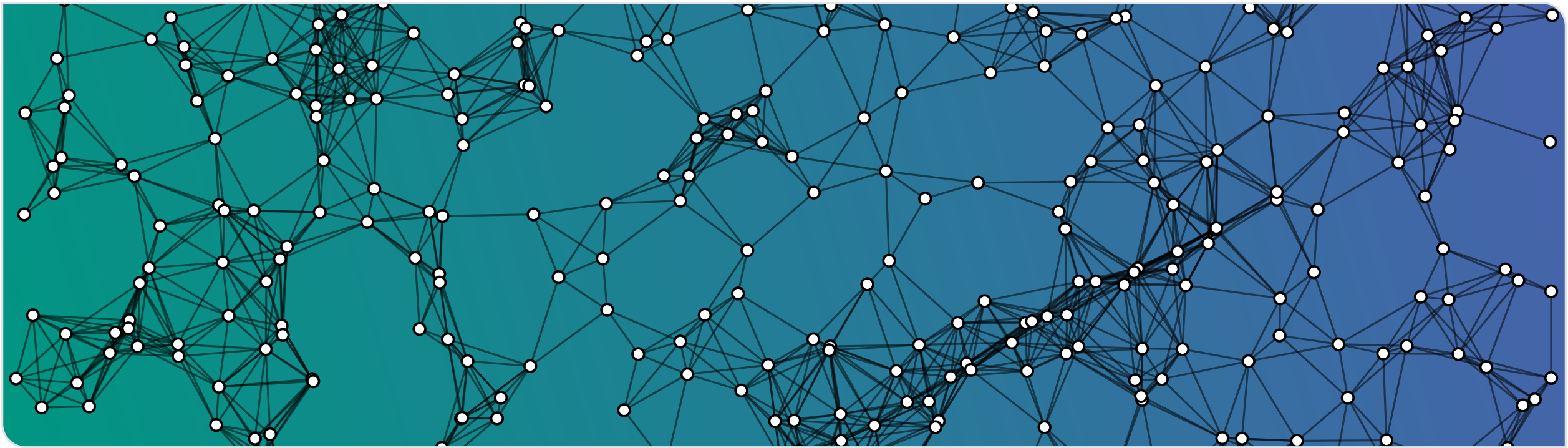
Karlsruher Institut für Technologie

Algorithmen 1

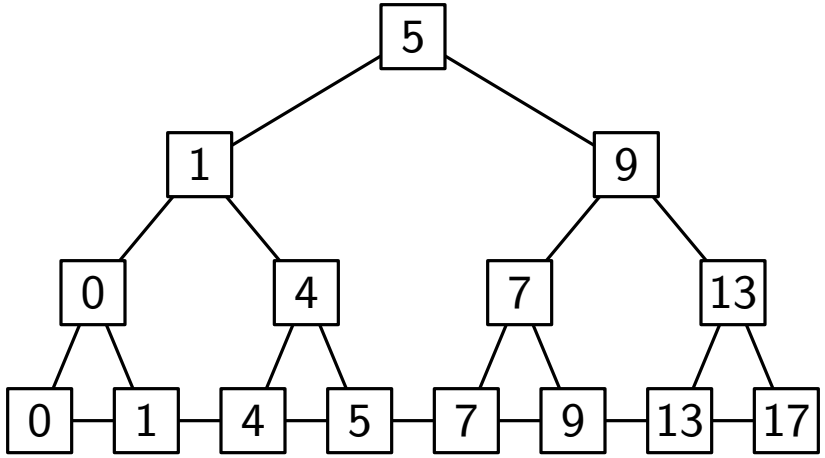
(2, 3)-Bäume – Implementierung



Letztes Mal

Suchbaum zum Verwalten von sortierten Folgen

- verzeigerte Struktur $\rightarrow$ gut zum Einfügen/Löschen
- Baumstruktur $\rightarrow$ simuliert binäre Suche

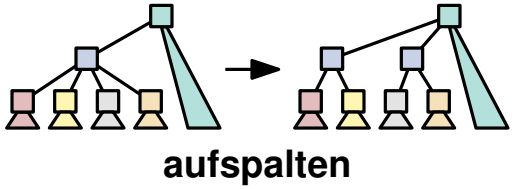


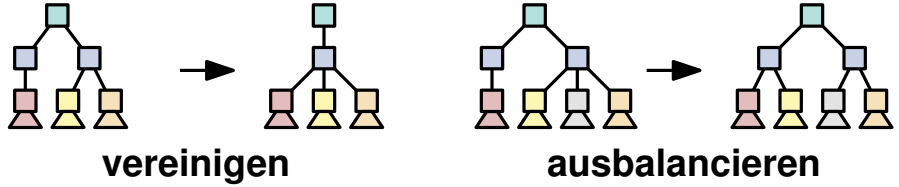
(2, 3)-Baum

- **fast binär:** jeder innere Knoten hat 2 oder 3 Kinder
- **balanciert:** jedes Blatt hat gleiche Tiefe

Erhaltung der Struktur beim Einfügen und Löschen

- Einfügen erzeugt ggf. Knoten mit 4 Kindern $\rightarrow$ aufspalten
- Löschen erzeugt ggf. Knoten mit nur 1 Kind
  $\rightarrow$ vereinigen oder ausbalancieren
- verschiebt das Problem Schrittweise nach oben
- $O(\log n)$ Höhe $\rightarrow$ $O(\log n)$ Laufzeit





Heute

Ausarbeitung der Details

- Was genau speichern wir bei jedem Knoten?
- Wie gehen wir mit Sonderfällen um (z.B. leerer Baum)?
- Wie müssen wir die Schlüssel updaten?
- Wie setzen wir die Datenstruktur in Pseudocode um?

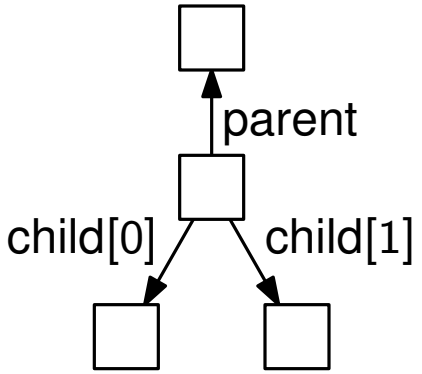
Anmerkungen

- es gibt meist nicht nur eine Antwort auf die Fragen
- Ziele, wenn wir mehrere Möglichkeiten haben etwas umzusetzen:
  - so einfach wie möglich
  - möglichst wenige Sonderbehandlungen
  - Korrektheit + richtige asymptotische Laufzeit
- Lernziel: Umsetzung eines Algorithmus auf hoher Abstraktionsebene in Pseudocode
- nicht das Lernziel: Pseudocode von $(2,3)$-Bäumen auswendig können

Was speichern wir bei jedem Knoten?

Baumstruktur

- jeder Knoten hat Zeiger zu bis zu 4 Kindern
- jeder Knoten hat einen Zeiger zum Elter

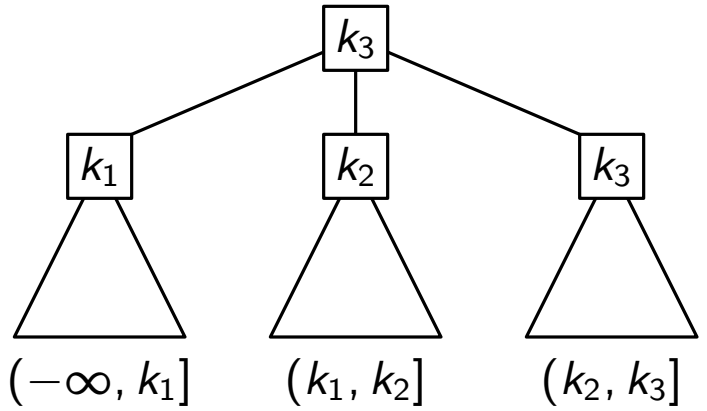


class NODE

Node parent
Array⟨*Node*, 4⟩ child
Key key

Navigation entsprechend der Schlüssel

- letztes Mal
  - jeder Knoten kennt mehrere Schlüssel
  - $\ell - 1$ Schlüssel für ℓ Kinder
- etwas einfacher
  - jeder Knoten a kennt nur einen Schlüssel a.key
  - a.key $=$ größter Schlüssel im Teilbaum unter a
  - Entscheidung für das richtige Kind beim Suchen: entsprechend der Schlüssel der Kinder

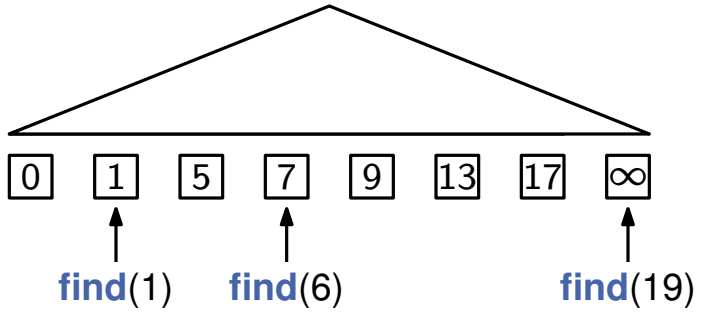


Invariante, die wir nach jeder Operation wiederherstellen

Suchen – Vorüberlegung & Initialisierung

Wonach genau suchen wir?

- Eingabe: ein Schlüssel k
- falls k in der Folge $\rightarrow$ entsprechendes Blatt zurückgeben
- k nicht in der Folge $\rightarrow$ Blatt mit nächst größerem Schlüssel

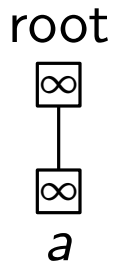


∞-Trick

- Was, wenn k größer als alle existierenden Schlüssel?
- füge ein Dummy-Blatt mit Schlüssel ∞ ein
- vermeidet den Sonderfall, wenn k zu groß
- vermeidet den Sonderfall, für einen leeren Baum

Initialisierung des $(2, 3)$-Baums

- Wurzelknoten mit einem Kind mit Schlüssel ∞
- Schlüssel der Wurzel: ebenfalls ∞



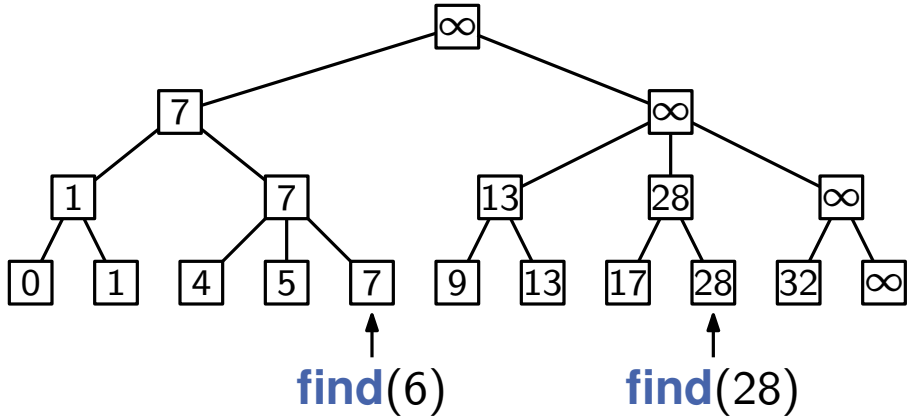

```
class (2, 3)-Tree
    Node root
    init()
        Node a
        a.key := ∞
        a.parent := root
        root.child[0] := a
        root.key := ∞
```

Suche

Erinnerung

- Ziel: finde erstes Blatt mit Schlüssel $\geq k$
- Invariante: a.key $=$ größter Schlüssel im Teilbaum unter a

```
find(Key k)
    Node a := root
    while a is not a leaf do
        i := min{i | k ≤ a.child[i].key}
        a := a.child[i]
    return a
```



Einfügen – Was wollen wir eigentlich genau?

Option 1: insert(k)

- durchsuchbare Menge von Schlüsseln
- Duplikate beim Einfügen überspringen

Option 2: set(k, v)

- Abbildung von Schlüsseln auf Werte
- speichere zusätzlich einen Wert an jedem Blatt
- Blatt mit Schlüssel existiert nicht $\rightarrow$ neues Blatt
- Blatt existiert $\rightarrow$ Wert überschreiben

Heute

- wir betrachten hier Option 1
- Option 2 lässt sich aber analog umsetzen

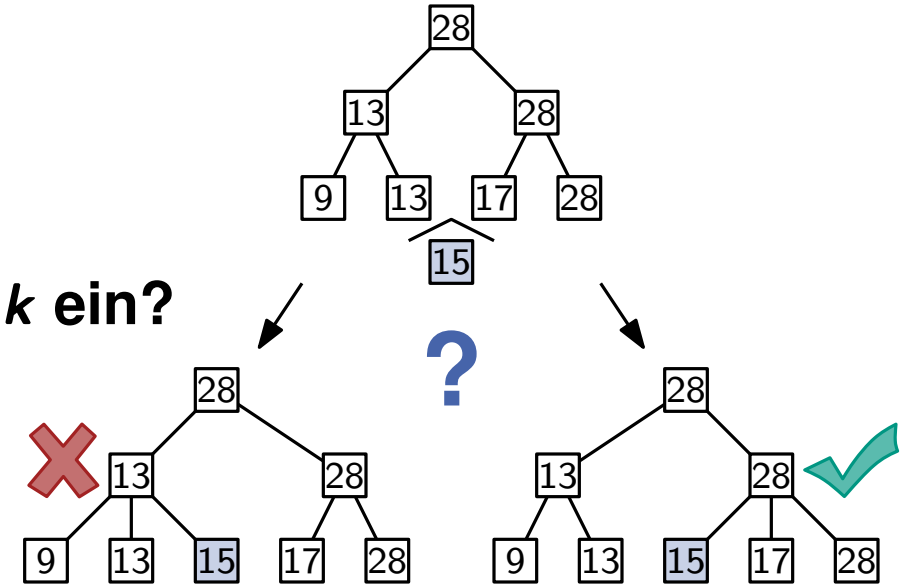
Einfügen – Vorüberlegung

Erinnerung: Neues Blatt einfügen

- Schritt 1: neues Blatt an der richtigen Stelle einfügen
- Schritt 2: Knoten mit 4 Kindern aufspalten

Schritt 1: Wo genau fügen wir den neuen Schlüssel k ein?

- an den Elter des Nachfolgers
- also: neues Blatt wird an **find**(k).parent gehängt



Anmerkung

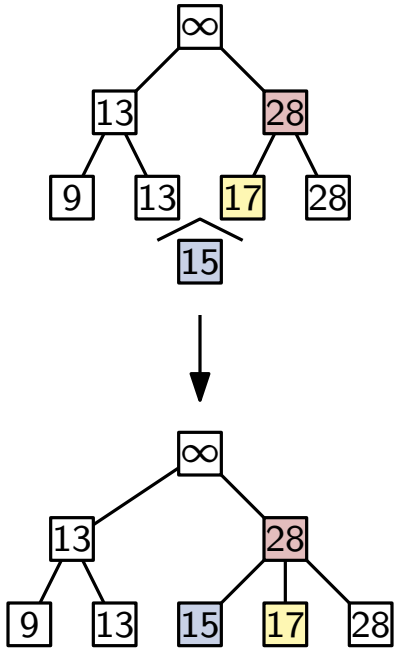
- eigentlich egal: an den Elter des Vorgängers hängen würde auch gehen
- der Elter des Nachfolgers hat aber ein paar Vorteile
  - **find**(k) liefert direkt den Nachfolger
  - Nachfolger existiert immer, dank des ∞-Knotens $\rightarrow$ kein Sonderfall
  - Schlüssel der inneren Knoten bleiben korrekt $\rightarrow$ kein Update nötig

Einfügen

Erinnerung: Neues Blatt einfügen

- Schritt 1: neues Blatt an der richtigen Stelle einfügen
- Schritt 2: Knoten mit 4 Kindern aufspalten

```
insert(Key k)
    if find(k).key = k then return
    Node a := find(k).parent
    Node b := {key :k, parent :a}   // new leaf
    insert b into a.child     // keeping it sorted
    // invariant: keys are correct (largest key in subtree)
    if a has four children then
        split(a)
```



Aufspalten: Knoten mit 4 Kindern

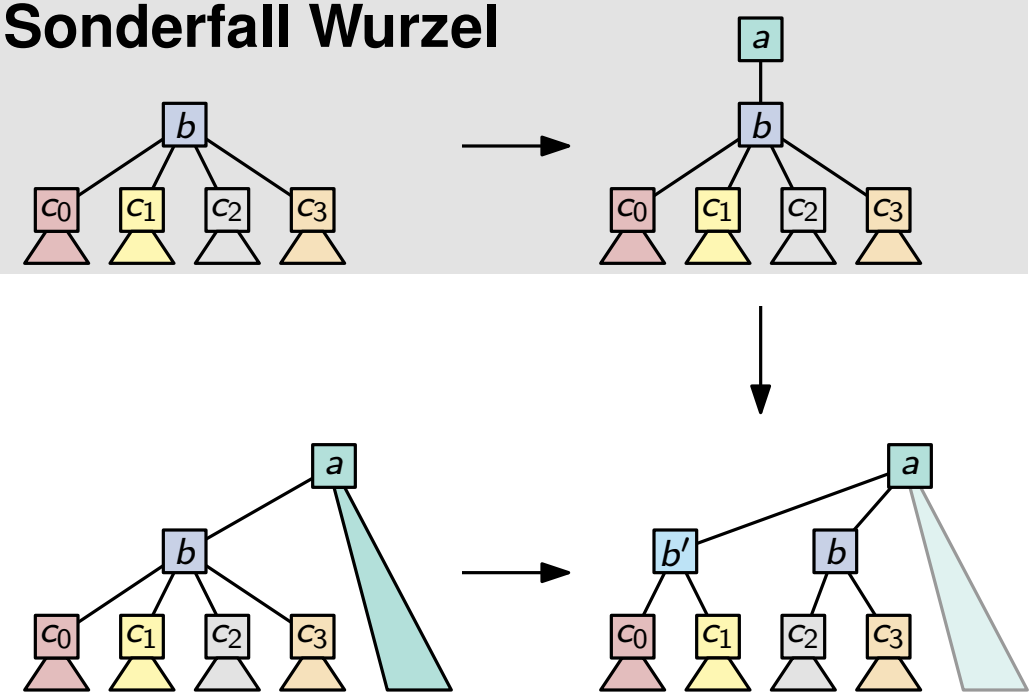
split(*Node* b)

if $b = \text{root}$ **then**
  Node $a := \{\text{key}: \infty; \text{parent}: \bot; \text{child}: \langle b \rangle\}$
  b.parent $:= a$
  root $:= a$

$a := b$.parent
$c_i := b$.child[i] (for $i \in \{0, 1\}$)
Node $b' := \{\text{key}: c_1.\text{key}, \text{parent}: a, \text{child}: \langle c_0, c_1 \rangle\}$
insert b' into a.child
c_0.parent, c_1.parent $:= b'$
remove c_0 and c_1 from b.child
// invariant: keys are correct (largest key in subtree)

if a has four children **then**
  split(a)

Sonderfall Wurzel



ggf. rekursiv aufspalten

Geht das noch etwas schöner?

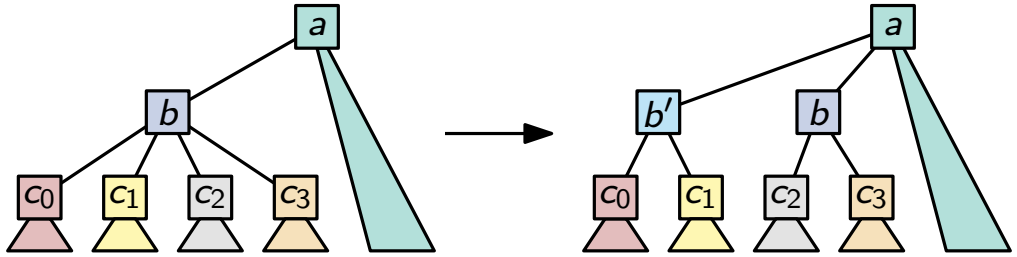
> *Node* b' $:= \{\text{key}: c_1.\text{key}, \text{parent}: a, \text{child}: \langle c_0, c_1 \rangle\}$
> insert b' into a.child
> c_0.parent, c_1.parent $:= b'$
> remove c_0 and c_1 from b.child

Probleme

- schwer zu parsen
- fehleranfällig
- Wiederverwendbarkeit

Eigentlich machen wir drei Dinge

- Knoten b' erstellen
- Kinder c_0, c_1 von b nach b' verschieben
- b' als Kind zu a hinzufügen



> create *Node* b'
> move children c_0 and c_1 from b to b'
> add b' as child to a

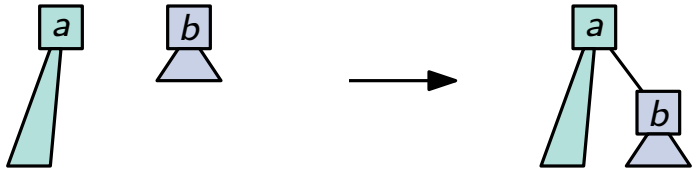
Besser

- lagere grundlegende Operationen in Subroutinen aus

Grundlegende Subroutinen

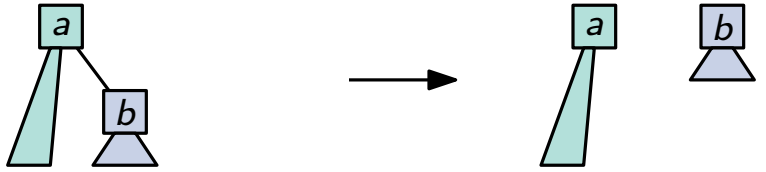
b als Kind von a einfügen

- b an der richtigen Position von a.child einfügen
- b.parent auf a setzen
- a.key auf b.key setzen, falls b das rechtes Kind ist

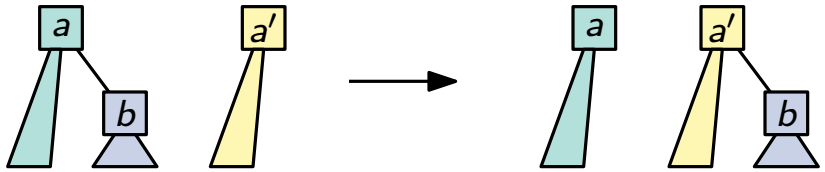


Kind b von a löschen

- b aus a.child entfernen
- b.parent auf $\perp$ setzen
- a.key aktualisieren, falls b das rechtes Kind war



Kind b von Elter a zu a' schieben: löschen + einfügen



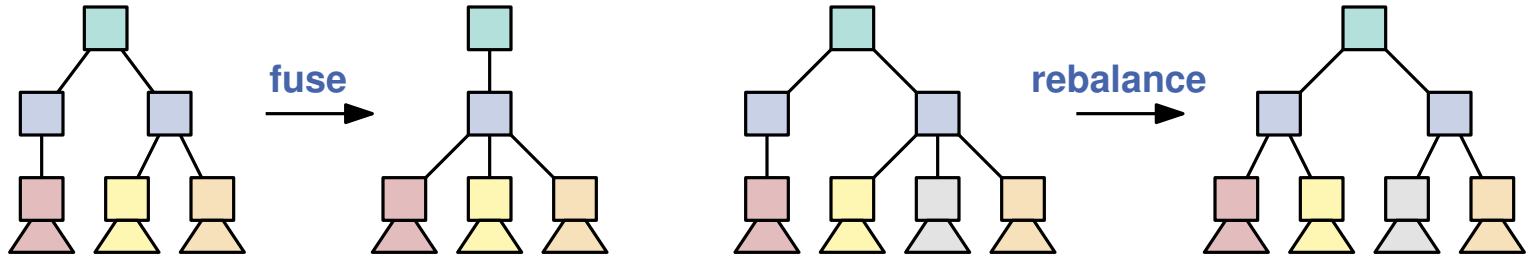
Anmerkung: Aktualisierung des Schlüssels

- machen wir hier nur lokal $\rightarrow$ wird insbesondere nicht an den Elter von a weitergegeben
- daher: Subroutinen immer bottom-up ausführen
- ggf. am Ende nochmal Restpfad zur Wurzel aktualisieren (werden wir beim Löschen gleich noch sehen)

Löschen – Vorüberlegung

Erinnerung

- Schritt 1: entsprechendes Blatt einfach löschen
- Schritt 2: Knoten mit nur einem Kind aufräumen
  (Verschmelzen oder Ausbalancieren)



Rekursive Aufrufe

- nach Verschmelzen: Elter hat ggf. nur noch ein Kind $\rightarrow$ rekursiv aufräumen
- Problem wird Schritt für Schritt weiter nach oben propagiert
- Stopp bei Ausbalancieren oder wenn der Elter bei Verschmelzen vorher drei Kinder hatte

Aktualisierung der Schlüssel

- benutze Operationen von eben $\rightarrow$ Schlüssel werden auf dem Weg nach oben aktualisiert
- stimmt nicht für den restlichen Pfad zur Wurzel, wenn die Rekursion stoppt
- ignorieren wir zunächst und fixen es später

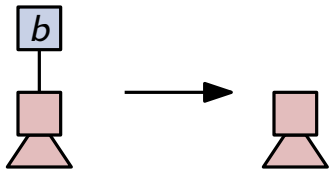
Existierendes Blatt entfernen

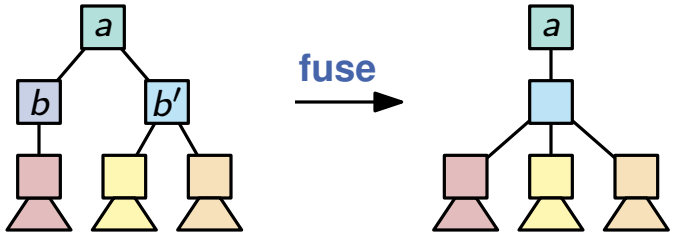
Erinnerung

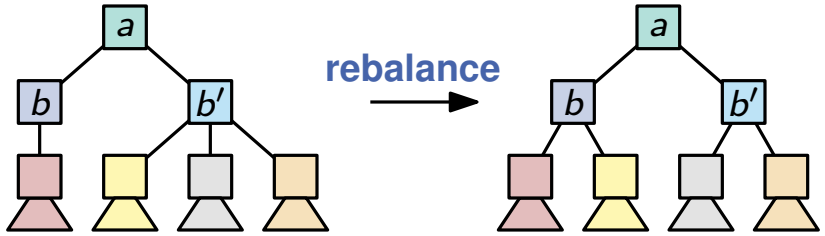
- Schritt 1: entsprechendes Blatt einfach löschen
- Schritt 2: Knoten mit nur einem Kind aufräumen
  (Verschmelzen oder Ausbalancieren)

```
remove(Key k)
    b := find(k)
    if b.key ≠ k then return
    a := b.parent
    remove b as child from a
    if a has only one child then
        fuseOrRebalance(a)
```

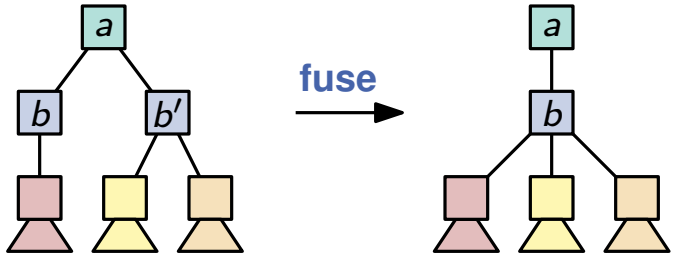
```
fuseOrRebalance(Node b)
    if b = root then
        root := b.child[0]
        remove child root from b
        return
    Node a := b.parent
    Node b' = successor or predecessor of b in a.child
    if b' has two children then
        fuse(b, b')
    else // b' has three children
        rebalance(b, b')
```





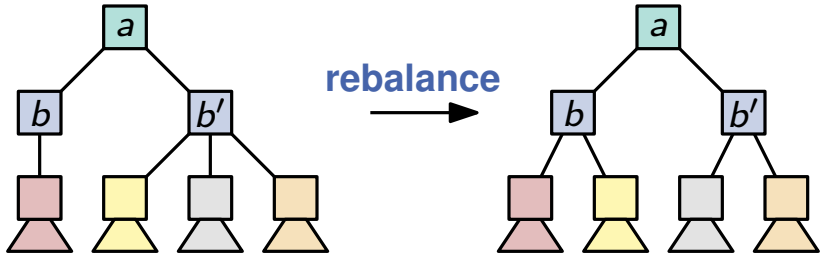


fuse und rebalance



fuse(*Node b*, *Node b′*)

$a := b$.parent
for *Node* $c \in b'$.child **do**
  move child c from b' to b
remove b' as child from a
if a has only one child **then**
  fuseOrRebalance(a)

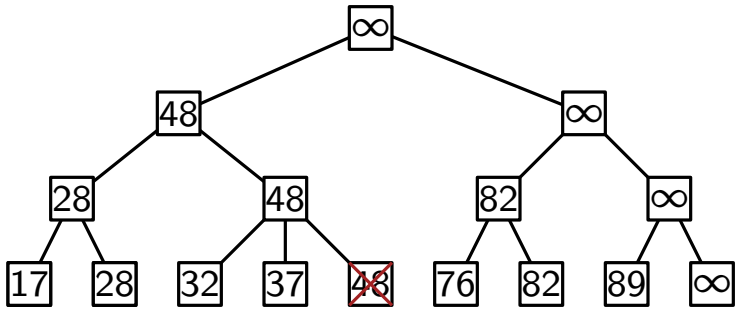


rebalance(*Node b*, *Node b′*)

$a := b$.parent
$\ell := \arg\min_{x \in \{b, b'\}} x$.key
$r := \arg\max_{x \in \{b, b'\}} x$.key
if ℓ has three children **do**
  move child ℓ.child[2] from ℓ to r
else
  move child r.child[0] from r to ℓ

fixKeysOnPathToRoot

```
fixKeysOnPathToRoot(Node a)
    // only ancestors of a have incorrect keys
    while a ≠ root do
        a := a.parent
        a.key := key of right-most child
    // all keys are correct
```

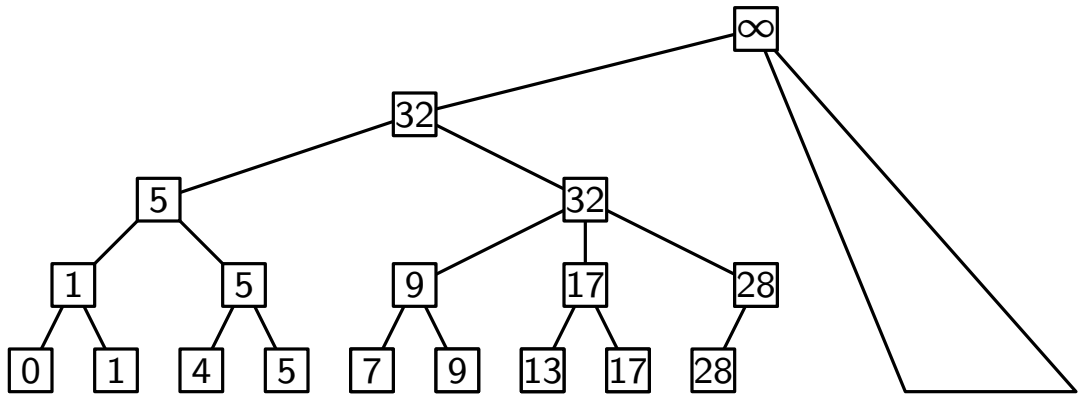



```
remove(Key k)
    // all keys are correct
    b := find(k)
    if b.key ≠ k then return
    a := b.parent
    remove b as child from a
    // only ancestors of a have incorrect keys
    if a has only one child then
        fuseOrRebalance(a)
    else
        fixKeysOnPathToRoot(a)
```

fixKeysOnPathToRoot

fixKeysOnPathToRoot(*Node a*)

```
// only ancestors of a have incorrect keys
while a ≠ root do
    a := a.parent
    a.key := key of right-most child
// all keys are correct
```

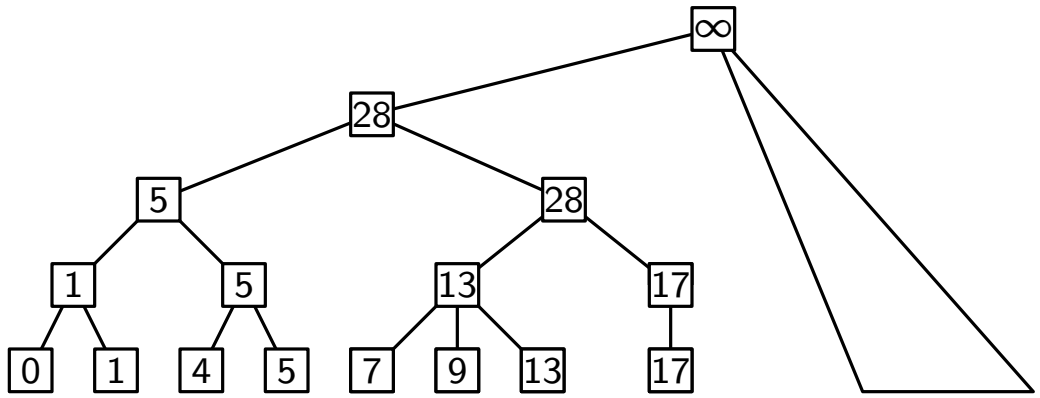


fuse(*Node b*, *Node b′*)

```
// only ancestors of b have incorrect keys
a := b.parent
for Node c ∈ b′.child do
    move child c from b′ to b
remove b′ as child from a
// only ancestors of a have incorrect keys
if a has only one child then
    fuseOrRebalance(a)
else
    fixKeysOnPathToRoot(a)
```

fixKeysOnPathToRoot

```
fixKeysOnPathToRoot(Node a)
    // only ancestors of a have incorrect keys
    while a ≠ root do
        a := a.parent
        a.key := key of right-most child
    // all keys are correct
```




```
remove(Key k)
    // all keys are correct
    b := find(k)
    if b.key ≠ k then return
    a := b.parent
    remove b as child from a
    // only ancestors of a have incorrect keys
    if a has only one child then
        fuseOrRebalance(a)
    else
        fixKeysOnPathToRoot(a)
```

Implementierung $(2, 3)$-Bäume

$(2, 3)$-Bäume

- Ausarbeitung der Details
- insbesondere Verwaltung und Aktualisierung der Schlüssel
- Endergebnis: Pseudocode

Der Weg ist das Ziel

- unwichtig: unser Endergebnis – der Pseudocode selbst
- wichtig: der Prozess, der uns dort hin gebracht hat
  - Sonderfälle vermeiden (∞-Knoten)
  - Ziel: Lesbarkeit, Wiederverwendbarkeit, geringe Fehleranfälligkeit
  - keine Scheu nochmal umzubauen, wenn es zu frickelig wird

Lernziel: von der Algorithmenidee zum Pseudocode

- gegeben die algorithmische Idee
- Details ausarbeiten $\rightarrow$ korrekten Pseudocode bauen