Kapitel ADS:V

V. Suchen

- Binary Search Tree
- AVL Tree
- Red-Black Tree
- B-Tree

Binary Search Tree

Definition

Ein Binärbaum T heißt Binary Search Tree (*Binärer Suchbaum*), wenn jeder seiner Knoten x folgende Bedingung erfüllt:

$$y.\mathit{key} \;\leq\; x.\mathit{key} \;\leq\; z.\mathit{key}$$

wobei y ein Knoten im linken Teilbaum von x ist und z ein Knoten im rechten.

Binary Search Tree

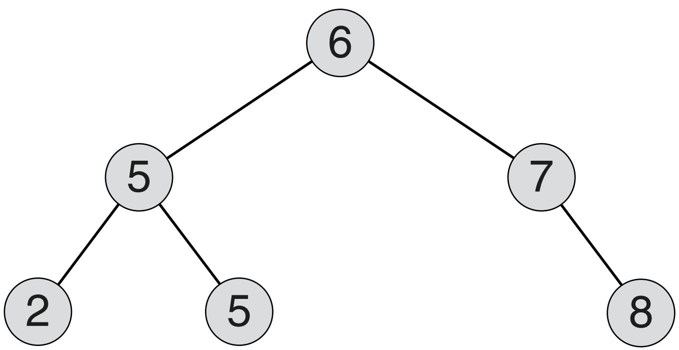
Definition

Ein Binärbaum T heißt Binary Search Tree (*Binärer Suchbaum*), wenn jeder seiner Knoten x folgende Bedingung erfüllt:

$$y.\textit{key} \;\leq\; x.\textit{key} \;\leq\; z.\textit{key}$$

wobei y ein Knoten im linken Teilbaum von x ist und z ein Knoten im rechten.

Beispiel:



Binary Search Tree

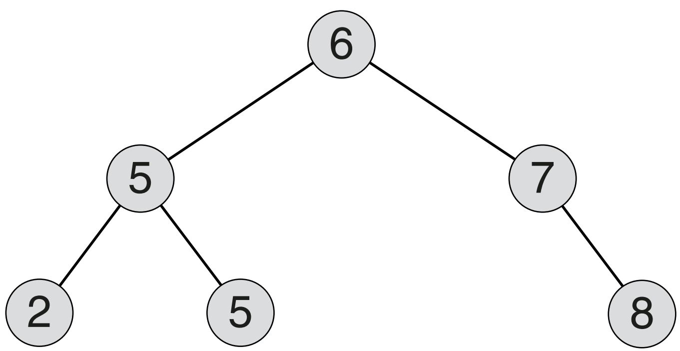
Definition

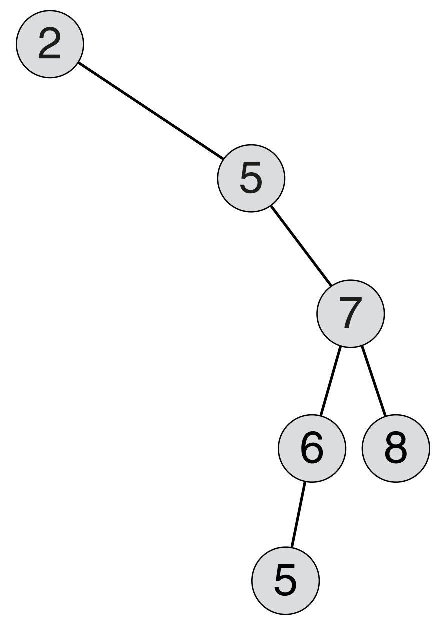
Ein Binärbaum T heißt Binary Search Tree (*Binärer Suchbaum*), wenn jeder seiner Knoten x folgende Bedingung erfüllt:

$$y.\textit{key} \ \leq \ x.\textit{key} \ \leq \ z.\textit{key}$$

wobei y ein Knoten im linken Teilbaum von x ist und z ein Knoten im rechten.

Beispiel:





Binary Search Tree

Implementierung

- Link-basierter Binärbaum

Binary Search Tree

Implementierung

- Link-basierter Binärbaum

Manipulation

- Knoten in Sortierreihenfolge besuchen
  Traversierung des Baumes mit DFS-Traverse (in-order).
- Knoten suchen (*Search*)
  Einen Knoten mit vorgegebenem Schlüssel suchen.
- Minimum, Maximum, oder Nachfolger (*Successor*) bestimmen
  Den Knoten mit kleinstem, größten oder nächstgrößten Sortierschlüssel bestimmten.
- Knoten einfügen (*Insert*)
  Einen Knoten an der richtigen Stelle im Baum einfügen.
- Knoten löschen (*Delete*)
  Einen bestimmten Knoten aus dem Baum löschen.
- Knoten verändern
  Den Schlüssel eines bestimmten Knotens ändern.

Bemerkungen:

- Die Bedingung, die ein Binary Search Tree erfüllen muss, wird auch „Binary Search Tree Property“ genannt.
- Die Implementierung eines Binary Search Tree ist link-basiert, damit alle Manipulationsoperationen effizient umsetzbar sind.
- Wenn die Menge zu durchsuchender Elemente statisch ist, genügt es, sie Array-basiert zu speichern, zu sortieren und die binäre Suche anzuwenden.

Binary Search Tree

Manipulation: Suche

Algorithmus: Tree Search.

Eingabe: x. Wurzel eines Binary Search Tree T.
k. Gesuchter Schlüssel.

Ausgabe: Knoten, der k als Schlüssel hat, oder *NIL*.

Binary Search Tree

Manipulation: Suche

Algorithmus: Tree Search.

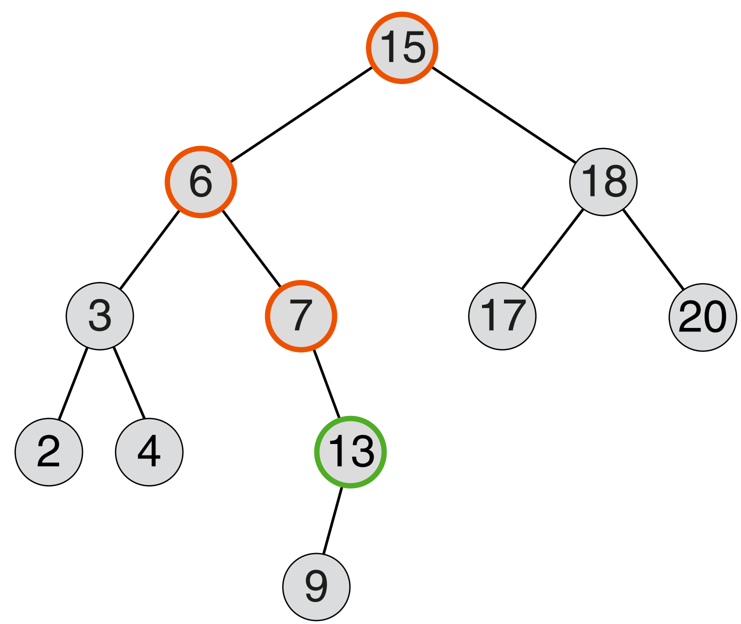
Eingabe: x. Wurzel eines Binary Search Tree T.
k. Gesuchter Schlüssel.

Ausgabe: Knoten, der k als Schlüssel hat, oder *NIL*.

TreeSearch(x, k)

```
WHILE x ≠ NIL AND k ≠ x.key THEN
  IF k < x.key THEN
    x = x.left
  ELSE
    x = x.right
  ENDIF
ENDDO
return(x)
```

Beispiel: *TreeSearch*$(T.root, 13)$



Binary Search Tree

Manipulation: Suche

Algorithmus: Tree Search.

Eingabe: x. Wurzel eines Binary Search Tree T.
k. Gesuchter Schlüssel.

Ausgabe: Knoten, der k als Schlüssel hat, oder *NIL*.

TreeSearch(x, k)

```
WHILE x ≠ NIL AND k ≠ x.key THEN
  IF k < x.key THEN
    x = x.left
  ELSE
    x = x.right
  ENDIF
ENDDO
return(x)
```

Laufzeit:

- Iterationen der While-Schleife?

Binary Search Tree

Manipulation: Suche

Algorithmus: Tree Search.

Eingabe: x. Wurzel eines Binary Search Tree T.
k. Gesuchter Schlüssel.

Ausgabe: Knoten, der k als Schlüssel hat, oder *NIL*.

TreeSearch(x, k)

```
WHILE x ≠ NIL AND k ≠ x.key THEN
  IF k < x.key THEN
    x = x.left
  ELSE
    x = x.right
  ENDIF
ENDDO
return(x)
```

Laufzeit:

- Iterationen der While-Schleife: Längster Pfad von der Wurzel zu einem Blattknoten.

Binary Search Tree

Manipulation: Suche

Algorithmus: Tree Search.

Eingabe: x. Wurzel eines Binary Search Tree T.
k. Gesuchter Schlüssel.

Ausgabe: Knoten, der k als Schlüssel hat, oder *NIL*.

TreeSearch(x, k)

```
WHILE x ≠ NIL AND k ≠ x.key THEN
  IF k < x.key THEN
    x = x.left
  ELSE
    x = x.right
  ENDIF
ENDDO
return(x)
```

Laufzeit:

- Iterationen der While-Schleife: Längster Pfad von der Wurzel zu einem Blattknoten.

➜ $O(h)$, wobei h die Baumhöhe ist.

Binary Search Tree

Manipulation: Minimum

Algorithmus: Tree Minimum.

Eingabe: x. Wurzel eines Binary Search Tree T.

Ausgabe: Knoten mit dem kleinsten Schlüssel, oder *NIL* falls der Baum leer ist.

Binary Search Tree

Manipulation: Minimum

Algorithmus: Tree Minimum.

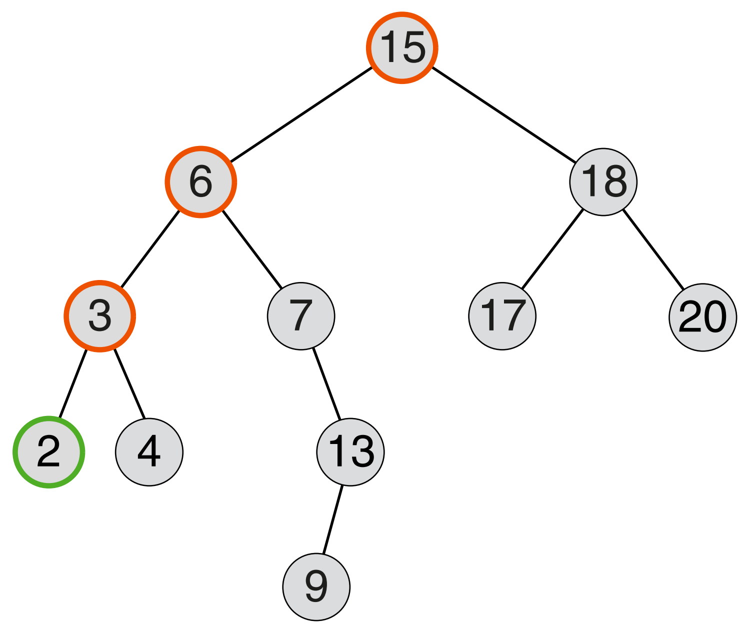
Eingabe: x. Wurzel eines Binary Search Tree T.

Ausgabe: Knoten mit dem kleinsten Schlüssel, oder *NIL* falls der Baum leer ist.

TreeMinimum(x)

```
IF x == NIL THEN return(NIL) ENDIF
WHILE x.left ≠ NIL DO
   x = x.left
ENDDO
return(x)
```

Beispiel: *TreeMinimum*$(T.$*root*$)$



Binary Search Tree

Manipulation: Maximum

Algorithmus: Tree Maximum.

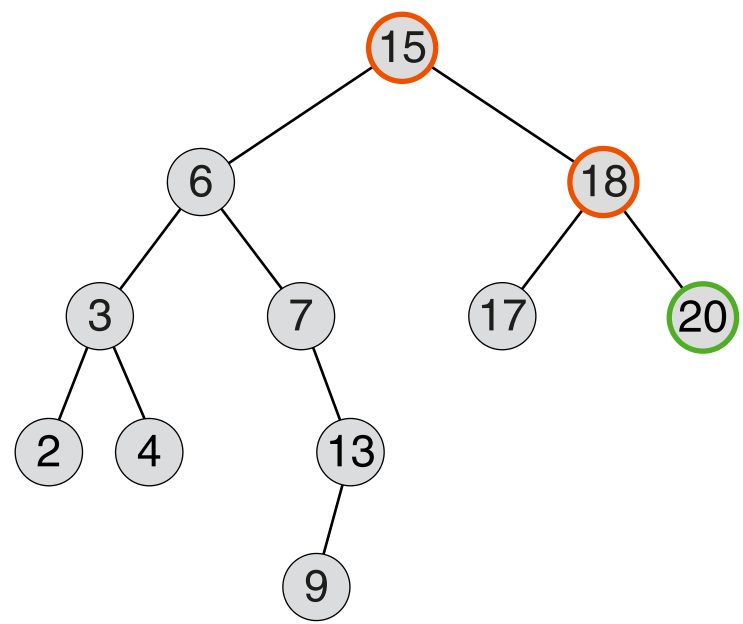
Eingabe: x. Wurzel eines Binary Search Tree T.

Ausgabe: Knoten mit dem größten Schlüssel, oder *NIL* falls der Baum leer ist.

TreeMaximum(x)

```
IF x == NIL THEN return(NIL) ENDIF
WHILE x.right ≠ NIL DO
    x = x.right
ENDDO
return(x)
```

Beispiel: *TreeMaximum*$(T.\textit{root})$



Binary Search Tree

Manipulation: Maximum

Algorithmus: Tree Maximum.

Eingabe: x. Wurzel eines Binary Search Tree T.

Ausgabe: Knoten mit dem größten Schlüssel, oder *NIL* falls der Baum leer ist.

TreeMaximum(x)

IF $x ==$ *NIL* **THEN** *return*(*NIL*) **ENDIF**
WHILE x.*right* $\neq$ *NIL* **DO**
    $x = x$.*right*
ENDDO
return(x)

Laufzeit:

- Iterationen der While-Schleife: Längster rechter Pfad von der Wurzel zu einem Blattknoten.

→ $O(h)$, wobei h die Baumhöhe ist.

- Analog für *TreeMinimum*.

Binary Search Tree

Manipulation: Nachfolger

Algorithmus: Tree Successor.

Eingabe: x. Knoten eines Binary Search Tree T.

Ausgabe: Knoten mit dem nächstgrößeren Schlüssel, oder *NIL*.

Binary Search Tree

Manipulation: Nachfolger

Algorithmus: Tree Successor.

Eingabe: x. Knoten eines Binary Search Tree T.

Ausgabe: Knoten mit dem nächstgrößeren Schlüssel, oder *NIL*.

Fallunterscheidung:

- x hat ein rechtes Kind.

  Suche den kleinsten Knoten im rechten Teilbaum von x.

- x hat kein rechtes Kind.

  Wandere solange Richtung Wurzel, bis der erste Knoten y erreicht ist, in dessen linken Teilbaum sich x befindet. Wird die Wurzel vorher erreicht, hat x keinen Nachfolger.

Binary Search Tree

Manipulation: Nachfolger

Algorithmus: Tree Successor.

Eingabe: x. Knoten eines Binary Search Tree T.

Ausgabe: Knoten mit dem nächstgrößeren Schlüssel, oder *NIL*.

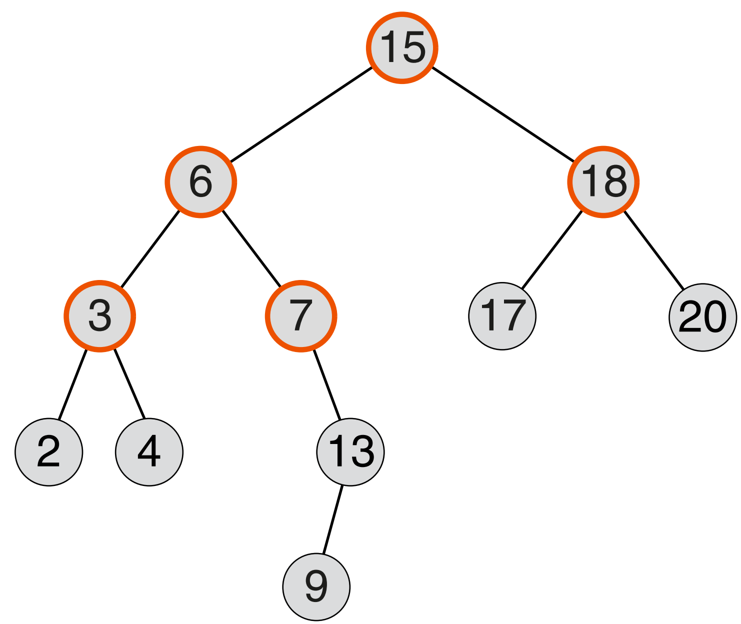
TreeSuccessor(x)

```
IF x.right ≠ NIL THEN
   return(TreeMinimum(x.right))
ENDIF

y = x.parent
WHILE y ≠ NIL AND x == y.right DO
   x = y
   y = y.parent
ENDDO

return(y)
```

Beispiel: *TreeSuccessor*(x)



Binary Search Tree

Manipulation: Nachfolger

Algorithmus: Tree Successor.

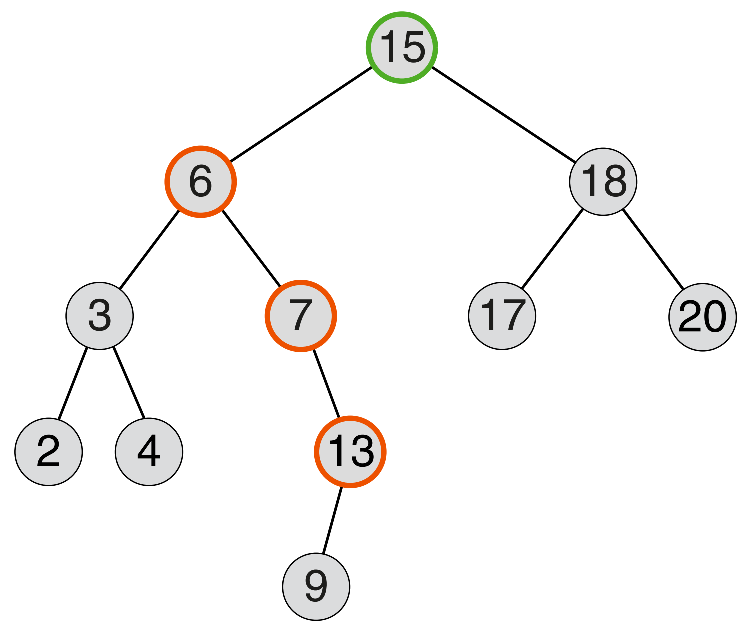
Eingabe: x. Knoten eines Binary Search Tree T.

Ausgabe: Knoten mit dem nächstgrößeren Schlüssel, oder *NIL*.

TreeSuccessor(x)

IF x.*right* $\neq$ *NIL* **THEN**
    return(*TreeMinimum*(x.*right*))
ENDIF
$y = x$.*parent*
WHILE $y \neq$ *NIL* **AND** $x == y$.*right* **DO**
    $x = y$
    $y = y$.*parent*
ENDDO
return(y)

Beispiel: *TreeSuccessor*(x), x.*key* $= 13$



Binary Search Tree

Manipulation: Nachfolger

Algorithmus: Tree Successor.

Eingabe: x. Knoten eines Binary Search Tree T.

Ausgabe: Knoten mit dem nächstgrößeren Schlüssel, oder *NIL*.

TreeSuccessor(x)

```
IF x.right ≠ NIL THEN
   return(TreeMinimum(x.right))
ENDIF

y = x.parent
WHILE y ≠ NIL AND x == y.right DO
   x = y
   y = y.parent
ENDDO

return(y)
```

Laufzeit:

- Iterationen der While-Schleife: Längster Pfad von der Wurzel zu einem Blattknoten

➜ $O(h)$, wobei h die Baumhöhe ist.

Binary Search Tree

Manipulation: Einfügen

Algorithmus: Tree Insert.

Eingabe: T. Binary Search Tree.
z. Einzufügender Knoten mit Schlüssel k.

Ausgabe: Um z erweiterter Binary Search Tree.

Vorgehen:

- Finde den Elter y des einzufügenden Knotens.
- Falls der Baum leer ist, füge den Knoten als neue Wurzel ein.
- Andernfalls, füge den Knoten gemäß der Binary Search Tree Property als linkes oder rechts Kind von y ein.

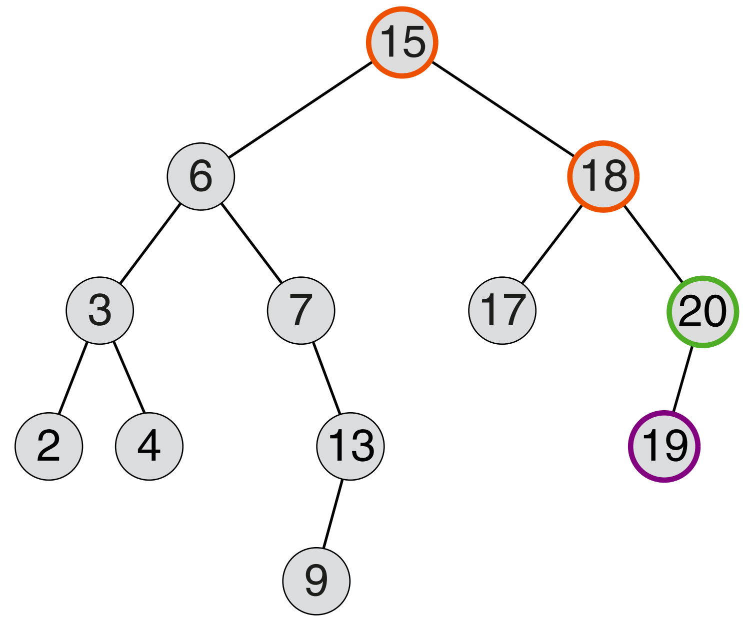
Binary Search Tree

Manipulation: Einfügen

TreeInsert(T, z)

```
y = NIL
x = T.root
WHILE x ≠ NIL DO
   y = x
   IF z.key < x.key THEN
      x = x.left
   ELSE
      x = x.right
   ENDIF
ENDDO
z.parent = y
IF y == NIL THEN
   T.root = z
ELSE IF z.key < y.key THEN
   y.left = z
ELSE
   y.right = z
ENDIF
```

Beispiel: *TreeInsert*(T, z), $z.key = 19$



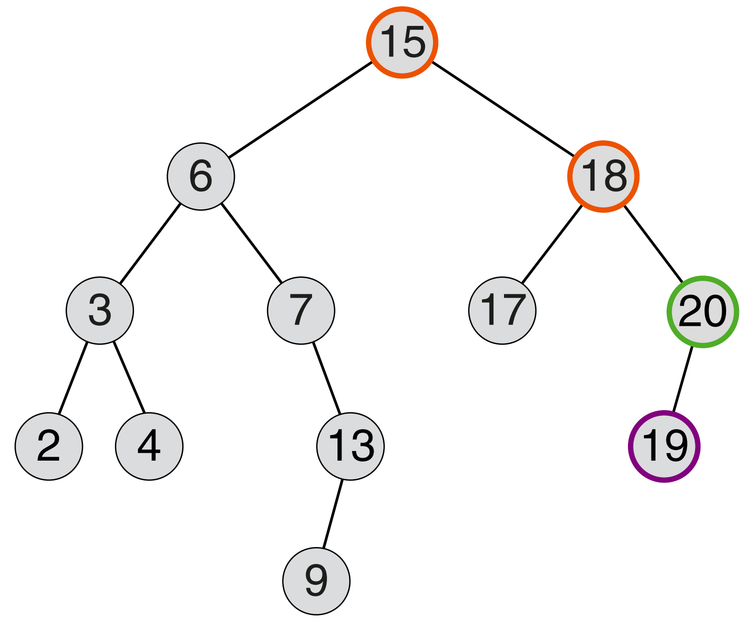
Binary Search Tree

Manipulation: Einfügen

TreeInsert(T, z)

```
y = NIL
x = T.root
WHILE x ≠ NIL DO
   y = x
   IF z.key < x.key THEN
      x = x.left
   ELSE
      x = x.right
   ENDIF
ENDDO
z.parent = y
IF y == NIL THEN
   T.root = z
ELSE IF z.key < y.key THEN
   y.left = z
ELSE
   y.right = z
ENDIF
```

Beispiel: *TreeInsert*(T, z), $z.key = 19$



Laufzeit:

- Iterationen der While-Schleife: Längster Pfad von der Wurzel zu einem Blattknoten

→ $O(h)$, wobei h die Baumhöhe ist.

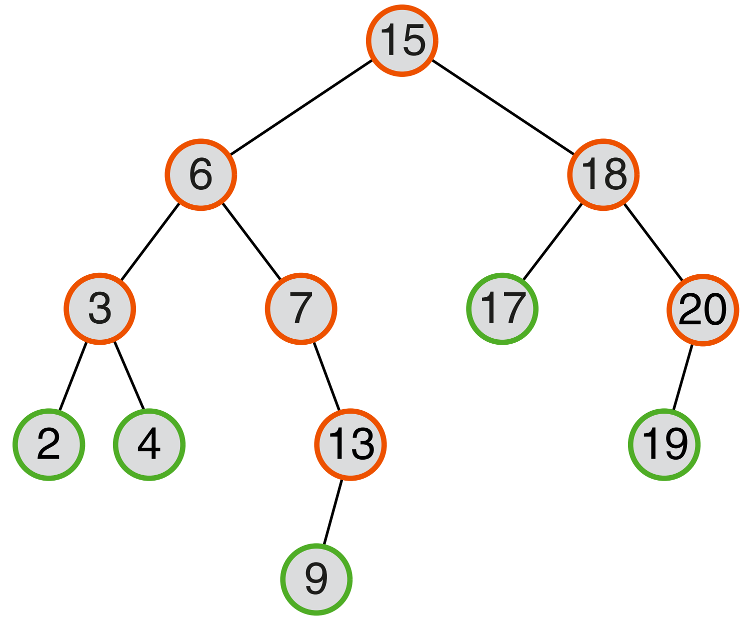
Binary Search Tree

Manipulation: Löschen

Vorüberlegungen:

- Äußere Knoten
  Können gefahrlos entfernt werden.
- Innere Knoten
  Das Löschen eines inneren Knotens hinterlässt einen Wald, aus dem ein gültiger Suchbaum gebildet werden muss.

Beispiel:



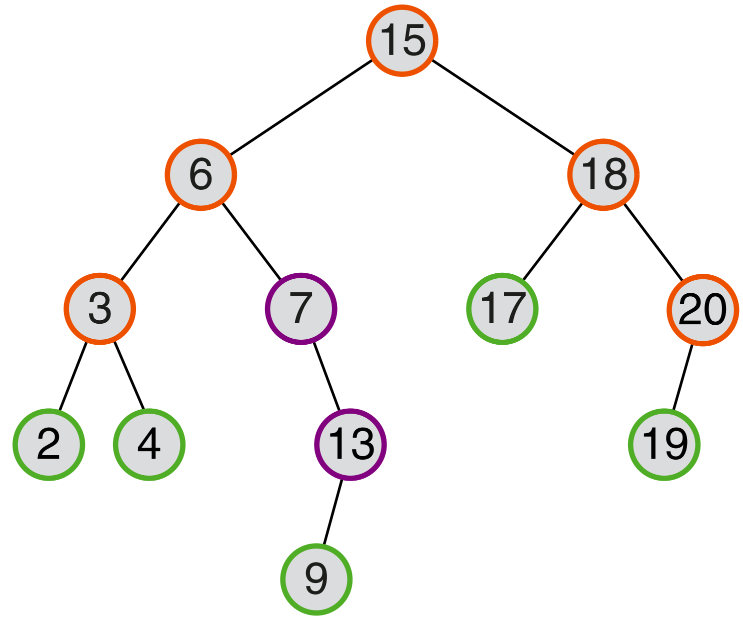
Binary Search Tree

Manipulation: Löschen

Vorüberlegungen:

- Äußere Knoten
  Können gefahrlos entfernt werden.
- Innere Knoten
  Das Löschen eines inneren Knotens hinterlässt einen Wald, aus dem ein gültiger Suchbaum gebildet werden muss.
- Idee: Ersetze den zu löschenden Knoten durch seinen Nachfolger.
  Ausnahme: Hat der Knoten nur ein Kind, kann er durch das Kind ersetzt werden.
- Problem: Der Nachfolger kann auch ein innerer Knoten sein.

Beispiel:



Binary Search Tree

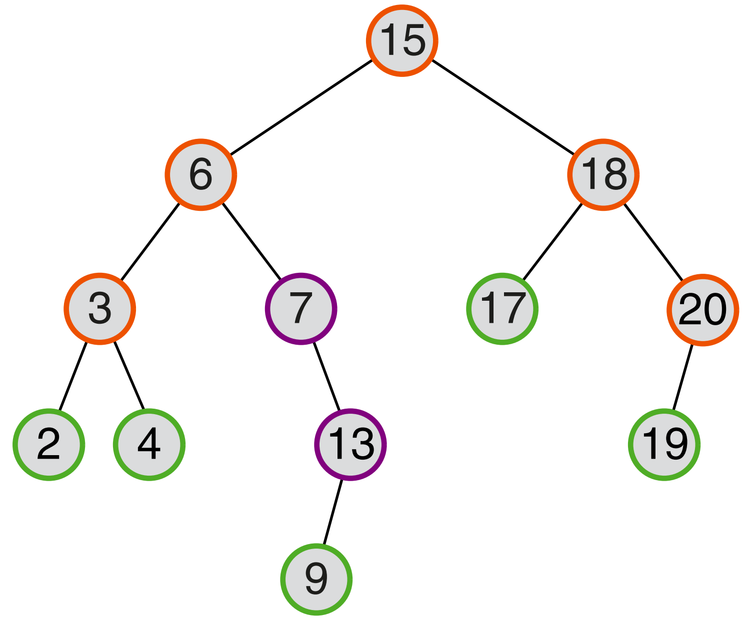
Manipulation: Löschen

Vorüberlegungen:

- Äußere Knoten
  Können gefahrlos entfernt werden.
- Innere Knoten
  Das Löschen eines inneren Knotens hinterlässt einen Wald, aus dem ein gültiger Suchbaum gebildet werden muss.
- Idee: Ersetze den zu löschenden Knoten durch seinen Nachfolger.
  Ausnahme: Hat der Knoten nur ein Kind, kann er durch das Kind ersetzt werden.
- Problem: Der Nachfolger kann auch ein innerer Knoten sein.
- Erkenntis: Der Nachfolger hat niemals ein linkes Kind.

➜ Der Nachfolger wird durch sein rechtes Kind ersetzt.

Beispiel:



Binary Search Tree

Manipulation: Löschen [Red-Black Tree]

Algorithmus: Transplant.

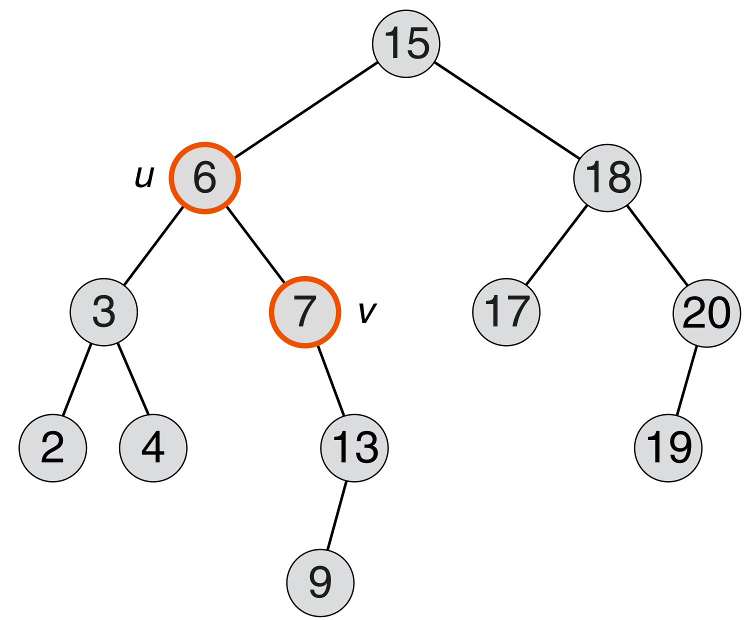
Eingabe: T. Binary Search Tree.
u, v. Wurzeln von Teilbäumen von T; ggf. $v =$ *NIL*.

Ausgabe: Binärbaum, bei dem u durch v ersetzt wurde.

Transplant(T, u, v)

IF u.*parent* $==$ *NIL* **THEN**
  T.*root* $= v$
ELSE IF $u == u$.*parent*.*left* **THEN**
  u.*parent*.*left* $= v$
ELSE
  u.*parent*.*right* $= v$
ENDIF
IF $v \neq$ *NIL* **THEN**
  v.*parent* $= u$.*parent*
ENDIF

Beispiel: *Transplant*(T, u, v)



Binary Search Tree

Manipulation: Löschen [Red-Black Tree]

Algorithmus: Transplant.

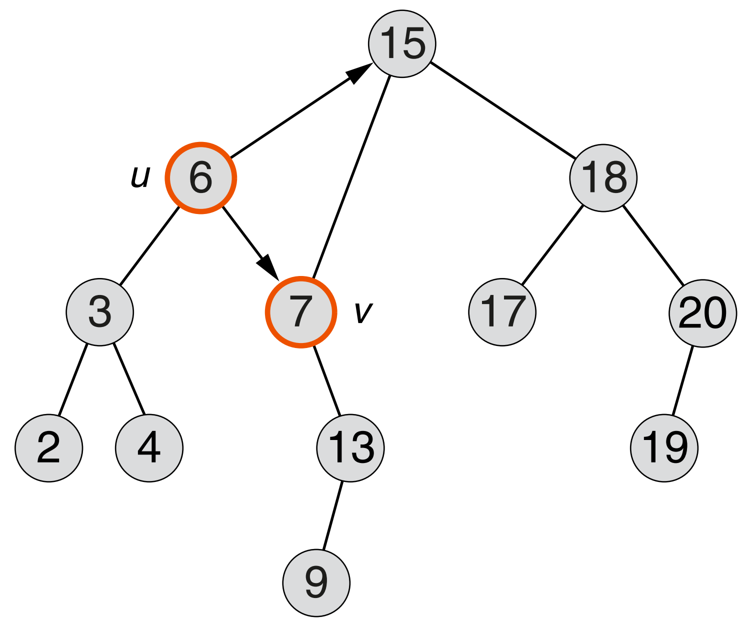
Eingabe: T. Binary Search Tree.
u, v. Wurzeln von Teilbäumen von T; ggf. $v =$ *NIL*.

Ausgabe: Binärbaum, bei dem u durch v ersetzt wurde.

Transplant(T, u, v)

IF u.*parent* $==$ *NIL* **THEN**
  T.*root* $= v$
ELSE IF $u == u$.*parent*.*left* **THEN**
  u.*parent*.*left* $= v$
ELSE
  u.*parent*.*right* $= v$
ENDIF
IF $v \neq$ *NIL* **THEN**
  v.*parent* $= u$.*parent*
ENDIF

Beispiel: *Transplant*(T, u, v)



Binary Search Tree

Manipulation: Löschen [Red-Black Tree]

Algorithmus: Transplant.

Eingabe: T. Binary Search Tree.
u, v. Wurzeln von Teilbäumen von T; ggf. $v =$ *NIL*.

Ausgabe: Binärbaum, bei dem u durch v ersetzt wurde.

Transplant(T, u, v)

```
IF u.parent == NIL THEN
  T.root = v
ELSE IF u == u.parent.left THEN
  u.parent.left = v
ELSE
  u.parent.right = v
ENDIF
IF v ≠ NIL THEN
  v.parent = u.parent
ENDIF
```

Laufzeit:

- Alle Anweisungen sind in $O(1)$.

➜ $O(1)$ Gesamtlaufzeit.

Binary Search Tree

Manipulation: Löschen

Algorithmus: Tree Delete.

Eingabe: T. Binary Search Tree.
z. Zu löschender Knoten.

Ausgabe: Binary Search Tree, bei dem z gelöscht wurde.

Fallunterscheidung:

1. z hat keine Kinder.
   Ersetze z bei seinem Elter durch NIL.

2. z hat ein Kind.
   Ersetze z bei seinem Elter durch (a) sein rechtes bzw. (b) sein linkes Kind.

3. z hat zwei Kinder.
   (a) zs Nachfolger y ist sein rechtes Kind.
   Ersetze z durch y.

   (b) zs Nachfolger y ist ein anderer Knoten seinem rechten Teilbaum.
   Ersetze y durch sein rechtes Kind und mache y zur Wurzel von zs rechtem Teilbaum.
   Ersetze z durch y.

Binary Search Tree

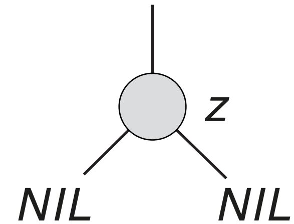
Manipulation: Löschen

TreeDelete(T, z)

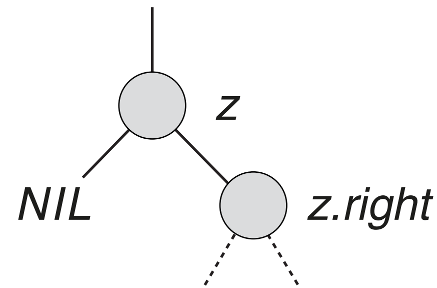
```
IF z.left == NIL THEN
    Transplant(T, z, z.right)
ELSE IF z.right == NIL THEN
    Transplant(T, z, z.left)
ELSE
    y = TreeMinimum(z.right)
    IF y.parent ≠ z THEN
        Transplant(T, y, y.right)
        y.right = z.right
        y.right.parent = y
    ENDIF
    Transplant(T, z, y)
    y.left = z.left
    y.left.parent = y
ENDIF
```

Fälle:

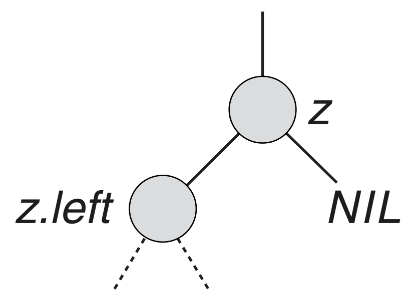
1.



2a.



2b.



Binary Search Tree

Manipulation: Löschen

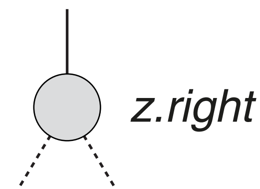
TreeDelete(T, z)

```
IF z.left == NIL THEN
    Transplant(T, z, z.right)
ELSE IF z.right == NIL THEN
    Transplant(T, z, z.left)
ELSE
    y = TreeMinimum(z.right)
    IF y.parent ≠ z THEN
        Transplant(T, y, y.right)
        y.right = z.right
        y.right.parent = y
    ENDIF
    Transplant(T, z, y)
    y.left = z.left
    y.left.parent = y
ENDIF
```

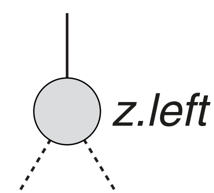
Fälle:

1.

2a.



2b.



Binary Search Tree

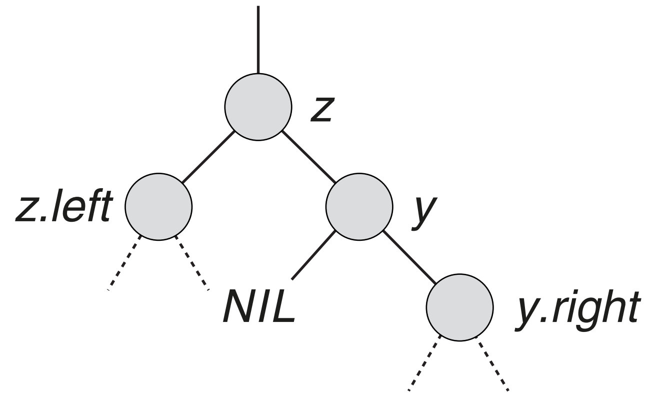
Manipulation: Löschen

TreeDelete(T, z)

```
IF z.left == NIL THEN
    Transplant(T, z, z.right)
ELSE IF z.right == NIL THEN
    Transplant(T, z, z.left)
ELSE
    y = TreeMinimum(z.right)
    IF y.parent ≠ z THEN
        Transplant(T, y, y.right)
        y.right = z.right
        y.right.parent = y
    ENDIF
    Transplant(T, z, y)
    y.left = z.left
    y.left.parent = y
ENDIF
```

Fälle:

3a.



Binary Search Tree

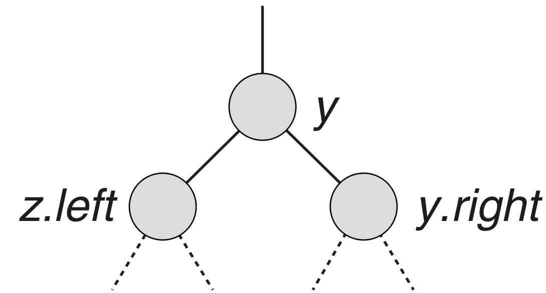
Manipulation: Löschen

TreeDelete(T, z)

```
IF z.left == NIL THEN
   Transplant(T, z, z.right)
ELSE IF z.right == NIL THEN
   Transplant(T, z, z.left)
ELSE
   y = TreeMinimum(z.right)
   IF y.parent ≠ z THEN
      Transplant(T, y, y.right)
      y.right = z.right
      y.right.parent = y
   ENDIF
   Transplant(T, z, y)
   y.left = z.left
   y.left.parent = y
ENDIF
```

Fälle:

3a.



Binary Search Tree

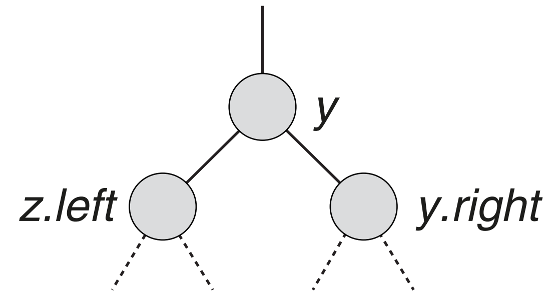
Manipulation: Löschen

TreeDelete(T, z)

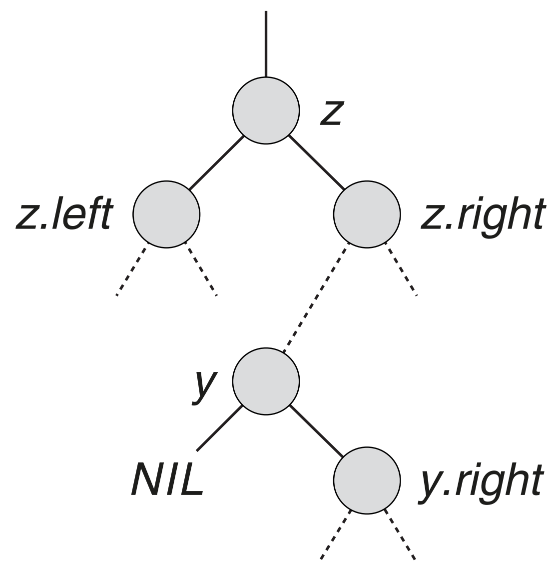
```
IF z.left == NIL THEN
    Transplant(T, z, z.right)
ELSE IF z.right == NIL THEN
    Transplant(T, z, z.left)
ELSE
    y = TreeMinimum(z.right)
    IF y.parent ≠ z THEN
        Transplant(T, y, y.right)
        y.right = z.right
        y.right.parent = y
    ENDIF
    Transplant(T, z, y)
    y.left = z.left
    y.left.parent = y
ENDIF
```

Fälle:

3a.



3b.



Binary Search Tree

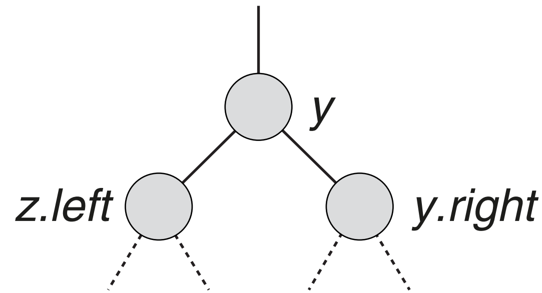
Manipulation: Löschen

TreeDelete(T, z)

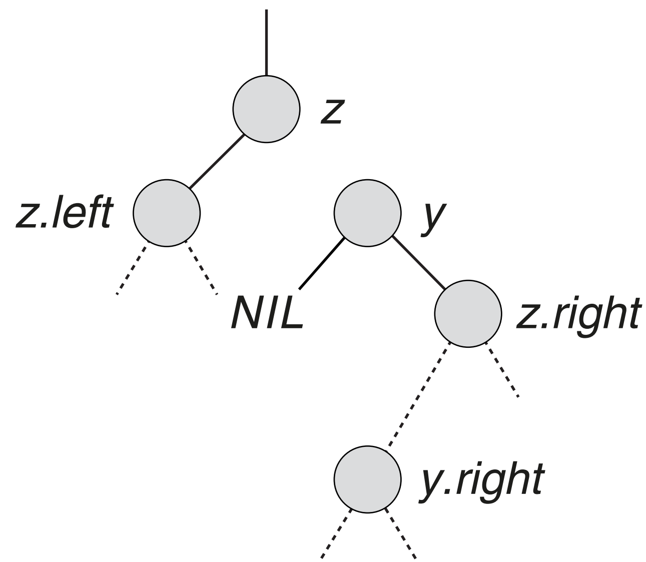
```
IF z.left == NIL THEN
   Transplant(T, z, z.right)
ELSE IF z.right == NIL THEN
   Transplant(T, z, z.left)
ELSE
   y = TreeMinimum(z.right)
   IF y.parent ≠ z THEN
      Transplant(T, y, y.right)
      y.right = z.right
      y.right.parent = y
   ENDIF
   Transplant(T, z, y)
   y.left = z.left
   y.left.parent = y
ENDIF
```

Fälle:

3a.



3b.



Binary Search Tree

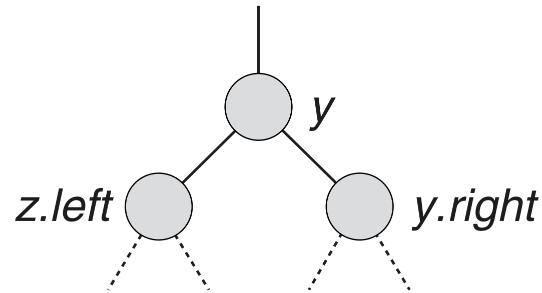
Manipulation: Löschen

TreeDelete(T, z)

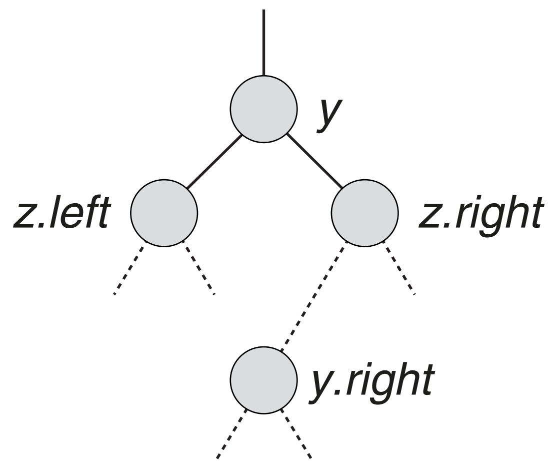
```
IF z.left == NIL THEN
   Transplant(T, z, z.right)
ELSE IF z.right == NIL THEN
   Transplant(T, z, z.left)
ELSE
   y = TreeMinimum(z.right)
   IF y.parent ≠ z THEN
      Transplant(T, y, y.right)
      y.right = z.right
      y.right.parent = y
   ENDIF
   Transplant(T, z, y)
   y.left = z.left
   y.left.parent = y
ENDIF
```

Fälle:

3a.



3b.



Binary Search Tree

Manipulation: Löschen

TreeDelete(T, z)

1. **IF** z.*left* $==$ *NIL* **THEN**
2.   *Transplant*$(T, z, z$.*right*$)$
3. **ELSE IF** z.*right* $==$ *NIL* **THEN**
4.   *Transplant*$(T, z, z$.*left*$)$
5. **ELSE**
6. $y =$ *TreeMinimum*$(z$.*right*$)$
7.   **IF** y.*parent* $\neq z$ **THEN**
8.   *Transplant*$(T, y, y$.*right*$)$
9. y.*right* $= z$.*right*
10. y.*right*.*parent* $= y$
11.   **ENDIF**
12.   *Transplant*(T, z, y)
13. y.*left* $= z$.*left*
14. y.*left*.*parent* $= y$
15. **ENDIF**

Laufzeit:

- Alle Zeilen außer Zeile 6 sind $O(1)$.
- Laufzeit von *TreeMinimum* ist $O(h)$.

→ $O(h)$, wobei h die Baumhöhe ist.

Die Laufzeit aller Manipulationsoperationen ist von der Höhe des Baums abhängig.

Best Case:

- Balancierter Baum.

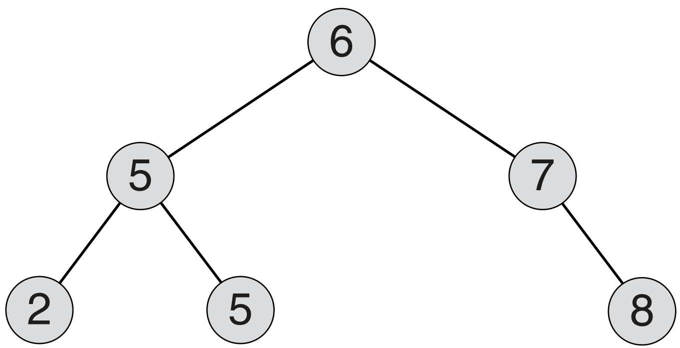
➜ $h \approx \lg n \;\to\; T(n) = O(\lg n)$.

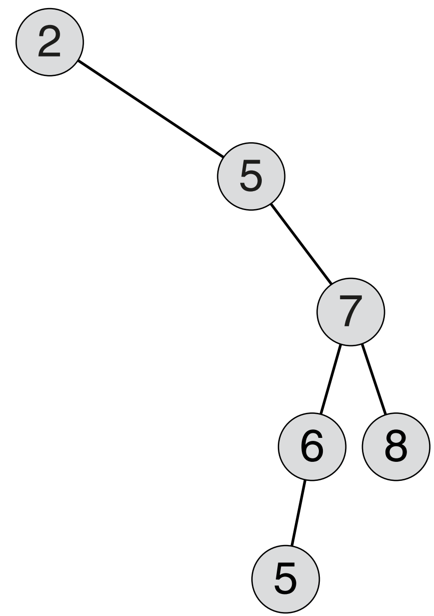
Worst Case:

- Alle inneren Knoten vom Grad 1.

➜ $h \approx n \;\to\; T(n) = O(n)$.

Beispiel:





Binary Search Tree

Laufzeit (Fortsetzung)

Average Case:

- Zufälliger Binary Search Tree mit n verschiedenen Knoten.

➜ $E[h] = \lg n \;\to\; T(n) = O(\lg n)$.

Beweisidee: Probabilistische Analyse der erwarteten Baumhöhe nach einer Sequenz zufälliger Einfügeoperationen.

Konstruktion

- Sei A eine Folge von n Schlüsseln.
- Wähle eine zufällige Permutation der Schlüssel in A.
- Füge die Schlüssel gemäß der Reihenfolge der Permutation nacheinander in einen leeren Binary Search Tree T ein.

- Weiteres Einfügen von Elementen in nicht-zufälliger Reihenfolge oder Löschen von Elementen machen den Binary Search Tree unzufällig.

AVL Tree

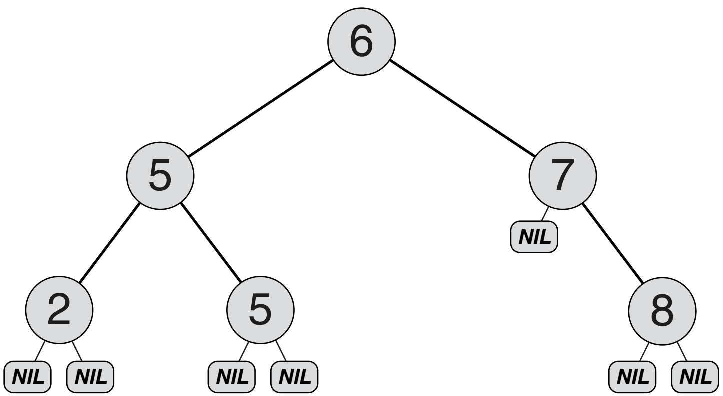
Definition

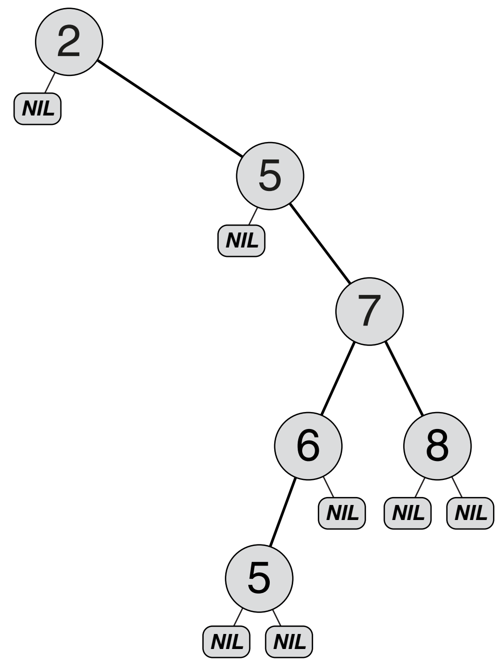
Ein Binary Search Tree ist ein AVL Tree, wenn für jeden internen Knoten x gilt

$$-1 \ \leq \ bf(x) \ \leq \ 1,$$

wobei $bf(x) = h(x.\mathit{right}) - h(x.\mathit{left})$ der Balancefaktor von x und $h(x)$ seine Höhe ist.

Beispiele:





AVL Tree

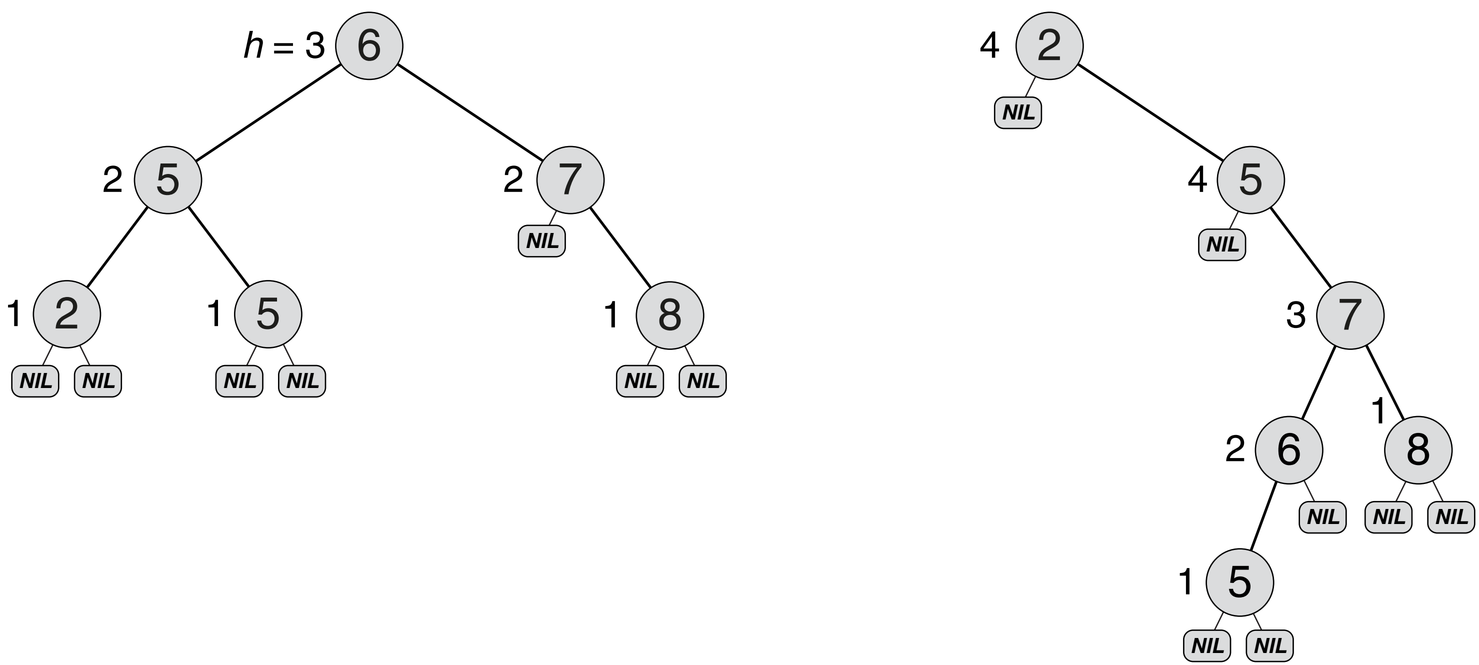
Definition

Ein Binary Search Tree ist ein AVL Tree, wenn für jeden internen Knoten x gilt

$$-1 \;\leq\; bf(x) \;\leq\; 1,$$

wobei $bf(x) = h(x.\textit{right}) - h(x.\textit{left})$ der Balancefaktor von x und $h(x)$ seine Höhe ist.

Beispiele:



AVL Tree

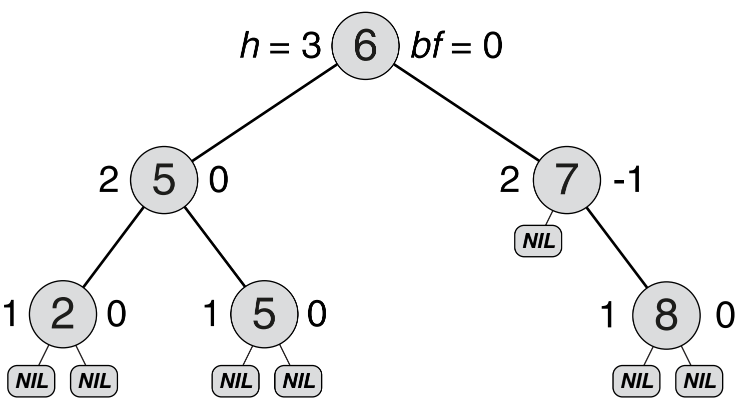
Definition

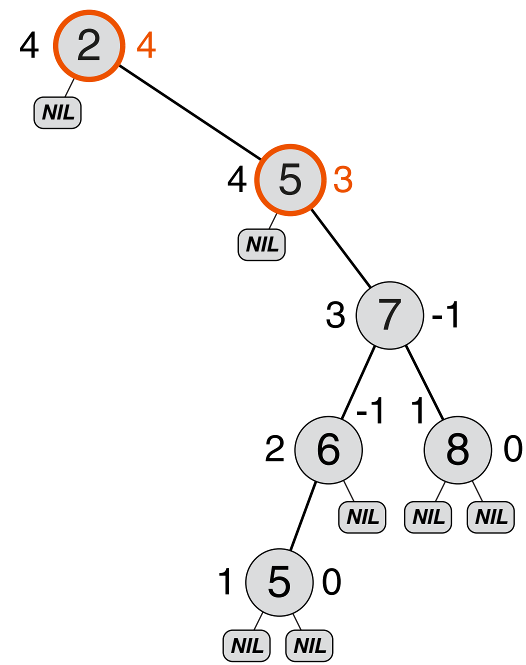
Ein Binary Search Tree ist ein AVL Tree, wenn für jeden internen Knoten x gilt

$$-1 \;\leq\; bf(x) \;\leq\; 1,$$

wobei $bf(x) = h(x.\textit{right}) - h(x.\textit{left})$ der Balancefaktor von x und $h(x)$ seine Höhe ist.

Beispiele:





AVL Tree

Satz 1

Ein AVL Tree mit n Knoten hat eine Höhe $h \leq 2 \lg n = O(\lg n)$.

AVL Tree

Satz 1

Ein AVL Tree mit n Knoten hat eine Höhe $h \;\leq\; 2\lg n \;=\; O(\lg n)$.

Beweis:

Behauptung: Der AVL Tree mit Wurzel x hat $\geq 2^{h(x)/2}$ Knoten.

AVL Tree

Satz 1

Ein AVL Tree mit n Knoten hat eine Höhe $h \le 2\lg n = O(\lg n)$.

Beweis:

Behauptung: Der AVL Tree mit Wurzel x hat $\geq 2^{h(x)/2}$ Knoten.

Für die Wurzel x eines AVL Trees mit n Knoten gilt:

$$\begin{array}{rrcll} & n & \geq & 2^{h(x)/2} & \text{(Behauptung)} \\ \Leftrightarrow & \lg n & \geq & h(x)/2 & \\ \Leftrightarrow & 2\lg n & \geq & h(x) & \square \end{array}$$

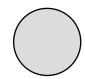
AVL Tree

Höhe eines AVL Tree

Behauptung: Der AVL Tree mit Wurzel x hat $\geq 2^{h(x)/2}$ Knoten.

Idee: Abschätzung der minimalen Zahl an Knoten eines AVL Trees der Höhe h.

Beispiel:



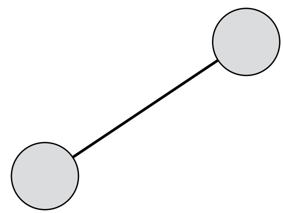
AVL Tree

Höhe eines AVL Tree

Behauptung: Der AVL Tree mit Wurzel x hat $\geq 2^{h(x)/2}$ Knoten.

Idee: Abschätzung der minimalen Zahl an Knoten eines AVL Trees der Höhe h.

Beispiel:



- Ein minimaler AVL Tree hat $bf(x) \neq 0$ für alle Knoten x.

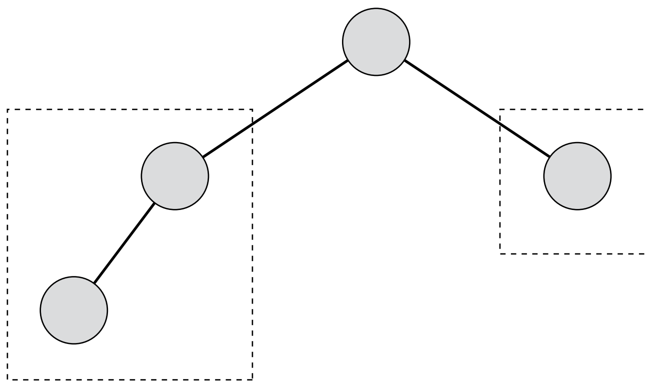
AVL Tree

Höhe eines AVL Tree

Behauptung: Der AVL Tree mit Wurzel x hat $\geq 2^{h(x)/2}$ Knoten.

Idee: Abschätzung der minimalen Zahl an Knoten eines AVL Trees der Höhe h.

Beispiel:



- Ein minimaler AVL Tree hat $bf(x) \neq 0$ für alle Knoten x.
- Jedes Kind der Wurzel eines minimalen AVL Trees ist ein minimaler AVL Tree.

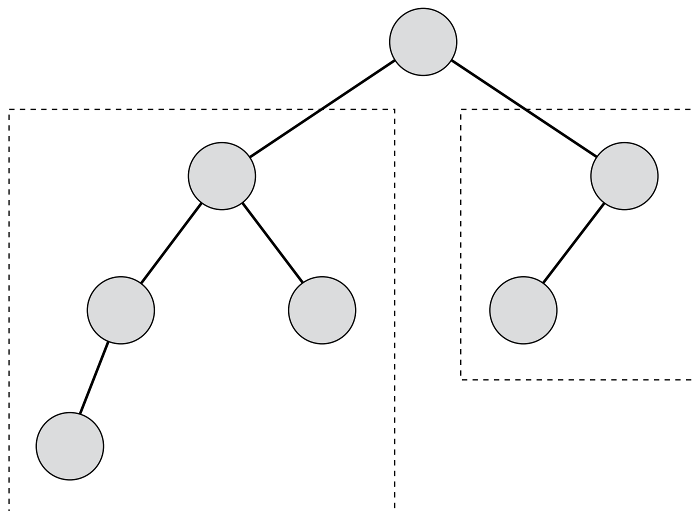
AVL Tree

Höhe eines AVL Tree

Behauptung: Der AVL Tree mit Wurzel x hat $\geq 2^{h(x)/2}$ Knoten.

Idee: Abschätzung der minimalen Zahl an Knoten eines AVL Trees der Höhe h.

Beispiel:



- Ein minimaler AVL Tree hat $bf(x) \neq 0$ für alle Knoten x.
- Jedes Kind der Wurzel eines minimalen AVL Trees ist ein minimaler AVL Tree.

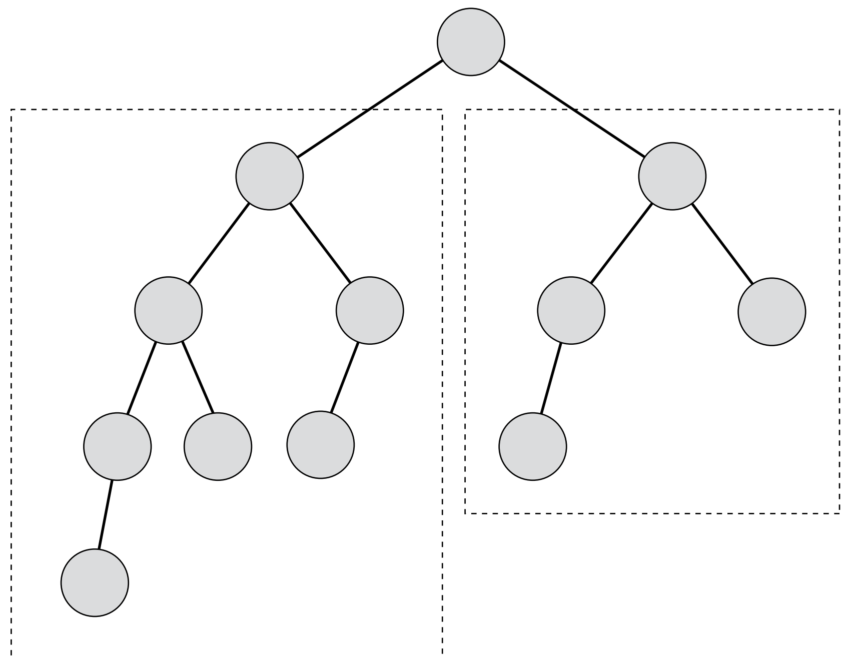
AVL Tree

Höhe eines AVL Tree

Behauptung: Der AVL Tree mit Wurzel x hat $\geq 2^{h(x)/2}$ Knoten.

Idee: Abschätzung der minimalen Zahl an Knoten eines AVL Trees der Höhe h.

Beispiel:



- Ein minimaler AVL Tree hat $bf(x) \neq 0$ für alle Knoten x.
- Jedes Kind der Wurzel eines minimalen AVL Trees ist ein minimaler AVL Tree.

AVL Tree

Höhe eines AVL Tree

Behauptung: Der AVL Tree mit Wurzel x hat $\geq 2^{h(x)/2}$ Knoten.

Idee: Abschätzung der minimalen Zahl an Knoten eines AVL Trees der Höhe h.

Sei $n(h)$ die Zahl der Knoten eines minimalen AVL Trees der Höhe h. Dann gilt:

$$
\begin{array}{rrcll}
 & n(h) & = & 1 + n(h-1) + n(h-2) & \\
\Rightarrow & n(h) & > & 2n(h-2) & \\
\Rightarrow & n(h) & > & 2^i n(h-2i) & \text{(für } i > 0\text{)} \\
\Rightarrow & n(h) & \geq & 2^{h/2} n(0) & \text{(so dass } h-2i = 0\text{)} \\
\Rightarrow & n(h) & \geq & 2^{h/2} \quad \square &
\end{array}
$$

Bemerkungen:

- Ein minimaler AVL Tree der Höhe h hat $F_{h+3} - 1$ Knoten, wobei F_i die i-te Fibonacci-Zahl ist. Nach Formel von Moivre-Binet ist

$$F_i = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^i - \left(\frac{1-\sqrt{5}}{2} \right)^i \right) \geq \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^i - 1 \right),$$

  so dass

$$n(h) \geq \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - 1 \right) - 1.$$

  Nach h aufgelöst folgt

$$h \leq 1.4405 \lg(n+2) - 0.3277.$$

AVL Tree
Manipulation

Algorithmen, die von Binary Search Trees geerbt werden:

- Knoten in Sortierreihenfolge besuchen
  Traversierung des Baumes mit DFS-Traverse (in-order).
- Knoten suchen (*Search*)
  Einen Knoten mit vorgegebenem Schlüssel suchen.
- Minimum, Maximum, oder Nachfolger (*Successor*) bestimmen
  Den Knoten mit kleinstem, größten oder nächstgrößten Sortierschlüssel bestimmten.
- Knoten verändern
  Den Schlüssel eines bestimmten Knotens ändern.

Laufzeit

- Die Algorithmen haben Laufzeit $O(h)$, wobei h die Höhe des Binary Search Tree ist.

→ Auf AVL Trees benötigen sie daher $O(\lg n)$ Zeit.

AVL Tree

Manipulation

Algorithmen, die auf AVL Trees zugeschnitten sind:

- Knoten einfügen (*Insert*)
  Einen Knoten an der richtigen Stelle im Baum einfügen.
- Knoten löschen (*Delete*)
  Einen bestimmten Knoten aus dem Baum löschen.

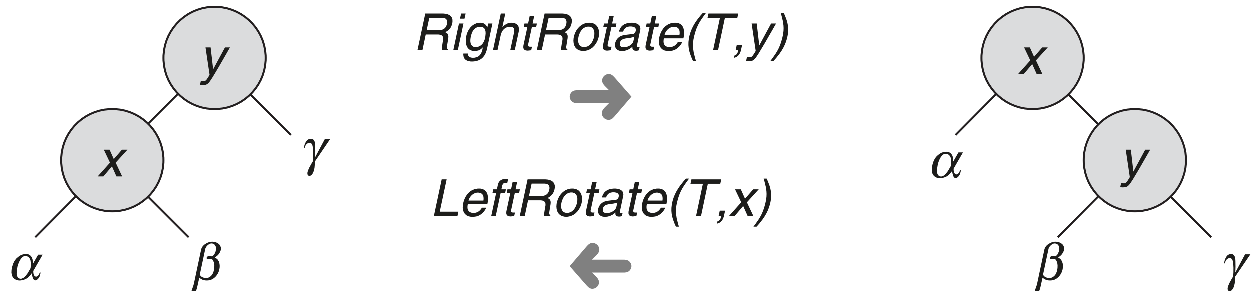
Jedes Einfügen oder Löschen eines Knotens erfordert gegebenenfalls die Wiederherstellung der AVL Tree-Bedingung:

- Einfügen / Löschen eines Knotens kann die Höhe seiner Vorgänger verändern.
- Das kann den Balancefaktor verändern und die AVL-Bedingung verletzen.

→ Der AVL Tree muss gegebenenfalls rekonfiguriert werden.

AVL Tree

Manipulation: Rotation



- Grundlegende Rekonfigurationsoperation für Binary Search Trees.
- Hält die Binary Search Tree-Bedingung aufrecht.
- Ändert die Reihenfolge einer In-Order-Traversierung nicht.
- Rechts- und Linksrotation sind inverse Operationen zueinander.

AVL Tree

Manipulation: Rotation

Algorithmus: Left Rotate.

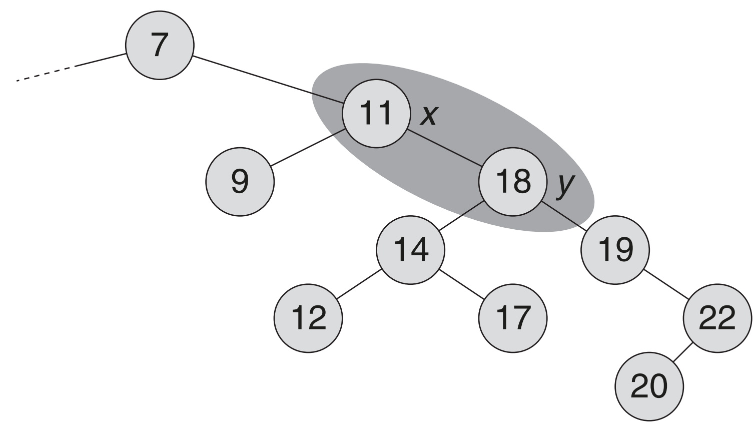
Eingabe: T. Binary Search Tree.
x. Wurzel eines Teilbaums von T, wobei x.*right* $\neq$ *NIL* bzw. T.*nil*

Ausgabe: Binary Search Tree, dessen Teilbaum x linkes Kind von x.*right* ist.

LeftRotate(T, x)

```
y = x.right
x.right = y.left
IF y.left ≠ T.nil THEN
   y.left.parent = x
ENDIF
y.parent = x.parent
IF x.parent == T.nil THEN
   T.root = y
ELSE IF x == x.parent.left THEN
   x.parent.left = y
ELSE
   x.parent.right = y
ENDIF
y.left = x
x.parent = y
```

Beispiel:



Laufzeit:

- $T(n) = O(1)$

AVL Tree

Manipulation: Rotation

Algorithmus: Left Rotate.

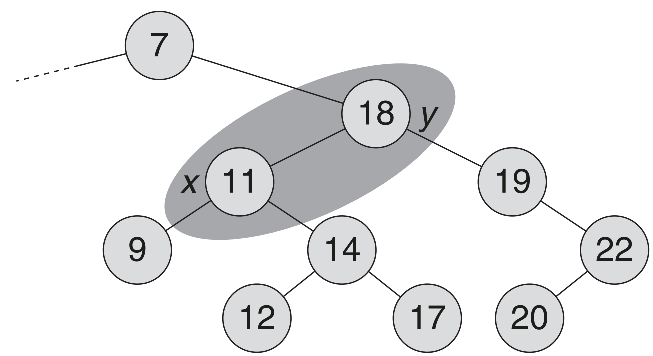
Eingabe: T. Binary Search Tree.
x. Wurzel eines Teilbaums von T, wobei x.*right* $\neq$ *NIL* bzw. T.*nil*

Ausgabe: Binary Search Tree, dessen Teilbaum x linkes Kind von x.*right* ist.

LeftRotate(T, x)

```
y = x.right
x.right = y.left
IF y.left ≠ T.nil THEN
   y.left.parent = x
ENDIF
y.parent = x.parent
IF x.parent == T.nil THEN
   T.root = y
ELSE IF x == x.parent.left THEN
   x.parent.left = y
ELSE
   x.parent.right = y
ENDIF
y.left = x
x.parent = y
```

Beispiel:



Laufzeit:

- $T(n) = O(1)$

AVL Tree

Manipulation: Einfügen

Algorithmus: AVL Insert.

Eingabe: T. AVL Tree.
z. Einzufügender Knoten mit Schlüssel k.

Ausgabe: Um z erweiterter AVL Tree.

AVL Tree

Manipulation: Einfügen

Algorithmus: AVL Insert.

Eingabe: T. AVL Tree.
z. Einzufügender Knoten mit Schlüssel k.

Ausgabe: Um z erweiterter AVL Tree.

AVLInsert(T, z)

1. *TreeInsert*(T, z)
2. *AVLInsertFixup*(T, z)

AVL Tree

Manipulation: Einfügen

Algorithmus: AVL Insert.

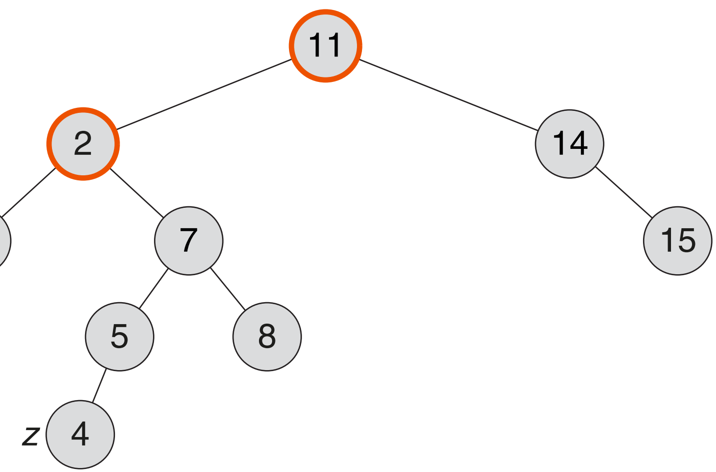
Eingabe: T. AVL Tree.
z. Einzufügender Knoten mit Schlüssel k.

Ausgabe: Um z erweiterter AVL Tree.

AVLInsert(T, z)

1. *TreeInsert*(T, z)
2. *AVLInsertFixup*(T, z)

Beispiel:



AVL Tree

Manipulation: Einfügen

Algorithmus: AVL Insert Fixup.

Eingabe: T. Potentiell invalider AVL Tree.
z. Knoten, der die Invalidität verursacht.

Ausgabe: Valider AVL Tree.

AVL Tree

Manipulation: Einfügen

Algorithmus: AVL Insert Fixup.

Eingabe: T. Potentiell invalider AVL Tree.
z. Knoten, der die Invalidität verursacht.

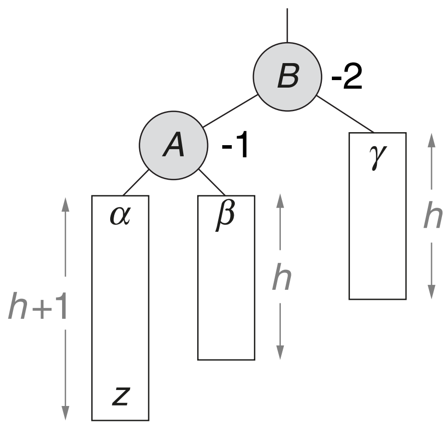
Ausgabe: Valider AVL Tree.

AVLInsertFixup(T, z):

1. Aktualisiere den Balancefaktor $bf(z)$ von z.
2. Fallunterscheidung:
   (a) $bf(z) = 0$: Terminiere.
   (b) $bf(z) = \pm 1$: Fahre mit $z = z.$*parent* bei Schritt 1 fort.
   (c) $bf(z) = \pm 2$: Fahre bei Schritt 3 fort.
3. Fallunterscheidung:
   (a) $bf(z) = -2$ und $bf(z.\textit{left}) = -1$: *RightRotate*(z)
   (b) $bf(z) = -2$ und $bf(z.\textit{left}) = +1$: *LeftRotate*$(z.\textit{left})$, dann *RightRotate*(z)
   (c) $bf(z) = +2$ und $bf(z.\textit{right}) = -1$: *RightRotate*$(z.\textit{right})$, dann *LeftRotate*$(z)$
   (d) $bf(z) = +2$ und $bf(z.\textit{right}) = +1$: *LeftRotate*$(z)$

AVL Tree
Manipulation: Einfügen

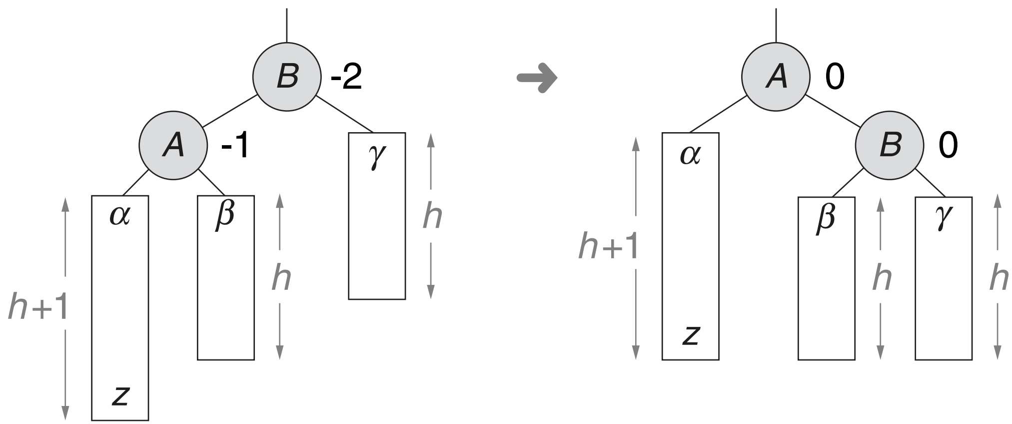
Beispiel: Fall 3a



AVL Tree

Manipulation: Einfügen

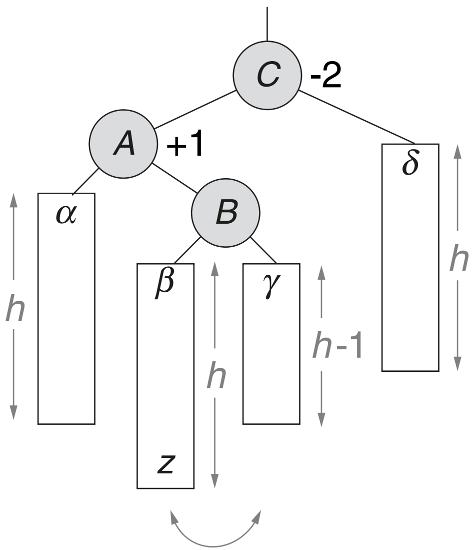
Beispiel: Fall 3a



AVL Tree

Manipulation: Einfügen

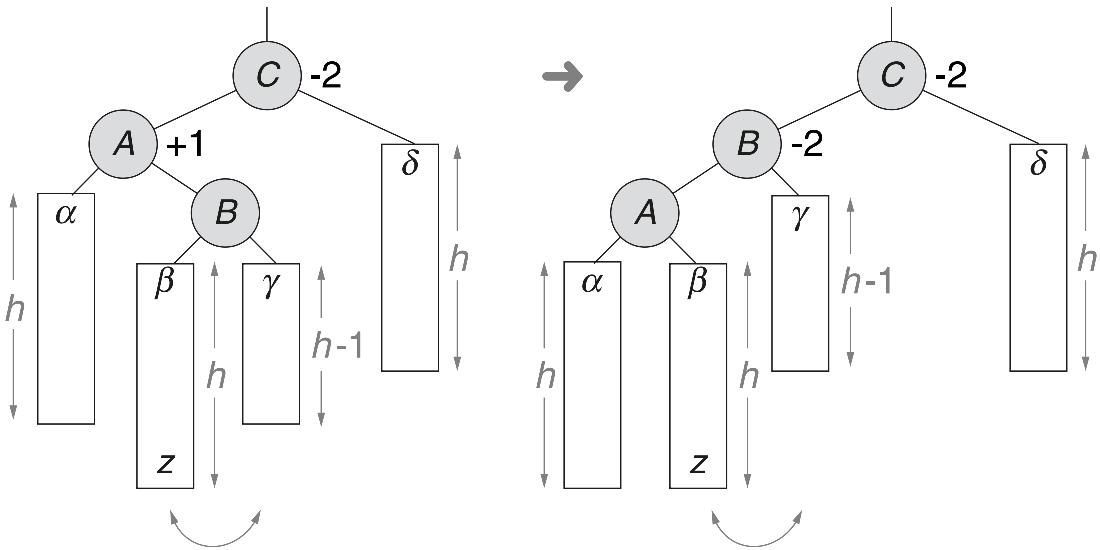
Beispiel: Fall 3b



AVL Tree

Manipulation: Einfügen

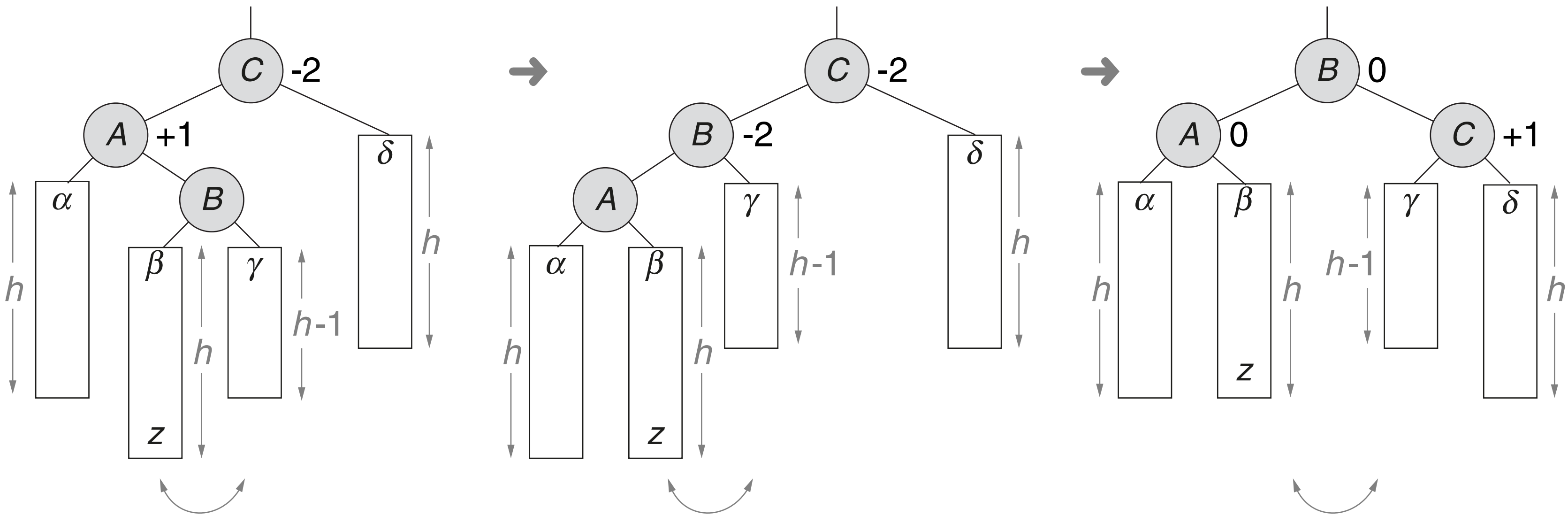
Beispiel: Fall 3b



AVL Tree

Manipulation: Einfügen

Beispiel: Fall 3b



AVL Tree

Manipulation: Löschen

Algorithmus: AVL Delete.

Eingabe: T. AVL Tree.
z. Zu löschender Knoten.

Ausgabe: AVL Tree, beim z gelöscht wurde.

AVL Tree

Manipulation: Löschen

Algorithmus: AVL Delete.

Eingabe: T. AVL Tree.
z. Zu löschender Knoten.

Ausgabe: AVL Tree, beim z gelöscht wurde.

AVLDelete(T, z)

1. $z =$ *TreeDelete*(T, z)
2. *AVLDeleteFixup*(T, z)

AVL Tree

Manipulation: Löschen

Algorithmus: AVL Delete.

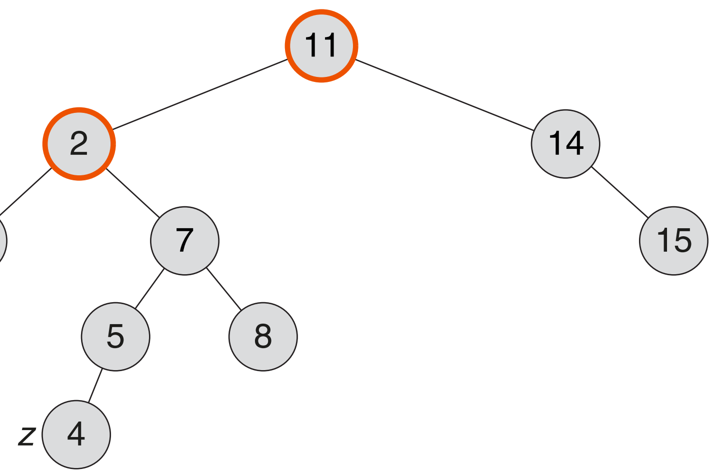
Eingabe: T. AVL Tree.
z. Zu löschender Knoten.

Ausgabe: AVL Tree, beim z gelöscht wurde.

AVLDelete(T, z)

1. $z =$ *TreeDelete*(T, z)
2. *AVLDeleteFixup*(T, z)

Beispiel:



- Modifiziere *TreeInsert*, so dass der Vaterknoten des Knotens, durch den z ersetzt wurde zurückgegeben wird.
- Korrigiere eventuelle Verletzungen der AVL Tree-Bedingung ausgehend von z.

AVL Tree

Manipulation: Einfügen

Algorithmus: AVL Delete Fixup.

Eingabe: T. Potentiell invalider AVL Tree.
z. Knoten, der die Invalidität verursacht.

Ausgabe: Valider AVL Tree.

Algorithmus: AVL Delete Fixup.

Eingabe: T. Potentiell invalider AVL Tree.
z. Knoten, der die Invalidität verursacht.

Ausgabe: Valider AVL Tree.

AVLDeleteFixup(T, z):

1. Aktualisiere den Balancefaktor $bf(z)$ von z.
2. Fallunterscheidung:
   (a) $bf(z) = 0$: Fahre mit $z = z$.*parent* bei Schritt 1 fort.
   (b) $bf(z) = \pm 1$: Terminiere.
   (c) $bf(z) = \pm 2$: Fahre bei Schritt 3 fort.
3. Fallunterscheidung:
   (a) $bf(z) = +2$ und $bf(z.\mathit{right}) \in \{0, +1\}$: *LeftRotate*$(z)$
   (b) $bf(z) = +2$ und $bf(z.\mathit{right}) = -1$: *RightRotate*$(z.\mathit{right})$, dann *LeftRotate*$(z)$
   (c) $bf(z) = -2$ und $bf(z.\mathit{left}) = +1$: *LeftRotate*$(z.\mathit{left})$, dann *RightRotate*(z)
   (d) $bf(z) = -2$ und $bf(z.\mathit{left}) \in \{-1, 0\}$: *RightRotate*$(z)$
4. Falls $bf(z) = 0$, fahre mit $z = z$.*parent* bei Schritt 1 fort.

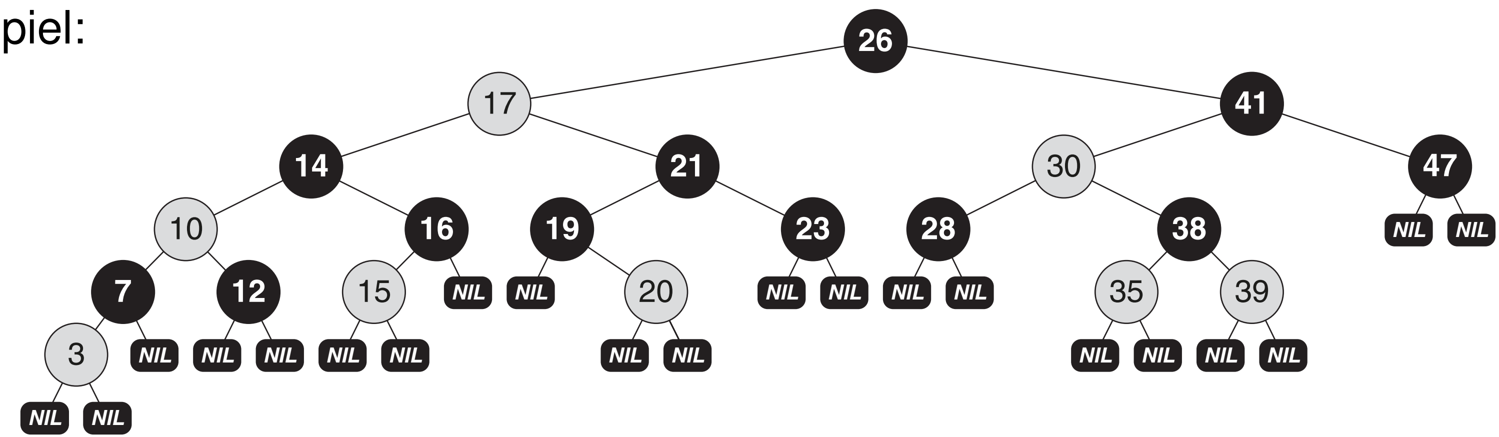
Red-Black Tree

Definition

Ein Red-Black Tree T (*Rot-Schwarz-Baum*) ist ein Binary Search Tree, dessen Knoten „gefärbt“ sind, so dass folgende Bedingungen erfüllt werden:

1. Ein Knoten ist rot oder schwarz; die Wurzel und alle Blätter sind schwarz.

Beispiel:



Red-Black Tree

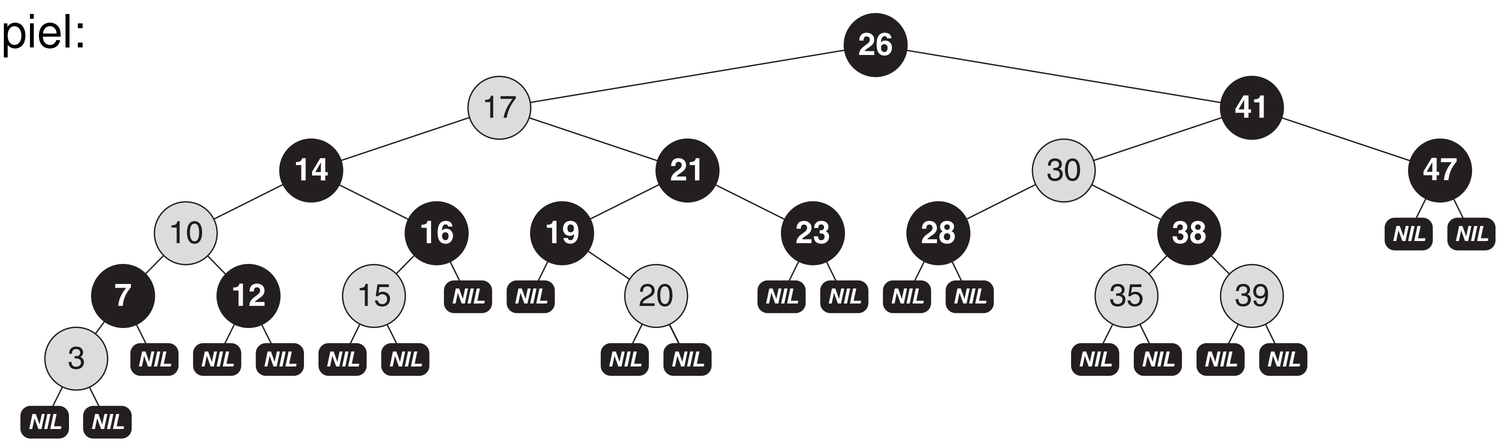
Definition

Ein Red-Black Tree T (*Rot-Schwarz-Baum*) ist ein Binary Search Tree, dessen Knoten „gefärbt“ sind, so dass folgende Bedingungen erfüllt werden:

1. Ein Knoten ist rot oder schwarz; die Wurzel und alle Blätter sind schwarz.

2. Wenn ein Knoten rot ist, sind seine Kinder schwarz.

Beispiel:



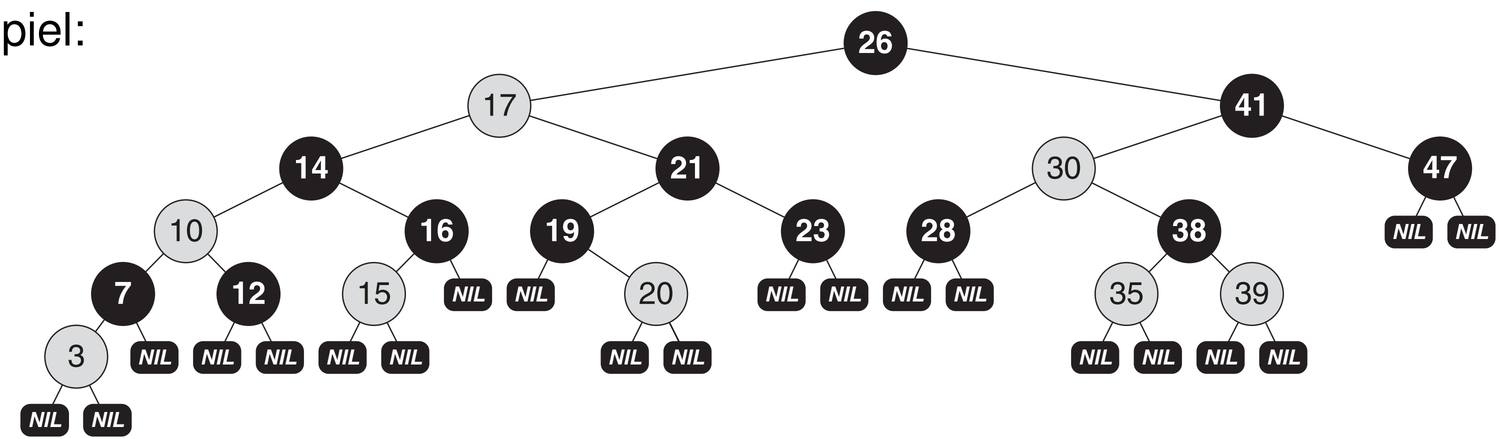
Red-Black Tree

Definition

Ein Red-Black Tree T (*Rot-Schwarz-Baum*) ist ein Binary Search Tree, dessen Knoten „gefärbt“ sind, so dass folgende Bedingungen erfüllt werden:

1. Ein Knoten ist rot oder schwarz; die Wurzel und alle Blätter sind schwarz.
2. Wenn ein Knoten rot ist, sind seine Kinder schwarz.
3. Für jeden Knoten gilt, dass all seine Pfade zu Blättern gleich viele schwarze Knoten enthalten.

Beispiel:



Red-Black Tree

Definition

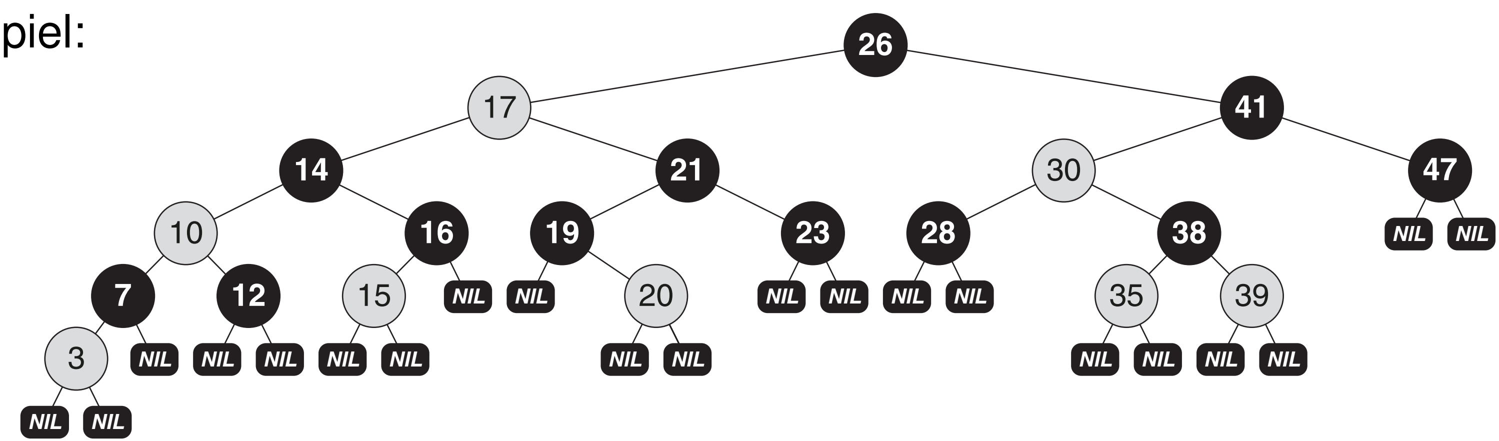
Ein Red-Black Tree T (*Rot-Schwarz-Baum*) ist ein Binary Search Tree, dessen Knoten „gefärbt“ sind, so dass folgende Bedingungen erfüllt werden:

1. Ein Knoten ist rot oder schwarz; die Wurzel und alle Blätter sind schwarz.

2. Wenn ein Knoten rot ist, sind seine Kinder schwarz.

3. Für jeden Knoten gilt, dass all seine Pfade zu Blättern gleich viele schwarze Knoten enthalten.

Beispiel:

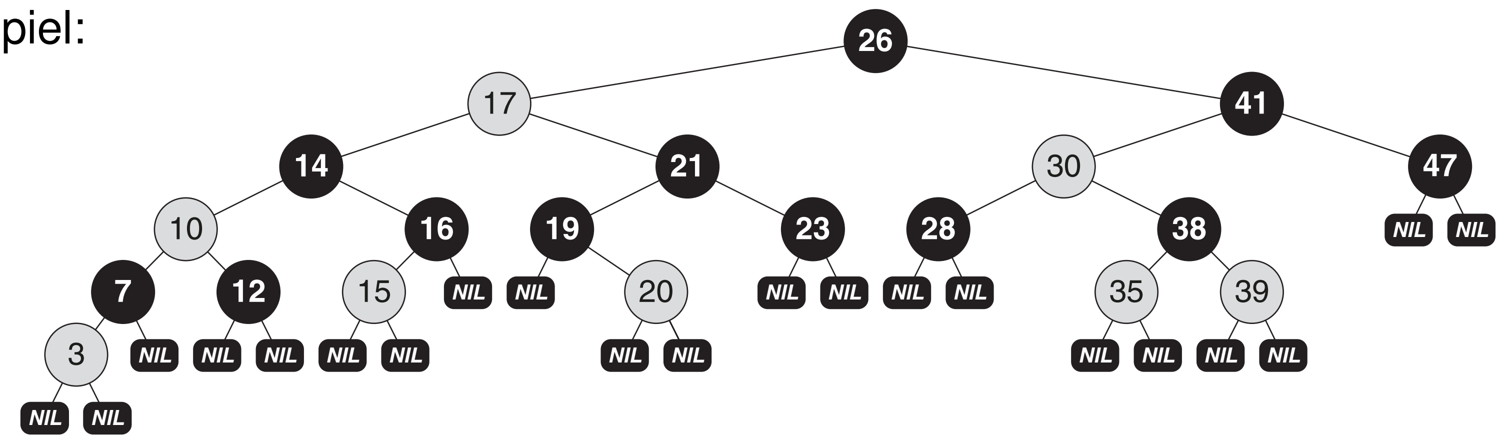


Red-Black Tree

Satz 2

Ein Red-Black Tree mit n internen Knoten hat eine Höhe $h \leq 2\lg(n+1) = O(\lg n)$.

Beispiel:



Red-Black Tree

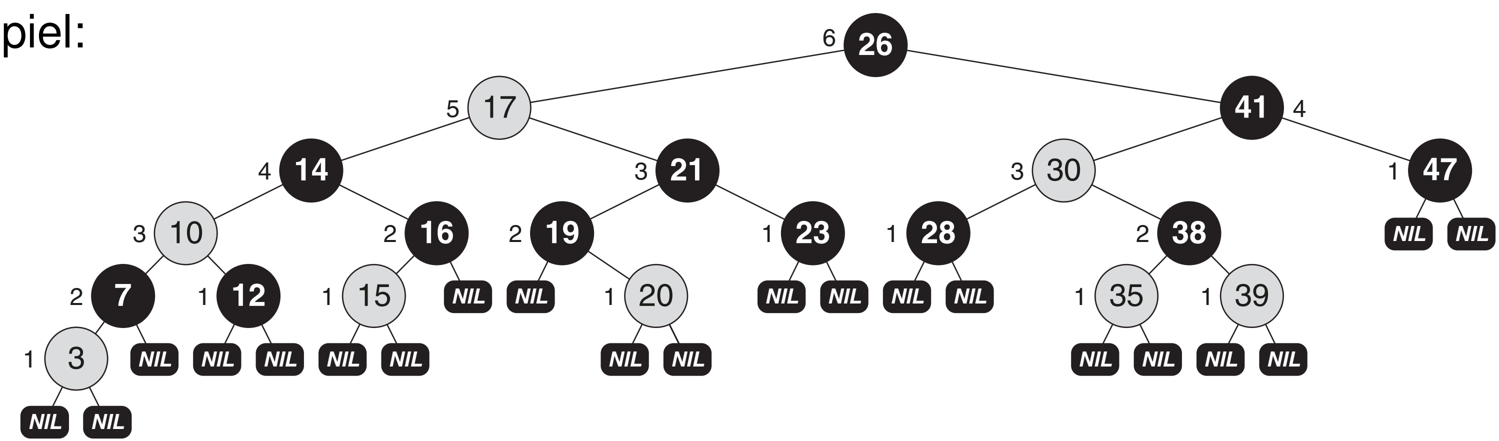
Satz 2

Ein Red-Black Tree mit n internen Knoten hat eine Höhe $h \leq 2\lg(n+1) = O(\lg n)$.

Beweis:

Sei $h(x)$ die Höhe eines Knotens x; der längste Pfad von x zu einem Blatt.

Beispiel:



Red-Black Tree

Satz 2

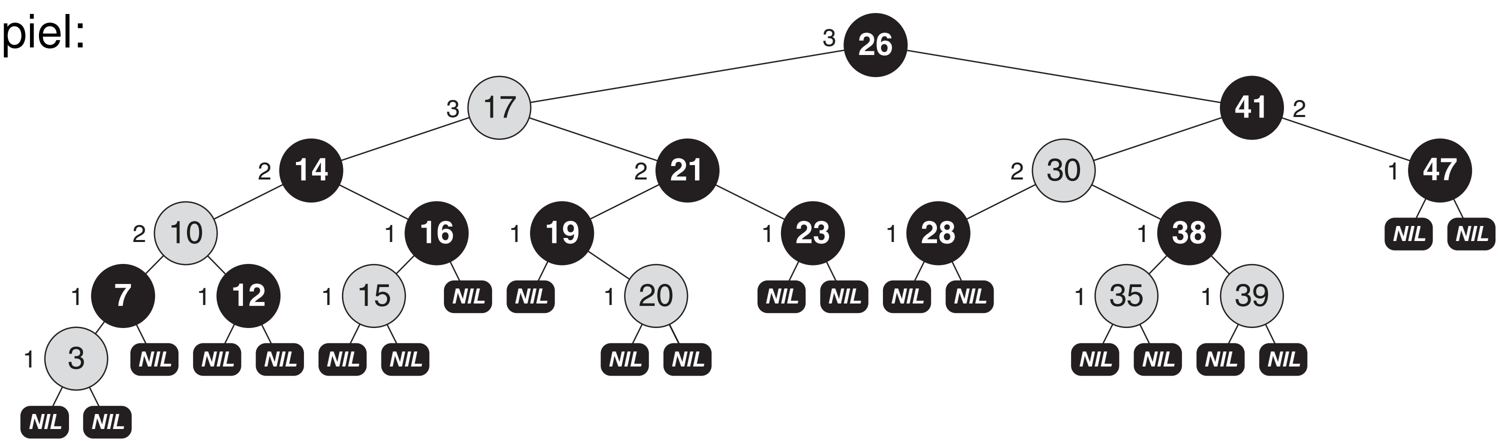
Ein Red-Black Tree mit n internen Knoten hat eine Höhe $h \leq 2\lg(n+1) = O(\lg n)$.

Beweis:

Sei $h(x)$ die Höhe eines Knotens x; der längste Pfad von x zu einem Blatt.

Sei $bh(x)$ die Schwarzhöhe (*black-height*) von x gemäß Bedingung 3: Die Zahl der schwarzen Knoten von x zu einem Blatt (exklusive x).

Beispiel:



Red-Black Tree

Satz 2

Ein Red-Black Tree mit n internen Knoten hat eine Höhe $h \leq 2\lg(n+1) = O(\lg n)$.

Beweis:

Sei $h(x)$ die Höhe eines Knotens x; der längste Pfad von x zu einem Blatt.

Sei $bh(x)$ die Schwarzhöhe (*black-height*) von x gemäß Bedingung 3: Die Zahl der schwarzen Knoten von x zu einem Blatt (exklusive x).

Behauptung 1: Jeder Knoten x mit Höhe $h(x)$ hat Schwarzhöhe $bh(x) \geq h(x)/2$.

Behauptung 2: Der Teilbaum mit Wurzel x hat $\geq 2^{bh(x)} - 1$ interne Knoten.

Red-Black Tree

Satz 2

Ein Red-Black Tree mit n internen Knoten hat eine Höhe $h \leq 2\lg(n+1) = O(\lg n)$.

Beweis:

Sei $h(x)$ die Höhe eines Knotens x; der längste Pfad von x zu einem Blatt.

Sei $bh(x)$ die Schwarzhöhe (*black-height*) von x gemäß Bedingung 3: Die Zahl der schwarzen Knoten von x zu einem Blatt (exklusive x).

Behauptung 1: Jeder Knoten x mit Höhe $h(x)$ hat Schwarzhöhe $bh(x) \geq h(x)/2$.

Behauptung 2: Der Teilbaum mit Wurzel x hat $\geq 2^{bh(x)} - 1$ interne Knoten.

Für die Wurzel x eines Red-Black Trees mit n Knoten gilt:

$$
\begin{array}{rrcll}
 & n & \geq & 2^{bh(x)} - 1 & \text{(Behauptung 2)} \\
 & & \geq & 2^{h(x)/2} - 1 & \text{(Behauptung 1)} \\
\Leftrightarrow & n+1 & \geq & 2^{h(x)/2} & \\
\Leftrightarrow & \lg(n+1) & \geq & h(x)/2 & \\
\Leftrightarrow & 2\lg(n+1) & \geq & h(x) & \square
\end{array}
$$

Red-Black Tree

Höhe eines Red-Black Tree

Behauptung 1: Jeder Knoten x mit Höhe $h(x)$ hat Schwarzhöhe $bh(x) \geq h(x)/2$.

Gemäß Bedingung 2 sind keine zwei aufeinanderfolgenden Knoten auf einem Pfad von x zu einem Blatt rot, so dass $\leq h(x)/2$ Knoten rot sein können, und damit $\geq h(x)/2$ schwarz sein müssen. $\square$

Red-Black Tree

Höhe eines Red-Black Tree

Behauptung 1: Jeder Knoten x mit Höhe $h(x)$ hat Schwarzhöhe $bh(x) \geq h(x)/2$.

Gemäß Bedingung 2 sind keine zwei aufeinanderfolgenden Knoten auf einem Pfad von x zu einem Blatt rot, so dass $\leq h(x)/2$ Knoten rot sein können, und damit $\geq h(x)/2$ schwarz sein müssen. $\square$

Behauptung 2: Der Teilbaum mit Wurzel x hat $\geq 2^{bh(x)} - 1$ interne Knoten.

Induktionsanfang:

Sei x ein Knoten mit $h(x) = 0$:

$\Rightarrow$ x ist ein Blatt

$\Rightarrow$ $bh(x) = 0$

$\Rightarrow$ $2^0 - 1 = 0$

$\leq$ Zahl interner Knoten des Teilbaums von x

Red-Black Tree

Höhe eines Red-Black Tree

Behauptung 1: Jeder Knoten x mit Höhe $h(x)$ hat Schwarzhöhe $bh(x) \geq h(x)/2$.

Gemäß Bedingung 2 sind keine zwei aufeinanderfolgenden Knoten auf einem Pfad von x zu einem Blatt rot, so dass $\leq h(x)/2$ Knoten rot sein können, und damit $\geq h(x)/2$ schwarz sein müssen. $\square$

Behauptung 2: Der Teilbaum mit Wurzel x hat $\geq 2^{bh(x)} - 1$ interne Knoten.

Induktionsanfang:

Sei x ein Knoten mit $h(x) = 0$:

$\Rightarrow$ x ist ein Blatt

$\Rightarrow$ $bh(x) = 0$

$\Rightarrow$ $2^0 - 1 = 0$

$\leq$ Zahl interner Knoten des Teilbaums von x

Induktionsschritt:

Sei x ein Knoten mit $h(x) > 0$:

$\Rightarrow$ x ist ein interner Knoten mit 2 Kindern

$\Rightarrow$ Jedes rote Kind hat Schwarzhöhe $bh(x)$, jedes schwarze Kind $bh(x) - 1$

$\Rightarrow$ Jedes Kind hat $\geq 2^{bh(x)-1} - 1$ interne Knoten (gemäß Prämisse)

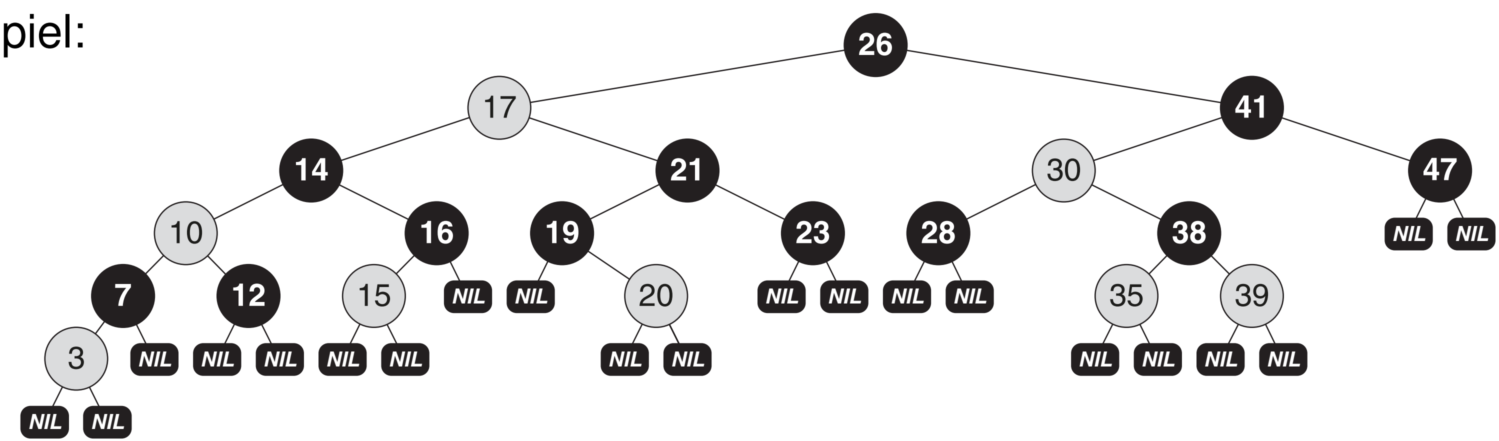
$\Rightarrow$ Zahl interner Knoten des Teilbaums von x:
$\geq 2 \cdot (2^{bh(x)-1} - 1) + 1 \;=\; 2^{bh(x)} - 1 \quad \square$

Red-Black Tree

Implementierung

- Es werden keine Schlüssel in Blättern gespeichert, alle sind *NIL*-Knoten.

Beispiel:

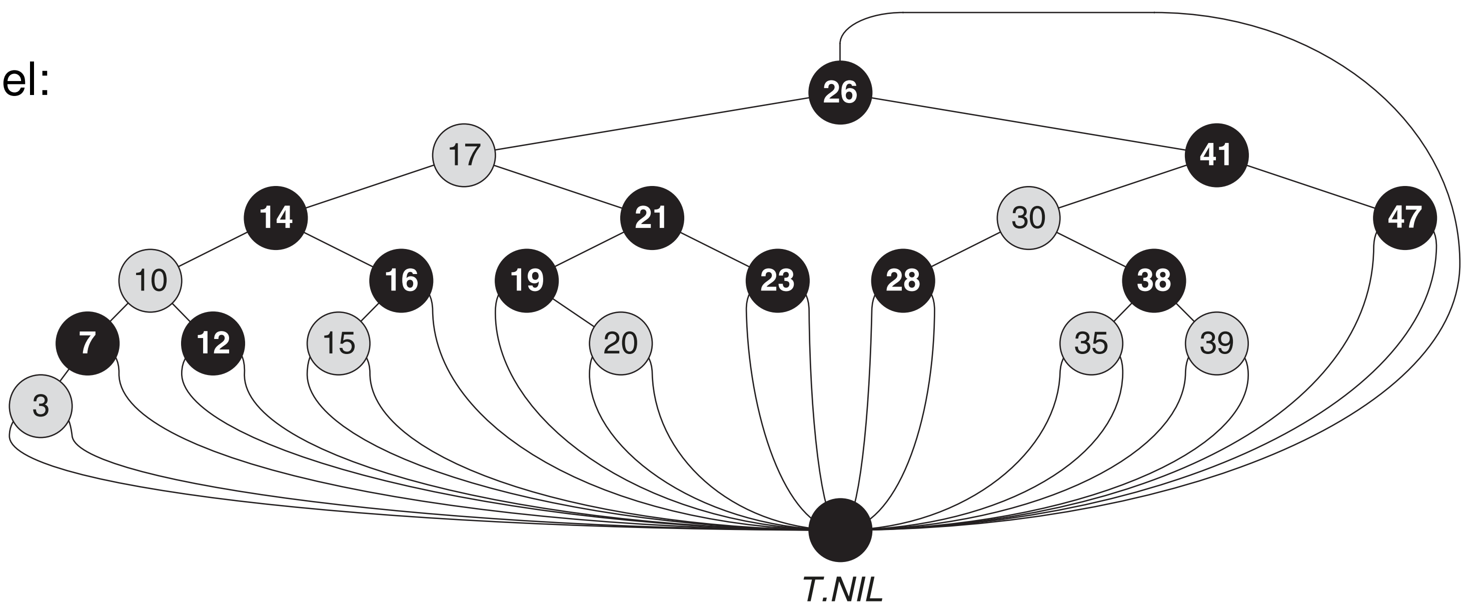


Red-Black Tree

Implementierung

- Es werden keine Schlüssel in Blättern gespeichert, alle sind *NIL*-Knoten.
- Nutzung eines Sentinel $T.nil$, um alle Blätter zu repräsentieren.
- Der Sentinel ist auch Elter der Wurzel.
- Die Farbe des Sentinel ist schwarz, alle anderen Attribute sind beliebig.

Beispiel:

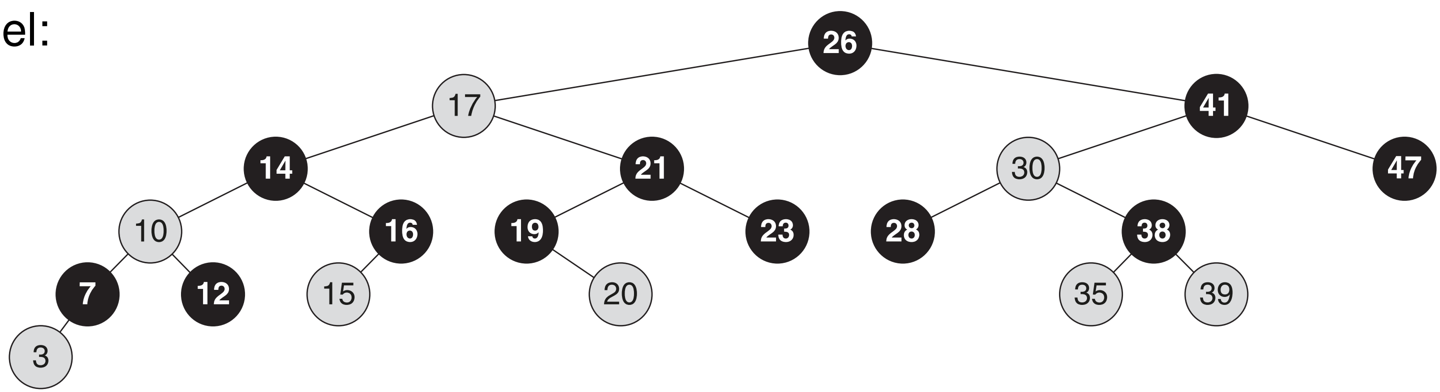


Red-Black Tree

Implementierung

- Es werden keine Schlüssel in Blättern gespeichert, alle sind *NIL*-Knoten.
- Nutzung eines Sentinel T.*nil*, um alle Blätter zu repräsentieren.
- Der Sentinel ist auch Elter der Wurzel.
- Die Farbe des Sentinel ist schwarz, alle anderen Attribute sind beliebig.
- Im Folgenden wird der Sentinel der Übersicht halber ausgeblendet.

Beispiel:



Red-Black Tree

Manipulation

Algorithmen, die von Binary Search Trees geerbt werden:

- Knoten in Sortierreihenfolge besuchen
  Traversierung des Baumes mit DFS-Traverse (in-order).

- Knoten suchen (*Search*)
  Einen Knoten mit vorgegebenem Schlüssel suchen.

- Minimum, Maximum, oder Nachfolger (*Successor*) bestimmen
  Den Knoten mit kleinstem, größten oder nächstgrößten Sortierschlüssel bestimmten.

- Knoten verändern
  Den Schlüssel eines bestimmten Knotens ändern.

Laufzeit

- Die Algorithmen haben Laufzeit $O(h)$, wobei h die Höhe des Binary Search Tree ist.

→ Auf Red-Black Trees benötigen sie daher $O(\lg n)$ Zeit.

Bemerkungen:

- Bei geerbten Algorithmen müssen Referenzen zu *NIL* durch T.*nil* ersetzt werden.

Red-Black Tree

Manipulation

Algorithmen, die auf Red-Black Trees zugeschnitten sind:

- Knoten einfügen (*Insert*)
  Einen Knoten an der richtigen Stelle im Baum einfügen.
- Knoten löschen (*Delete*)
  Einen bestimmten Knoten aus dem Baum löschen.

Jedes Einfügen oder Löschen eines Knotens erfordert gegebenenfalls die Wiederherstellung der Red-Black Tree-Bedingungen:

- Welche Farbe sollte ein einzufügender Knoten bekommen?
  - Rot: Könnte Bedingung 2 verletzen.
  - Schwarz: Könnte Bedingung 3 verletzen.
- Was geschieht, wenn ein Knoten einer Farbe gelöscht wird?
  - Rot: Kein Problem, Bedingungen 1, 2, und 3 können nicht verletzt werden.
  - Schwarz: Jede der Bedingungen könnte verletzt werden.

→ Der Red-Black Tree muss gegebenenfalls rekonfiguriert werden.

Red-Black Tree

Manipulation: Einfügen

Algorithmus:	Red-Black Insert.
Eingabe:	T. Red-Black-Tree.
	z. Einzufügender Knoten mit Schlüssel k.
Ausgabe:	Um z erweiterter Red-Black Tree.

Red-Black Tree

Manipulation: Einfügen

Algorithmus: Red-Black Insert.

Eingabe: T. Red-Black-Tree.
z. Einzufügender Knoten mit Schlüssel k.

Ausgabe: Um z erweiterter Red-Black Tree.

RBInsert(T, z)

1. *TreeInsert*(T, z)
2. z.*left* $= T$.*nil*
3. z.*right* $= T$.*nil*
4. z.*color* $=$ *RED*
5. *RBInsertFixup*(T, z)

Red-Black Tree

Manipulation: Einfügen

Algorithmus: Red-Black Insert.

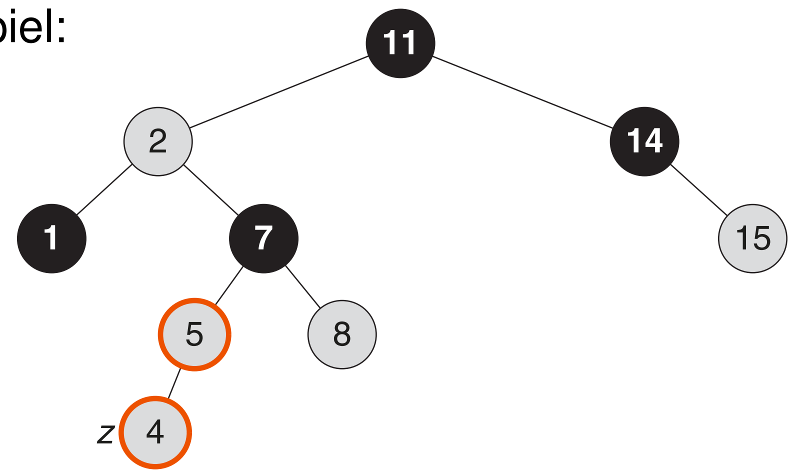
Eingabe: T. Red-Black-Tree.
z. Einzufügender Knoten mit Schlüssel k.

Ausgabe: Um z erweiterter Red-Black Tree.

RBInsert(T, z)

1. *TreeInsert*(T, z)
2. z.*left* $= T$.*nil*
3. z.*right* $= T$.*nil*
4. z.*color* $=$ *RED*
5. *RBInsertFixup*(T, z)

Beispiel:



- Nach dem Einfügen wird z rot gefärbt und anschließend eventuelle Verletzungen der Red-Black Tree-Bedingungen korrigiert.
- Ersetze in *TreeInsert* alle Referenzen zu *NIL* durch T.*nil*.

Red-Black Tree

Manipulation: Einfügen

Algorithmus: Red-Black Insert Fixup.

Eingabe: T. Potentiell invalider Red-Black Tree.
z. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black Tree.

Red-Black Tree

Manipulation: Einfügen

Algorithmus: Red-Black Insert Fixup.

Eingabe: T. Potentiell invalider Red-Black Tree.
z. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black Tree.

Fallunterscheidung:

1. z ist die Wurzel von T.
   Färbe die Wurzel schwarz.

2. zs Elter ist schwarz.
   Es ist nichts zu tun.

3. zs Elter ist rot und sein Onkel y ist rot.
   Färbe zs Großelter rot und dessen Kinder schwarz. Fahre beim Großelter fort.

4. zs Elter ist rot, sein Onkel y ist schwarz und z ist rechtes Kind.
   Linksrotation über zs Elter. Fahre mit zs vorigem Elter bei Fall 5 fort.

5. zs Elter ist rot, sein Onkel y ist schwarz und z ist linkes Kind.
   Färbe zs Elter schwarz und zs Großelter rot. Rechtsrotation über zs Großelter.

Red-Black Tree

Manipulation: Einfügen

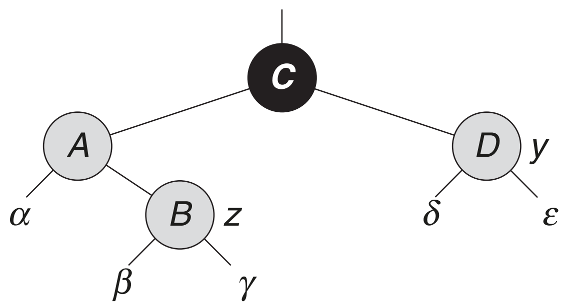
Algorithmus: Red-Black Insert Fixup.

Eingabe: T. Potentiell invalider Red-Black Tree.
z. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black Tree.

Fallunterscheidung:

3. zs Elter ist rot und sein Onkel y ist rot.
   Färbe zs Großelter rot und dessen Kinder schwarz. Fahre beim Großelter fort.



Red-Black Tree

Manipulation: Einfügen

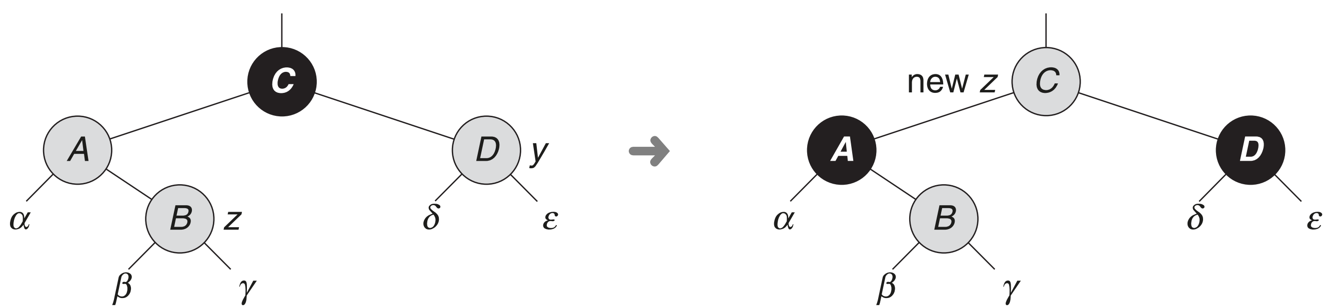
Algorithmus: Red-Black Insert Fixup.

Eingabe: T. Potentiell invalider Red-Black Tree.
z. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black Tree.

Fallunterscheidung:

3. zs Elter ist rot und sein Onkel y ist rot.
   Färbe zs Großelter rot und dessen Kinder schwarz. Fahre beim Großelter fort.



Red-Black Tree

Manipulation: Einfügen

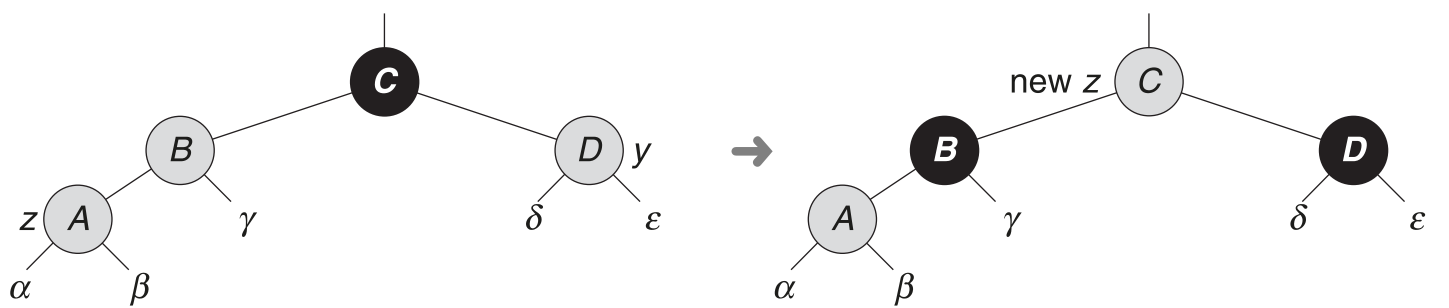
Algorithmus: Red-Black Insert Fixup.

Eingabe: T. Potentiell invalider Red-Black Tree.
z. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black Tree.

Fallunterscheidung:

3. zs Elter ist rot und sein Onkel y ist rot.
   Färbe zs Großelter rot und dessen Kinder schwarz. Fahre beim Großelter fort.



Red-Black Tree

Manipulation: Einfügen

Algorithmus: Red-Black Insert Fixup.

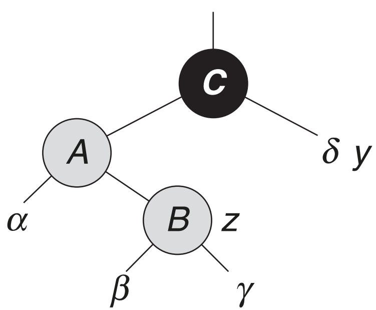
Eingabe: T. Potentiell invalider Red-Black Tree.
z. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black Tree.

Fallunterscheidung:

4. zs Elter ist rot, sein Onkel y ist schwarz und z ist rechtes Kind.
   Linksrotation über zs Elter. Fahre mit zs vorigem Elter bei Fall 5 fort.

5. zs Elter ist rot, sein Onkel y ist schwarz und z ist linkes Kind.
   Färbe zs Elter schwarz und zs Großelter rot. Rechtsrotation über zs Großelter.



Red-Black Tree

Manipulation: Einfügen

Algorithmus: Red-Black Insert Fixup.

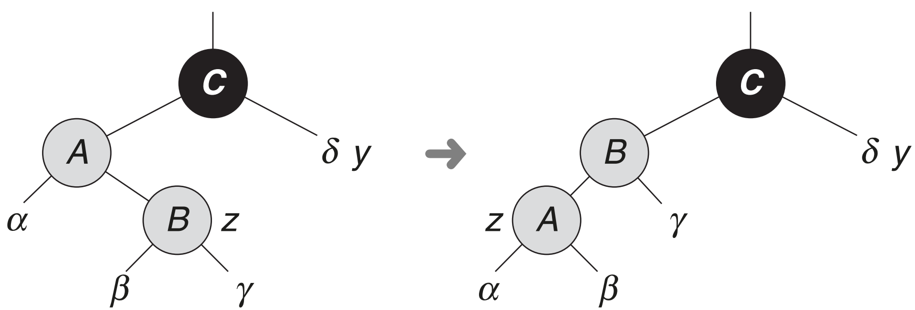
Eingabe: T. Potentiell invalider Red-Black Tree.
z. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black Tree.

Fallunterscheidung:

4. zs Elter ist rot, sein Onkel y ist schwarz und z ist rechtes Kind.
   Linksrotation über zs Elter. Fahre mit zs vorigem Elter bei Fall 5 fort.

5. zs Elter ist rot, sein Onkel y ist schwarz und z ist linkes Kind.
   Färbe zs Elter schwarz und zs Großelter rot. Rechtsrotation über zs Großelter.



Red-Black Tree

Manipulation: Einfügen

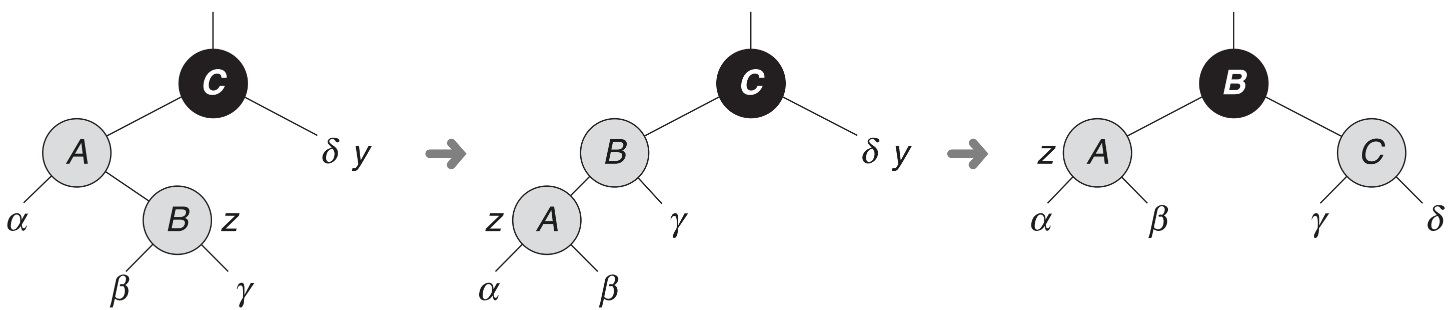
Algorithmus: Red-Black Insert Fixup.

Eingabe: T. Potentiell invalider Red-Black Tree.
z. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black Tree.

Fallunterscheidung:

4. zs Elter ist rot, sein Onkel y ist schwarz und z ist rechtes Kind.
   Linksrotation über zs Elter. Fahre mit zs vorigem Elter bei Fall 5 fort.
5. zs Elter ist rot, sein Onkel y ist schwarz und z ist linkes Kind.
   Färbe zs Elter schwarz und zs Großelter rot. Rechtsrotation über zs Großelter.



Red-Black Tree

Manipulation: Einfügen

RBInsertFixup(T, z)

```
WHILE z.parent.color == RED DO
  IF z.parent == z.parent.parent.left THEN
    y = z.parent.parent.right
    IF 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
        LeftRotate(T, z)
      ENDIF
      z.parent.color = BLACK
      z.parent.parent.color = RED
      RightRotate(T, z.parent.parent)
    ENDIF
  ELSE
    // THEN-clause with "right" and "left" exchanged.
  ENDIF
ENDDO
T.root.color = BLACK
```

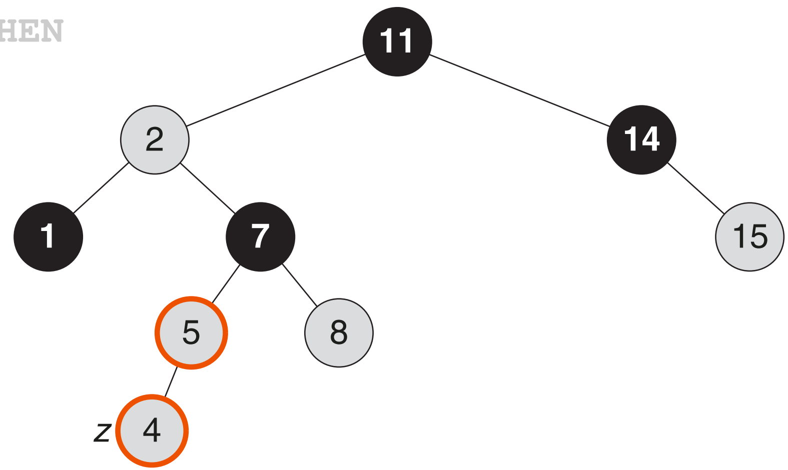
Red-Black Tree

Manipulation: Einfügen

RBInsertFixup(T, z)

```
WHILE z.parent.color == RED DO
  IF z.parent == z.parent.parent.left THEN
    y = z.parent.parent.right
    IF 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
        LeftRotate(T, z)
      ENDIF
      z.parent.color = BLACK
      z.parent.parent.color = RED
      RightRotate(T, z.parent.parent)
    ENDIF
  ELSE
    // THEN-clause with "right" and "left" exchanged.
  ENDIF
ENDDO
T.root.color = BLACK
```

Beispiel:



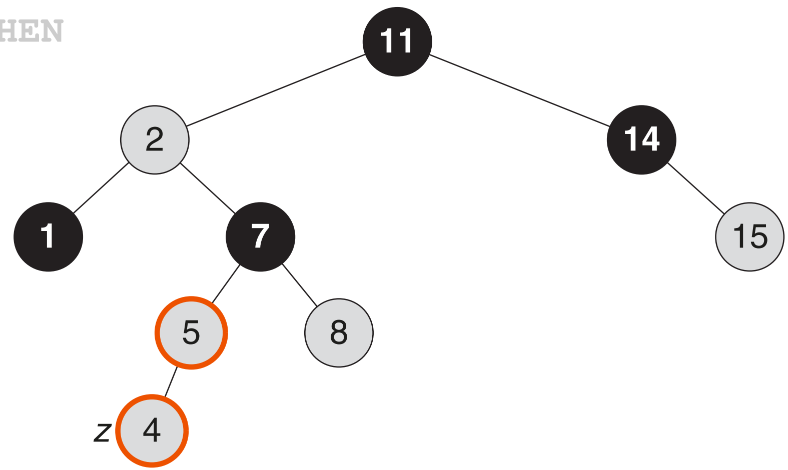
Red-Black Tree

Manipulation: Einfügen

RBInsertFixup(T, z)

```
WHILE z.parent.color == RED DO
  IF z.parent == z.parent.parent.left THEN
    y = z.parent.parent.right
    IF 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
        LeftRotate(T, z)
      ENDIF
      z.parent.color = BLACK
      z.parent.parent.color = RED
      RightRotate(T, z.parent.parent)
    ENDIF
  ELSE
    // THEN-clause with "right" and "left" exchanged.
  ENDIF
ENDDO
T.root.color = BLACK
```

Beispiel:



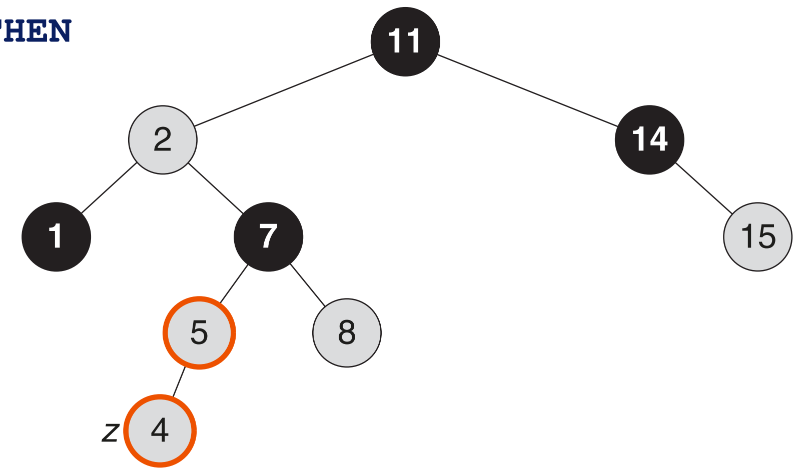
Red-Black Tree

Manipulation: Einfügen

RBInsertFixup(T, z)

```
WHILE z.parent.color == RED DO
  IF z.parent == z.parent.parent.left THEN
    y = z.parent.parent.right
    IF 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
        LeftRotate(T, z)
      ENDIF
      z.parent.color = BLACK
      z.parent.parent.color = RED
      RightRotate(T, z.parent.parent)
    ENDIF
  ELSE
    // THEN-clause with "right" and "left" exchanged.
  ENDIF
ENDDO
T.root.color = BLACK
```

Beispiel:



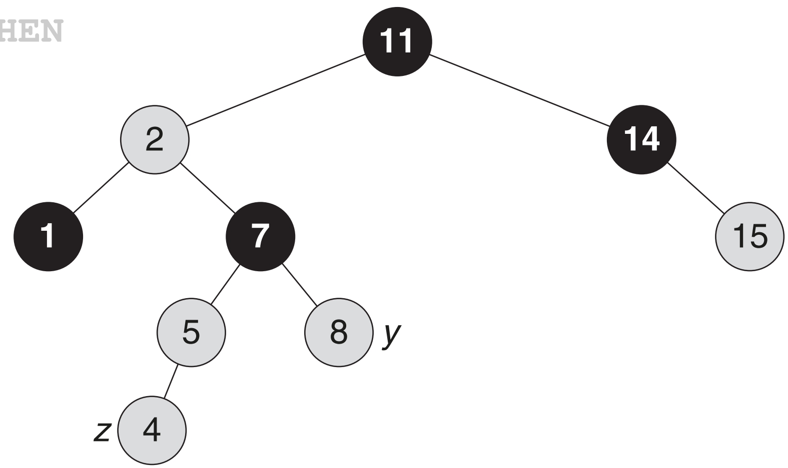
Red-Black Tree

Manipulation: Einfügen

RBInsertFixup(T, z)

```
WHILE z.parent.color == RED DO
  IF z.parent == z.parent.parent.left THEN
    y = z.parent.parent.right
    IF 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
        LeftRotate(T, z)
      ENDIF
      z.parent.color = BLACK
      z.parent.parent.color = RED
      RightRotate(T, z.parent.parent)
    ENDIF
  ELSE
    // THEN-clause with "right" and "left" exchanged.
  ENDIF
ENDDO
T.root.color = BLACK
```

Beispiel:



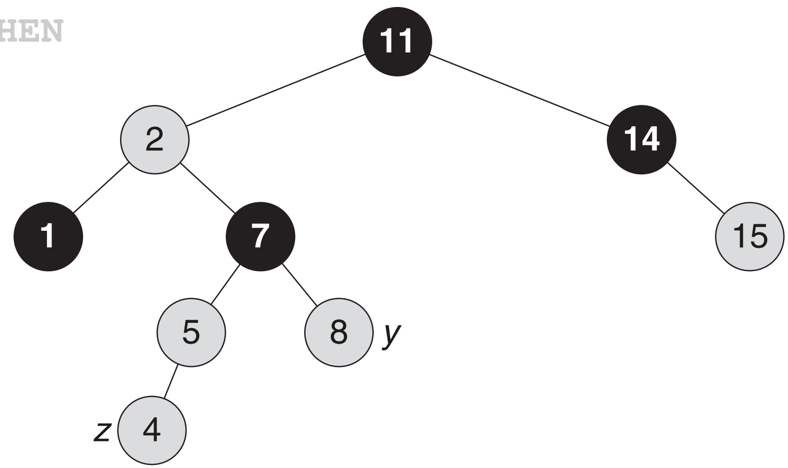
Red-Black Tree

Manipulation: Einfügen

RBInsertFixup(T, z)

```
WHILE z.parent.color == RED DO
  IF z.parent == z.parent.parent.left THEN
    y = z.parent.parent.right
    IF 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
        LeftRotate(T, z)
      ENDIF
      z.parent.color = BLACK
      z.parent.parent.color = RED
      RightRotate(T, z.parent.parent)
    ENDIF
  ELSE
    // THEN-clause with "right" and "left" exchanged.
  ENDIF
ENDDO
T.root.color = BLACK
```

Beispiel:



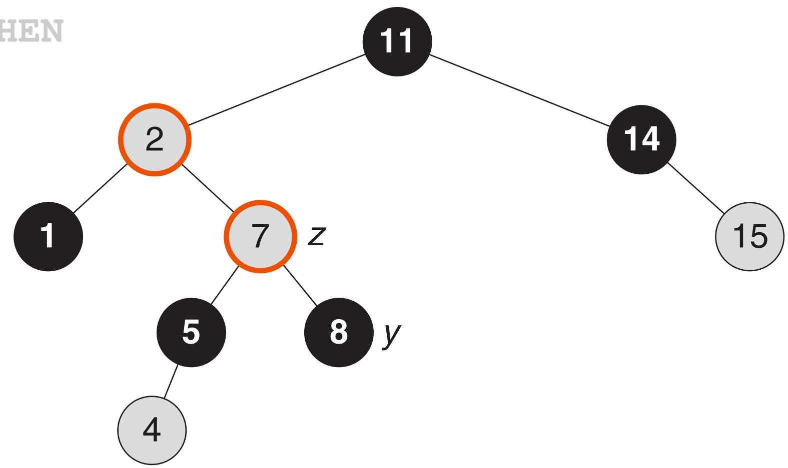
Red-Black Tree

Manipulation: Einfügen

RBInsertFixup(T, z)

```
WHILE z.parent.color == RED DO
  IF z.parent == z.parent.parent.left THEN
    y = z.parent.parent.right
    IF 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
        LeftRotate(T, z)
      ENDIF
      z.parent.color = BLACK
      z.parent.parent.color = RED
      RightRotate(T, z.parent.parent)
    ENDIF
  ELSE
    // THEN-clause with "right" and "left" exchanged.
  ENDIF
ENDDO
T.root.color = BLACK
```

Beispiel:



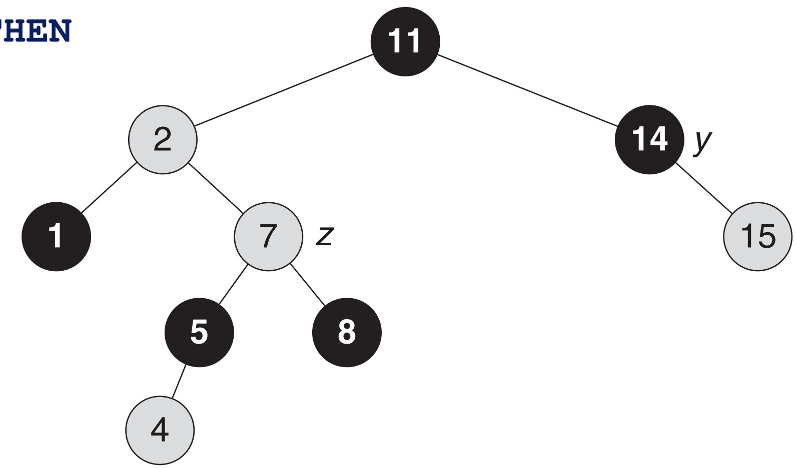
Red-Black Tree

Manipulation: Einfügen

RBInsertFixup(T, z)

```
WHILE z.parent.color == RED DO
  IF z.parent == z.parent.parent.left THEN
    y = z.parent.parent.right
    IF 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
        LeftRotate(T, z)
      ENDIF
      z.parent.color = BLACK
      z.parent.parent.color = RED
      RightRotate(T, z.parent.parent)
    ENDIF
  ELSE
    // THEN-clause with "right" and "left" exchanged.
  ENDIF
ENDDO
T.root.color = BLACK
```

Beispiel:



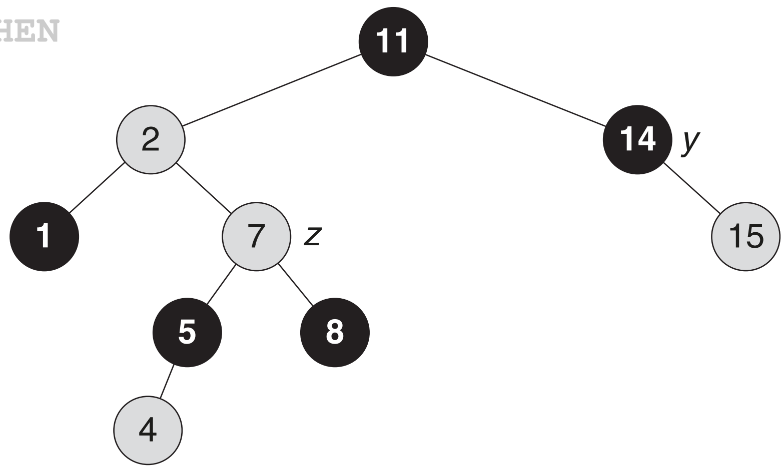
Red-Black Tree

Manipulation: Einfügen

RBInsertFixup(T, z)

```
WHILE z.parent.color == RED DO
  IF z.parent == z.parent.parent.left THEN
    y = z.parent.parent.right
    IF 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
        LeftRotate(T, z)
      ENDIF
      z.parent.color = BLACK
      z.parent.parent.color = RED
      RightRotate(T, z.parent.parent)
    ENDIF
  ELSE
    // THEN-clause with "right" and "left" exchanged.
  ENDIF
ENDDO
T.root.color = BLACK
```

Beispiel:



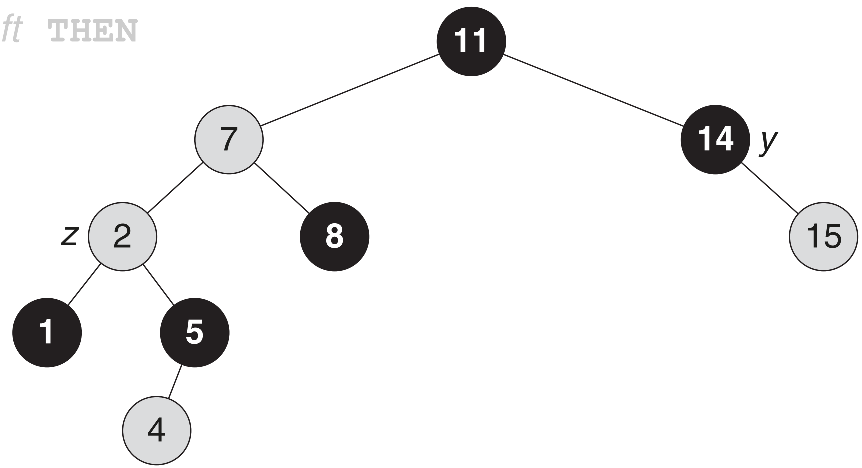
Red-Black Tree

Manipulation: Einfügen

RBInsertFixup(T, z)

```
WHILE z.parent.color == RED DO
  IF z.parent == z.parent.parent.left THEN
    y = z.parent.parent.right
    IF 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
        LeftRotate(T, z)
      ENDIF
      z.parent.color = BLACK
      z.parent.parent.color = RED
      RightRotate(T, z.parent.parent)
    ENDIF
  ELSE
    // THEN-clause with "right" and "left" exchanged.
  ENDIF
ENDDO
T.root.color = BLACK
```

Beispiel:



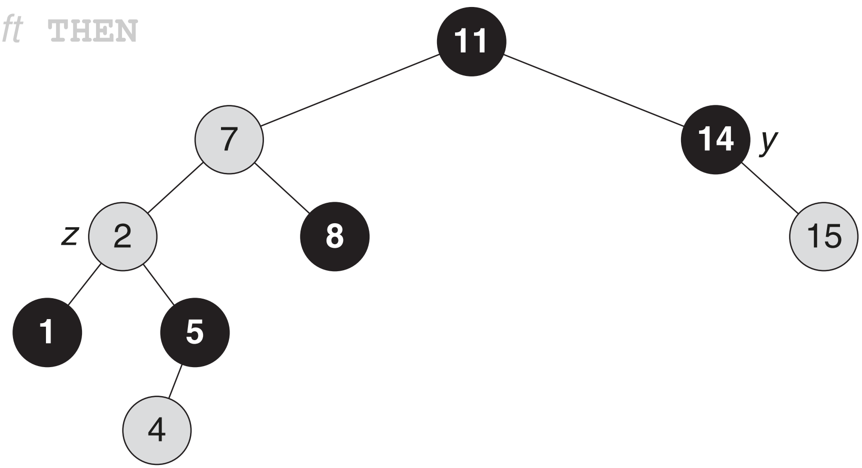
Red-Black Tree

Manipulation: Einfügen

RBInsertFixup(T, z)

```
WHILE z.parent.color == RED DO
  IF z.parent == z.parent.parent.left THEN
    y = z.parent.parent.right
    IF 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
        LeftRotate(T, z)
      ENDIF
      z.parent.color = BLACK
      z.parent.parent.color = RED
      RightRotate(T, z.parent.parent)
    ENDIF
  ELSE
    // THEN-clause with "right" and "left" exchanged.
  ENDIF
ENDDO
T.root.color = BLACK
```

Beispiel:



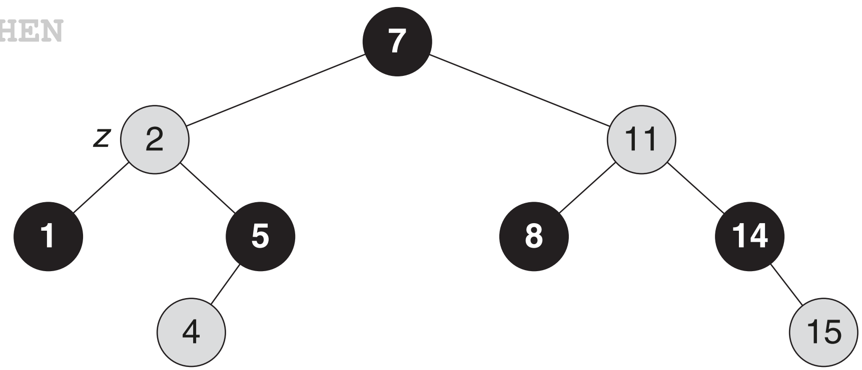
Red-Black Tree

Manipulation: Einfügen

RBInsertFixup(T, z)

```
WHILE z.parent.color == RED DO
  IF z.parent == z.parent.parent.left THEN
    y = z.parent.parent.right
    IF 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
        LeftRotate(T, z)
      ENDIF
      z.parent.color = BLACK
      z.parent.parent.color = RED
      RightRotate(T, z.parent.parent)
    ENDIF
  ELSE
    // THEN-clause with "right" and "left" exchanged.
  ENDIF
ENDDO
T.root.color = BLACK
```

Beispiel:



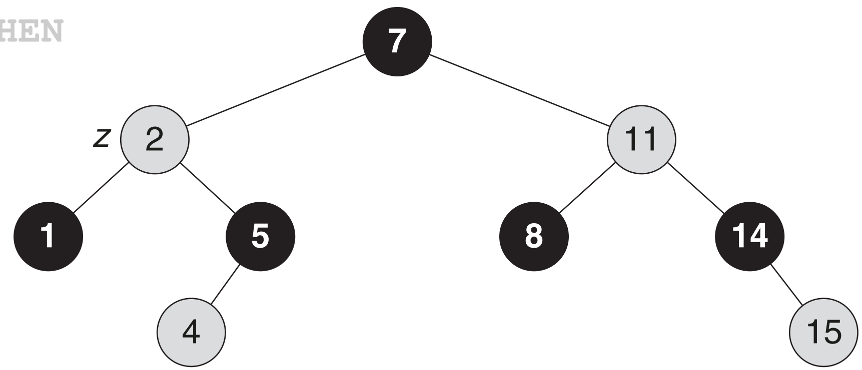
Red-Black Tree

Manipulation: Einfügen

RBInsertFixup(T, z)

```
WHILE z.parent.color == RED DO
  IF z.parent == z.parent.parent.left THEN
    y = z.parent.parent.right
    IF 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
        LeftRotate(T, z)
      ENDIF
      z.parent.color = BLACK
      z.parent.parent.color = RED
      RightRotate(T, z.parent.parent)
    ENDIF
  ELSE
    // THEN-clause with "right" and "left" exchanged.
  ENDIF
ENDDO
T.root.color = BLACK
```

Beispiel:



Red-Black Tree

Manipulation: Löschen [Binary Search Tree]

Algorithmus: Red-Black Transplant.

Eingabe: T. Red-Black Tree.
u, v. Wurzeln eines Teilbaums von T

Ausgabe: Binärbaum, bei dem u durch v ersetzt wurde.

RBTransplant(T, u, v)

```
1. IF u.parent == T.nil THEN
2.   T.root = v
3. ELSE IF u == u.parent.left THEN
4.   u.parent.left = v
5. ELSE
6.   u.parent.right = v
7. ENDIF
8. v.parent = u.parent
```

Unterschiede zu *Transplant*:

- Zeile 1: Referenz zu *NIL* durch T.*nil* ersetzt.
- Die Zuweisung in Zeile 8 ist nicht mehr abhängig vom Elter: Nutzung des Sentinels.

Red-Black Tree

Manipulation: Löschen

Algorithmus: Tree Delete.

Eingabe: T. Red-Black Tree.
z. Zu löschender Knoten.

Ausgabe: Red-Black Tree, bei dem z gelöscht wurde.

Fallunterscheidung:

1. z hat keine Kinder.
   Ersetze z bei seinem Elter durch T.*nil*.

2. z hat ein Kind.
   Ersetze z bei seinem Elter durch (a) sein rechtes bzw. (b) sein linkes Kind.

3. z hat zwei Kinder.
   (a) zs Nachfolger y ist sein rechtes Kind.
   Ersetze z durch y.

   (b) zs Nachfolger y ist ein anderer Knoten seinem rechten Teilbaum.
   Ersetze y durch sein rechtes Kind und mache y zur Wurzel von zs rechtem Teilbaum.
   Ersetze z durch y.

Red-Black Tree

Manipulation: Löschen

RBDelete(T, z)

```
y = z
yOriginalColor = y.color
IF z.left == T.nil THEN
   x = z.right
   RBTransplant(T, z, z.right)
ELSE IF z.right == T.nil THEN
   x = z.left
   RBTransplant(T, z, z.left)
ELSE
   y = TreeMinimum(z.right)
   yOriginalColor = y.color
   x = y.right
   IF y.parent == z THEN
      x.parent = y
   ELSE
      RBTransplant(T, y, y.right)
      y.right = z.right
      y.right.parent = y
   ENDIF
   RBTransplant(T, z, y)
   y.left = z.left
   y.left.parent = y
   y.color = z.color
ENDIF
IF yOriginalColor == BLACK THEN
   RBDeleteFixup(T, x)
ENDIF
```

Red-Black Tree

Manipulation: Löschen

RBDelete(T, z)

RBDelete „enthält“ *TreeDelete*.

```
y = z
yOriginalColor = y.color
IF z.left == T.nil THEN
   x = z.right
   RBTransplant(T, z, z.right)
ELSE IF z.right == T.nil THEN
   x = z.left
   RBTransplant(T, z, z.left)
ELSE
   y = TreeMinimum(z.right)
   yOriginalColor = y.color
   x = y.right
   IF y.parent == z THEN
      x.parent = y
   ELSE
      RBTransplant(T, y, y.right)
      y.right = z.right
      y.right.parent = y
   ENDIF
   RBTransplant(T, z, y)
   y.left = z.left
   y.left.parent = y
   y.color = z.color
ENDIF
IF yOriginalColor == BLACK THEN
   RBDeleteFixup(T, x)
ENDIF
```

Red-Black Tree

Manipulation: Löschen

RBDelete(T, z)

```
y = z
yOriginalColor = y.color
IF z.left == T.nil THEN
  x = z.right
  RBTransplant(T, z, z.right)
ELSE IF z.right == T.nil THEN
  x = z.left
  RBTransplant(T, z, z.left)
ELSE
  y = TreeMinimum(z.right)
  yOriginalColor = y.color
  x = y.right
  IF y.parent == z THEN
    x.parent = y
  ELSE
    RBTransplant(T, y, y.right)
    y.right = z.right
    y.right.parent = y
  ENDIF
  RBTransplant(T, z, y)
  y.left = z.left
  y.left.parent = y
  y.color = z.color
ENDIF
IF yOriginalColor == BLACK THEN
  RBDeleteFixup(T, x)
ENDIF
```

RBDelete „enthält“ *TreeDelete*.

Dazu kommt Code zur Vorbereitung der Korrektur von Verletzungen der Red-Black Tree-Bedingungen.

Red-Black Tree

Manipulation: Löschen

RBDelete(T, z)

```
y = z
yOriginalColor = y.color
IF z.left == T.nil THEN
   x = z.right
   RBTransplant(T, z, z.right)
ELSE IF z.right == T.nil THEN
   x = z.left
   RBTransplant(T, z, z.left)
ELSE
   y = TreeMinimum(z.right)
   yOriginalColor = y.color
   x = y.right
   IF y.parent == z THEN
      x.parent = y
   ELSE
      RBTransplant(T, y, y.right)
      y.right = z.right
      y.right.parent = y
   ENDIF
   RBTransplant(T, z, y)
   y.left = z.left
   y.left.parent = y
   y.color = z.color
ENDIF
IF yOriginalColor == BLACK THEN
   RBDeleteFixup(T, x)
ENDIF
```

RBDelete „enthält“ *TreeDelete*.

Dazu kommt Code zur Vorbereitung der Korrektur von Verletzungen der Red-Black Tree-Bedingungen.

- Hilfsvariablen y und *yOriginalColor*.

  Knoten, der fallabhängig gelöscht oder verschoben wird und seine initiale Farbe.

- Hilfsvariable x.

  Knoten, der y ersetzt; ys einziges Kind oder T.*nil*. x ist Ursache für mögliche Verletzungen.

- Fälle 1 und 2.

  $y = z$ wird durch x ersetzt.

- Fall 3.

  y wird durch x ersetzt, an die Stelle zs verschoben und erhält zs Farbe.

Red-Black Tree

Manipulation: Löschen

RBDelete(T, z)

```
y = z
yOriginalColor = y.color
IF z.left == T.nil THEN
   x = z.right
   RBTransplant(T, z, z.right)
ELSE IF z.right == T.nil THEN
   x = z.left
   RBTransplant(T, z, z.left)
ELSE
   y = TreeMinimum(z.right)
   yOriginalColor = y.color
   x = y.right
   IF y.parent == z THEN
      x.parent = y
   ELSE
      RBTransplant(T, y, y.right)
      y.right = z.right
      y.right.parent = y
   ENDIF
   RBTransplant(T, z, y)
   y.left = z.left
   y.left.parent = y
   y.color = z.color
ENDIF
IF yOriginalColor == BLACK THEN
   RBDeleteFixup(T, x)
ENDIF
```

RBDelete „enthält" *TreeDelete*.

Dazu kommt Code zur Vorbereitung der Korrektur von Verletzungen der Red-Black Tree-Bedingungen.

- Hilfsvariablen y und *yOriginalColor*.

  Knoten, der fallabhängig gelöscht oder verschoben wird und seine initiale Farbe.

- Hilfsvariable x.

  Knoten, der y ersetzt; ys einziges Kind oder T.*nil*. x ist Ursache für mögliche Verletzungen.

- Fälle 1 und 2.

  $y = z$ wird durch x ersetzt.

- Fall 3.

  y wird durch x ersetzt, an die Stelle zs verschoben und erhält zs Farbe.

- Vorbereitung der Korrektur von x.

  Setzen von x.*parent* auf den initialen Elter von y. Betrachte die letzte Zeile von *RBTransplant*.

Red-Black Tree

Manipulation: Löschen

Algorithmus: Red-Black Delete Fixup.

Eingabe: T. Potentiell invalider Red-Black-Tree.
x. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black-Tree.

Fallunterscheidung:

1. x ist Wurzel von T oder x ist rot.
   Färbe x schwarz.
2. xs Geschwister w ist rot.
   Färbe w schwarz und xs Elter rot. Linksrotation über xs Elter. Fahre bei Fällen 3, 4 oder 5 fort.
3. xs Geschwister w ist schwarz und beide Kinder von w sind schwarz.
   Färbe w rot und fahre bei xs Elter fort.
4. xs Geschwister w ist schwarz, ws linkes Kind rot und das rechte schwarz.
   Färbe ws linkes Kind schwarz und w rot. Rechtsrotation über w. Fahre bei Fall 5 fort.
5. xs Geschwister w ist schwarz und ws rechtes Kind rot.
   Färbe w wie xs Elter und den Elter und ws rechtes Kind schwarz. Linksrotation über xs Elter. Fahre bei x fort.

Red-Black Tree

Manipulation: Löschen

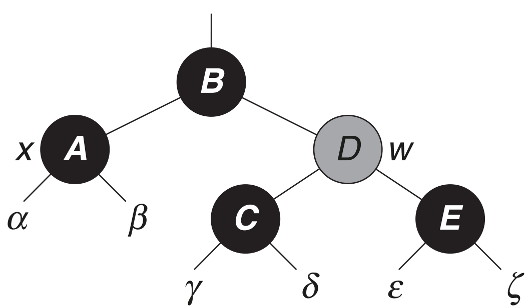
Algorithmus: Red-Black Delete Fixup.

Eingabe: T. Potentiell invalider Red-Black-Tree.
x. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black-Tree.

Fallunterscheidung:

2. xs Geschwister w ist rot.
   Färbe w schwarz und xs Elter rot. Linksrotation über xs Elter. Fahre bei Fällen 3, 4 oder 5 fort.



Red-Black Tree

Manipulation: Löschen

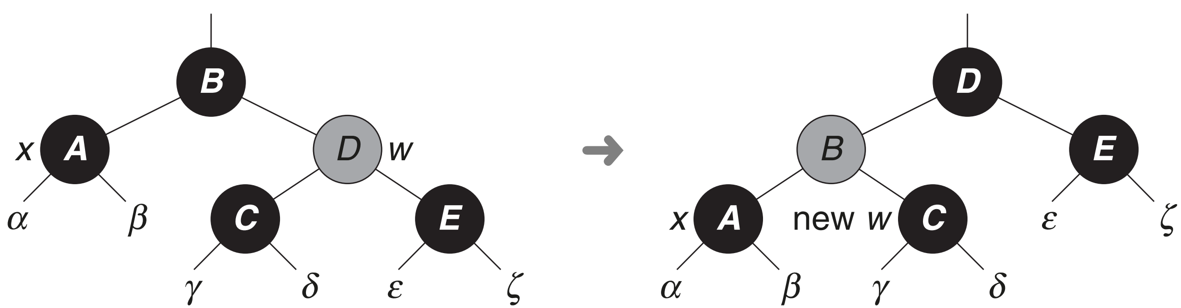
Algorithmus: Red-Black Delete Fixup.

Eingabe: T. Potentiell invalider Red-Black-Tree.
x. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black-Tree.

Fallunterscheidung:

2. xs Geschwister w ist rot.
   Färbe w schwarz und xs Elter rot. Linksrotation über xs Elter. Fahre bei Fällen 3, 4 oder 5 fort.



Red-Black Tree

Manipulation: Löschen

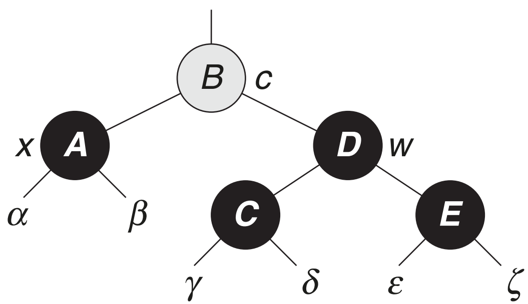
Algorithmus: Red-Black Delete Fixup.

Eingabe: T. Potentiell invalider Red-Black-Tree.
x. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black-Tree.

Fallunterscheidung:

3. xs Geschwister w ist schwarz und beide Kinder von w sind schwarz.
   Färbe w rot und fahre bei xs Elter fort.



Red-Black Tree

Manipulation: Löschen

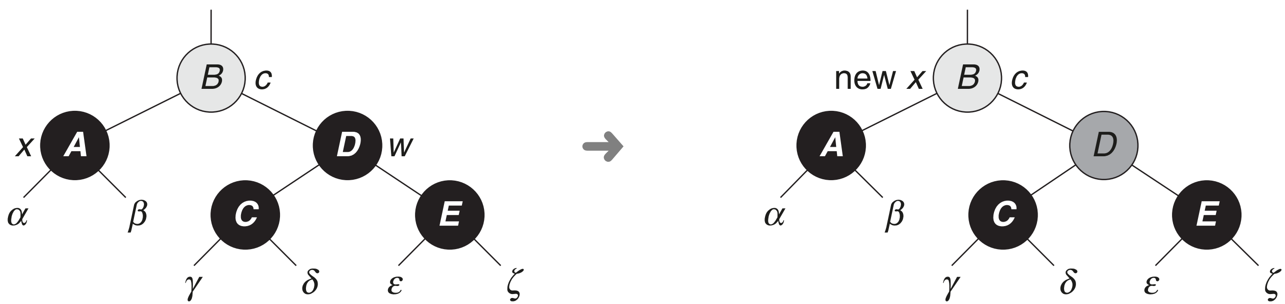
Algorithmus: Red-Black Delete Fixup.

Eingabe: T. Potentiell invalider Red-Black-Tree.
x. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black-Tree.

Fallunterscheidung:

3. xs Geschwister w ist schwarz und beide Kinder von w sind schwarz.
   Färbe w rot und fahre bei xs Elter fort.



Red-Black Tree

Manipulation: Löschen

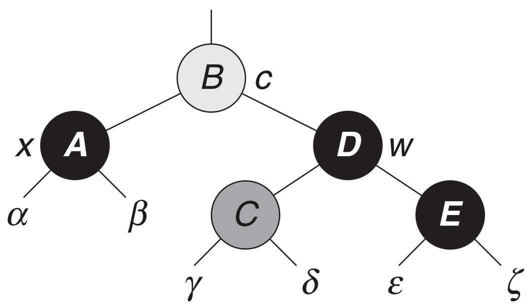
Algorithmus: Red-Black Delete Fixup.

Eingabe: T. Potentiell invalider Red-Black-Tree.
x. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black-Tree.

Fallunterscheidung:

4. xs Geschwister w ist schwarz, ws linkes Kind rot und das rechte schwarz.
   Färbe ws linkes Kind schwarz und w rot. Rechtsrotation über w. Fahre bei Fall 5 fort.



Red-Black Tree

Manipulation: Löschen

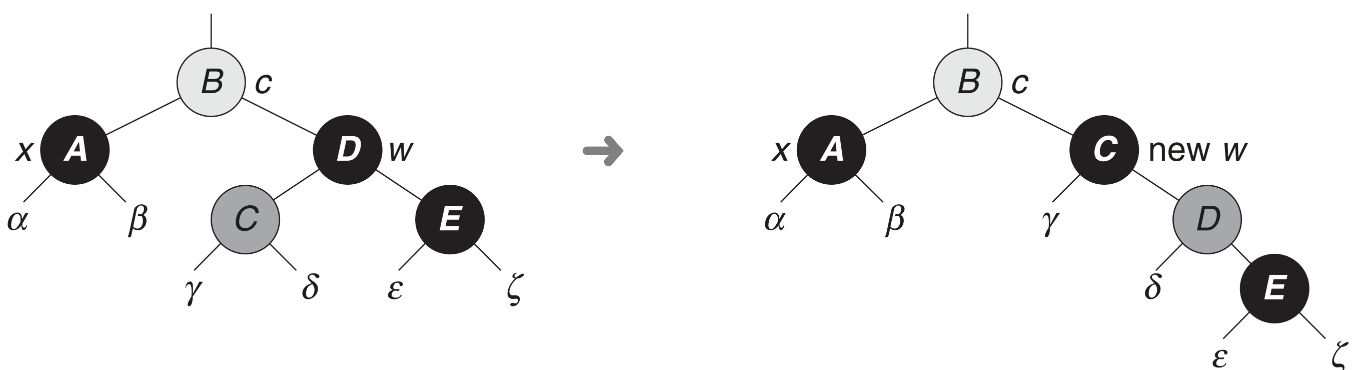
Algorithmus: Red-Black Delete Fixup.

Eingabe: T. Potentiell invalider Red-Black-Tree.
x. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black-Tree.

Fallunterscheidung:

4. xs Geschwister w ist schwarz, ws linkes Kind rot und das rechte schwarz.
Färbe ws linkes Kind schwarz und w rot. Rechtsrotation über w. Fahre bei Fall 5 fort.



Red-Black Tree

Manipulation: Löschen

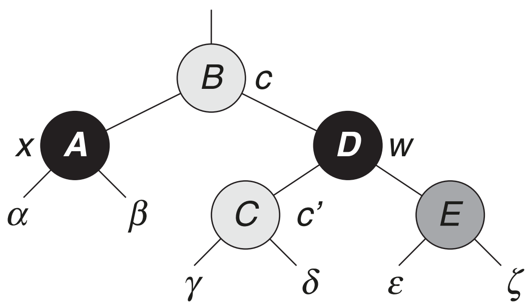
Algorithmus: Red-Black Delete Fixup.

Eingabe: T. Potentiell invalider Red-Black-Tree.
x. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black-Tree.

Fallunterscheidung:

4. xs Geschwister w ist schwarz und ws rechtes Kind rot.
   Färbe w wie xs Elter und den Elter und ws rechtes Kind schwarz. Linksrotation über xs Elter. Fahre bei x fort.



Red-Black Tree

Manipulation: Löschen

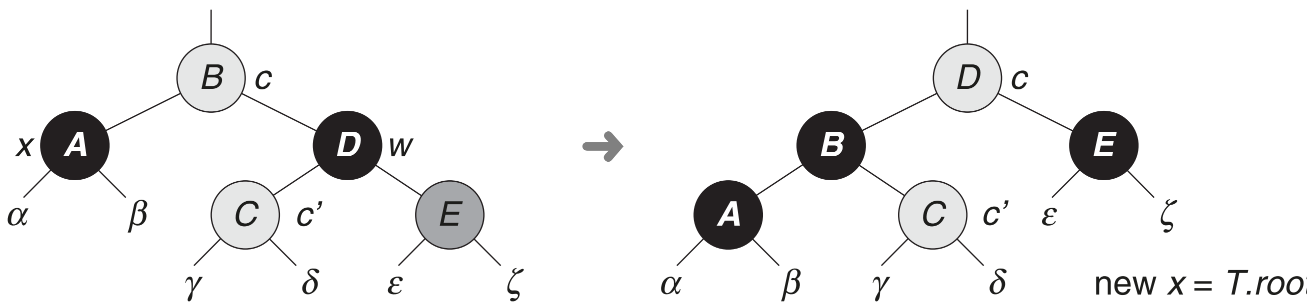
Algorithmus: Red-Black Delete Fixup.

Eingabe: T. Potentiell invalider Red-Black-Tree.
x. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black-Tree.

Fallunterscheidung:

4. xs Geschwister w ist schwarz und ws rechtes Kind rot.
   Färbe w wie xs Elter und den Elter und ws rechtes Kind schwarz. Linksrotation über xs Elter. Fahre bei x fort.



Bemerkungen:

- *RBDeleteFixup*(T, x)

```
WHILE x ≠ T.root AND x.color == BLACK DO
  IF x == x.parent.left THEN
    w = x.parent.right
    IF w.color == RED THEN
      w.color = BLACK
      x.parent.color = RED
      LeftRotate(T, x.parent)
      w = x.parent.right
    ENDIF
    IF w.left.color == BLACK AND w.right.color == BLACK THEN
      w.color = RED
      x = x.parent
    ELSE
      IF w.right.color == BLACK THEN
        w.left.color = BLACK
        w.color = RED
        RightRotate(T, w)
        w = x.parent.right
      ENDIF
      w.color = x.parent.color
      x.parent.color = BLACK
      w.right.color = BLACK
      LeftRotate(T, x.parent)
      x = T.root
    ENDIF
  ELSE
    // THEN-clause with "right" and "left" exchanged.
  ENDIF
ENDDO
x.color = BLACK
```