# Datenstrukturen und Algorithmen

## Vorlesung 11: Rot-Schwarz-Bäume

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

http://www-i2.informatik.rwth-aachen.de/i2/dsal12/

21. Mai 2012

RWTH AACHEN UNIVERSITY





## Übersicht







## Übersicht







## Rot-Schwarz-Bäume (1)

**Rot-Schwarz-Eigenschaft**

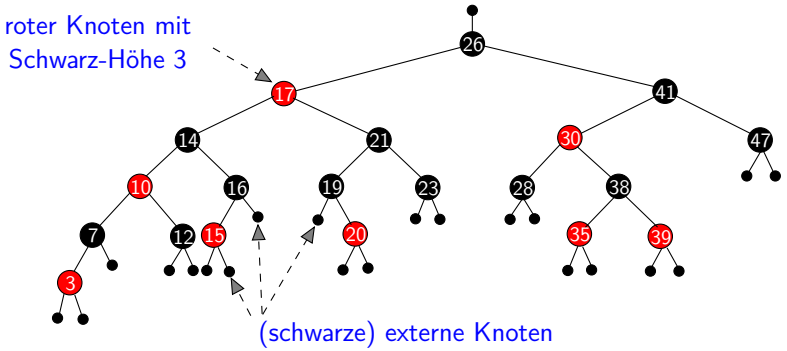
Ein binärer Suchbaum, dessen Knoten jeweils zusätzlich eine Farbe haben, hat die Rot-Schwarz-Eigenschaft, wenn:

1. Jeder Knoten ist entweder rot oder schwarz.
2. Die Wurzel ist schwarz.
3. Jedes (externe) Blatt, d. h. `null`, ist schwarz.
4. Ein roter Knoten hat nur schwarze Kinder.
5. Für jeden Knoten enthalten alle Pfade, die an diesem Knoten starten und in einem Blatt des Teilbaumes dieses Knotens enden, die gleiche Anzahl schwarzer Knoten.

Solche Bäume heißen dann Rot-Schwarz-Bäume (RBT, red-black tree).

- Anders als bei den bekannten BST wird hier zur Vereinfachung `null` als „externes“ Blatt, insbesondere ohne Daten, behandelt.
- In Algorithmen verwenden wir daher `null.color` (`== BLACK`).

# Rot-Schwarz-Bäume (2)




**Definition**

- Die Schwarz-Höhe $bh(x)$ eines Knotens $x$ ist die Anzahl schwarzer Knoten bis zu einem (externen) Blatt, $x$ ausgenommen.

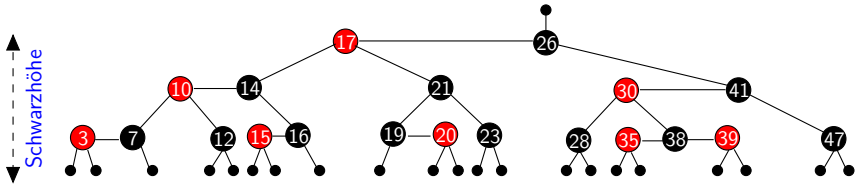
- Die Schwarz-Höhe eines externen Blattes $bh(\texttt{null}) = 0$.





# Rot-Schwarz-Bäume (3)

Zeichnet man die roten Knoten auf der selben Höhe wie ihren Vater, dann erhält man:




- Die externen Knoten werden in Zeichnungen oft weggelassen.

**Definition**

Die Schwarzhöhe $bh(t)$ eines RBT $t$ ist die Schwarz-Höhe seiner Wurzel.





# Elementare Eigenschaften von Rot-Schwarz-Bäumen

**Lemma**

*Ein Rot-Schwarz-Baum $t$ mit Schwarzhöhe $h = bh(t)$ hat:*

- *Mindestens $2^h - 1$ innere Knoten.*
- *Höchstens $4^h - 1$ innere Knoten.*

*Beweis: Induktion über $h$.*

**Theorem**

Ein RBT mit $n$ inneren Knoten hat höchstens die Höhe $2 \cdot \log(n+1)$.

- Damit ist ein RBT ein ziemlich balancierter BST.

$\Rightarrow$ Suchen benötigt also nur $\Theta(\log n)$ statt $\Theta(n)$ Zeit.

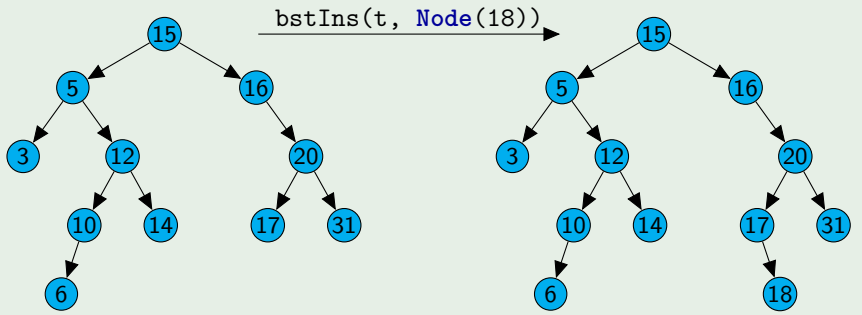
- Für `bstMin`, `bstSucc`, etc. gilt dasselbe.
- Mit Einfügen und Löschen werden wir uns noch beschäftigen.





# Erinnerung: Einfügen in einen BST – Beispiel

**Beispiel**



# Einfügen von Schlüssel $k$ in einen RBT – Strategie

## Einfügen

Zum Einfügen in einen RBT gehen wir zunächst wie beim BST vor:

- Finde einen geeigneten, freien Platz.
- Hänge den neuen Knoten an.

Es bleibt die Frage nach der Farbe:

- Färben wir den neuen Knoten schwarz, dann verletzen wir in der Regel die Schwarz-Höhen-Bedingung.
- Färben wir ihn aber rot, dann könnten wir eine Verletzung der Farbbedingungen bekommen (die Wurzel ist schwarz, rote Knoten haben keine roten Kinder).

$\Rightarrow$ Wir färben den Knoten rot – ein Schwarz-Höhen-Verletzung wäre schwieriger zu behandeln.

- Rot ist sozusagen eine *lokale* Eigenschaft, schwarz eine *globale*.

- Behebe daher im letzten Schritt die Farbverletzung.





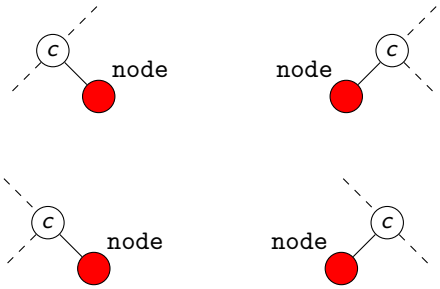
# Einfügen in einen RBT – Algorithmus

```
1 void rbtIns(Tree t, Node node) { // Füge node in den Baum t ein
2   bstIns(t, node); // Einfügen wie beim BST
3   node.left = null;
4   node.right = null;
5   node.color = RED; // eingefügter Knoten immer zunächst rot
6   // stelle Rot-Schwarz-Eigenschaft ggf. wieder her
7   rbtInsFix(t, node);
8 }
```





# Einfügen – Was kann passieren? (1)



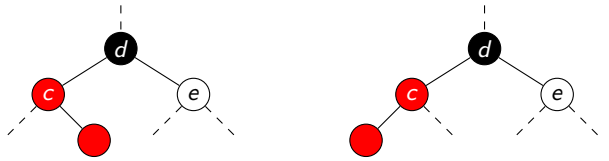

- Der neu eingefügte Knoten ist immer rot.
- Ist die Farbe des Vaterknotens $c$ schwarz (z. B. die Wurzel), haben wir kein Problem.
- Ist $c$ aber rot, dann liegt eine *Rot-Rot-Verletzung* vor, die wir behandeln müssen.
- Die unteren Fälle lassen sich analog (symmetrisch) zu den oberen lösen, daher betrachten wir nur die beiden oberen Situationen.





# Einfügen – Was kann passieren? (2)

Wir müssen nun Großvater $d$ und Onkel $e$ mit berücksichtigen:




- Der Großvater des eingefügten Knotens war schwarz, denn es handelte sich vor dem Einfügen um einen korrekten RBT.

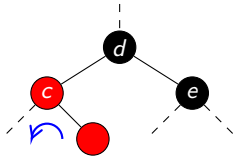
## Fall 1

Ist $e$ rot, dann können wir durch Umfärben von $c$ und $e$ auf schwarz sowie $d$ auf rot die Rot-Schwarz-Eigenschaft lokal wieder herstellen.

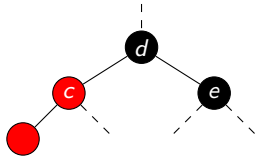
- *Zwei Ebenen* weiter oben könnte nun aber eine *Rot-Rot-Verletzung* vorliegen, die nach dem selben Schema iterativ aufgelöst werden kann.
- Ist $d$ allerdings die Wurzel, dann färben wir sie einfach wieder schwarz. Dadurch erhöht sich die Schwarzhöhe des Baumes um 1.

# Einfügen – Was kann passieren? (3)

Ist der Onkel $e$ dagegen schwarz, dann erhalten wir Fall 2 und 3:




Fall 2




Fall 3

## Fall 2

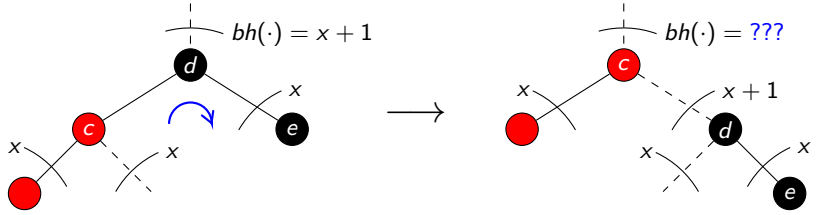
Dieser Fall lässt sich durch Linksrotation um $c$ auf Fall 3 reduzieren.

- Die Schwarz-Höhe des linken Teilbaumes von $d$ ändert sich dadurch nicht.
- Der bisherige Vaterknoten $c$ wird dabei zum linken, roten Kind, das eine Rot-Rot-Verletzung mit dem *neuen Vater* ($c$ im rechten Bild) hat, die wir mit Fall 3 beheben können.





# Einfügen – Was kann passieren? (4)



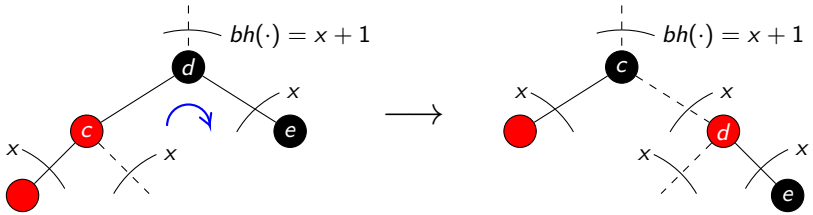

## Fall 3

- Zunächst rotieren wir um $d$ nach rechts, wobei wir die Schwarz-Höhen im Auge behalten.
- Um die Schwarz-Höhen der Kinder von $c$ wieder in Einklang zu bringen, färben wir $d$ rot. Da dessen linkes Kind ursprünglich am roten $c$ hing, ist das soweit unproblematisch.





# Einfügen – Was kann passieren? (4)




## Fall 3

- Zunächst rotieren wir um $d$ nach rechts, wobei wir die Schwarz-Höhen im Auge behalten.
- Um die Schwarz-Höhen der Kinder von $c$ wieder in Einklang zu bringen, färben wir $d$ rot. Da dessen linkes Kind ursprünglich am roten $c$ hing, ist das soweit unproblematisch.
- Färben wir nun $c$ schwarz, dann haben wir wieder einen gültigen RBT. Die Schwarzhöhe des Gesamtbaumes ist unverändert.





# Einfügen in einen RBT – Algorithmus Teil 2

```
1  // Behebe eventuelle Rot-Rot-Verletzung mit Vater
2  void rbtInsFix(Tree t, Node node) {
3    // solange noch eine Rot-Rot-Verletzung besteht
4    while (node.parent.color == RED) {
5      if (node.parent == node.parent.parent.left) {
6        // der von uns betrachtete Fall
7        leftAdjust(t, node);
8        // möglicherweise wurde node = node.parent.parent gesetzt
9      } else {
10       // der dazu symmetrischer Fall
11       rightAdjust(t, node);
12     }
13   }
14   t.root.color = BLACK; // Wurzel bleibt schwarz
15 }
```

# Einfügen in einen RBT – Algorithmus Teil 3

```
1 void leftAdjust(Tree t, Node &node) {
2   Node uncle = node.parent.parent.right;
3   if (uncle.color == RED) { // Fall 1
4     node.parent.parent.color = RED; // Großvater
5     node.parent.color = BLACK; // Vater
6     uncle.color = BLACK; // Onkel
7     node = node.parent.parent; // prüfe Rot-Rot weiter oben
8   } else { // Fall 2 und 3
9     if (node == node.parent.right) { // Fall 2
10      // dieser Knoten wird das linke, rote Kind:
11      node = node.parent;
12      leftRotate(t, node);
13    } // Fall 3
14    rightRotate(t, node.parent.parent);
15    node.parent.color = BLACK;
16    node.parent.right.color = RED;
17  }
18 }
```





# Einfügen in einen RBT – Analyse

**Zeitkomplexität Einfügen**

Die Worst-Case Laufzeit von `rbtIns` für ein RBT mit $n$ inneren Knoten ist $O(\log n)$.

**Beweisskizze:**

- Die Worst-Case Laufzeit von `bstIns` ist $O(\log n)$.
- Die Schleife in `rbtInsFix` wird nur wiederholt wenn Fall 1 auftrittt. Dann steigt der Zeiger `node` zwei Ebenen im Baum auf.
- Die maximal Anzahl der Schleifen ist damit $O(\log n)$.
- Maximal 2 Rotationen werden ausgeführt, da die Schleife in `rbtInsFix` terminiert, wenn die Fälle 2 oder 3 auftreten. (Fall 1 involviert keine Rotationen.)

$\Rightarrow$ Die Gesamtanzahl der Rotationen ist konstant und eine Rotation läuft in $O(1)$.

- Somit benötigt `rbtIns` eine Gesamtzeit $O(\log n)$.





# Erinnerung: Löschen im BST – Strategie

**Löschen**

Um Knoten `node` aus dem BST zu löschen, verfahren wir folgendermaßen:

`node` hat keine Kinder:
Ersetze im Vaterknoten von `node` den Zeiger auf `node` durch `null`.

`node` hat ein Kind:
Wir schneiden `node` aus, indem wir den Vater und das Kind direkt miteinander verbinden.

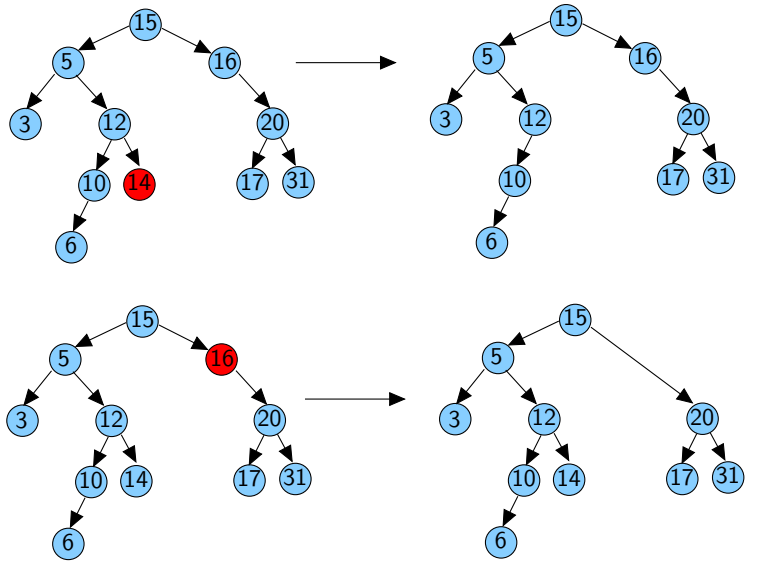
`node` hat zwei Kinder:
Wir finden den Nachfolger von `node`, entfernen ihn aus seiner ursprünglichen Position und ersetzen `node` durch den Nachfolger.

- Der Nachfolger hat höchstens ein Kind.

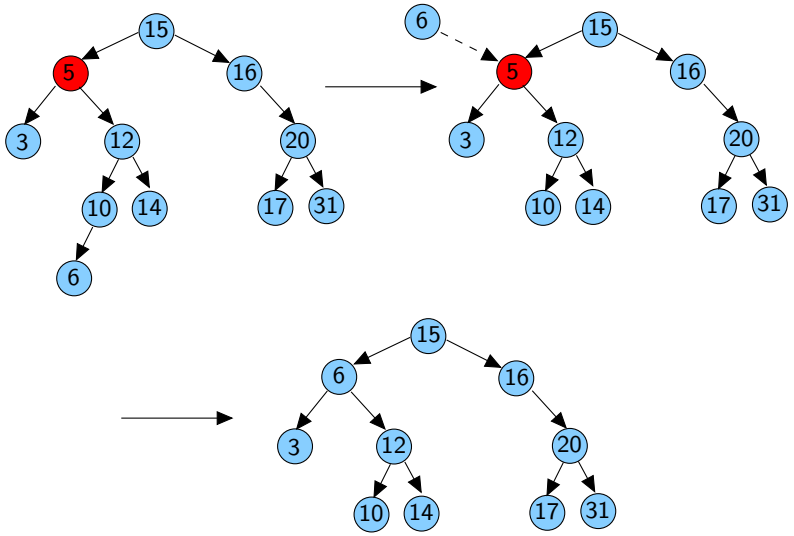




# Löschen im BST: Die beiden einfachen Fälle



# Löschen im BST: Der aufwändigere Fall








# Löschen im RBT – Strategie (1)

## Löschen

Damit der RBT auch nach dem Löschen noch ein RBT ist, müssen wir das Löschverfahren für BSTs ergänzen:

- Haben wir einen roten Knoten (wirklich) gelöscht, bleibt alles beim alten, da:
  1. im Baum werden keine Schwarzhöhen geändert
  2. es sind keine benachbarten roten Knoten entstanden
  3. der gelöschte Knoten war rot, und damit bleibt die Wurzel schwarz.

⇒ Also: das Löschen eines roten Knotens liefert ein RBT.

- Haben wir dagegen einen schwarzen Knoten gelöscht, dann haben wir auf dem Pfad von der Wurzel zum gelöschten Knoten, bzw. zum an seine Stelle getretenen Knoten einen Schwarzwert zu viel.

⇒ Behebe diese Schwarz-Höhen-Verletzung.





# Löschen im RBT – Strategie (2)

## Löschen

Insbesondere für den Fall `node` hat zwei Kinder:

1. Wir finden den Nachfolger von `node`
2. Entfernen ihn (mittels `rbtDel`) aus seiner ursprünglichen Position
3. Und beheben dabei die möglich auftretende Farbverletzung
4. Ersetzen `node` durch den Nachfolger, und
5. Übernehmen dabei die möglicherweise *neue* Farbe von `node`.

Dadurch ändert sich *nichts mehr* an den Farben, der *RBT bleibt gültig*!





# Erinnerung: Löschen im BST – Algorithmus

```
1 // Entfernt node aus dem Baum.
2 // Danach kann node ggf. auch aus dem Speicher entfernt werden.
3 void bstDel(Tree t, Node node) {
4   if (node.left && node.right) { // zwei Kinder
5     Node tmp = bstMin(node.right);
6     bstDel(t, tmp); // höchstens ein Kind, rechts
7     bstSwap(t, node, tmp);
8   } else if (node.left) { // ein Kind, links
9     bstReplace(t, node, node.left);
10  } else { // ein Kind, oder kein Kind (node.right == null)
11    bstReplace(t, node, node.right);
12  }
13 }
```

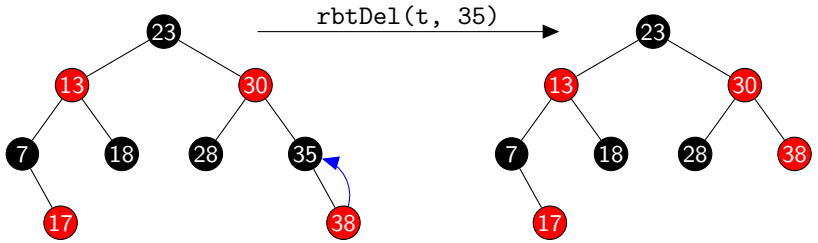
# Löschen im RBT – Algorithmus Teil 1

```
1  // Entfernt node aus dem Baum.
2  void rbtDel(Tree t, Node node) {
3    if (node.left && node.right) { // zwei Kinder
4      Node tmp = bstMin(node.right);
5      rbtDel(t, tmp); // höchstens ein Kind, rechts
6      bstSwap(t, node, tmp);
7      tmp.color = node.color; // übernimm die Farbe
8    } else if (node.left) { // ein Kind, links
9      bstReplace(t, node, node.left);
10     if (node.color == BLACK) { // gelöschter Knoten war schwarz
11       rbtDelFix(t, node.left);
12     }
13   } else { // ein Kind, oder kein Kind (node.right == null)
14     bstReplace(t, node, node.right);
15     if (node.color == BLACK) {
16       // wenn node.right == null übergebe node (statt null)
17       rbtDelFix(t, node.right || node);
18     }
19   }
20 }
```





# Löschen im RBT – Beispiel 1

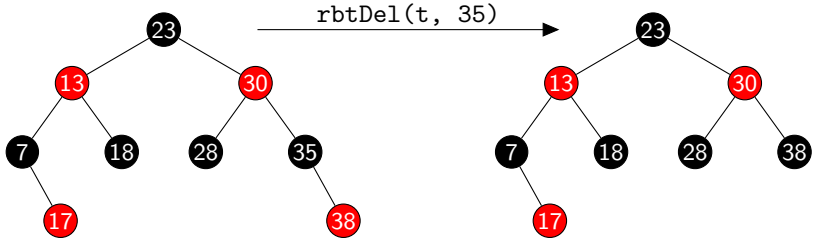



- Wie beim BST tritt der rechte Teilbaum von 35 an dessen Stelle.
- Der nun fehlende Schwarzwert wird 38 zugeschoben.
  Einfaches Umfärben auf schwarz genügt hier. Wäre 38 bereits schwarz gewesen, hätten wir die Verletzung aufwändiger weiter oben beheben müssen.





# Löschen im RBT – Beispiel 1

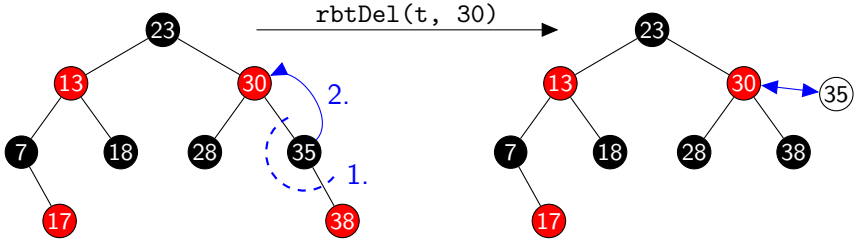



- Wie beim BST tritt der rechte Teilbaum von 35 an dessen Stelle.
- Der nun fehlende Schwarzwert wird 38 zugeschoben.
  Einfaches Umfärben auf schwarz genügt hier. Wäre 38 bereits schwarz gewesen, hätten wir die Verletzung aufwändiger weiter oben beheben müssen.



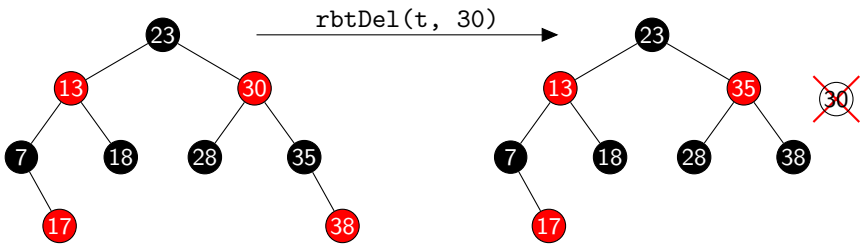


# Löschen im RBT – Beispiel 2




- Da 30 zwei Kinder hat, finde den Nachfolger.
- Lösche 35 und stelle die RBT-Eigenschaft wieder her (siehe voriges Beispiel).
- Ersetze 30 durch die nun freie 35, wobei die Farbe von 30 übernommen wird.

# Löschen im RBT – Beispiel 2




- Da 30 zwei Kinder hat, finde den Nachfolger.
- Lösche 35 und stelle die RBT-Eigenschaft wieder her (siehe voriges Beispiel).
- Ersetze 30 durch die nun freie 35, wobei die Farbe von 30 übernommen wird.





# Löschen im RBT – Algorithmus Teil 2

```
1 // Auf dem Pfad von der Wurzel nach node fehlt ein Schwarzwert,
2 // man könnte sagen: node ist "doppelt-schwarz".
3 void rbtDelFix(Tree t, Node node) {
4   // solange der Schwarzwert nicht eingefügt werden kann
5   while (node.parent && node.color == BLACK) {
6     if (node == node.parent.left) { // ist linkes Kind
7       delLeftAdjust(t, node);
8       // möglicherweise ist node = node.parent gesetzt worden
9     } else { // ist/war(!) rechtes Kind
10      // symmetrischer Fall
11      delRightAdjust(t, node);
12    }
13  }
14  node.color = BLACK; // dieser Knoten war rot (bzw. Wurzel).
15 }
```





# Löschen im RBT – Algorithmus Teil 3a

```
1 void delLeftAdjust(Tree t, Node &node) {
2   // brother existiert wegen Schwarzhöhe immer
3   // (Spezialfall, delRightAdjust: node.parent.right == null)
4   Node brother = node.parent.right; // ( || node; )
5   if (brother.color == RED) { // Fall 1: Reduktion auf 2,3,4
6     brother.color = BLACK;
7     node.parent.color = RED; // Vater
8     leftRotate(t, node.parent);
9     brother = node.parent.right; // neuer Bruder
10  }
```

$\rightarrow$





# Löschen im RBT – Algorithmus Teil 3b

```
9   if (brother.left.color == BLACK &&
10      brother.right.color == BLACK) { // Fall 2
11    brother.color = RED;
12    node = node.parent; // Doppel-schwarz weiter oben...
13  } else { // Fall 3 und 4
14    if (brother.right.color == BLACK) // Fall 3
15      brother.left.color = BLACK;
16      brother.color = RED;
17      rightRotate(t, brother);
18      brother = node.parent.right; // ab jetzt von hier aus
19    } // Fall 4
20    brother.color = node.parent.color;
21    node.parent.color = BLACK;
22    brother.right.color = BLACK;
23    leftRotate(t, node.parent);
24    node = t.root; // Fertig.
25  }
26 }
```

# Löschen im RBT – Analyse

**Zeitkomplexität Löschen**

Die Worst-Case Laufzeit von `rbtDel` für ein RBT mit $n$ inneren Knoten ist $O(\log n)$.

**Beweisskizze:**

- Die Laufzeit von `rbtDel` ohne Aufrufe von `rbtDelFix` ist $O(\log n)$.
- Die Fälle 2, 3, und 4 brauchen höchstens 3 Rotationen und eine konstante Anzahl Farbänderungen.
- Die Schleife in `rbtDelFix` wird nur wiederholt wenn Fall 2 auftrittt. Dann steigt der Zeiger `node` eine Ebene im Baum auf.
- Die maximal Anzahl der Schleifen ist damit $O(\log n)$.
- Somit benötigt `rbtDel` eine Gesamtzeit $O(\log n)$.





# Komplexität der RBT-Operationen

| Operation | Zeit |
|---|---|
| `bstSearch` | $\Theta(h)$ |
| `bstSucc` | $\Theta(h)$ |
| `bstMin` | $\Theta(h)$ |
| `bstIns` | $\Theta(h)$ |
| `bstDel` | $\Theta(h)$ |

| Operation | Zeit |
|---|---|
| `rbtIns` | $\Theta(\log n)$ |
| `rbtDel` | $\Theta(\log n)$ |

- Alle anderen Operationen wie beim BST, wobei $h = \log n$.

Alle Operationen sind logarithmisch in der Größe des Rot-Schwarz-Baumes