Datenstrukturen & Algorithmen

Peppo Brambilla
Universität Bern
Frühling 2018

Übersicht

Rot-schwarz Bäume

- Eigenschaften
- Rotationen
- Einfügen
- (Löschen)

Einführung

- Binäre Suchbäume
  - Höhe h
  - $O(h)$ für Operationen auf dynamische Mengen
  - Baum kann zu Liste degenerieren
  - h ist im Worst Case $O(n)$
- Balancierte Suchbäume
  - Garantieren, dass Baum nicht zu Liste degenerieren kann
  - Garantieren $h = O(\lg n)$
  - Erzwungen mittels zusätzlichen Bedingungen an Datenstruktur

Rot-schwarz Bäume

- Eine Variante von balancierten Suchbäumen
- Garantieren, dass Höhe im Worst Case $O(\lg n)$ statt wie vorher $O(n)$
  - Alle Operationen sind $O(\lg n)$

Rot-schwarz Bäume

- Zeiger gleich wie binäre Suchbäume

$$(\mathrm{left}, \mathrm{right}, \mathrm{p})$$

- Konvention
  - Falls ein Kind nicht existiert, wird Zeiger auf Wächterelement $T.nil$ gespeichert
  - Alle Knoten ausser $T.nil$ werden als innere Knoten betrachtet
  - Vater von Wurzel zeigt auch auf $T.nil$

Rot-schwarz Bäume

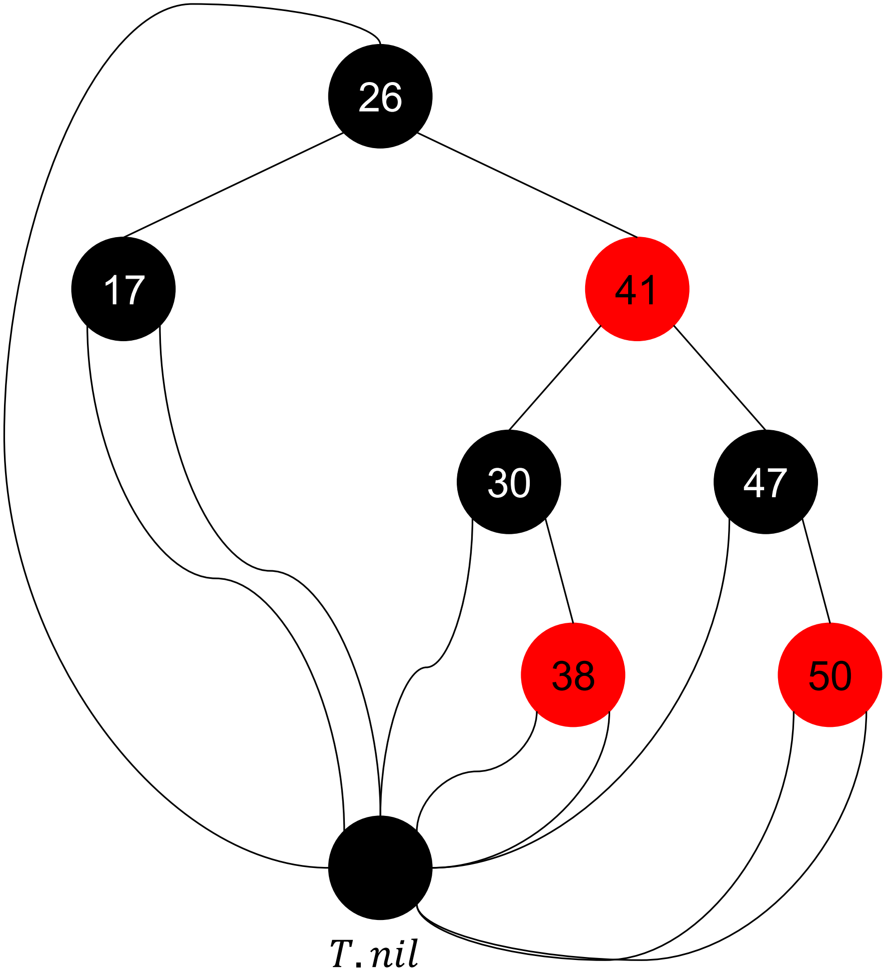
Hauptidee

- Jeder Knoten hat ein Bit Zusatzinformation
  - Farbe des Knotens: rot oder schwarz
- Bedingungen an Farbreihenfolge entlang Pfaden von Wurzel zu Blatt
  - Garantieren, dass kein Pfad mehr als doppelt so lang als alle anderen ist
  - Baum ist balanciert
- Einfügen und Löschen müssen Bedingungen erhalten

Eigenschaften

1. Jeder Knoten ist rot oder schwarz
2. Die Wurzel ist schwarz.
3. Jedes Blatt ($T.nil$) ist schwarz.
4. Wenn ein Knoten rot ist,
   dann sind seine beiden Kinder schwarz.
5. Für jeden Knoten gilt:
   Alle Pfade vom Knoten zu seinen Blättern
   haben dieselbe Anzahl schwarzer Knoten.

Beispiel

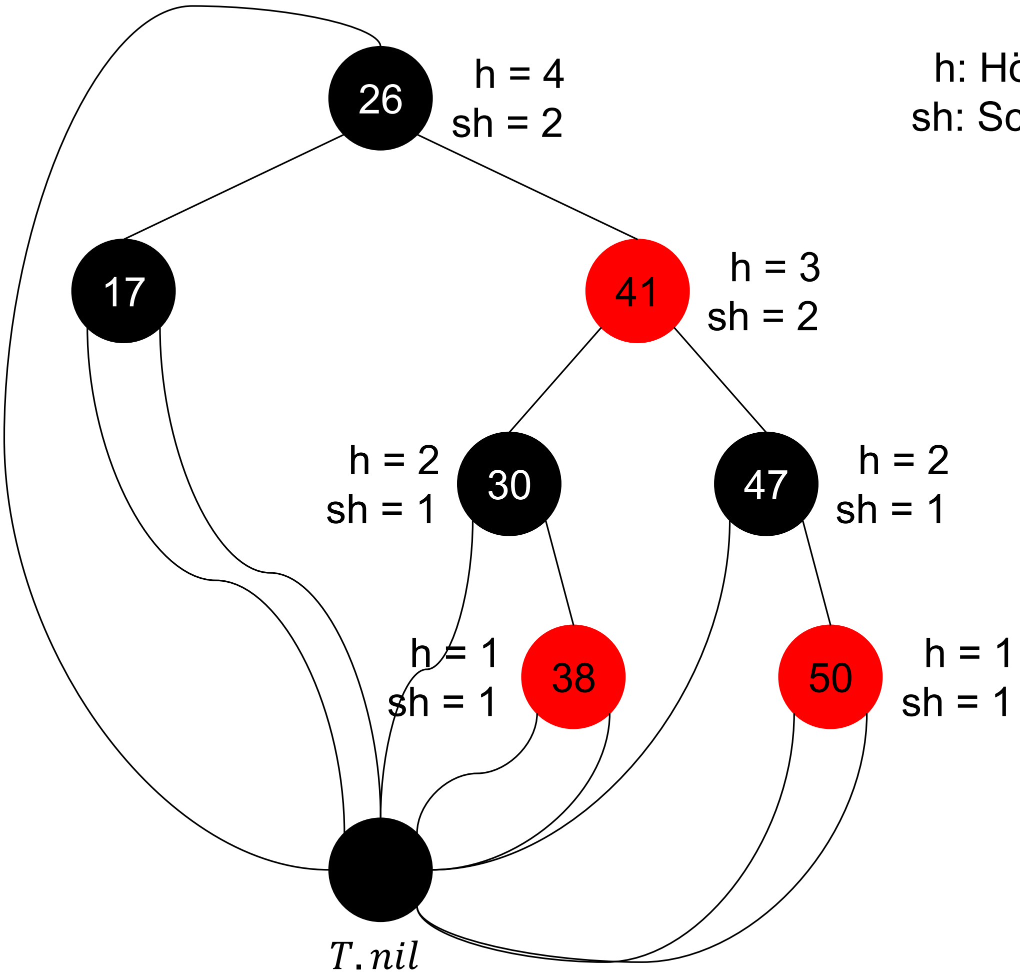


Alle internen Knoten haben **zwei** Kinder wegen Verbindung zum nil Element.

Schwarz-Höhe

- Höhe eines Knotens
  - Länge des längsten Pfades vom Knoten zu einem Blatt
- Schwarz-Höhe $\mathrm{sh}(x)$ eine Knotens x
  - Anzahl Schwarzer Knoten auf irgendeinem Pfad von x zum Blatt $T.nil$
  - x selbst nicht gezählt
  - Blatt $T.nil$ mitgezählt
  - Wegen Eigenschaft 5 wohldefiniert

Beispiel



h: Höhe
sh: Schwarz-Höhe

Höhe von rot-schwarz Bäumen

- Lemma: Eigenschaften von rot-schwarz Bäumen garantieren, dass Höhe des Baumes $h \leq 2\lg(n+1) = O(\lg n)$
- Garantiert effiziente Operationen!
- Beweis mittels 2 Behauptungen

Höhe von rot-schwarz Bäumen

1. Behauptung: Knoten mit Höhe h hat mindestens Schwarz-Höhe $h/2$

- Beweis: Eigenschaft 4
  - Höchstens $h/2$ rote Knoten auf einem Pfad von Knoten zu jedem Blatt
  - Mindestens $h/2$ Schwarze Knoten

Höhe von rot-schwarz Bäumen

2. Behauptung: Unterbaum von jedem Knoten x hat $\geq 2^{\mathrm{sh}(x)} - 1$ interne Knoten

Induktion über Höhe h

Verankerung: $h = 0$

$\Rightarrow x$ ist Blatt $\Rightarrow \mathrm{sh}(x) = 0$

Unterbaum hat keine internen Knoten

$\Rightarrow 0 \geq 2^{\mathrm{sh}(x)} - 1 = 2^0 - 1 = 0$

Höhe von rot-schwarz Bäumen

Induktionsschritt: $h > 0$

$\Rightarrow x$ hat 2 Kinder

Kinder haben

- Höhe $< h$
- Schwarz-Höhe $\begin{cases} \mathrm{sh}(x) - 1 & \text{falls Kind schwarz} \\ \mathrm{sh}(x) & \text{falls Kind rot} \end{cases}$

Schwarz-Höhe eines Kindes also mindestens $\mathrm{sh}(x) - 1$

Induktionsannahme:

Jedes Kind hat $\geq 2^{\mathrm{sh}(x)-1} - 1$ interne Knoten.

$\Rightarrow$ Unterbaum von x hat

Vater

$$\geq \underbrace{2\left(2^{\mathrm{sh}(x)-1} - 1\right) + 1}_{= 2^{\mathrm{sh}(x)}-1} \text{ interne Knoten}$$

Höhe von rot-schwarz Bäumen

- Zusammenfassend
  1. Knoten mit Höhe h hat Schwarz-Höhe $\geq h/2$
  2. Für jeden Knoten x gilt: Unterbaum von x hat $\geq 2^{\mathrm{sh}(x)} - 1$ interne Knoten
- Sei b Schwarz-Höhe der Wurzel, n Anzahl Knoten im ganzen Baum

$$\underbrace{n \geq 2^b - 1}_{2.} \geq \underbrace{2^{h/2} - 1}_{1.\ b \geq h/2} \Rightarrow n + 1 \geq 2^{h/2}$$

$$\begin{aligned} n + 1 &\geq 2^{h/2} && \mid \lg \\ \lg(n+1) &\geq h/2 && \mid \cdot 2 \\ 2\lg(n+1) &\geq h && \Rightarrow h = O(\lg n) \end{aligned}$$

Operationen

- Nicht-modifizierende Operationen (Suchen, Vorgänger, Nachfolger, Minimum, Maximum) gleich wie binäre Suchbäume
  - Aufwand $O(\lg n)$
- Einfügen und Löschen sind komplizierter
- Einfügen
  - Roter Knoten: kann Eigenschaft 4 verletzen
  - Schwarzer Knoten: kann Eigenschaft 5 verletzen
- Löschen
  - Rote Knoten: kein Problem
  - Schwarze Knoten: kann Eigenschaften 2, 4, oder 5 verletzen

Übersicht

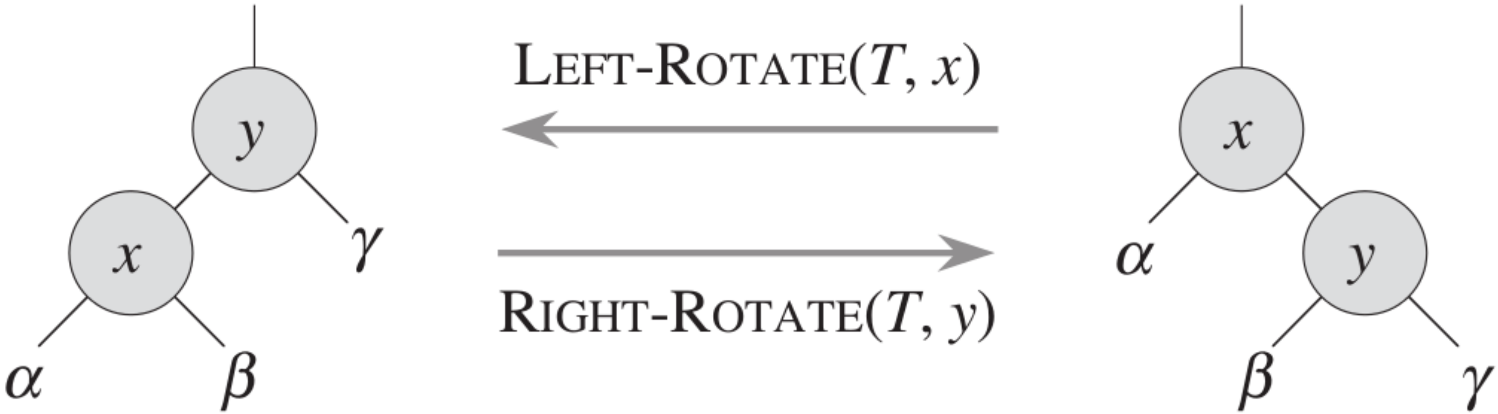
Rot-schwarz Bäume

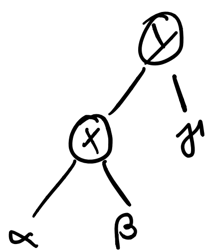
- Eigenschaften
- Rotationen
- Einfügen
- (Löschen)

Rotationen

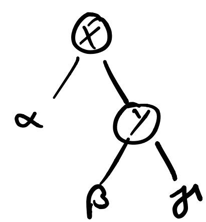
- Einfügen und Löschen zerstören rot-schwarz Eigenschaften
- Benötigen Rotationen, um rot-schwarz Eigenschaften wiederherzustellen
  - Ändern Baumstruktur
  - Bewahren binäre Suchbaum-Eigenschaft
- Es gibt Links- und Rechtsrotationen
  - Invers zueinander
  - Implementiert durch Umhängen von Zeigern

Rotationen





$x \le y$ (1)
$\alpha \le y$ (2)
$\beta \le y$ (3)
$\alpha \le x$ (4)
$x \le \beta$ (5)
$y \le \gamma$ (6)

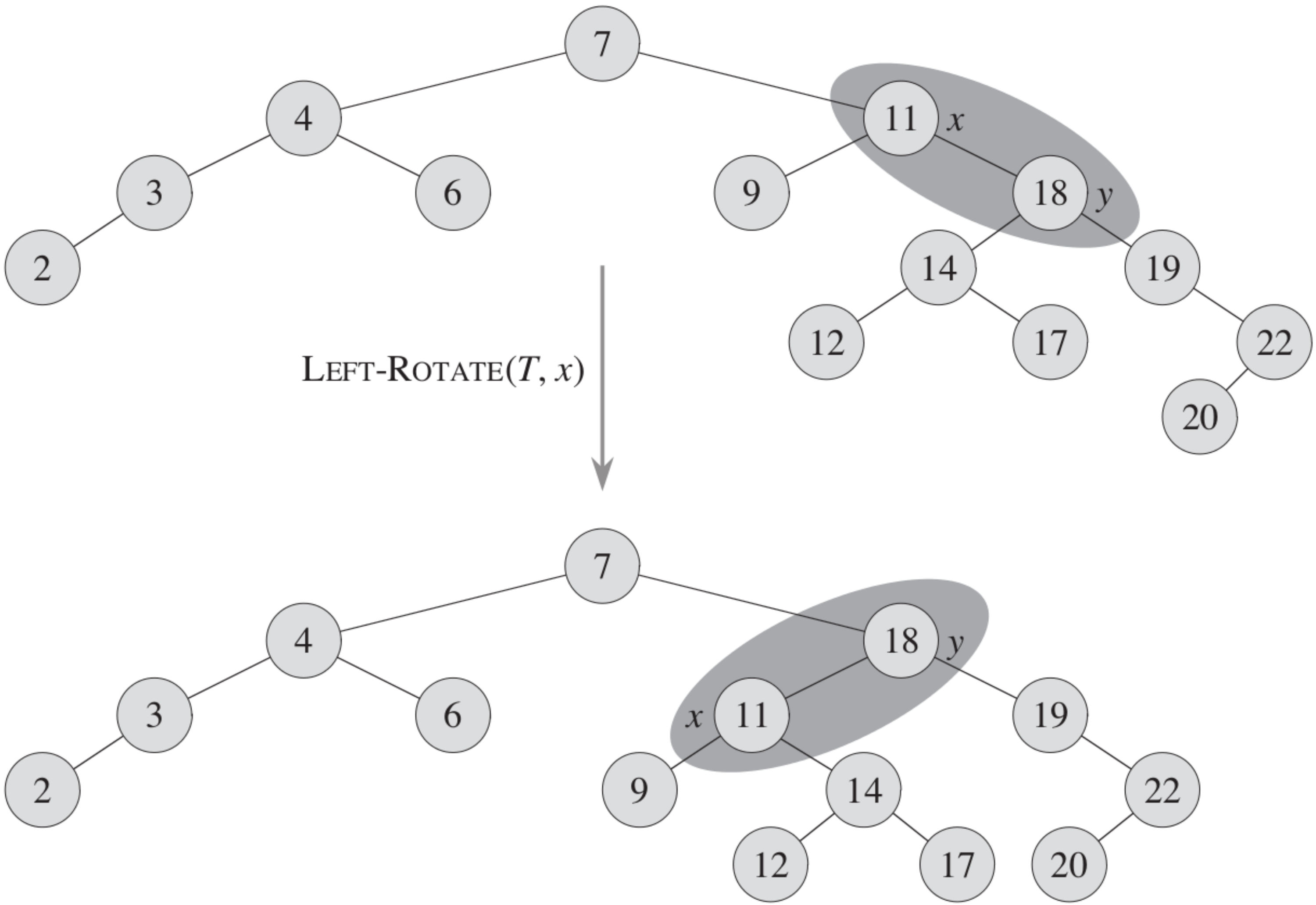


$\alpha \le x = (4)$
$x \le y = (1)$
$x \le \beta = (5)$
$x \le \gamma = (1) + (6)$
$\beta \le y = (3)$
$y \le \gamma = (6)$

Rotationen

```
LEFT-ROTATE(T, x) [x.right ≠ T.nil]
1   y = x.right                   // bestimme y
2   x.right = y.left              // 2–4: mache linken Teilbaum von y
3   if y.left ≠ T.nil             //      zu rechtem Teilbaum von x
4       y.left.p = x
5   y.p = x.p                     // 5–10: verbinde Vater von x mit y
6   if x.p == T.nil
7       T.root = y
8   elseif x == x.p.left
9       x.p.left = y
10  else x.p.right = y
11  y.left = x                    // 11–12: mache x zu linkem Kind von y
12  x.p = y
```

Beispiel


7
4
11
x
3
6
9
18
y
2
14
19
12
17
22
20
LEFT-ROTATE(T, x)
7
4
18
y
3
6
x
11
19
2
9
14
22
12
17
20

Rotationen

- Aufwand konstant, da eine konstante Anzahl Zeiger verändert werden
- Rotationen werden auch in anderen balancierten Suchbäumen verwendet
  - AVL Bäume, Splay Bäume
    http://en.wikipedia.org/wiki/AVL_tree
    http://en.wikipedia.org/wiki/Splay_tree

Übersicht

Rot-schwarz Bäume

- Eigenschaften
- Rotationen
- Einfügen
- (Löschen)

Einfügen

- Zwei Schritte

1. Einfügen (fast) wie in binären Suchbaum
2. Wiederherstellen der rot-schwarz Eigenschaften

Einfügen

RB-INSERT(T, z)

$y = T.nil$ $x = T.root$	Starte bei Wurzel x, traversiere Baum nach unten. y ist immer Vater von x.

while $x \neq T.nil$
    $y = x$
    if $z.key < x.key$
        $x = x.left$
    else $x = x.right$
	x traversiere Baum nach unten,
	wie beim Suchen nach $z.key$.
	Am Schluss der Schleife immer $y = x.p$

$z.p = y$	y wird Vater des eingefügten Elements.
if $y == T.nil$ $T.root = z$	Baum war leer, z wird Wurzel.
elseif $z.key < y.key$ $y.left = z$ **else** $y.right = z$	z wird je nach Schlüssel linkes oder rechtes Kind von y.
$z.left = T.nil$ $z.right = T.nil$ $z.color =$ RED	verbinde z mit $T.nil$ und färbe z rot.
RB-INSERT-FIXUP(T, z)	stelle rot-schwarz Eigenschaften wieder her.

Eigenschaften

1. Ok
2. Verletzt falls z die Wurzel ist, sonst Ok
3. Ok
4. Verletzt falls $z.p$ rot ist: sowohl z wie auch $z.p$ sind rot
5. Ok

Wiederherstellen der Eigenschaften

RB-INSERT-FIXUP(T, z)

```
while z.p.color == RED
    if z.p == z.p.p.left                              // Vater von z is linkes Kind
        y = z.p.p.right                               // y ist Onkel von z
        if y.color == RED                             // Fall 1: Onkel ist rot
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else if z == z.p.right                        // Fall 2: z ist rechtes Kind
                z = z.p
                LEFT-ROTATE(T, z)
            z.p.color = BLACK                         // Fall 3: z ist linkes Kind
            z.p.p.color = RED
            RIGHT-ROTATE(T, z.p.p)
    else (same as then clause with                    // Vater von z is rechtes Kind
            "right" and "left" exchanged)
T.root.color = BLACK
```

Schleifeninvariante

- Zu Beginn jeder Iteration der while-Schleife gilt
  a. Knoten z ist rot
  b. Falls $z.p$ Wurzel, dann ist $z.p$ schwarz
  c. Es gibt höchstens eine rot-schwarz Verletzung
     - Eigenschaft 2: z is die Wurzel und rot
     - Eigenschaft 4: sowohl z wie auch $z.p$ sind rot

Schleifeninvariante

Initialisierung

a. Bei Initialisierung ist z roter Knoten, der eingefügt wurde

b. Falls $z.p$ Wurzel, dann schwarz vor Einfügen, und Farbe wurde nicht geändert

c. Zwei Fälle

   – Falls Eigenschaft 2 verletzt: rote Wurzel ist Knoten der eingefügt wurde, keine andere Verletzung

   – Falls Eigenschaft 4 verletzt: z und $z.p$ sind rot, keine andere Verletzung

Schleifeninvariante

Terminierung

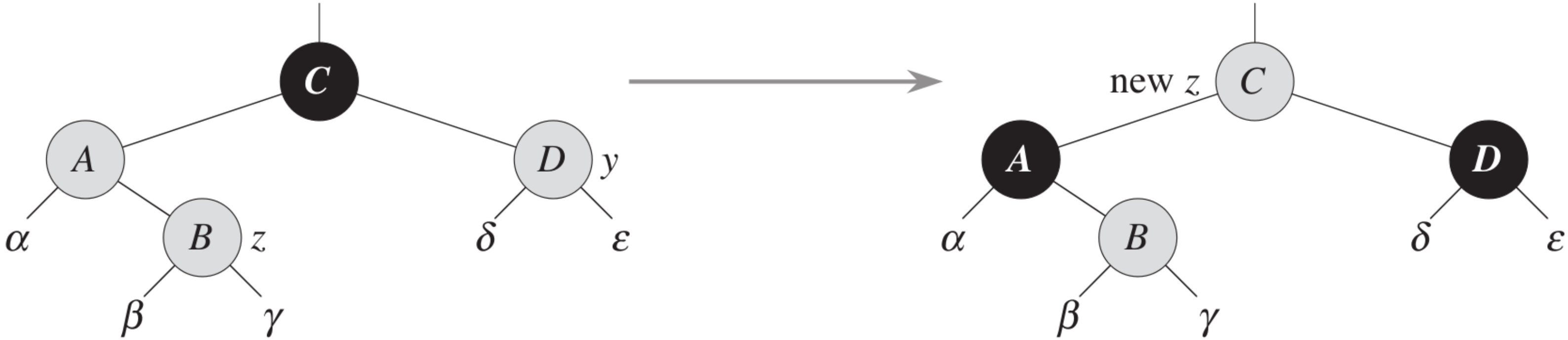
- Schleife endet weil $z.p$ schwarz ist, also ist Eigenschaft 4 ok
- Letzte Zeile garantiert Eigenschaft 2
- Deshalb: rot-schwarz Eigenschaften bei Terminierung erfüllt

Schleifeninvariante

Fortsetzung

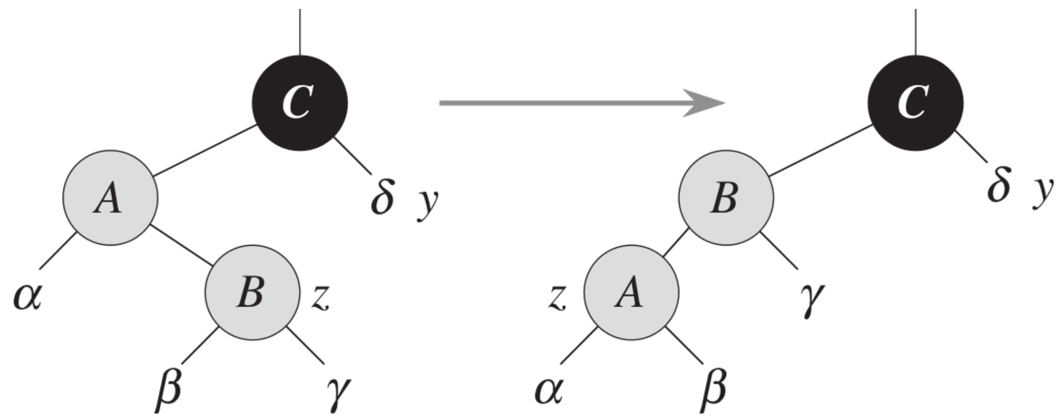
- Fallunterscheidung
  - 6 Fälle, 3 symmetrisch zu den anderen 3
  - Je 3 Fälle für $z.p$ entweder linkes oder rechtes Kind
  - Beschreibung hier nimmt an $z.p$ ist linkes Kind
  - y bezeichnet den Onkel von z, also den Bruder von $z.p$
- Fälle schliessen sich gegenseitig nicht aus

Fall 1: Onkel y ist rot



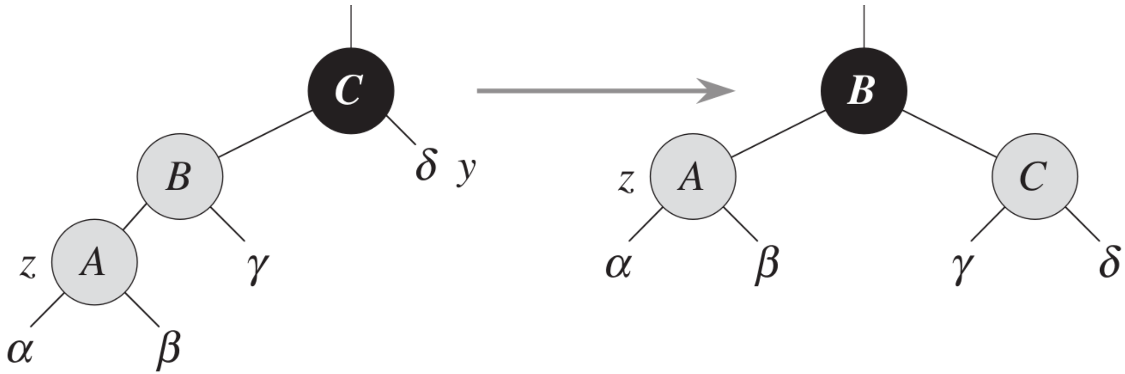
- $z.p.p$ muss schwarz sein, weil sowohl z wie $z.p$ rot sind, es aber nur eine Verletzung von Eigenschaft 4 gibt
- Färbe $z.p$ und y schwarz → Eigenschaft 4 ok, aber Eigenschaft 5 kann verletzt sein
- Färbe $z.p.p$ rot → Eigenschaft 5 ok, aber 4 ev. nicht
- Nächste Iteration hat $z.p.p$ als neues z

Fall 2: Onkel y schwarz, z rechtes Kind



- Linksrotation um $z.p$ → z ist nun linkes Kind, und z und $z.p$ sind rot
- Sind nun in Fall 3

Fall 3: Onkel y schwarz, z linkes Kind



- Färbe $z.p$ schwarz und $z.p.p$ rot
- Rechtsrotation um $z.p.p$
- Eigenschaft 4 ok: nicht mehr zwei aufeinanderfolgende rote Knoten
- $z.p$ ist jetzt schwarz → keine weiteren Iterationen

Analyse

- $O(\lg n)$ für RB-Insert
- Für RB-Insert-Fixup
  - Jede Iteration ist $O(1)$
  - Jede Iteration ist entweder die letzte (Fälle 2 und 3) oder bewegt z zwei Stufen nach oben (Fall 1)
  - $O(\lg n)$ Stufen → $O(\lg n)$ Zeit
  - Höchstens 2 Rotationen insgesamt!
- Total: Einfügen in rot-schwarz Baum ist $O(\lg n)$

Übersicht

Rot-schwarz Bäume

- Eigenschaften
- Rotationen
- Einfügen
- (Löschen)

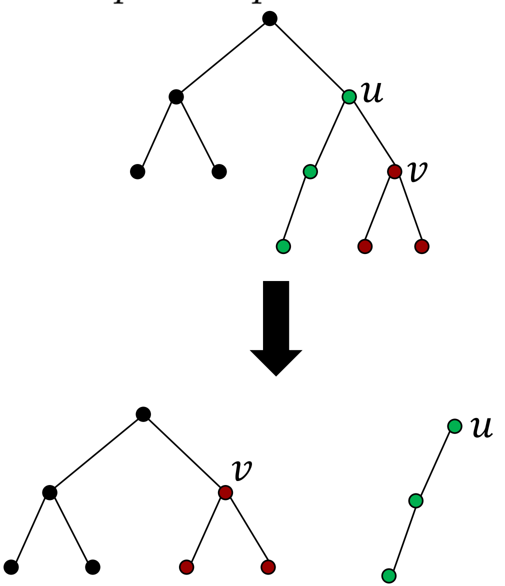
Löschen

- Zwei Schritte

1. Löschen (fast) wie aus binärem Suchbaum
2. Wiederherstellen der rot-schwarz Eigenschaften

Löschen

```
RB-TRANSPLANT(T, u, v)
if u.p == T.nil
        T.root = v
elseif u == u.p.left
        u.p.left = v
else u.p.right = v
v.p = u.p
```



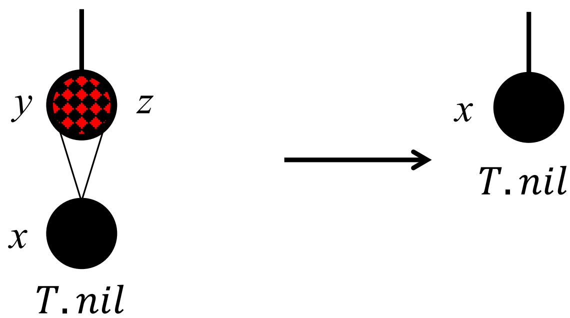

```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
        x = z.right
        RB-TRANSPLANT(T, z, z.right)                Fall 1 & 2
elseif z.right == T.nil
        x = z.left
        RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)                      Fall 3
        y-original-color = y.color
        x = y.right
        if y.p == z
                x.p = y
        else RB-TRANSPLANT(T, y, y.right)
                y.right = z.right                   Fall 3b
                y.right.p = y
        RB-TRANSPLANT(T, z, y)
        y.left = z.left
        y.left.p = y
        y.color = z.color
if y-original-color == BLACK
        RB-DELETE-FIXUP(T, x)
```

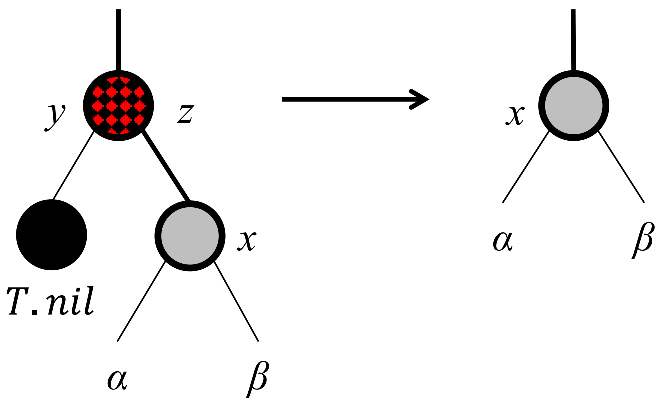
Löschen

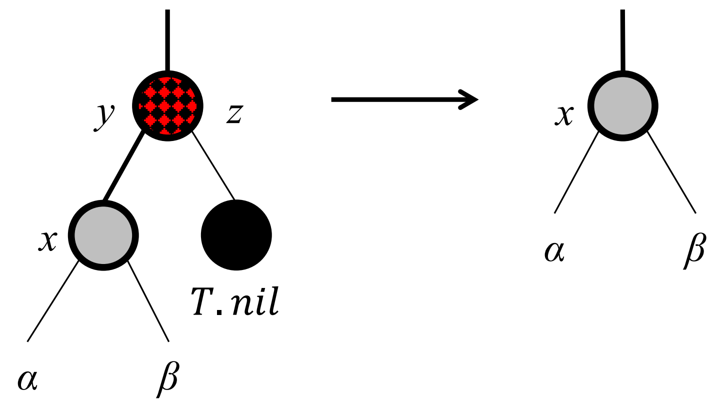
- y wurde ausgeschnitten oder verschoben
- x ist Knoten der ursprüngliche Position von y einnimmt.
  - einziges Kind von y bevor y ausgeschnitten bzw. verschoben wurde
  - oder der Wächter, falls y keine Kinder hatte
- $x.p$ zeigt Position von y's ursprünglichem Vater

Löschen

y wird ausgeschnitten

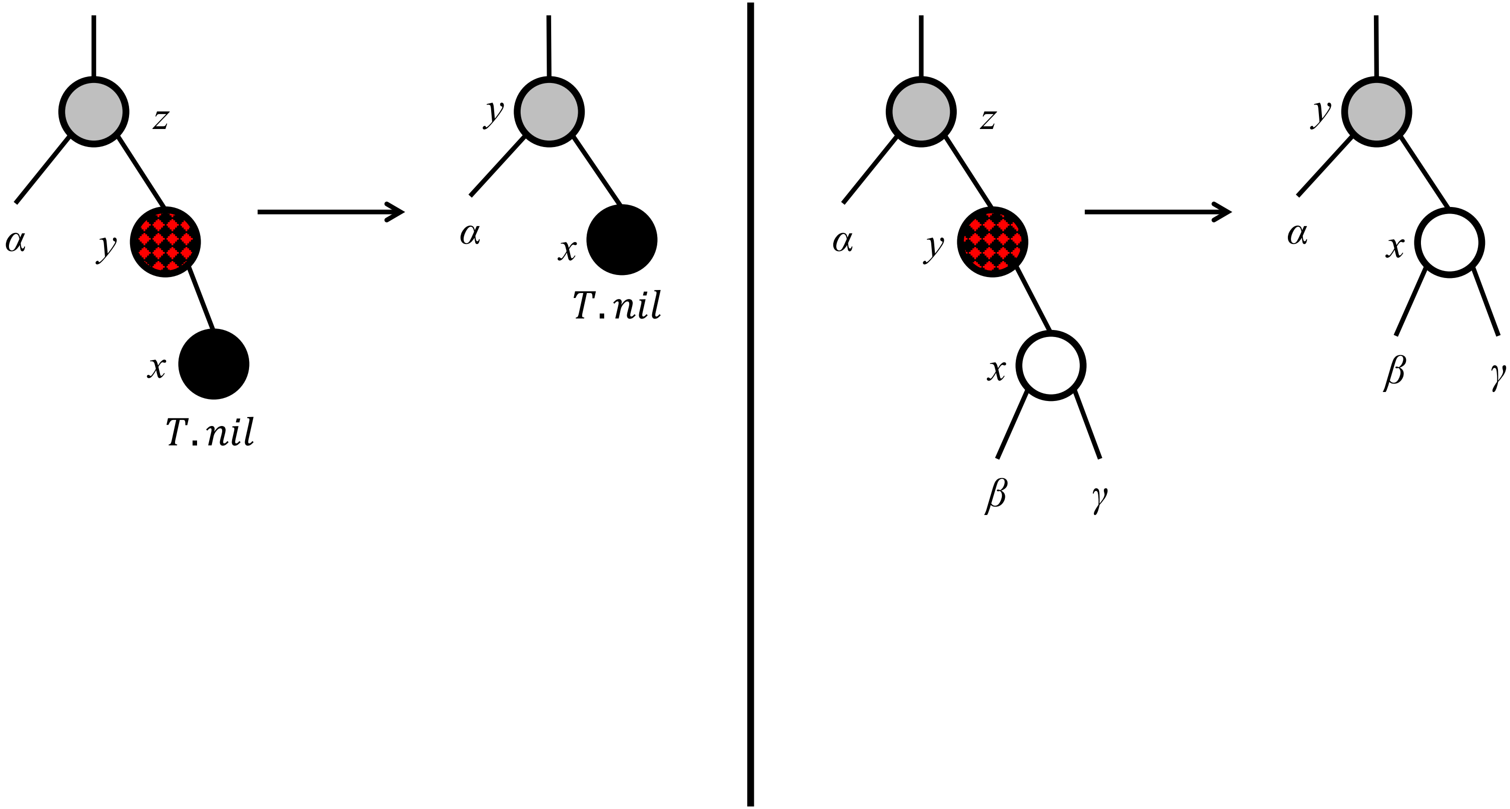






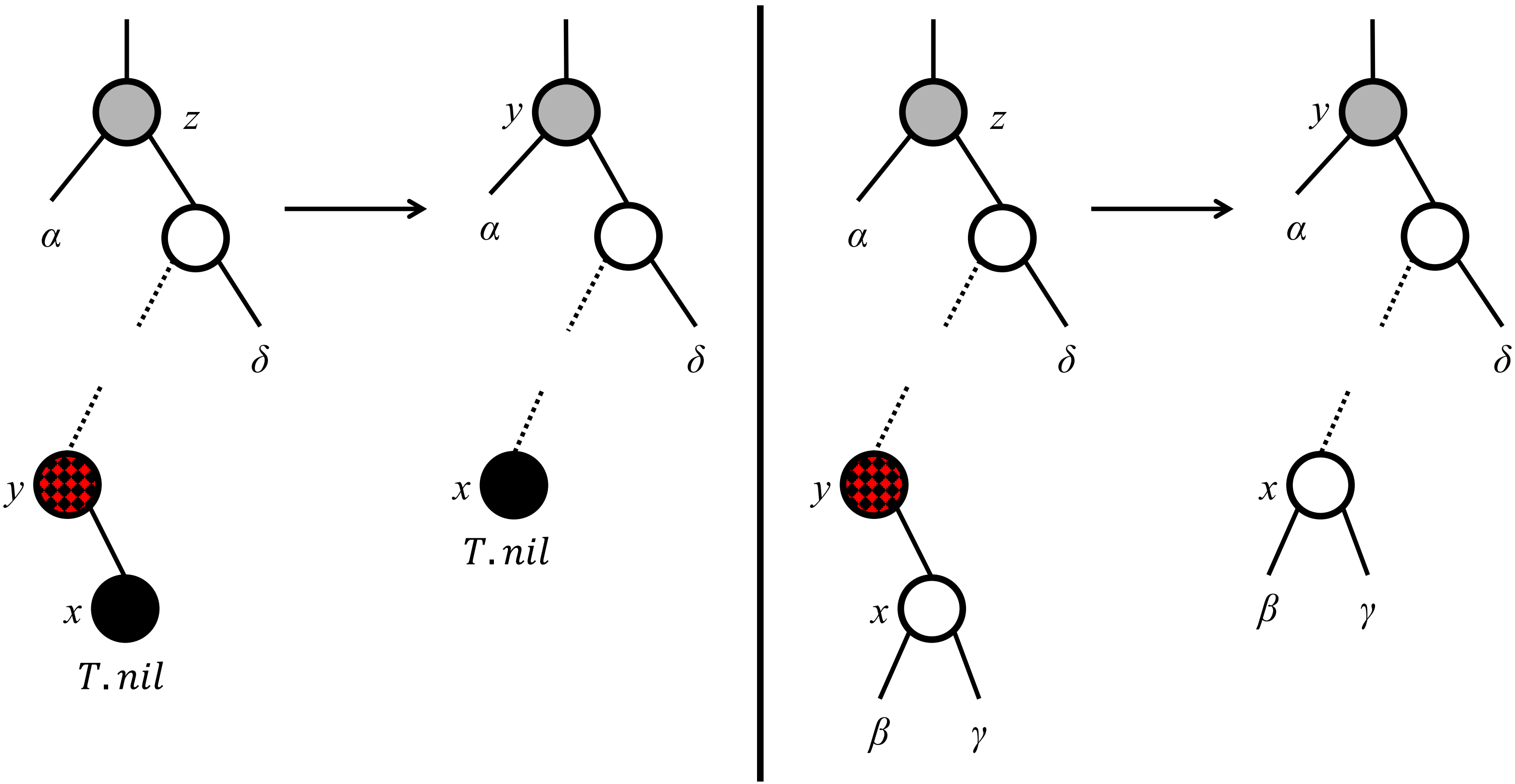
Löschen

y wird verschoben und übernimmt Farbe von z



Löschen

y wird verschoben und
übernimmt Farbe von z



Rot-schwarz Verletzungen

- Falls y rot, keine Verletzungen
  - Keine Veränderung der Schwarz-Höhen
  - Keine neuen roten Knoten
- Falls y schwarz, mögliche Verletzungen
  1. Ok
  2. Falls y die Wurzel und x rot ist, dann wurde Wurzel rot
  3. Ok
  4. Verletzt falls $x.p$ und x rot sind
  5. Jeder Pfad, der y enthielt, hat jetzt einen schwarzen Knoten weniger

Rot-schwarz Verletzungen

- Idee zur Reparatur: x bekommt ein „zusätzliches“ Schwarz
- Alle Pfade, die x enthalten, erhalten $+1$ schwarz
- Eigenschaft 5 repariert, aber Eigenschaft 1 verletzt
  - x ist doppelt schwarz oder rot & schwarz

Reparatur

- Interpretation
  - Der Knoten, auf den x zeigt, hat ein zusätzliches („gedachtes") Schwarz, aber sein Farbattribut wird nicht geändert
- Idee: bewege x im Baum nach oben, bis
  - x auf einen rot & schwarzen Knoten zeigt.
    → Färbe x schwarz
  - x auf die Wurzel zeigt.
    → Zusätzliches Schwarz wird einfach entfernt
  - geeignete Rotationen und Umfärbungen durchgeführt werden können.

Reparatur

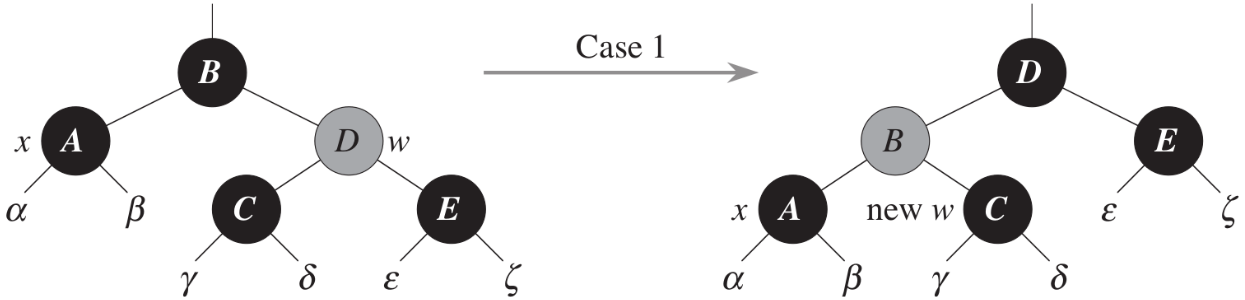
```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK                                   // Fall 1
            x.p.color = RED                                   // Fall 1
            LEFT-ROTATE(T, x.p)                               // Fall 1
            w = x.p.right                                     // Fall 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                                     // Fall 2
            x = x.p                                           // Fall 2
        else if w.right.color == BLACK
                w.left.color = BLACK                          // Fall 3
                w.color = RED                                 // Fall 3
                RIGHT-ROTATE(T, w)                            // Fall 3
                w = x.p.right                                 // Fall 3
            w.color = x.p.color                               // Fall 4
            x.p.color = BLACK                                 // Fall 4
            w.right.color = BLACK                             // Fall 4
            LEFT-ROTATE(T, x.p)                               // Fall 4
            x = T.root                                        // Fall 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

Reparatur

- In der while Schleife
  - zeigt x immer auf einen nicht-Wurzel, doppelt schwarzen Knoten
  - ist w der Bruder von x
  - kann w nicht $T.nil$ sein, weil das Eigenschaft 5 am Knoten $x.p$ verletzen würde
- 8 Fälle, je 4 symmetrisch
  - Hier: betrachten Fälle wo x ein linkes Kind ist
  - Fälle schliessen sich gegenseitig nicht aus
- Idee
  - Jede Transformation erhält Eigenschaft 5
  - Jede Transformation verschiebt x zu $x.p$, oder führt zu Termination der Schleife in der nächsten Iteration

Fall 1

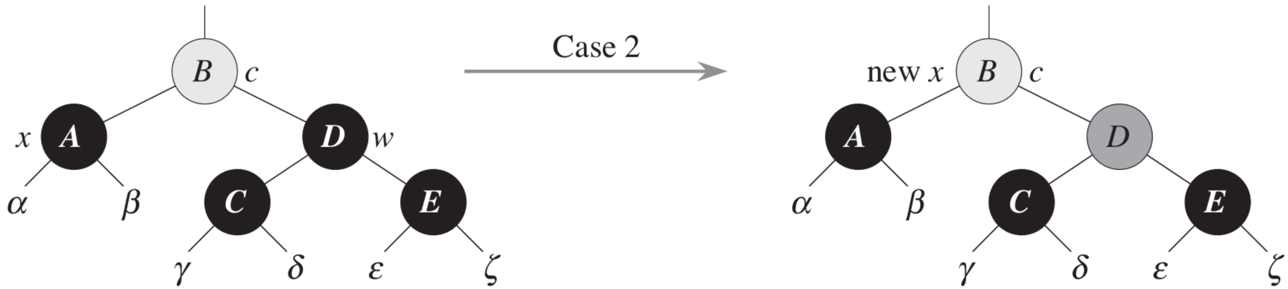
Bruder w von x ist rot



- w muss schwarze Kinder haben
- Färbe w schwarz und $x.p$ rot
- Linksrotation auf $x.p$
- Neuer Bruder von x war Kind von w bevor Rotation → muss schwarz sein
- Gehe zu Fall 2, 3 oder 4

Fall 2

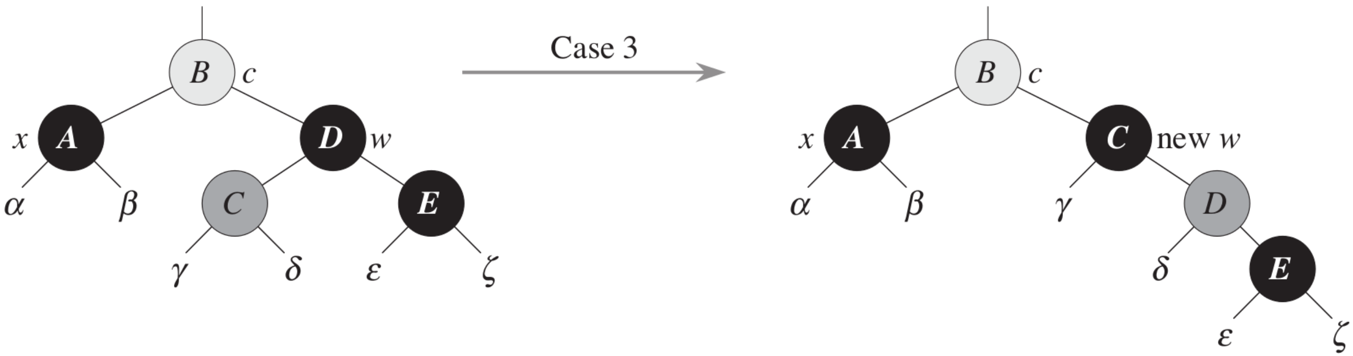
w ist schwarz und Kinder von w sind schwarz



- Entferne 1 Schwarz von x (→ einfach schwarz) und 1 von w (→ rot)
- Bewege Schwarz zu $x.p$
- Nächste Iteration mit $x.p$ als neuem x
  - Falls vorher Fall 1, dann war $x.p$ rot → neues x ist rot & schwarz → Farbattribut von neuem x ist rot → Schleife terminiert, neues x wird schwarz in letzter Zeile

Fall 3

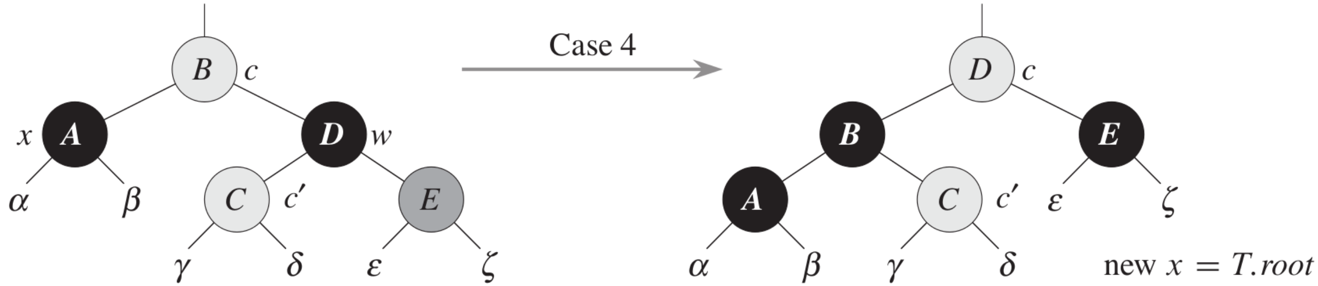
w ist schwarz, linkes Kind rot, rechtes schwarz



- Färbe w rot und linkes Kind von w schwarz
- Rechtsrotation auf w
- Neuer Bruder w von x ist schwarz mit rotem Kind → Fall 4

Fall 4

w ist schwarz, w's rechtes Kind ist rot



- Färbe w gleich wie $x.p$
- Färbe $x.p$ schwarz und rechtes Kind von w schwarz
- Linksrotation auf $x.p$
- Entferne extra schwarz auf x (→ x jetzt einfach schwarz) ohne rot-schwarz Eigenschaften zu verletzen
- Fertig. Lass x auf Wurzel zeigen, Schleife terminiert

Analyse

- RB-Delete ist $O(\lg n)$
- RB-Delete-Fixup
  - Nur Fall 2 erfordert mehr Iterationen
    - x bewegt sich eine Stufe nach oben
    - $O(\lg n)$ Iterationen
  - Fälle 1, 3 und 4 haben eine Rotation
    → höchstens 3 Rotationen total
  - Total $O(\lg n)$

Zusammenfassung

- Binäre Suchbäume
  - Binäre Suchbaum-Eigenschaft
  - Wörterbuchoperationen (Suchen, Einfügen, Löschen)
  - Prioritätswarteschlangen (Minimum, Maximum)
  - Sortierte Ausgabe (Vorgänger, Nachfolger)
- Ohne Zusatzbedingungen: Höhe $h = O(n)$
  - Kann zu Liste degenerieren
- Balancierte Bäume
  - Z.B. rot-schwarz Bäume
  - Zusatzbedingungen an Struktur
  - Garantiert logarithmische Höhe $h = O(\lg n)$
  - Zusatzbedingungen müssen nach Einfügen, Löschen wiederhergestellt werden, Aufwand $O(\lg n)$

Nächstes Mal

- Kapitel 15: dynamisches Programmieren