

# Algorithmen und Datenstrukturen

Wintersemester 2022/23

14. Vorlesung

## Rot-Schwarz-Bäume

# Vorlesungsumfrage

Sehr geehrter Herr Prof. Dr. Wolff,  
die Rücklaufquote der Umfrage "Algorithmen und  
Datenstrukturen" liegt aktuell bei 18%  
(Teilnehmerzahl insgesamt: 356).

Dieser Wert liegt unter einer definierten Schwelle  
von 50%. Wir möchten Sie daher bitten, Ihre  
Teilnehmer nochmals zu ermuntern, sich an der  
Umfrage zu beteiligen.

Bitte beachten Sie, dass die Umfrage am **15.12.2022**  
**23:59:00** geschlossen wird.

Ihr EvaSys Administrator

**Bitte teilnehmen!**

# Dynamische Menge

verwaltet Elemente einer  
sich ändernden Menge  $M$



<b>Abstrakter Datentyp</b>	<i>Funktionalität</i>
ptr Insert(key $k$ , info $i$ ) Delete(ptr $x$ )	} Änderungen
ptr Search(key $k$ ) ptr Minimum() ptr Maximum() ptr Predecessor(ptr $x$ ) ptr Successor(ptr $x$ )	} Anfragen

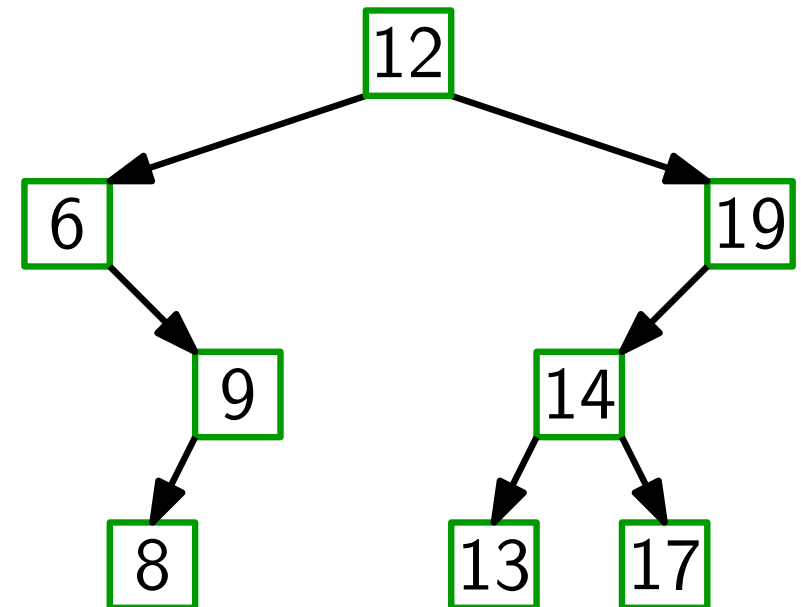
**Implementierung:** je nachdem...

# Binäre Suchbäume

**Satz.** Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in  $O(h)$  Zeit, wobei  $h$  die momentane Höhe des Baums ist.

**Aber:** Im schlechtesten Fall gilt  $h \in \Theta(n)$ .

**Ziel:** Suchbäume *balancieren!*  
 $\Rightarrow h \in O(\log n)$



*Binärer-Suchbaum-Eigenschaft:*

Für jeden Knoten  $v$  gilt:

alle Knoten im linken Teilbaum von  $v$  haben Schlüssel  $\leq v.key$   
 rechten  $\geq$

# Balanciermethoden

*Beispiele*

nach **Gewicht**

**BB[ $\alpha$ ]-Bäume**

für jeden Knoten ist das Gewicht  
(= Anzahl der Knoten) von linkem  
u. rechtem Teilbaum ungefähr gleich.

nach **Höhe**

**AVL-Bäume\***

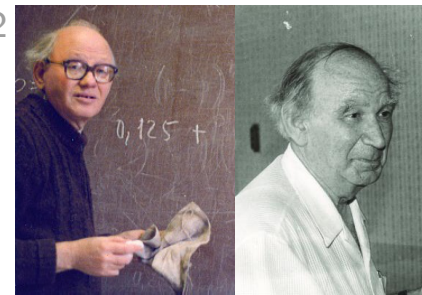
für jeden Knoten ist die Höhe  
von linkem und rechtem Teilbaum ungefähr gleich.

\*) Georgi M. Adelson-Velski & Jewgeni M. Landis, Doklady Akademii Nauk SSSR, 1962

nach **Grad**

**(2, 3)-Bäume**

alle Blätter haben dieselbe Tiefe, aber innere  
Knoten können verschieden viele Kinder haben.



1922–2014 1921–1997

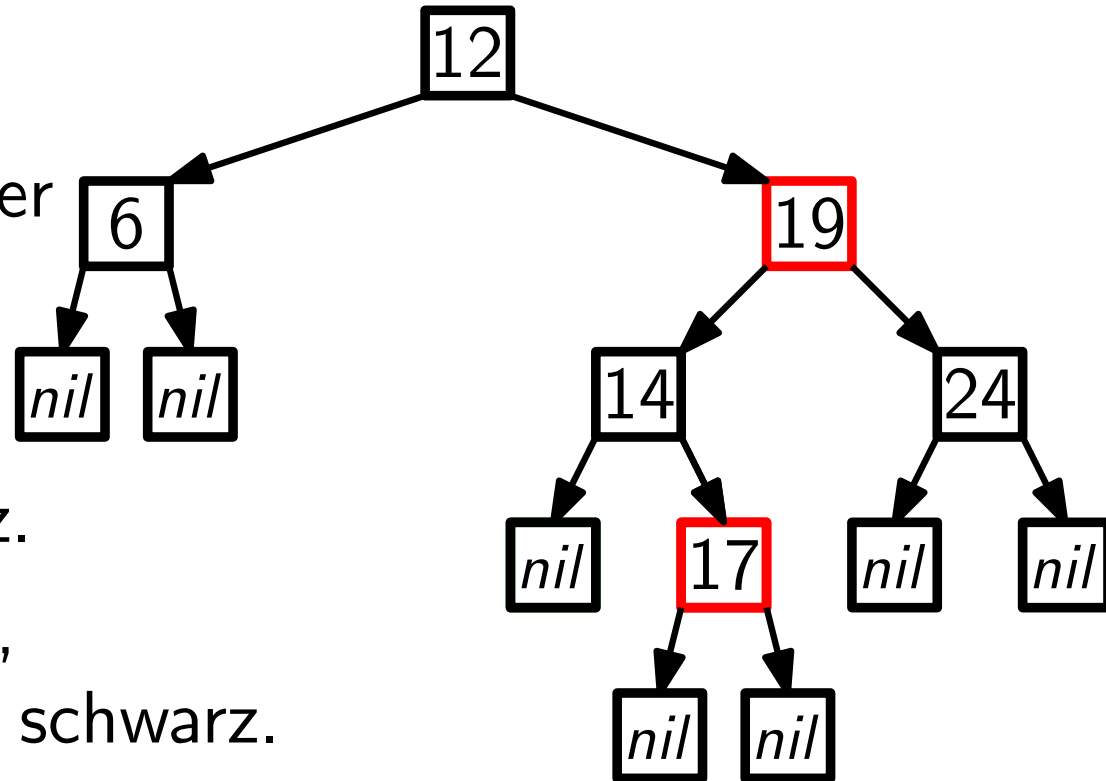
nach **Knotenfarbe** **Rot-Schwarz-Bäume**

jeder Knoten ist entw. „gut“ oder „schlecht“; der Anteil  
schlechter Knoten darf in keinem Teilbaum zu groß sein.

# Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden *Rot-Schwarz-Eigenschaften*:

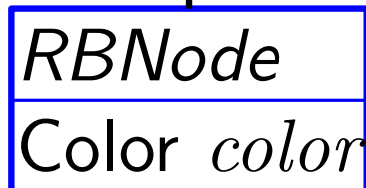
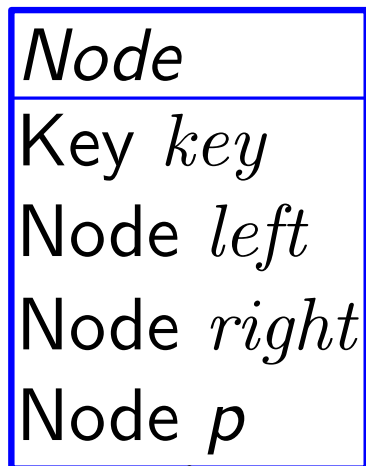
- (E1) Jeder Knoten ist entweder rot oder schwarz.
- (E2) Die Wurzel ist schwarz.
- (E3) Alle Blätter sind schwarz.
- (E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.



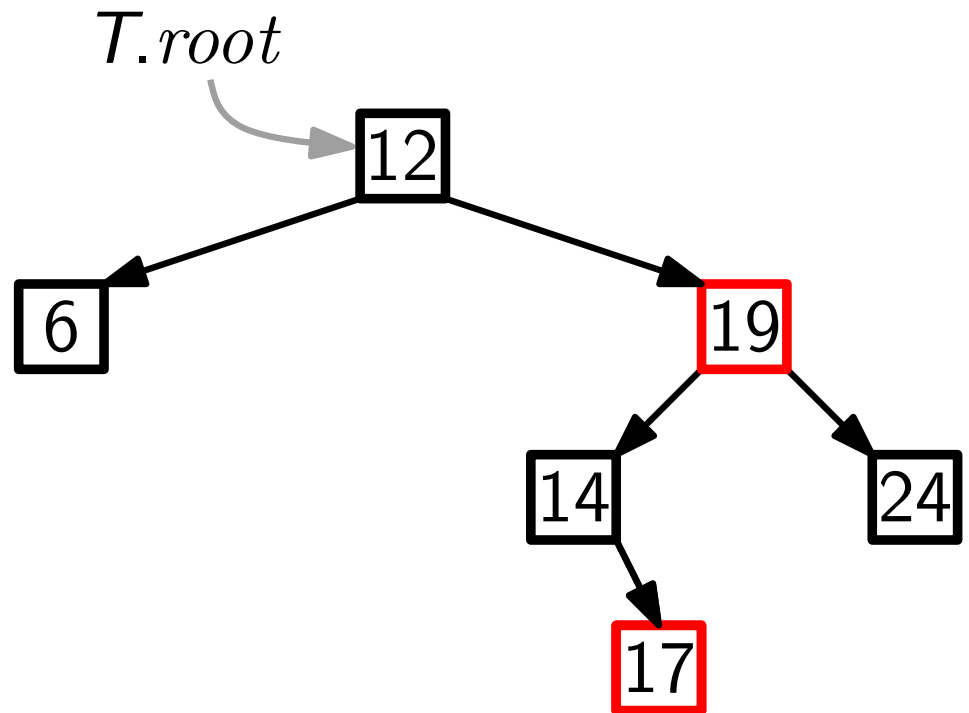
Aus (E4) folgt: Auf keinem Wurzel-Blatt-Pfad folgen zwei rote Knoten direkt auf einander.



# Technisches Detail



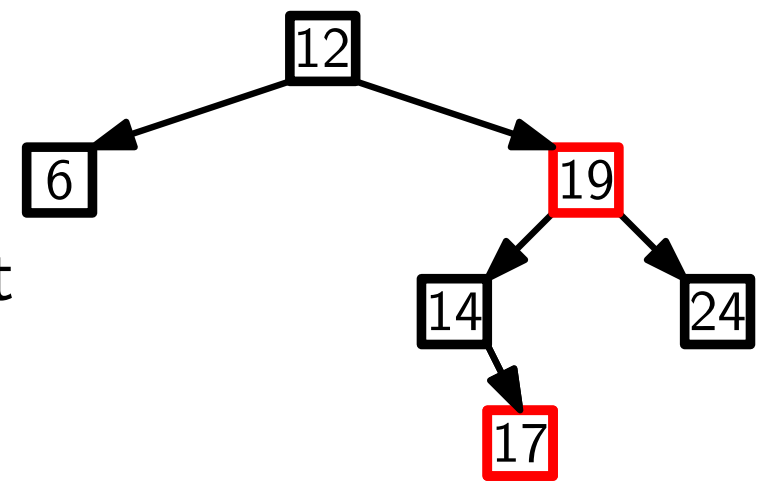
*T.root, T.nil*



Zweck: um Baum-Operationen prägnanter aufschreiben zu können. (Wir zeichnen den Dummy-Knoten i.A. nicht.)



# (Schwarz-) Höhe



**Definition:** Die *Länge* eines Pfades ist die Anz. seiner Kanten.

**Definition:** Sei  $B$  ein Baum.

Knoten  $u$  ist *unter* Knoten  $v$ , wenn  $u$  in dem Teilbaum  $B_v$  von  $B$  mit Wurzel  $v$  enthalten ist.

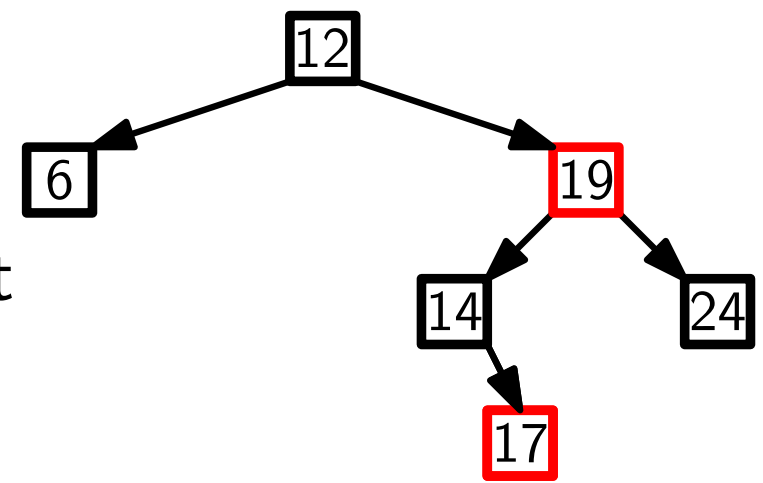
**Beispiel:** „17“ ist unter „19“, „14“ ist *nicht* unter „6“.

**Definition:** Die *Höhe* eines Knotens  $v$  ist die Länge eines längsten Pfades von  $v$  zu einem Blatt unter  $v$ .

**Definition!** Die *Höhe*  $\text{Höhe}(v)$  eines Knotens  $v$  ist die Anz. der Knoten (ohne  $v$ ) auf dem längsten Pfad zu einem Blatt (inkl. Blatt) in  $B_v$ .

**Beispiel:** „12“ hat Höhe

# (Schwarz-) Höhe



**Definition:** Die *Länge* eines Pfades ist die Anz. seiner Kanten.

**Definition:** Sei  $B$  ein Baum.

Knoten  $u$  ist *unter* Knoten  $v$ , wenn  $u$  in dem Teilbaum  $B_v$  von  $B$  mit Wurzel  $v$  enthalten ist.

**Beispiel:** „17“ ist unter „19“, „14“ ist *nicht* unter „6“.

**Definition:** Die *Höhe* eines Knotens  $v$  ist die Länge eines längsten Pfades von  $v$  zu einem Blatt unter  $v$ .

wohl-  
definiert  
wg. (E5)

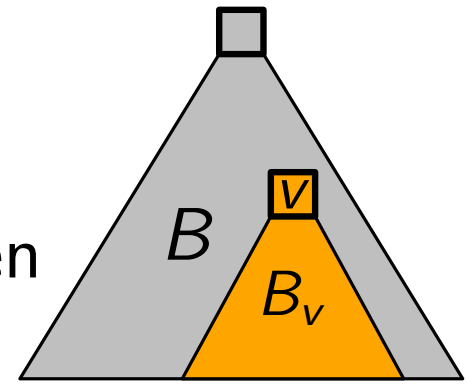
**Definition:** Die *Schwarz-Höhe*  $s\text{Höhe}(v)$  eines Knotens  $v$  ist die Anz. der *schwarzen* Knoten (ohne  $v$ ) auf *jedem* ~~längsten~~ Pfad zu einem Blatt (inkl. Blatt) in  $B_v$ .

**Beispiel:** „12“ hat Höhe 4 (!) und Schwarz-Höhe 2.

**Folgerung:**  $v$  Knoten  $\Rightarrow s\text{Höhe}(v) \leq \text{Höhe}(v) \leq 2 \cdot s\text{Höhe}(v)$ .

Höhe  $\in \Theta(\log n)!!$

**Lemma.** Ein Rot-Schwarz-Baum  $B$  mit  $n$  inneren Knoten hat Höhe  $\leq 2 \log_2(n + 1)$ .



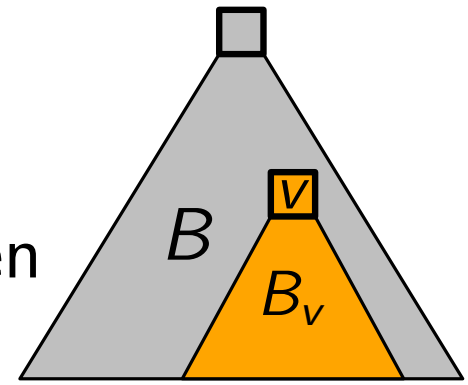
*Beweis.*

Behauptung: Für jeden Knoten  $v$  von  $B$  gilt:

$B_v$  hat  $\geq 2^{\text{Höhe}(v)} - 1$  innere Knoten.

Höhe  $\in \Theta(\log n)!!$

**Lemma.** Ein Rot-Schwarz-Baum  $B$  mit  $n$  inneren Knoten hat Höhe  $\leq 2 \log_2(n + 1)$ .



**Beweis.**

Behauptung: Für jeden Knoten  $v$  von  $B$  gilt:

$B_v$  hat  $\geq 2^{\text{sHöhe}(v)} - 1$  innere Knoten.

Beweis durch vollständige Induktion über  $\text{Höhe}(v)$ .

$\text{Höhe}(v) = 0$ . Dann  $B_v = B.\text{nil}$  und  $\text{sHöhe}(v) = 0$ .

$B_v$  hat  $2^0 - 1 = 0$  innere Knoten. ✓

$\text{Höhe}(v) > 0$ . Beide Kinder von  $v$  haben  $\text{Höhe} < \text{Höhe}(v)$ .

$\Rightarrow$  können Ind.-Annahme anwenden.

$\Rightarrow$  Anz. innere Knoten von  $B_v$  ist mind.

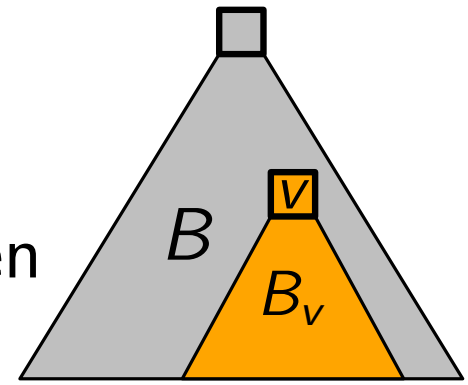
$2 \cdot (2^{\text{sHöhe}(v)-1} - 1) + 1 = 2^{\text{sHöhe}(v)} - 1$ . ✓

sHöhe der Kinder von  $v$  ist mind.

Anz. innerer Knoten unter einem Kind von  $v$

Höhe  $\in \Theta(\log n)!!$

**Lemma.** Ein Rot-Schwarz-Baum  $B$  mit  $n$  inneren Knoten hat Höhe  $\leq 2 \log_2(n + 1)$ .



*Beweis.*

Behauptung: Für jeden Knoten  $v$  von  $B$  gilt:

$B_v$  hat  $\geq 2^{\text{sHöhe}(v)} - 1$  innere Knoten.

$v := B.\text{root} \Rightarrow \underbrace{\# \text{ innere Knoten}(B)}_n \geq 2^{\text{sHöhe}(B)} - 1.$

$\Rightarrow \text{sHöhe}(B) \leq \log_2(n + 1)$

Wegen R-S-Eig. (E4) gilt: Höhe( $B$ )  $\leq 2 \cdot \text{sHöhe}(B)$ .

$\Rightarrow \text{Höhe}(B) \leq 2 \log_2(n + 1)$  □

**Also:**

Rot-Schwarz-Bäume sind *balanciert!* Fertig?!

Nee: **Insert & Delete können R-S-Eig. verletzen!**

# Einfügen

Node Insert(key  $k$ )

$y = nil$

$x = root$

**while**  $x \neq nil$  **do**

$y = x$

**if**  $k < x.key$  **then**

$x = x.left$

**else**  $x = x.right$

$z = \text{new Node}(k, y)$

**if**  $y == nil$  **then**  $root = z$

**else**

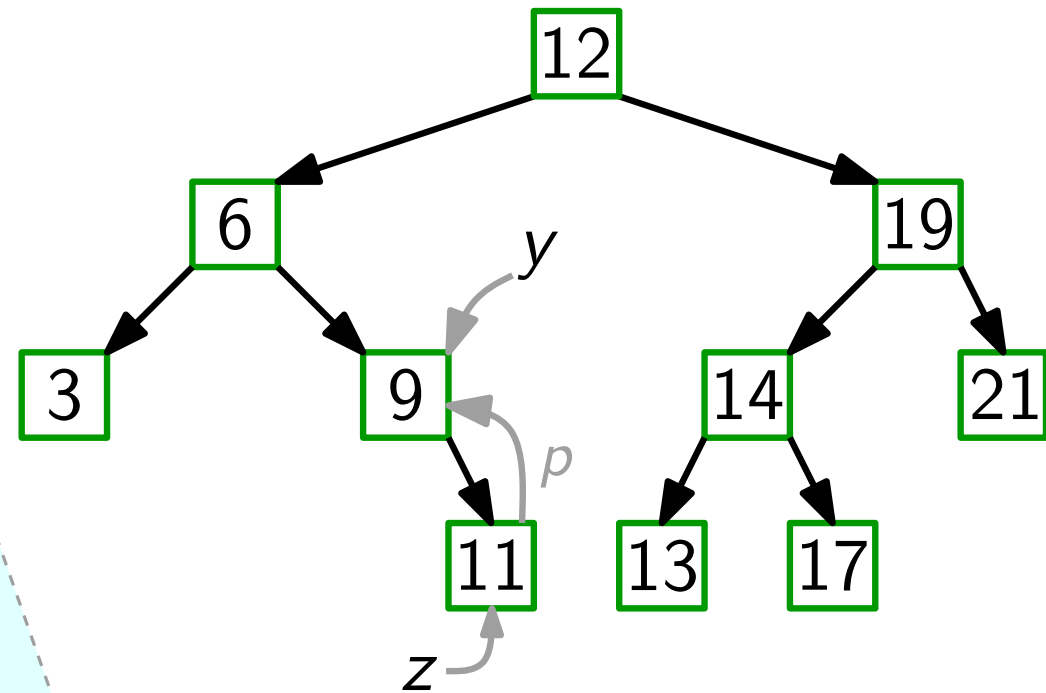
**if**  $k < y.key$  **then**  $y.left = z$

**else**  $y.right = z$

**return**  $z$

Insert(11)

$x == nil$



Node(Key  $k$ , Node  $par$ )

$key = k$

$p = par$

$right = left = nil$

## Einfügen

**Laufzeit?** (ohne RBInsertFixup)  $O(h) = O(\log n)$

RB RB

Node Insert(key  $k$ )

$y = T.nil$

$x = root$

**while**  $x \neq T.nil$  **do**

$y = x$

**if**  $k < x.key$  **then**

$x = x.left$

**else**  $x = x.right$

$z = \text{new Node}(k, y, \text{red})$

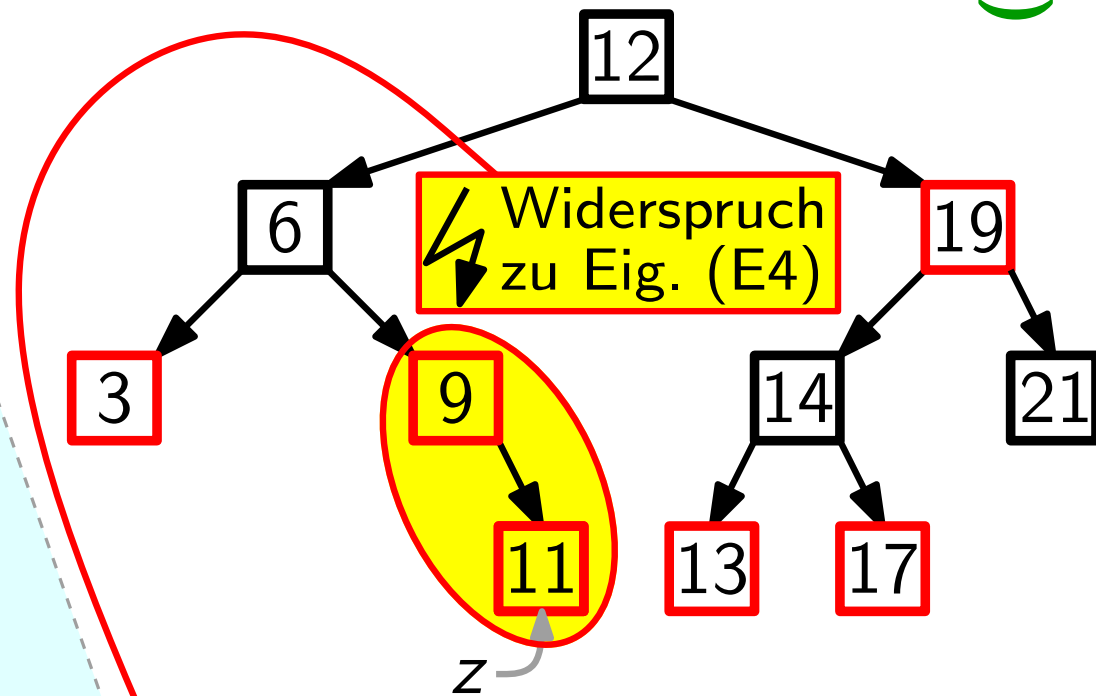
**if**  $y == T.nil$  **then**  $root = z$

**else**

**if**  $k < y.key$  **then**  $y.left = z$

**else**  $y.right = z$

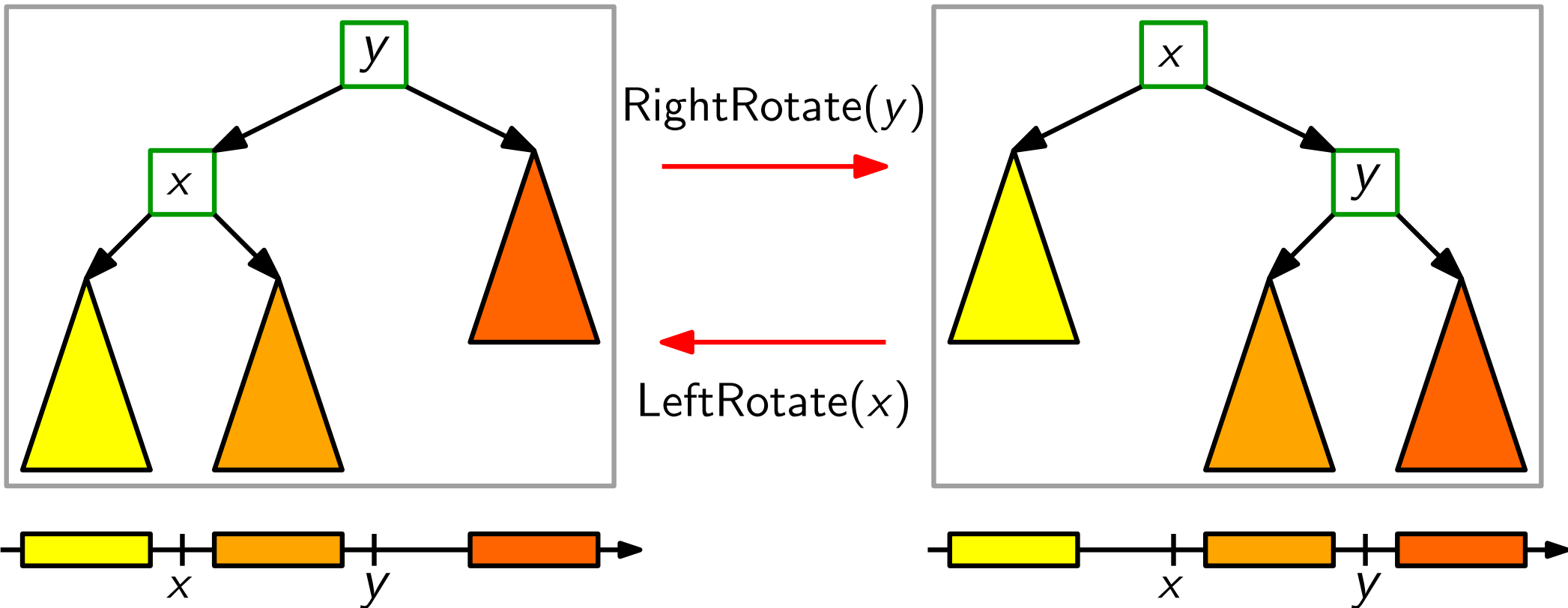
    RBInsertFixup( $z$ )  
**return**  $z$



Node(Key  $k$ , Node  $par$ )  
 $key = k$   
 $p = par$   
 $right = left = T.nil$

RBNode(..., Color  $c$ )  
 $super(k, par)$   
 $color = c$

# Exkurs: Rotationen



**Also:** Binärer-Suchbaum-Eig. bleibt beim Rotieren erhalten!

**Aufgabe:** Schreiben Sie Pseudocode für `LeftRotate(x)`!

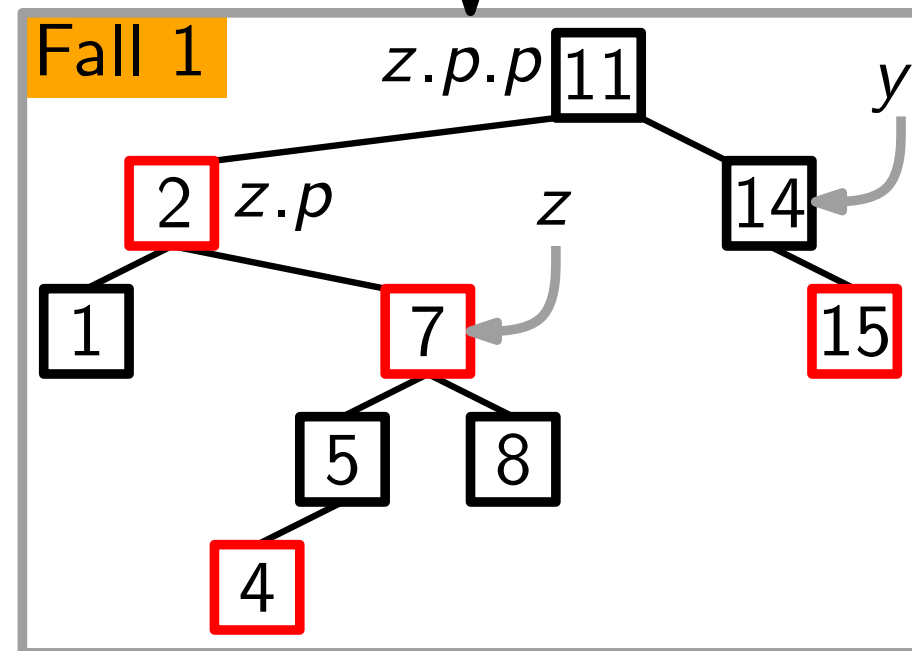
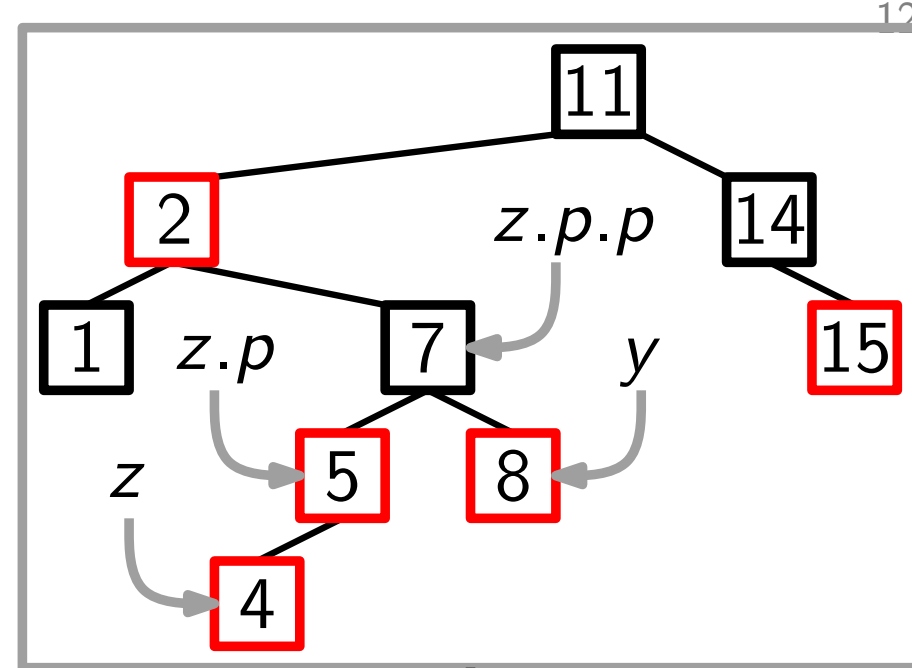
**Laufzeit:**  $O(1)$ .





# RBInsertFixup(Node z)

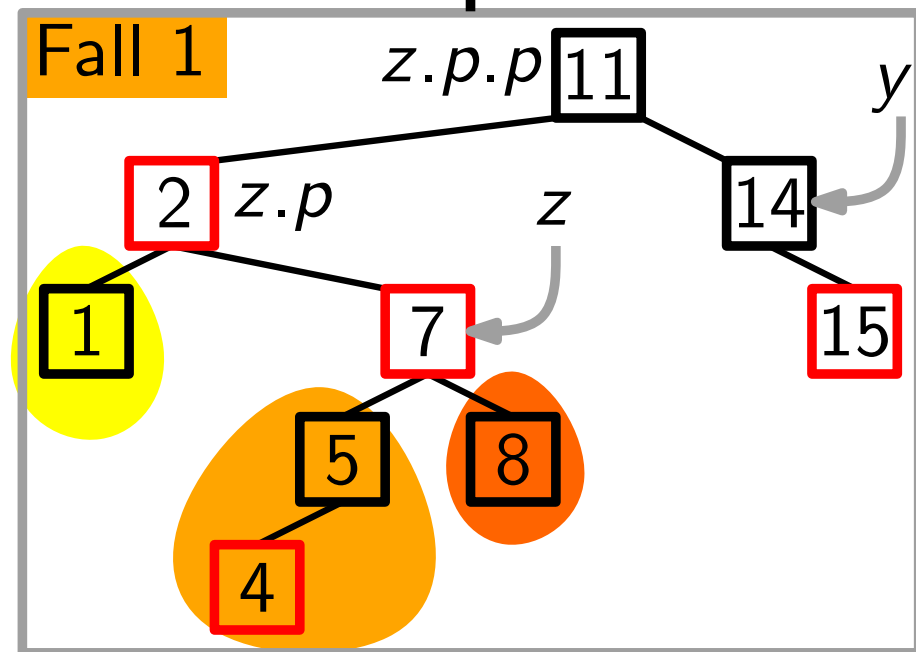
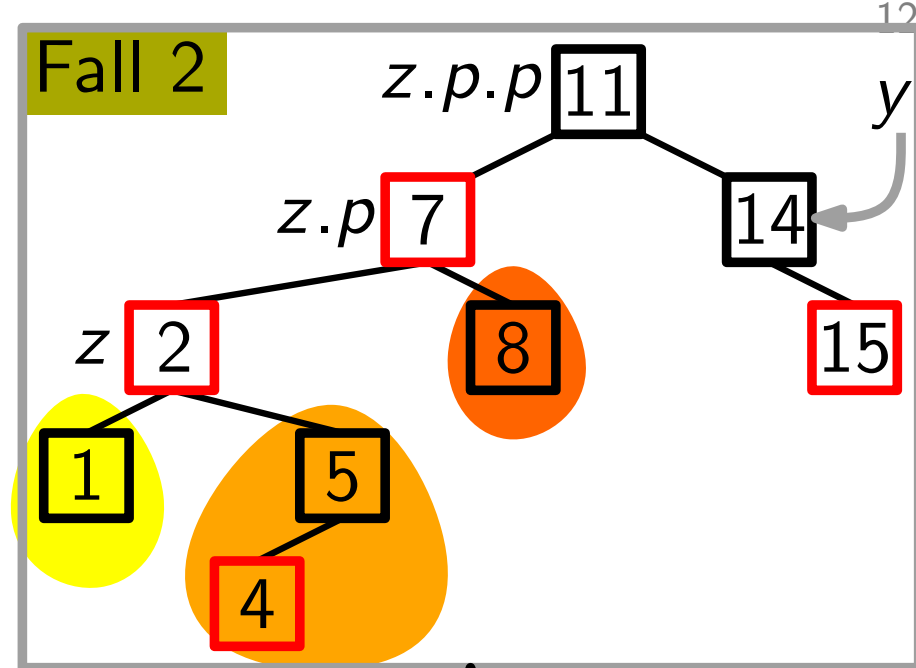
```
while z.p.color == red do
  if z.p == z.p.p.left then
    y = z.p.p.right // Tante von z
    if y.color == red then
      z.p.color = black
      z.p.p.color = red
      y.color = black
      z = z.p.p
    else
      if z == z.p.right then
        z = z.p
        LeftRotate(z)
        z.p.color = black
        z.p.p.color = red
        RightRotate(z.p.p)
      else ... // wie oben, aber re. & li. vertauscht
  root.color = black
```



# RBInsertFixup(Node z)

```

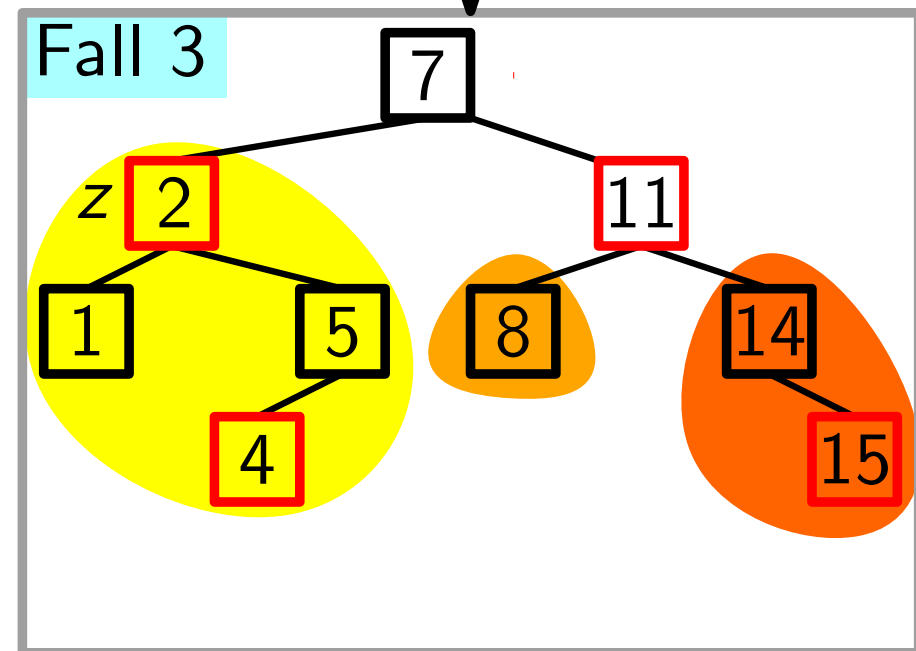
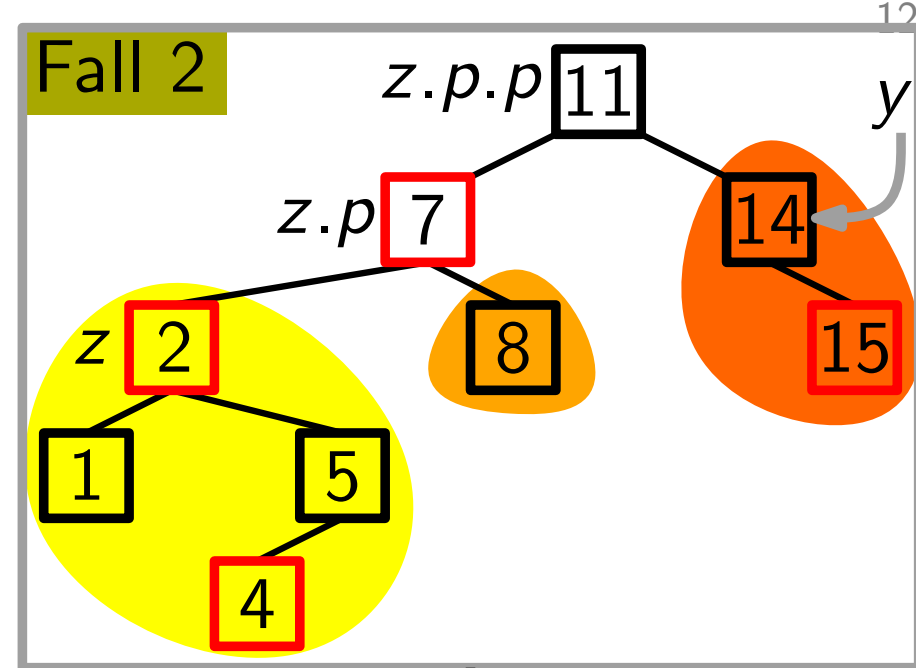
while z.p.color == red do
  if z.p == z.p.p.left then
    y = z.p.p.right // Tante von z
    if y.color == red then
      z.p.color = black
      z.p.p.color = red
      y.color = black
      z = z.p.p
    else
      if z == z.p.right then
        z = z.p
        LeftRotate(z)
        z.p.color = black
        z.p.p.color = red
        RightRotate(z.p.p)
      else ... // wie oben, aber re. & li. vertauscht
  root.color = black
  
```



# RBInsertFixup(Node z)

```

while z.p.color == red do
  if z.p == z.p.p.left then
    y = z.p.p.right // Tante von z
    if y.color == red then
      z.p.color = black
      z.p.p.color = red
      y.color = black
      z = z.p.p
    else
      if z == z.p.right then
        z = z.p
        LeftRotate(z)
        z.p.color = black
        z.p.p.color = red
        RightRotate(z.p.p)
      else ... // wie oben, aber re. & li. vertauscht
  root.color = black
  
```



# Korrektheit

**Zu zeigen:** RBInsertFixup stellt R-S-Eigenschaft wieder her.

**Schleifeninvariante** (gültig am Anfang der while-Schleife)

- $z$  ist rot.
- Falls  $z.p$  die Wurzel ist, dann ist  $z.p$  schwarz.
- Falls R-S-Eig. verletzt sind, dann entweder (E2) oder (E4).
  - Falls (E2) verletzt ist, dann weil  $z = root$  und  $z$  rot ist.
  - Falls (E4) verletzt ist, dann weil  $z$  und  $z.p$  rot sind.

**Zeige:**

- Initialisierung
- Aufrechterhaltung
- Terminierung

Viel Arbeit! Siehe [CLRS, Kapitel 13.3].

# Laufzeit RBInsertFixup

```

while z.p.color == red do
  if z.p == z.p.p.left then
    y = z.p.p.right
    if y.color == red then
      z.p.color = black
      z.p.p.color = red
      y.color = black
      z = z.p.p
    else
      if z == z.p.p.right then
        z = z.p.p
        LeftRotate(z)
        z.p.color = black
        z.p.p.color = red
        RightRotate(z.p.p)
      else ... // wie oben, aber re. & li. vertauscht
  root.color = black
  
```

Insgesamt:

- Fall 1  $O(h)$  mal
- Fall 2  $\leq 1$  mal
- Fall 3  $\leq 1$  mal

$O(\log n)$  Umfärbungen  
und  $\leq 2$  Rotationen

$O(1)$

Klettert im Baum  
2 Ebenen nach oben.

$O(1)$

Führt zum Abbruch  
der while-Schleife.

$O(1)$

# Löschen in (farblosen) binären Suchbäumen

Sei  $z$  der zu löschende Knoten. Wir betrachten drei Fälle:

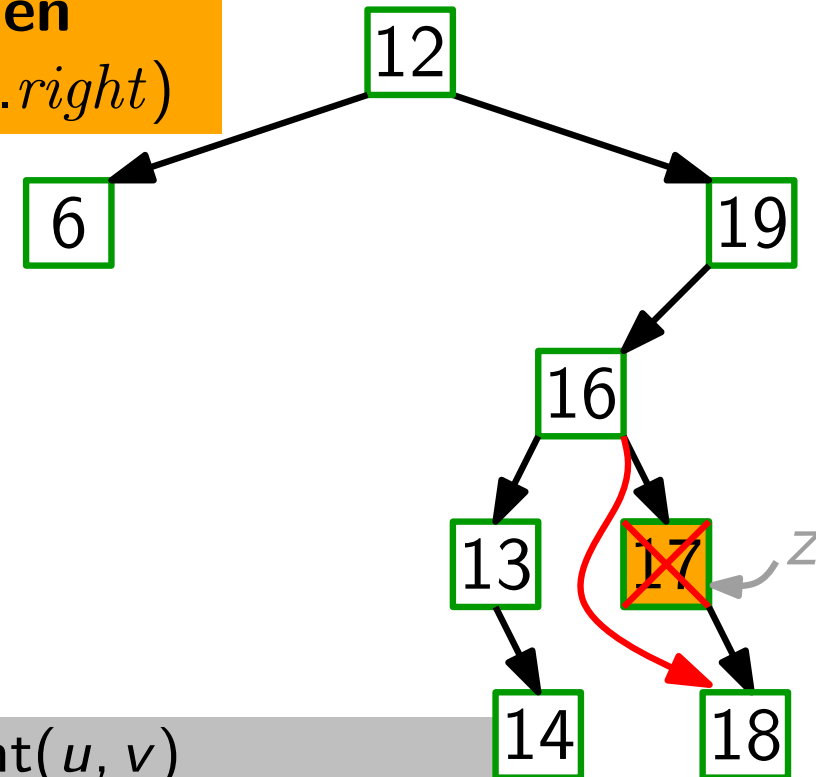
1.  $z$  hat kein li. Kind.

```
if z.left == nil then
  Transplant(z, z.right)
```

Setze  $z.right$  an die Stelle von  $z$ .  
Lösche  $z$ .

2.  $z$  hat kein re. Kind.

3.  $z$  hat zwei Kinder.



```
Transplant(u, v)
```

```
if u.p == nil then root = v
```

```
else
```

```
  if u == u.p.left then
```

```
    u.p.left = v
```

```
  else u.p.right = v
```

```
if v ≠ nil then v.p = u.p
```

Setze  $v$   
an die  
Stelle  
von  $u$ .



# Löschen (Übersicht)

Delete(Node z)

```
if z.left == nil then           // kein linkes Kind
|   Transplant(z, z.right)
else
|   if z.right == nil then      // kein rechtes Kind
|   |   Transplant(z, z.left)
|   else                         // zwei Kinder
|   |   y = Successor(z)
|   |   if y.p ≠ z then
|   |   |   Transplant(y, y.right)
|   |   |   y.right = z.right
|   |   |   y.right.p = y
|   |   Transplant(z, y)
|   |   y.left = z.left
|   |   y.left.p = y
```



## RBDelete(Node z)

```
if z.left == T.nil then
    RBTransplant(z, z.right)
else
    if z.right == T.nil then
        RBTransplant(z, z.left)
    else
        y = Successor(z)
        if y.p == z then
            else
                RBTransplant(y, y.right)
                y.right = z.right
                y.right.p = y
            RBTransplant(z, y)
            y.left = z.left
            y.left.p = y
```

## RBTransplant(u, v)

```
if u.p == T.nil then root = v
else
    if u == u.p.left then
        u.p.left = v
    else u.p.right = v
if v != nil then v.p = u.p
```

### Achtung – Trick!

Das kann dazu führen, dass manchmal der Elternzeiger von  $T.nil$  temporär auf den Elternknoten von  $u$  gesetzt wird.

```
RBDelete(Node z)
```

```
y = z; origcolor = y.color
```

```
if z.left == T.nil then
```

```
    x = z.right
```

```
    RBTransplant(z, z.right)
```

```
else
```

```
    if z.right == T.nil then
```

```
        x = z.left
```

```
        RBTransplant(z, z.left)
```

```
    else
```

```
        y = Successor(z)
```

```
        origcolor = y.color
```

```
        x = y.right
```

```
        if y.p == z then x.p = y
```

```
        else
```

```
            RBTransplant(y, y.right)
```

```
            y.right = z.right
```

```
            y.right.p = y
```

```
        RBTransplant(z, y)
```

```
        y.left = z.left
```

```
        y.left.p = y; y.color = z.color
```

```
if origcolor == black then RBDeleteFixup(x)
```

**y** zeigt auf den Knoten, der entweder gelöscht oder verschoben wird.

**x** zeigt auf den Knoten, der die Stelle von *y* einnimmt – das ist entweder das einzige Kind von *y* oder *T.nil*.

➔ Falls *y* ursprünglich *rot* war, bleiben alle R-S-Eig. erhalten:

- Keine Schwarzhöhe hat sich verändert.
- Keine zwei roten Knoten sind Nachbarn geworden.
- *y* rot  $\Rightarrow y \neq$  Wurzel  $\Rightarrow$  Wurzel bleibt schwarz.

# RBDeleteFixup

*Was kann schief gehen, wenn  $y$  schwarz war?*

- (E2)  $y$  war Wurzel, und ein rotes Kind von  $y$  wurde Wurzel.
- (E4)  $x$  und  $x.p$  sind rot.
- (E5) Falls  $y$  verschoben wurde, haben jetzt alle Pfade, die vorher  $y$  enthielten, einen schwarzen Knoten zu wenig.

„Repariere“ Knoten  $x$  zählt eine schwarze Einheit extra  
 (E5): (ist also „rot-schwarz“ oder „doppelt schwarz“)

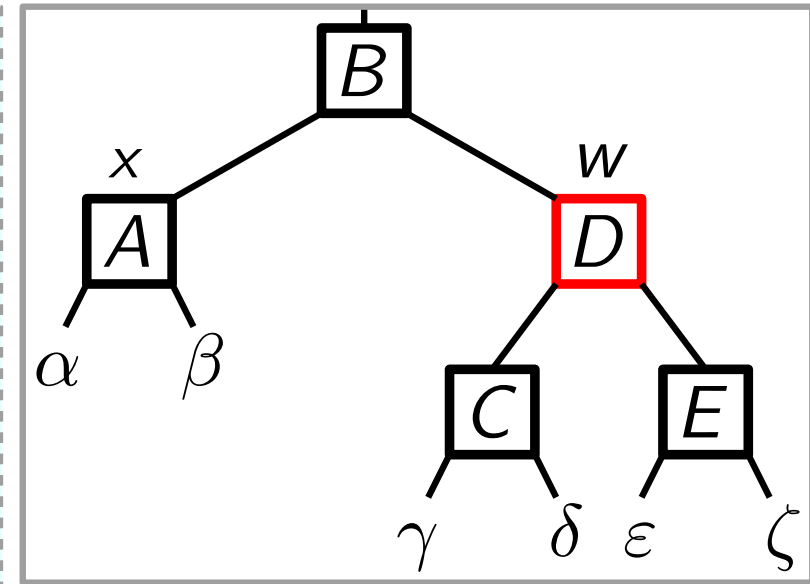
- Ziel:** Schiebe die überzählige schwarze Einheit nach oben, bis:
- $x$  ist rot-schwarz  $\Rightarrow$  mach  $x$  schwarz.
  - $x$  ist Wurzel  $\Rightarrow$  schwarze Extra-Einheit verfällt.
  - Problem wird lokal durch Umfärben & Rotieren gelöst.

# RBDeleteFixup(RBNode x)

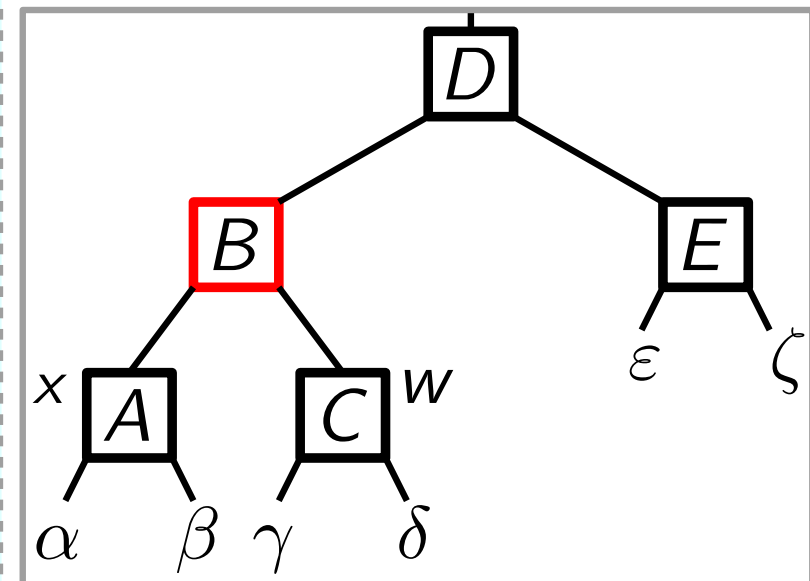
```

while x ≠ root and x.color == black do
  if x == x.p.left then
    w = x.p.right // Schwester von x
    if w.color == red then
      w.color = black
      x.p.color = red
      LeftRotate(x.p)
      w = x.p.right
    if w.left.color == black and
       w.right.color == black then
      w.color = red
      x = x.p
    else // kommt gleich!!
  else // wie oben; nur left ↔ right
x.color = black
  
```

Ziel:  
 $w \rightarrow$  schwarz  
 ohne R-S-Eig.  
 zu verletzen.



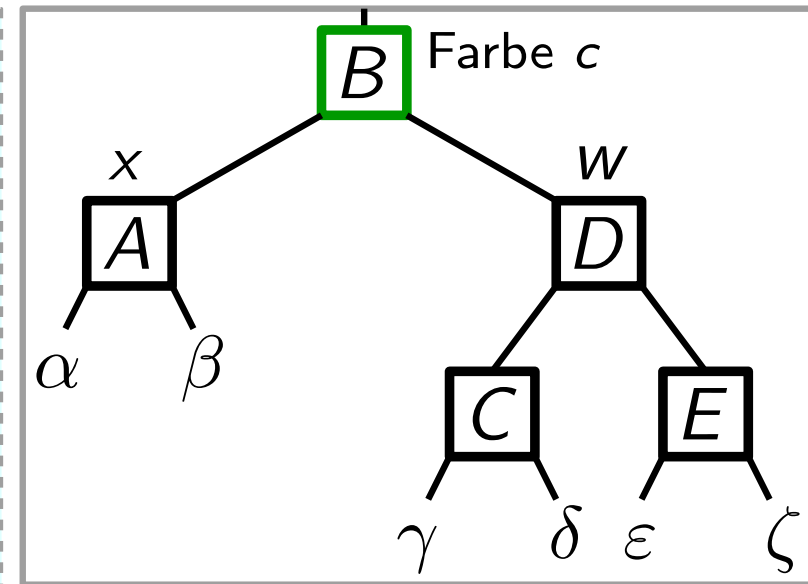
↓ Fall 1



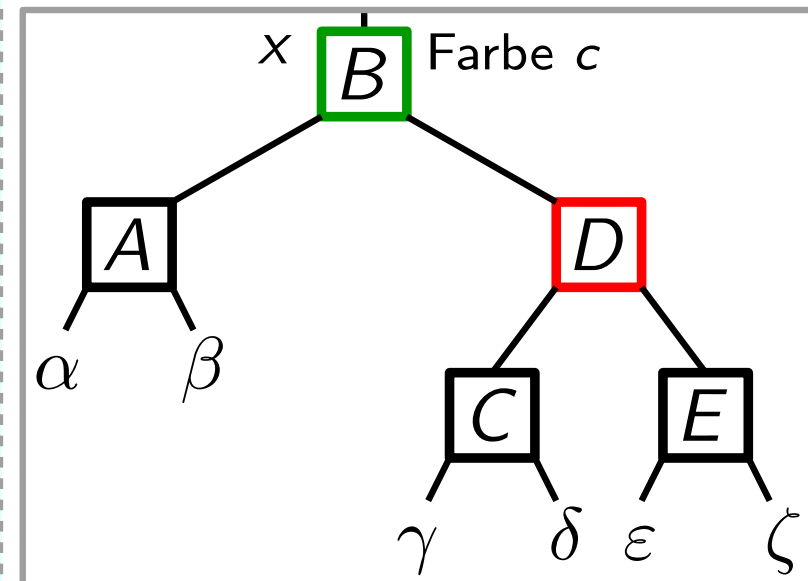
# RBDeleteFixup(RBNode x)

```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
     $w = x.p.\text{right}$  // Schwester von x
    if  $w.\text{color} == \text{red}$  then
       $w.\text{color} = \text{black}$ 
       $x.p.\text{color} = \text{red}$ 
      LeftRotate( $x.p$ )
       $w = x.p.\text{right}$ 
      Ziel:
       $w \rightarrow$  schwarz
      ohne R-S-Eig.
      zu verletzen.
    if  $w.\text{left}.\text{color} == \text{black}$  and
        $w.\text{right}.\text{color} == \text{black}$  then
       $w.\text{color} = \text{red}$ 
      Schw. Einheit
      raufschieben.
       $x = x.p$ 
    else // kommt gleich!!
  else // wie oben; nur left  $\leftrightarrow$  right
   $x.\text{color} = \text{black}$ 
  
```



↓ Fall 2



**Bem.:** Anz. der schw. Knoten (inkl. Extra-Einh. bei x) bleibt auf allen Pfaden gleich!

# RBDeleteFixup (Forts.)

**else**

**if**  $w.right.color == black$  **then**

$w.left.color = black$

$w.color = red$

RightRotate( $w$ )

$w = x.p.right$

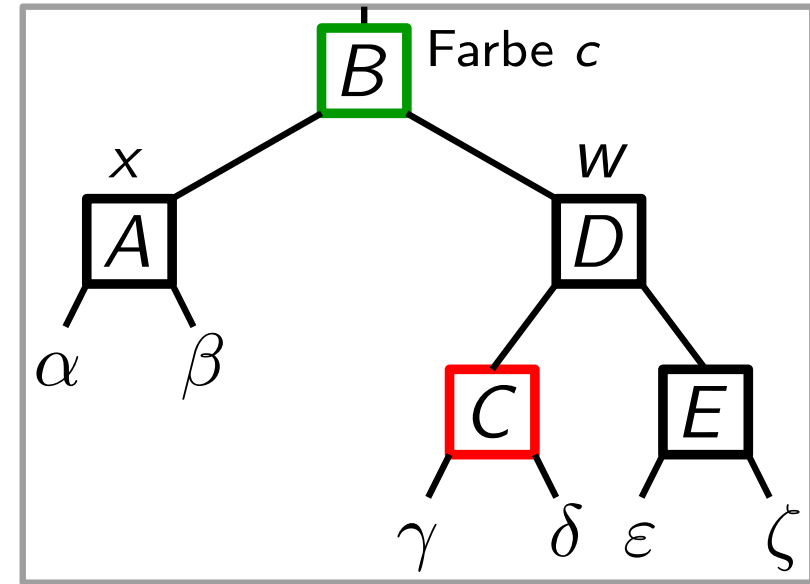
$w.color = x.p.color$

$x.p.color = black$

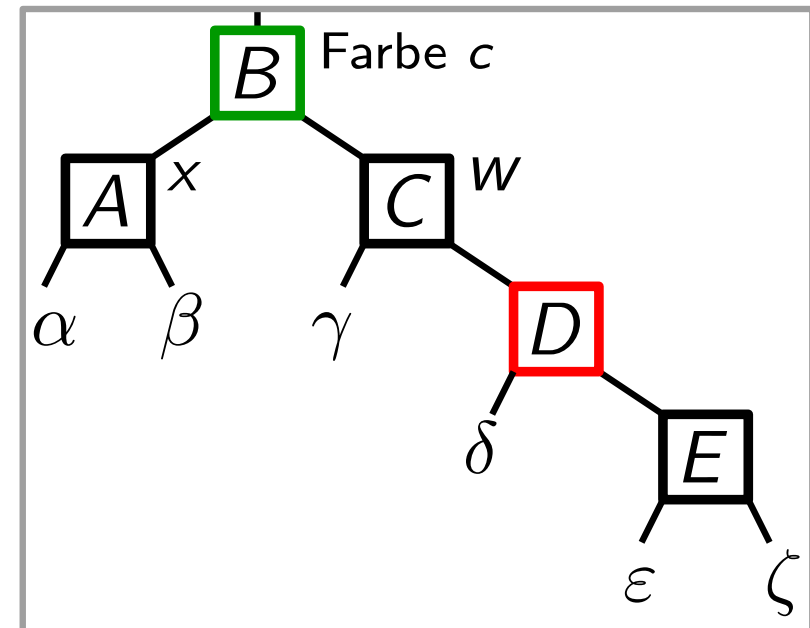
$w.right.color = black$

LeftRotate( $x.p$ )

$x = root$



Fall 3



# RBDeleteFixup (Forts.)

**else**

**if**  $w.right.color == black$  **then**

$w.left.color = black$

$w.color = red$

RightRotate( $w$ )

$w = x.p.right$

$w.color = x.p.color$

$x.p.color = black$

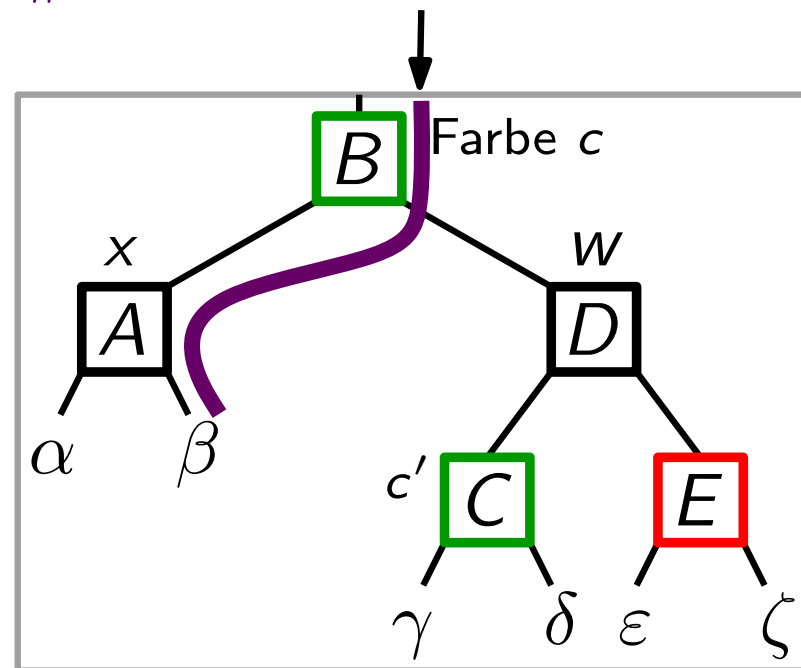
$w.right.color = black$

LeftRotate( $x.p$ )

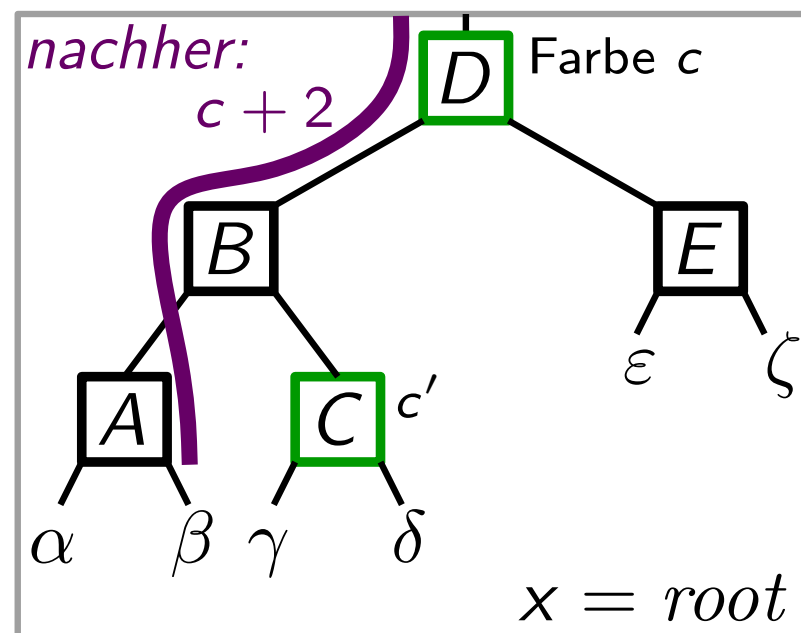
$x = root$

**Bem.:** Anz. der schwarzen Knoten  
(inkl. der Extra-Einheit bei  $x$ )  
bleibt auf allen Pfaden gleich!

vorher:  
# schwarze Einheiten =  $c + 2$



Fall 4



# RBDeleteFixup (Forts.)

**else**

**if**  $w.right.color == black$  **then**

$w.left.color = black$

$w.color = red$

RightRotate( $w$ )

$w = x.p.right$

$w.color = x.p.color$

$x.p.color = black$

$w.right.color = black$

LeftRotate( $x.p$ )

$x = root$

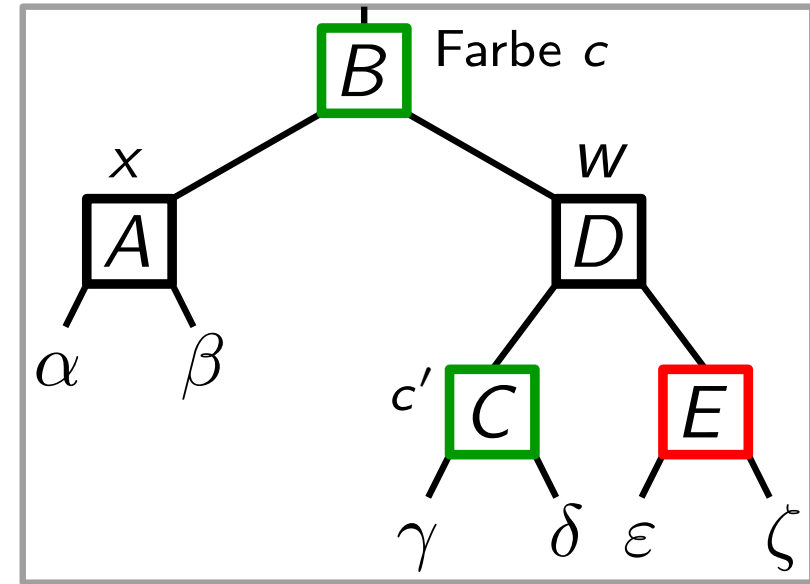
**Laufzeit?**

Fall 1:  $O(h)$  Umfärbungen

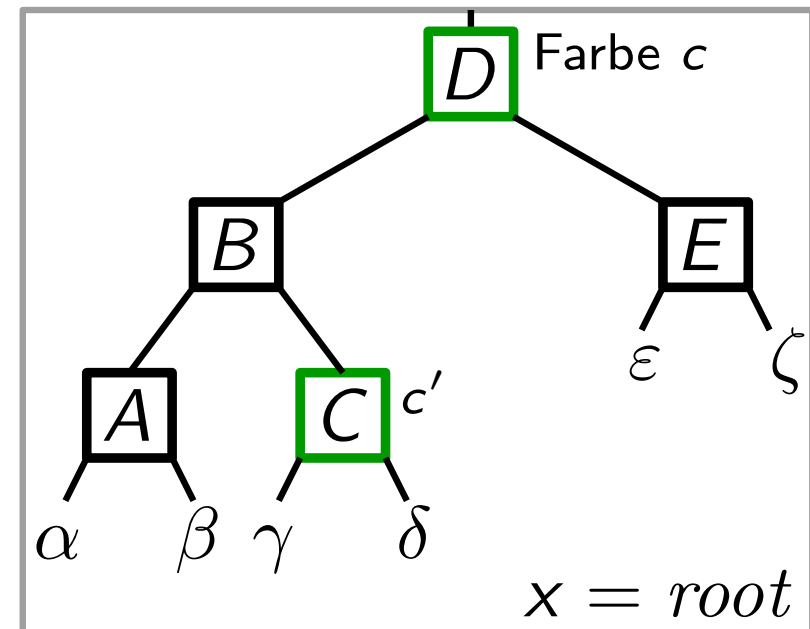
Fall 2: 1 Rotation +  $O(1)$

Fall 3: 1 Rotation +  $O(1)$

Fall 4: 1 Rotation +  $O(1)$



↓ Fall 4





# Zusammenfassung

$$\text{Laufzeit RBDelete} \in O(h) + \underbrace{\text{Laufzeit RBDeleteFixup}}_{O(h)} = O(h)$$

RBDelete erhält die Rot-Schwarz-Eigenschaften.

Also gilt (siehe Lemma):  $h \in O(\log n)$



Laufzeit RBDelete  $\in O(\log n)$

## Satz.

Rot-Schwarz-Bäume implementieren alle dynamische-Menge-Operationen in  $O(\log n)$  Zeit, wobei  $n$  die momentane Anz. der Schlüssel ist.