Grundlagen der Algorithmen und Datenstrukturen

Kapitel 7

Christian Scheideler + Helmut Seidl

SS 2009

Wörterbuch

S: Menge von Elementen

Jedes Element e identifiziert über e.key().

Operationen:

- S.insert(Element e): $S = S \cup \{e\}$;
- S.removeKey(Key k): $S = S \setminus \{e\}$; wobei e das Element ist mit e.key()==k
- S.find(Key k): Falls es ein $e \in S$ gibt mit e.key()==k, dann gib e aus, sonst gib $\perp$ aus

Suchstruktur

S: Menge von Elementen

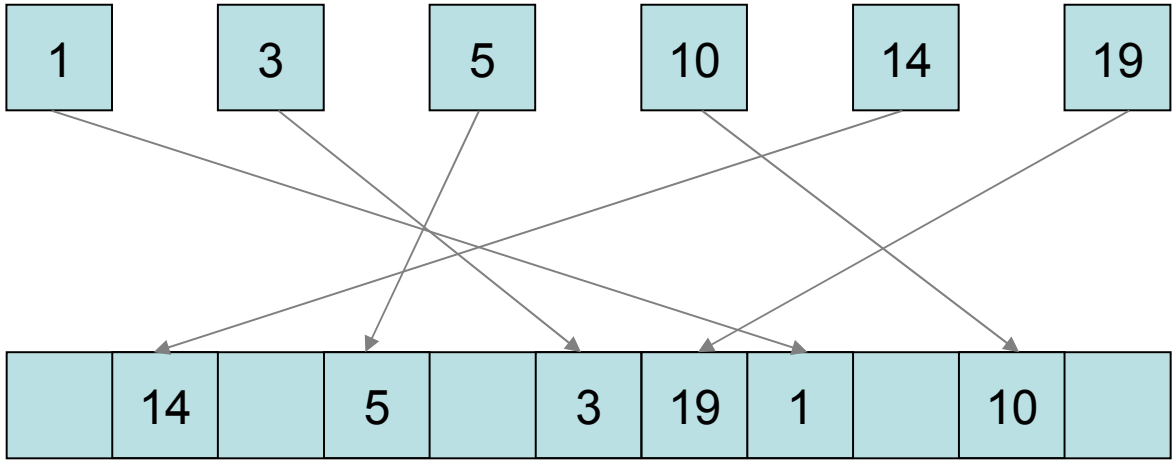
Jedes Element e identifiziert über key(e).

Operationen:

- S.insert(Element e): $S = S \cup \{e\}$;
- S.removeKey(Key k): $S = S \setminus \{e\}$; wobei e das Element ist mit e.key()==k
- S.locate(Key k): gib $e \in S$ aus mit minimalem e.key() so dass e.key() ≥ k

Wörterbuch vs. Suchstruktur

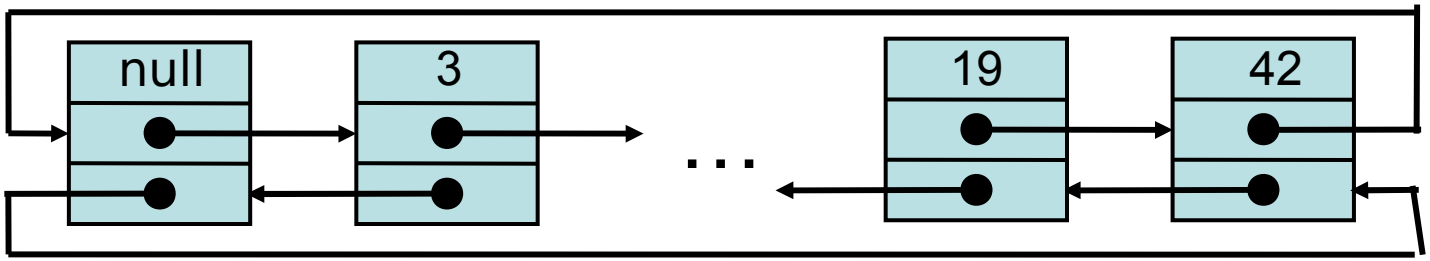
- Wörterbuch effizient über Hashing realisierbar (insert, remove und find kosten amortisiert / worst case O(1) Zeit)



- Hashing zerstört Ordnung und erlaubt daher keine effiziente locate-Operation

Suchstruktur

Erste Lösung: sortierte Liste (mit Wächter)

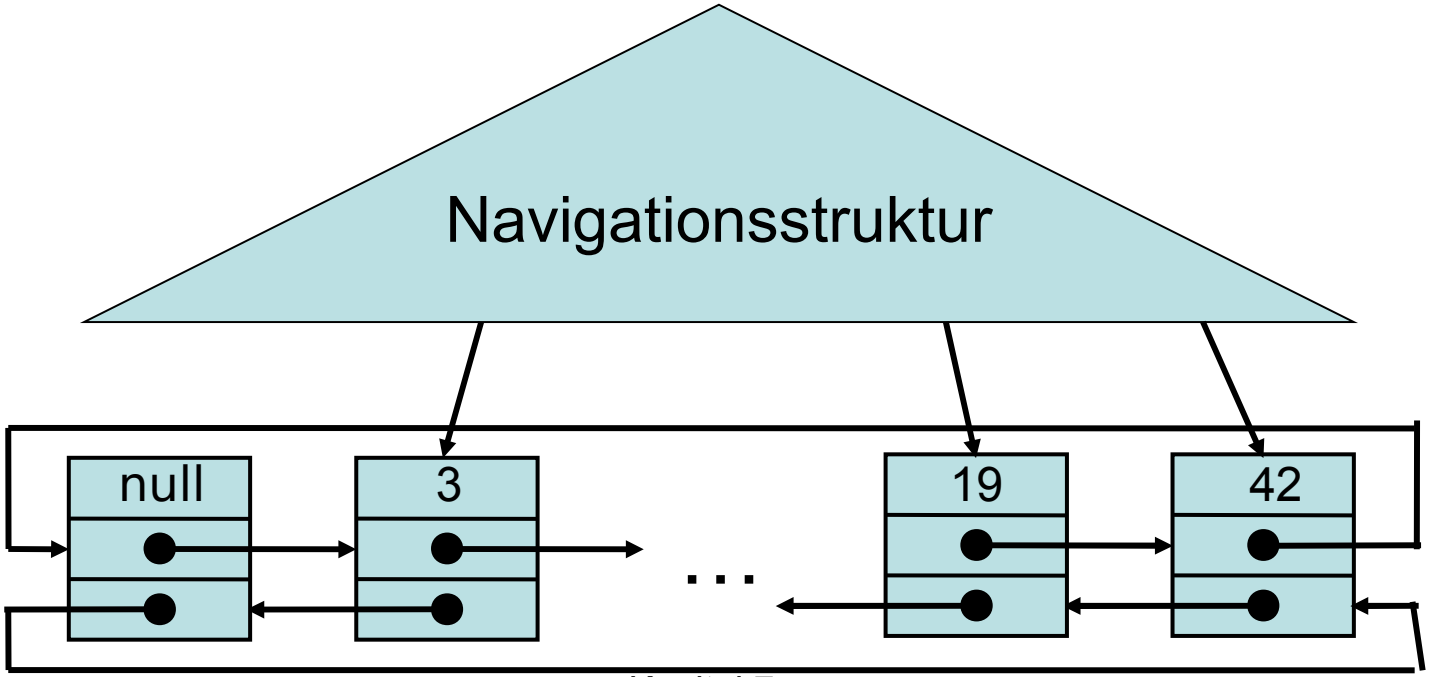


Problem: insert, remove, locate kosten im worst case $\Theta(n)$ Zeit

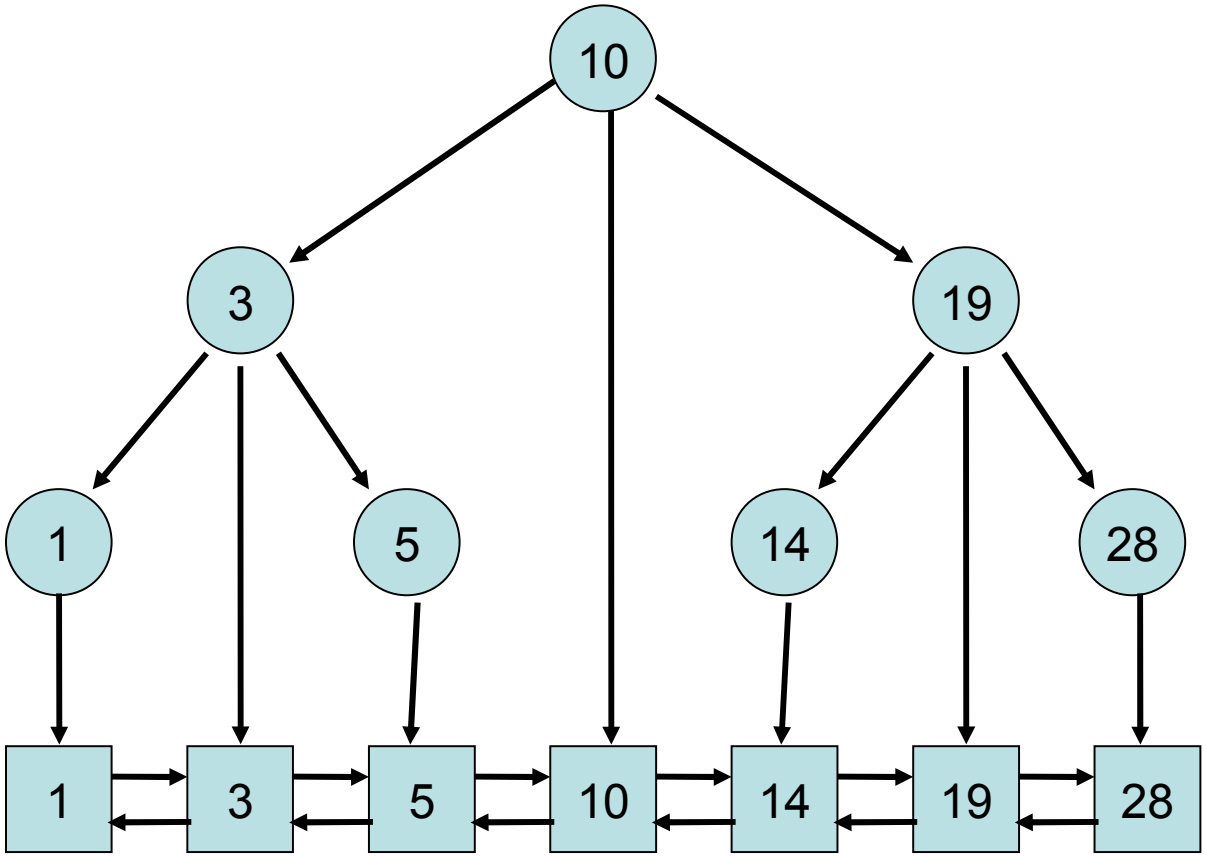
Einsicht: Wenn locate effizient zu implementieren wäre, dann auch alle anderen Operationen

Suchstruktur

Idee: füge Navigationsstruktur hinzu, die locate effizient macht

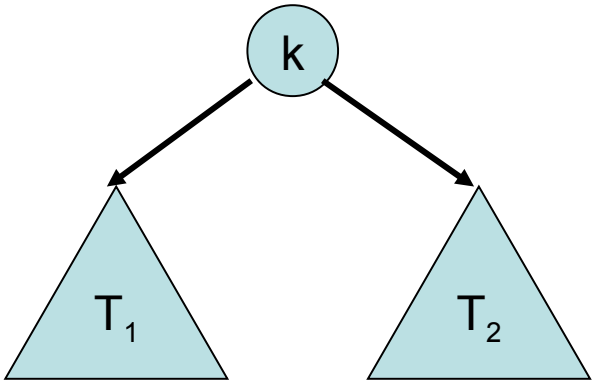


Binärer Suchbaum (ideal)



Binärer Suchbaum

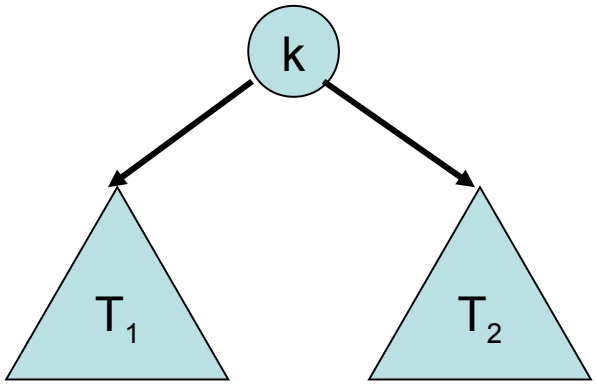
Suchbaum-Regel:



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

Damit locate Operation einfach zu implementieren.

locate(k) Operation



Für alle Schlüssel k´ in T_1 und k´´ in T_2: k´< k < k´´

Locate-Strategie:

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

Binärer Suchbaum

Formal: für einen Baumknoten v sei

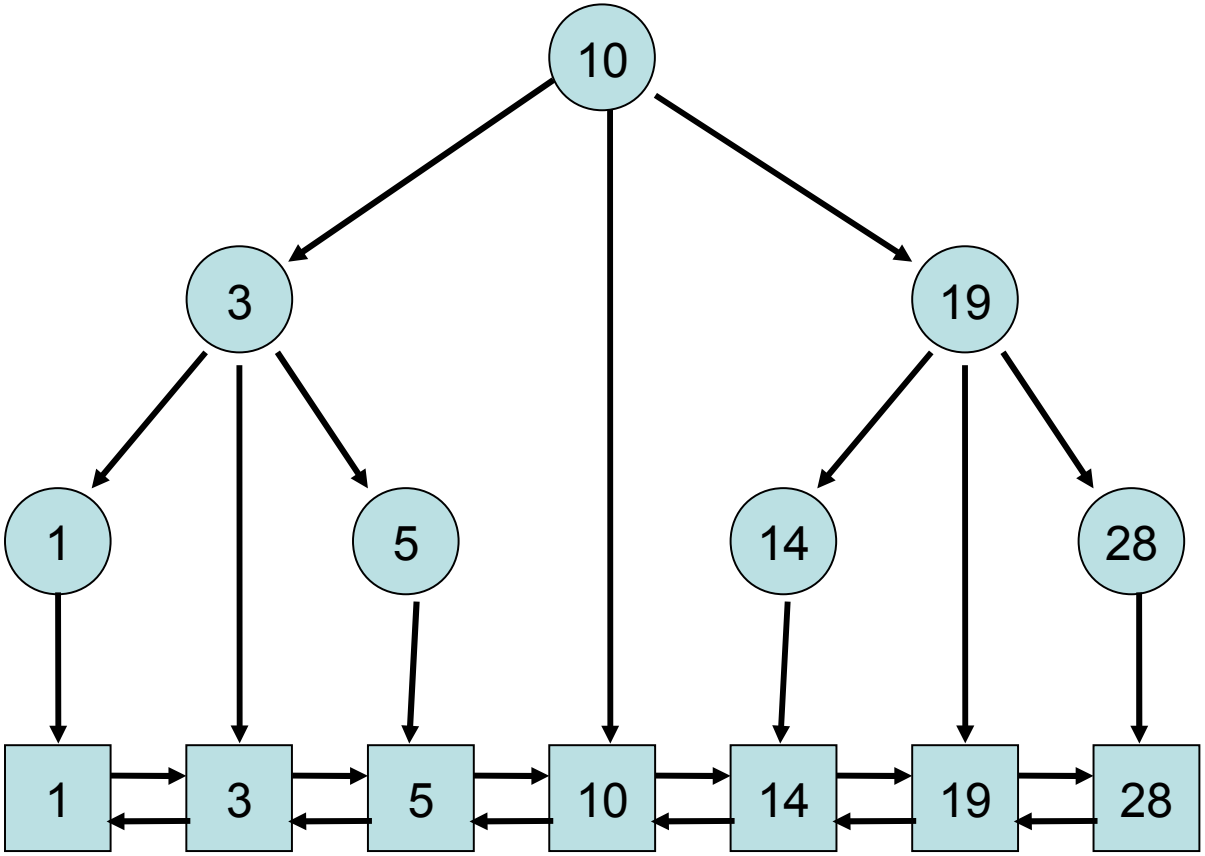
- key(v) der Schlüssel in v
- d(v) die Anzahl Kinder von v

- Suchbaum-Invariante: (s.o.)

- Grad-Invariante:
  Alle Baumknoten haben max. zwei Kinder

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

locate(9)

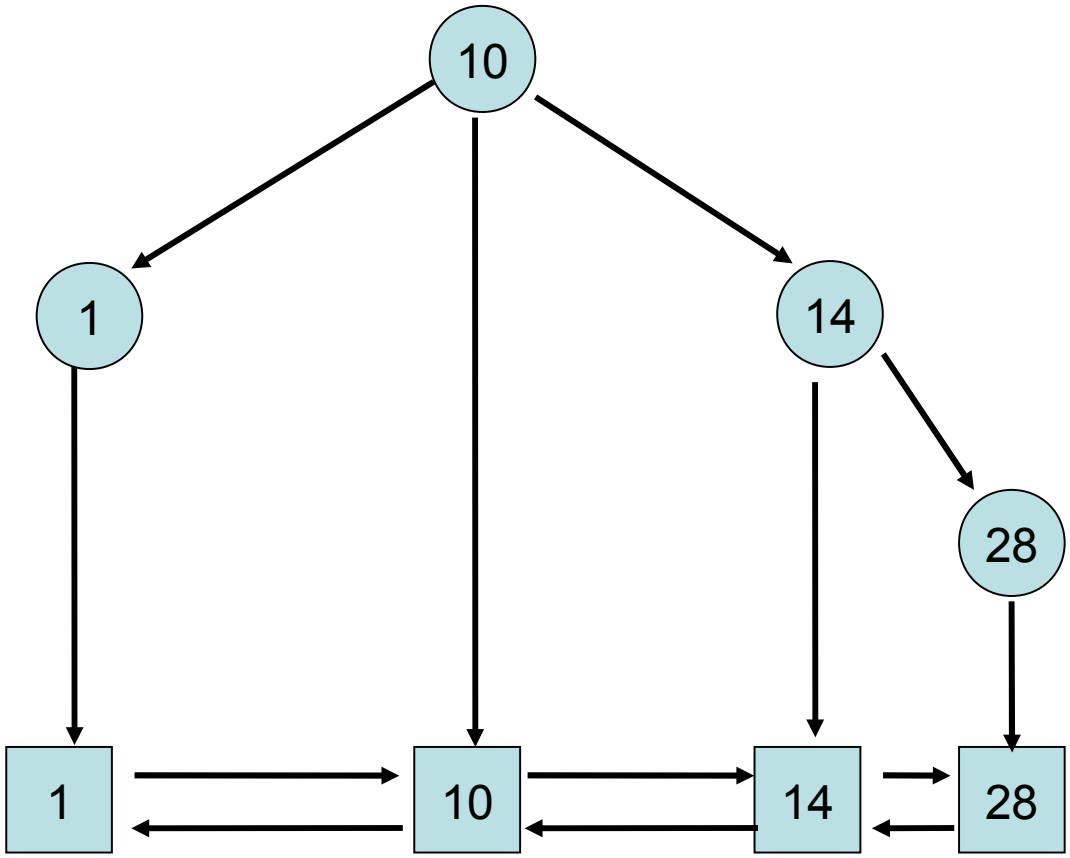


Insert und Remove Operationen

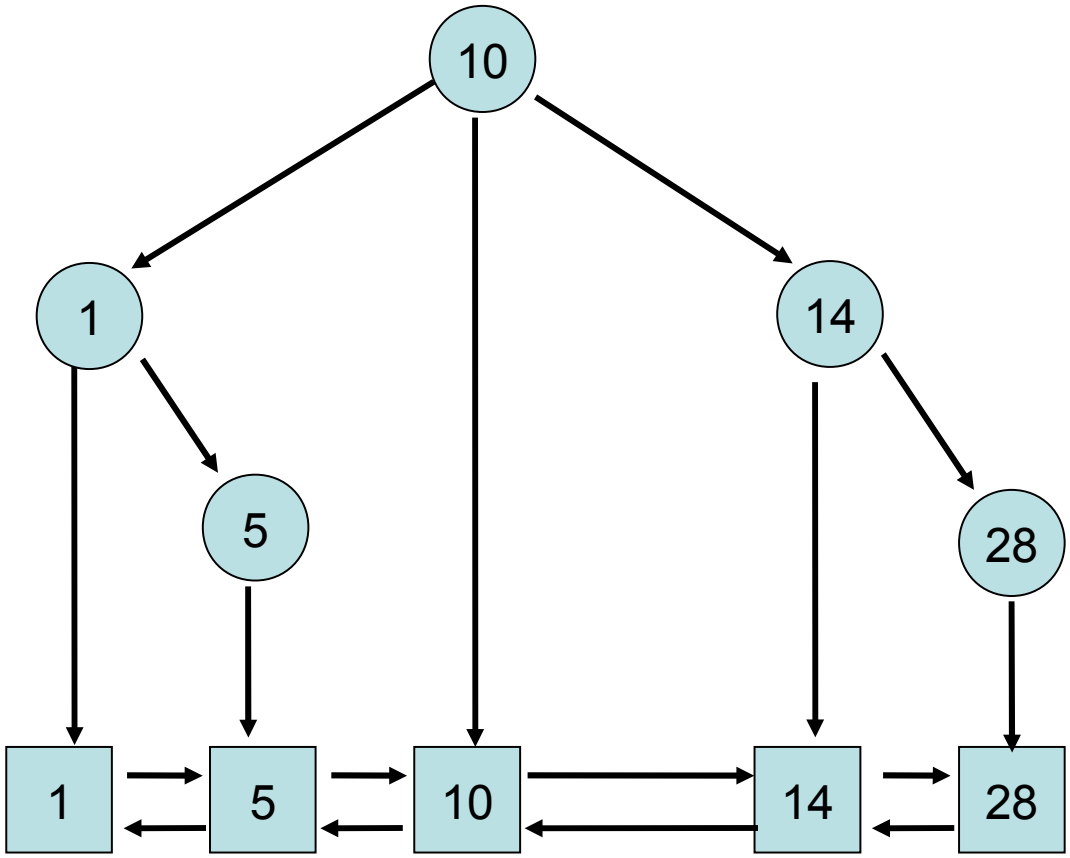
Strategie:

- insert(e):
  Suche einen Knoten entweder mit e.key() < key und linkem Nachfolger null oder mit e.key() > key und rechtem Nachfolger null. Füge für e ein neues Suchbaumblatt ein, so dass Suchbaum-Regel erfüllt ist.
- removeKey(k):
  Suche einen Knoten mit key == k. Lösche den Inhalt des Knoten. Hat er keine Nachfolger, lösche den ganzen Knoten. Hat er einen Nachfolger, verkürze. Hat er zwei Nachfolger, überschreibe ihn mit dem größten Knoten des linken Nachfolgers.

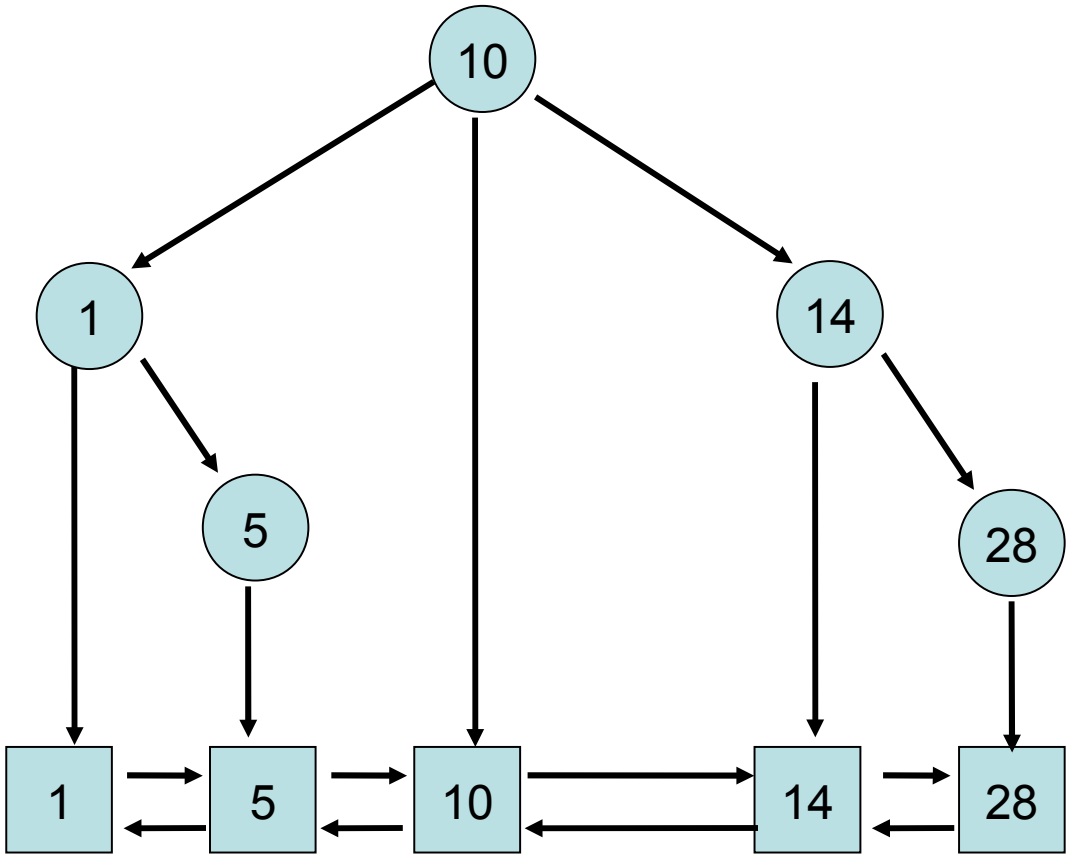
Insert(5)



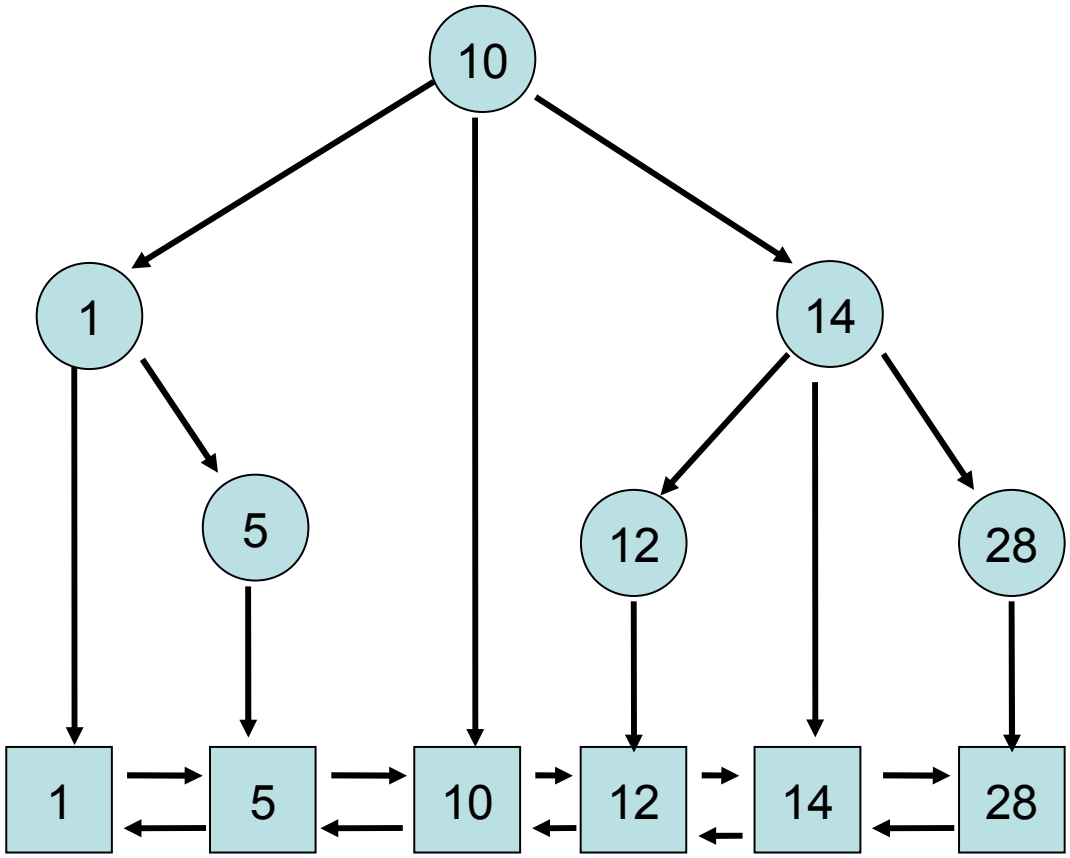
Insert(5)



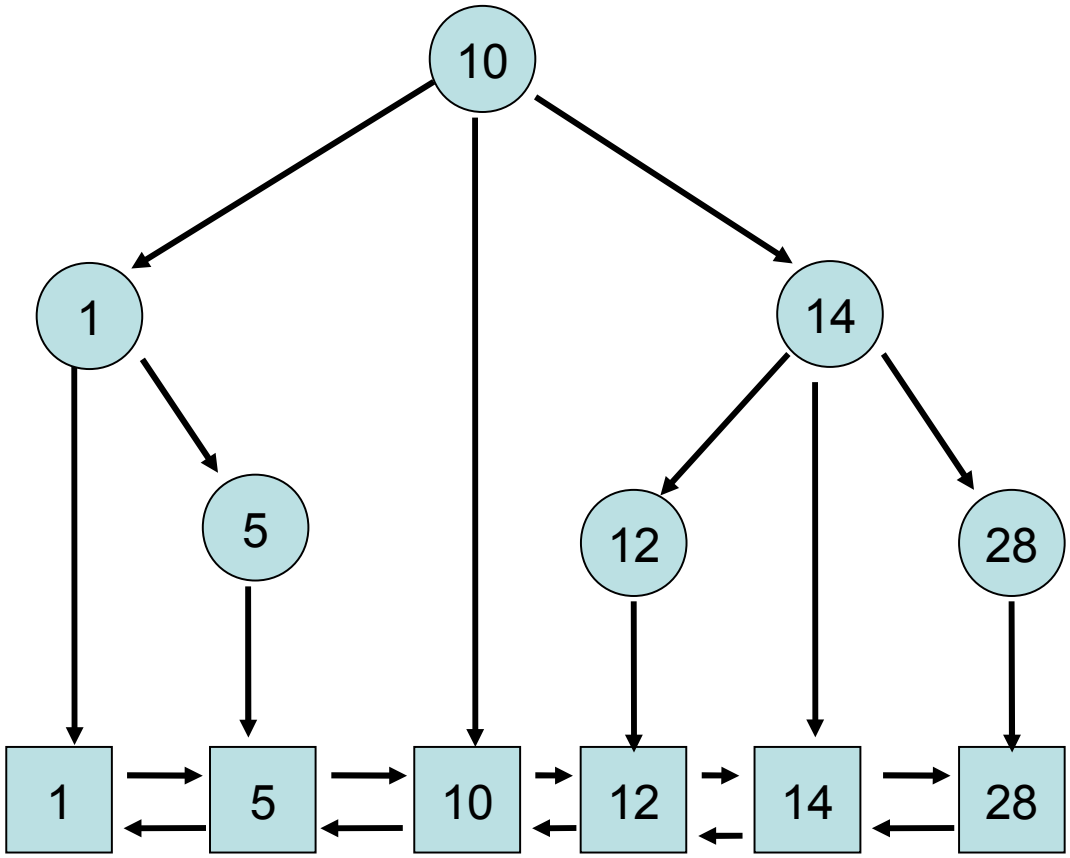
Insert(12)



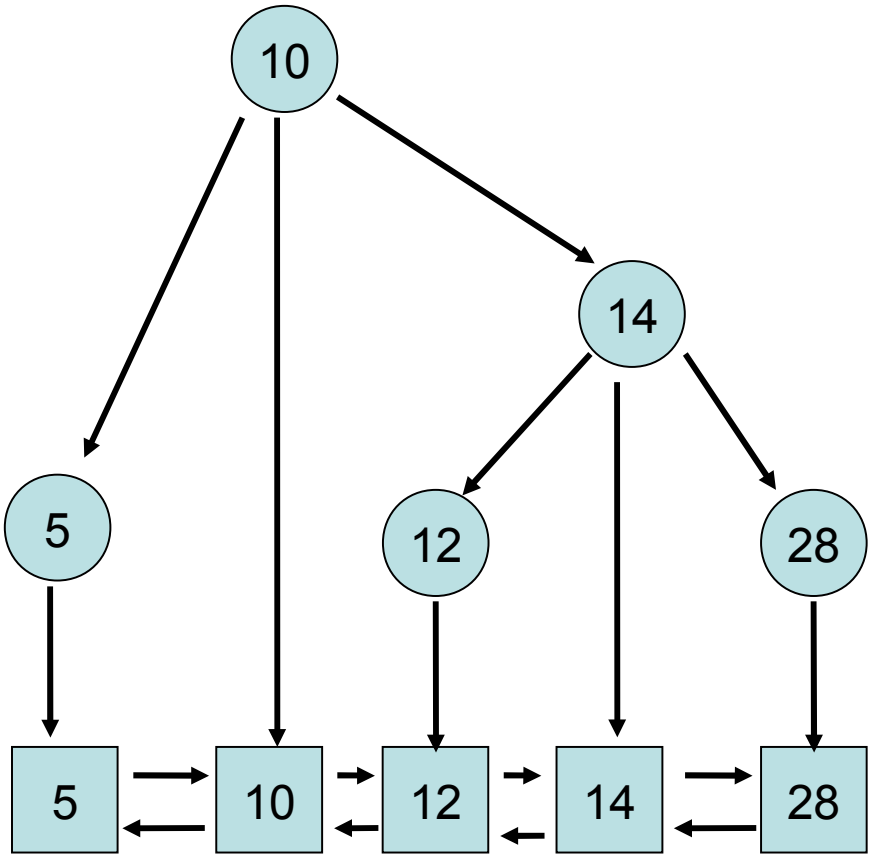
Insert(12)



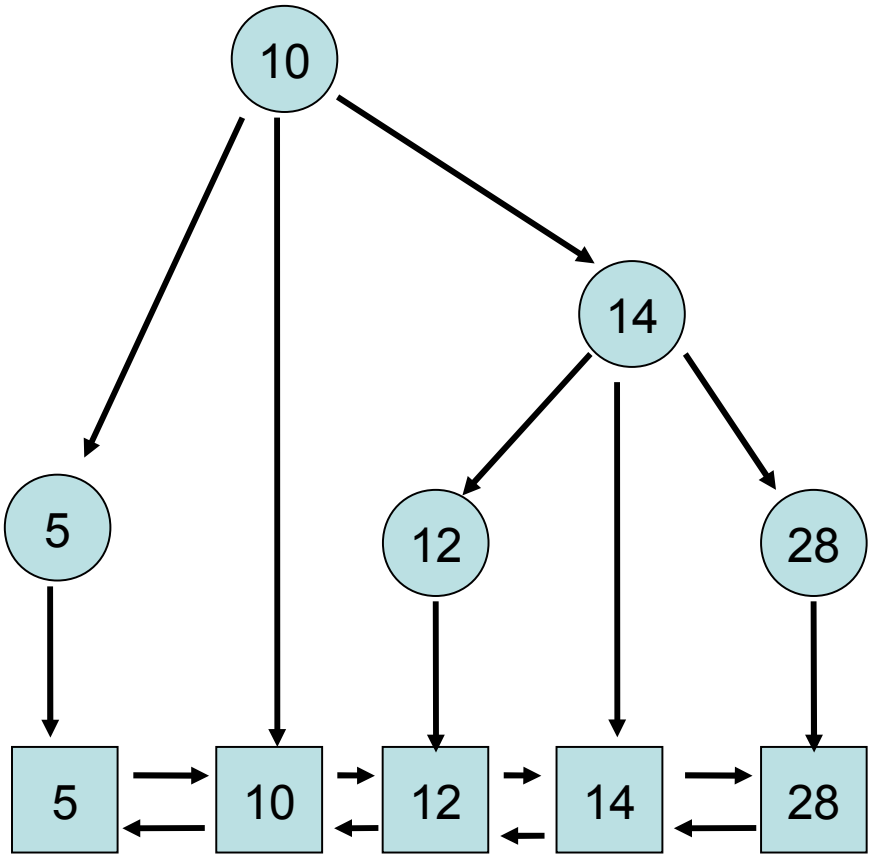
removeKey(1)



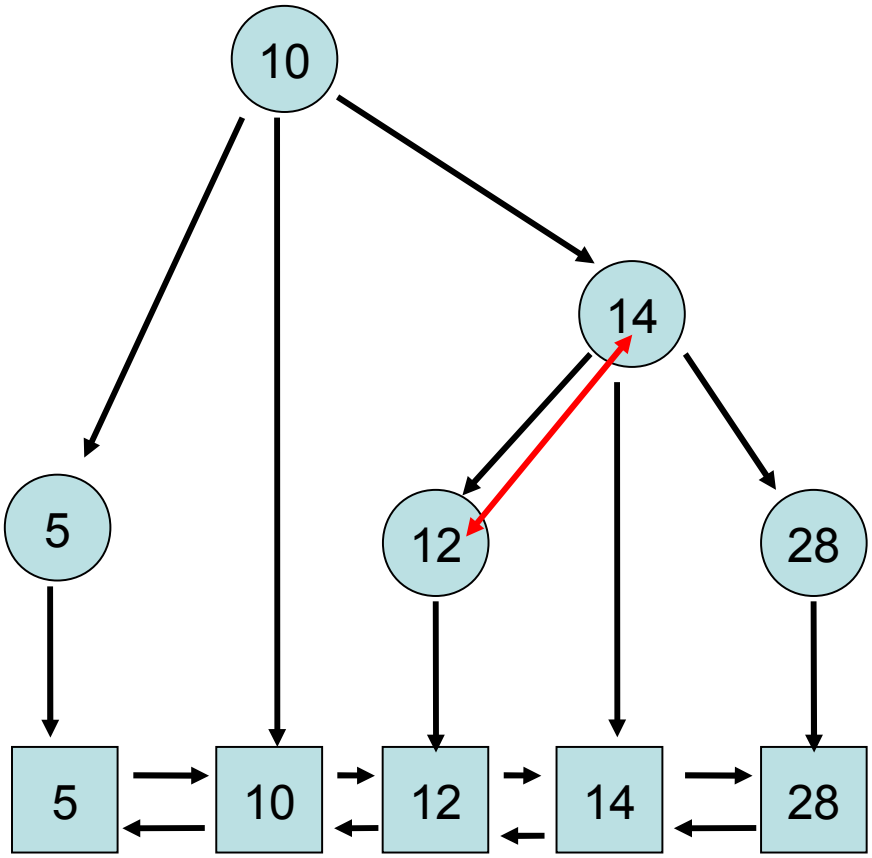
removeKey(1)



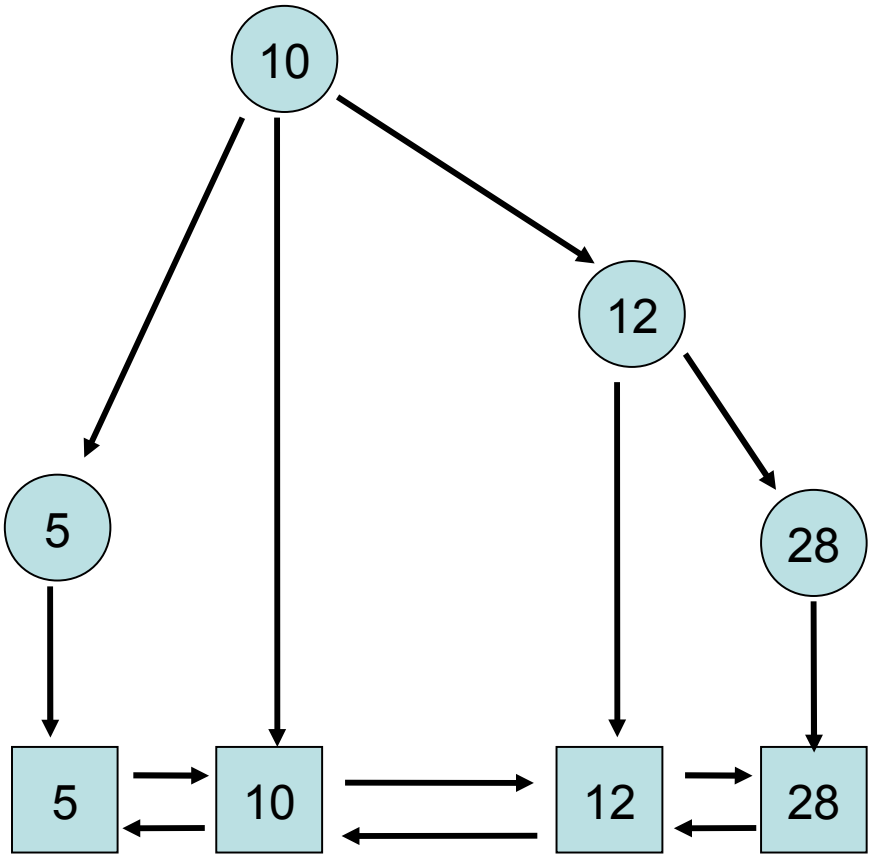
removeKey(14)



removeKey(14)



removeKey(14)



Datenstruktur

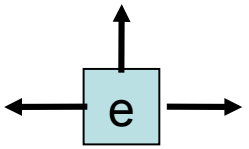
```
// Item: Typ für □ , TreeItem: Typ für ○

interface Elem<K extends Comparable<K>> { K key();}

class Item<K extends Comparable<K>, E extends Elem<K>> {
// doppelt verketttete Liste von Elementen und Baumknoten
      TreeItem<K,E> t = null;
      E e = null;
      Item<K,E> prev, next;

      Item () { prev = next = this;}
      Item (E e, TreeItem<K,E> t, Item<K,E> prev, Item<K,E> next) {
              this.e = e; this.t = t; this.prev = prev; this.next = next;}

      Item<K,E> insertBefore(E e, TreeItem<K,E> t) {...}
      Item<K,E> insertAfter(E e, TreeItem<K,E> t) {...}
      void remove() {...}
}
```

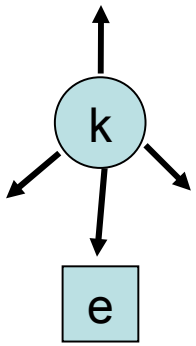


Datenstruktur

```
class TreeItem<K extends Comparable<K>, E extends Elem<K>> {
      // Baumknoten mit Schlüssel, Item, Vater und Nachfolgern

K key;
Item<K,E> item = null;
TreeItem<K,E> pa = null;
TreeItem<K,E> lc = null;
TreeItem<K,E> rc = null;

TreeItem(K key,TreeItem<K,E> pa) {
            this.key = key; this.pa = pa;
            }
   ...
}
```



Datenstruktur

```
class SearchTree<K extends Comparable<K>,
                 E extends Elem<K>> {

// erzeuge leere doppeltverkettete Liste

    Item<K,E> l;
    TreeItem<K,E>;
    SearchTree() {
        l = new Item<K,E>();
        t = new TreeItem(null,null);      // leerer Baumknoten mit
        t.item = l;                       // Verweis auf leeren Listenknoten
        l.t = t;
    }
   ...
}
```

Locate Operation

```
// in SearchTree
Item<K,E> locate(K key) {
   if (t.rc == null) return l;
    return t.rc.locate(key);  // starte Suche in Wurzel
}
// in TreeItem
Item<K,E> locate(K key) {
    switch (key.compareTo (this.key)) {
     case -1: if (lc == null) return item;
                   else return lc.locate(key);
      case 0: return item;
     case +1: if (rc == null) return item.next;
                    return rc.locate(key);
    }
    return null;
}
```

Insert Operation

```
// in SearchTree

void insert(Element e) {
    K key = e.key();
   if (t.rc == null) {
        t.rc = new TreeItem<K,E> (key,t);
        t.rc.item = l.insertAfter(e, t.rc);
   } else t.rc.insert(e);
}
```

Insert Operation

```
// in TreeItem
void insert (E e) {
   K key = e.key();
   if (key.compareTo(this.key) < 0)  if (lc == null) {
            lc = new TreeItem<K,E>(key,this);
            lc.item = item.insertBefore(e,lc);
        } else lc.insert(e);
  else if (rc == null) {
            rc = new TreeItem<K,E>(key,this);
            rc.item = item.insertAfter(e,rc);
        } else rc.insert(e);
}
```

RemoveKey Operation

```
// in SearchTree

void removeKey(K key) {
   if (t.rc == null) return;
   t.rc. removeKey(key);
}
// in TreeItem
void removeKey(K key) {
    switch (key.compareTo(this.key)) {
    case -1: if (lc == null) break;
                else lc.removeKey(key); break;
    case +1: if (rc == null) break;
                else rc.removeKey(key); break;
                ...
```

RemoveKey Operation

```
   ...
   case 0: item.remove();
            if (lc == null)
                  if (this == pa.lc)  pa.lc = rc;      // Hochkopieren des Nachfolgers
                  else pa.rc = rc;                     // Hochkopieren des Nachfolgers
                  if (rc!= null) rc.pa = pa;           // Korrektur des Vaterverweises
             } else if (rc == null) {
                  if (this == pa.lc)   pa.lc = lc;
                  else  pa.rc = lc;
                  lc.pa = pa;
            } else lc.removeMax(this);
      }
}
```

RemoveKey Operation

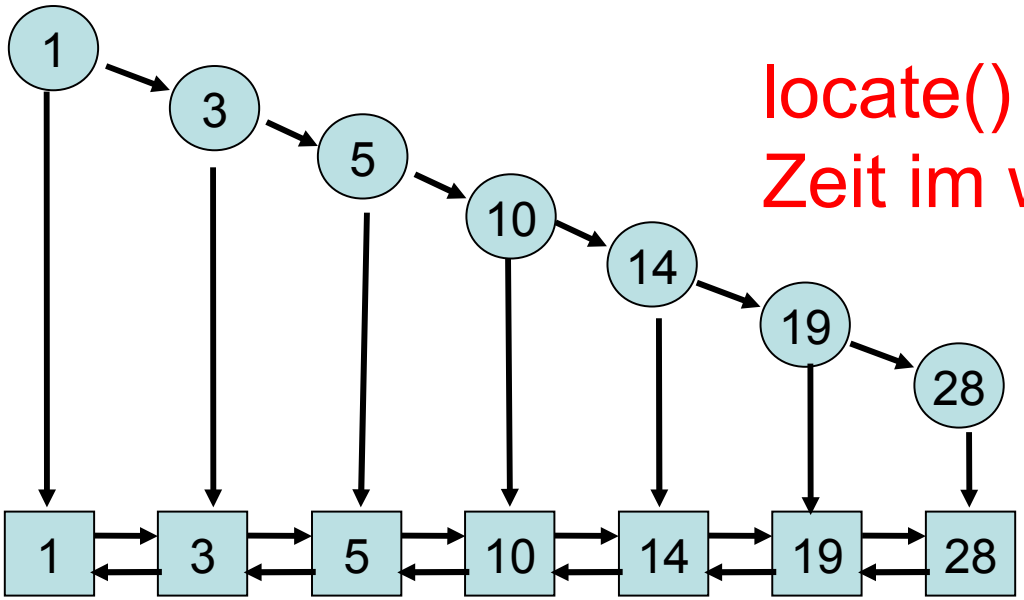
```
// in TreeItem

void removeMax(TreeItem<K,E> t) {
    if (rc == null) {
        item.t = t;
        t.item = item;  t.key = item.e.key();
        if (this == pa.lc) {
                    pa.lc = lc;
                    if (lc != null) lc.pa = pa;
        } else {
                  pa.rc = lc;
                  if (lc != null) lc.pa = pa;
       }
   } else rc.removeMax(t);
}
```

Binärbaum

Problem: Binärbaum kann entarten!

Beispiel: Zahlen werden in sortierter Folge eingefügt



locate() benötigt Θ (n)
Zeit im worst case

(a,b)-Bäume

Problem: Binärbaum kann entarten!

Lösung: (a,b)-Baum

Idee:

- Alle Knoten v außer der Wurzel enthalten d(v)-1 Schlüssel mit $a \leq d(v) \leq b$, wobei $a \geq 2$ und $b \geq 2a-1$ ist
- Alle Blätter sind in derselben Ebene

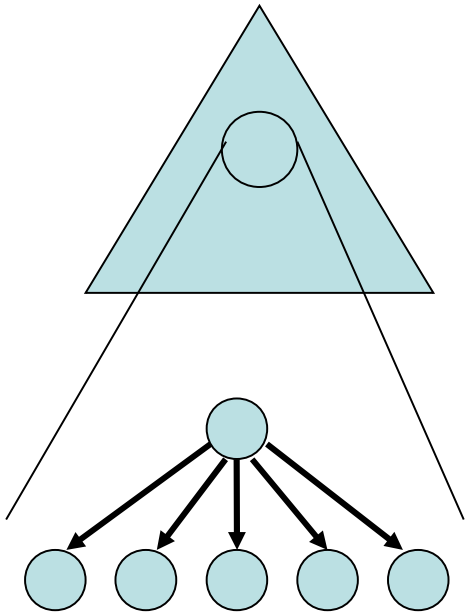
(a,b)-Bäume

Formal: für einen Baumknoten v sei

- $d(v)-1$ die Anzahl der Schlüssel in v
- $t(v)$ die Tiefe von v (Wurzel hat Tiefe 0)

- Form-Invariante:
  Für alle Blätter v,w: $t(v)=t(w)$

- Grad-Invariante:
  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

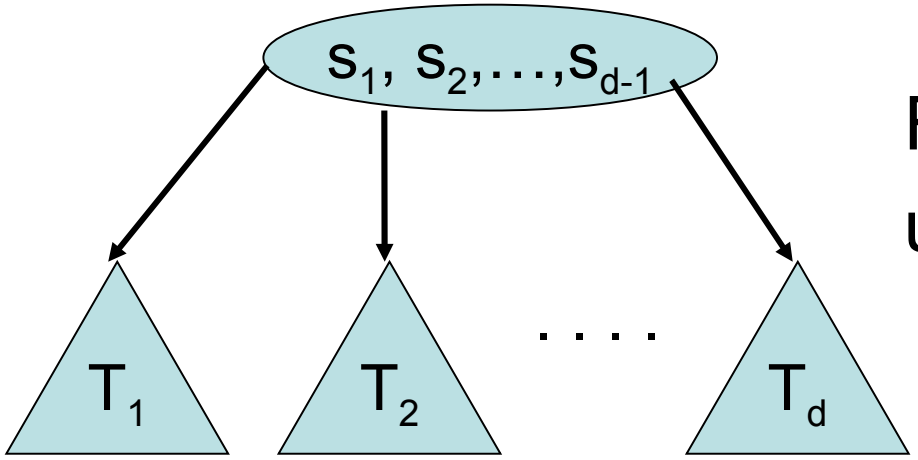
Lemma 7.1: Ein (a,b)-Baum für n Elemente hat Tiefe max. $1+\log_a (n/2)$

Beweis:

- Die Wurzel hat Grad ≥ 2 und jeder andere innere Knoten hat Grad $\geq a$.
- Bei Tiefe t gibt es mindestens $2a^{t-1}$ Blätter
- $n \geq 2a^{t-1} \Leftrightarrow t \leq 1+\log_a n/2$

(a,b)-Bäume

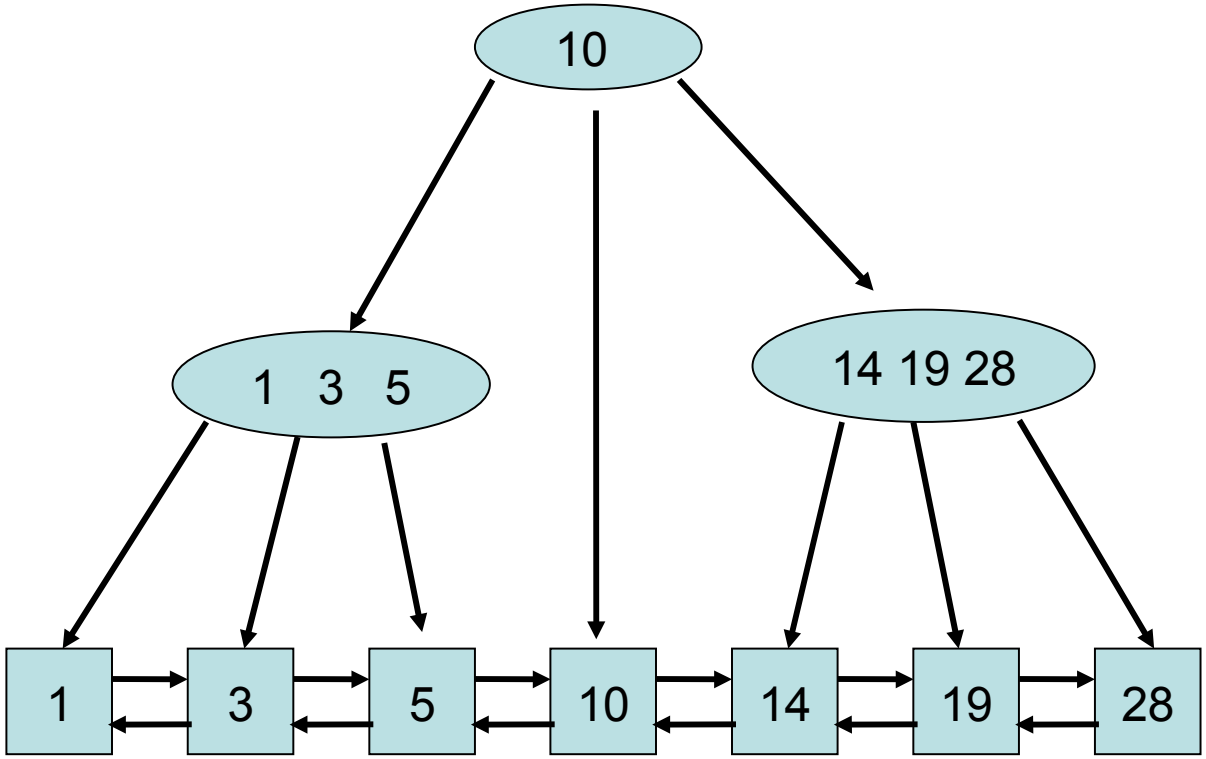
(a,b)-Suchbaum-Regel:



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

Damit locate Operation einfach zu implementieren.

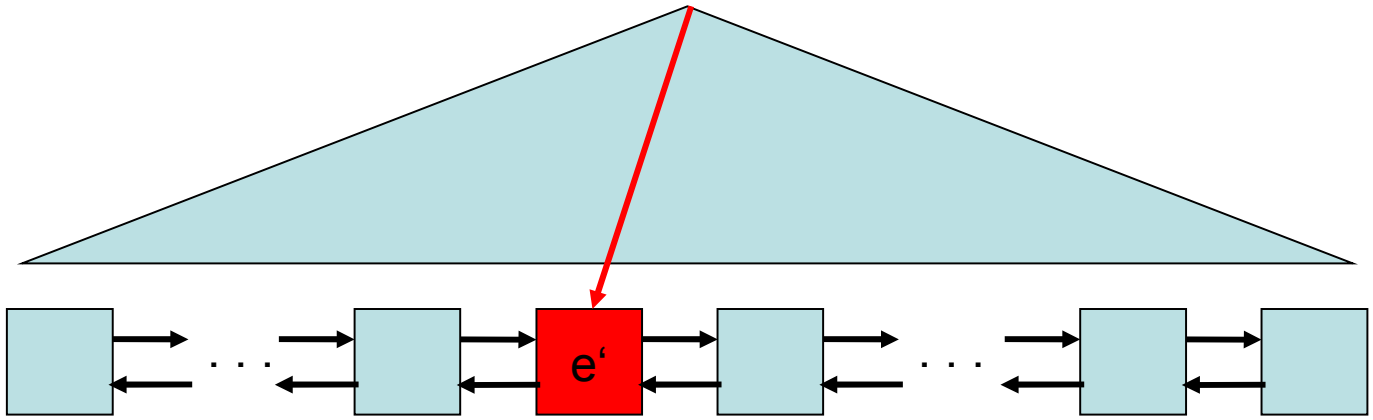
Locate(9)



Insert(e) Operation

Strategie:

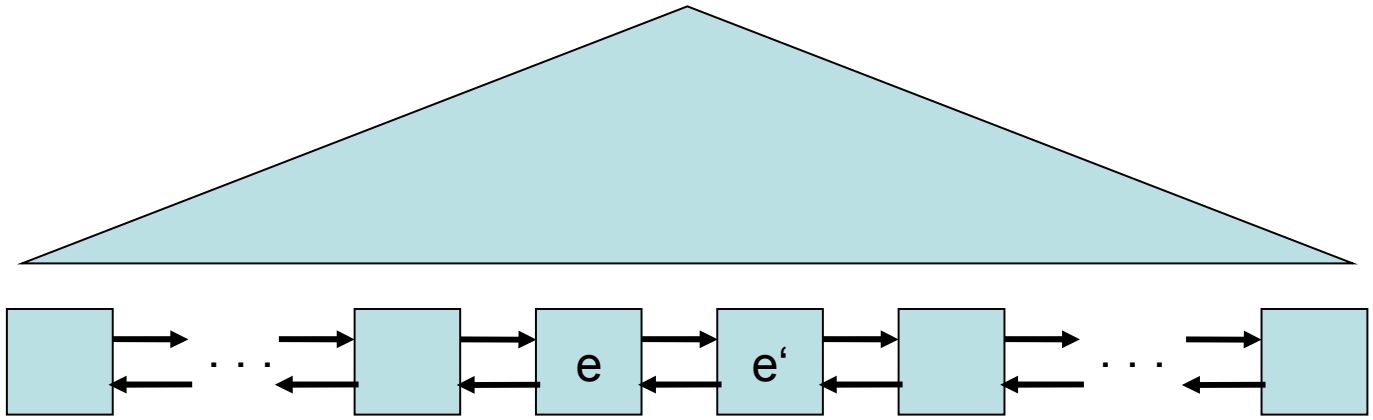
- Suche Blatt v, in das e.key() gehört. Finde benachbartes Element e´ in dem Blatt. Falls e'.key()>e.key(), füge e vor e´ ein, ansonsten dahinter.



Insert(e) Operation

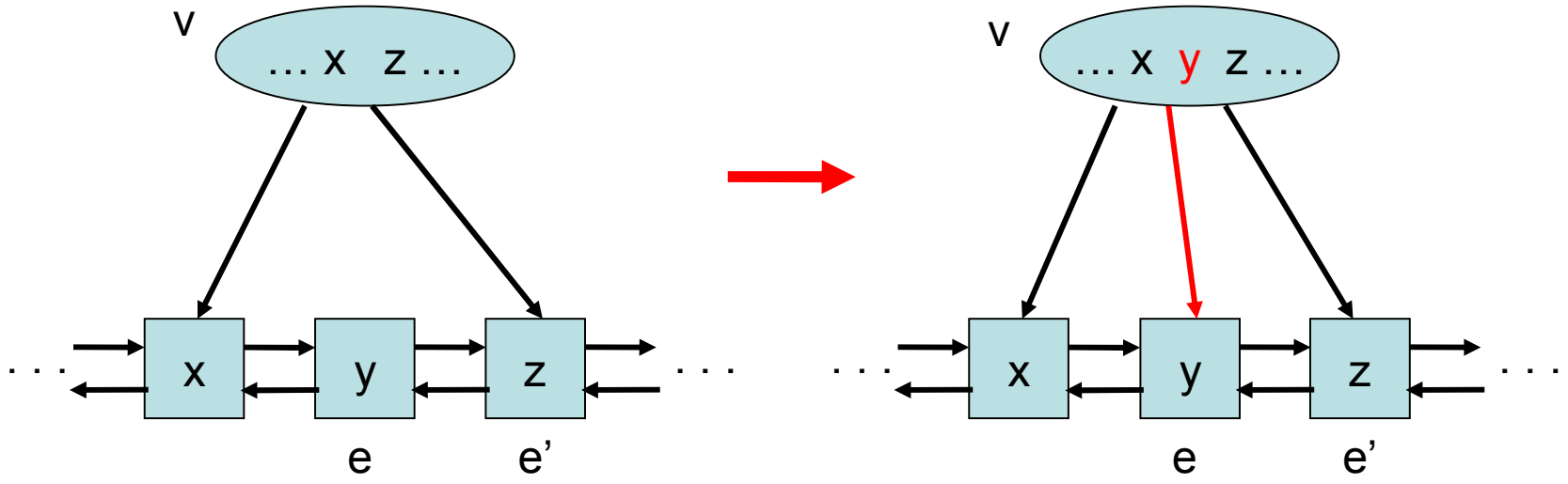
Strategie:

- Suche Blatt v, in das e.key() gehört. Finde benachbartes Element e´ in dem Blatt. Falls e'.key()>e.key(), füge e vor e´ ein, ansonsten dahinter.



Insert(e) Operation

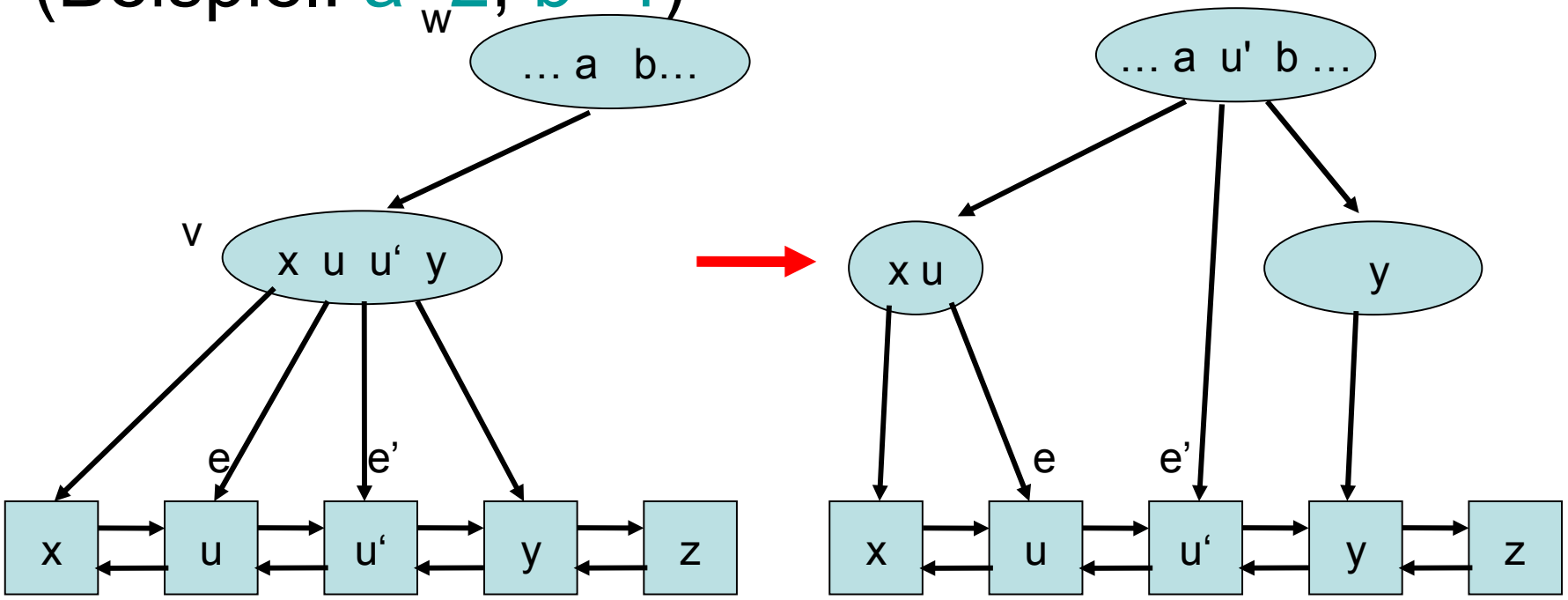
- Füge e.key() und Verweis auf e in Baumknoten v ein. Falls v nachher weniger als b Elemente hat, dann fertig.



Insert(e) Operation

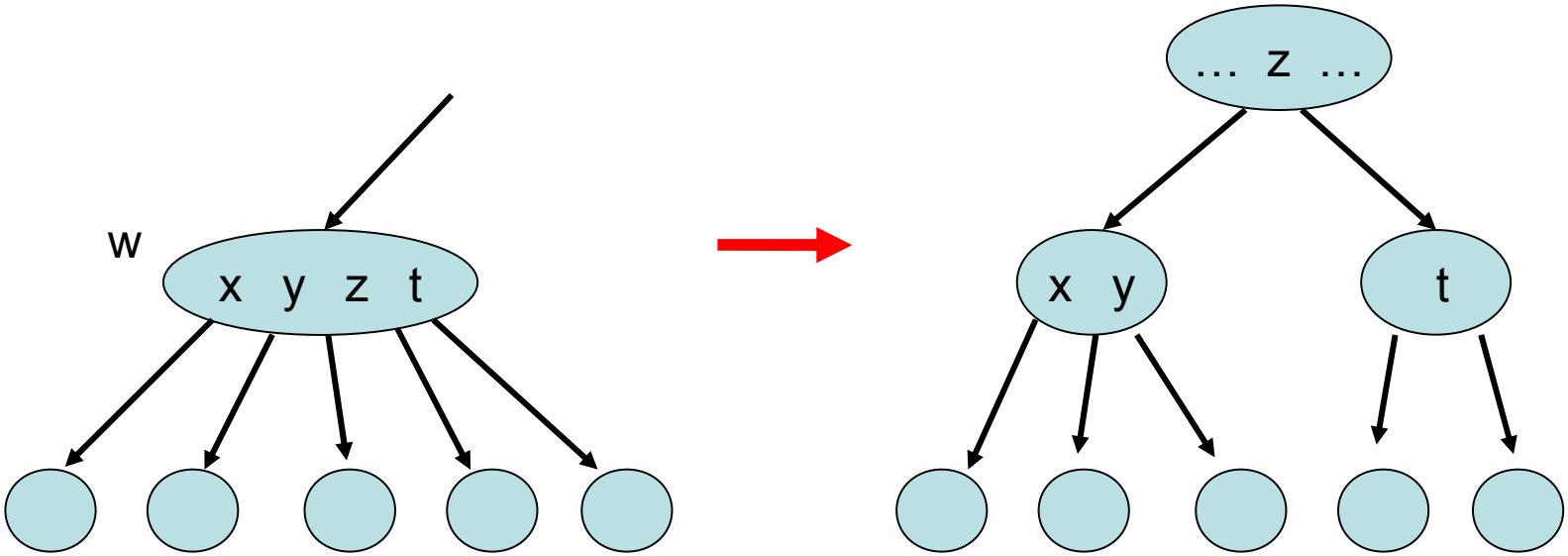
- Falls v nachher b Elemente enthält, teile v in zwei Knoten auf.

(Beispiel: a=2, b=4)



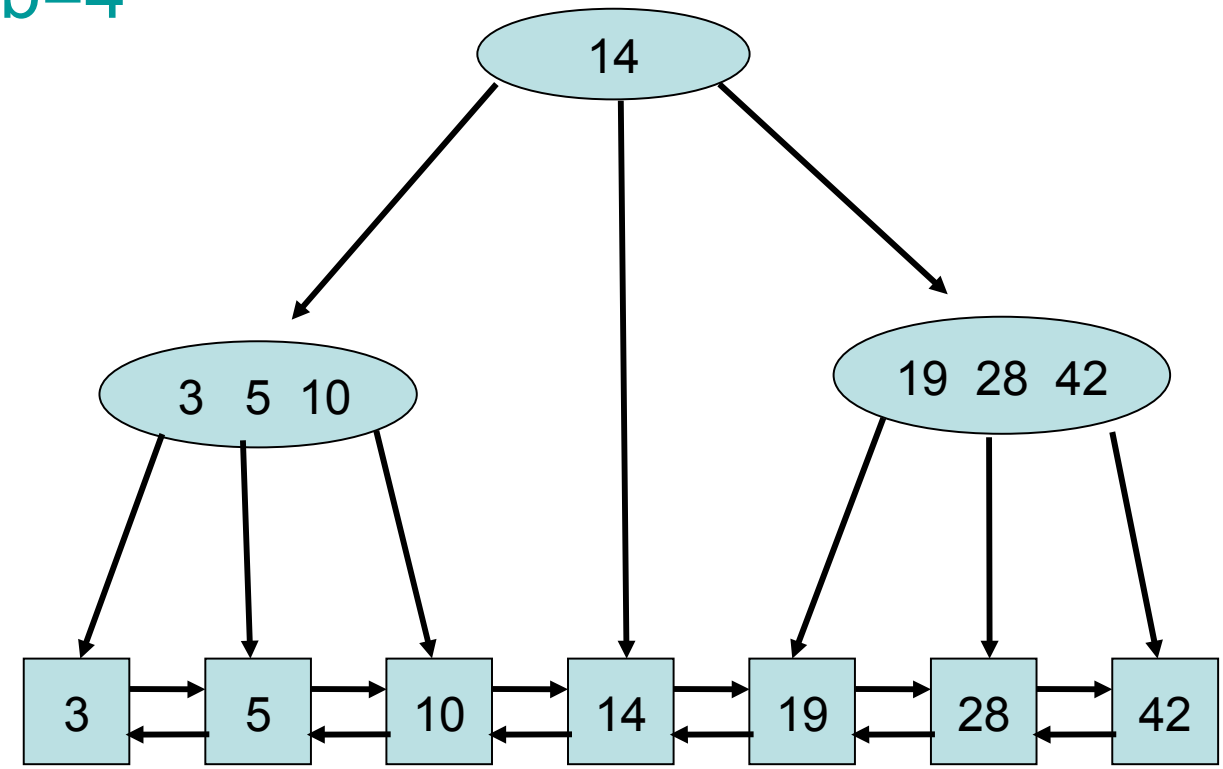
Insert(e) Operation

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



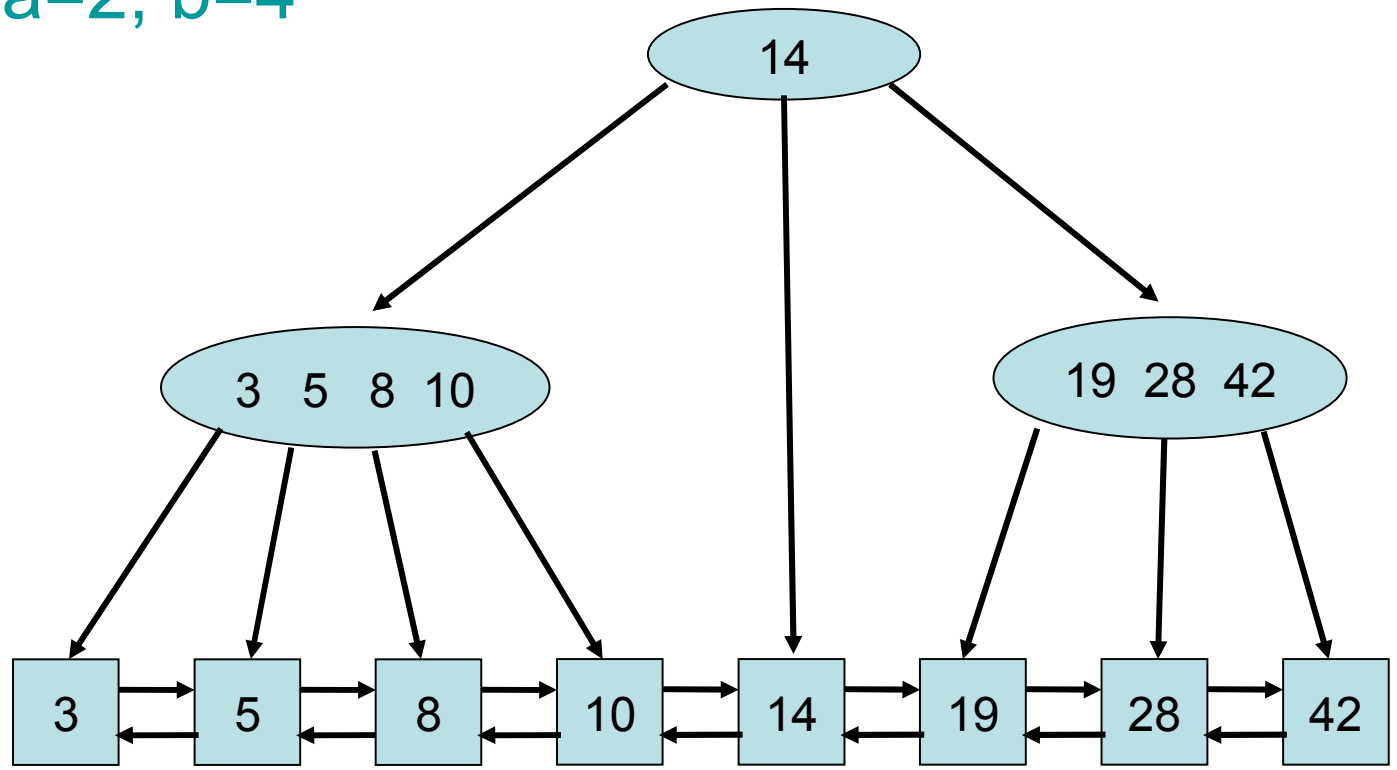
Insert(8)

a=2, b=4



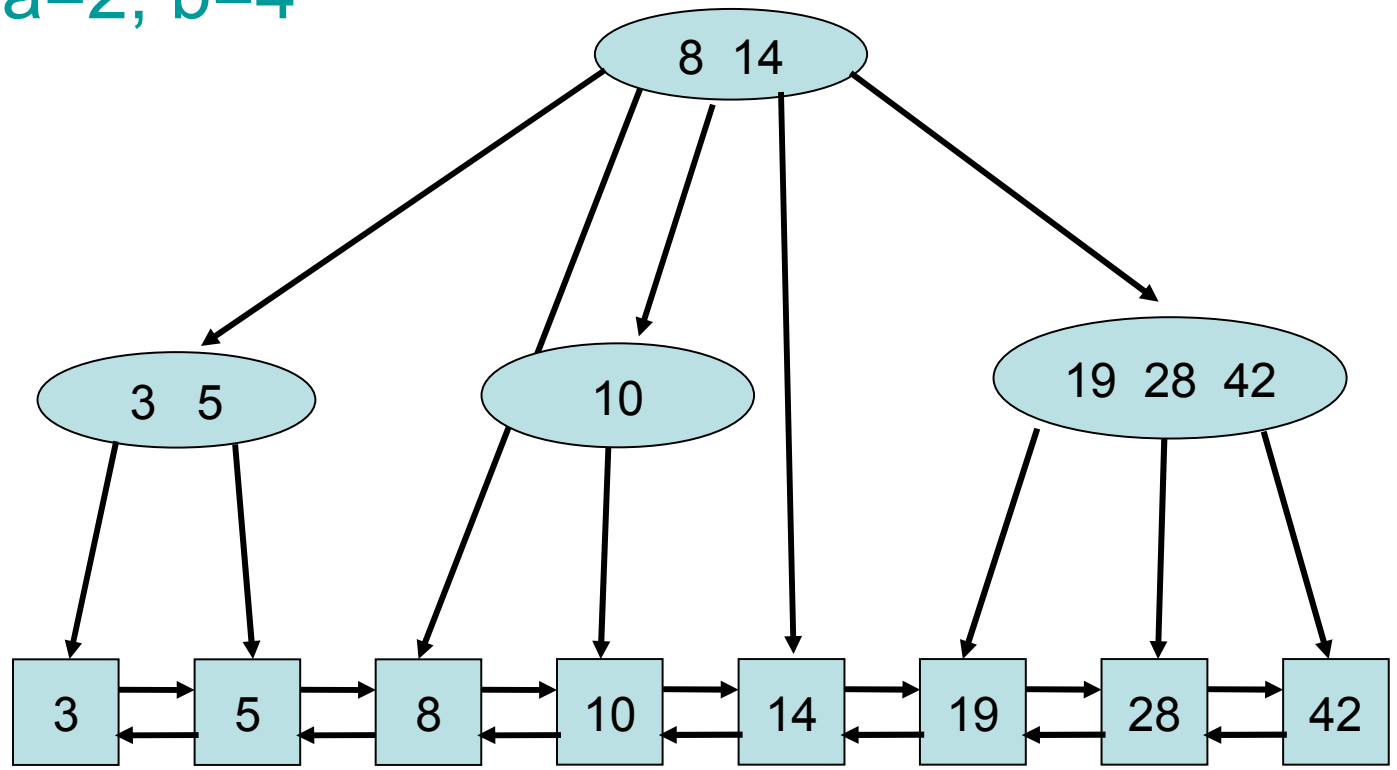
Insert(8)

a=2, b=4



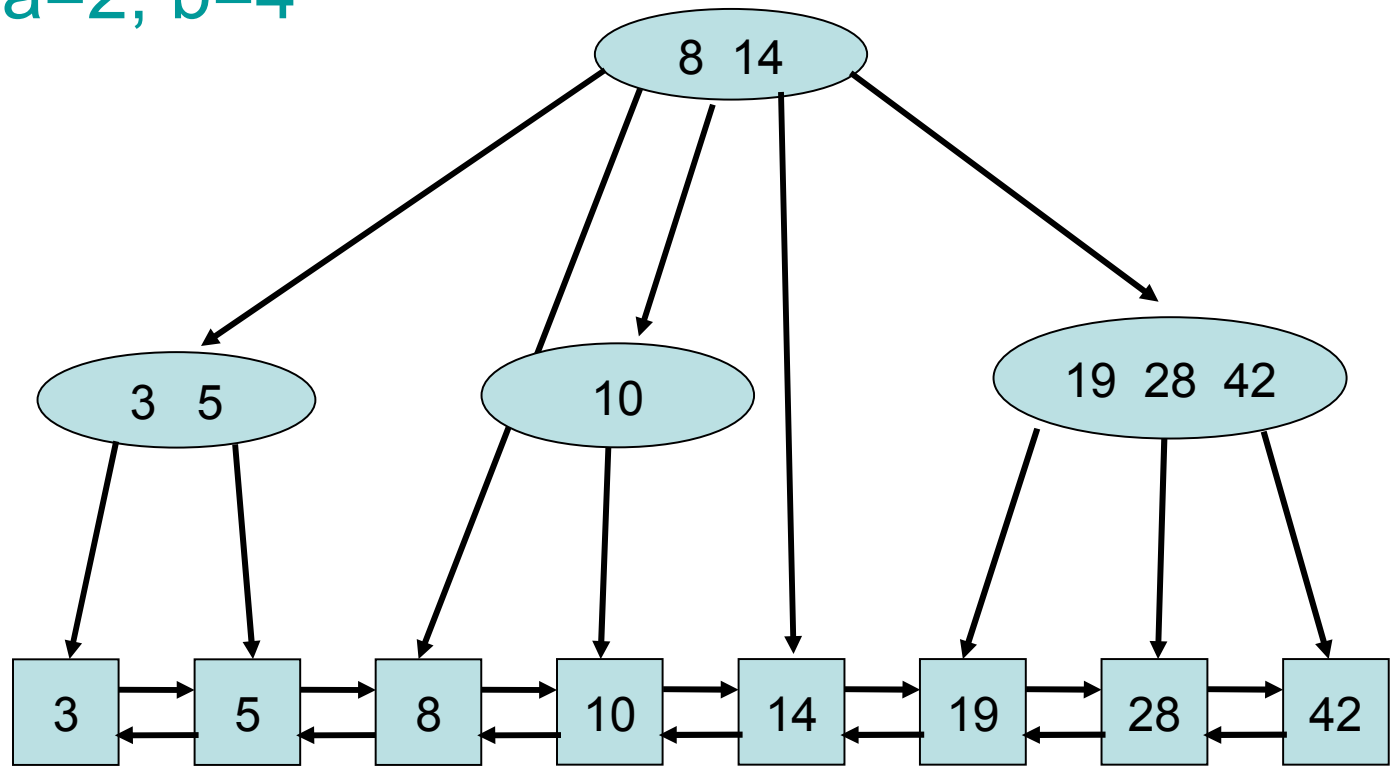
Insert(8)

a=2, b=4



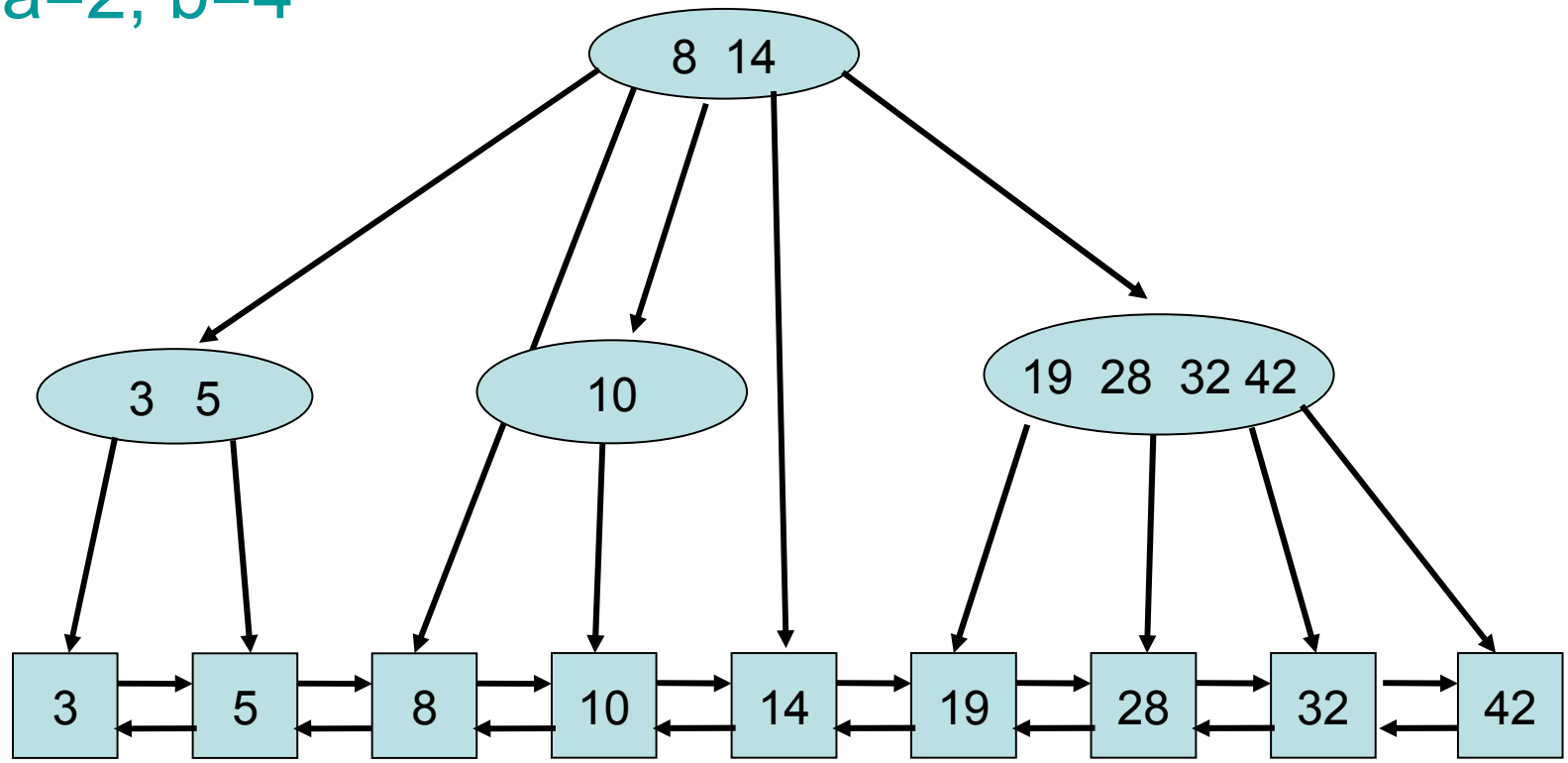
Insert(32)

a=2, b=4



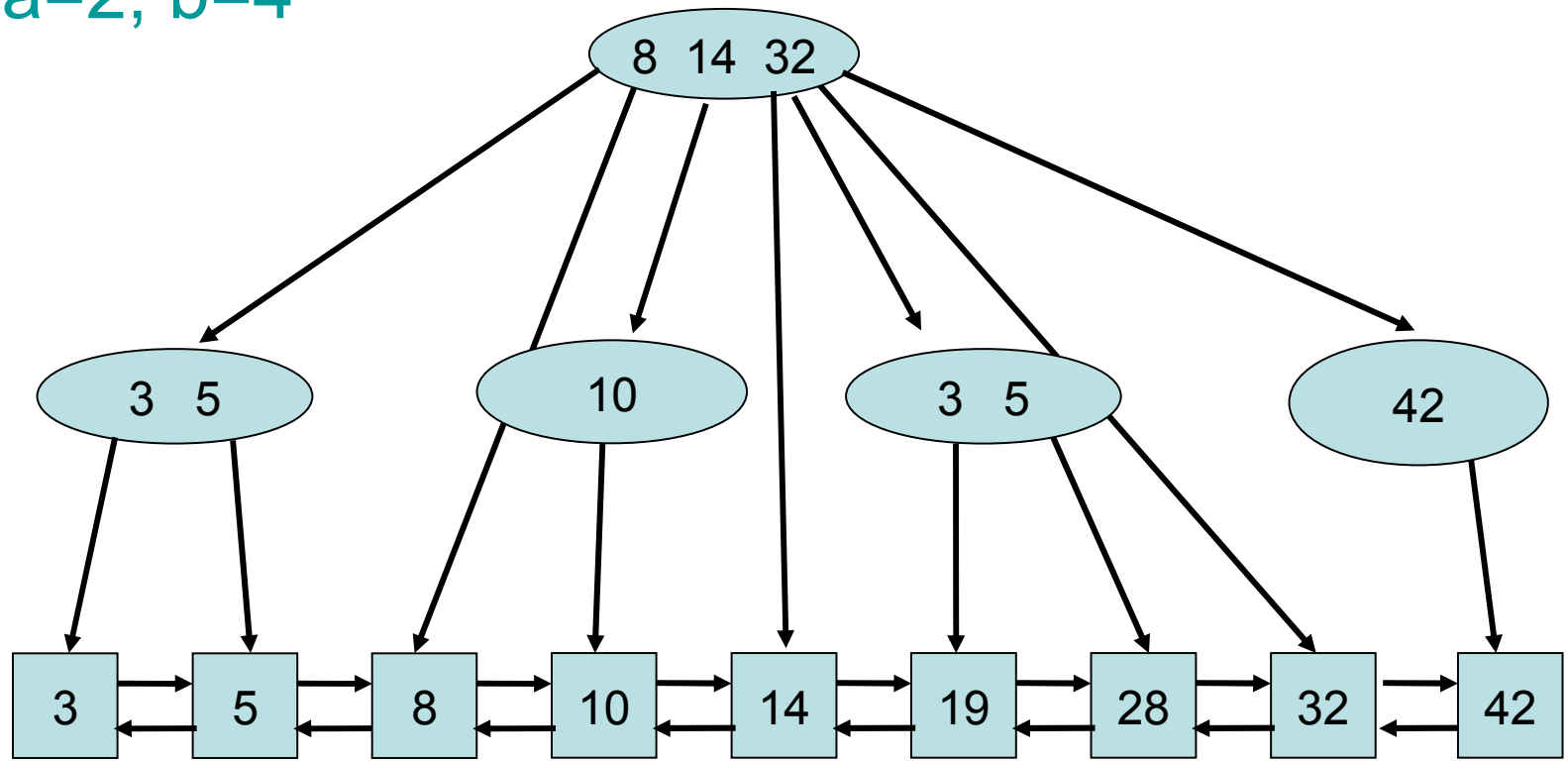
Insert(32)

a=2, b=4



Insert(32)

a=2, b=4



Insert Operation

- Form-Invariante:
  Für alle Blätter v,w: t(v)=t(w)
  Erfüllt durch insert!

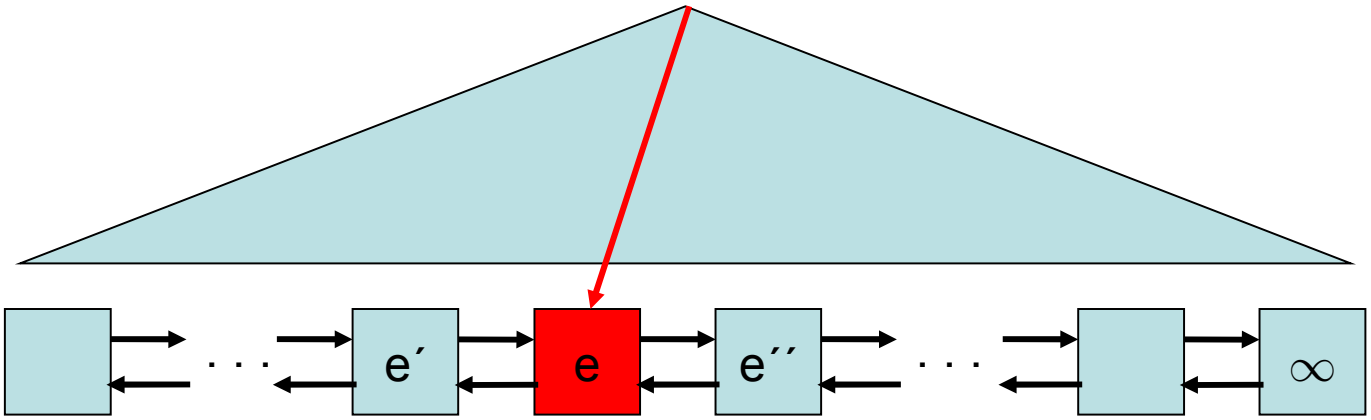
- Grad-Invariante:
  Für alle inneren Knoten v außer Wurzel: $d(v) \in [a,b]$,
  für Wurzel r: $d(r) \in [2,b]$

  1) Insert splittet Knoten mit b Schlüsseln in Knoten mit b/2 bzw. (b-1)/2 Schlüsseln. Wenn $b \geq 2a-1$, dann beide Werte $\geq a-1$.
  2) Wenn Wurzel b Schlüssel enthält, wird eine neue Wurzel mit einem Schlüssel d.h. Grad 2 erzeugt.

removeKey(k) Operation

Strategie:

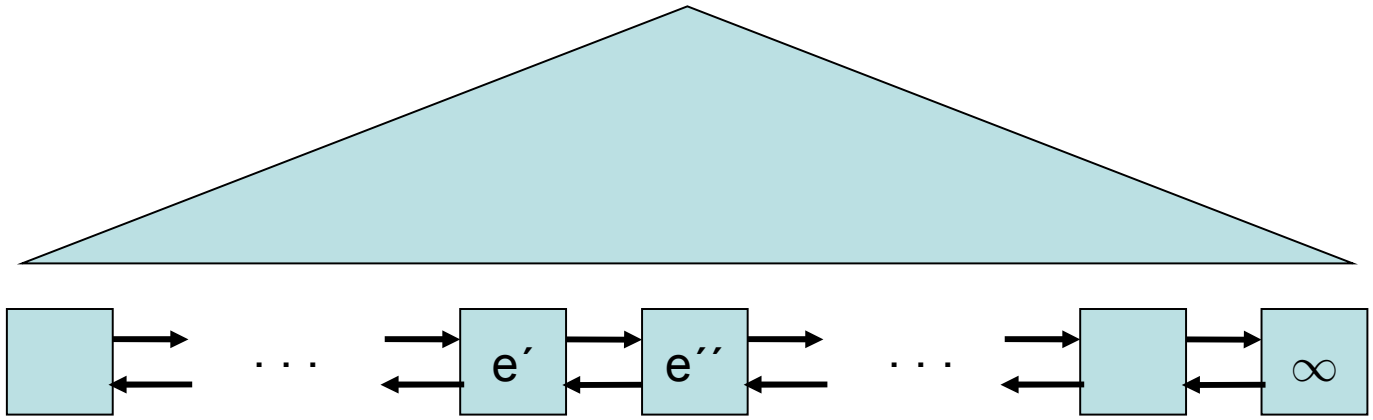
- Erst locate(k) bis Element e in Liste erreicht. Falls e.key()==k, entferne e aus Liste, ansonsten stop.



RemoveKey(k) Operation

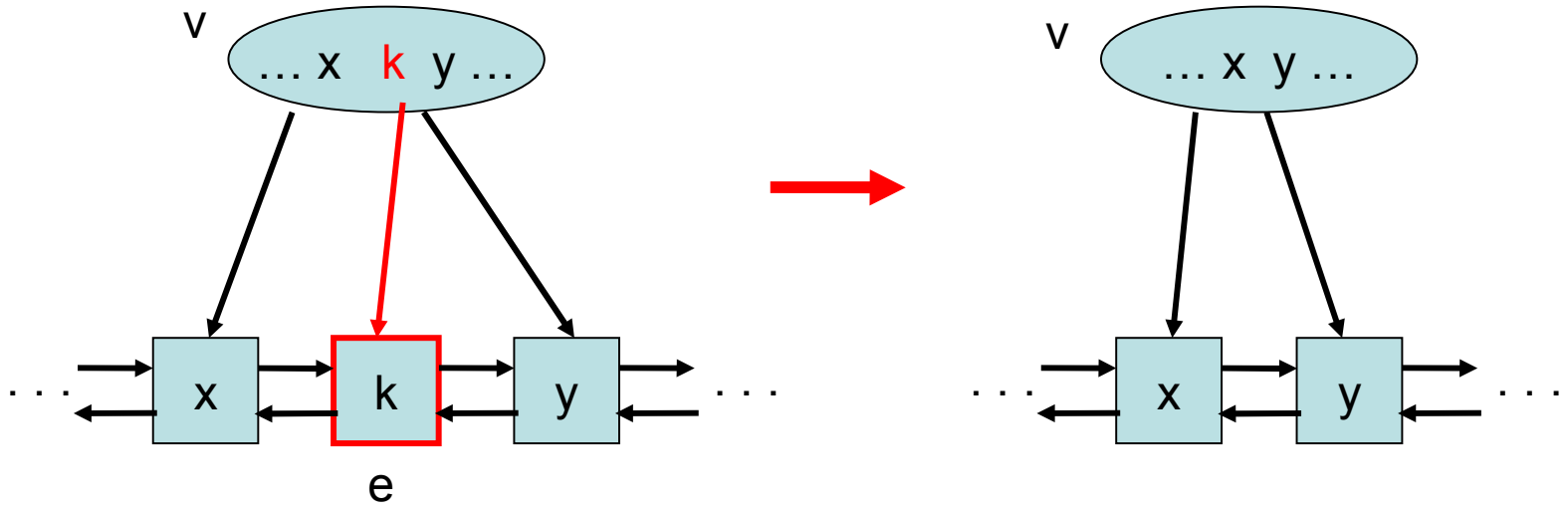
Strategie:

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



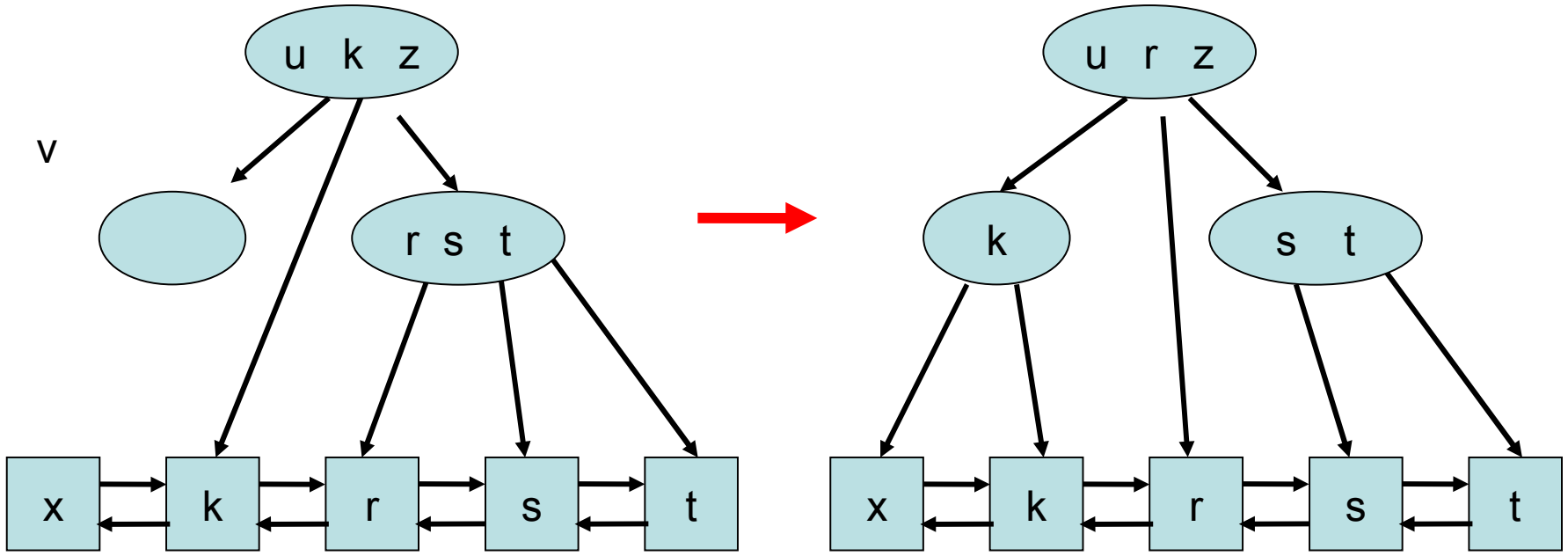
RemoveKey(k) Operation

- Entferne Verweis auf e und Schlüssel k vom Baumknoten v mit e. Falls v ein Blatt ist und noch a-1 Schlüssel enthält, dann fertig.



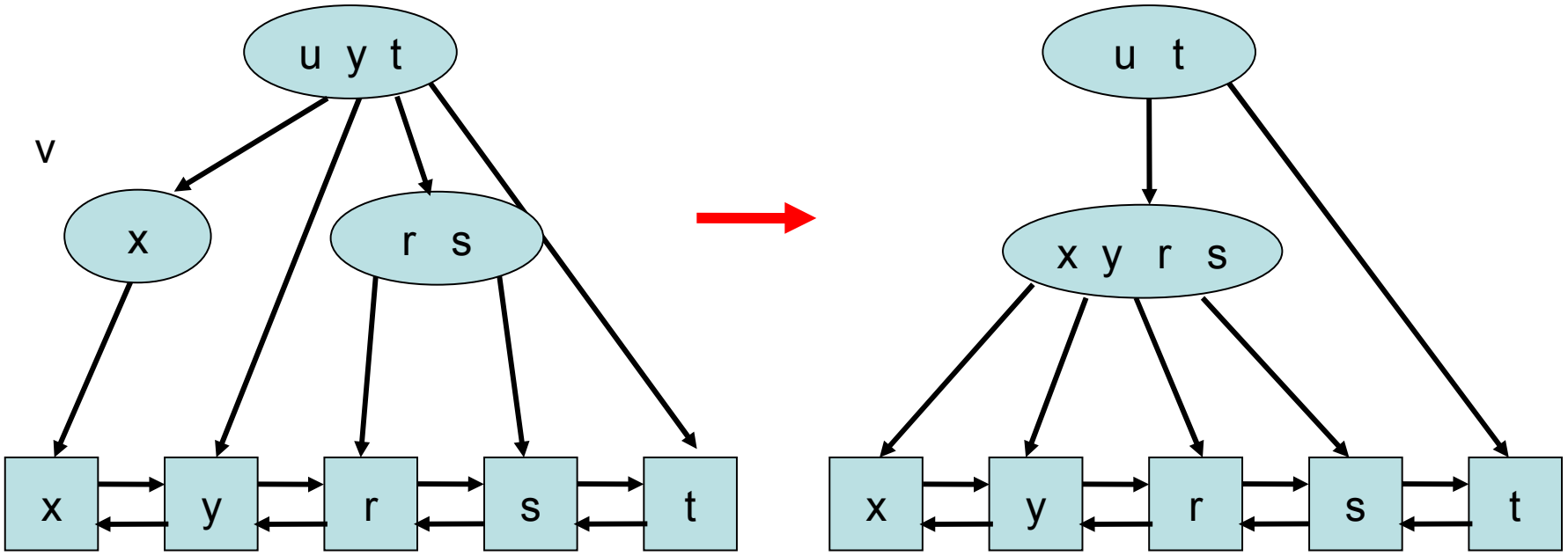
RemoveKey(k) Operation

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



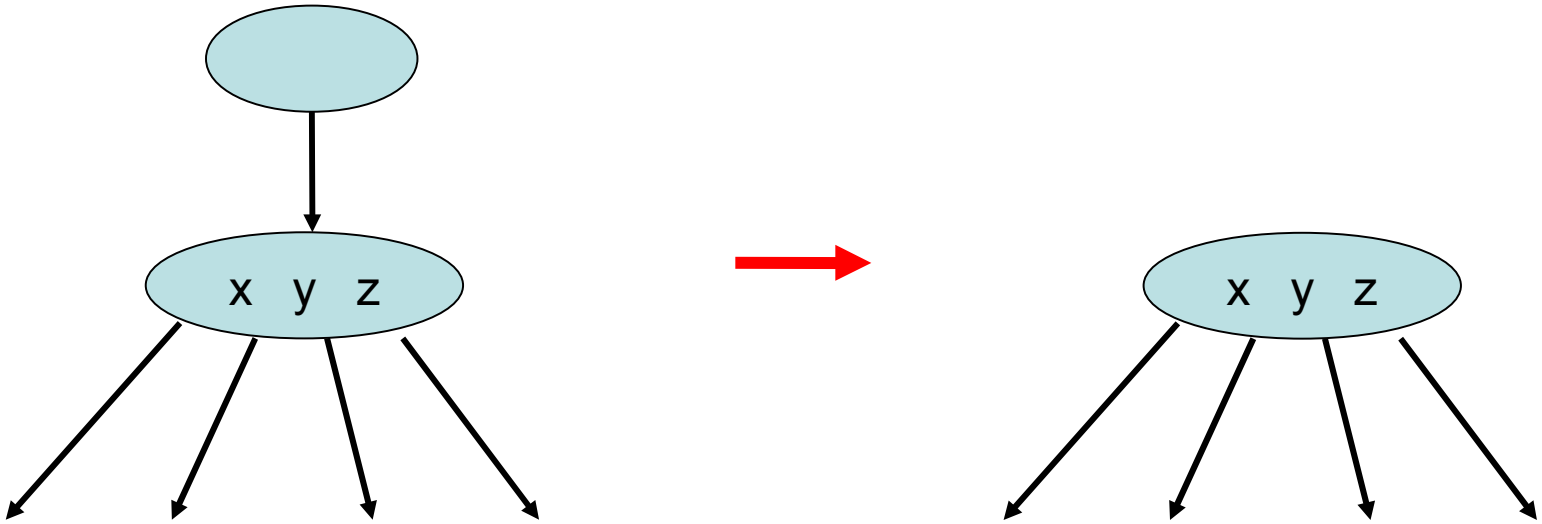
RemoveKey(k) Operation

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



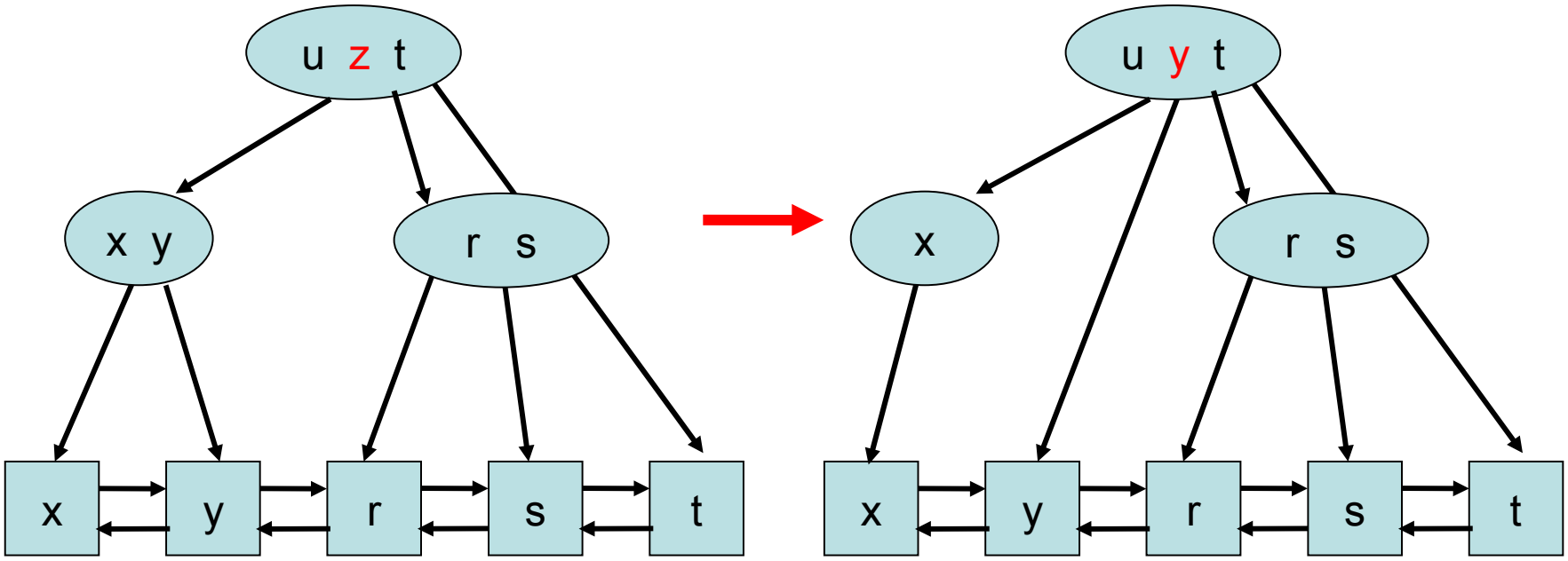
Remove(k) Operation

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



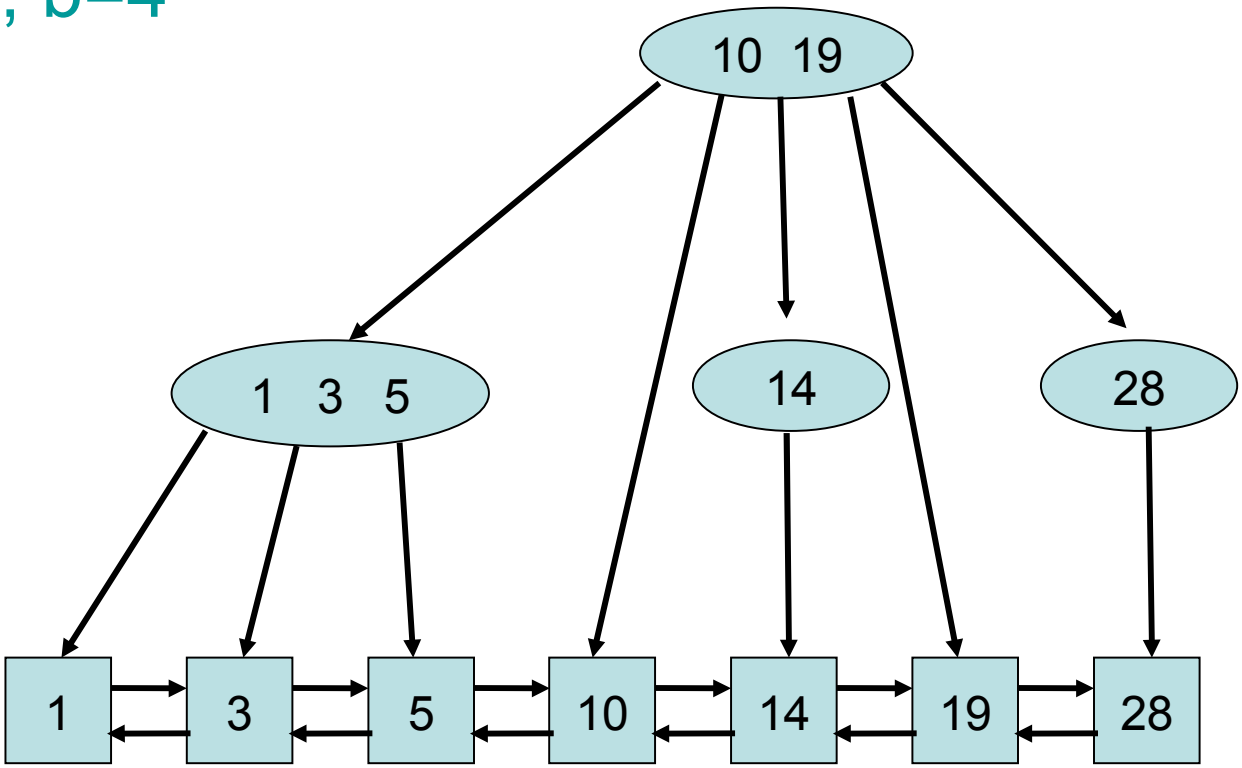
Remove(k) Operation

- Falls y in innerem Knoten gelöscht wird, ersetze durch max. Schlüssel in linkem Teilbaum. (Beispiel: a=2, b=4)



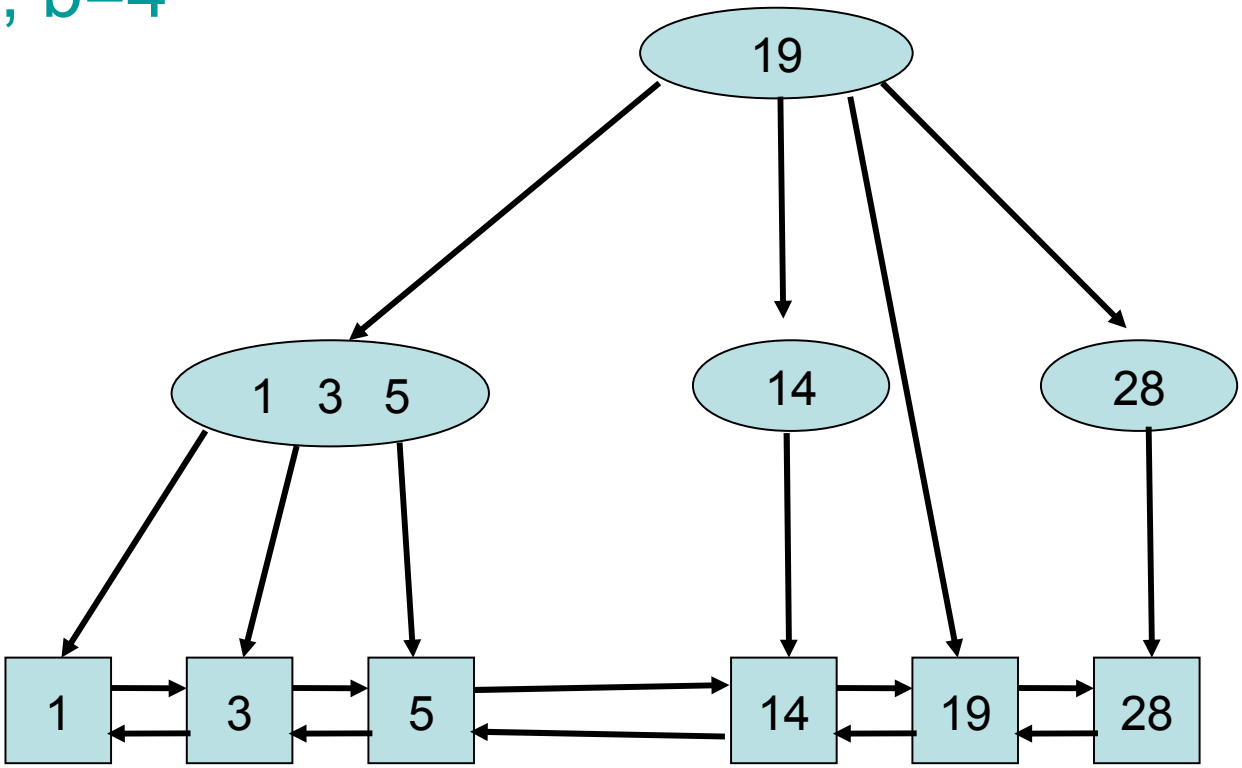
Remove(10)

a=2, b=4



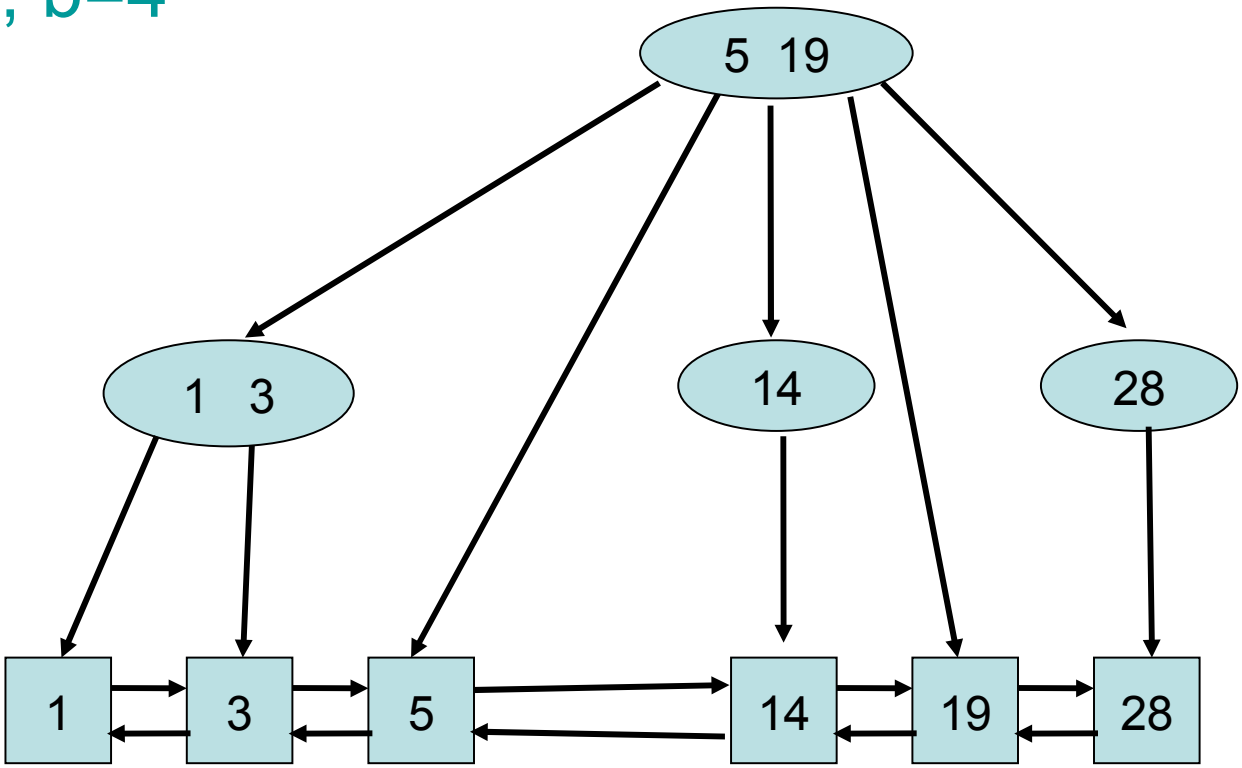
Remove(10)

a=2, b=4



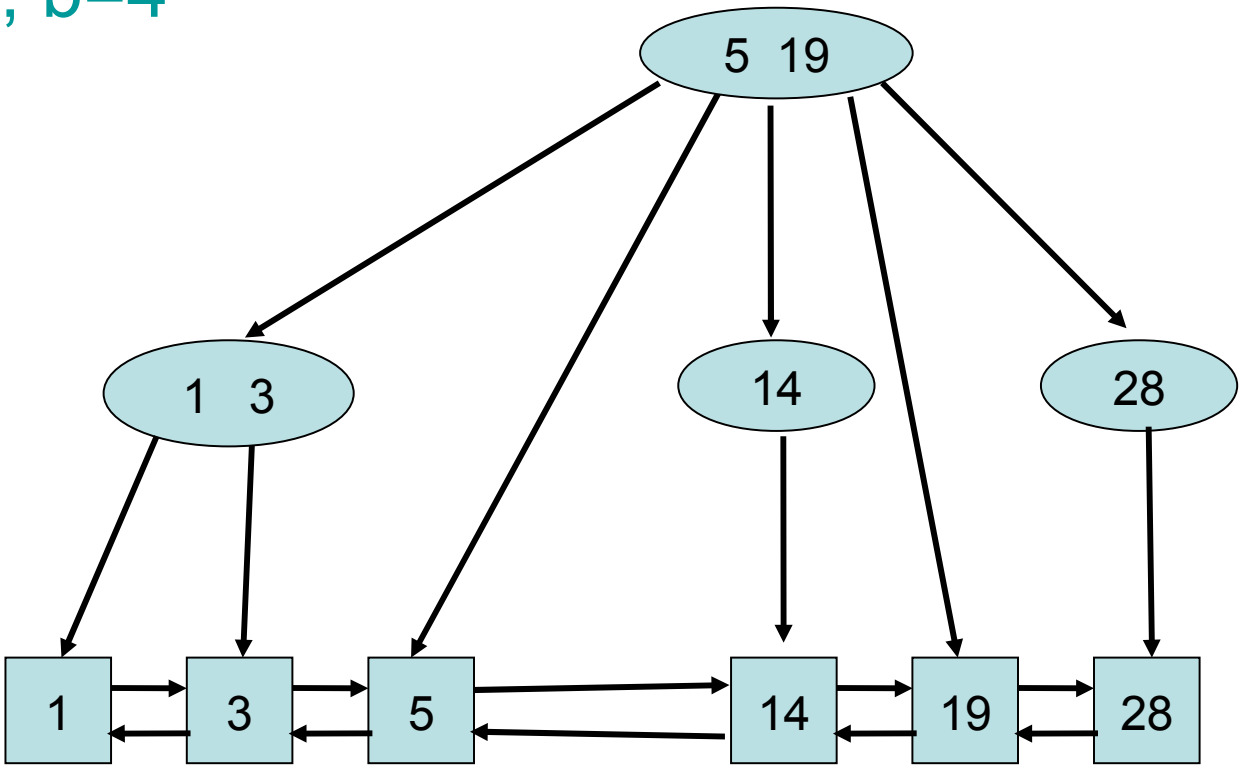
Remove(10)

a=2, b=4



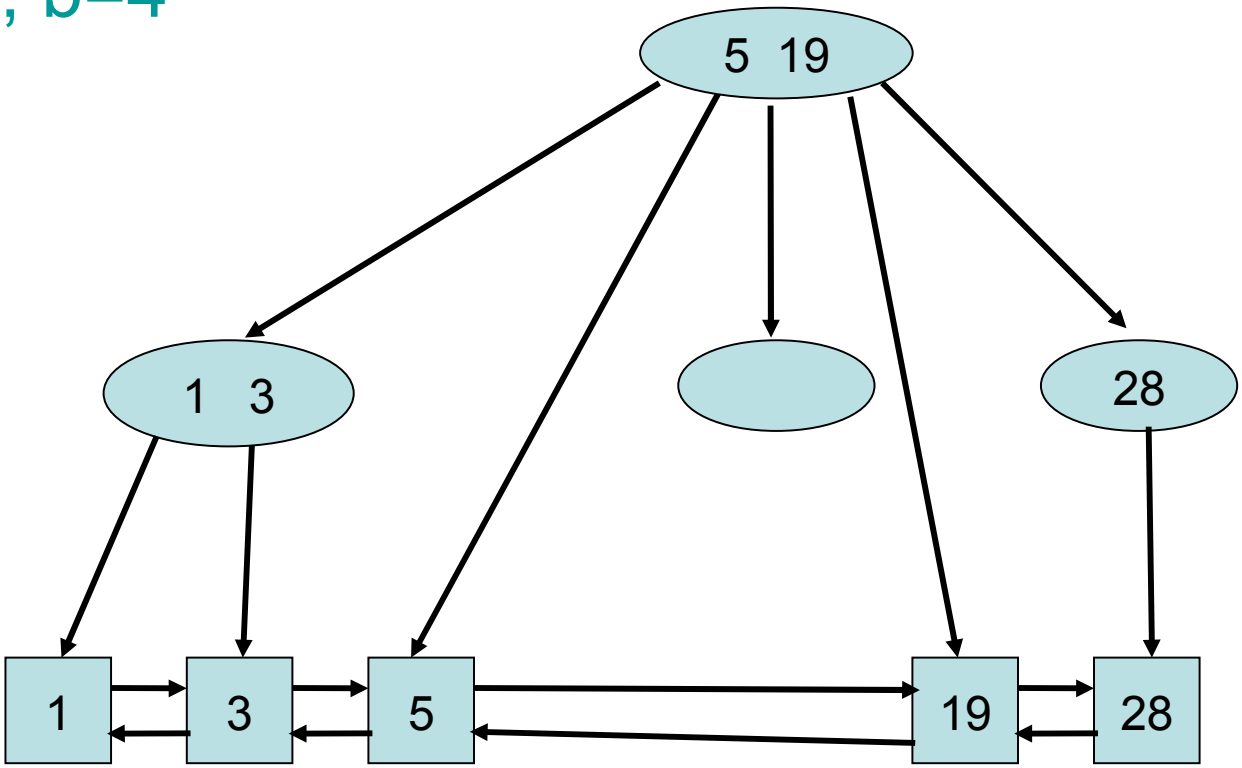
Remove(14)

a=2, b=4



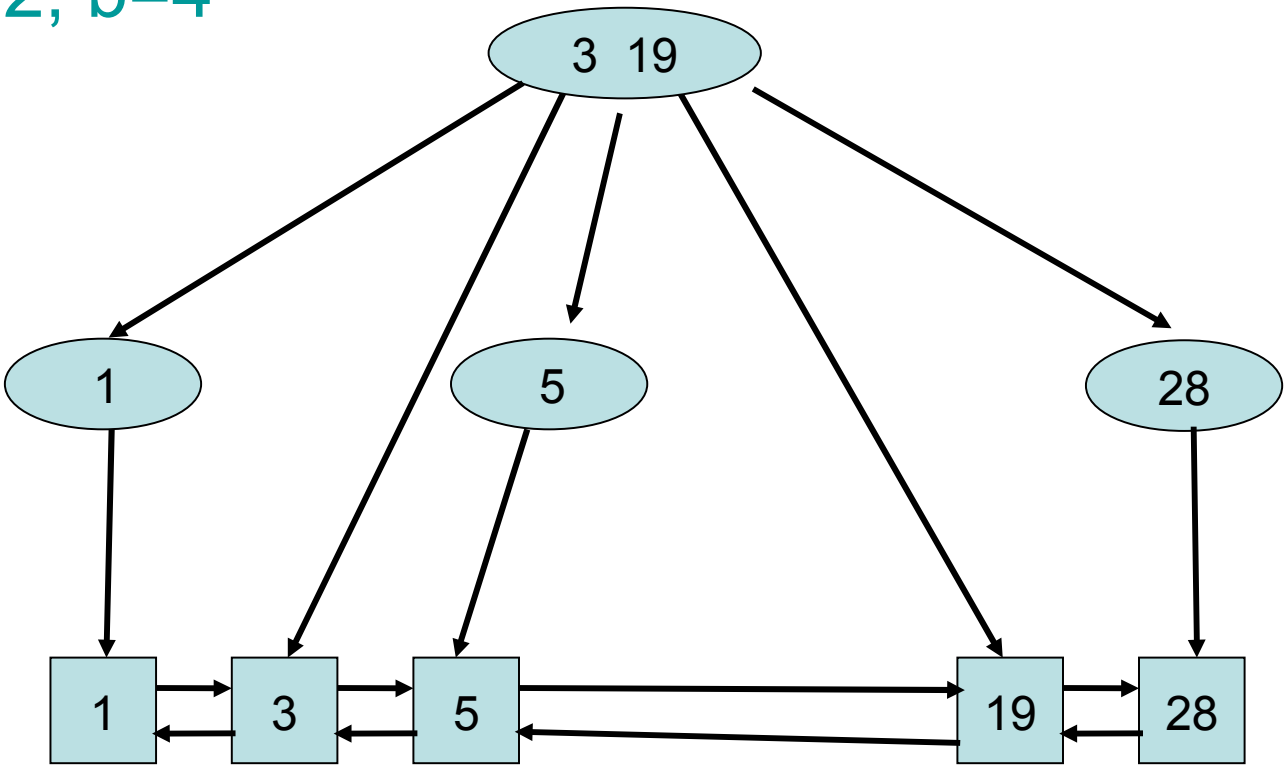
Remove(14)

a=2, b=4



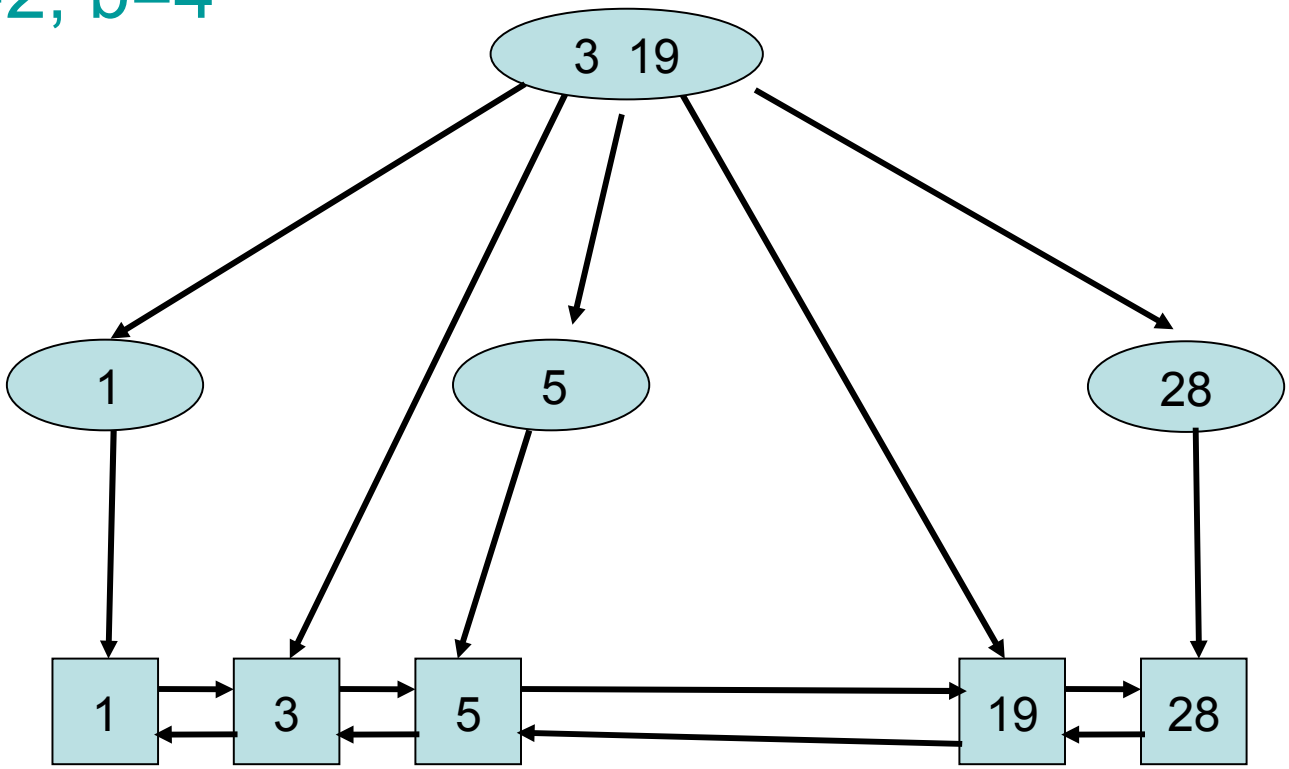
Remove(14)

a=2, b=4



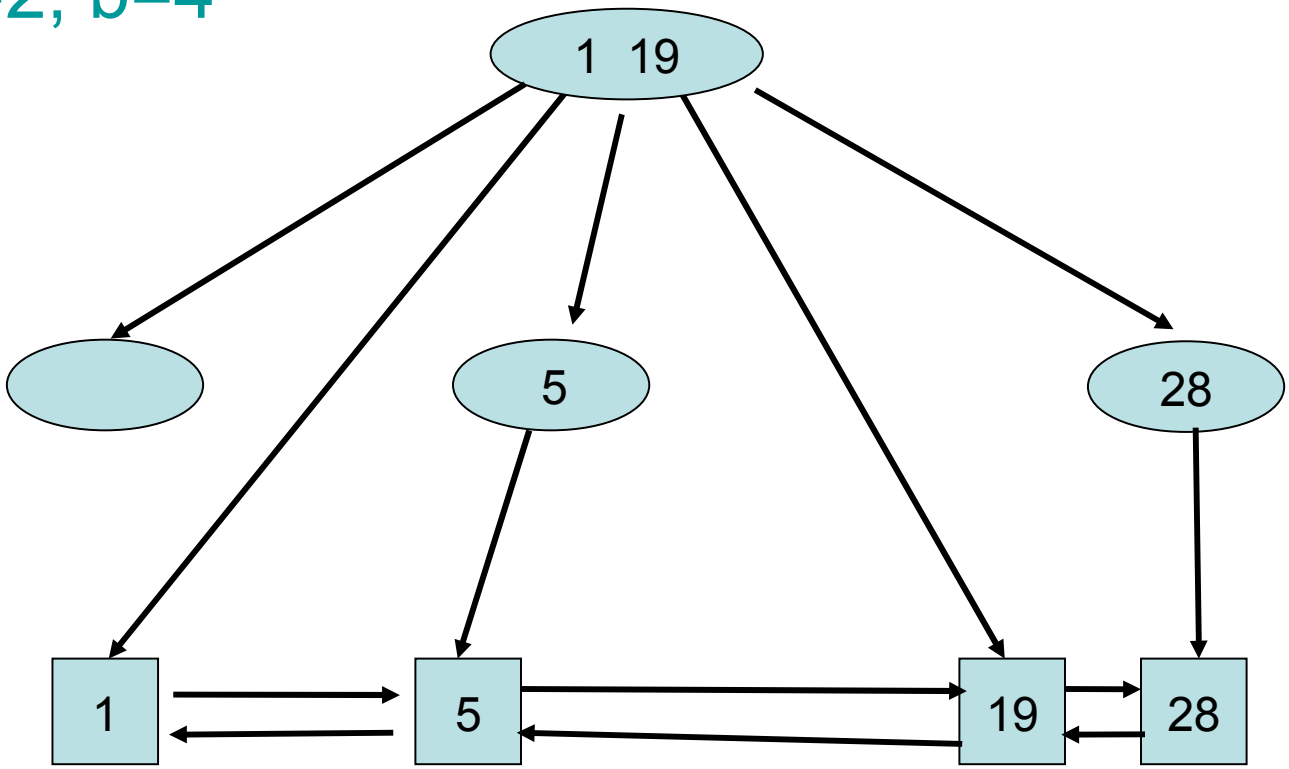
Remove(3)

a=2, b=4



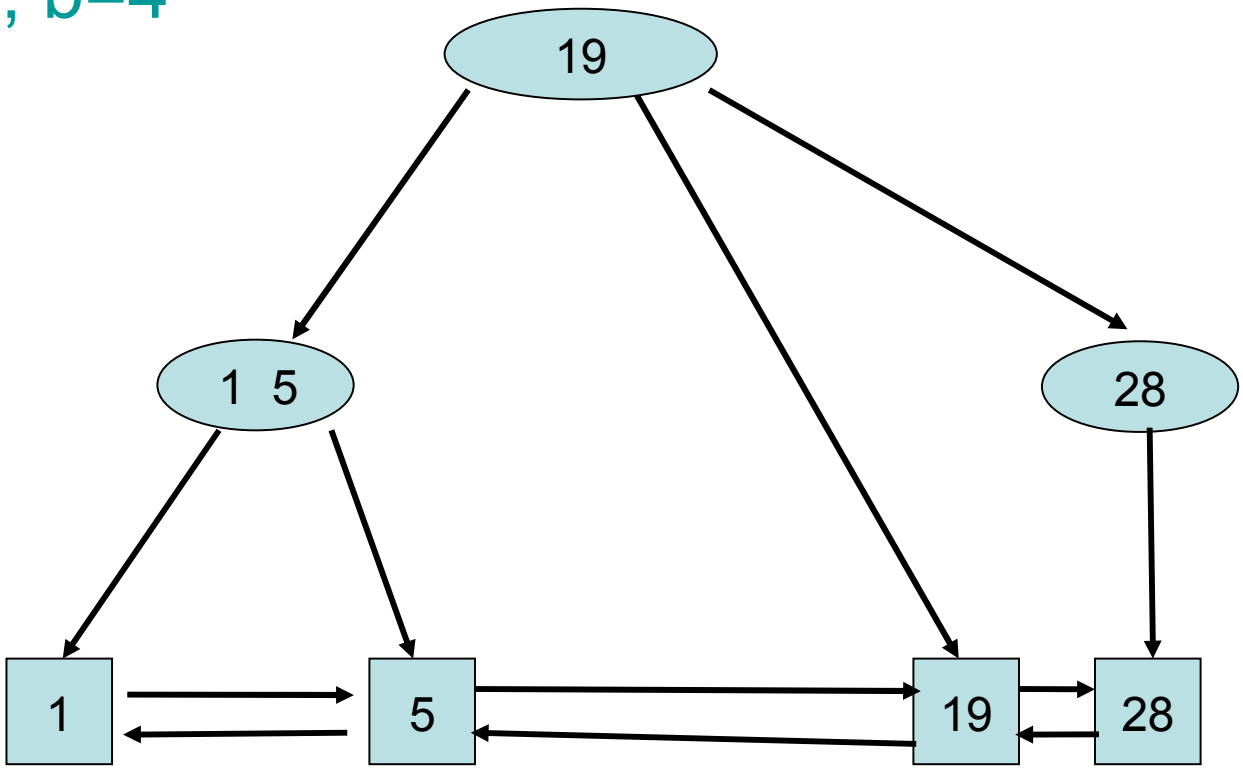
Remove(3)

a=2, b=4



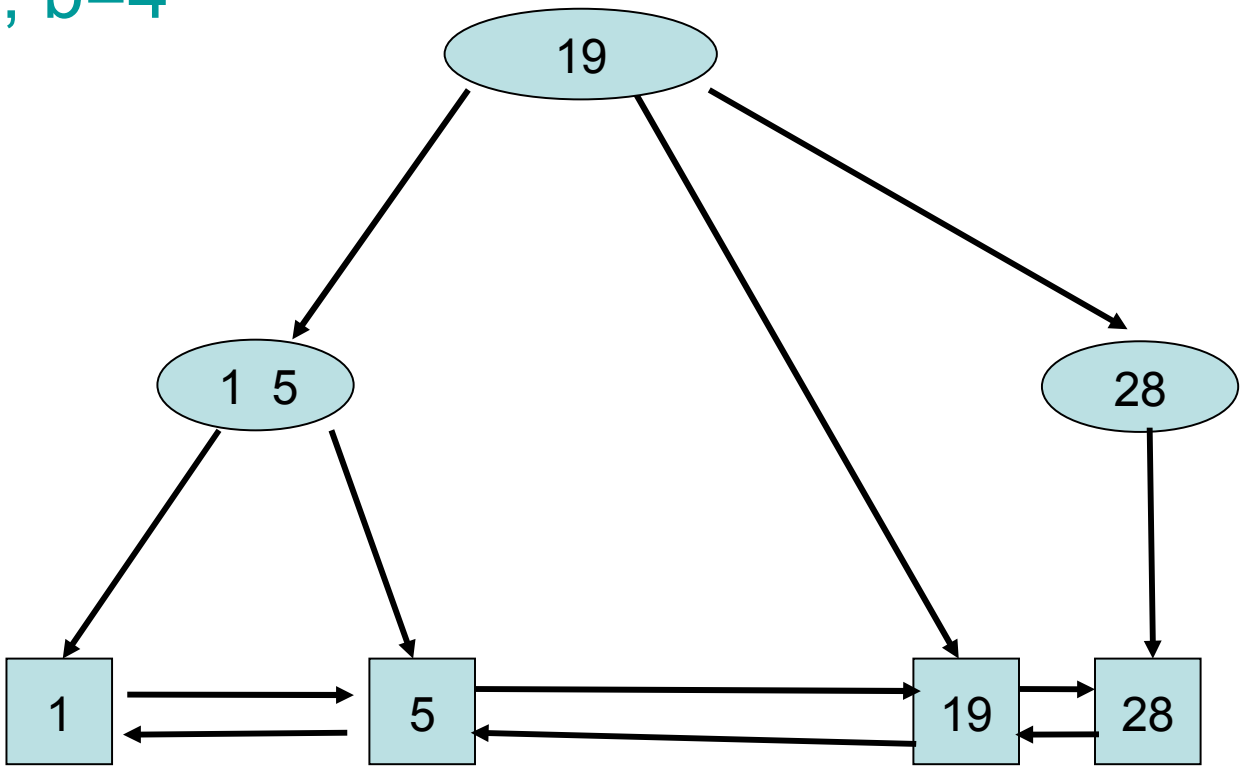
Remove(3)

a=2, b=4



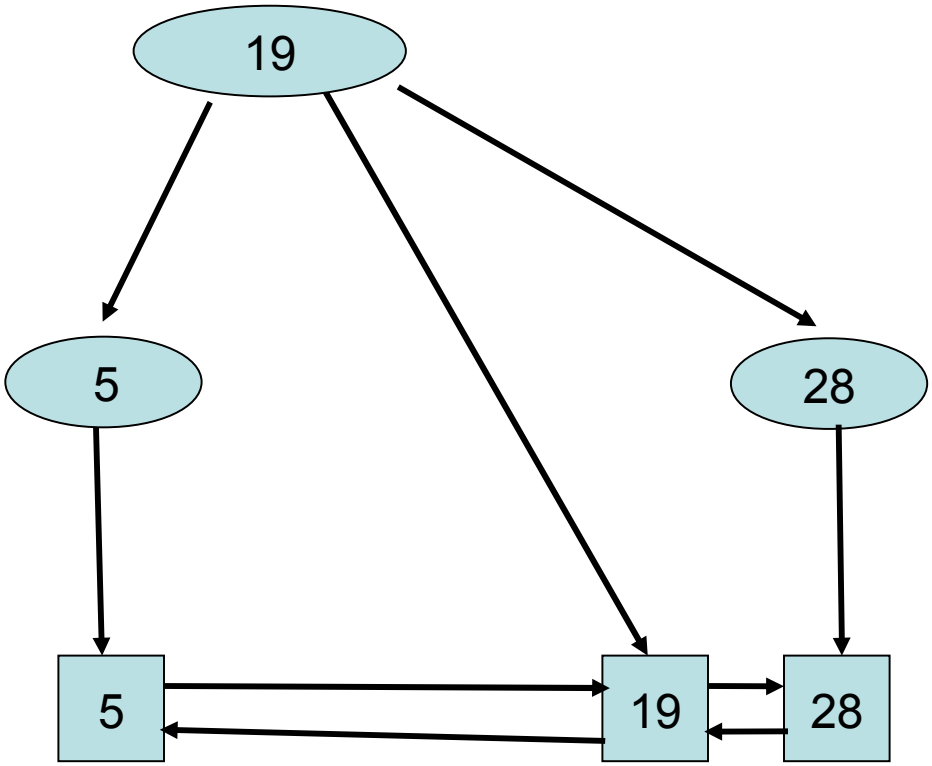
Remove(1)

a=2, b=4



Remove(1)

a=2, b=4



Remove Operation

- Form-Invariante:
  Für alle Blätter v,w: t(v)=t(w)
  Erfüllt durch Remove!

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

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

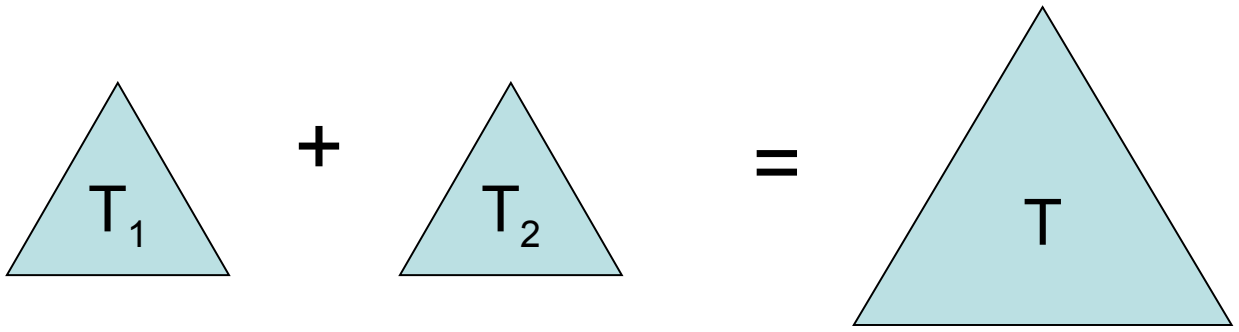
Mehr Operationen

- min/max Operation:
  Verwende die first und last Operationen, um das kleinste oder größte Element auszugeben. Zeit O(1).

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

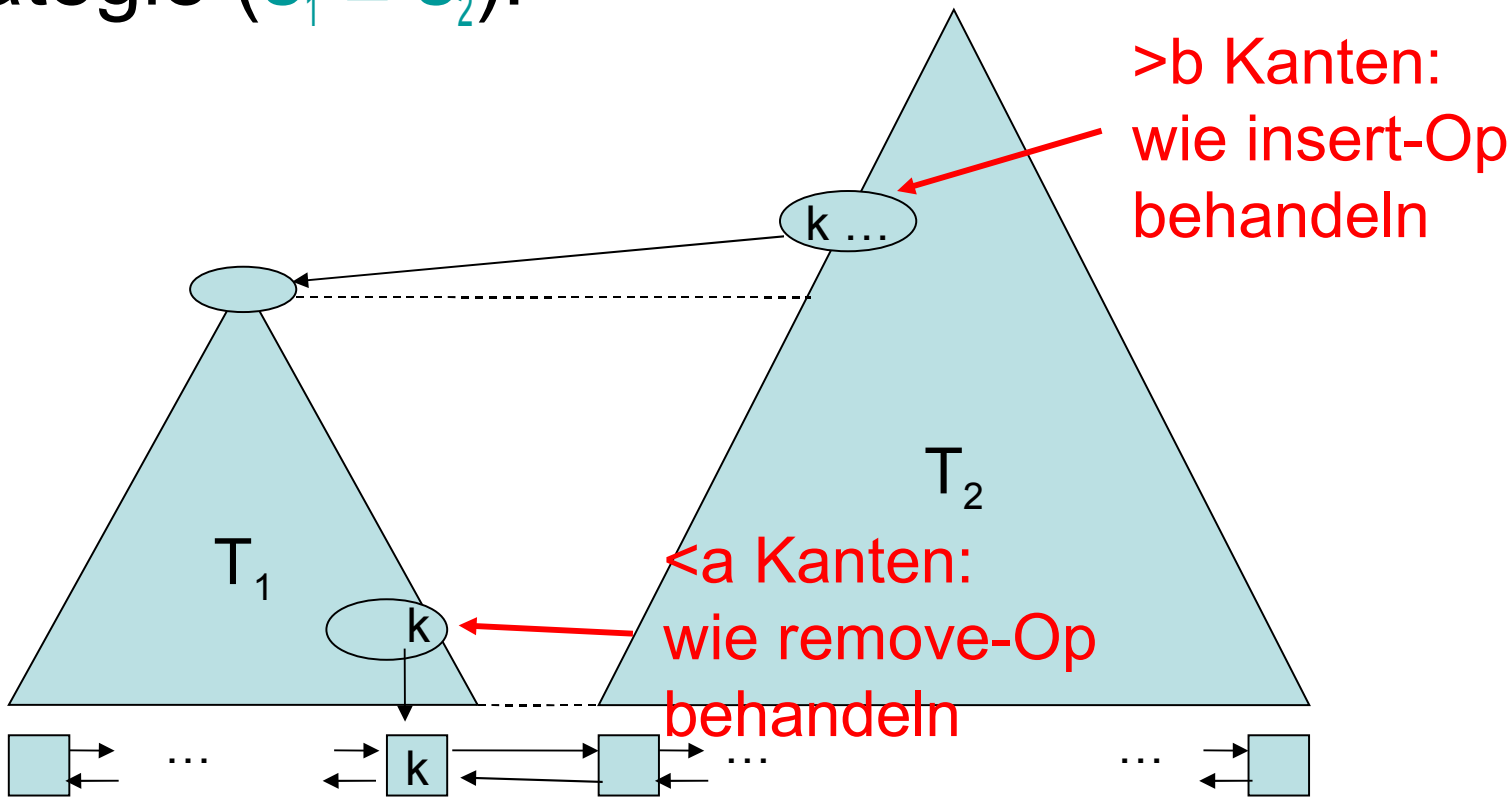
Mehr Operationen

- Konkatenation:
  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)



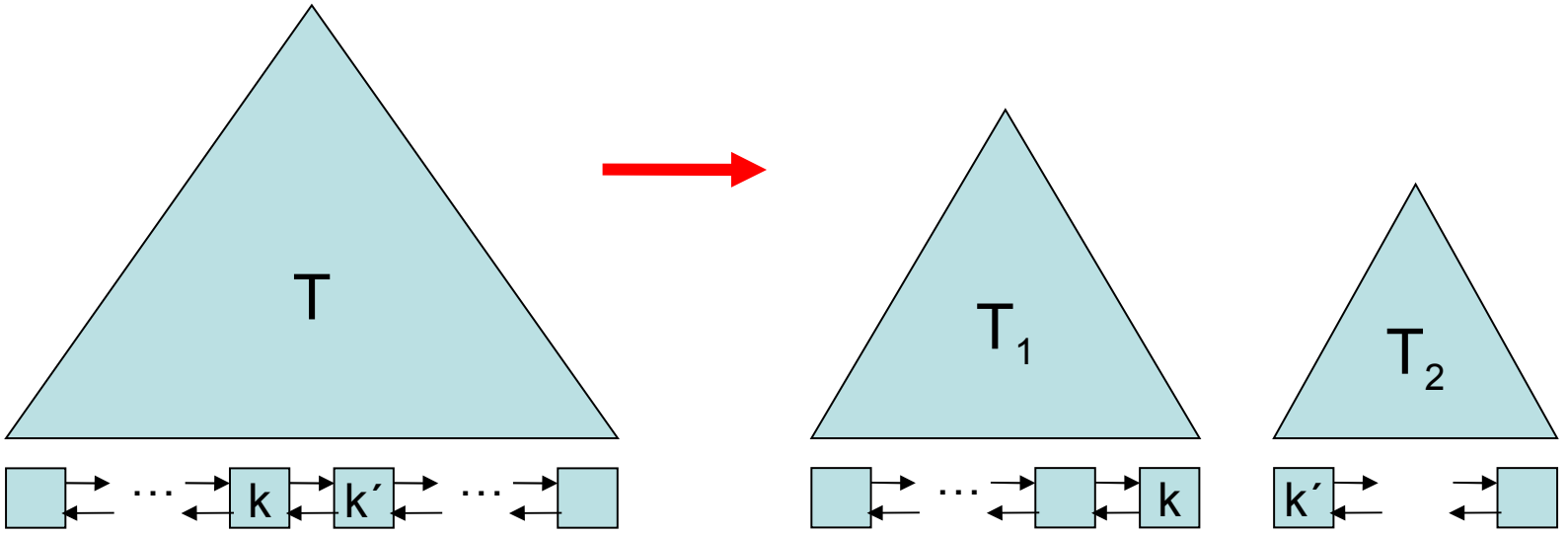
Mehr Operationen

- Konkatenation:
  Strategie ($s_1 \leq s_2$):



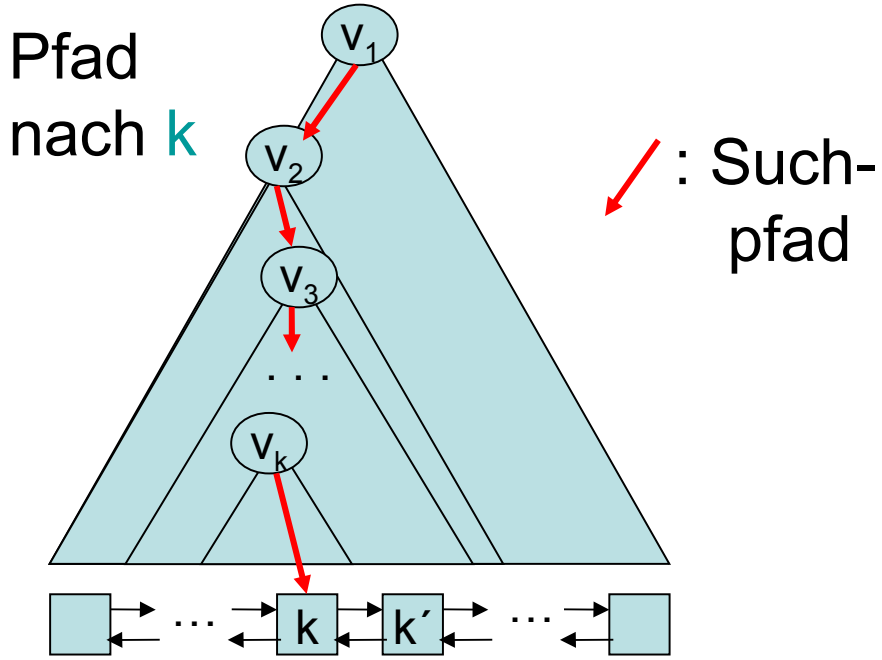
Mehr Operationen

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



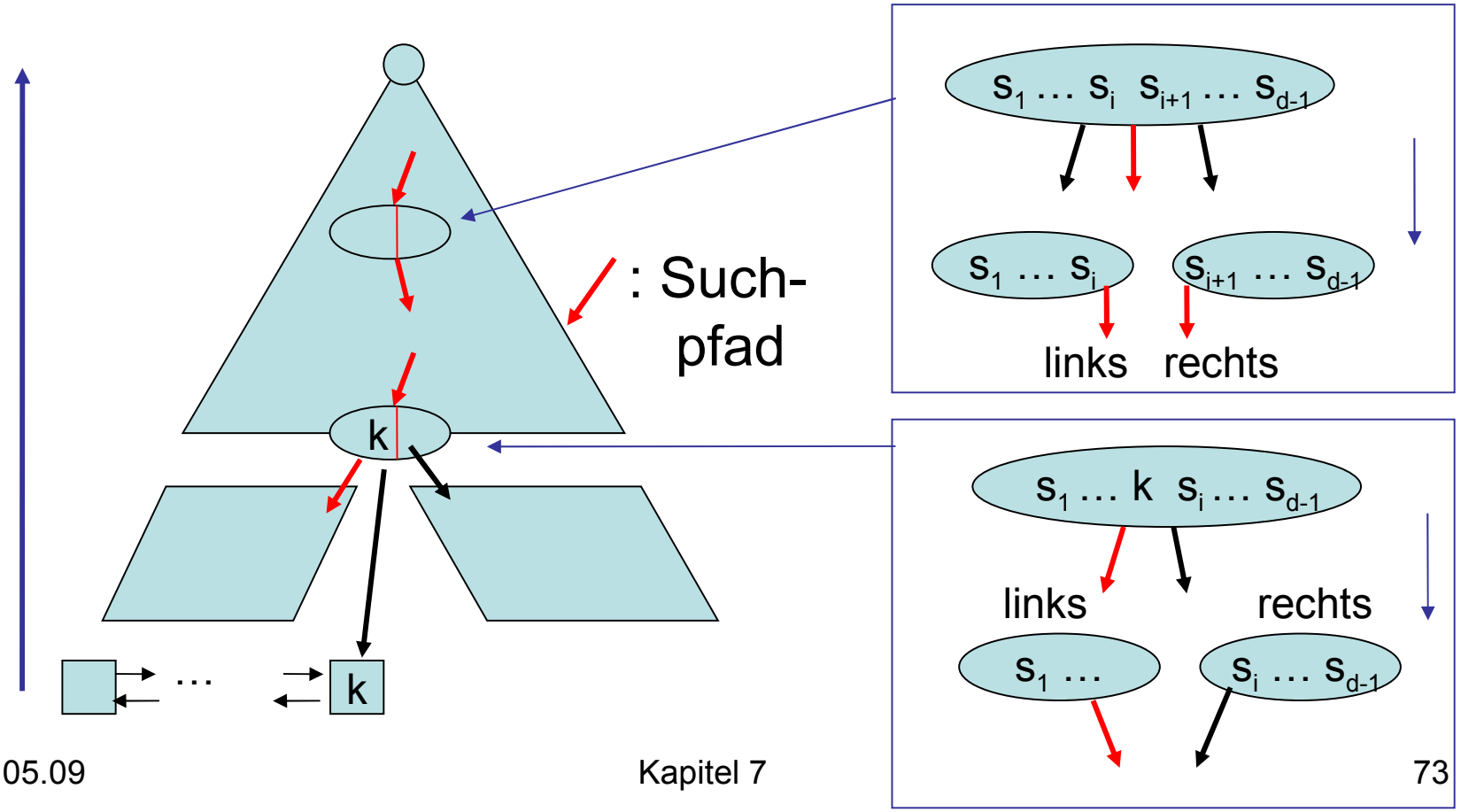
Split-Operation

1. Suche nach k



Split-Operation

2. Aufspaltung entlang Suchpfad

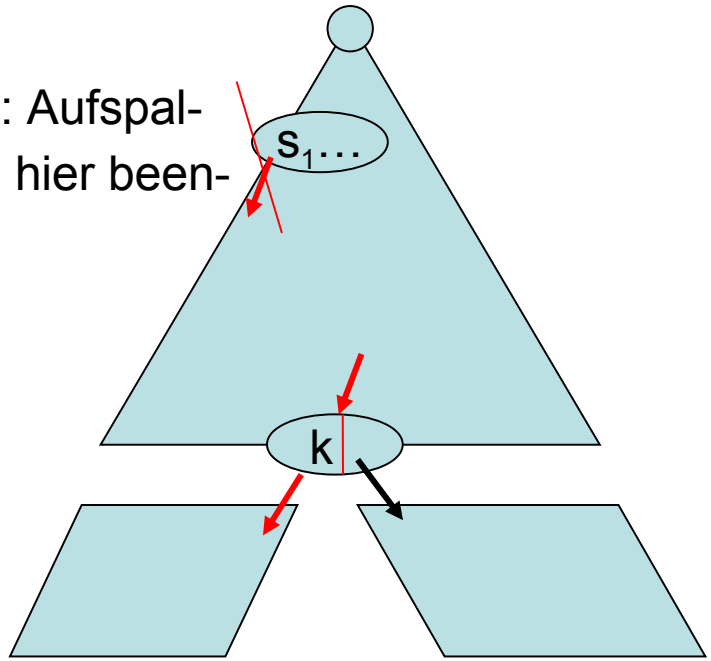


Split-Operation

2. Abbruch bei Aufspaltung:

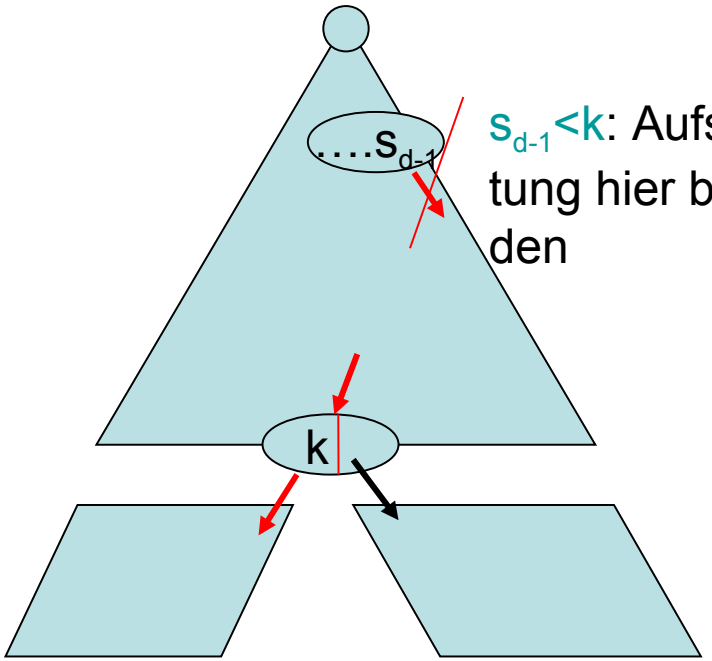
Fall 1:

s_1>k: Aufspaltung hier beenden



Fall 2:

s_{d-1}<k: Aufspaltung hier beenden



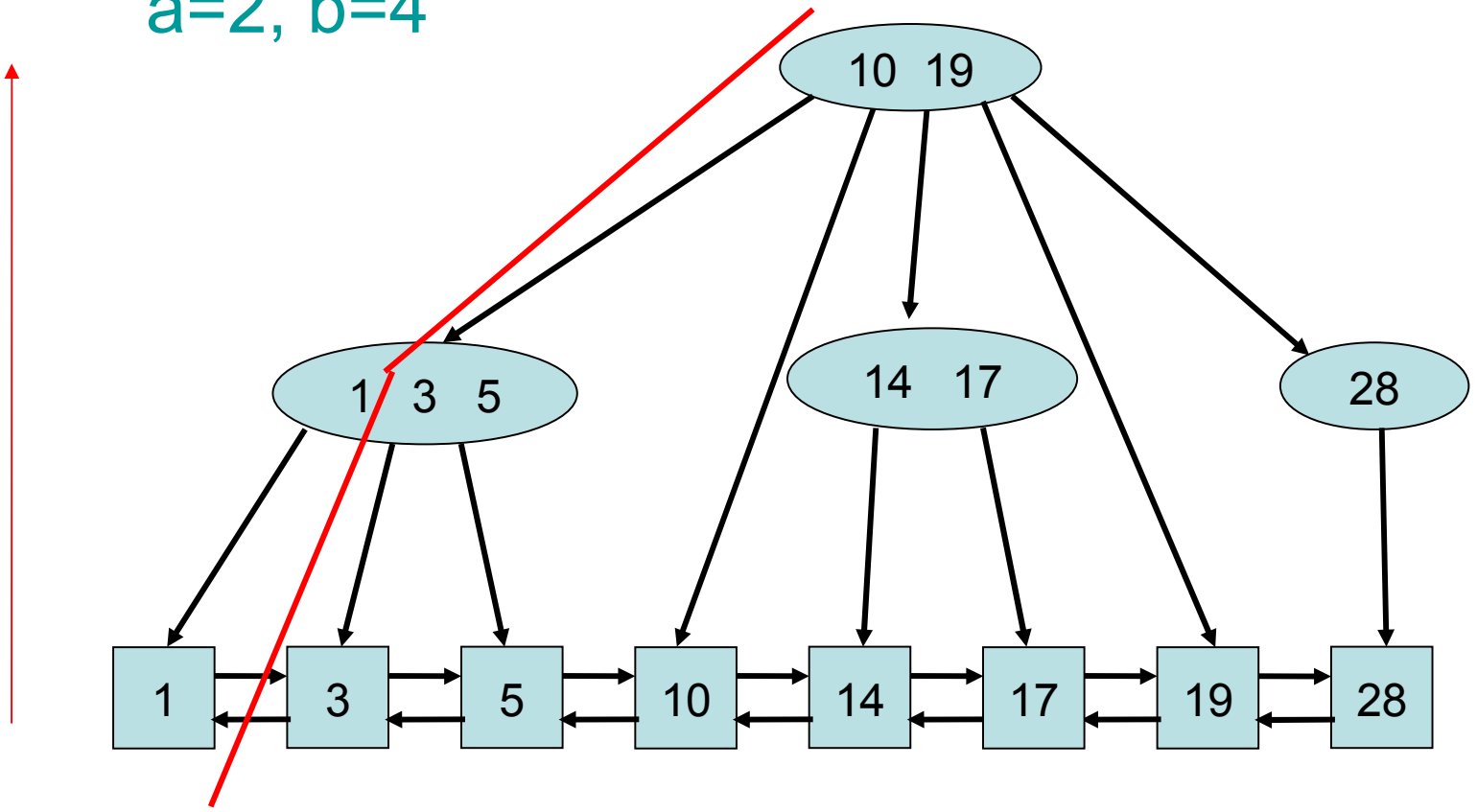
Split-Operation

1. Gradreparatur in den beiden Teilbäumen

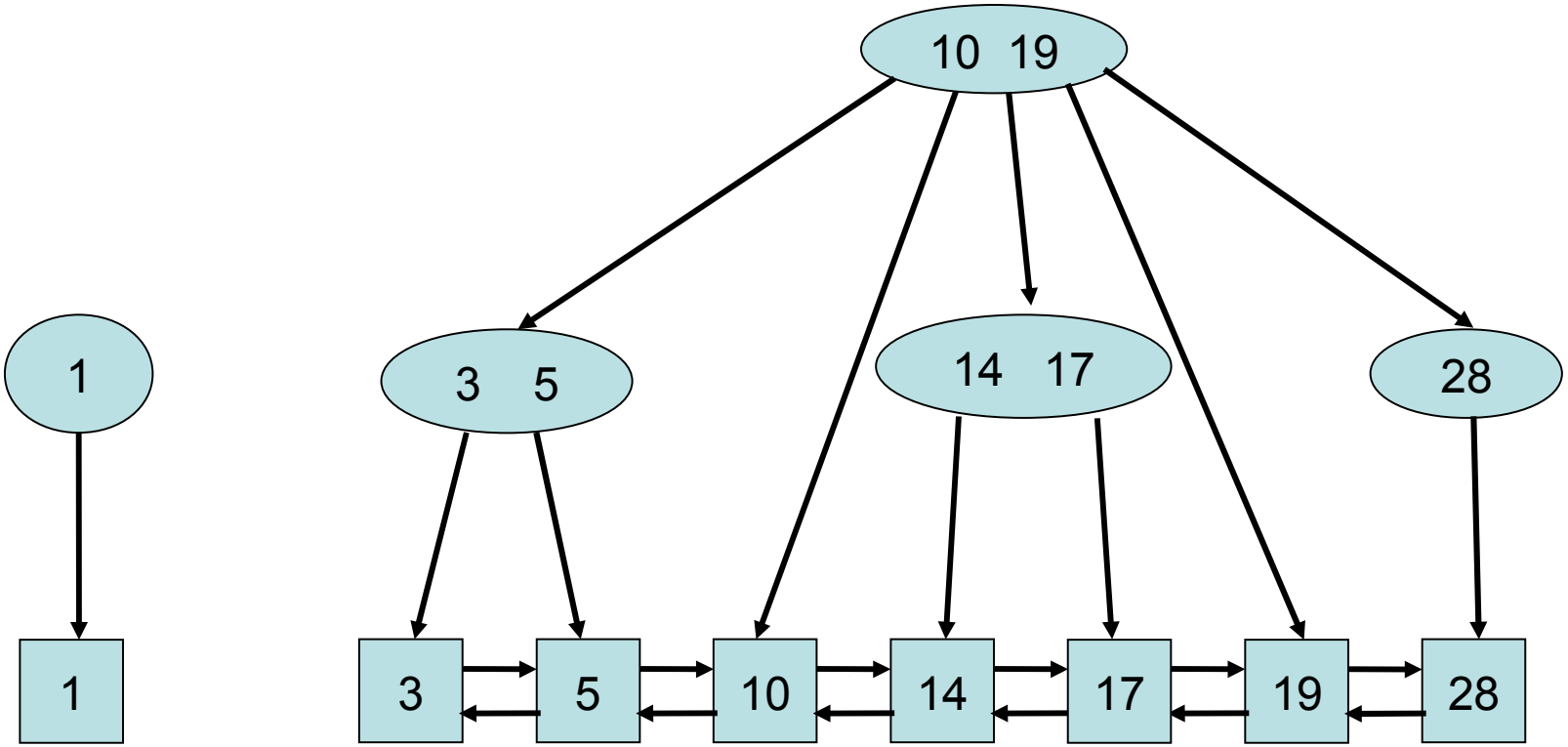
Strategie: bottom-up entlang Suchpfad, um zu gültigen (a,b)-Bäumen zurückzukeh-ren. (Wie bei insert und remove Operationen.)

Split(1)

a=2, b=4

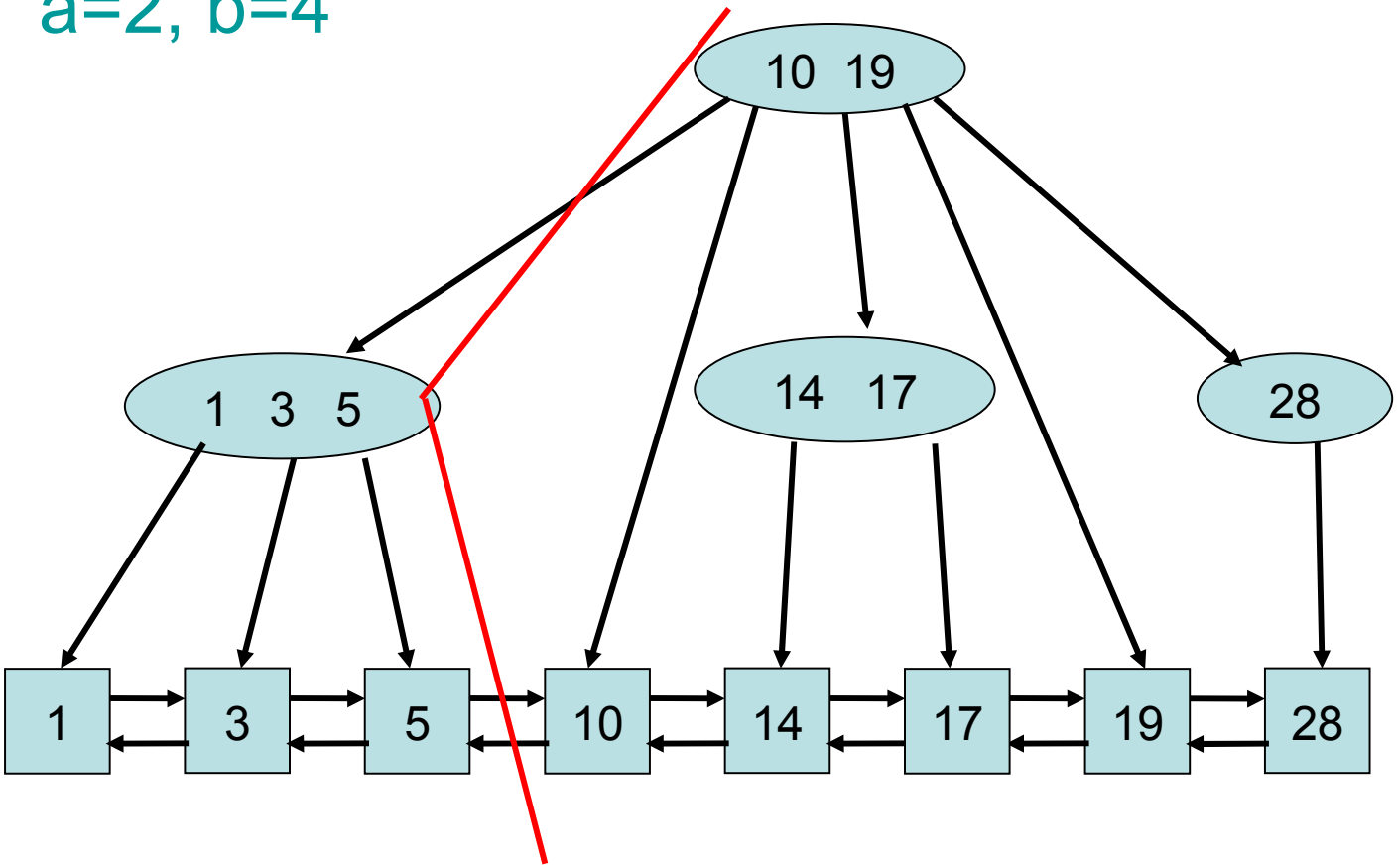


Split(1)

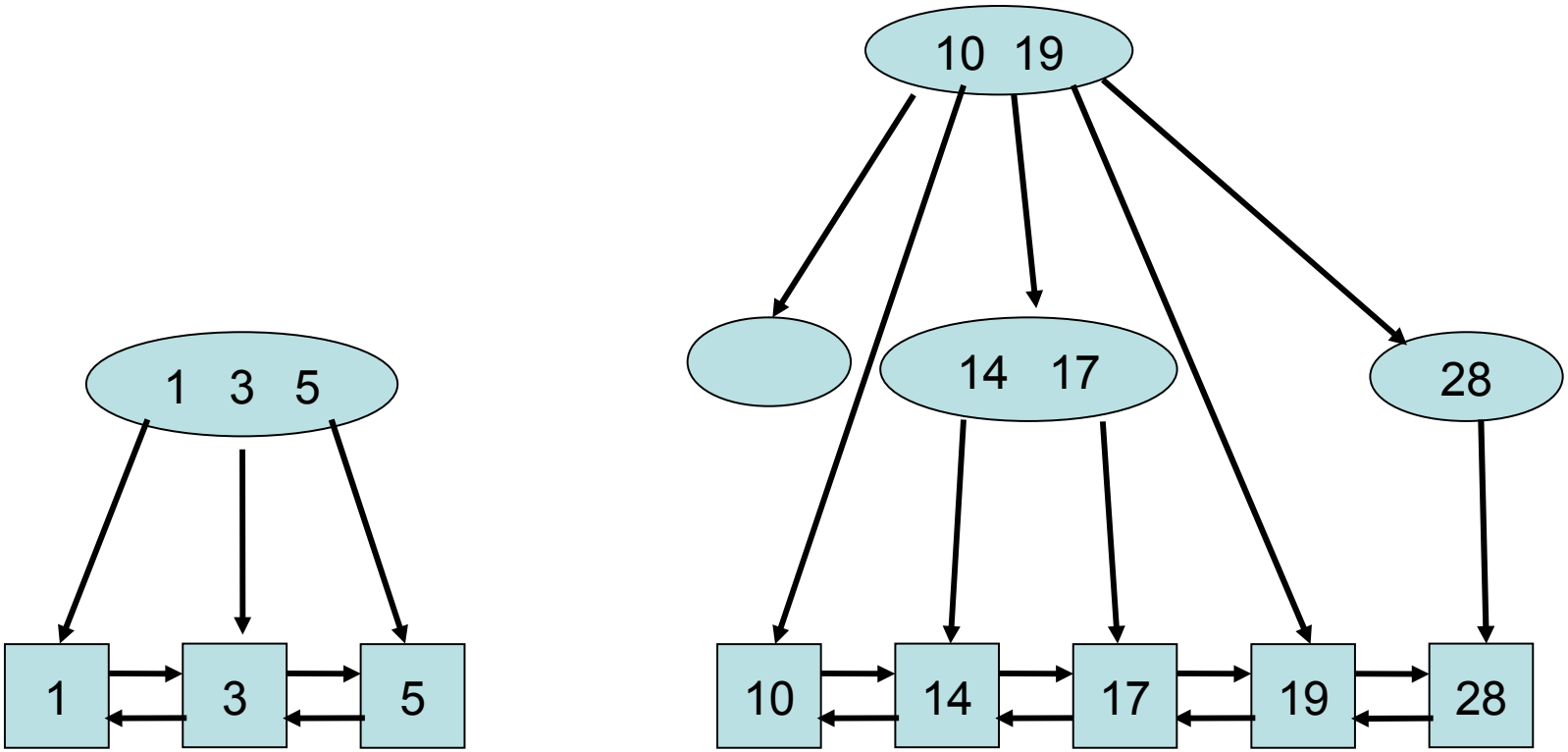


Split(5)

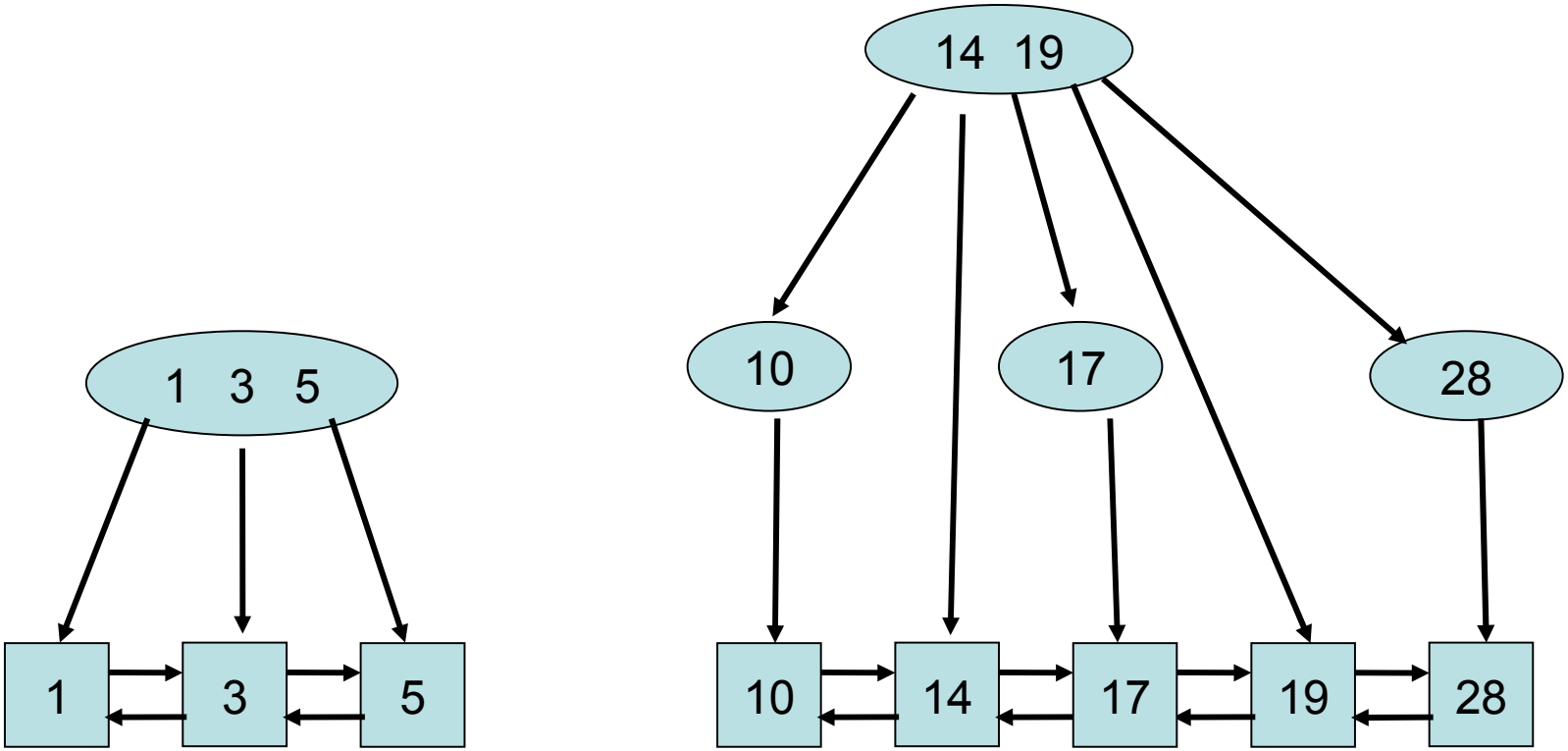
a=2, b=4



Split(5)

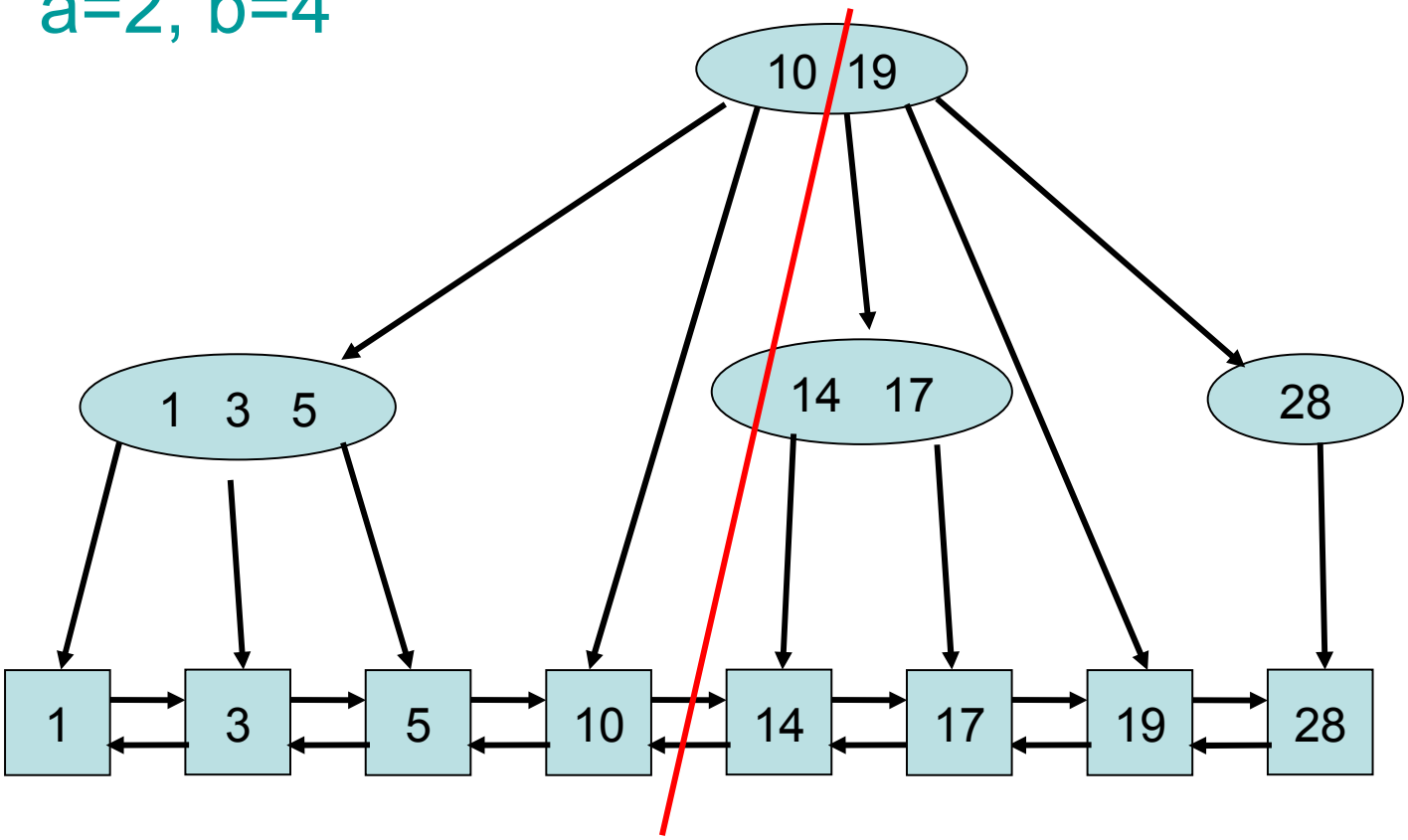


Split(5)

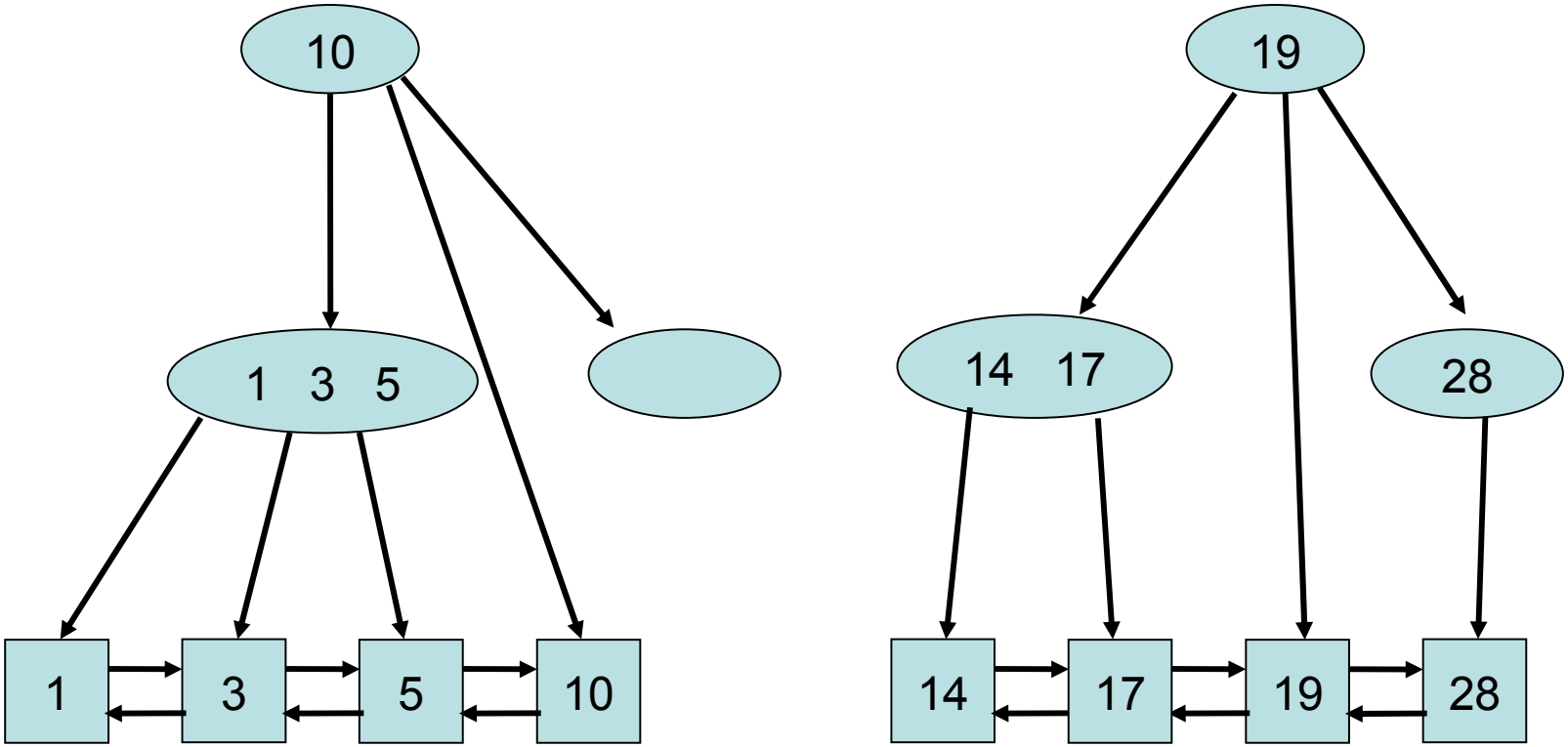


Split(10)

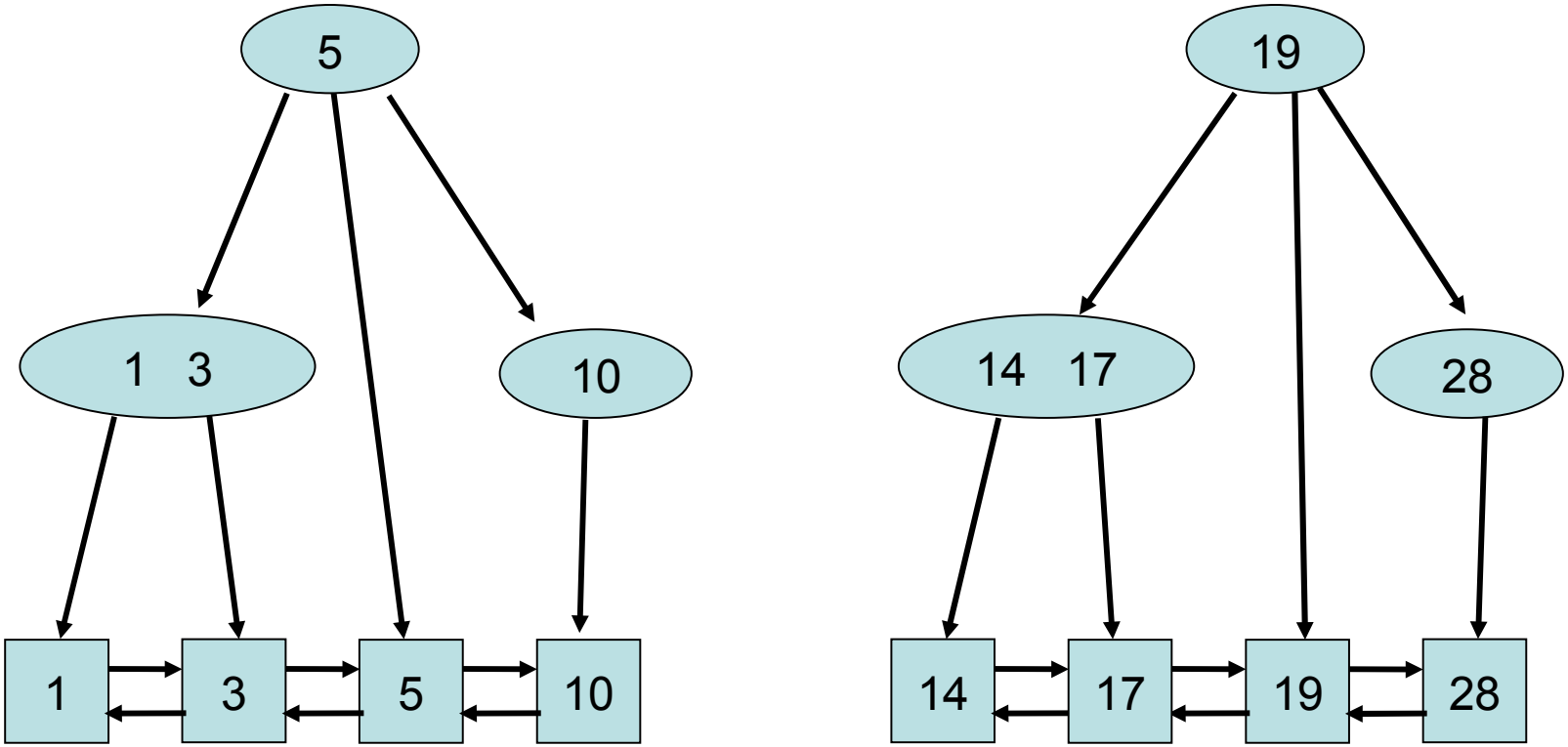
a=2, b=4



Split(10)



Split(10)



n Update-Operationen

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

Beweis: Übung.

n Update-Operationen

Theorem 7.3: Betrachte einen (a,b)-Baum mit $b \geq 2a$, der anfangs leer ist. Für jede Folge von n insert und remove Operationen ist die Gesamtanzahl der split und merge Operationen O(n).

Beweis:

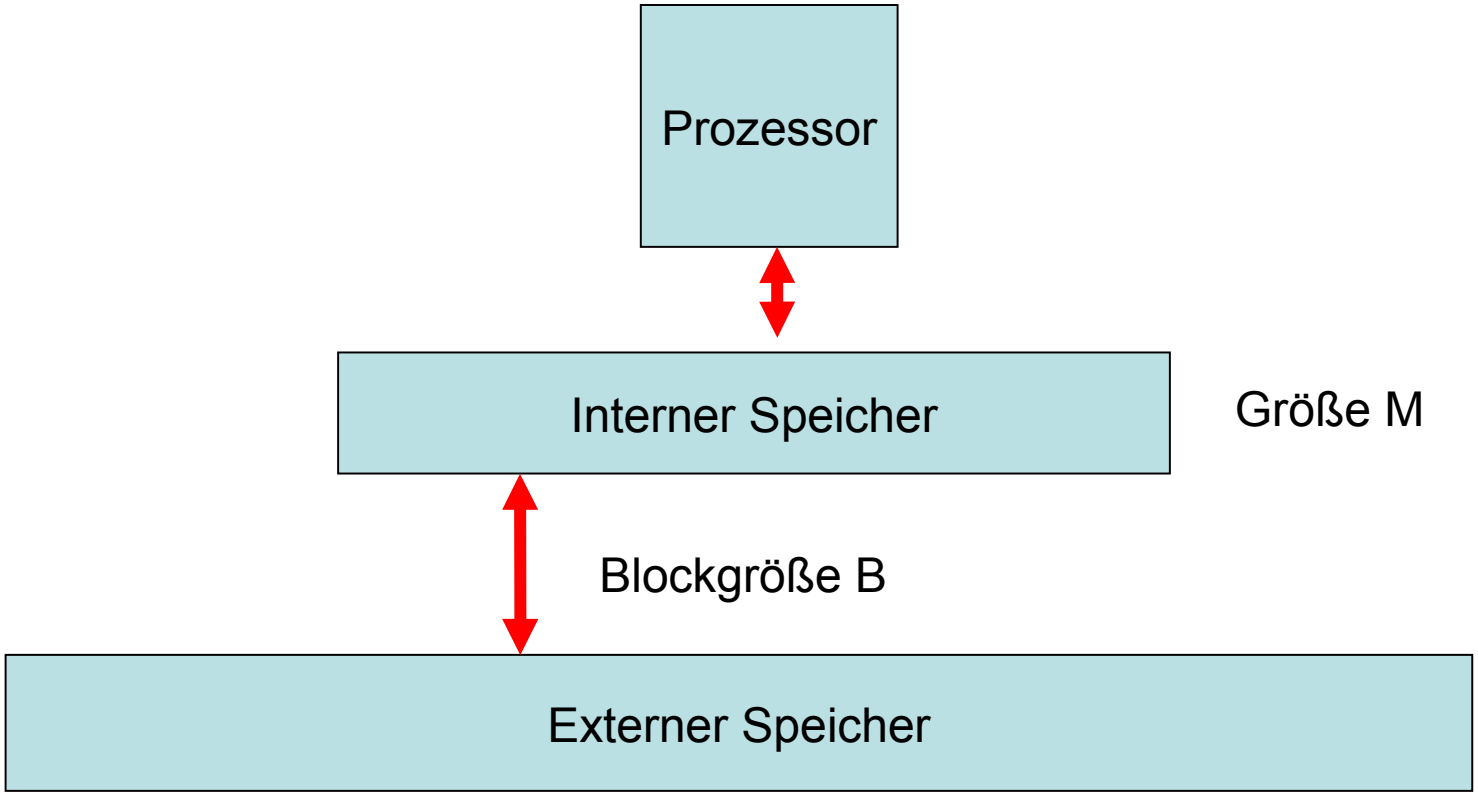
Amortisierte Analyse
(wird nicht in Vorlesung behandelt)

Zusammenfassung

- Binäre Suchbäume
- (a,b)-Bäume
- Zusätzliche Operationen

- Weiter mit Graphrepräsentationen (Kapitel 8)

Externer (a,b)-Baum



Externer (a,b)-Baum

Problem: Minimiere Blocktransfers zwischen internem und externem Speicher

Lösung:

- verwende $b=B$ (Blockgröße) und $a=b/2$
- halte oberste $\log_a(M/b)$ Ebenen des (a,b)-Baums im internen Speicher (Speicher $<= M$)
- Lemma 7.1: Tiefe des (a,b)-Baums max.$1+\lfloor \log_a (n+1)/2 \rfloor$
- $\log_a[(n+1)/2] - \log_a(M/b) = \log_a[(n+1)/(2M)] + \log_a b$
- $\log_a b = O(1)$
- Kosten für insert, remove und locate Operationen: $O(\log_B(n/M))$ Blocktransfers