Kapitel ADS:V

V. Suchen

- Binary Search Tree
- AVL Tree
- Red-Black Tree
- Maschinenmodell (Erweiterung)
- B-Tree

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.

Binary Search Tree

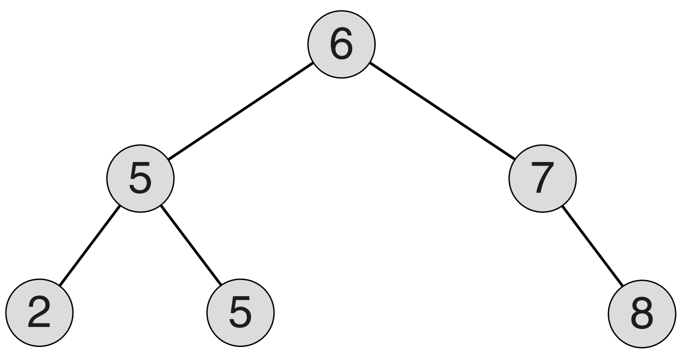
Definition

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

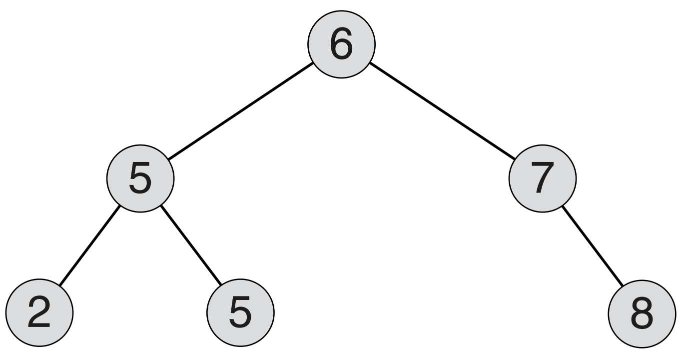
Definition

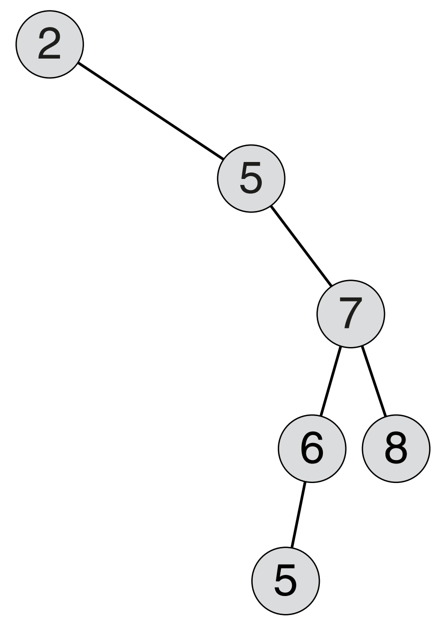
Ein Binärbaum T heißt Binary Search Tree (*Binärer Suchbaum*), wenn jeder seiner Knoten x folgende Bedingung erfüllt:

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

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, größtem oder nächstgrößerem Sortierschlüssel bestimmen.
- Knoten einfügen (*Insert*)
  Einen Knoten an der richtigen Stelle im Baum einfügen.
- Knoten löschen (*Delete*)
  Einen bestimmten Knoten aus dem Baum löschen.

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.

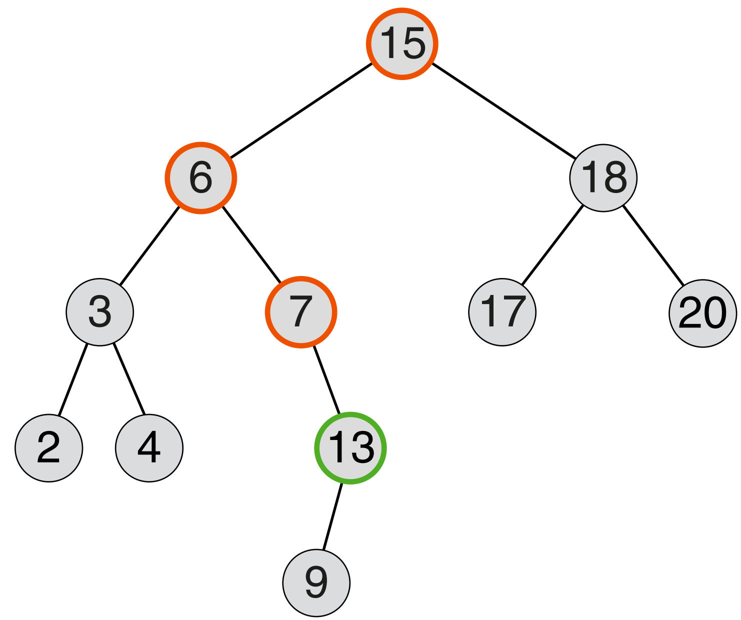
Eingabe: x. Wurzel eines Binary Search Tree T.
k. Gesuchter Schlüssel.

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 $x \neq$ *NIL* **AND** $k \neq x.$*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.

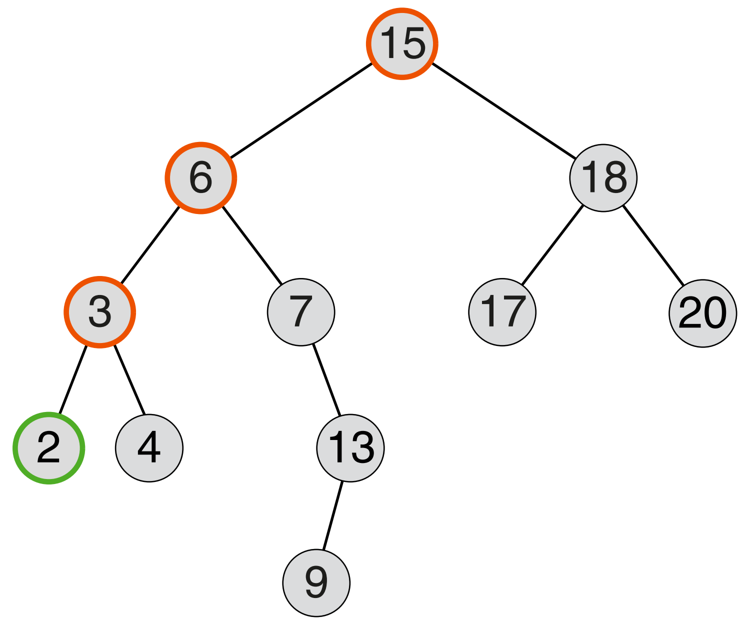
Eingabe: x. Wurzel eines Binary Search Tree T.

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.

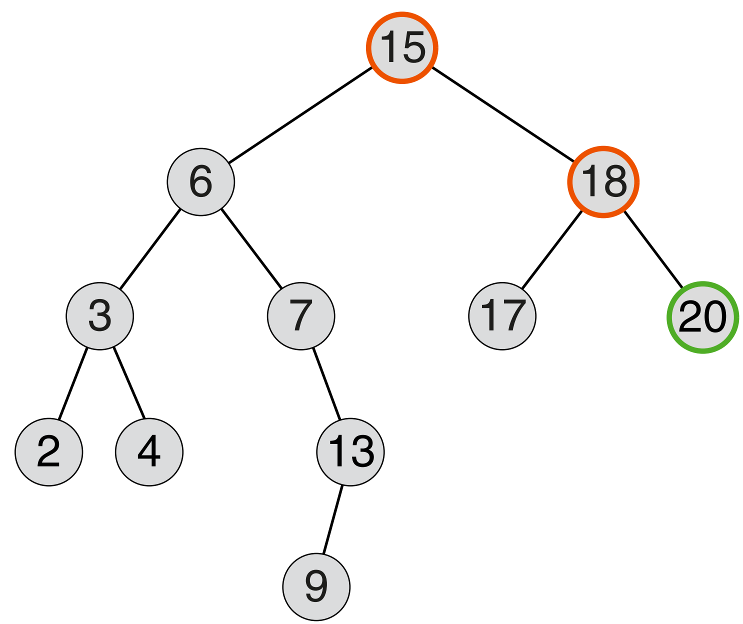
Eingabe: x. Wurzel eines Binary Search Tree T.

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*$(T.root)$



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)

1. **IF** $x ==$ *NIL* **THEN** *return*(*NIL*) **ENDIF**
2. **WHILE** x.*right* $\neq$ *NIL* **DO**
3. $x = x$.*right*
4. **ENDDO**
5. *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*.

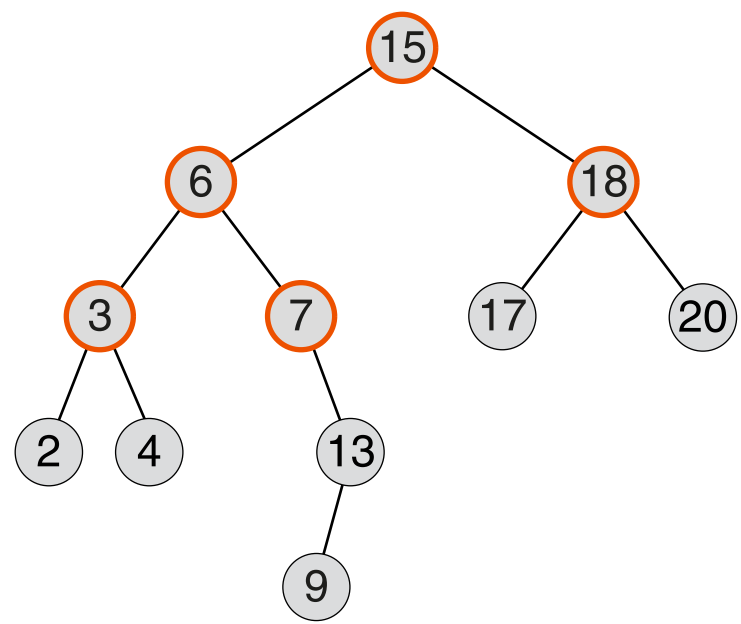
TreeSuccessor(x)

```
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*.

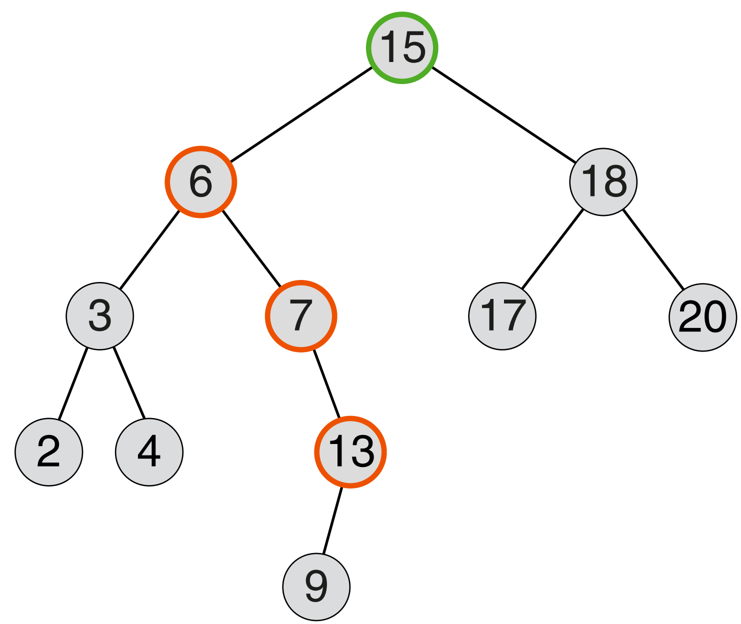
TreeSuccessor(x)

```
IF x.right ≠ 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), 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* $\neq$ *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)

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.

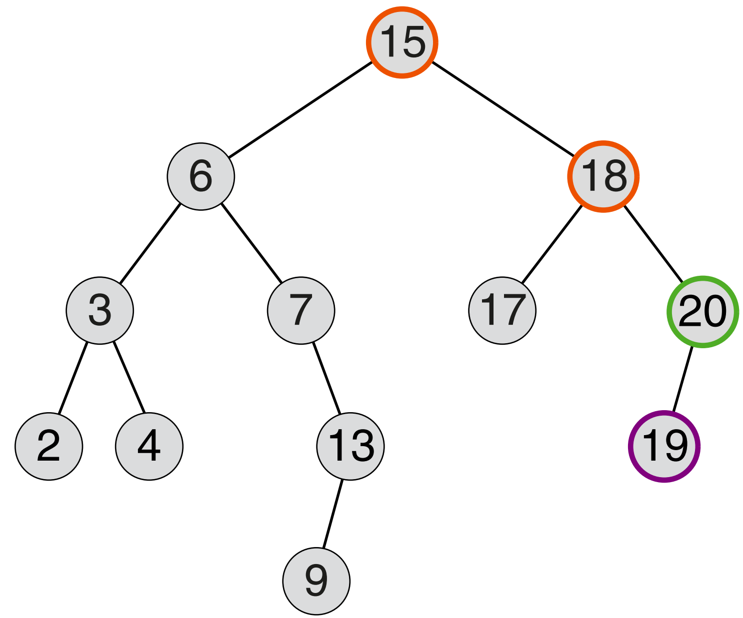
Binary Search Tree

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), $z.key = 19$



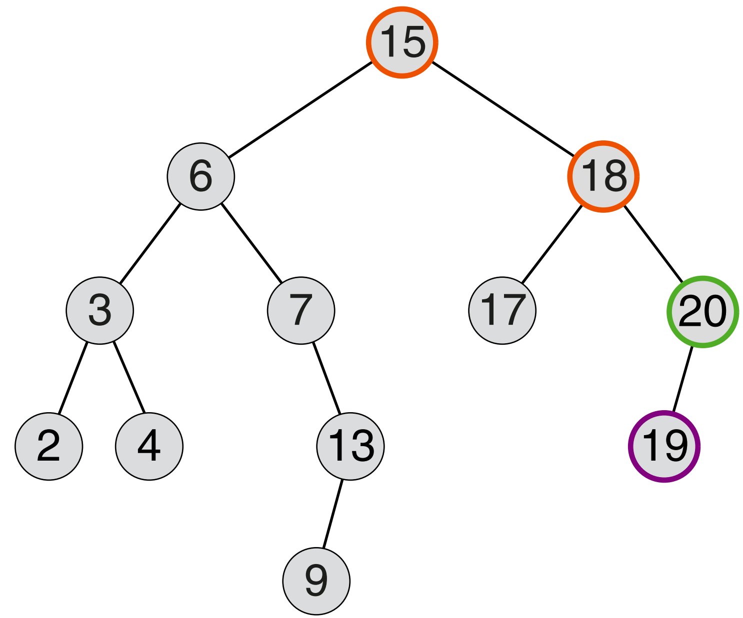
Binary Search Tree

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), $z.\mathit{key} = 19$



Laufzeit:

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

→ $O(h)$, wobei h die Baumhöhe ist.

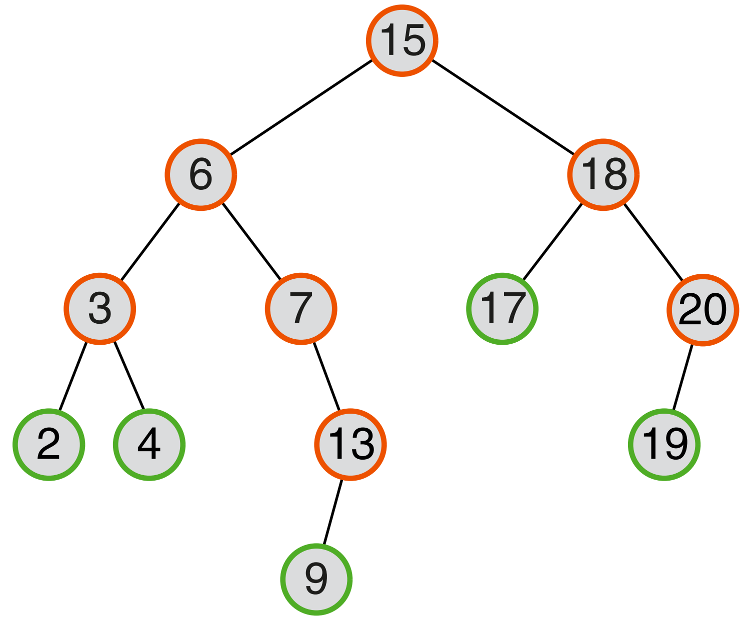
Binary Search Tree

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:



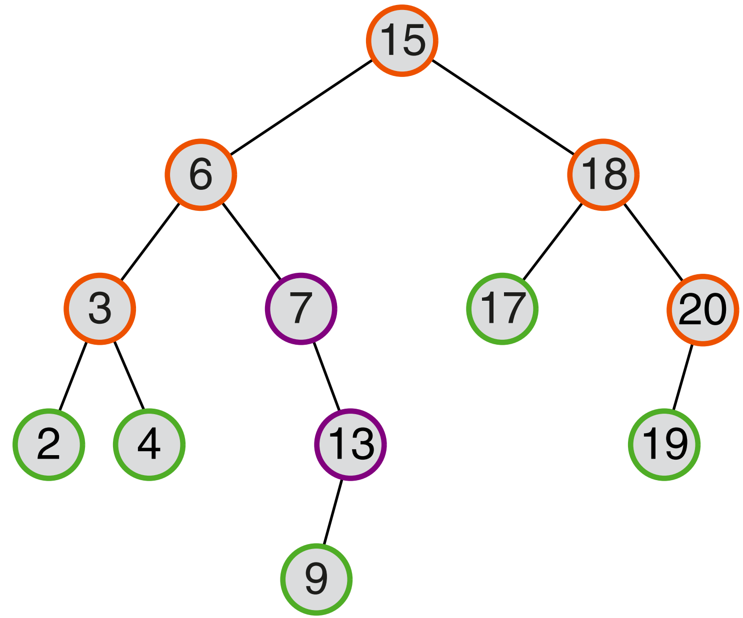
Binary Search Tree

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

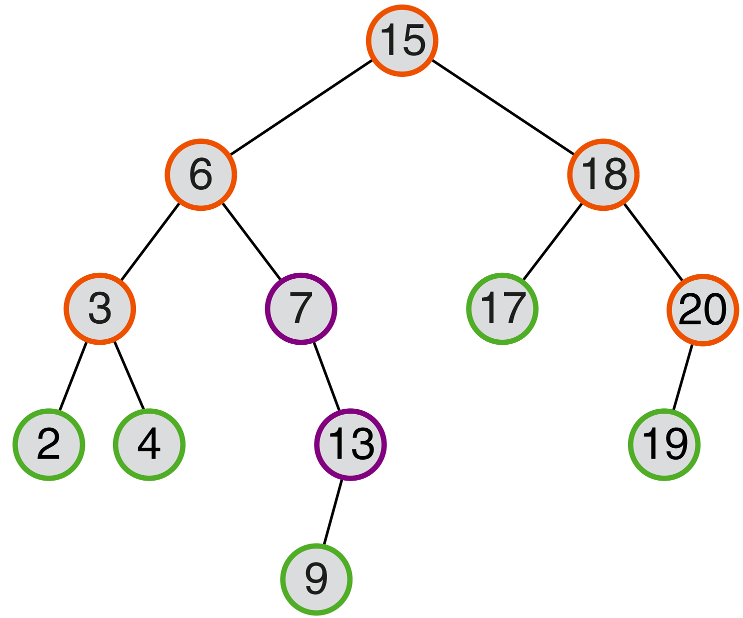
Manipulation: Löschen

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.

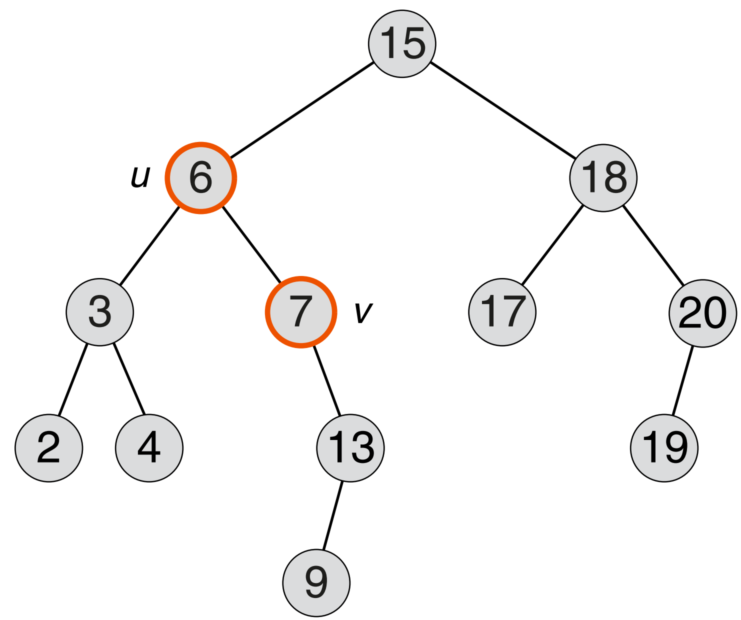
Eingabe: T. Binary Search Tree.
u, v. Wurzeln von Teilbäumen von T; ggf. $v =$ *NIL*.

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.

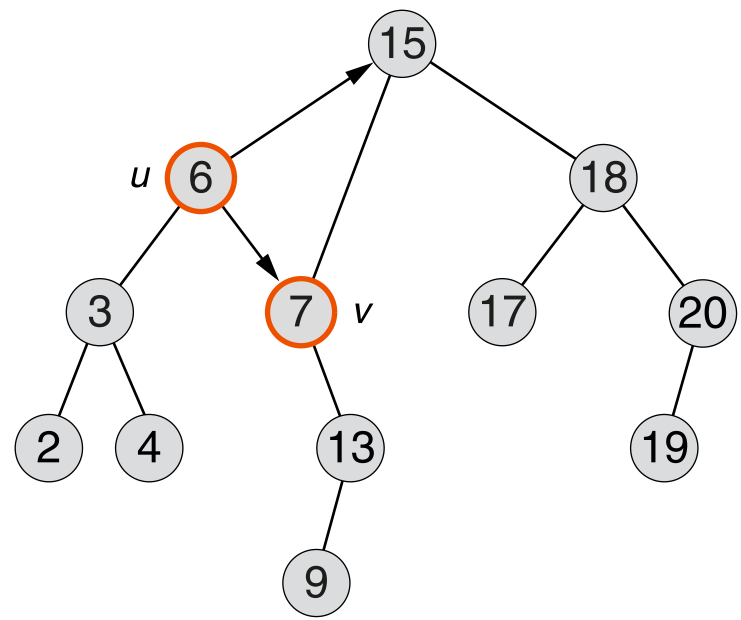
Eingabe: T. Binary Search Tree.
u, v. Wurzeln von Teilbäumen von T; ggf. $v =$ *NIL*.

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 $v \neq$ *NIL* **THEN**
  $v.$*parent* $= u.$*parent*
ENDIF

Laufzeit:

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

→ $O(1)$ Gesamtlaufzeit.

Bemerkungen:

- In der Literatur wird alternativ auch die Möglichkeit aufgezeigt, einfach die Schlüssel des zu löschenden Knotens und seines Nachfolgers *auszutauschen*.
- Der Vorteil der vorliegenden Methode besteht darin, dass der zu löschende Knoten nicht berührt wird und intakt im Speicher verbleibt, so dass Referenzen anderer Prozesse auf den Knoten valide bleiben.

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 in seinem rechten Teilbaum.
   Ersetze y durch sein rechtes Kind und mache y zur Wurzel von zs rechtem Teilbaum.
   Ersetze z durch y.

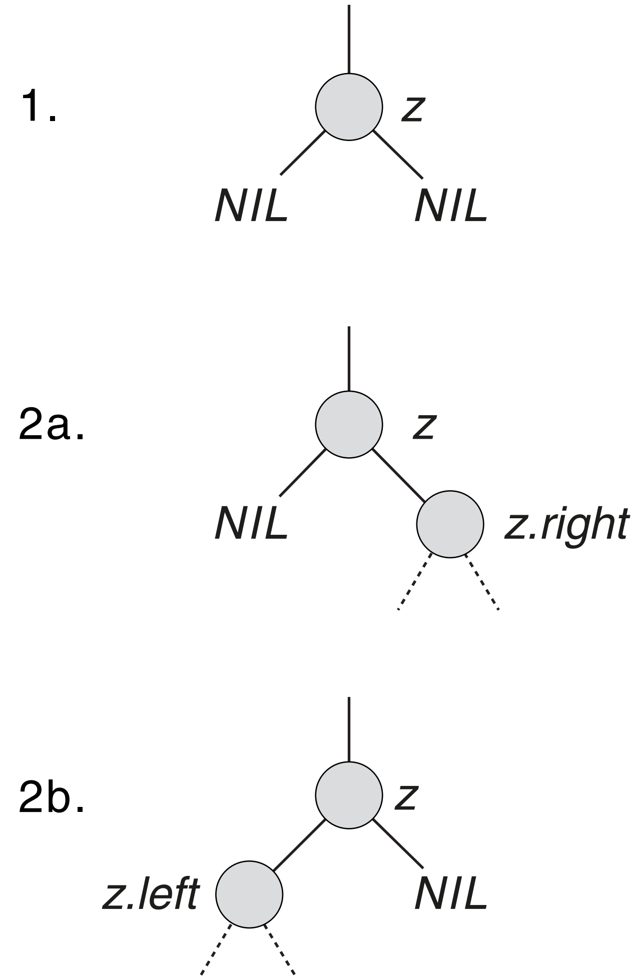
Binary Search Tree

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:



Binary Search Tree

Manipulation: Löschen

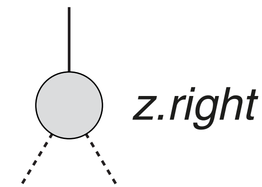
TreeDelete(T, z)

```
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
```

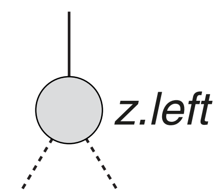
Fälle:

1.

2a.



2b.



Binary Search Tree

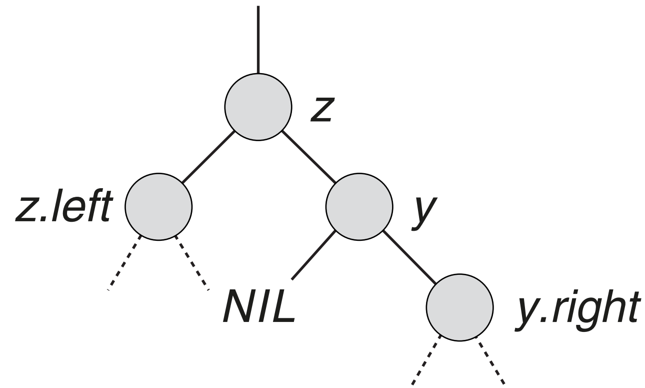
Manipulation: Löschen

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

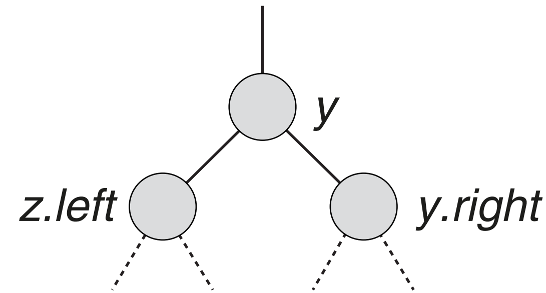
Manipulation: Löschen

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* $\neq z$ **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.



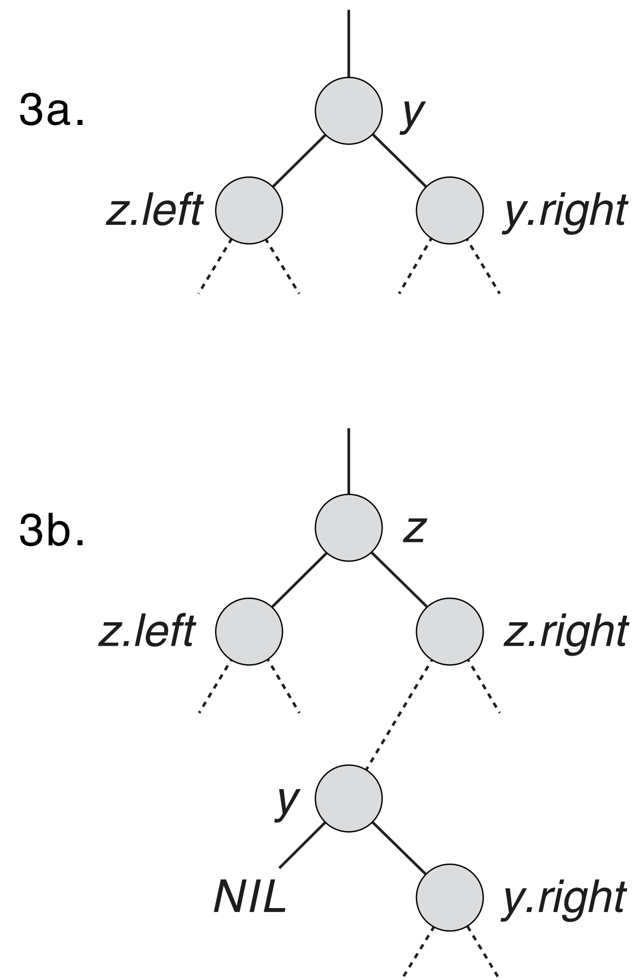
Binary Search Tree

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:



Binary Search Tree

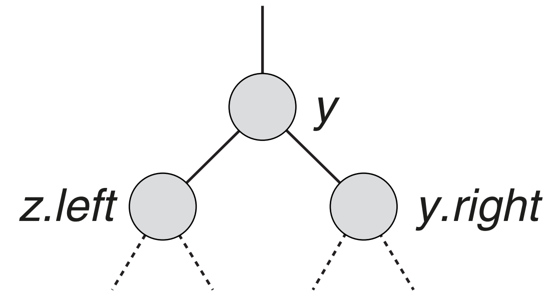
Manipulation: Löschen

TreeDelete(T, z)

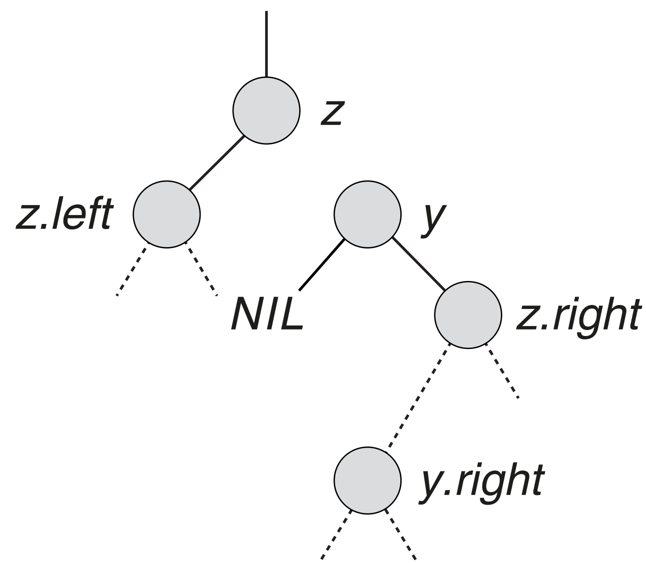
```
IF z.left == NIL THEN
   Transplant(T, z, z.right)
ELSE IF z.right == NIL THEN
   Transplant(T, z, z.left)
ELSE
   y = TreeMinimum(z.right)
   IF y.parent ≠ z THEN
      Transplant(T, y, y.right)
      y.right = z.right
      y.right.parent = y
   ENDIF
   Transplant(T, z, y)
   y.left = z.left
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

Fälle:

3a.



3b.



Binary Search Tree

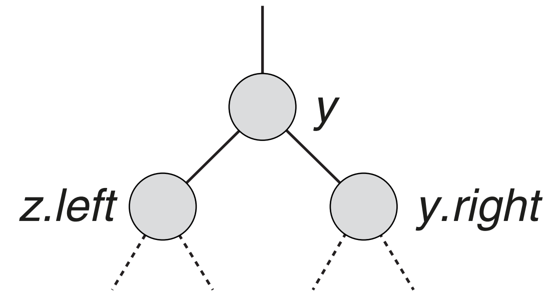
Manipulation: Löschen

TreeDelete(T, z)

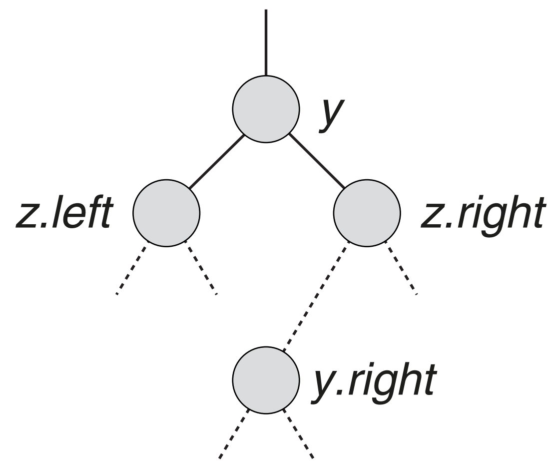
```
IF z.left == NIL THEN
   Transplant(T, z, z.right)
ELSE IF z.right == NIL THEN
   Transplant(T, z, z.left)
ELSE
   y = TreeMinimum(z.right)
   IF y.parent ≠ z THEN
      Transplant(T, y, y.right)
      y.right = z.right
      y.right.parent = y
   ENDIF
   Transplant(T, z, y)
   y.left = z.left
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

Binary Search Tree

Laufzeit

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

Best Case:

- Balancierter Baum.

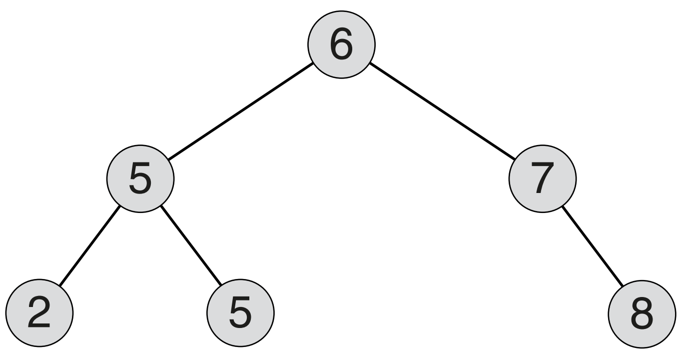
➜ $h \approx \lg n \;\rightarrow\; T(n) = O(\lg n)$.

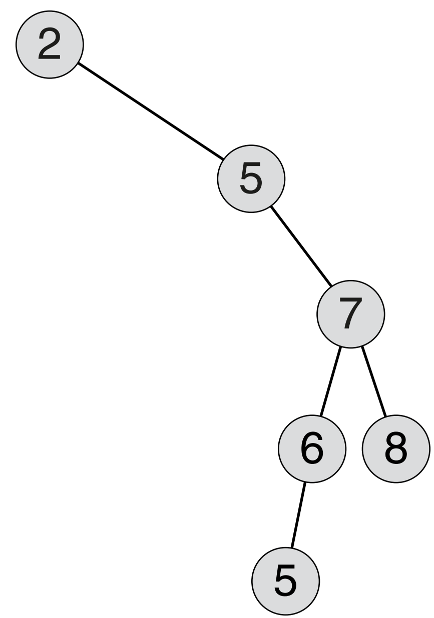
Worst Case:

- Alle inneren Knoten vom Grad 1.

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

Beispiel:





Binary Search Tree

Laufzeit (Fortsetzung)

Average Case:

- ❑ Zufälliger Binary Search Tree mit n verschiedenen Knoten.

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

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 inneren 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.

AVL Tree

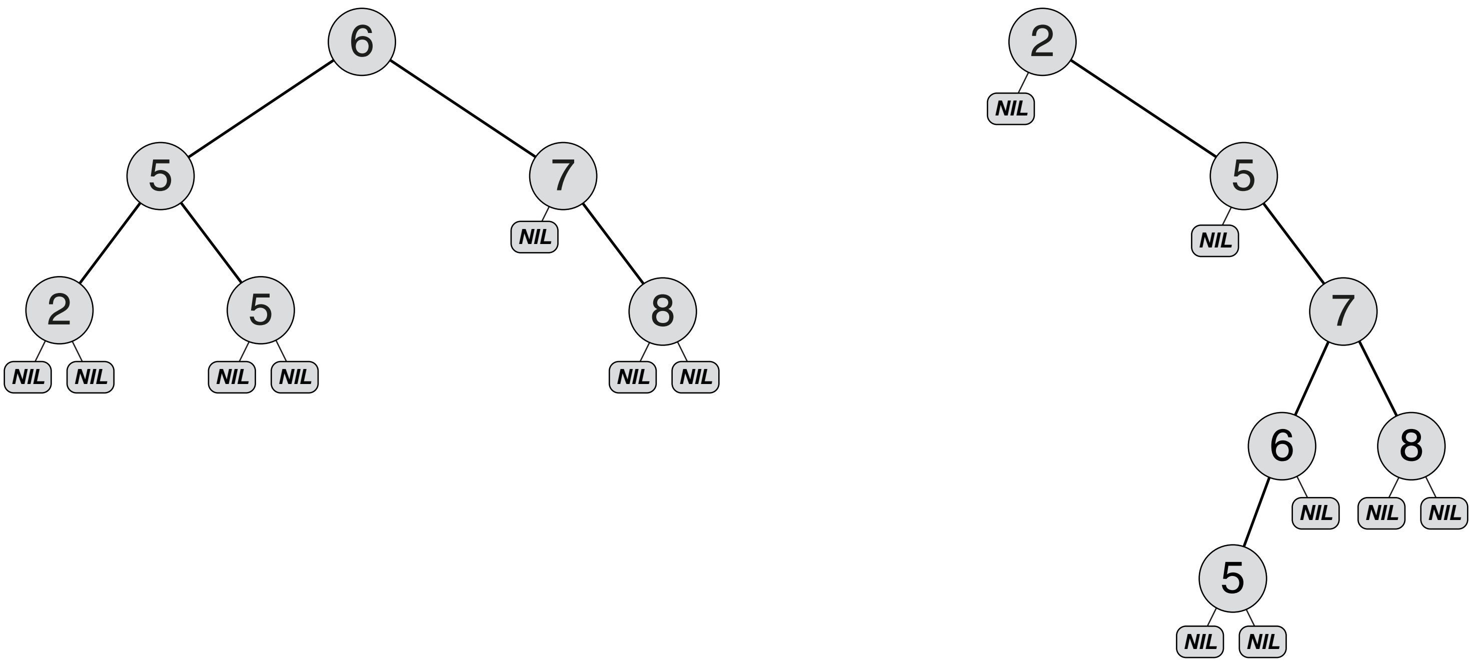
Definition

Ein Binary Search Tree ist ein AVL Tree, wenn für jeden inneren 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

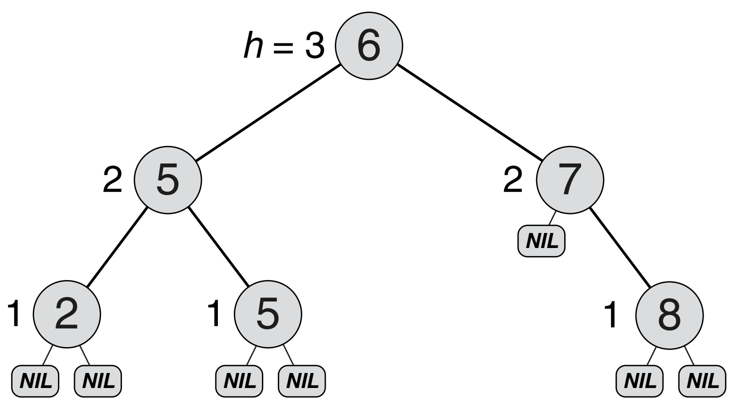
Definition

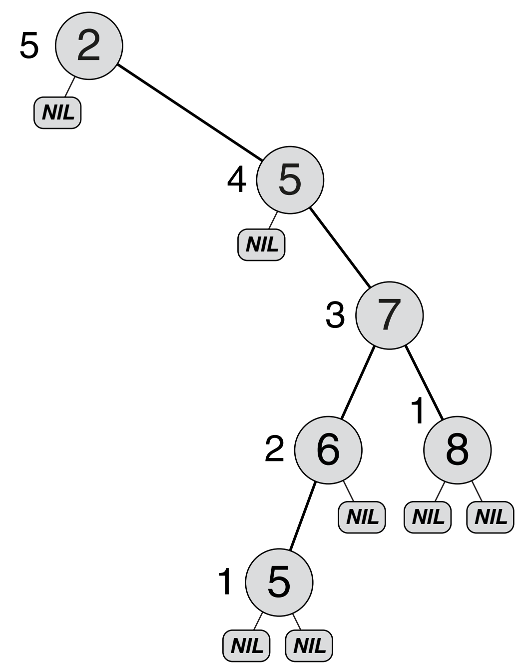
Ein Binary Search Tree ist ein AVL Tree, wenn für jeden inneren Knoten x gilt

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

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

Beispiele:





AVL Tree

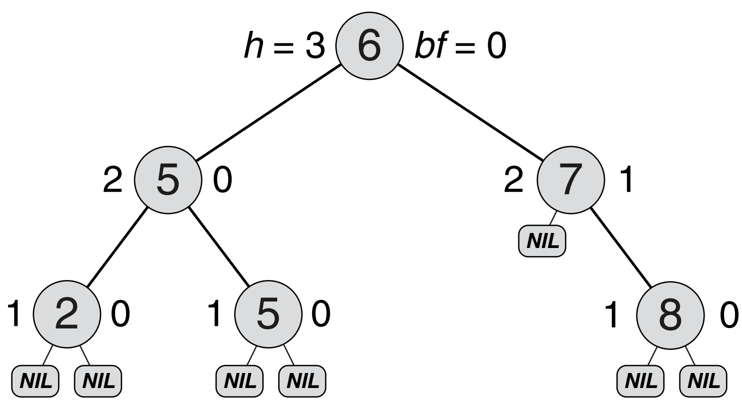
Definition

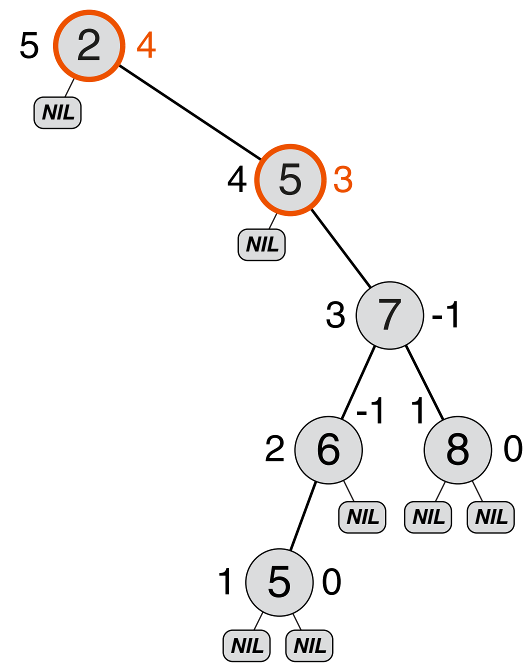
Ein Binary Search Tree ist ein AVL Tree, wenn für jeden inneren Knoten x gilt

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

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

Beispiele:





Bemerkungen:

- Der AVL-Baum ist benannt nach den sowjetischen Mathematikern Georgi Maximowitsch Adel'son-Vel'skiĭ und Jewgeni Michailowitsch Landis, die die Datenstruktur im Jahr 1962 vorstellten [Adel'son-Vel'skiĭ and Landis 1962] (aus dem Russischen).
- Es werden knotenorientierte binäre Suchbäume von blattorientierten unterschieden: Wenn die Inhalte der Menge [zu speichernder Elemente] in den Knoten abgespeichert werden und die externen Knoten leer sind, nennt man die Art der Speicherung knotenorientiert. Um auszudrücken, dass sie nicht zur Menge gehören, bezeichnet man in diesem Fall die externen Knoten zur besseren Unterscheidung als externe Blätter. Ein externes Blatt stellt einen Einfügepunkt dar. Bei der blattorientierten Speicherung sind die Inhalte der Menge in den Blättern abgespeichert, und die Knoten stellen nur Hinweisschilder für die Navigation dar, die möglicherweise mit den Schlüsseln der Menge wenig zu tun haben. [Wikipedia]
- Leider wird die jeweils verwendete Betrachtung oft nicht explizit genannt oder sogar vermischt verwendet, was zu großer Verwirrung führen kann. Hier verwenden wir die knotenorientierte Sicht. Der Übersicht halber blenden wir die leeren externen Blätter (die *NIL*-Knoten) in den meisten Fällen jedoch aus.

AVL Tree

Satz 1

Ein AVL Tree mit n Knoten hat eine Höhe $h \leq 1.44 \cdot \lg n + 1 \ = \ O(\lg n)$.

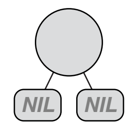
AVL Tree

Satz 1

Ein AVL Tree mit n Knoten hat eine Höhe $h \leq 1.44 \cdot \lg n + 1 \; = \; O(\lg n)$.

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

Beispiel:



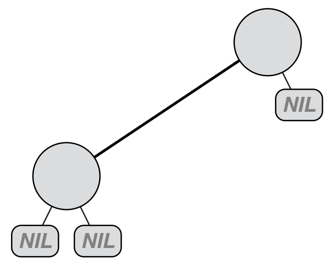
AVL Tree

Satz 1

Ein AVL Tree mit n Knoten hat eine Höhe $h \leq 1.44 \cdot \lg n + 1 \;=\; O(\lg n)$.

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

Beispiel:



- Für alle Knoten x, die ein Nicht-Blatt als Kind haben, gilt: $bf(x) \neq 0$.

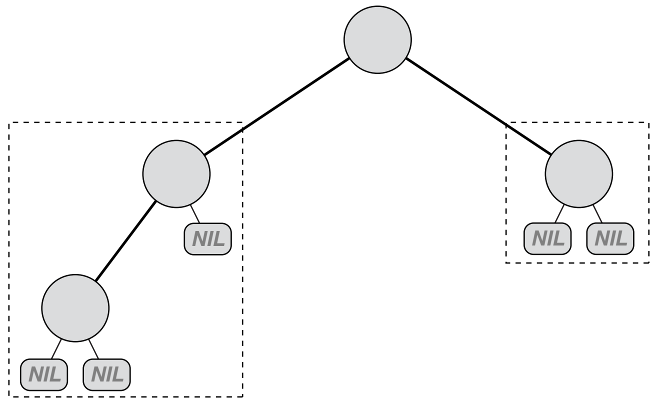
AVL Tree

Satz 1

Ein AVL Tree mit n Knoten hat eine Höhe $h \leq 1.44 \cdot \lg n + 1 \; = \; O(\lg n)$.

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

Beispiel:



- Für alle Knoten x, die ein Nicht-Blatt als Kind haben, gilt: $bf(x) \neq 0$.
- Jedes Kind der Wurzel eines minimalen AVL Trees ist ein minimaler AVL Tree.

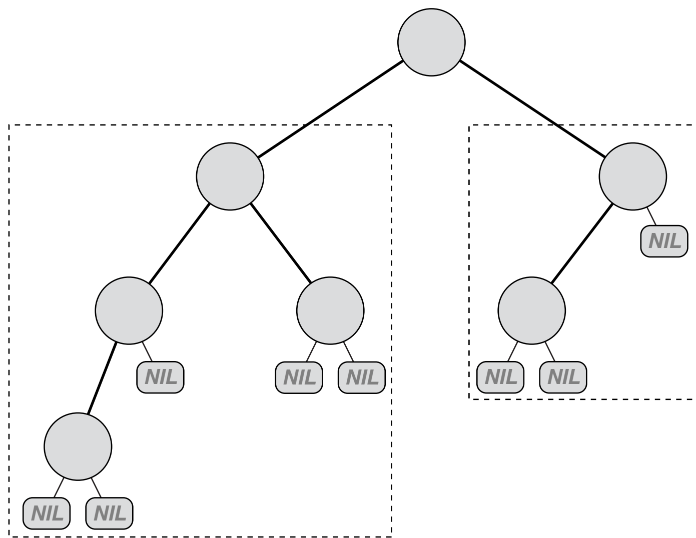
AVL Tree

Satz 1

Ein AVL Tree mit n Knoten hat eine Höhe $h \leq 1.44 \cdot \lg n + 1 \; = \; O(\lg n)$.

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

Beispiel:



- Für alle Knoten x, die ein Nicht-Blatt als Kind haben, gilt: $bf(x) \neq 0$.
- Jedes Kind der Wurzel eines minimalen AVL Trees ist ein minimaler AVL Tree.

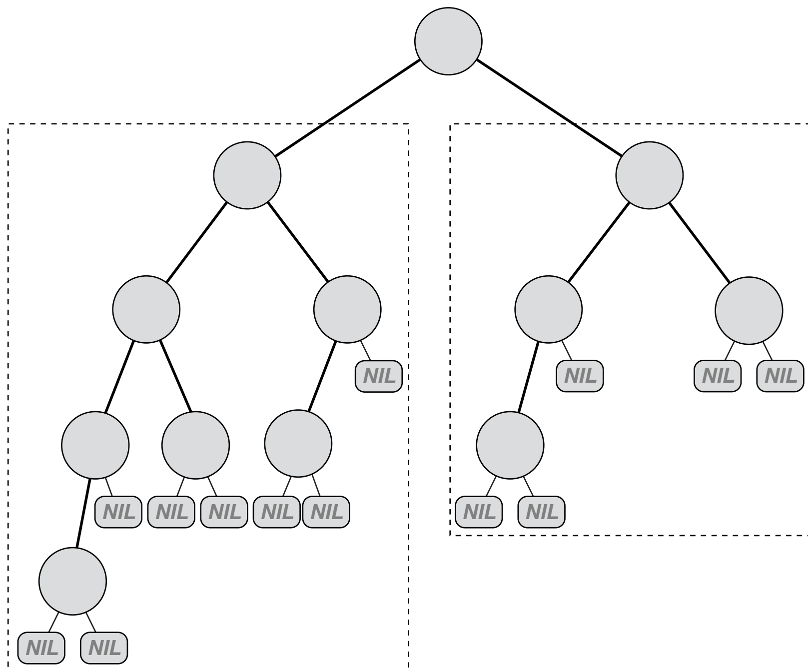
AVL Tree

Satz 1

Ein AVL Tree mit n Knoten hat eine Höhe $h \leq 1.44 \cdot \lg n + 1 \;=\; O(\lg n)$.

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

Beispiel:



- Für alle Knoten x, die ein Nicht-Blatt als Kind haben, gilt: $bf(x) \neq 0$.
- Jedes Kind der Wurzel eines minimalen AVL Trees ist ein minimaler AVL Tree.

AVL Tree

Satz 1

Ein AVL Tree mit n Knoten hat eine Höhe $h \leq 1.44 \cdot \lg n + 1 \;=\; O(\lg n)$.

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

➜ Die Zahl der externen Blätter eines minimalen AVL Trees der Höhe h entspricht der Fibonacci-Zahl F_{h+2} und die Zahl der inneren Knoten $F_{h+2} - 1$.

AVL Tree

Satz 1

Ein AVL Tree mit n Knoten hat eine Höhe $h \leq 1.44 \cdot \lg n + 1 \;=\; O(\lg n)$.

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

➜ Die Zahl der externen Blätter eines minimalen AVL Trees der Höhe h entspricht der Fibonacci-Zahl F_{h+2} und die Zahl der inneren Knoten $F_{h+2} - 1$.

Nach Formel von Moivre-Binet berechnet sich die h-te Fibonacci-Zahl F_h wie folgt:

$$F_h = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^h - \left(\frac{1-\sqrt{5}}{2} \right)^h \right) \approx 0.4472... \cdot (1.618...)^h,$$

AVL Tree

Satz 1

Ein AVL Tree mit n Knoten hat eine Höhe $h \leq 1.44 \cdot \lg n + 1 \;=\; O(\lg n)$.

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

➜ Die Zahl der externen Blätter eines minimalen AVL Trees der Höhe h entspricht der Fibonacci-Zahl F_{h+2} und die Zahl der inneren Knoten $F_{h+2} - 1$.

Nach Formel von Moivre-Binet berechnet sich die h-te Fibonacci-Zahl F_h wie folgt:

$$F_h = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^h - \left(\frac{1-\sqrt{5}}{2}\right)^h\right) \approx 0.4472... \cdot (1.618...)^h,$$

Sei $n(h)$ die Zahl der externen Blätter eines minimalen AVL Trees der Höhe h:

$$n(h) \geq F_{h+2} \approx 0.4472 \cdot 1.618^{h+2} = 1.171 \cdot 1.618^h$$

AVL Tree

Satz 1

Ein AVL Tree mit n Knoten hat eine Höhe $h \leq 1.44 \cdot \lg n + 1 \;=\; O(\lg n)$.

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

➜ Die Zahl der externen Blätter eines minimalen AVL Trees der Höhe h entspricht der Fibonacci-Zahl F_{h+2} und die Zahl der inneren Knoten $F_{h+2} - 1$.

Nach Formel von Moivre-Binet berechnet sich die h-te Fibonacci-Zahl F_h wie folgt:

$$F_h = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^h - \left(\frac{1-\sqrt{5}}{2} \right)^h \right) \approx 0.4472... \cdot (1.618...)^h,$$

Sei $n(h)$ die Zahl der externen Blätter eines minimalen AVL Trees der Höhe h:

$$n(h) \geq F_{h+2} \approx 0.4472 \cdot 1.618^{h+2} = 1.171 \cdot 1.618^h$$

Daraus folgt:

$$h \leq 1.44 \cdot \lg n + 1. \quad \square$$

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, größtem oder nächstgrößerem Sortierschlüssel bestimmen.

Laufzeit

- Traversierung ist in $O(n)$.
- Suchen, Minimum, Maxium und Nachfolger sind in $O(h)$, wobei h die Höhe des Binary Search Trees 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.

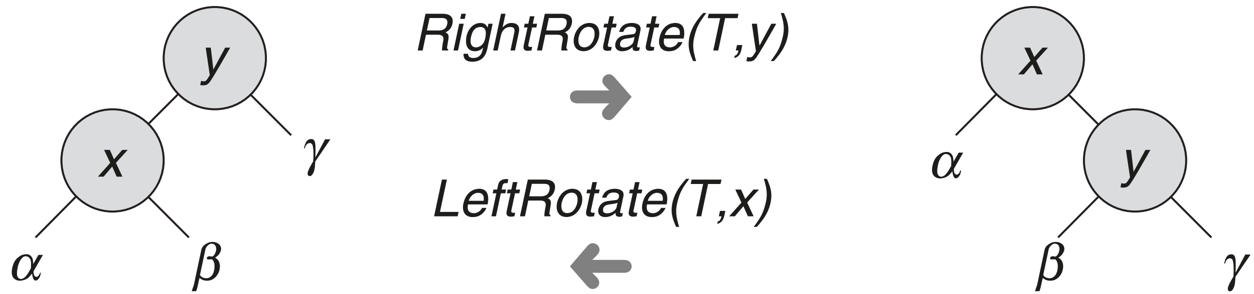
Jedes Einfügen oder Löschen eines Knotens kann Seiteneffekte haben:

- Einfügen / Löschen eines Knotens kann die Höhe seiner Vorfahren ändern.
- Das ändert ihren jeweiligen Balancefaktor und verletzt eventuell die AVL-Bedingung.

➜ Der AVL Tree muss gegebenenfalls rekonfiguriert werden.

AVL Tree

Manipulation: Rotation [Animation]



- 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.

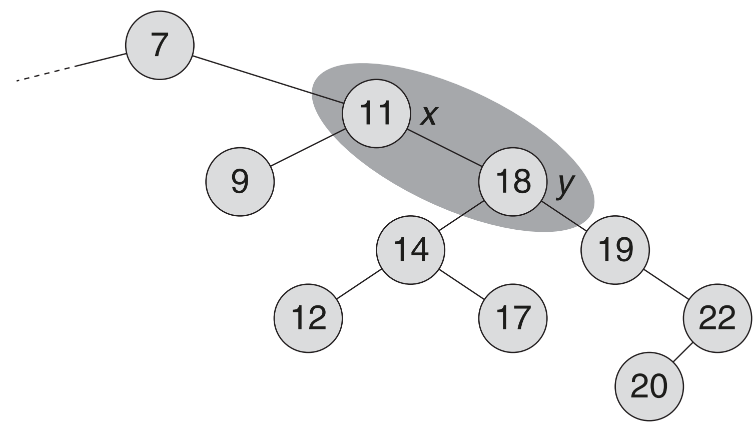
Eingabe: T. Binary Search Tree.
x. Wurzel eines Teilbaums von T, wobei $x.\mathit{right} \neq \mathit{NIL}$

Ausgabe: Rekonfigurierter Binary Search Tree.

LeftRotate(T, x)

```
y = x.right
x.right = y.left
IF y.left ≠ NIL THEN
   y.left.parent = x
ENDIF
y.parent = x.parent
IF x.parent == 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.

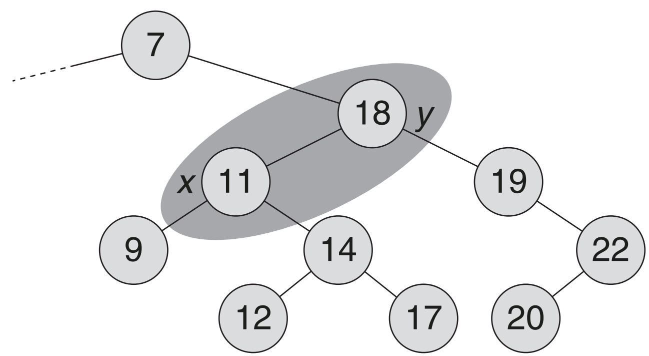
Eingabe: T. Binary Search Tree.
x. Wurzel eines Teilbaums von T, wobei x.*right* $\neq$ *NIL*

Ausgabe: Rekonfigurierter Binary Search Tree.

LeftRotate(T, x)

```
y = x.right
x.right = y.left
IF y.left ≠ NIL THEN
   y.left.parent = x
ENDIF
y.parent = x.parent
IF x.parent == 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*$(T, z.$*parent*$)$

AVL Tree

Manipulation: Einfügen

Algorithmus: AVL Insert.

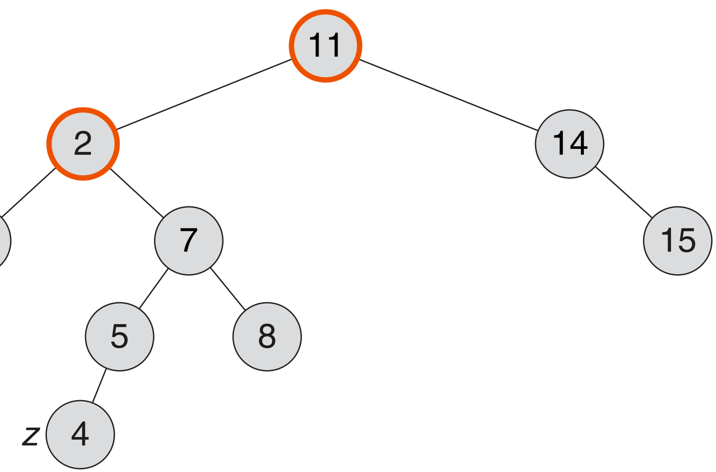
Eingabe: T. AVL Tree.
z. Einzufügender Knoten mit Schlüssel k.

Ausgabe: Um z erweiterter AVL Tree.

AVLInsert(T, z)

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

Beispiel:



AVL Tree
Manipulation: Einfügen

Algorithmus:	AVL Insert Fixup.
Eingabe:	T. Potentiell invalider AVL Tree. x. Elter des eingefügten Knotens z.
Ausgabe:	Valider AVL Tree.

AVL Tree

Manipulation: Einfügen [Löschen]

Algorithmus: AVL Insert Fixup.

Eingabe: T. Potentiell invalider AVL Tree.
x. Elter des eingefügten Knotens z.

Ausgabe: Valider AVL Tree.

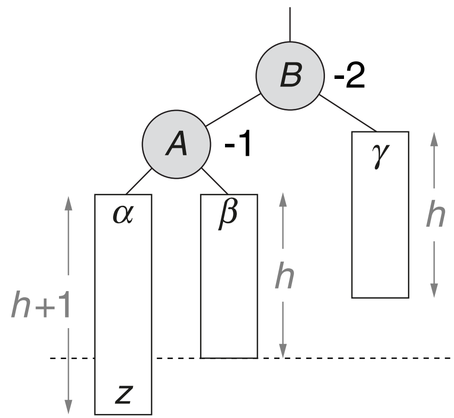
AVLInsertFixup(T, x):

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

AVL Tree

Manipulation: Einfügen

Fall 3a:

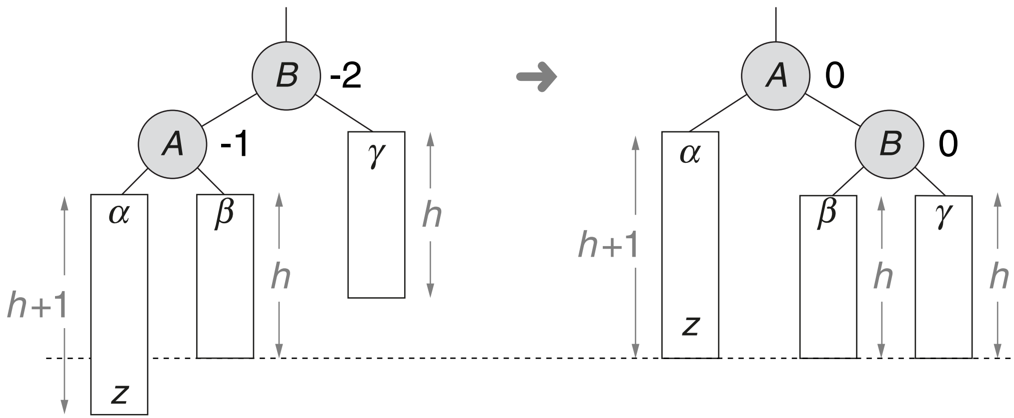


- Wenn Knoten B nach Einfügen von Knoten z den Balancefaktor -2 aufweist
  - war er zuvor unbalanciert mit Faktor -1.
  - war sein linkes Kind A zuvor balanciert.

AVL Tree

Manipulation: Einfügen

Fall 3a:



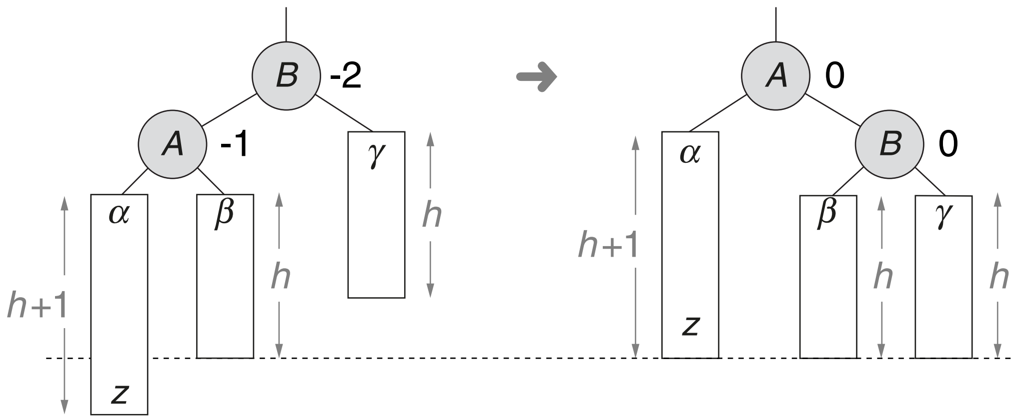
- Wenn Knoten B nach Einfügen von Knoten z den Balancefaktor -2 aufweist
  - war er zuvor unbalanciert mit Faktor -1.
  - war sein linkes Kind A zuvor balanciert.

➜ Wenn Knoten A Faktor -1 aufweist, stellt eine Rechtsrotation über B die Balance für A und B wieder her.

AVL Tree

Manipulation: Einfügen [Löschen]

Fall 3a:



- Wenn Knoten B nach Einfügen von Knoten z den Balancefaktor -2 aufweist
  - war er zuvor unbalanciert mit Faktor -1.
  - war sein linkes Kind A zuvor balanciert.

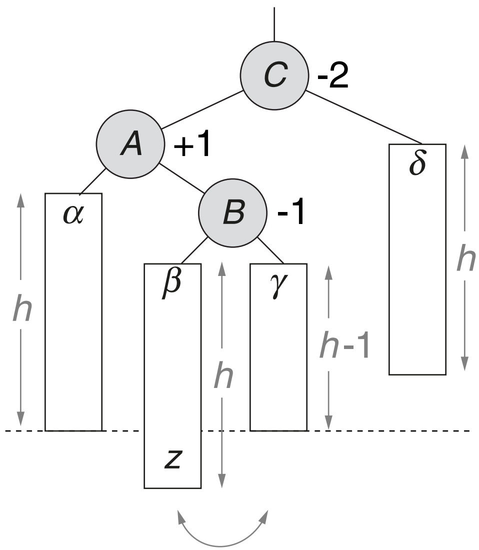
➜ Wenn Knoten A Faktor -1 aufweist, stellt eine Rechtsrotation über B die Balance für A und B wieder her.

➜ Die Tiefe des Teilbaums bleibt unverändert: Elterknoten bleiben balanciert.

AVL Tree

Manipulation: Einfügen

Fall 3b:

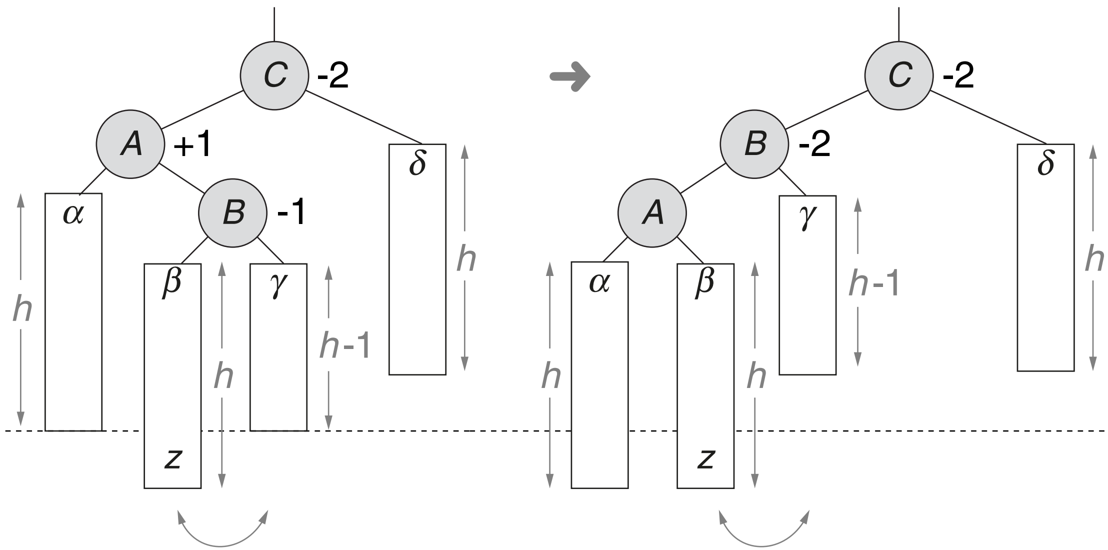


- Wenn Knoten A Faktor +1 aufweist, genügt eine einfache Rotation nicht.

AVL Tree

Manipulation: Einfügen

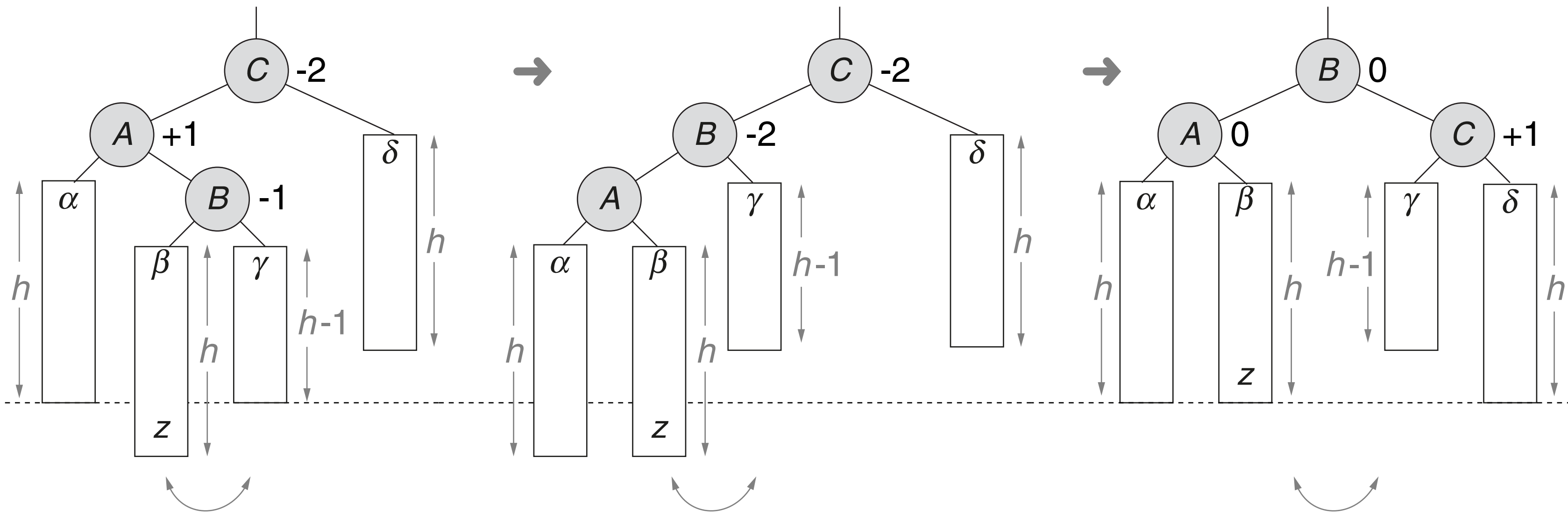
Fall 3b:



- Wenn Knoten A Faktor +1 aufweist, genügt eine einfache Rotation nicht.

➜ Durch eine vorherige Linksrotation über A, gelingt die Herstellung eines Zustands analog zu Fall 3a.

Fall 3b:



- ❑ Wenn Knoten A Faktor +1 aufweist, genügt eine einfache Rotation nicht.
- ➜ Durch eine vorherige Linksrotation über A, gelingt die Herstellung eines Zustands analog zu Fall 3a.
- ➜ Eine anschließende Rotation über Knoten C stellt die Balance wieder her.
  Dieses Vorgehen wird als „Doppelrotation“ bezeichnet.
- ➜ Die Tiefe des Teilbaums bleibt unverändert: Elterknoten bleiben balanciert.

Bemerkungen:

- Fälle 3c und d sind symmetrisch zu den Fällen 3a und b und können dementsprechend analog, durch Vertauschen von „Links“ und „Rechts“, gelöst werden.
- Nach dem Einfügen muss maximal eine (Doppel-)Rotation durchgeführt werden.
- Das Einfügen des $(n+1)$-ten Knotens in einen AVL-Baum mit n Knoten hat im Worst Case logarithmischen Aufwand, beispielsweise wenn jede Ebene bis hinauf zur Wurzel überprüft werden muss. Da aber hierfür die Wahrscheinlichkeit von Ebene zu Ebene nach oben hin exponentiell abnimmt, ist der reine Modifikationsaufwand (Ändern von Balance-Faktoren und Rotationen) beim Einfügen im Mittel konstant. [Wikipedia]

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

- Modifikation von *TreeInsert*

  Rückgabe des Elters des gelöschten Knotens z bzw. des vormaligen Elters des Nachfolgers y, durch den z ersetzt wird.

AVL Tree

Manipulation: Löschen

Algorithmus: AVL Delete.

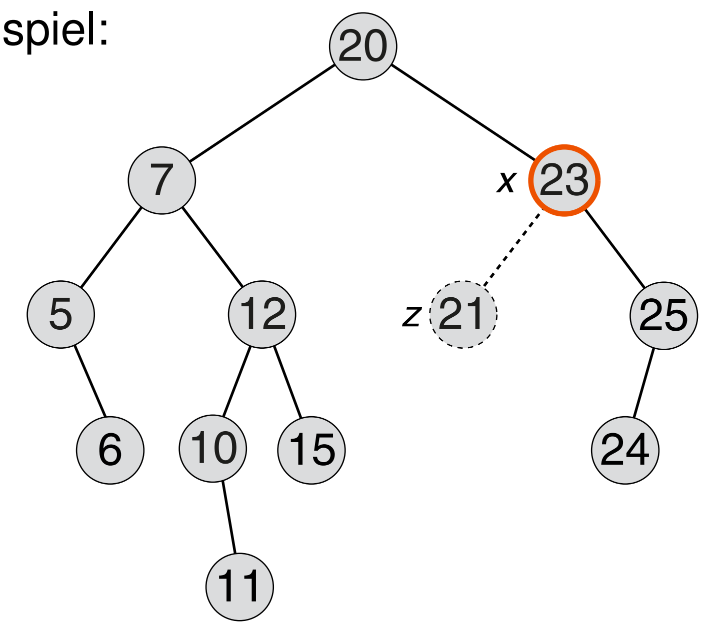
Eingabe: T. AVL Tree.
z. Zu löschender Knoten.

Ausgabe: AVL Tree, beim z gelöscht wurde.

AVLDelete(T, z)

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

Beispiel:



- Modifikation von *TreeInsert*
  Rückgabe des Elters des gelöschten Knotens z bzw. des vormaligen Elters des Nachfolgers y, durch den z ersetzt wird.

AVL Tree

Manipulation: Löschen [Löschen]

Algorithmus: AVL Delete Fixup.

Eingabe: T. Potentiell invalider AVL Tree.
x. Elterknoten des gelöschten Knotens z, bzw. seines Nachfolgers.

Ausgabe: Valider AVL Tree.

AVL Tree

Manipulation: Löschen [Einfügen]

Algorithmus: AVL Delete Fixup.

Eingabe: T. Potentiell invalider AVL Tree.
x. Elterknoten des gelöschten Knotens z, bzw. seines Nachfolgers.

Ausgabe: Valider AVL Tree.

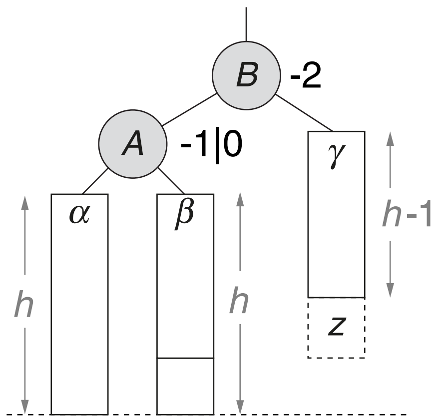
AVLDeleteFixup(T, x):

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

AVL Tree

Manipulation: Löschen

Fall 3a:

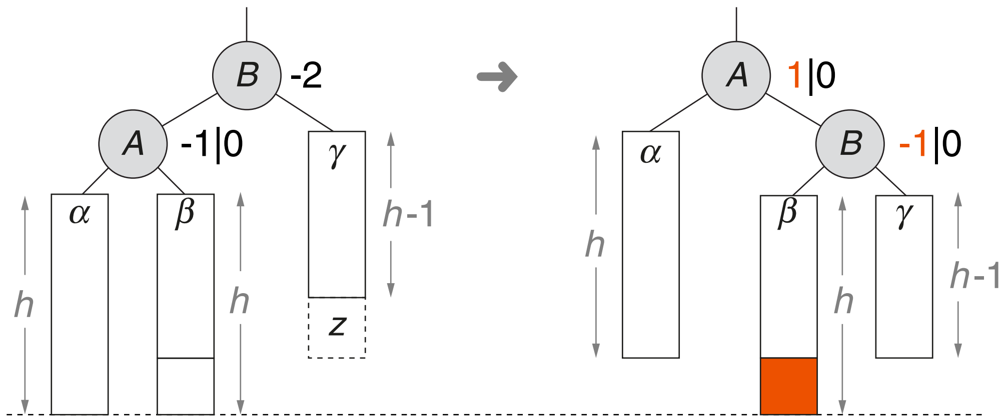


- Wenn Knoten B nach Löschen von Knoten z den Balancefaktor -2 aufweist
  - war er zuvor unbalanciert mit Faktor -1.
  - wurde aus dem rechten Teilbaum z entfernt.

AVL Tree

Manipulation: Löschen [Einfügen]

Fall 3a:



- Wenn Knoten B nach Löschen von Knoten z den Balancefaktor -2 aufweist
  - war er zuvor unbalanciert mit Faktor -1.
  - wurde aus dem rechten Teilbaum z entfernt.

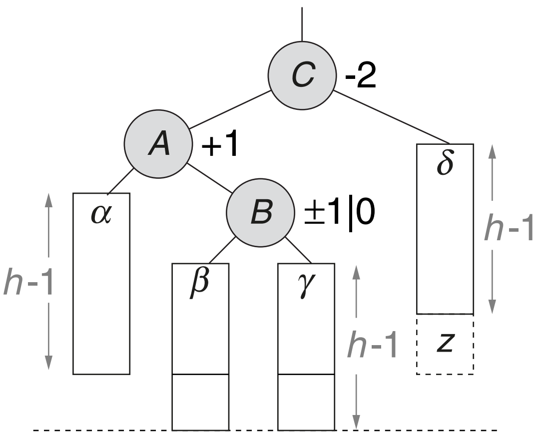
➜ Wenn Knoten A Faktor -1 oder 0 aufweist, stellt eine Rechtsrotation über B die Balance für A und B wieder her.

➜ Die Tiefe des Teilbaums hat sich eventuell verändert: Elterknoten prüfen.

AVL Tree

Manipulation: Löschen

Fall 3b:

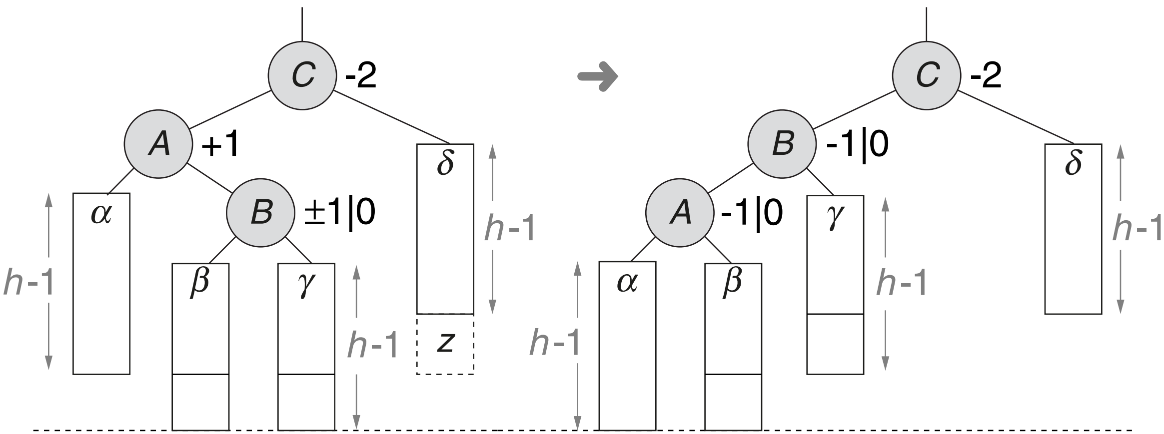


- Wenn Knoten A Faktor +1 aufweist, genügt eine einfache Rotation nicht.

AVL Tree

Manipulation: Löschen

Fall 3b:



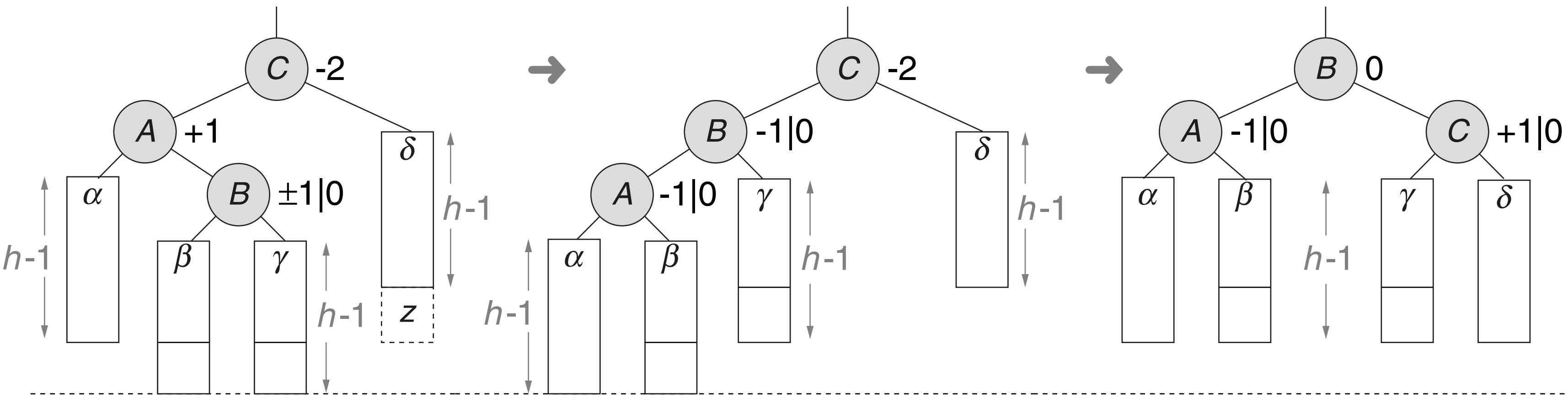
- Wenn Knoten A Faktor +1 aufweist, genügt eine einfache Rotation nicht.

➜ Durch eine vorherige Linksrotation über A, gelingt die Herstellung eines Zustands analog zu Fall 3a.

AVL Tree

Manipulation: Löschen [Einfügen]

Fall 3b:



- Wenn Knoten A Faktor +1 aufweist, genügt eine einfache Rotation nicht.

→ Durch eine vorherige Linksrotation über A, gelingt die Herstellung eines Zustands analog zu Fall 3a.

→ Eine anschließende Rotation über Knoten C stellt die Balance wieder her.
Dieses Vorgehen wird als „Doppelrotation“ bezeichnet.

→ Die Tiefe des Teilbaums hat sich verändert: Elterknoten prüfen.

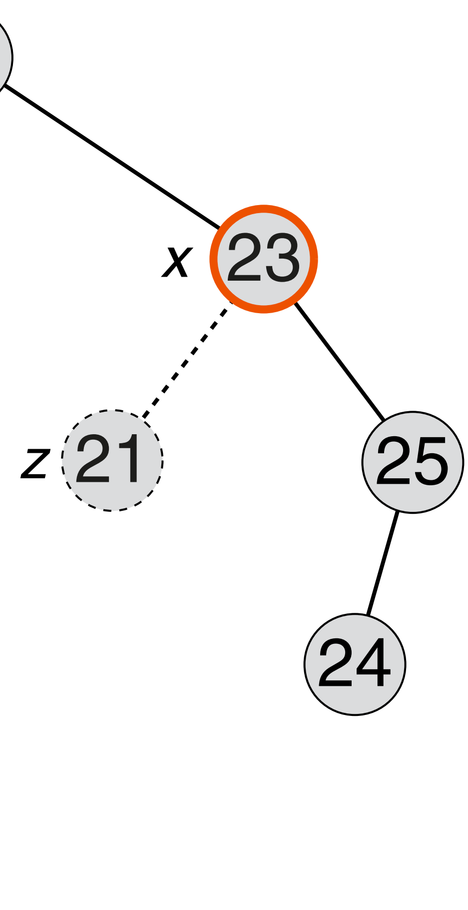
Bemerkungen:

- Fälle 3c und d sind symmetrisch zu den Fällen 3a und b und können dementsprechend analog, durch Vertauschen von „Links“ und „Rechts“, gelöst werden.
- Nach dem Löschen müssen $O(h)$ (Doppel-)Rotationen durchgeführt werden.
- Der Aufwand fürs Löschen ist im schlechtesten Fall logarithmisch; im Mittel aber ist er konstant, da die Wahrscheinlichkeit für die Notwendigkeit, die Balance auf der nächsthöheren Ebene überprüfen zu müssen, nach oben hin exponentiell abnimmt. [Wikipedia]

AVL Tree

Manipulation: Löschen

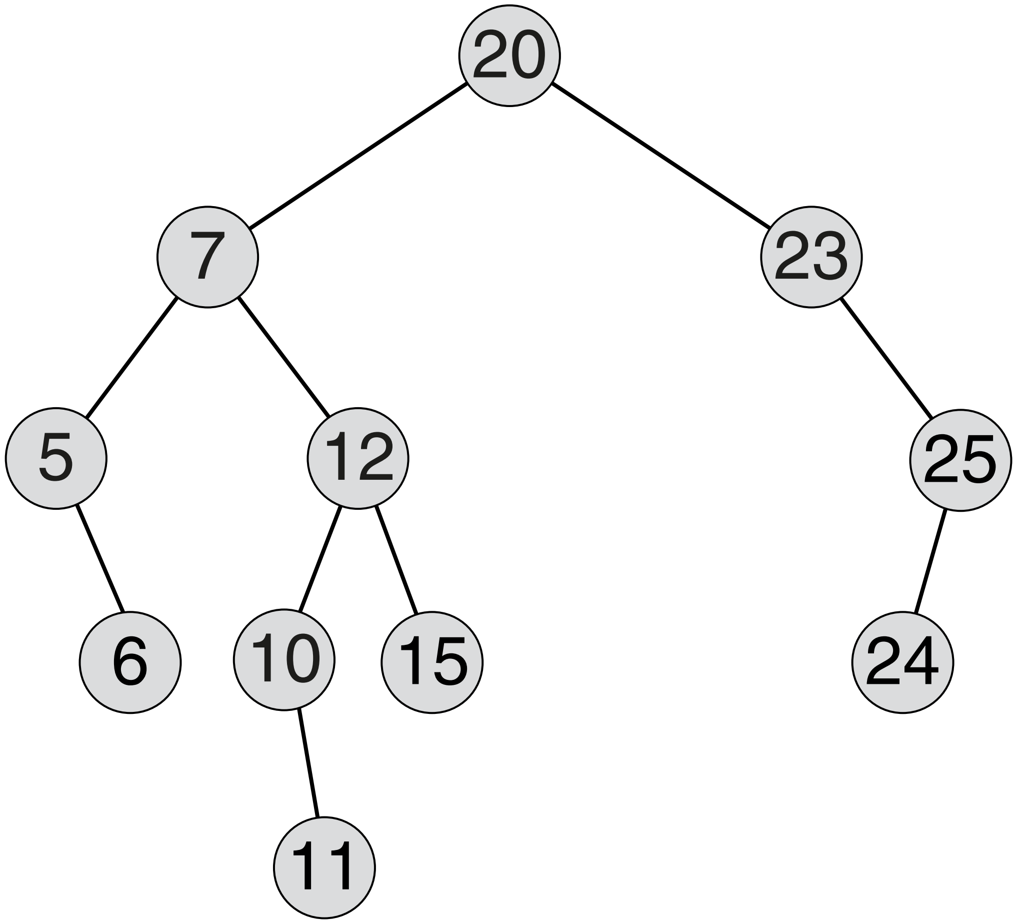
Beispiel:



AVL Tree

Manipulation: Löschen

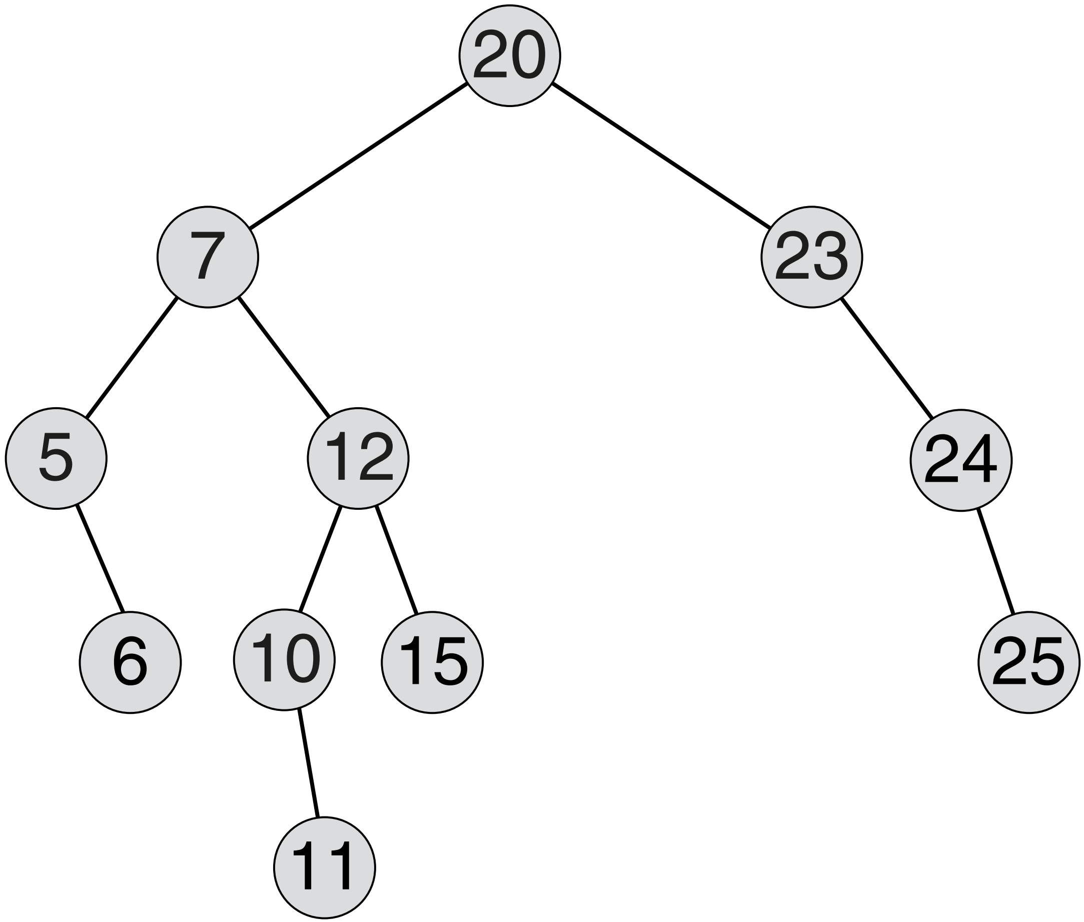
Beispiel:



AVL Tree

Manipulation: Löschen

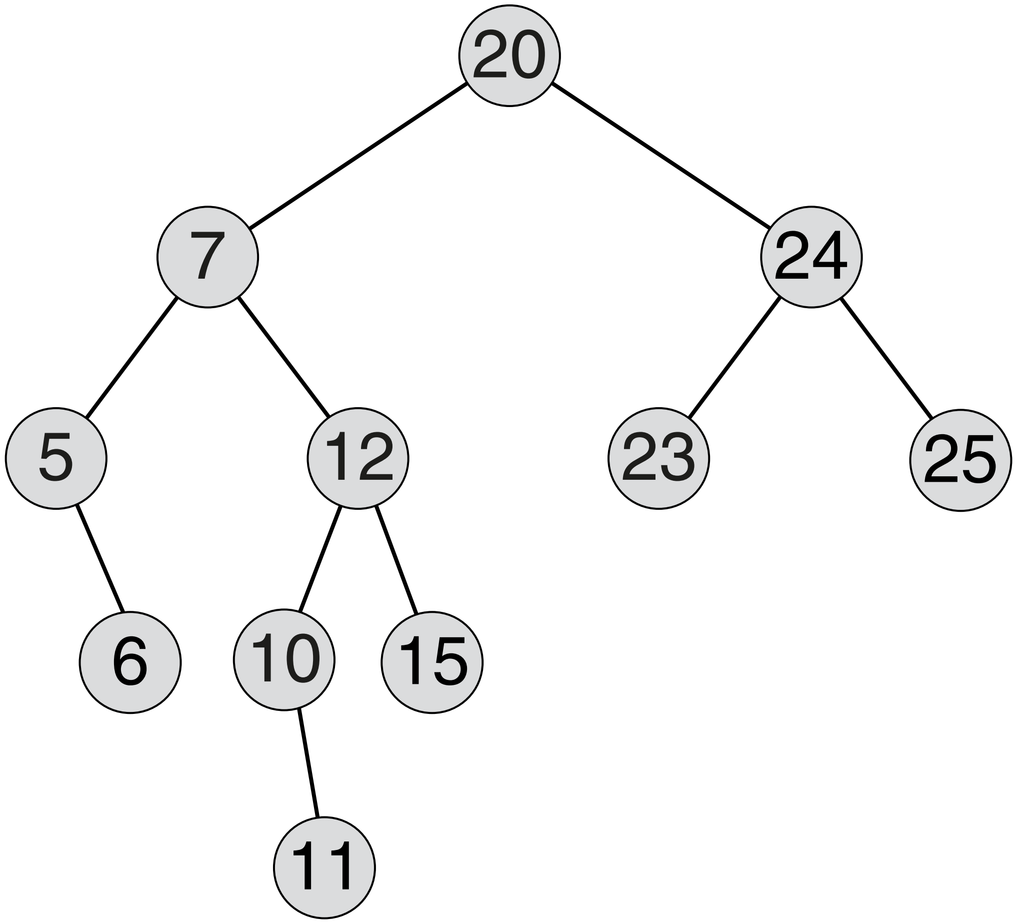
Beispiel:



AVL Tree

Manipulation: Löschen

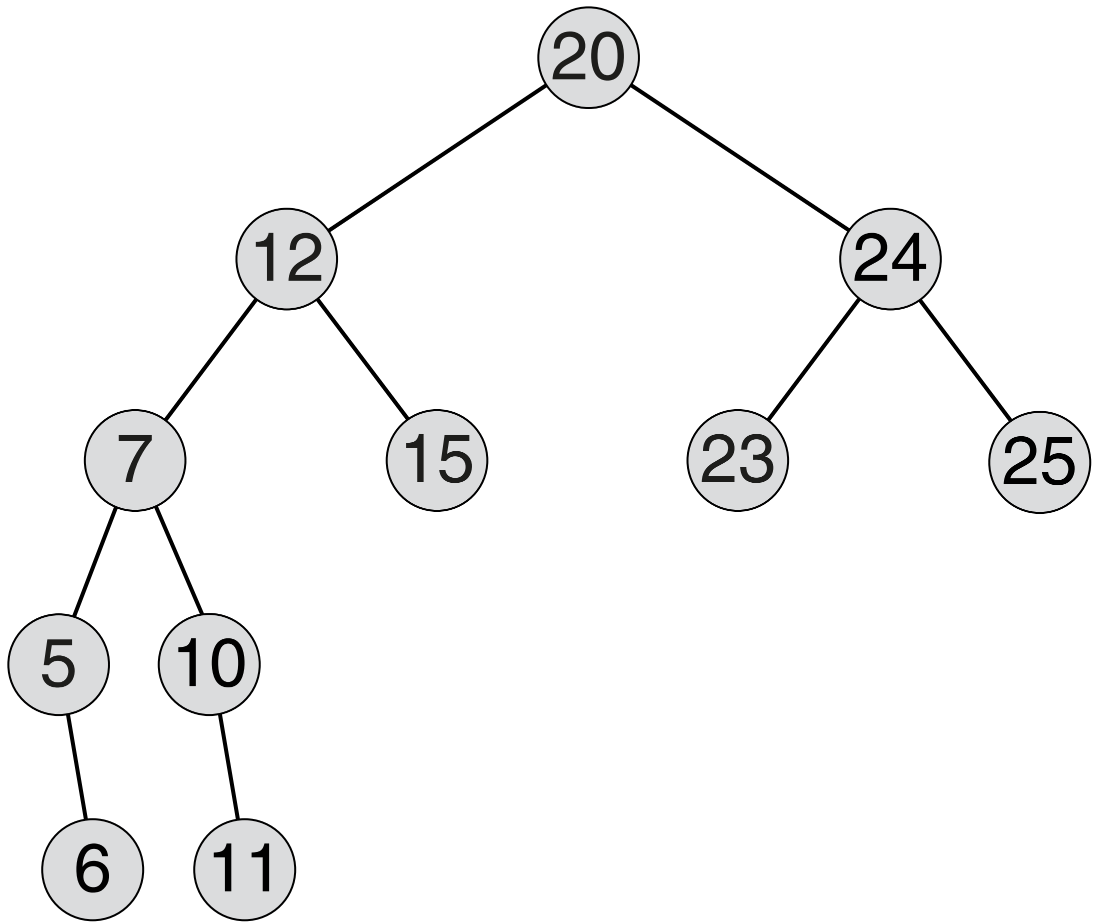
Beispiel:



AVL Tree

Manipulation: Löschen

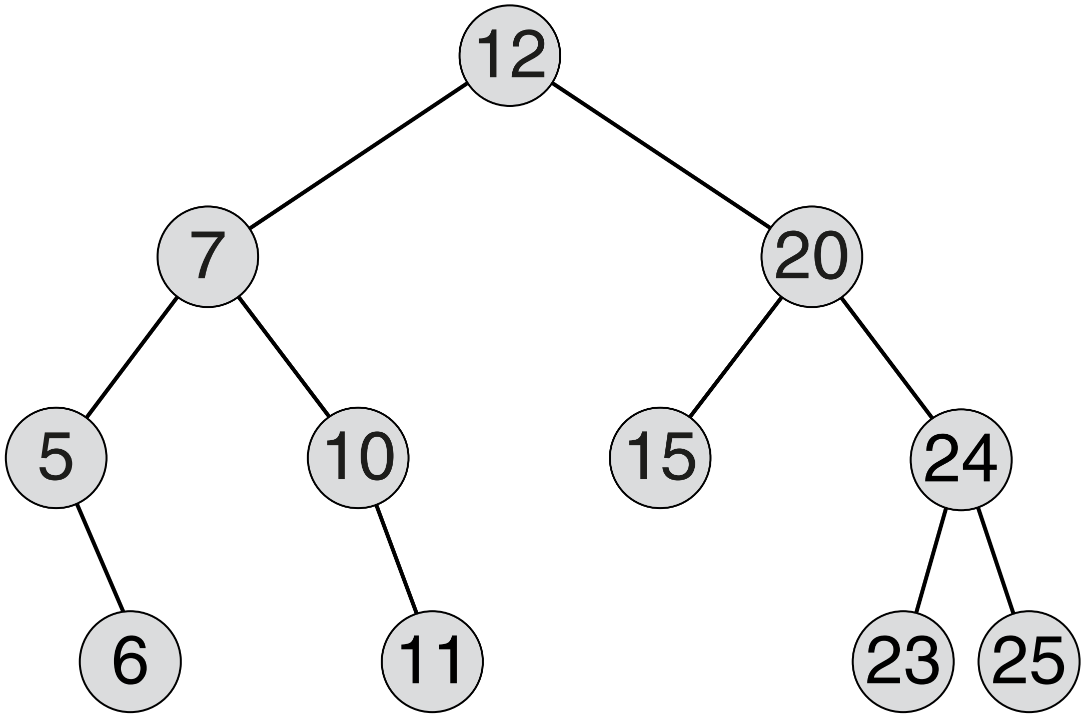
Beispiel:



AVL Tree

Manipulation: Löschen

Beispiel:



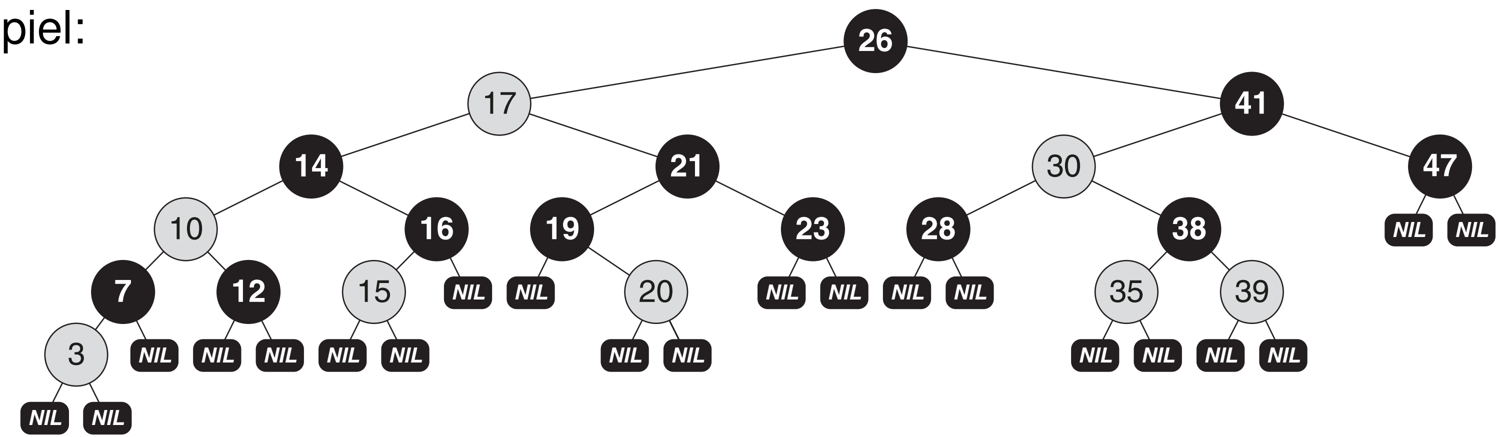
Red-Black Tree

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.

Beispiel:



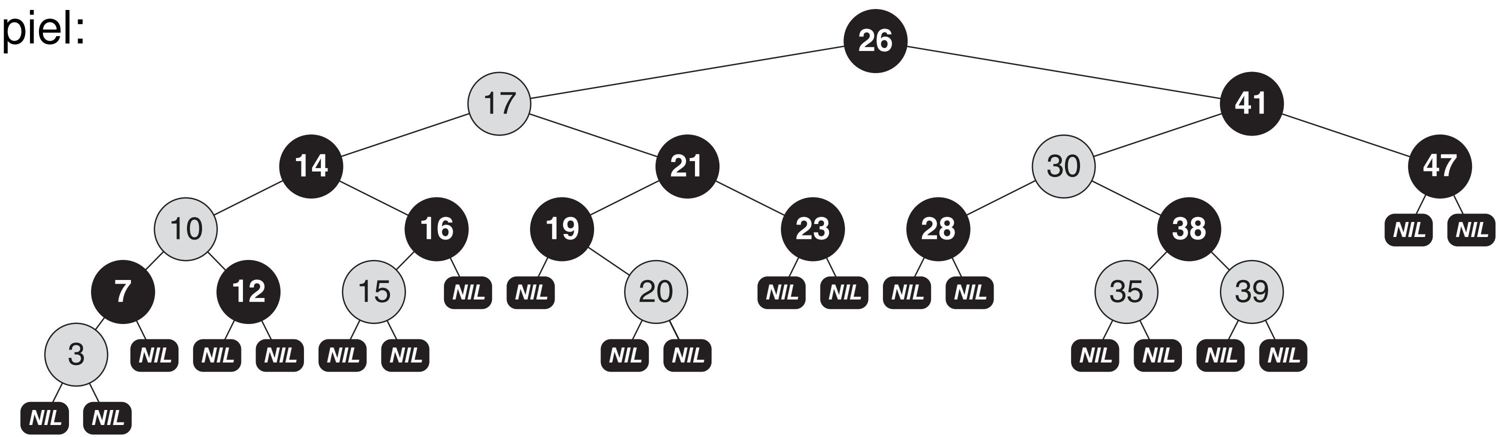
Red-Black Tree

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:



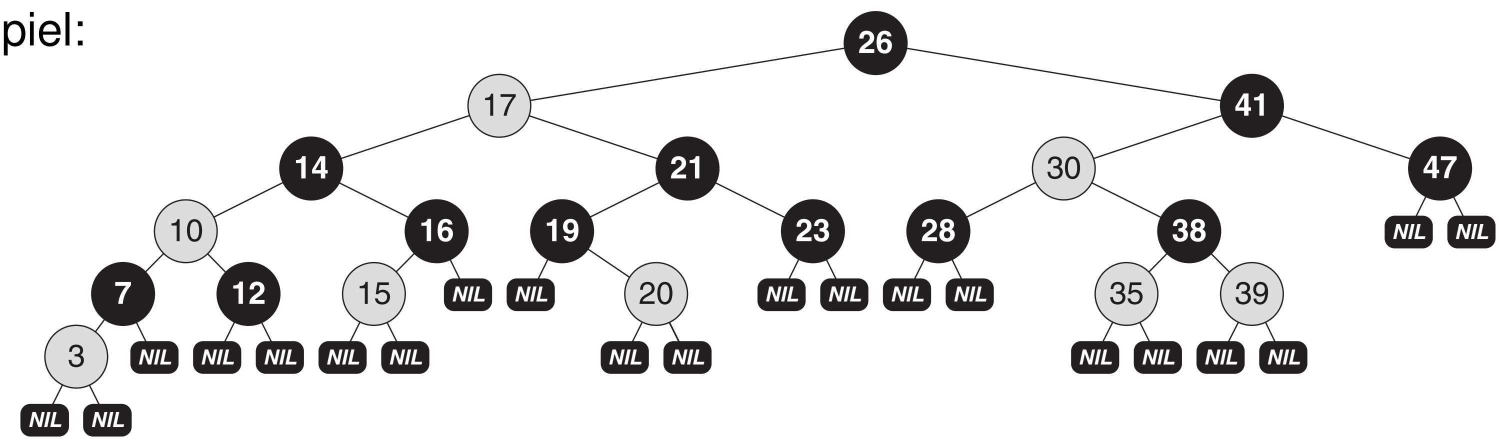
Red-Black Tree

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:

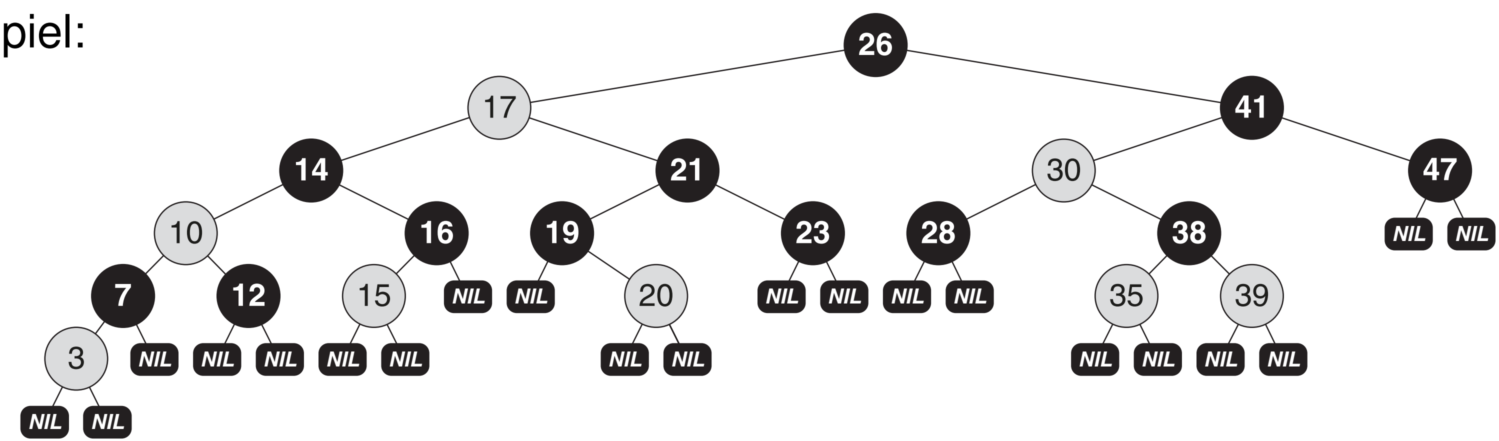


Red-Black Tree

Satz 2

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

Beispiel:



Red-Black Tree

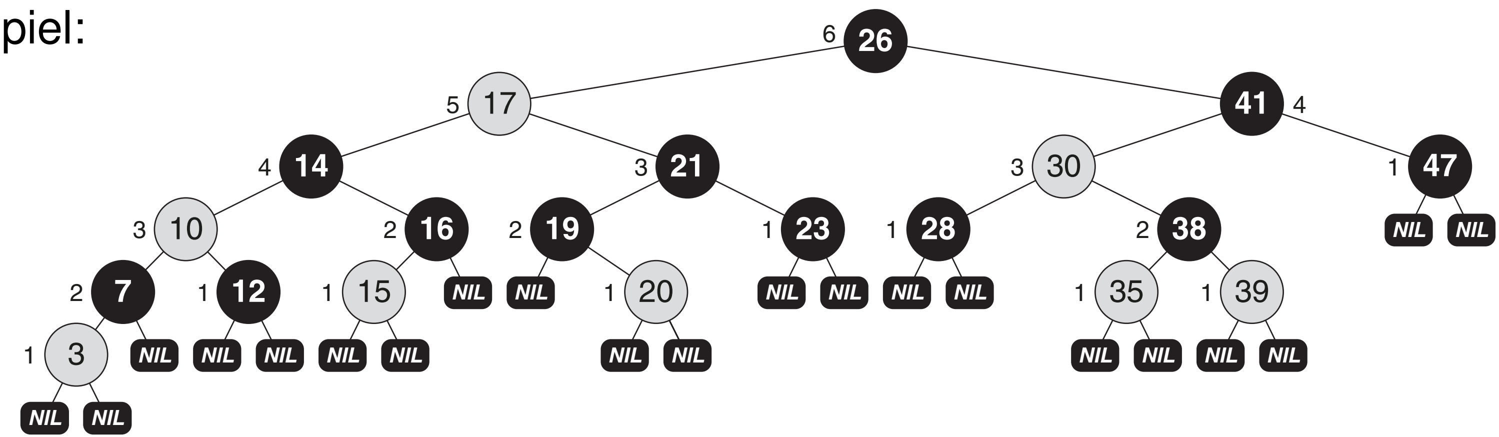
Satz 2

Ein Red-Black Tree mit n inneren 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

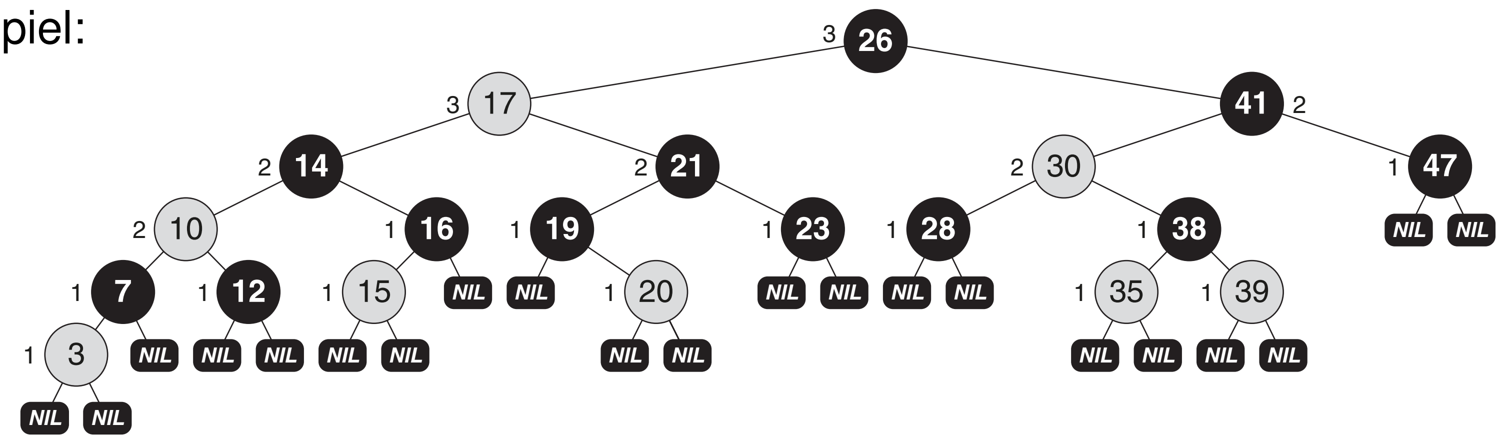
Ein Red-Black Tree mit n inneren Knoten hat eine Höhe $h \leq 2\lg(n+1) = O(\lg n)$.

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 auf dem längsten Pfad zu einem Blatt (exklusive x).

Beispiel:



Red-Black Tree

Satz 2

Ein Red-Black Tree mit n inneren 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 auf dem längsten Pfad 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 $\geq 2^{bh(x)} - 1$ innere Knoten.

Red-Black Tree

Satz 2

Ein Red-Black Tree mit n inneren 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 auf dem längsten Pfad 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 $\geq 2^{bh(x)} - 1$ innere 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 $\geq 2^{bh(x)} - 1$ innere 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 innerer 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 $\geq 2^{bh(x)} - 1$ innere 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 innerer Knoten des Teilbaums von x

Induktionsschritt:

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

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

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

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

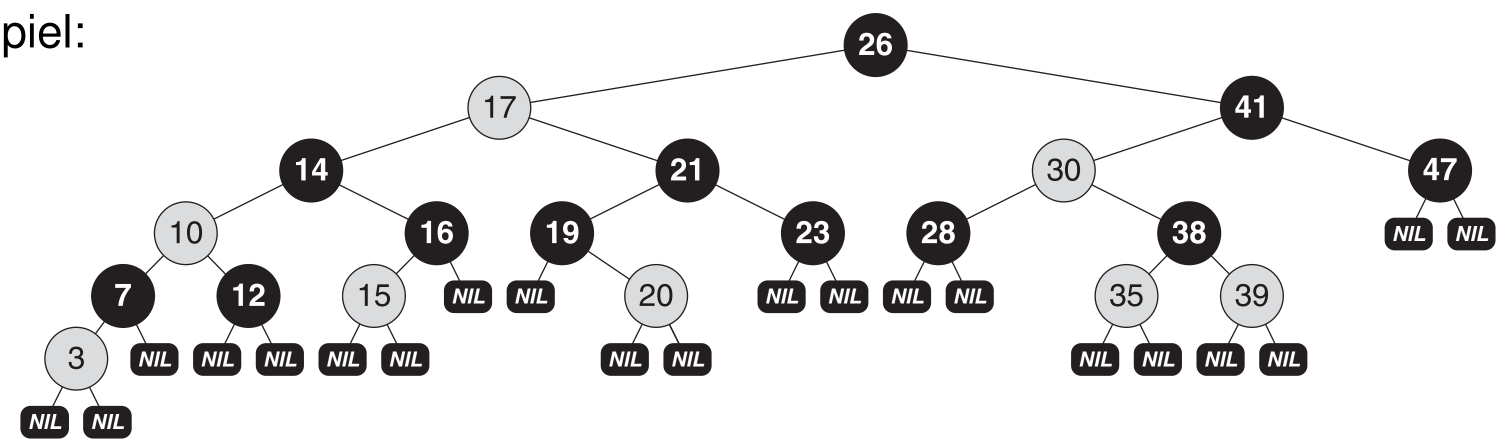
$\Rightarrow$ Zahl innerer Knoten des Teilbaums von x:
$\geq 2 \cdot (2^{bh(x)-1} - 1) + 1 \;=\; 2^{bh(x)} - 1 \quad \square$

Red-Black Tree

Implementierung

- Knotenorientierte Speicherung der Elemente.

Beispiel:

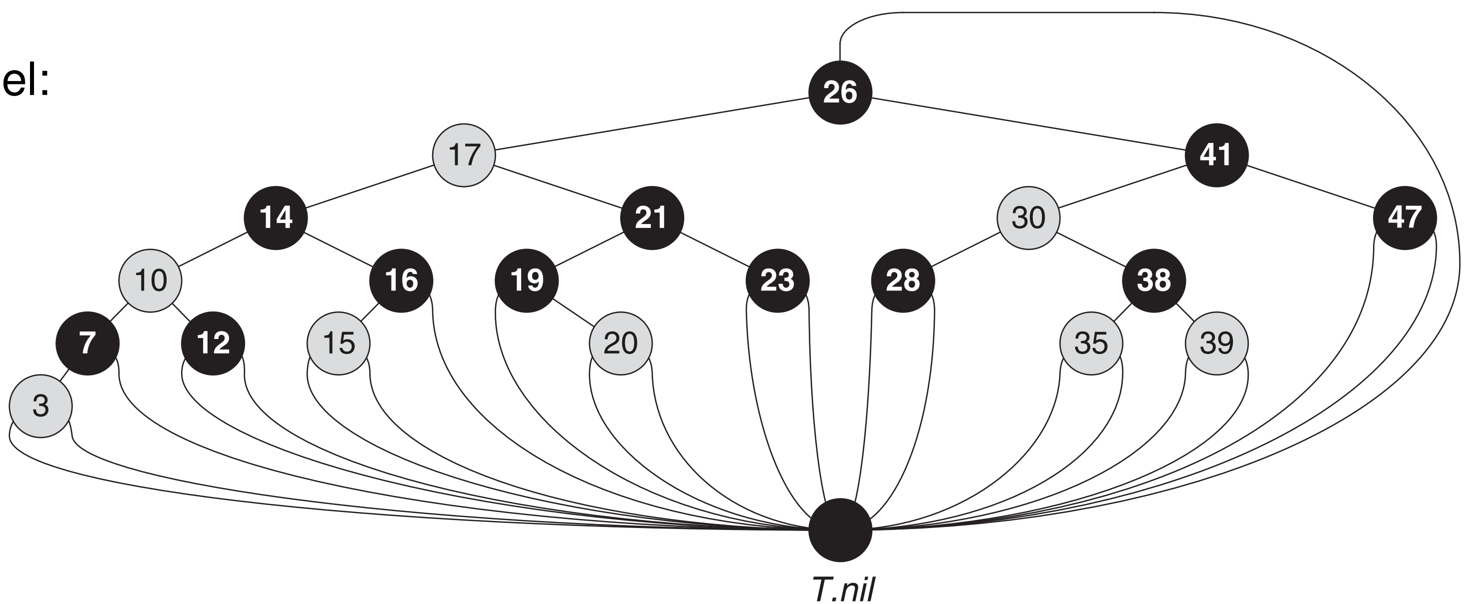


Red-Black Tree

Implementierung

- Knotenorientierte Speicherung der Elemente.
- 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:

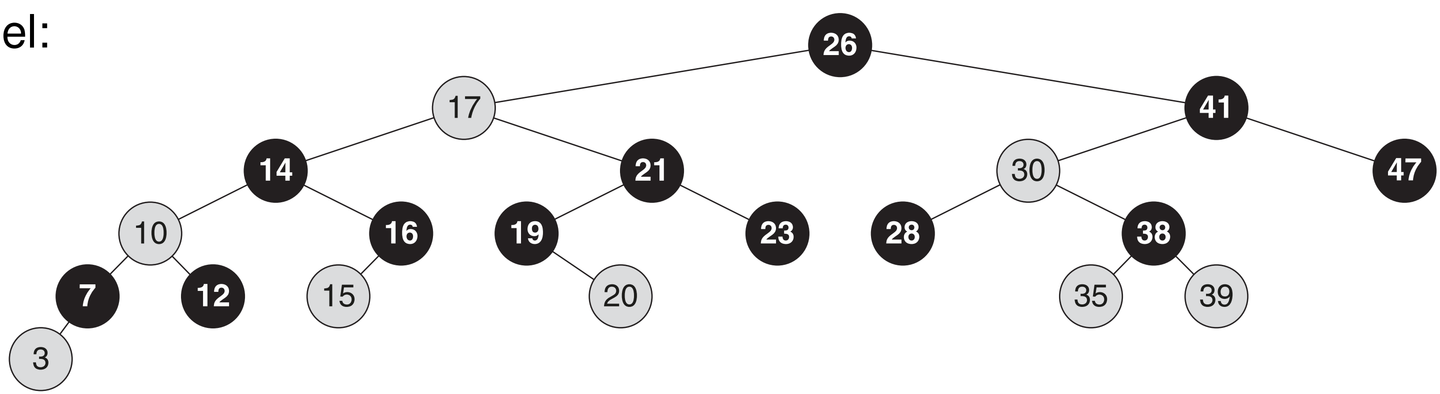


Red-Black Tree

Implementierung

- Knotenorientierte Speicherung der Elemente.
- 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, größtem oder nächstgrößerem Sortierschlüssel bestimmen.

Laufzeit

- Traversierung ist in $O(n)$.

- Suchen, Minimum, Maxium und Nachfolger sind in $O(h)$, wobei h die Höhe des Binary Search Trees 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 kann Seiteneffekte haben:

- 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.

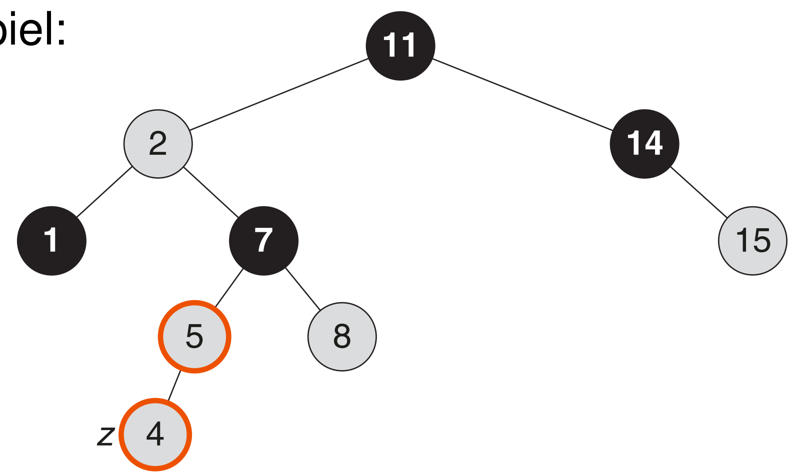
Eingabe: T. Red-Black-Tree.
z. Einzufügender Knoten mit Schlüssel k.

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

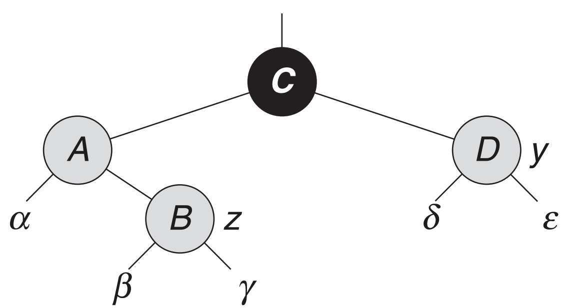
Algorithmus: Red-Black Insert Fixup.

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

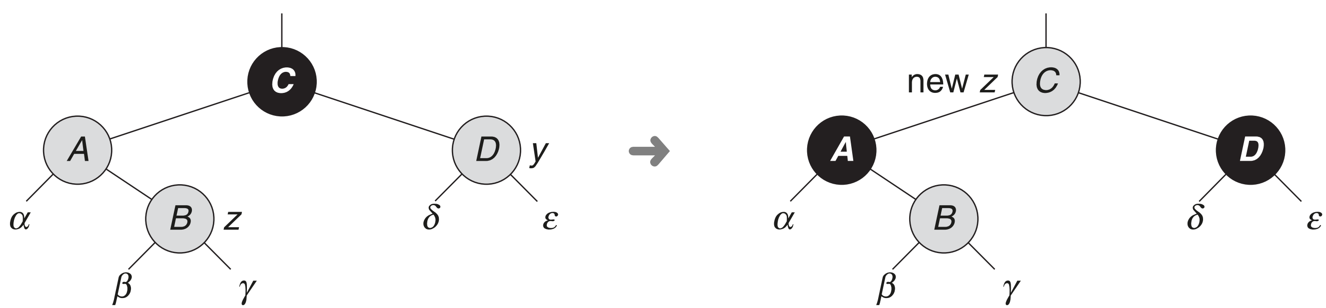
Algorithmus: Red-Black Insert Fixup.

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

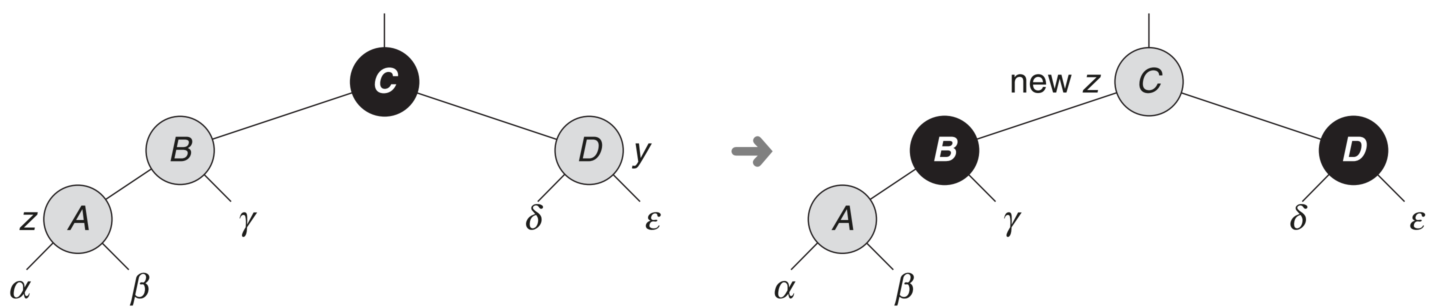
Algorithmus: Red-Black Insert Fixup.

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

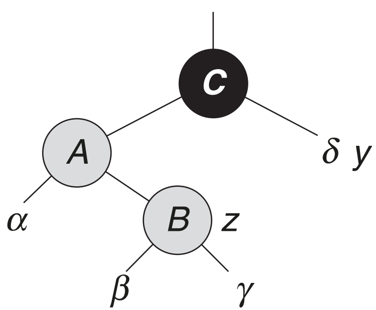
Algorithmus: Red-Black Insert Fixup.

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.

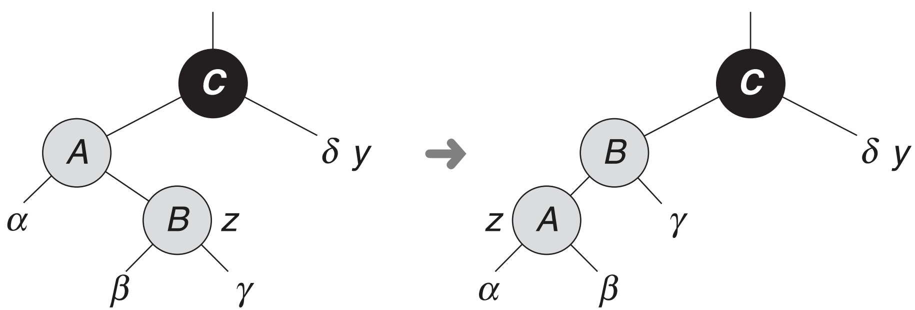
Eingabe: T. Potentiell invalider Red-Black Tree.
z. Knoten, der die Invalidität verursacht.

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

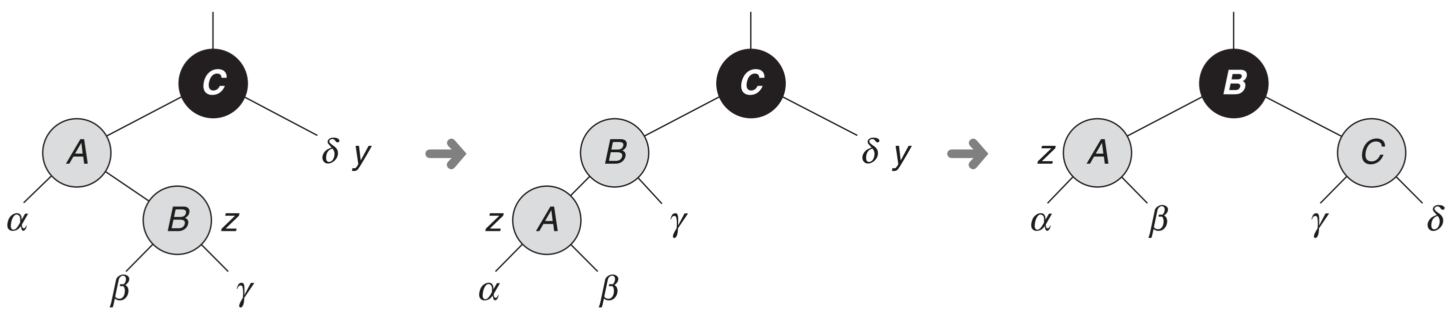
Algorithmus: Red-Black Insert Fixup.

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
```

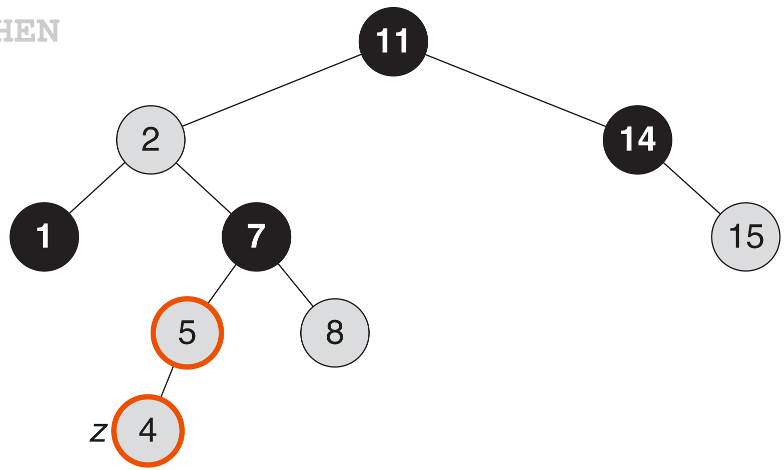
Red-Black Tree

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:



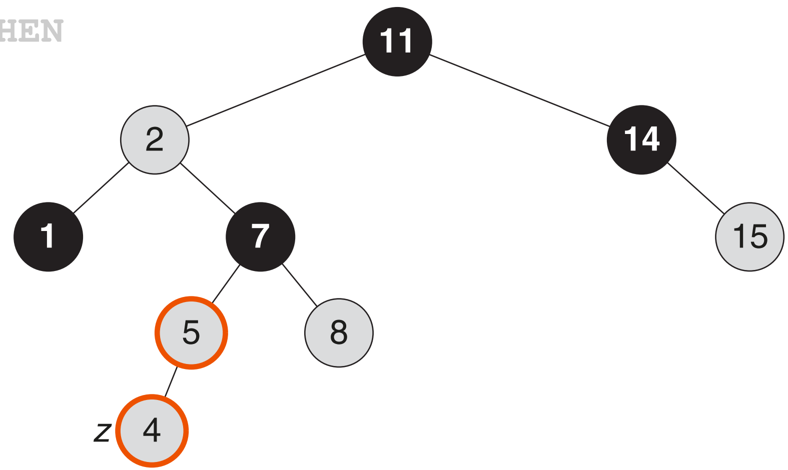
Red-Black Tree

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:



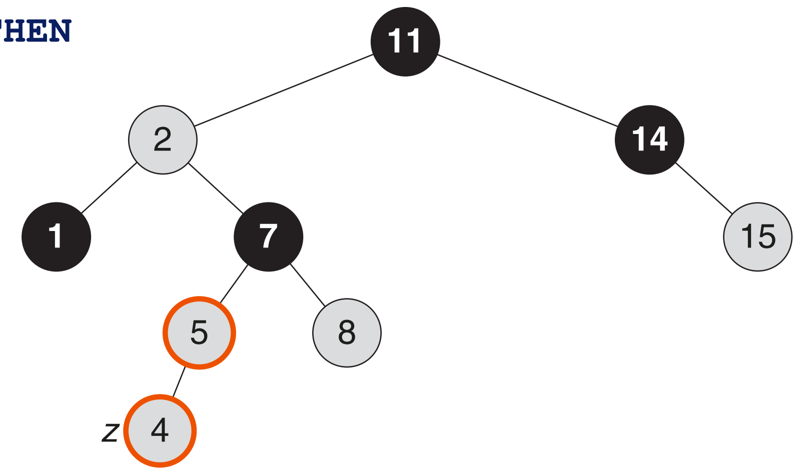
Red-Black Tree

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:



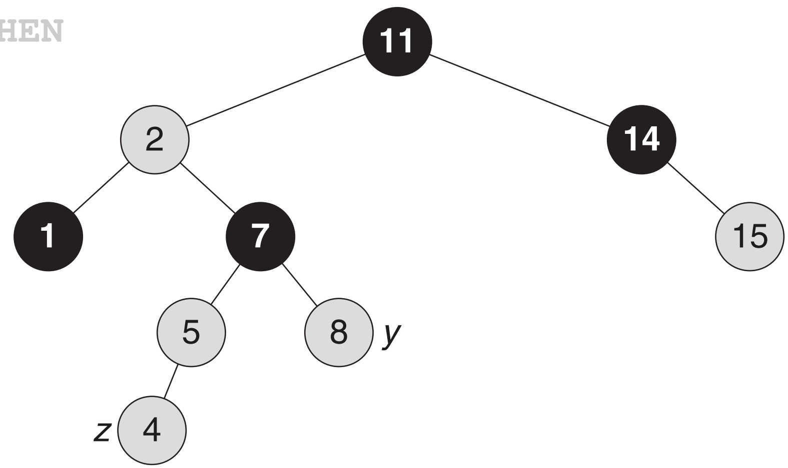
Red-Black Tree

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:



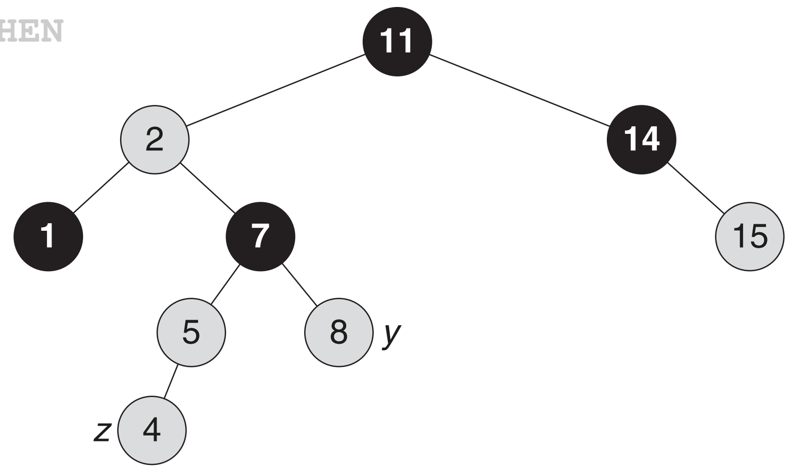
Red-Black Tree

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:



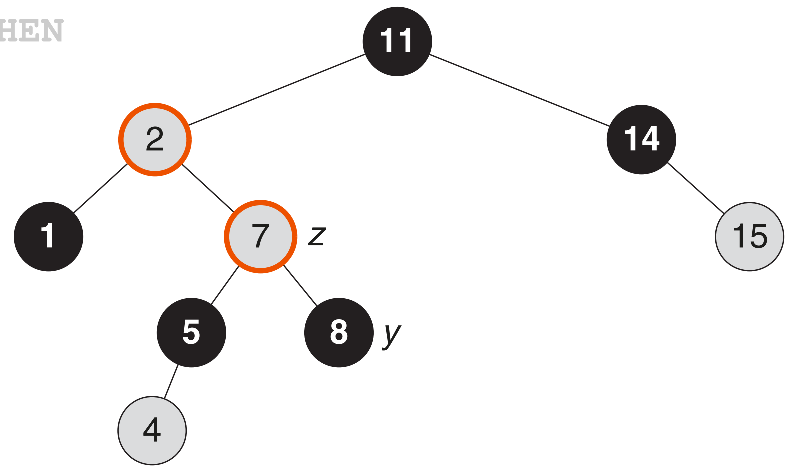
Red-Black Tree

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:



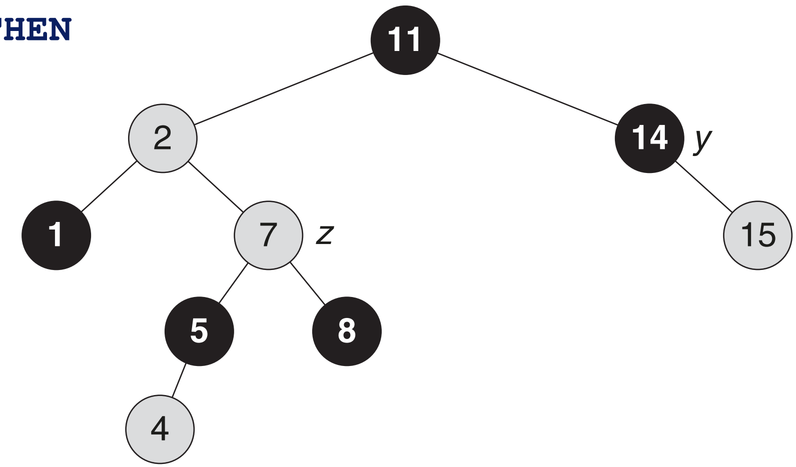
Red-Black Tree

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:



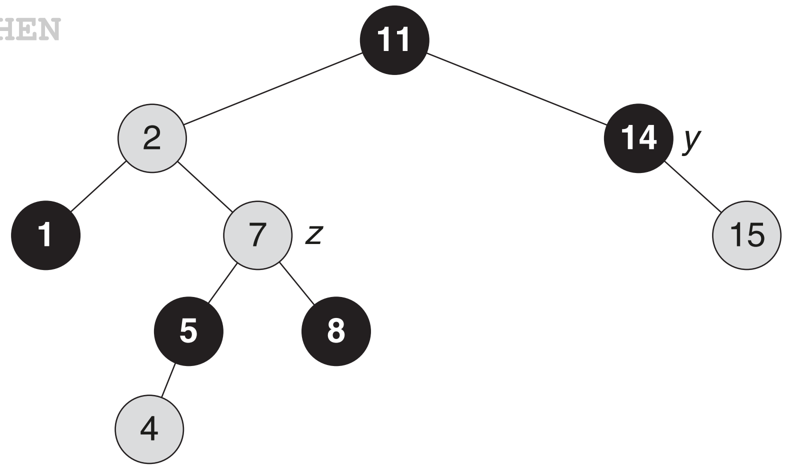
Red-Black Tree

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:



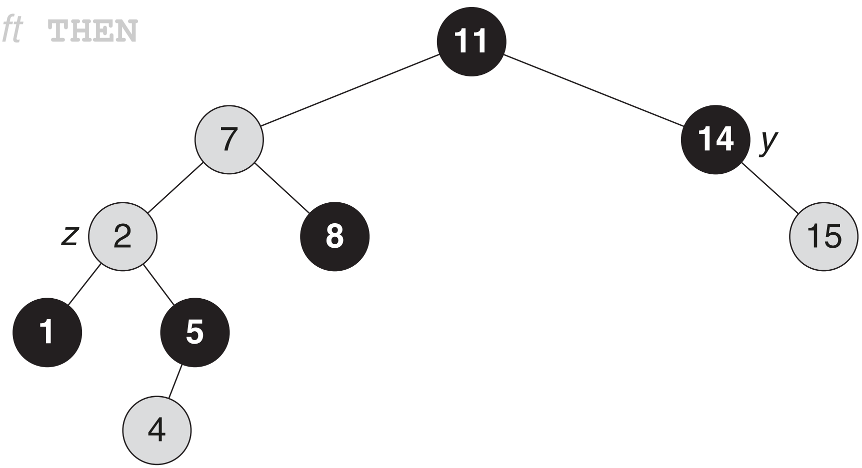
Red-Black Tree

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:



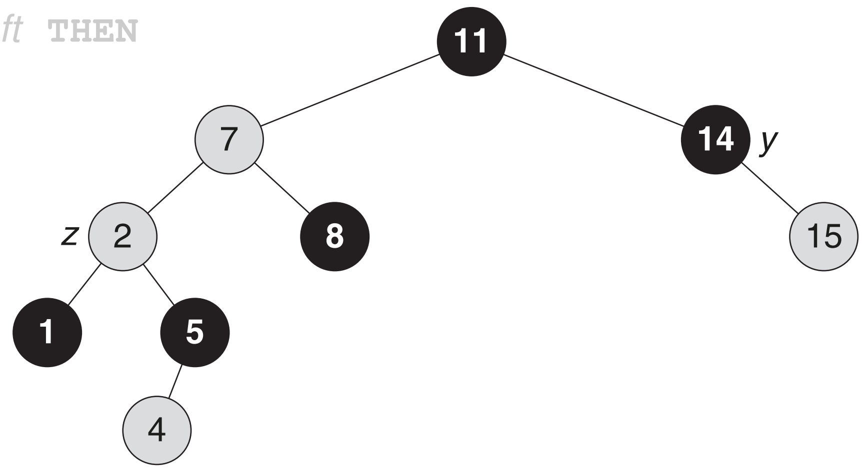
Red-Black Tree

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:



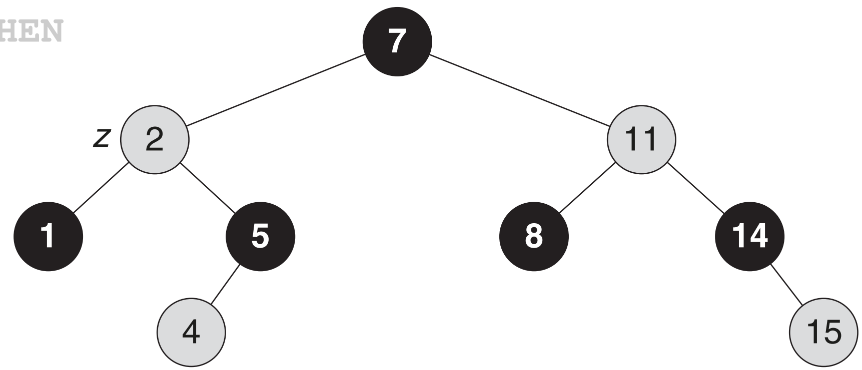
Red-Black Tree

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:



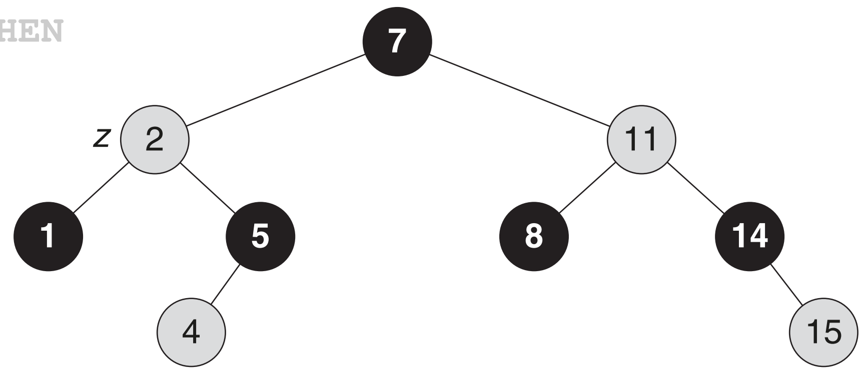
Red-Black Tree

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:



Bemerkungen:

- Laufzeit: *RBInsert* und *RBInsertFixup* benötigen jeweils im Worst Case $O(h)$ Laufzeit, wenn wiederholt Fall 1 auftritt.
- Es werden niemals mehr als zwei Rotationen ausgeführt.

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: Red-Black 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, wobei y die Farbe von z erhält.
   (b) zs Nachfolger y ist ein anderer Knoten in seinem rechten Teilbaum.
   Ersetze y durch sein rechtes Kind x und ersetze z durch y, wobei y die Farbe von z erhält.

➜ Wenn z (Fälle 1 und 2) bzw. zs Nachfolger y (Fall 3) schwarz war, könnten die Red-Black Tree-Bedingungen beim Nachfolger x von z bzw. y verletzt sein.

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. x Geschwister w ist rot.
   Färbe w schwarz und x Elter rot. Linksrotation über x Elter. Fahre bei Fällen 3, 4 oder 5 fort.
3. x Geschwister w ist schwarz und beide Kinder von w sind schwarz.
   Färbe w rot und fahre bei x Elter fort.
4. x 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. x Geschwister w ist schwarz und ws rechtes Kind rot.
   Färbe w wie x Elter und den Elter und ws rechtes Kind schwarz. Linksrotation über x Elter.
   Fahre bei $x = T$.*root* fort.

Red-Black Tree

Manipulation: Löschen

Algorithmus: Red-Black Delete Fixup.

Eingabe: T. Potentiell invalider Red-Black-Tree.

x. Knoten, der die Invalidität verursacht.

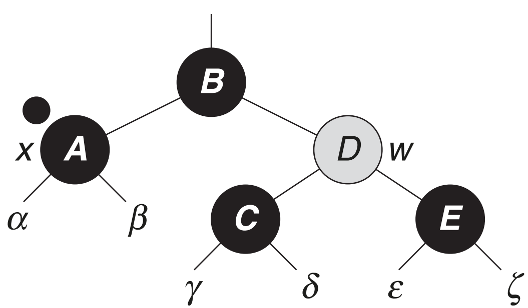
Ausgabe: Valider Red-Black-Tree.

Fallunterscheidung:

2. x Geschwister w ist rot.

   Färbe w schwarz und x Elter rot. Linksrotation über x Elter. Fahre bei Fällen 3, 4 oder 5 fort.

➜ x Geschwister ist schwarz, Reduktion auf nachfolgende Fälle.



Red-Black Tree

Manipulation: Löschen

Algorithmus: Red-Black Delete Fixup.

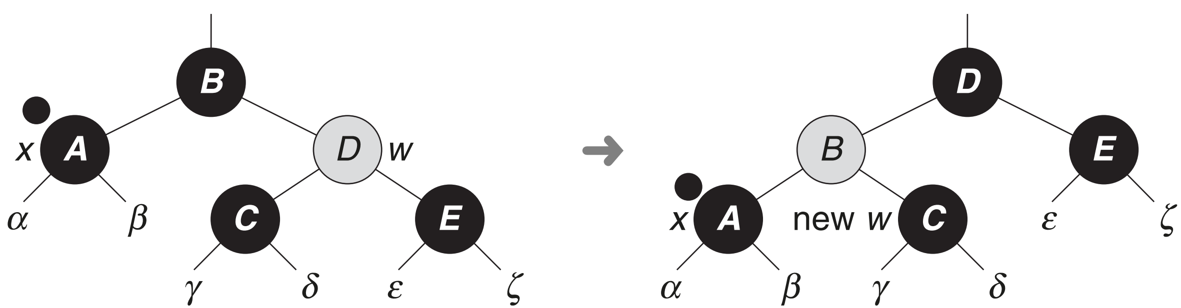
Eingabe: T. Potentiell invalider Red-Black-Tree.
x. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black-Tree.

Fallunterscheidung:

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

→ x Geschwister ist schwarz, Reduktion auf nachfolgende Fälle.



Red-Black Tree

Manipulation: Löschen

Algorithmus: Red-Black Delete Fixup.

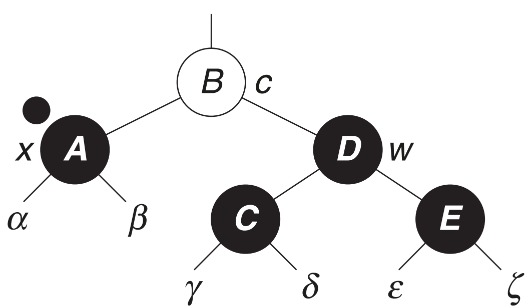
Eingabe: T. Potentiell invalider Red-Black-Tree.
x. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black-Tree.

Fallunterscheidung:

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

➜ Ein schwarzer Knoten weniger pro Teilbaum; schwarzer Chip zu B propagiert.



Red-Black Tree

Manipulation: Löschen

Algorithmus: Red-Black Delete Fixup.

Eingabe: T. Potentiell invalider Red-Black-Tree.

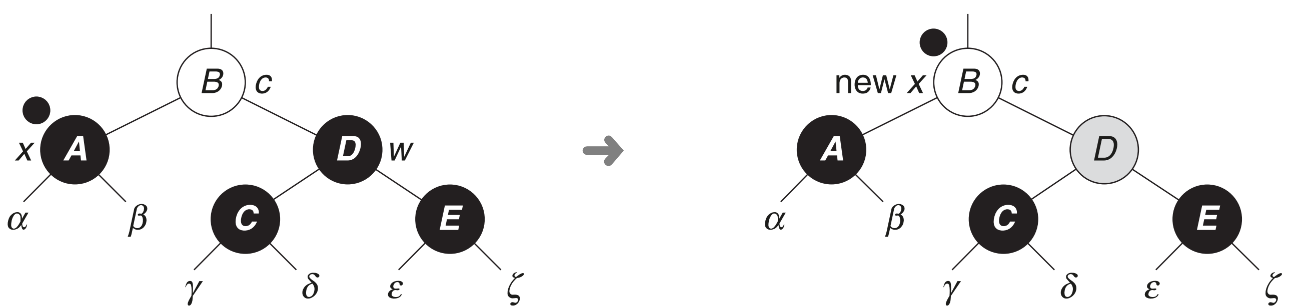
x. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black-Tree.

Fallunterscheidung:

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

➜ Ein schwarzer Knoten weniger pro Teilbaum; schwarzer Chip zu B propagiert.



Red-Black Tree

Manipulation: Löschen

Algorithmus: Red-Black Delete Fixup.

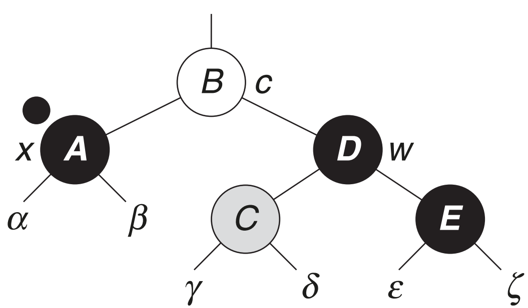
Eingabe: T. Potentiell invalider Red-Black-Tree.
x. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black-Tree.

Fallunterscheidung:

4. x 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.

➜ Reduktion auf Fall 5.



Red-Black Tree

Manipulation: Löschen

Algorithmus: Red-Black Delete Fixup.

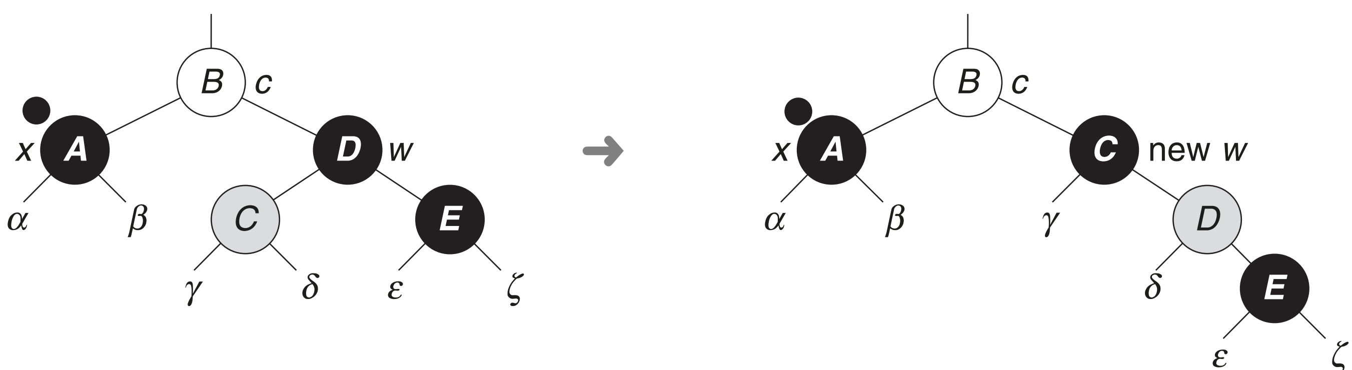
Eingabe: T. Potentiell invalider Red-Black-Tree.
x. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black-Tree.

Fallunterscheidung:

4. x 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.

→ Reduktion auf Fall 5.



Red-Black Tree

Manipulation: Löschen

Algorithmus: Red-Black Delete Fixup.

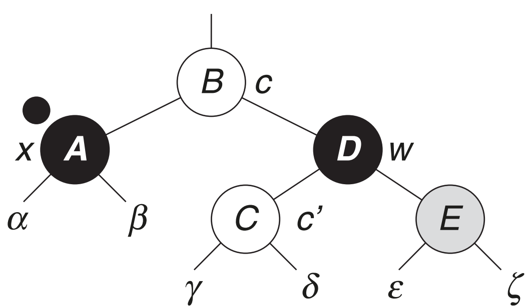
Eingabe: T. Potentiell invalider Red-Black-Tree.
x. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black-Tree.

Fallunterscheidung:

5. x Geschwister w ist schwarz und ws rechtes Kind rot.
   Färbe w wie x Elter und den Elter und ws rechtes Kind schwarz. Linksrotation über x Elter. Fahre bei $x = T$.*root* fort.

➜ Zusätzlicher schwarzer Knoten erzeugt und Teilbaum balanciert.



Red-Black Tree

Manipulation: Löschen

Algorithmus: Red-Black Delete Fixup.

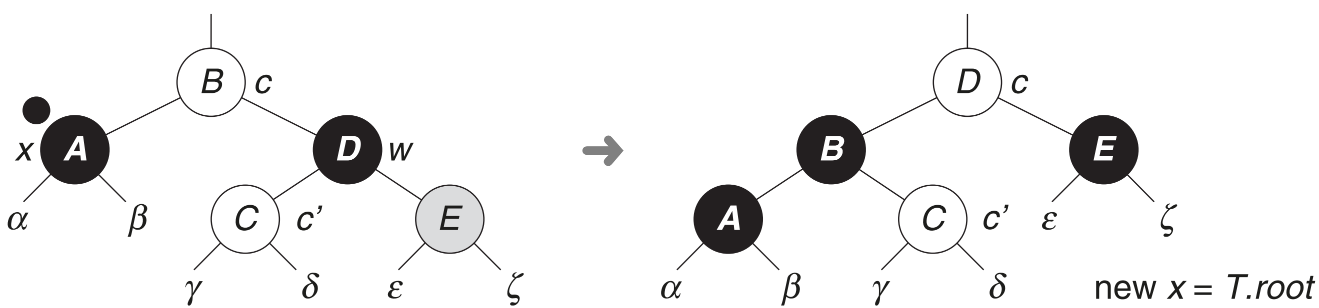
Eingabe: T. Potentiell invalider Red-Black-Tree.
x. Knoten, der die Invalidität verursacht.

Ausgabe: Valider Red-Black-Tree.

Fallunterscheidung:

5. x Geschwister w ist schwarz und ws rechtes Kind rot.
   Färbe w wie x Elter und den Elter und ws rechtes Kind schwarz. Linksrotation über x Elter. Fahre bei $x = T.$*root* fort.

➜ Zusätzlicher schwarzer Knoten erzeugt und Teilbaum balanciert.



Bemerkungen:

- Wenn y schwarz war, muss der Baum so rekonfiguriert werden, dass ein anderer Knoten schwarz gefärbt werden kann, um die vorherige Schwarzhöhe auf allen Pfaden, die y beinhaltet haben, wieder herzustellen.

- Um dies auszudrücken, erhält x einen schwarzen Chip. Dieser Chip dient nur der Anschaulichkeit; er wird im Code nicht explizit kodiert, sondern implizit durch die Variable x: Jeder Knoten, auf den x zeigt, hat aktuell den Chip.

- Fälle 3-5 rekonfigurieren den Baum mit dem Ziel, den schwarzen Chip in Richtung Wurzel zu propagieren bzw. zu einem roten Knoten. Keiner der Fälle ändert etwas an der Verteilung der Schwarzhöhen im Baum.

- Es ist leicht, zu prüfen, dass keine der Rekonfigurationen die Anzahl der schwarzen Knoten auf Pfaden von der Wurzel der gezeigten Teibäume zu den Teilbäumen $\alpha, \beta, \ldots, \zeta$ ändert: Es genügt, für jeden Pfad die Zahl der schwarzen Knoten zu zählen, wobei c und c' der Hilfsfunktion *count*(c) gezählt werden, wobei *count*$($*RED*$) = 0$ und *count*$($*BLACK*$) = 1$ gilt.

- Ein weißer Knoten ist ein Knoten mit beliebiger Färbung c.

- Laufzeit: *RBDelete* und *RBDeleteFixup* benötigen jeweils im Worst Case $O(h)$ Laufzeit, wenn wiederholt Fall 3 auftritt.

- Es werden niemals mehr als drei Rotationen ausgeführt.

Bemerkungen: (Fortsetzung)

- *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
```


Red-Black Tree

Binary Search Tree Hierarchy