Algorithmen und Datenstrukturen
Kapitel 6B: Balancierte Bäume

Prof. Dr. Wolfgang Mühlbauer
Fakultät für Informatik
wolfgang.muehlbauer@th-rosenheim.de

Wintersemester 2019/2020

Übersicht

- Binäre Suchbäume
  - Siehe Kapitel 7A

- **Balancierte Binärbäume**
  - **Rot-Schwarz-Bäume**
  - Rotationen
  - Einfügen in Rot-Schwarz-Bäumen

- B-Bäume
  - Siehe Kapitel 7C

Balancierte Binärbäume

- **Basisoperationen** (Einfügen, Löschen, Suchen) von binären Suchbäumen: $\boldsymbol{O(h)}$
  - Worst Case: Binärer Suchbaum zu linearer Liste degeneriert (*h* = *n*).
  - Datenstruktur nur effizient, falls binärer Suchbaum ***balanciert.***

- **Idee**: Halte binären Suchbaum balanciert!
  - Balanciert: Höhe $h = \Theta(\log n)$!
  - Ggfs. Reorganisation des Baumes nach Einfügen / Löschen.
  - Kompromiss zwischen
    - Aufwand zur Reorganisation des Baumes und
    - schneller Laufzeit für Basisoperationen.

- **Verschiedene Ansätze**
  - AVL Bäume, 1962
  - Red-Black Trees (dt. "Rot-Schwarz-Bäume"), 1972

Red-Black Tree (dt. "Rot-Schwarz-Baum")

- **Attribute** eines Knoten
  - **`color`** (1Bit): "rot" und "schwarz" als mögliche Farben eines Knotens
  - Andere Attribute wie bislang: **`key, left, right, p`**
  - Schlüssel ***nur in inneren Knoten*** gespeichert.
    - Anheften von externen Blättern, um Algorithmus kompakter zu formulieren.
    - Wo man normal auf "null" verweist, zeigt man nun auf ein spezielles, externes Terminierungsblatt "T.NULL" (siehe nächste Folien).

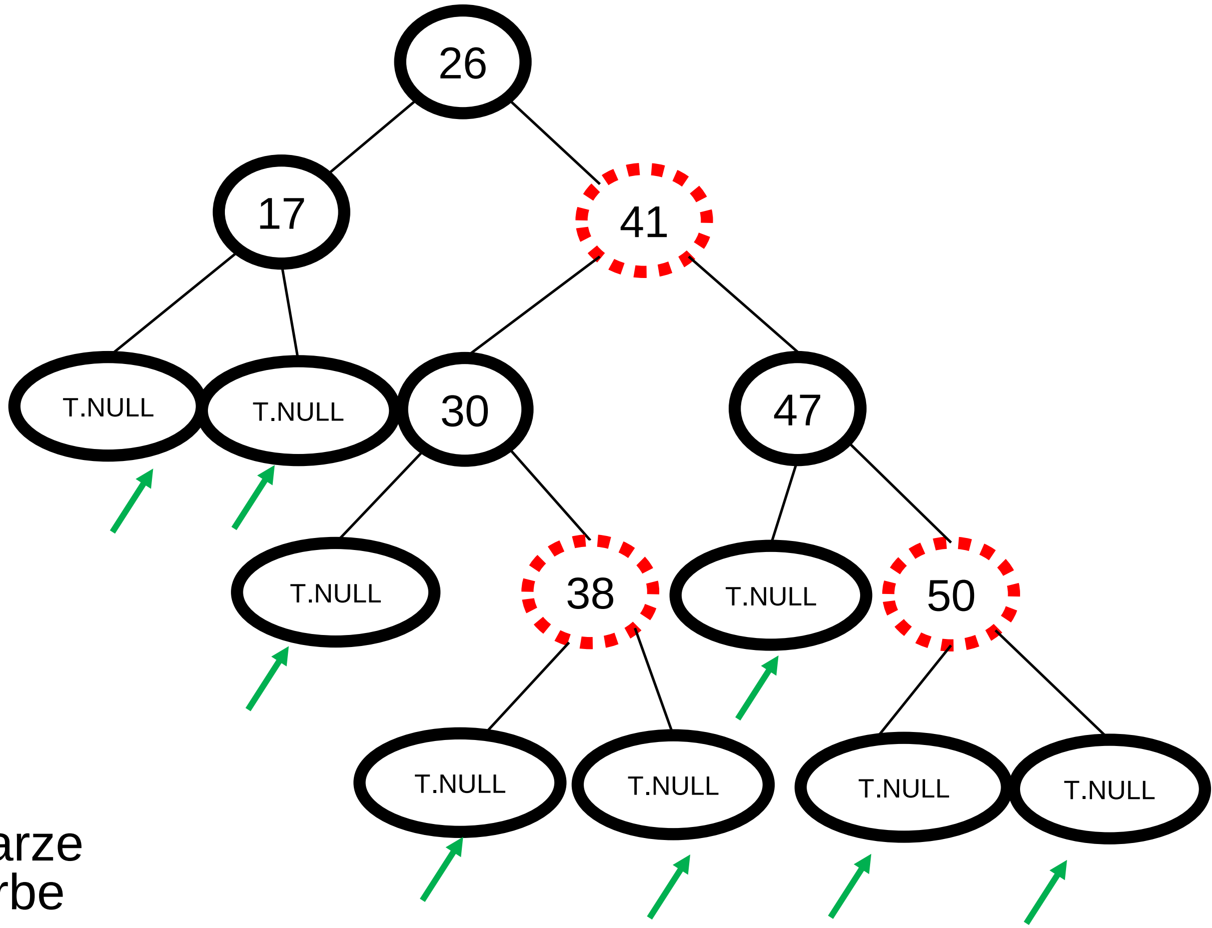
- **Red-Black Tree Eigenschaften:**
  1) Jeder Knoten ist rot oder schwarz.
  2) Die Wurzel ist schwarz.
  3) Jedes Blatt ist schwarz.
  4) Falls ein Knoten rot ist, dann sind seine beiden Kinder schwarz.
  5) Für jeden Knoten enthalten ***alle*** Pfade von diesem Knoten zu seinen ***nachfolgenden*** Blättern ***gleich viele schwarze Knoten***.

- Erfüllt Baum diese Eigenschaften, ist er einigermaßen gut balanciert!

Beispiel: RB-Tree

- **Red-Black Tree Eigenschaften:**
  1) Jeder Knoten rot oder schwarz.
  2) Wurzel schwarz.
  3) Jedes Blatt schwarz.
  4) Knoten rot → alle Kinder schwarz.
  5) Für jeden Knoten: In ***allen*** Pfade zu ***nachfolgenden*** Blättern ***gleich viele schwarze Knoten***.

- **Konvention für Klausur**
  - Sowohl rote als auch schwarze Knoten mit der gleichen Farbe zeichnen.
  - Rote Knoten dann gestrichelt zeichnen.

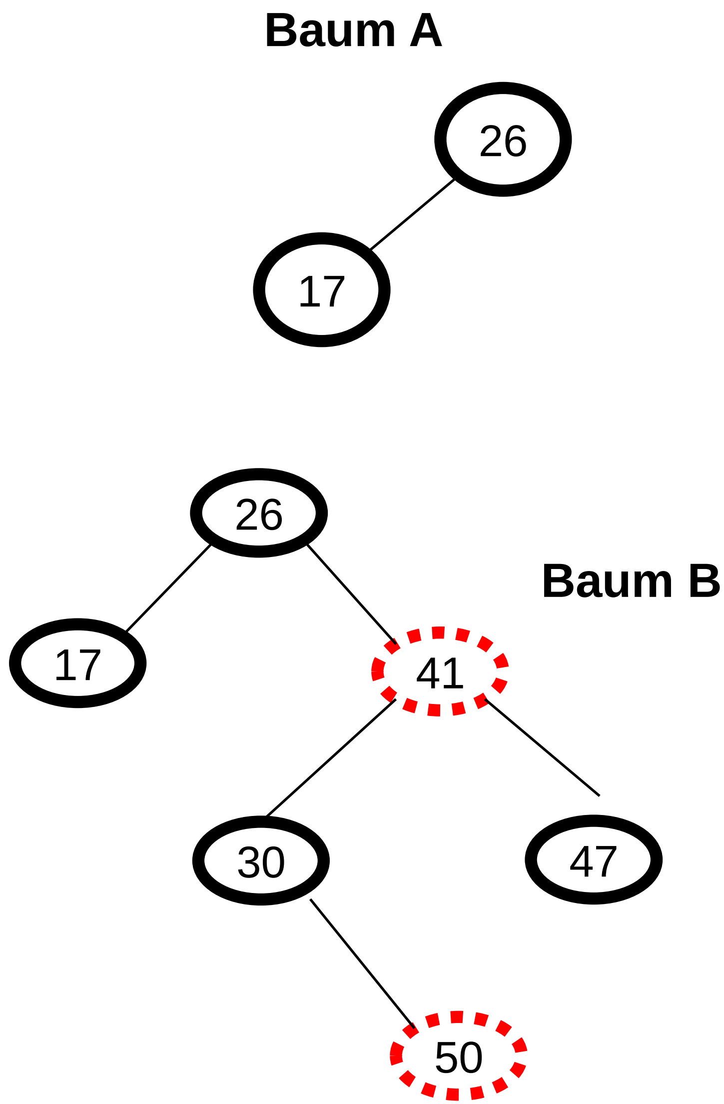


Was sind hier die Blätter? Es wird künstlich erreicht, dass alle Blätter schwarz sind.

Publikums-Joker

Sind Bäume A und B Rot-Schwarz-Bäume?

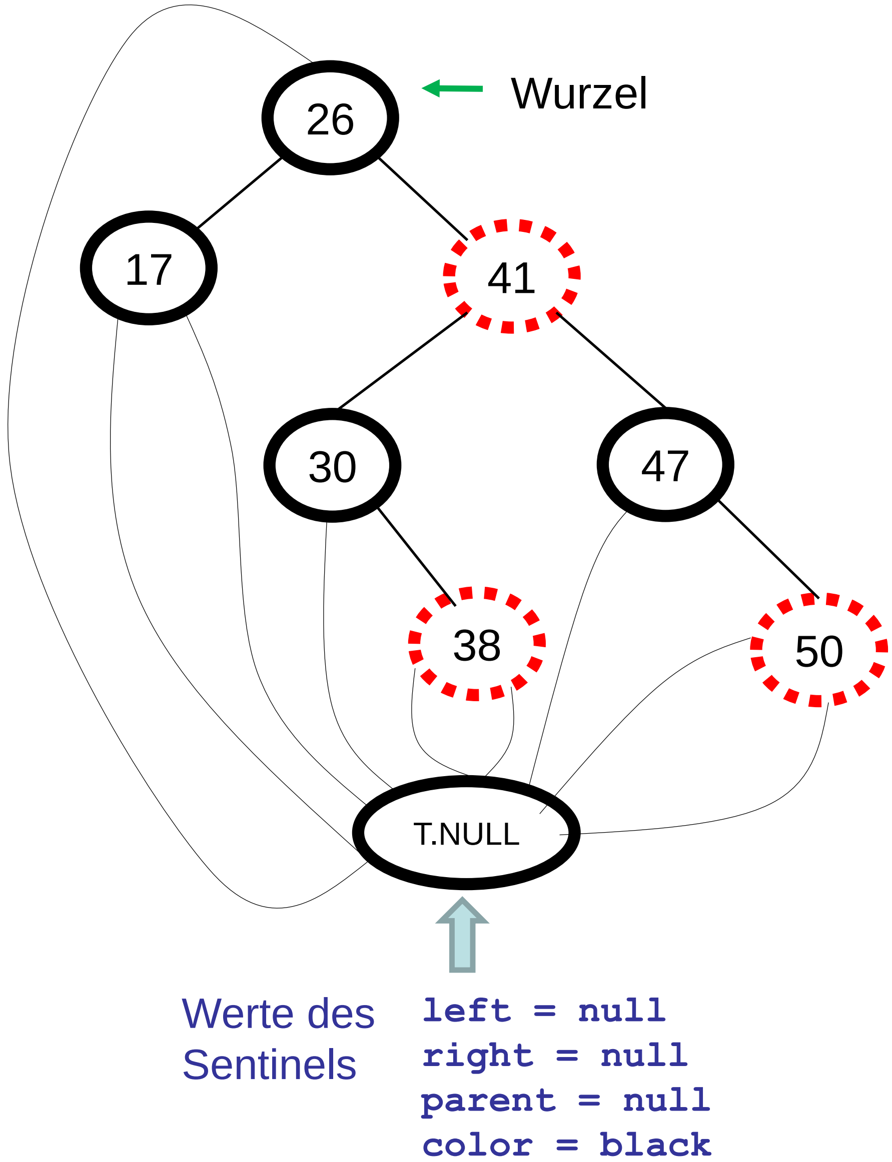
A. Baum A: Ja Baum B: Ja
B. Baum A: Nein Baum B: Ja
C. Baum A: Ja Baum B: Nein
D. Baum A: Nein Baum B: Nein



Wächterwert: T.NULL-Knoten

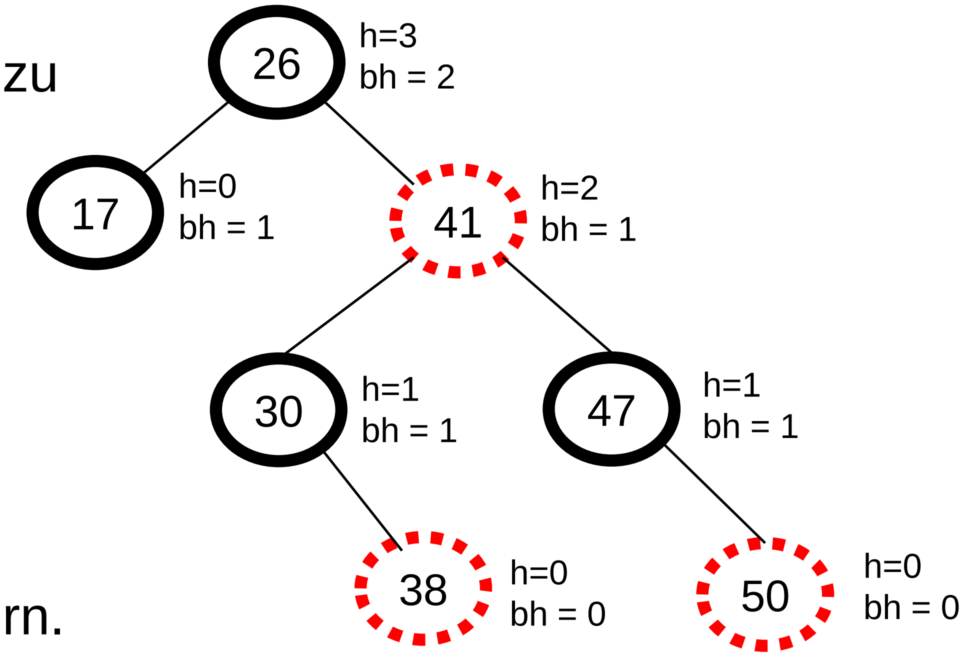
- ***Wächterwert***
  - Programmiertrick
  - Vermeidet `if`-Abfragen und gesonderte Fallunterscheidungen.
  - Vereinfacht Code.

- **`T.NULL`**
  - Schwarzer Knoten
  - Alle anderen Attribute `null`.
  - 1 solches `T.NULL` Objekt genügt, siehe rechte Grafik.
  - Meist im Folgenden nicht gezeichnet.



Höhe und Schwarztiefe

- **Höhe *h*** eines Knotens ***x***
  - Anzahl Knoten in **längstem** Pfad zu einem Blatt (ohne T.NULL)



- **Schwarztiefe *bh(x)*** (engl.: *black-height*) eines Knoten ***x***
  - Anzahl ***schwarzer*** Knoten auf Pfaden von *x* zu beliebigen Blättern.
    - x *wird mitgezählt,* T.NULL *nicht.*
  - Wegen Eigenschaft 5) wohldefiniert.
    - Schwarztiefen sind für jeden Pfad gleich.

T.NULL bzw. schwarze Blätter sind hier und im Folgenden jeweils nicht gezeichnet.

Warum ist RB-Tree immer gut balanciert?

- RB-Tree mit *n* Knoten hat Höhe $h \leq 2\log(n+1)$

- **Beweisidee:** Induktion
  - Details z.B: [1] oder https://de.wikipedia.org/wiki/Rot-Schwarz-Baum#H.C3.B6henbeweis

- **Vergleich zu AVL-Bäumen:**
  - Bei AVL-Bäumen gilt: Tiefe des rechten Teilbaumes und die Tiefe des linken Teilbaumes unterscheiden sich in jedem um maximal 1.
  - AVL-Baum mit *n* Knoten hat Höhe $1{,}44\ \log(n+1)$
  - Etwas bessere Balancierung, aber oft größerer "Overhead".

- **Konsequenz:**
  - RB-Tree recht gut balanciert $\rightarrow$ Suche in $O(h) = O(\log n)$!
  - Wie erreicht man, dass Baum nach Löschen und Einfügen balanciert bleibt?

Übung:

Zeichnen Sie einen Rot-Schwarz-Baum mit folgenden Eigenschaften:

- Die Höhe ist 5
- Der Baum enthält 14 Knoten
- Der Baum erfüllt die Rot-Schwarz Eigenschaften, ist aber recht schlecht balanciert.

Übersicht

- Binäre Suchbäume
  - Siehe Kapitel 7A

- Balancierte Binärbäume
  - Rot-Schwarz-Bäume
  - **Rotationen**
  - Einfügen in Rot-Schwarz-Bäumen

- B-Bäume
  - Siehe Kapitel 7C

Operationen auf RB-Trees

- `MIN, MAX, SUCCESSOR, PREDECESSOR, GET`
  - Identisch wie bei binären Suchbäumen.
  - Laufzeit jeweils in Zeit $\boldsymbol{O}(\log \boldsymbol{n})$, falls balanciert.

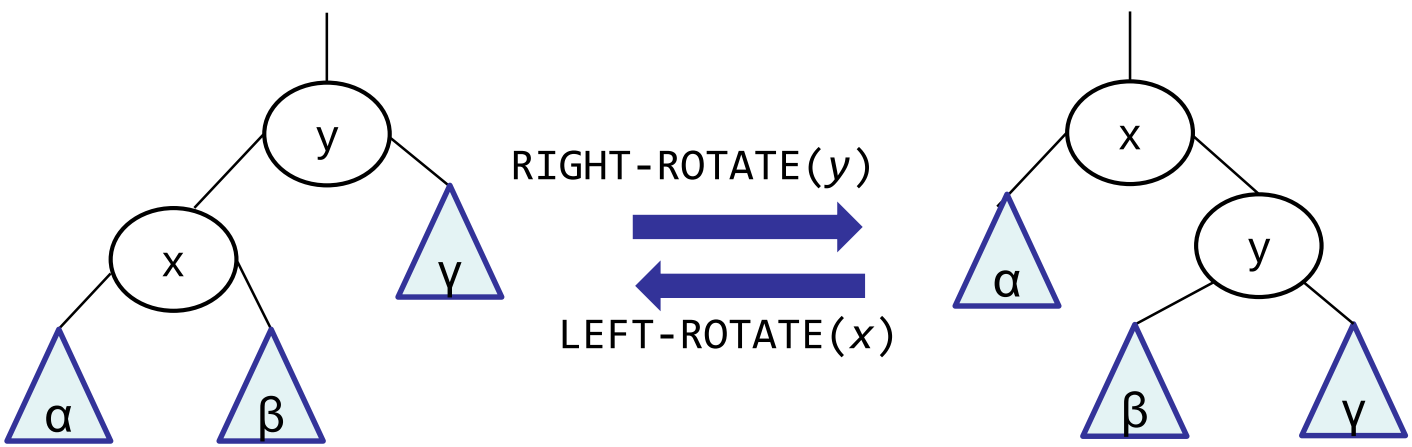
- `PUT`
  - Suche Einfügeposition: $\boldsymbol{O}(\log \boldsymbol{n})$ falls balanciert.
  - Wie färbt man neuen Knoten?
    - **Rot:** Verletzung von Eigenschaft 4?
    - **Schwarz:** Verletzung von Eigenschaft 5?
  - Wiederherstellung der Red-Black-Tree Eigenschaften?

- `DELETE`
  - Wie bei binären Suchbäumen ggfs. Vor- bzw. Nachfolger bestimmen.
  - Ggfs. Wiederherstellung der Red-Black-Tree Eigenschaften

- ***Wichtige Hilfsmethode*** beim Löschen und Einfügen: **Rotation**

Rotation

- Bei Einfügen als auch bei Löschen von Knoten benötigt.
  - "Repariert" Eigenschaften eines RB-Trees.
- ***Ändert nur Zeiger***, kopiert keine eigentlichen Daten.
  - "Minimal-invasiv"
- ***Erhält zentrale Eigenschaft eines binären Suchbaums***
  - "Schlüssel im linken Teilbaum sind < als Schlüssel im rechten Teilbaum".
- Links- und Rechtsrotationen operieren auf einem Knoten *x*.

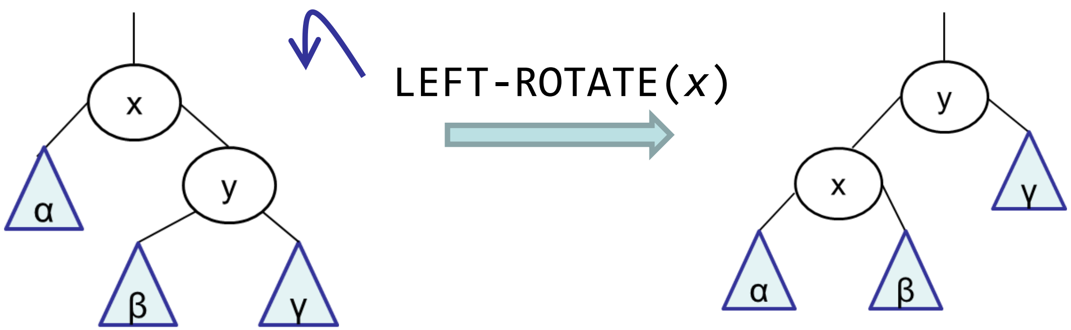


Rotation anhand einer Linksrotation

Siehe Java Library: TreeMap.java / rotateLeft

Linksrotation um Knoten x

```
LEFT-ROTATE(x)                  // Annahme: x.right ≠ T.NULL
    y = x.right                 // bestimme y
    x.right = y.left            // linkes Kind von y wird rechtes Kind von x
    if y.left ≠ T.NULL
        y.left.p = x
    y.p = x.p                   // y bekommt Eltern von x
    if x.p == T.NULL            // Fall: x war die Wurzel
        T.root = y
    elseif x == x.p.left        // Fall: x war linkes Kind
        x.p.left = y
    else                        // Fall: x war rechtes Kind
        x.p.right = y
    y.left = x                  // mache x zu linkem Kind von y
    x.p = y
```



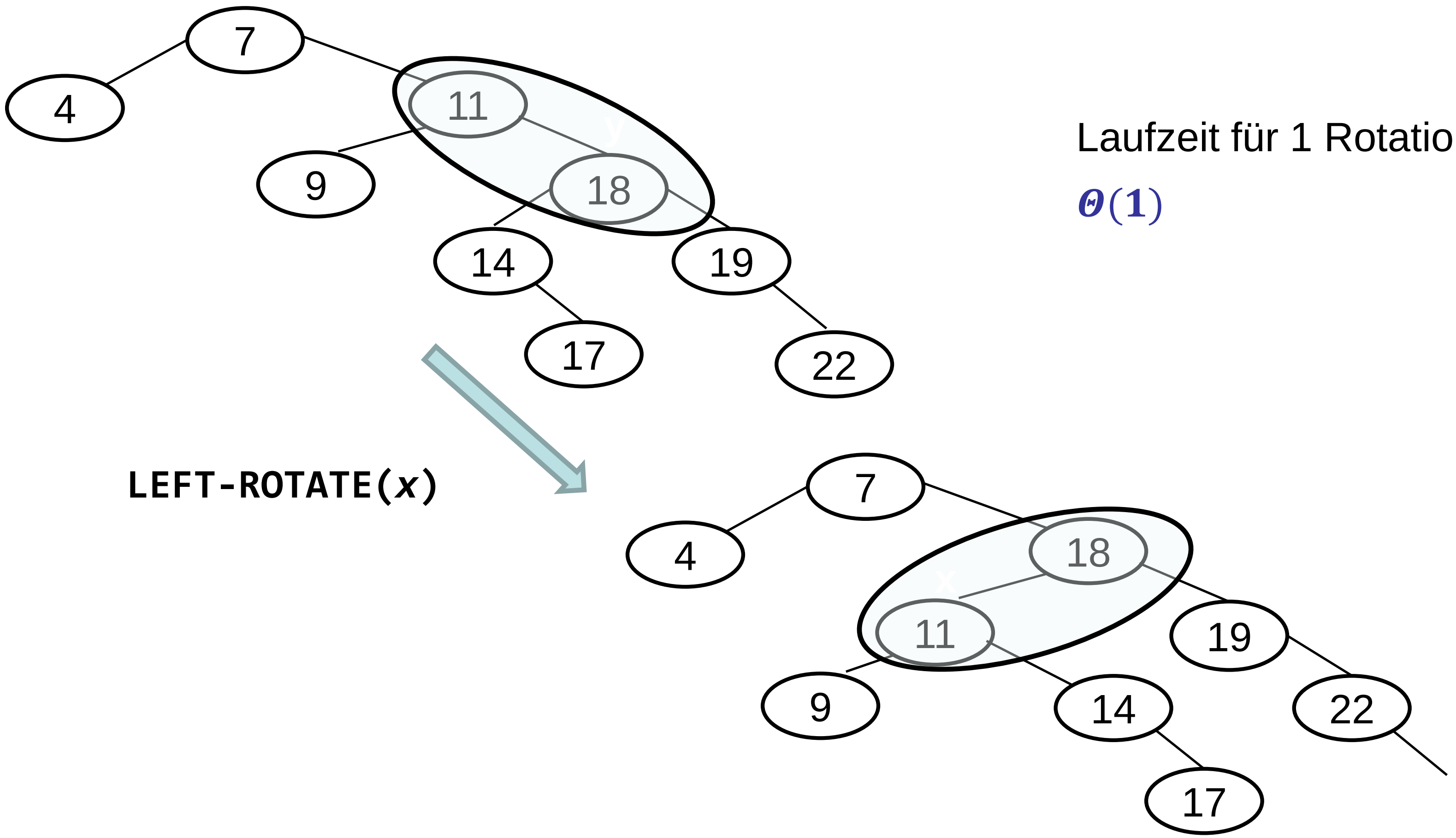
Hinweis:
Elternknoten der Wurzel: *T.NULL*

Laufzeit?

Konstant, d.h. $O(1)$

Rotation: Übung

- Verändert Rotation die In-Order Reihenfolge der Schlüssel?



Laufzeit für 1 Rotation:

$\Theta(1)$

Publikums-Joker

Welche Aussage **ist falsch?**

A. Die Laufzeit für die Durchführung einer Rotation ist unabhängig davon, wie viele Elemente der Baum speichert.

B. Rotationen verändern die Höhe des Baumes.

C. Die In-Order Traversierung nach einer Rotation ergibt die gleiche Reihenfolge wie vor der Rotation.

D. Links- und Rechtsrotationen sind gleich schnell.

Übersicht

- Binäre Suchbäume
  - Siehe Kapitel 7A
- Balancierte Binärbäume
  - Rot-Schwarz-Bäume
  - Rotationen
  - **Einfügen in Rot-Schwarz-Bäumen**
- B-Bäume
  - Siehe Kapitel 7C

Einfügen

- Einfügen ist in $O(\log n)$ möglich!

- **Ansatz**
  - Suche Einfügeposition.
  - Färbe einzufügenden Knoten ***z*** rot. Warum?
  - Aufruf von `FIXUP(z)` stellt Eigenschaften des RB-Baumes für Teilbaum bei *z* her.
  - Ggfs. ***iterative*** Reparatur von *z* bis zur Wurzel.

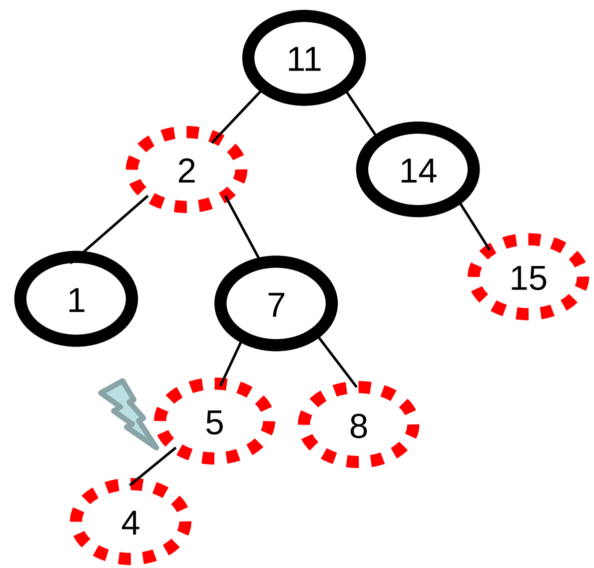
Füge Knoten z ein

```
PUT(z)
    y = T.NULL
    x = root
    while x ≠ T.NULL
        y = x
        if z.key < x.key
            x = x.left
        else
            x = x.right
    z.p = y
    if y == T.NULL // tree was empty
        root = z
    elseif z.key < y.key
        y.left = z
    else
        y.right = z
    z.left = T.NULL
    z.right = T.NULL
    z.color = RED
    FIXUP(z)
```

Nach Einfügen: Welche Eigenschaften verletzt?

- Nach Einfügen von **z**: Was **könnte** verletzt sein?
  - 1) Nein!
  - 2) **Ja!** Aber nur falls Baum bislang leer war und *z* die Wurzel wird.
  - 3) Nein! (Blätter hier nicht gezeichnet)
  - 4) **Ja!** Falls *z.p* rot ist, gibt es ein Problem!
  - 5) Nein!

Bsp.: Einfügen von *z*=4



- <u>Wiederholung: RB-Eigenschaften:</u>
  1) Jeder Knoten rot oder schwarz.
  2) Wurzel schwarz.
  3) Jedes Blatt schwarz.
  4) Knoten rot → alle Kinder schwarz.
  5) Für jeden Knoten: In ***allen*** Pfade zu ***nachfolgenden*** Blättern ***gleich viele schwarze Knoten***.

`FIXUP` (z): Wiederherstellung des RB-Baumes

- Wird aufgerufen nach Einfügen von Knoten *z:* `PUT(z)`

- Korrektur: Rotationen und Umfärbungen.

- Arbeitet ***iterativ***
  - Startet beim eingefügten Knoten *z* und versucht lokale Korrektur beim Teilbaum mit *z* als Wurzel.
  - Geht dann Ebene für Ebene nach oben, solange bis Abbruchkriterium
  - Jeweils ggfs. lokale Korrektur, bis zur Wurzel.

- Nimmt an, dass der RB-Baum intakt ist bis auf "Störung", die gerade durch Einfügen erzeugt wurde.

- Laufzeit: $O(\log n)$
  - Höhe: $\mathrm{h} = O(\log n)$
  - Je Iterationsschritt maximal 2 Rotationen (jede davon: $O(1)$)

FIXUP: Überblick

```
FIXUP(z)
  while z.p.color == RED
     if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
           z.p.color = BLACK
           y.color = BLACK
           z.p.p.color = RED
           z = z.p.p
        else
           if z == z.p.right
                z = z.p
                LEFT-ROTATE(z)
           z.p.color = BLACK
           z.p.p.color = RED
           RIGHT-ROTATE(z.p.p)
     else
         ". . ."

  root.color = BLACK
```

Repariere Schaden bei Knoten z ggfs. rekursiv bis oben zur Wurzel.

// Elter von *z* ist linkes Kind

`// y: "Onkel" von z`

Fall 1: Onkel ist rot

Fall 2: Onkel y ist schwarz, z ist rechtes Kind
→ macht *z* zu linkem Kind (geht über in Fall 3)

Fall 3: Onkel *y* ist schwarz, *z* ist linkes Kind

Elter von z ist rechtes Kind
Wie oben, einfach "right" durch "left" tauschen

Siehe Java Library: TreeMap.java / fixAfterInsertion

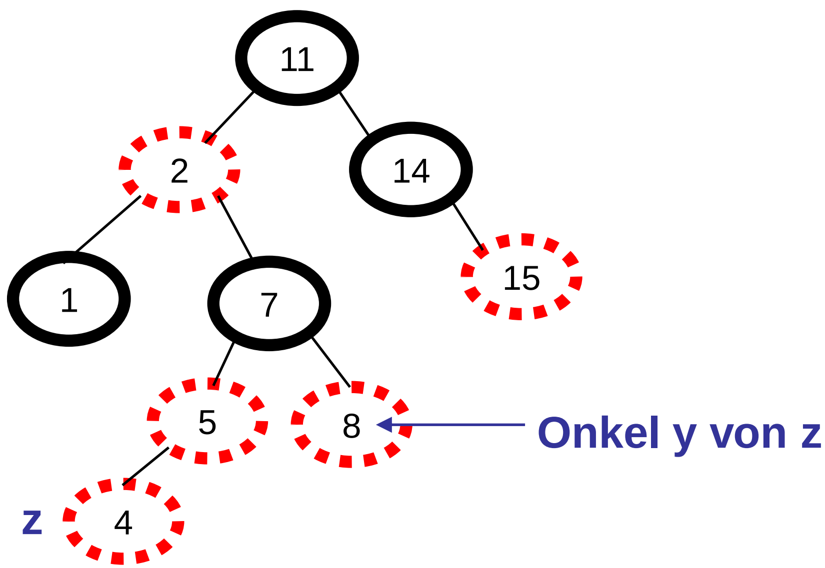
Annahme für die folgenden Folien!!!

- Man betrachtet immer 3 Generationen
  - Großeltern, Eltern, Kind == aktueller Knoten
- **Der Elternknoten von z ist linkes Kind**
  - ***`z.p == z.p.p.left`***
- Ansonsten müssten man im Folgenden grundsätzlich links und rechts vertauschen.

Was ist der Onkel von z?

Bruder/Schwester vom Elternknoten von z!

Code, kein Klausurstoff

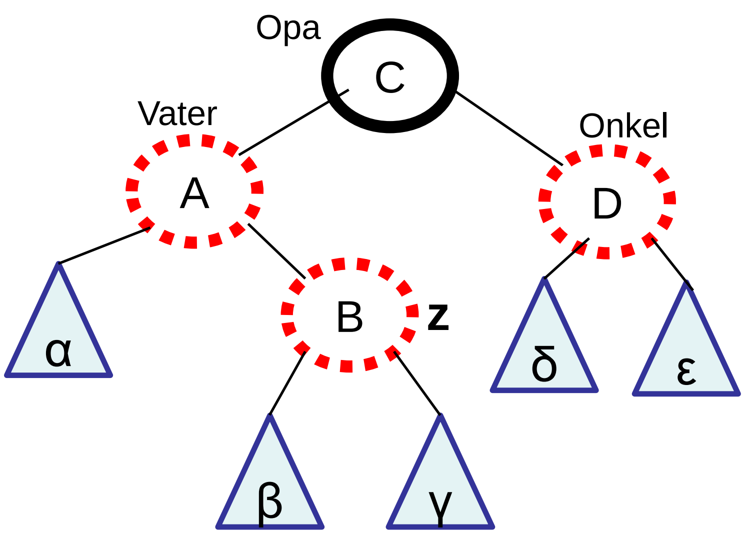


Hinweis: *z* sei der eingefügte Knoten

Fall 1: Onkel y ist rot.

- Opa von *z* (hier *C*) muss schwarz sein. Warum?

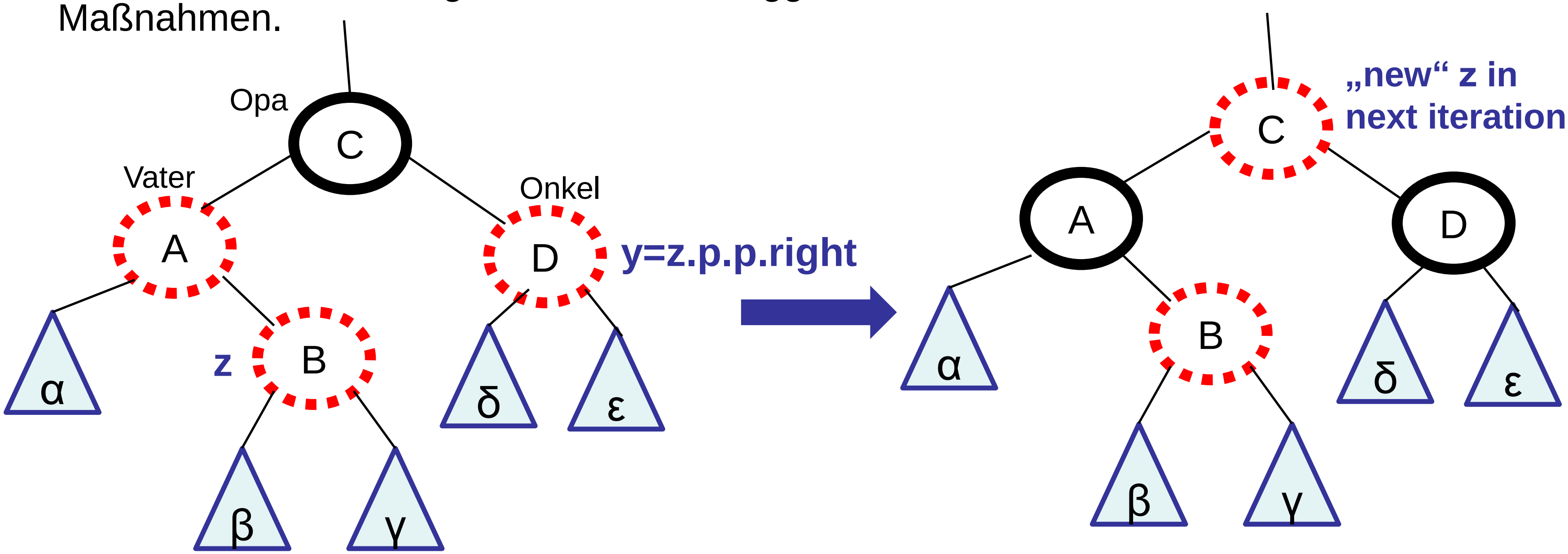
- Der Vater von *z* (hier *A*) ist ebenfalls rot.
  - Ansonsten wäre (durch Einfügen von z) Bedingung 4 nicht verletzt.
  - Es gilt: $\mathrm{bh}(A) = \mathrm{bh}(D)$, d.h. die Schwarztiefen beider Knoten sind gleich.



Fall 1: Onkel y ist rot

- **Aktion**
  - Färbe Vater *A* und Onkel *D* schwarz.
  - Färbe Großvater *C* rot, um Eigenschaft 5 wiederherzustellen.
- **Achtung!** *C* ist nur ein Teilbaum
  - Weiter oben werden ggfs. RB-Eigenschaften gestört → Iteriere weiter in Richtung Wurzel, setze dazu *z* auf *C*
  - Welche Eigenschaft könnte verletzt sein?
  - Deshalb muss z auf C gesetzt werden → ggfs. weitere Maßnahmen.

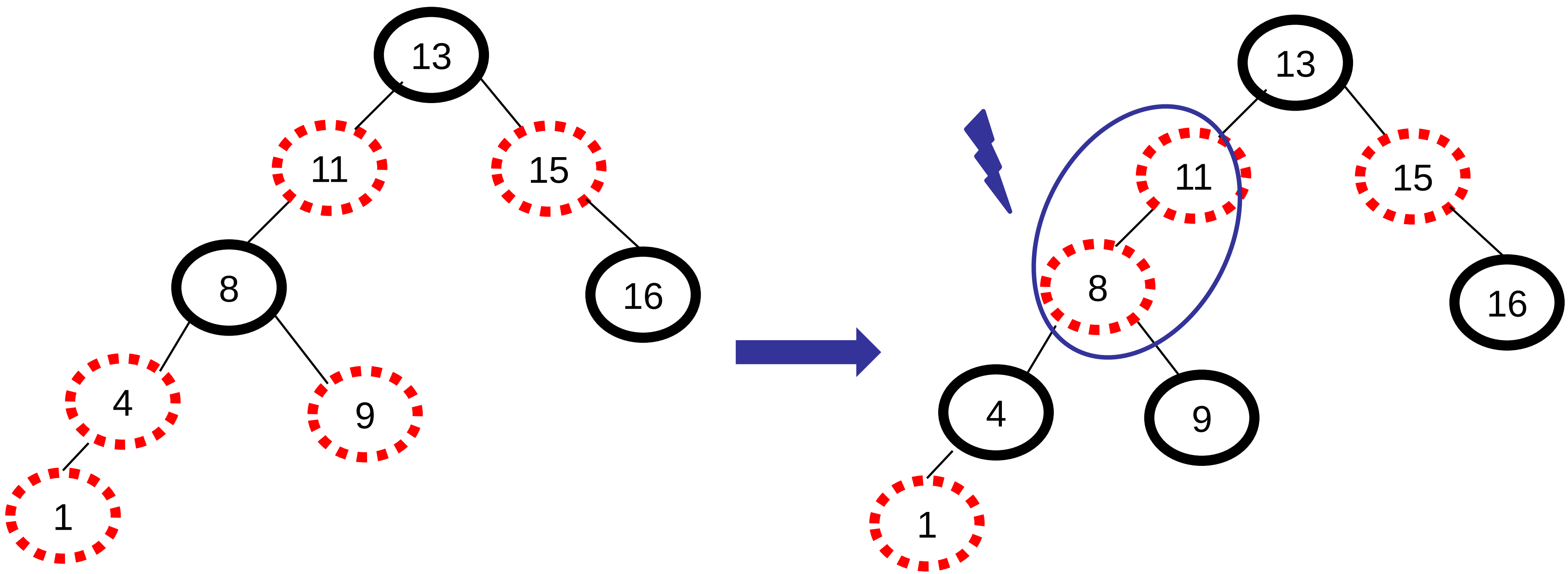
```
AKTION
z.p.color = BLACK
y.color = BLACK
z.p.p.color = RED
z = z.p.p
```



Fall 1: Übung

- Knoten 1 wird eingefügt, *z* zeigt auf diesen Knoten.
- Hier ist *z* übrigens linkes Kind!
- Wie sieht das Ergebnis aus?

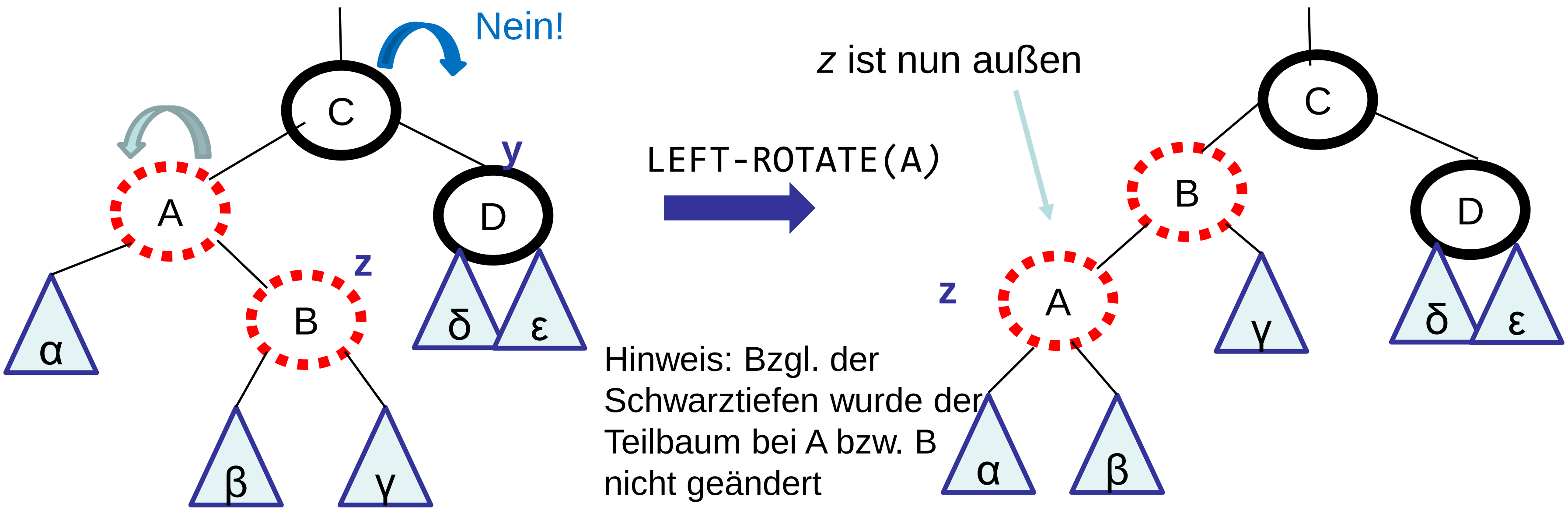
Welche Eigenschaft ist nun verletzt?



Fall 2: Onkel schwarz oder nicht vorhanden, z innen

- ***Ziel:***
  - Bringe Knoten *z* (hier *B*) durch Rotation ***nach "außen"***
  - Dadurch geht Fall 2 in Fall 3 über.
- Keine Farbänderung eines Knoten!
- ***Ansatz:***
  - "Schalte" *z* ***zunächst*** auf Vater von *z* weiter.
  - Linksrotation um ursprünglichen Vater *(hier A)* von *z*.
- Hinweis: Kein Onkel entspricht schwarzem Knoten T.NULL

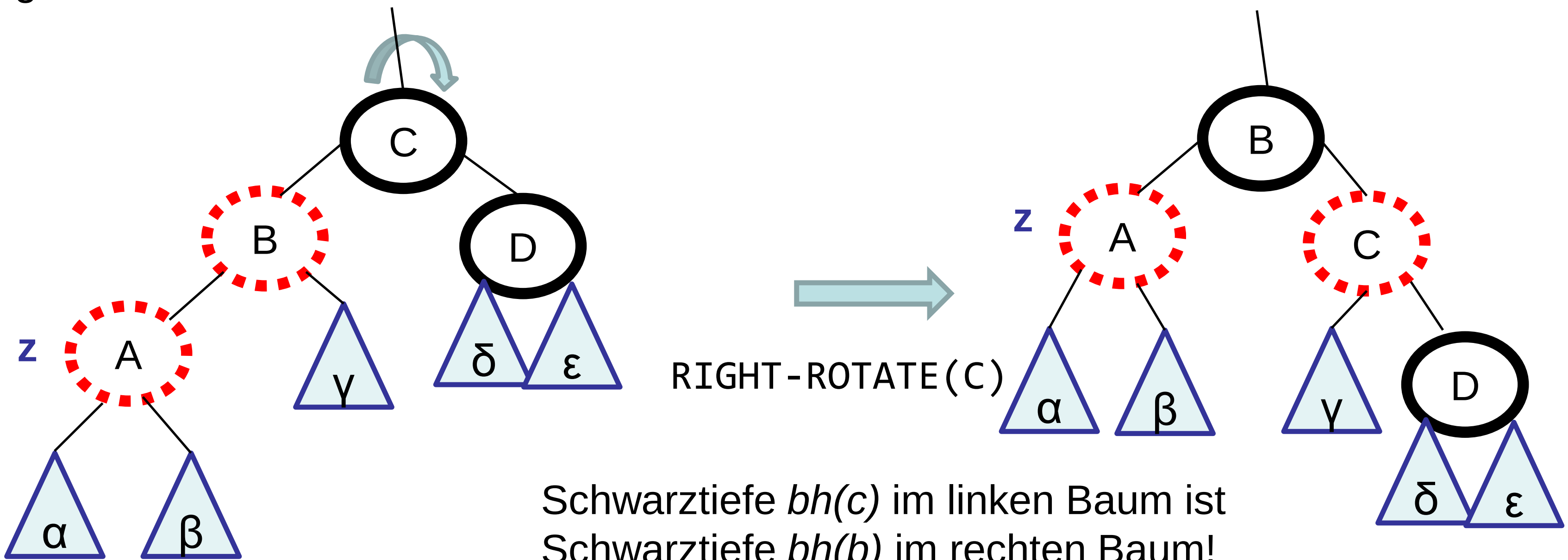
```
AKTION
z = z.p
LEFT-ROTATE(z)
```



Fall 3: Onkel y schwarz bzw. nicht vorhanden, z außen

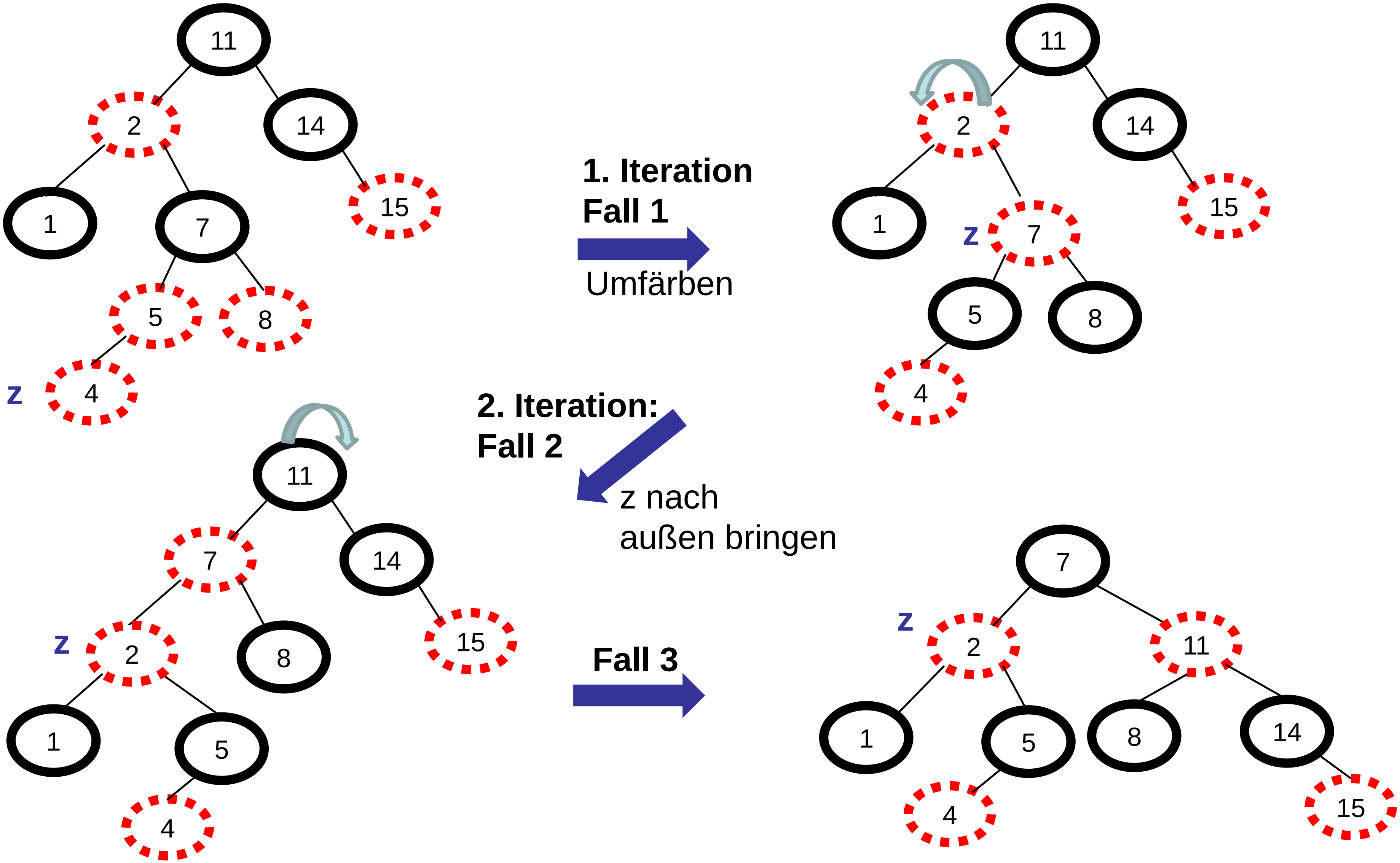
- ***Annahme***
  - z sei nun sicher linkes Kind
- ***Farbenänderung***
  - Vater *B* von *z* wird schwarz
  - Opa *C* von *z* wird rot
- **Ansatz**
  - Rechtsrotation um Opa *C*
  - z bleibt bei gleichem Knoten, der aber 1 Ebene nach oben wandert.
- Ergebnis: Keine 2 rote Knoten mehr hintereinander.

```
AKTION
z.p.color = BLACK
z.p.p.color = RED
RIGHT-ROTATE(z.p.p)
```



Schwarztiefe *bh(c)* im linken Baum ist Schwarztiefe *bh(b)* im rechten Baum!

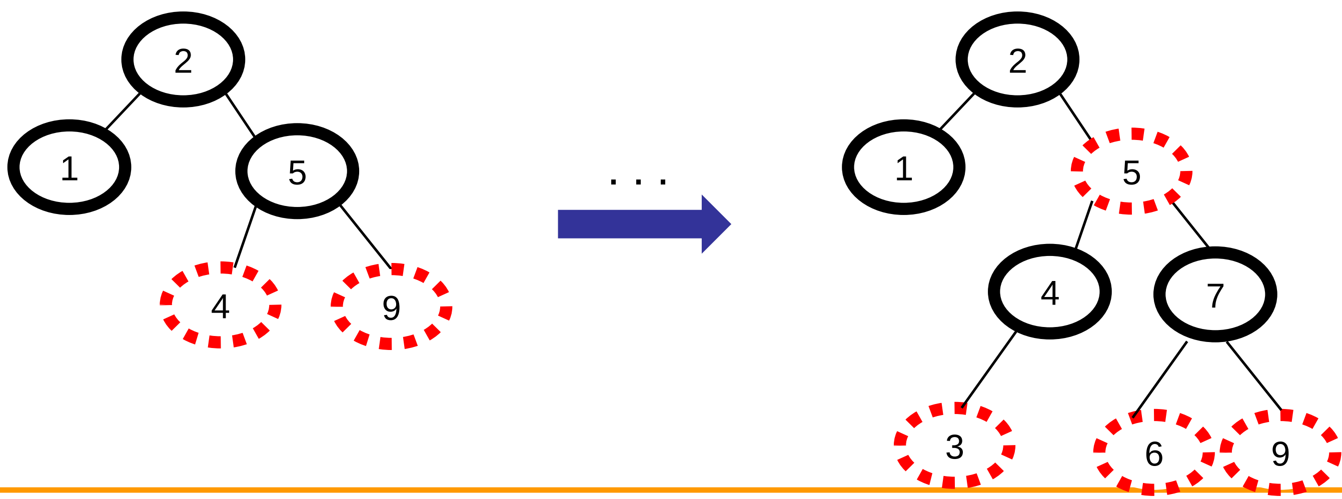
Beispiel: Einfügen von 4



Einfügen in RB-Trees

- ***Beispiel***
  - Füge Schlüssel 3, dann Schüssel 6, dann Schlüssel 7 ein
  - Lösung mit Animation nachvollziehen: https://www.cs.usfca.edu/~galles/visualization/RedBlack.html
  - Beinhaltet alle 3 Fälle!

- ***Terminierung***
  - Egal ob Fall 1, 2 oder 3: Der Knoten *z* ist danach rot.
  - Falls Elternknoten *z.p* rot ist: Weitermachen!
  - Sonst: Terminierung, da entweder
    - an Wurzel angelangt oder
    - Baum erfüllt Rot-Schwarz-Eigenschaften



FIXUP: Überblick

```
FIXUP(z)
  while z.p.color == RED
      if z.p == z.p.p.left
          y = z.p.p.right
          if y.color == RED
              z.p.color = BLACK
              y.color = BLACK
              z.p.p.color = RED
              z = z.p.p
          else
              if z == z.p.right
                      z = z.p
                      LEFT-ROTATE(z)
              z.p.color = BLACK
              z.p.p.color = RED
              RIGHT-ROTATE(z.p)
      else
           ". . ."

  root.color = BLACK
```

Repariere Schaden bei Knoten z ggfs. rekursiv bis oben zur Wurzel.

Terminierung

// Elter von *z* ist linkes Kind

// *y*: "Onkel" von z

Fall 1: Onkel ist rot

Fall 2: Onkel y ist schwarz, z ist rechtes Kind
→ macht *z* zu linkem Kind (geht über in Fall 3)

Fall 3: Onkel *y* ist schwarz, *z* ist linkes Kind

Elter von z ist rechtes Kind
Wie oben, einfach "right" durch "left" tauschen

Siehe Java Library: TreeMap.java / fixAfterInsertion

Diskussion

- Laufzeit FIXUP: $O(\log n)$
  - Jede Iteration der while-Schleife: $O(1)$
  - Jede Iteration ist entweder die letzte oder sie schiebt z um 2 Ebenen nach oben (Fall 1).
  - $O(\log n)$ Ebenen $\rightarrow O(\log n)$ Laufzeit

- Laufzeit für PUT: $O(\log n)$

- **Alle Basisoperationen benötigen: $\boldsymbol{O(log\ n)}$ im Worst Case!**
  - Kein Worst Case von $\boldsymbol{O(n)}$ wie bei Hashtabellen oder Arrays möglich!
  - Aber Laufzeit ist auch nicht durchschnittlich konstant wie bei Hashtabellen.

- Java verwendet Red-Black Trees
  - `TreeMap` und `TreeSet`

- Animation
  - https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

Publikums-Joker

Welche der folgenden Aussagen bzgl. dem Einfügen eines Elements *z* in einen Red-Black Tree ist **falsch**?

A. Das Finden der Einfügeposition funktioniert ähnlich wie bei einem normalen binären Suchbaum.

B. Der einzufügende Knoten wird zunächst rot eingefärbt.

C. In jeder Iteration der Methode RB-INSERT-FIXUP wandert der Zeiger z, der zu Beginn der Iterationen auf das einzufügende Element zeigt, um genau 1 Ebene nach oben Richtung Wurzel.

D. Der Algorithmus kann terminieren, obwohl der Zeiger z, der zu Beginn der Iterationen auf das einzufügende Element zeigt, noch gar nicht unmittelbar unter der Wurzel angelangt ist (also noch gar nicht Kind der Wurzel) ist.

Löschen

- Ähnlich wie bei binären Suchbaum.

- Nach Löschen können RB-Tree Eigenschaften verletzt sein.

- Pseudocode in etwa doppelt so lange wie beim Einfügen.
  - Wird in Vorlesung nicht besprochen.

- Weiterführende Informationen

  https://www.geeksforgeeks.org/red-black-tree-set-3-delete-2/

Zusammenfassung

- Baum als Datenstruktur

- Binäre Suchbäume
  - Suchen, Einfügen und Entfernen von Schlüsseln
  - Traversieren von Bäumen
  - Laufzeitanalyse

- Balancierte Binärbäume
  - Rote-Schwarz-Bäume
  - Suche, Einfügen, Löschen in linearer Zeit

- B-Bäume
  - Definition und Anwendung
  - Suchen, Einfügen und Entfernen von Schlüsseln

Quellenverzeichnis

[1] Cormen, Leiserson, Rivest and Stein. *Introduction to Algorithms*, Third Edition, The MIT Press, 2009.

[2] Ottmann, Widmayer. *Algorithmen und Datenstrukturen*, Kapitel 5.1, 5. Auflage, Spektrum Akademischer Verlag, 2012.

[3] Quelle: https://cs124.quora.com/xkcd-comics