1 Advanced Data Structures

1.1 Rot-Schwarz-Bäume

Definition — Rot-Schwarz-Baum

- Ist ein binärer Suchbaum mitden 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 oder Halbblatt 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
- Einigermaßen ausbalanciert $\Rightarrow$ Höhe $O(log\ n)$

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

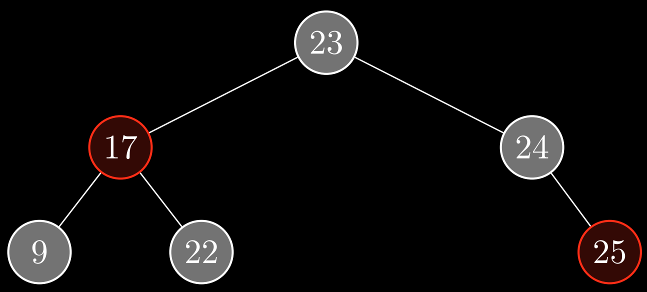


Abbildung 1: Beispielhafte Darstellung wie in 1.3

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;

```
x=T.root; px=T.sent;
WHILE x != nil DO                    //Bis zum passenden Blatt gehen
    px=x;
    IF x.key > z.key THEN
        x=x.left;
    ELSE
        x=x.right;
z.parent=px;
IF px==T.sent THEN                   // Einfügen
    T.root=z
ELSE
    IF px.key > z.key THEN
        px.left=z;
    ELSE
        px.right=z;
z.color=red;                         // ab hier anders als bei BST-Insert
fixColorsAfterInsertion(T,z);
```

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

Hilfsmethode `rotateLeft`

rotateLeft(T,x)

```
y = x.right;
x.right = y.left;
IF y.left != nil THEN
    y.left.parent = x;
y.parent = x.parent;
IF x.parent == T.sent THEN
    T.root = y;
ELSE
    IF x == x.parent.left THEN
        x.parent.left = y;
    ELSE
        x.parent.right = y;
y.left = x;
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

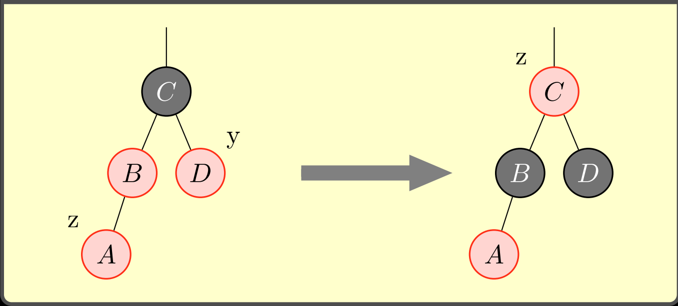
Also müssen wir:

- 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 $\Longrightarrow$ wir müssen den Algorithmus starten

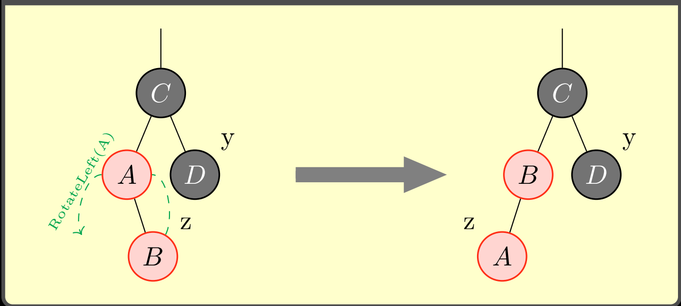
Fälle falls $z.parent$ rot ist:

1. Onkel ist ebenfalls rot $\Rightarrow$ 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 $\Rightarrow$ zu Fall 3 durch Rotation konvertieren
3. Onkel ist schwarz und z hat gleiche Kindrichtung wie z.p $\Rightarrow$ 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

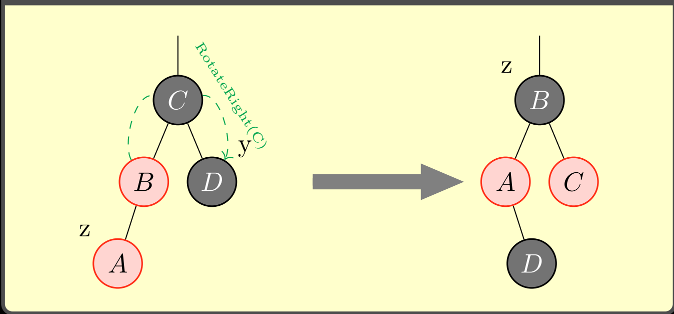
Fall 1



Fall 2



Fall 3



fixColorsAfterInsertion(T,z)

```
WHILE z.parent.color == red DO                  // solange der Elternknoten rot ist
    IF z.parent == z.parent.parent.left THEN    // Linkes Kind (if-Fall)
        y = z.parent.parent.right;
        IF y != nil AND y.color == red THEN     // Fall 1
            z.parent.color = black;
            y.color = black;
            z.parent.parent.color = red;
            z = z.parent.parent;                // rekursiv nach oben weiterführen
        ELSE
            IF z == z.parent.right THEN         // Fall 2
                z = z.parent;
                rotateLeft(T,z);
            z.parent.color = black;             // Fall 3
            z.parent.parent.color = red;
            rotateRight(T, z.parent.parent);
    ELSE
        // Tauschen von rechts und links
    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)

```
IF u.parent==nil THEN
    T.root=v;
ELSE
    IF u==u.parent.left THEN
        u.parent.left=v;
    ELSE
        u.parent.right=v;
IF v != nil THEN
    v.parent=u.parent;
```

delete(T,z)

```
y=z;
y-original-color=y.color;
IF z.left == nil;
    x = z.right;
    transplant(T,z,z.right);
ELSE IF z.right == nil;
    x = z.left;
    transplant(T,z,z.left);
ELSE
    y TREE-MINIMUM(z.right);
    y-original-color=y.color;
    x=y.right;
    IF y.p == z
        x.p = y;
    ELSE
        transplant(T,y,y.right);
        y.right=z.right;
        y.right.p=y;
    transplant(T,z,y);
    y.left=z.left;
    y.left.p=y;
    y.color=z.color;
IF y-original-color == BLACK
    deleteFixup(T,x);
```

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

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

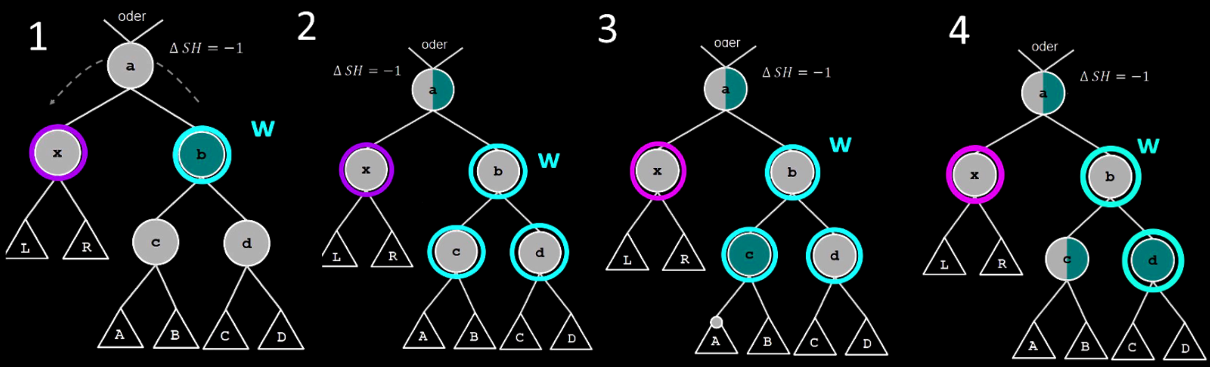


Abbildung 2: Fälle für Fixup bei delete in RSB

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 $\Delta SH = -1$; verfahre 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)

```
WHILE x != T.root and x.color == BLACK
    IF x==x.p.left
        w=x.p.right;
        IF w.color == RED                           // Case 1
            w.color=BLACK;
            x.p.color=RED;
            rotateLeft(T,x.p);
            w=x.p.right;
        IF w.left.color == w.right.color == BLACK   // Case 2
            w.color = RED;
            x=x.p;
        ELSE IF w.right.color == BLACK              // Case 3
            w.left.color = BLACK;
            w.color=RED;
            rotateRight(T,w);
            w=x.p.right;
        w.color=x.p.color;                          // Case 4
        x.p.color=BLACK;
        w.right.color=BLACK;
        rotateLeft(T,x.p);
        x=T.root;
    ELSE
        // same as above with "right" and "left" exchanged
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: $B(x) =$ Höhe(rechter Teilbaum) - Höhe(linker Teilbaum)

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

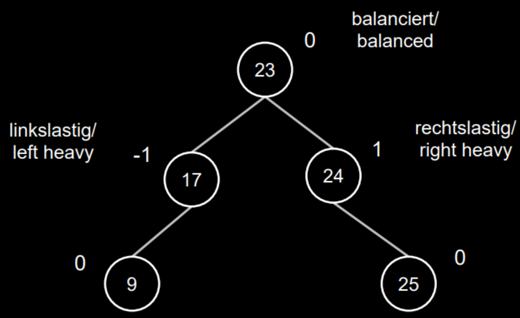


Abbildung 3: Beispiel Balance AVL Baum

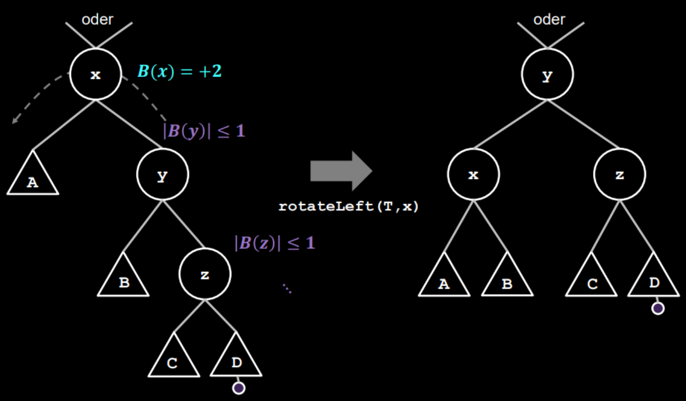
AVL vs. Rot-Schwarz-Bäume

AVL	• Einfügen und Löschen verletzen in der Regel öfter die Baum-Bedingung • Aufwendiger zum Rebalancieren
Rot-Schwarz	• Suchen dauert evtl. länger
Konklusion	• AVL geeigneter, wenn mehr Such-Operationen und weniger Einfügen und Löschen
Gemeinsamkeiten	• AVL $\subset$ Rot-Schwarz • AVL-Baum $\Rightarrow$ Rot-Schwarz-Baum mit Höhe $\lceil \frac{h+1}{2} \rceil$ • Für jede Höhe $h \geq 3$ gibt es einen RBT, der kein AVL-Baum ist (AVL $\neq$ RBT)

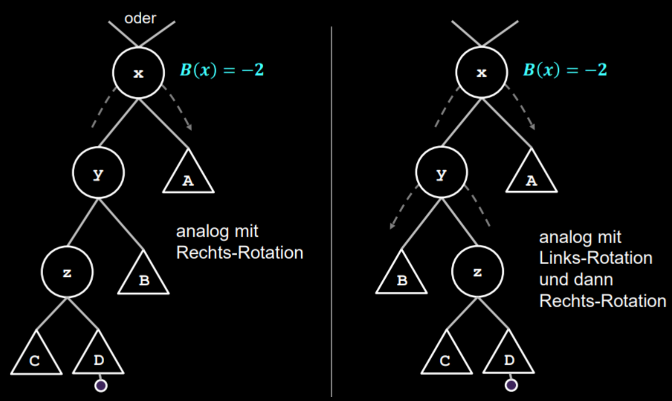
Einfügen

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

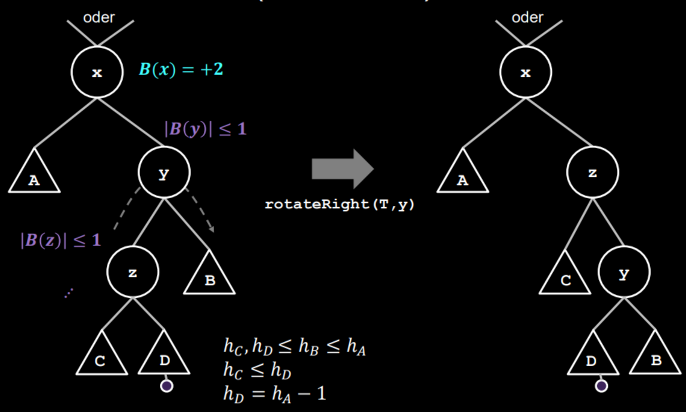
Rebalancieren: Fall I



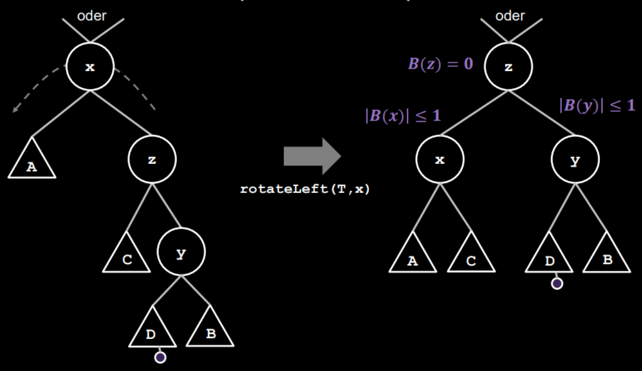
Rebalancieren: Fälle III+IV



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 ($\subseteq$)

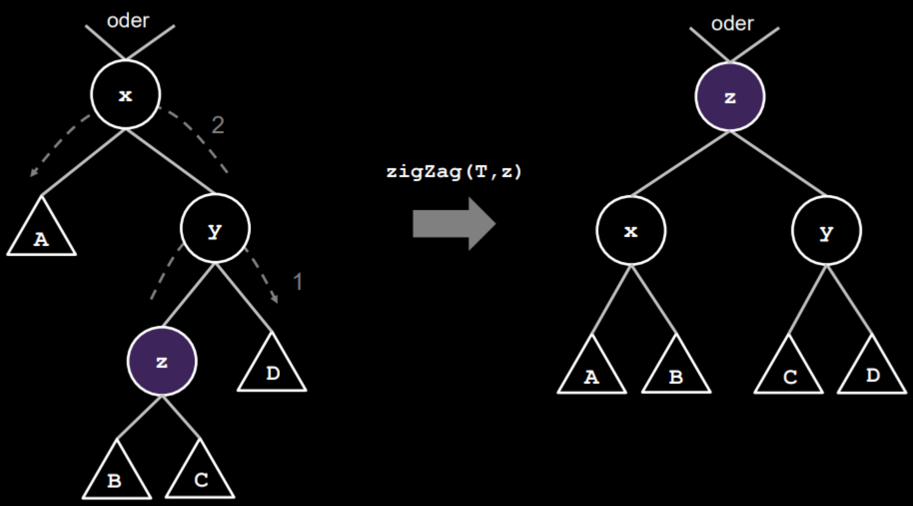
Splay-Operationen

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

splay(T,z)

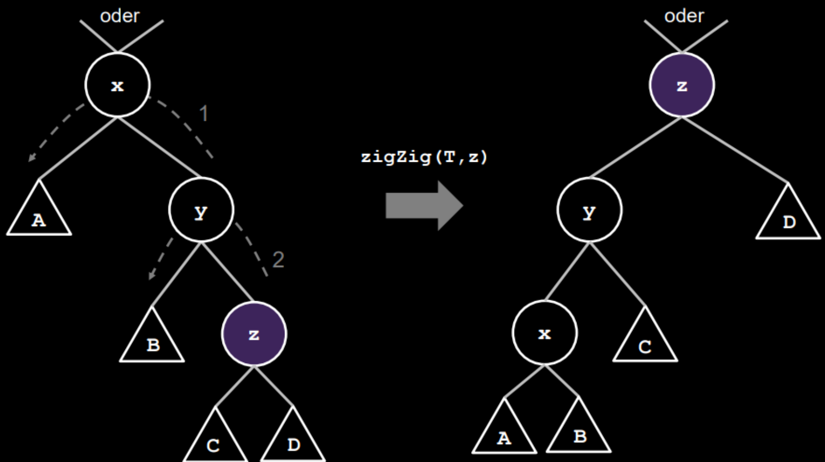
```
WHILE z != T.root DO
    IF z.parent.parent == nil THEN
        zig(T,z);
    ELSE
        IF z == z.parent.parent.left.left OR
           z == z.parent.parent.right.right THEN
           zigZig(T,z);
        ELSE
            zigZag(T,z);
```

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



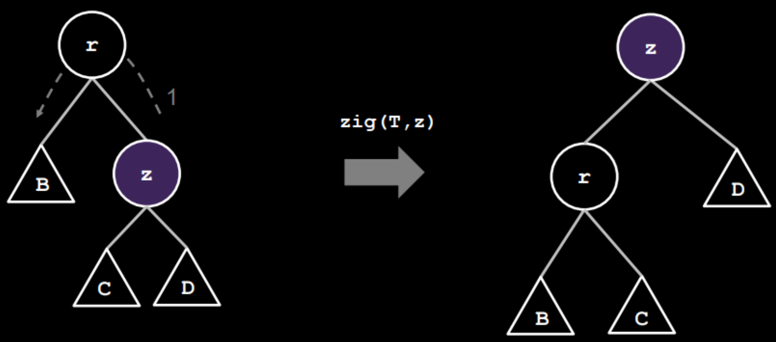
Zig-Zig-Operation

=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

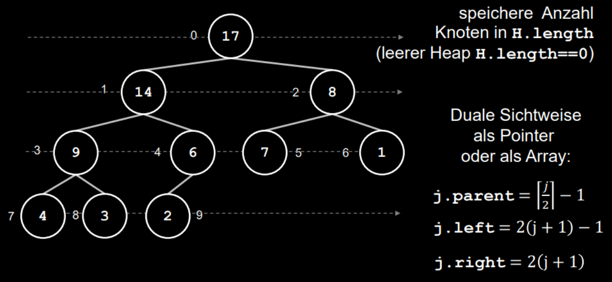
- Amortisierte Laufzeit: Laufzeit pro Operation über mehrere Operationen hinweg
- Worst-Case-Laufzeit pro Operation: $O(log_n\ n)$

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.parent.key` $\geq$ `x.key`
  - Maximum des Heaps steht damit in der Wurzel
- $h \leq log\ n$, da Baum fast vollständig

Heaps durch Arrays



0	1	2	3	4	5	6	7	8	9
17	14	8	9	6	7	1	4	3	2

Einfügen

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

Code:

insert(H,k) // als unbeschränktes Array

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

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

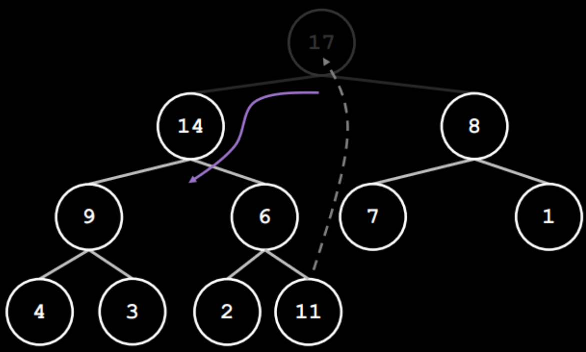


Abbildung 4: Beispiel Löschen des Maximums im Binären Max-Heap

heapify(H, i)

```
maxind = i;
IF i.left < H.length AND H.A[i]<H.A[i.left] THEN
    maxind = i.left;
IF i.right < H.length AND H.A[maxind]<H.A[i.right] THEN
    maxind = i.right;
IF maxind != i THEN
    SWAP(H.A, i, maxind);
    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

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

Heap-Sort

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

heapSort(H.A)

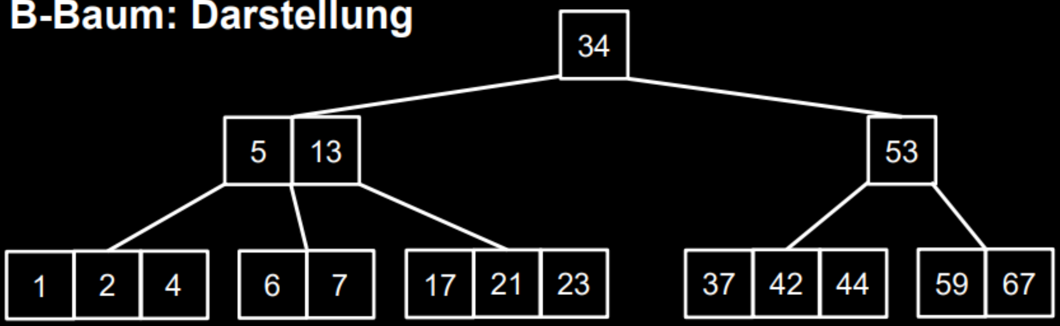
```
buildHeap(H.A)                  // Bauen des Heaps
WHILE !isEmpty(H) DO
    PRINT extract-max(H);       // Ausgabe des Maximums bis Heap leer ist
```

1.5 B-Bäume

Definition — B-Baum

- Jeder B-Baum hat einen angebenen Grad also z.B. $t = 2$
- Eigenschaften:
  - Wurzel zwischen $[1, \cdots, 2t-1]$ Werte
  - Knoten zwischen $[t-1, \cdots, 2t-1]$ 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: $k_0 \leq key[0] \leq k_1 \leq key[1] \leq ... \leq k_{n-1} \leq key[n-1] \leq k_n$

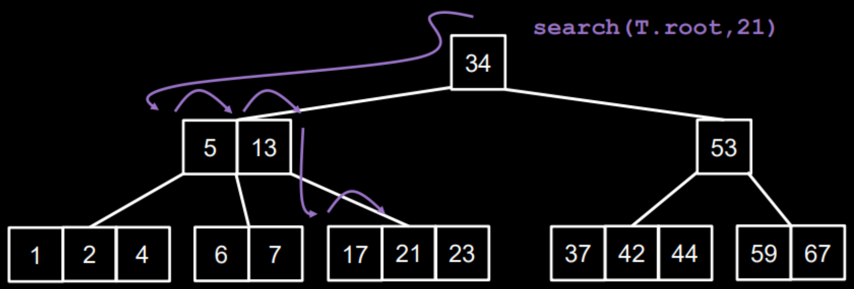
B-Baum: Darstellung



`x.n` – Anzahl Werte eines Knoten `x`
`x.key[0],…,x.key[x.n-1]` – (geordnete) Werte in Knoten `x`
`x.child[0],…,x.child[x.n]` – Zeiger auf Kinder in Knoten `x`

- 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)

```
WHILE x != nil DO
    i = 0;
    WHILE i < x.n AND x.key[i] < k DO
        i++;
    IF i < x.n AND x.key[i] == k THEN
        return(x, i);
    ELSE
        x = x.child[i];
return nil;
```

Einfügen

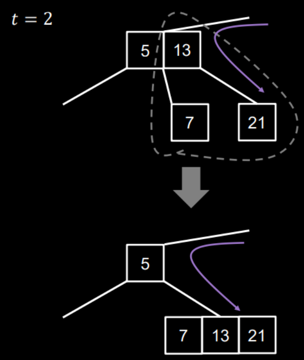
- Einfügen erfolgt immer in einem Blatt
- Falls das Blatt voll ist, muss jedoch gesplittet werden
- ⇒ Beim Durchlaufen des Baumes an jeder notwendigen vollen Position splitten
- Splitten:
  - Bricht volle Node auf und fügt mittleren Wert zur Elternnode hinzu
  - Aus den anderen Werten entstehen nun jeweils 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)

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

Löschen

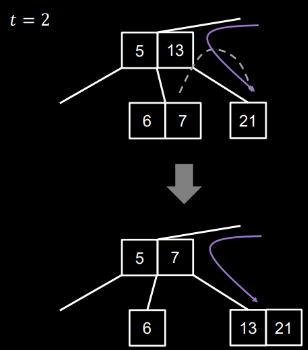
- Wenn Blatt noch mehr als $t-1$ Werte, kann der Wert einfach entfernt werden
- Allerdings durchlaufen wir hier den Baum auch wieder von oben und stellen gewisse Voraussetzungen her
- Durchlaufen des Baumes von oben und Anwendung der folgenden Algorithmen



Allgemeines Verschmelzen:

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

  $\Rightarrow$ keine Änderung oberhalb notwendig



Allgemeines Rotieren/Verschieben:

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

Code:

delete(T, k)

```
Wenn Wurzel nur 1 Wert und beide Kinder t-1 Werte,
verschmelze Wurzel und Kinder (reduziert Höhe um 1)
Suche rekursiv Löschposition:
    Wenn zu besuchendes Kind nur t-1 Werte,
    verschmelze es oder rotiere/verschiebe
Entferne Wert k im inneren Knoten/Blatt
// 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