

# 1 Advanced Data Structures

## 1.1 Rot-Schwarz-Bäume

### Definition – Rot-Schwarz-Baum

- Ist ein binärer Suchbaum mit den zusätzlichen Eigenschaften:
  - Jeder Knoten hat die Farbe rot oder schwarz
  - Die Wurzel ist schwarz
  - Wenn ein Knoten rot ist, sind seine Kinder schwarz („Nicht-Rot-Rot-Regel“)
  - Für jeden Knoten hat jeder Pfad zu einem Blatt die selbe Anzahl an schwarzen Knoten
- Halbblätter im RBT sind schwarz
- Schwarzhöhe eines Knoten:  
Eindeutige Anzahl von schwarzen Knoten auf dem Weg zu einem Blatt im Teilbaum des Knoten
- Für leeren Baum gibt Schwarzhöhe = 0 ( $SH(nil) = 0$ )

### Höhe eines Rot-Schwarz-Baums

- $h \leq 2 \cdot \log_2(n + 1)$  ( $n$  Knoten)
- In jedem Unterteilbaum gleiche Anzahl schwarzer Knoten
- Maximal zusätzlich gleiche Anzahl roter Knoten auf diesem Pfad
- Einigermäßen ausbalanciert  $\Rightarrow$  Höhe  $O(\log n)$

Alle folgenden Algorithmen arbeiten mithilfe eines Sentinels (zeigt auf sich selbst)

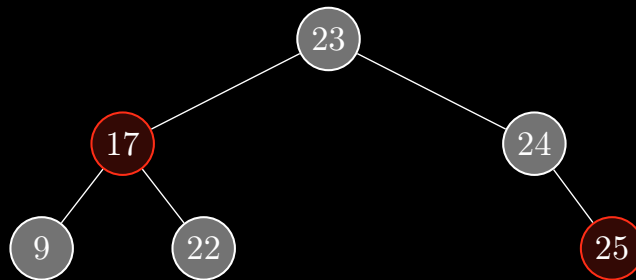


Abbildung 1: Beispielhafte Darstellung wie in 1.3

## Einfügen

1. Finde Elternknoten wie im BST (BST-Einfüge Algorithmus)
2. Knoten als Kind von Elternknoten anfügen
3. Färbe den neuen Knoten rot
4. Wiederherstellen der Rot-Schwarz-Bedingung

```
insert(T,z) //z.left==z.right==nil;
```

```
1  x=T.root; px=T.sent;
2  WHILE x != nil DO           //Bis zum passenden Blatt gehen
3      px=x;
4      IF x.key > z.key THEN
5          x=x.left;
6      ELSE
7          x=x.right;
8  z.parent=px;
9  IF px==T.sent THEN         // Einfügen
10     T.root=z
11 ELSE
12     IF px.key > z.key THEN
13         px.left=z;
14     ELSE
15         px.right=z;
16 z.color=red;                // ab hier anders als bei BST-Insert
17 fixColorsAfterInsertion(T,z);
```

Laufzeit:  $\Theta(h)$  ( $h$  jedoch  $\log n$ )

Hilfsmethode rotateLeft

```
rotateLeft(T,x)
```

```
1  y = x.right;
2  x.right = y.left;
3  IF y.left != nil THEN
4      y.left.parent = x;
5  y.parent = x.parent;
6  IF x.parent == T.sent THEN
7      T.root = y;
8  ELSE
9      IF x == x.parent.left THEN
10         x.parent.left = y;
11     ELSE
12         x.parent.right = y;
13 y.left = x;
14 x.parent = y;
```

**fixColorsAfterInsertion** Beim Aufrufen werden zwei Bedingungen potentiell verletzt:

1. root ist nicht mehr schwarz
2. wenn eine Node rot ist müssen beide Kinder schwarz sein

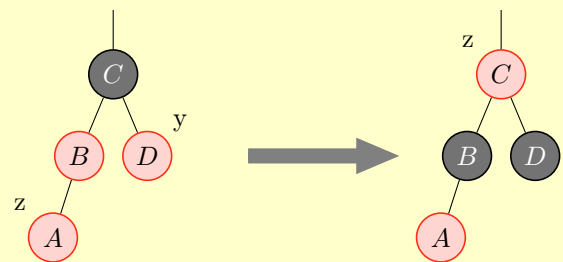
Also müssen wir:

- nach Rotation die root des Baumes ggf auf schwarz setzen
- Überprüfen, ob RSB-Bedingung verletzt wurde
- Bloß wenn das parent auch rot ist kommen wir in Verlegenheit  $\implies$  wir müssen den Algorithmus starten

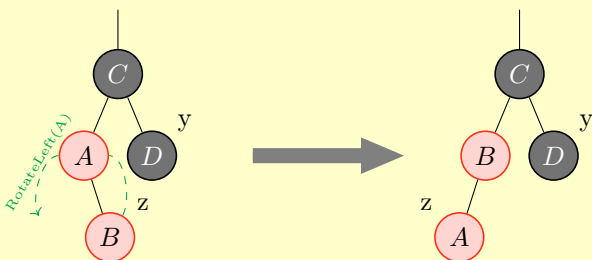
Fälle falls  $z.parent$  rot ist:

1. Onkel ist ebenfalls rot  $\implies$  parent und Onkel werden schwarz gefärbt und Grandparent wird rot gefärbt, z wird zu z.p.p, ggf. Verletzung durch Farbe des neuen z
2. Onkel ist schwarz und z hat andere Kindrichtung wie z.p  $\implies$  zu Fall 3 durch Rotation konvertieren
3. Onkel ist schwarz und z hat gleiche Kindrichtung wie z.p  $\implies$  z.p wird schwarz, z.p.p wird rot, und es wird um z.p.p entgegen der Kindrichtung gedreht. Schleife terminiert nach Fall 3

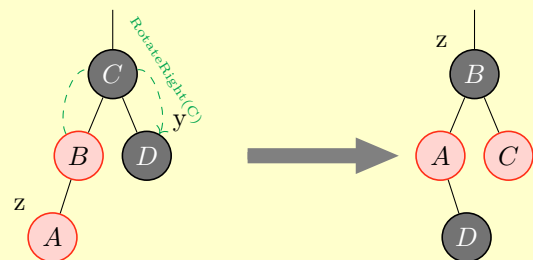
**Fall 1**



**Fall 2**



**Fall 3**



### fixColorsAfterInsertion(T,z)

```

1  WHILE z.parent.color == red DO           // solange der Elternknoten rot ist
2      IF z.parent == z.parent.parent.left THEN // Linkes Kind (if-Fall)
3          y = z.parent.parent.right;
4          IF y != nil AND y.color == red THEN // Fall 1
5              z.parent.color = black;
6              y.color = black;
7              z.parent.parent.color = red;
8              z = z.parent.parent;         // rekursiv nach oben weiterführen
9      ELSE
10         IF z == z.parent.right THEN // Fall 2
11             z = z.parent;
12             rotateLeft(T,z);
13             z.parent.color = black; // Fall 3
14             z.parent.parent.color = red;
15             rotateRight(T, z.parent.parent);
16     ELSE
17         // Tauschen von rechts und links
18     T.root.color = black;                // Setzen der Wurzel auf Schwarz

```

## Löschen

- analog zum binären Suchbaum, aber neue Node erbt Farbe der alten Node
- Wenn „neue“ Node schwarz war  $\Rightarrow$  Fixup
- Verschiedene Fälle, die auch gegenseitig Voraussetzungen füreinander sind

Hilfsroutine transplant:

### transplant(T,u,v)

```
1 IF u.parent==nil THEN
2   T.root=v;
3 ELSE
4   IF u==u.parent.left THEN
5     u.parent.left=v;
6   ELSE
7     u.parent.right=v;
8 IF v != nil THEN
9   v.parent=u.parent;
```

### delete(T,z)

```
1 y=z;
2 y-original-color=y.color;
3 IF z.left == nil;
4   x = z.right;
5   transplant(T,z,z.right);
6 ELSE IF z.right == nil;
7   x = z.left;
8   transplant(T,z,z.left);
9 ELSE
10  y TREE-MINIMUM(z.right);
11  y-original-color=y.color;
12  x=y.right;
13  IF y.p == z
14    x.p = y;
15  ELSE
16    transplant(T,y,y.right);
17    y.right=z.right;
18    y.right.p=y;
19  transplant(T,z,y);
20  y.left=z.left;
21  y.left.p=y;
22  y.color=z.color;
23 IF y-original-color == BLACK
24   deleteFixup(T,x);
```

Laufzeit:  $O(h) = O(\log n)$

Delete kann die RSB-Bedingung verletzen. Das fixup hat vier fälle:

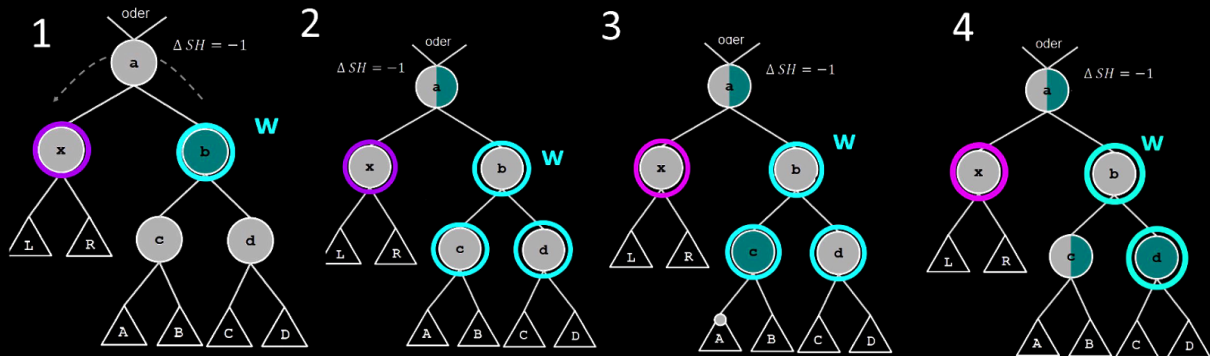


Abbildung 2: Fälle für Fixup bei delete in RSB

1. Knoten  $w$  (Bruder von Knoten  $x$ ) ist rot

- Da  $w$  schwarze Kinder haben muss, können wir die Farben von  $w$  und  $x.p$  wechseln und dann eine Linksdrehung auf  $x.p$  ausführen, ohne eine der rot-schwarzen Eigenschaften zu verletzen.
- Das neue Geschwisterchen von  $x$ , das vor der Rotation eines der Kinder von  $w$  ist, ist jetzt schwarz. Wir haben also Fall 1 in Fall 2, 3 oder 4 konvertiert. Die Fälle 2, 3 und 4 treten auf, wenn der Knoten  $w$  schwarz ist, aber unterschiedliche Farben der Kinder.

2.  $w$  ist schwarz und beide Kinder von  $w$  sind schwarz

- entfernen eines Schwarz von  $x$  und  $w$ , wobei  $x$  nur ein Schwarz hat und  $w$  rot bleibt
- Um Entfernen aus  $x$  und  $w$  zu kompensieren ein zusätzliches Schwarz hinzufügen.
- $w$  konnte entweder Schwarz oder rot sein.
- Wenn  $x.p (= a)$  schwarz, dann Vaterknoten  $\Delta SH = -1$ ; verfahren rekursiv mit  $x.p (= a)$  als neuem  $x$ ;  $\Rightarrow$  wir wiederholen die while-Schleife mit  $x.p$  als neuem Knoten  $x$ .

3.  $w$  ist schwarz,  $w$ 's linkes Kind ist rot und  $w$ 's rechtes Kind ist schwarz

- Farben von  $w$  und seinem linken Kind links ändern und dann eine Rechtsdrehung auf  $w$  ausführen
- Das neue Geschwister  $w$  von  $x$  ist jetzt ein schwarzer Knoten mit einem roten rechten Kind

$\Rightarrow$  Fall 3 in Fall 4 transformiert.

4.  $w$  ist schwarz und das rechte Kind von  $w$  ist rot

- $w$  erbt die Farbe von  $x.p$
- $x.p$  wird schwarz
- $w.right$  wird schwarz

$\Rightarrow$  LinksRotation von  $x.p$

- $x$  als Wurzel festlegen, wird der Whileloop beendet, wenn die Schleifenbedingung getestet wird.

### deleteFixup(T,z)

```
1  WHILE x != T.root and x.color == BLACK
2      IF x==x.p.left
3          w=x.p.right;
4          IF w.color == RED // Case 1
5              w.color=BLACK;
6              x.p.color=RED;
7              rotateLeft(T,x.p);
8              w=x.p.right;
9          IF w.left.color == w.right.color == BLACK // Case 2
10             w.color = RED;
11             x=x.p;
12         ELSE IF w.right.color == BLACK // Case 3
13             w.left.color = BLACK;
14             w.color=RED;
15             rotateRight(T,w);
16             w=x.p.right;
17             w.color=x.p.color; // Case 4
18             x.p.color=BLACK;
19             w.right.color=BLACK;
20             rotateLeft(T,x.p);
21             x=T.root;
22     ELSE
23         // same as above with "right" and "left" exchanged
24     x.color=BLACK;
```

### Worst-Case-Laufzeiten

Einfügen  $\Theta(\log n)$

Löschen  $\Theta(\log n)$

Suchen  $\Theta(\log n)$

## 1.2 AVL-Bäume

### Definition – AVL-Baum

Binärer Suchbaum, aber für Balance in **jedem** Knoten nur  $-1, 0$ , oder  $1$  erlaubt.

Balance für  $x$ :  $B(x) = \text{Höhe}(\text{rechter Teilbaum}) - \text{Höhe}(\text{linker Teilbaum})$

$h \leq 1.441 \cdot \log n$  (optimierte Konstanten - 1,441 vs 2 (RBT))

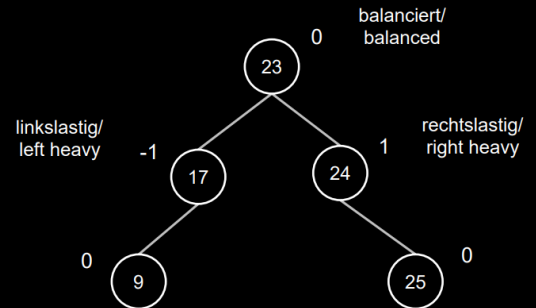


Abbildung 3: Beispiel Balance AVL Baum

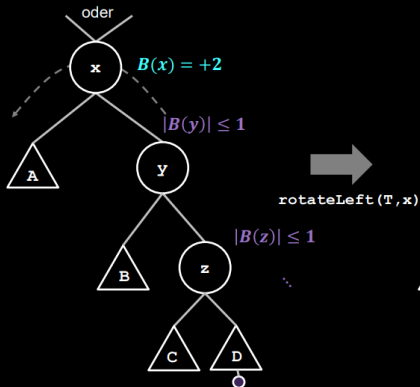
### AVL vs. Rot-Schwarz-Bäume

AVL	<ul style="list-style-type: none"><li>• Einfügen und Löschen verletzen in der Regel öfter die Baum-Bedingung</li><li>• Aufwendiger zum Rebalancieren</li></ul>
Rot-Schwarz	<ul style="list-style-type: none"><li>• Suchen dauert evtl. länger</li></ul>
Konklusion	<ul style="list-style-type: none"><li>• AVL geeigneter, wenn mehr Such-Operationen und weniger Einfügen und Löschen</li></ul>
Gemeinsamkeiten	<ul style="list-style-type: none"><li>• <math>AVL \subset \text{Rot-Schwarz}</math></li><li>• AVL-Baum <math>\Rightarrow</math> Rot-Schwarz-Baum mit Höhe <math>\lceil \frac{h+1}{2} \rceil</math></li><li>• Für jede Höhe <math>h \geq 3</math> gibt es einen RBT, der kein AVL-Baum ist (<math>AVL \neq \text{RBT}</math>)</li></ul>

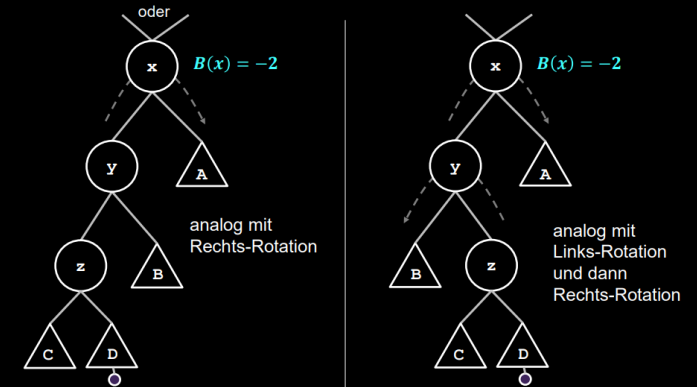
## Einfügen

- Einfügen funktioniert wie beim Binary Search Tree mit Sentinel
- Erfordert danach jedoch Rebalancieren weiter oben im Baum
- Rebalancieren: (verschiedene Fälle)

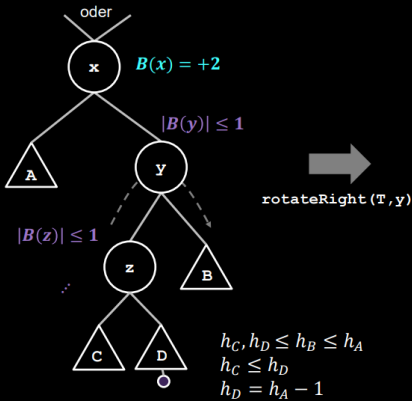
### Rebalancieren: Fall I



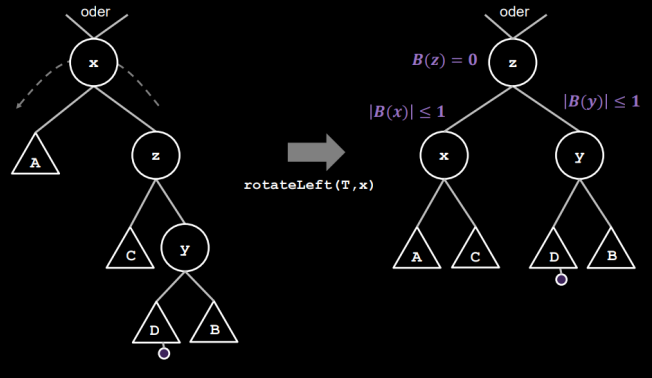
### Rebalancieren: Fälle III+IV



### Rebalancieren: Fall II (erste Rotation)



### Rebalancieren: Fall II (zweite Rotation)



## Löschen

- Analog zum binären Suchbaum
- Rebalancieren (eventuell bis zur Wurzel) notwendig

## Worst-Case-Laufzeiten

Einfügen  $\Theta(\log n)$

Löschen  $\Theta(\log n)$

Suchen  $\Theta(\log n)$

- theoretisch bessere Konstanten als RBT
- in Praxis aber nur unwesentlich schneller



## 1.3 Splay-Bäume

### Definition — Splay-Baum

- selbst-organisierender Baum
- Ansatz: Einmal angefragte Werte werden wahrs. noch öfter angefragt
- Angefragte Werte nach oben schieben
- Splay-Bäume sind eine Untermenge von Rot-Schwarz-Bäumen ( $\subseteq$ )

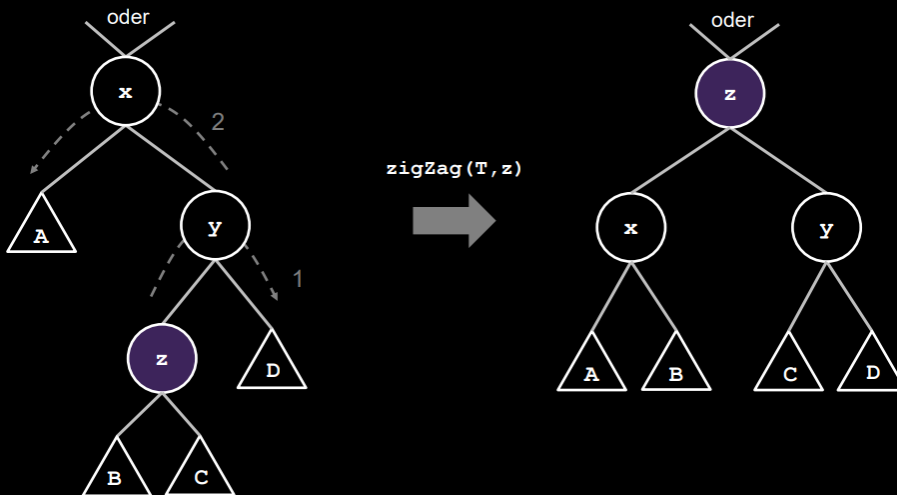
### Splay-Operationen

- Suchen oder Einfügen: Spüle gesuchten oder neu eingefügten Knoten an die Wurzel
- Splay: Folge von Zig-, Zig-Zig-, Zig-Zag-Operationen

#### splay(T,z)

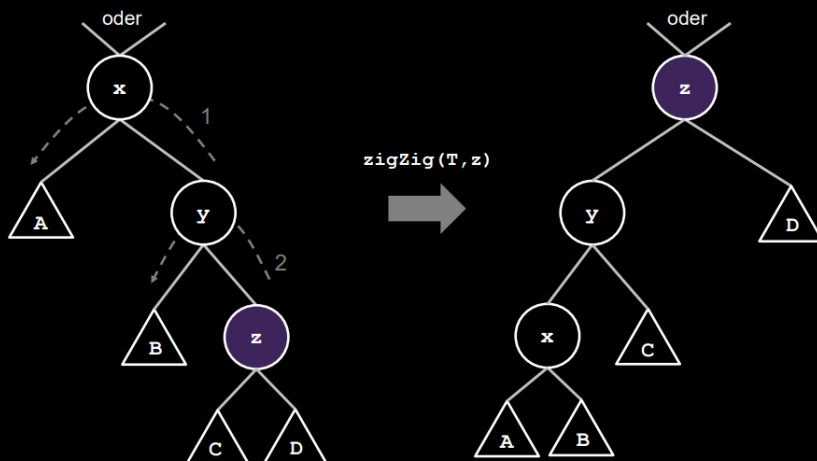
```
1 WHILE z != T.root DO
2   IF z.parent.parent == nil THEN
3     zig(T,z);
4   ELSE
5     IF z == z.parent.parent.left.left OR
6       z == z.parent.parent.right.right THEN
7       zigZig(T,z);
8     ELSE
9       zigZag(T,z);
```

### Zig-Zag-Operation =Rechts-Links- oder Links-Rechts-Rotation



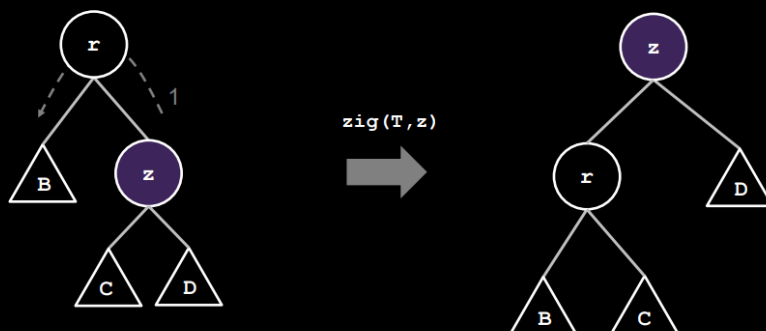
## Zig-Zig-Operation

=Links-Links- oder Rechts-Rechts-Rotation



## Zig-Operation

=einfache Links- oder Rechts-Rotation



### Suchen

- Laufzeit:  $O(h)$
- Suche des Knotens wie im BST
- Hochspülen des gefundenen Knotens (alternativ zuletzt besuchter Knoten, falls nicht gefunden)

### Einfügen

- Laufzeit:  $O(h)$
- Suche der Position wie im BST
- Einfügen und danach hochspülen des eingefügten Knotens

### Löschen

- Laufzeit:  $O(h)$
- 1. Spüle gesuchten Knoten per Splay-Operation nach oben
- 2. Lösche den gesuchten Knoten (Wenn einer der beiden entstehenden Teilbäume leer, dann fertig)
- 3. Spüle den größten Knoten im linken Teilbaum nach oben (kann kein rechtes Kind haben)
- 4. Hänge rechten Teilbaum an größten Knoten aus 3. an

### Laufzeit Splay-Bäume

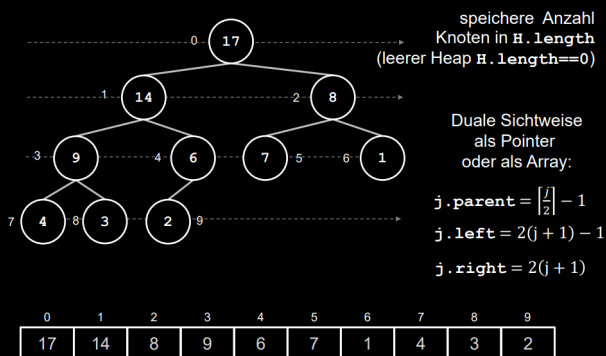
- Amortisierte Laufzeit: Laufzeit pro Operation über mehrere Operationen hinweg
- Worst-Case-Laufzeit pro Operation:  $O(\log_n n)$

## 1.4 Binäre Max-Heaps

### Definition — Binäre Max-Heaps

- Heaps sind keine BSTs
- Eigenschaften von binären Max-Heaps:
  - bis auf das unterste Level vollständig und dort von links gefüllt ist
  - Für alle Knoten gilt:  $x.\text{parent}.key \geq x.key$
  - Maximum des Heaps steht damit in der Wurzel
- $h \leq \log n$ , da Baum fast vollständig

### Heaps durch Arrays



### Einfügen

**Idee — Einfügen** Einfügen und danach Vertauschen nach oben, bis Max-Eigenschaft wieder erfüllt ist

Code:

`insert(H,k) // als unbeschränktes Array`

```
1 H.length = H.length + 1;
2 H.A[H.length-1] = k;
3 i = H.length - 1;
4 WHILE i > 0 AND H.A[i] > H.A[i.parent]
5     SWAP(H.A, i, i.parent);
6     i = i.parent;
```

Laufzeit:  $O(h) = O(\log n)$

## Lösche Maximum

1. Ersetze Maximum durch „letztes“ Blatt
2. Vertausche Knoten durch Maximum der beiden Kinder (heapify)

### extract-max(H)

```
1 IF isEmpty(H) THEN return error "underflow";
2 ELSE
3   max = H.A[0];
4   H.A[0] = H.A[H.length - 1];
5   H.length = H.length - 1;
6   heapify(H, 0);
7   return max;
```

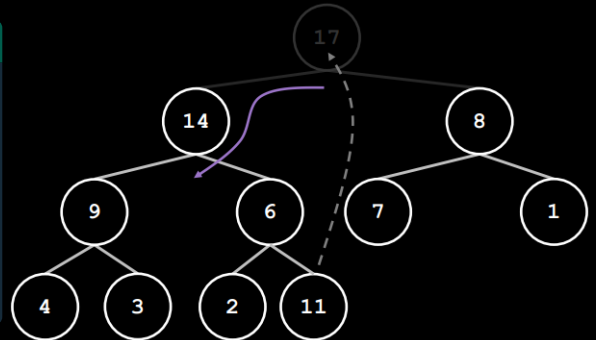


Abbildung 4: Beispiel Löschen des Maximums im Binären Max-Heap

### heapify(H, i)

```
1 maxind = i;
2 IF i.left < H.length AND H.A[i] < H.A[i.left] THEN
3   maxind = i.left;
4 IF i.right < H.length AND H.A[maxind] < H.A[i.right] THEN
5   maxind = i.right;
6 IF maxind != i THEN
7   SWAP(H.A, i, maxind);
8   heapify(H, maxind);
```

## Heap-Konstruktion aus Array

- Blätter sind für sich triviale Max-Heaps
- Bauen von Max-Heaps für Teilbäume mithilfe Rekursion per **heapify**
- (Array nicht unbedingt in richtiger Reihenfolge)

### buildHeap(H.A) // Array in H.A

```
1 H.length = A.length;
2 FOR i = ceil((H.length-1)/2) - 1 DOWNTO 0 DO
3   heapify(H.A, i);
```

## Heap-Sort

- Idee: Bauen des Heaps aus Array und dann (wiederholte) Extraktion des Maximums

### heapSort(H.A)

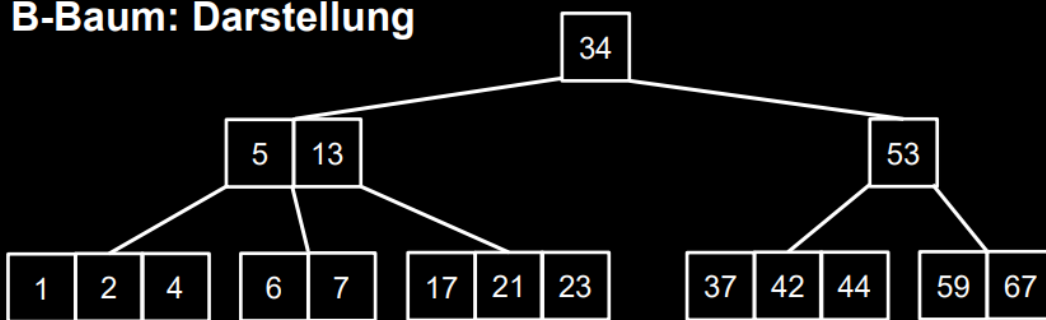
```
1 buildHeap(H.A)           // Bauen des Heaps
2 WHILE !isEmpty(H) DO
3   PRINT extract-max(H);   // Ausgabe des Maximums bis Heap leer ist
```

## 1.5 B-Bäume

### Definition – B-Baum

- Jeder B-Baum hat einen angegebenen Grad also z.B.  $t = 2$
- Eigenschaften:
  - Wurzel zwischen  $[1, \dots, 2t - 1]$  Werte
  - Knoten zwischen  $[t - 1, \dots, 2t - 1]$  Werte
  - Werte innerhalb eines Knotens aufsteigend geordnet
  - Blätter haben alle die gleiche Höhe
  - Jeder innere Knoten mit  $n$  Werten hat  $n + 1$  Kinder, sodass gilt:  
 $k_0 \leq \text{key}[0] \leq k_1 \leq \text{key}[1] \leq \dots \leq k_{n-1} \leq \text{key}[n] \leq k_n$

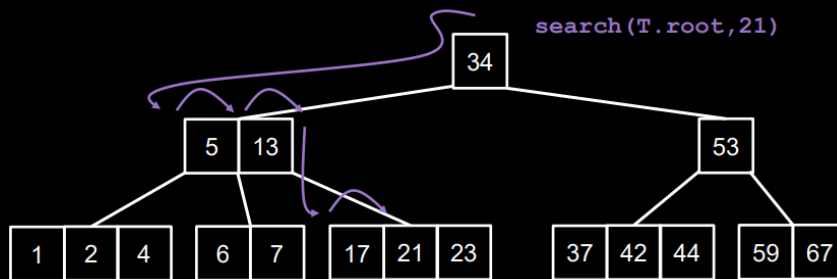
### B-Baum: Darstellung



- |   |   |
|---|---|
| <b><code>x.n</code></b>                           | – Anzahl Werte eines Knoten <b><code>x</code></b>   |
| <b><code>x.key[0], ..., x.key[x.n-1]</code></b>   | – (geordnete) Werte in Knoten <b><code>x</code></b> |
| <b><code>x.child[0], ..., x.child[x.n]</code></b> | – Zeiger auf Kinder in Knoten <b><code>x</code></b> |

- Höhe B-Baum:  $h \leq \log_t \frac{n+1}{2}$  (Grad  $t$  und  $n$  Werte)
- B-Baum wird für größere  $t$  flacher

## Suche



search(x, k)

```
1 WHILE x != nil DO
2     i = 0;
3     WHILE i < x.n AND x.key[i] < k DO
4         i++;
5     IF i < x.n AND x.key[i] == k THEN
6         return(x, i);
7     ELSE
8         x = x.child[i];
9 return nil;
```

## Einfügen

- Einfügen erfolgt immer in einem Blatt
- Falls das Blatt voll ist, muss jedoch gesplittet werden
- $\Rightarrow$  Beim Durchlaufen des Baumes an jeder notwendigen vollen Position splitten
- Splitten:
  - Bricht volle Node auf und fügt mittleren Wert zur Elternnode hinzu
  - Aus den anderen Werten entstehen nun jeweils eigene Kinder
  - An der Wurzel splitten erzeugt neue Wurzel und erhöht Baumhöhe um eins
- Ablauf zusammengefasst:
  1. Start bei Wurzel, falls kein Platz mehr splitten
  2. Durchlaufen des Baumes bis zur richtigen Position und immer, falls voll, splitten
  3. Einfügen der Node (fertig)

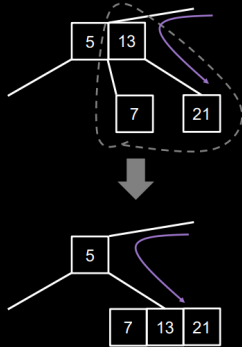
insert(T, z)

```
1 Wenn Wurzel schon 2t-1 Werte hat, dann splitte Wurzel
2 Suche rekursiv Einfügeposition:
3     Wenn zu besuchendes Kind 2t-1 Werte hat, splitte es erst
4 Füge z in Blatt ein
```

## Löschen

- Wenn Blatt noch mehr als  $t - 1$  Werte, kann der Wert einfach entfernt werden
- Allerdings durchlaufen wir hier den Baum auch wieder von oben und stellen gewisse Voraussetzungen her
- Durchlaufen des Baumes von oben und Anwendung der folgenden Algorithmen

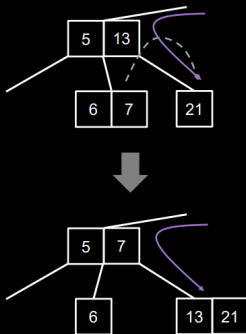
$t = 2$



Allgemeines Verschmelzen:

- Kind und rechter und linker Geschwisterknoten (sofern existent) nur  $t - 1$  Werte
- Wenn Elternknoten vorher min.  $t$  Werte  
⇒ keine Änderung oberhalb notwendig

$t = 2$



Allgemeines Rotieren/Verschieben:

- Kind nur  $t - 1$  Werte
- Geschwister jedoch mehr als  $t - 1$  Werte
- keine Änderung oberhalb notwendig

Code:

```
delete(T, k)
```

```
1 Wenn Wurzel nur 1 Wert und beide Kinder t-1 Werte,  
2 verschmelze Wurzel und Kinder (reduziert Höhe um 1)  
3 Suche rekursiv Löschposition:  
4     Wenn zu besuchendes Kind nur t-1 Werte,  
5     verschmelze es oder rotiere/verschiebe  
6 Entferne Wert k im inneren Knoten/Blatt  
7 // Ohne Probleme, aufgrund vorheriger Anpassung
```

## Laufzeiten

Einfügen  $\Theta(\log_t n)$

Löschen  $\Theta(\log_t n)$

Suchen  $\Theta(\log_t n)$

- Nur vorteilhaft wenn Daten blockweise eingelesen werden
- $\mathcal{O}$ -Notation versteckt hier konstanten Faktor  $t$  für Suche innerhalb eines Knotens