RUB

Vorlesung Datenstrukturen

Sortierte Folgen

Maike Buchin

30.5., 1.6., 13.6.2017

Sortierte Folgen

RUB

Häufiges Szenario: in einer Menge von Objekten mit Schlüsseln (aus geordnetem Universum) sollen Elemente gesucht, eingefügt und gelöscht werden.

Gesucht: Datenstruktur, um dies effizient zu tun

Formal: Sortierte Folgen verwalten Mengen S von Einträgen e mit Schlüsseln $k = key(e) \in Key$ unter den Operationen

- S.locate$(k : Key)$ – gebe Element mit kleinstem Schlüssel größer gleich k in S zurück
- Q.insert$(e : Element)$ – falls bereits ein Element mit gleichem Schlüssel in S vorhanden, ersetze dieses durch e; andernfalls füge e zu S neu hinzu
- Q.remove$(k : Key)$ – lösche Element $e \in S$ mit $key(e) = k$ aus S, falls dies in S vorhanden

Wir werden zeigen: alle Operationen in $O(\log n)$.

Sortierte Folgen

RUB

Häufiges Szenario: in einer Menge von Objekten mit Schlüsseln (aus geordnetem Universum) sollen Elemente gesucht, eingefügt und gelöscht werden.

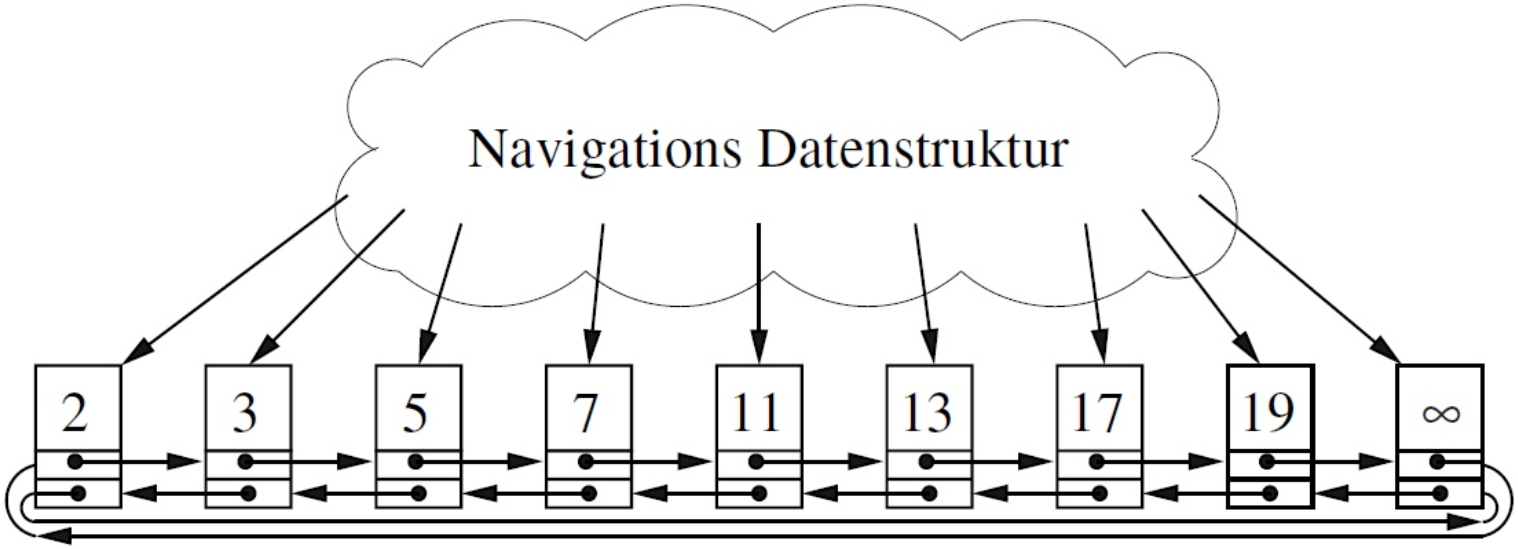
Gesucht: Datenstruktur, um dies effizient zu tun

Vergleich mit bekannten Datenstrukturen:

- flexibler und daher schneller als sortierte Arrays oder Listen
- langsamer, aber mächtiger als Hashtabellen: locate(k) funktioniert auch, falls k nicht vorhanden; sortierte Ausgabe in $O(n)$
- allgemeiner als Prioritätswarteschlangen: diese erlauben nur das minimale Element effizient zu finden und löschen

Sortierte Folgen

Grundsatz:



doppelt-verkette Liste

- Navigations Datenstruktur erlaubt effizientes locate
- doppelt verkette Liste erlaubt effizientes Durchlaufen

Häufigste Navigations Datenstruktur: **Suchbäume**

Sortierte Folgen

RUB

Beispiel Anwendung: Best first Heuristik für online binpacking

Gegenstände versch. Größe sollen in möglichst wenige Behälter (engl. bins) mit bestimmter Maximalkapazität gepackt werden.
Online Szenario: Gegenstände kommen nach und nach an.
Best-first Heuristik: Tue Gegenstand in genügend großen Behälter mit minimaler Restkapazität.

$\rightarrow$ verwende Sortierte Folge von Behälter mit Restkapazitäten als Schlüsseln.

7.1 Binäre Suchbäume

RUB

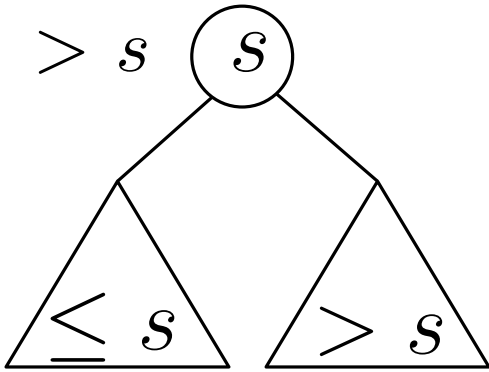
Ein binärer Suchbaum ist ein binärer Baum in dessen Blätter die Einträge einer sortierten Folge gespeichert sind. In inneren Knoten stehen sog. 'Spaltschlüssel', welche die Suche steuern.

In einem binären Suchbaum mit mind. zwei Blättern hat jeder innere Knoten genau zwei Kinder: rechtes und linkes Kind.

Für Spaltschlüssel s an innerem Knoten gilt: alle Schlüssel sind im linken Unterbaum $\leq s$ und im rechten Unterbaum $> s$.

Suche nach Schlüssel k: **Laufzeit:** $O(\text{Höhe}) = O(n)$

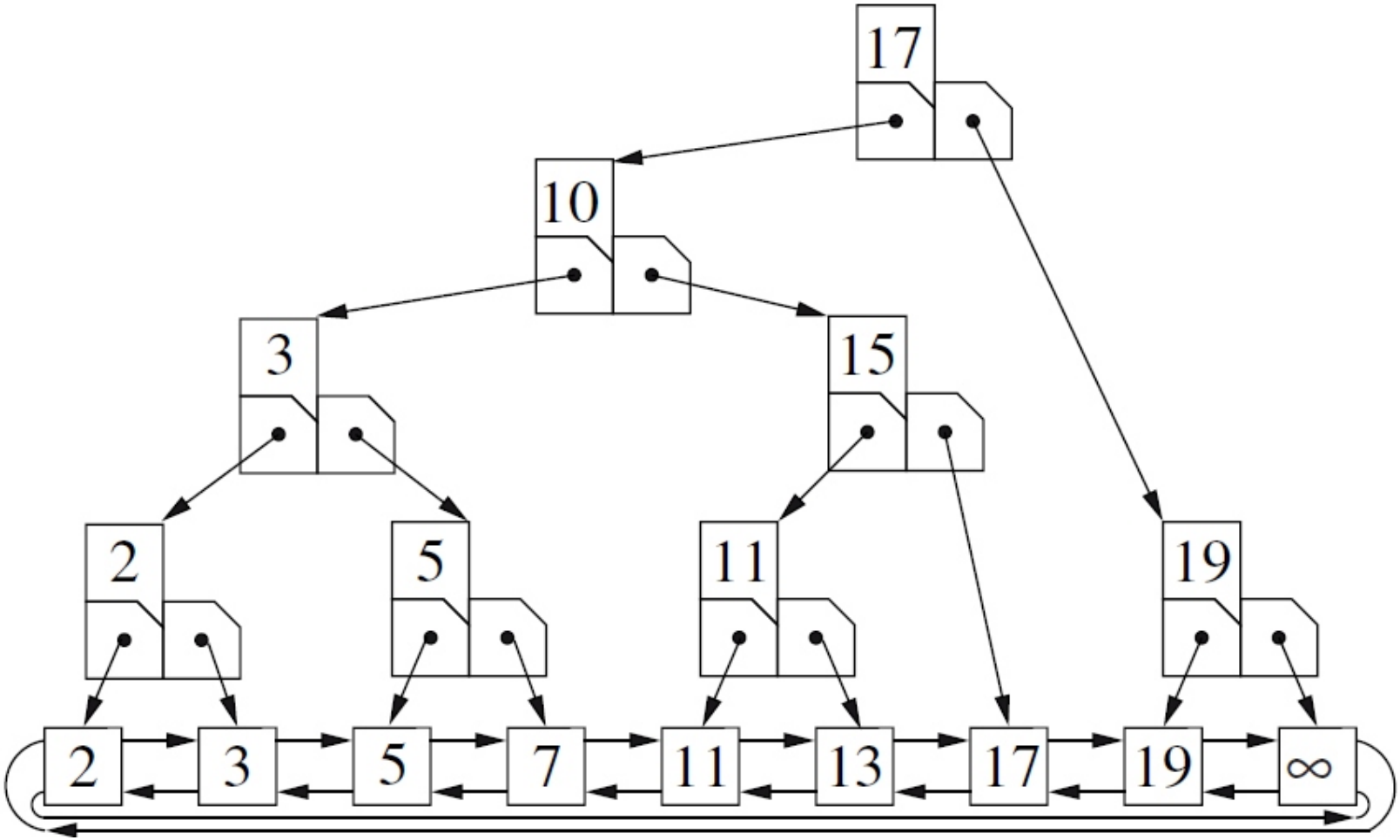
beginne in der Wurzel und wiederhole bis Blatt erreicht:
gehe zum linken bzw. rechten Kind falls k $\leq$ bzw. $> s$



Korrektheit folgt aus der Invariante: Unterbaum mit aktuellem Knoten als Wurzel enthält kleinsten Schlüssel $k' \geq k$ oder rechtestes Blatt ist Listenvorgänger von k.
(falls Spaltschlüssel auch Blätter gilt sogar nur erster Teil der Invariante)

RUB

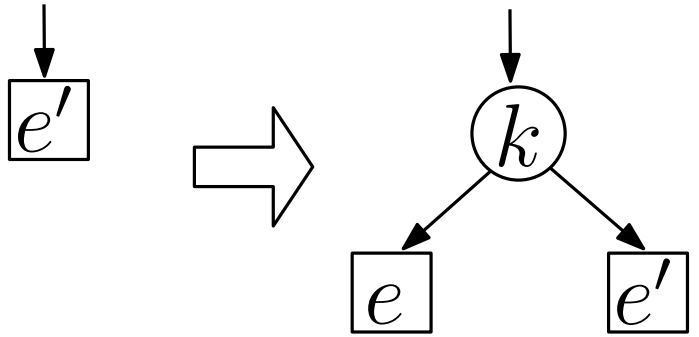
Beispiel:



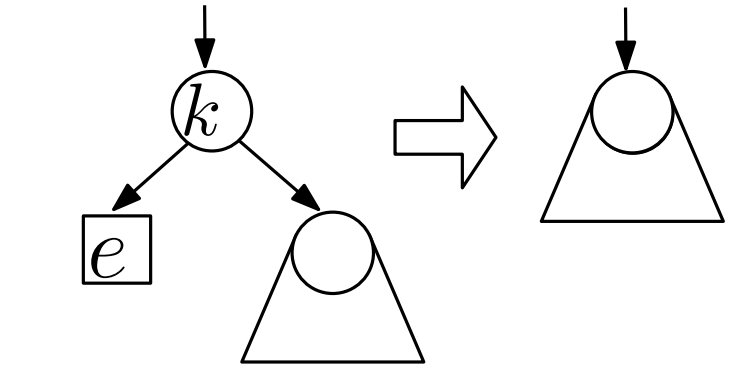
Suche nach Schlüsseln $1, 9, 13, 25$

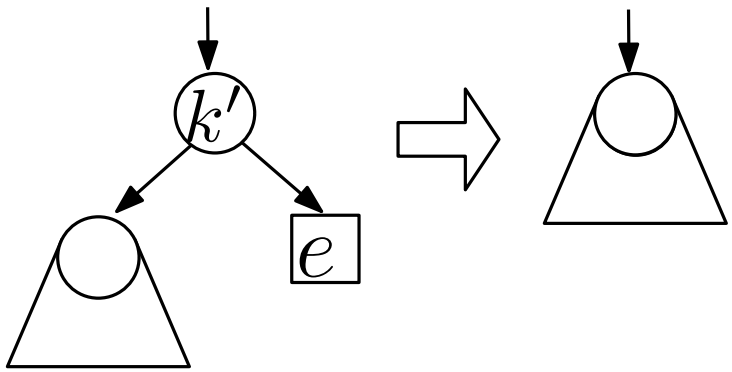
7.1 Binäre Suchbäume

insert(e)
 $e' =$ locate(key(e))
 if key$(e) =$ key(e') then
  replace e' by e
 else
  replace leaf by new inner node with children e, e'



remove(k)
 $e =$ locate(k)
 if key$(e) = k$ then
  if e is leftchild then
   replace parent(e) with its right child
  else
   replace parent(e) with its left child
   replace k with k' on path to root





RUB

Balancierte Binäre Suchbäume

Ein Binärbaum mit n Blättern hat Höhe zwischen $\lceil \log n \rceil$ und n.
Ein binärer Suchbaum mit Tiefe

- $\lceil \log n \rceil$ heißt *perfekt balanciert*
- $O(\log n)$ heißt *balanciert*

Frage: Wie kann man einen binären Suchaum balancieren?

Eine Lösung: Optimismus, d.h. bei Einfügen in zufälliger Reihenfolge ist die mittlere Tiefe $O(\log n)$.
Skizze: Einfügen in binären Suchbaum ist analog zum Aufteilen einer Folge mit Quicksort.

RUB

Balancierte Binäre Suchbäume

Ein Binärbaum mit n Blättern hat Höhe zwischen $\lceil \log n \rceil$ und n.
Ein binärer Suchbaum mit Tiefe

- $\lceil \log n \rceil$ heißt *perfekt balanciert*
- $O(\log n)$ heißt *balanciert*

Frage: Wie kann man einen binären Suchaum balancieren?

Eine Lösung: Optimismus, d.h. bei Einfügen in zufälliger Reihenfolge ist die mittlere Tiefe $O(\log n)$.
Skizze: Einfügen in binären Suchbaum ist analog zum Aufteilen einer Folge mit Quicksort.

Andere Lösung: Es gibt mehrere Methoden, eine logarithmische Tiefe auch im schlechtesten Fall zu erreichen:

- (a, b)-Bäume – unterschiedliche Grade
- rot-schwarz Bäume } Rotationen
- AVL-Bäume } Rotationen

7.2 (a, b) Bäume

RUB

Ein $(a, b)-$**Baum** ist ein Suchbaum in dem alle inneren Knoten ausser der Wurzel Grad zwischen a und b haben. Die Wurzel hat Grad 1 nur im Falle eines trivialen Baumes mit einem Blatt; sonst hat die Wurzel Grad zwischen 2 und b.

Wenn $a \geq 2$ und $b \geq 2a - 1$ erlaubt uns die Flexibiltät bei der Wahl der Grade effizient die **Invariante** sicher zu stellen, dass alle Blätter gleiche Tiefe haben.

7.2 (a, b) Bäume

RUB

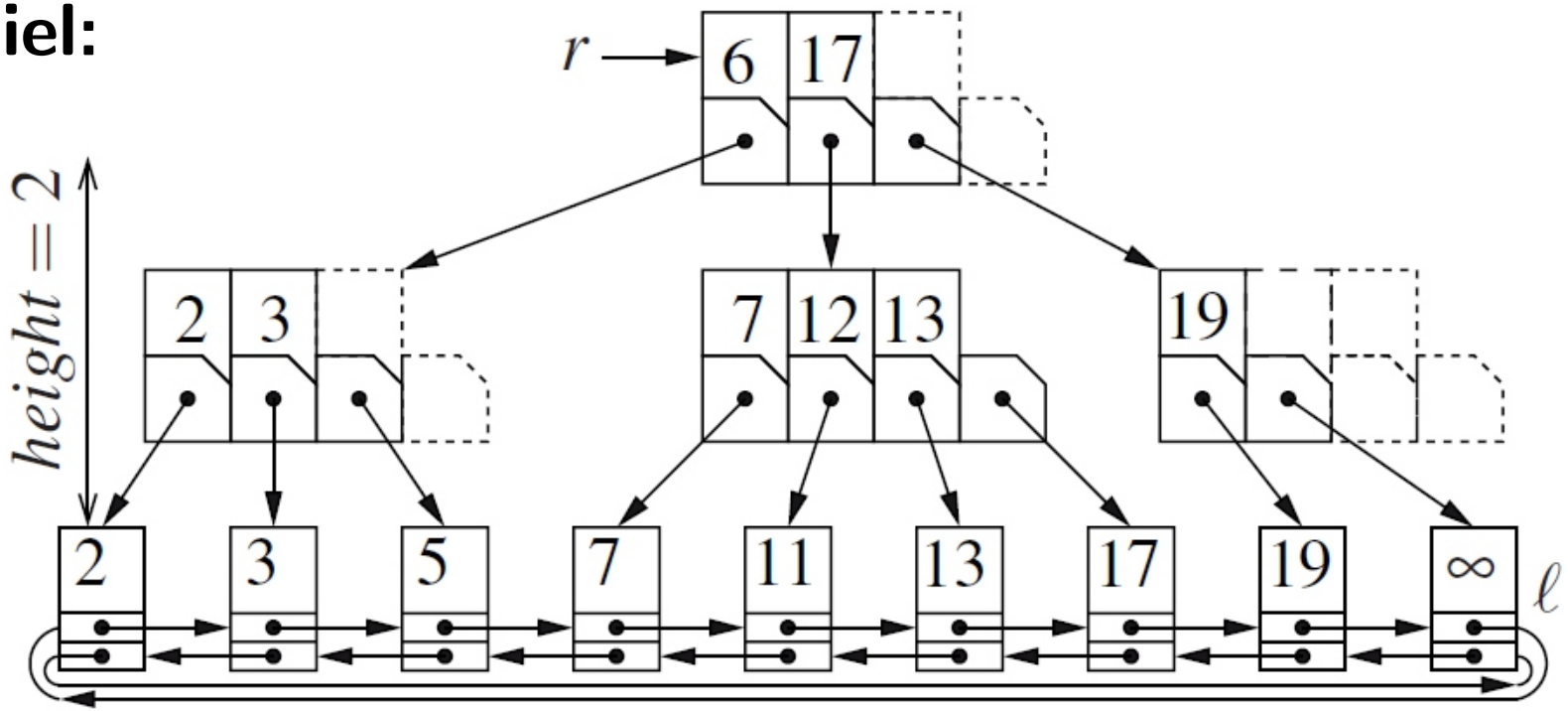
Ein $(a, b)-$**Baum** ist ein Suchbaum in dem alle inneren Knoten ausser der Wurzel Grad zwischen a und b haben. Die Wurzel hat Grad 1 nur im Falle eines trivialen Baumes mit einem Blatt; sonst hat die Wurzel Grad zwischen 2 und b.

Betrachte **Knoten mit Grad** d.
Diesem ordnen wir ein Array $c[1 \ldots d]$ von Kindzeigern und ein sortiertes Array $s[1 \ldots d-1]$ von Spaltschlüsseln zu.
Zur Vereinfachung definieren wir $s[0] = 0$ und $s[d] = \infty$.
Wir verlangen, dass für $1 \leq i \leq d$ die Schlüssel der Einträge im Unterbaum mit Wurzel $c[i]$ zwischen dem $(i-1)$-ten (ausschließlich) und dem i-ten (einschließlich) Spaltschlüssel liegen.

Zur Vereinfachung der Implementierung enthält ein (a, b)-Baum immer einen **Dummyknoten** mit Schlüssel ∞ als rechtestes Blatt.

7.2 (a, b) Bäume

Beispiel:



7.2 (a, b) Bäume

RUB

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

Beweis: Der Baum hat $n+1$ Blätter ($+1$ für Dummyblatt ∞). Falls $n = 0$ hat die Wurzel Grad 1 und es gibt ein Blatt. Falls $n \geq 1$ hat die Wurzel Grad ≥ 2. Sei h Tiefe des Baumes. Wegen Grad ≥ 2 bzw. $\geq a$ sonst, ist die Anzahl Blätter $n + 1 \geq 2a^{h-1}$. Also gilt $h \leq 1 + \log_a(n + 1/2)$. Zudem ist h ganzzahlig.

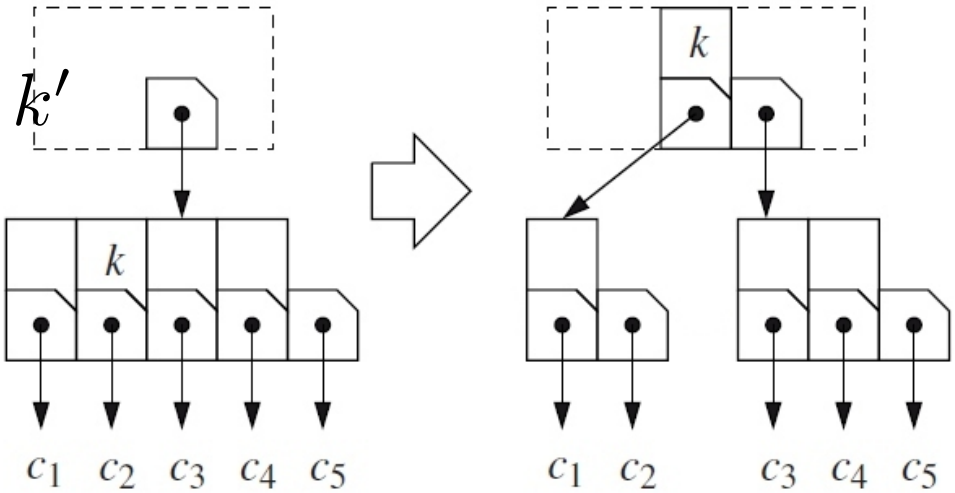
7.2 (a, b) Bäume

RUB

Suche in einem $(a, b)-$Baum ist ähnlich wie in einem binären Suchbaum:
Es sei k der gesuchte Schlüssel und $k' \geq k$ der kleinste vorkommende Schlüssel. Statt einfachem Vergleich führe binäre Suche mit $\lceil \log b \rceil$ Vergleichen auf den Spaltschlüsseln in jedem Knoten aus. Korrektheit folgt mit gleicher Invariante.

Einfügen von Element e mit Schlüssel k:

- $k' =$ key(locate(k))
- falls $k < k'$
  - füge k vor k' in doppelt verketteter Liste ein
  - füge neues Blatt und Spaltschlüssel im Knoten ein
  - falls nun Grad $> b$
    - spalte den Knoten mittig
    - falls Elternknoten existiert, spalte diesen ggfs rekursiv
    - sonst erzeuge neue Wurzel vom Grad 2
- andernfalls ($k = k'$)
  überschreibe den Eintrag an k'



7.2 (a, b) Bäume

RUB

Einfügen von Element e mit Schlüssel k:

- $k' = \text{key}(\text{locate}(k))$
- falls $k < k'$
  - füge k vor k' in doppelt verketteter Liste ein
  - füge neues Blatt und Spaltschlüssel im Knoten ein
  - falls nun Grad $> b$
    - spalte den Knoten mittig
    - falls Elternknoten existiert, spalte diesen ggfs rekursiv
    - sonst erzeuge neue Wurzel vom Grad 2
- andernfalls $(k = k')$
  überschreibe den Eintrag an k'

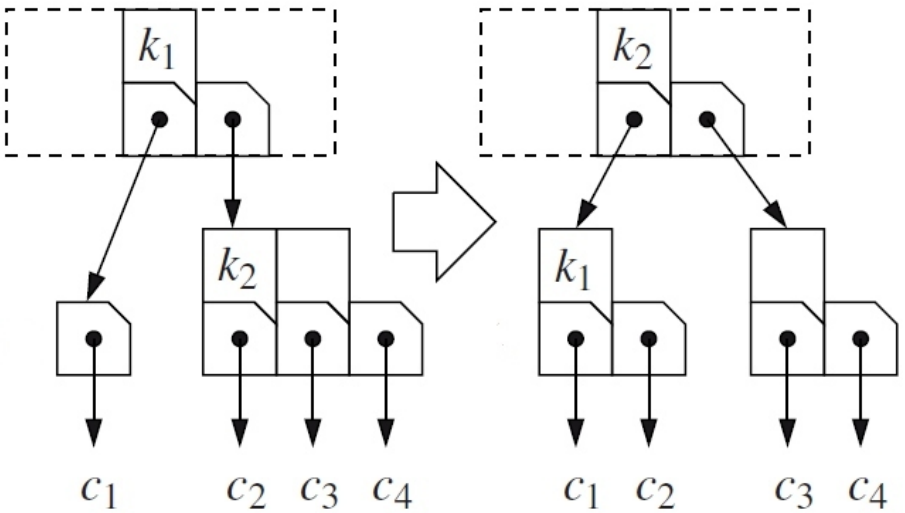
Laufzeit: $O(\log n)$

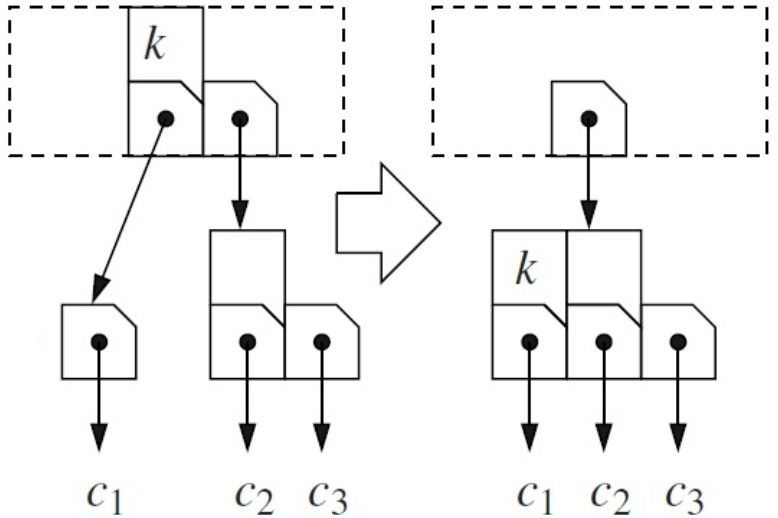
Beob.: Invariante (alle Blätter gleiche Tiefe) bleibt erhalten
(eine neue Wurzel erhöht die Tiefe aller Knoten um 1)

Korrektheit: Für gespaltene Knoten gilt Grad $a \leq d \leq b$

Löschen des Elements mit Schlüssel k:

- if $k =$ key(locate(k))
  - lösche das gefundene Blatt und den Zeiger darauf
  - falls nun der Knoten zu kleinen Grad hat kombiniere ihn mit einem seiner Geschwister
    - falls Geschwister Grad $> a$: hänge ein Kind um
    - falls Geschwister Grad $= a$: verschmelze mit diesem; kombiniere ggfs. rekursiv den Elternknoten





7.2 (a, b) Bäume

RUB

Löschen des Elements mit Schlüssel k:

- if $k =$ key(locate(k))
  - lösche das gefundene Blatt und den Zeiger darauf
  - falls nun der Knoten zu kleinen Grad hat kombiniere ihn mit einem seiner Geschwister
    - falls Geschwister Grad $> a$: hänge ein Kind um
    - falls Geschwister Grad $= a$: verschmelze mit diesem; kombiniere ggfs. rekursiv den Elternknoten

Laufzeit: $O(\log n)$

Beob.: Invariante (alle Blätter gleiche Tiefe) bleibt erhalten (falls durch rekursives Kombinieren eine Wurzel vom Grad 1 entsteht, wird diese bei $n > 1$ Einträgen gelöscht und die Tiefe aller Knoten verringert sich)

Korrektheit: Für kombinierte Knoten gilt Grad $a \leq d \leq b$

7.2 (a, b) Bäume

RUB

Satz: Für ganze Zahlen a und b mit $a \geq 2$ und $b \geq 2a - 1$ gilt: (a, b)-Bäume führen die Operationen insert, remove und locate auf sortierten Folgen mit n Einträgen in Zeit $O(\log n)$ aus.

7.3 Weitere Operationen auf Suchbäumen

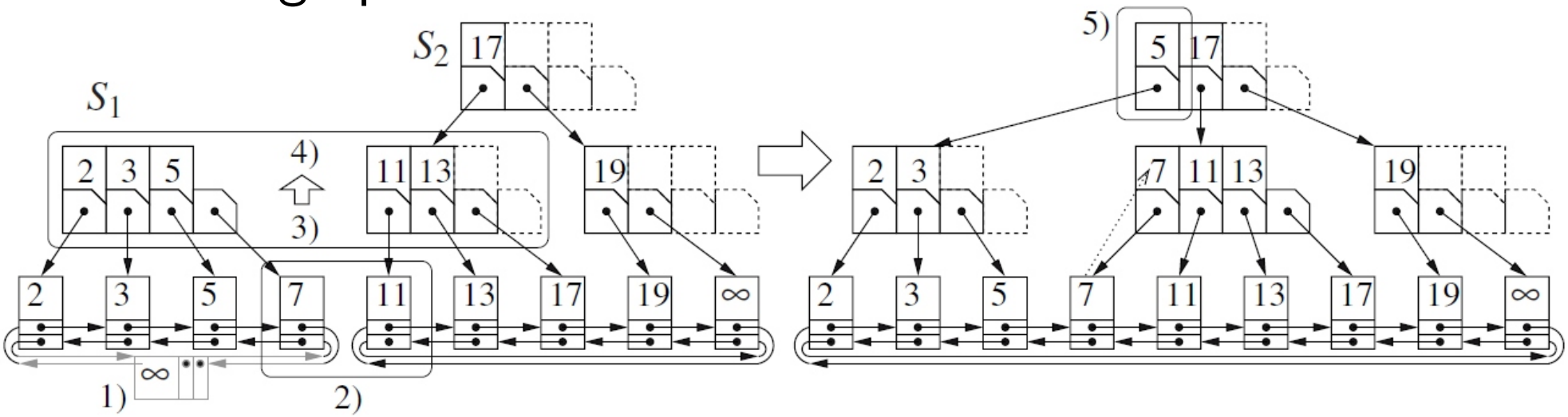
RUB

- **min/max** über first/last auf doppelt verketteter Liste
- **Bereichsanfragen**: finde alle Einträge mit Schlüssel in $[x, y]$
  locate(x) und durchlaufe dopVerkListe bis y
- **Aufbau** aus sortierter Liste: in $O(n)$ Zeit (evtl. Übung)

7.3.1 Konkatenation

Zwei sortierte Folgen S_1, S_2 lassen sich konkatenieren, falls $\max S_1 \leq \min S_2$. Sind S_1, S_2 als (a, b)-Bäume dargestellt, so geht dies in $O(1 + |H_1 - H_2|) = O(\log(S_1| + |S_2|))$ Zeit.

- konkateniere die doppelt verketteten Listen
- oBdA $H_1 = \text{height}(S_1) \geq H_2 = \text{height}(S_2)$.
  verschmelze Wurzel von S_2 mit rechtesten Knoten auf Ebene $H_1 - H_2$ in S_2 mit $\max S_1$ als Spaltschlüssel.
  Wenn der verschmolzene Knoten nun zu hohen Grad hat, wird er gespalten wie beim insert.

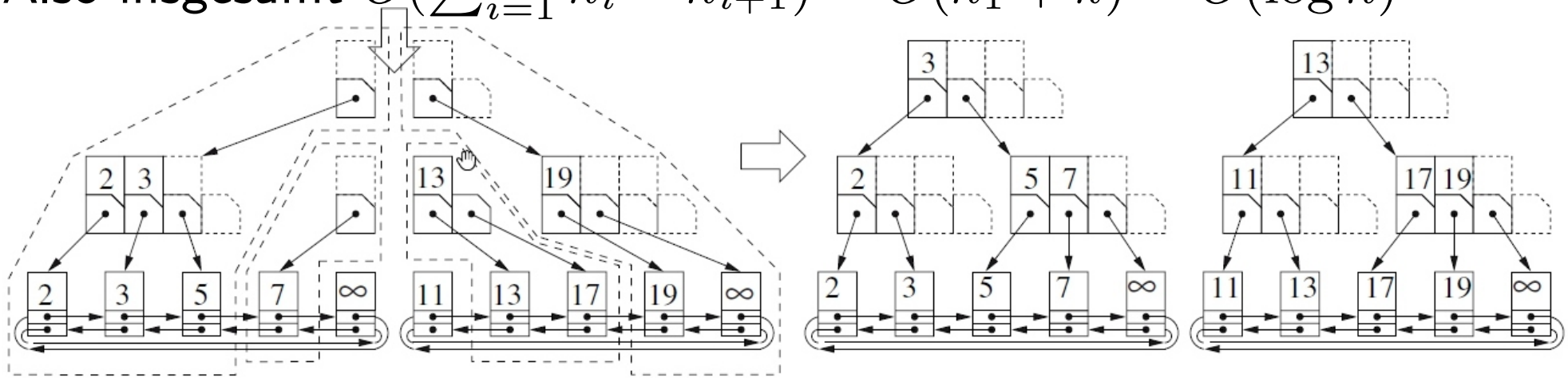


7.3.2 Spalten

Sei Folge $S = \langle w, \ldots, x, y, \ldots, z \rangle$ gegeben.
Spalte an y in $S_1 = \langle w, \ldots, x \rangle$ und $S_2 = \langle y, \ldots, z \rangle$.

Betrachte dazu Pfad P von der Wurzel r zu Blatt y.
Jeder Knoten v auf P wird in zwei Knoten v_ℓ und v_r gespalten (mit möglicherweise 0 oder 1 Kind). Jeder entstandene Teilknoten mit mind. einem Kind liefert eigenen (a, b)-Baum.
Die Bäume links von P werden sukzessive von klein nach gross konkateniert $\rightarrow$ (a, b)-Baum für S_1 [analog für S_2]

Laufzeit: Seien $h_1 > \ldots > h_k$ die Höhen der zu verschmelzenden linken Bäume. Die i-te Konkatenation kostet $O(h_i - h_{i+1})$.
Also insgesamt $O(\sum_{i=1}^{k} h_i - h_{i+1}) = O(h_1 + k) = O(\log n)$



7.4 Amortisierte Analyse

RUB

Eine Einfügung oder Löschung im $(a,b)-$Baum kostet $\Theta(\log n)$

Genauer: nach der Suche müssen wir

- im besten Fall nur die doppelt verkettete Liste und den untersten Knoten aktualisieren
- im schlimmsten Fall können sich Spaltungen bzw. Verschmelzungen bis zur Wurzel hochziehen

Falls $b \geq 2a$ können wir eine bessere amortisierte Laufzeit bzgl. Spaltungen und Verschmelzungen zeigen.

Satz: Betrachte einen anfangs leeren (a,b)-Baum mit $b \geq 2a$. Für jede Folge aus n Einfügungen und Löschungen werden insgesamt nur $O(n)$ split- oder fuse-Operationen ausgeführt.

Beweis: für $(2,4)$-Baum mit Bankkontomethode.

7.4 Amortisierte Analyse

RUB

Wir denken uns ein **Bankkonto**, auf dem bei jeder Operation konstant viele *'Jetons'* eingezahlt und abgehoben werden. Konkret: bei split und fuse -1, bei insert $+2$ und bei remove $+1$. Der Kontostand darf niemals negativ werden. Dann können wir folgern, dass nur $O(n)$ viele split und fuse durchgeführt werden.

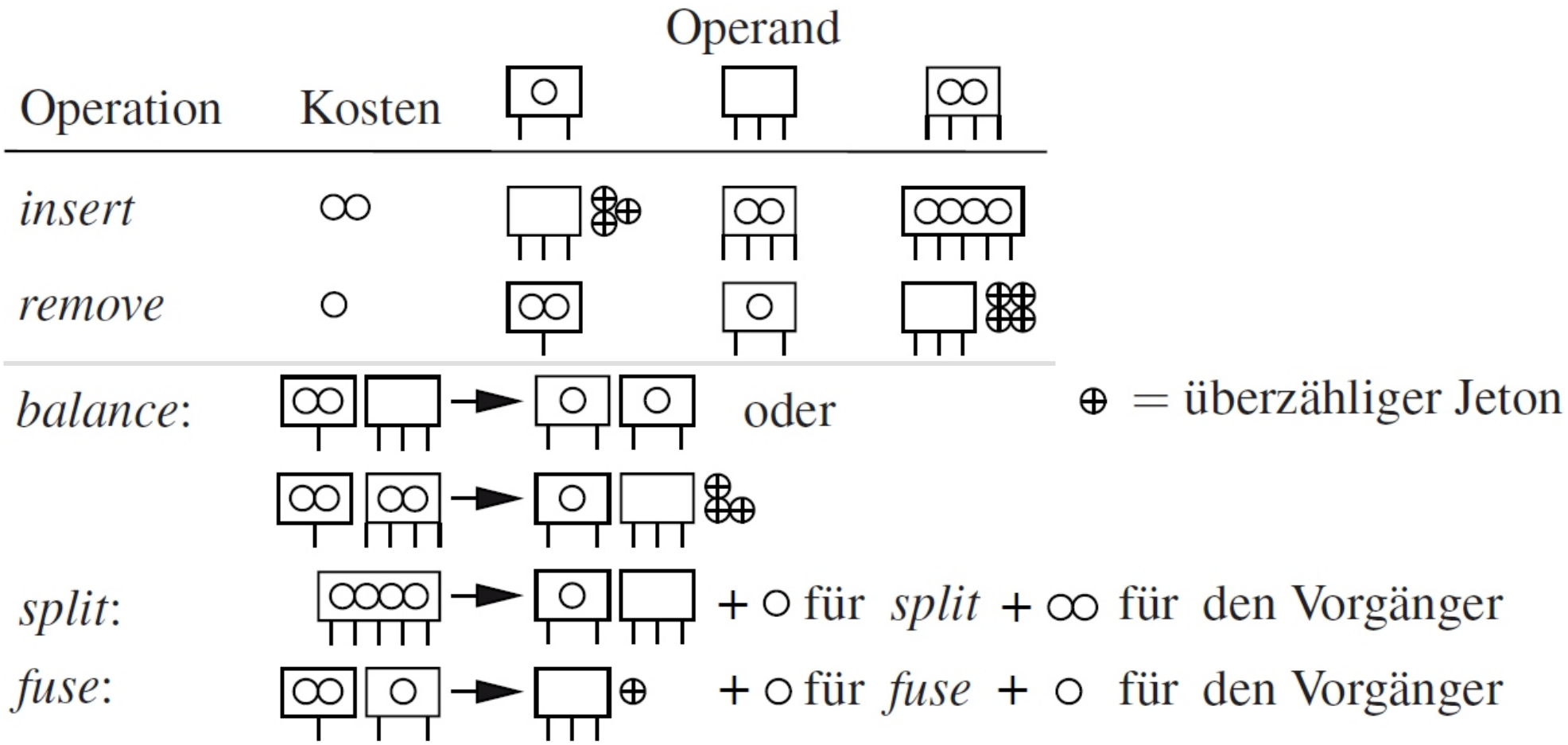
Wir müssen noch zeigen, dass diese Einzahlungen ausreichen, um alle split- und fuse-Operationen zu bezahlen.

Balance-Operationen passieren bei jedem Löschen nur einmal, so dass wir sie nicht weiter berücksichtigen müssen.

7.4 Amortisierte Analyse

Zur Analyse des Kontostandes verteilen wir die Jetons auf Baumknoten, und zeigen dass folgende **Jeton-Invariante** erhalten bleibt:

Grad	1	2	3	4	5
Jetons	○○	○		○○	○○○○



RUB

Inkrementieren eines Binärzählers ($\rightarrow$ 2.7.1)

Mit der Bankkontomethode kann man ebenfalls zeigen, dass beim Inkrementieren eines Binärzählers amortisiert nur konstant viele Bits umgesetzt werden.

Zur Erinnerung:

$i := 0$
while $(i < n \; and \; a[i] = 1)$ **do** $a[i] = 0$; i++
if $i < n$ **then** $a[i] = 1$

Wir zeigen, dass der Rumpf der While-Schleife bei n Inkrementierungen startend in 0 nur $O(n)$ mal ausgeführt wird.

Zahle 1 Jeton pro Inkrementierung und zeige dass die **Invariante** aufrechterhalten bleibt: Kontostand $=$ Anzahl Einsen.

Betrachte Inkrementierung mit Kosten k: $01 \ldots 1 \rightarrow 10 \ldots 0$

Kontostand erniedrigt um $k - 1$, gemeinsam mit 1 Jeton Einzahlung können damit die Kosten k gezahlt werden.

7.5 Erweiterte Suchbäume

RUB

Vorgängerzeiger: hat jeder Baumknoten zusätzlich einen Zeiger auf seinen Elternknoten (also doppelt gerichtete Zeiger im Baum), so können Einträge über Griffe direkt gelöscht und eingefügt werden (ohne Suche) und kosten amortisiert nur $O(1)$

RUB

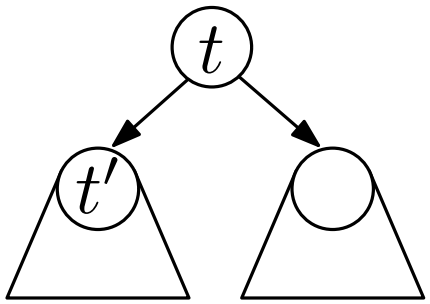
Größe von Teilbäumen:

Angenommen, jeder Baumknoten t speichert ein Feld $t.size$ mit der Anzahl der Blätter im Unterbaum mit Wurzel t.
Dann lässt sich das Auswahlproblem effizient lösen: Eintrag von Rang k kann in Zeit proportional zur Baumhöhe gefunden werden.
Der Einfachheit halber zeigen wir dies für binäre Suchbäume.

Halte bei der Suche die **Invariante** aufrecht:

Eintrag von Rang k ist im Unterbaum mit Wurzel t.

Vorgehen: führe ebenfalls $i =$ Anzahl Elemente links vom Unterbaum in t mit. Sei $i' = t'.size$, dann



- falls $i' + i \leq k$: ersetze t durch linkes Kind
- sonst: ersetze t durch rechtes Kind; $i := i + i'$

Wird ein Blatt erreicht, folgt aus der Invariante, das dieses das Element von Rang k ist.