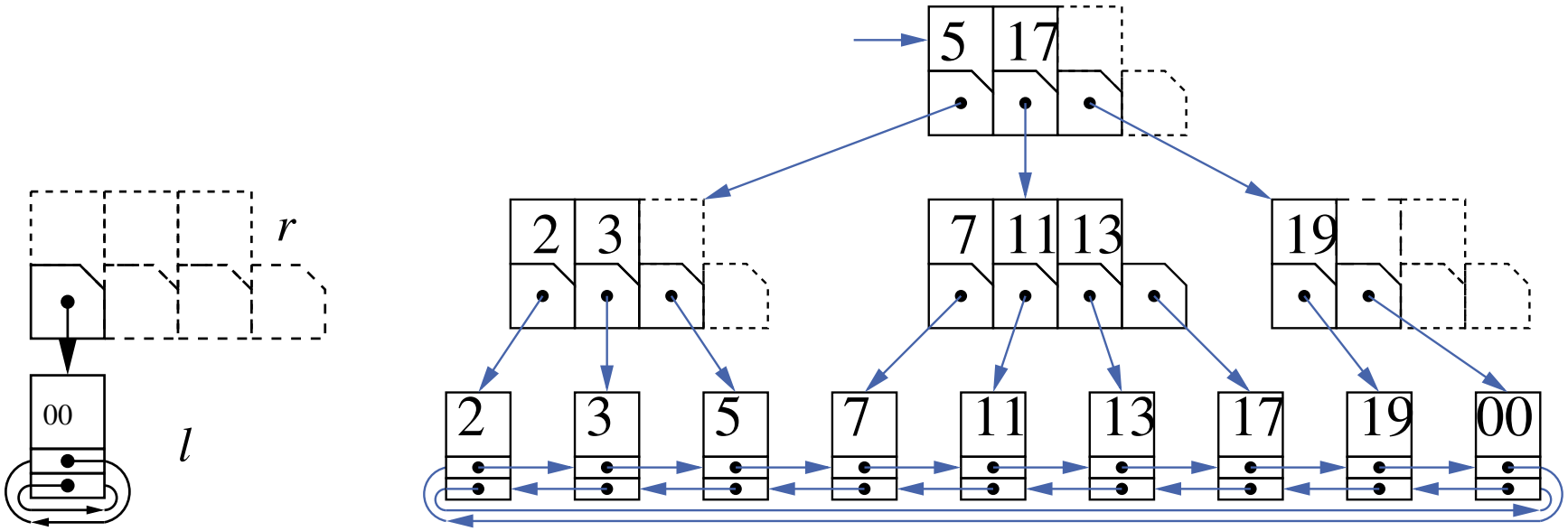
Suchbäume balancieren

Perfekte Balance: schwer aufrechtzuerhalten

Flexible Höhe $O(\log n)$: balancierte binäre Suchbäume.
Nicht hier (Variantenzoo).

Flexibler Knotengrad: (a, b)-Bäume.
$\approx$ Grad zwischen a und b.
Höhe $\approx \log_a n$

(a, b)-Bäume



Blätter: Listenelemente (wie gehabt). Alle mit gleicher Tiefe!

Innere Knoten: Grad $a..b$

Wurzel: Grad $2..b$, (Grad 1 für $\langle\rangle$)

Items

Class ABHandle : **Pointer** to ABItem or Item

Class ABItem(splitters : Sequence **of** Key, children : Sequence **of** ABHandle)

d=|children| : $1..b$ // outdegree

s=splitters : **Array** $[1..b-1]$ **of** Key

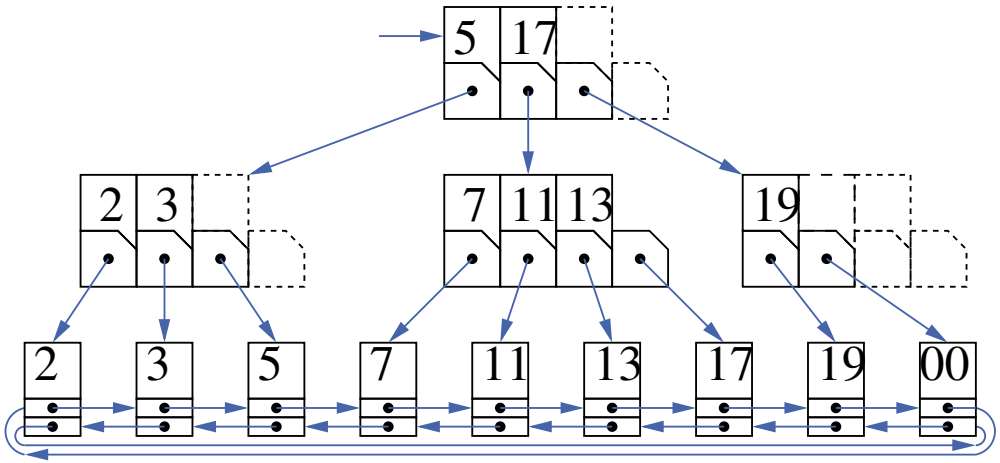
c=children : **Array** $[1..b]$ **of** Handle

Invariante:

e über $c[i]$ erreichbar

$\Rightarrow s[i-1] < key(e) \leq s[i]$ mit

$s[0] = -\infty$, $s[d] = s[d+1] = \infty$



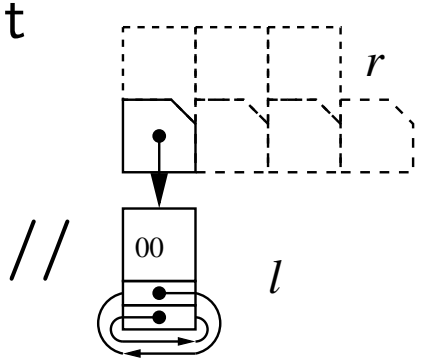
Initialisierung

Class ABTree($a \geq 2 : \mathbb{N}$, $b \geq 2a-1 : \mathbb{N}$) **of** Element

ℓ=$\langle\rangle$: List **of** Element

r : ABItem($\langle\rangle$, $\langle \ell$.head$\rangle$)

height=1 : $\mathbb{N}$



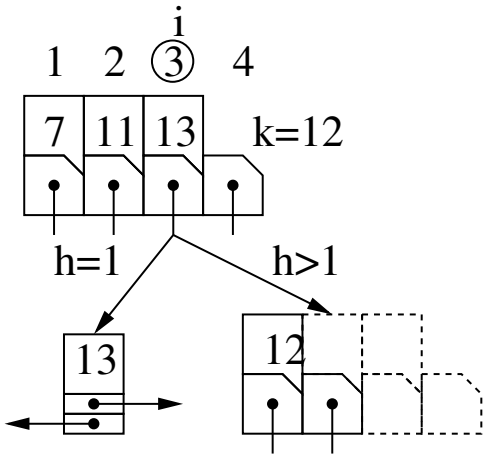
// Locate the smallest Item with key $k' \geq k$

Function locate(k : Key) : Handle **return** r.locateRec(k, height)

Locate

Function ABItem::locateLocally(k : Key) : $\mathbb{N}$
    return $\min\{i \in 1..d : k \leq s[i]\}$

Function ABItem::locateRec(k : Key, h : $\mathbb{N}$) : Handle
    i:= locateLocally(k)
    if $h = 1$ **then**
        if $c[i] \rightarrow e \geq k$ Then **return** $c[i]$
        else return $c[i] \rightarrow$ next
    else
        return $c[i] \rightarrow$locateRec(k, $h-1$) //



Invariante: im Wesentlichen analog zu binären Suchbäumen

Locate – Laufzeit

$O(b \cdot \text{height})$

Übung: $b \to \log b$?

Lemma: $\text{height} = h \leq 1 + \left\lfloor \log_a \frac{n+1}{2} \right\rfloor$

Beweis:

Fall $n = 1$: $\text{height} = 1$.

Fall $n > 1$:

Wurzel hat Grad ≥ 2 und

Innere Knoten haben Grad $\geq a$.

$\Rightarrow \geq 2a^{h-1}$ Blätter.

Es gibt $n+1$ Blätter.

Also $n+1 \geq 2a^{h-1}$

$\Rightarrow h \leq 1 + \log_a \frac{n+1}{2}$

Rundung folgt, weil h eine ganze Zahl ist. $\square$

Einfügen – Algorithmenskizze

Procedure insert(e)

Finde Pfad Wurzel $\rightsquigarrow$ nächstes Element e'

ℓ.insertBefore(e, e')

füge key(e) als neuen Splitter in Vorgänger u

if $u.d = b+1$ **then**

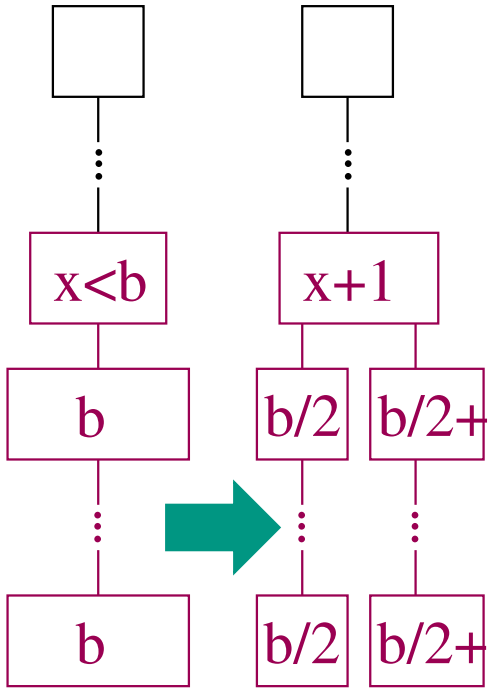
spalte u in 2 Knoten mit Graden

$\lfloor (b+1)/2 \rfloor$, $\lceil (b+1)/2 \rceil$

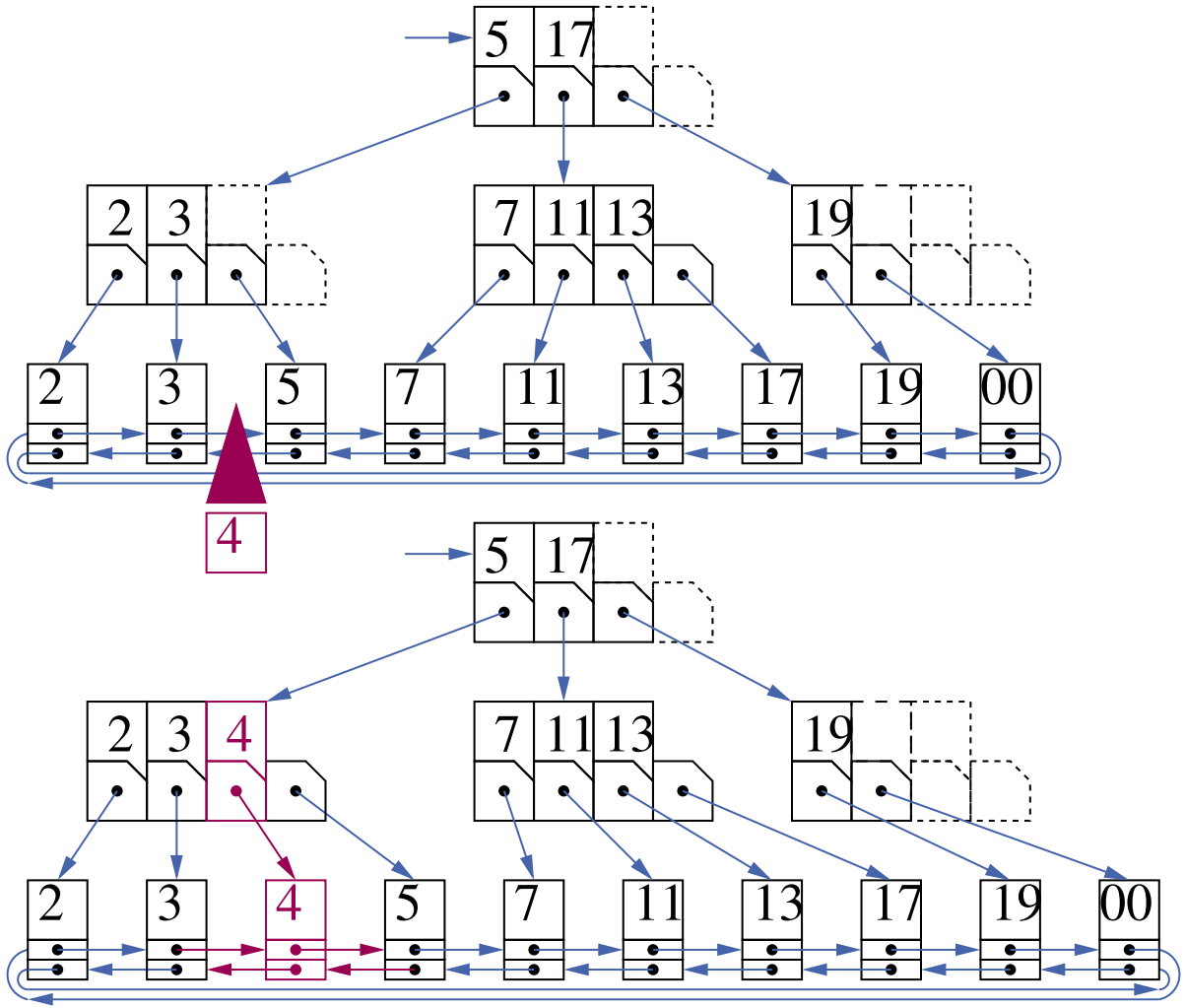
Weiter oben einfügen, spalten

...

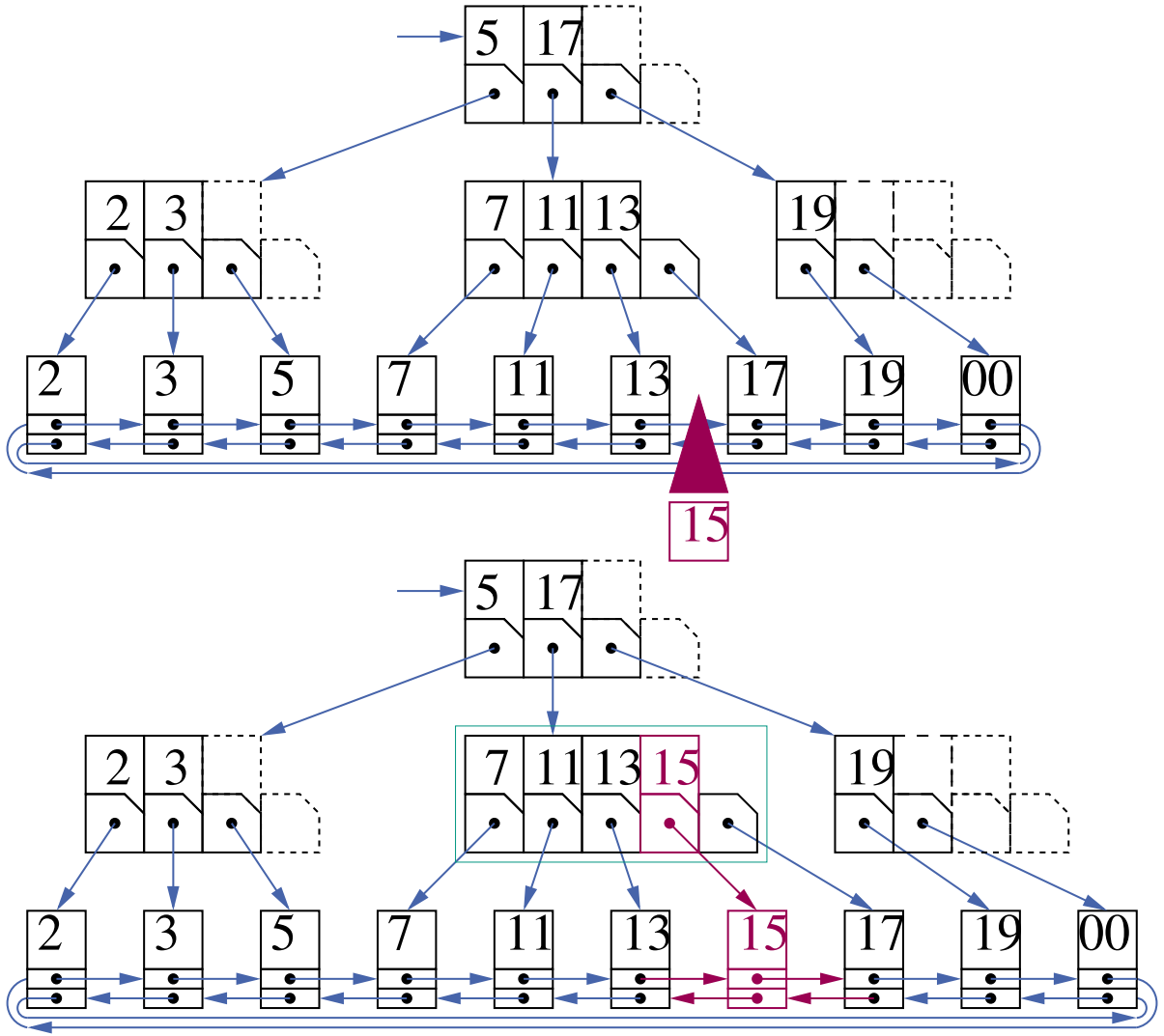
ggf. neue Wurzel



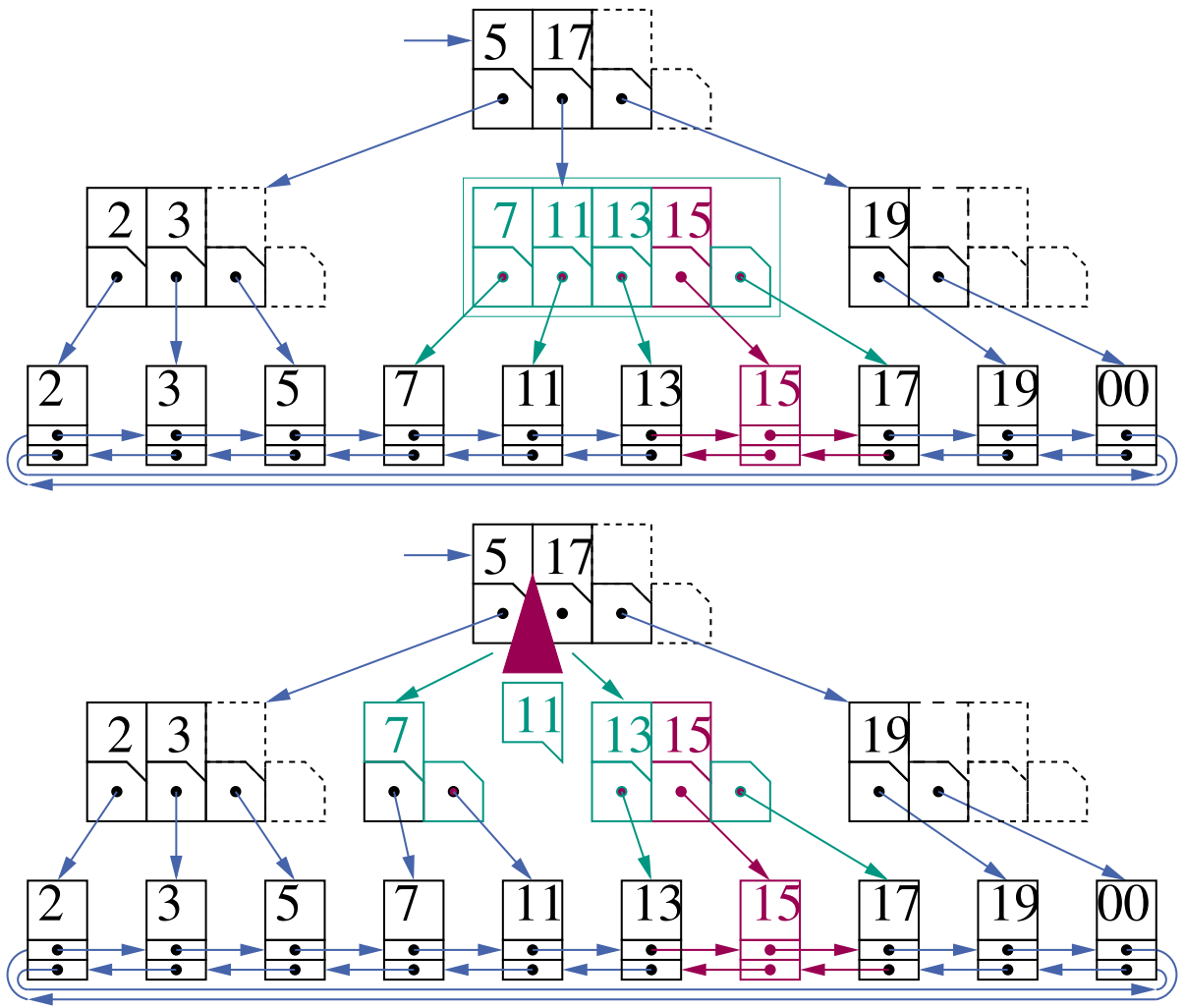
Einfügen – Beispiel



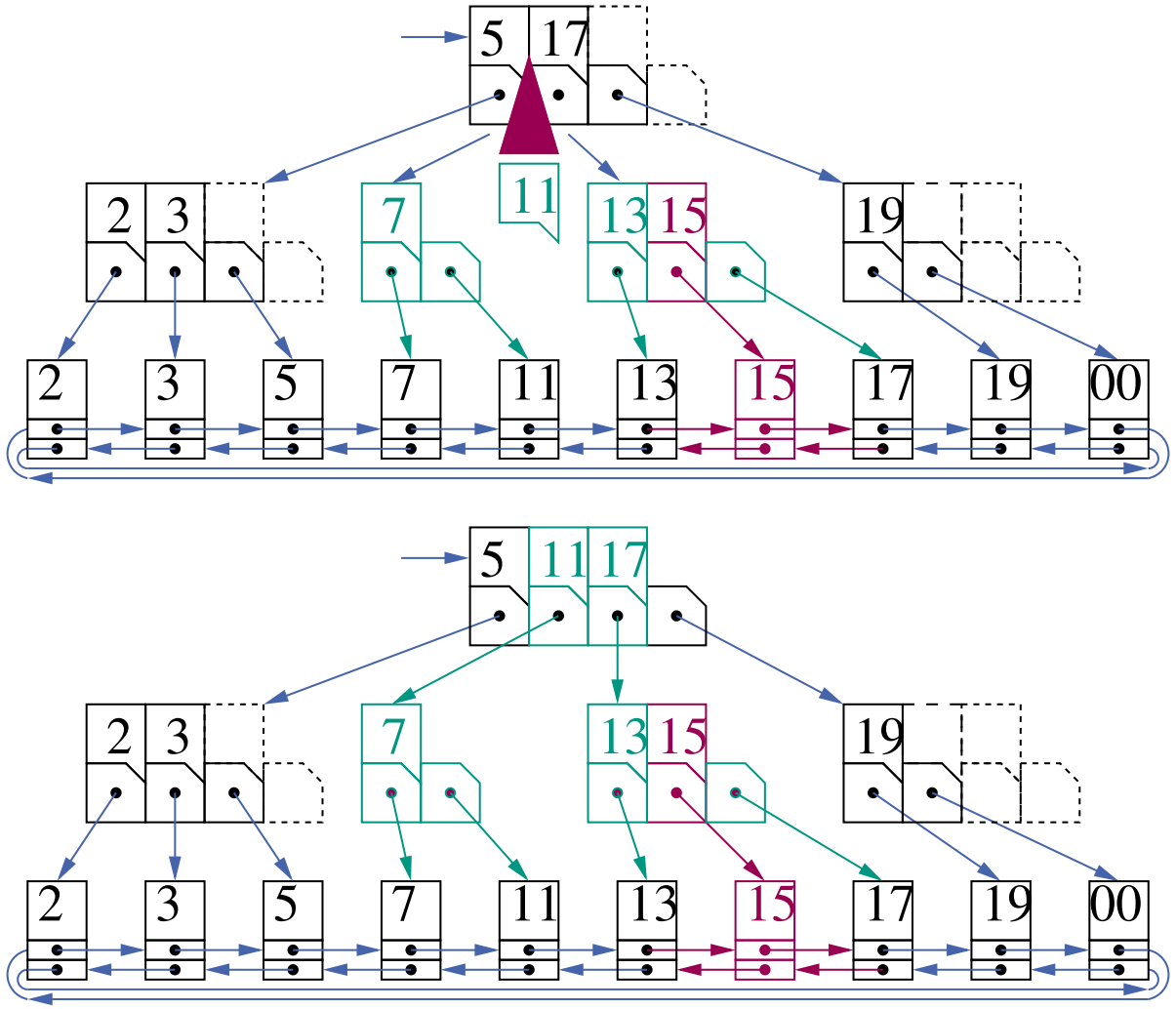
Einfügen – Beispiel



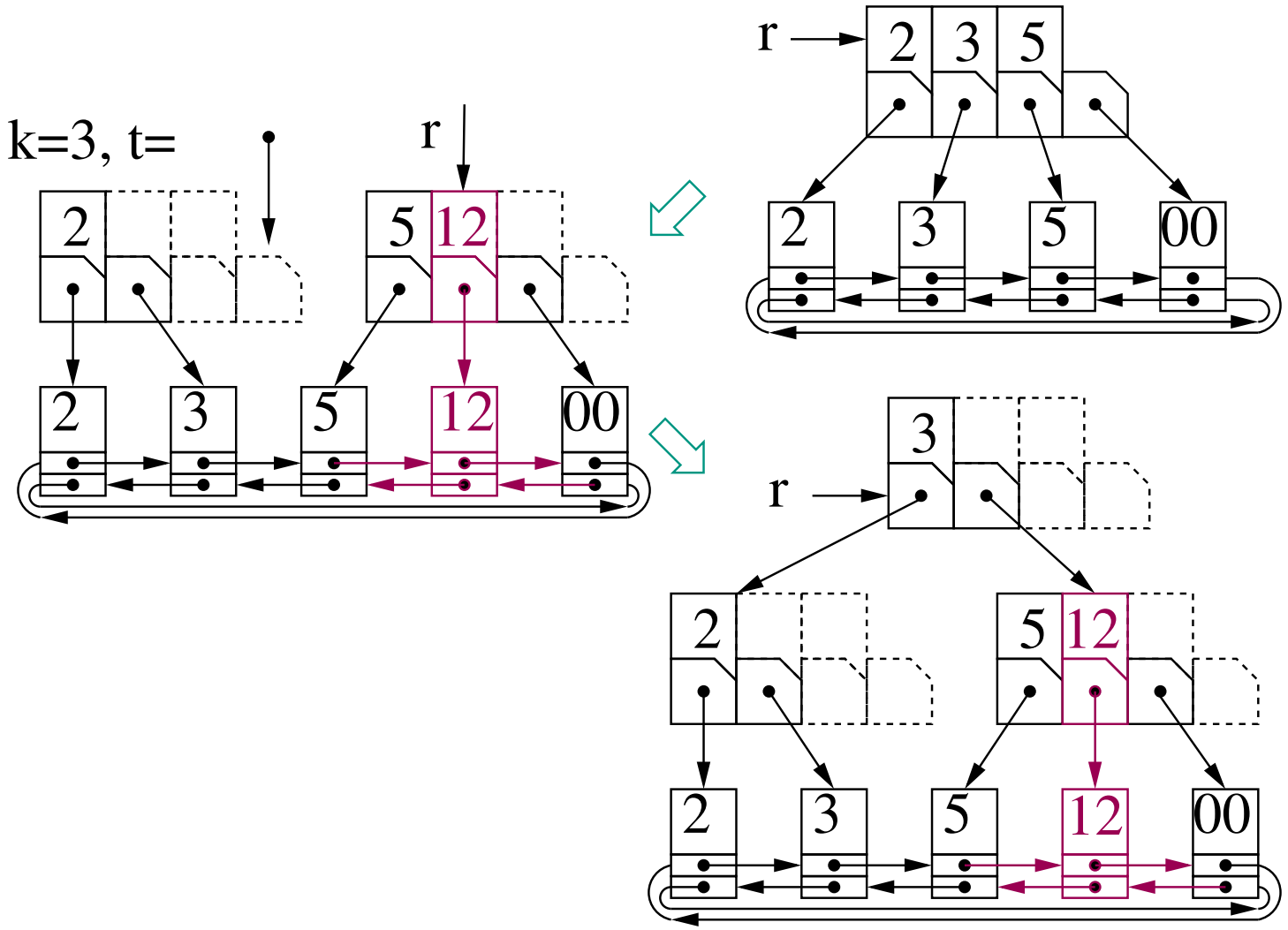
Einfügen – Beispiel



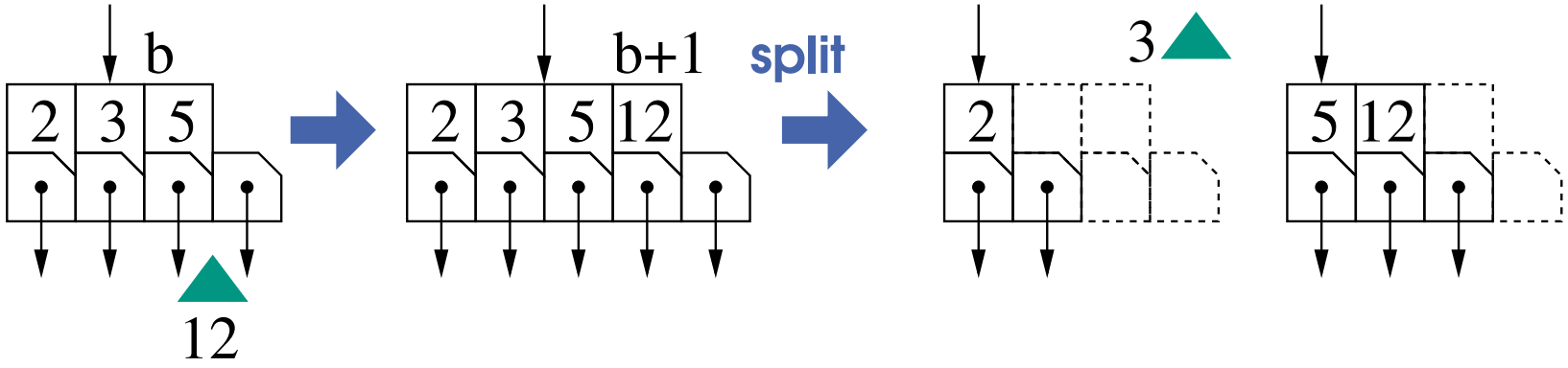
Einfügen – Beispiel



Einfügen – Beispiel



Einfügen – Korrektheit



Nach dem Spalten müssen zulässige Items entstehen:

$$\left\lfloor \frac{b+1}{2} \right\rfloor \overset{!}{\geq} a \Leftrightarrow b \geq 2a-1$$

Weil $\left\lfloor \frac{(2a-1)+1}{2} \right\rfloor = \left\lfloor \frac{2a}{2} \right\rfloor = a$

Einfügen – Implementierungsdetails

- Spalten pflanzt sich von unten nach oben fort. Aber wir speichern nur Zeiger nach unten.
  Lösung: Rekursionsstapel speichert Pfad.
- Einheitlicher Itemdatentyp mit Kapazität für b Nachfolger.
  einfacher, schneller, Speicherverwaltung!
- Baue nie explizit temporäre Knoten mit $b+1$ Nachfolgern.

Einfügen – Pseudocode

```
// ℓ: "the list"
// r: root
// height (of tree)
Procedure ABTree::insert(e : Element)
    (k, t):= r.insertRec(e, height, ℓ)
    if t ≠ null then
        r:= allocate ABItem(⟨k⟩, ⟨r, t⟩)
        height++
```

```
Function ABItem::insertRec(e : Element, h : ℕ, ℓ : List of Element) :
        Key×ABHandle
    i:= locateLocally(e)
    if h = 1 then (k,t):= (key(e), ℓ.insertBefore(e, c[i]))      // base
    else          (k,t):= c[i] → insertRec(e, h−1, ℓ)            // recurse
                  if t = null then return (⊥, null )
    s′:= ⟨s[1],...,s[i−1], k, s[i],...,s[d−1]⟩                   // new splitter
    c′:= ⟨c[1],...,c[i−1], t, c[i],...,c[d]⟩                     // new child
    if d < b then (s,c,d):= (s′,c′,d+1); return (⊥, null )
    else                                                         // split this node
        d:= ⌊(b+1)/2⌋
        s:= s′[b+2−d..b]
        c:= c′[b+2−d..b+1]
        return (s′[b+1−d],
                allocate ABItem(s′[1..b−d], c′[1..b+1−d]))
```

Entfernen – Algorithmenskizze

Procedure remove(e)

Finde Pfad Wurzel$\rightarrow e$

ℓ.remove(e)

entferne key(e) in Vorgänger u

if $u.d = a-1$ **then**

finde Nachbarn u'

if $u'.d + a - 1 \leq b$ **then**

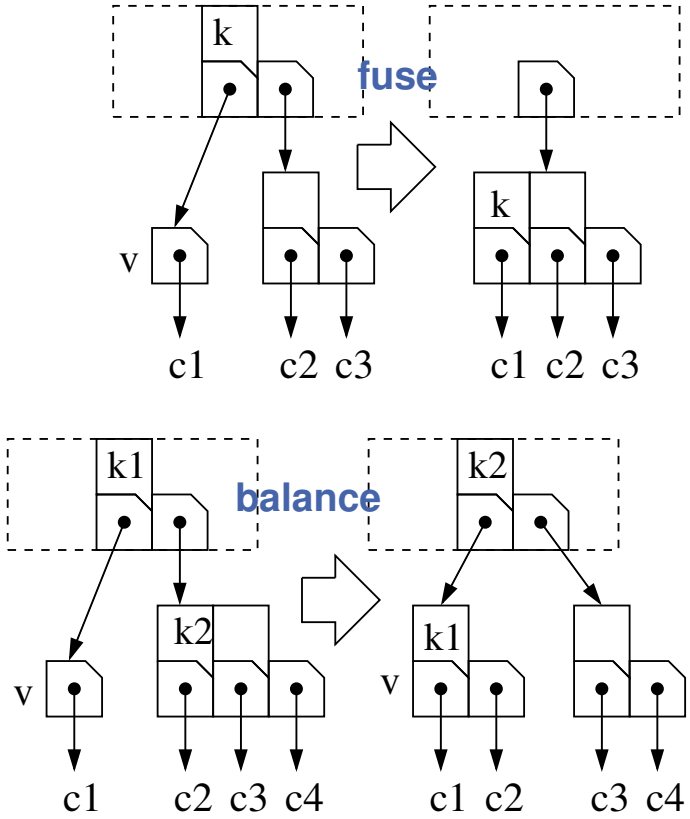
fuse(u', u)

Weiter oben splitter entfernen

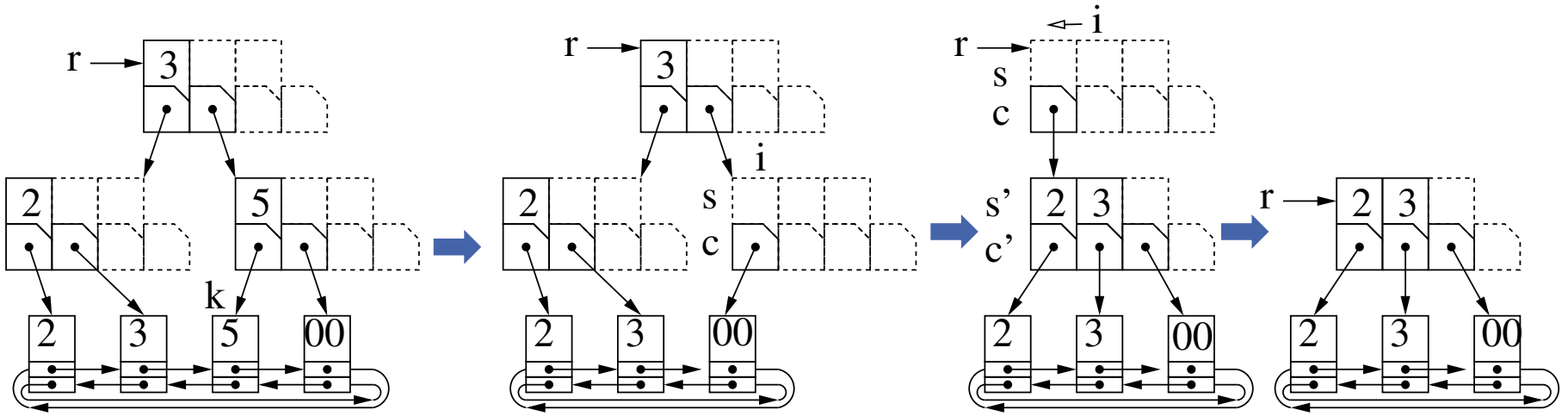
...

ggf. Wurzel entfernen

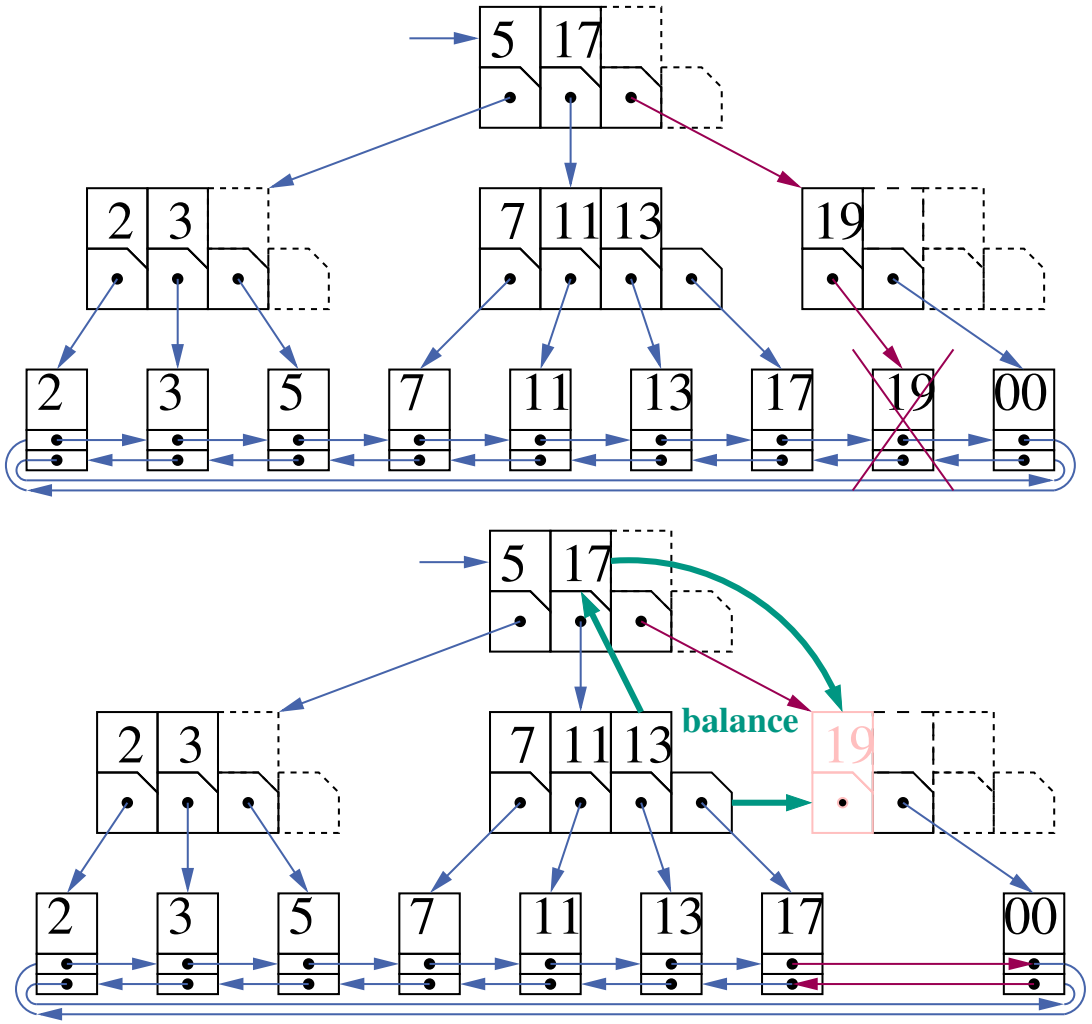
else balance(u', u)



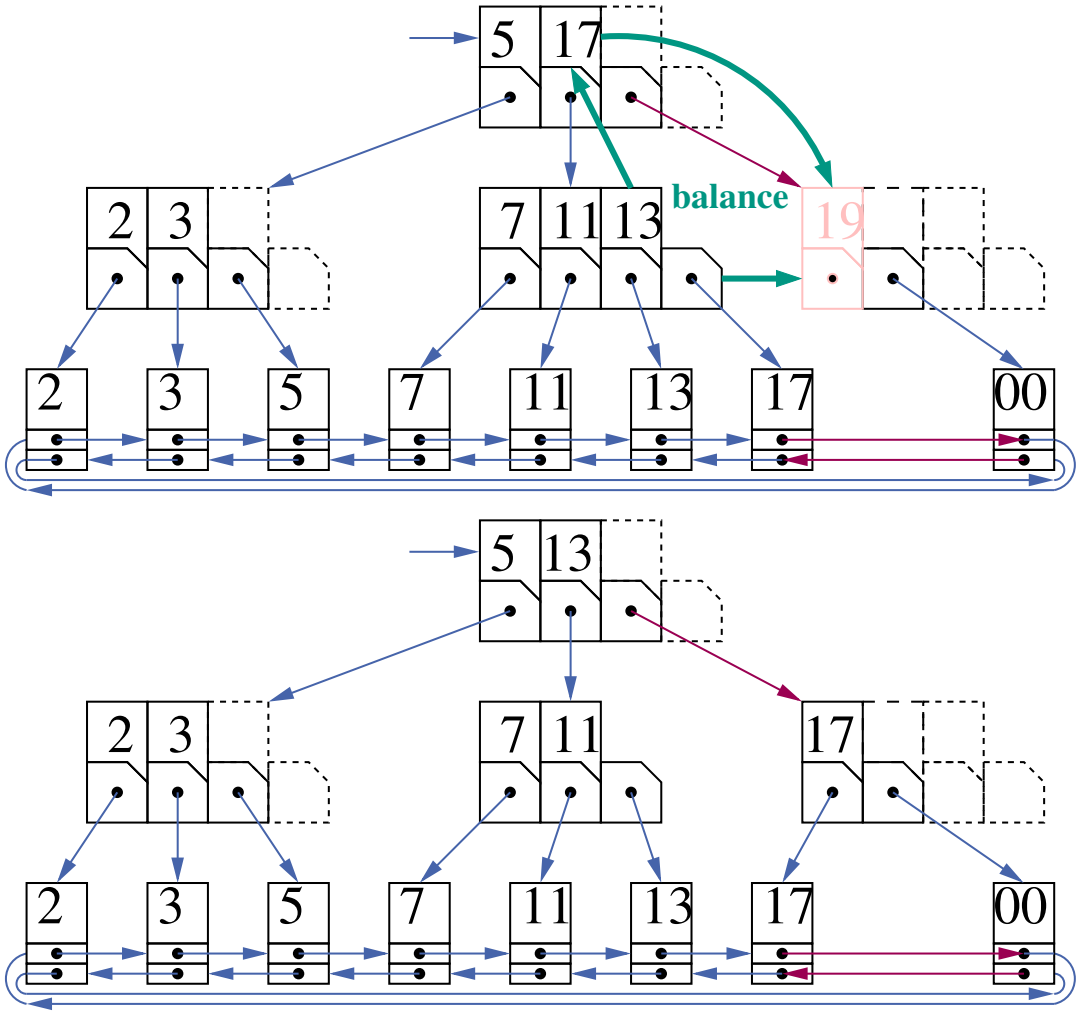
Entfernen – Beispiel



Entfernen – Beispiel



Entfernen – Beispiel

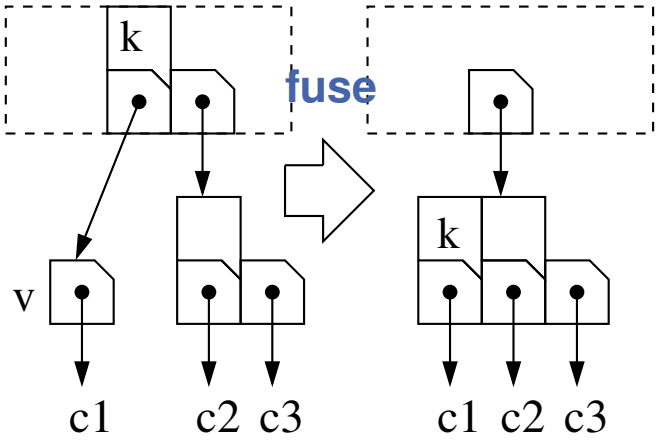


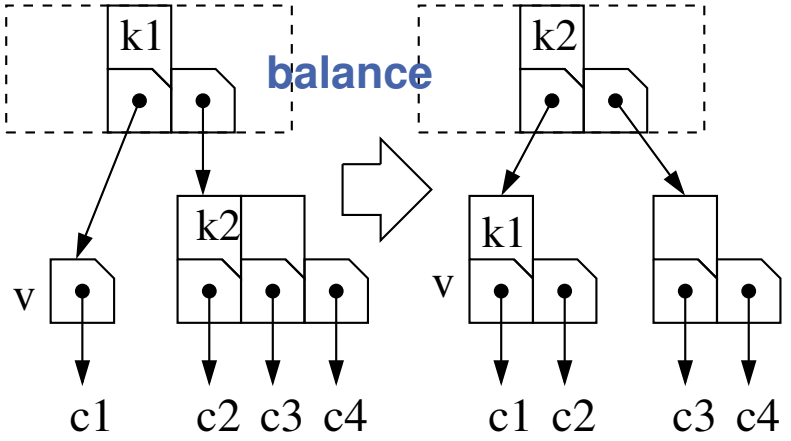
Entfernen – Korrektheit

Nach fuse
müssen zulässige Items entstehen:

$a + (a-1) \overset{!}{\leq} b \Leftrightarrow b \geq 2a-1$

hatten wir schon!





Einfügen und Entfernen – Laufzeit

$$\begin{aligned} &\mathrm{O}(b \cdot \text{Höhe}) = \mathrm{O}(b \log_a n) \\ &= \mathrm{O}(\log n) \text{ für } \{a, b\} \subseteq \mathrm{O}(1) \end{aligned}$$

(a,b)-Bäume
Implementierungsdetails

Etwas kompliziert...
Wie merkt man sich das?
Gar nicht!
Man merkt sich:

- Invarianten
  Höhe, Knotengrade
- Grundideen
  split, balance, fuse

Den Rest leitet man
sich nach Bedarf neu her.

```
Procedure ABTree::remove(k : Key)
    r.removeRec(k, height, ℓ)
    if r.d = 1 ∧ height > 1 then r' := r; r := r'.c[1]; dispose r'
Procedure ABItem::removeRec(k : Key, h : ℕ, ℓ : List of Element)
    i := locateLocally(k)
    if h = 1 then
        if key(c[i] → e) = k then
            ℓ.remove(c[i])
            removeLocally(i)
    else
        c[i] → removeRec(e, h − 1, ℓ)
        if c[i] → d < a then
            if i = d then i−−
            s' := concatenate(c[i] → s, ⟨s[i]⟩, c[i + 1] → s))
            c' := concatenate(c[i] → c, c[i + 1] → c)
            d' := |c'|
            if d' ≤ b then          // fuse
                (c[i + 1] → s, c[i + 1] → c, c[i + 1] → d) := (s', c', d')
                dispose c[i];   removeLocally(i)
            else                    // balance
                m := ⌈d'/2⌉
                (c[i] → s, c[i] → c, c[i] → d) := (s'[1..m − 1], c'[1..m], m)
                (c[i + 1] → s, c[i + 1] → c,   c[i + 1] → d) :=
                    (s'[m + 1..d' − 1], c'[m + 1..d'],   d' − m)
                s[i] := s'[m]
Procedure ABItem::removeLocally(i : ℕ)
    c[i..d − 1] := c[i + 1..d]
    s[i..d − 2] := s[i + 1..d − 1]
    d−−
```

Mehr Operationen

min, max, rangeSearch(a, b): $\langle \text{min}, \ldots, a, \ldots, b, \ldots, \text{max} \rangle$
hatten wir schon
build: Übung! Laufzeit $\mathrm{O}(n)$!
(Navigationstruktur für sortierte Liste aufbauen)
concat, **split**: nicht hier. Zeit $\mathrm{O}(\log n)$
Idee: Ganze Teilbäume umhängen
merge(N, M): sei $n = |N| \leq m = |M|$ Zeit $\mathrm{O}\left(n \log \frac{m}{n}\right)$
nicht hier. Idee: z. B. Fingersuche

Amortisierte Analyse von insert und remove

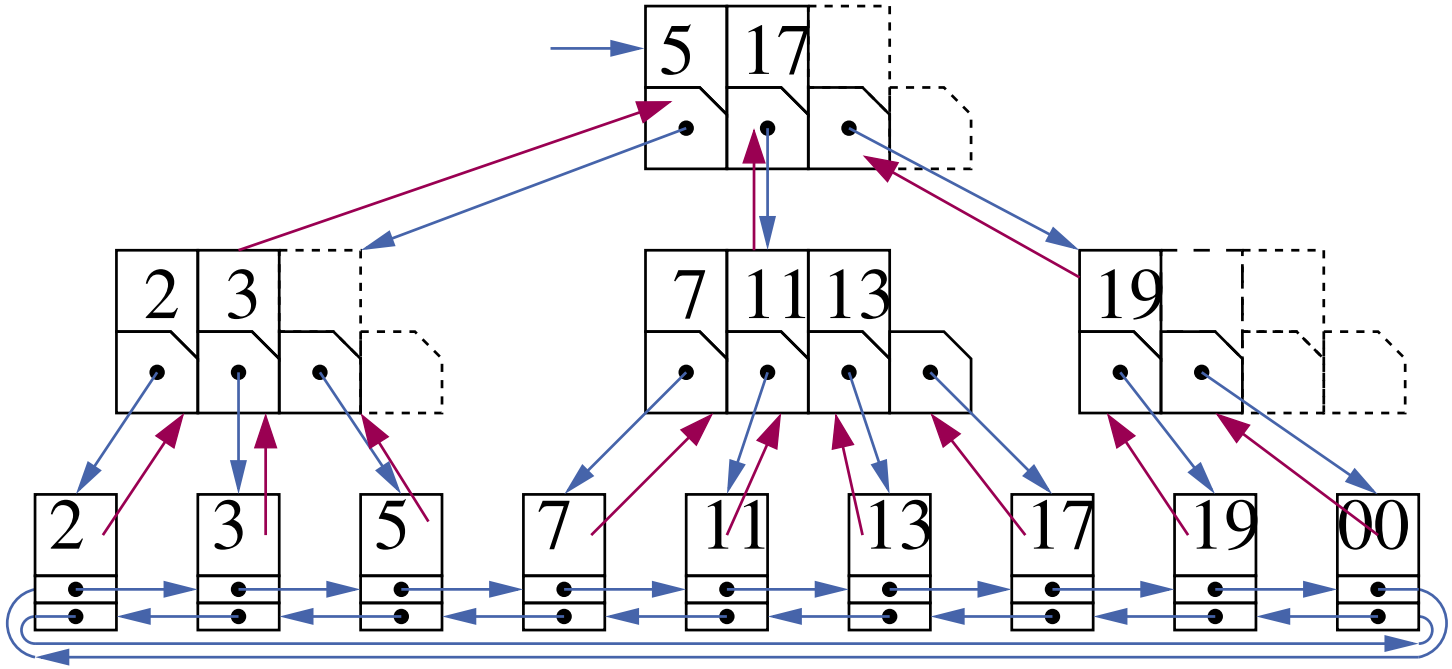
nicht hier.
Grob gesagt: Abgesehen von der Suche fällt nur konstant viel Arbeit an (summiert über alle Operationsausführungen).

Erweiterte (augmentierte) Suchbäume

Idee: zusätzliche Infos verwalten $\rightsquigarrow$ mehr (schnelle) Operationen.
Nachteil: Zeit- und Platzverschwendung,
wenn diese Operationen nicht wichtig sind. gold plating

Elternzeiger

Idee (Binärbaum): Knoten speichern Zeiger auf Elternknoten



Anwendungen: schnelleres remove, insertBefore, insertAfter,
falls man ein handle des Elements kennt.
Man spart die Suche.

Frage: was speichert man bei (a, b)-Bäumen?

Teilbaumgrößen

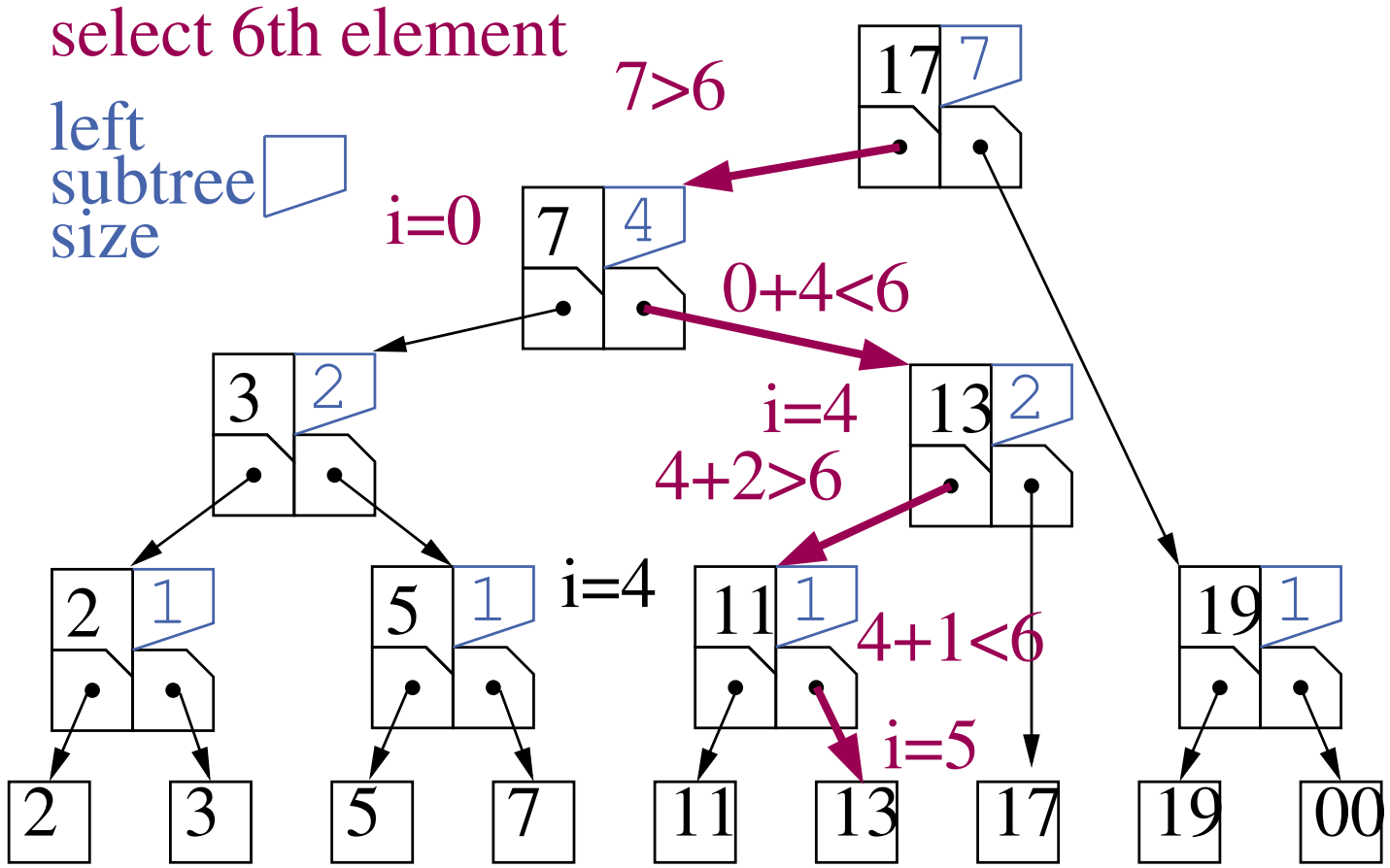
Idee (Binärbaum): speichere, wie viele Blätter von links erreichbar.
(Etwas anders als im Buch!)

// return k-th Element in subtree rooted at h
Function selectRec(h, k)
    if $h \rightarrow$ leftSize $\geq k$ **then return** select(ℓ, k)
    else return select($r, k -$ leftSize)

Zeit: $O(\log n)$

Übung: Was ist anders bei (a, b)-Bäumen?
Übung: Rang eines Elements e bestimmen.
Übung: Größe eines Bereichs $a..b$ bestimmen.

Beispiel


select 6th element
left subtree size
7>6
i=0
0+4<6
i=4
4+2>6
i=4
4+1<6
i=5
17 7
7 4
3 2
13 2
2 1
5 1
11 1
19 1
2
3
5
7
11
13
17
19
00

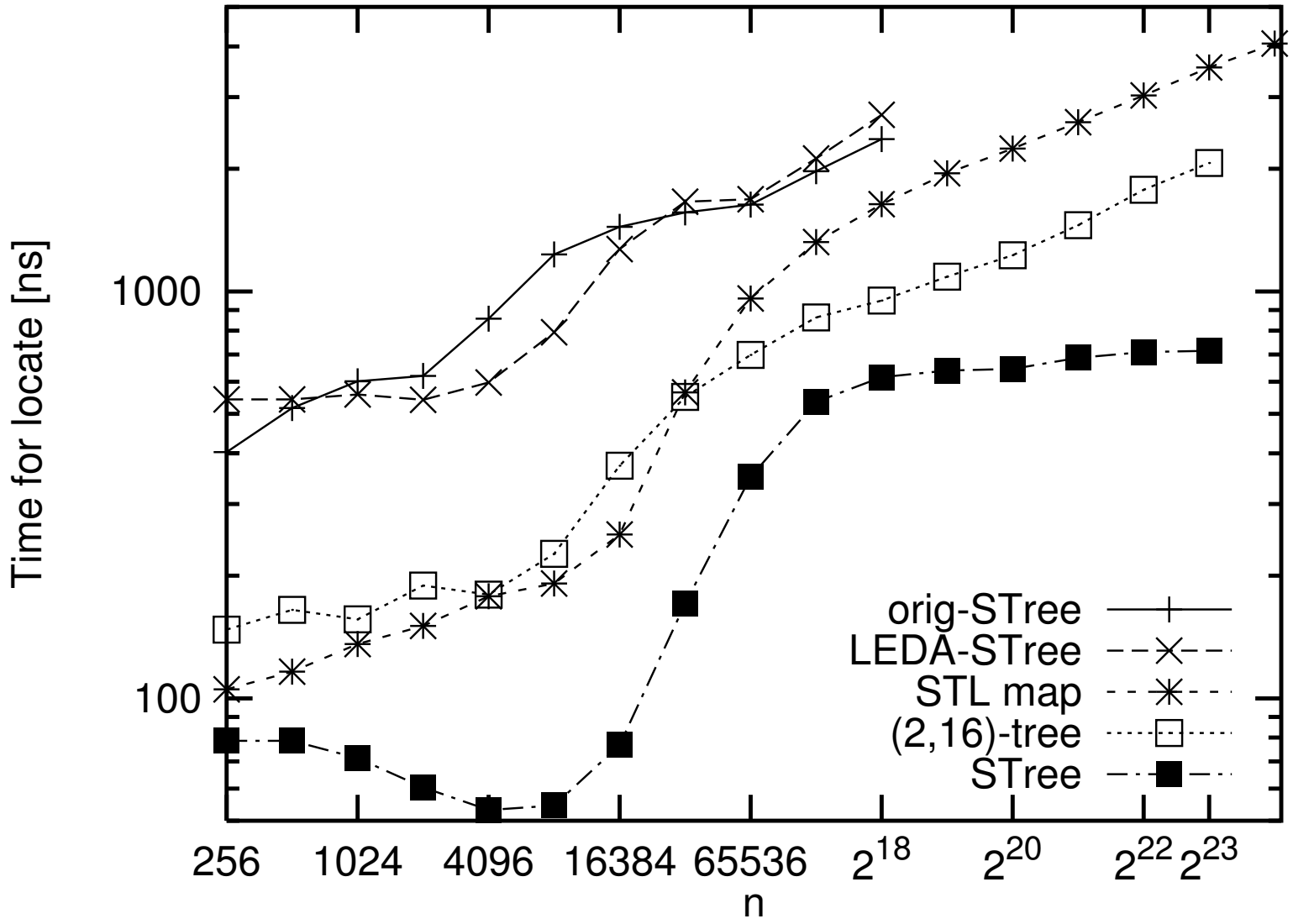
Zusammenfassung

- Suchbäume erlauben viele effiziente Operationen auf sortierten Folgen.
- Oft logarithmische Ausführungszeit
- Der schwierige Teil: logarithmische Höhe erzwingen.
- Augmentierungen $\rightsquigarrow$ zusätzliche Operationen

Mehr zu sortierten Folgen

- Karteikasten $\rightsquigarrow$ Array mit Löchern
- (a, b)-Bäume sind wichtig für externe Datenstrukturen
- Ganzzahlige Schlüssel aus $1..U$
  $\rightsquigarrow$ Grundoperationen in Zeit $\mathrm{O}(\log \log U)$
- Verallgemeinerungen: Zeichenketten, mehrdimensionale Daten

Ein paar Zahlen



Was haben wir noch gelernt?

- Invarianten, Invarianten, Invarianten
- Komplexe verzeigerte Datenstrukturen
- Datenstruktur-Augmentierung
- Unterschied Interface$\leftrightarrow$Repräsentation
- Tradeoff Array, sortierte Liste, Hash-Tabelle