

## 4. Advanced Data Structures

### Rot-Schwarz-Bäume (RS-Bäume)

#### Anwendung: Linux Completely Fair Scheduling

Verwendet RS-Bäume, um Worst-Case-Laufzeit ( $\log n$ ) zu erreichen

`key`: virtual run time eines Prozesses in Sekunden

1. Nächsten Prozess holen
2. Addiere zugewiesene Zeit
3. Füge mit aktualisierter Zeit wieder ein

#### Baumkunde

Ein RS-Baum ist ein binärer Suchbaum, sodass gilt:

1. Jeder Knoten ist rot oder schwarz (`x.color=red/black`)
2. Die Wurzel ist schwarz, wenn der Baum nicht leer ist
3. Wenn ein Knoten rot ist, sind seine Kinder schwarz (Nicht-Rot-Rot-Regel)
4. Für jeden Knoten hat jeder Pfad im Teilbaum zu einem Blatt/Halbblatt die gleiche Anzahl an schwarzen Knoten

#### Halbblätter

Halbblätter sind Knoten mit nur einem Kind

Halbblätter im RS-Baum sind schwarz, sonst wird direkt mindestens eine Regel verletzt

#### Schwarzhöhe eines Knoten

Die Schwarzhöhe eines Knoten `x` ist die eindeutige Anzahl an schwarzen Knoten auf dem Weg zu einem Blatt/Halbblatt im Teilbaum des Knoten

Für leeren Baum setzt man  $SH(nil) = 0$

#### Höhe eines RS-Baums

Ein RS-Baum mit  $n$  Knoten hat maximale Höhe  $h \leq 2 \cdot \log_2(n + 1)$

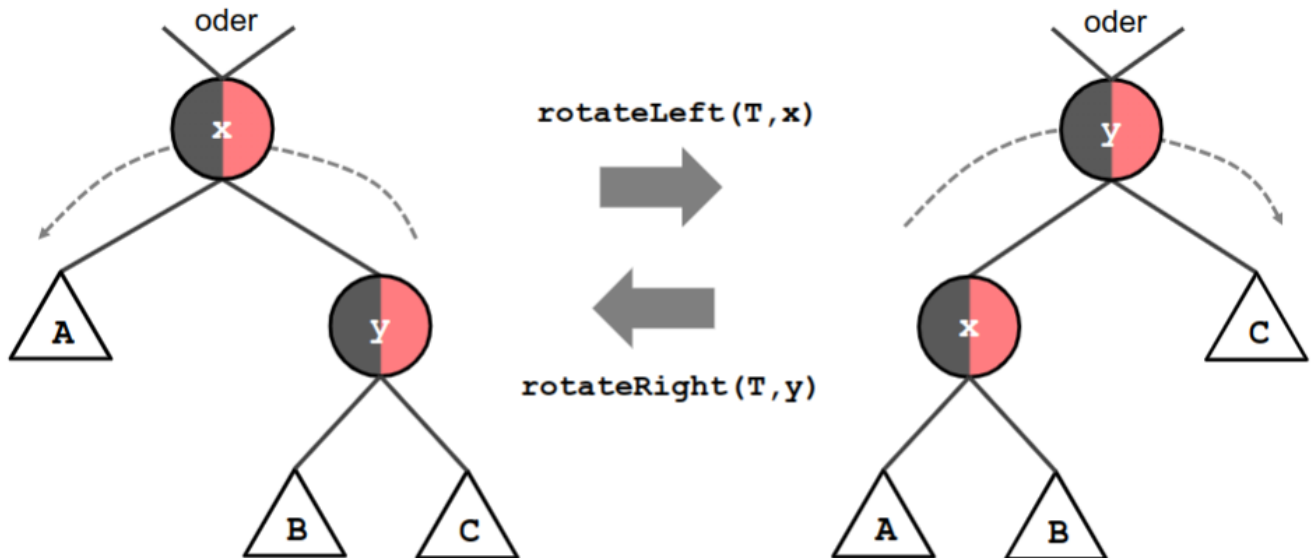
Intuition:

1. In jedem Unterteilbaum gleiche Anzahl schwarzer Knoten auf jedem Pfad
2. Maximal zusätzlich gleiche Anzahl roter Knoten auf diesem Pfad
3. Daher einigermaßen ausbalanciert und Höhe  $O(\log n)$

#### Implementierungen mittels Sentinel

- `T.root.parent = T.sent`
- `T.sent.key=nil; T.sent.color=black;`
- `T.sent.parent = T.sent, T.sent.left=T.sent; T.sent.right=T.sent;`
- Alles immer wohldefiniert dank Einführung von `T.sent` (Sentinel des Baumes T)

## Rotation



```

rotateLeft(T,x) // x.right!=nil
    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;

```

- Rot-Schwarz-Baum-Bedingungen sind nach Rotation eventuell verletzt
- Laufzeit =  $\Theta(1)$

## Einfügen

1. Finde Elternknoten `y` wie im BST
2. Färbe neuen Knoten `z` rot
3. Stelle RS-Baum-Bedingung wieder her

```

insert(T,z) // z.left==z.right==nil;
    x=T.root; px=T.sent;
    WHILE x != nil DO
        px=x;
        IF x.key > z.key THEN
            x=x.left
        ELSE
            x=x.right;
    z.parent=px;
    IF px==T.sent THEN
        T.root=z
    ELSE
        IF px.key > z.key THEN
            px.left=z
        ELSE
            px.right=z;
    z.color=red;
    fixColorsAfterInsertion(T,z);

```

- Funktioniert wie beim BST mit Sentinel

## Aufräumen

```

fixColorsAfterInsertion(T,z)
    WHILE z.parent.color==red DO
        IF z.parent==z.parent.parent.left THEN
            y=z.parent.parent.right;
            IF y!=nil AND y.color==red THEN
                z.parent.color=black;
                y.color=black;
                z.parent.parent.color=red;
                z=z.parent.parent;
            ELSE
                IF z==z.parent.right THEN
                    z=z.parent;
                    rotateLeft(T,z);
                z.parent.color=black;
                z.parent.parent.color=red;
                rotateRight(T,z.parent.parent);
            ELSE
                // do the same, but exchange left and right
        T.root.color=black;

```

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

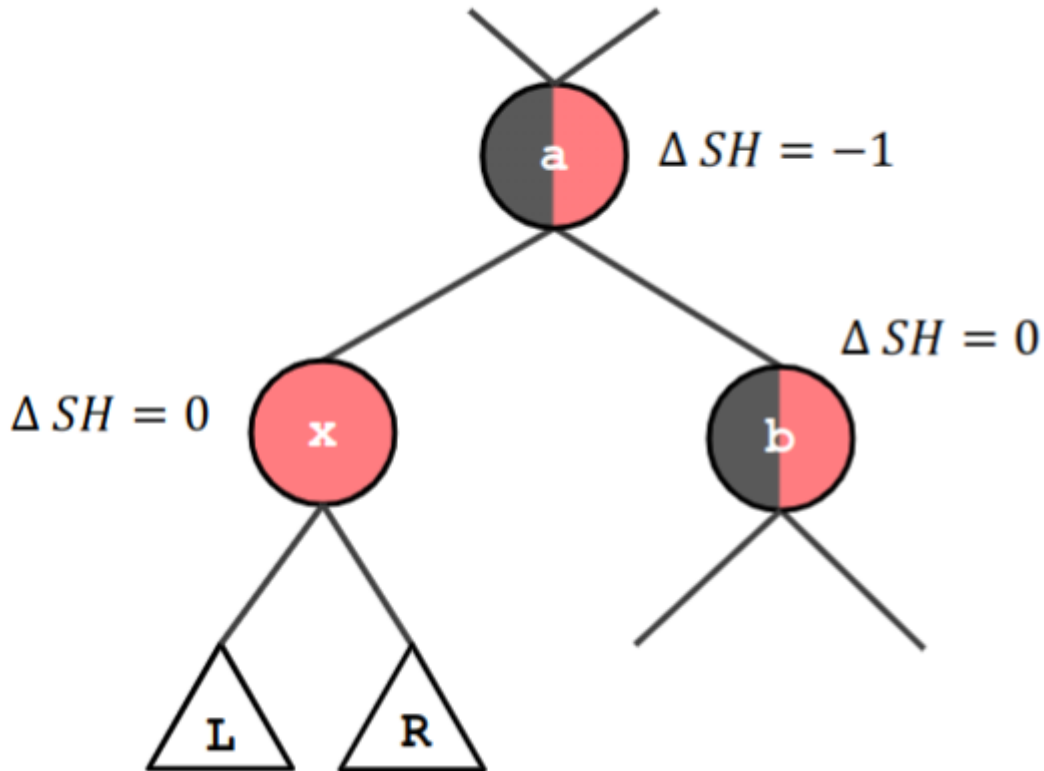
# Löschen

- Größtenteils analog zum BST
- Sei  $z$  der entfernte Knoten und  $y$  der Knoten, der  $z$  ersetzt.  
Dann erbt  $y$  die Farbe von  $z$ , wenn  $y$  schwarz war, müssen Farben angepasst werden

## Fixup bei $y.\text{color} == \text{black}$

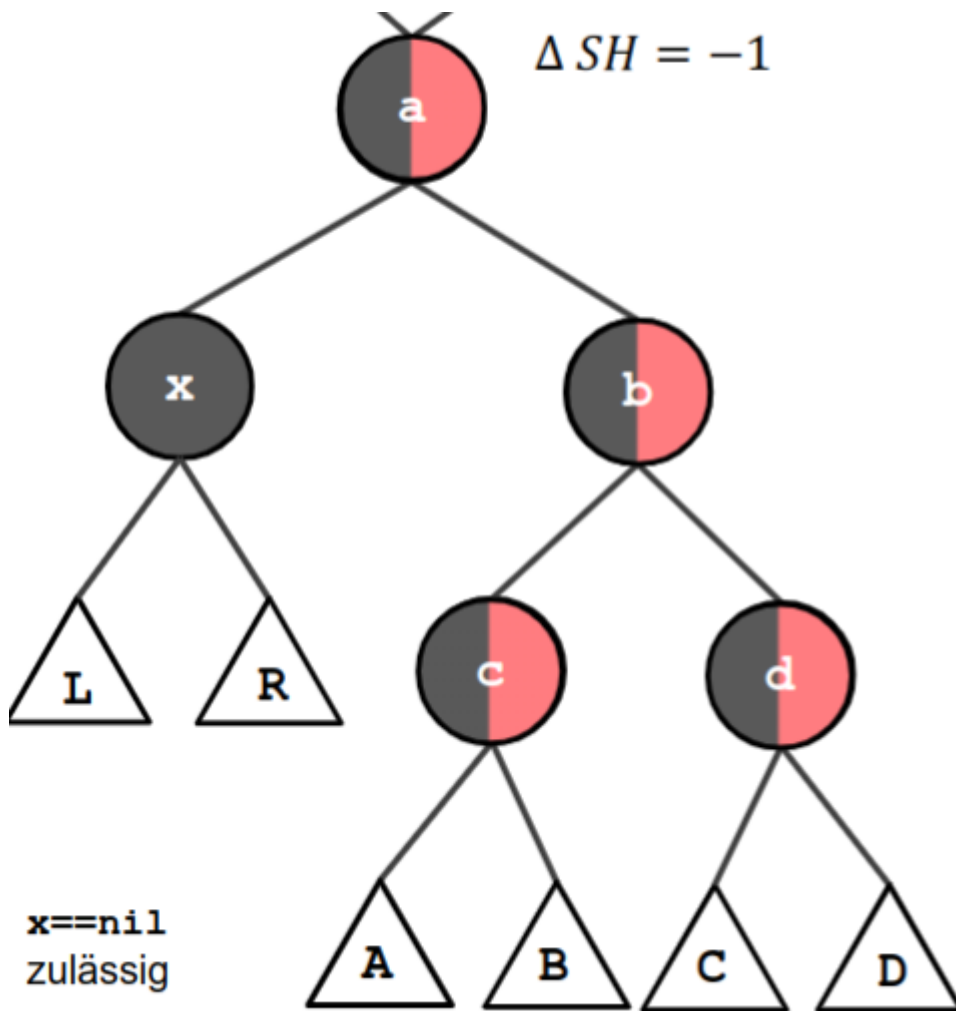
$\Delta SH := SH(\text{linker Teilbaum}) - SH(\text{rechter Teilbaum})$  f.a. Knoten

Beim Löschen kann die Schwarzhöhe nur sinken



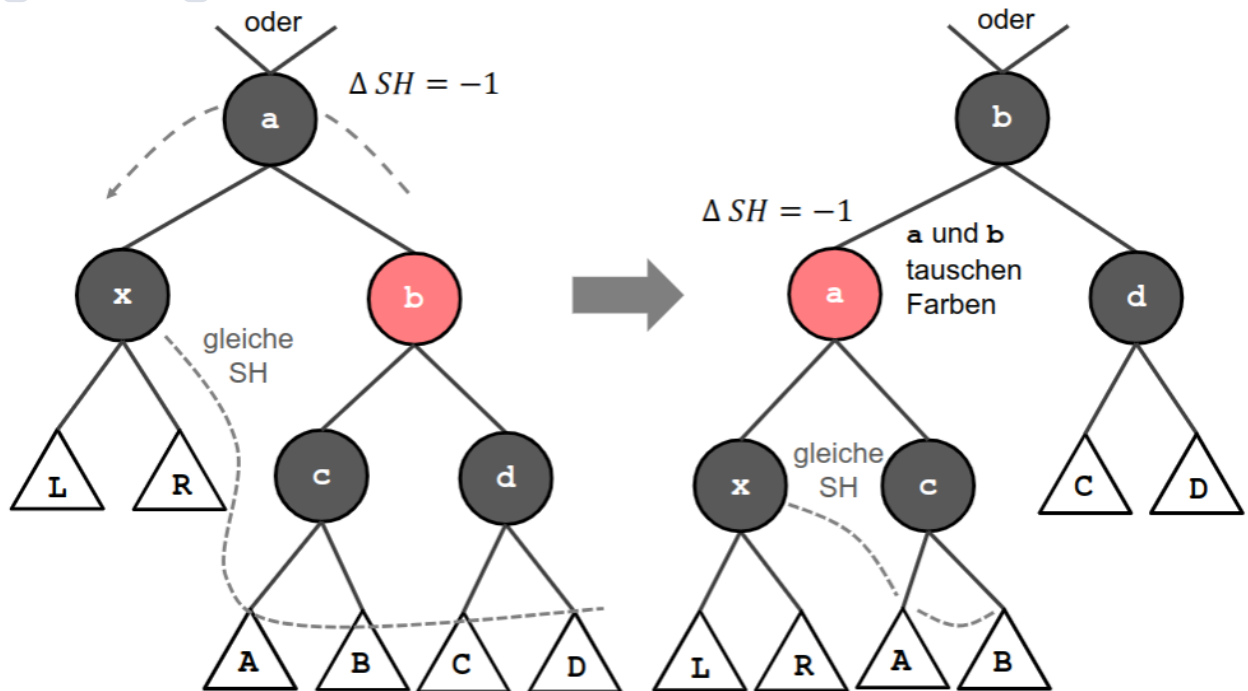
Fall  $\Delta SH = 1$  in  $a$  ist analog

Wenn  $x$  rot ist, setze  $x$  schwarz und fertig, somit nur schwarzer Fall zu betrachten



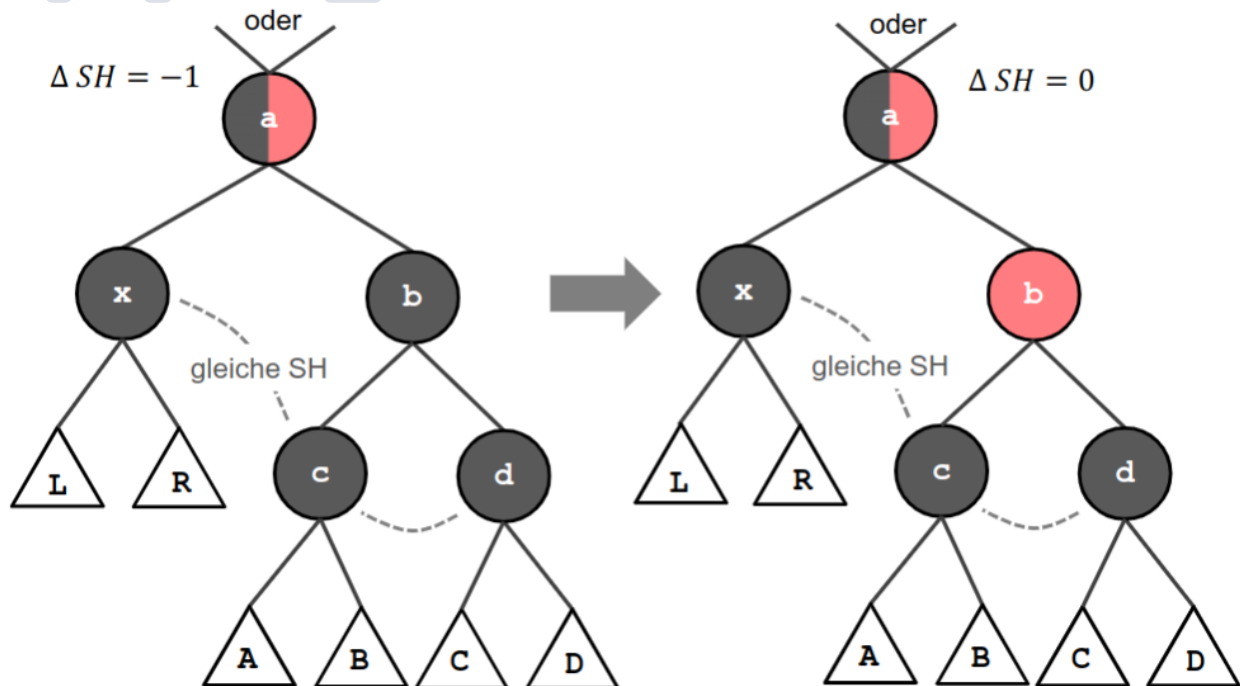
Fallunterscheidung:

1. **a** schwarz, **b** rot



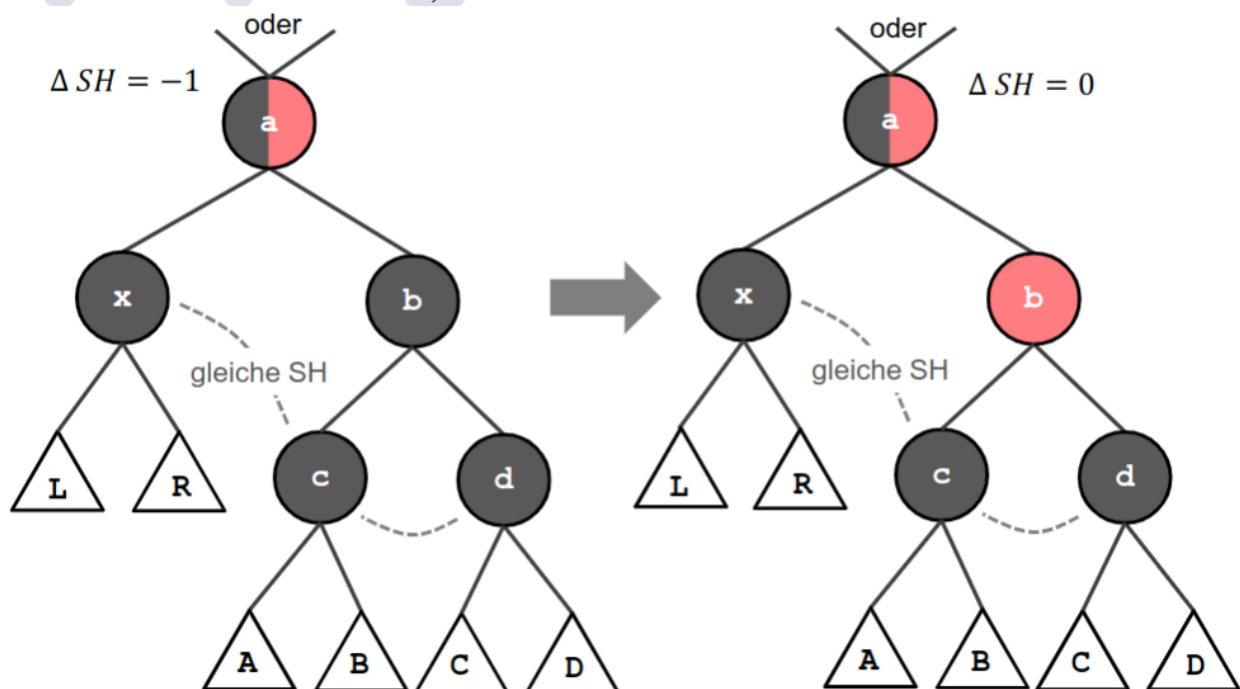
Wird zu allen Fällen außer 2b)

2. a) a rot, b schwarz, c,d nicht rot



b auf schwarz setzen, um ursprüngliche SH zu erreichen

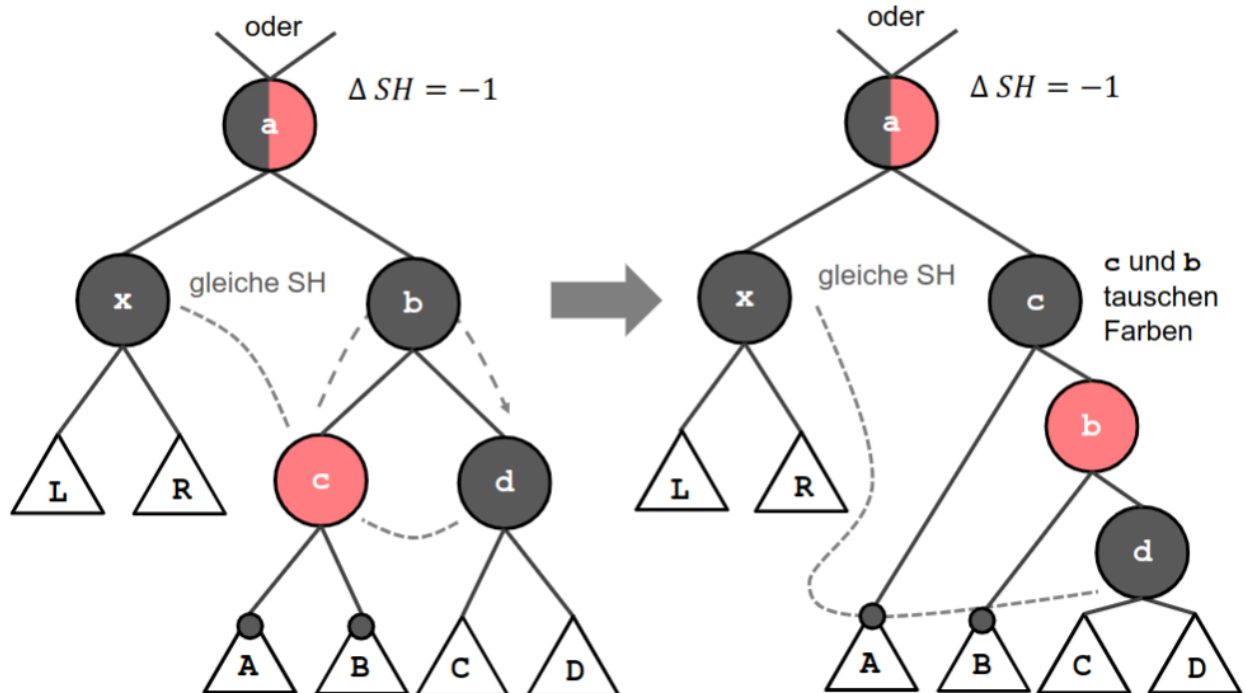
3. b) a schwarz, b schwarz, c,d nicht rot



Wenn a schwarz ist, dann gilt für Elternknoten  $\Delta SH = \pm 1$ .

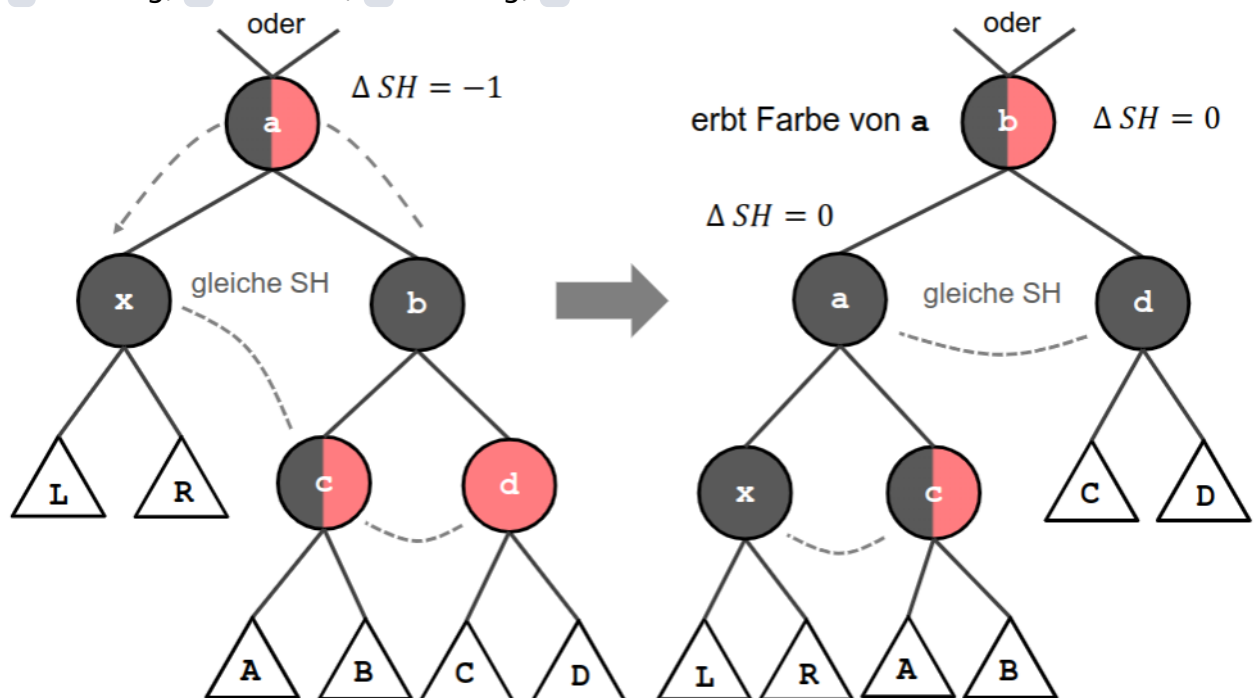
Verfahre also rekursiv mit a als neuem x

4. **a** beliebig, **b** schwarz, **c** rot, **d** nicht rot



Wird zu Fall 4

5. **a** beliebig, **b** schwarz, **c** beliebig, **d** rot



Ursprüngliche SH von vor der Entfernung wieder hergestellt

## Algorithmus

```
transplant(T,u,v) // with sentinel, also works for v==nil
  IF u.parent==T.sent 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)
    a=z.parent; dsh=nil;
    IF z.left==z.right==nil THEN // z leaf
        IF z.color==black AND z!=T.root THEN
            IF z.parent.left==z THEN dsh=right ELSE dsh=left;
            transplant(T,z,nil);
        ELSE IF z.left==nil THEN // z half leaf
            y=z.right;
            transplant(T,z,z.right);
            y.color=z.color;
        ELSE IF z.right==nil THEN // z half leaf
            y=z.left;
            transplant(T,z,z.left);
            y.color=z.color;
        ELSE // z has two children
            y=z.right; a=y; wentleft=false;
            WHILE y.left != nil DO
                a=y; y=y.left; wentleft=true;
            IF y.parent != z THEN
                transplant(T,y,y.right);
                y.right=z.right;
                y.right.parent=y;
            transplant(T,z,y);
            y.left=z.left;
            y.left.parent=y;
            IF y.color==black THEN
                IF wentleft THEN dsh=right ELSE dsh=left;
            y.color=z.color;
        IF dsh!=nil THEN fixColorsAfterDeletion(T,a,dsh);

```

- `a` ist ein Zeiger auf den Knoten, in dem die tiefste Imbalance entstehen könnte
- `dsh` =  $\Delta SH$  für Knoten `a`: `nil` = 0, `left` = 1, `right` = -1
- In den Fällen, in denen `z` ein Halbblatt ist, muss `y.color==red` sein, da sonst die SH-Regel verletzt wäre.  
Dann kann man einfach `y` umhängen und die Farbe von `z` kopieren
- In den Fällen, in denen `z` kein (Halb-)Blatt ist, muss eine Fallunterscheidung stattfinden, je nachdem, ob `y` rechtes oder linkes Kind ist, davon ist dann auch die eventuelle Imbalance abhängig

```

fixColorsAfterDeletion(T,a,dsh)
    IF dsh==right THEN // extra black node on the right

```



```

x=a.left; b=a.right; c=b.left; d=b.right;
IF x!=nil AND x.color==red THEN // x is red, easy to solve
    x.color=black;
ELSE IF a.color==black AND b.color==red THEN // case 1
    rotateLeft(T,a);
    a.color=red; b.color=black;
    fixColorsAfterDeletion(T,a,dsh);
ELSE IF a.color==red AND b.color==black // case 2a
    AND (c==nil OR c.color=black)
    AND (d==nil OR d.color=black) THEN
    a.color=black; b.color=red;
ELSE IF a.color==black AND b.color==black // case 2b
    AND (c==nil OR c.color==black)
    AND (d==nil OR d.color==black) THEN
    b.color=red;
    IF a==a.parent.left THEN dsh=left
    ELSE IF a==a.parent.right THEN dsh=right ELSE dsh=nil;
    fixColorsAfterDeletion(T,a.parent,dsh);
ELSE IF b.color==black AND c!=nil AND c.color==red // case 3
    AND (d==nil OR d.color==black) THEN
    rotateRight(T,b);
    c.color=black; b.color=black;
    fixColorsAfterDeletion(T,a,dsh);
ELSE IF b.color==black AND d!=nil AND d.color==red THEN // case
4
    rotateLeft(T,a);
    b.color=a.color; a.color=black; d.color=black;
ELSE // dsh==left, extra black node on the left
    // do the same, but exchange left and right

```

- `dsh=right` impliziert `b!=nil`
- Der letzte `ELSE`-branch ist für den linkslastigen Fall
- Außer in Fall 2b) führen rekursive Aufrufe im nächsten Schritt zum Rekursionsende

## Laufzeiten

y suchen hat wie beim BST Laufzeit  $O(h) = O(\log n)$

Falls Rekursion in Fixup eintritt, ist die Laufzeit konstant

↪ Gesamtlaufzeit Löschen =  $O(h) = O(\log n)$

## Worst-Case-Laufzeiten RSB

- Einfügen:  $\Theta(\log n)$
- Löschen:  $\Theta(\log n)$
- Suchen:  $\Theta(\log n)$

# AVL-Bäume

Optimierte Konstanten:

- RS-Bäume:  $h \leq 2 \cdot \log n$
- AVL-Bäume:  $h \leq 1.441 \cdot \log n$

Balance in Knoten  $x$  mit angehängtem rechtem und linkem Teilbaum:

$$B(x) = \text{height}(\text{rechter Teilbaum}) - \text{height}(\text{linker Teilbaum})$$

Konvention:  $\text{height}(\text{leerer Baum}) = -1$

Ein AVL-Baum ist ein binärer Suchbaum, sodass für die Balance  $B(x)$  in jedem Knoten  $x$  gilt:  $B(x) \in \{-1, 0, +1\}$

## Höhe

Ein AVL-Baum mit  $n$  Knoten hat die maximale Höhe  $h \leq 1.441 \cdot \log_2 n$

## AVL-Baum vs. RS-Baum

- AVL-Baum:  
Differenz von rechtem und linkem Teilbaum desselben Knotens  $\leq 1$   
Einfügen und Löschen verletzen in der Regel öfter die Baum-Bedingung, mehr Aufwand zum Rebalancieren
- RS-Baum:  
Höhenfaktor von rechtem und linkem Teilbaum desselben Knotens  $\leq 2$   
Suchen dauert eventuell länger  
AVL-Bäume geeigneter, wenn mehr Such-Operationen und weniger Einfüge- und Löschoperationen

## AVL $\subset$ RS, AVL $\neq$ RS

Jeder nicht-leere AVL-Baum der Höhe  $h$  lässt sich als RS-Baum mit Schwarzhöhe  $\lceil \frac{h+1}{2} \rceil$  darstellen.

Für gerade  $h$  gibt es sogar einen Baum mit roter Wurzel, Schwarzhöhe  $\frac{h}{2}$ , der alle anderen RS-Baumbedingungen erfüllt.

Für jede Höhe  $h \geq 3$  gibt es einen RS-Baum, der kein AVL-Baum ist.

## Einfügen

Funktioniert wie beim BST mit Sentinel, zuzüglich eventuellem Rebalancieren

```
insert(T,z) // z.left==z.right==nil;
    x=T.root; px=T.sent;
    WHILE x != nil DO
        px=x;
        IF x.key > z.key THEN
            x=x.left
        ELSE
```

```

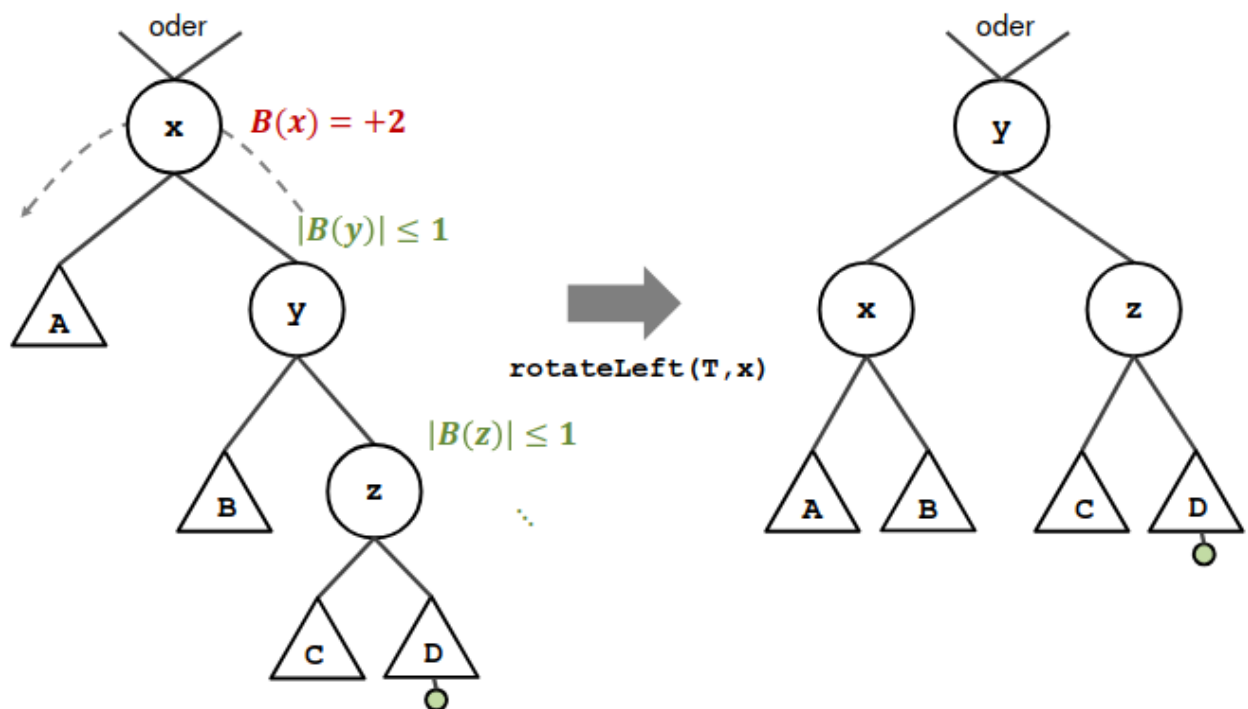
        x=x.right;
    z.parent=px;
    IF px==T.sent THEN
        T.root=z
    ELSE
        IF px.key > z.key THEN
            px.left=z
        ELSE
            px.right=z;
    fixBalanceAfterInsertion(T,z);

```

## Rebalacieren

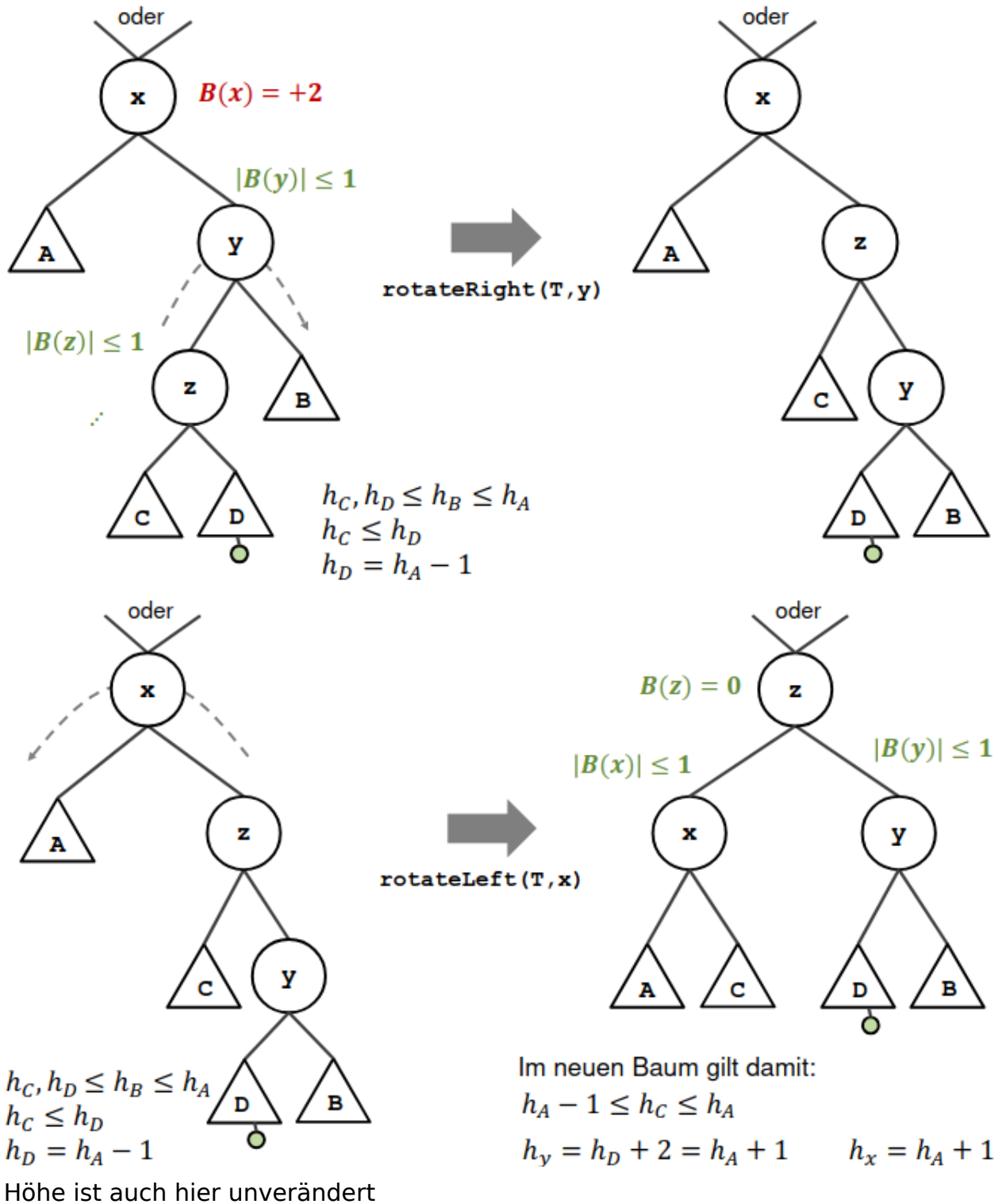
- Da immer in Blättern eingefügt wird, werden nur die Knoten direkt darüber eventuell nicht mehr balanciert
- Das Einfügen muss nicht immer dazu führen, dass die Balance nicht mehr vorhanden ist
- Unterscheide 4 Fälle:

### 1. Einfache Rotation:

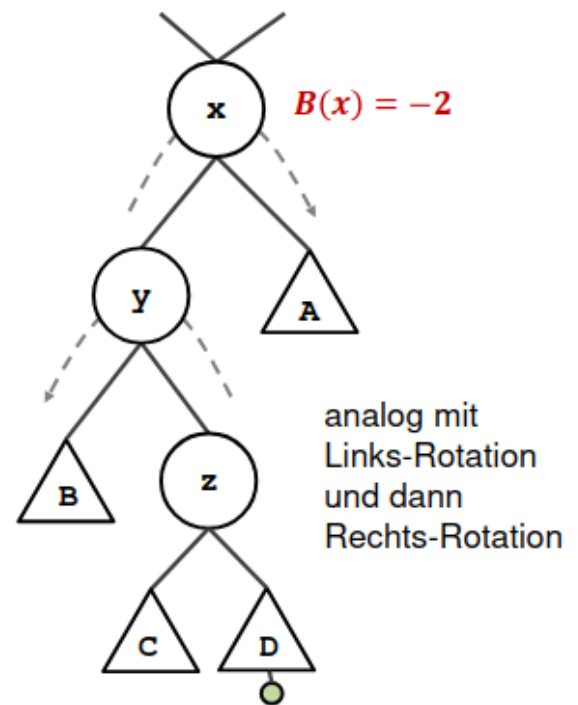
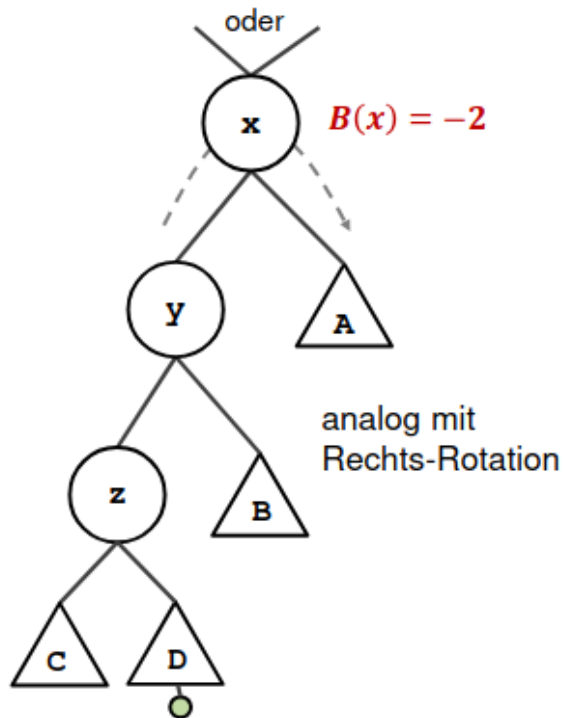


Nach dem Einfügen und Rotieren ist die Höhe gleich der Höhe vor dem Einfügen

2. Hier finden zwei Rotationen statt:



3. Fall 3 und 4 sind spiegelverkehrt und analog zu den Fällen 1 und 2



## Laufzeit

Gesamtlaufzeit  $O(h) = O(\log n)$

Suche hat Laufzeit  $O(h)$ , Rebalancieren ist nur einmal nötig, also ist das konstant

## Löschen

Analog zum BST, aber Rebalancierung eventuell bis in die Wurzel nötig

Gesamtlaufzeit  $O(h) = O(\log n)$

## Worst-Case-Laufzeiten

- Einfügen:  $\Theta(\log n)$
- Löschen:  $\Theta(\log n)$
- Suchen:  $\Theta(\log n)$

AVL-Bäume haben bessere theoretische Konstanten als Rot-Schwarz-Bäume, sind je nach Daten und Operationen aber in der Praxis nur unwesentlich schneller.

## Splay-Bäume

Selbst-organisierende Datenstrukturen

## Selbst-Organisierende Listen

Ansatz: einmal angefragte Werte werden voraussichtlich noch öfter angefragt

Variante für Bäume: Splay trees

## Anwendung: SQUID

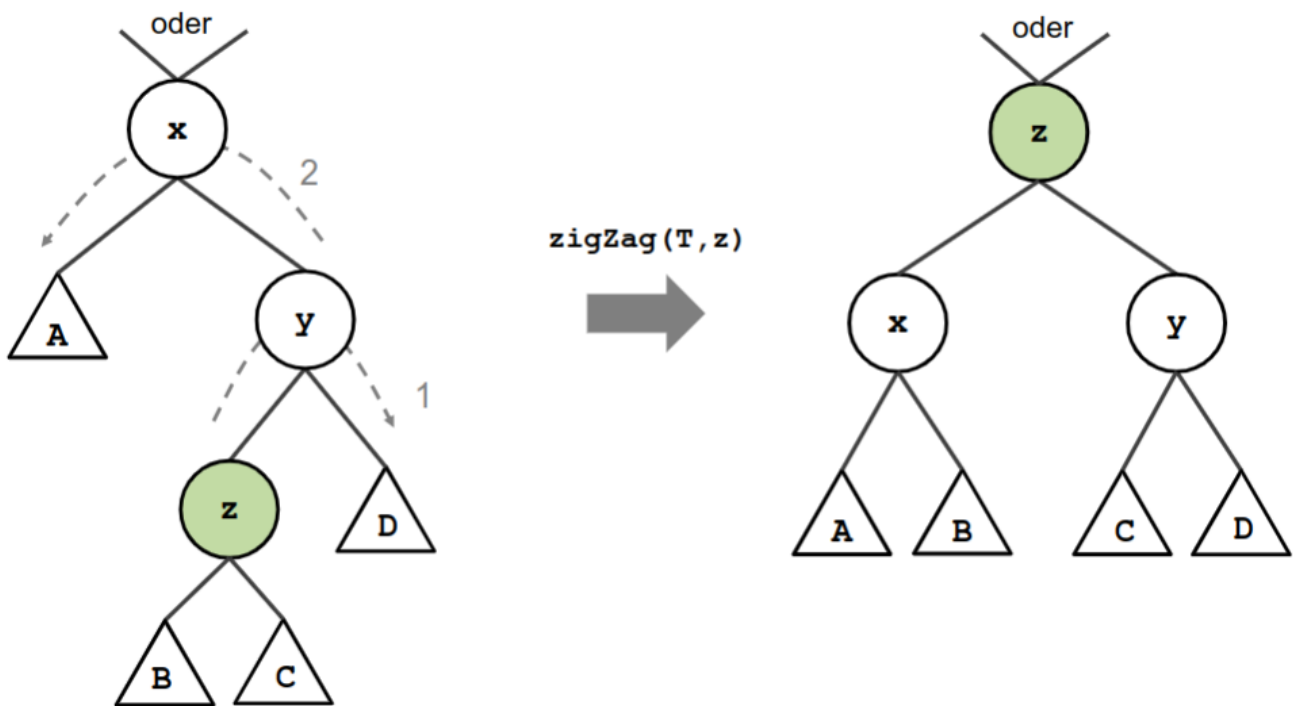
- Web-Cache-Proxy
- Speichert Access Control Listen (ACL) für http-Zugriffe als Splay-Tree

## Splay-Operationen

- Splay-Bäume bilden Untermenge der BST
- Spüle gesuchten oder neu eingefügten Knoten an die Wurzel
- $\text{splay}(T, z)$  = Folge von Zig-, Zig-Zig und Zig-Zag-Operationen

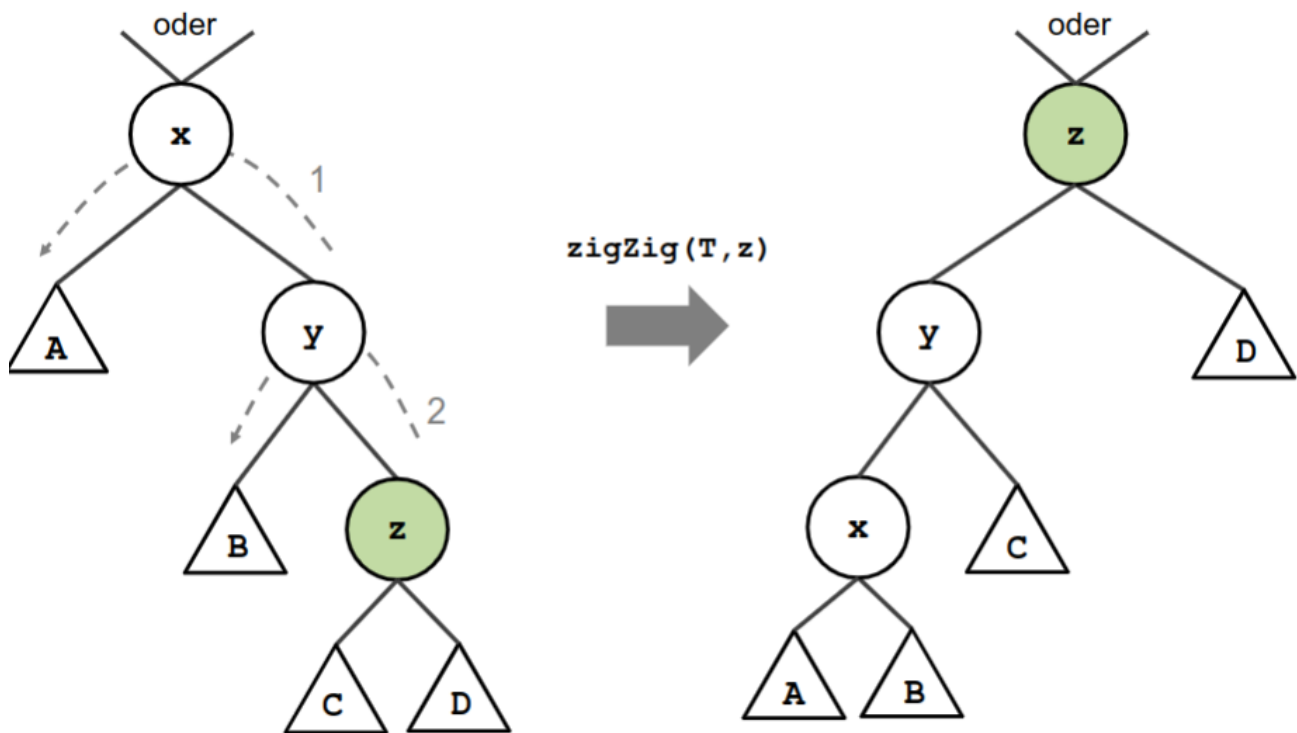
## Zig-Zag-Operation

Rechts-Links- oder Links-Rechts-Rotation



## Zig-Zig-Operation

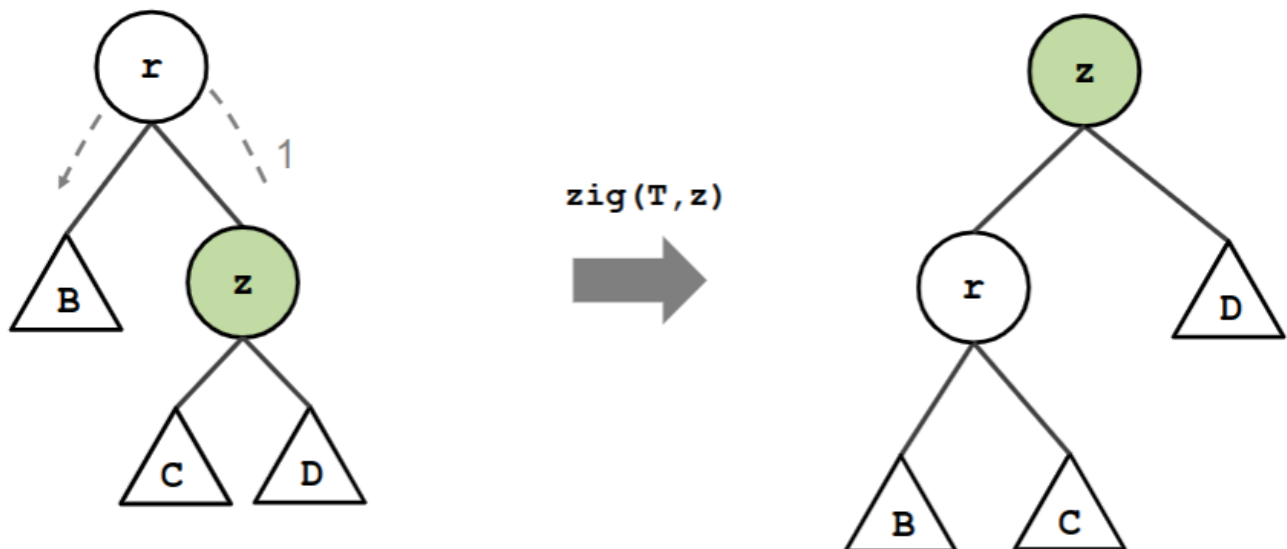
## Links-Links- oder Rechts-Rechts-Rotation



## Zig-Operation

Einfache Links- oder Rechts-Rotation

Wird verwendet, falls  $z$  direkt unter Wurzel hängt



## Splay-Operation

```
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);

zigZig(T,z)
    IF z==z.parent.left THEN
        rotateRight(T,z.parent.parent);
        rotateRight(T,z.parent);
    ELSE
        rotateLeft(T,z.parent.parent);
        rotateLeft(T,z.parent);

zigZag(T,z) // disclaimer: Not official, I wrote this!
    IF z==z.parent.left THEN
        rotateRight(T,z.parent);
        rotateLeft(T,z.parent);
    ELSE
        rotateLeft(T,z.parent);
        rotateRight(T,z.parent);

zig(T,z) // disclaimer: Not official, I wrote this!
    IF z==z.parent.left THEN
        rotateRight(T,z.parent);
    ELSE
        rotateLeft(T,z.parent);

```

Gesamtlaufzeit  $O(h)$

## Suchen

```

search(T,k)
    x=T.root;
    WHILE x != nil AND x.key != k DO
        IF x.key < k THEN
            x=x.right
        ELSE
            x=x.left;
// now search is done
    IF x==nil THEN
        return nil
    ELSE
        splay(T,x);
        return T.root;

```



Suche und Splayen haben Laufzeit  $O(h) \rightsquigarrow$  Gesamtlaufzeit  $O(h)$

Alternative: Bei erfolgloser Suche letzten besuchten Knoten nach oben splayen

## Einfügen

1. Suche analog zum Einfügen bei BST Einfügepunkt
2. Spüle eingefügten Knoten  $x$  per Splay-Operation nach oben

## Laufzeit

1. Position im BST suchen:  $O(h)$
2.  $\text{splay}(T, x) : O(h)$   
 $\rightsquigarrow$  Gesamtlaufzeit  $O(h)$

## Löschen

1. Spüle gesuchten Knoten  $x$  per Splay-Operation nach oben
2. Lösche  $x$   
Wenn einer der beiden Teilbäume leer ist, fertig
3. Spüle den "größten" Knoten  $y$  im linken Teilbaum per Splay-Operation nach oben  
 $y$  kann kein rechtes Kind haben, da größter Wert im linken Teilbaum
4. Hänge rechten Teilbaum an  $y$  an

## Laufzeit

1.  $\text{splay}(T, x) : O(h)$
2.  $x$  löschen:  $O(1)$
3.  $y$  im linken Teilbaum  $L$  finden,  $\text{splay}(L, y) : O(h) + O(h) = O(h)$
4. Anhängen:  $O(1)$   
 $\rightsquigarrow$  Gesamtlaufzeit  $O(h)$

## Laufzeit Splay-Bäume

- Amortisierte Laufzeit:  
Laufzeit pro Operation über mehrere Operationen hinweg
- Für  $m \geq n$  Operationen auf einem Splay-Baum mit maximal  $n$  Knoten ist die Worst-Case-Laufzeit  $O(m \cdot \log_2 n)$ , also  $O(\log_2 n)$  pro Operation.
- Zusätzlich: Oft gesuchte Elemente werden sehr schnell gefunden

## (Binäre Max-)Heaps

Ein binärer Max-Heap ist ein binärer Baum, der

1. bis auf das unterste Level vollständig und im untersten Level von links gefüllt ist
2. Für alle Knoten  $x \neq T.\text{root}$  gilt:  $x.\text{parent.key} \geq x.\text{key}$

- Heaps sind keine BSTs, linke Kinder können größere Werte als rechte Kinder haben!
- Bei Min-Heaps sind die Werte in Elternknoten jeweils kleiner

## Eigenschaften

- Da Baum (fast) vollständig ist, gilt  $h \leq \log n$
- Maximum des Heaps steht in der Wurzel

## Heaps durch Arrays

- speichere Anzahl Knoten in `H.length` (leerer Heap `H.length==0`)
- Duale Sichtweise als Pointer oder als Array ( $j$  ist Index im Array):
  - $j.parent = \left\lceil \frac{j}{2} \right\rceil - 1$
  - $j.left = 2(j+1) - 1$
  - $j.right = 2(j+1)$

## Einfügen

- Position durch Baumstruktur vorgegeben
- Vertausche nach oben, bis Max-Eigenschaft wieder erfüllt

```
insert(H,k) // as (unlimited) 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. Stelle Max-Eigenschaften wieder her, indem Knoten nach unten gegen das Maximum der beiden Kinder getauscht wird (`heapify`)

```
extract-max(H) // as (unlimited) array
  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;

heapify(H,i) // as (unlimited) array
    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);

```

Laufzeit beider Algorithmen  $O(h) = O(\log n)$

## Heap-Konstruktion aus Array

Blätterindizes:  $\lceil \frac{n-1}{2} \rceil, \dots, n-1$

Blätter sind für sich triviale Max-Heaps

Baue rekursiv per `heapify` Max-Heaps für Teilbäume

```

buildHeap(H) // array A has already been copied to H.A
    H.length=A.length;
    FOR i = ceil((H.length-1)/2)-1 DOWNT0 0 DO
        heapify(H,i);

```

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

## Heap-Sort

Gibt Einträge in Array `A` in absteigender Größe aus

```

heapSort(H) // array A has already been copied to H.A
    buildHeap(H);
    WHILE !isEmpty(H) DO PRINT extract-max(H);

```

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

Alternativ: speichere in jeder `WHILE`-Iteration `max=extract-max(H)` in

`H.A[H.length]=max`, um sortierte Liste am Ende aufsteigend im Array `A` zu haben.

## Abstrakter Datentyp Priority Queue

- `new(Q)` : erzeugt neue, leere Priority Queue namens `Q`
- `isEmpty(Q)` : gibt an, ob Queue `Q` leer
- `max(Q)` : gibt "größtes" Element aus Queue `Q` zurück, Fehler wenn leer

- `extract-max(Q)` : gibt "größtes" Element aus `Q` zurück, löscht es aus `Q`, Fehler wenn leer
- `insert(Q,k)` : fügt Wert `k` zu Queue `Q` hinzu  
Implementation kann Priority Heap verwenden (Java)

## B-Bäume

Ein B-Baum von Grad  $t$  ist ein Baum, bei dem

1. jeder Knoten außer der Wurzel zwischen  $t - 1$  und  $2t - 1$  Werte `key[0], key[1], ...` hat, die Wurzel hat zwischen 1 und  $2t - 1$  Werte
2. die Werte innerhalb eines Knoten aufsteigend geordnet sind
3. die Blätter alle die gleiche Höhe haben
4. jeder innerer Knoten mit  $n$  Werten  $n + 1$  Kinder hat, sodass für alle Werte  $k_j$  aus dem  $j$ -ten Kind gilt:  $k_0 \leq key[0] \leq k_1 \leq key[1] \leq \dots \leq k_{n-1} \leq key[n-1] \leq k_n$

## Darstellung

- `x.n` : Anzahl Werte des Knotens `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

- Mindestens 1 Wert in Wurzel
- Mindestens 2 Knoten in Tiefe 1 mit jeweils mindestens  $t$  Kindern
- Mindestens  $2t$  Knoten in nächster Tiefe mit jeweils mindestens  $t$  Kindern
- Mindestens  $2t^2$  Knoten in nächster Tiefe mit jeweils mindestens  $t$  Kindern, usw.
- In jedem Knoten außer Wurzel mindestens  $t - 1$  Werte
- Anzahl Werte  $n$  im B-Baum im Vergleich zur Höhe  $h$ :  $n \geq 2t^h - 1$ , also  $\log_t \frac{n+1}{2} \geq h$
- $\rightsquigarrow$  Ein B-Baum vom Grad  $t$  mit  $n$  Werten hat maximale Höhe  $h \leq \log_t \frac{n+1}{2}$
- Für größere  $t$  also flacher als vollständiger Binärbaum

## Anwendung

- MySQL speichert Werte in B-Bäumen
- Lesen/Schreiben in Blöcken: mehrere Werte (z.B. Index-Einträge) auf einmal

## Suche

```
search(x,k)
    WHILE x != nil DO
        i=0;
        WHILE i < x.n AND x.key[i] < k DO i=i+1;
        IF i < x.n AND x.key[i]==k THEN
```

```

        return (x,i);
    ELSE
        x=x.child[i];
return nil;

```

\2. While-Schleife: Maximal  $2t \in O(1)$  Iterationen

Laufzeit  $O(t \cdot h) = O(\log_t n)$

## Baumkunde

- B-Baum vom Grad  $t$ : max.  $2t$ , min.  $t$  Kinder pro Knoten  $\neq$  Wurzel  
Alternative Definition: max.  $t$ , min.  $\frac{t}{2}$  Kinder pro Knoten  $\neq$  Wurzel
- 2-3-4-Baum/(2,4)-Baum: B-Baum mit  $t = 2$
- B+-Baum: alle Werte in Blättern, innerer Knoten enthalten Werte erneut  
Vorteil: innere Knoten speichern nur kurzen Schlüssel, nicht auch noch Daten(-zeiger)  
Nachteil: Findet Werte erst im Blatt  
Alternativer Name: B\*-Baum
- Alternative Bedeutung B\*-Baum: B-Baum mit Füllgrad min.  $\frac{2}{3}$  pro Knoten  $\neq$  Wurzel

## Einfügen

### Idee

- Einfügen erfolgt immer in einem Blatt
- Wenn Blatt weniger als  $2t - 1$  Werte hat, dann einfügen und fertig
- Wenn nicht:

### Splitten

- Wenn Blatt bereits  $2t - 1$  Werte, dann teile es in zwei Blätter mit je  $t - 1$  Werten, füge mittleren Wert im Elternknoten ein
- Wenn dadurch Elternknoten mehr als  $2t - 1$  Werte hat, rekursiv nach oben
- Splitten an der Wurzel: Neue Wurzel wird erzeugt, Höhe des Baumes wächst um 1  
B-Baum-Einfügen splittet beim Suchen und läuft nur einmal hinab, sonst werden teure Disk-Operationen zweimal ausgeführt, einmal beim ab-, einmal beim aufsteigen.

## Informeller Algorithmus

```

insert(T,z)
    Wenn Wurzel schon  $2t-1$  Werte, dann splitte Wurzel
    Suche rekursiv Einfügeposition:
        Wenn zu besuchendes Kind  $2t-1$  Werte, splitte es erst
    Füge z in Blatt ein

```

Laufzeit  $O(t \cdot h) = O(\log_t n)$

Schleifeninvariante:

Bei der Suche hat der aktuelle Knoten immer weniger als  $2t - 1$  Werte, da sonst vorher gesplitted.

Eventuelles Splitten ist also problemlos möglich.

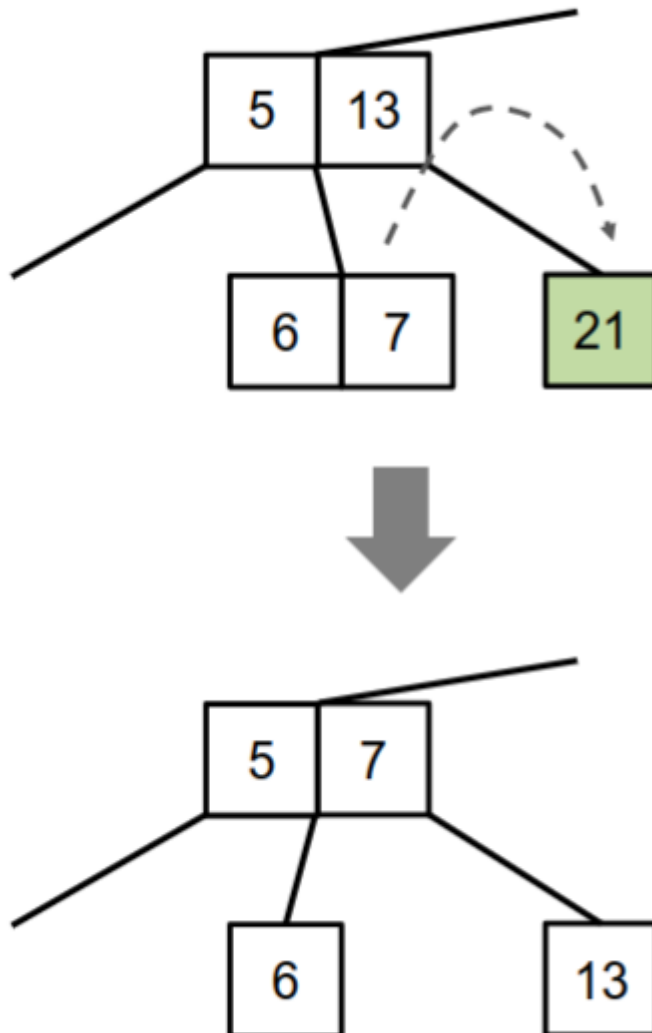
Auch das Blatt hat am Ende weniger als  $2t - 1$  Werte.

## Löschen

### Löschen im Blatt

- Wenn Blatt noch mehr als  $t - 1$  Werte hat, dann einfach entfernen
- Wenn  $t - 1$  Werte im Blatt mit zu löschendem Wert sind, linker oder rechter Geschwisterknoten hat mind.  $t$  Werte, dann rotiere Werte von Geschwisterknoten und Elternknoten:

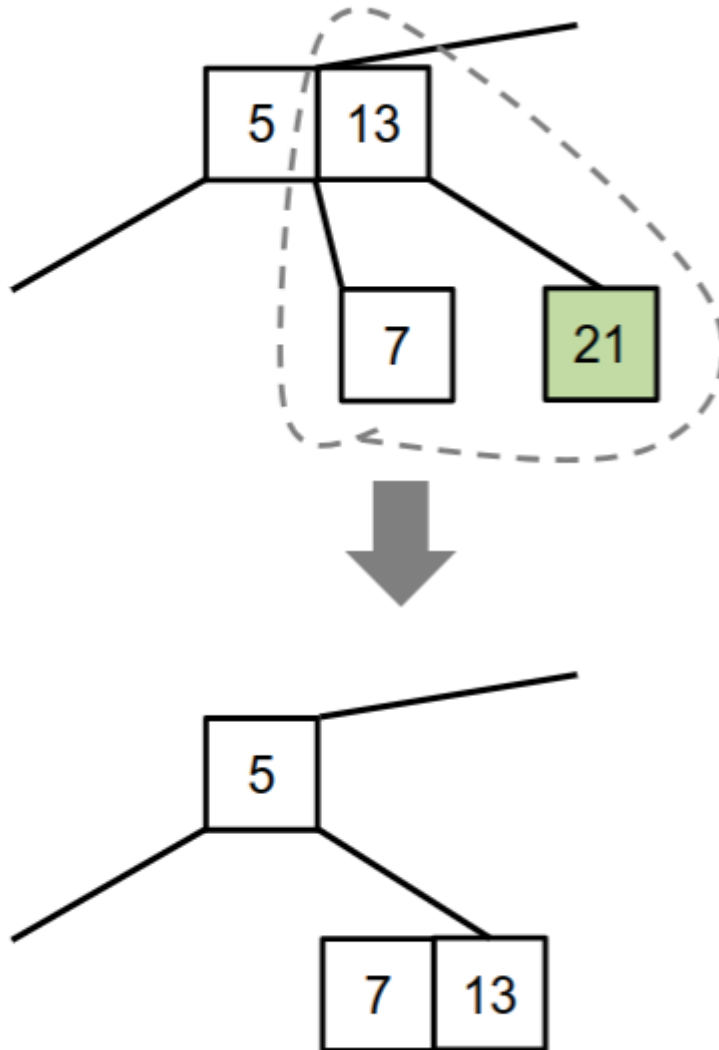
$t = 2$



- Wenn  $t - 1$  Werte im Blatt mit zu löschendem Wert sind, linker oder rechter Geschwisterknoten haben auch  $t - 1$  Werte, dann verschmelze einen Geschwisterknoten mit Wert aus Elternknoten, dieser hat nun eventuell zu wenig

Werte:

$$t = 2$$

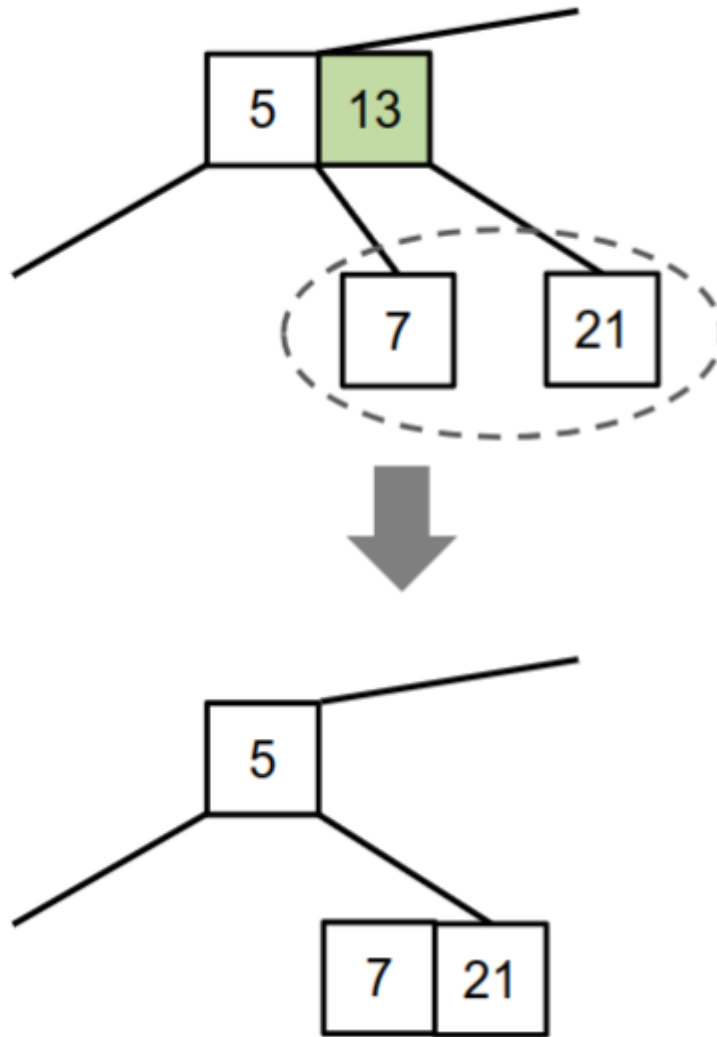


maximal  $t - 2 + t - 1 + 1 = 2t - 2$  Werte

## Löschen im inneren Knoten

- Verschieben:  
Wenn sich mehr als  $t - 1$  Werte in einem der beiden Kindknoten befinden, dann größten Wert (vom linken Kind) bzw. kleinsten Wert (vom rechten Kind) nach oben kopieren
- Verschmelzen:  
Wenn sich jeweils  $t - 1$  Werte in beiden Kindknoten befinden, dann Kindknoten verschmelzen, eventuell hat Elternknoten nun zu wenig Werte:

$t = 2$



B-Baum-Löschen läuft auch nur einmal hinab, stelle dazu sicher, dass zu besuchendes Kind mindestens  $t$  Werte hat.

## Allgemeines Verschmelzen ohne Löschen

Zu besuchendes Kind, rechter, linker Geschwisterknoten (sofern existent) haben nur  $t - 1$  Werte.

Dann ist Verschmelzen ohne weitere Änderungen möglich, wenn der Elternknoten vorher mindestens  $t$  Werte hat.

## Allgemeines Rotieren/Verschieben ohne Löschen

Zu besuchendes Kind hat nur  $t - 1$  Werte, aber ein Geschwisterknoten hat mehr als  $t - 1$  Werte, dann kann man dies ohne Änderungen oberhalb tun

## Informeller Algorithmus

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

```
    Wenn Wurzel nur 1 Wert und beide Kinder  $t-1$  Werte haben, verschmelze  
    Wurzel und Kinder (reduziert Höhe um 1)
```

```
    Suche rekursiv Löschposition:
```

```
        Wenn zu besuchendes Kind nur  $t-1$  Werte hat, verschmelze es oder
```



Laufzeit  $O(t \cdot h) = O(\log_t n)$

Schleifeninvariante:

Aktueller Knoten hat zu diesem Zeitpunkt mindestens  $t$  Werte, sonst wäre er vorher verschmolzen worden oder es wäre rotiert worden.

Beim Verschmelzen/Verschieben des Kindes kann die Anzahl der Werte im aktuellen Knoten nicht unter  $t - 1$  fallen.

Entfernen aus Blatt problemlos möglich, da mindestens  $t$  Werte vorhanden.

Entfernen im inneren Knoten durch Verschieben oder Verschmelzen.

## Worst-Case-Laufzeiten

- Einfügen:  $\Theta(\log_t n)$
- Löschen:  $\Theta(\log_t n)$
- Suchen:  $\Theta(\log_t n)$
- $O$ -Notation versteckt konstanten Faktor  $t$  für Suche innerhalb eines Knoten:  
 $t \cdot \log_t n = t \cdot \frac{\log_2 n}{\log_2 t}$  ist in der Regel größer als  $\log_2 n$ , also nur vorteilhaft, wenn Daten blockweise eingelesen werden.