Algorithmen und Datenstrukturen

Marc Fischlin, SS 2024

04

Fortgeschrittene Datenstrukturen

Rot-Schwarz-Bäume

Binäre Suchbäume

Operation	Laufzeit*
Einfügen	$\Theta(h)$
Löschen	$\Theta(h)$
Suchen	$\Theta(h)$

Können wir $h = O(\log n)$ garantieren?

Rot-Schwarz-Bäume in Java

OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX H

SUMMARY: NESTED | FIELD | CONSTR | METHOD   DETAIL: FIELD | CONSTR | METHOD

Module java.base
Package java.util

Class TreeMap<K,V>

java.lang.Object
    java.util.AbstractMap<K,V>
        java.util.TreeMap<K,V>

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All Implemented Interfaces:

Serializable, Cloneable, Map<K,V>, NavigableMap<K,V>, SequencedMap<K,V>, So

```
public class TreeMap<K,V>
extends AbstractMap<K,V>
implements NavigableMap<K,V>, Cloneable, Serializable
```

A Red-Black tree based NavigableMap implementation. The map is sorted accordin
Comparator provided at map creation time, depending on which constructor is use

This implementation provides guaranteed log(n) time cost for the containsKey, ge
adaptations of those in Cormen, Leiserson, and Rivest's *Introduction to Algorithms*

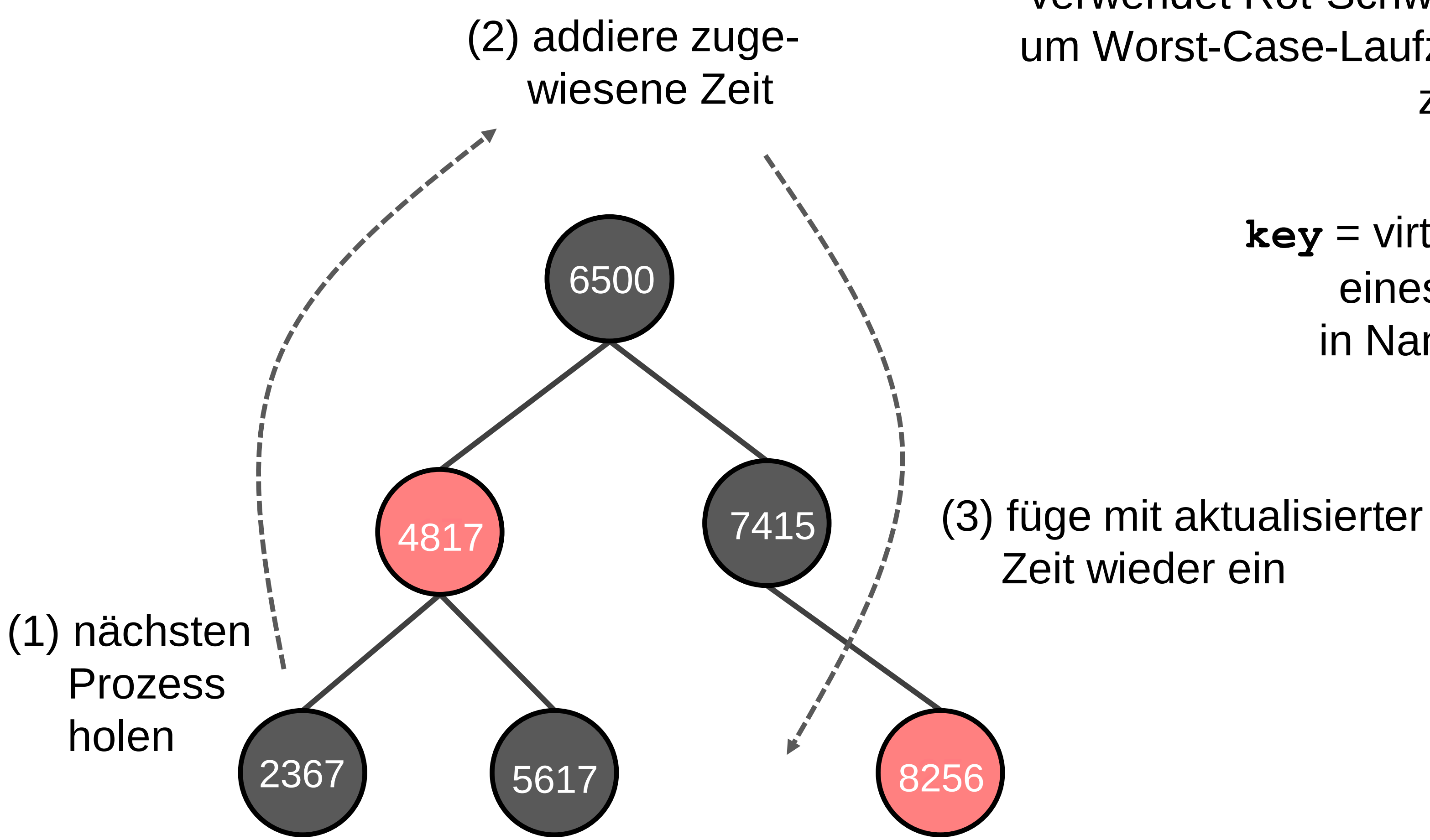
Quelle: Wikipedia

Class TreeMap in Java 22

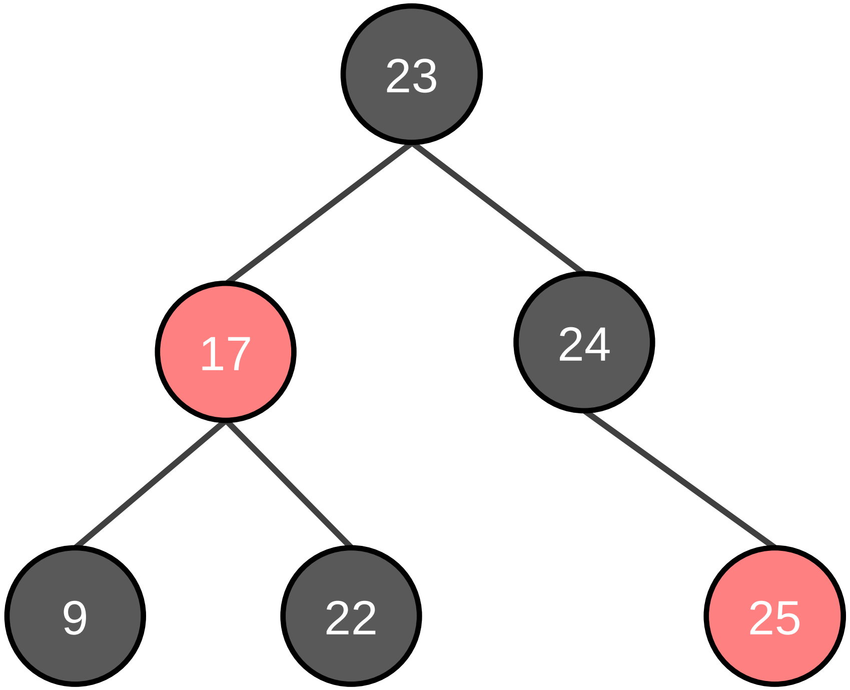
Anwendung: Linux Completely Fair Scheduling

verwendet Rot-Schwarz-Bäume, um Worst-Case-Laufzeit $O(\log n)$ zu erreichen

`key` = virtual run time eines Prozesses in Nanosekunden



Rot-Schwarz-Bäume



zusätzlicher Knoten-Eintrag
`x.color=red` oder `black`

Ein **Rot-Schwarz-Baum** ist ein binärer Suchbaum, so dass gilt:

(1) Jeder Knoten hat die Farbe rot oder schwarz,

(2) Die Wurzel ist schwarz*,

(3) Wenn ein Knoten rot ist, sind seine Kinder schwarz („Nicht-Rot-Rot"-Regel),

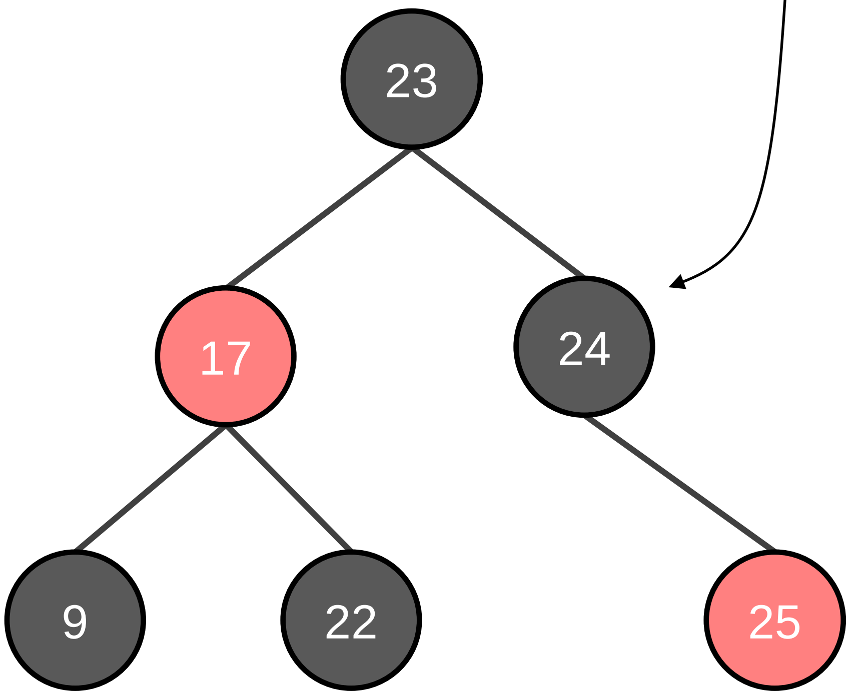
(4) Für jeden Knoten hat jeder Pfad im Teilbaum zu einem Blatt oder Halbblatt die gleiche Anzahl von schwarzen Knoten („gleiche Anzahl schwarz").

*sofern Baum nicht leer

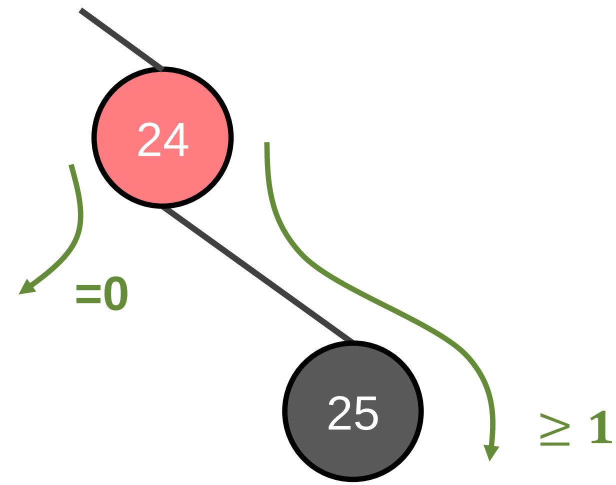
Bestimmte Farben

⇔ rote Knoten haben genau 0 oder 2 Kinder

Halbblätter im Rot-Schwarz-Baum sind schwarz

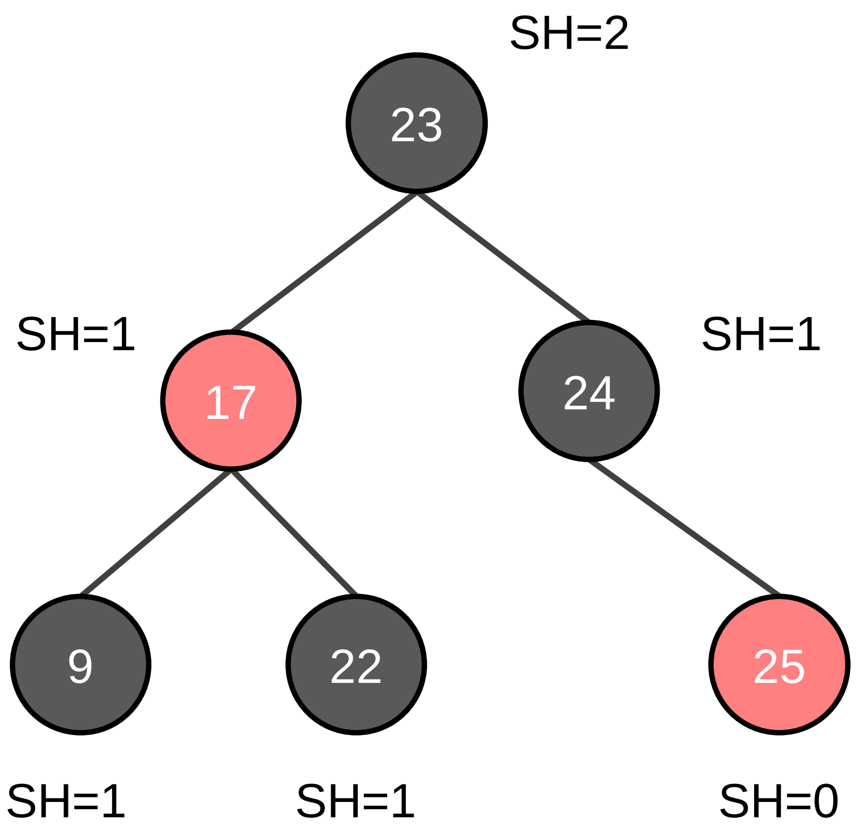


Wenn Halbblatt rot wäre,
dann wäre (einziges) Kind schwarz.



Dann gäbe es
im kinderlosen Pfad keinen schwarzen Knoten,
im anderen aber mindestens einen schwarzen Knoten.

„Schwarzhöhe“ eines Knoten



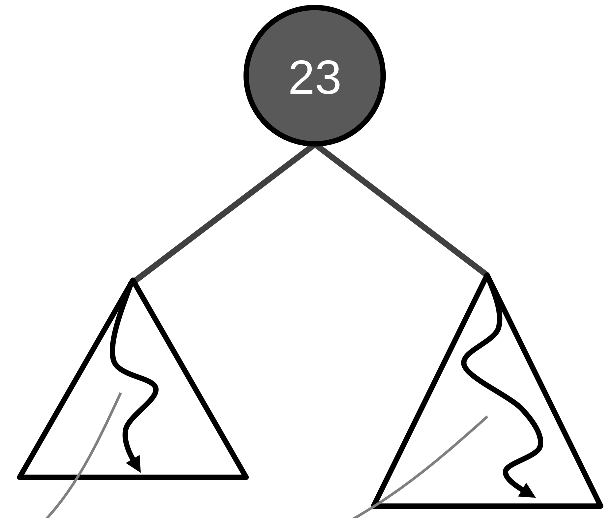
Die **Schwarzhöhe** eines Knoten $\mathbf{x}$ ist die (eindeutige) Anzahl von schwarzen Knoten auf dem Weg zu einem Blatt oder Halbblatt im Teilbaum des Knoten

Für leeren Baum setzt man

$$\text{SH}(nil) = 0$$

Höhe eines Rot-Schwarz-Baums

Ein Rot-Schwarz-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 $O(\log n)$

Höhe eines Rot-Schwarz-Baums: Beweis (I)

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

Zeige zuerst:
Teilbaum in Knoten $\mathbf{x}$ hat mindestens $2^{SH(x)} - 1$ Knoten

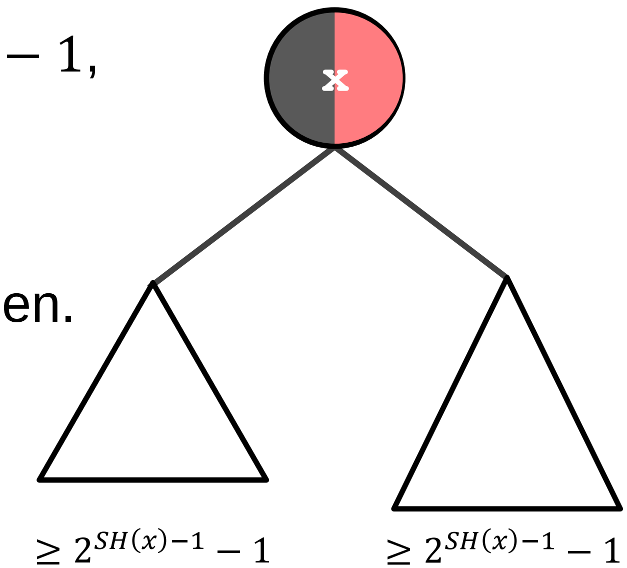
Beweis per Induktion:

Leerer Baum hat $n = 0$ Knoten und mind. $2^{SH(x)} - 1 = 2^0 - 1 = 0$ Knoten.

Teilbäume haben Schwarzhöhe $SH(x)$ oder $SH(x) - 1$, je nachdem ob $\mathbf{x}$ rot oder schwarz.

Induktionsvoraussetzung:
Also jeweils mind. $2^{SH(x)-1} - 1$ Knoten in Teilbäumen.

Insgesamt mindestens
$(2^{SH(x)-1} - 1) + (2^{SH(x)-1} - 1) + 1 = 2^{SH(x)} - 1$
Knoten im Teilbaum von $\mathbf{x}$.



Höhe eines Rot-Schwarz-Baums: Beweis (II)

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

Zeige zuerst:
Teilbaum in Knoten $\mathbf{x}$ hat mindestens $2^{SH(x)} - 1$ Knoten

Sei h Höhe der Baumes mit Wurzel $\mathbf{r}$ und n Knoten.

Dann $SH(\mathbf{r}) \geq h/2$, da maximal die Hälfte der Knoten auf längstem Pfad rot.

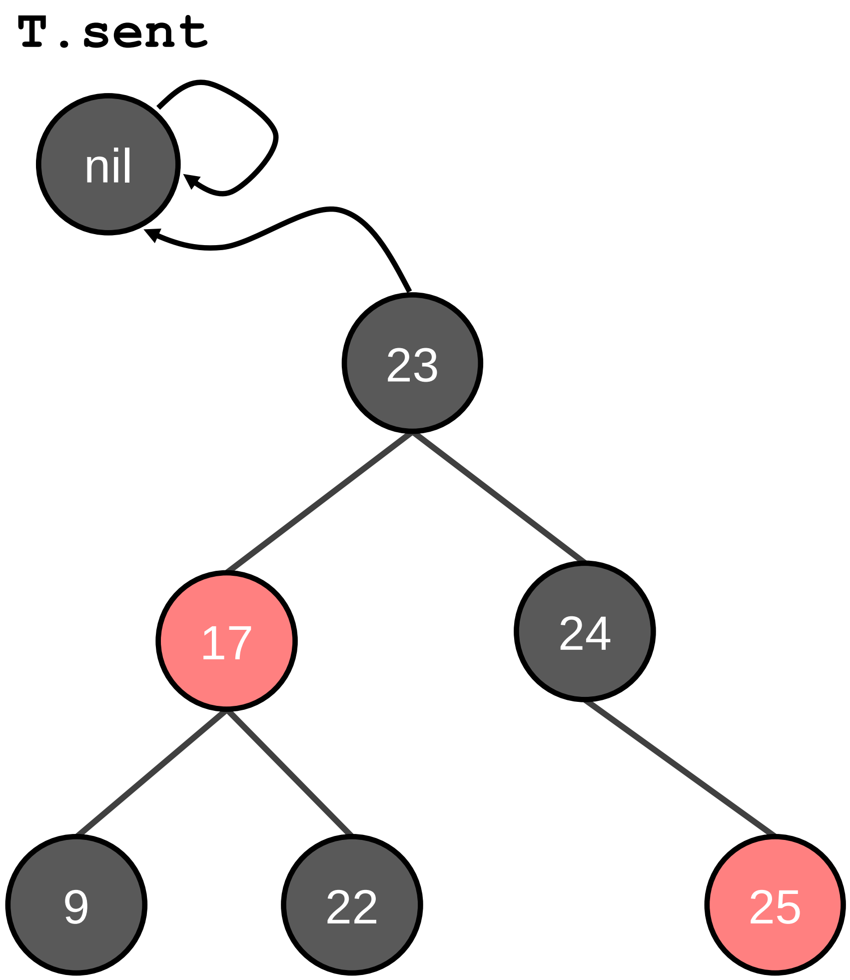
Dann $n \geq 2^{h/2} - 1$ bzw. $\log_2(n+1) \geq h/2$. ■

Kann ein Rot-Schwarz-Baum nur aus schwarzen Knoten bestehen? Wie sieht er dann aus?

Gibt es für jedes n einen Rot-Schwarz-Baum, der die obere Schranke $2 \cdot \log(n+1)$ für die Höhe auch (fast) erreicht?

Wie sieht ein Algorithmus aus, der überprüft, ob ein BST auch ein Rot-Schwarz-Baum ist?

Implementierungen mittels Sentinel

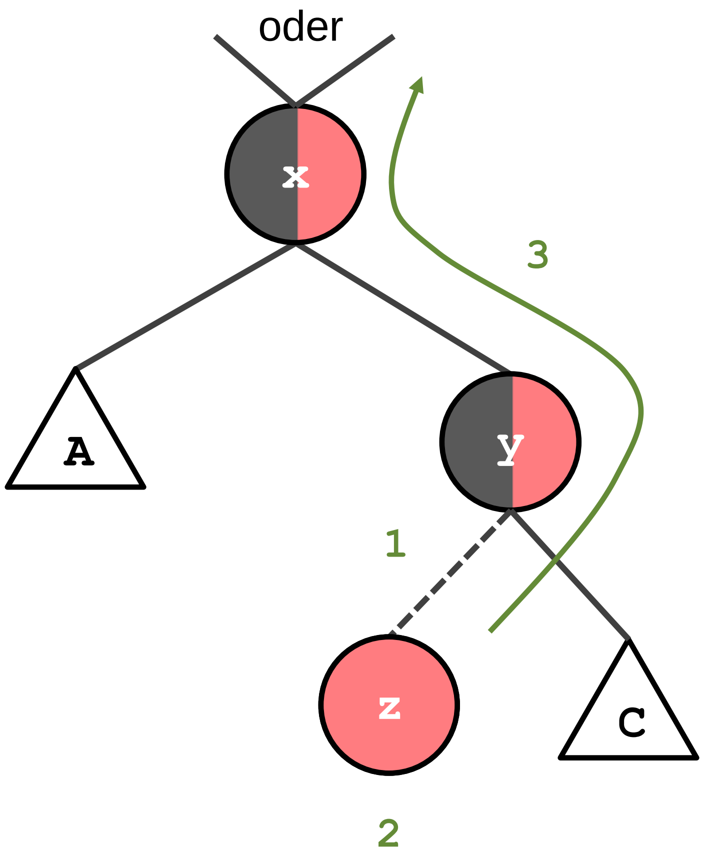


`T.root.parent=T.sent`

`T.sent.key=nil`
`T.sent.color=black`
`T.sent.parent=T.sent`
`T.sent.left=T.sent`
`T.sent.right=T.sent`

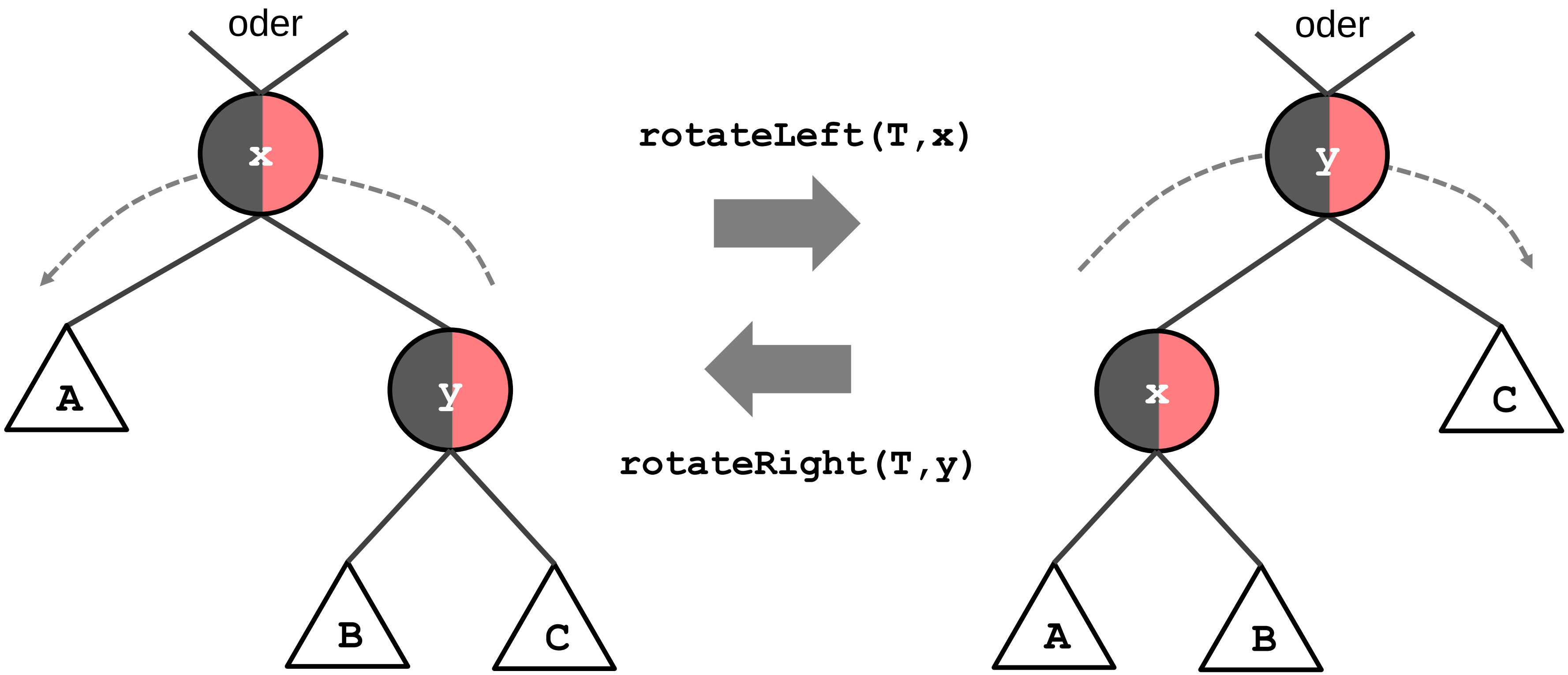
Beispiel:
`x.parent.parent.left`
immer wohldefiniert

Einfügen



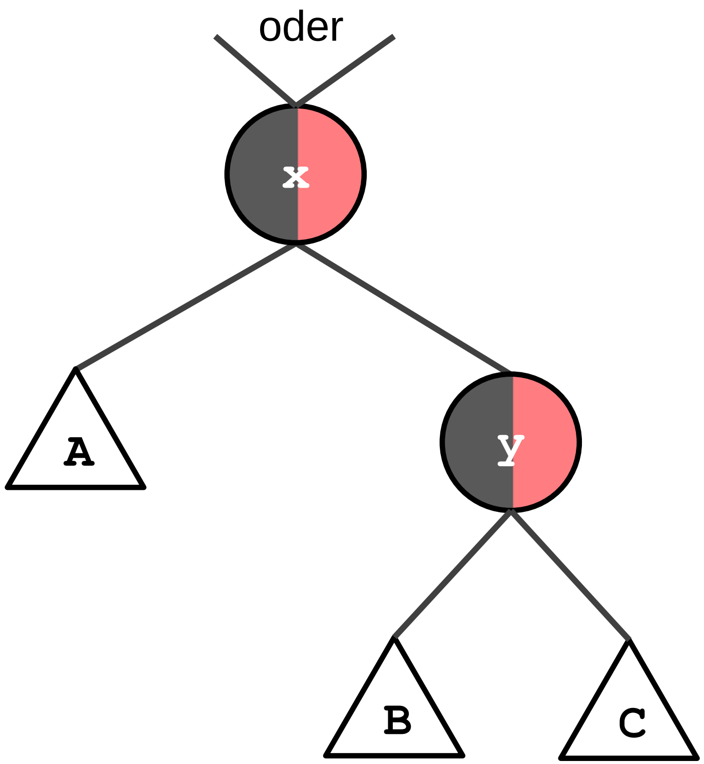
1. Finde Elternknoten **y** wie im BST
2. Färbe neuen Knoten **z** rot
3. Stelle RS-Baum-Bedingung wieder her

Einfügen: Rotation



Achtung: Rot-Schwarz-Baum-Bedingungen
sind nach Rotation evtl. verletzt

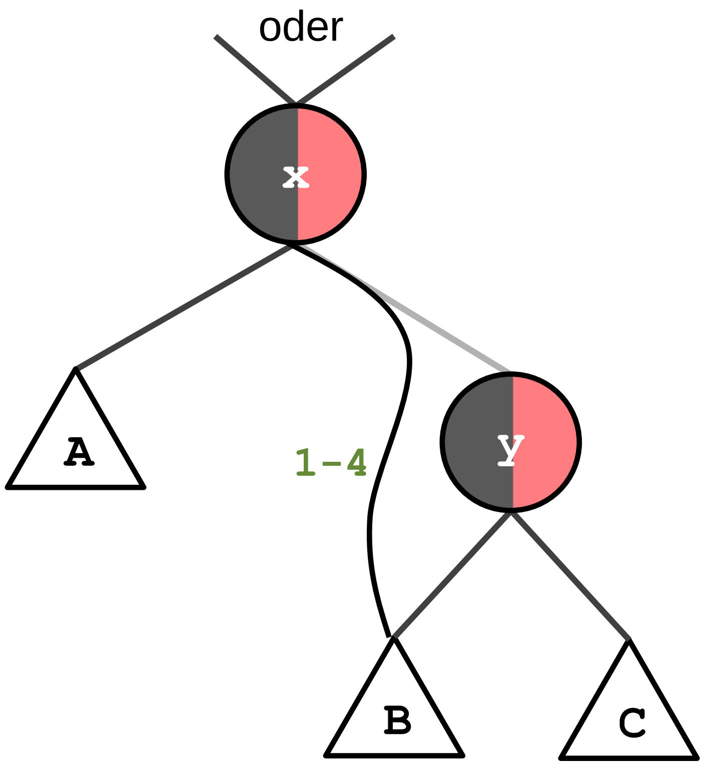
Rotation: Algorithmus (I)




```
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;
```

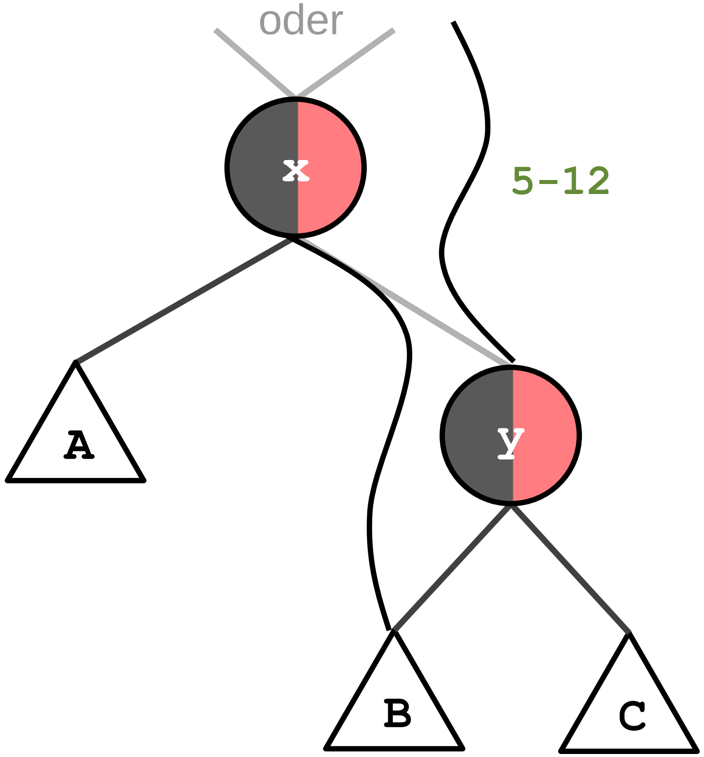
Laufzeit = $\Theta(1)$

Rotation: Algorithmus (II)



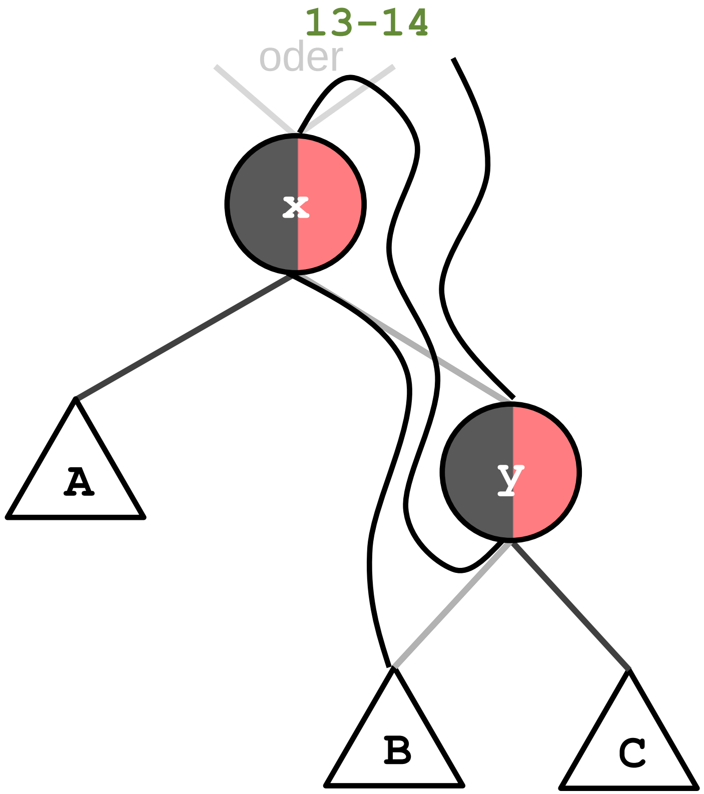

```
rotateLeft(T,x) //x.right!=nil

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;
```

Rotation: Algorithmus (III)




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

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;
```

Rotation: Algorithmus (IV)




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

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;
```

Einfügen

Funktioniert wie beim binären Suchbaum (mit Sentinel)

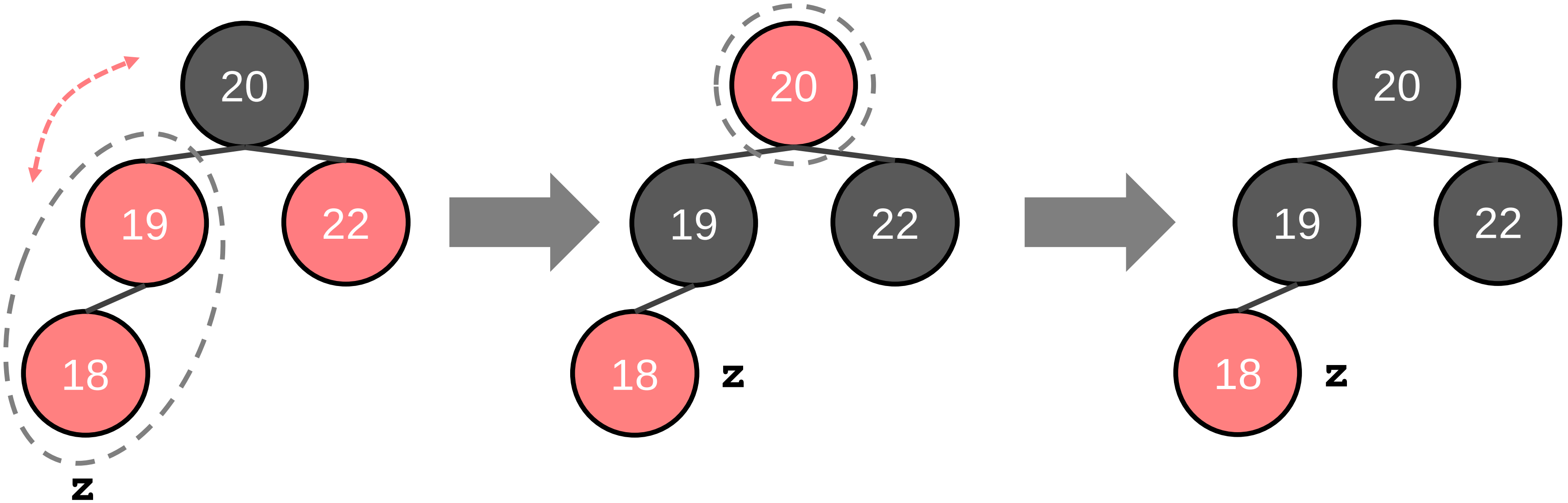
Änderung: Farbe des neuen Knoten auf rot setzen, dann RSB-Bedingung wieder herstellen

```
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);
```

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
      … //exchange left and right
T.root.color=black;
```

Aufräumen: Idee für einfache Bäume (I)

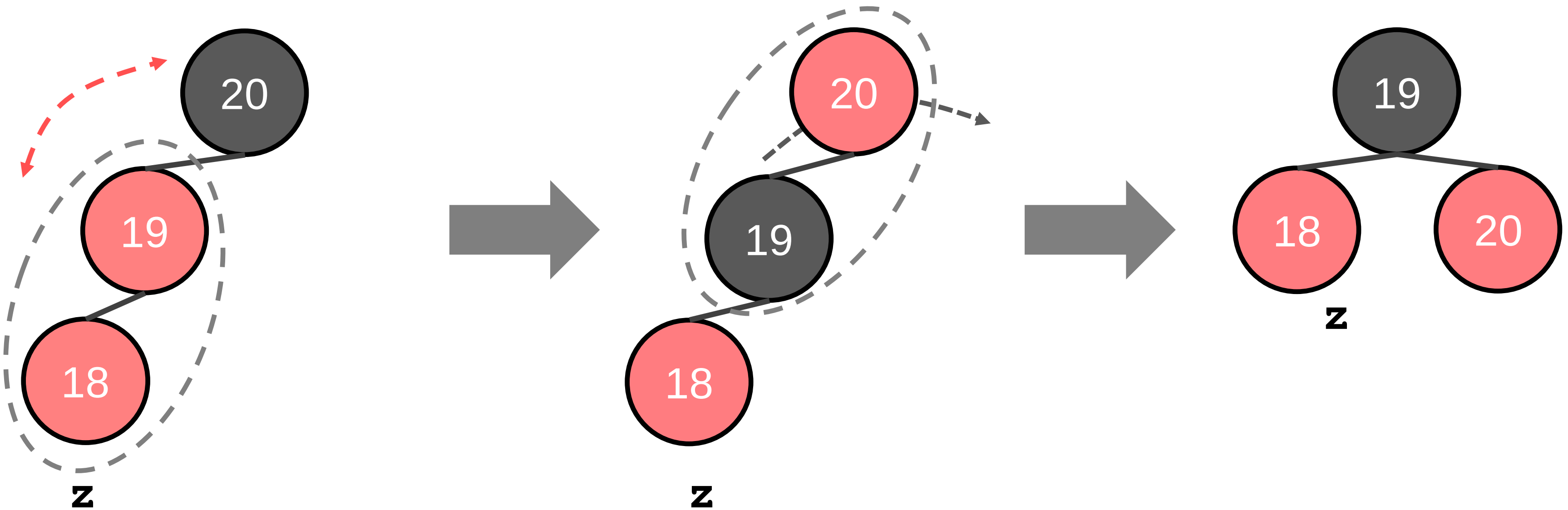


Schwarzhöhe des Baumes bleibt erhalten

Achtung: Zweiter Schritt geht hier nur, weil 20 Wurzel ist

Wenn Teilbaum, dann stattdessen „rekursiv in Knoten 20 aufräumen“

Aufräumen: Idee für einfache Bäume (II)



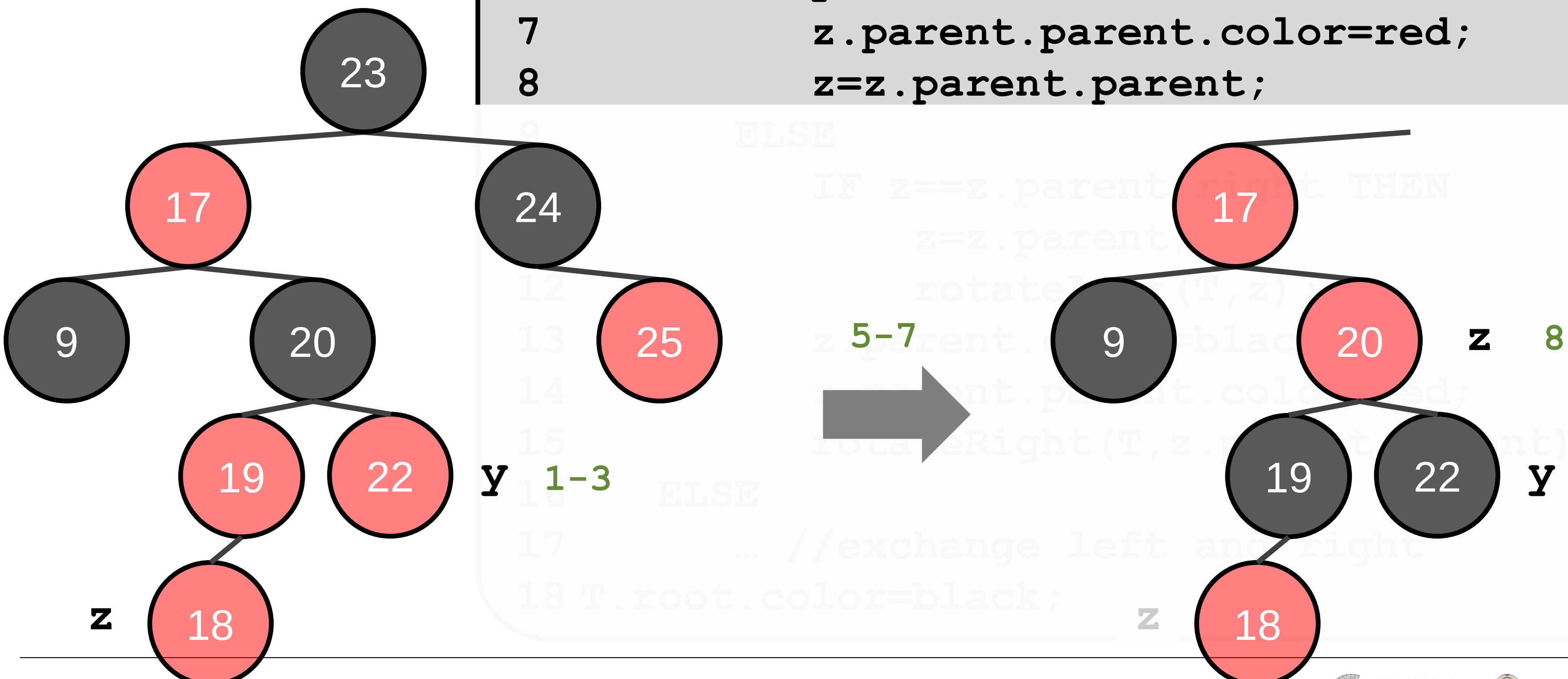
Schwarzhöhe des
Baumes bleibt erhalten

Selbst wenn Teilbaum,
dann kein weiteres Aufräumen nötig

Aufräumen (I)

z nicht Wurzel,
nicht Kind der Wurzel

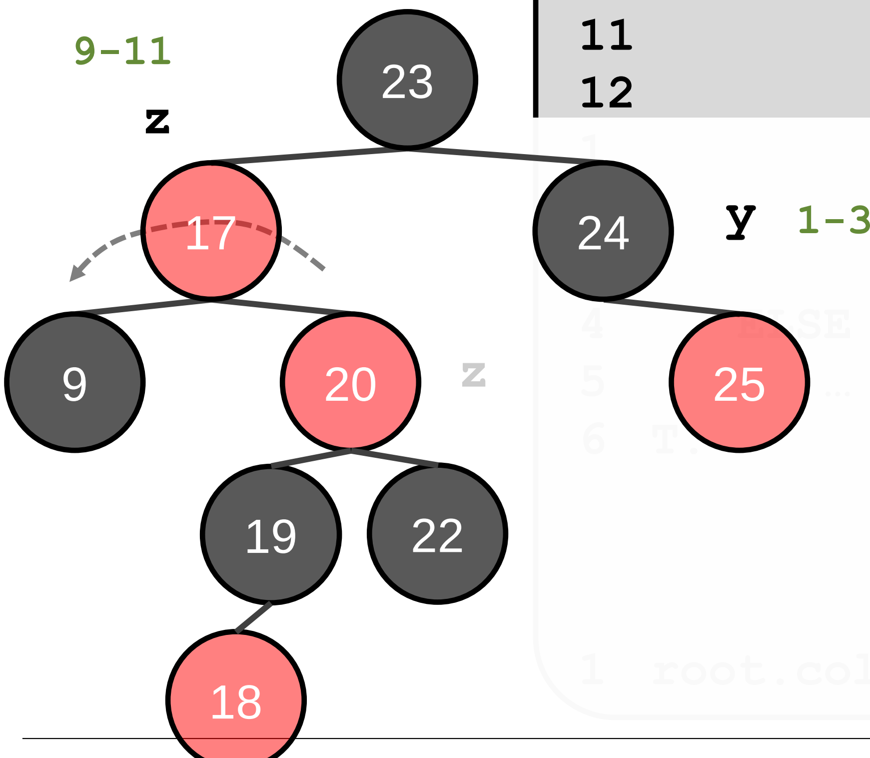
```
fixColorsAfterInsertion(T,z)

1  WHILE z.parent.color==red DO
2    IF z.parent==z.parent.parent.left THEN
3        y=z.parent.parent.right;
4        IF y!=nil AND y.color==red THEN
5           z.parent.color=black;
6           y.color=black;
7           z.parent.parent.color=red;
8           z=z.parent.parent;
```

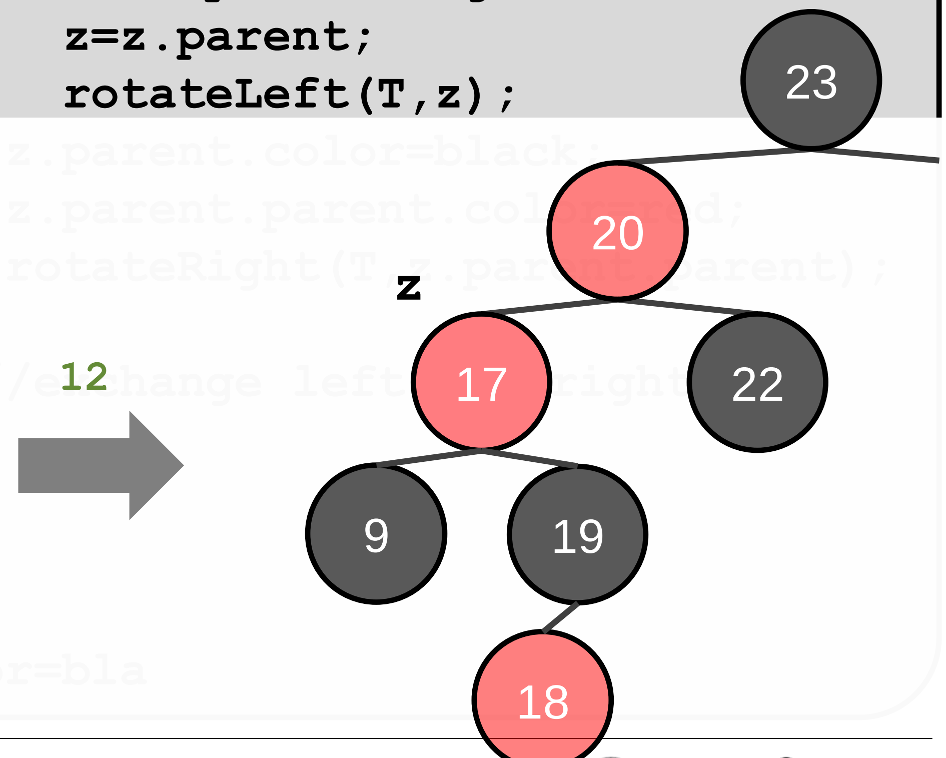


Aufräumen (II)

nächste
WHILE-Iteration

```
fixColorsAfterInsertion(T,z)
1  WHILE z.parent.color==red DO
2     IF z.parent==z.parent.parent.left THEN
3          y=z.parent.parent.right;
4          IF y!=nil AND y.color==red THEN
                 …
9          ELSE
10           IF z==z.parent.right THEN
11                z=z.parent;
12                rotateLeft(T,z);
```

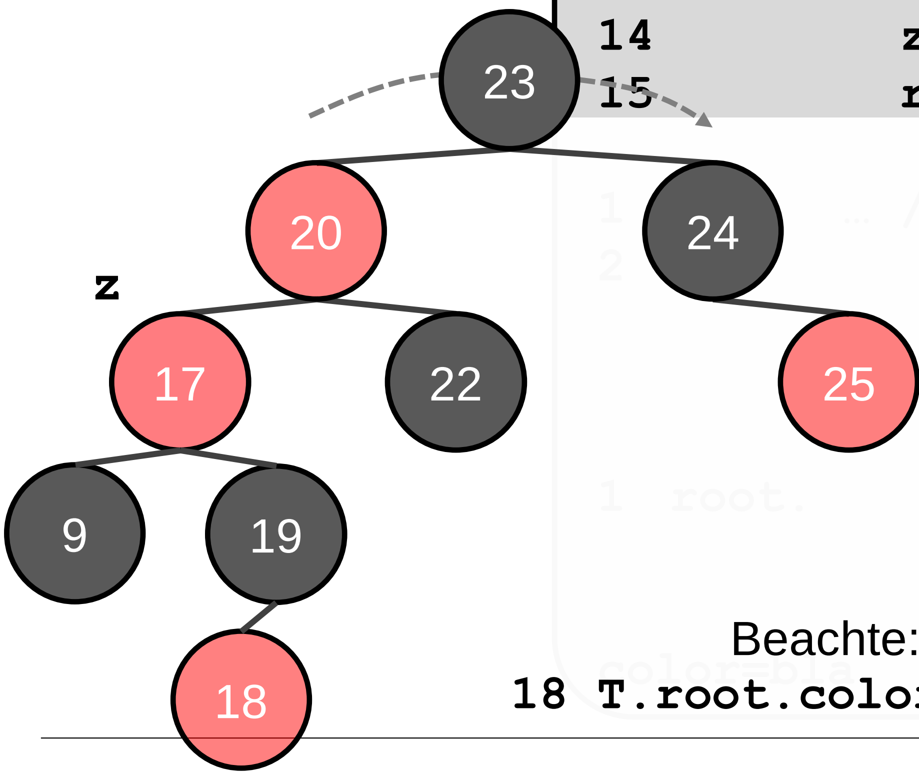


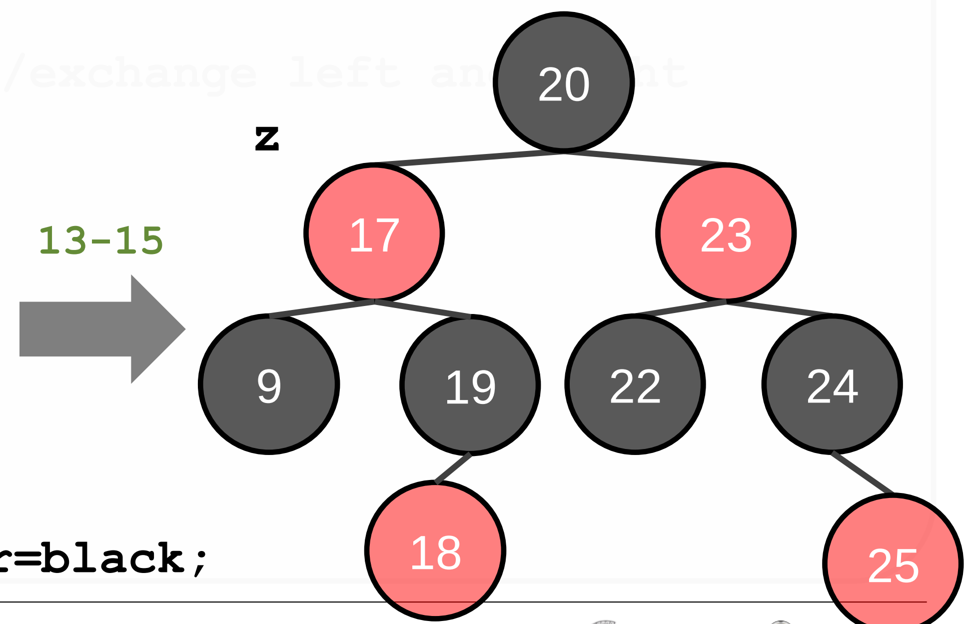


Aufräumen (III)

`WHILE`-Schleife
`z.parent.color=red`
hier im Beispiel
danach beendet

```
fixColorsAfterInsertion(T,z)
        ...

9          ELSE
10            IF z==z.parent.right THEN
11               z=z.parent;
12               rotateLeft(T,z);
13            z.parent.color=black;
14            z.parent.parent.color=red;
15            rotateRight(T,z.parent.parent);
```





Beachte:
`18 T.root.color=black;`

Aufräumen: Schleifeninvariante (I)

Schleifeninvariante:

1. **`z.color==red`**
2. Wenn **`z.parent`** Wurzel, dann **`z.parent.color==black`**
3. Wenn der aktuelle Baum kein Rot-Schwarz-Baum ist,
   dann weil **`z`** als Wurzel die Farbe rot hat,
   oder weil „Nicht-Rot-Rot-Regel“ für **`z,z.parent`** verletzt ist.*

*anderen Regeln: Schwarzhöhe und jeder Knoten rot oder schwarz

Gilt zu Beginn, da

1. neuer Knoten **`z`** zunächst auf rot gesetzt wird,
2. Wurzel im Baum zu Beginn schwarz ist,
3. „Schwarzhöhen“-Regel und „Rot-oder-Schwarz“-Regel
   von neuem roten Knoten **`z`** nicht verletzt wird,
   und alle anderen Knoten „Nicht-Rot-Rot“-Regel erfüllen

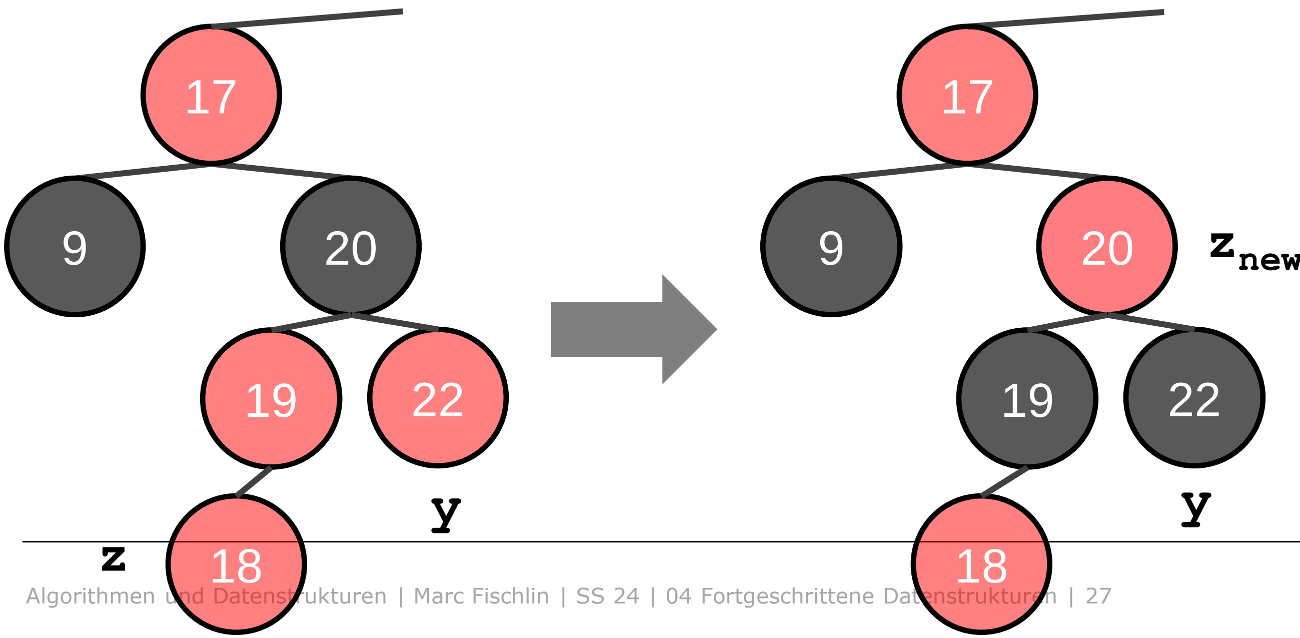
Aufräumen: Schleifeni

`y!=nil&y.color==red`

```
      …
4     IF y!=nil AND y.color==red THEN
5           z.parent.color=black;
6           y.color=black;
7           z.parent.parent.color=red;
8           z=z.parent.parent;
      …
```

Schleifeninvariante:

1. **`z.color==red`**
2. Wenn **`z.parent`** Wurzel, dann **`z.parent.color==black`**
3. Wenn der aktuelle Baum kein Rot-Schwarz-Baum ist, dann weil **`z`** als Wurzel die Farbe rot hat, oder weil „Nicht-Rot-Rot-Regel" für **`z,z.parent`** verletzt ist.



1. z_{new} wird wieder rot (7/8)

2. Farbe von z_{new}.parent ändert sich nicht

3. Schwarzhöhen-Regel bleibt erhalten und nur z_{new} wird rot

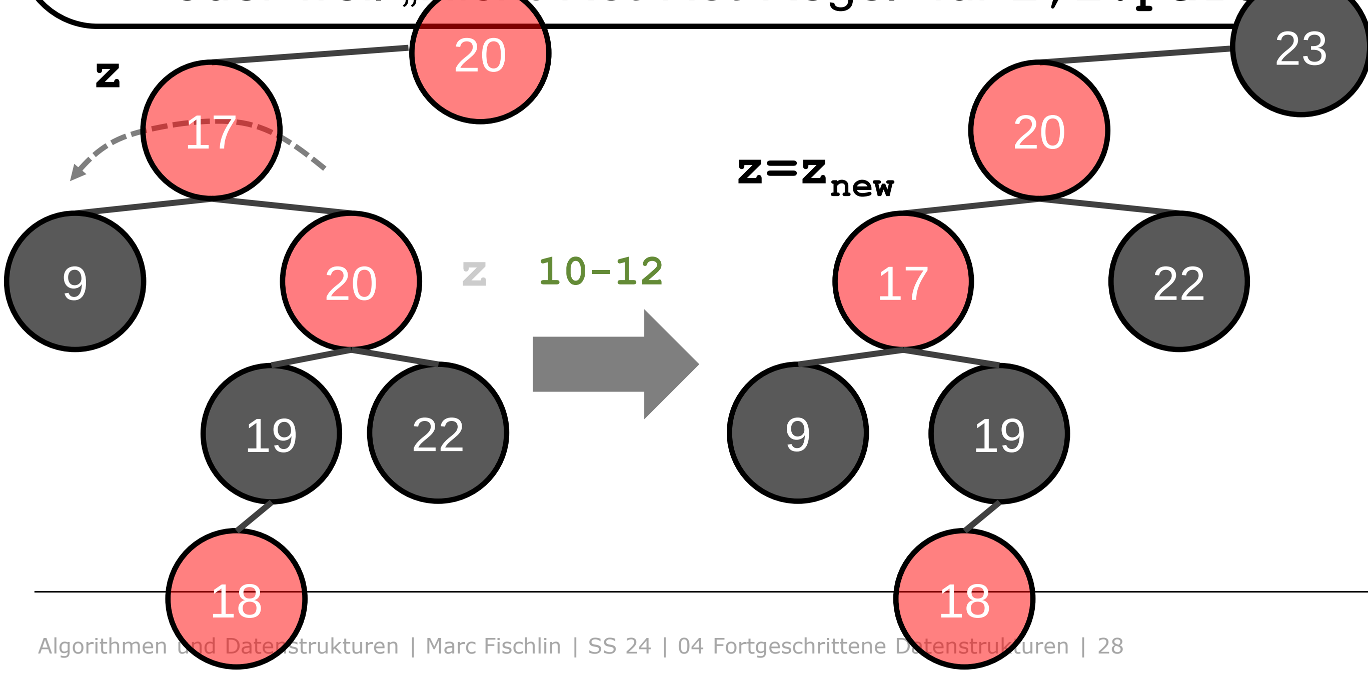
Aufräumen: Schleife[…]

y==nil|y.color==blac[…]

```
9      ELSE
10        IF z==z.parent.right THEN
11           z=z.parent;
12           rotateLeft(T,z);
13        z.parent.color=black;
14        z.parent.parent.color=red;
15        rotateRight(T,z.parent.parent);
```

Schleifeninvariante:

1. **`z.color==red`**
2. Wenn **`z.parent`** Wurzel, dann **`z.parent.color==black`**
3. Wenn der aktuelle Baum kein Rot-Schwarz-Baum ist, dann weil **`z`** als Wurzel die Farbe rot hat, oder weil „Nicht-Rot-Rot-Regel“ für **`z,z.parent`** verletzt ist.



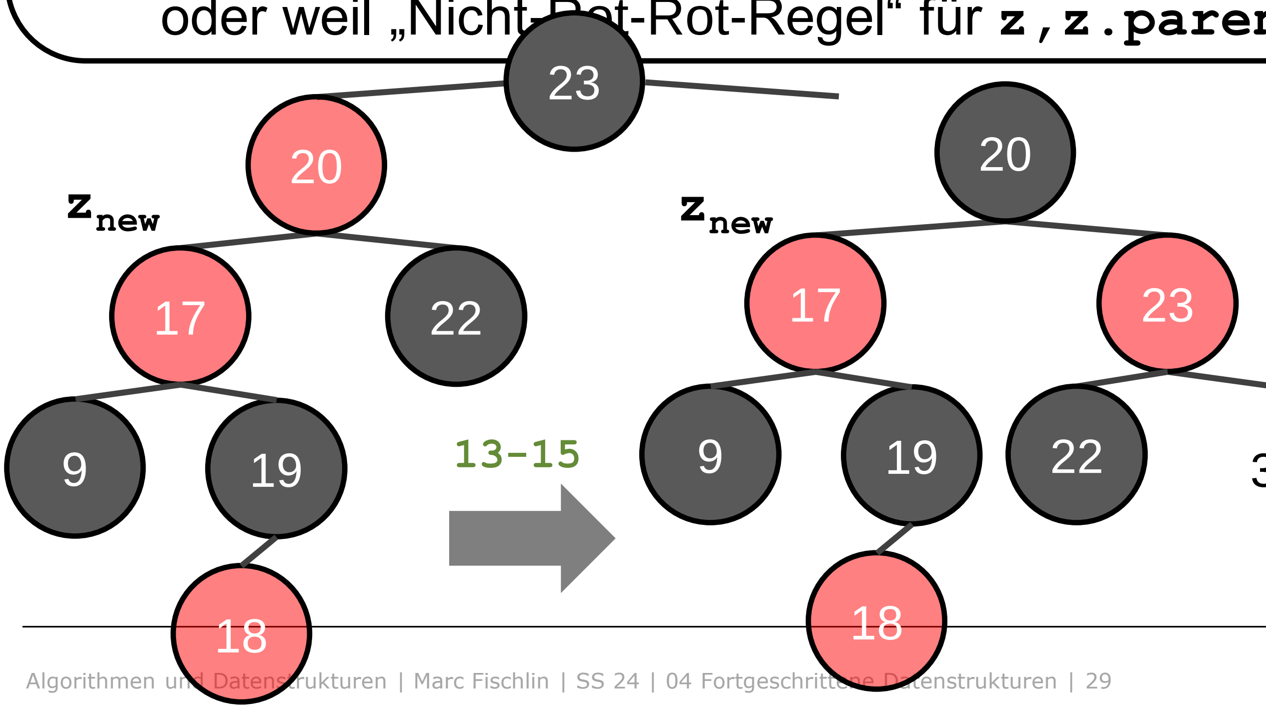
Aufräumen: Schleife…

`y==nil|y.color==blac…`

```
9        ELSE
10          IF z==z.parent.right THEN
11             z=z.parent;
12             rotateLeft(T,z);
13          z.parent.color=black;
14          z.parent.parent.color=red;
15          rotateRight(T,z.parent.parent);
```

Schleifeninvariante:

1. `z.color==red`
2. Wenn `z.parent` Wurzel, dann `z.parent.color==black`
3. Wenn der aktuelle Baum kein Rot-Schwarz-Baum ist, dann weil `z` als Wurzel die Farbe rot hat, oder weil „Nicht-Rot-Rot-Regel" für `z,z.parent` verletzt ist.



1. z_{new} wird wieder rot, da `z.parent.color==red` in `WHILE`-Schleife
2. Farbe von z_{new}`.parent` wegen 13 schwarz
3. SH-Regel erhalten und neues rotes z_{new} wird Kind schwarzes Knotens

Aufräumen: Terminierung

Terminierung ⇒ Korrektheit

Schleifeninvariante:

1. **`z.color==red`**
2. Wenn **`z.parent`** Wurzel, dann **`z.parent.color==black`**
3. Wenn der aktuelle Baum kein Rot-Schwarz-Baum ist, dann weil **`z`** als Wurzel die Farbe rot hat, oder weil „Nicht-Rot-Rot-Regel“ für **`z,z.parent`** verletzt ist.

```
fixColorsAfterInsertion(T,z)
1  WHILE z.parent.color==red DO
     …
18 T.root.color=black;
```

Wenn **`WHILE`**-Schleife terminiert, dann weil **`z.parent.color==black`**.

Wenn kein RB-Baum, dann wäre **`z`** als Wurzel rot, aber **`18`** setzt schwarz (und dies kann andere Rot-Schwarz-Baum-Bedingungen nicht verletzen).

Aufräumen: Laufzeit

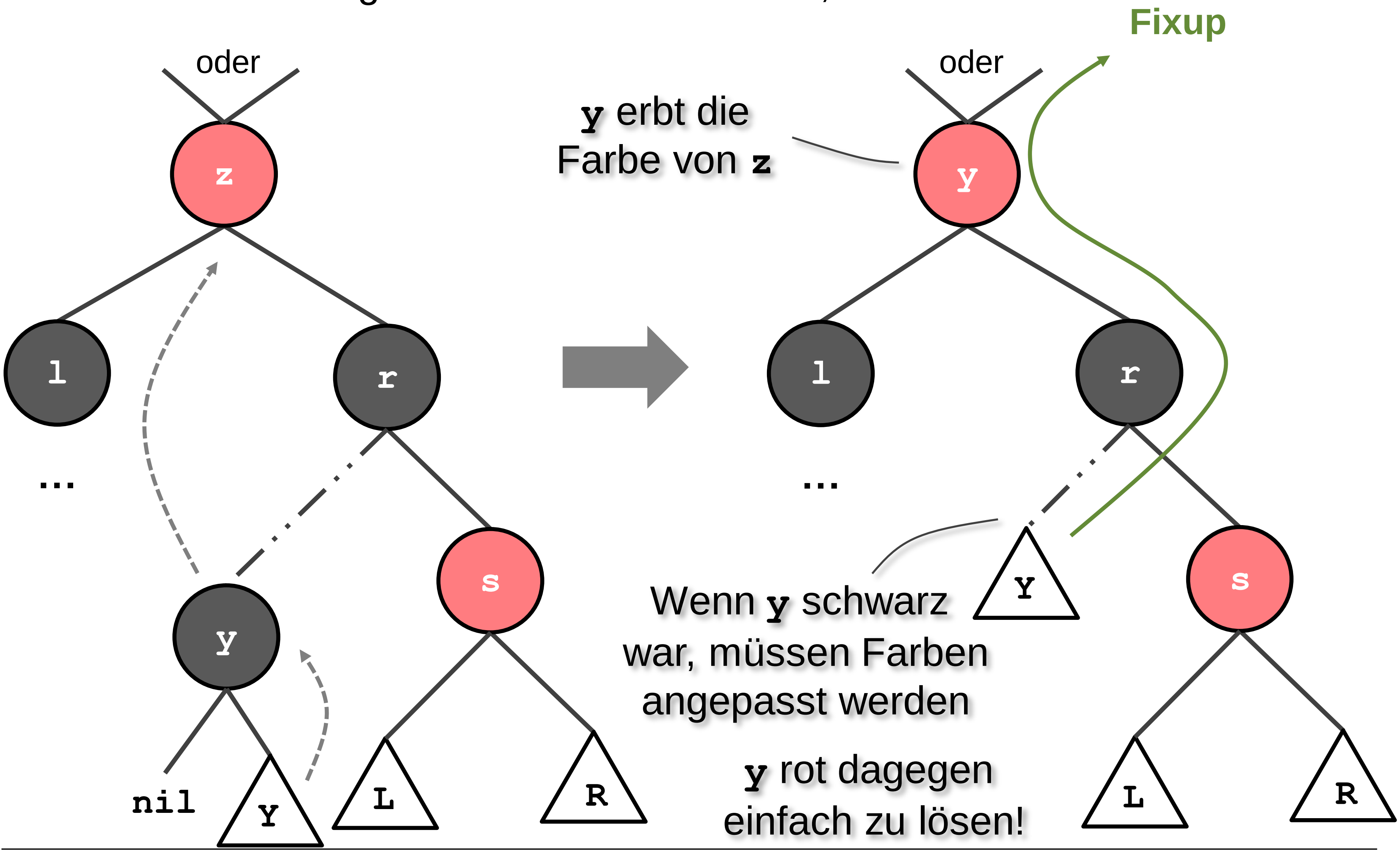
Entweder **`z`** geht zwei Level nach oben, oder **`WHILE`**-Schleife terminiert

maximal $O(h)$ viele Iterationen mit jeweils konstanter Laufzeit

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

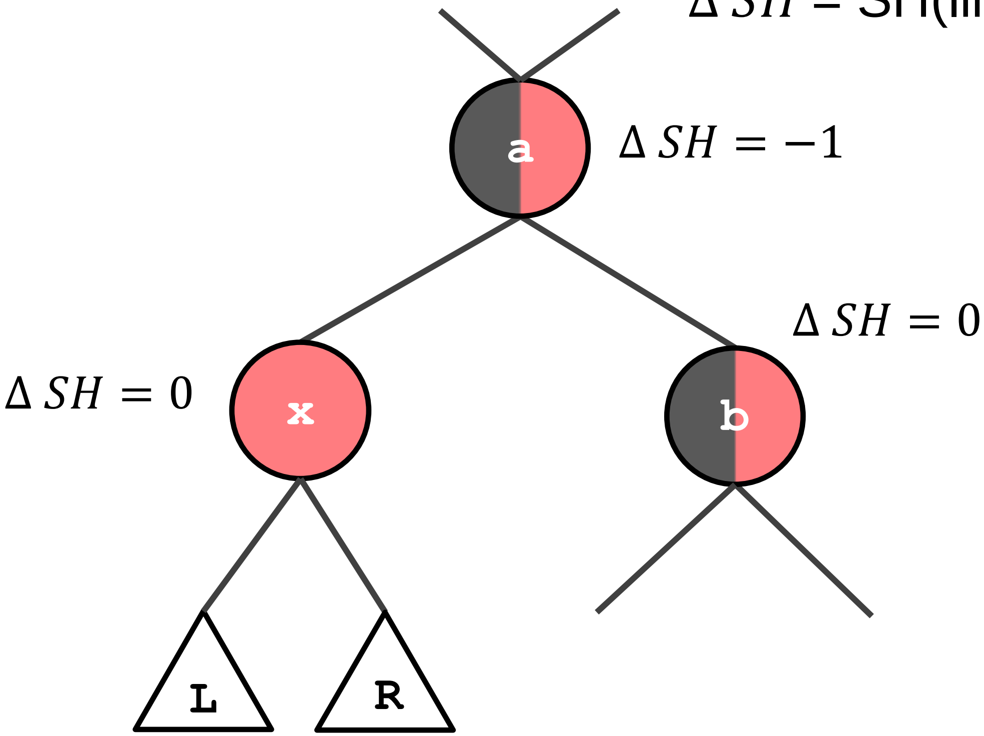
```
fixColorsAfterInsertion(T,z)

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
     … //exchange left and right
T.root.color=black;
```

Löschen analog zum binären Suchbaum, aber:



Löschen: Fixup bei `y.color==black`

In jedem Knoten:
$\Delta\, SH$ = SH(linker Teilbaum)–SH(rechter Teilbaum)



Beispiel:
(rechtslastige)
Ungleichheit
in Knoten **`a`** fixen.

Anderer Fall $\Delta SH = +1$ analog.

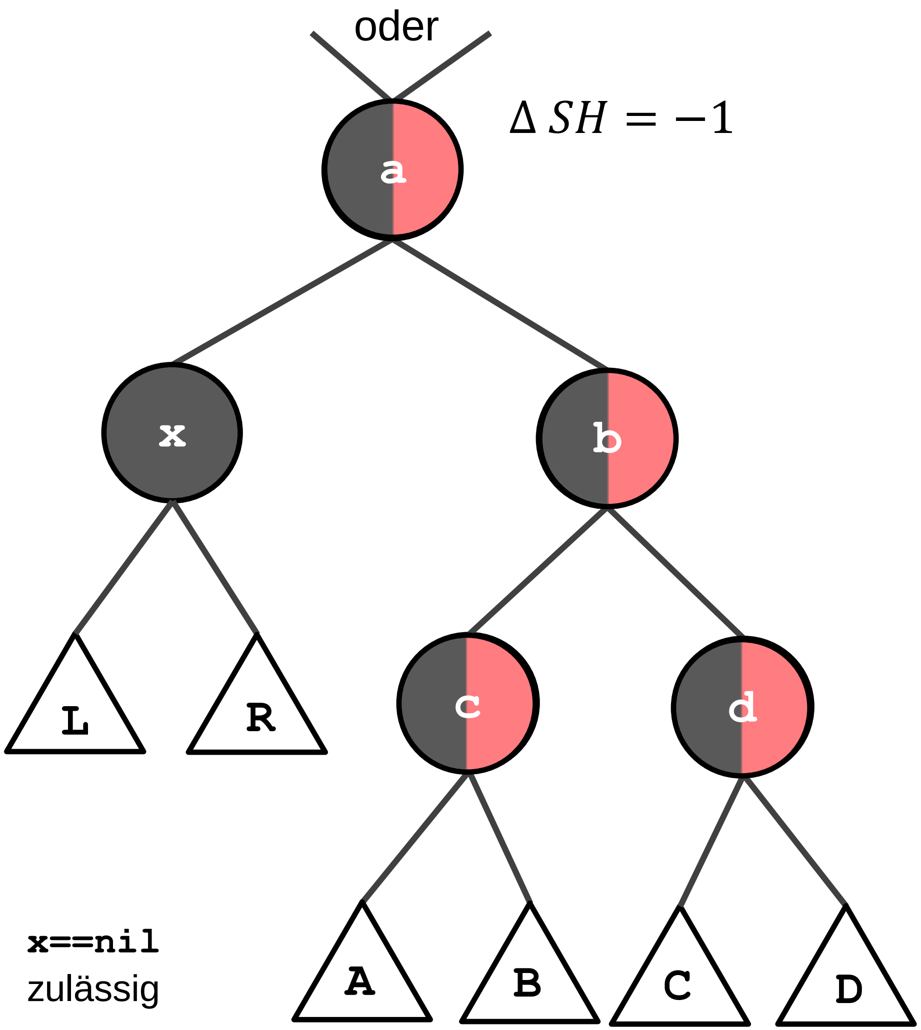
Fallunterscheidung nach Farbkombination der Knoten.

Beachte: Beim Löschen kann SH in Knoten nur sinken, also war ursprüngliche SH in **`a`** um 1 höher

Wenn **`x`** rot, dann schwarz setzen; daher nur Fall **`x`** schwarz zu betrachten.

Löschen: Fixup Fallunterscheidung

Algorithmen im Anhang



`x==nil`
zulässig

Fall I: `a` schwarz, `b` rot

Fall IIa: `a` rot, `b` schwarz,
`c,d` nicht rot

Fall IIb: `a` schwarz, `b` schwarz,
`c,d` nicht rot

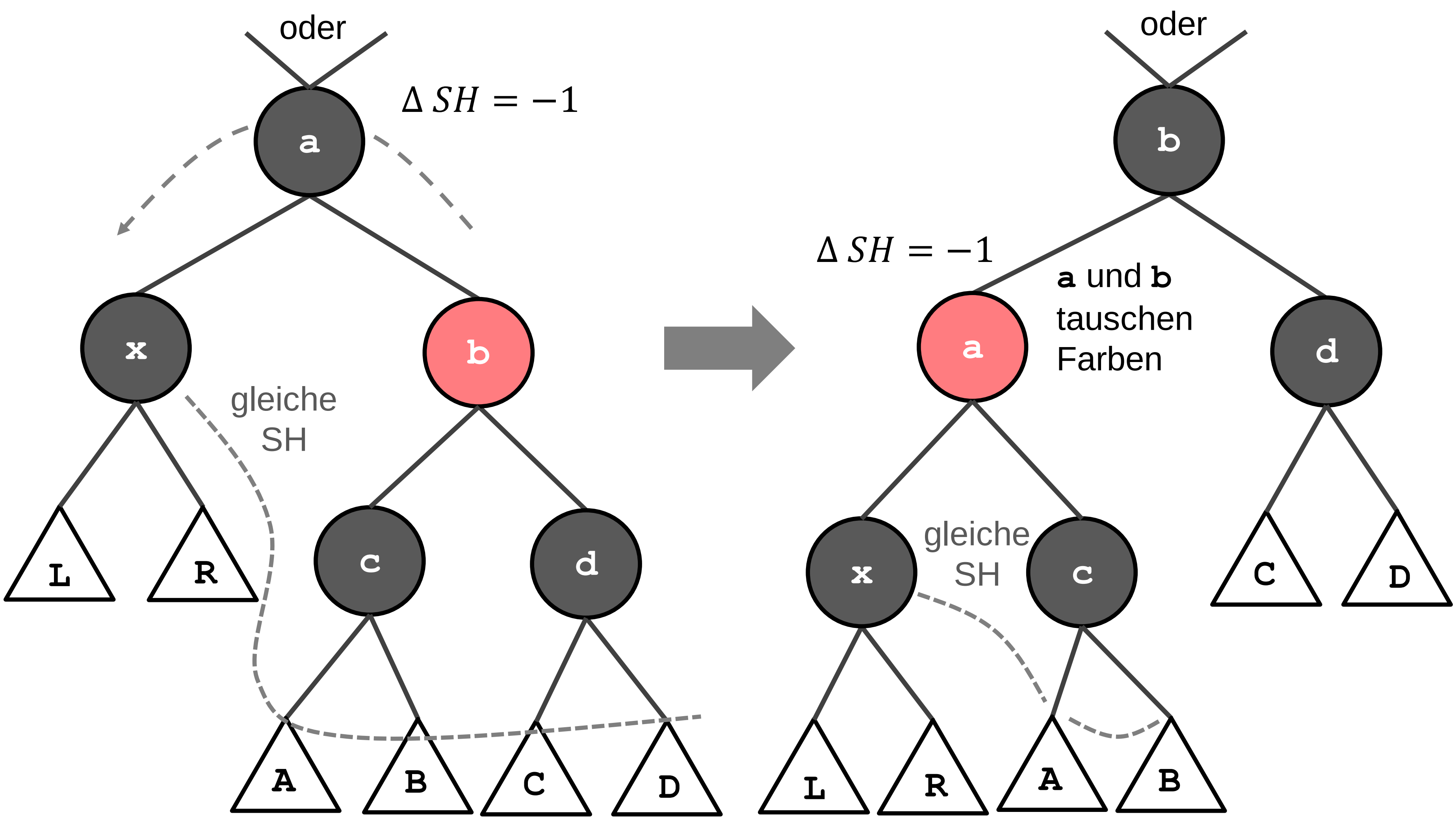
Fall III: `a` beliebig, `b` schwarz,
`c` rot, `d` nicht rot

Fall IV: `a` beliebig, `b` schwarz,
`c` beliebig, `d` rot

nicht rot = schwarz oder `nil`

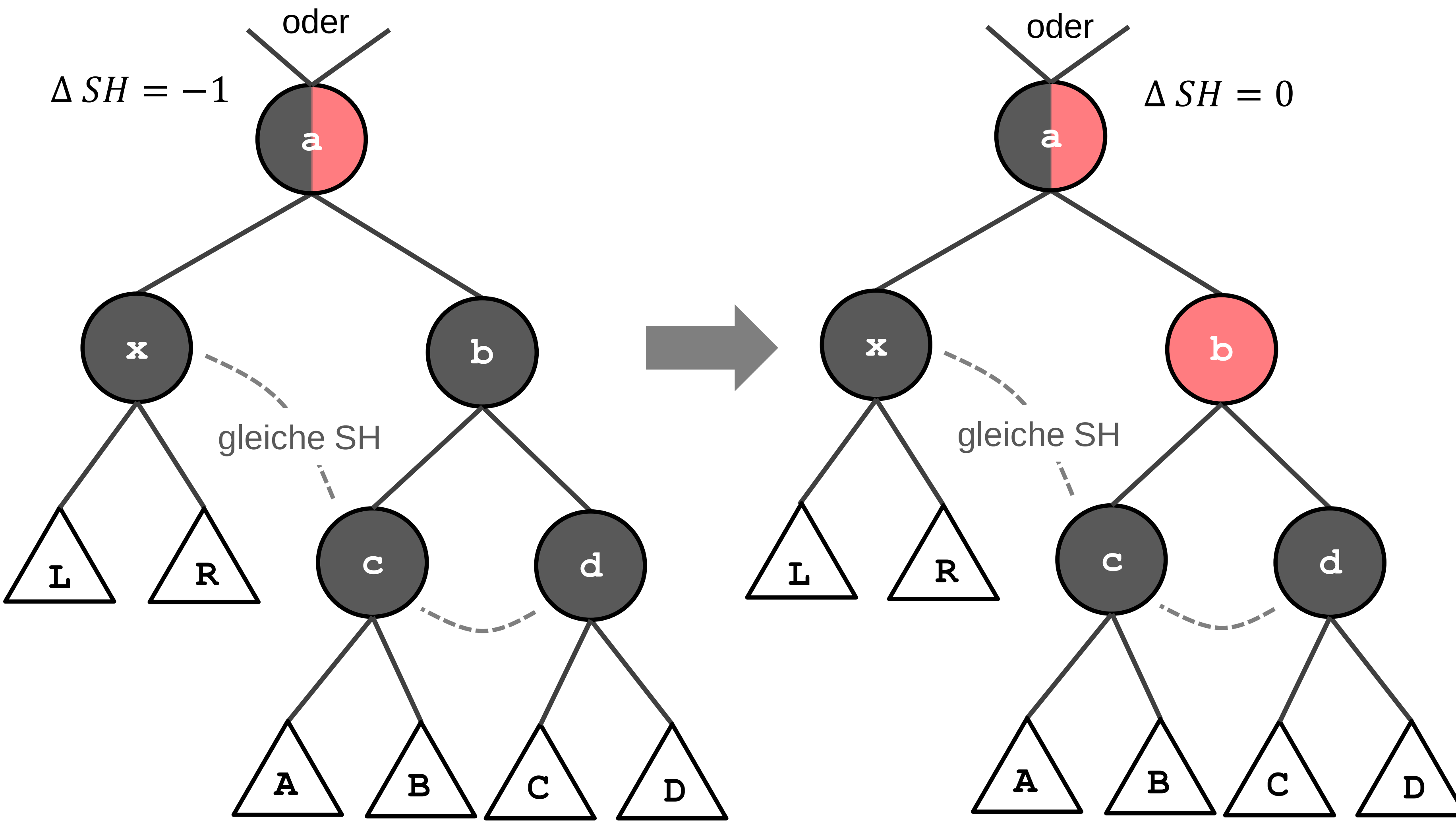
Löschen: Fixup (Fall I)

wird zu Fall IIa, III oder IV;
nie zu Fall IIb



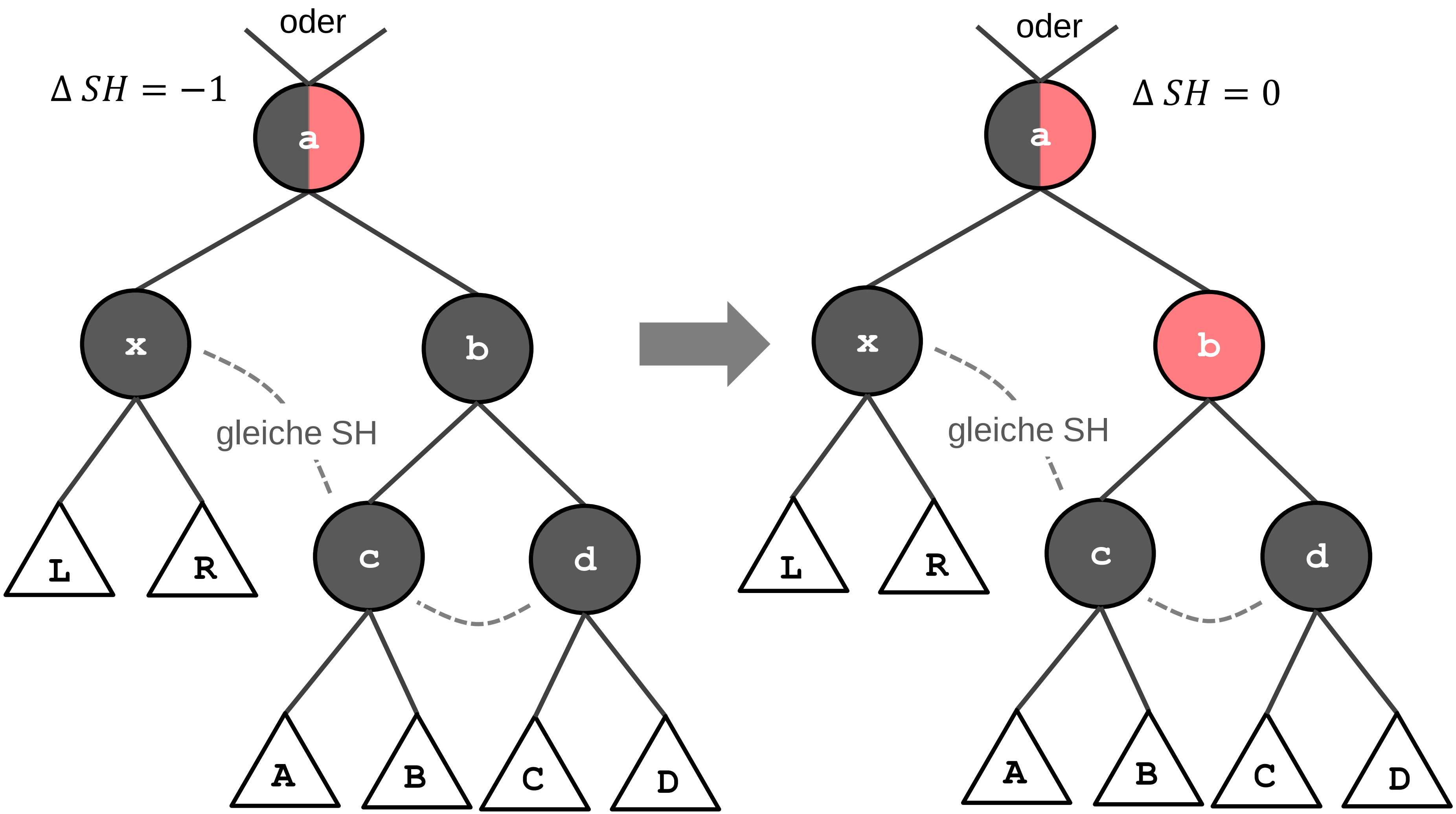
Löschen: Fixup (Fall IIa)

Wenn **a** rot, dann auf schwarz setzen und damit ursprüngliche SH erreicht



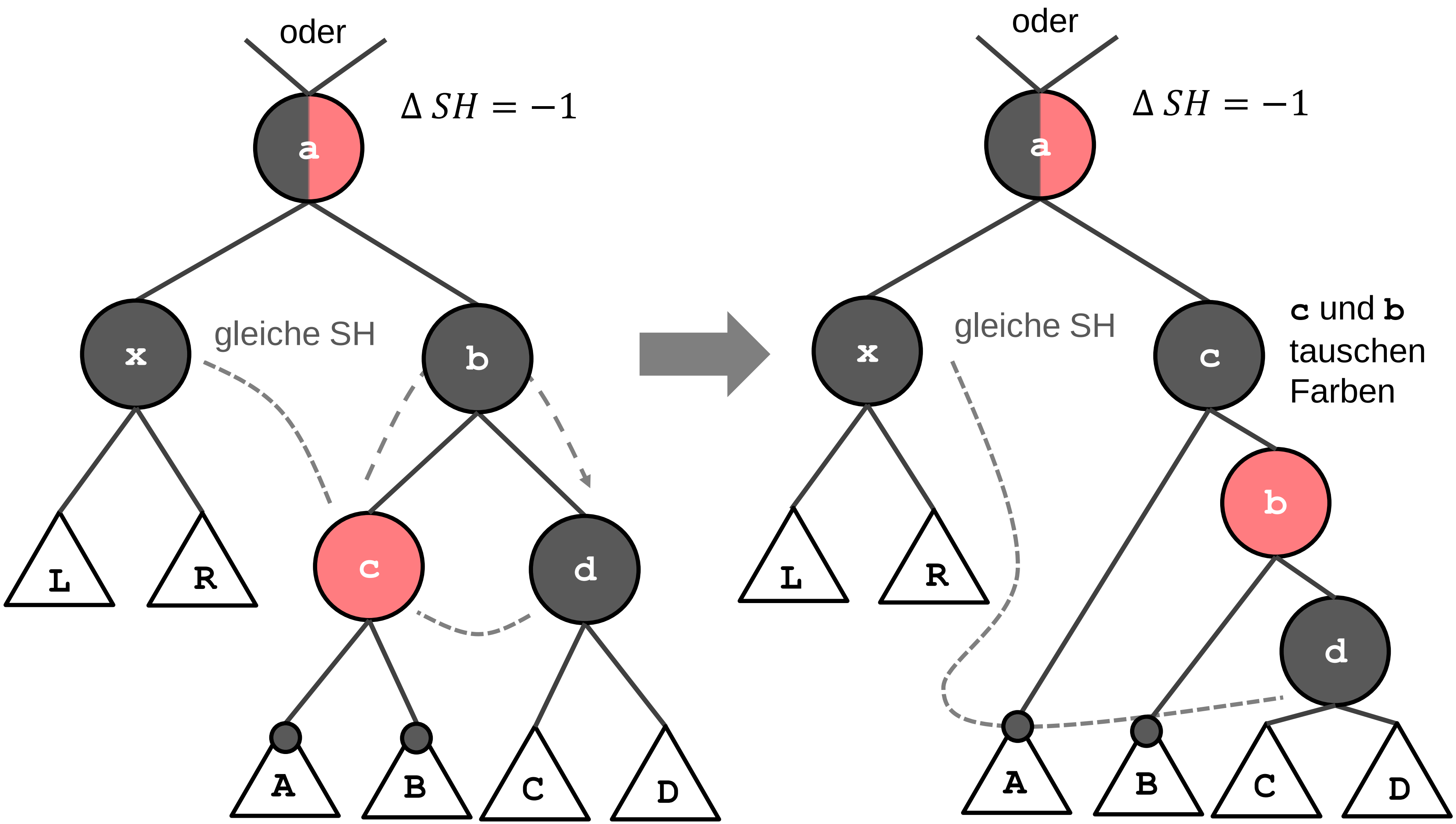
Löschen: Fixup (Fall IIb)

Wenn **a** schwarz, dann Vaterknoten $\Delta\, SH = \pm 1$;
verfahre rekursiv mit **a** als neuem **x**

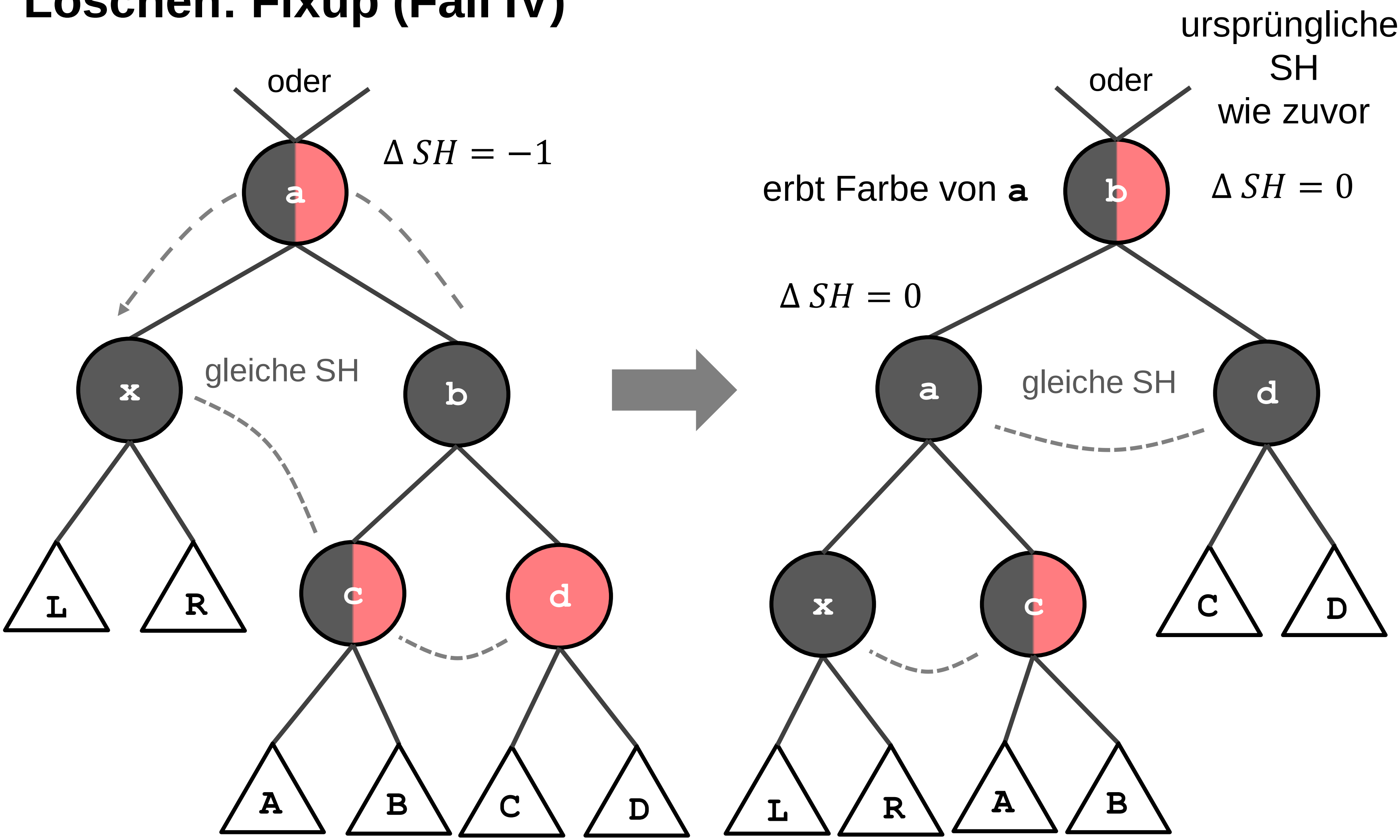


Löschen: Fixup (Fall III)

wird zu Fall IV



Löschen: Fixup (Fall IV)



Löschen: Laufzeiten

`y` suchen hat (wie beim binären Suchbaum) Laufzeit $O(h) = O(\log n)$

Fixup geht nur in Fall IIb in Rekursion, dann aber einen Level höher

(In Fall I wird **`a`** zwar einen Level tiefer rotiert, aber dann bricht Rekursion nach Fällen IIa, III oder IV danach ab)

In jeder Rekursion konstante Laufzeit, also Gesamtlaufzeit $O(h) = O(\log n)$

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

Worst-Case-Laufzeiten

Rot-Schwarz-Bäume

Operation	Laufzeit
Einfügen	$\Theta(\log n)$
Löschen	$\Theta(\log n)$
Suchen	$\Theta(\log n)$

Wie vereinigen Sie zwei Rot-Schwarz-Bäume `T0` und `T1` mit gleicher Schwarzhöhe, wenn `x0.key<=x1.key` für alle Knoten `x0` in `T0` und `x1` in `T1`?

Was ist, wenn die beiden Bäume nicht die gleiche Schwarzhöhe haben?

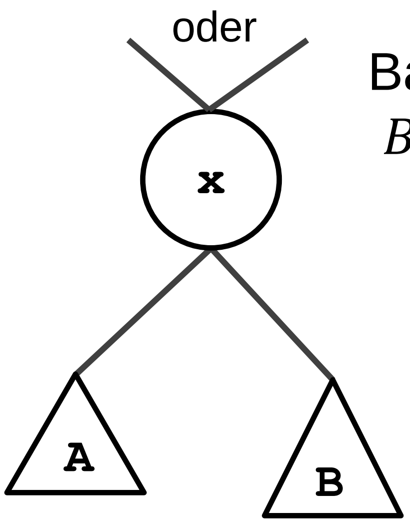
AVL-Bäume

Georgi Maximowitsch **A**delson-**V**elski und Jewgeni Michailowitsch **L**andis

„Optimierte Konstanten“:

$h \leq 2 \cdot \log n$ (Rot-Schwarz-Bäume) vs. $h \leq 1.441 \cdot \log n$ (AVL-Bäume)

AVL-Bäume



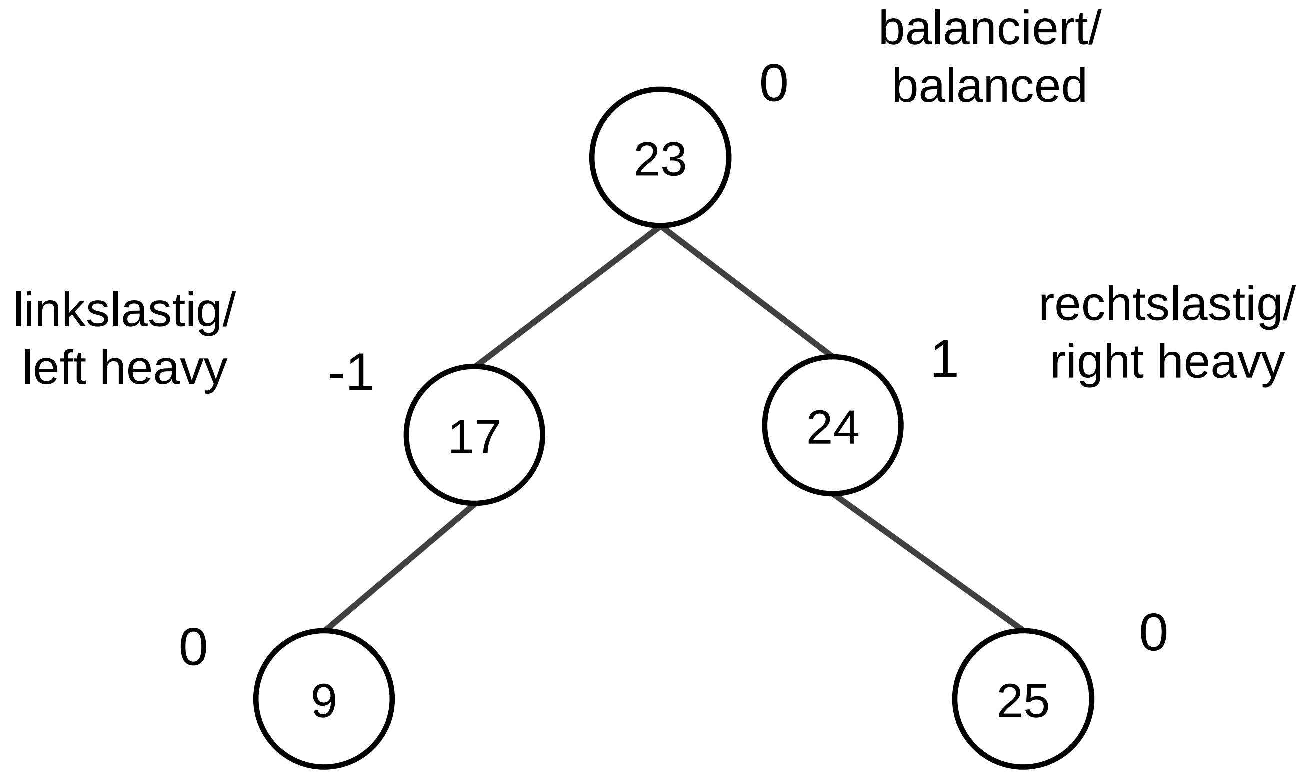
Balance in Knoten x:

$B(x) = Höhe(rechter\ Teilbaum) - Höhe(linker\ Teilbaum)$

Ein **AVL-Baum** ist ein binärer Suchbaum, so dass für die Balance $B(x)$ in jede Knoten x gilt: $B(x) \in \{-1, 0, +1\}$.

Konvention: $Höhe(leerer\ Baum) = -1$

Beispiel: AVL-Baum



Höhe AVL-Bäume (I)

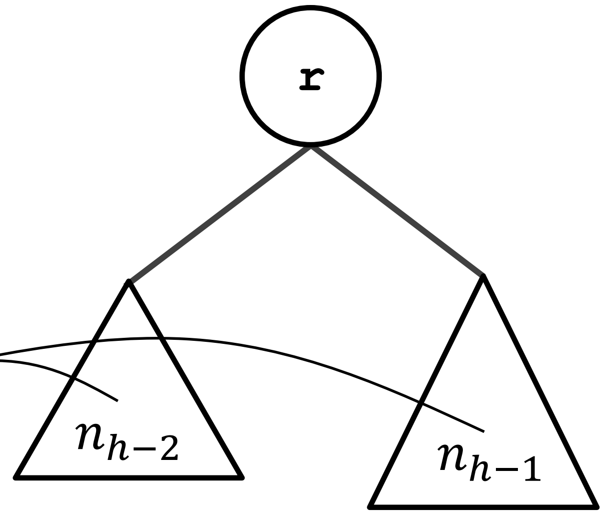
Ein AVL-Baum mit n Knoten hat maximale Höhe $h \leq 1{,}441 \cdot \log_2 n$.

Beweisidee:

Sei n_h minimale Anzahl von Knoten in einem AVL-Baum der Höhe h.

Dann: $n_0 = 1, n_1 = 2, n_2 = 4, \ldots$

Allgemein: $n_h = 1 + n_{h-1} + n_{h-2}$



wegen Balance-Differenz von 0,±1

Höhe AVL-Bäume (II)

Ein AVL-Baum mit n Knoten hat maximale Höhe $h \leq 1{,}441 \cdot \log_2 n$.

Fibonacci-Zahlen:

$F_0 = 1, \quad F_1 = 1,$

$F_n = F_{n-1} + F_{n-2}$

h	0	1	2	3	4	5	6
F_h	1	1	2	3	5	8	13
n_h	1	2	4	7	12	20	33

Dann: $n_0 = 1, n_1 = 2, n_2 = 4, \ldots$

$$F_h + 1 \approx \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+3}$$

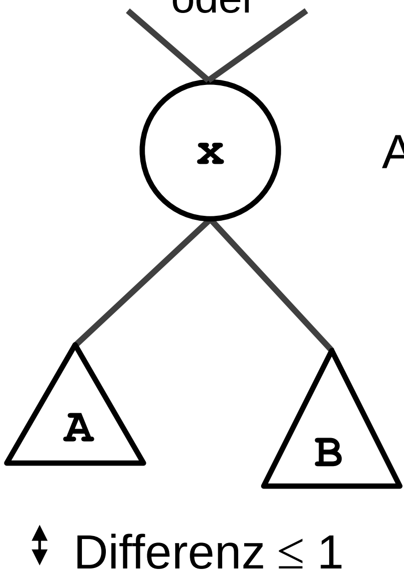
Allgemein: $n_h = 1 + n_{h-1} + n_{h-2}$

Folglich: $n_h = F_{h+2} - 1$

$h \approx 1{,}441 \cdot \log_2 n.$ ■

AVL vs. Rot-Schwarz

oder



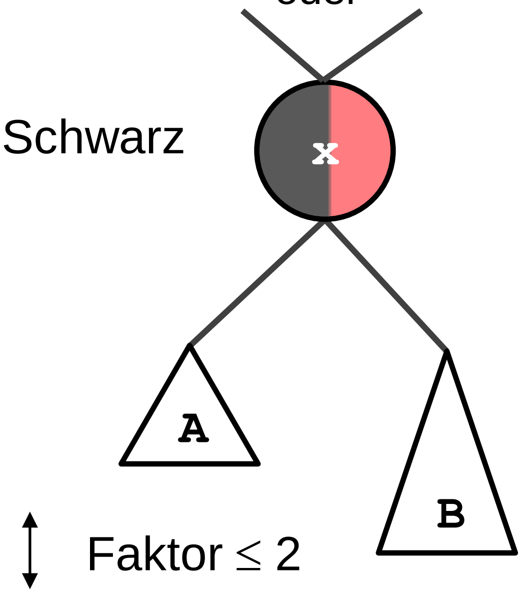
AVL

Differenz ≤ 1

Einfügen und Löschen verletzen
in der Regel öfter die Baum-Bedingung,
mehr Aufwand zum Rebalancieren

oder

Rot-Schwarz



Faktor ≤ 2

Suchen dauert evtl. länger

AVL-Bäume geeigneter, wenn mehr Such-Operationen
und weniger Einfüge- und Lösch-Operationen

AVL $\subset$ Rot-Schwarz (I)

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

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

Beweis per Induktion:

Gilt für Ein-Knoten-Baum mit schwarzer oder roter Wurzel ($h = 0$).

$$SH = 1 = \left\lceil \frac{h+1}{2} \right\rceil$$

$$SH = 0 = \frac{h}{2}$$

AVL $\subset$ Rot-Schwarz (II)

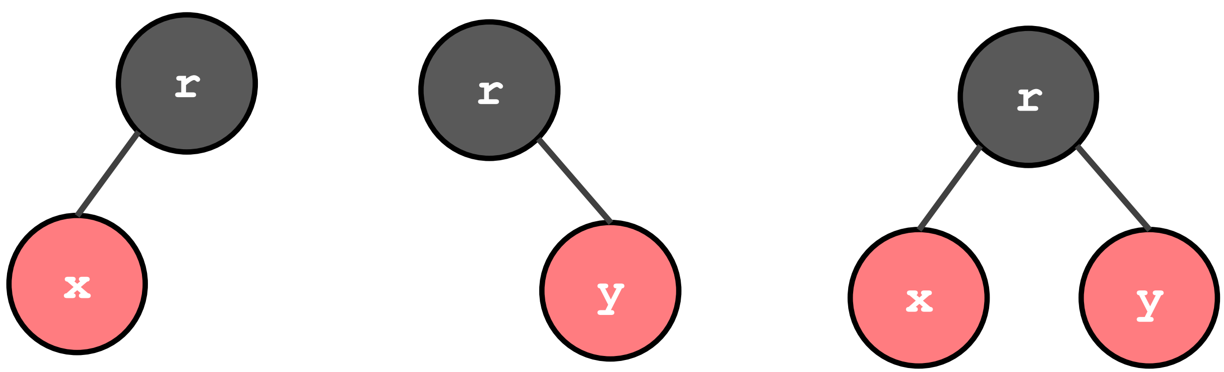
Jeder nicht-leere AVL-Baum der Höhe h lässt sich als Rot-Schwarz-Baum mit Schwarzhöhe $\left\lceil\frac{h+1}{2}\right\rceil$ darstellen.

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

Beweis per Induktion:

Gilt für Baum mit schwarzer Wurzel ($h = 1$).

$$SH = 1 = \left\lceil\frac{h+1}{2}\right\rceil$$



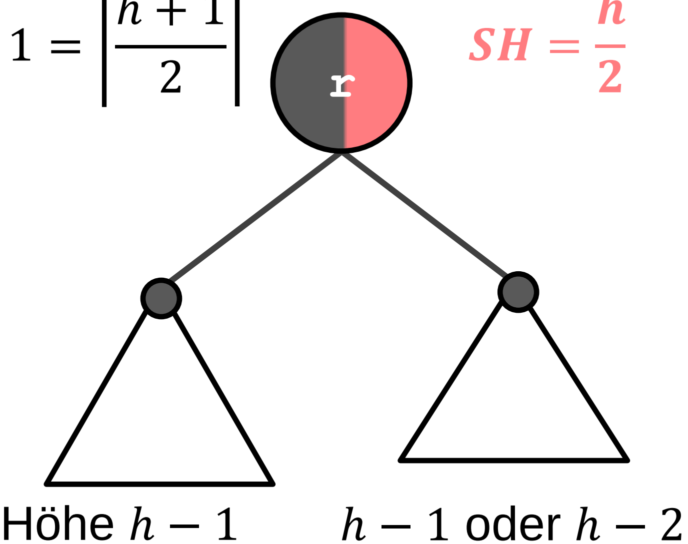
AVL ⊂ Rot-Schwarz (III)

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

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

Induktionsschritt: $h \geq 2$ gerade

$$SH = \frac{h}{2} + 1 = \left\lceil\frac{h+1}{2}\right\rceil$$



Wähle für linken Teilbaum RS-Baum mit schwarzer Wurzel mit $\mathrm{SH} = \left\lceil\frac{(h-1)+1}{2}\right\rceil = \frac{h}{2}$

Wähle für rechten Teilbaum RS-Baum mit schwarzer Wurzel mit $\mathrm{SH} = \frac{h}{2}$ für Höhe $h-1$ bzw. mit $\mathrm{SH} = \left\lceil\frac{(h-2)+1}{2}\right\rceil = \frac{h}{2}$ für Höhe $h-2$

(umgekehrter Fall analog)

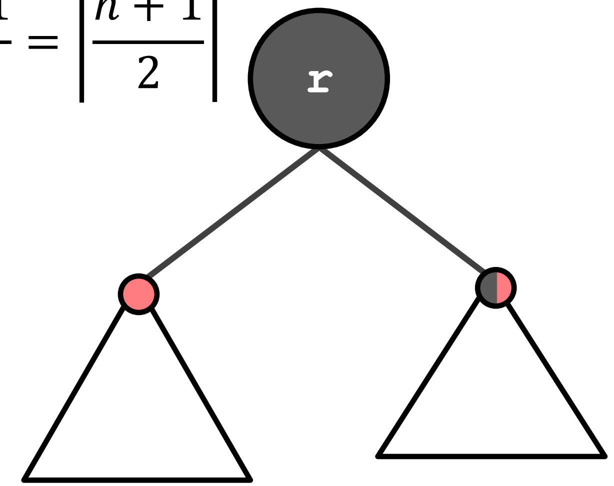
AVL ⊂ Rot-Schwarz (IV)

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

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

Induktionsschritt: $h \geq 3$ ungerade

$$SH = \frac{h+1}{2} = \left\lceil \frac{h+1}{2} \right\rceil$$



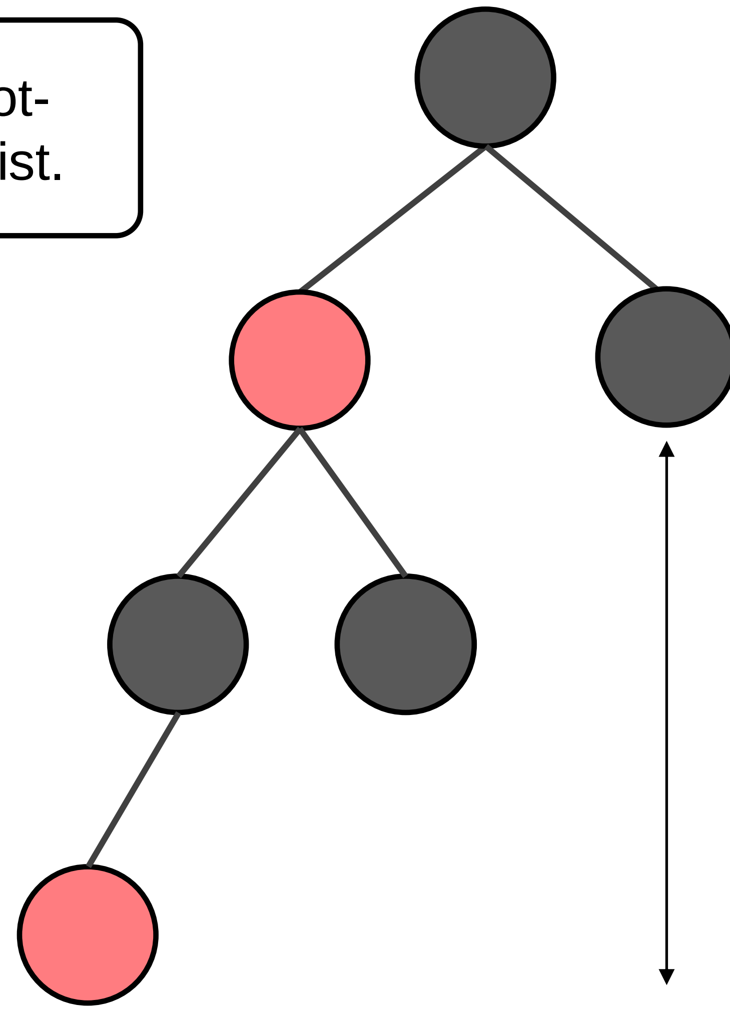
(umgekehrter Fall analog)

Wähle für linken Teilbaum RS-Baum mit roter Wurzel mit $\mathrm{SH} = \frac{h-1}{2}$

Wähle für rechten Teilbaum RS-Baum mit roter Wurzel mit $\mathrm{SH} = \frac{h-1}{2}$ für Höhe $h-1$ bzw. mit schwarzer Wurzel mit $\mathrm{SH} = \left\lceil \frac{h+1-2}{2} \right\rceil = \frac{h-1}{2}$ für Höhe $h-2$ ■

AVL $\neq$ Rot-Schwarz

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



Höhen
unter-
schied
= 2

Für größere h verwende zweimal diesen Teilbaum und hänge beide Teilbäume an eine neue (schwarze) Wurzel

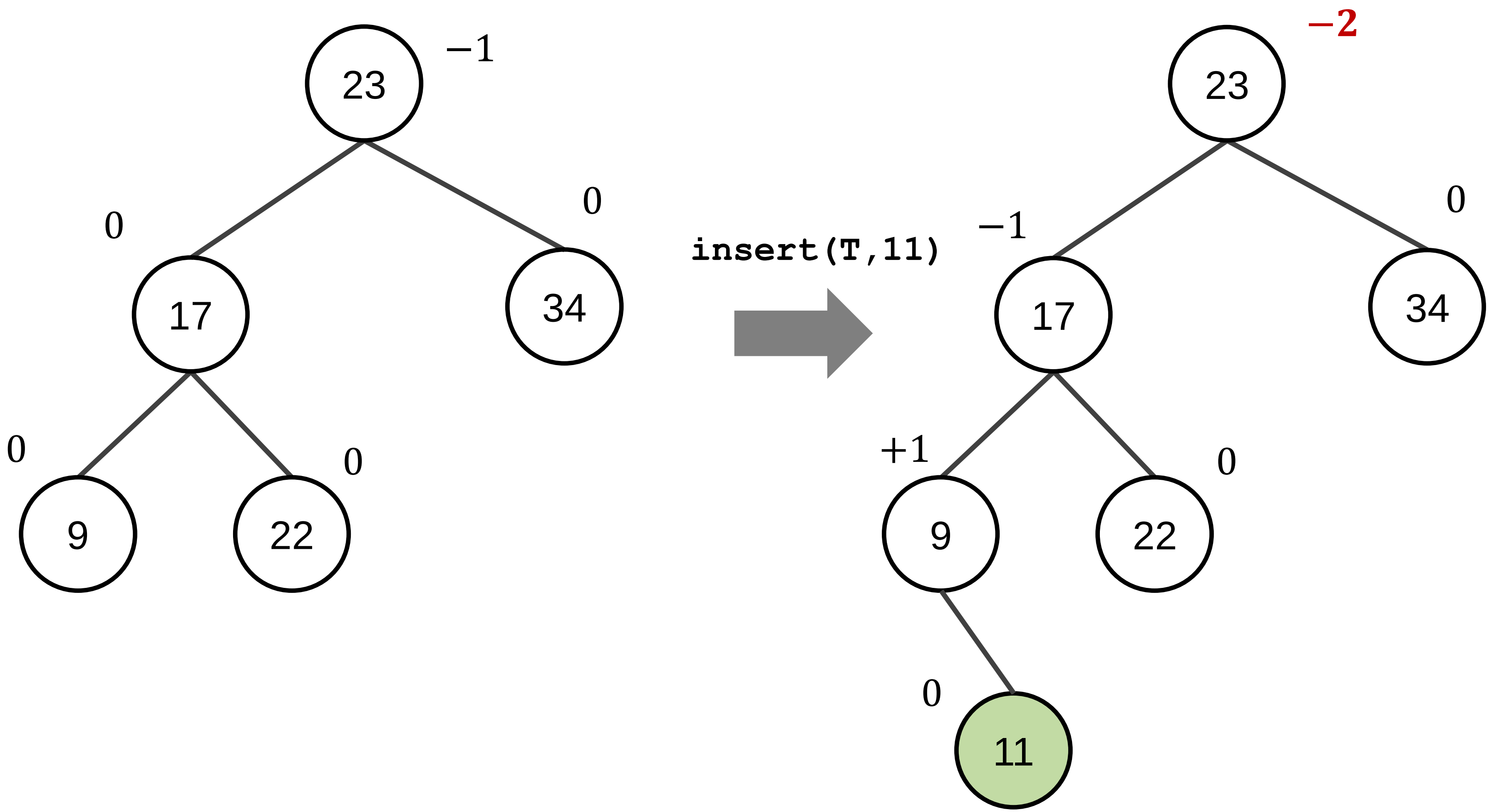
Einfügen

Funktioniert wie beim binären Suchbaum (mit Sentinel)

```
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);
```

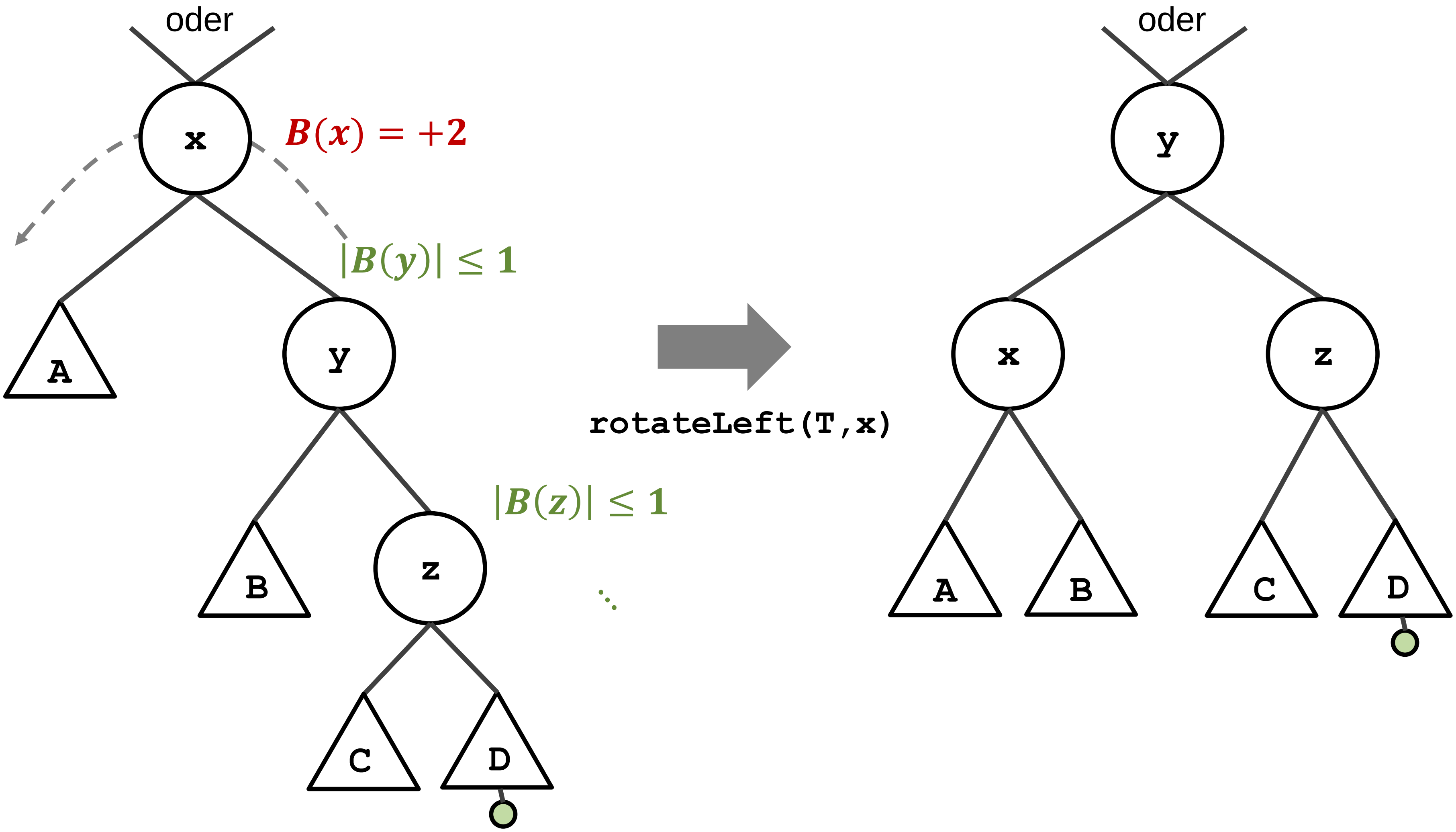
Änderung: evtl. Rebalancieren

Einfügen: Beispiel

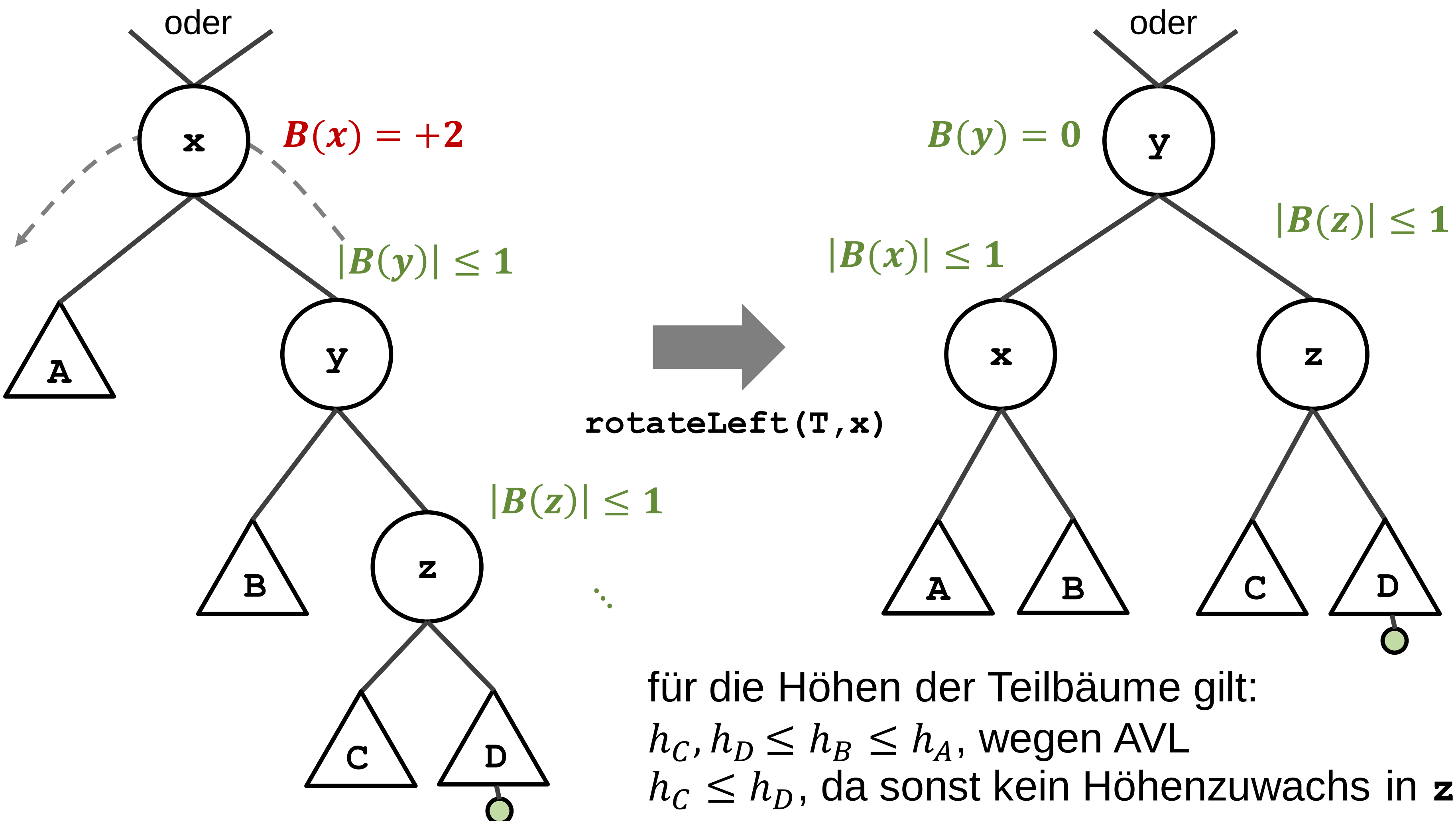


erfordert Rebalancieren,
evtl. weiter oben im Baum

Rebalancieren: Fall I



Rebalancieren: Fall I (Analyse I)



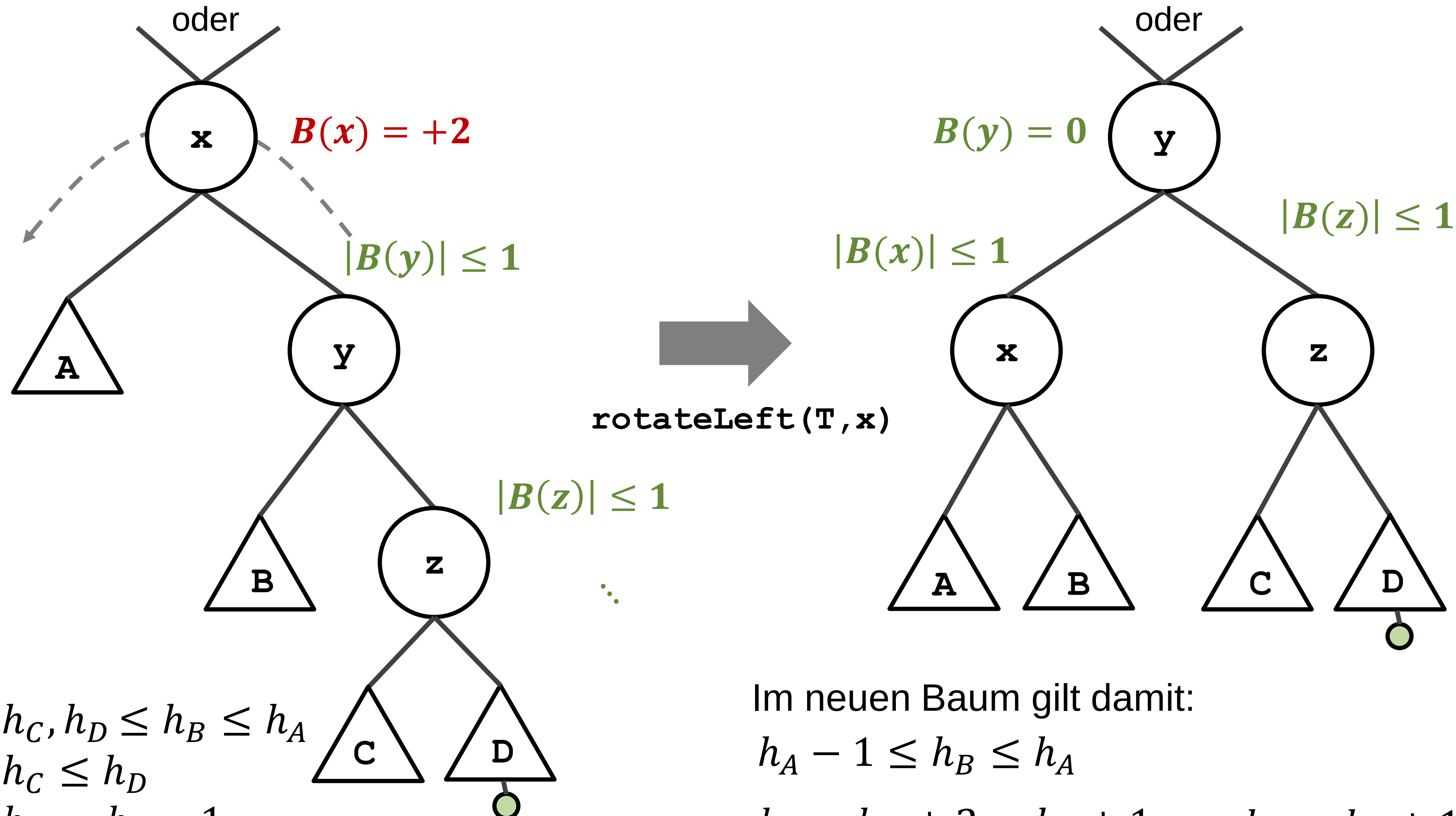
für die Höhen der Teilbäume gilt:

$h_C, h_D \leq h_B \leq h_A$, wegen AVL

$h_C \leq h_D$, da sonst kein Höhenzuwachs in **z**

$h_D = h_A - 1$, da nur dann $B(x) = +2$

Rebalancieren: Fall I (Analyse II)



$h_C, h_D \leq h_B \leq h_A$

$h_C \leq h_D$

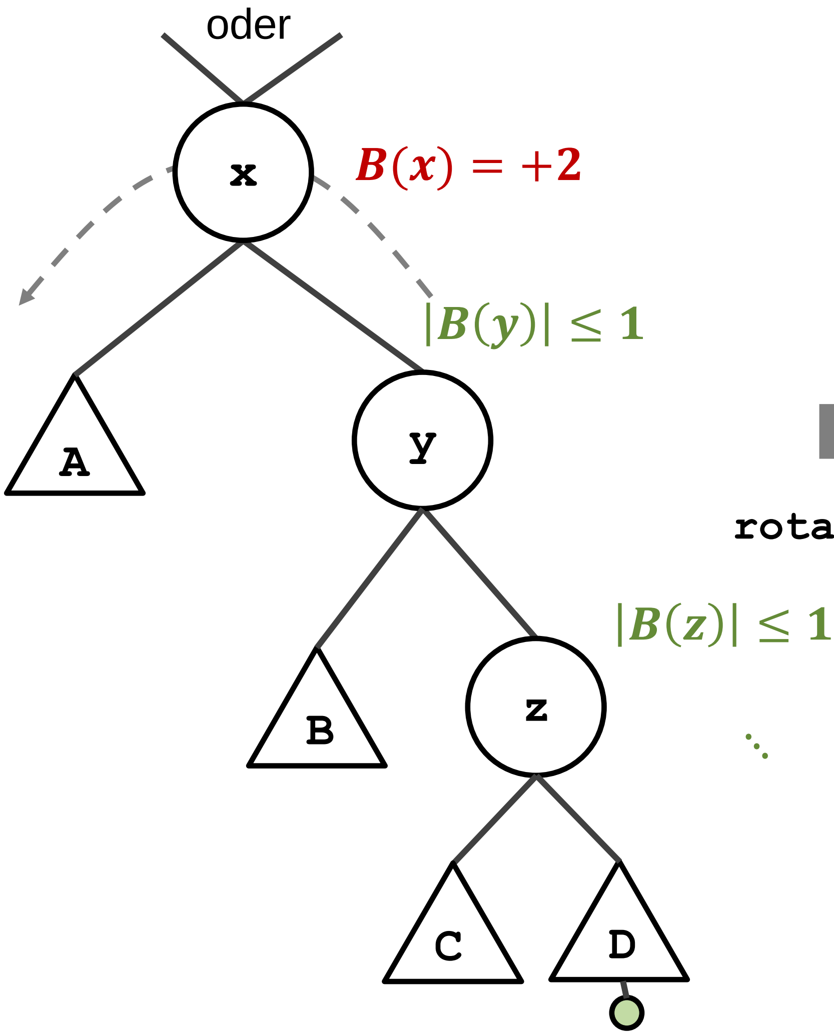
$h_D = h_A - 1$

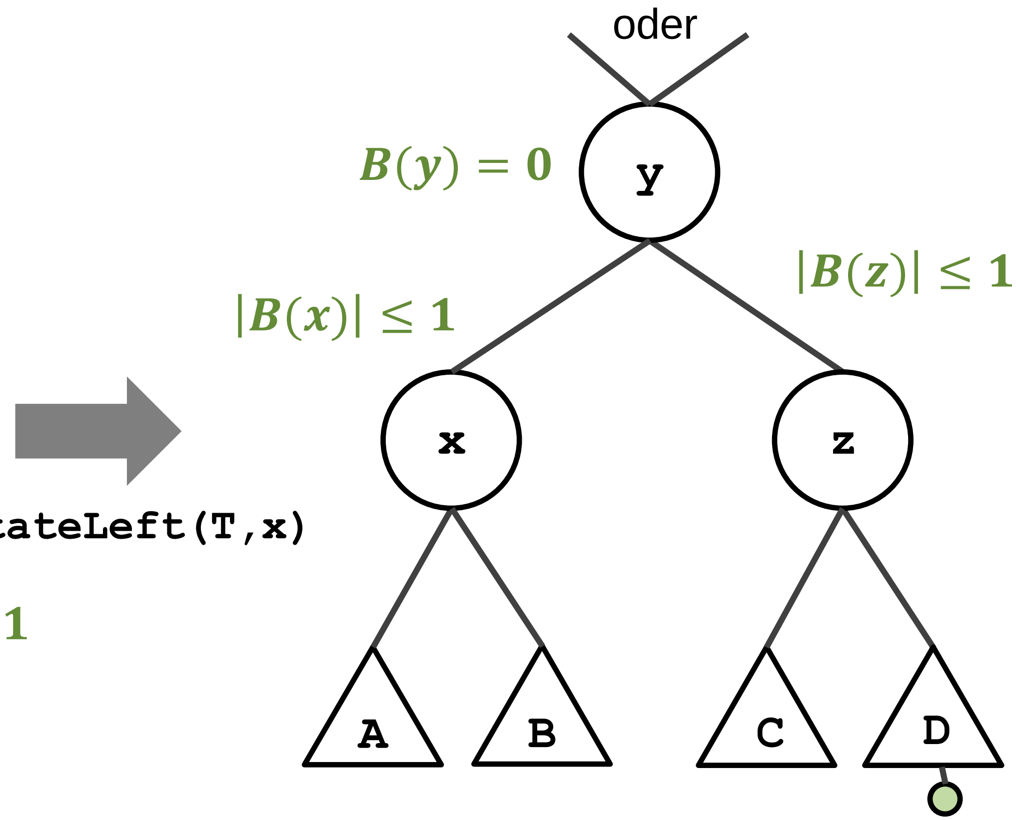
Im neuen Baum gilt damit:

$h_A - 1 \leq h_B \leq h_A$

$h_z = h_D + 2 = h_A + 1 \qquad h_x = h_A + 1$

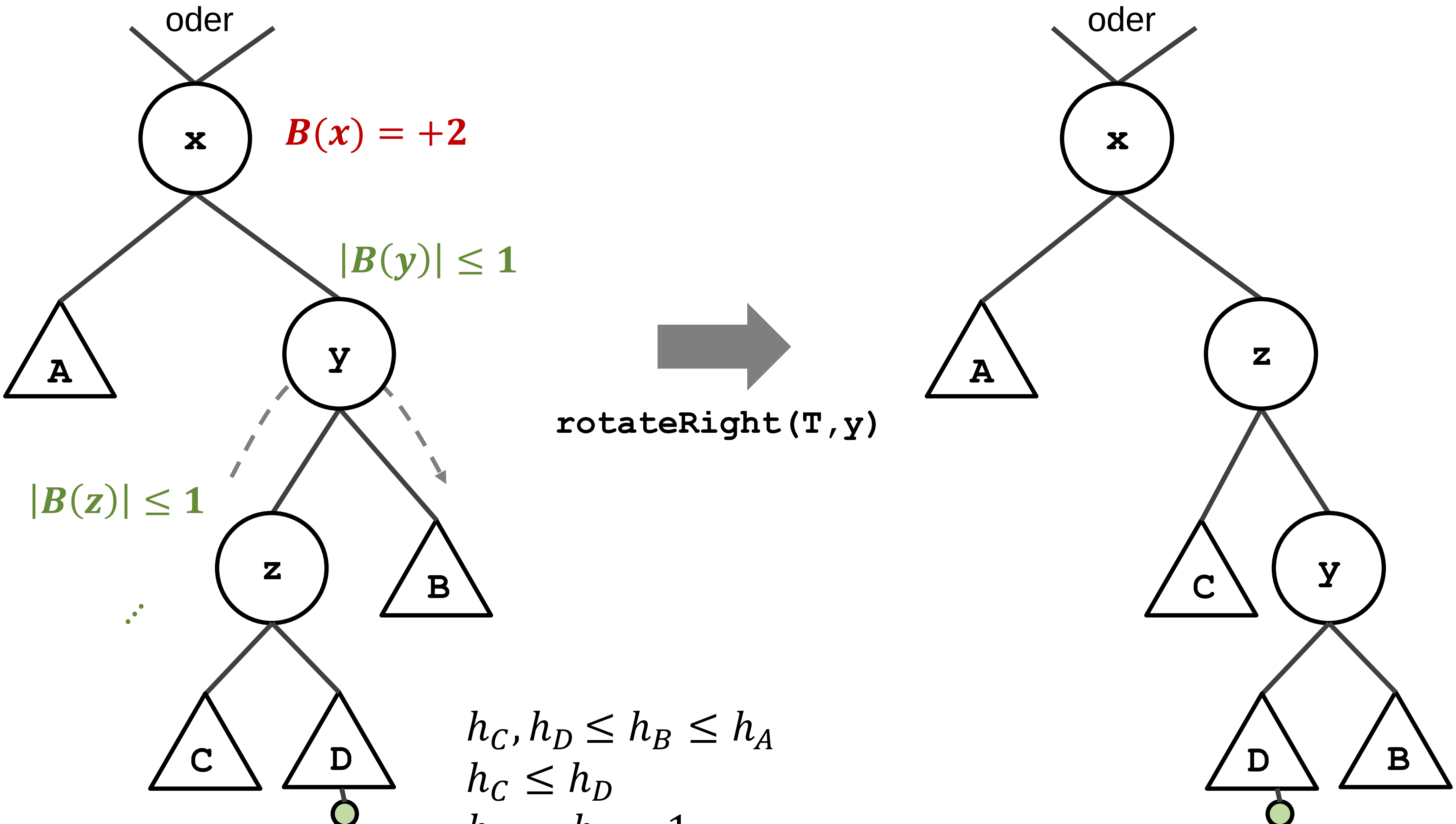
Rebalancieren: Fall I (Analyse III)





Höhe(nach Einfügen und Rotation)
=
Höhe(vor Einfügen)

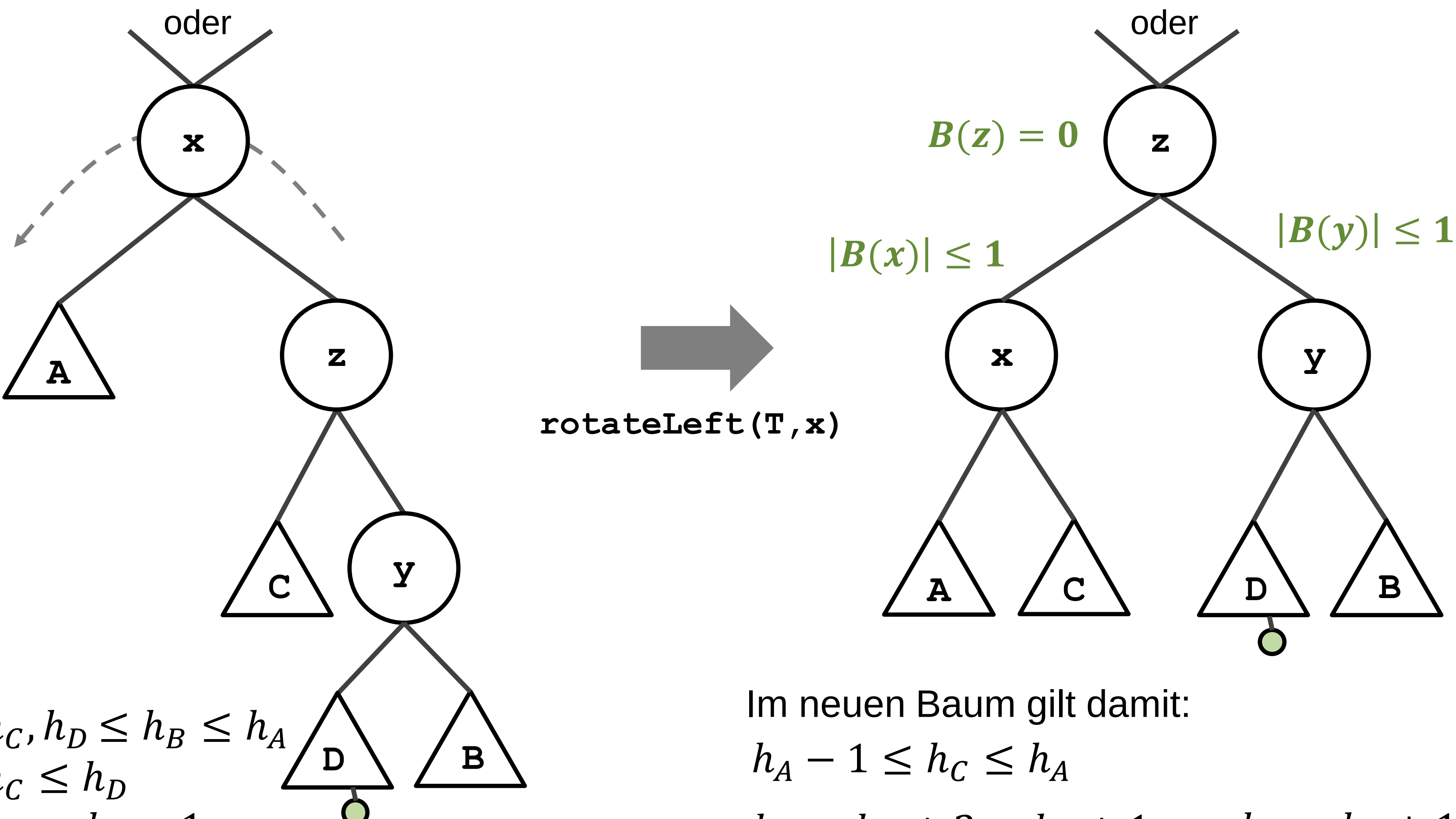
Rebalancieren: Fall II (erste Rotation)



$$h_C, h_D \leq h_B \leq h_A$$
$$h_C \leq h_D$$
$$h_D = h_A - 1$$

Rebalancieren: Fall II (zweite Rotation)

Höhe auch hier unverändert



$h_C, h_D \leq h_B \leq h_A$

$h_C \leq h_D$

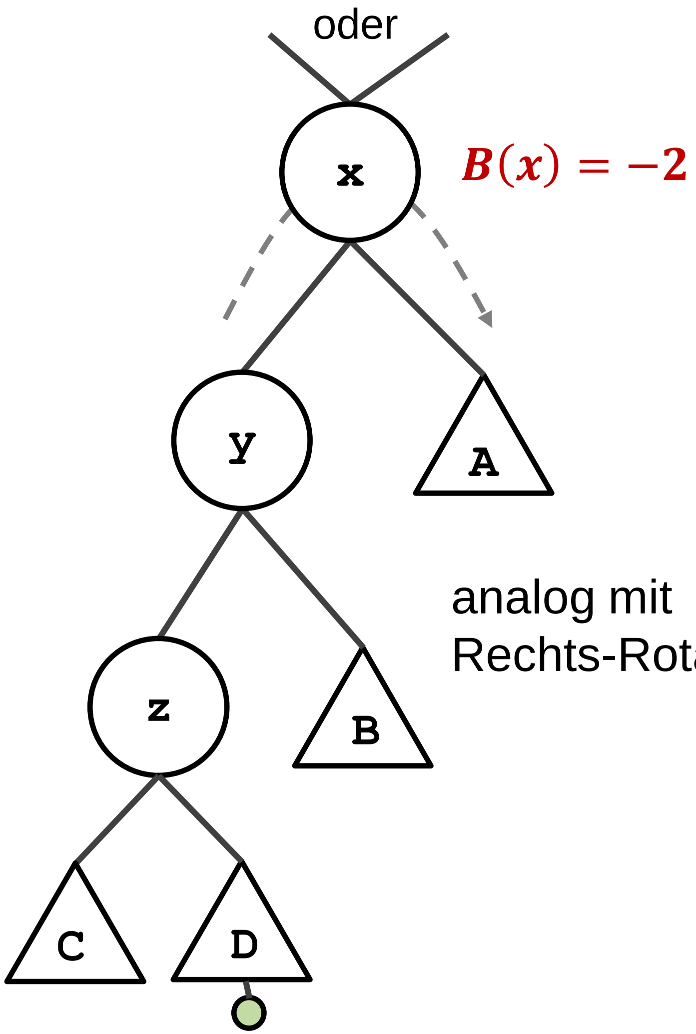
$h_D = h_A - 1$

Im neuen Baum gilt damit:

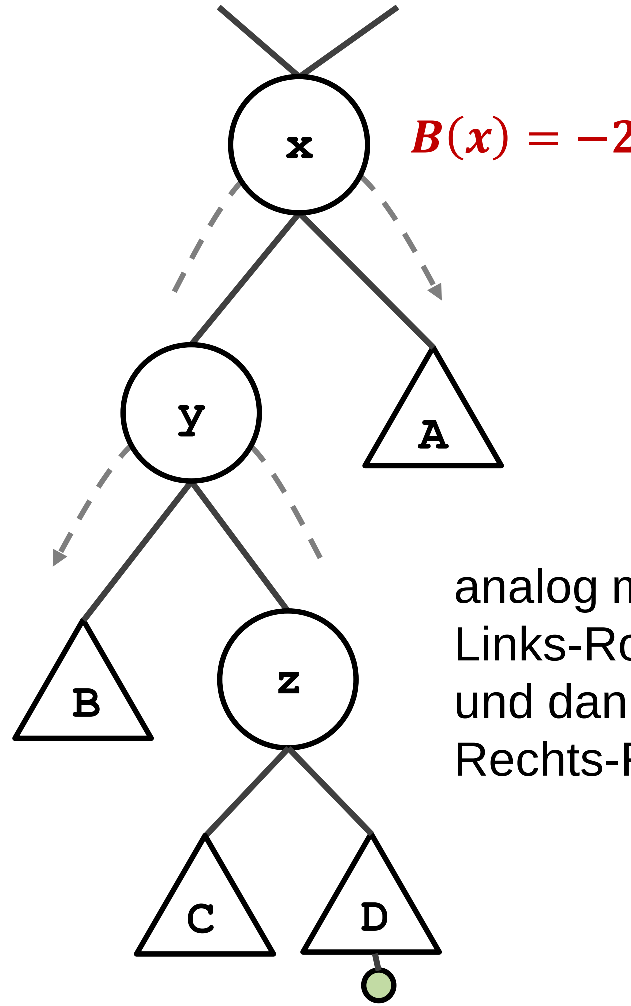
$h_A - 1 \leq h_C \leq h_A$

$h_y = h_D + 2 = h_A + 1 \qquad h_x = h_A + 1$

Rebalancieren: Fälle III+IV



analog mit
Rechts-Rotation



analog mit
Links-Rotation
und dann
Rechts-Rotation

Einfügen: Laufzeit

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

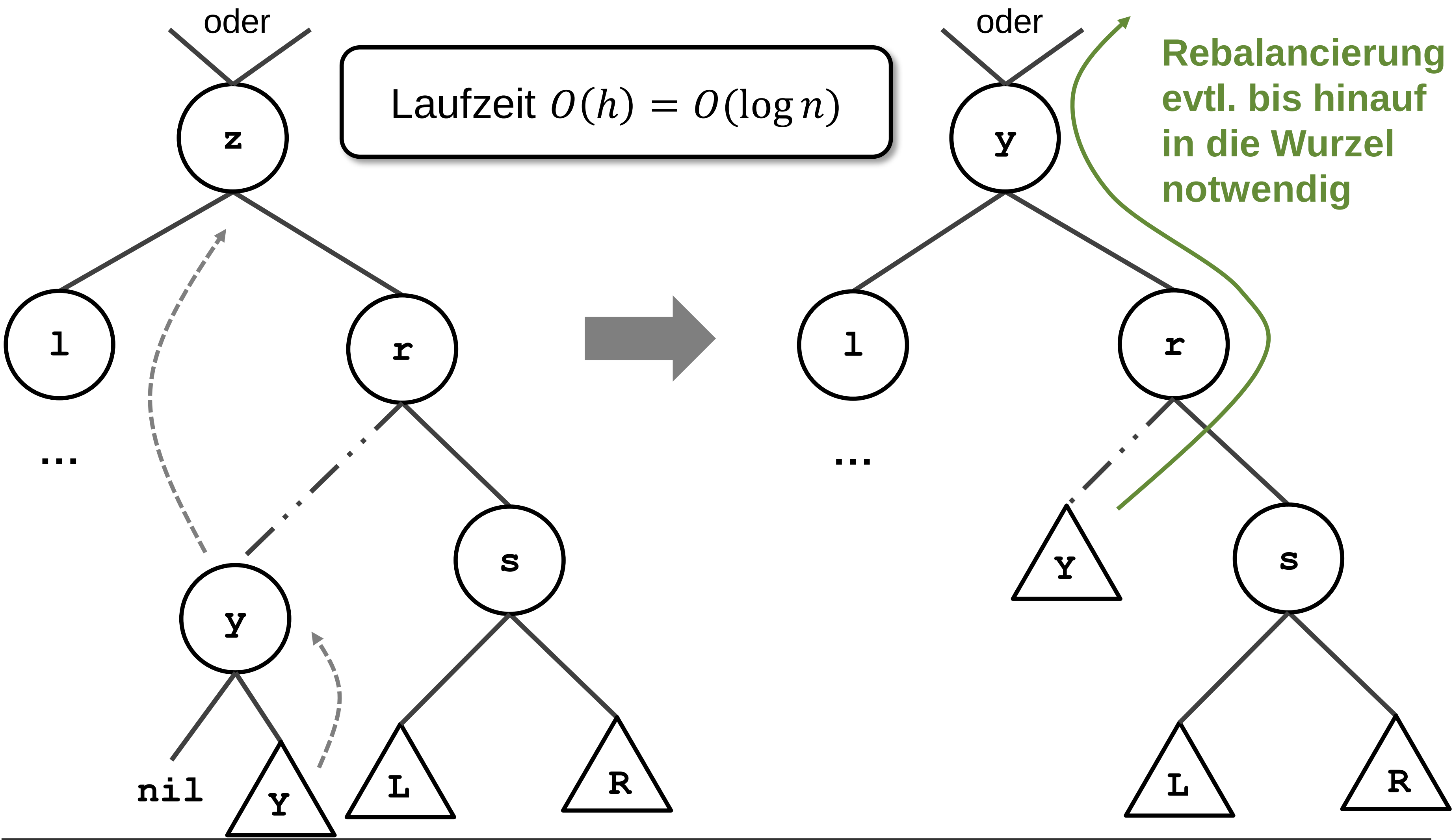
```
insert(T,z)
//z.left==z.right==nil;

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);
```

Laufzeit = $O(h)$

Laufzeit = $O(h)$, da Suche nach unbalanciertem Knoten Richtung Wurzel in $O(h)$, und Rebalancieren nur einmal nötig

Löschen analog zum binären Suchbaum, aber:

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



Worst-Case-Laufzeiten

AVL-Bäume

Operation	Laufzeit
Einfügen	$\Theta(\log n)$
Löschen	$\Theta(\log n)$
Suchen	$\Theta(\log n)$

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

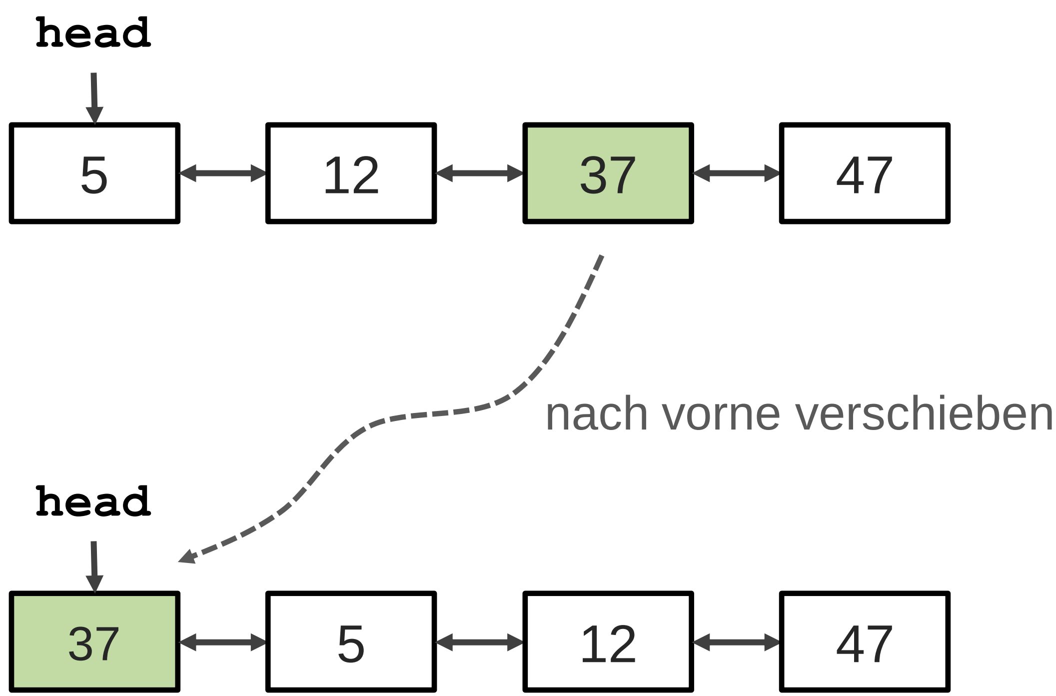
Splay-Bäume

(selbst-organisierende Datenstrukturen)

Selbst-Organisierende Listen

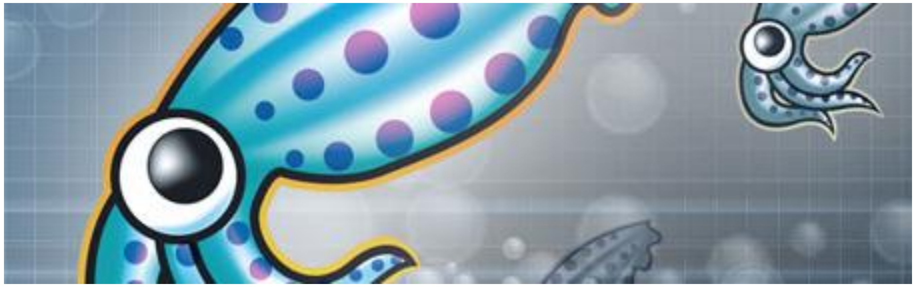
Ansatz: einmal angefragte Werte werden voraussichtlich noch öfter angefragt

`search(L,37)`



bekannte Variante für Bäume: Splay Trees

Anwendung: SQUID



Quelle: **www.squid-cache.org/**

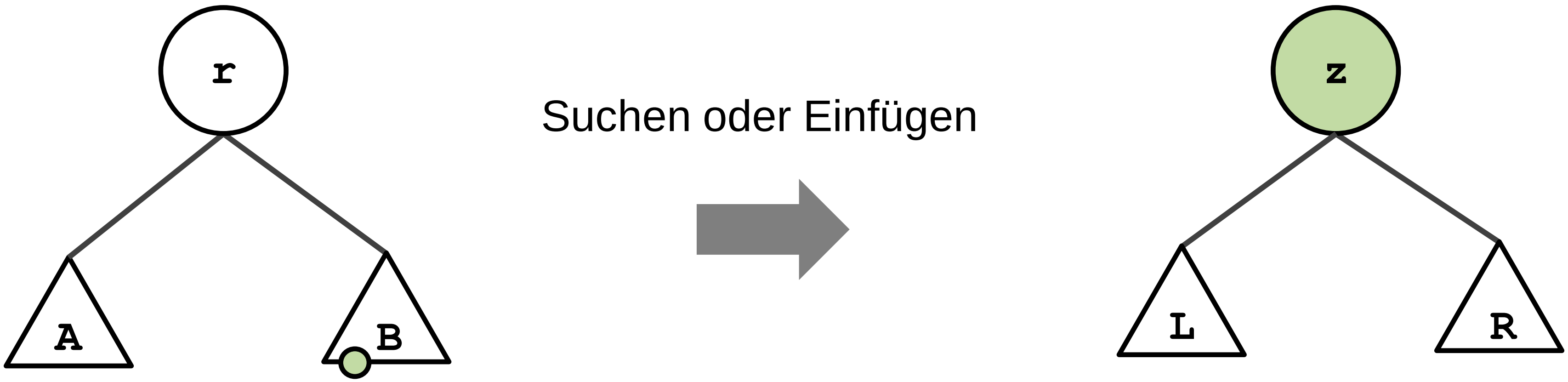
SQUID: Web-Cache-Proxy

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

```
2018/03/17 18:29:45| WARNING: (B) '127.0.0.1' is a subnetwork
                     of (A) '127.0.0.1'
2018/03/17 18:29:45| WARNING: because of this '127.0.0.1' is ignored
                     to keep splay tree searching predictable
2018/03/17 18:29:45| WARNING: You should probably remove '127.0.0.1’
                     from the ACL named 'localhost'
```

Splay-Operation

Splay-Bäume bilden Untermenge der BST

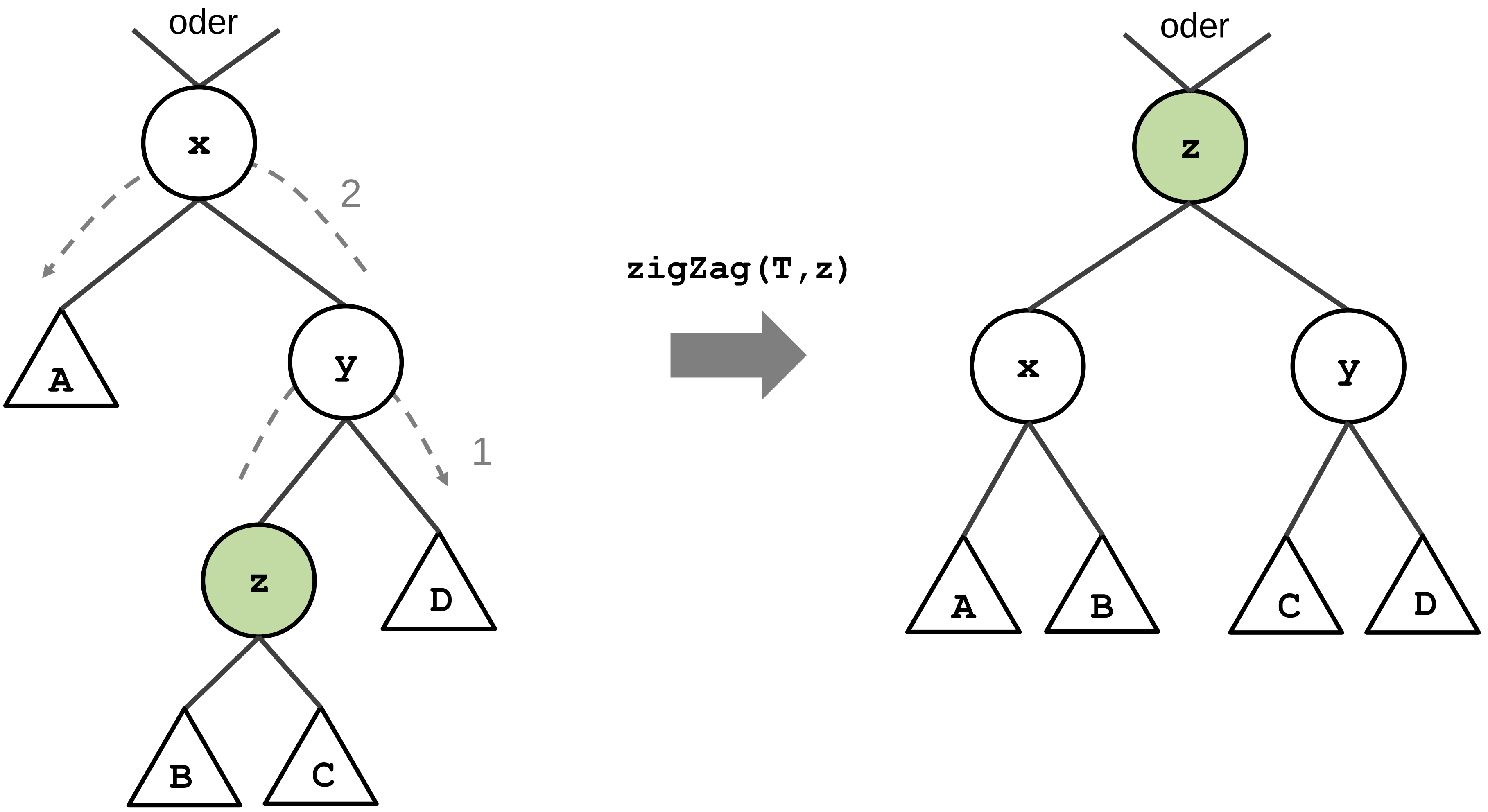


Spüle gesuchten oder neu eingefügten Knoten an die Wurzel

`splay(T,z)` = Folge von Zig-, Zig-Zig- und Zig-Zag-Operationen

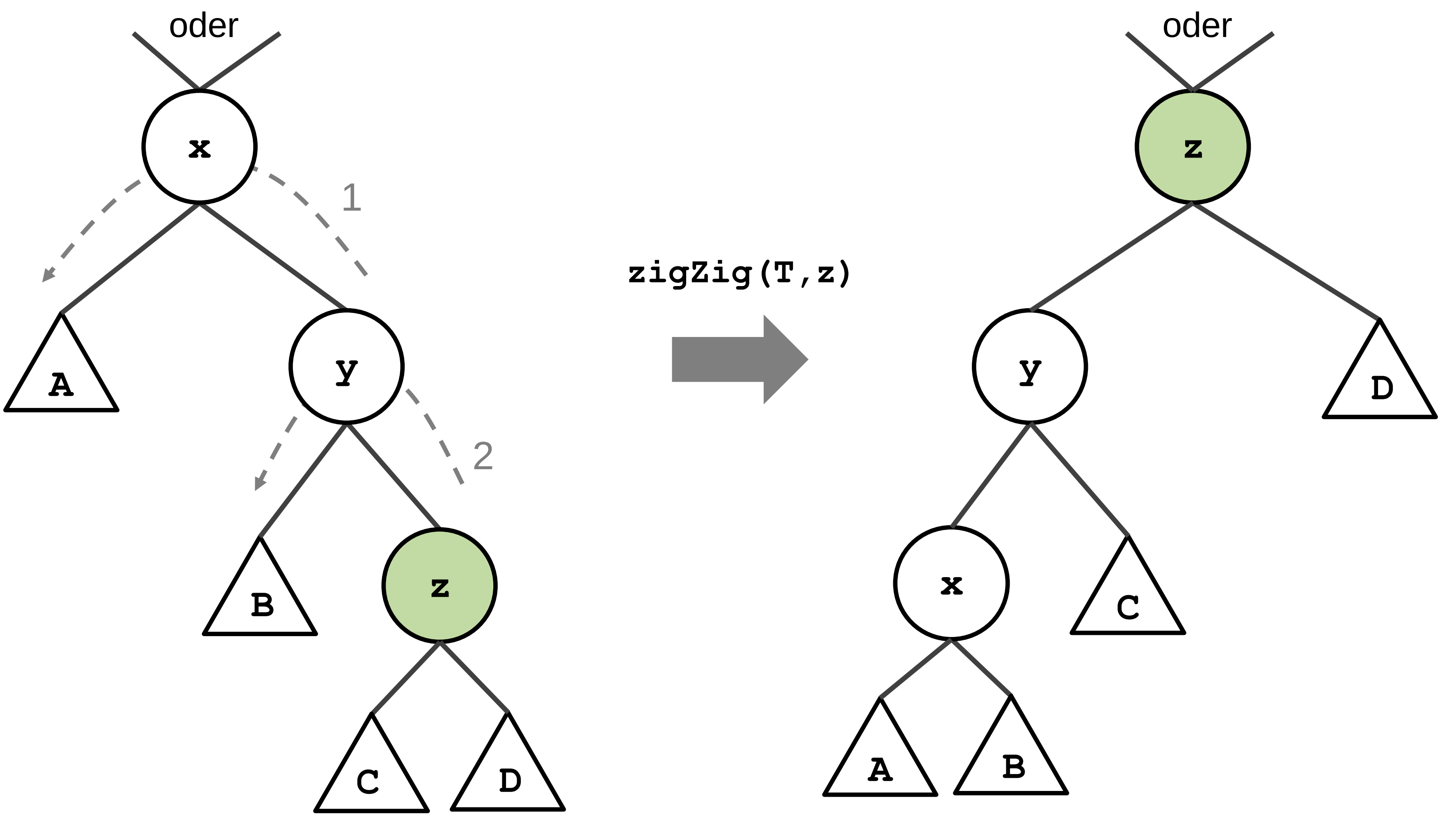
Zig-Zag-Operation

=Rechts-Links- oder Links-Rechts-Rotation



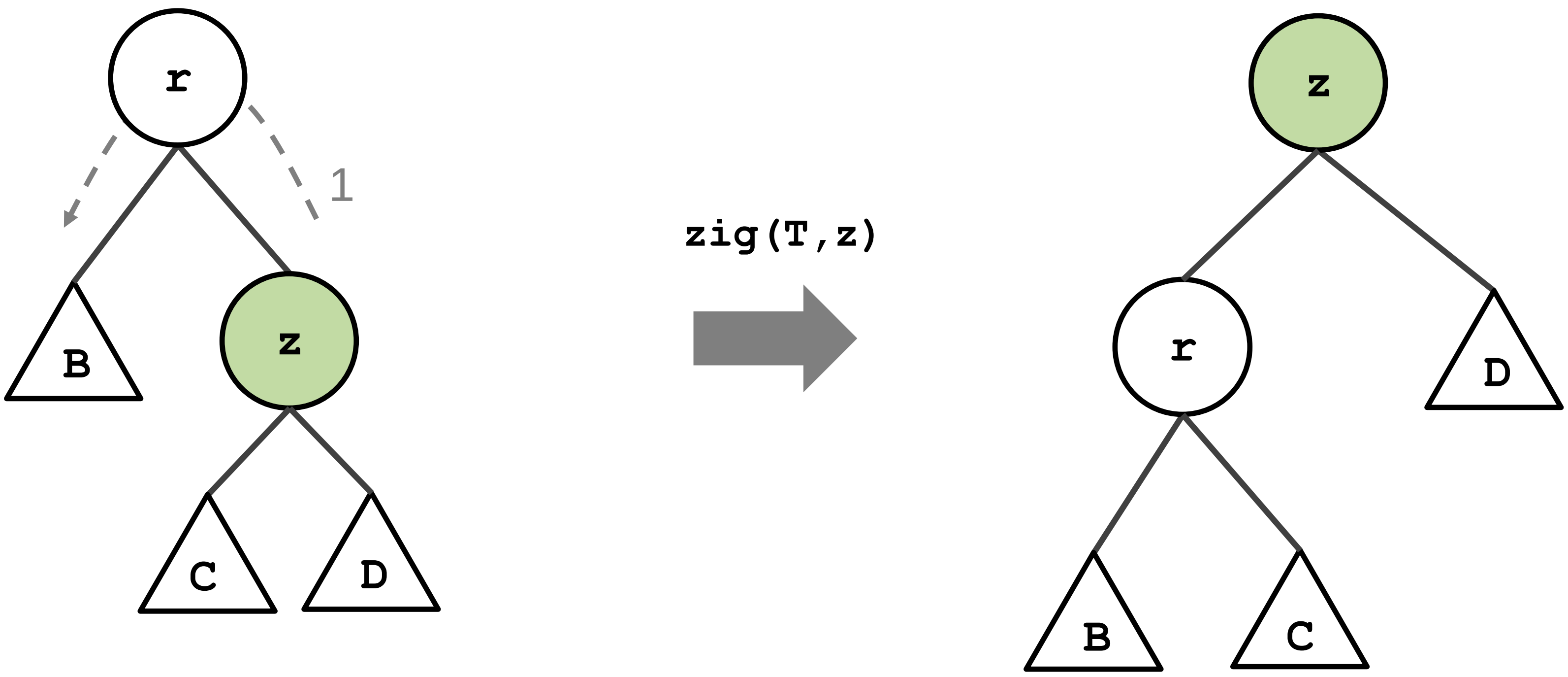
Zig-Zig-Operation

=Links-Links- oder Rechts-Rechts-Rotation



Zig-Operation

=einfache Links- oder Rechts-Rotation



(Falls **`z`** direkt unter der Wurzel hängt)

Splay-Operation

Gesamtlaufzeit $O(h)$

```
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);
```

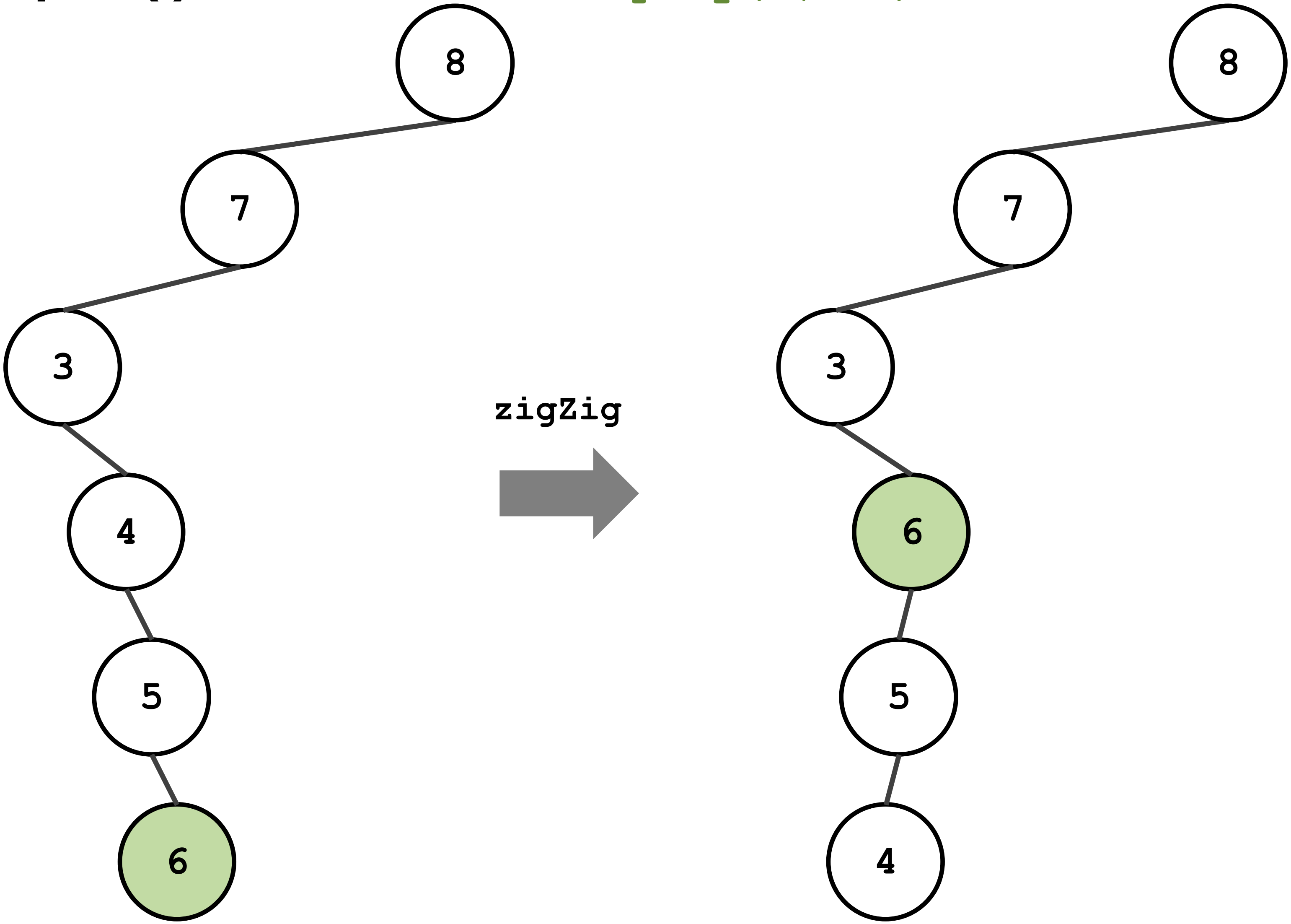
Laufzeit:
Bei jeder Iteration wird **`z`** mindestens einen Level nach oben rotiert

```
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);
```

`zig` und **`zigZag`** analog

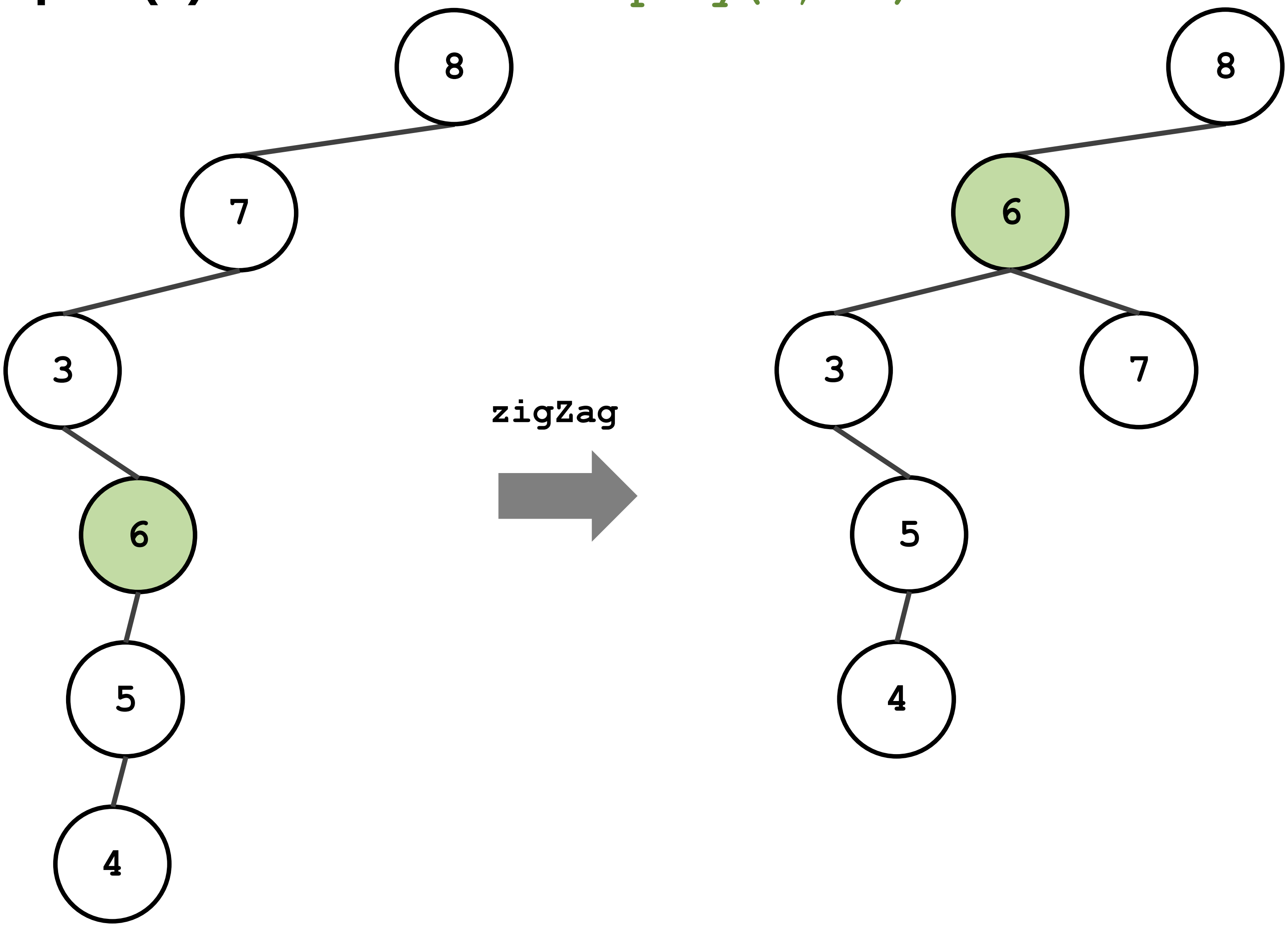
Beispiel (I)

splay(T,→6)



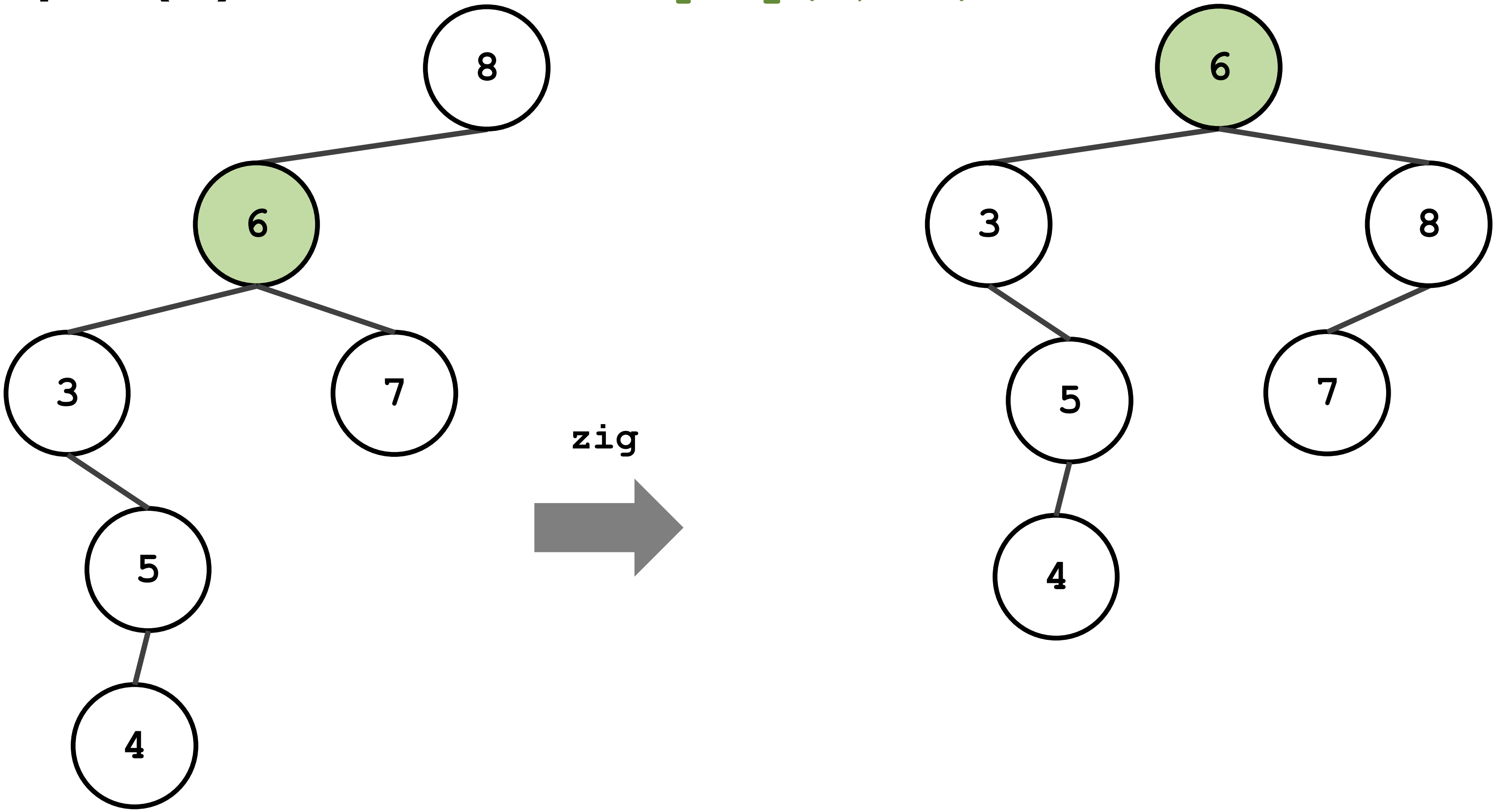
Beispiel (II)

`splay(T,→6)`



Beispiel (III)

`splay(T,→6)`



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;

IF x==nil THEN
    return nil
ELSE
    splay(T,x);
    return T.root;
```

Laufzeit $O(h)$

Alternativ: „splaye“ dann letzten besuchten Knoten bei erfolgloser Suche nach oben

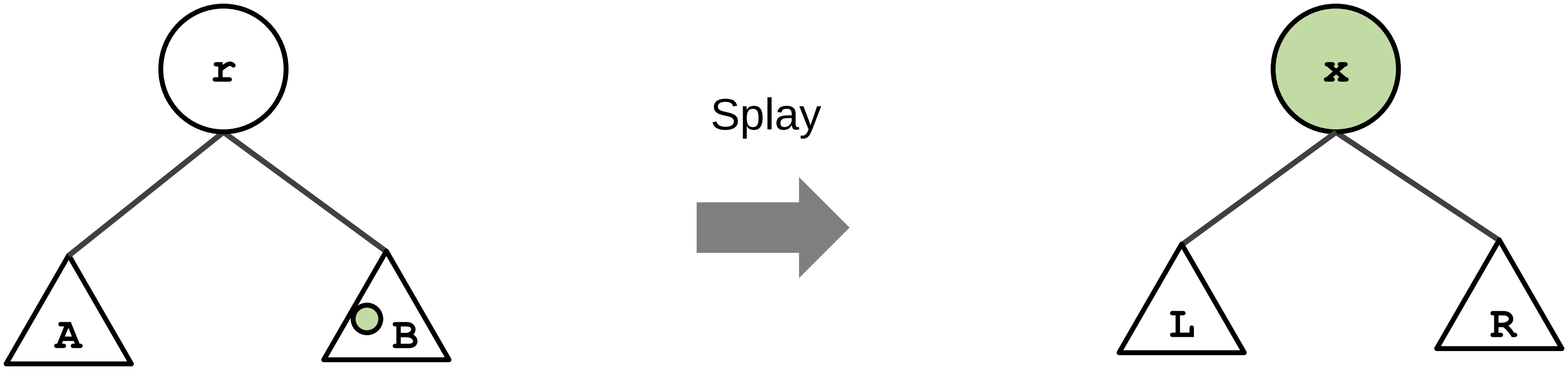
Laufzeit $O(h)$

Gesamtlaufzeit $O(h)$

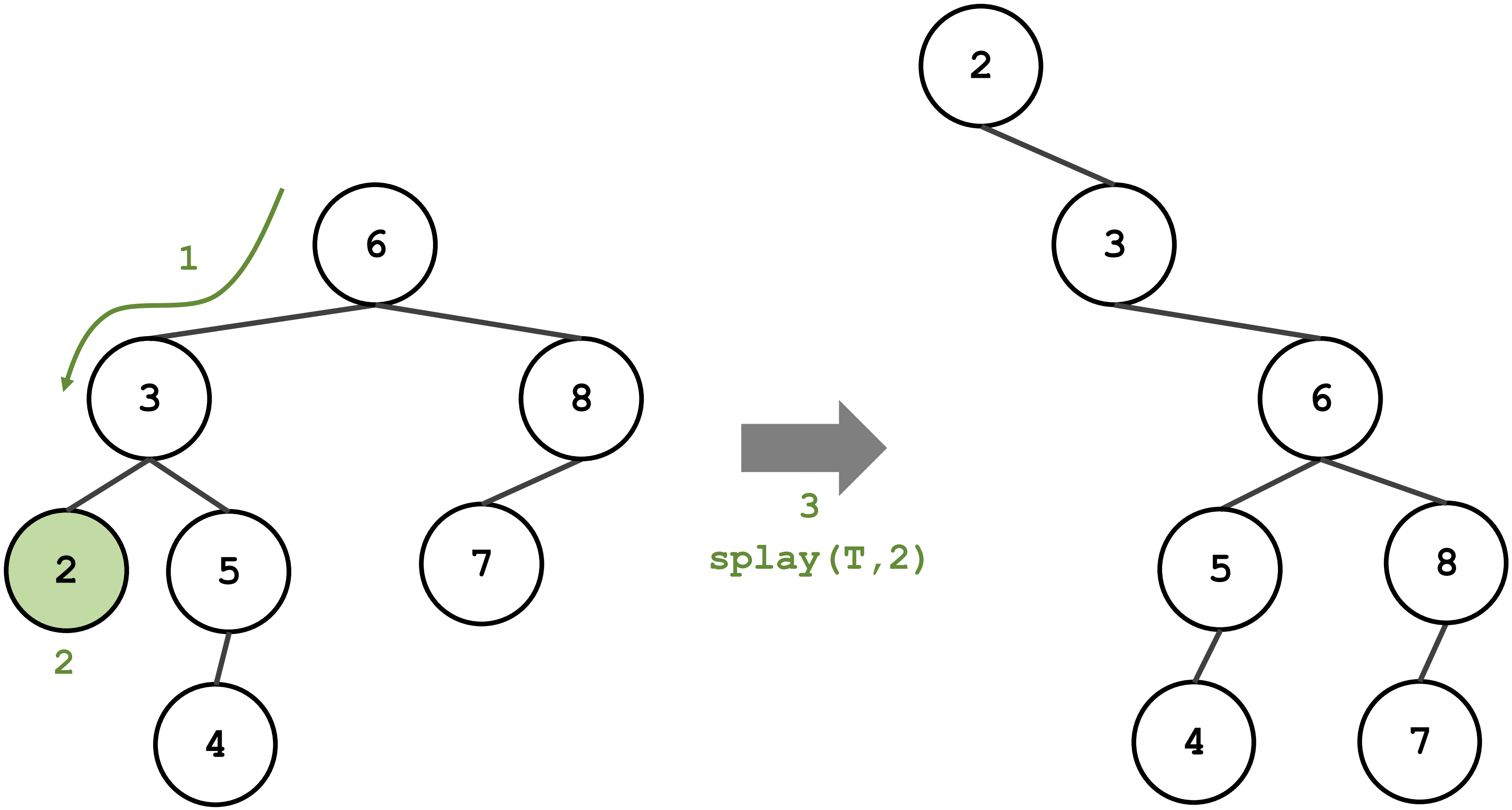
Einfügen

1. Suche analog zum Einfügen bei BST Einfügepunkt

2. Spüle eingefügten Knoten **x** per Splay-Operation nach oben



Beispiel `insert(T,2)`



Einfügen: Laufzeit

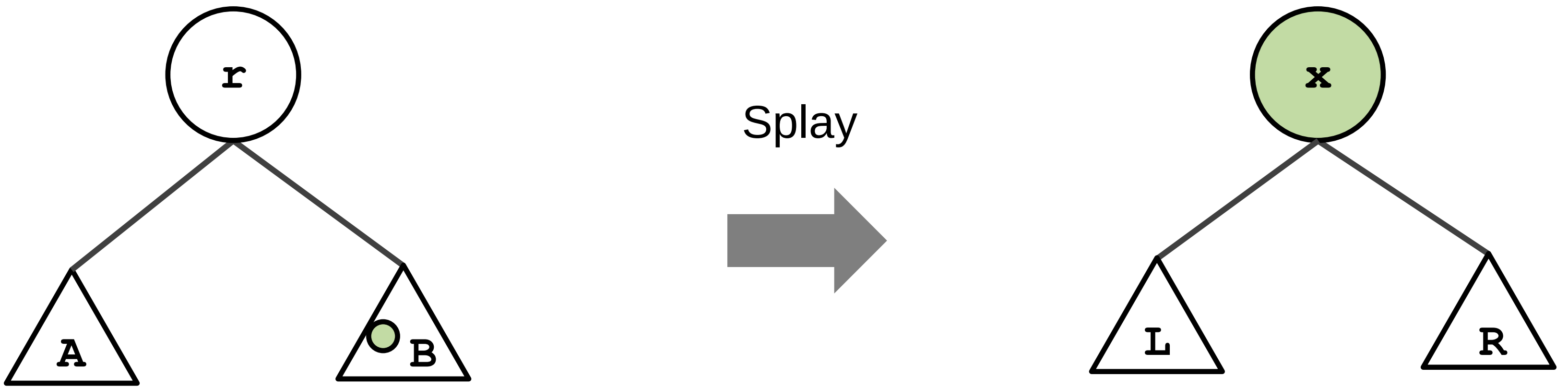
1. Position im BST suchen $O(h)$

2. `splay(T,x)` $O(h)$

Gesamtlaufzeit $O(h)$

Löschen (I)

1. Spüle gesuchten Knoten **x** per Splay-Operation nach oben

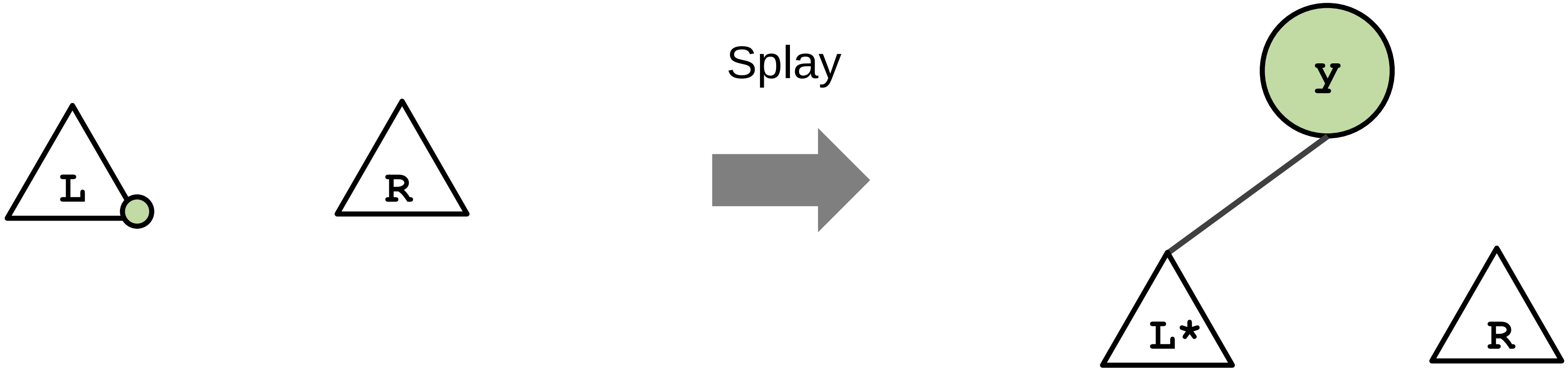


2. Lösche **x**

Wenn einer der beiden Teilbäume leer, dann fertig

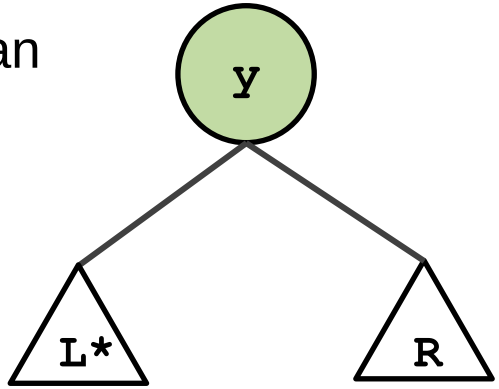
Löschen (II)

3. Spüle den „größten“ Knoten **`y`** in **`L`** per Splay-Operation nach oben



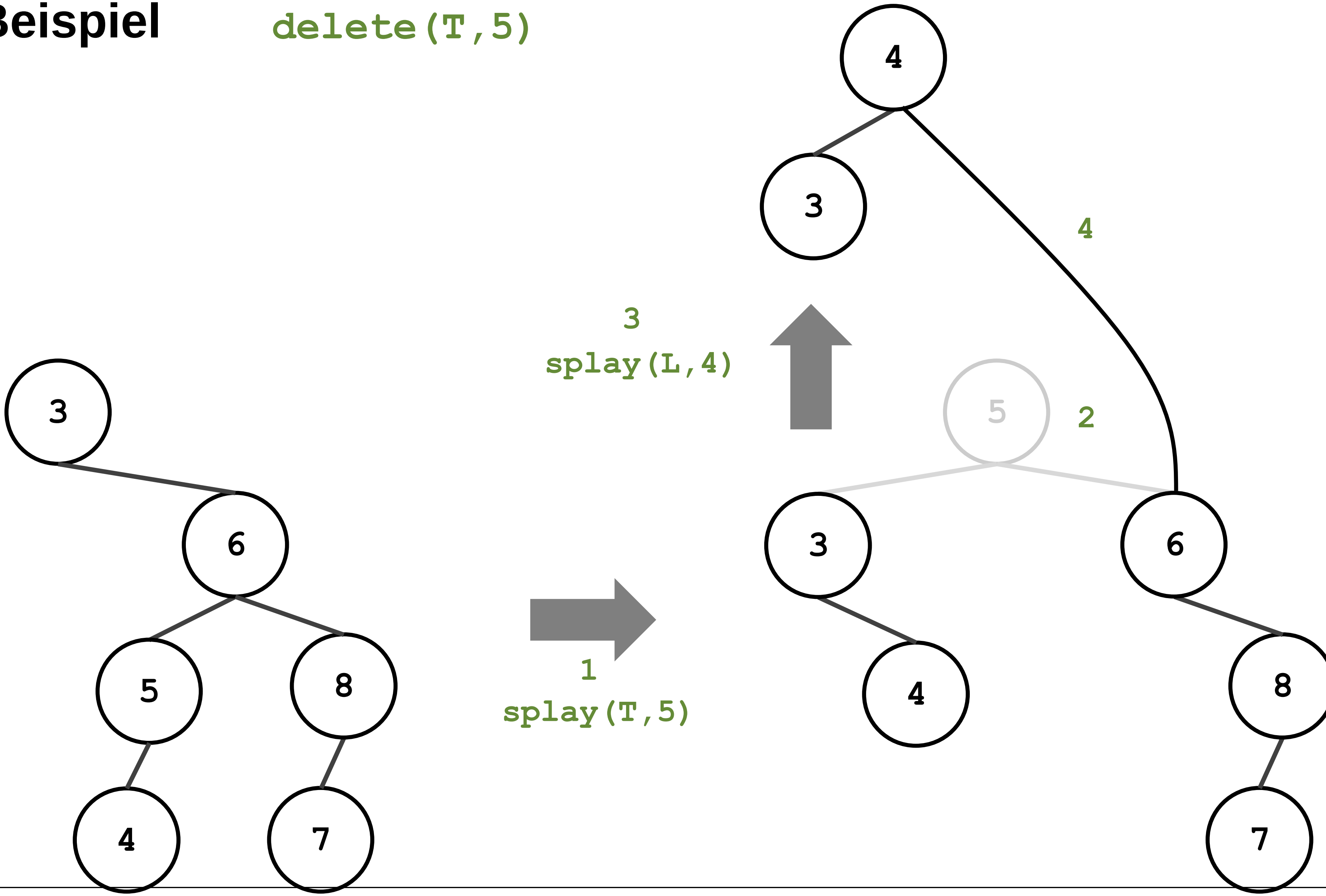
`y` kann keinen rechtes Kind haben, da größter Wert in **`L`**

4. Hänge **`R`** an **`y`** an



Beispiel

delete(T,5)



Löschen: Laufzeit

1. **`splay(T,x)`** $O(h)$

2. **`x`** löschen $O(1)$

3. Max-Knoten **`y`** in **`L`** bestimmen
   und **`splay(L,y)`** $O(h) + O(h) = O(h)$

4. Anhängen $O(1)$

Gesamtlaufzeit $O(h)$

Laufzeit Splay-Bäume

Amortisierte Laufzeit:
Laufzeit pro Operation über mehrere Operationen hinweg

Operationen: Suchen, Löschen, Einfügen

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.

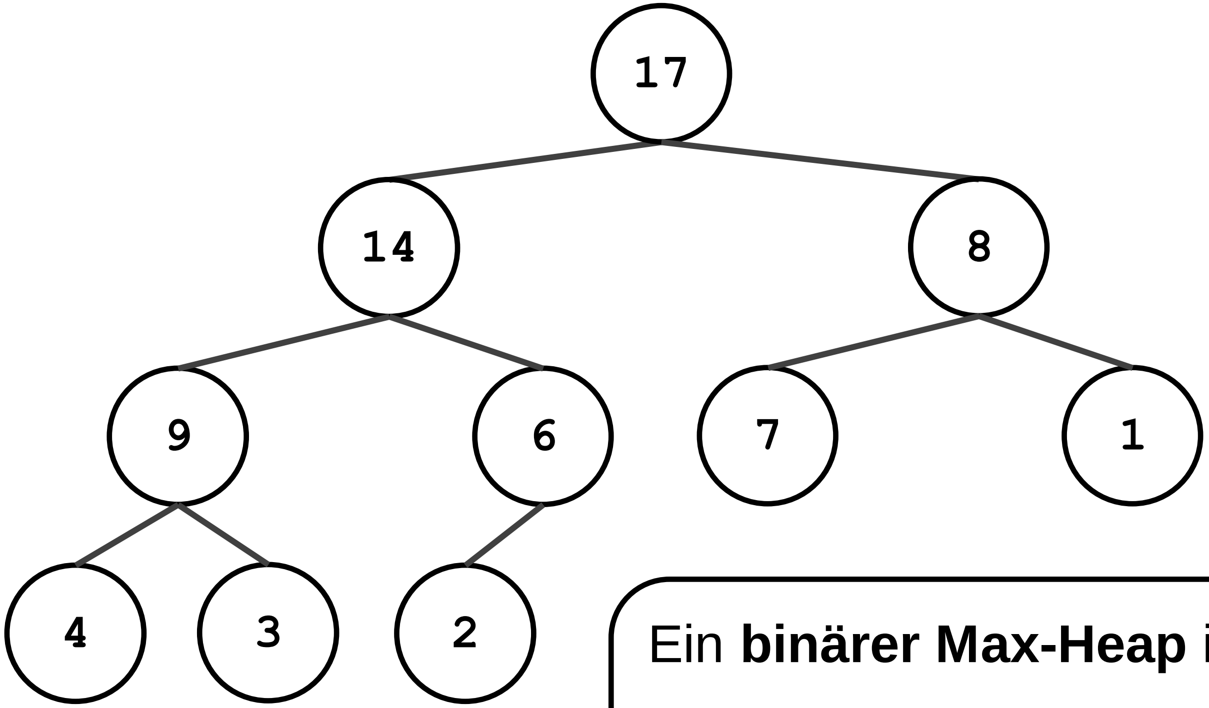
Zusätzlich: Oft gesuchte Elemente werden sehr schnell gefunden

Ändern Sie den Algorithmus zur Suche bei Splay-Bäumen so ab, dass auch bei erfolgloser Suche der letzte besuchte Knoten nach oben gespült wird.

Könnte man statt des Maximums in **`L`** beim Löschen auch das Minimum in **`R`** nehmen?

(Binäre Max-)Heaps

Binärer Max-Heap



Achtung:
Heaps sind keine BSTs,
linke Kinder können
größere Werte
als rechte Kinder
haben!

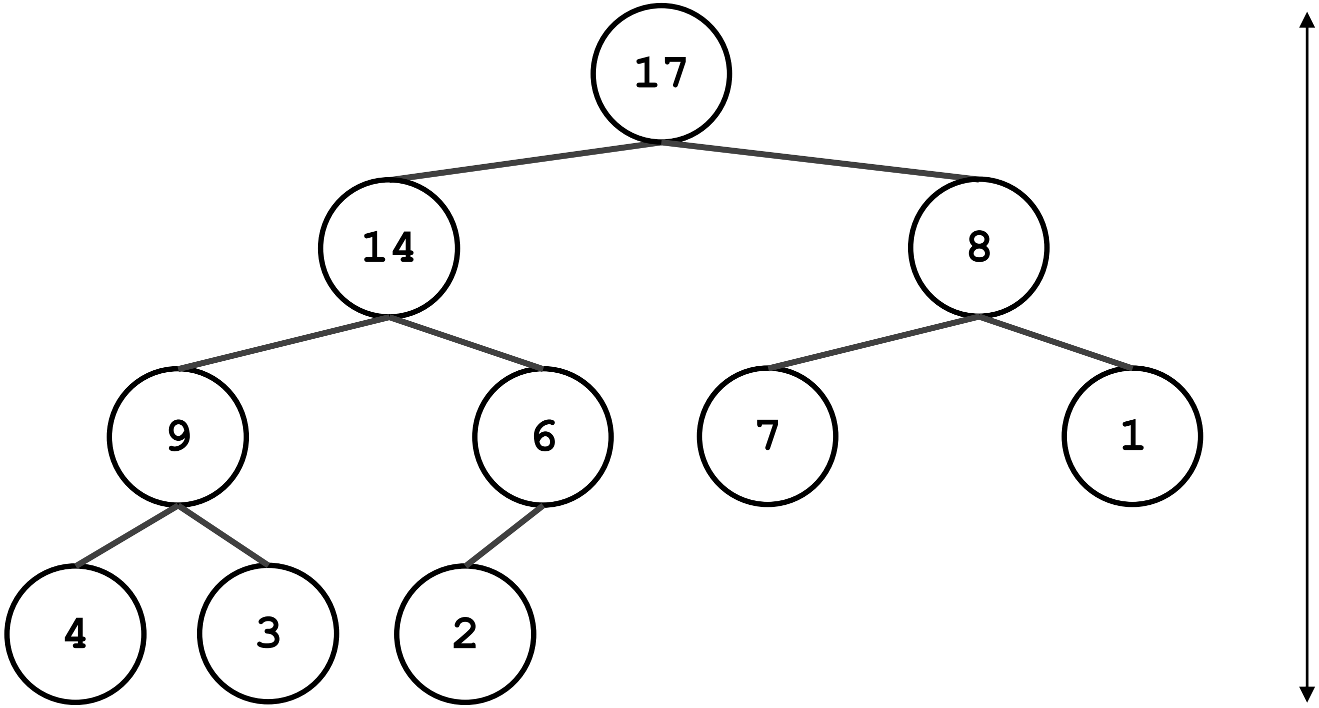
Ein **binärer Max-Heap** ist ein binärer Baum, der

(1) „bis auf das unterste Level vollständig und im untersten Level von links gefüllt ist" und

(2) Für alle Knoten **`x`** $\neq$ **`T.root`** gilt:
`x.parent.key` $\geq$ **`x.key`**

Bei Min-Heaps sind die Werte in Elternknoten jeweils kleiner

Ein Haufen Eigenschaften

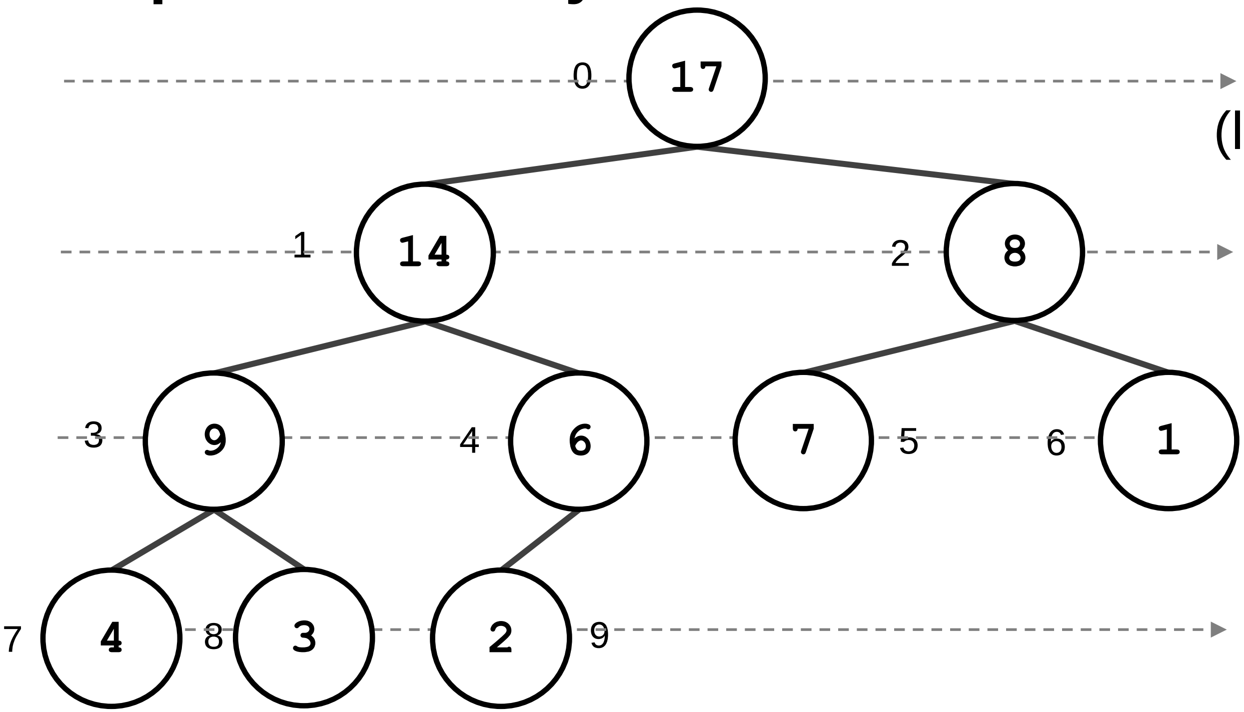


Da Baum (fast) vollständig, gilt

$$h \leq \log n$$

Maximum des Heaps steht in der Wurzel

Heaps durch Arrays



speichere Anzahl Knoten in **`H.size`**
(leerer Heap **`H.size==0`**)

Duale Sichtweise
als Pointer
oder als Array:

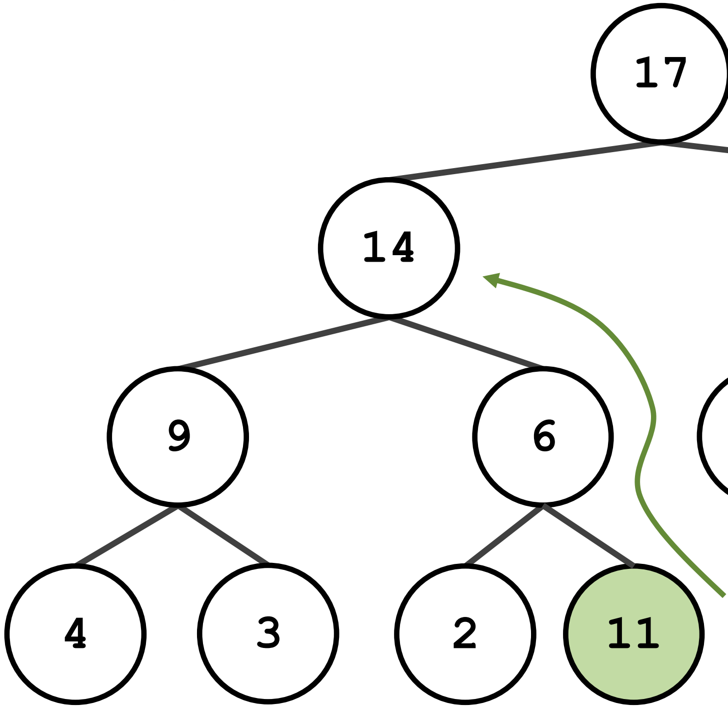
$\texttt{\textbf{j.parent}} = \left\lceil \frac{j}{2} \right\rceil - 1$

$\texttt{\textbf{j.left}} = 2(\mathrm{j}+1) - 1$

$\texttt{\textbf{j.right}} = 2(\mathrm{j}+1)$

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

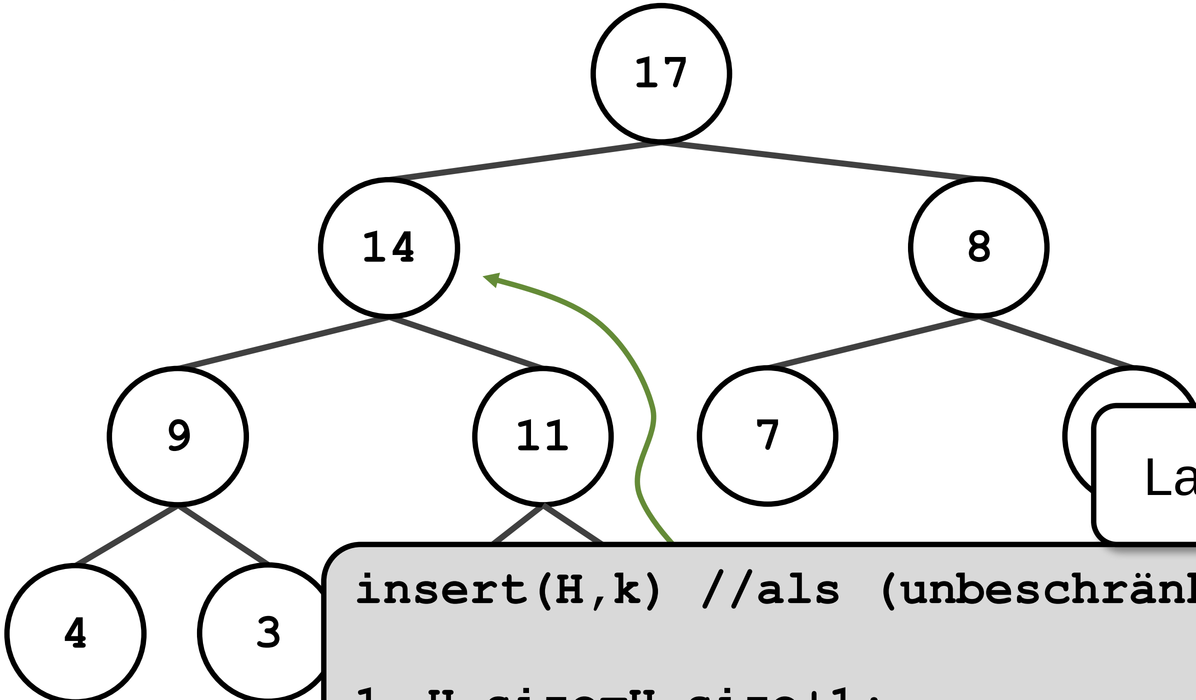
Einfügen



**Vertausche nach oben,
bis Max-Eigenschaft wieder erfüllt**

Position durch
Baumstruktur
vorgegeben

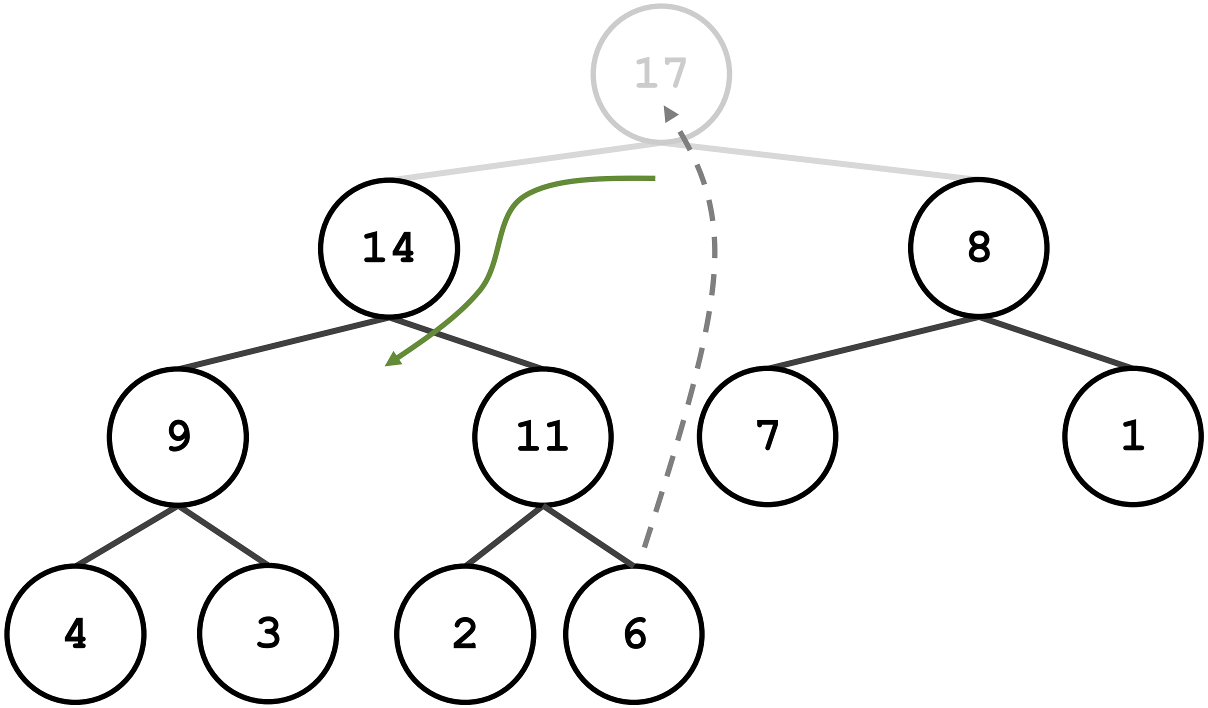
Einfügen: Algorithmus



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

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

H.size=H.size+1;
H.A[H.size-1]=k;

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

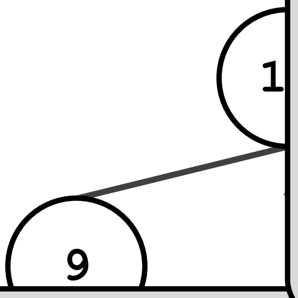
Lösche Maximum



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`**)

Lösche Max: Algorithmen



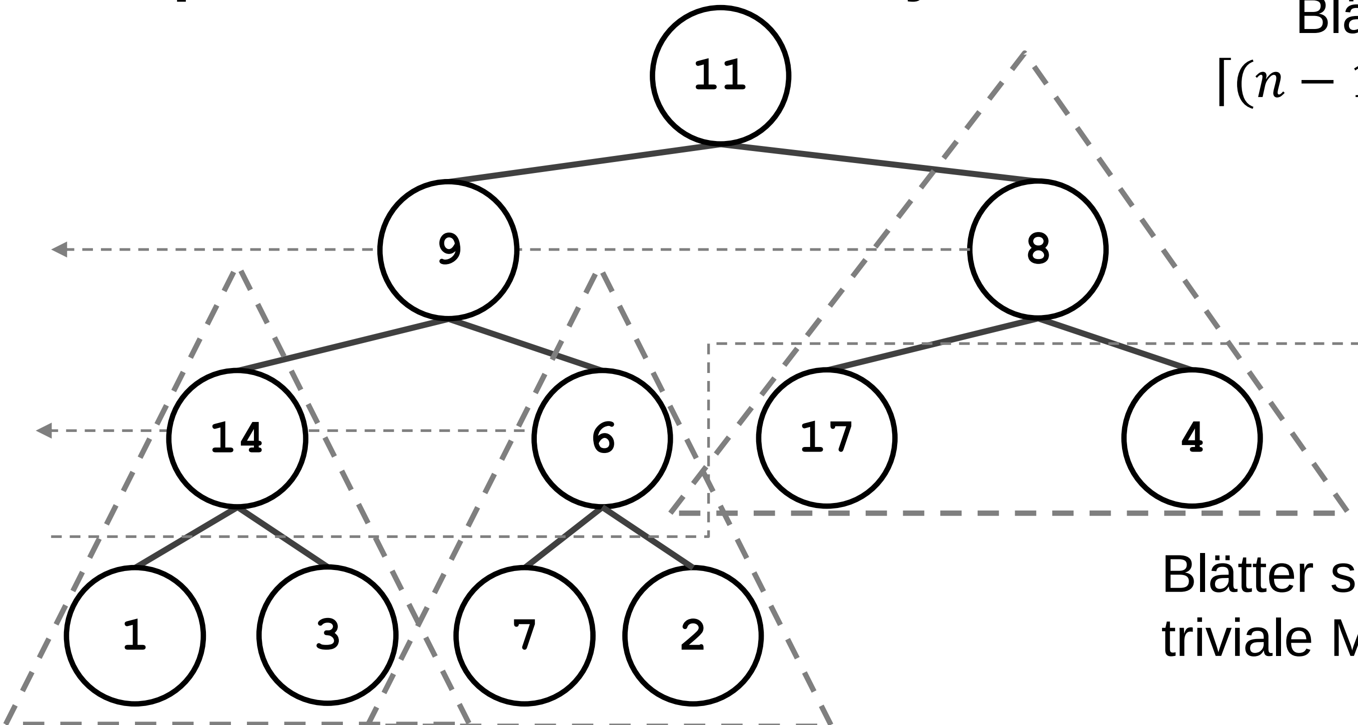

```
extract-max(H) //als (unbeschränktes) Array

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

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

```
heapify(H,i) //als (unbeschränktes) Array

maxind=i;
IF i.left<H.size AND H.A[i]<H.A[i.left] THEN
    maxind=i.left;
IF i.right<H.size 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 $O(h) = O(\log n)$

Heap-Konstruktion aus Array



Blätterindizes:
$[(n-1)/2], \ldots, n-1.$

Blätter sind für sich triviale Max-Heaps

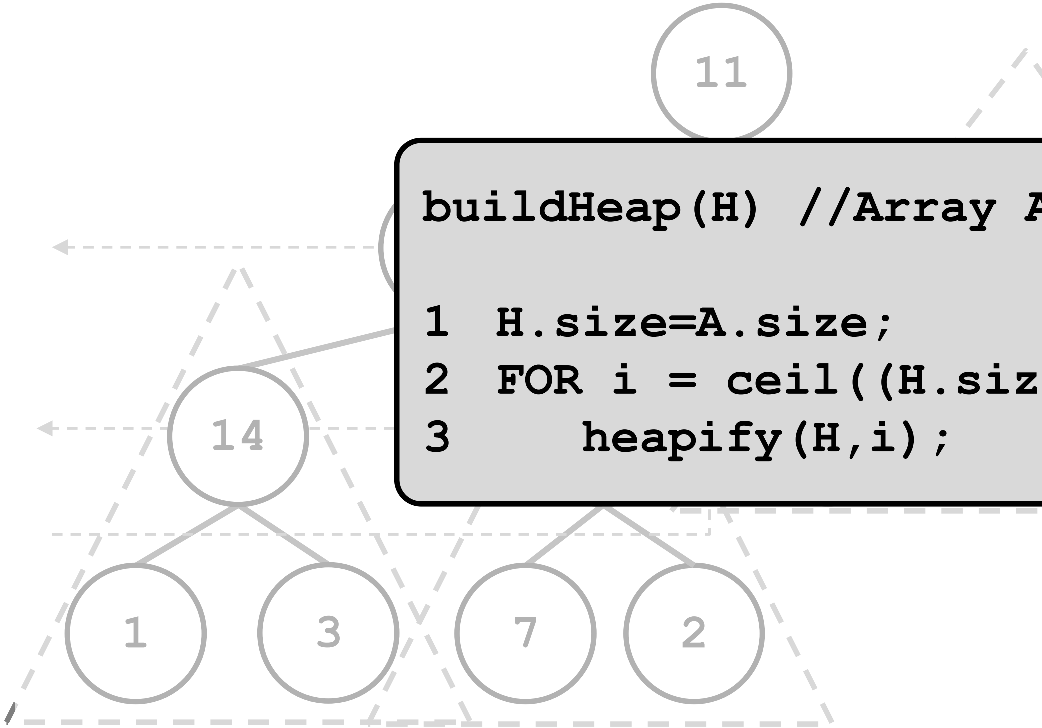
Baue rekursiv per **`heapify`** Max-Heaps für Teilbäume

0	1	2	3	4	5	6	7	8	9	10
11	9	8	14	6	17	4	1	3	7	2

Heap-Konstruktion: Algorithmus

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

```
buildHeap(H) //Array A schon nach H.A kopiert

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

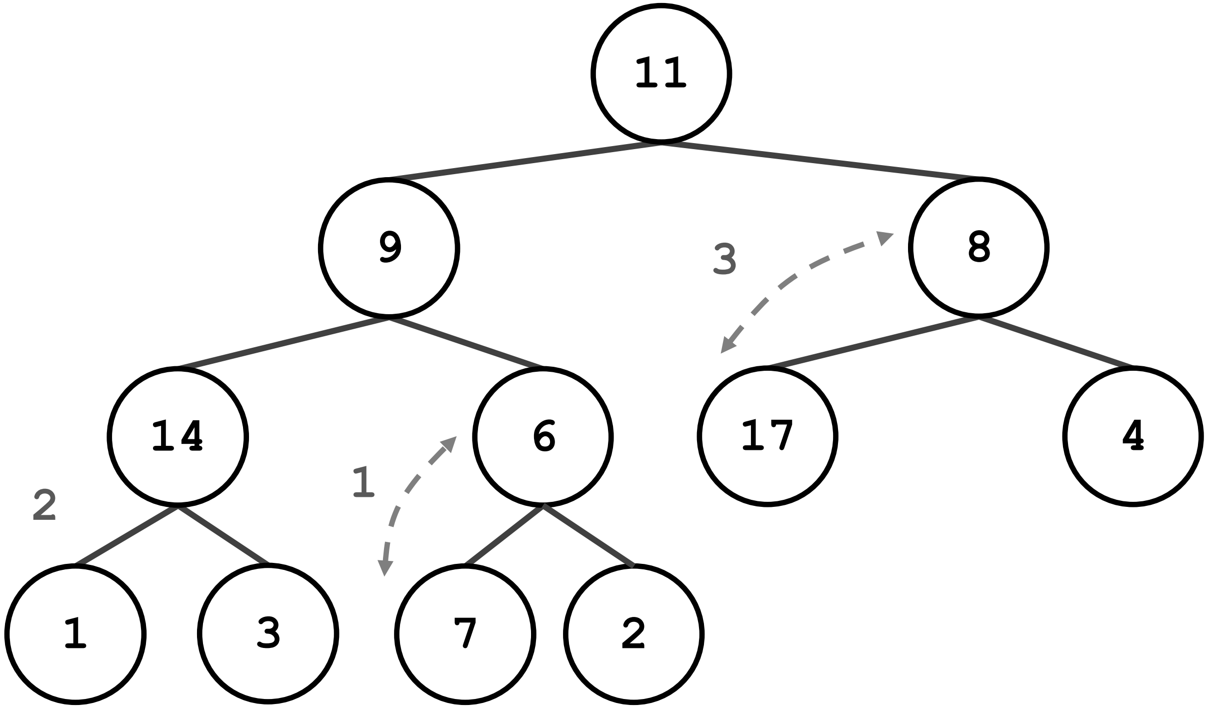


Blätter sind für sich triviale Max-Heaps

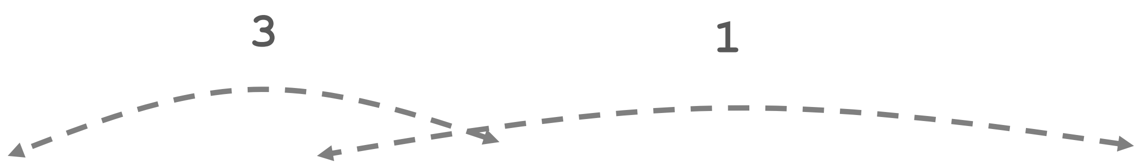
Baue rekursiv per `heapify` Max-Heaps für Teilbäume

0	1	2	3	4	5	6	7	8	9	10
11	9	8	14	6	17	4	1	3	7	2

Heap-Konstruktion: Beispiel (I)

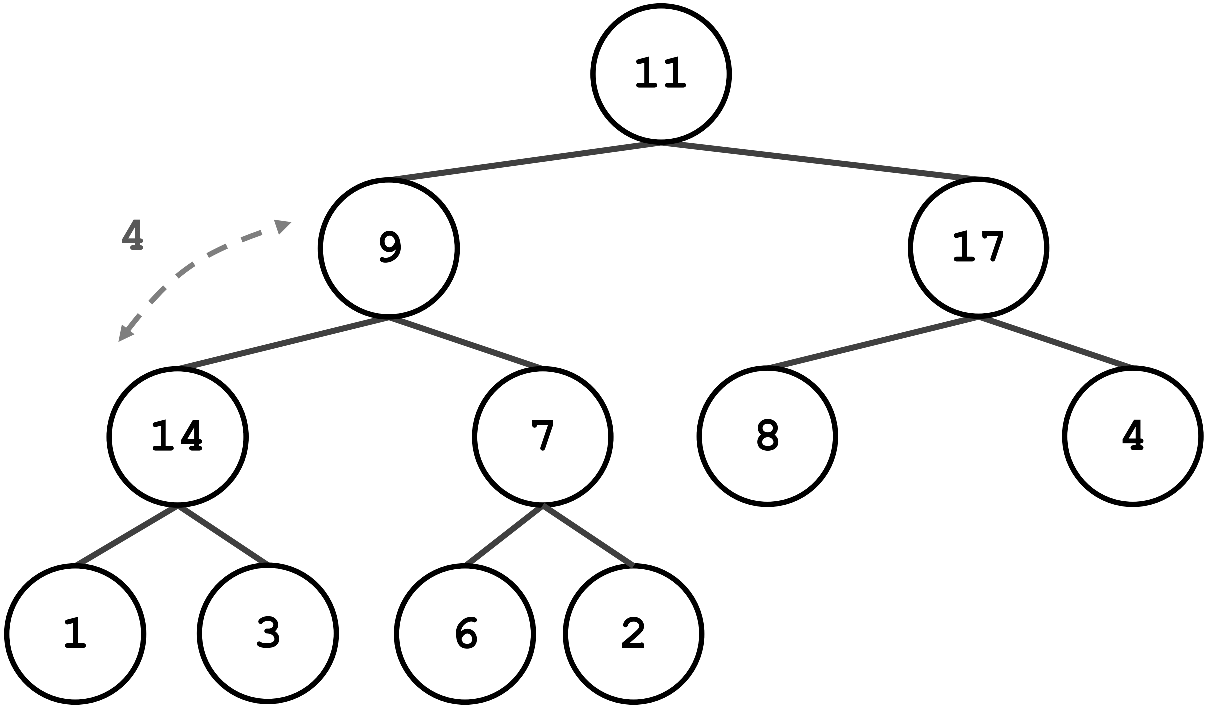


`heapify` auf den Knoten **`6`**, **`14`** und **`8`**

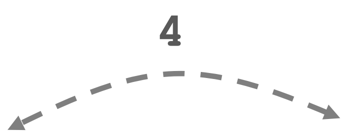


0	1	2	3	4	5	6	7	8	9	10
11	9	8	14	6	17	4	1	3	7	2

Heap-Konstruktion: Beispiel (II)

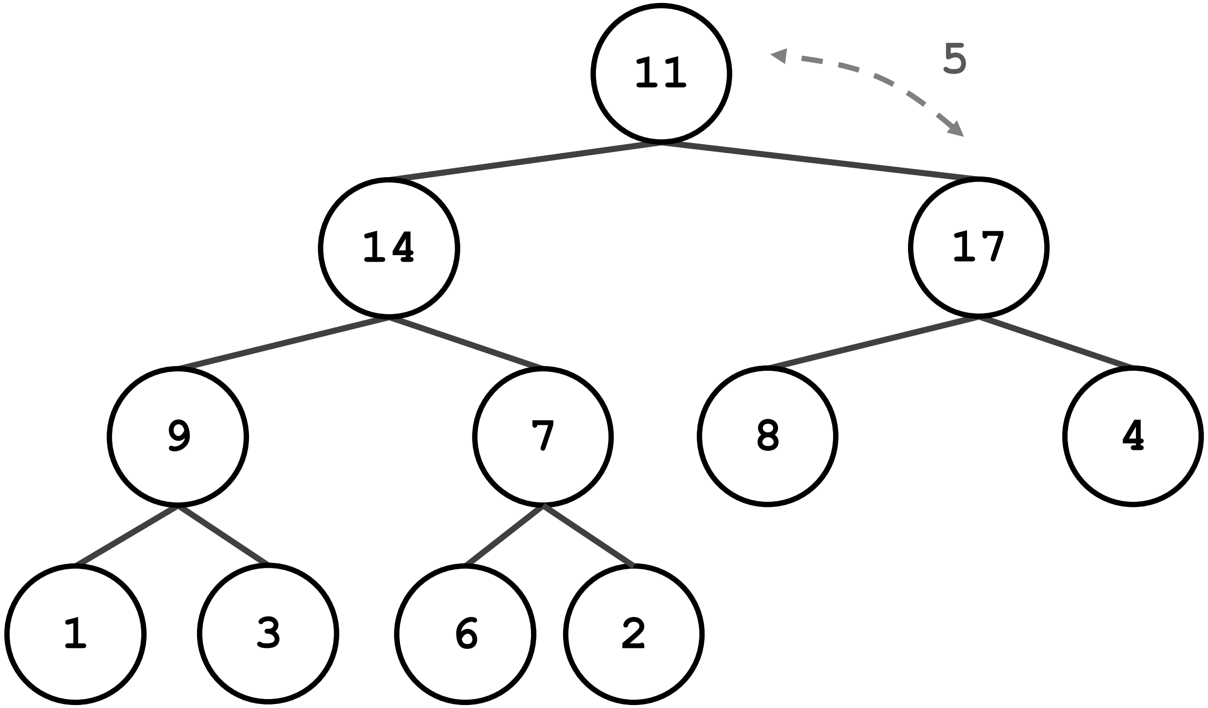


`heapify` auf dem Knoten **`9`**

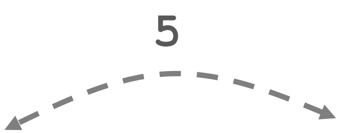


0	1	2	3	4	5	6	7	8	9	10
11	9	17	14	7	8	4	1	3	6	2

Heap-Konstruktion: Beispiel (III)

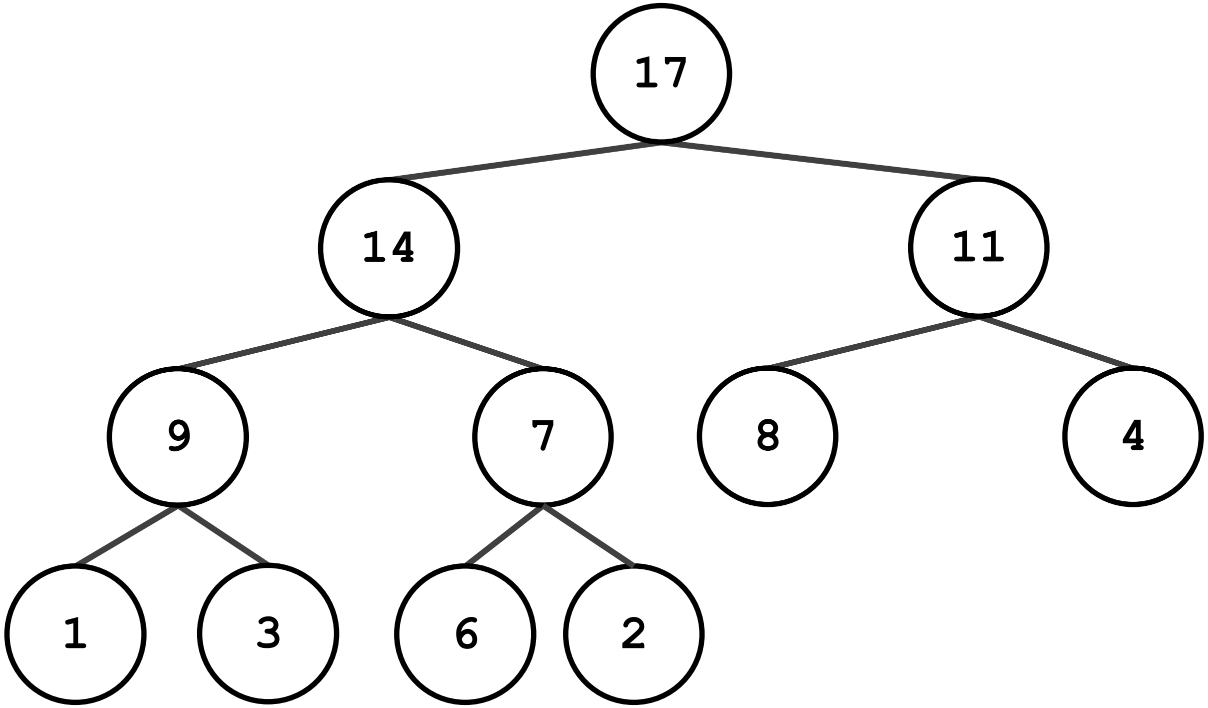


`heapify` auf dem Knoten **`11`**



0	1	2	3	4	5	6	7	8	9	10
11	14	17	9	7	8	4	1	3	6	2

Heap-Konstruktion: Beispiel (IV)



0	1	2	3	4	5	6	7	8	9	10
17	14	11	9	7	8	4	1	3	6	2

Heap-Sort

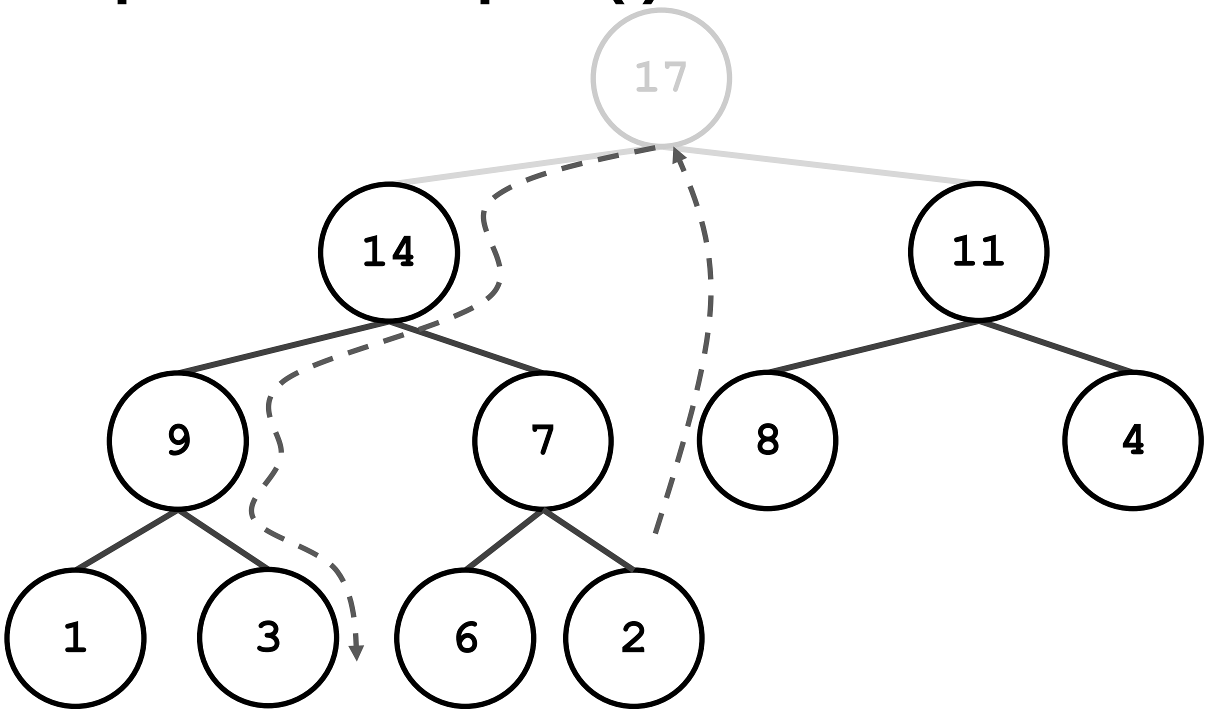
Laufzeit $O(n \cdot h) = O(n \cdot \log n)$

```
heapSort(H) //Array A schon nach H.A kopiert

1  buildHeap(H);
2  WHILE !isEmpty(H) DO PRINT extract-max(H);
```

Gibt Einträge in Array **`A`** in absteigender Größe aus

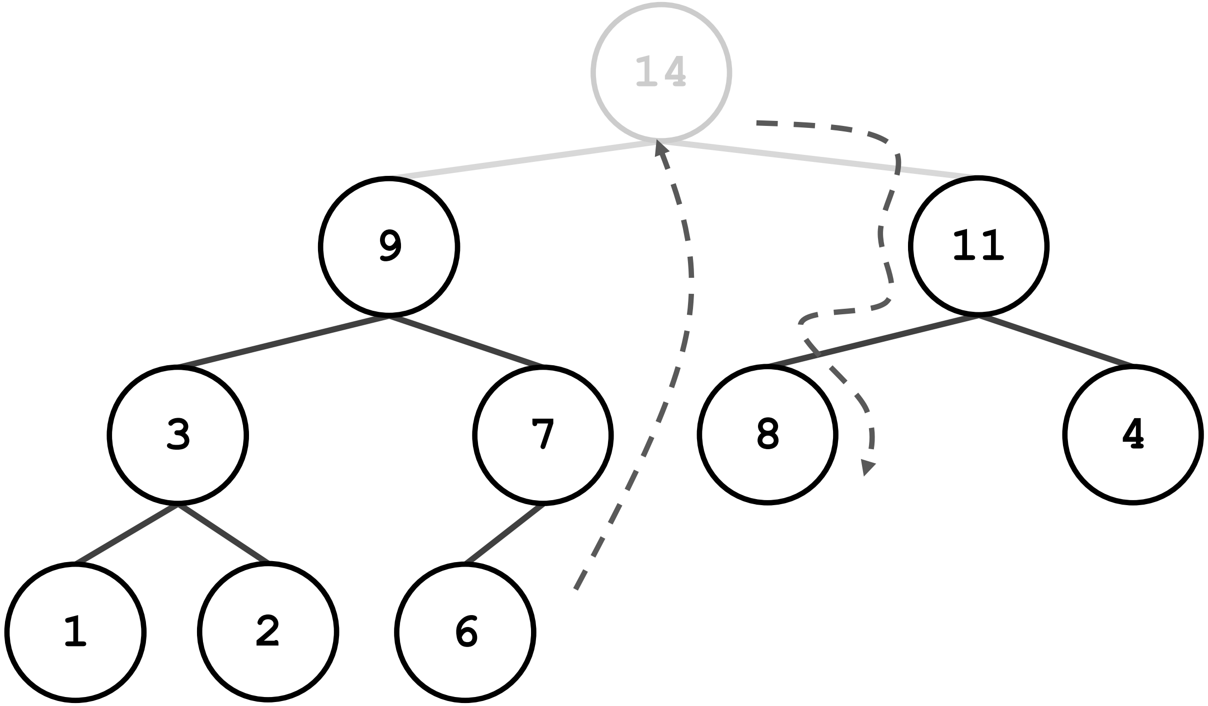
Alternativ: speichere in jeder **`WHILE`**-Iteration **`max=extract-max(H)`** in **`H.A[H.size]=max`**, um sortierte Liste am Ende aufsteigend im Array **`A`** zu haben.

Heap-Sort: Beispiel (I)



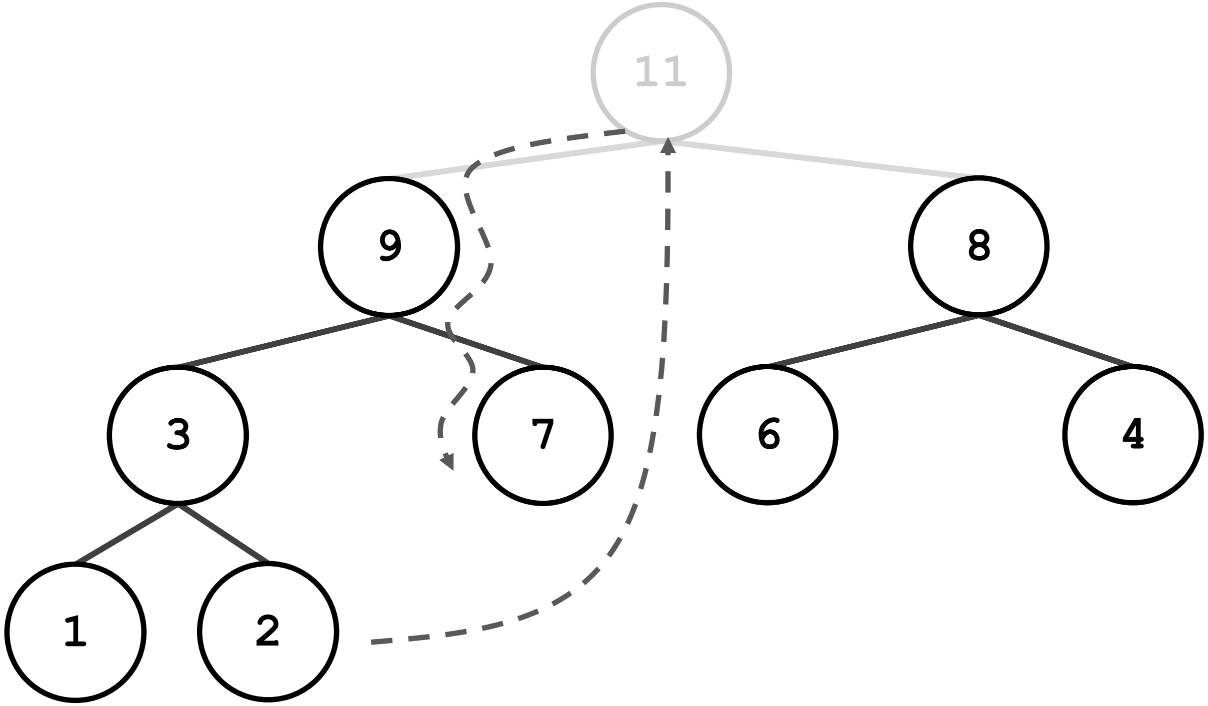
Ausgabe: 17

Heap-Sort: Beispiel (II)



Ausgabe: 17 14

Heap-Sort: Beispiel (III)



Ausgabe: 17 14 11 usw.

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 (bzw. Fehlermeldung, wenn Queue leer)
`extract-max(Q)`	-	gibt „größtes“ Element aus Queue **`Q`** zurück und löscht es aus Queue (bzw. Fehlermeldung, wenn Queue leer)
`insert(Q,k)`	-	fügt Wert **`k`** zu Queue **`Q`** hinzu

Priority Queues in Java

OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX H

SUMMARY: NESTED | FIELD | CONSTR | METHOD   DETAIL: FIELD | CONSTR | METHOD

Module java.base
Package java.util

Class PriorityQueue<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractQueue<E>
            java.util.PriorityQueue<E>

Type Parameters:

E - the type of elements held in this queue

All Implemented Interfaces:

Serializable, Iterable<E>, Collection<E>, Queue<E>

```
public class PriorityQueue<E>
extends AbstractQueue<E>
implements Serializable
```

An unbounded priority queue based on a priority heap. The elements of the priorit
ordering, or by a Comparator provided at queue construction time, depending on w
permit null elements. A priority queue relying on natural ordering also does not p
may result in ClassCastException).

Java 22 Class Priority Queue

Mit den Heaps erreichen wir ohne aufwändiges Balancieren dennoch Laufzeiten $O(\log n)$ für die Operationen Einfügen und Löschen. Sind AVL-Bäume+Co also überflüssig?

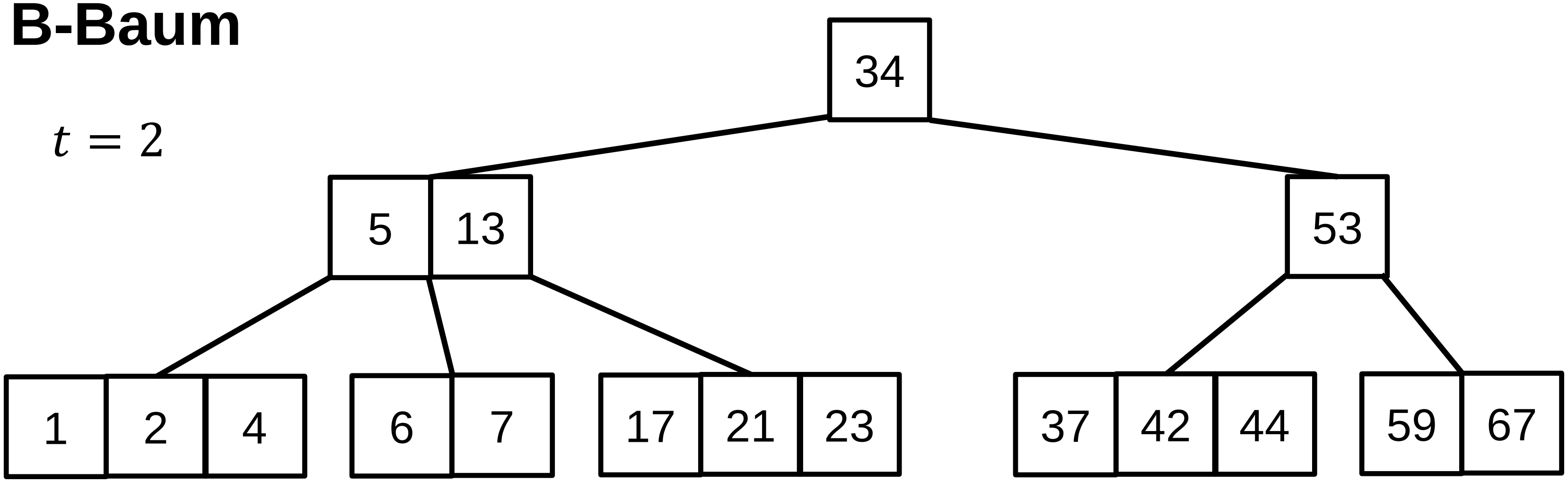
Warum kann man für Heap-Sort nicht einfach am Ende das Array ausgeben?

B-Bäume

B für Balanced?
B für Bayer-McCreight (1972)?
B für Boeing?
…

B-Baum

$t = 2$



Ein **B-Baum** (vom 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 und die Wurzel zwischen 1 und $2t-1$,

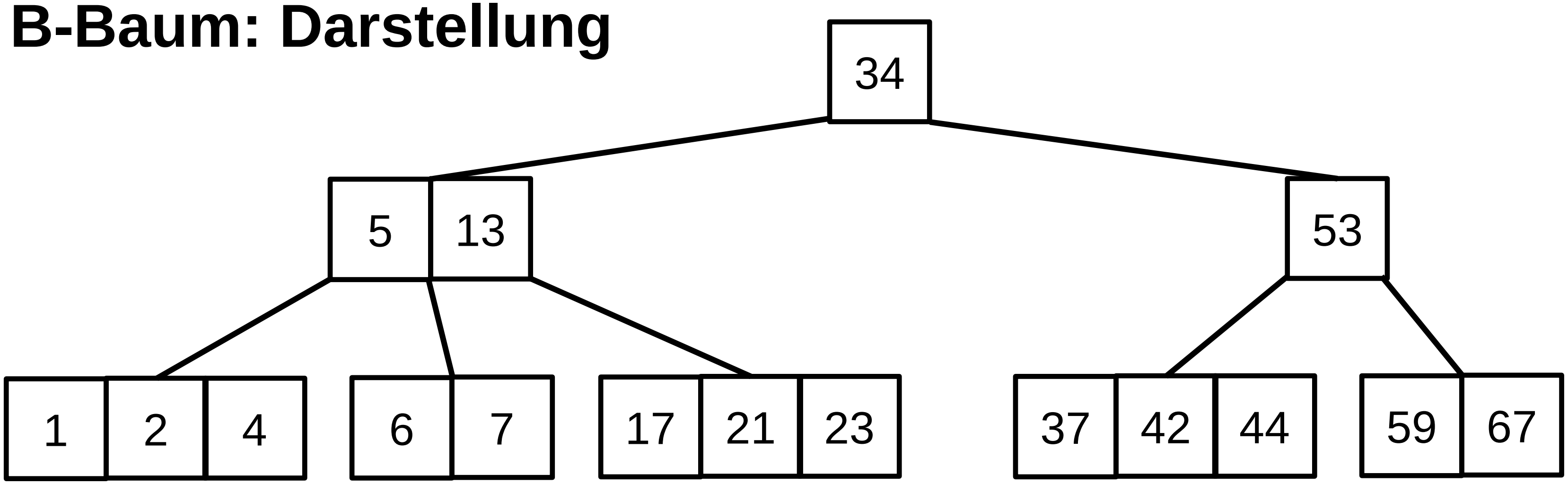
(2) die Werte innerhalb eines Knoten aufsteigend geordnet sind,

(3) die Blätter alle die gleiche Höhe haben, und

(4) jeder innere Knoten mit m Werten $m+1$ Kinder hat, so dass für alle Werte k_j aus dem j-ten Kind gilt:

$k_0 \leq$ **`key[0]`** $\leq k_1 \leq$ **`key[1]`** $\leq \cdots \leq k_{m-1} \leq$ **`key[m-1]`** $\leq k_m$

B-Baum: Darstellung

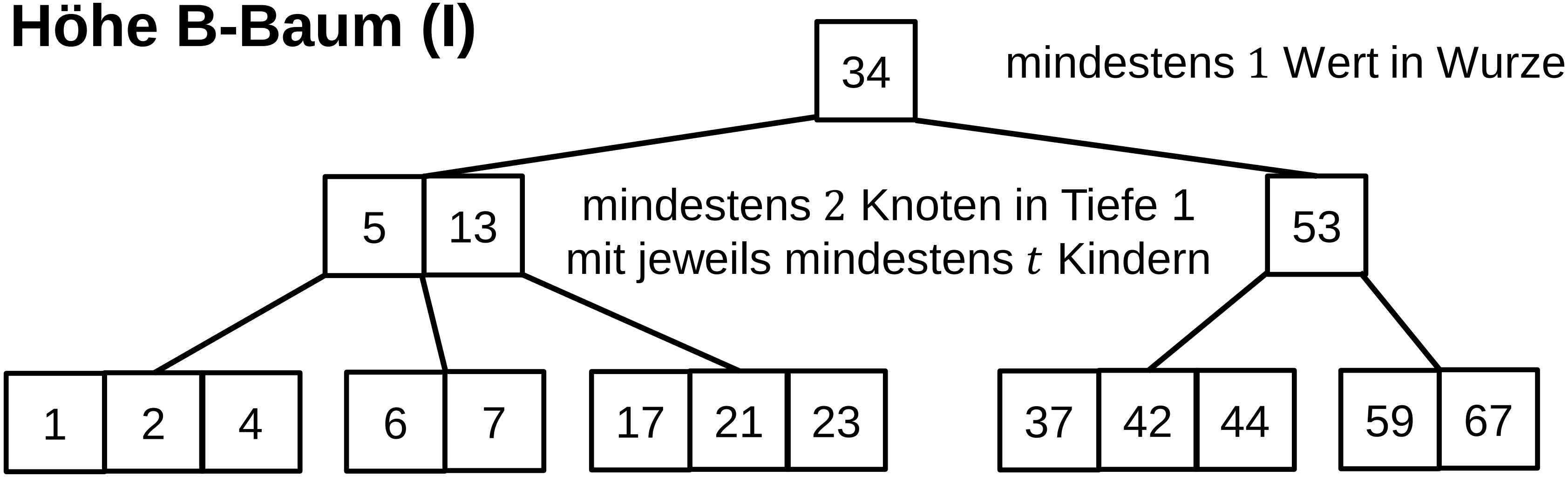


`x.n` – Anzahl Werte eines Knoten **`x`**

`x.key[0],…,x.key[x.m-1]` – (geordnete) Werte in Knoten **`x`**

`x.child[0],…,x.child[x.m]` – Zeiger auf Kinder in Knoten **`x`**

Höhe B-Baum (I)



mindestens $2t$ Knoten in nächster Tiefe mit jeweils mindestens t Kindern,
mindestens $2t^2$ Knoten in nächster Tiefe mit jeweils mindestens t Kindern,
mindestens $2t^3$ Knoten in nächster Tiefe mit jeweils mindestens t Kindern, usw.

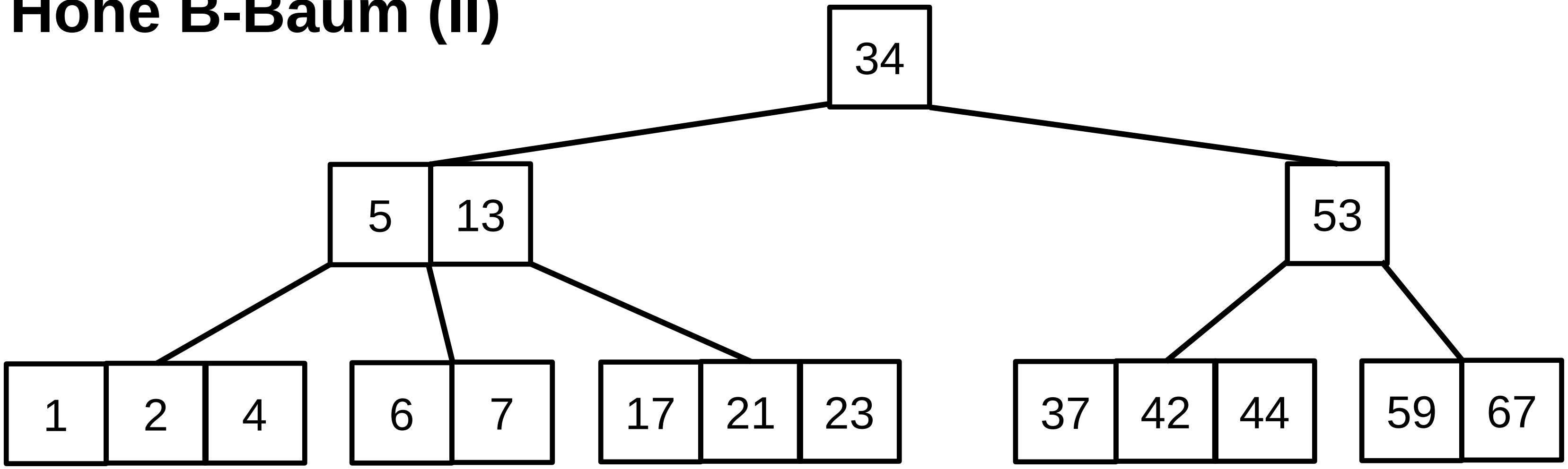
Und in jedem Knoten außer Wurzel mindestens $t-1$ Werte.

Anzahl Werte n im B-Baum im Vergleich zu Höhe h:

$$n \geq 1 + (t-1) \cdot \sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1) \cdot \frac{t^h - 1}{t-1} = 2t^h - 1$$

also: $\log_t \frac{n+1}{2} \geq h$. ∎

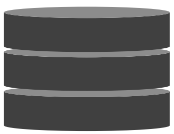
Höhe B-Baum (II)



Ein B-Baum vom Grad t mit n Werten hat maximale Höhe $h \leq \log_t \frac{n+1}{2}$.

Für größere Werte für t also „flacher“ als (vollständiger) Binärbaum.

Anwendung



Lesen/Schreiben in Blöcken von 512 bis 16K Bytes → mehrere Werte (z.B. Index-Einträge) auf einmal

Beispiel:
4KB Blöcke / 32 Bytes pro Eintrag = 128 Einträge pro Knoten, $t = 64$.
Bei 10^{12} Werten hat B-Baum die Höhe $\mathrm{h} \leq 7$.

Quelle: **de.wikipedia.org**

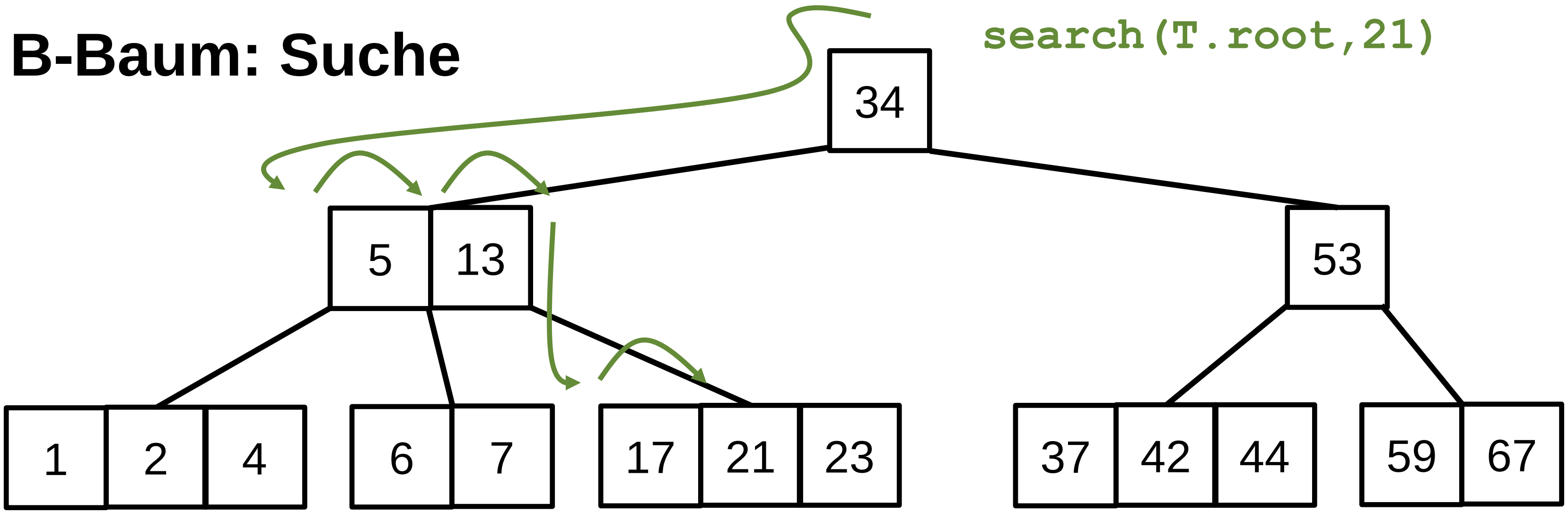
10.3.1 How MySQL Uses Indexes

Indexes are used to find rows with specific column values quickly. Without an index, MySQL must begin with the first row and then read through the entire table to find the relevant rows. The larger the table, the more this costs. If the table has an index for the columns in question, MySQL can quickly determine the position to seek to in the middle of the data file without having to look at all the data. This is much faster than reading every row sequentially.

Most MySQL indexes (`PRIMARY KEY`, `UNIQUE`, `INDEX`, and `FULLTEXT`) are stored in B-trees. Exceptions: Indexes on spatial data types use R-trees; `MEMORY` tables also support hash indexes; `InnoDB` uses inverted lists for `FULLTEXT` indexes.

Reference Manual

B-Baum: Suche




```
search(x,k)

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

maximal
$2t = O(1)$
Iterationen

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

Baumkunde

alternative Definition:
max t und mind $t/2$ Kinder
pro innerem Knoten $\neq$ Wurzel

B-Baum vom Grad t

max $2t$, mind t Kinder
pro innerem Knoten $\neq$ Wurzel

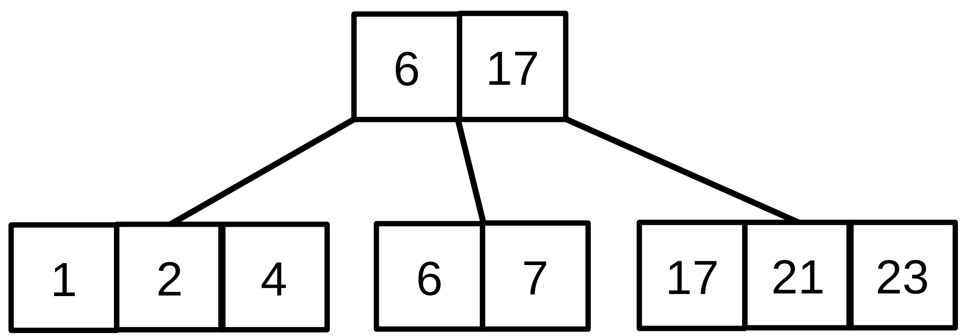
2-3-4-Baum
oder (2,4)-Baum
=
B-Baum für $t = 2$

alternativer Bedeutung B*-Baum:
B-Baum, mit Füllgrad mind. 2/3
pro innerem Knoten $\neq$ Wurzel

alternativer Name:
B*-Baum

B+-Baum

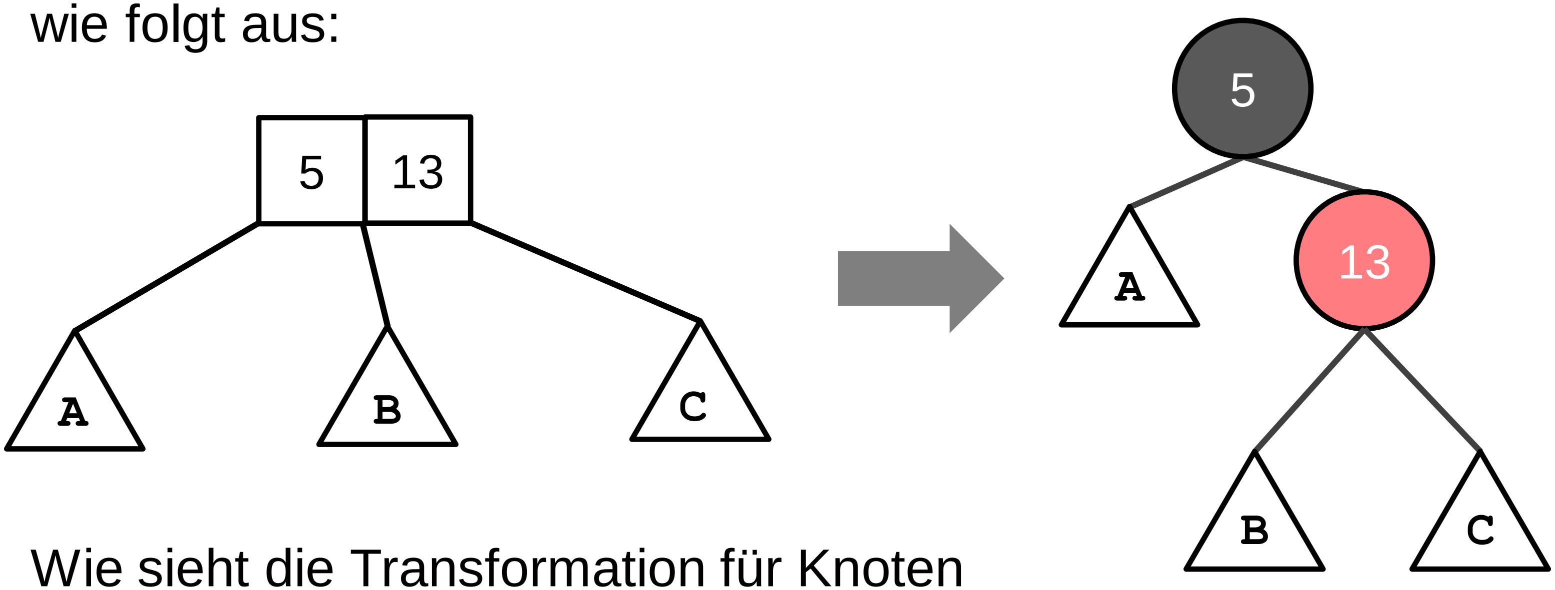
alle Werte in Blättern, innere
Knoten enthalten Werte erneut



Vorteil: innere Knoten speichern
nur kurzen Schlüssel, nicht
auch noch Daten(-zeiger)

Nachteil: Findet Werte erst im Blatt

Man kann B-Bäume vom Grad $t = 2$ in Rot-Schwarz-Bäume überführen. Für Knoten mit zwei Werten sieht die Transformation wie folgt aus:

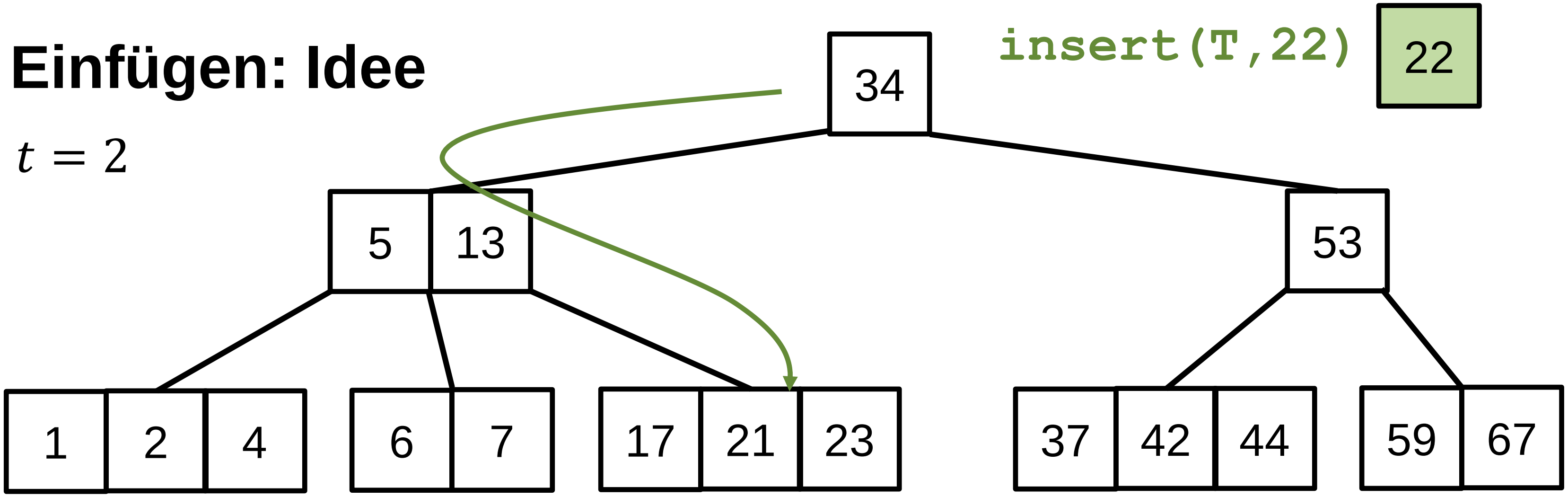


Wie sieht die Transformation für Knoten mit einem bzw. drei Werten aus?

Argumentieren Sie, dass bei rekursiver Anwendung auf die Teilbäume ein korrekter Rot-Schwarz-Baum entsteht. Welche Höhe hat er?

Einfügen: Idee

$t = 2$

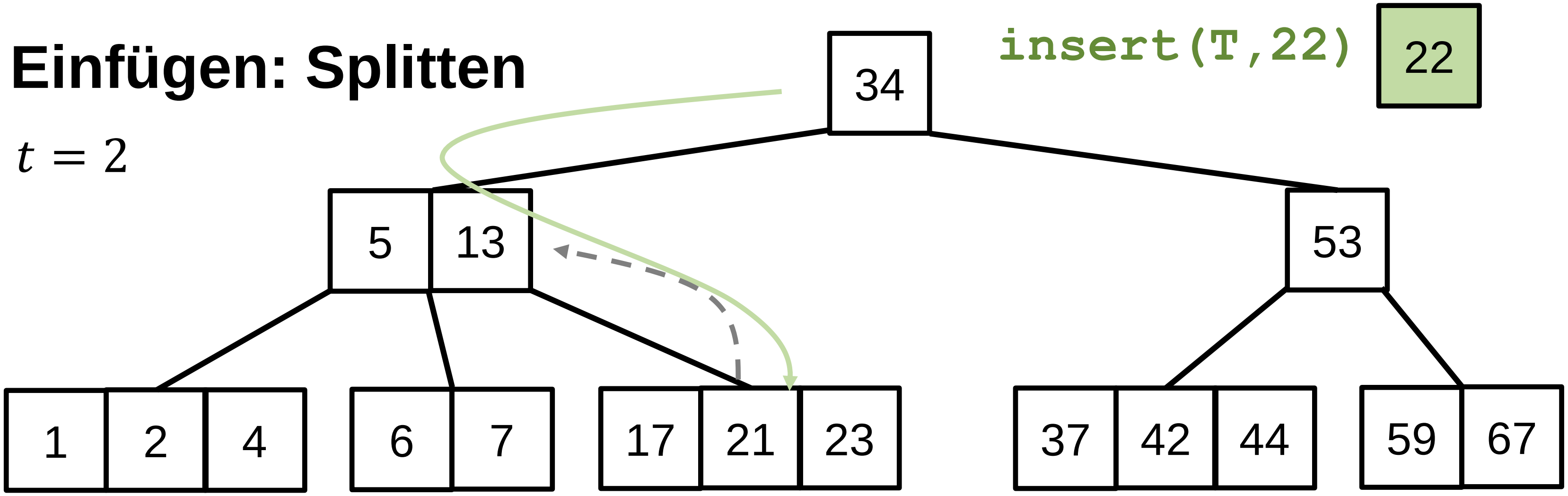


Prinzip: Einfügen erfolgt immer in einem Blatt

Wenn Blatt weniger als $2t - 1$ Werte, dann einfügen – fertig.
Sonst…

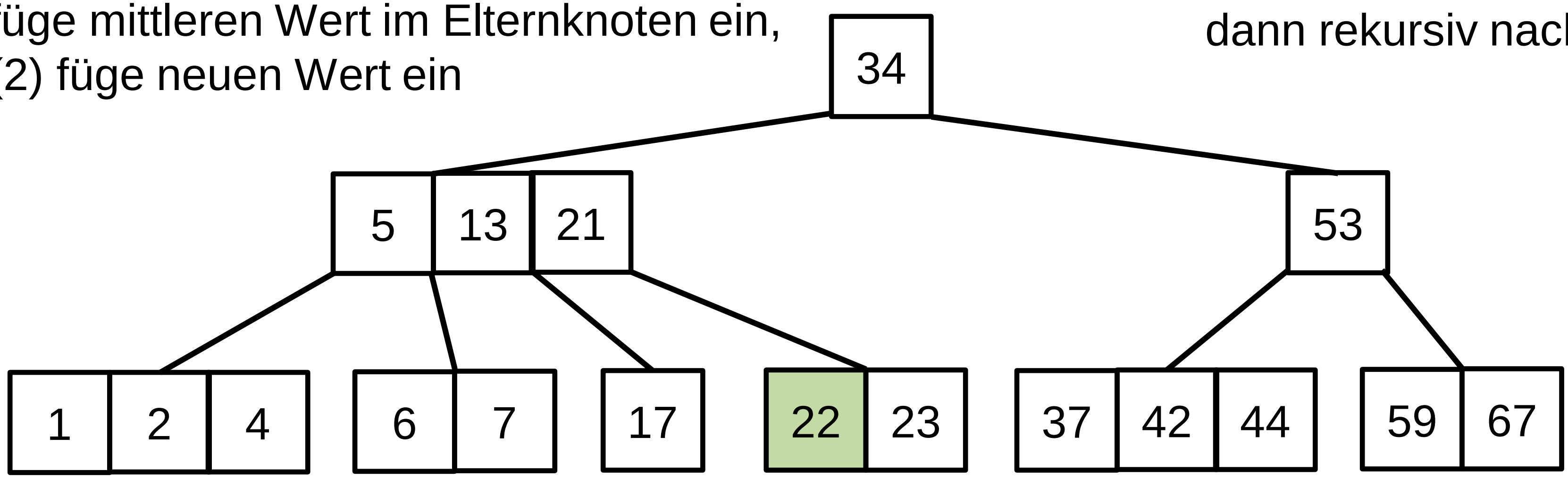
Einfügen: Splitten

$t = 2$



Wenn Blatt bereits $2t - 1$ Werte, dann
(1) teile in zwei Blätter mit je $t - 1$ Werten, füge mittleren Wert im Elternknoten ein,
(2) füge neuen Wert ein

Wenn dadurch Elternknoten mehr als $2t - 1$ Werte, dann rekursiv nach oben



An der Wurzel splitten

$t = 2$

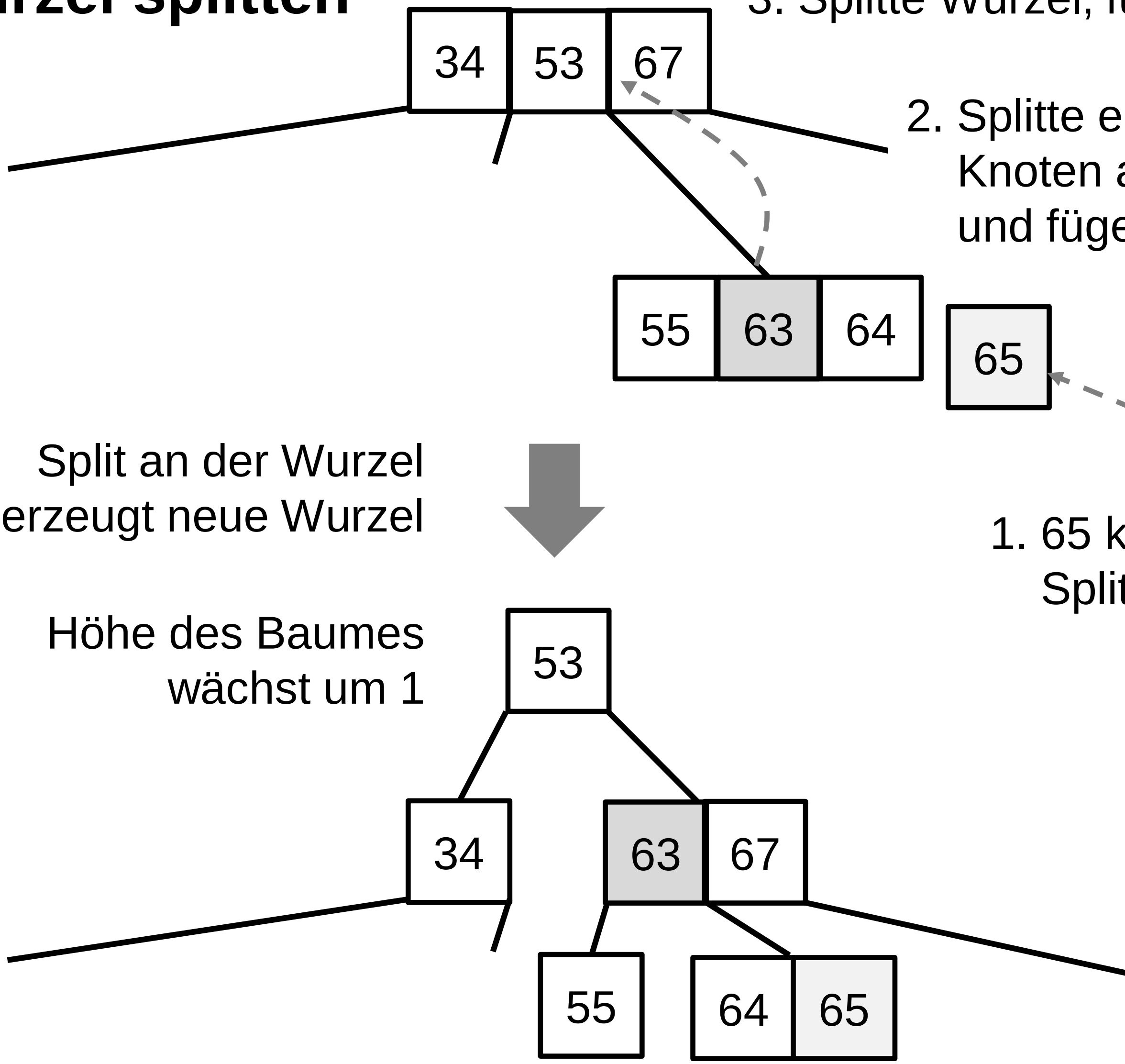
3. Splitte Wurzel, füge 63 ein

2. Splitte erst den
   Knoten an der 63
   und füge 65 rechts ein

1. 65 kommt von
   Split weiter unten

Split an der Wurzel
erzeugt neue Wurzel

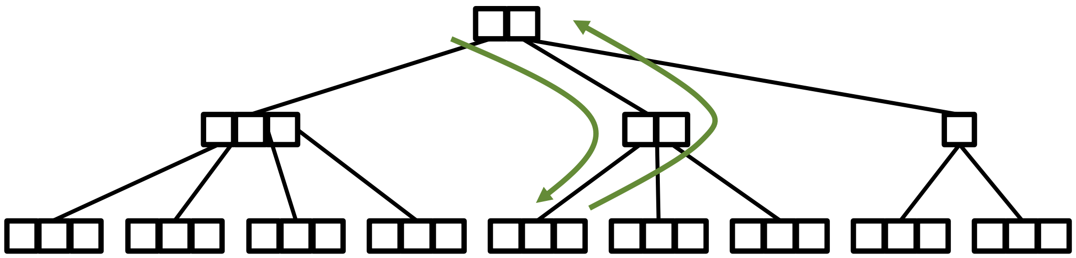
Höhe des Baumes
wächst um 1



„Suchen, dann Splitten“ vs. „Suchen und Splitten“

Suche, splitte dann:

1. Suche (abwärts)
2. Splitte (aufwärts)



Teure Disk-Operationen werden zweimal ausgeführt, einmal beim Absteigen, einmal auf Aufsteigen.

B-Baum-Einfügen splittet daher beim Suchen und läuft nur einmal hinab.

B-Baum Einfügen (informell)

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

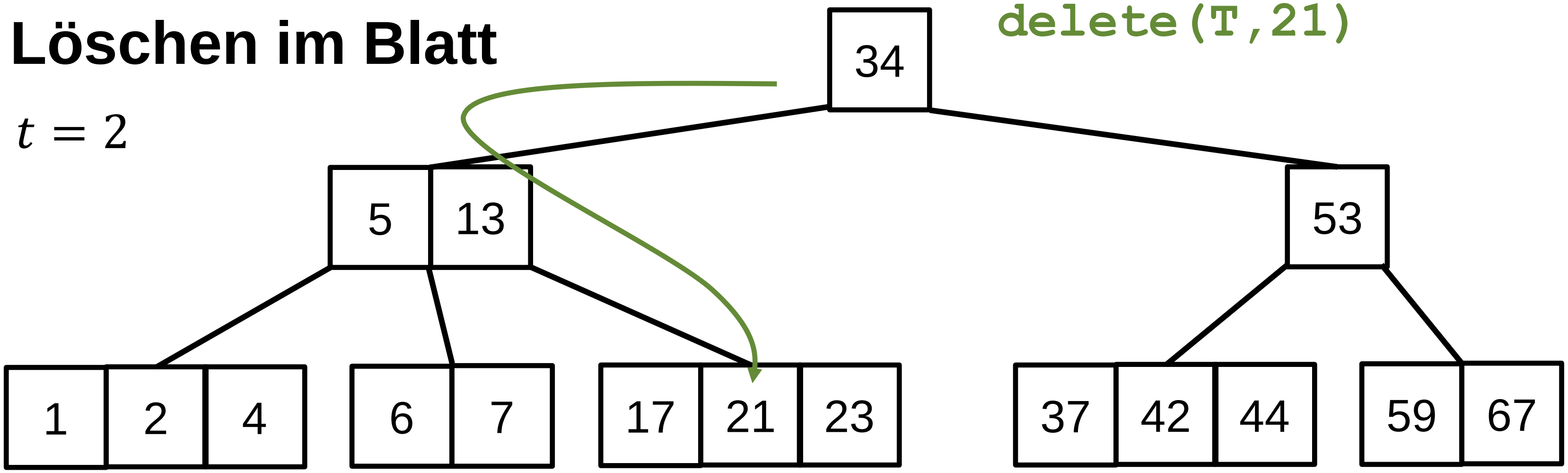
```
insert(T,z)

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

Aktueller Knoten hat zu diesem Zeitpunkt weniger als $2t-1$ Werte, sonst wäre er vorher geteilt worden. Beim Splitten des Kindes kann der mittlere Wert daher problemlos im aktuellen Knoten eingefügt werden.

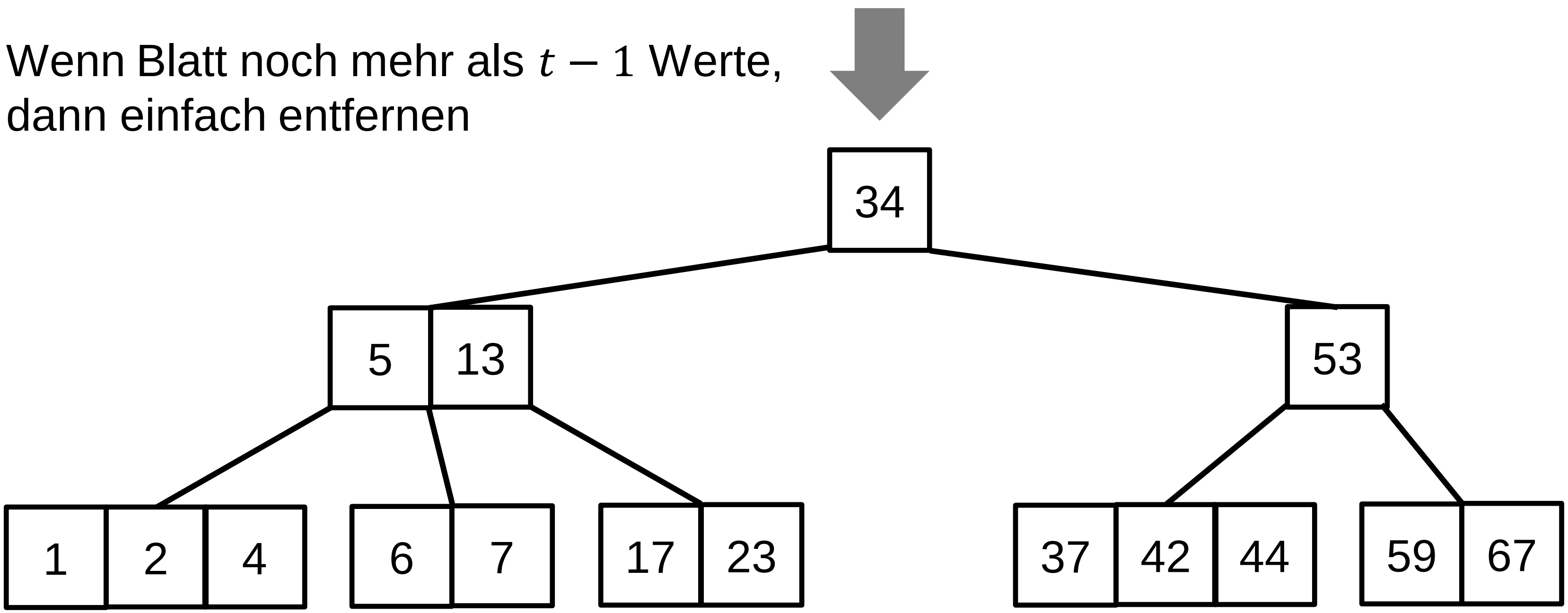
Auch Blatt hat zu diesem Zeitpunkt weniger als $2t-1$ Werte.

Löschen im Blatt

$t = 2$

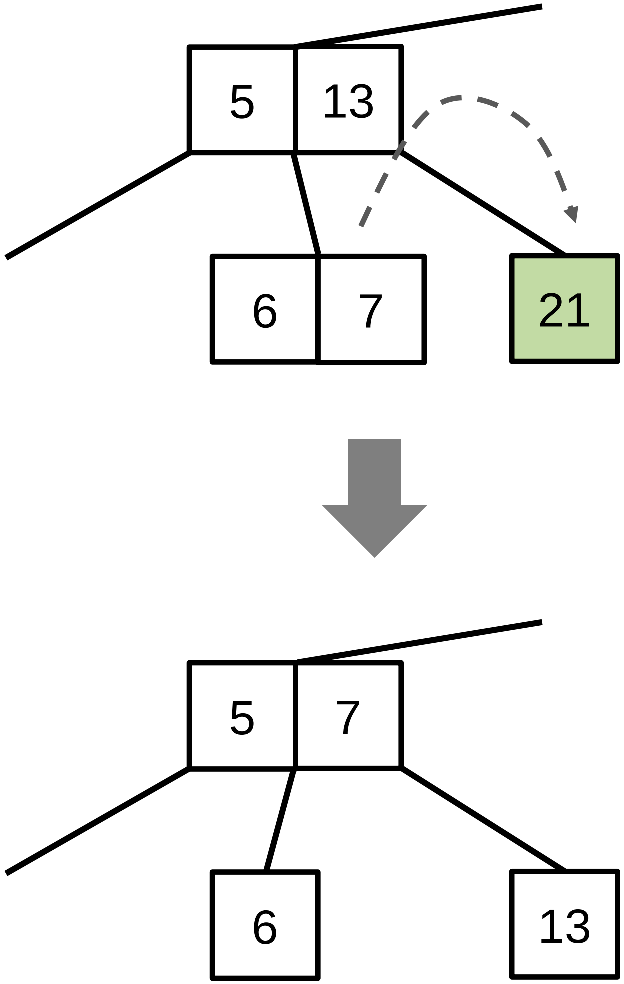


Wenn Blatt noch mehr als $t - 1$ Werte, dann einfach entfernen



Löschen im „zu leeren“ Blatt: Rotieren

$t = 2$



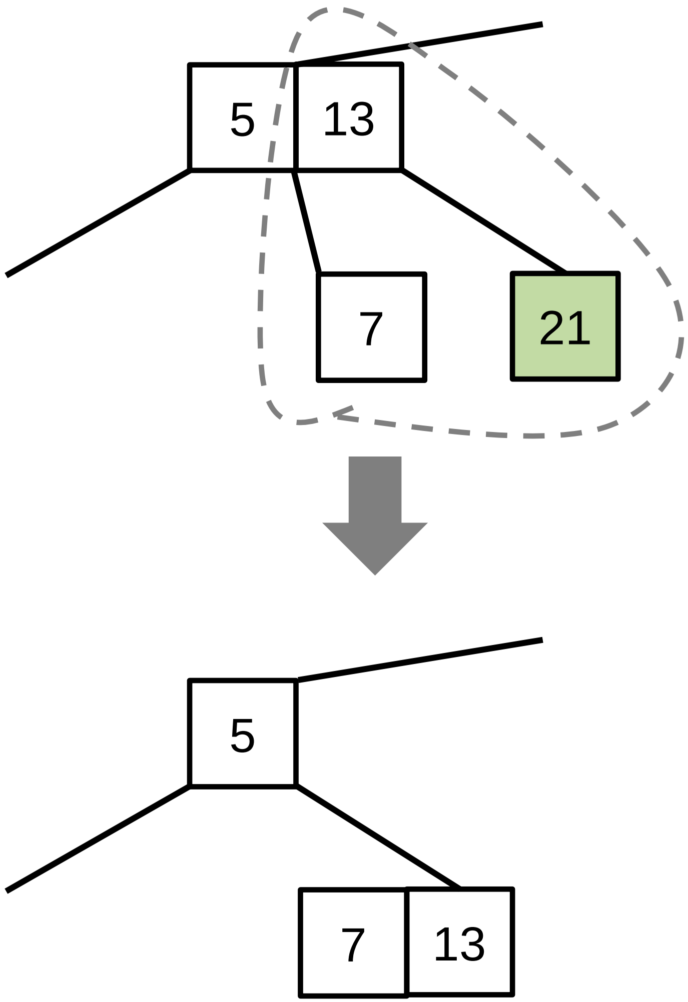
jeweils mind. $t - 1$ Werte

Wenn $t - 1$ Werte im Blatt mit zu löschendem Wert, und linker oder rechter Geschwisterknoten hat mind. t Werte, dann rotiere Werte von Geschwisterknoten und Elternknoten

Wenn linker Geschwisterknoten, dann rotiere größten Wert in dem Knoten. Wenn rechter Geschwisterknoten, dann rotiere kleinsten Wert in dem Knoten.

Löschen im „zu leeren“ Blatt: Verschmelzen

$t = 2$



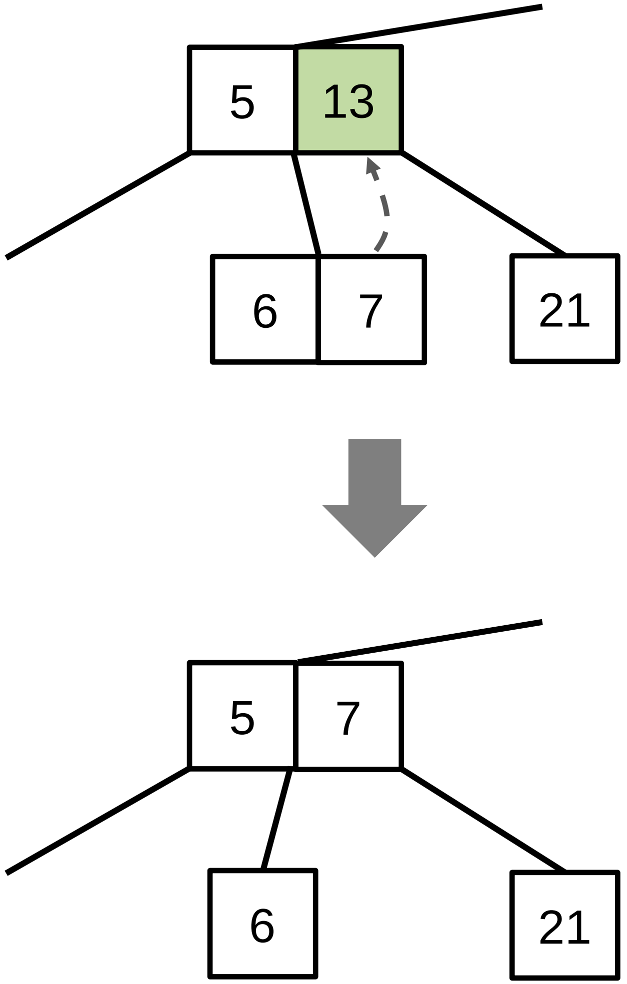
Wenn $t-1$ Werte im Blatt
mit zu löschendem Wert,
und linker und rechter
Geschwisterknoten auch $t-1$ Werte,
dann verschmelze ein Geschwister
mit Wert aus Elternknoten

Eventuell hat Elternknoten
nun zu wenig Werte

maximal $\mathrm{t}-2+\mathrm{t}-1+1=2t-2$ Werte

Löschen im inneren Knoten: Verschieben

$t = 2$

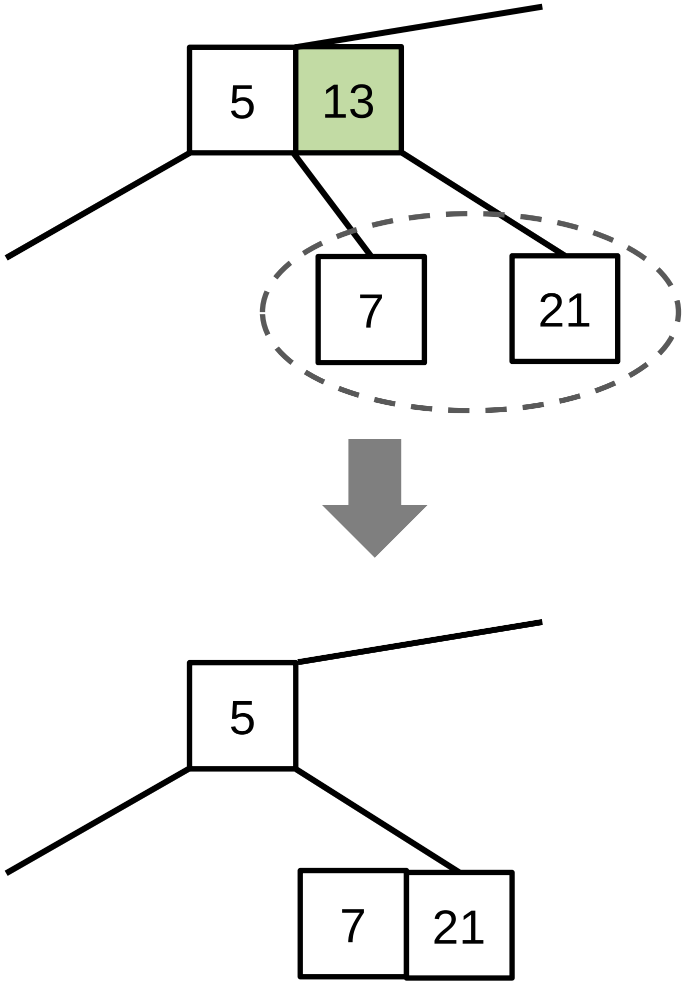


jeweils mind. $t - 1$ Werte

Wenn mehr als $t - 1$ Werte in einem der beiden Kindknoten, dann größten Wert (vom linken Kind) bzw. kleinsten Wert (vom rechten Kind) nach oben kopieren

Löschen im inneren Knoten: Verschmelzen

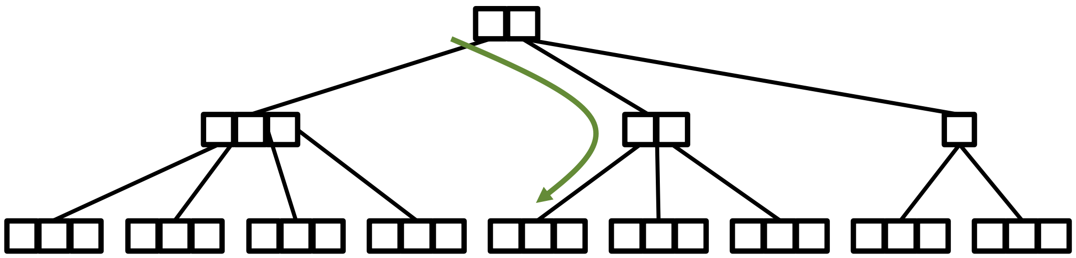
$t = 2$



Wenn jeweils $t-1$ Werte in beiden Kindknoten, dann Kindknoten verschmelzen

Eventuell hat Elternknoten nun zu wenig Werte

„Löschen und Splitten“

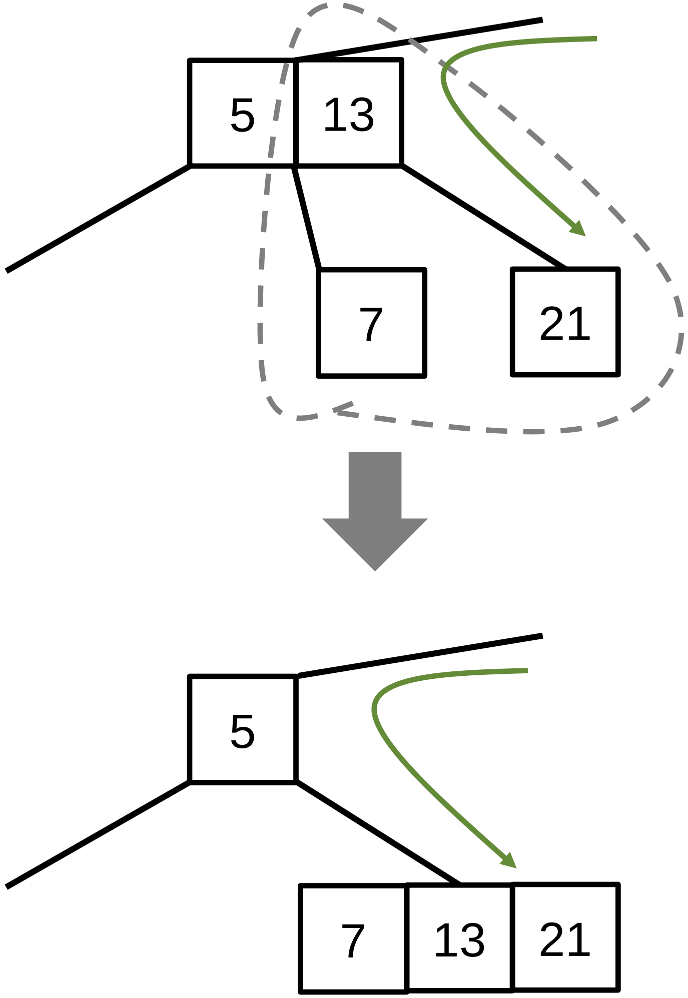


B-Baum-Löschen läuft auch nur einmal hinab.

Stelle dazu sicher, dass zu
besuchendes Kind mindestens t Werte hat.

Allgemeines Verschmelzen (ohne Löschen)

$t = 2$



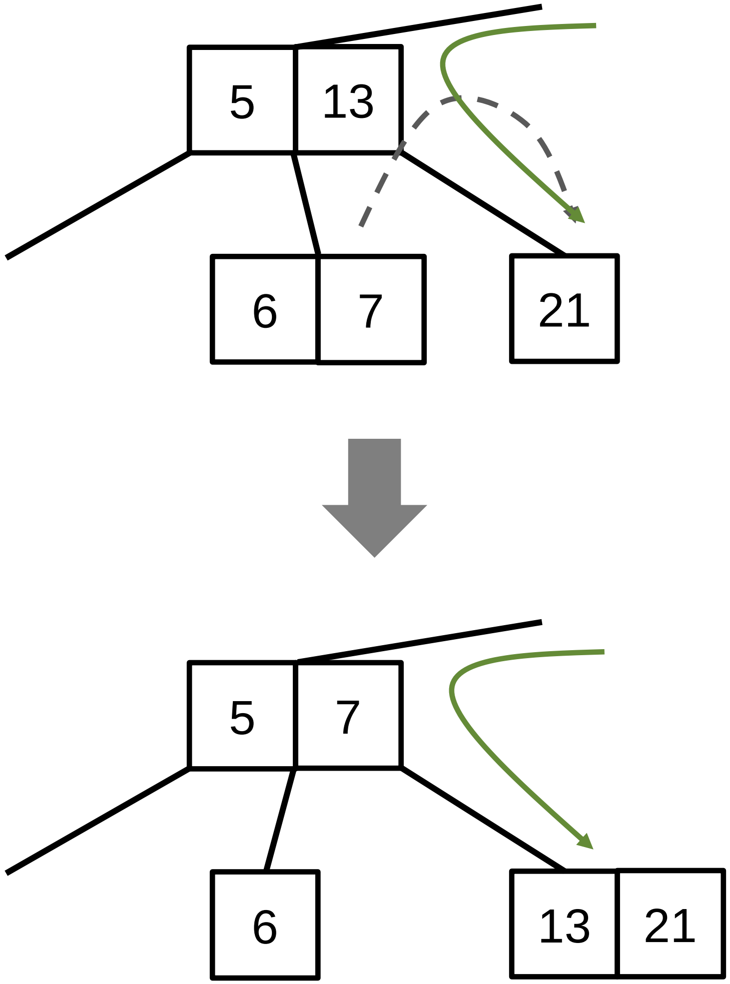
mind. t Werte, max. $(t-1) + (t-1) + 1 = 2t - 1$ Werte

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

Wenn Elternknoten vorher mindestens t Werte, dann keine Änderung oberhalb nötig

Allgemeines Rotieren/Verschieben (ohne Löschen)

$t = 2$



mind. $t-1$ Werte mind. t Werte

Zu besuchendes Kind nur $t-1$ Werte, aber ein Geschwisterknoten mehr als $t-1$ Werte

Keine Änderung oberhalb nötig

B-Baum Löschen (informell)

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

```
delete(T,k)

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

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, da mindestens t Werte. Entfernen im inneren Knoten durch Verschieben oder Verschmelzen.

Worst-Case-Laufzeiten

B-Bäume

Operation	Laufzeit
Einfügen	$\Theta(\log_t n)$
Löschen	$\Theta(\log_t n)$
Suchen	$\Theta(\log_t n)$

Achtung:
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 in der Regel nur vorteilhaft, wenn Daten blockweise eingelesen werden.

Anhang:
Algorithmen Löschen in Rot-Schwarz-Bäumen

Achtung: wird/wurde nicht in Vorlesung explizit vorgestellt

Löschen: Transplant mit Sentinels

```
transplant(T,u,v) //with Sent

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;
```

Zur Erinnerung:
funktioniert auch, wenn
v==nil

Löschen: Algorithmus (I)

a Zeiger auf Knoten, in dem „tiefste“ Imbalance entstehen könnte

dsh =ΔSH für Knoten **a**
(**nil**=0, **left**=+1, **right**=–1)

```
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
   y=z.left;
   transplant(T,z,z.left);
   y.color=z.color;
ELSE …
```

In den Fällen muss **y.color==red** sein, sonst SH-Regel verletzt; einfach **y** umhängen und Farbe von **z** kopieren

Löschen: Algorithmus (II)

Analog zu BST

```
...
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);
```

Fallunterscheidung nach **y** rechtes oder linkes Kind

erzeugt Imbalance,
je nachdem, ob **y**
rechtes oder linkes Kind war

Löschen: Farben korrigieren (I)

```
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.color=black;
  ELSE IF a.color==black AND b.color==red THEN
     rotateLeft(T,a);
     a.color=red; b.color=black;
     fixColorsAfterDeletion(T,a,dsh);
  ELSE IF a.color==red AND b.color==black
          AND (c==nil OR c.color=black)
          AND (d==nil OR d.color=black) THEN
     a.color=black; b.color=red;
ELSE IF …
```

notwendig: **b!=nil** für **dsh=right**

einfacher Fall: **x** ist rot

Fall I: **a** schwarz, **b** rot

Fall IIa: **a** rot, **b** schwarz, **c,d** nicht rot

außer im Fall IIb führen rekursive Aufrufe im nächsten Schritt zum Rekursionsende

Löschen: Farben korrigieren (II)

```
fixColorsAfterDeletion(T,a,dsh)
…
  ELSE IF a.color==black AND b.color==black
       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
       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
    rotateLeft(T,a);
    b.color=a.color; a.color=black; d.color=black;
ELSE // dsh==left, extra black node on the left
  … //exchange left and right
```

Fall IIb: **`a`**, **`b`** schwarz, **`c`**,**`d`** nicht rot

Fall III: **`b`** schwarz, **`c`** rot, **`d`** nicht rot

Fall IV: **`b`** schwarz, **`d`** rot

Fall linkslastig