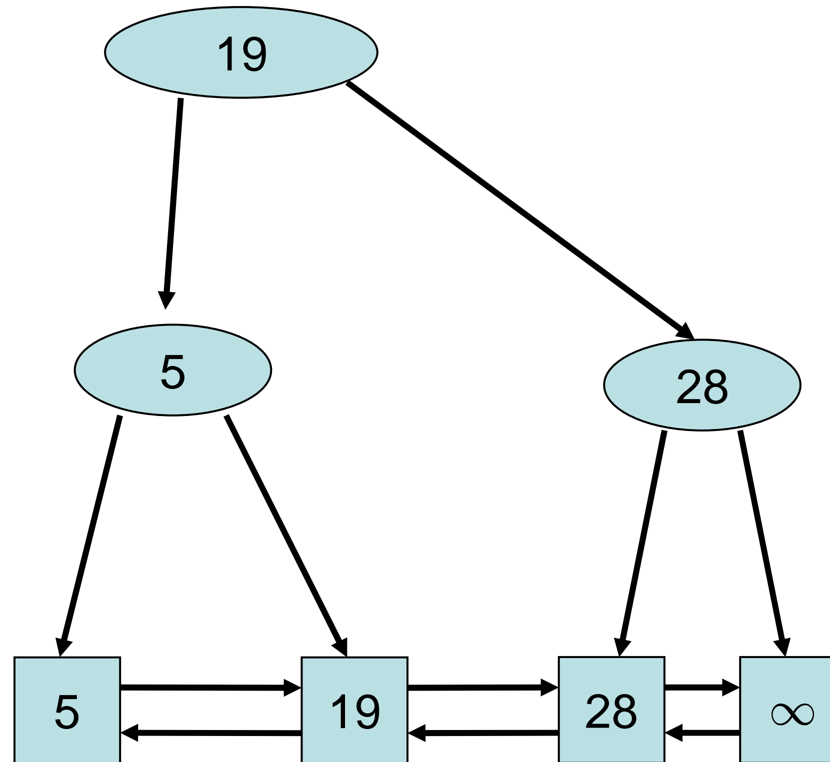
Delete(1)

a=2, b=4



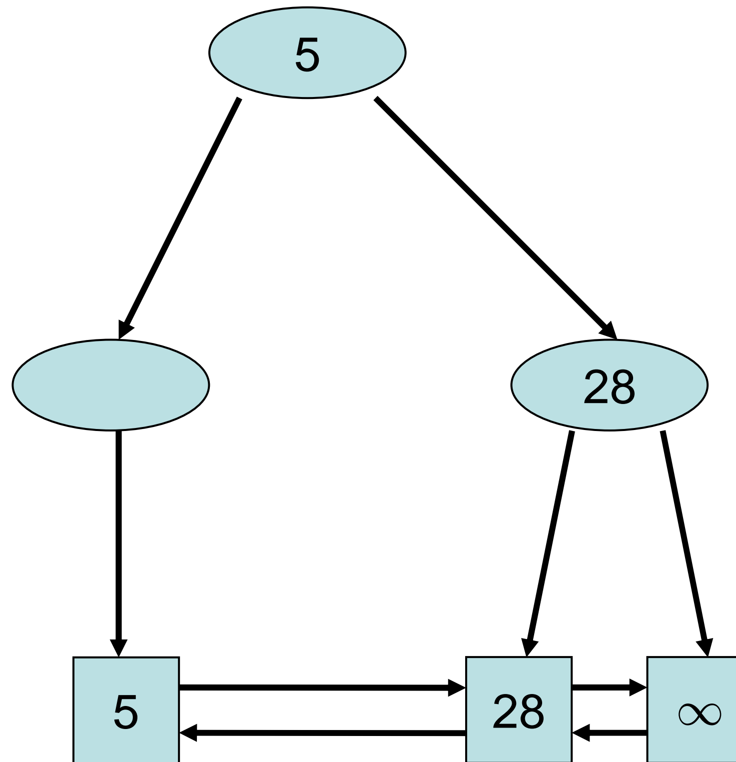
Delete(19)

a=2, b=4



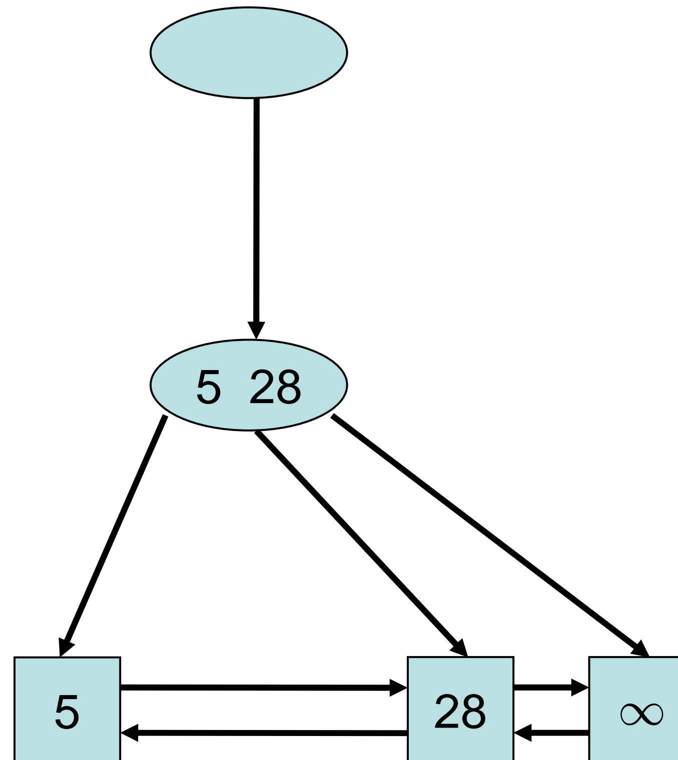
Delete(19)

a=2, b=4



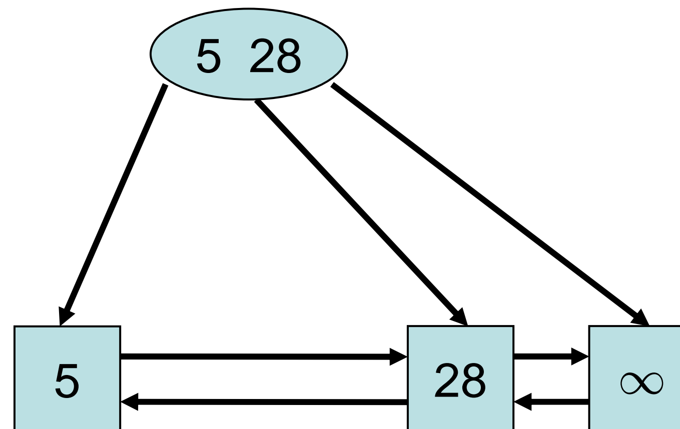
Delete(19)

a=2, b=4



Delete(19)

a=2, b=4



Delete Operation

- Form-Regel:
  Für alle Blätter v,w: $t(v)=t(w)$
  Erfüllt durch Delete!

- Grad-Regel:
  Für alle inneren Knoten v außer Wurzel: $d(v) \in [a,b]$, für Wurzel r: $d(r) \in [2,b]$

  1) Delete verschmilzt Knoten mit Grad $a-1$ mit Knoten mit Grad a. Wenn $b>=2a-1$, dann hat resultierender Knoten Grad $<=b$.
  2) Delete verschiebt Kante von Knoten mit Grad $>a$ nach Knoten mit Grad $a-1$. Auch OK.
  3) Wurzel gelöscht: Kinder vorher verschmolzen, Grad vom verbeibenden Kind $>=a$ (und $<=b$), also auch OK.

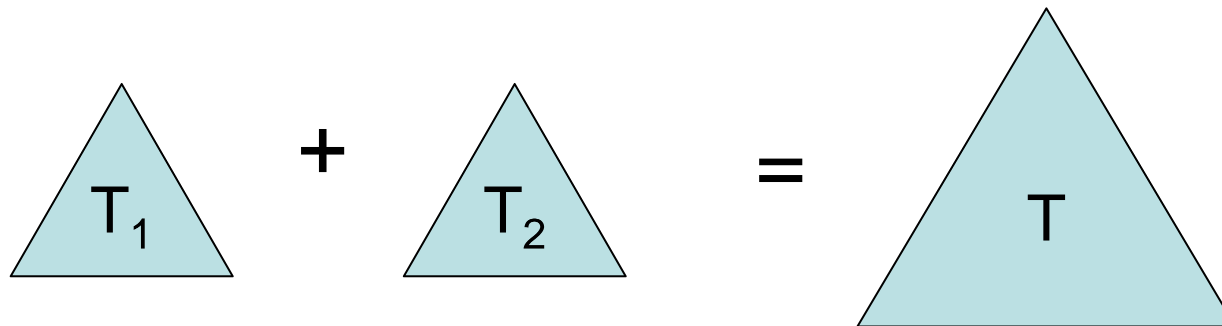
Mehr Operationen

- min/max Operation:
  Listenenden bekannt: Zeit O(1).

- Bereichsanfragen:
  Um alle Elemente im Bereich [x,y] zu suchen, führe search(x) aus und durchlaufe dann die Liste, bis ein Element >y gefunden wird.
  Zeit O(log n + Ausgabegröße).

Mehr Operationen

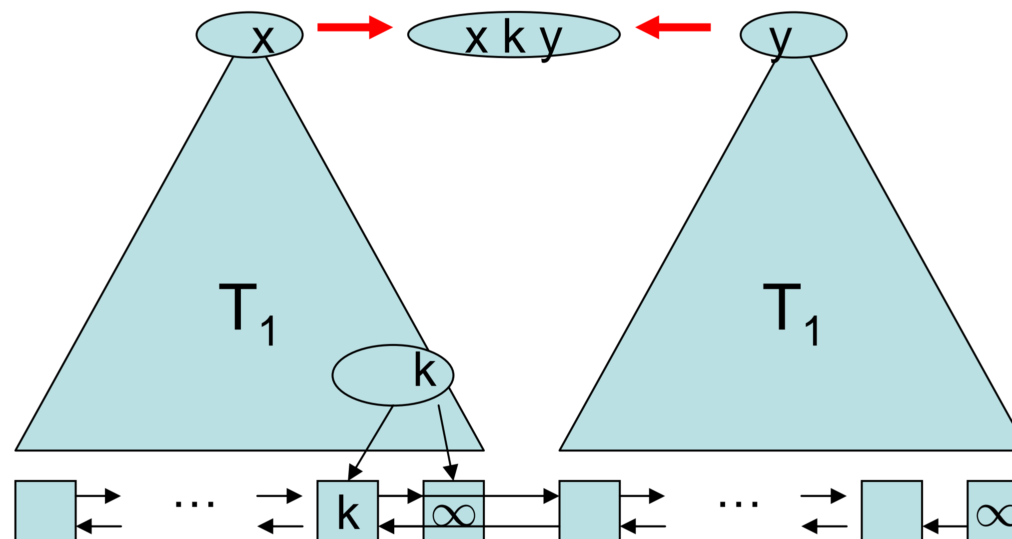
- Concatenate-Operation:
  Ziel: Verknüpfe zwei (a,b)-Bäume T_1 und T_2 mit s_1 und s_2 Elementen zu (a,b)-Baum T (Schlüssel in T_1 <= Schlüssel in T_2)



Mehr Operationen

Concatenate-Operation:

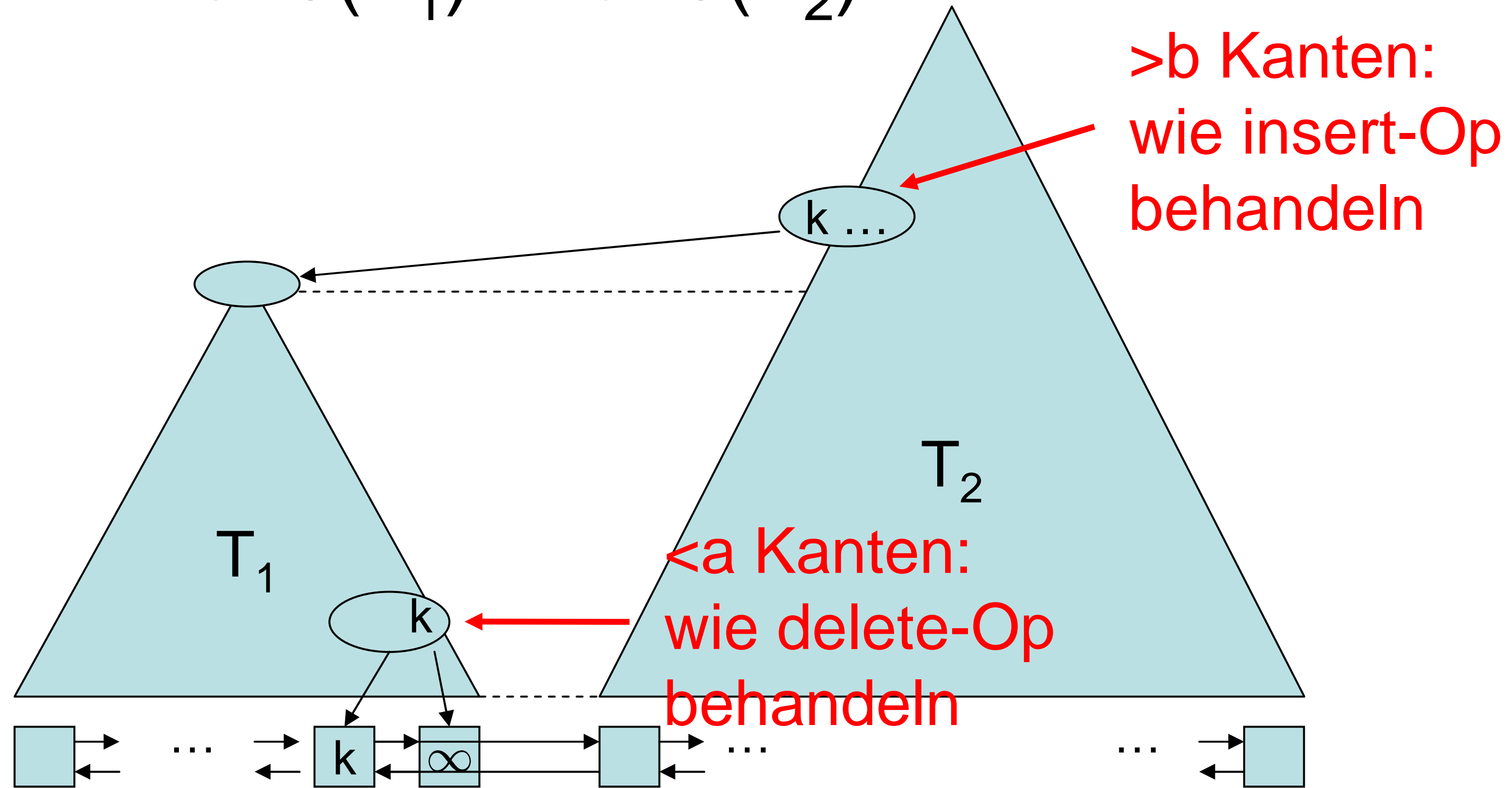
Fall 1: Höhe(T_1)=Höhe(T_2)



Mehr Operationen

Concatenate-Operation:

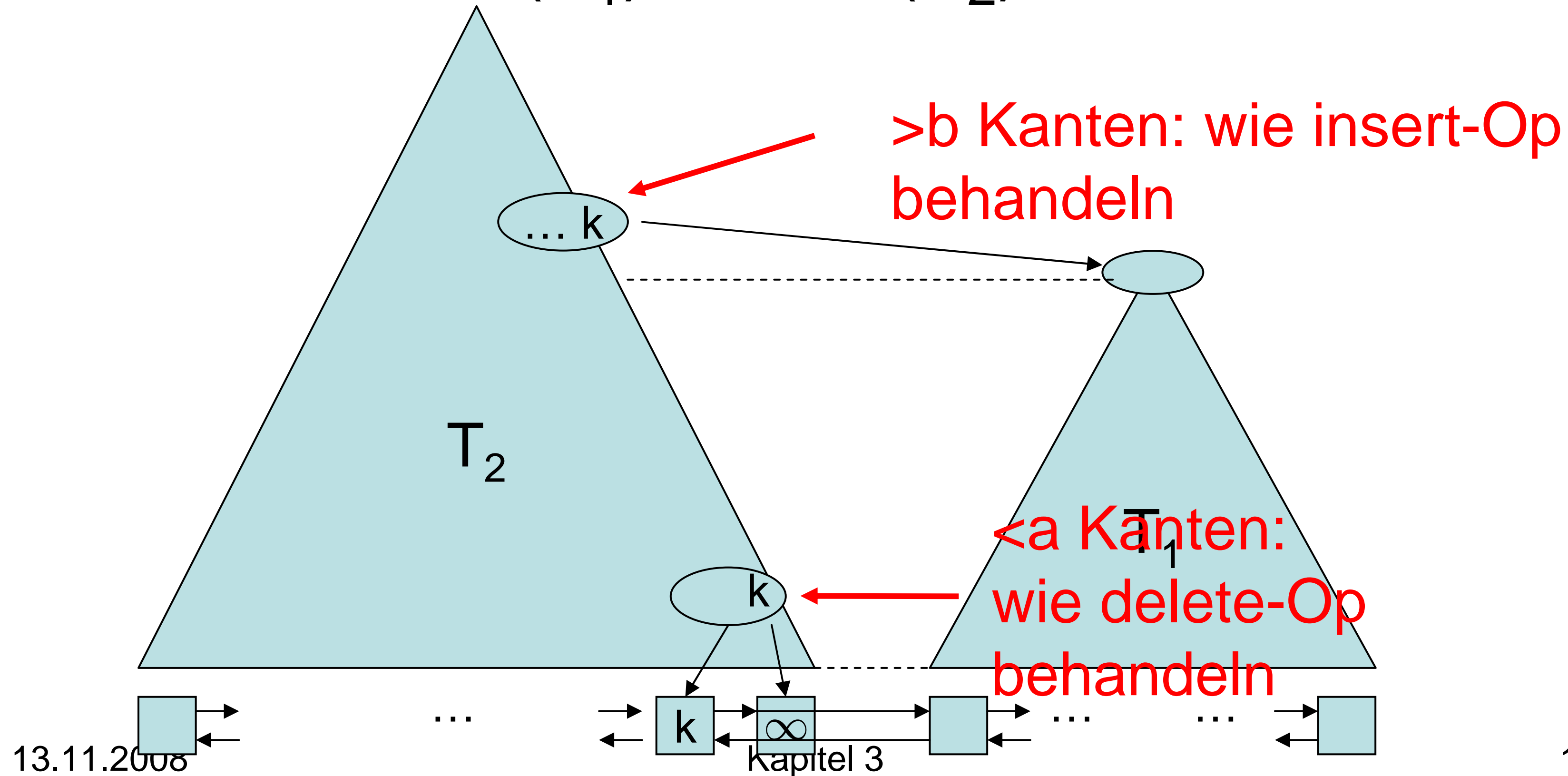
Fall 2: Höhe(T_1)<Höhe(T_2):



Mehr Operationen

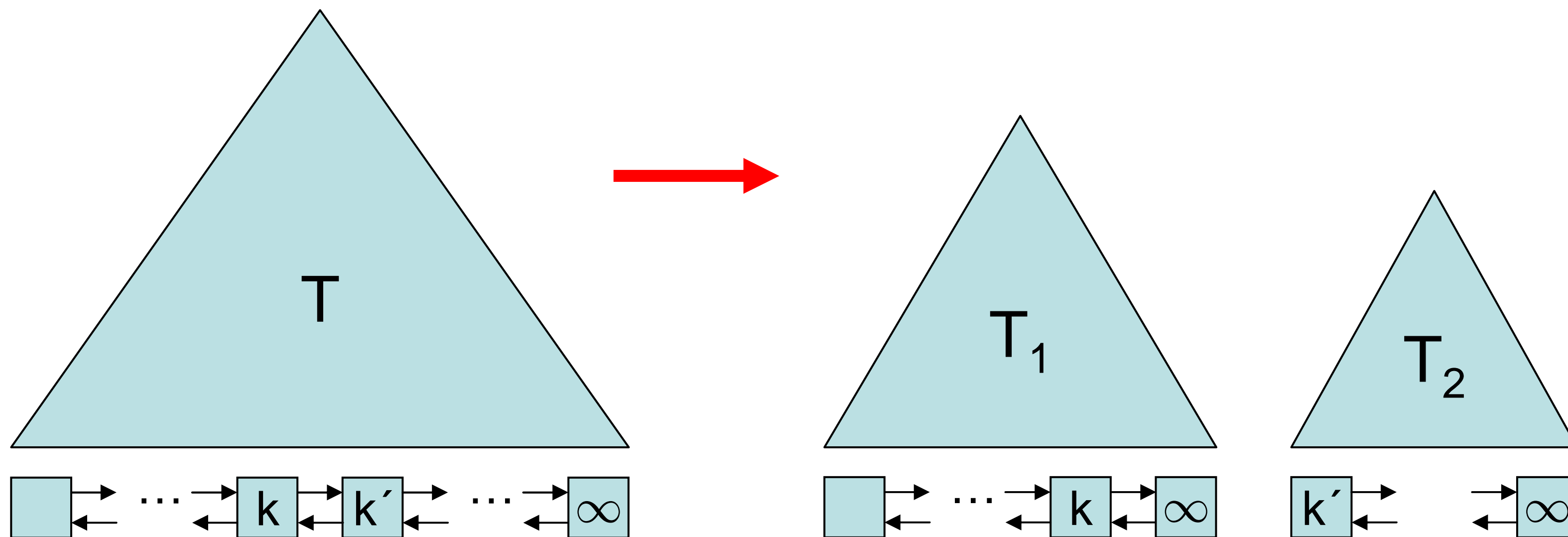
Concatenate-Operation:

Fall 2: Höhe(T_1)>Höhe(T_2):



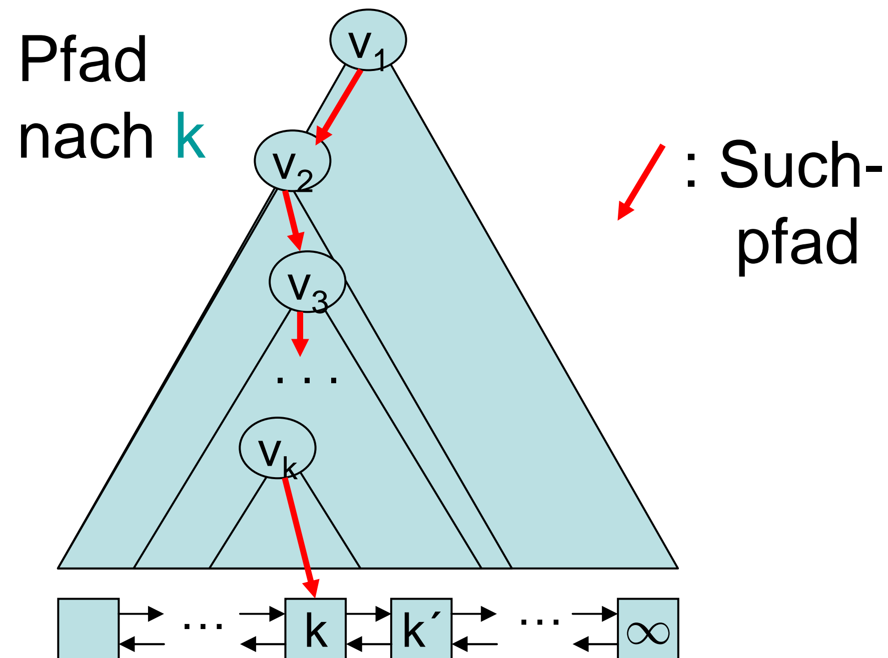
Mehr Operationen

- Cut-Operation:
  Ziel: Spalte (a,b)-Baum T in (a,b)-Bäume T_1 und T_2 bei Schlüssel k auf.



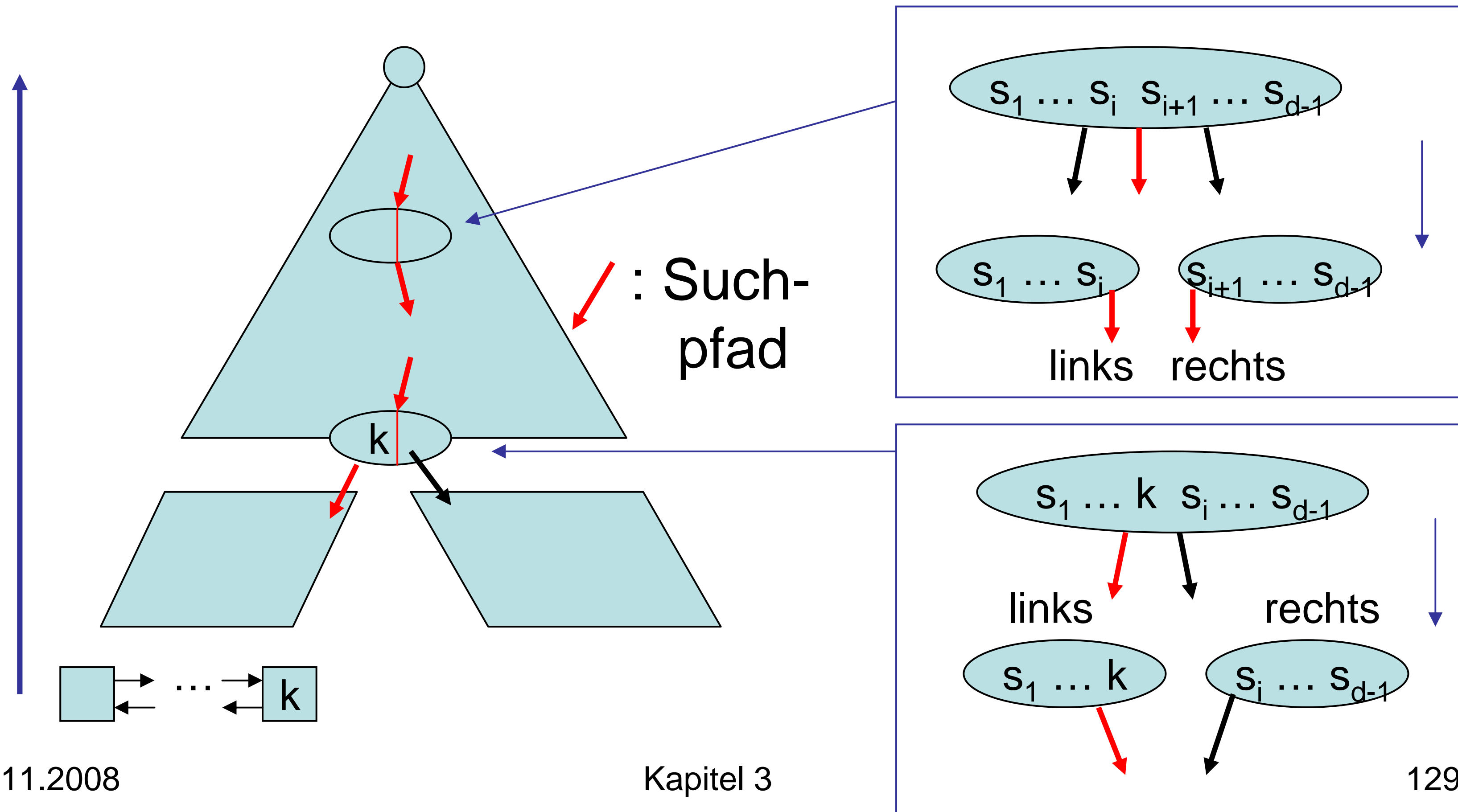
Cut-Operation

1. Suche nach k



Cut-Operation

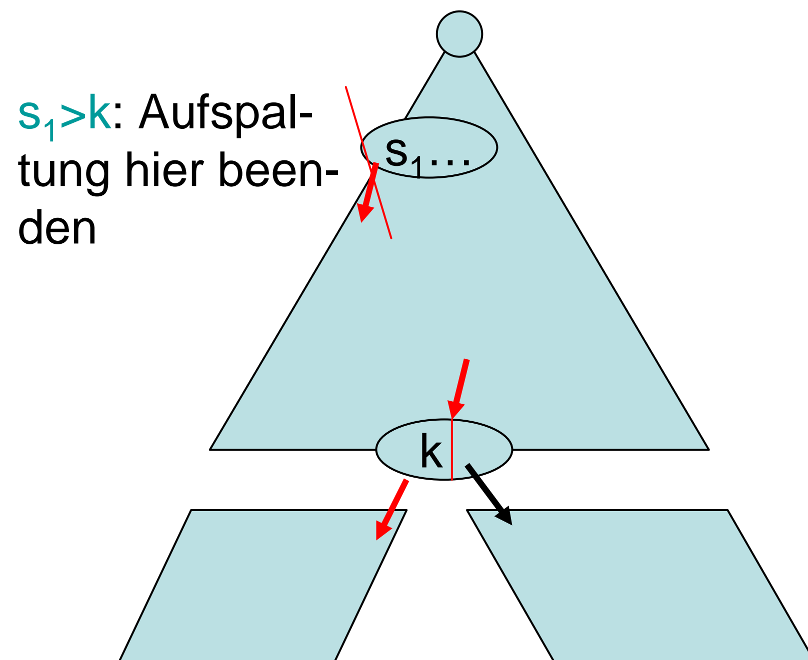
2. Aufspaltung entlang Suchpfad



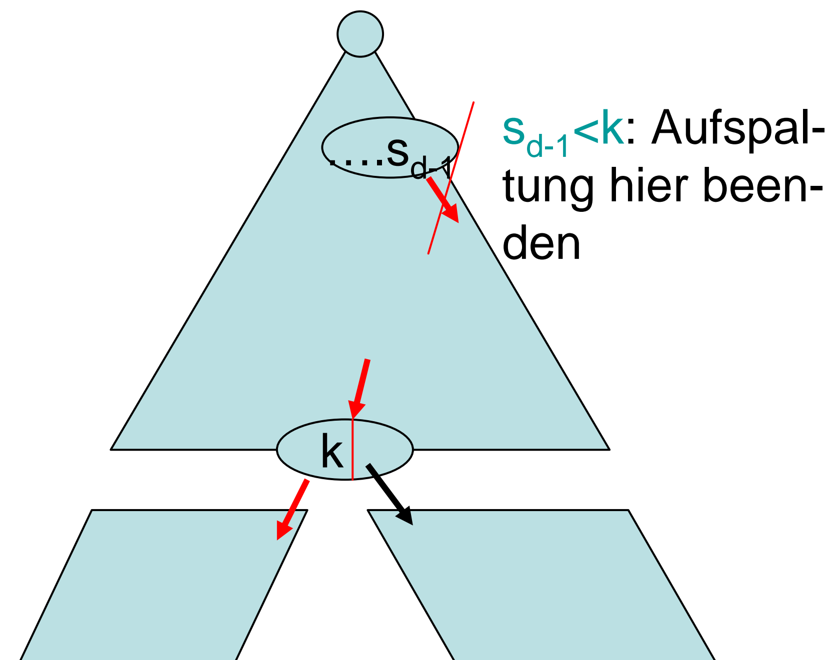
Cut-Operation

2. Abbruch bei Aufspaltung:

Fall 1:

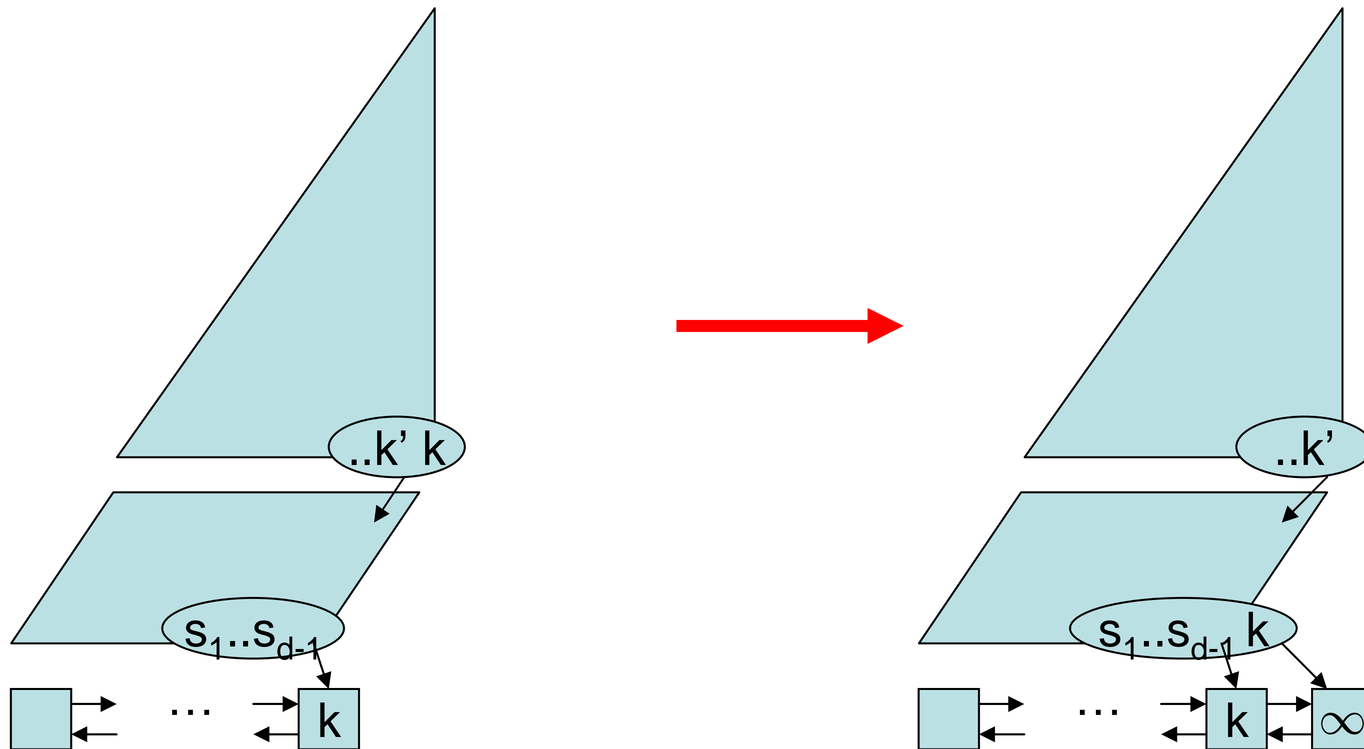


Fall 2:



Cut-Operation

3. Einfügung von ∞ in linken Teilbaum

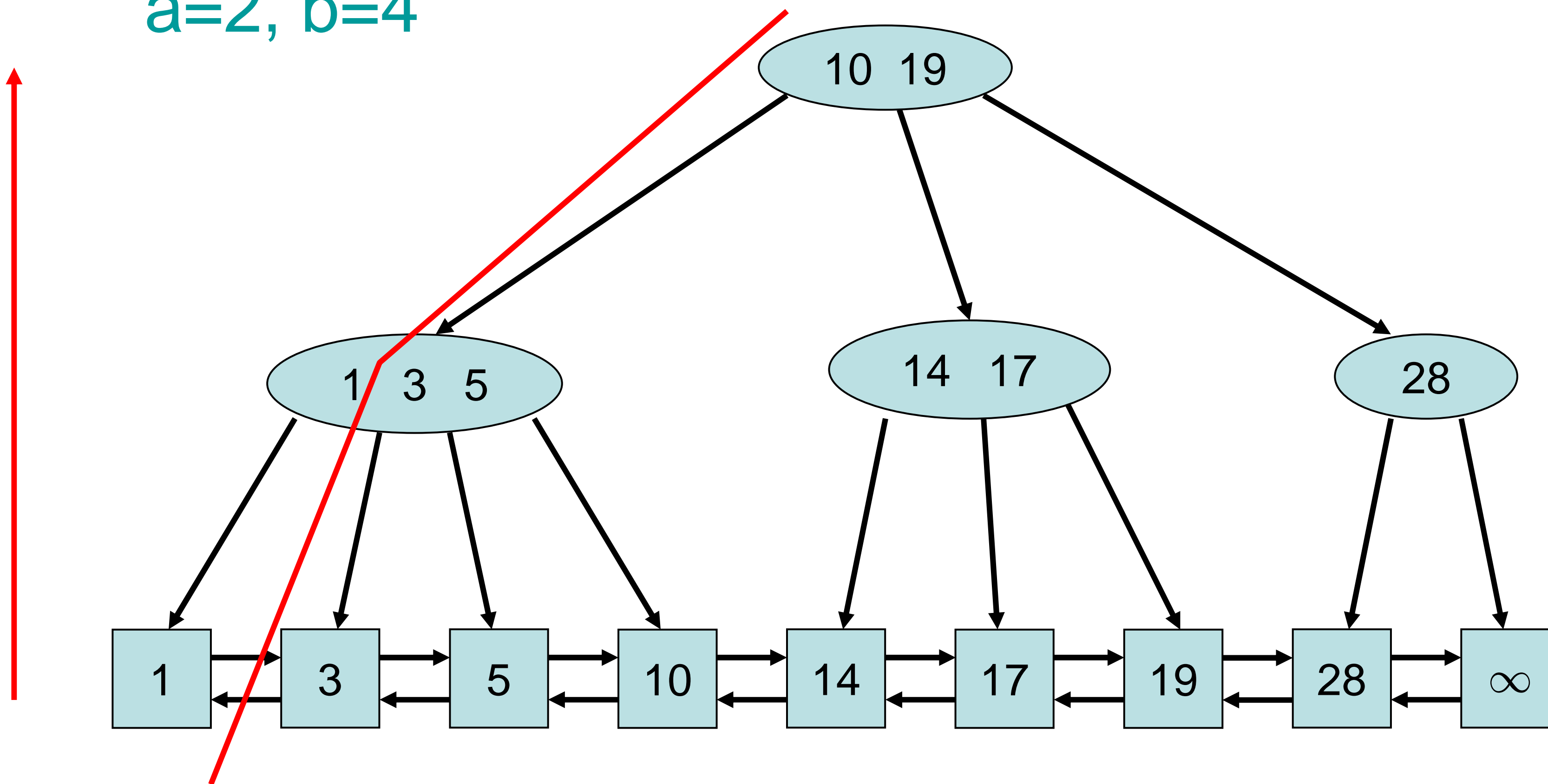


Cut-Operation

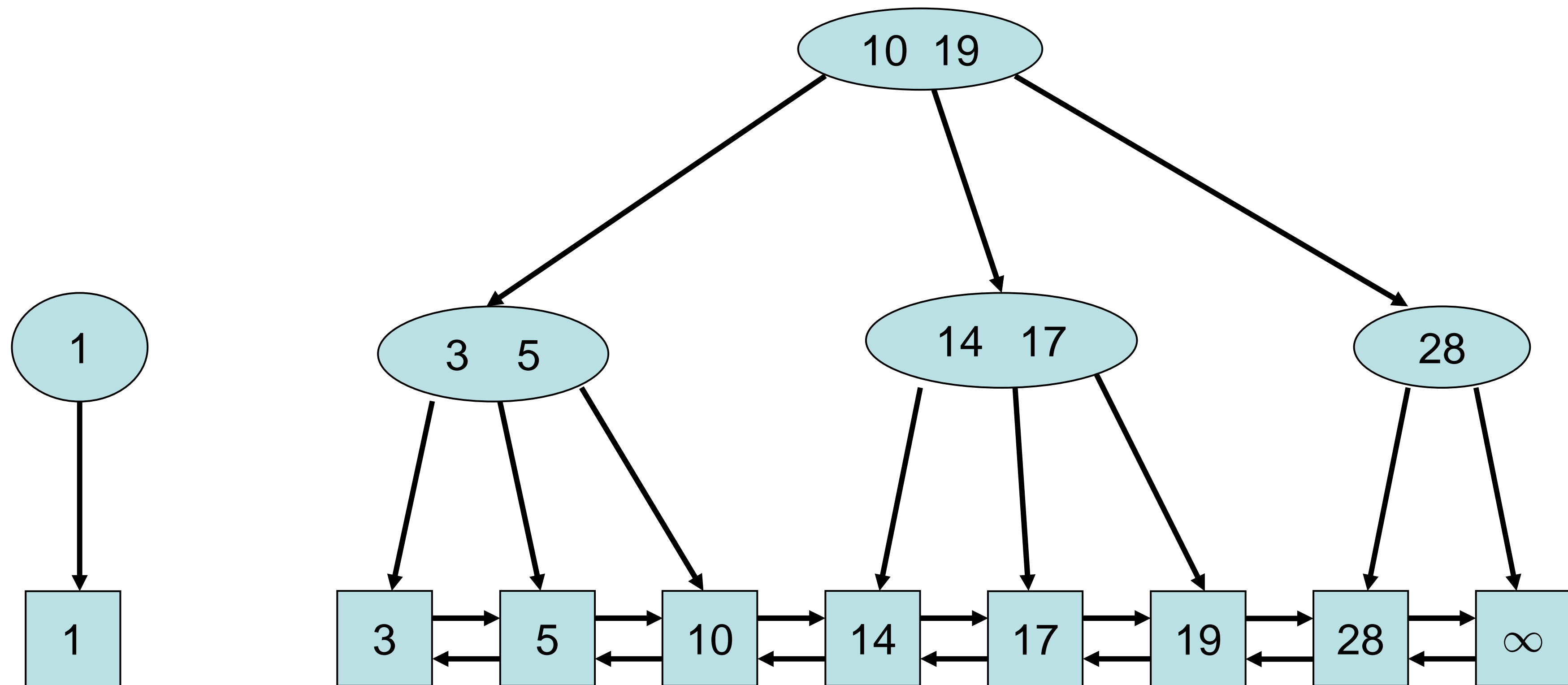
4. Gradreparatur in den beiden Teilbäumen

Strategie: bottom-up entlang Suchpfad, um zu gültigen (a,b)-Bäumen zurückzukehren. (Wie bei insert und delete Operationen.)

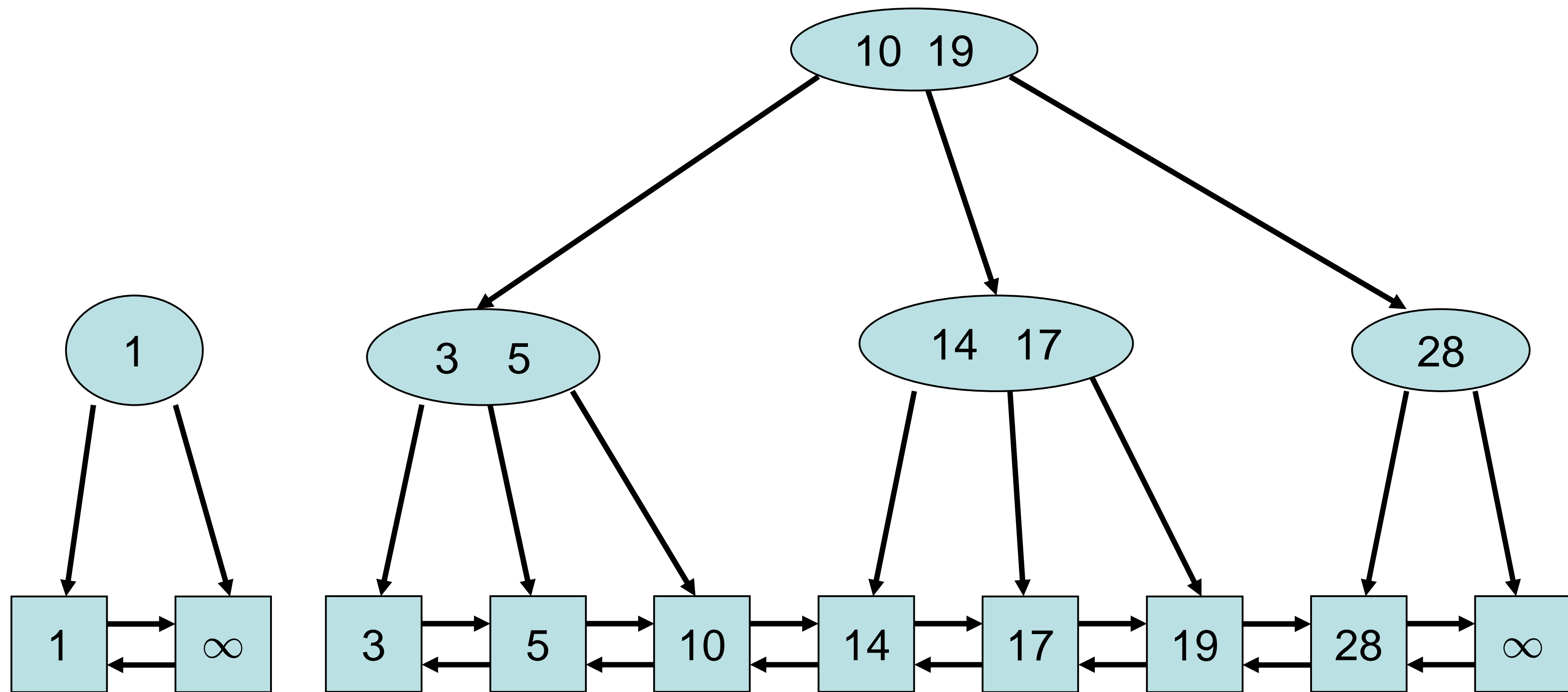
Cut(1)



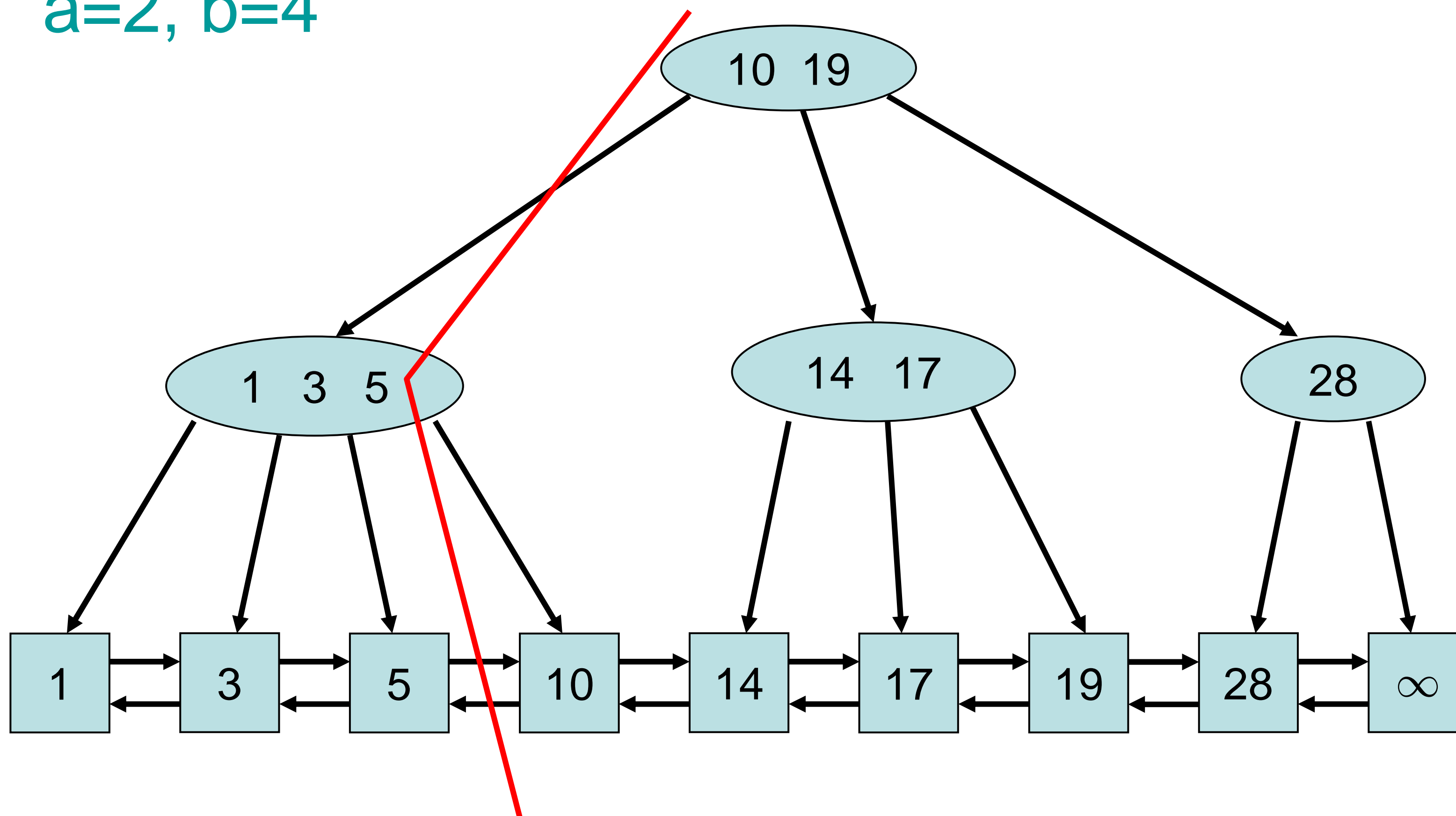
Cut(1)



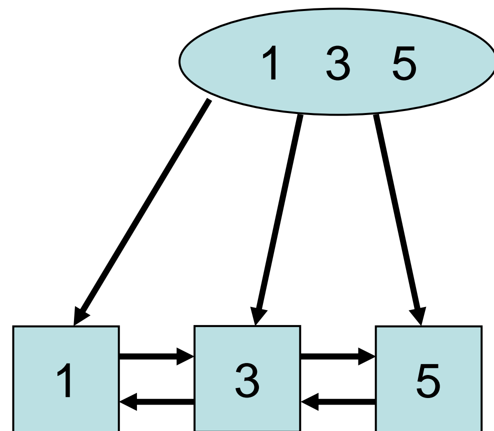
Cut(1)

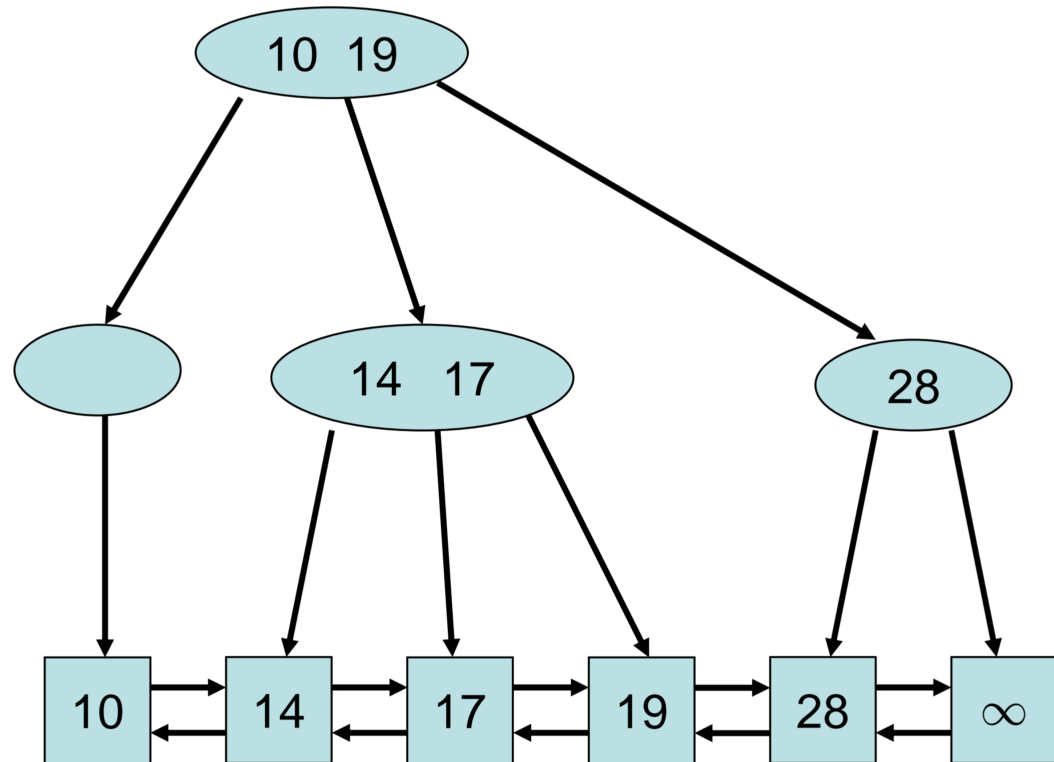


Cut(5)

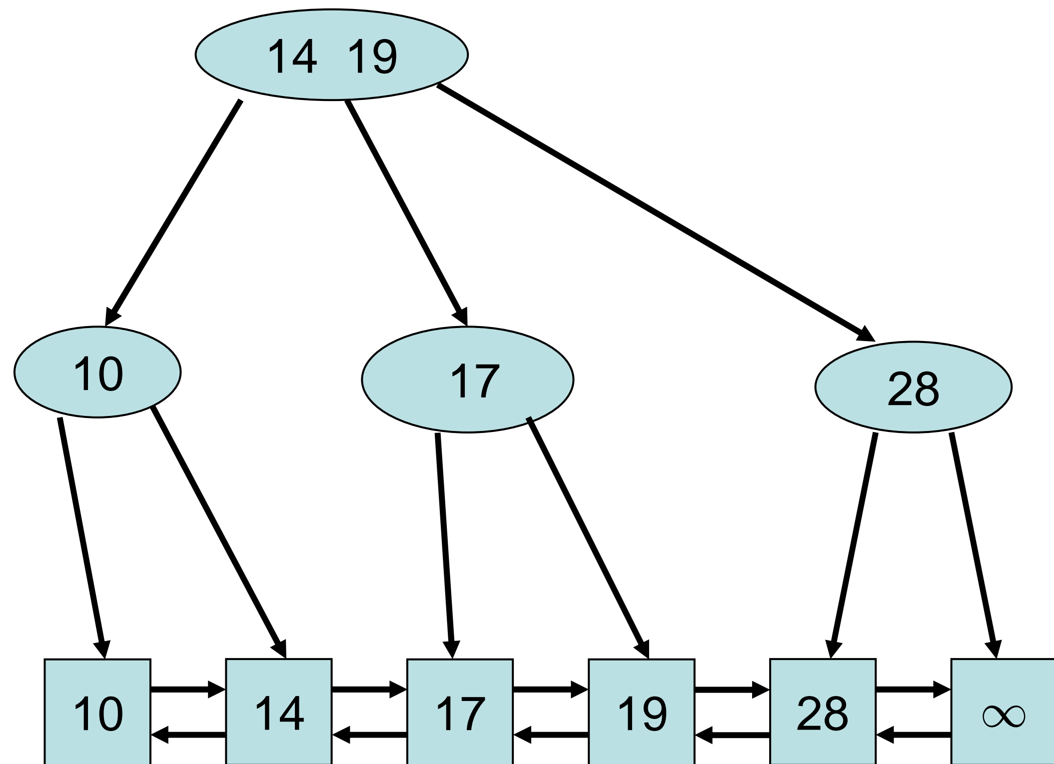


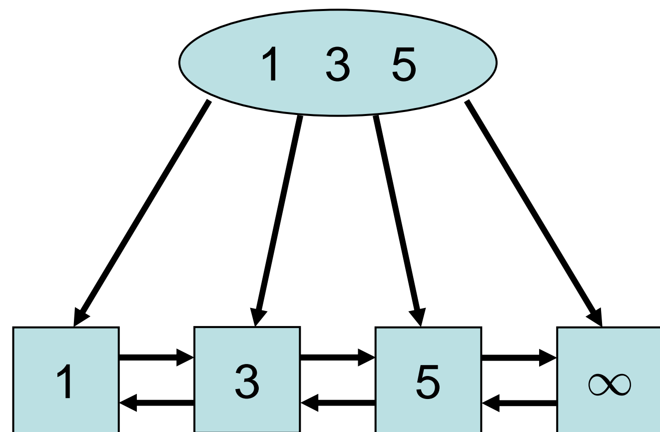
Cut(5)



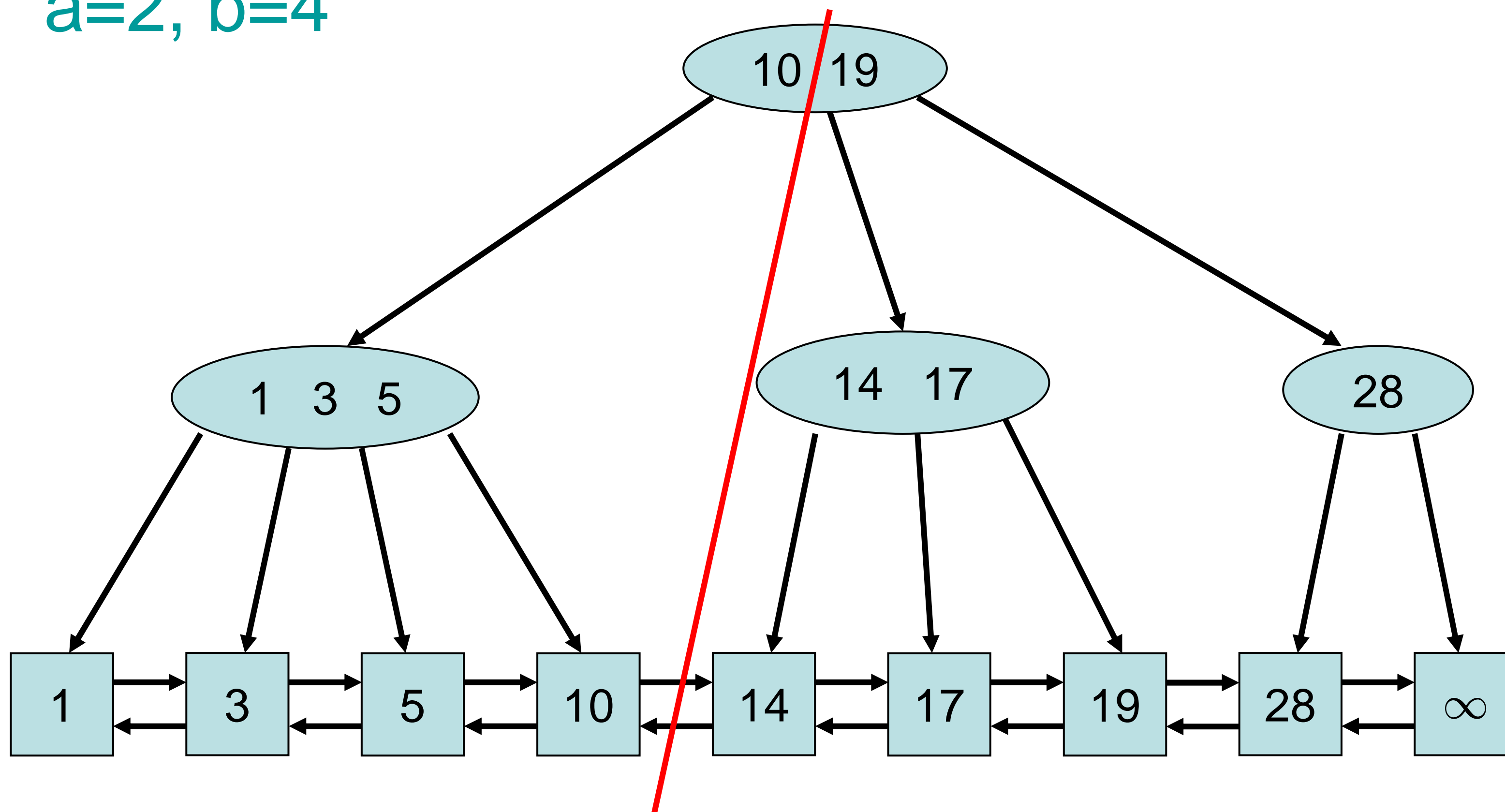


Cut(5)

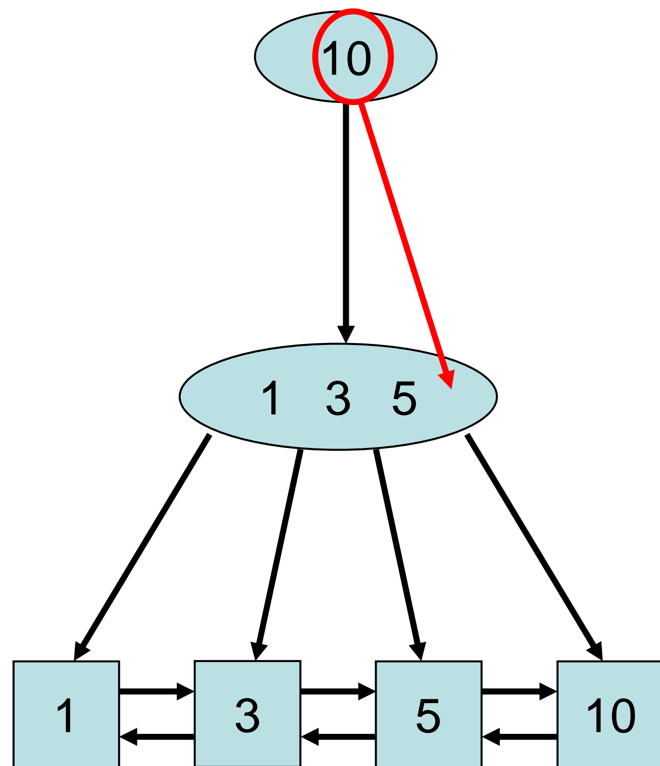


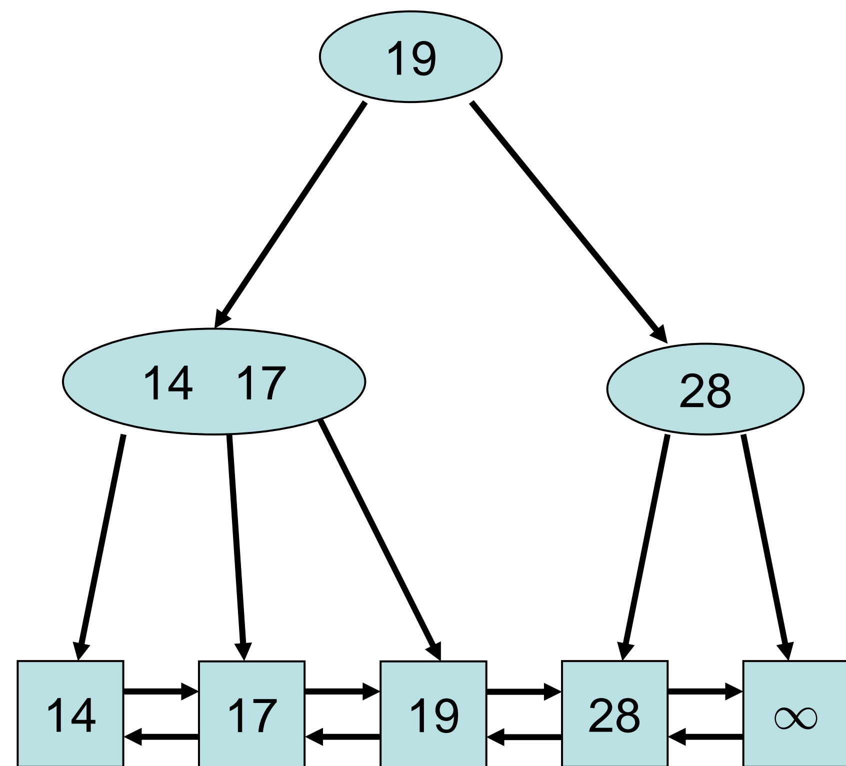


Cut(10)

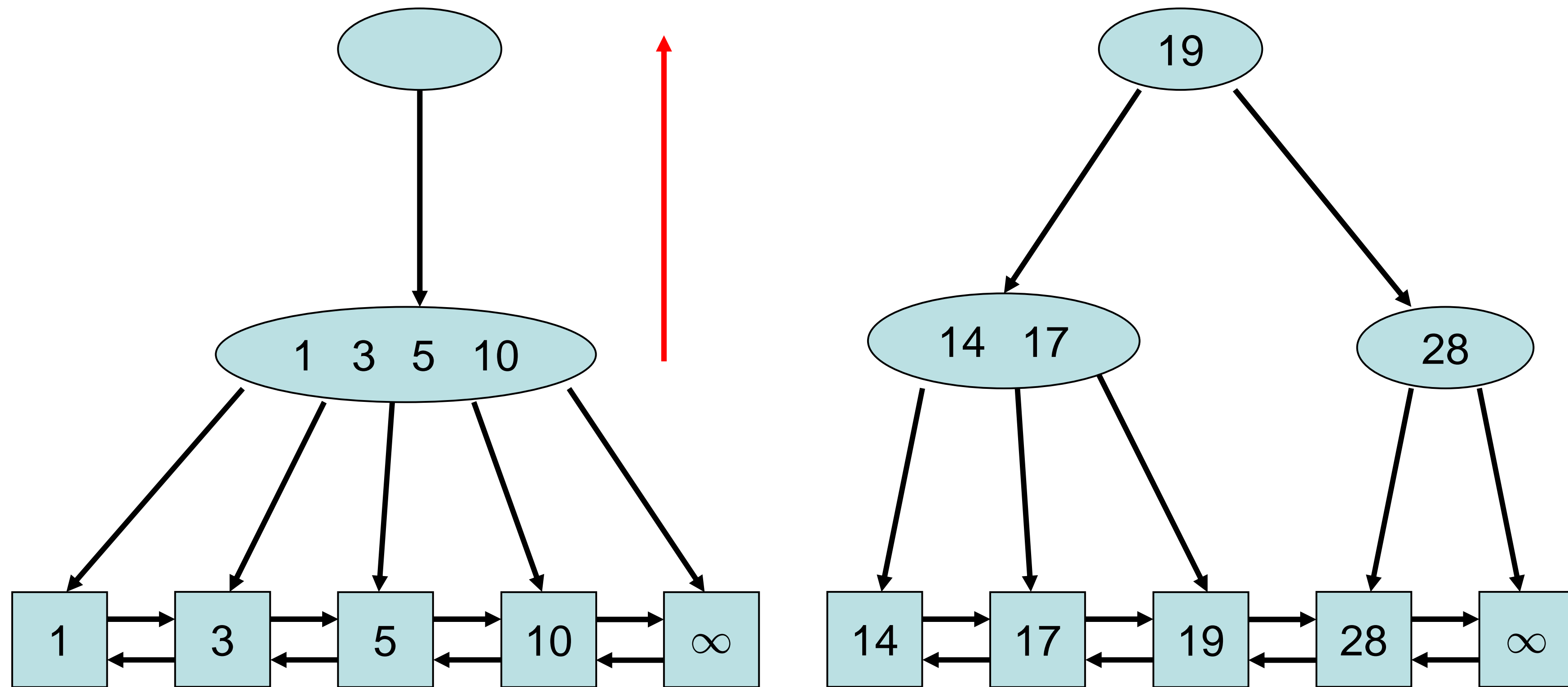


Cut(10)

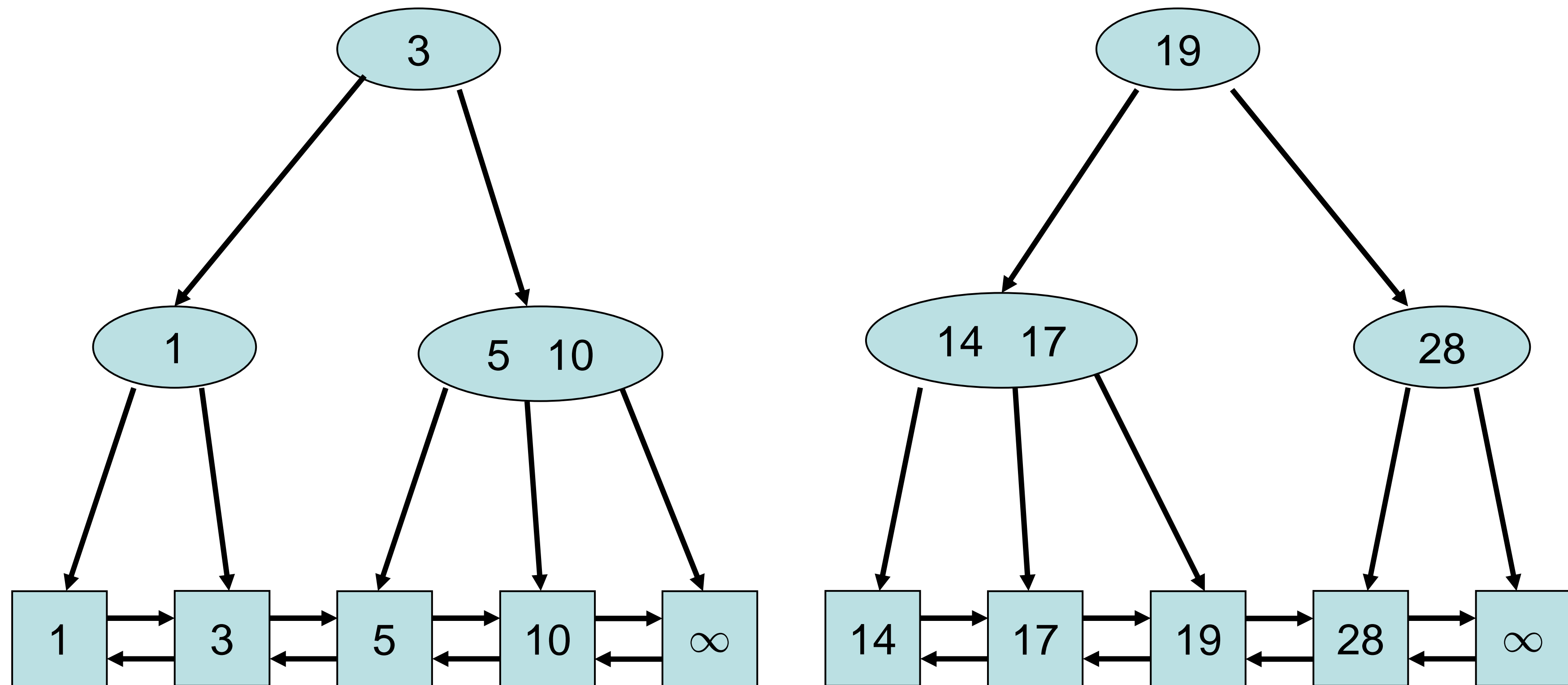




Cut(10)



Cut(10)



n Update-Operationen

Theorem 3.11: Es gibt eine Folge von n insert und delete Operationen im (2,3)-Baum, so dass Gesamtanzahl der split und merge Operationen $\Omega(n \log n)$ ist.

Beweis: Übung.

n Update-Operationen

Theorem 3.12: Betrachte einen (a,b)-Baum mit b>=2a, der anfangs leer ist. Für jede Folge von n insert und delete Operationen ist die Gesamtanzahl der split und merge Operationen O(n).

Beweis:

Amortisierte Analyse

Binärbaum

Problem: Binärbaum kann entarten!

Lösungen:

- Splay-Baum
  (sehr effektive Heuristik)
- Treaps
  (mit hoher Wkeit gut balanciert)
- (a,b)-Baum
  (garantiert gut balanciert)
- Rot-Schwarz-Baum
  (konstanter Reorganisationsaufwand)
- Gewichtsbalancierter Baum
  (kompakt einbettbar in Feld)

Rot-Schwarz-Baum

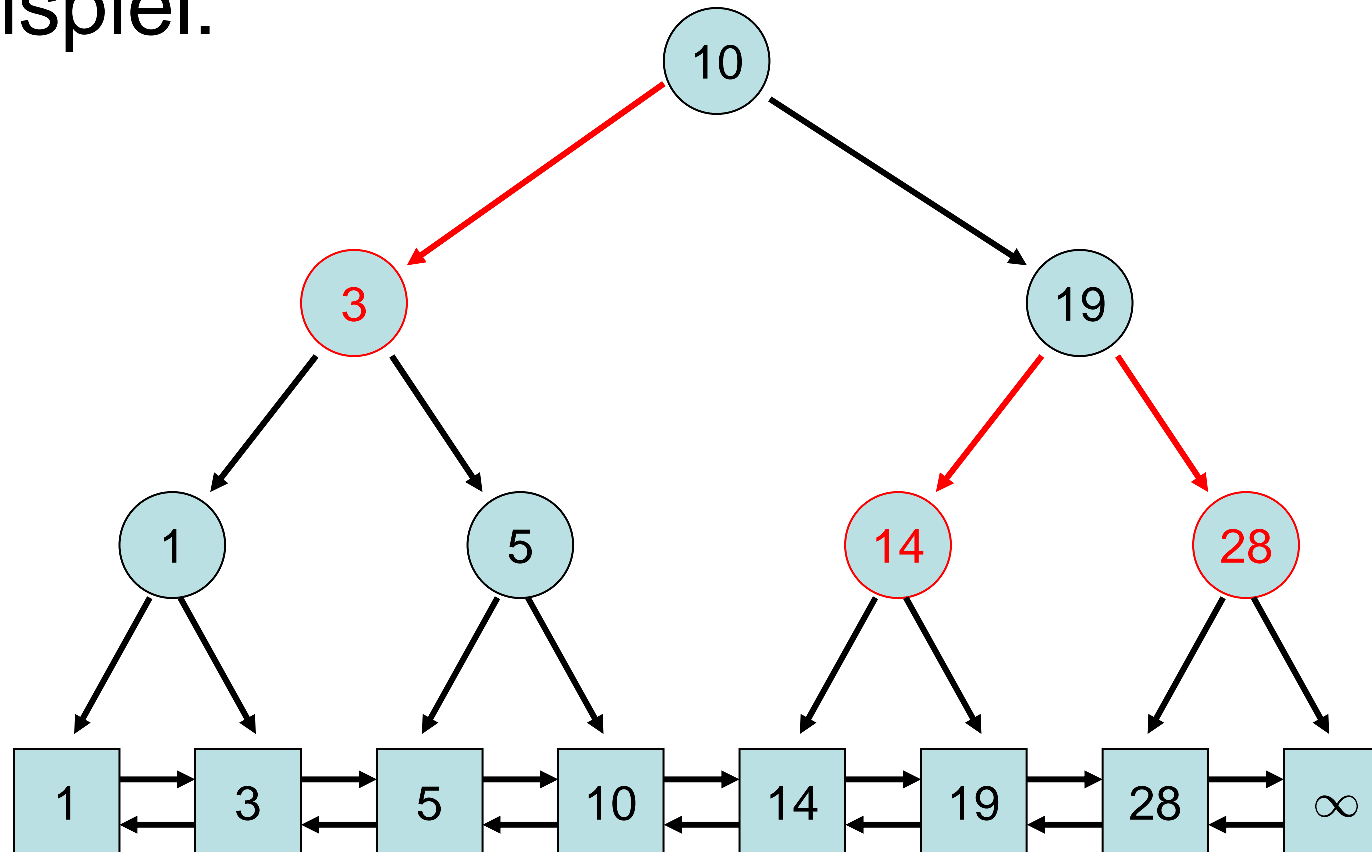
Rot-Schwarz-Bäume sind binäre Suchbäume mit roten und schwarzen Knoten, so dass gilt:

- Wurzelregel: Die Wurzel ist schwarz.
- Externe Regel: Jeder Listenknoten ist schwarz.
- Interne Regel: Die Kinder eines roten Knotens sind schwarz.
- Tiefenregel: Alle Listenknoten haben dieselbe "Schwarztiefe"

"Schwarztiefe" eines Knotens: Anzahl der schwarzen Baumknoten (außer der Wurzel) auf dem Pfad von der Wurzel zu diesem Knoten.

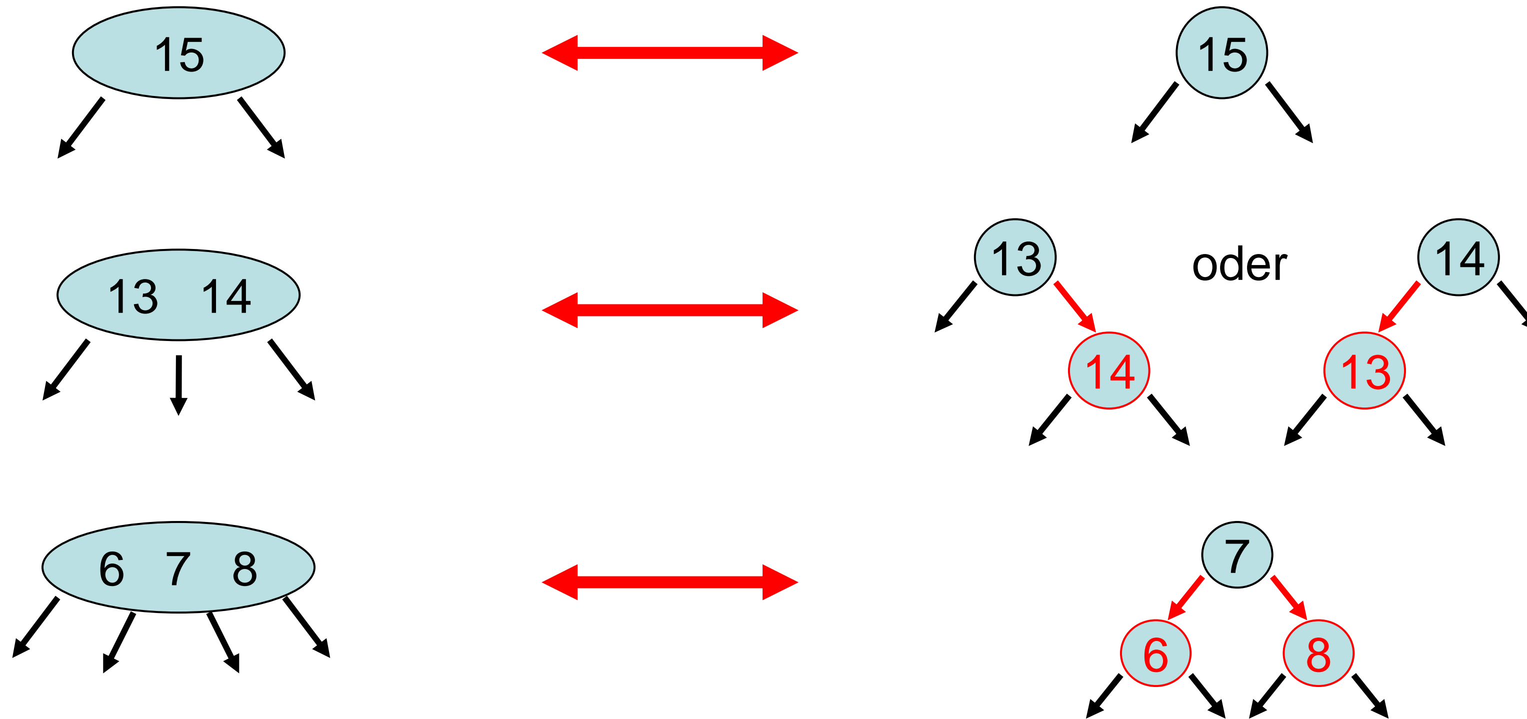
Rot-Schwarz-Baum

Beispiel:



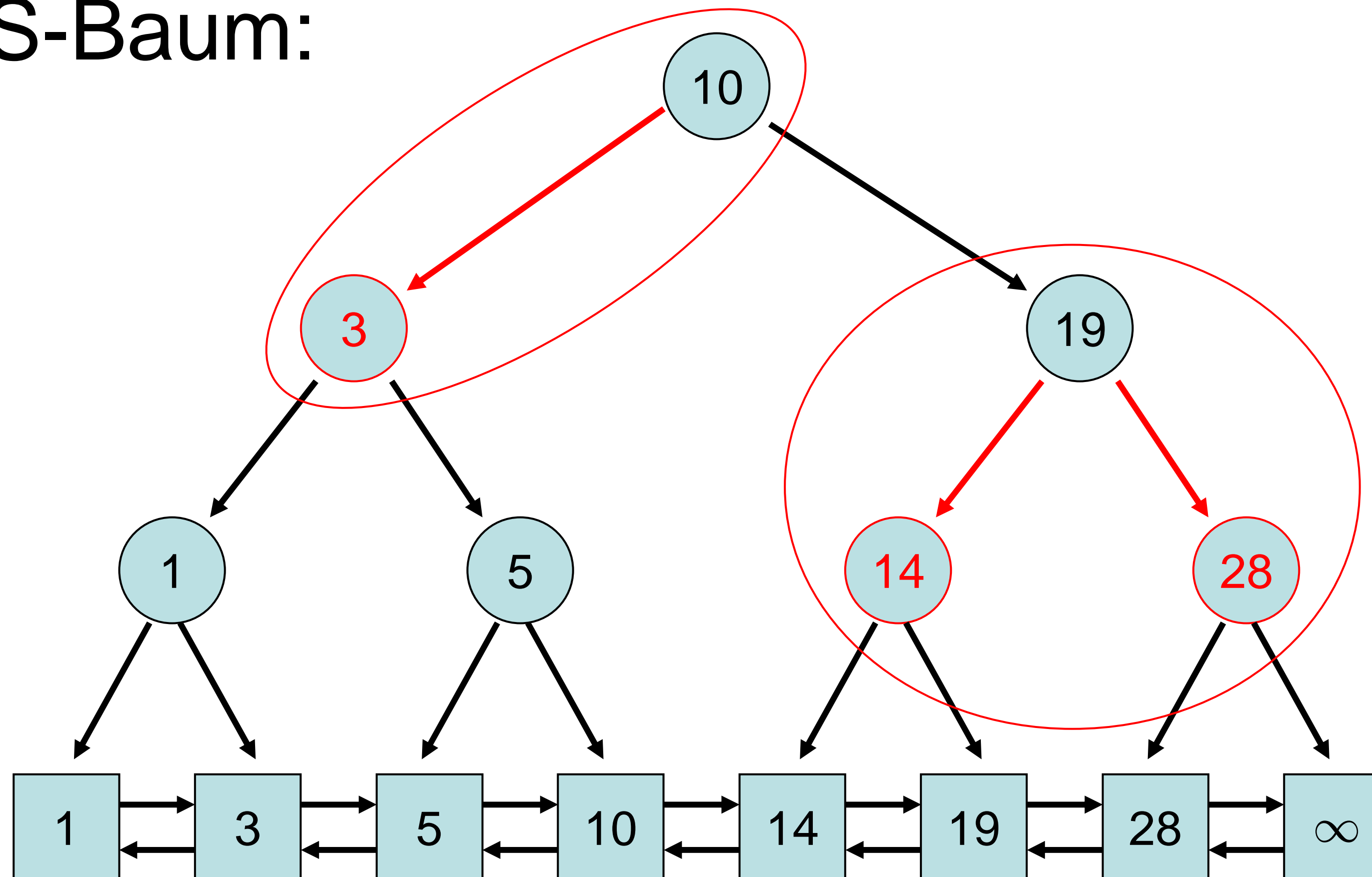
Rot-Schwarz-Baum

Es gibt einen direkten Zusammenhang zwischen (2,4)- und Rot-Schwarz-Bäumen:



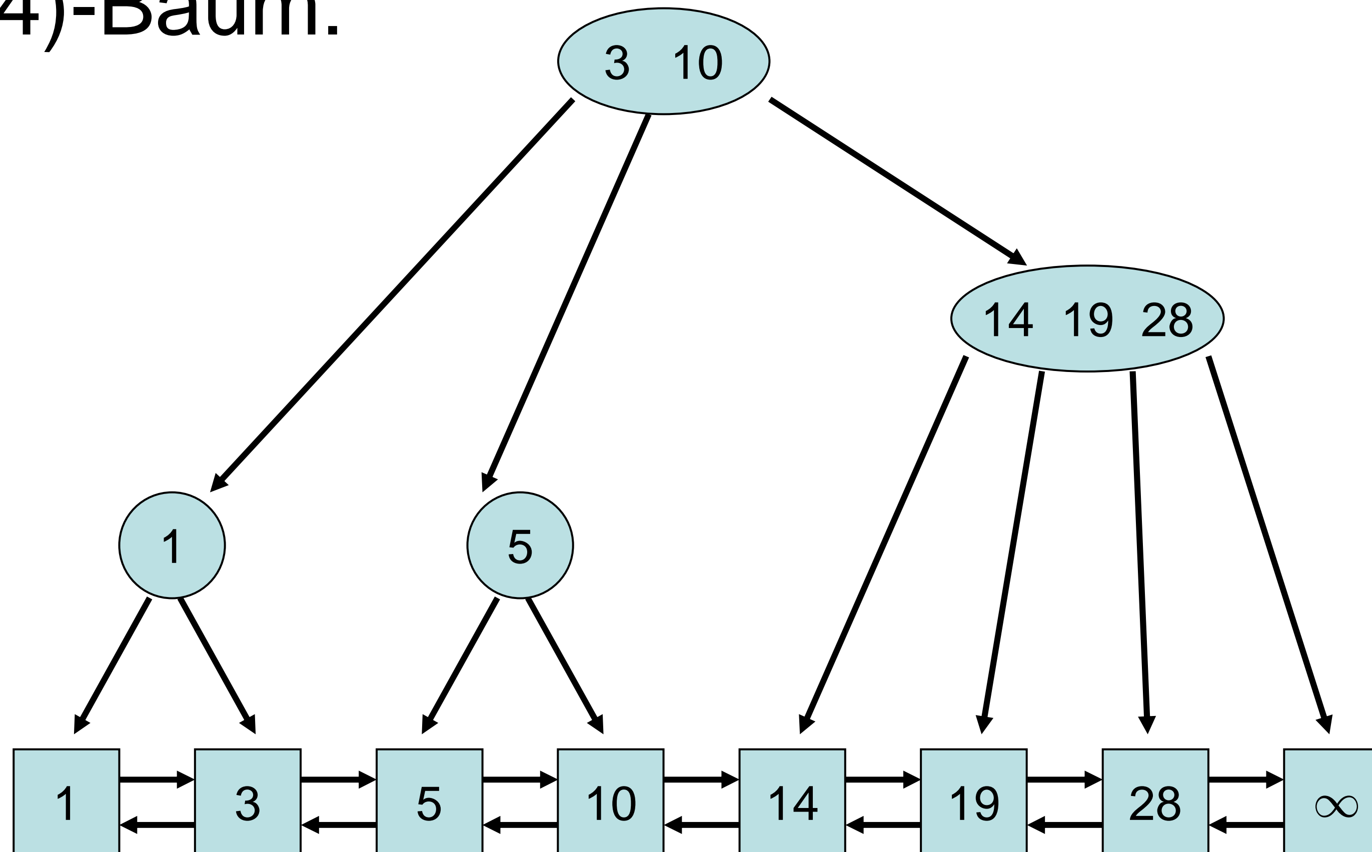
Rot-Schwarz-Baum

R-S-Baum:



Rot-Schwarz-Baum

(2,4)-Baum:



Rot-Schwarz-Baum

Lemma 3.13: Die Tiefe eines Rot-Schwarz-Baums T mit n Elementen ist O(log n).

Beweis:

Wir zeigen: log(n+1)<=t<=2log(n+1) für die Tiefe t des Rot-Schwarz-Baums.

- d: Schwarztiefe der Listenknoten
- T': (2,4)-Baum zu T
- T' hat Tiefe exakt d überall und d<=log(n+1)
- Aufgrund der internen Eigenschaft gilt t<=2d
- Außerdem ist t>=log(n+1), da Rot-Schwarz-Baum ein Binärbaum ist.

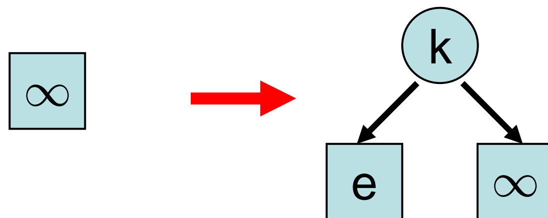
Rot-Schwarz-Baum

search(k): wie im binären Suchbaum

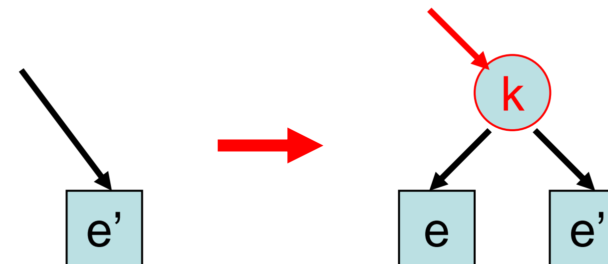
insert(e):

- führe search(k) mit k=key(e) aus
- füge e vor Nachfolger e' in Liste ein

Fall 1: Baum leer



Fall 2: Baum nicht leer



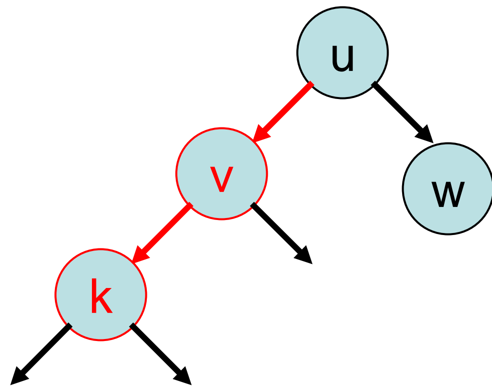
Rot-Schwarz-Baum

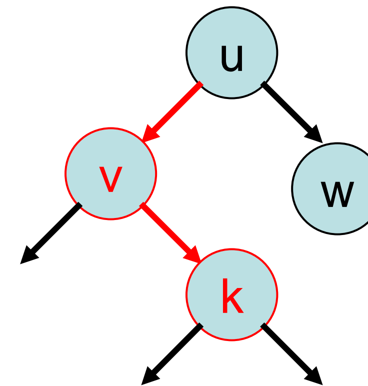
insert(e):

- führe search(k) mit k=key(e) aus
- füge e vor Nachfolger e' in Liste ein
  (bewahrt alles bis auf evtl. interne Regel)
- interne Regel verletzt (Fall 2 vorher): 2 Fälle
  - Fall 1: Vater von k in T hat schwarzen Bruder
    (Restrukturierung, aber beendet Reparatur)
  - Fall 2: Vater von k in T hat roten Bruder
    (setzt Reparatur nach oben fort, aber keine Restrukturierung)

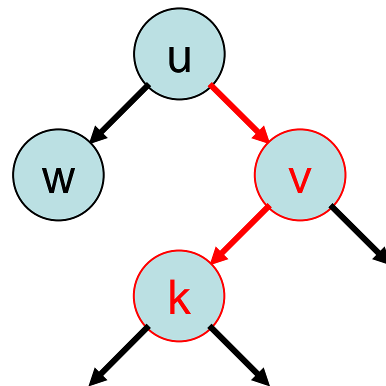
Rot-Schwarz-Baum

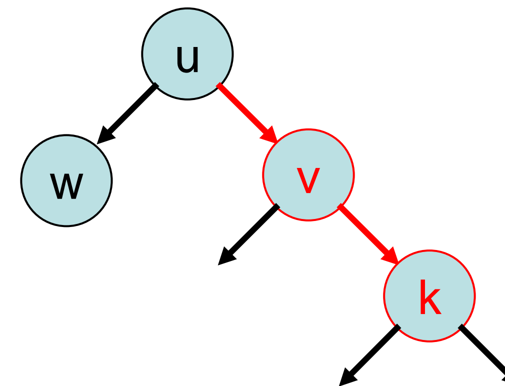
Fall 1: Vater v von k in T hat schwarzen Bruder w





Alternativen

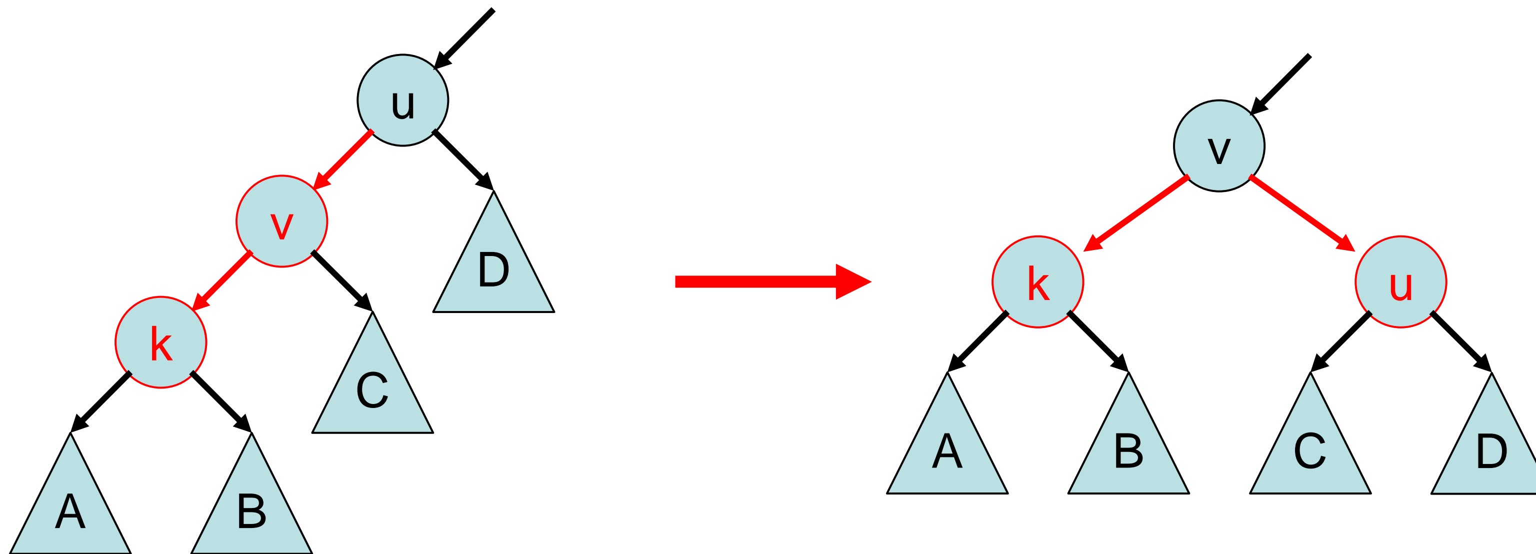




Rot-Schwarz-Baum

Fall 1: Vater v von k in T hat schwarzen Bruder w

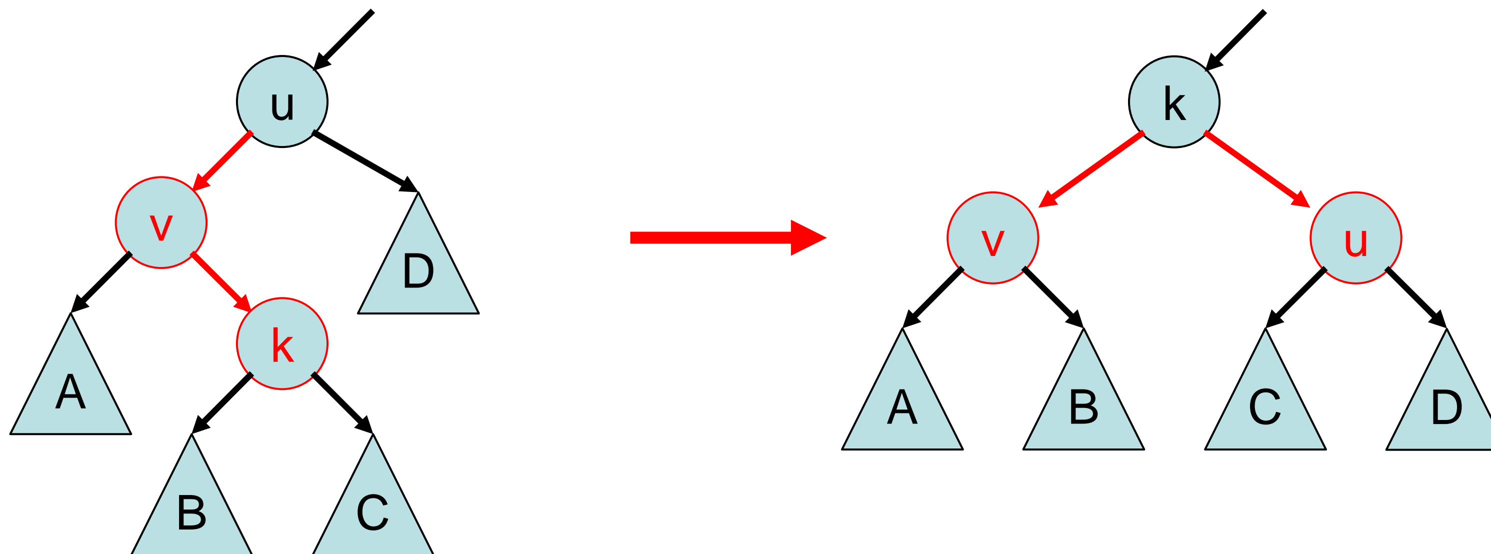
Lösung:



Rot-Schwarz-Baum

Fall 1: Vater v von k in T hat schwarzen Bruder w

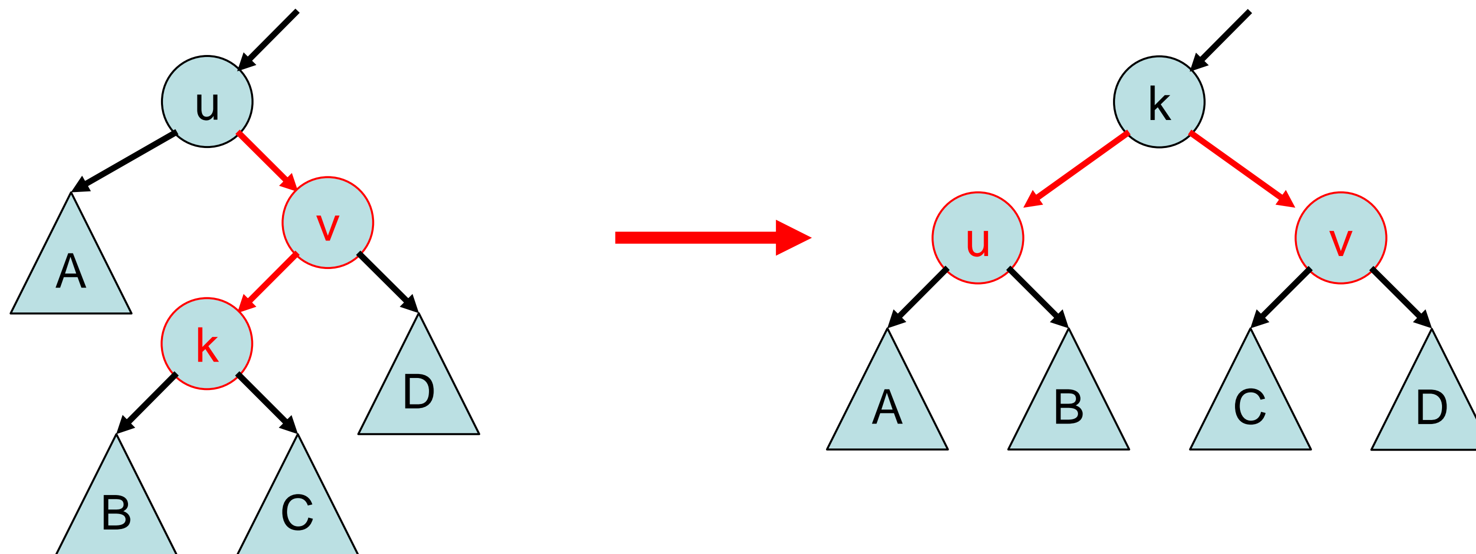
Lösung:



Rot-Schwarz-Baum

Fall 1: Vater v von k in T hat schwarzen Bruder w

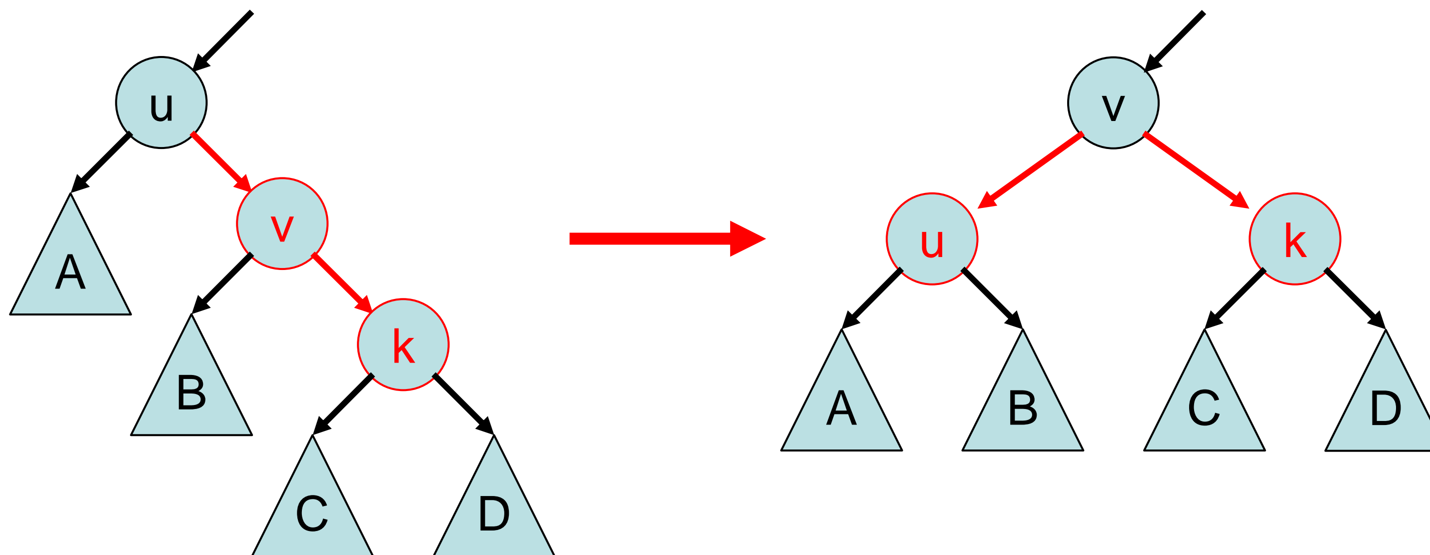
Lösung:



Rot-Schwarz-Baum

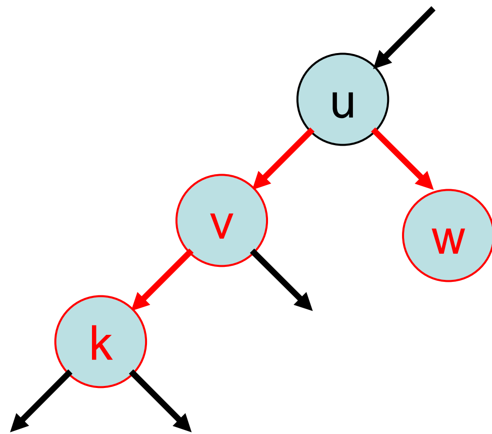
Fall 1: Vater v von k in T hat schwarzen Bruder w

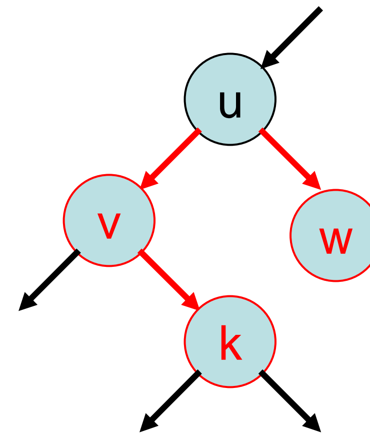
Lösung:



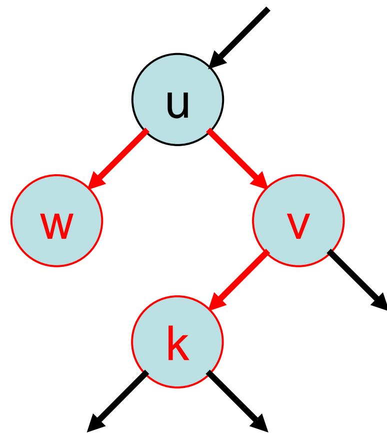
Rot-Schwarz-Baum

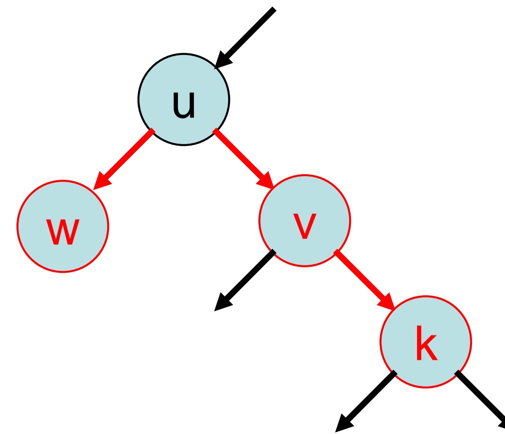
Fall 2: Vater v von k in T hat roten Bruder w





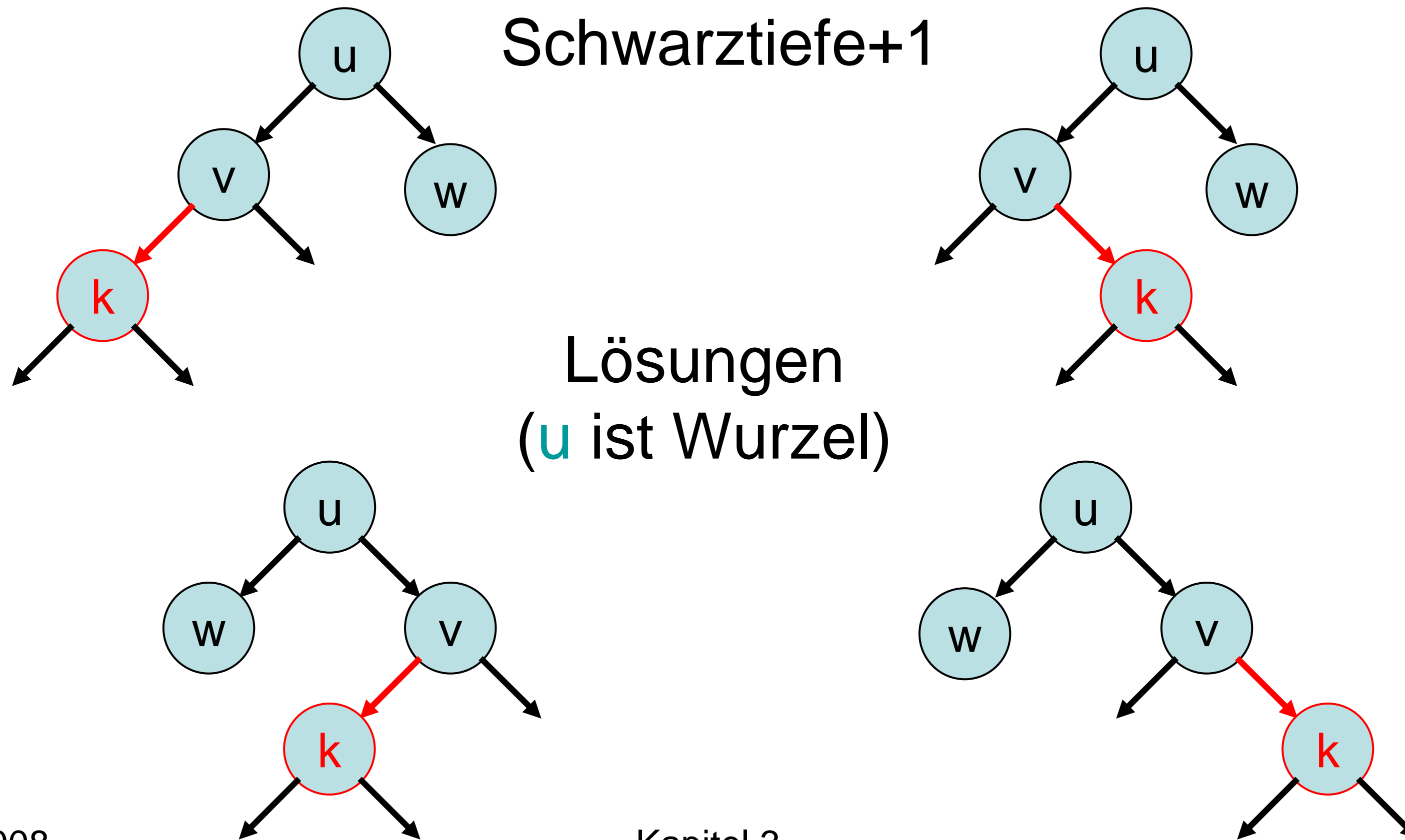
Alternativen





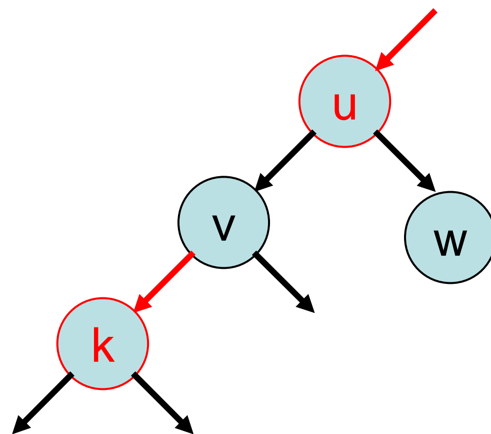
Rot-Schwarz-Baum

Fall 2: Vater v von k in T hat roten Bruder w

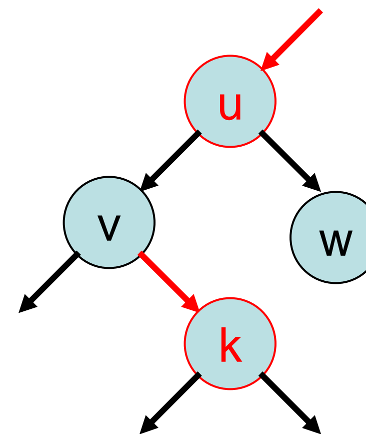


Rot-Schwarz-Baum

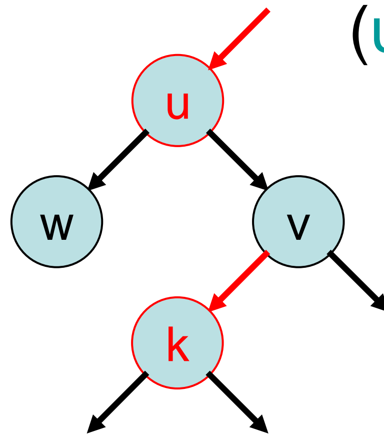
Fall 2: Vater v von k in T hat roten Bruder w

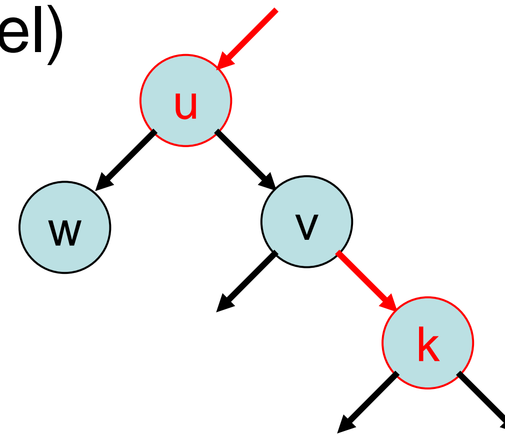


bewahrt Schwarztiefe!



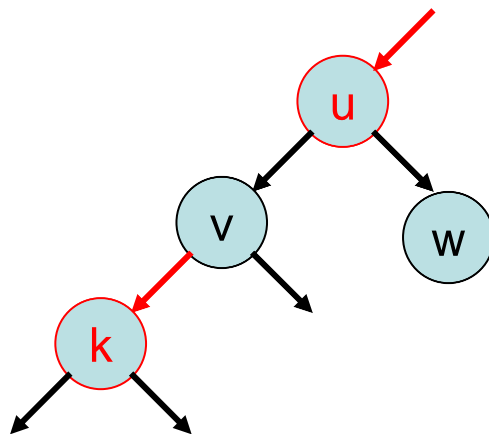
Lösungen (u keine Wurzel)



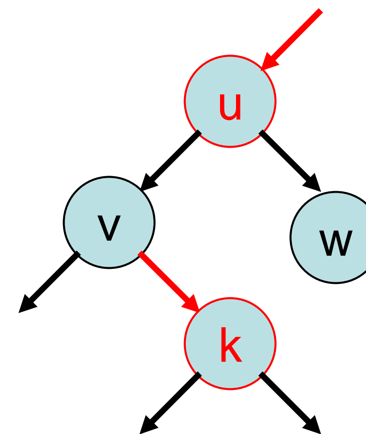


Rot-Schwarz-Baum

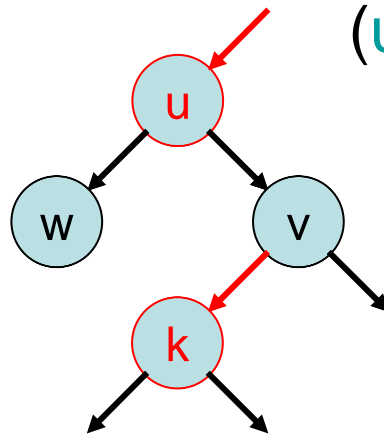
Fall 2: Vater v von k in T hat roten Bruder w

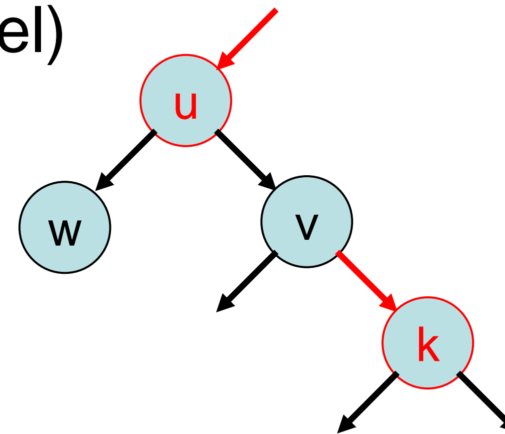


weiter mit u
wie mit k



Lösungen
(u keine Wurzel)

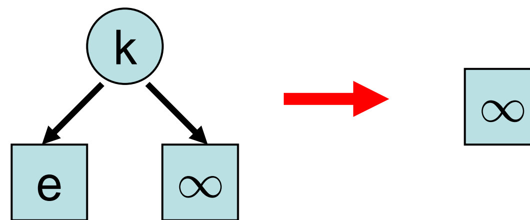




Rot-Schwarz-Baum

delete(k):

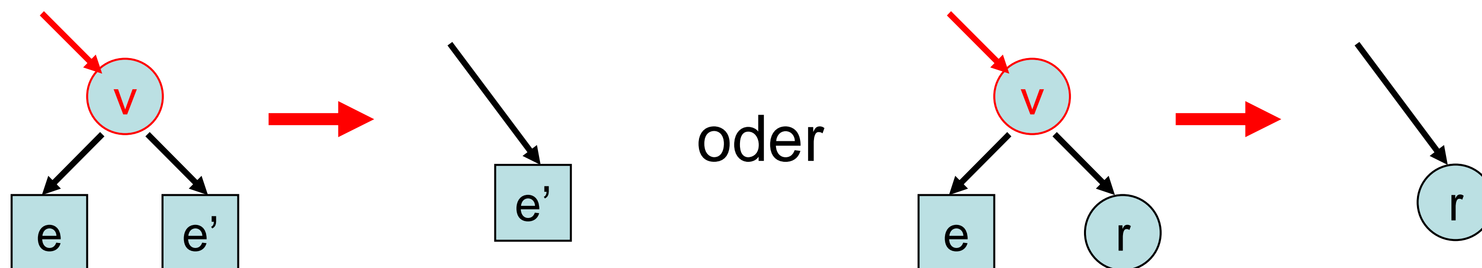
- führe search(k) auf Baum aus
- lösche Element e mit key(e)=k wie im binären Suchbaum
- Fall 1: Baum ist dann leer



Rot-Schwarz-Baum

delete(k):

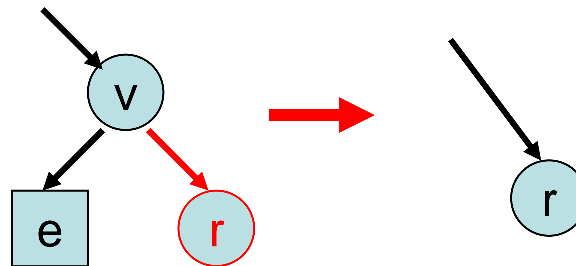
- führe search(k) auf Baum aus
- lösche Element e mit key(e)=k wie im binären Suchbaum
- Fall 2: Vater v von e ist rot (d.h. Bruder schwarz)



Rot-Schwarz-Baum

delete(k):

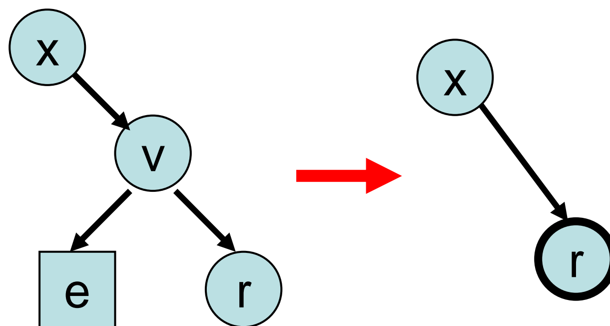
- führe search(k) auf Baum aus
- lösche Element e mit key(e)=k wie im binären Suchbaum
- Fall 3: Vater v von e ist schwarz und Bruder rot



Rot-Schwarz-Baum

delete(k):

- führe search(k) auf Baum aus
- lösche Element e mit key(e)=k wie im binären Suchbaum
- Fall 4: Vater v von e und Bruder r sind schwarz



Tiefenregel verletzt!
r heißt dann doppelt schwarz

Rot-Schwarz-Baum

delete(k):

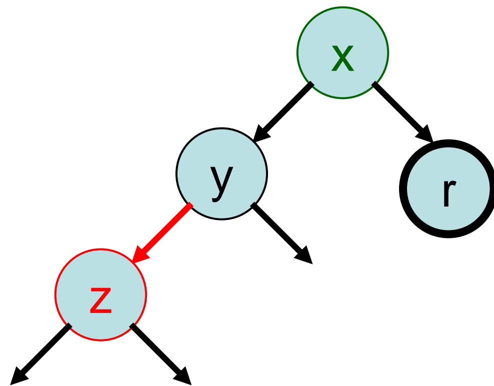
- führe search(k) auf Baum aus
- lösche Element e mit key(e)=k wie im binären Suchbaum
- falls Vater v von e und Bruder r sind schwarz, dann 3 weitere Fälle:
  - Fall 1: Bruder y von r ist schwarz und hat rotes Kind
  - Fall 2: Bruder y von r ist schwarz und beide Kinder von y sind schwarz (evtl. weiter, aber keine Restrukt.)
  - Fall 3: Bruder y von r ist rot

Rot-Schwarz-Baum

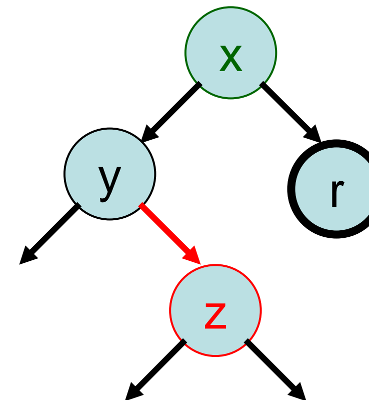
Fall 1: Bruder y von r ist schwarz, hat rotes Kind z

O.B.d.A. sei r rechtes Kind von x (links: analog)

Alternativen: (x: beliebig gefärbt)



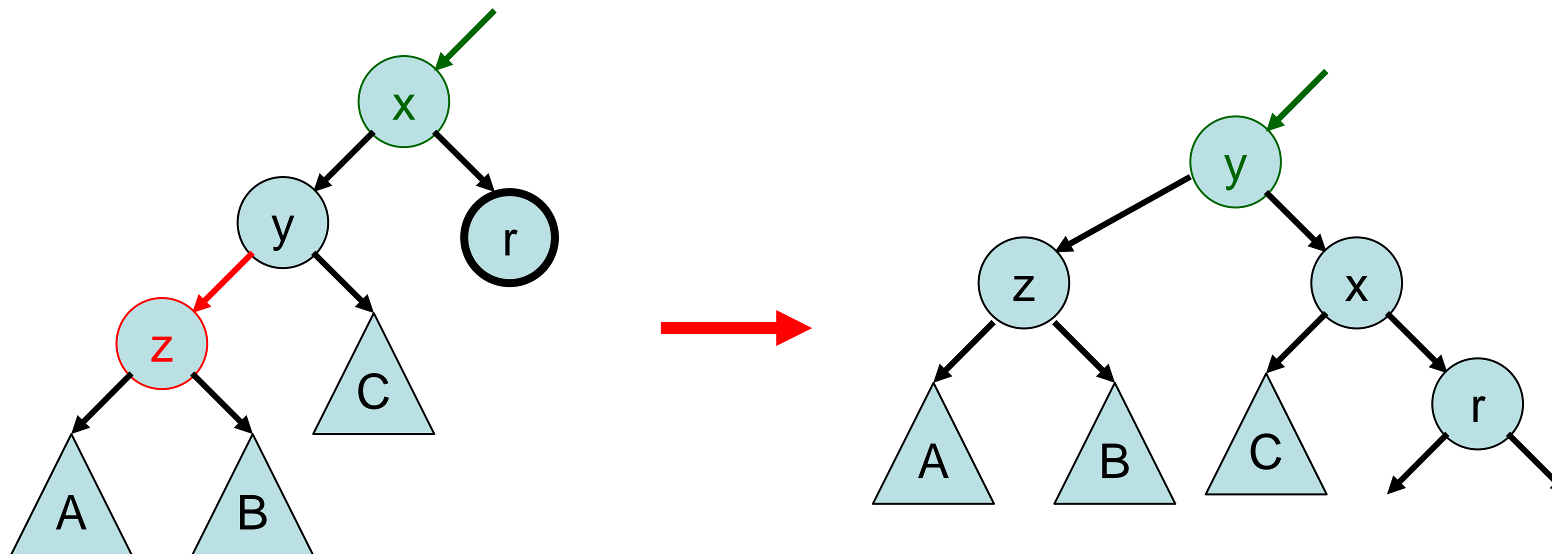
oder



Rot-Schwarz-Baum

Fall 1: Bruder y von r ist schwarz, hat rotes Kind z

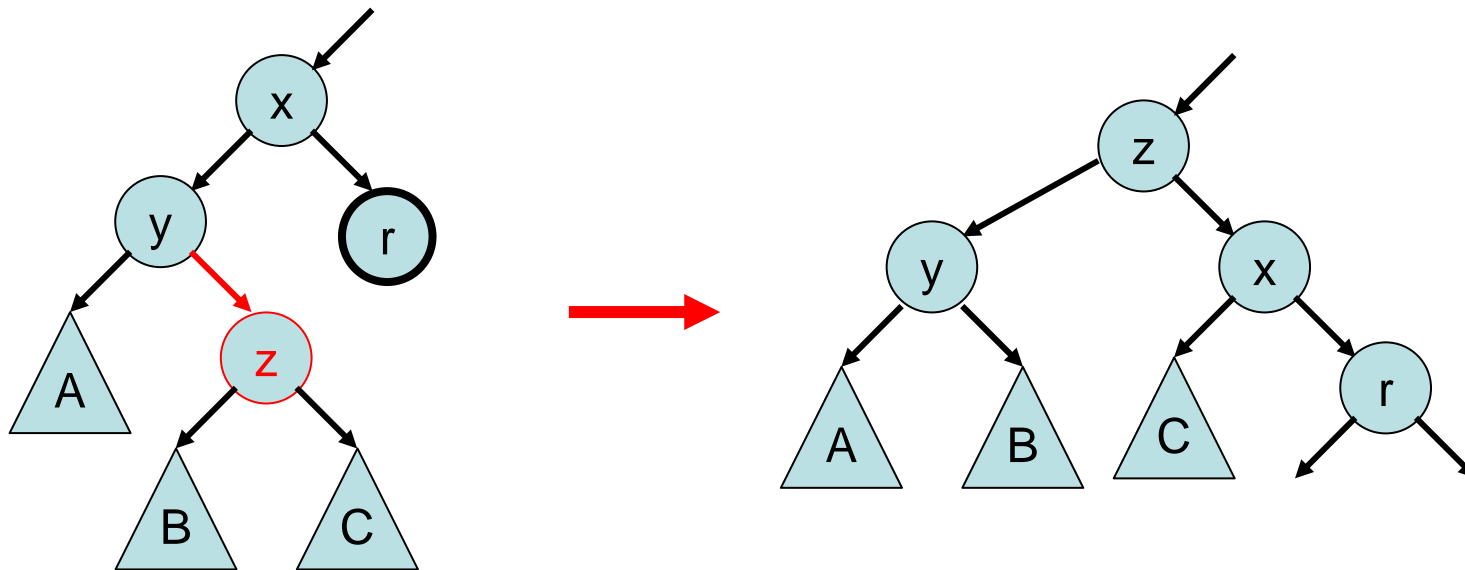
O.B.d.A. sei r rechtes Kind von x (links: analog)



Rot-Schwarz-Baum

Fall 1: Bruder y von r ist schwarz, hat rotes Kind z

O.B.d.A. sei r rechtes Kind von x (links: analog)

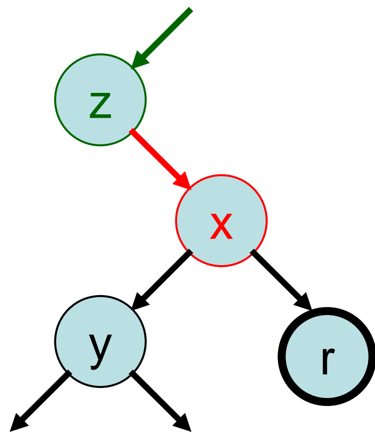


Rot-Schwarz-Baum

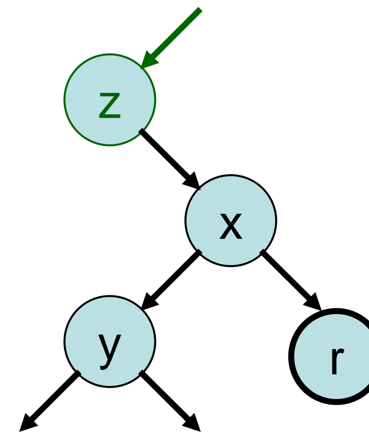
Fall 2: Bruder y von r ist schwarz und beide Kinder von y sind schwarz

O.B.d.A. sei r rechtes Kind von x (links: analog)

Alternativen: (z beliebig gefärbt)



oder

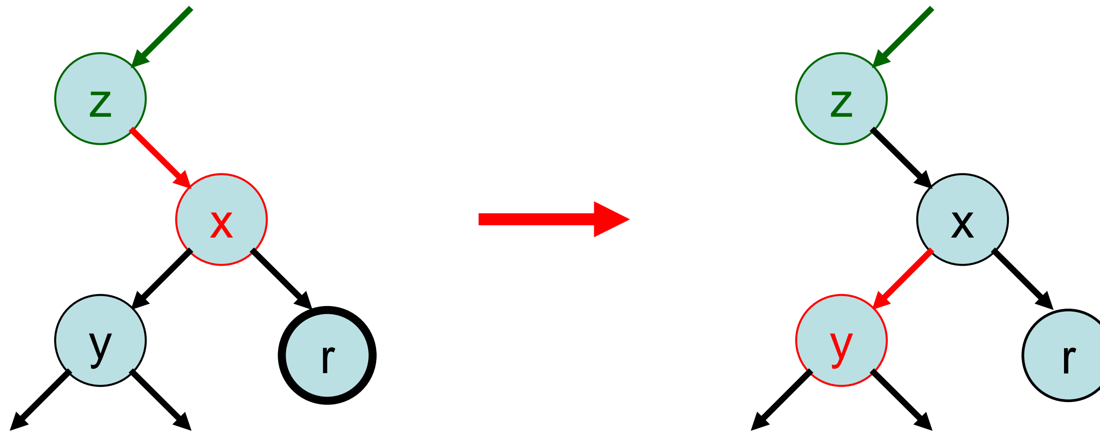


Rot-Schwarz-Baum

Fall 2: Bruder y von r ist schwarz und beide Kinder von y sind schwarz

O.B.d.A. sei r rechtes Kind von x (links: analog)

2a)

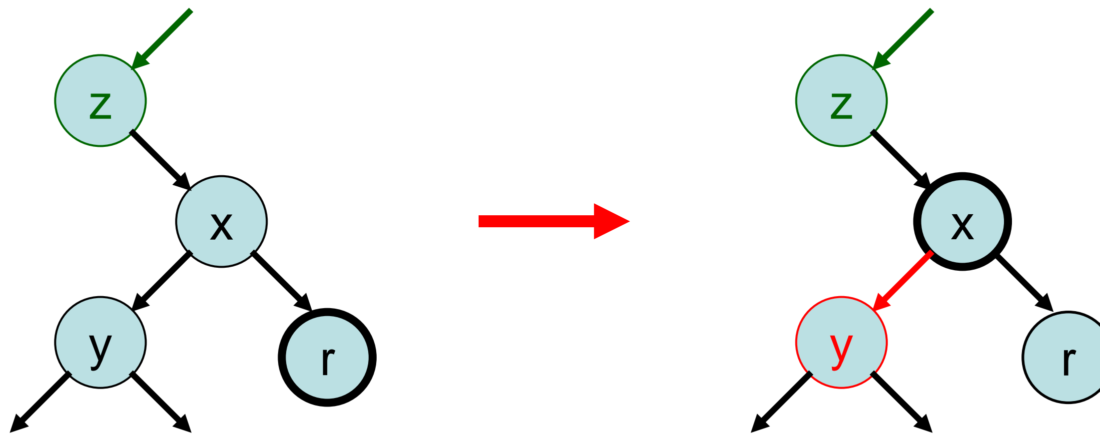


Rot-Schwarz-Baum

Fall 2: Bruder y von r ist schwarz und beide Kinder von y sind schwarz

O.B.d.A. sei r rechtes Kind von x (links: analog)

2b)



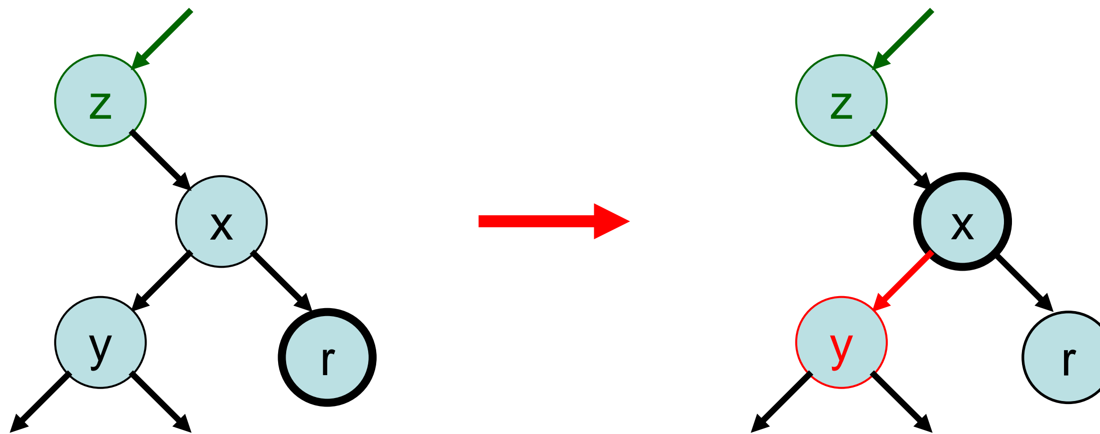
x ist Wurzel: fertig (Schwarztiefe-1)

Rot-Schwarz-Baum

Fall 2: Bruder y von r ist schwarz und beide Kinder von y sind schwarz

O.B.d.A. sei r rechtes Kind von x (links: analog)

2b)

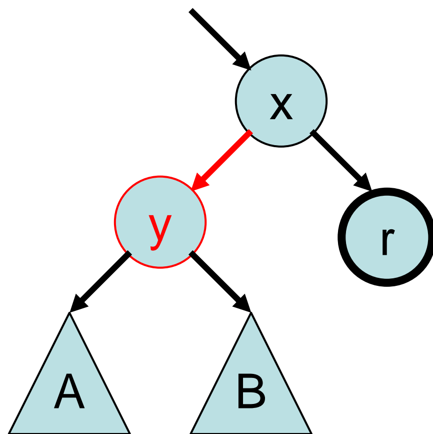


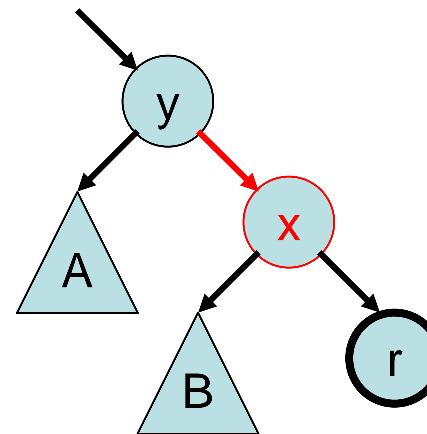
x keine Wurzel: weiter wie mit r

Rot-Schwarz-Baum

Fall 3: Bruder y von r ist rot

O.B.d.A. sei r rechtes Kind von x (links: analog)





Fall 1 oder 2a
$\rightarrow$ terminiert dann

Rot-Schwarz-Baum

Laufzeiten der Operationen:

- search(k): $O(\log n)$
- insert(e): $O(\log n)$
- delete(k): $O(\log n)$

Restrukturierungen:

- insert(e): max. 1
- delete(k): max. 2

Binärbaum

Problem: Binärbaum kann entarten!

Lösungen:

- Splay-Baum
  (sehr effektive Heuristik)
- Treaps
  (mit hoher Wkeit gut balanciert)
- (a,b)-Baum
  (garantiert gut balanciert)
- Rot-Schwarz-Baum
  (konstanter Reorganisationsaufwand)
- Gewichtsbalancierter Baum
  (kompakt einbettbar in Feld)

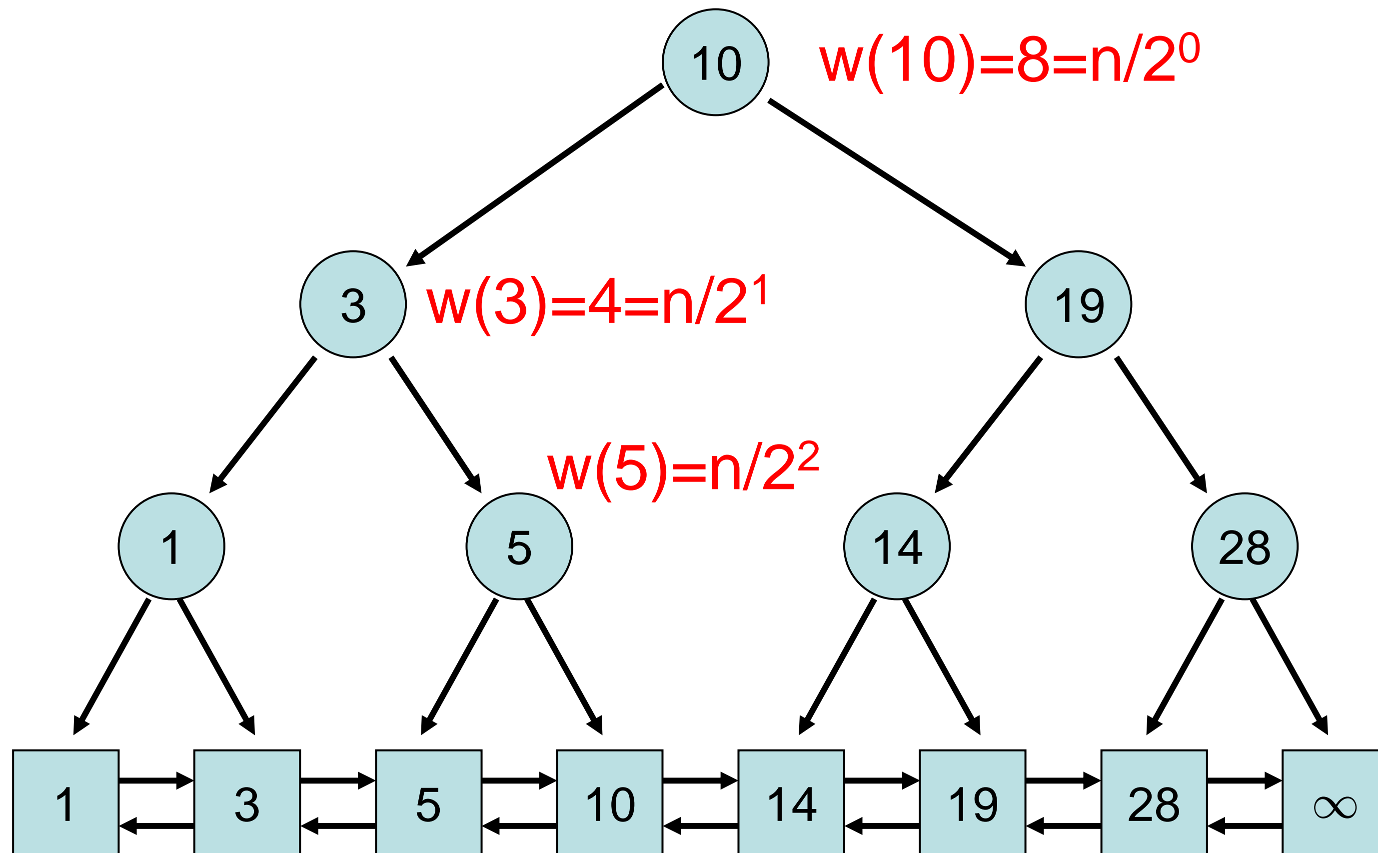
Gewichtsbalancierter Baum

- n: Anzahl der Elemente im Baum (gerundet auf einen Wert $(1+\varepsilon)^k$ für ein $k \in \mathbb{N}$ und eine feste Konstante $0<\varepsilon<1/2$)
- $w(v)$: Anzahl Listenknoten unter Baum $T(v)$ von v

- Suchbaum-Regel: (s.o.)
- Grad-Regel:
  Alle Baumknoten haben zwei Kinder (sofern #Elemente >1)
- Schlüssel-Regel:
  Für jedes Element e in der Liste gibt es genau einen Baumknoten v mit $key(v)=key(e)$.
- Gewichts-Regel:
  Für jeden Knoten v der Tiefe d gilt

$$w(v) \in [(1-\varepsilon/\log n)^d(1-\varepsilon)\, n/2^d,\ (1+\varepsilon/\log n)^d(1+\varepsilon)\, n/2^d]$$

Gewichtsbalancierter Baum



Gewichtsbalancierter Baum

Gewichtsregel impliziert Tiefe max. $\log n + O(1)$.

search(k): wie im binären Suchbaum

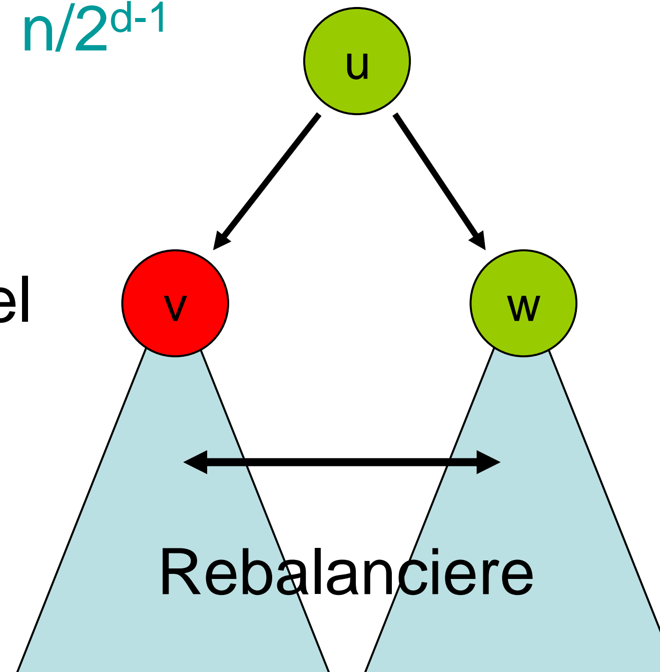
insert(e):

- füge e ein wie im binären Suchbaum
- Gewichtsregel nirgendwo verletzt: fertig
- sonst sei v der höchste Knoten im Baum mit verletzter Gewichtsregel

Gewichtsbalancierter Baum

max. $(1+\varepsilon/\log n)^{d-1}(1+\varepsilon)\ n/2^{d-1}$
Blätter unter u

v keine Wurzel

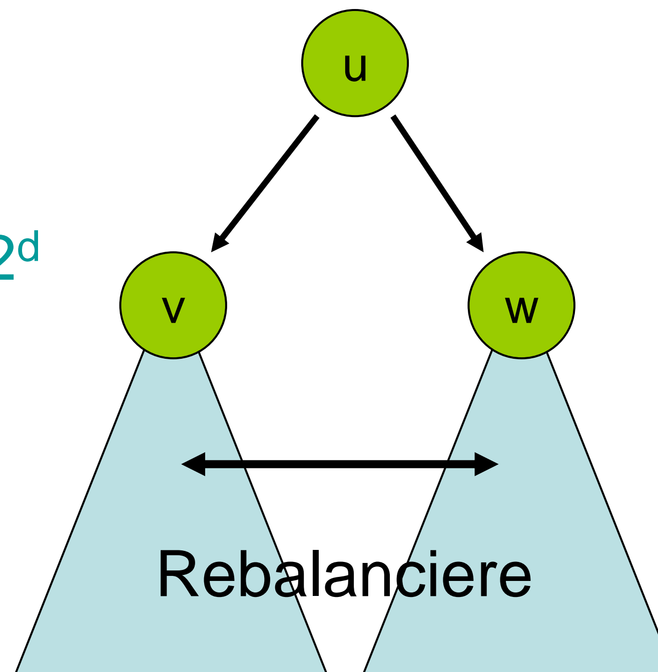


Aufwand: $\Theta(n/2^d)$

Gewichtsbalancierter Baum

max. $(1+\varepsilon/\log n)^{d-1}(1+\varepsilon)\ n/2^d$
Blätter unter v und w

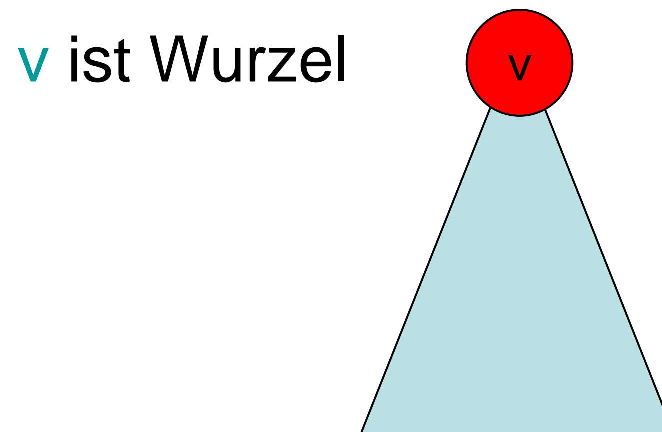
Grenze für v und w:
$(1+\varepsilon/\log n)^{d}(1+\varepsilon)\ n/2^d$



mind. $(\varepsilon/\log n)\cdot n/2^d$ inserts, bis v wieder übergewichtig

Gewichtsbalancierter Baum

v ist Wurzel



Erhöhe n um $(1+\varepsilon)$-Faktor und rebalanciere kompletten Baum nach neuem n (Aufwand: $\Theta(n)$).

- Obere Grenze für altes n: $(1+\varepsilon)(1+\varepsilon)^k$ für ein $k \in \mathbb{N}$
- Untere Grenze für neues n: $(1-\varepsilon)(1+\varepsilon)^{k+1}$
- Differenz: $\varepsilon\ (1+\varepsilon)^{k+1} = \varepsilon\ n_{neu}$ ← viele deletes notwendig!

Gewichtsbalancierter Baum

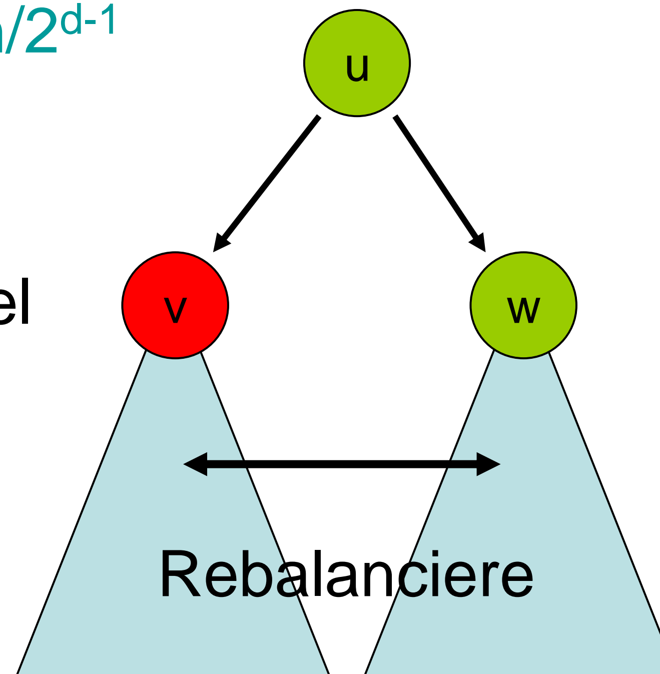
delete(k):

- führe zunächst delete(k) durch wie im binären Suchbaum
- Gewichtsregel nirgendwo verletzt: fertig
- sonst sei v der höchste Knoten im Baum mit verletzter Gewichtsregel

Gewichtsbalancierter Baum

min. $(1-\varepsilon/\log n)^{d-1}(1-\varepsilon)\ n/2^{d-1}$
Blätter unter u

v keine Wurzel

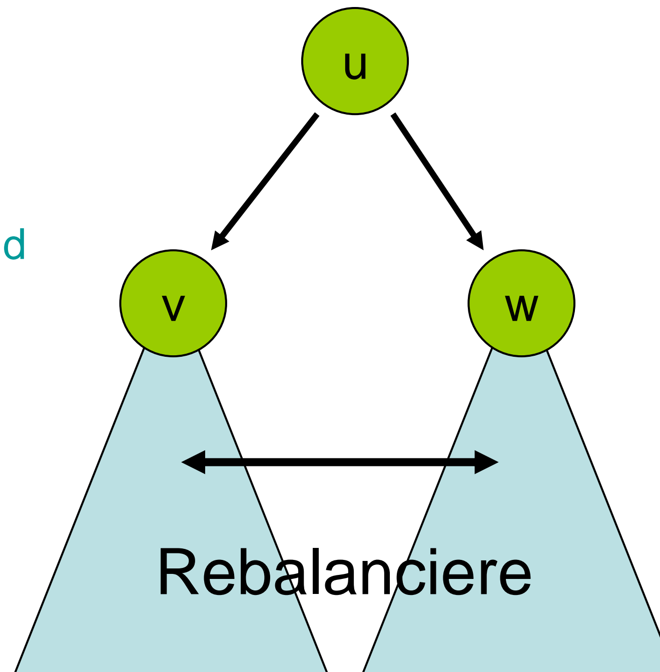


Aufwand: $\Theta(n/2^d)$

Gewichtsbalancierter Baum

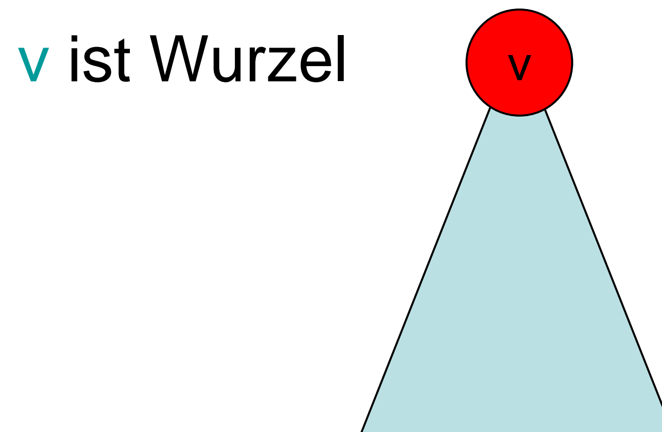
mind. $(1-\varepsilon/\log n)^{d-1}(1-\varepsilon)\ n/2^d$ Blätter unter v und w

Grenze für v und w:
$(1-\varepsilon/\log n)^{d}(1-\varepsilon)\ n/2^d$



mind. $(\varepsilon/\log n)(1-\varepsilon)\cdot n/2^d$ deletes, bis v wieder untergewichtig

Gewichtsbalancierter Baum



Erniedrige n um $(1+\varepsilon)$-Faktor und rebalanciere kompletten Baum nach neuem n.

- Untere Grenze für altes n: $(1-\varepsilon)(1+\varepsilon)^k$ für ein $k \in \mathbb{N}$
- Obere Grenze für neues n: $(1+\varepsilon)(1+\varepsilon)^{k-1}$
- Differenz: $\varepsilon\,(1+\varepsilon)^k = \varepsilon\, n_{alt}$ ← viele inserts notwendig!

Gewichtsbalancierter Baum

Es gilt:

- Für jeden Knoten v der Tiefe d nur Reorganisation alle $\Theta((\varepsilon/\log n)\ n/2^d)$ viele insert/delete Operationen.
- Knoten v hat zu jeder Zeit $\Theta(n/2^d)$ Knoten unter sich. Das ist auch Reorganisationsaufwand.
- Pro insert/delete Operation maximal $O(\log n)$ Knoten betroffen (d.h. sie erfahren Gewichtsveränderung).

Amortisierter Aufwand pro insert/delete Operation:

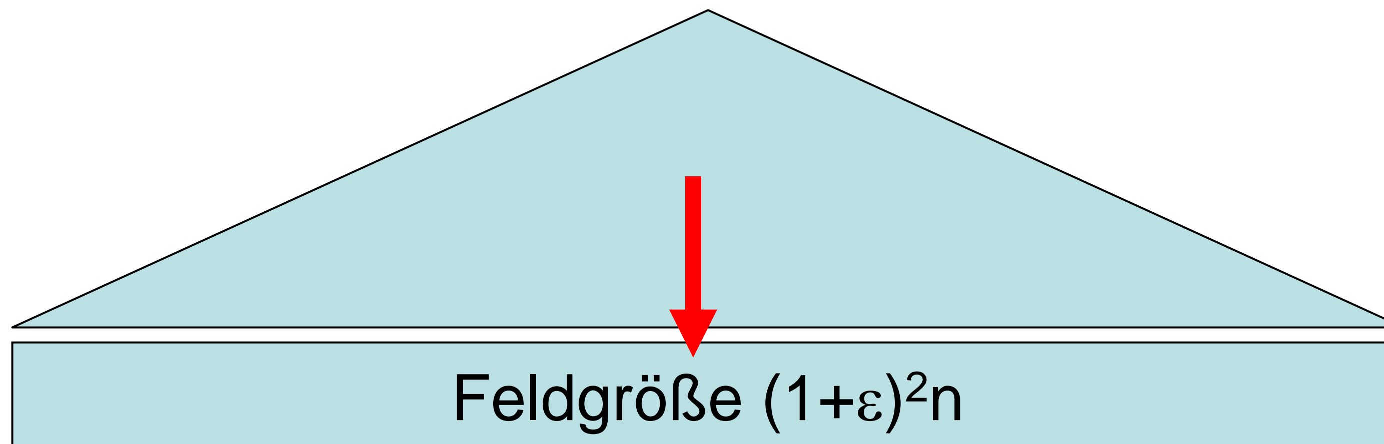
$$O\left(\sum_d \frac{(n/2^d)}{(\varepsilon/\log n)\ (n/2^d)} \right) = O(\log^2 n / \varepsilon)$$

Gewichtsbalancierter Baum

Vorteil: Baum kann in Feld eingebettet werden

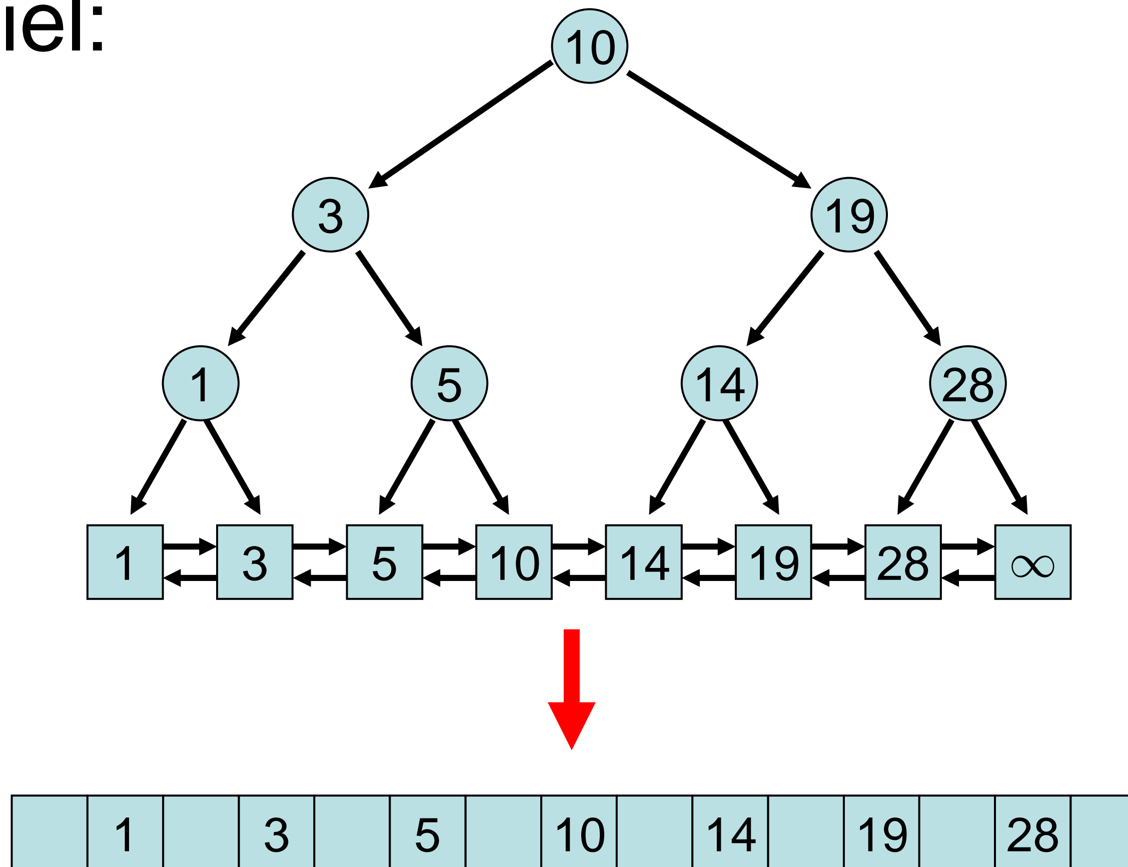
Regel: Gib Knoten v der Tiefe d Feld der Größe $(1+\varepsilon)^2 n/2^d$, v selbst in der Mitte des Feldes

Reicht, da $(1+\varepsilon/\log n)^d(1+\varepsilon)\, n/2^d \le (1+\varepsilon)^2 n/2^d$



Gewichtsbalancierter Baum

Beispiel:



Gewichtsbalancierter Baum

Wir können also auch effizient sortierte Felder verwalten.

Anwendungen:

- Bereichsanfragen mit wenig I/O Operationen
- Inhalt editierter Files bleibt aufeinanderfolgend, d.h. keine Fragmentierung

Ausblick

Weiter mit Wörterbüchern und Hashing