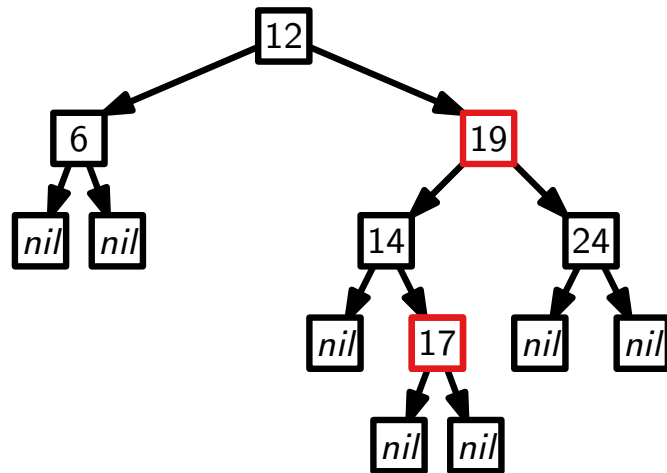
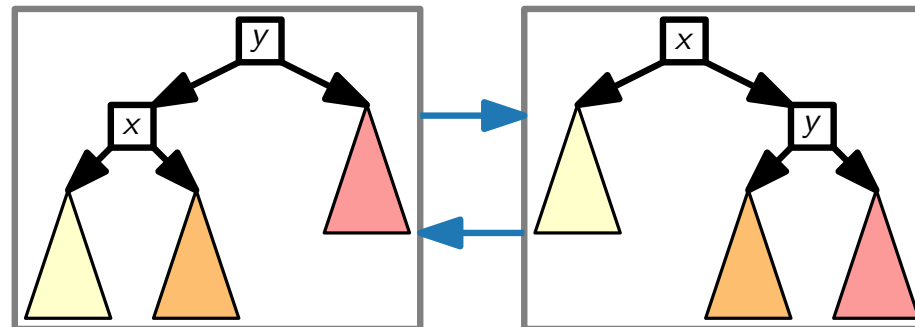


Algorithmen und Datenstrukturen

Vorlesung 14: Rot-Schwarz-Bäume



Diana Sieper



Wintersemester 2025

Dynamische Menge



Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M , wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation

ptr INSERT(key k , value v)
DELETE(ptr x)

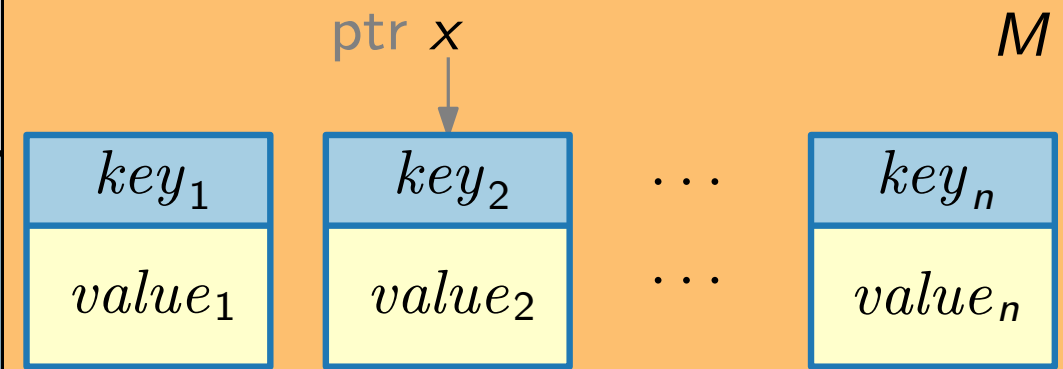
ptr SEARCH(key k)
ptr MINIMUM()
ptr MAXIMUM()
ptr PREDECESSOR(ptr x)
ptr SUCCESSOR(ptr x)

Pointer

Funktionalität

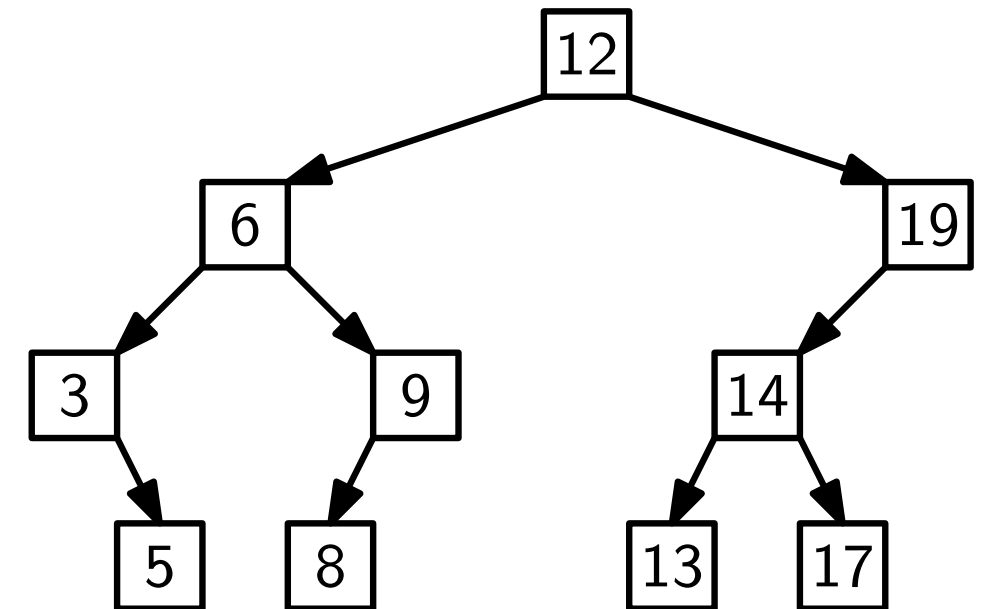
Veränderungen

Anfragen



Binäre Suchbäume

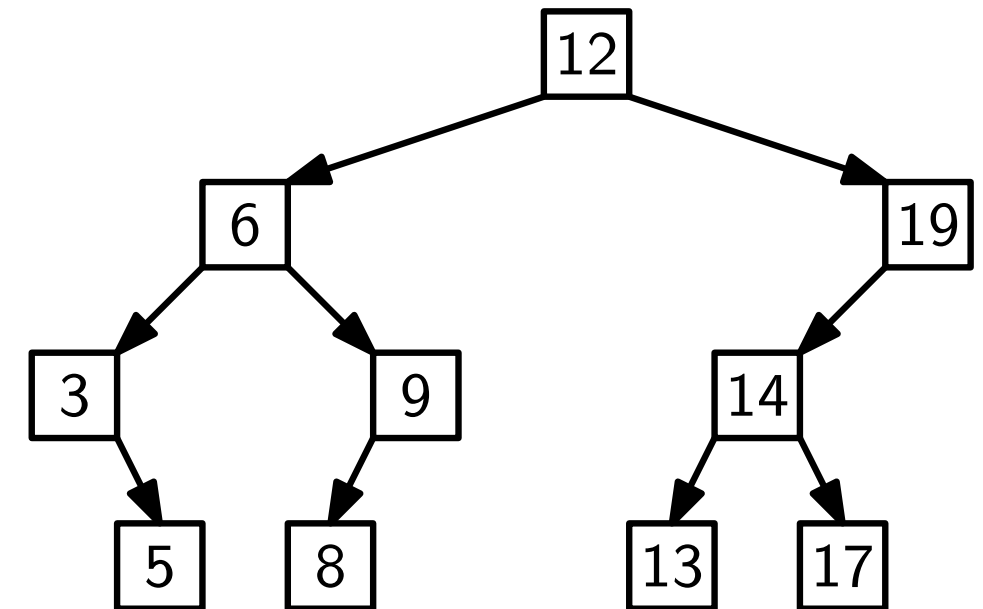
Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $\mathcal{O}(h)$ Zeit, wobei h die momentane Höhe des Baums ist.



Binäre Suchbäume

Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $\mathcal{O}(h)$ Zeit, wobei h die momentane Höhe des Baums ist.

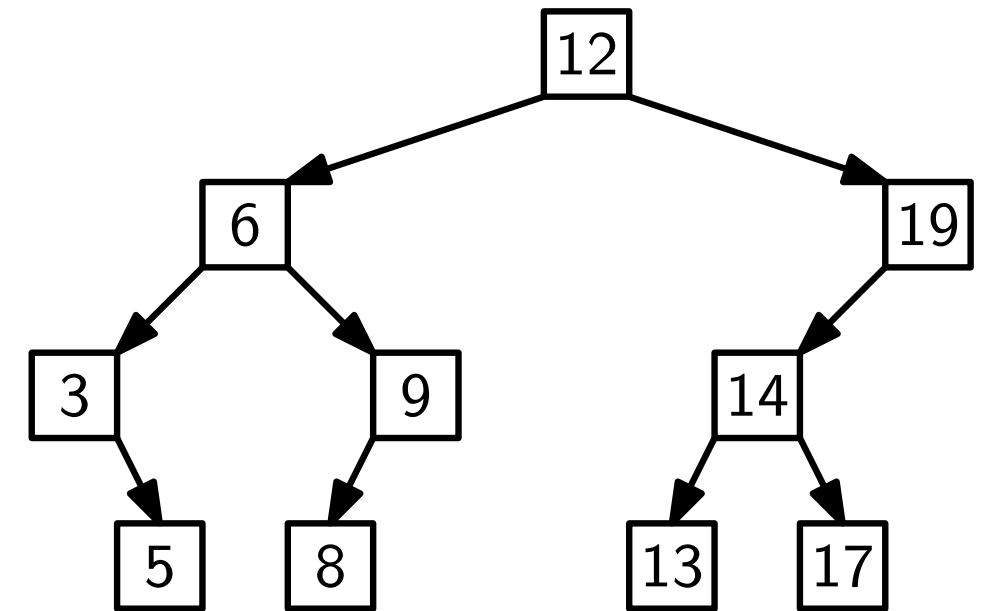
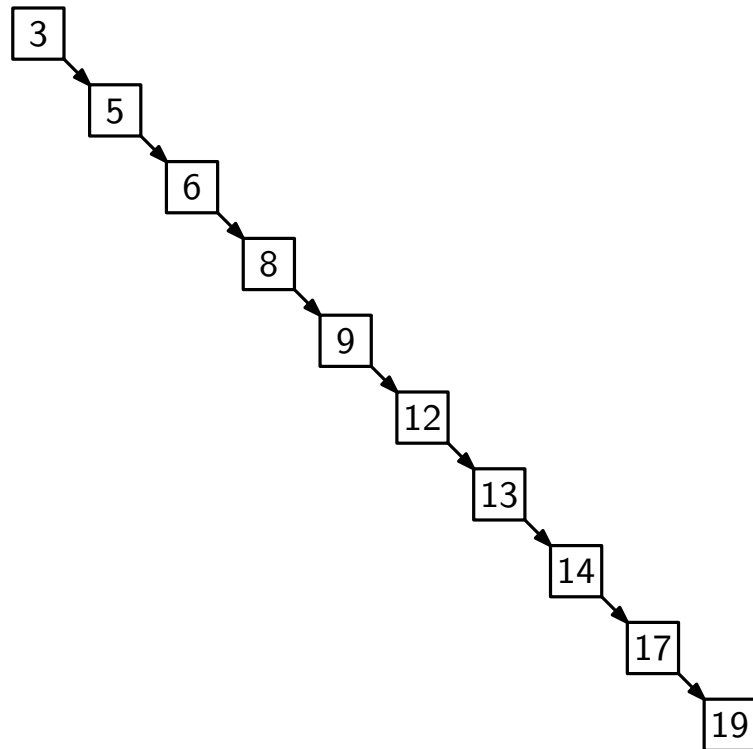
Aber:



Binäre Suchbäume

Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $\mathcal{O}(h)$ Zeit, wobei h die momentane Höhe des Baums ist.

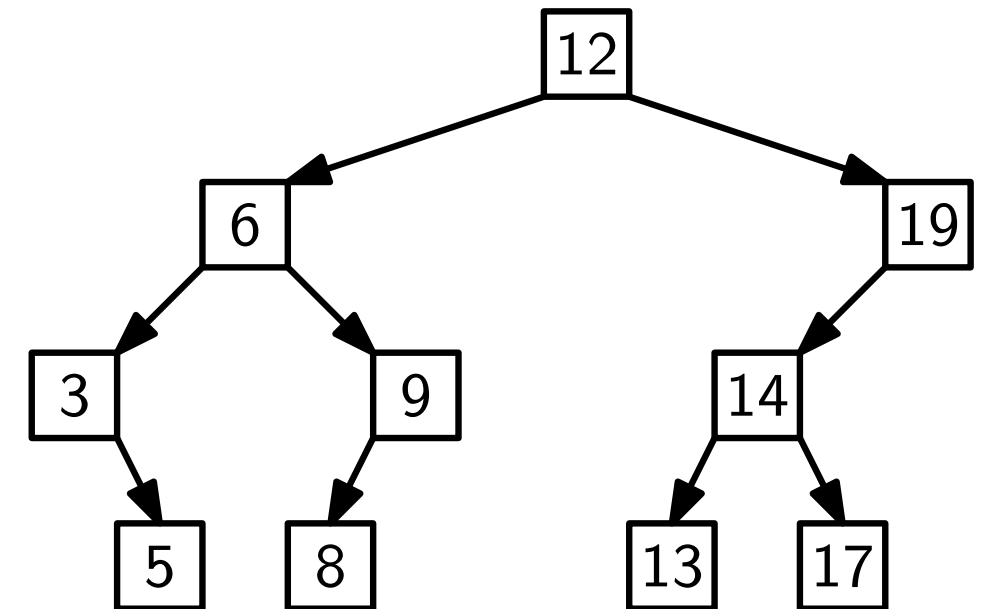
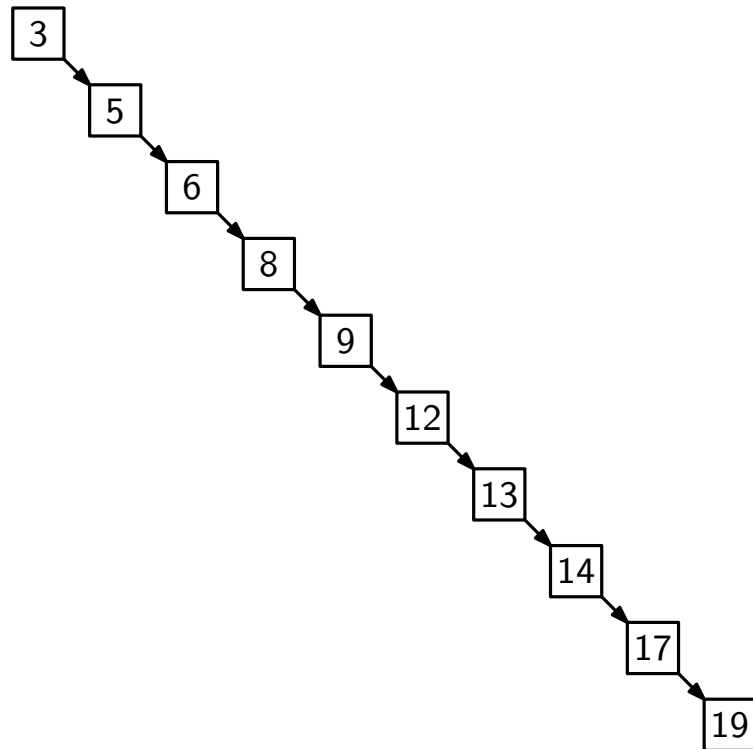
Aber:



Binäre Suchbäume

Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $\mathcal{O}(h)$ Zeit, wobei h die momentane Höhe des Baums ist.

Aber: Im schlechtesten Fall gilt $h \in \Theta(n)$.

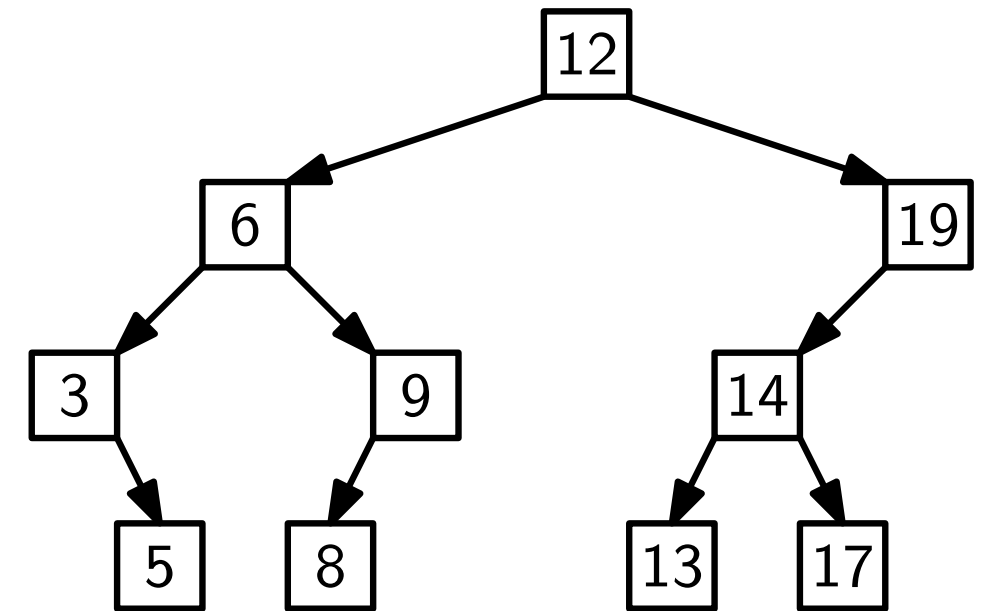
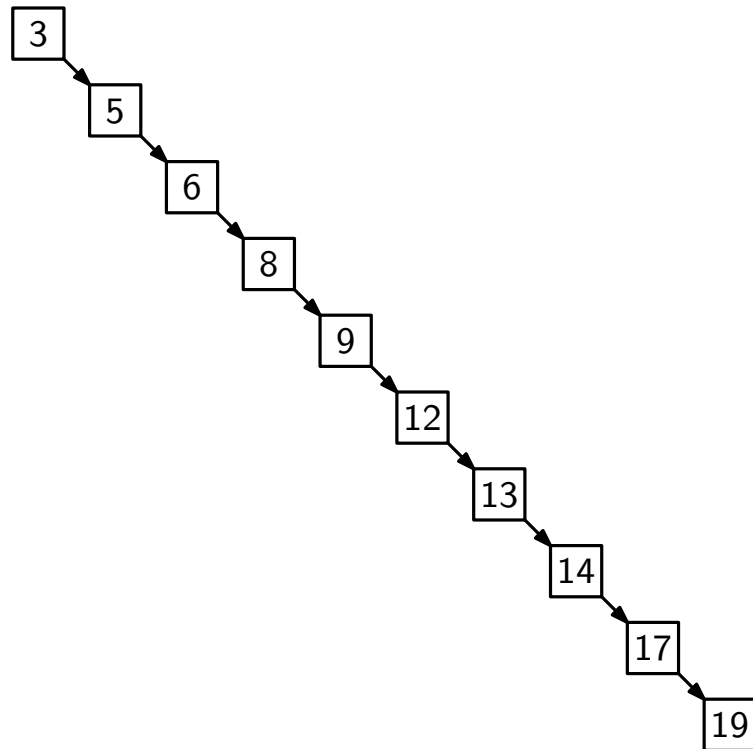


Binäre Suchbäume

Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $\mathcal{O}(h)$ Zeit, wobei h die momentane Höhe des Baums ist.

Aber: Im schlechtesten Fall gilt $h \in \Theta(n)$.

Ziel:

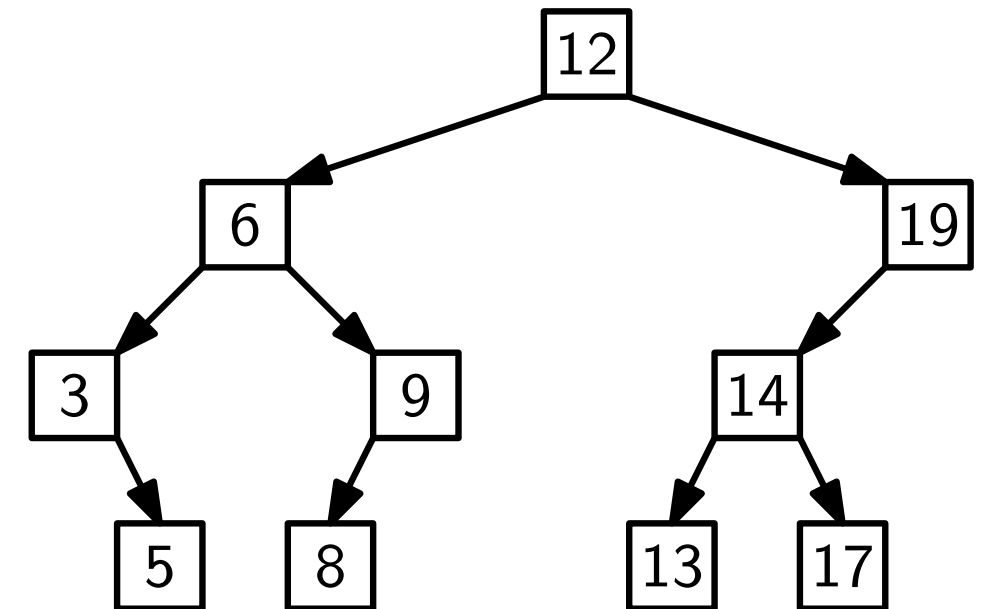
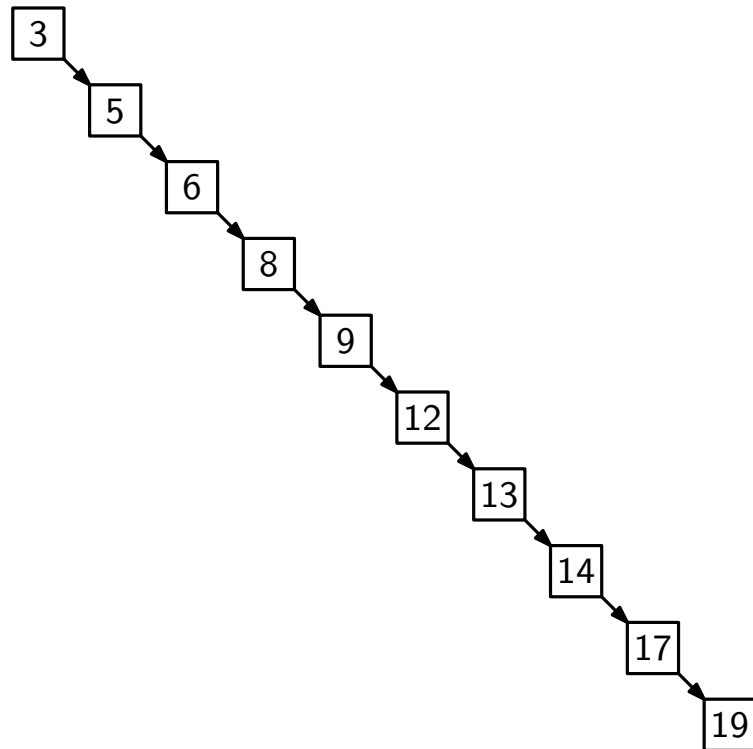


Binäre Suchbäume

Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $\mathcal{O}(h)$ Zeit, wobei h die momentane Höhe des Baums ist.

Aber: Im schlechtesten Fall gilt $h \in \Theta(n)$.

Ziel: Suchbäume **balancieren**

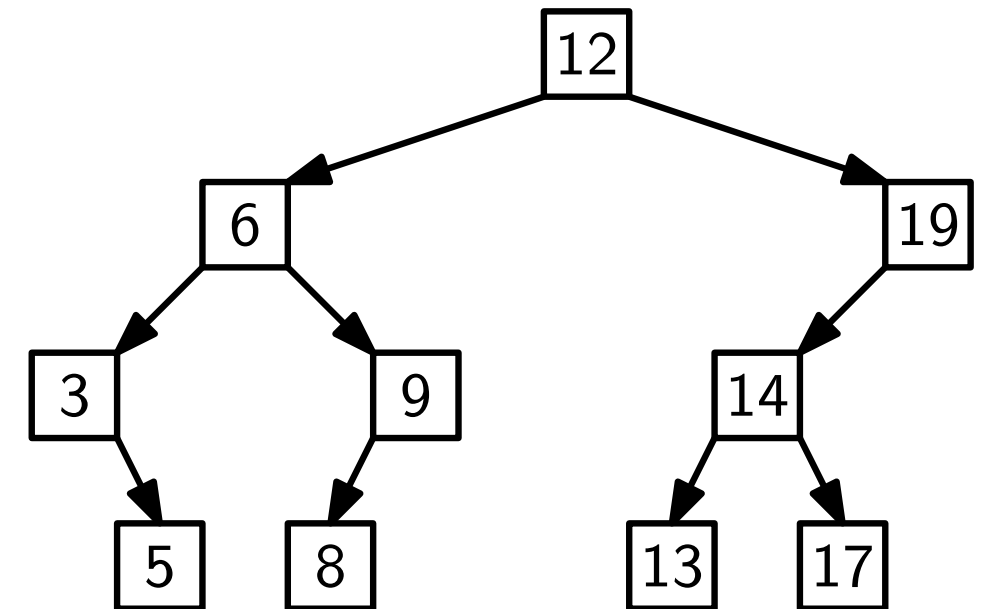
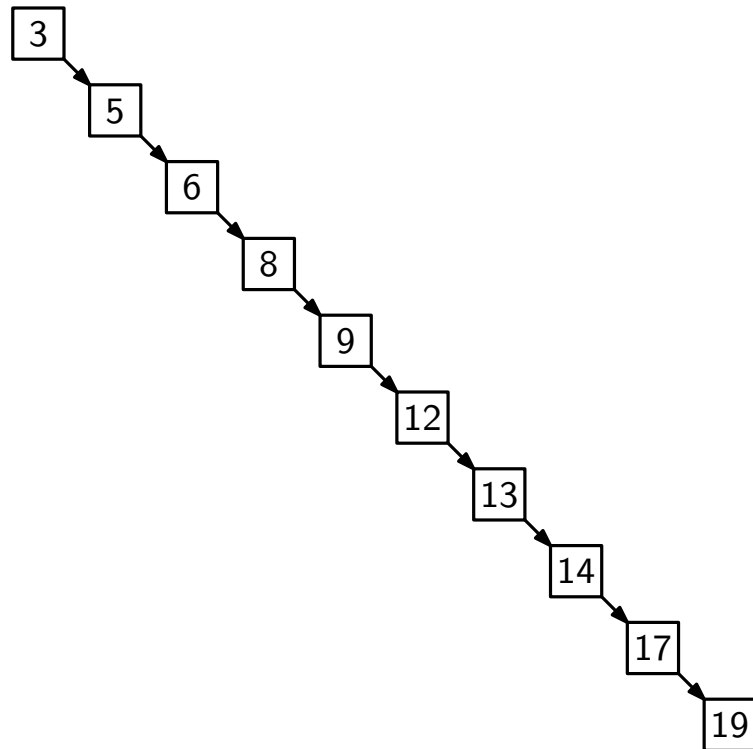


Binäre Suchbäume

Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $\mathcal{O}(h)$ Zeit, wobei h die momentane Höhe des Baums ist.

Aber: Im schlechtesten Fall gilt $h \in \Theta(n)$.

Ziel: Suchbäume **balancieren** $\Rightarrow h \in \mathcal{O}(\log n)$

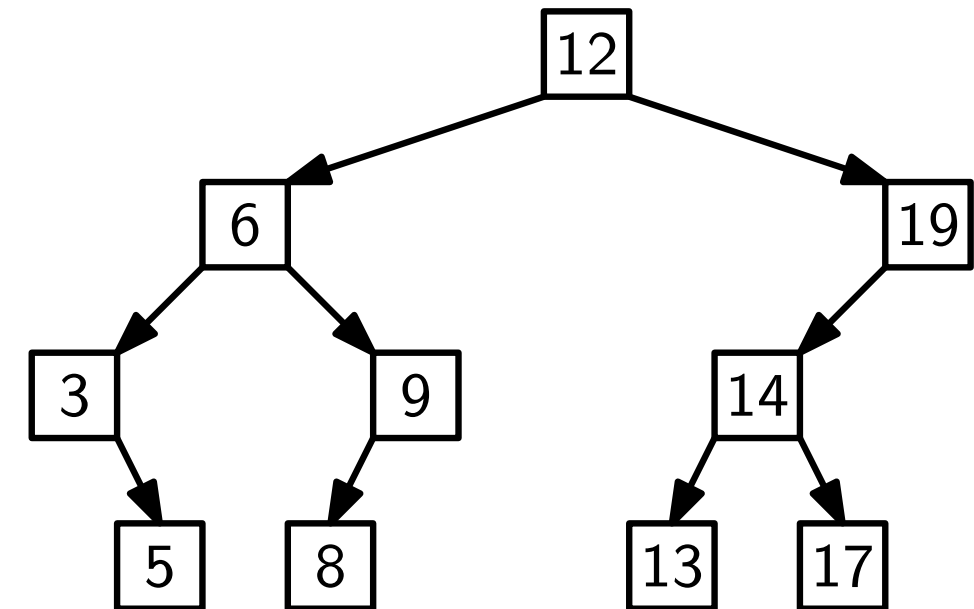
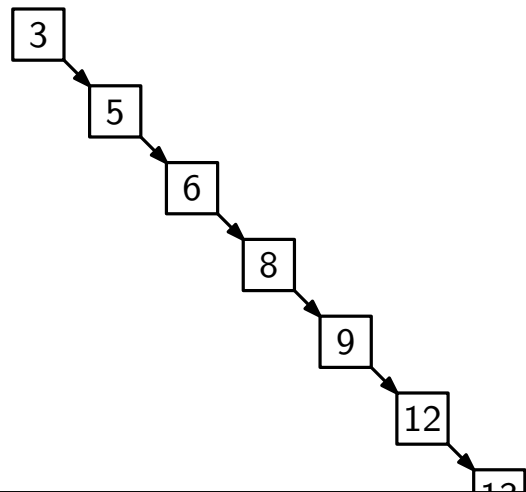


Binäre Suchbäume

Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $\mathcal{O}(h)$ Zeit, wobei h die momentane Höhe des Baums ist.

Aber: Im schlechtesten Fall gilt $h \in \Theta(n)$.

Ziel: Suchbäume **balancieren** $\Rightarrow h \in \mathcal{O}(\log n)$



Binärer-Suchbaum-Eigenschaft.

Für jeden Knoten v gilt:
 alle Knoten im linken Teilbaum von v haben Schlüssel $\leq v.key$
 rechten $\geq$

Balanciermethoden

nach **Gewicht**

für jeden Knoten ist das Gewicht
(= Anzahl der Knoten) des linken
und rechten Teilbaums ungefähr gleich.



Balanciermethoden

nach **Gewicht**

für jeden Knoten ist das Gewicht
(= Anzahl der Knoten) des linken
und rechten Teilbaums ungefähr gleich.

nach **Höhe**

für jeden Knoten ist die Höhe
des linken und rechten Teilbaums ungefähr gleich.



Balanciermethoden

nach **Gewicht**

für jeden Knoten ist das Gewicht
(= Anzahl der Knoten) des linken
und rechten Teilbaums ungefähr gleich.



nach **Höhe**

für jeden Knoten ist die Höhe
des linken und rechten Teilbaums ungefähr gleich.

nach **Grad**

alle Blätter haben dieselbe Tiefe, aber innere Knoten
können verschieden viele Kinder haben.

Balanciermethoden

nach **Gewicht**

für jeden Knoten ist das Gewicht
(= Anzahl der Knoten) des linken
und rechten Teilbaums ungefähr gleich.



nach **Höhe**

für jeden Knoten ist die Höhe
des linken und rechten Teilbaums ungefähr gleich.

nach **Grad**

alle Blätter haben dieselbe Tiefe, aber innere Knoten
können verschieden viele Kinder haben.

nach **Knotenfarbe**

jeder Knoten ist entweder **gut** oder **schlecht**; der Anteil
schlechter Knoten darf in keinem Teilbaum zu groß sein.

Balanciermethoden

Beispiele

nach Gewicht

für jeden Knoten ist das Gewicht
(= Anzahl der Knoten) des linken
und rechten Teilbaums ungefähr gleich.



nach Höhe

für jeden Knoten ist die Höhe
des linken und rechten Teilbaums ungefähr gleich.

nach Grad

alle Blätter haben dieselbe Tiefe, aber innere Knoten
können verschieden viele Kinder haben.

nach Knotenfarbe

jeder Knoten ist entweder **gut** oder **schlecht**; der Anteil
schlechter Knoten darf in keinem Teilbaum zu groß sein.

Balanciermethoden

Beispiele

nach Gewicht

für jeden Knoten ist das Gewicht
(= Anzahl der Knoten) des linken
und rechten Teilbaums ungefähr gleich.

$BB[\alpha]$ -Bäume



nach Höhe

für jeden Knoten ist die Höhe
des linken und rechten Teilbaums ungefähr gleich.

nach Grad

alle Blätter haben dieselbe Tiefe, aber innere Knoten
können verschieden viele Kinder haben.

nach Knotenfarbe

jeder Knoten ist entweder **gut** oder **schlecht**; der Anteil
schlechter Knoten darf in keinem Teilbaum zu groß sein.

Balanciermethoden

Beispiele

nach Gewicht

$BB[\alpha]$ -Bäume

für jeden Knoten ist das Gewicht
(= Anzahl der Knoten) des linken
und rechten Teilbaums ungefähr gleich.



nach Höhe

AVL-Bäume

für jeden Knoten ist die Höhe
des linken und rechten Teilbaums ungefähr gleich.

nach Grad

alle Blätter haben dieselbe Tiefe, aber innere Knoten
können verschieden viele Kinder haben.

nach Knotenfarbe

jeder Knoten ist entweder **gut** oder **schlecht**; der Anteil
schlechter Knoten darf in keinem Teilbaum zu groß sein.

Balanciermethoden

Beispiele

nach Gewicht

für jeden Knoten ist das Gewicht
(= Anzahl der Knoten) des linken
und rechten Teilbaums ungefähr gleich.

$BB[\alpha]$ -Bäume



nach Höhe

für jeden Knoten ist die Höhe
des linken und rechten Teilbaums ungefähr gleich.

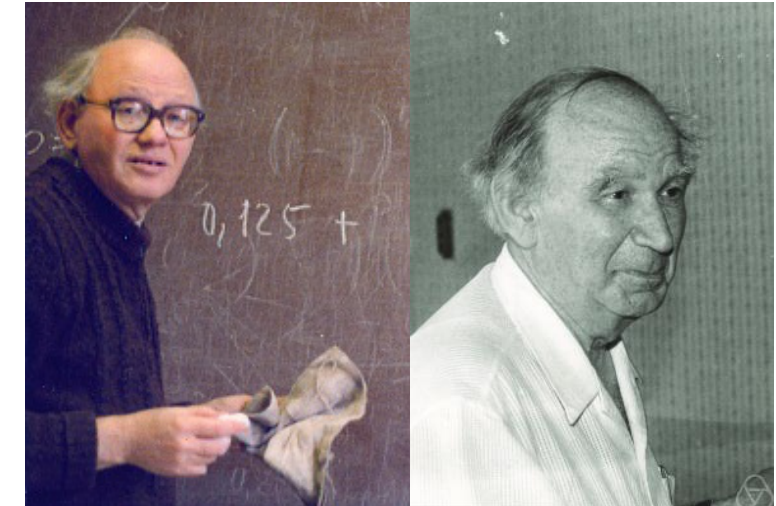
AVL-Bäume

nach Grad

alle Blätter haben dieselbe Tiefe, aber innere Knoten
können verschieden viele Kinder haben.

nach Knotenfarbe

jeder Knoten ist entweder **gut** oder **schlecht**; der Anteil
schlechter Knoten darf in keinem Teilbaum zu groß sein.



Georgi M. Adelson-Velski
1922–2014

Jewgeni M. Landis
1921–1997

Balanciermethoden

Beispiele

nach Gewicht

für jeden Knoten ist das Gewicht
(= Anzahl der Knoten) des linken
und rechten Teilbaums ungefähr gleich.

$BB[\alpha]$ -Bäume



nach Höhe

für jeden Knoten ist die Höhe
des linken und rechten Teilbaums ungefähr gleich.

AVL-Bäume

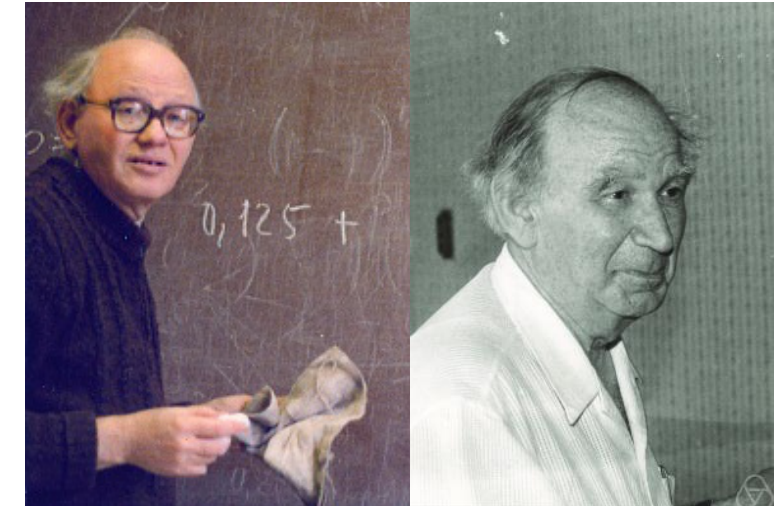
nach Grad

alle Blätter haben dieselbe Tiefe, aber innere Knoten
können verschieden viele Kinder haben.

$(2, 3)$ -Bäume

nach Knotenfarbe

jeder Knoten ist entweder **gut** oder **schlecht**; der Anteil
schlechter Knoten darf in keinem Teilbaum zu groß sein.



Georgi M. Adelson-Velski
1922–2014

Jewgeni M. Landis
1921–1997

Balanciermethoden

Beispiele

nach Gewicht

für jeden Knoten ist das Gewicht
(= Anzahl der Knoten) des linken
und rechten Teilbaums ungefähr gleich.

$BB[\alpha]$ -Bäume



nach Höhe

für jeden Knoten ist die Höhe
des linken und rechten Teilbaums ungefähr gleich.

AVL-Bäume

nach Grad

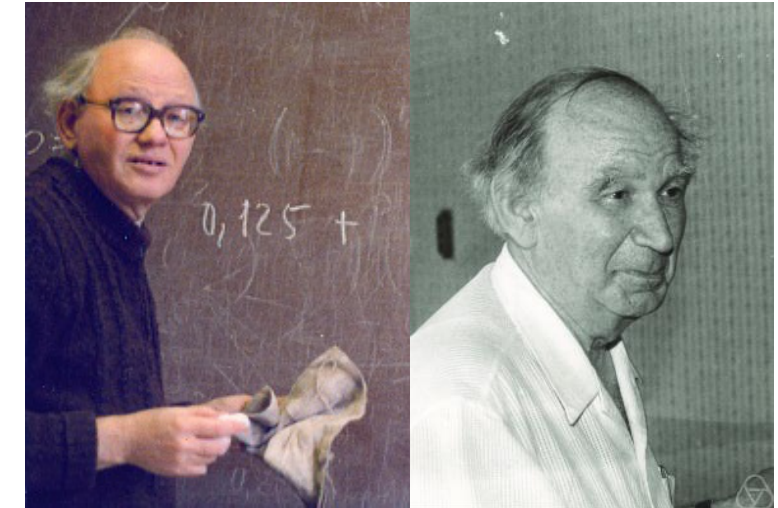
alle Blätter haben dieselbe Tiefe, aber innere Knoten
können verschieden viele Kinder haben.

(2, 3)-Bäume

nach Knotenfarbe

jeder Knoten ist entweder **gut** oder **schlecht**; der Anteil
schlechter Knoten darf in keinem Teilbaum zu groß sein.

Rot-Schwarz-Bäume



Georgi M. Adelson-Velski
1922–2014

Jewgeni M. Landis
1921–1997

Balanciermethoden

Beispiele

nach Gewicht

für jeden Knoten ist das Gewicht
(= Anzahl der Knoten) des linken
und rechten Teilbaums ungefähr gleich.

$BB[\alpha]$ -Bäume



nach Höhe

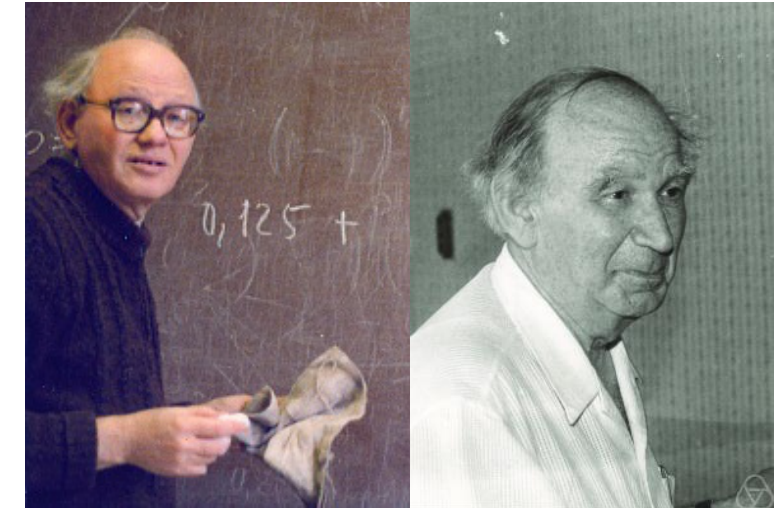
für jeden Knoten ist die Höhe
des linken und rechten Teilbaums ungefähr gleich.

AVL-Bäume

nach Grad

alle Blätter haben dieselbe Tiefe, aber innere Knoten
können verschieden viele Kinder haben.

$(2, 3)$ -Bäume



Georgi M. Adelson-Velski
1922–2014

Jewgeni M. Landis
1921–1997

nach Knotenfarbe

Rot-Schwarz-Bäume

jeder Knoten ist entweder **gut** oder **schlecht**; der Anteil
schlechter Knoten darf in keinem Teilbaum zu groß sein.

Rot-Schwarz-Bäume: Anwendungen

Red-black trees are also particularly valuable in functional programming, where they are one of the most common persistent data structures, used to construct associative arrays and sets that can retain previous versions after mutations. The persistent version of red-black trees requires $O(\log n)$ space for each insertion or deletion, in addition to time.

Rot-Schwarz-Bäume: Anwendungen

Red-black trees are also particularly valuable in functional programming, where they are one of the most common persistent data structures, used to construct associative arrays and sets that can retain previous versions after mutations. The persistent version of red-black trees requires $O(\log n)$ space for each insertion or deletion, in addition to time.

JAVA 8 improvements

The inner representation of the `HashMap` has changed a lot in JAVA 8. Indeed, the implementation in JAVA 7 takes 1k lines of code whereas the implementation in JAVA 8 takes 2k lines. Most of what I've said previously is true except the linked lists of entries. In JAVA8, you still have an array but it now stores Nodes that contains the exact same information as Entries and therefore are also linked lists:

Here is a part of the Node implementation in JAVA 8:

```
static class Node<K,V> implements Map.Entry<K,V> {  
    final int hash;  
    final K key;  
    V value;  
    Node<K,V> next;  
}
```

So what's the big difference with JAVA 7? Well, Nodes can be extended to `TreeNode`s. A `TreeNode` is a red-black tree structure that stores really more information so that it can add, delete or get an element in $O(\log(n))$.

Rot-Schwarz-Bäume: Anwendungen

Red-black trees are also particularly valuable in functional programming, where they are one of the most common persistent data structures, used to construct associative arrays and sets that can retain previous versions after mutations. The persistent version of red-black trees requires $O(\log n)$ space for each insertion or deletion, in addition to time.

JAVA 8 improvements

The inner representation of the `HashMap` has changed a lot in JAVA 8. Indeed, the implementation in JAVA 7 takes 1k lines of code whereas the implementation in JAVA 8 takes 2k lines. Most of what I've said previously is true except the linked lists of entries. In JAVA8, you still have an array but it now stores Nodes that contains the exact same information as Entries and therefore are also linked lists:

Here is a part of the Node implementation in JAVA 8:

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;
```

So what's the big difference with JAVA 7? Well, Nodes can be extended to `TreeNode`s. A `TreeNode` is a red-black tree structure that stores really more information so that it can add, delete or get an element in $O(\log(n))$.

Red Black Trees are from a class of self balancing BSTs and as answered by others, any such self balancing tree can be used. I would like to add that Red-black trees are widely used as system symbol tables. For example they are used in implementing the following:

- Java: `java.util.TreeMap` , `java.util.TreeSet` .
- C++ STL: `map`, `multimap`, `multiset`.
- Linux kernel: completely fair scheduler, `linux/rbtree.h`

Rot-Schwarz-Bäume: Anwendungen

Red-black trees are also particularly valuable in functional programming, where they are one of the most common persistent data structures, used to construct associative arrays and sets that can retain previous versions after mutations. The persistent version of red-black trees requires $O(\log n)$ space for each insertion or deletion, in addition to time.

Data indexing in database engines uses RB trees directly or indirectly.

For example, MySQL uses B+ trees, which can be seen as a type of B tree. An RB tree is similar in structure to a B tree of order 4.

JAVA 8 improvements

The inner representation of the HashMap has changed a lot in JAVA 8. Indeed, the implementation in JAVA 7 takes 1k lines of code whereas the implementation in JAVA 8 takes 2k lines. Most of what I've said previously is true except the linked lists of entries. In JAVA8, you still have an array but it now stores Nodes that contains the exact same information as Entries and therefore are also linked lists:

Here is a part of the Node implementation in JAVA 8:

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;
```

So what's the big difference with JAVA 7? Well, Nodes can be extended to TreeNodes. A TreeNode is a red-black tree structure that stores really more information so that it can add, delete or get an element in $O(\log(n))$.

Red Black Trees are from a class of self balancing BSTs and as answered by others, any such self balancing tree can be used. I would like to add that Red-black trees are widely used as system symbol tables. For example they are used in implementing the following:

- Java: java.util.TreeMap , java.util.TreeSet .
- C++ STL: map, multimap, multiset.
- Linux kernel: completely fair scheduler, linux/rbtree.h

Rot-Schwarz-Bäume: Anwendungen

Completely Fair Scheduler

From Wikipedia, the free encyclopedia

The **Completely Fair Scheduler (CFS)** is a [process scheduler](#) that was merged into the 2.6.23 (October 2007) release of the [Linux kernel](#) and is the default scheduler of the tasks of the `SCHED_NORMAL` class (i.e., tasks that have no real-time execution constraints). It handles [CPU](#) resource allocation for executing [processes](#), and aims to maximize overall CPU utilization while also maximizing interactive performance.

In contrast to the previous [O\(1\) scheduler](#) used in older Linux 2.6 kernels, which maintained and switched [run queues](#) of active and expired tasks, the CFS scheduler implementation is based on per-CPU run queues, whose nodes are time-ordered schedulable entities that are kept sorted by [red-black trees](#). The CFS does away with the old notion of per-priorities fixed time-slices and instead it aims at giving a fair share of CPU time to tasks (or, better, schedulable entities).^{[1][2]}

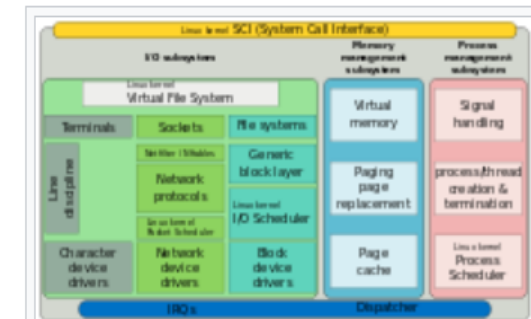
Contents [\[hide\]](#)

- [Algorithm](#)
- [History](#)
- [See also](#)
- [References](#)
- [External links](#)

Algorithm

Completely Fair Scheduler

Original author(s)	Ingo Molnár
Developer(s)	Linux kernel developers
Written in	C
Operating system	Linux kernel
Type	process scheduler
License	GPL-2.0
Website	kernel.org ↗

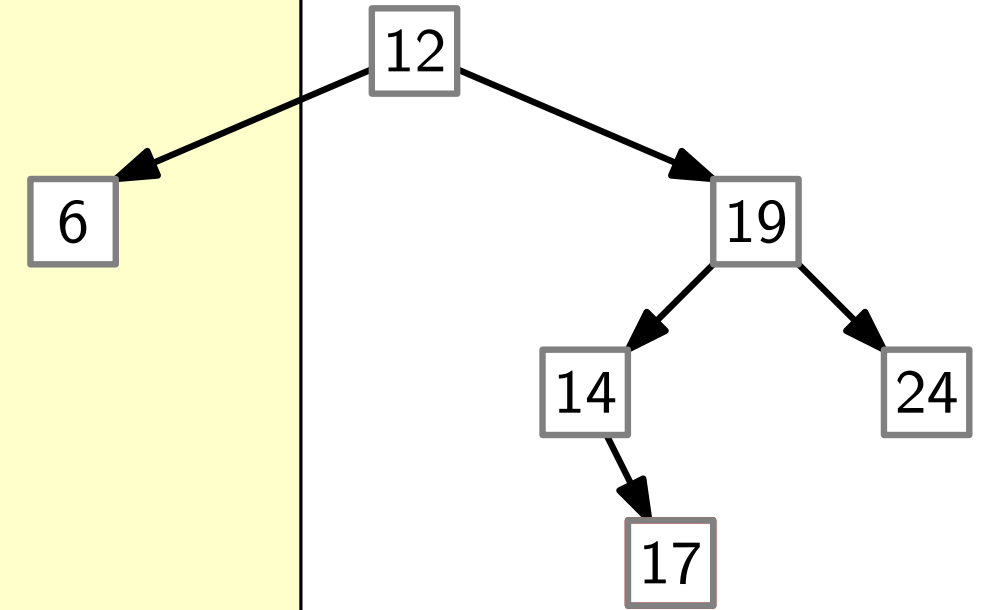


Location of the "Completely Fair Scheduler" (a process scheduler) in a simplified structure of the Linux kernel. [↗](#)

Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden

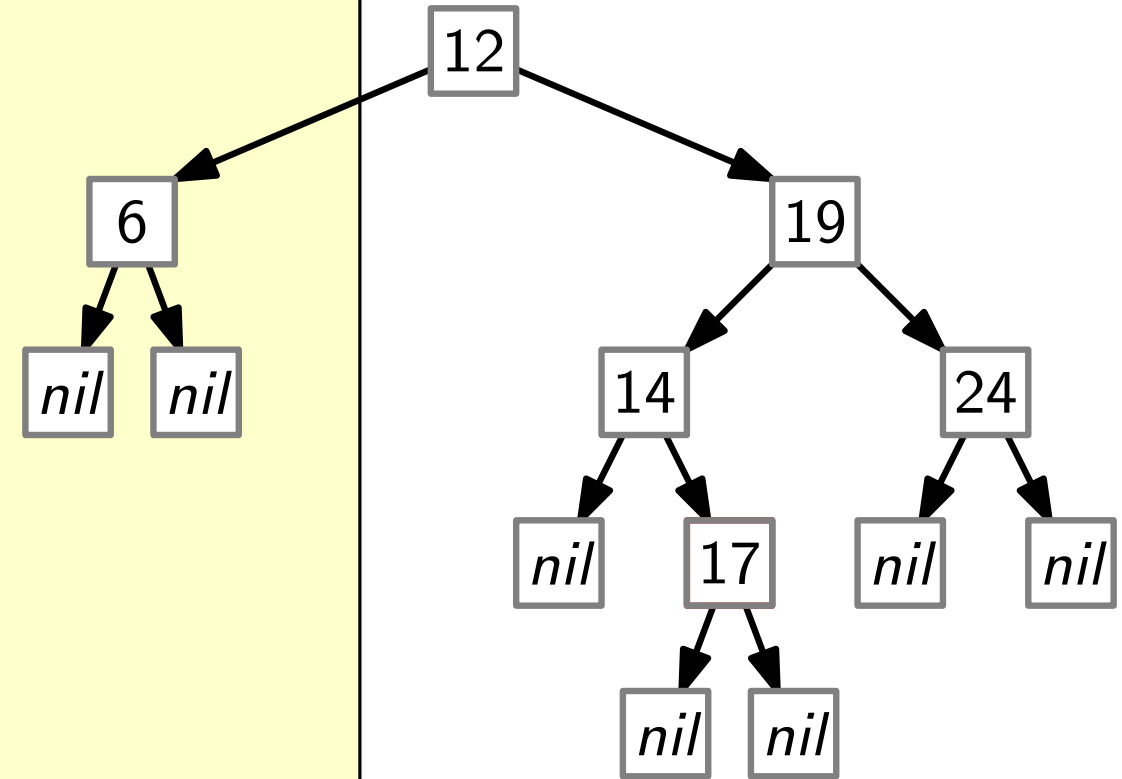
Rot-Schwarz-Eigenschaften:



Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden

Rot-Schwarz-Eigenschaften:

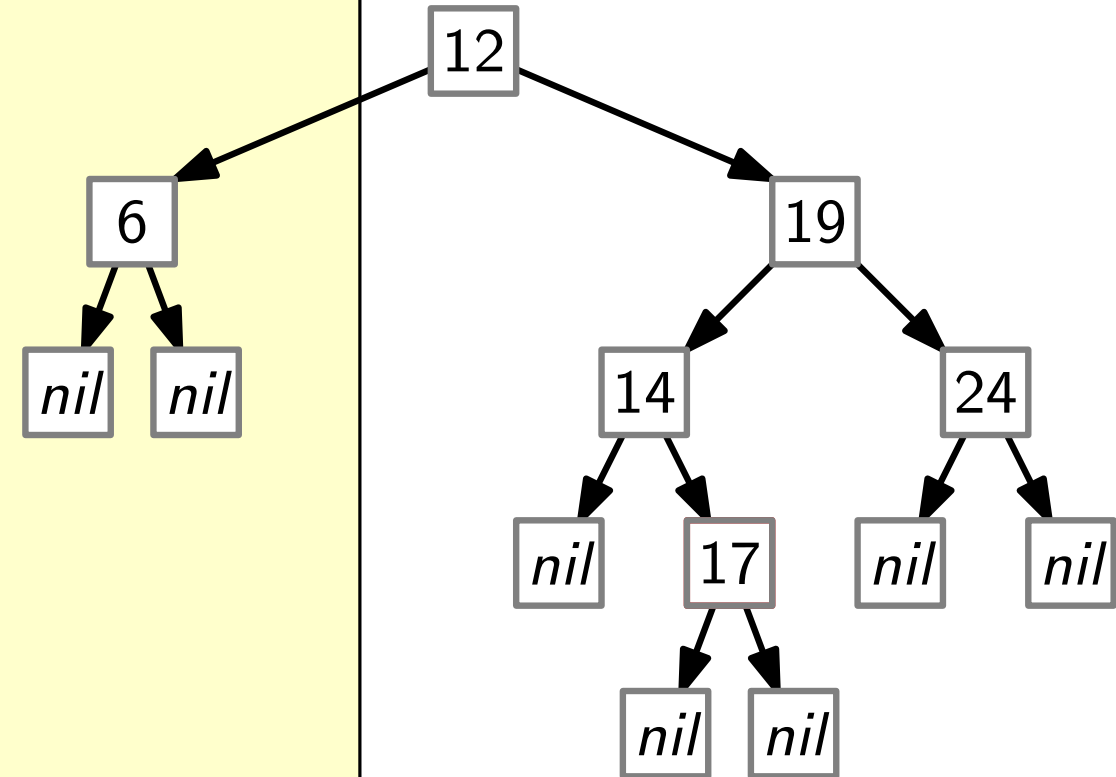


Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden

Rot-Schwarz-Eigenschaften:

(E1) Jeder Knoten ist entweder rot oder schwarz.

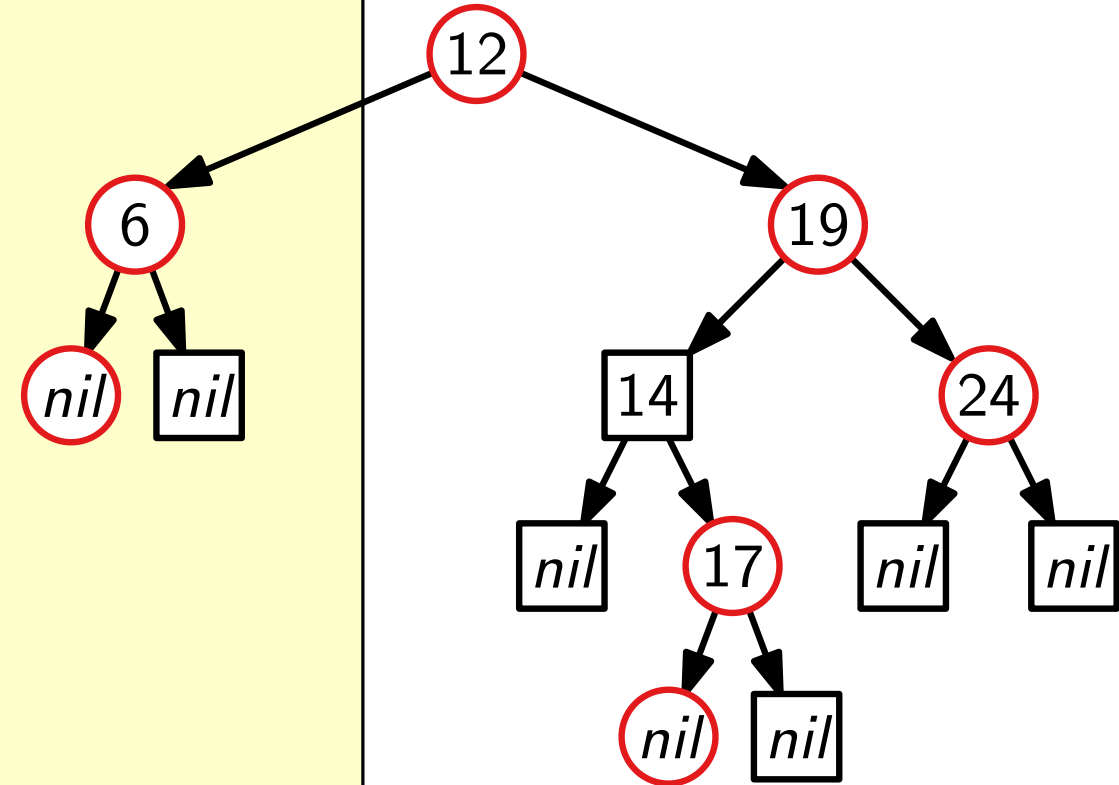


Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden

Rot-Schwarz-Eigenschaften:

(E1) Jeder Knoten ist entweder rot oder schwarz.



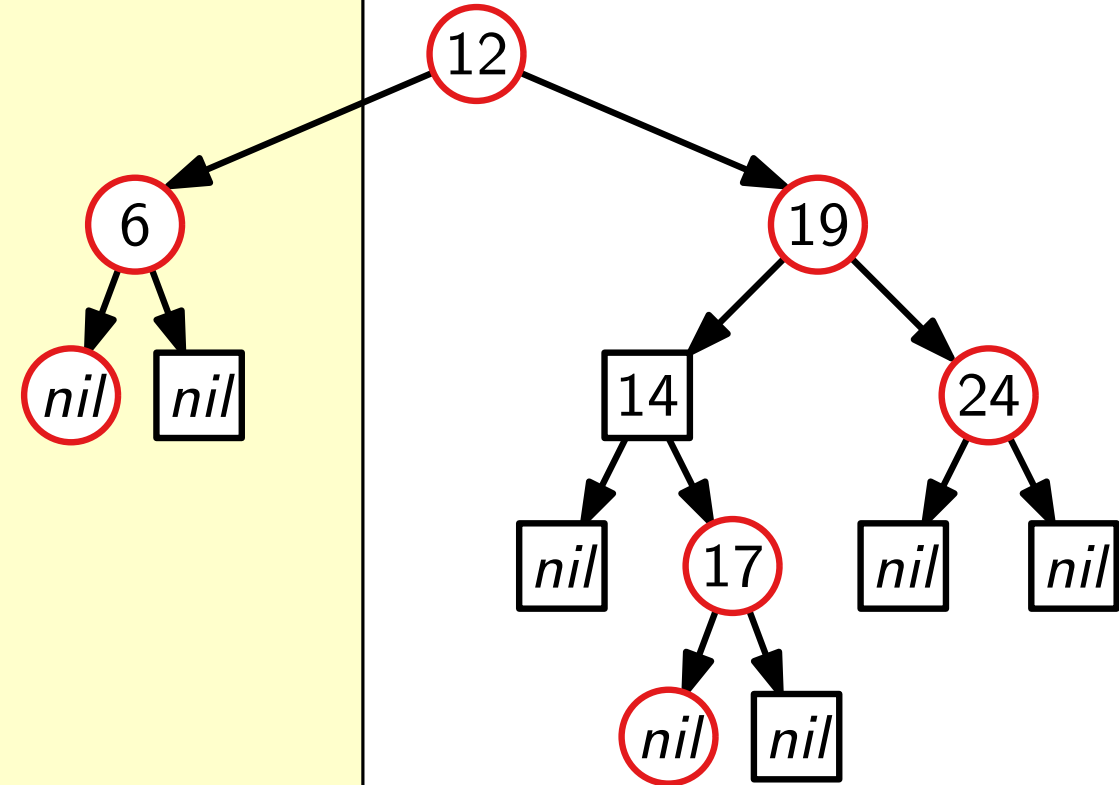
Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden

Rot-Schwarz-Eigenschaften:

(E1) Jeder Knoten ist entweder rot oder schwarz.

(E2) Die Wurzel ist schwarz.



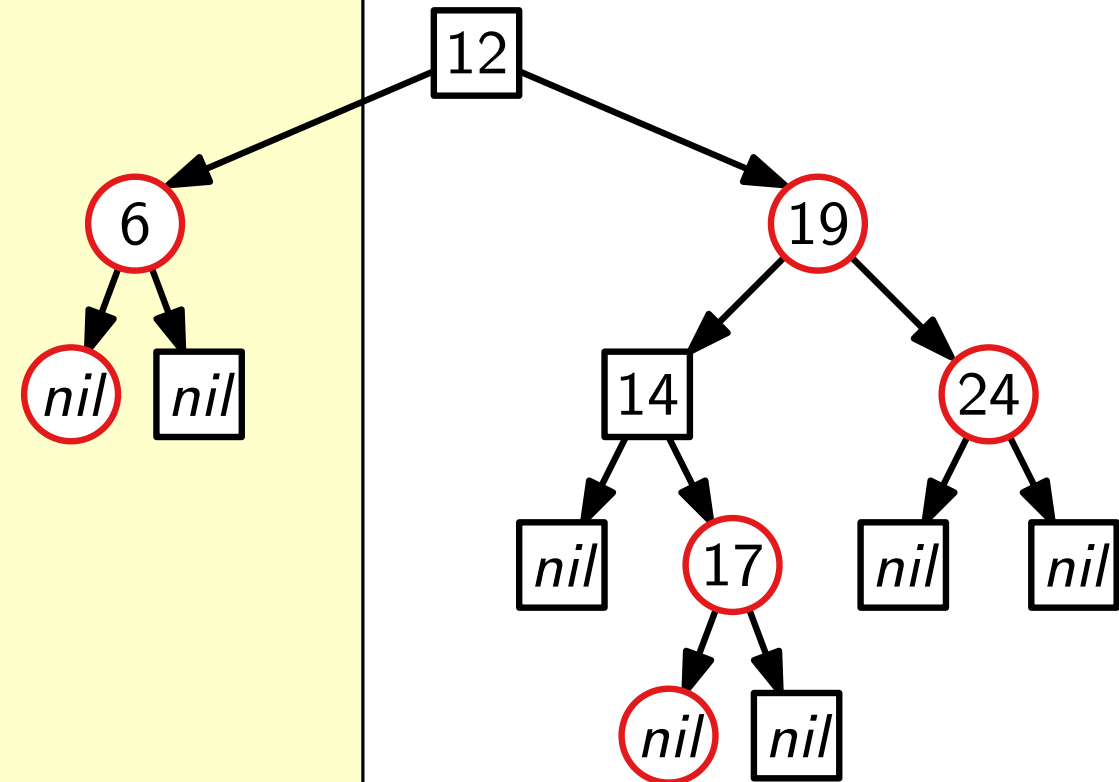
Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden

Rot-Schwarz-Eigenschaften:

(E1) Jeder Knoten ist entweder rot oder schwarz.

(E2) Die Wurzel ist schwarz.



Rot-Schwarz-Bäume: Eigenschaften

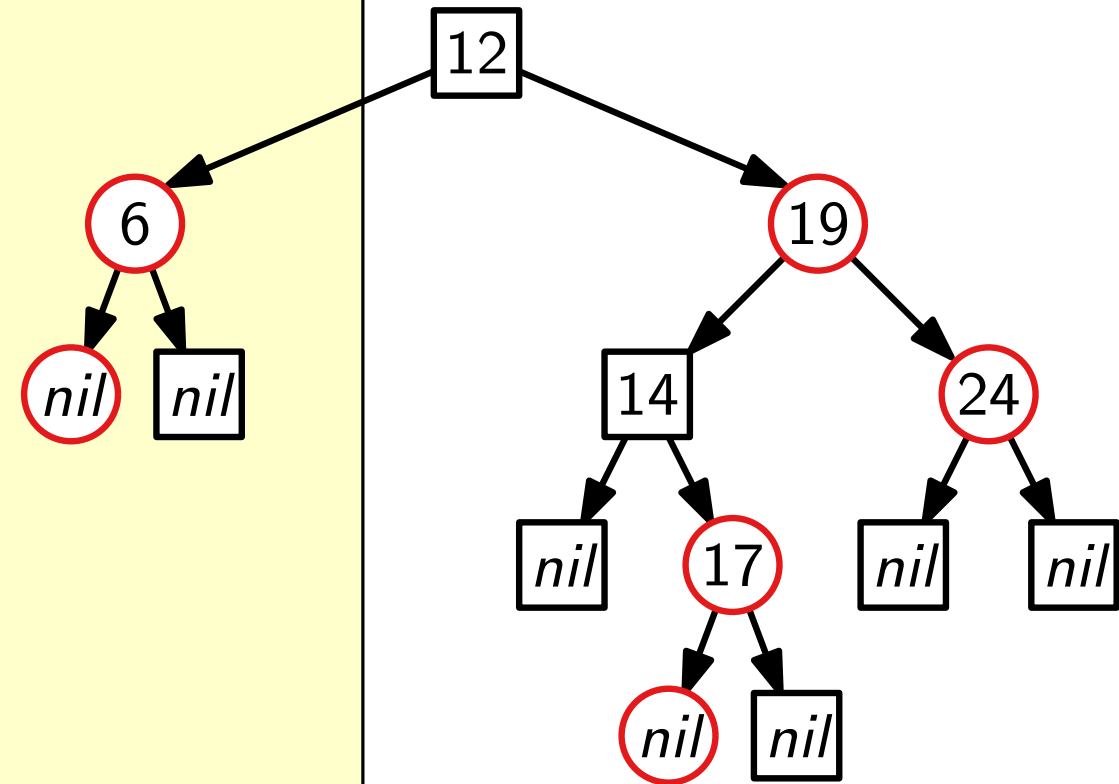
Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden

Rot-Schwarz-Eigenschaften:

(E1) Jeder Knoten ist entweder rot oder schwarz.

(E2) Die Wurzel ist schwarz.

(E3) Alle *nil*-Knoten sind schwarz. 



Rot-Schwarz-Bäume: Eigenschaften

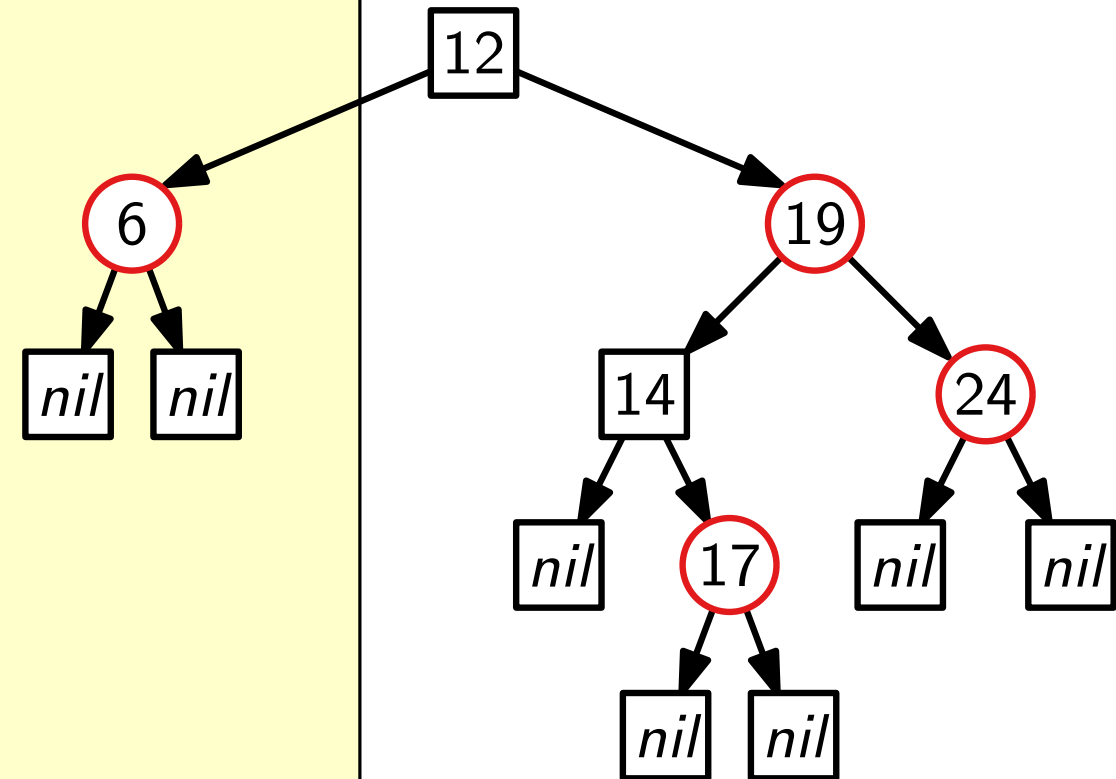
Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden

Rot-Schwarz-Eigenschaften:

(E1) Jeder Knoten ist entweder rot oder schwarz.

(E2) Die Wurzel ist schwarz.

(E3) Alle *nil*-Knoten sind schwarz. nil

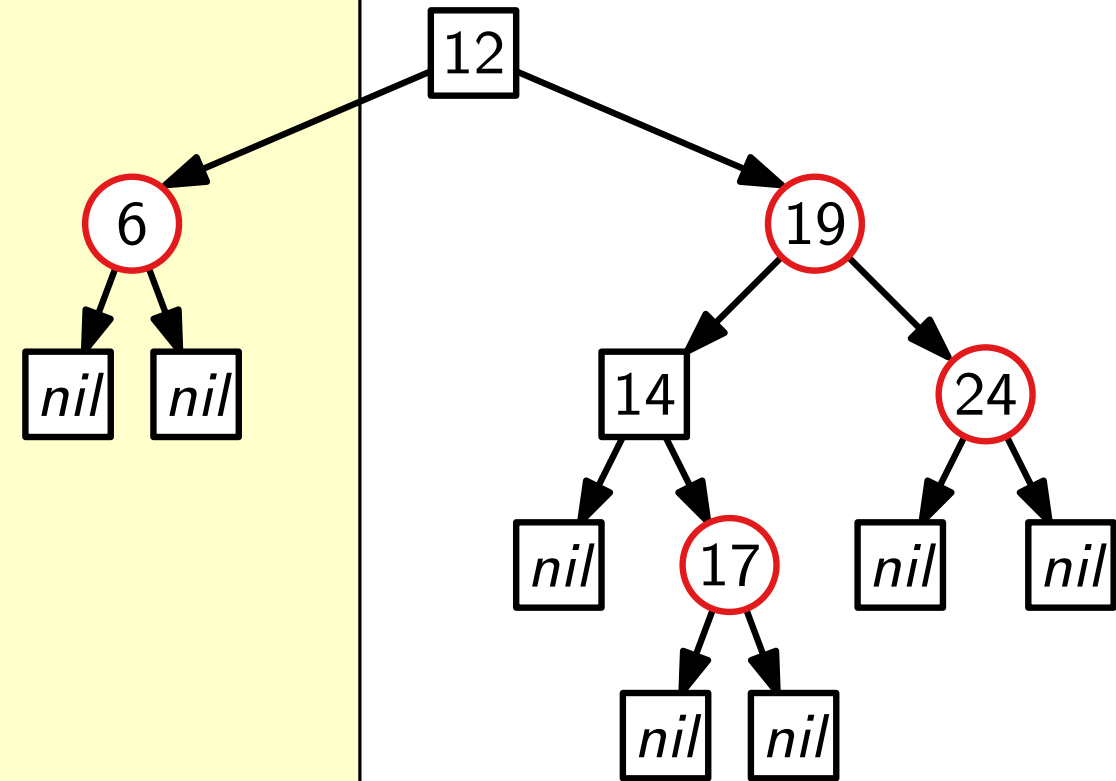


Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden

Rot-Schwarz-Eigenschaften:

- (E1)** Jeder Knoten ist entweder rot oder schwarz.
- (E2)** Die Wurzel ist schwarz.
- (E3)** Alle *nil*-Knoten sind schwarz. nil
- (E4)** Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.

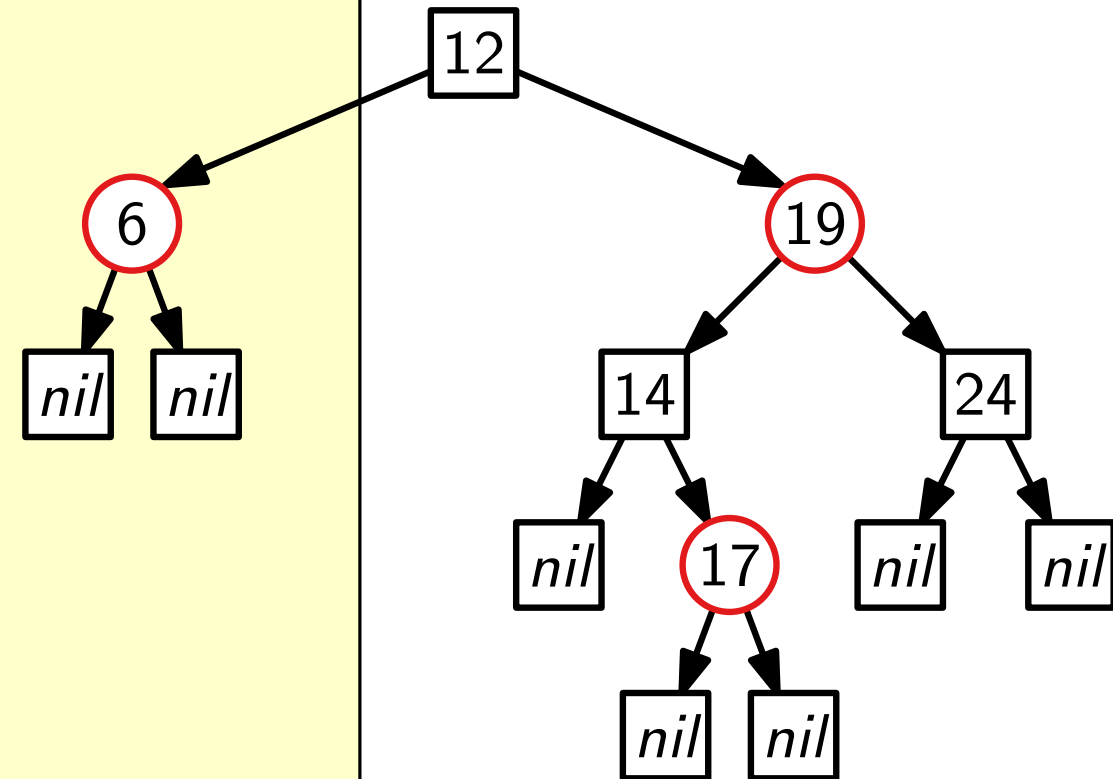


Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden

Rot-Schwarz-Eigenschaften:

- (E1)** Jeder Knoten ist entweder rot oder schwarz.
- (E2)** Die Wurzel ist schwarz.
- (E3)** Alle *nil*-Knoten sind schwarz. *nil*
- (E4)** Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.

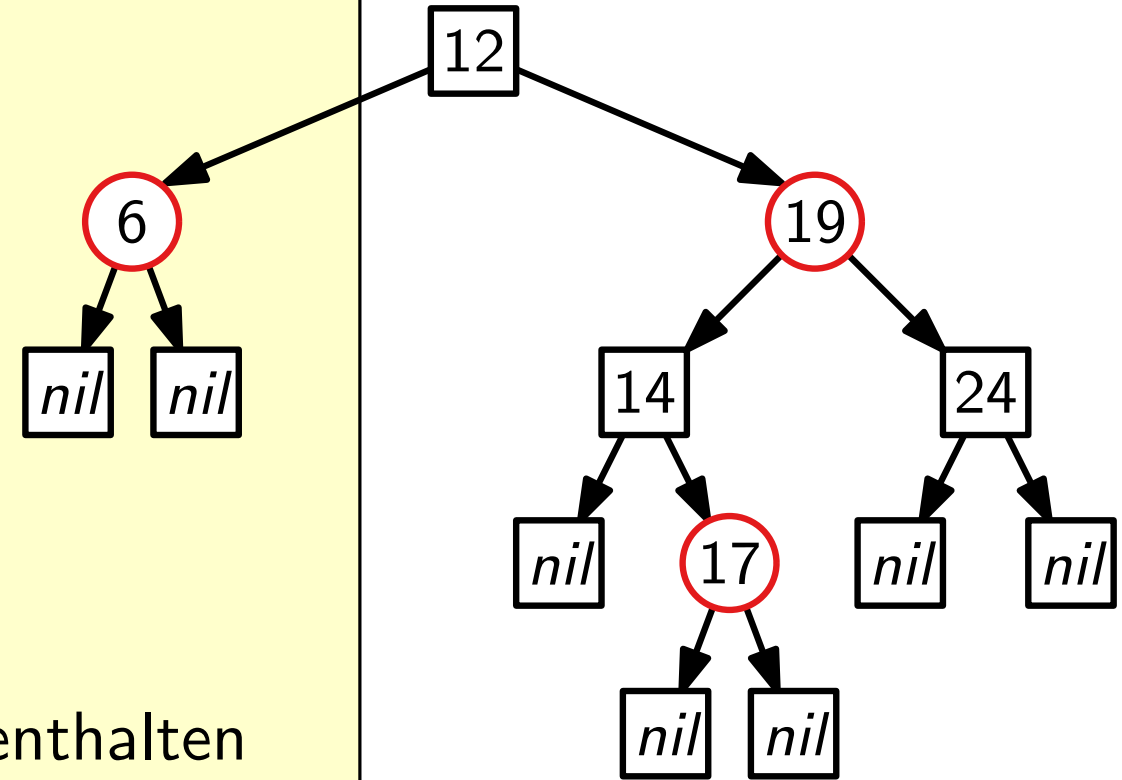


Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden

Rot-Schwarz-Eigenschaften:

- (E1) Jeder Knoten ist entweder rot oder schwarz.
- (E2) Die Wurzel ist schwarz.
- (E3) Alle *nil*-Knoten sind schwarz. *nil*
- (E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.

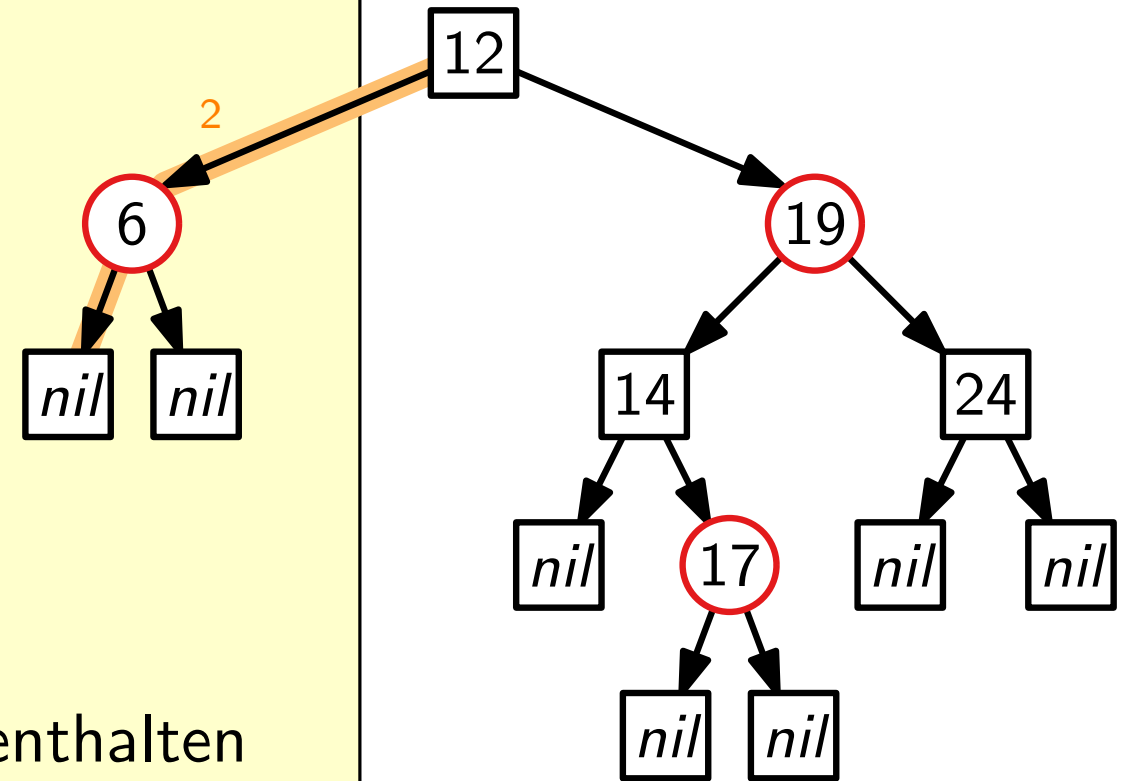


Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden

Rot-Schwarz-Eigenschaften:

- (E1) Jeder Knoten ist entweder rot oder schwarz.
- (E2) Die Wurzel ist schwarz.
- (E3) Alle *nil*-Knoten sind schwarz. nil
- (E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.

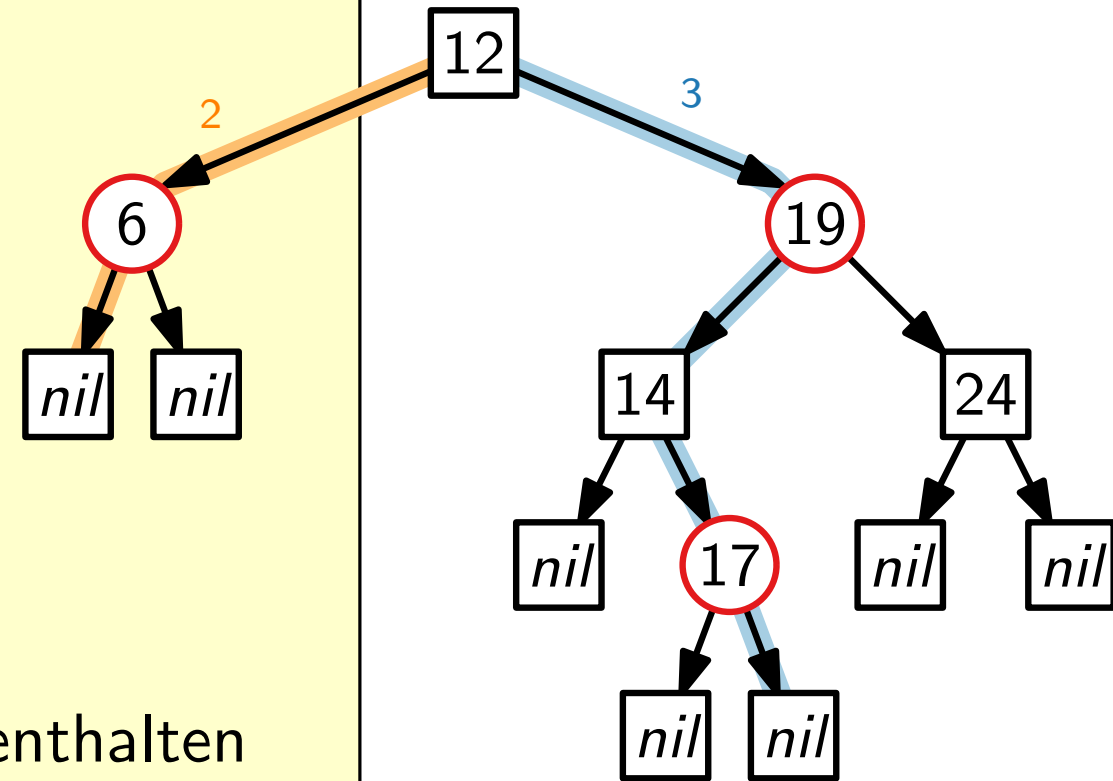


Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden

Rot-Schwarz-Eigenschaften:

- (E1) Jeder Knoten ist entweder rot oder schwarz.
- (E2) Die Wurzel ist schwarz.
- (E3) Alle *nil*-Knoten sind schwarz. nil
- (E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.

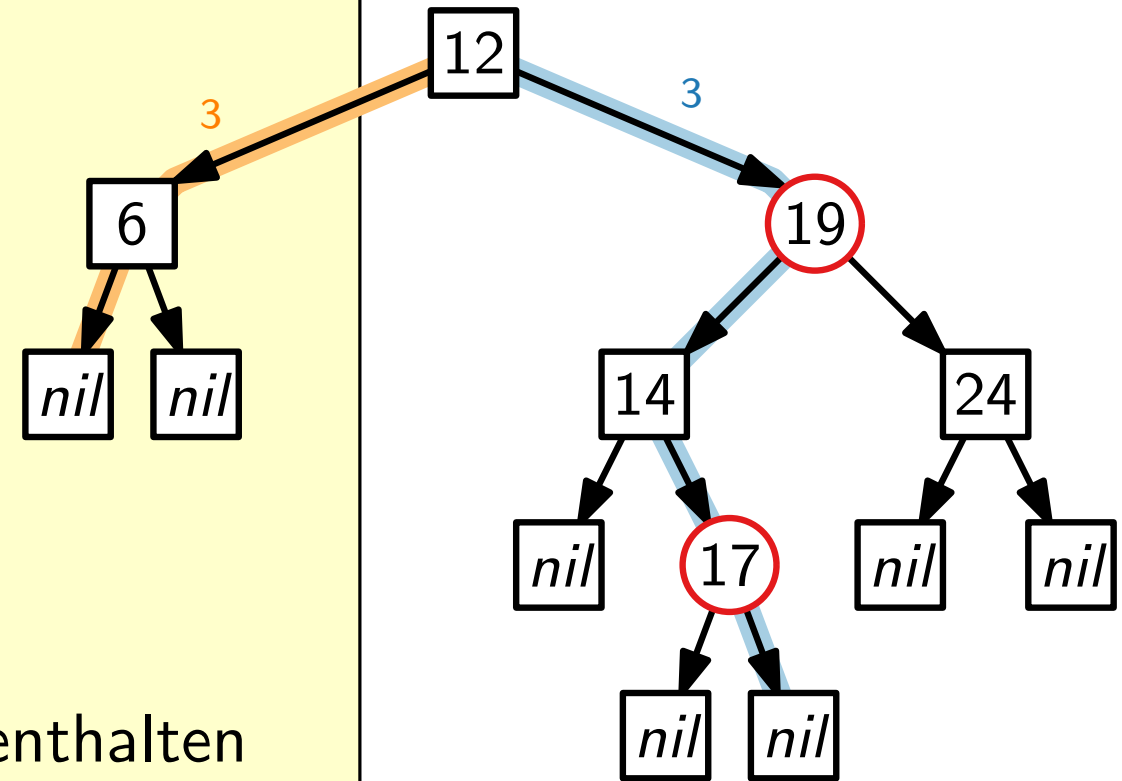


Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden

Rot-Schwarz-Eigenschaften:

- (E1)** Jeder Knoten ist entweder rot oder schwarz.
- (E2)** Die Wurzel ist schwarz.
- (E3)** Alle *nil*-Knoten sind schwarz. nil
- (E4)** Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.
- (E5)** Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.

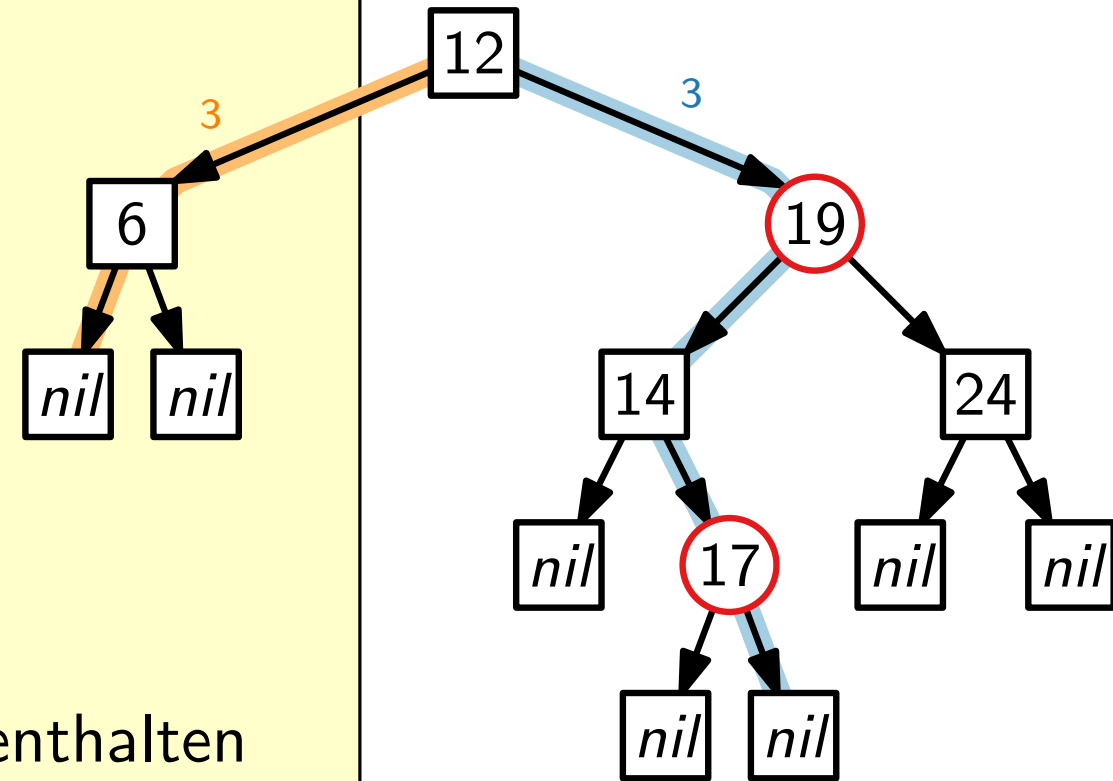


Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden

Rot-Schwarz-Eigenschaften:

- (E1)** Jeder Knoten ist entweder rot oder schwarz.
- (E2)** Die Wurzel ist schwarz.
- (E3)** Alle *nil*-Knoten sind schwarz. nil
- (E4)** Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.
- (E5)** Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.



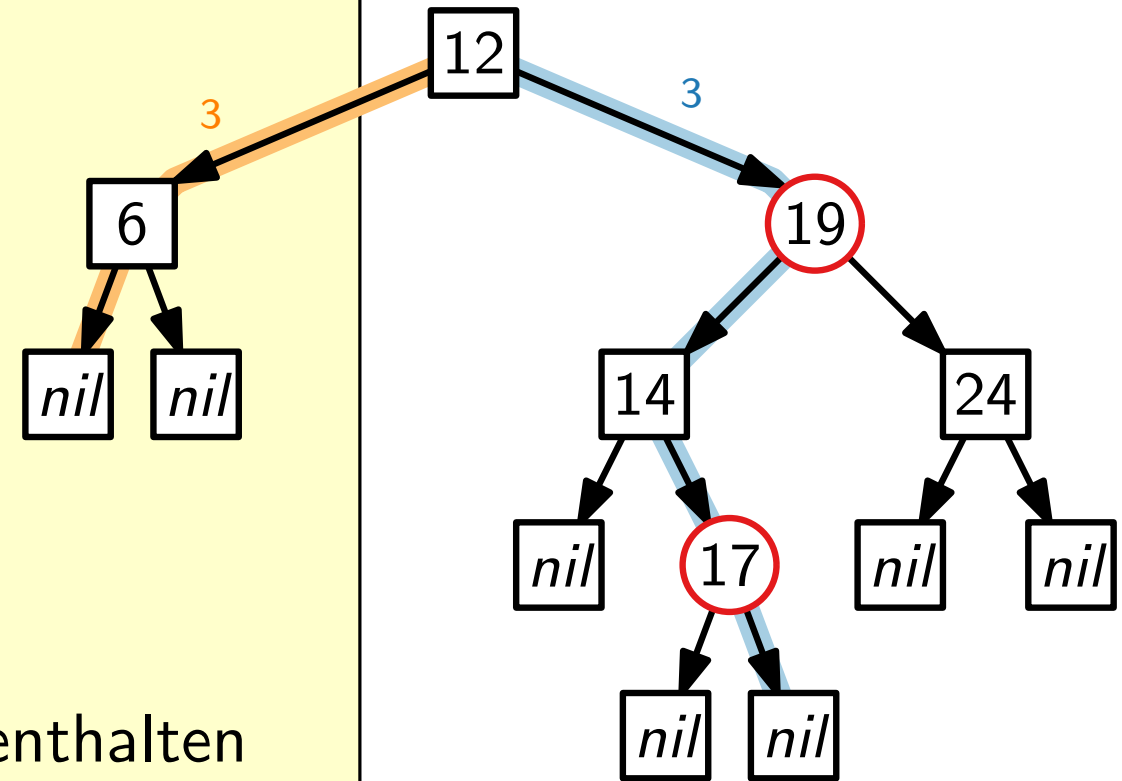
Aus **(E4)** folgt:

Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden

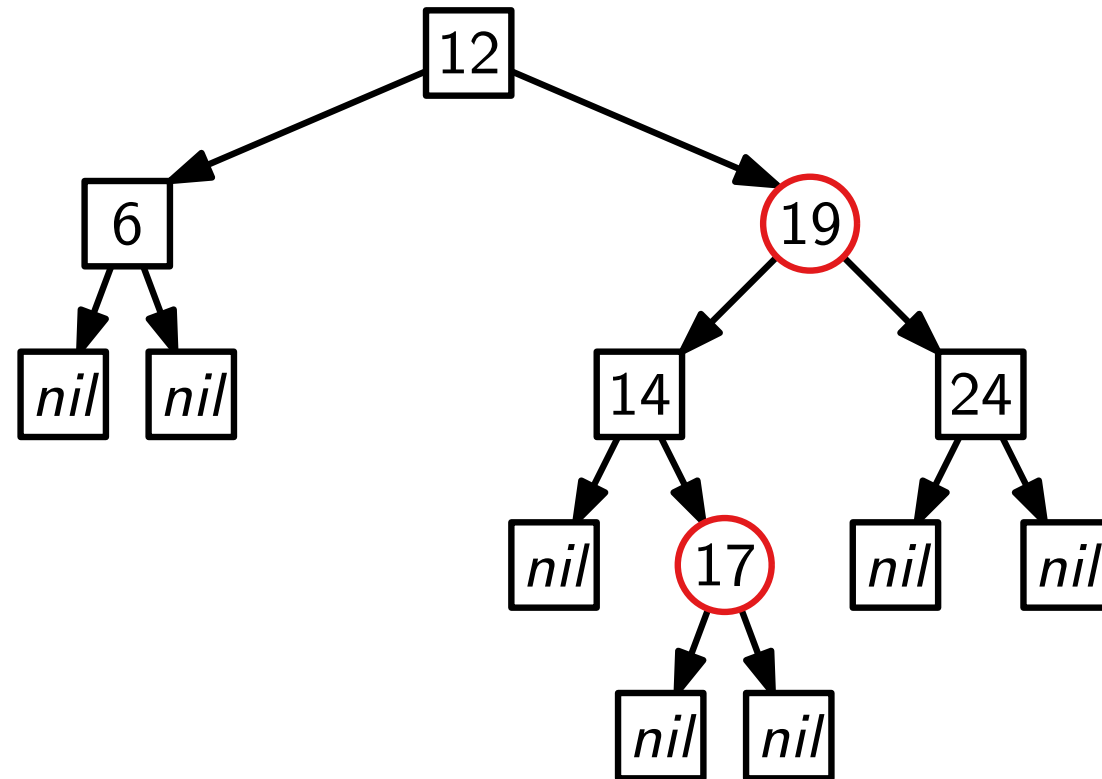
Rot-Schwarz-Eigenschaften:

- (E1)** Jeder Knoten ist entweder rot oder schwarz.
- (E2)** Die Wurzel ist schwarz.
- (E3)** Alle *nil*-Knoten sind schwarz. nil
- (E4)** Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.
- (E5)** Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.

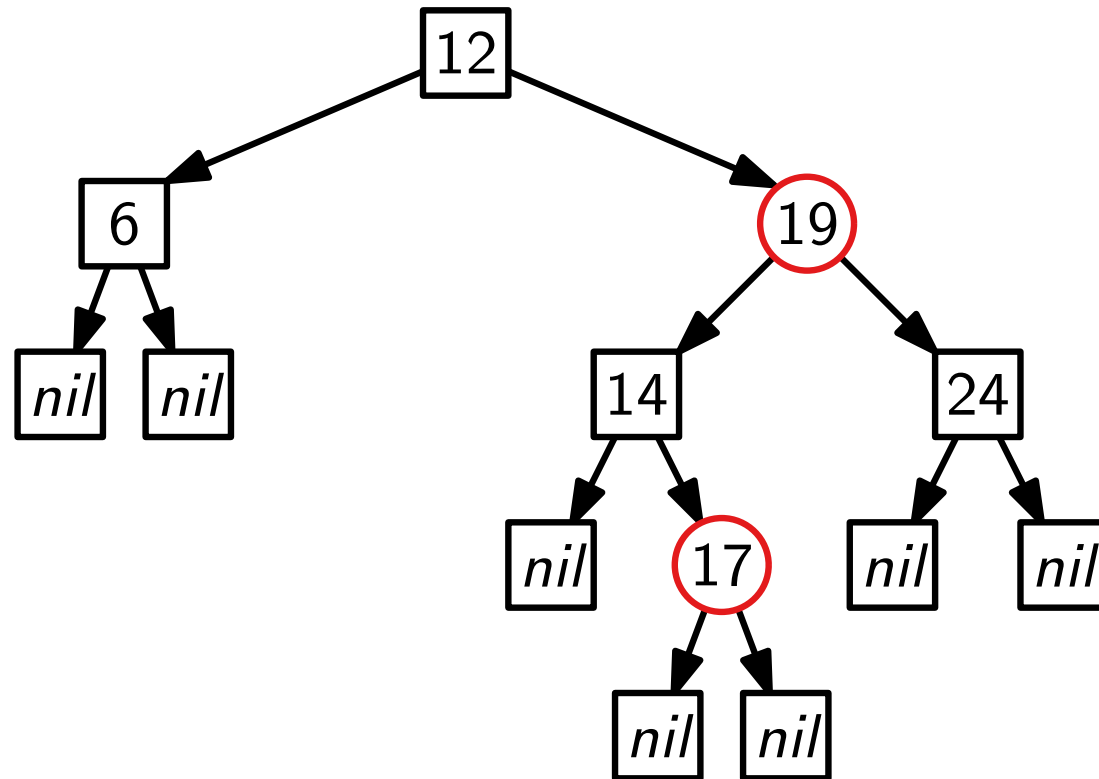
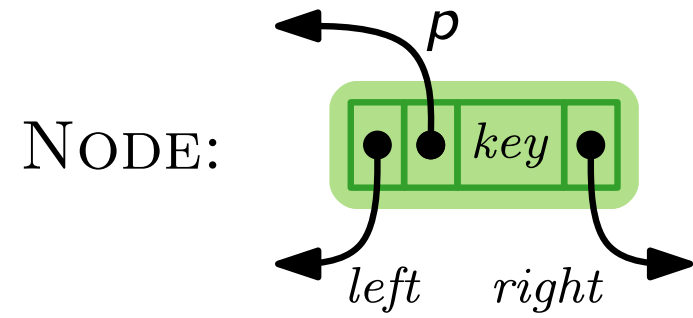


Aus **(E4)** folgt: Auf keinem Wurzel-Blatt-Pfad folgen zwei rote Knoten direkt aufeinander.

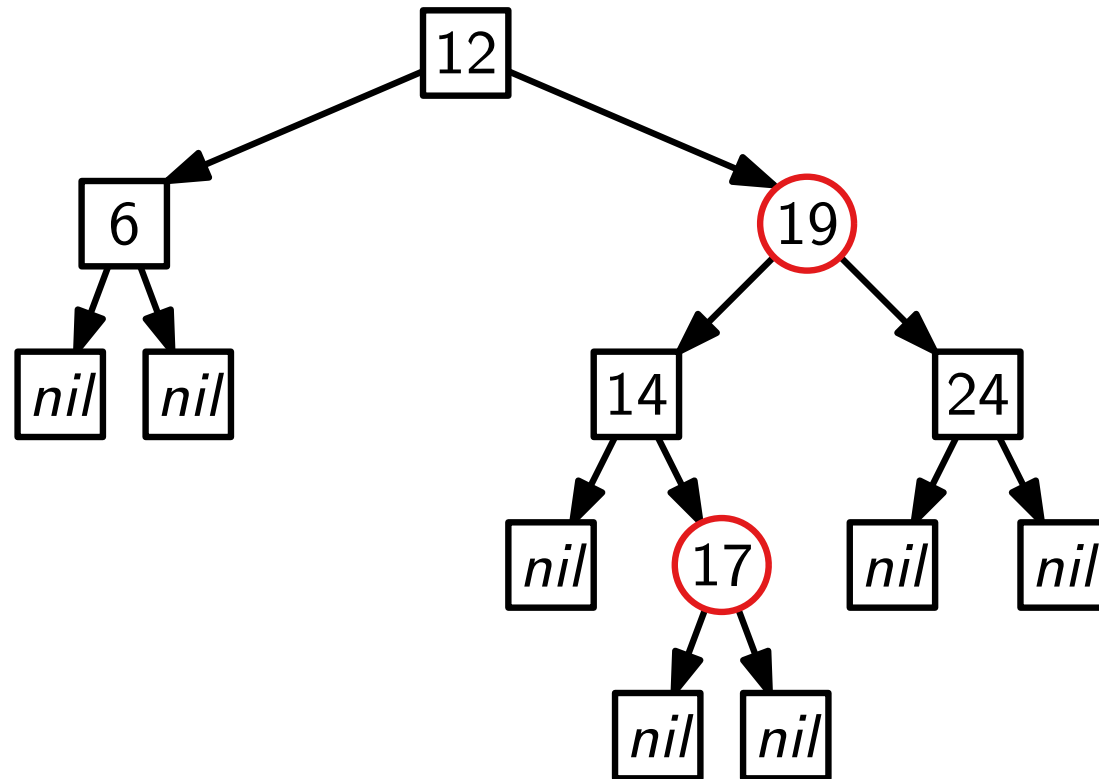
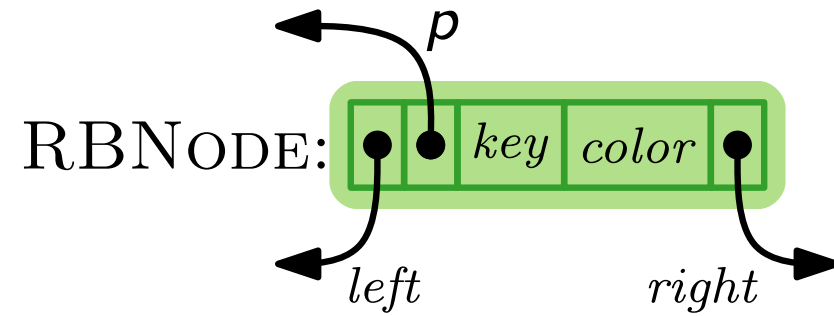
Technisches Detail



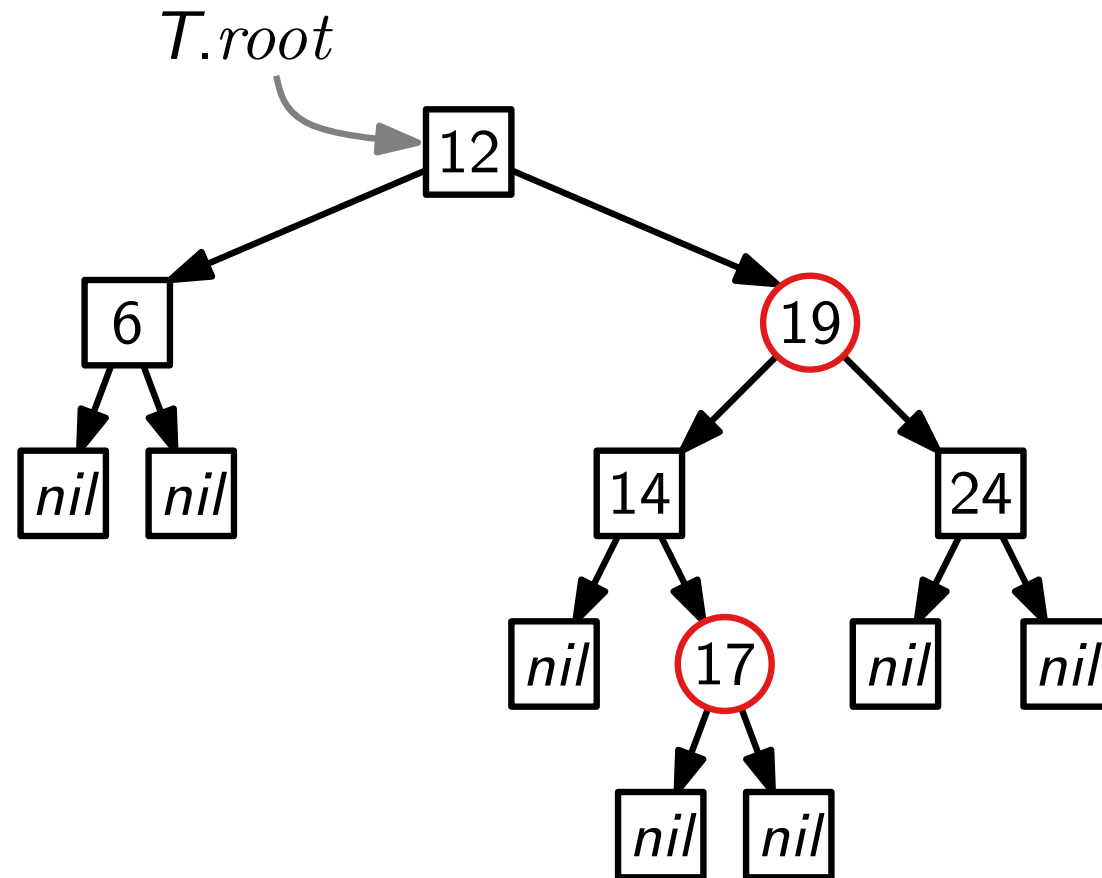
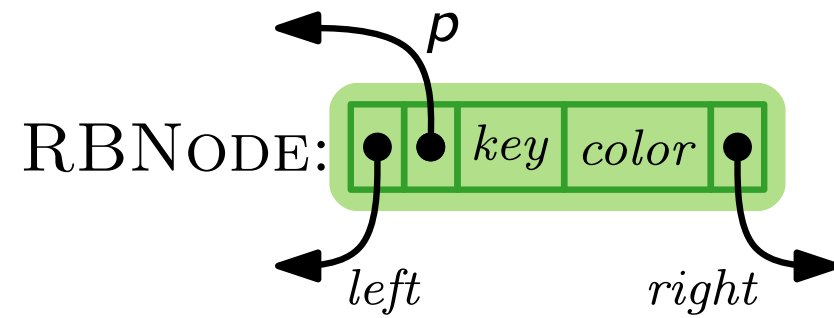
Technisches Detail



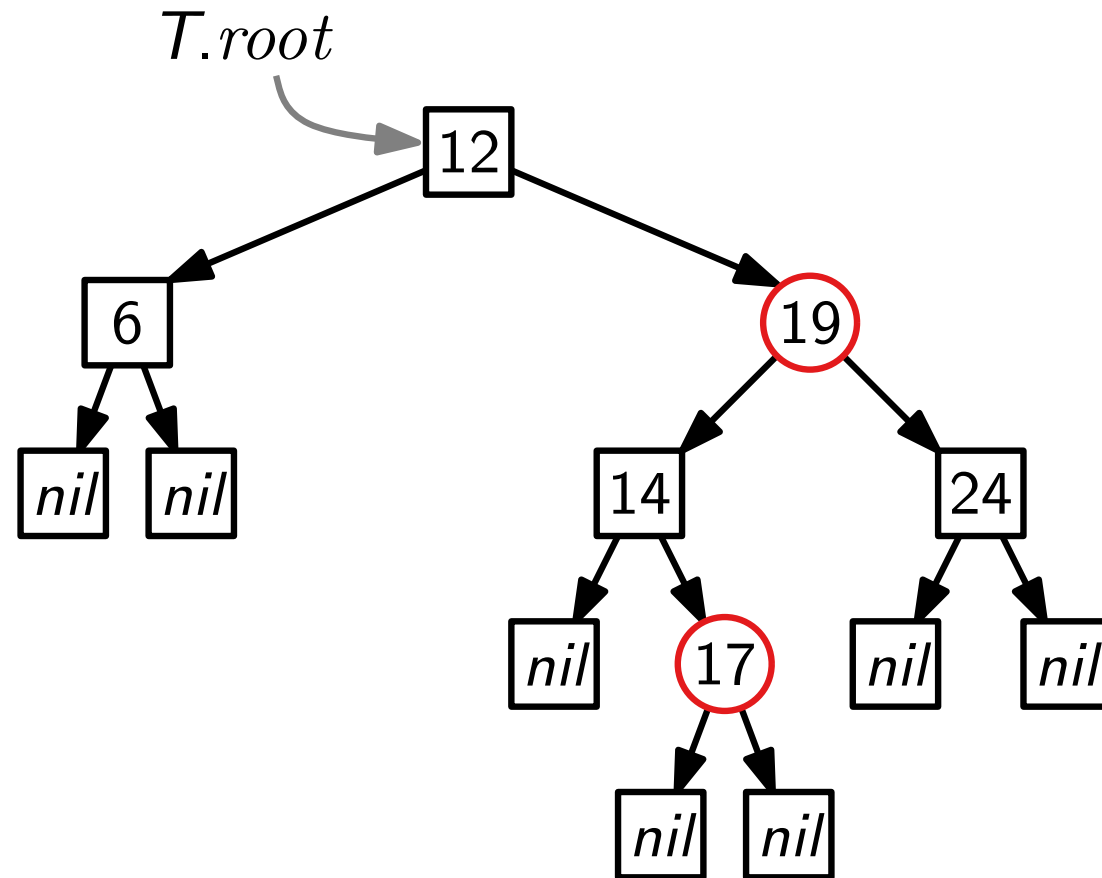
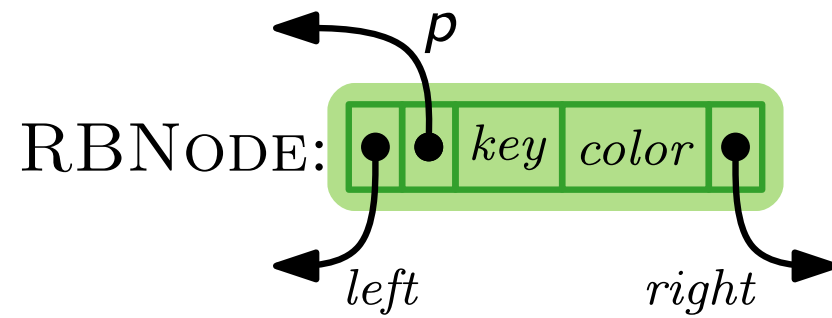
Technisches Detail



Technisches Detail



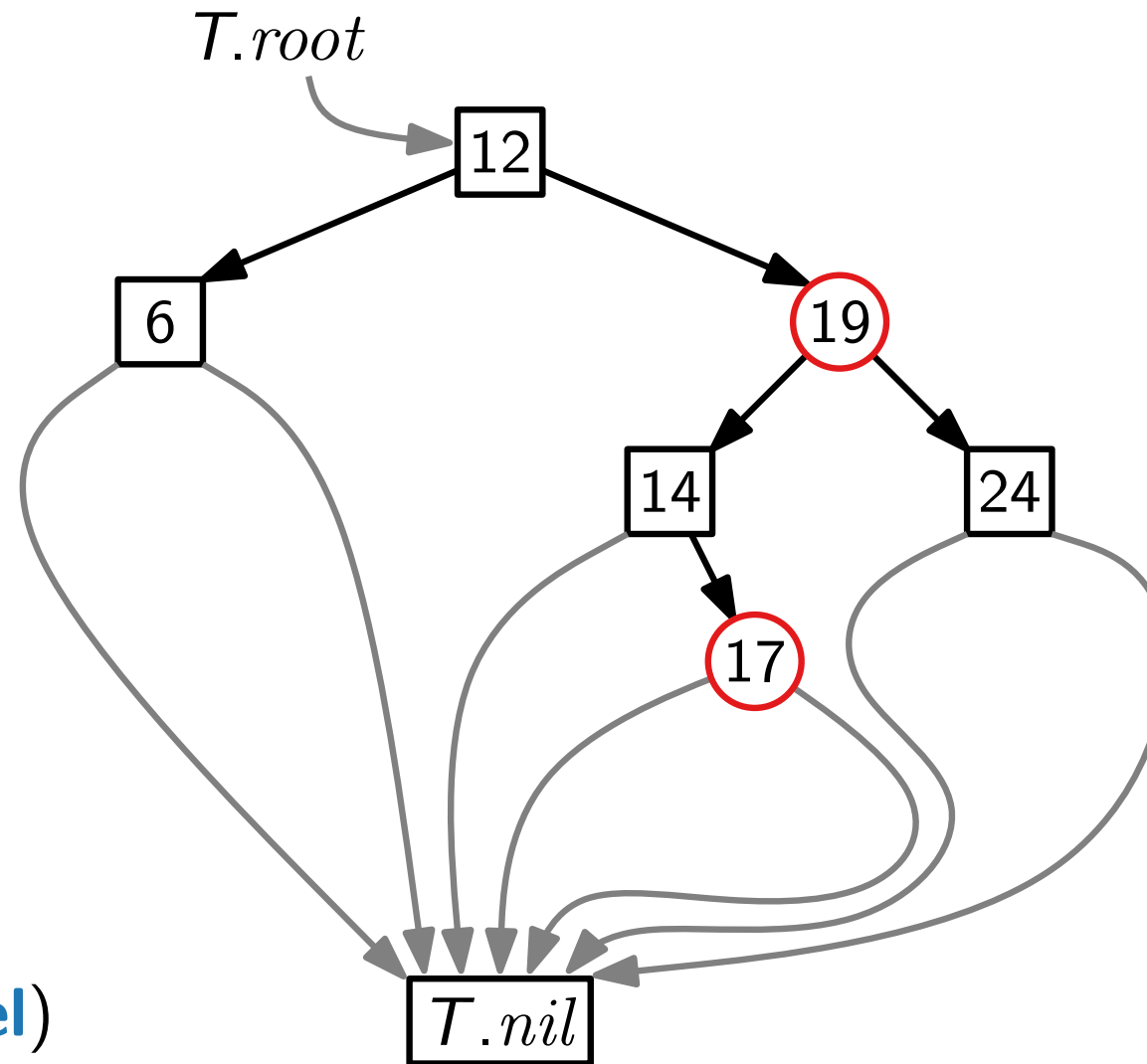
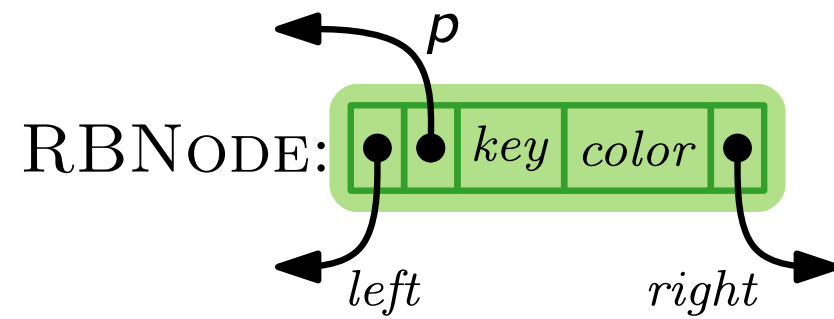
Technisches Detail



Dummy-Knoten (engl. **sentinel**)

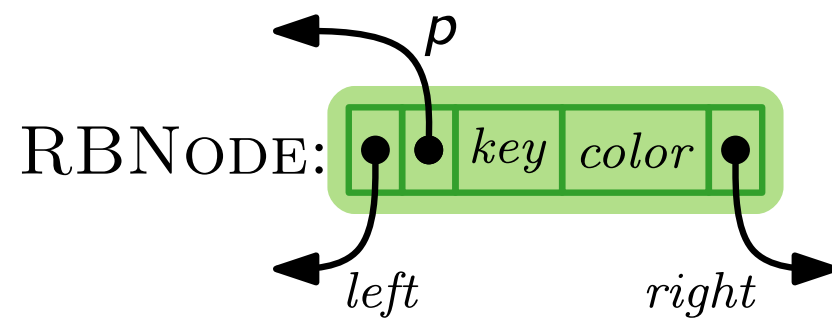
$T.nil$

Technisches Detail

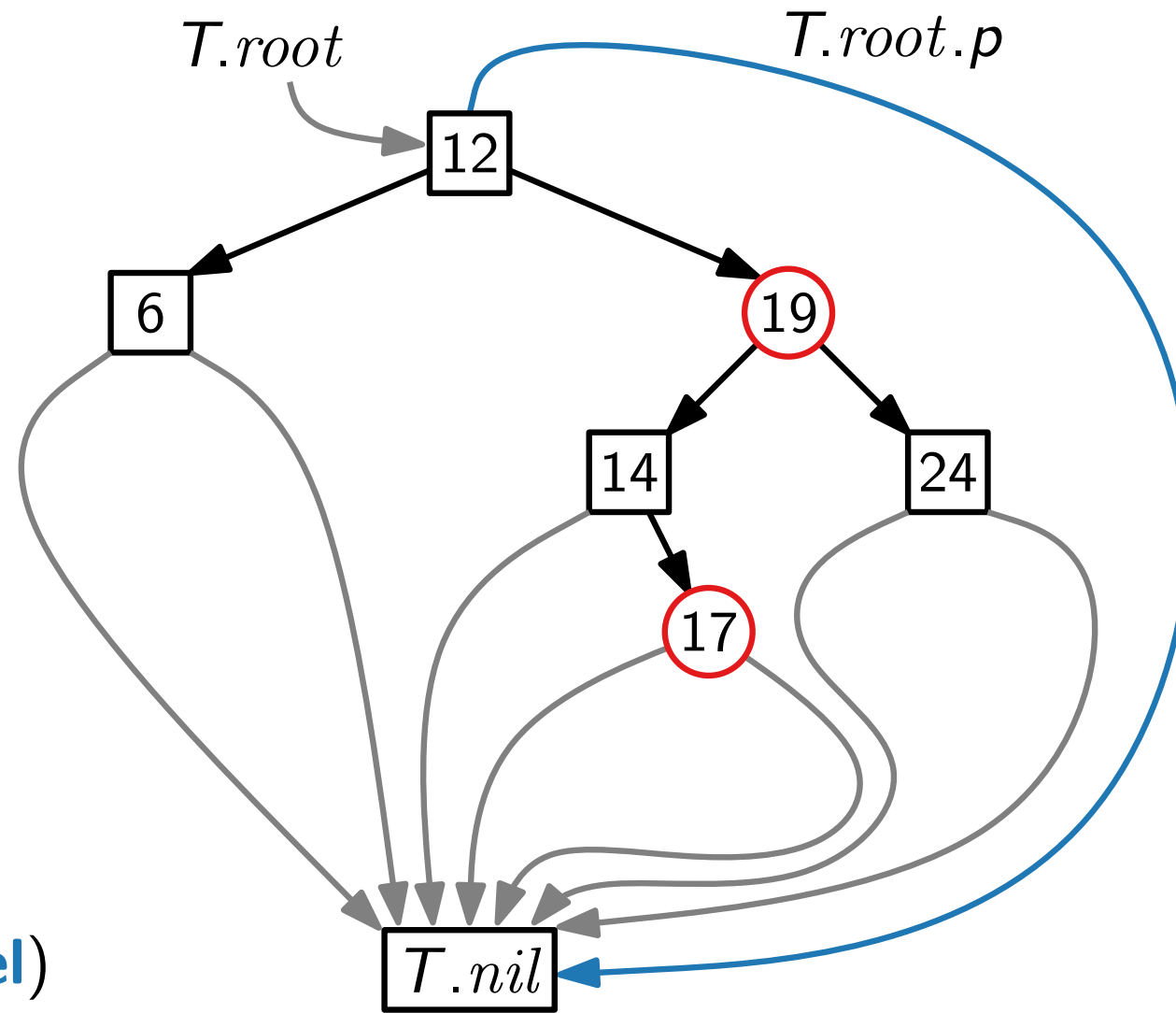


Dummy-Knoten (engl. **sentinel**)

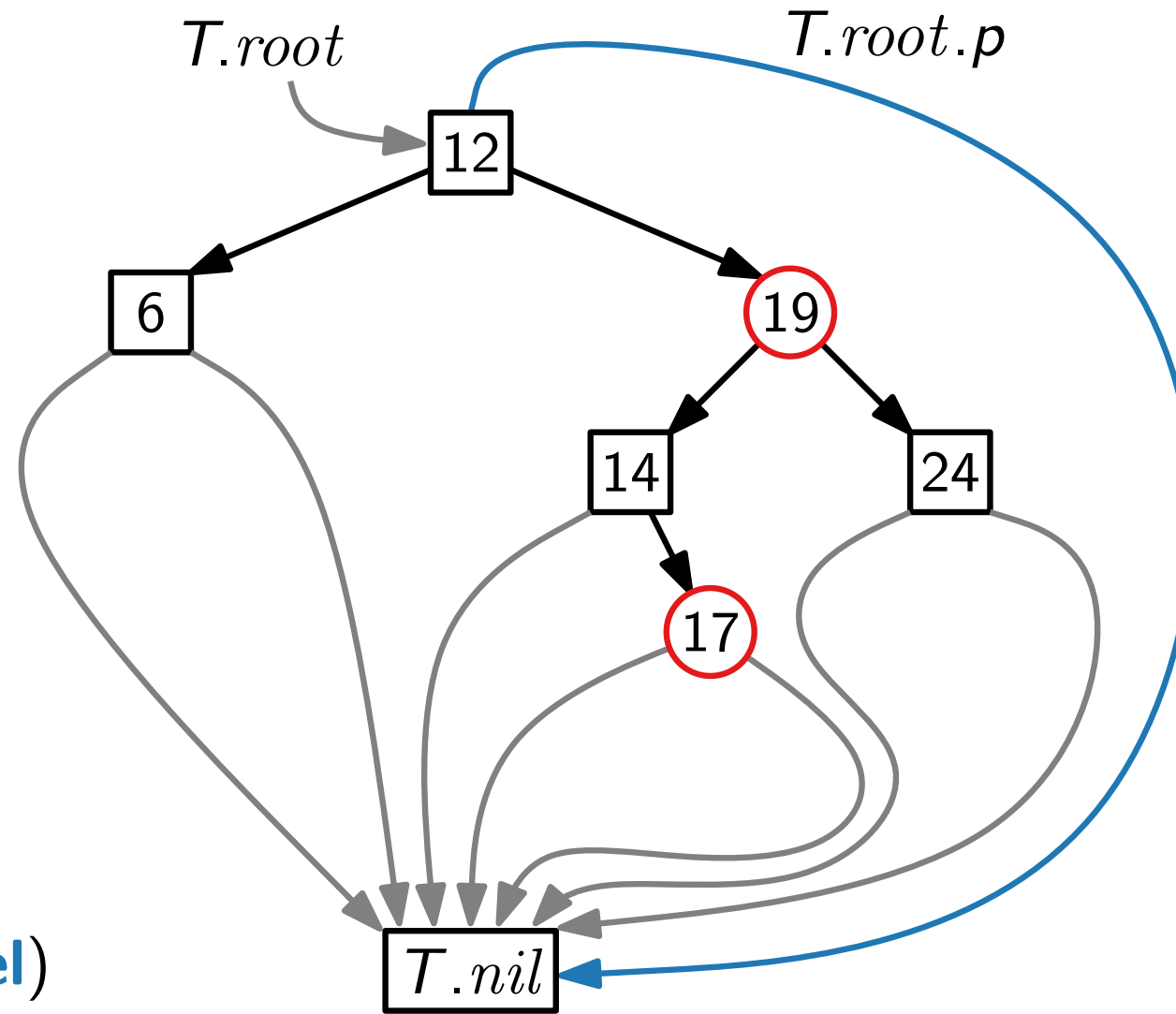
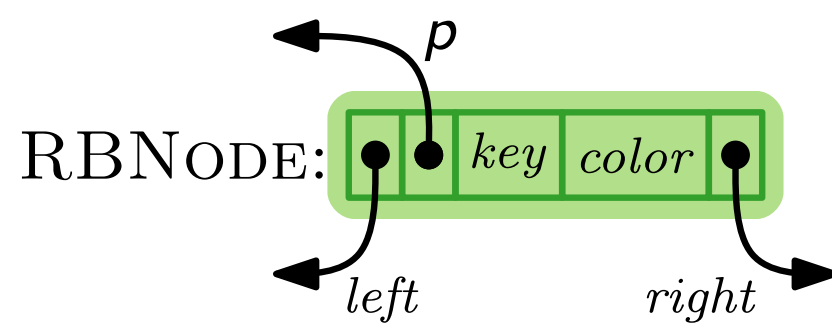
Technisches Detail



Dummy-Knoten (engl. **sentinel**)



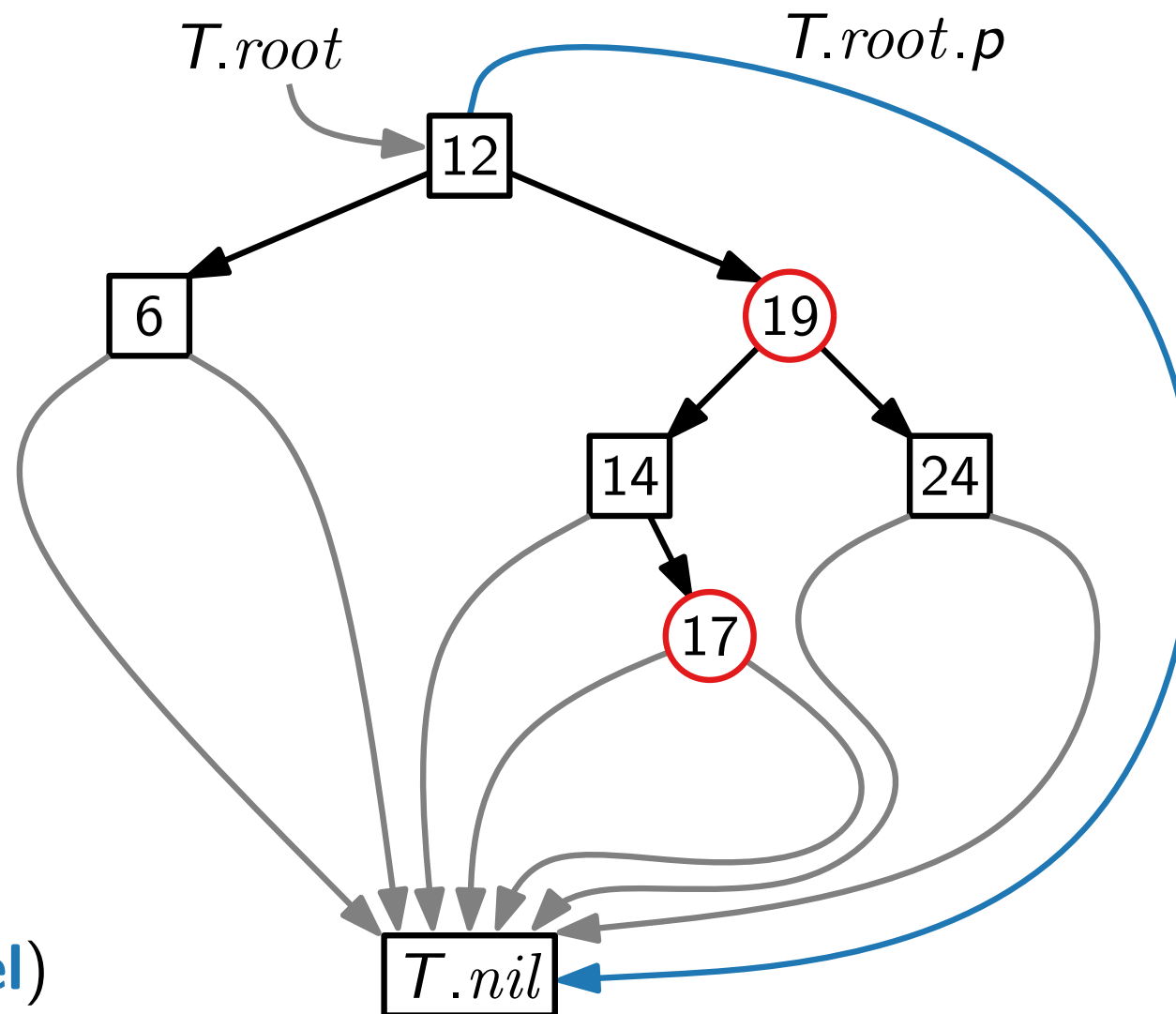
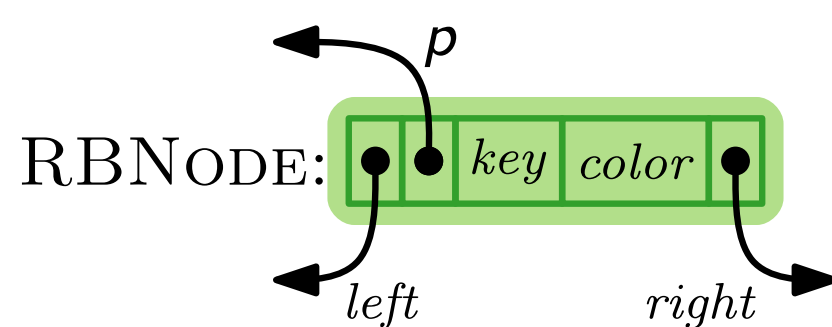
Technisches Detail



Dummy-Knoten (engl. **sentinel**)

Zweck: um Baum-Operationen prägnanter aufschreiben zu können.

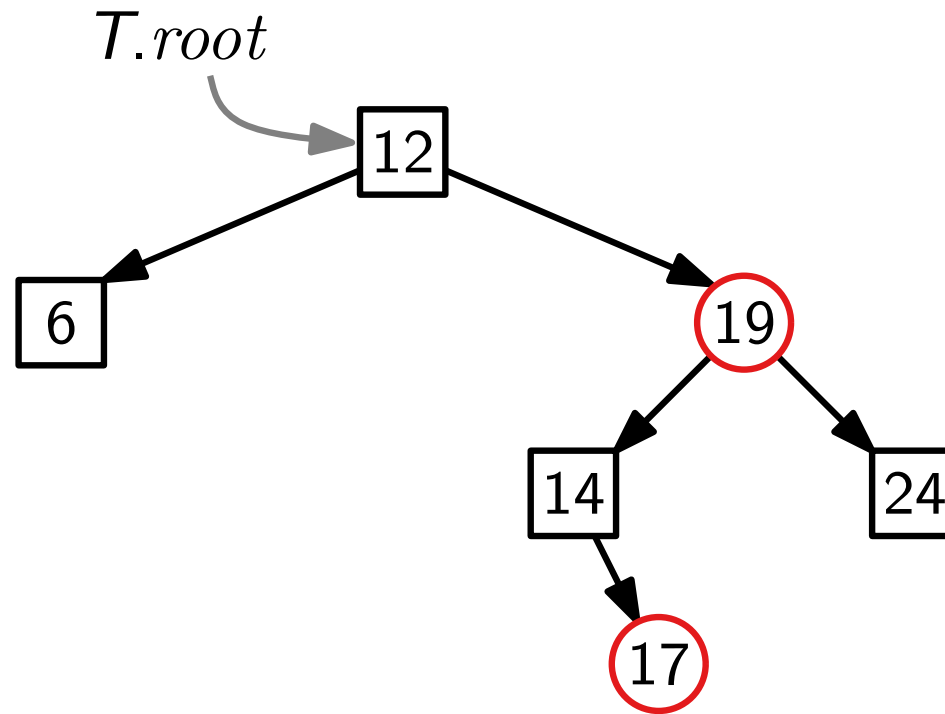
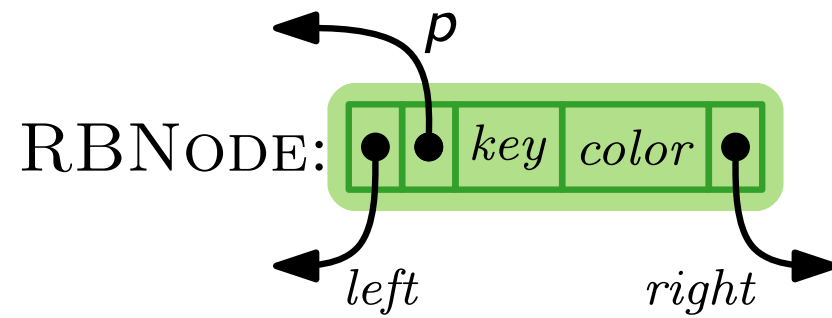
Technisches Detail



Dummy-Knoten (engl. **sentinel**)

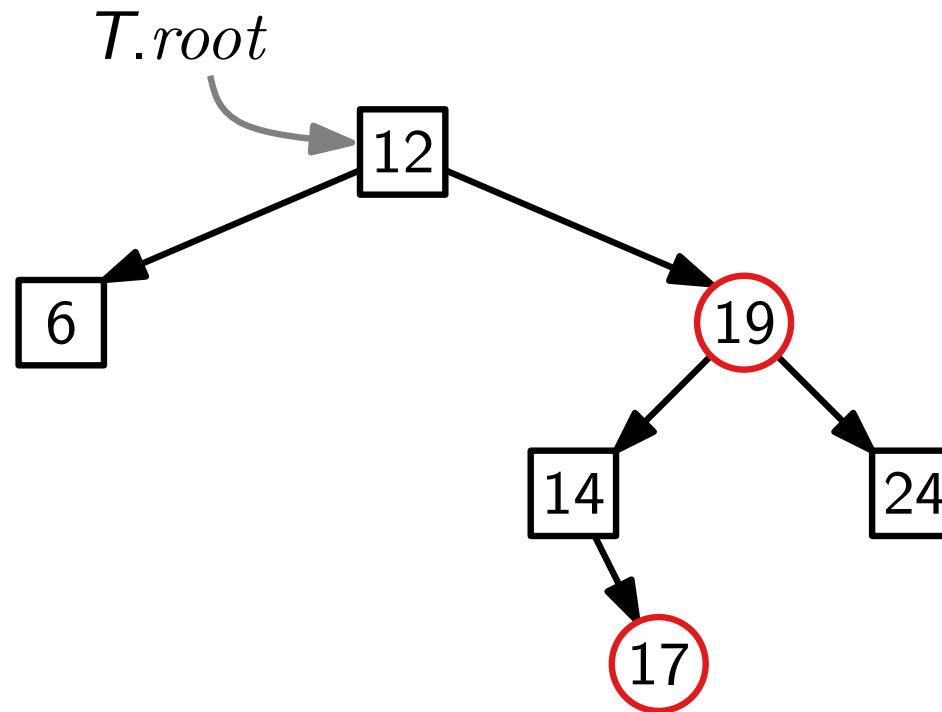
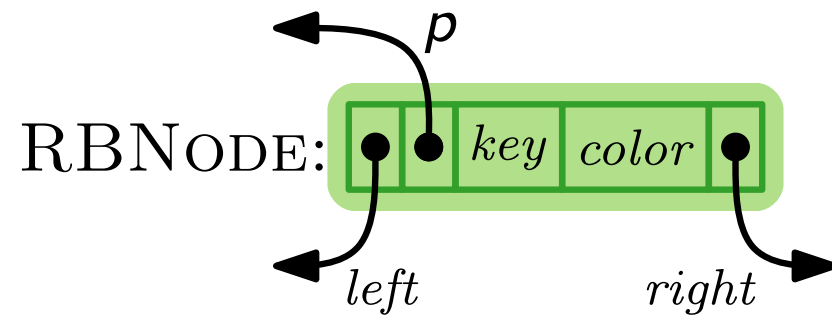
Zweck: um Baum-Operationen prägnanter aufschreiben zu können.
(Wir zeichnen den Dummy-Knoten i.A. nicht.)

Technisches Detail



Zweck: um Baum-Operationen prägnanter aufschreiben zu können.
(Wir zeichnen den Dummy-Knoten i.A. nicht.)

Technisches Detail



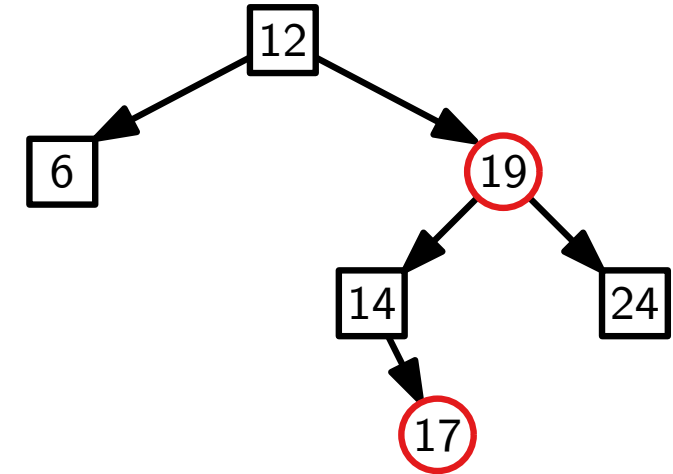
Achtung: **(E3)** ist hier erfüllt, weil es sich auf den Dummy-Knoten bezieht!

(E3) Alle *nil*-Knoten sind schwarz. *nil*

Zweck: um Baum-Operationen prägnanter aufschreiben zu können.
(Wir zeichnen den Dummy-Knoten i.A. nicht.)

(Schwarz-) Höhe

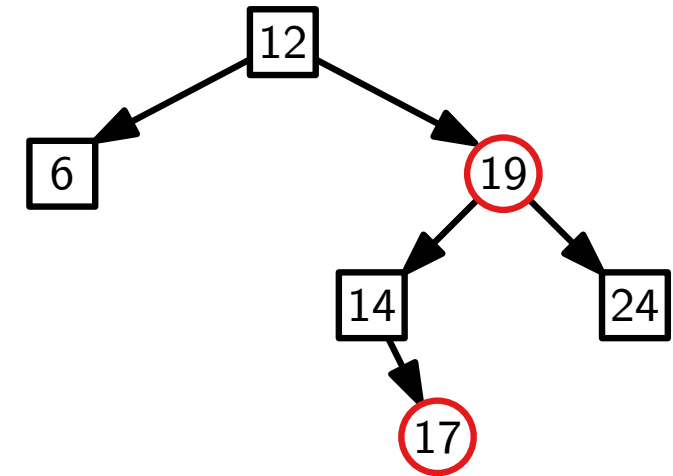
Definition. Die **Länge** eines Pfades ist die Anzahl seiner Kanten.



(Schwarz-) Höhe

Definition. Die **Länge** eines Pfades ist die Anzahl seiner Kanten.

Definition. Sei T ein Baum.
Knoten u ist **unter** Knoten v , wenn u in dem Teilbaum T_v von T mit Wurzel v enthalten ist.

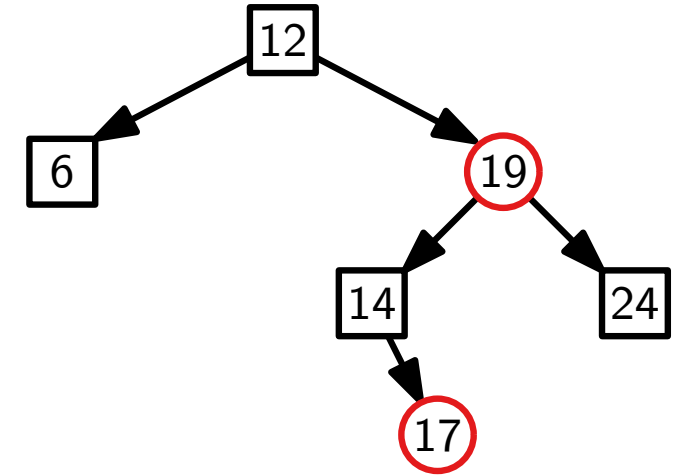


(Schwarz-) Höhe

Definition. Die **Länge** eines Pfades ist die Anzahl seiner Kanten.

Definition. Sei T ein Baum.
Knoten u ist **unter** Knoten v , wenn u in dem Teilbaum T_v von T mit Wurzel v enthalten ist.

Beispiel: 17 ist unter 19

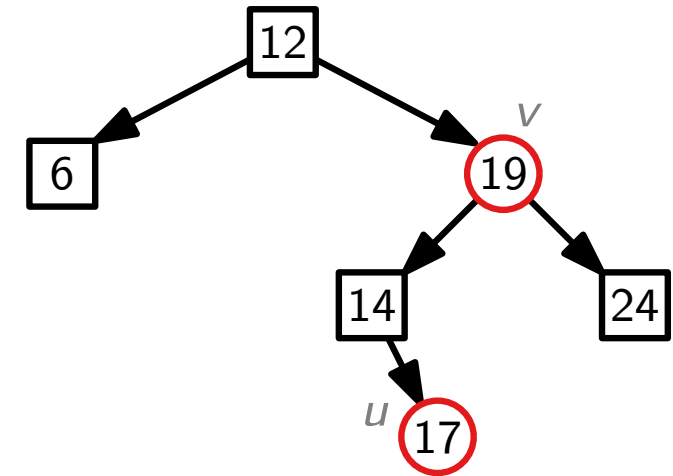


(Schwarz-) Höhe

Definition. Die **Länge** eines Pfades ist die Anzahl seiner Kanten.

Definition. Sei T ein Baum.
Knoten u ist **unter** Knoten v , wenn u in dem Teilbaum T_v von T mit Wurzel v enthalten ist.

Beispiel: 17 ist unter 19

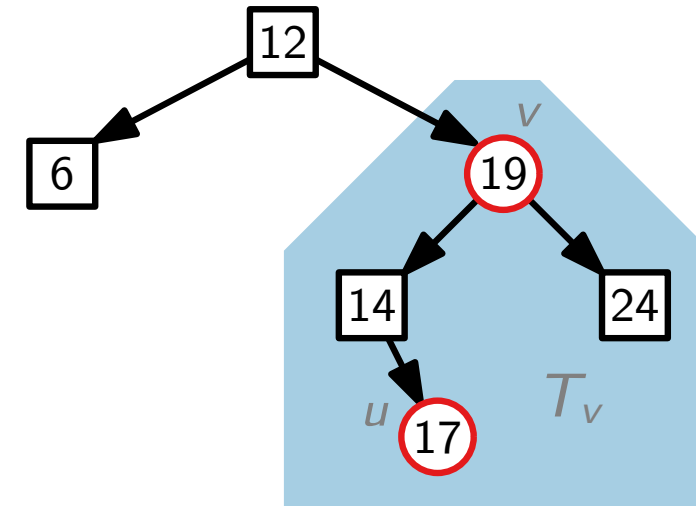


(Schwarz-) Höhe

Definition. Die **Länge** eines Pfades ist die Anzahl seiner Kanten.

Definition. Sei T ein Baum.
Knoten u ist **unter** Knoten v , wenn u in dem Teilbaum T_v von T mit Wurzel v enthalten ist.

Beispiel: 17 ist unter 19

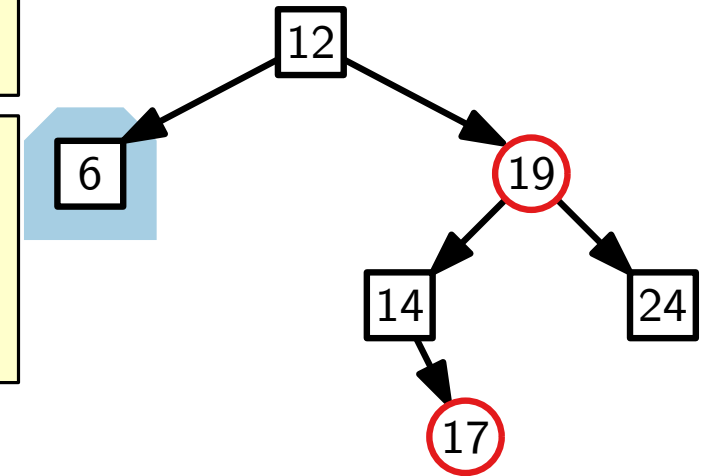


(Schwarz-) Höhe

Definition. Die **Länge** eines Pfades ist die Anzahl seiner Kanten.

Definition. Sei T ein Baum.
Knoten u ist **unter** Knoten v , wenn u in dem Teilbaum T_v von T mit Wurzel v enthalten ist.

Beispiel: 17 ist unter 19 , 14 ist **nicht** unter 6.



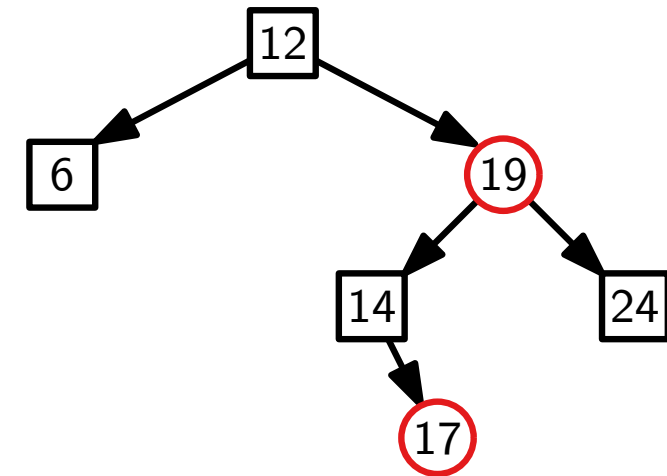
(Schwarz-) Höhe

Definition. Die **Länge** eines Pfades ist die Anzahl seiner Kanten.

Definition. Sei T ein Baum.
Knoten u ist **unter** Knoten v , wenn u in dem Teilbaum T_v von T mit Wurzel v enthalten ist.

Beispiel: 17 ist unter 19, 14 ist **nicht** unter 6.

Definition. Die **Höhe** eines Knotens v ist die Länge eines längsten Pfades von v zu einem Blatt unter v .



(Schwarz-) Höhe

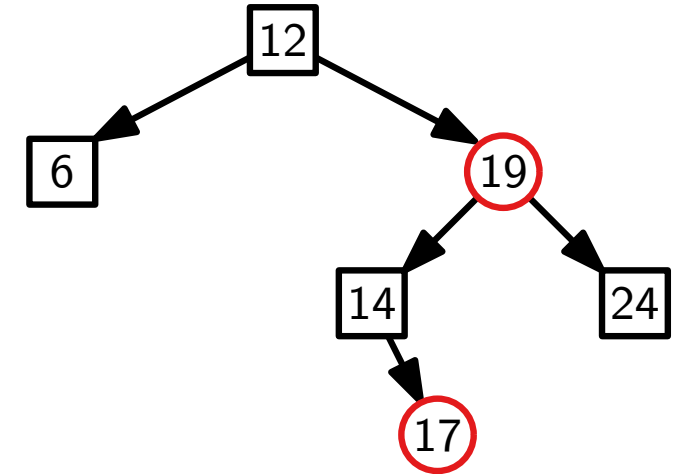
Definition. Die **Länge** eines Pfades ist die Anzahl seiner Kanten.

Definition. Sei T ein Baum.
Knoten u ist **unter** Knoten v , wenn u in dem Teilbaum T_v von T mit Wurzel v enthalten ist.

Beispiel: 17 ist unter 19 , 14 ist **nicht** unter 6.

Definition. Die **Höhe** eines Knotens v ist die Länge eines längsten Pfads von v zu einem Blatt unter v .

Definition'. Die **Höhe** $\text{Höhe}(v)$ eines Knotens v ist die Anz. der Knoten auf dem längsten Pfad zu einem Blatt (exkl. *nil*-Knoten) in T_v .



(Schwarz-) Höhe

Definition. Die **Länge** eines Pfades ist die Anzahl seiner Kanten.

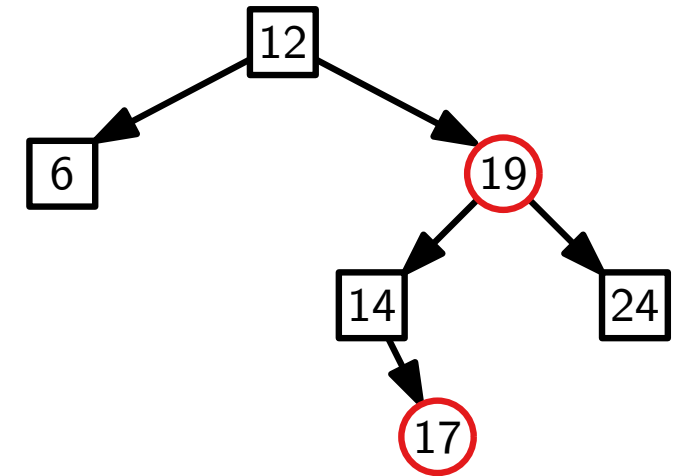
Definition. Sei T ein Baum.
Knoten u ist **unter** Knoten v , wenn u in dem Teilbaum T_v von T mit Wurzel v enthalten ist.

Beispiel: 17 ist unter 19, 14 ist **nicht** unter 6.

Definition. Die **Höhe** eines Knotens v ist die Länge eines längsten Pfads von v zu einem Blatt unter v .

Definition'. Die **Höhe** $\text{Höhe}(v)$ eines Knotens v ist die Anz. der Knoten auf dem längsten Pfad zu einem Blatt (exkl. *nil*-Knoten) in T_v .

Beispiel: „12“ hat Höhe



(Schwarz-) Höhe

Definition. Die **Länge** eines Pfades ist die Anzahl seiner Kanten.

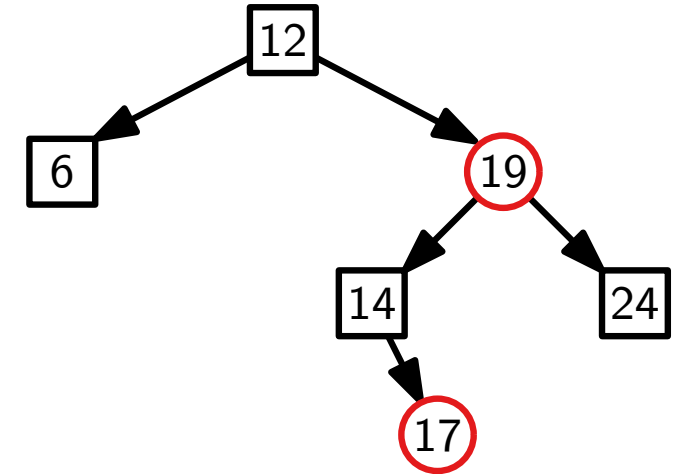
Definition. Sei T ein Baum.
Knoten u ist **unter** Knoten v , wenn u in dem Teilbaum T_v von T mit Wurzel v enthalten ist.

Beispiel: 17 ist unter 19 , 14 ist **nicht** unter 6.

Definition. Die **Höhe** eines Knotens v ist die Länge eines längsten Pfads von v zu einem Blatt unter v .

Definition'. Die **Höhe** $\text{Höhe}(v)$ eines Knotens v ist die Anz. der Knoten auf dem längsten Pfad zu einem Blatt (exkl. *nil*-Knoten) in T_v .

Beispiel: „12“ hat Höhe 4 (!)



(Schwarz-) Höhe

Definition. Die **Länge** eines Pfades ist die Anzahl seiner Kanten.

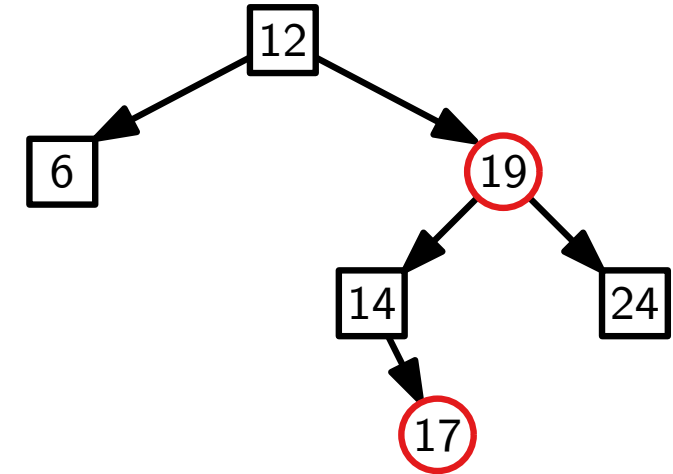
Definition. Sei T ein Baum.
Knoten u ist **unter** Knoten v , wenn u in dem Teilbaum T_v von T mit Wurzel v enthalten ist.

Beispiel: 17 ist unter 19, 14 ist **nicht** unter 6.

Definition. Die **Höhe** eines Knotens v ist die Länge eines längsten Pfads von v zu einem Blatt unter v .

Definition'. Die **Schwarz-Höhe** $shöhe(v)$ eines Knotens v ist die Anz. der schwarzen Knoten auf **jedem** Pfad zu einem Blatt (exkl. *nil*-Knoten) in T_v .

Beispiel: „12“ hat Höhe 4 (!)



(Schwarz-) Höhe

Definition. Die **Länge** eines Pfades ist die Anzahl seiner Kanten.

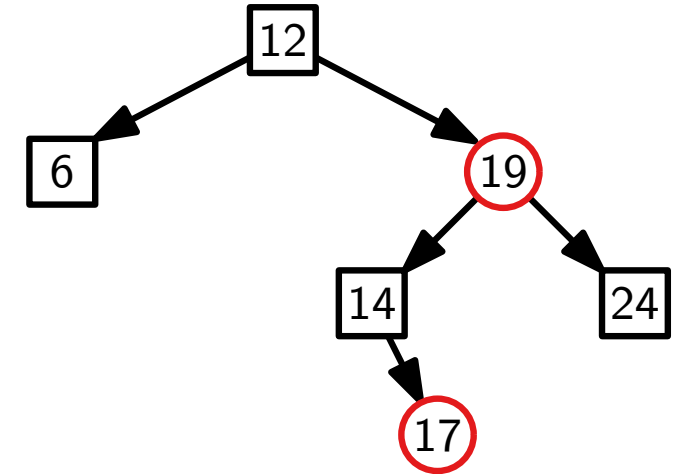
Definition. Sei T ein Baum.
Knoten u ist **unter** Knoten v , wenn u in dem Teilbaum T_v von T mit Wurzel v enthalten ist.

Beispiel: 17 ist unter 19, 14 ist **nicht** unter 6.

Definition. Die **Höhe** eines Knotens v ist die Länge eines längsten Pfads von v zu einem Blatt unter v .

Definition'. Die **Schwarz-Höhe** $shöhe(v)$ eines Knotens v ist die Anz. der schwarzen Knoten auf **jedem** Pfad zu einem Blatt (exkl. *nil*-Knoten) in T_v .

Beispiel: „12“ hat Höhe 4 (!) und Schwarz-Höhe 2



(Schwarz-) Höhe

Definition. Die **Länge** eines Pfades ist die Anzahl seiner Kanten.

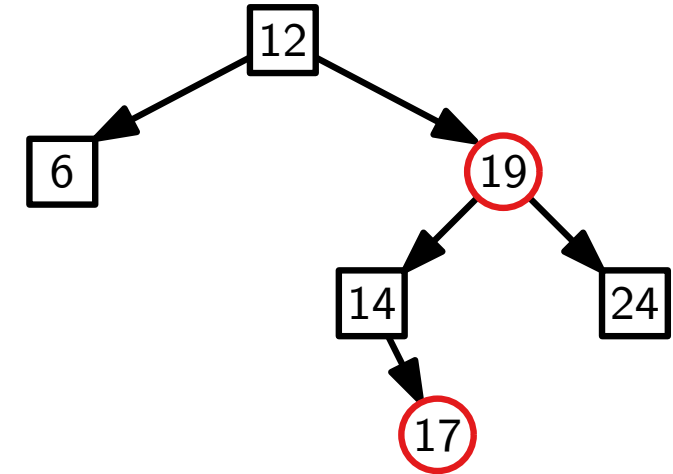
Definition. Sei T ein Baum.
Knoten u ist **unter** Knoten v , wenn u in dem Teilbaum T_v von T mit Wurzel v enthalten ist.

Beispiel: 17 ist unter 19, 14 ist **nicht** unter 6.

Definition. Die **Höhe** eines Knotens v ist die Länge eines längsten Pfads von v zu einem Blatt unter v .

Definition'. Die **Schwarz-Höhe** $shöhe(v)$ eines Knotens v ist die Anz. der schwarzen Knoten auf **jedem** Pfad zu einem Blatt (exkl. *nil*-Knoten) in T_v .

Beispiel: „12“ hat Höhe 4 (!) und Schwarz-Höhe 2.



(Schwarz-) Höhe

Definition. Die **Länge** eines Pfades ist die Anzahl seiner Kanten.

Definition. Sei T ein Baum.
Knoten u ist **unter** Knoten v , wenn u in dem Teilbaum T_v von T mit Wurzel v enthalten ist.

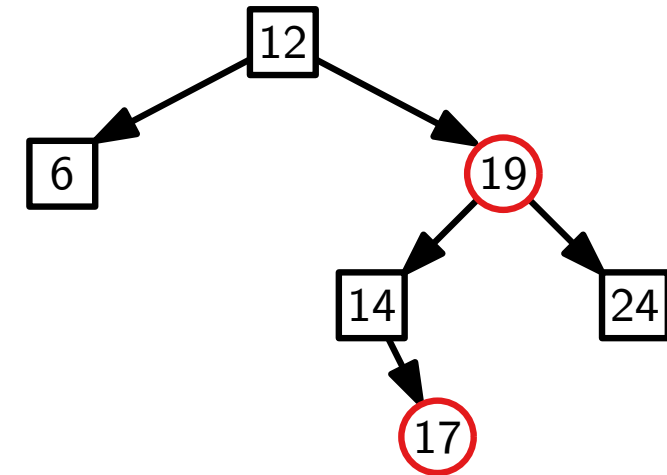
Beispiel: 17 ist unter 19, 14 ist **nicht** unter 6.

Definition. Die **Höhe** eines Knotens v ist die Länge eines längsten Pfads von v zu einem Blatt unter v .

Definition'. Die **Schwarz-Höhe** $shöhe(v)$ eines Knotens v ist die Anz. der schwarzen Knoten auf **jedem** Pfad zu einem Blatt (exkl. *nil*-Knoten) in T_v .

Beispiel: „12“ hat Höhe 4 (!) und Schwarz-Höhe 2.

Folgerung: v Knoten $\Rightarrow shöhe(v) \leq$



(Schwarz-) Höhe

Definition. Die **Länge** eines Pfades ist die Anzahl seiner Kanten.

Definition. Sei T ein Baum.
Knoten u ist **unter** Knoten v , wenn u in dem Teilbaum T_v von T mit Wurzel v enthalten ist.

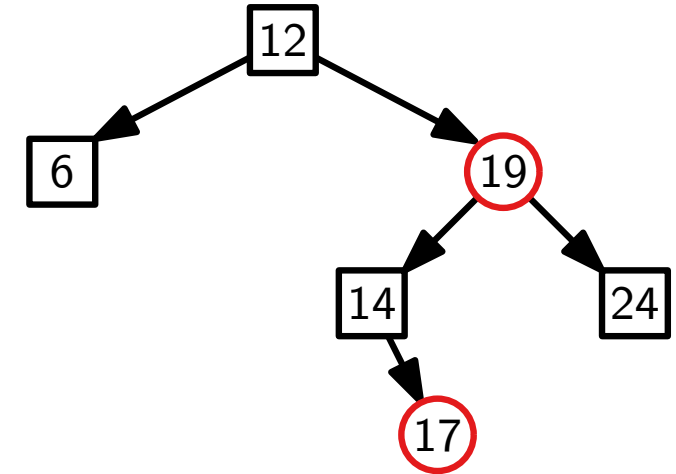
Beispiel: 17 ist unter 19, 14 ist **nicht** unter 6.

Definition. Die **Höhe** eines Knotens v ist die Länge eines längsten Pfads von v zu einem Blatt unter v .

Definition'. Die **Schwarz-Höhe** $shöhe(v)$ eines Knotens v ist die Anz. der schwarzen Knoten auf **jedem** Pfad zu einem Blatt (exkl. *nil*-Knoten) in T_v .

Beispiel: „12“ hat Höhe 4 (!) und Schwarz-Höhe 2.

Folgerung: v Knoten $\Rightarrow shöhe(v) \leq Höhe(v)$



(Schwarz-) Höhe

Definition. Die **Länge** eines Pfades ist die Anzahl seiner Kanten.

Definition. Sei T ein Baum.
Knoten u ist **unter** Knoten v , wenn u in dem Teilbaum T_v von T mit Wurzel v enthalten ist.

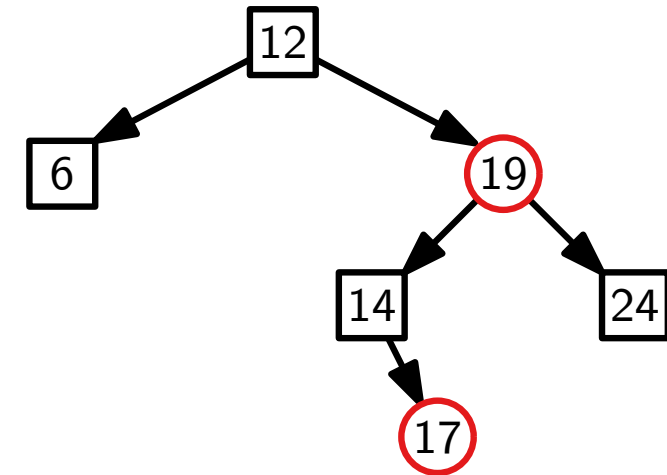
Beispiel: 17 ist unter 19, 14 ist **nicht** unter 6.

Definition. Die **Höhe** eines Knotens v ist die Länge eines längsten Pfads von v zu einem Blatt unter v .

Definition'. Die **Schwarz-Höhe** $shöhe(v)$ eines Knotens v ist die Anz. der schwarzen Knoten auf **jedem** Pfad zu einem Blatt (exkl. *nil*-Knoten) in T_v .

Beispiel: „12“ hat Höhe 4 (!) und Schwarz-Höhe 2.

Folgerung: v Knoten $\Rightarrow shöhe(v) \leq Höhe(v) \leq$



(Schwarz-) Höhe

Definition. Die **Länge** eines Pfades ist die Anzahl seiner Kanten.

Definition. Sei T ein Baum.
Knoten u ist **unter** Knoten v , wenn u in dem Teilbaum T_v von T mit Wurzel v enthalten ist.

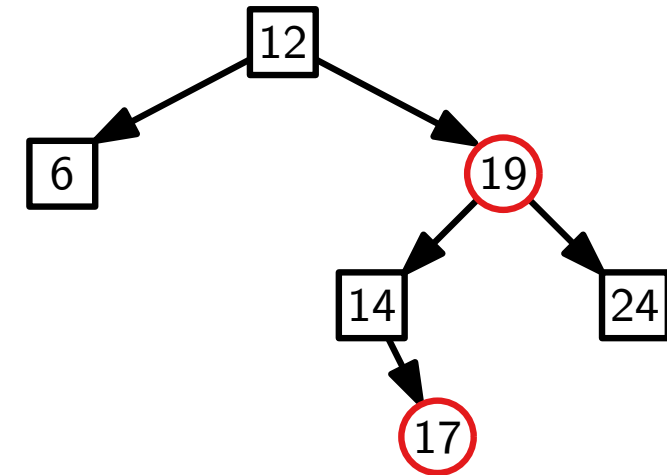
Beispiel: 17 ist unter 19, 14 ist **nicht** unter 6.

Definition. Die **Höhe** eines Knotens v ist die Länge eines längsten Pfads von v zu einem Blatt unter v .

Definition'. Die **Schwarz-Höhe** $\text{sHöhe}(v)$ eines Knotens v ist die Anz. der schwarzen Knoten auf **jedem** Pfad zu einem Blatt (exkl. *nil*-Knoten) in T_v .

Beispiel: „12“ hat Höhe 4 (!) und Schwarz-Höhe 2.

Folgerung: v Knoten $\Rightarrow \text{sHöhe}(v) \leq \text{Höhe}(v) \leq 2 \cdot \text{sHöhe}(v)$.



(Schwarz-) Höhe

Definition. Die **Länge** eines Pfades ist die Anzahl seiner Kanten.

Definition. Sei T ein Baum.
Knoten u ist **unter** Knoten v , wenn u in dem Teilbaum T_v von T mit Wurzel v enthalten ist.

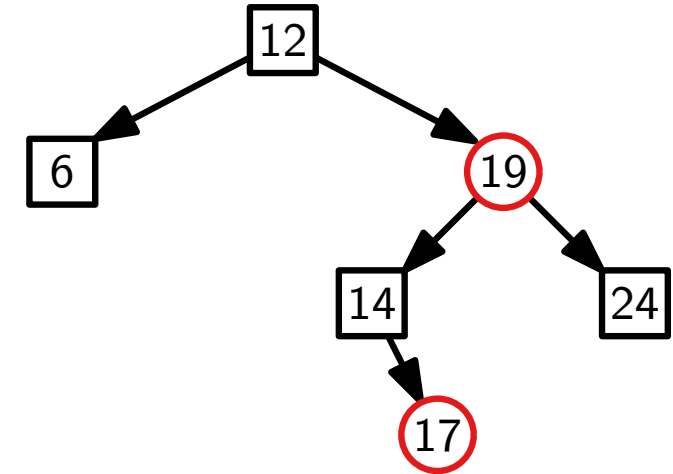
Beispiel: 17 ist unter 19, 14 ist **nicht** unter 6.

Definition. Die **Höhe** eines Knotens v ist die Länge eines längsten Pfads von v zu einem Blatt unter v .

Definition'. Die **Schwarz-Höhe** $\text{sHöhe}(v)$ eines Knotens v ist die Anz. der schwarzen Knoten auf **jedem** Pfad zu einem Blatt (exkl. *nil*-Knoten) in T_v .

Beispiel: „12“ hat Höhe 4 (!) und Schwarz-Höhe 2.

Folgerung: v Knoten $\Rightarrow \text{sHöhe}(v) \leq \overset{\text{(E4)}}{\text{Höhe}(v)} \leq 2 \cdot \text{sHöhe}(v)$.



Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

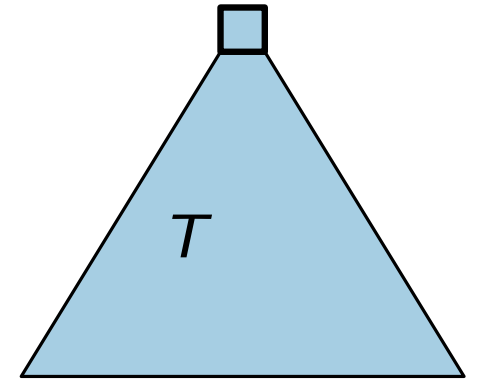
Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis. Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

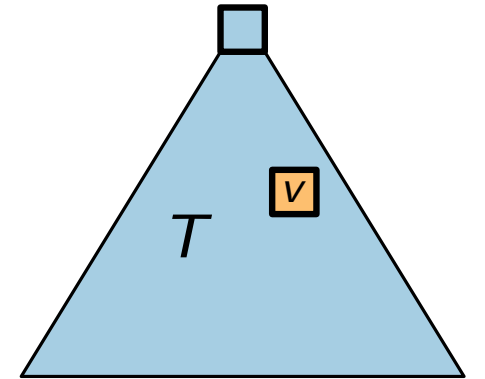


Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

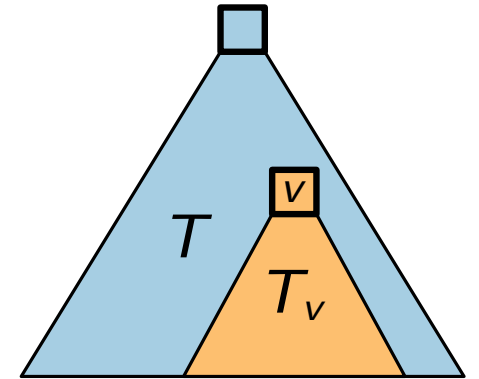


Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.



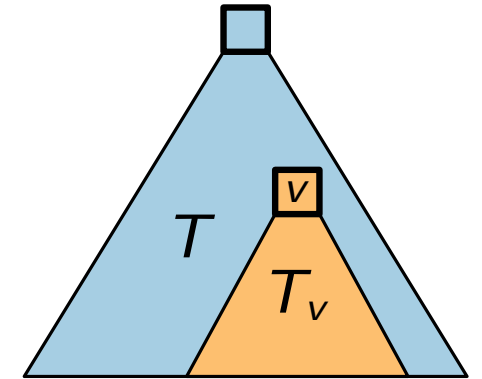
Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

Beweis durch vollständige Induktion über $\text{Höhe}(v)$.



Höhe $\in \Theta(\log n)$

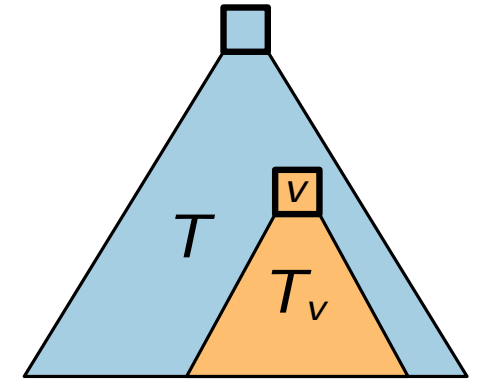
Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{Höhe}(v)} - 1$ innere Knoten.

Beweis durch vollständige Induktion über $\text{Höhe}(v)$.

$\text{Höhe}(v) = 0$.



Höhe $\in \Theta(\log n)$

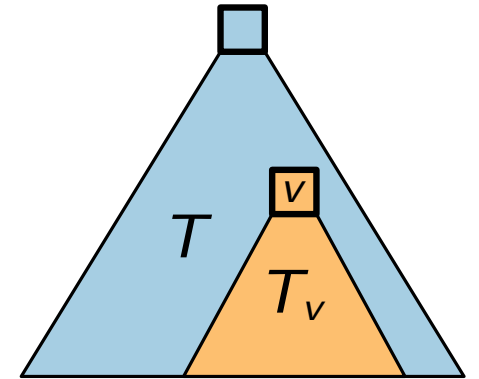
Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{Höhe}(v)} - 1$ innere Knoten.

Beweis durch vollständige Induktion über $\text{Höhe}(v)$.

$\text{Höhe}(v) = 0$. Dann $T_v = T.\text{nil}$



Höhe $\in \Theta(\log n)$

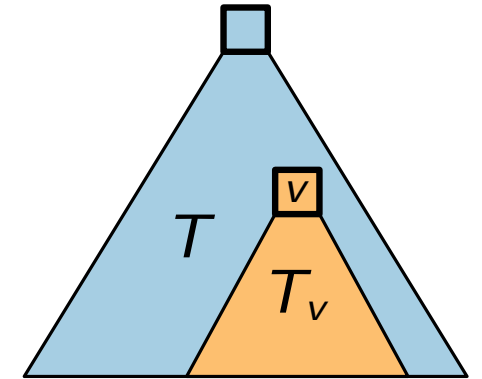
Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

Beweis durch vollständige Induktion über $\text{Höhe}(v)$.

$\text{Höhe}(v) = 0$. Dann $T_v = T.\text{nil}$ und $\text{sHöhe}(v) = 0$.



Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

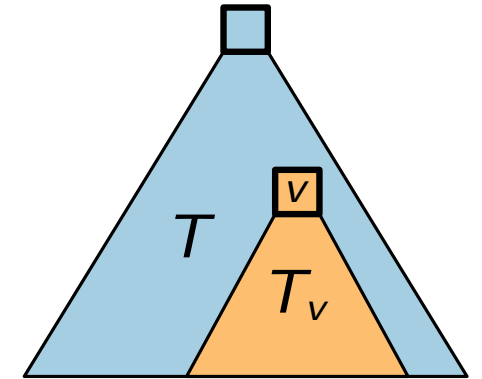
Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

Beweis durch vollständige Induktion über $\text{Höhe}(v)$.

$\text{Höhe}(v) = 0$. Dann $T_v = T.\text{nil}$ und $\text{sHöhe}(v) = 0$.

T_v hat $0 = 2^0 - 1$ innere Knoten.



Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

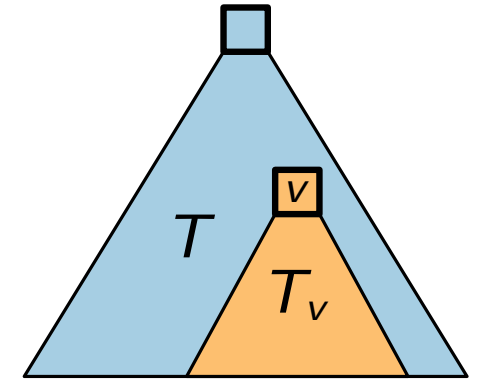
Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

Beweis durch vollständige Induktion über $\text{Höhe}(v)$.

$\text{Höhe}(v) = 0$. Dann $T_v = T.\text{nil}$ und $\text{sHöhe}(v) = 0$.

T_v hat $0 = 2^0 - 1$ innere Knoten.



Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

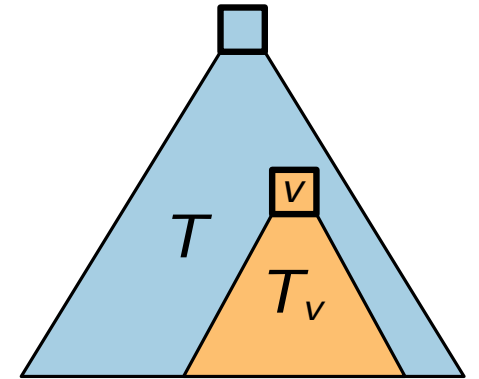
Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

Beweis durch vollständige Induktion über $\text{Höhe}(v)$.

$\text{Höhe}(v) = 0$. Dann $T_v = T.\text{nil}$ und $\text{sHöhe}(v) = 0$.

T_v hat $0 = 2^0 - 1$ innere Knoten.

$\text{Höhe}(v) > 0$.



Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

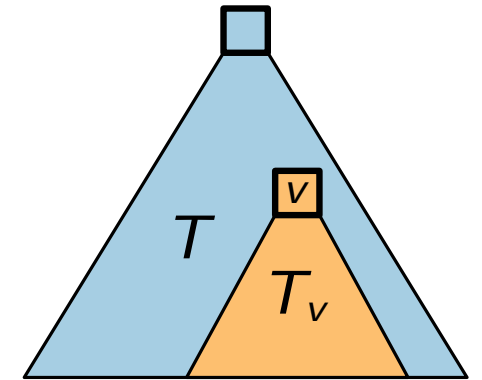
Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

Beweis durch vollständige Induktion über $\text{Höhe}(v)$.

$\text{Höhe}(v) = 0$. Dann $T_v = T.\text{nil}$ und $\text{sHöhe}(v) = 0$.

T_v hat $0 = 2^0 - 1$ innere Knoten.

$\text{Höhe}(v) > 0$. Beide Kinder von v haben $\text{Höhe} < \text{Höhe}(v)$.



Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

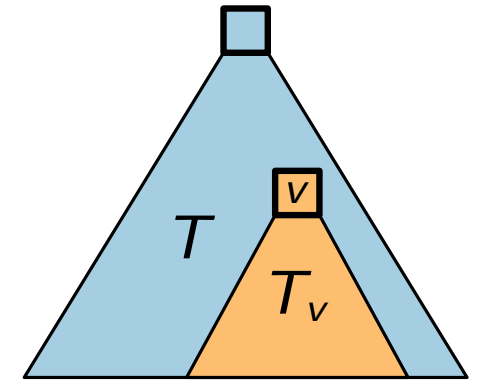
Beweis durch vollständige Induktion über $\text{Höhe}(v)$.

$\text{Höhe}(v) = 0$. Dann $T_v = T.\text{nil}$ und $\text{sHöhe}(v) = 0$.

T_v hat $0 = 2^0 - 1$ innere Knoten.

$\text{Höhe}(v) > 0$. Beide Kinder von v haben $\text{Höhe} < \text{Höhe}(v)$.

$\Rightarrow$ können Ind.-Annahme anwenden.



Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

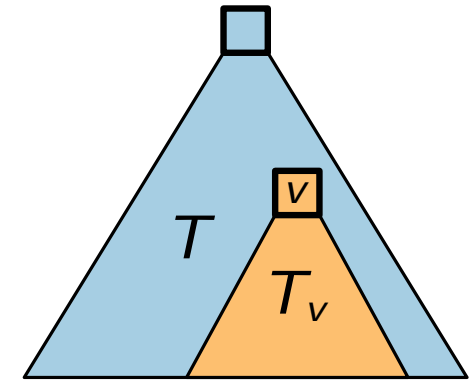
Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

Beweis durch vollständige Induktion über $\text{Höhe}(v)$.

$\text{Höhe}(v) = 0$. Dann $T_v = T.\text{nil}$ und $\text{sHöhe}(v) = 0$.
 T_v hat $0 = 2^0 - 1$ innere Knoten.

$\text{Höhe}(v) > 0$. Beide Kinder von v haben $\text{Höhe} < \text{Höhe}(v)$.
 $\Rightarrow$ können Ind.-Annahme anwenden.
 $\Rightarrow$ # innere Knoten von T_v ist mind.
 $2 \cdot (2^{\text{sHöhe}(v)-1} - 1) + 1 =$



Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

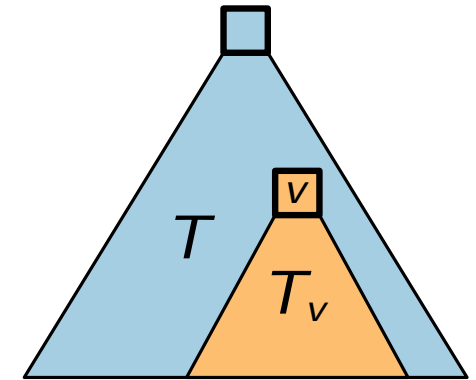
Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

Beweis durch vollständige Induktion über $\text{Höhe}(v)$.

$\text{Höhe}(v) = 0$. Dann $T_v = T.\text{nil}$ und $\text{sHöhe}(v) = 0$.
 T_v hat $0 = 2^0 - 1$ innere Knoten.

$\text{Höhe}(v) > 0$. Beide Kinder von v haben $\text{Höhe} < \text{Höhe}(v)$.
 $\Rightarrow$ können Ind.-Annahme anwenden.
 $\Rightarrow$ # innere Knoten von T_v ist mind.
 $2 \cdot (2^{\text{sHöhe}(v)-1} - 1) + 1 =$

sHöhe der Kinder von v ist mind.



Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

Beweis durch vollständige Induktion über $\text{Höhe}(v)$.

$\text{Höhe}(v) = 0$. Dann $T_v = T.\text{nil}$ und $\text{sHöhe}(v) = 0$.
 T_v hat $0 = 2^0 - 1$ innere Knoten.

$\text{Höhe}(v) > 0$. Beide Kinder von v haben $\text{Höhe} < \text{Höhe}(v)$.

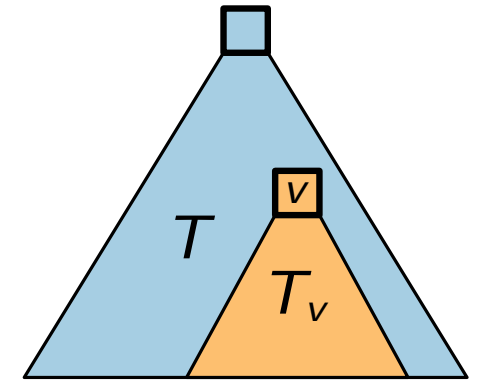
$\Rightarrow$ können Ind.-Annahme anwenden.

$\Rightarrow$ # innere Knoten von T_v ist mind.

$$2 \cdot (2^{\text{sHöhe}(v)-1} - 1) + 1 =$$

sHöhe der Kinder von v ist mind.

Anz. innerer Knoten unter einem Kind von v (IA)



Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

Beweis durch vollständige Induktion über $\text{Höhe}(v)$.

$\text{Höhe}(v) = 0$. Dann $T_v = T.\text{nil}$ und $\text{sHöhe}(v) = 0$.
 T_v hat $0 = 2^0 - 1$ innere Knoten.

$\text{Höhe}(v) > 0$. Beide Kinder von v haben $\text{Höhe} < \text{Höhe}(v)$.

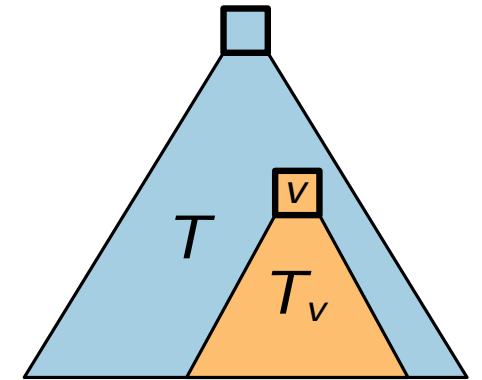
$\Rightarrow$ können Ind.-Annahme anwenden.

$\Rightarrow$ # innere Knoten von T_v ist mind.

$$2 \cdot (2^{\text{sHöhe}(v)-1} - 1) + 1 = 2^{\text{sHöhe}(v)} - 1.$$

sHöhe der Kinder von v ist mind.

Anz. innerer Knoten unter
einem Kind von v (IA)



Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

Beweis durch vollständige Induktion über $\text{Höhe}(v)$.

$\text{Höhe}(v) = 0$. Dann $T_v = T.\text{nil}$ und $\text{sHöhe}(v) = 0$.
 T_v hat $0 = 2^0 - 1$ innere Knoten. ✓

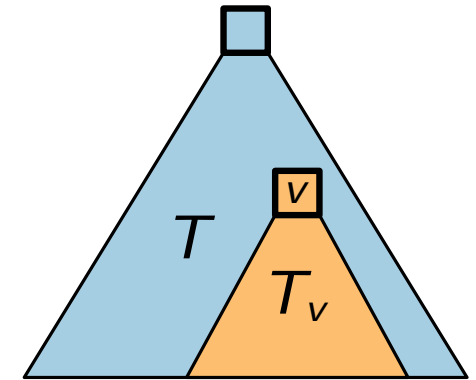
$\text{Höhe}(v) > 0$. Beide Kinder von v haben $\text{Höhe} < \text{Höhe}(v)$.
 $\Rightarrow$ können Ind.-Annahme anwenden.

$\Rightarrow$ # innere Knoten von T_v ist mind.

$$2 \cdot (2^{\text{sHöhe}(v)-1} - 1) + 1 = 2^{\text{sHöhe}(v)} - 1. \quad \checkmark$$

sHöhe der Kinder von v ist mind.

Anz. innerer Knoten unter
einem Kind von v (IA)



Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

Beweis durch vollständige Induktion über $\text{Höhe}(v)$.

$\text{Höhe}(v) = 0$. Dann $T_v = T.\text{nil}$ und $\text{sHöhe}(v) = 0$.
 T_v hat $0 = 2^0 - 1$ innere Knoten.

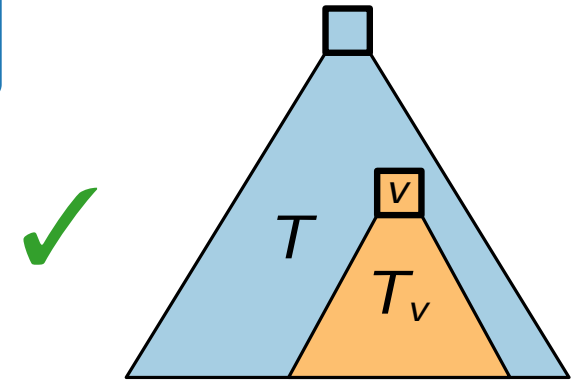
$\text{Höhe}(v) > 0$. Beide Kinder von v haben $\text{Höhe} < \text{Höhe}(v)$.
 $\Rightarrow$ können Ind.-Annahme anwenden.

$\Rightarrow$ # innere Knoten von T_v ist mind.

$$2 \cdot (2^{\text{sHöhe}(v)-1} - 1) + 1 = 2^{\text{sHöhe}(v)} - 1.$$

sHöhe der Kinder von v ist mind.

Anz. innerer Knoten unter
einem Kind von v (IA)



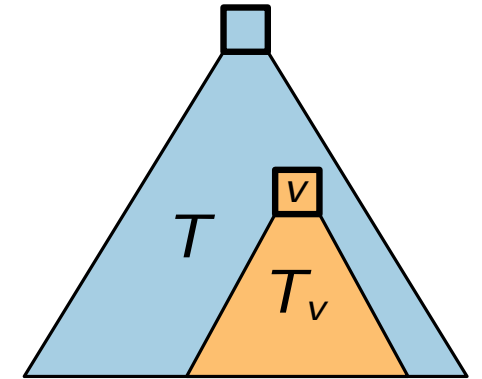
Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

$v = T.\text{root} \Rightarrow$



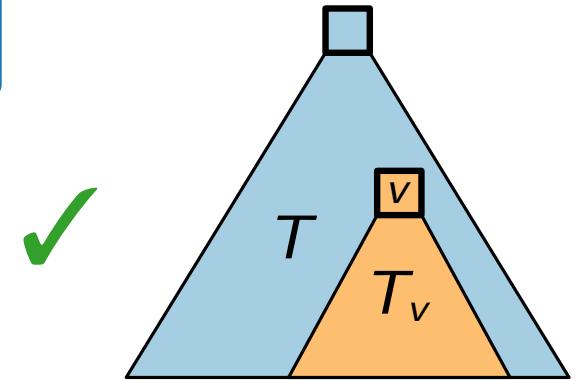
Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

$v = T.\text{root} \Rightarrow \# \text{ innere Knoten}(T) \geq 2^{\text{sHöhe}(T)} - 1.$



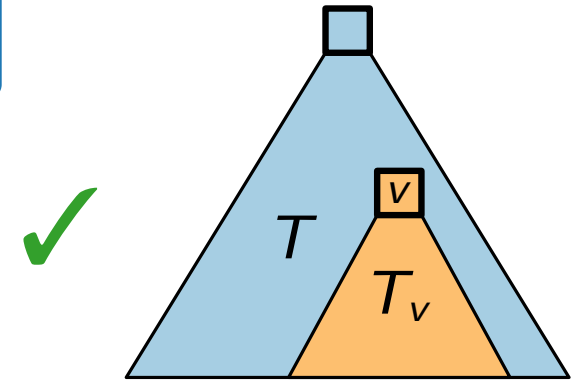
Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

$v = T.\text{root} \Rightarrow \underbrace{\# \text{ innere Knoten}(T)}_n \geq 2^{\text{sHöhe}(T)} - 1.$



Höhe $\in \Theta(\log n)$

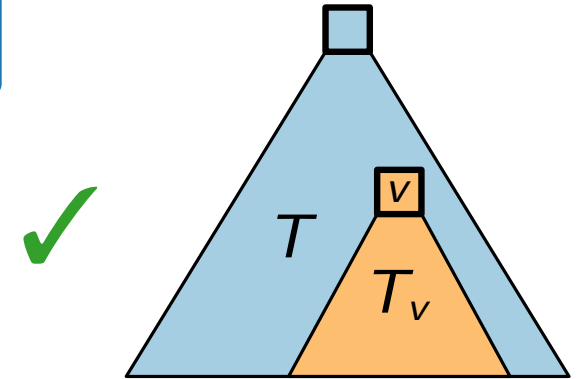
Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

$$v = T.\text{root} \Rightarrow \underbrace{\# \text{ innere Knoten}(T)}_n \geq 2^{\text{sHöhe}(T)} - 1.$$

$$\Rightarrow \text{sHöhe}(T) \leq$$



Höhe $\in \Theta(\log n)$

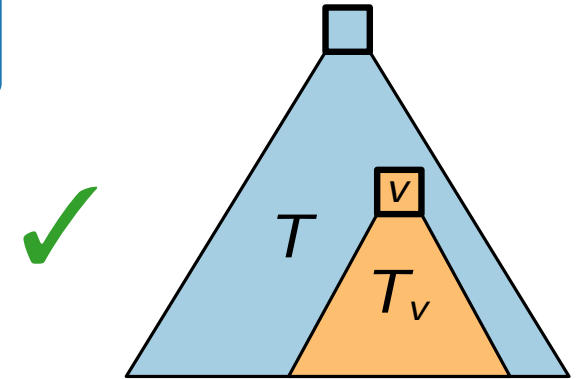
Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

$$v = T.\text{root} \Rightarrow \underbrace{\# \text{ innere Knoten}(T)}_n \geq 2^{\text{sHöhe}(T)} - 1.$$

$$\Rightarrow \text{sHöhe}(T) \leq \log_2(n + 1)$$



Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

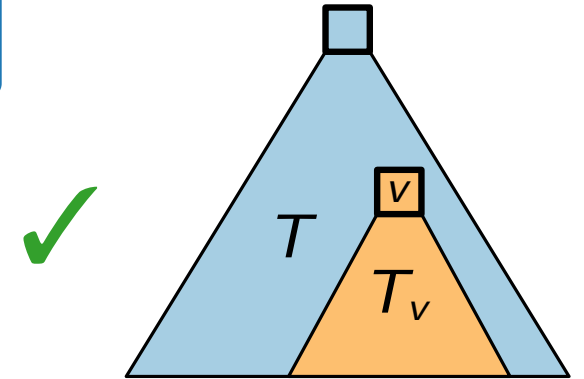
Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

$$v = T.\text{root} \Rightarrow \underbrace{\# \text{ innere Knoten}(T)}_n \geq 2^{\text{sHöhe}(T)} - 1.$$

$$\Rightarrow \text{sHöhe}(T) \leq \log_2(n + 1)$$

Wegen R-S-Eig. **(E4)** gilt:



Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

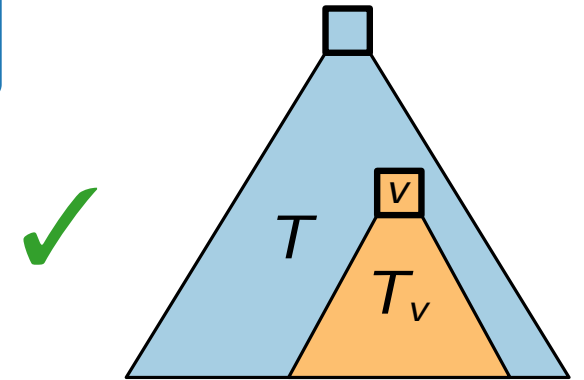
Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

$$v = T.\text{root} \Rightarrow \underbrace{\# \text{ innere Knoten}(T)}_n \geq 2^{\text{sHöhe}(T)} - 1.$$

$$\Rightarrow \text{sHöhe}(T) \leq \log_2(n + 1)$$

Wegen R-S-Eig. **(E4)** gilt: $\text{Höhe}(T) \leq 2 \cdot \text{sHöhe}(T)$.



Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

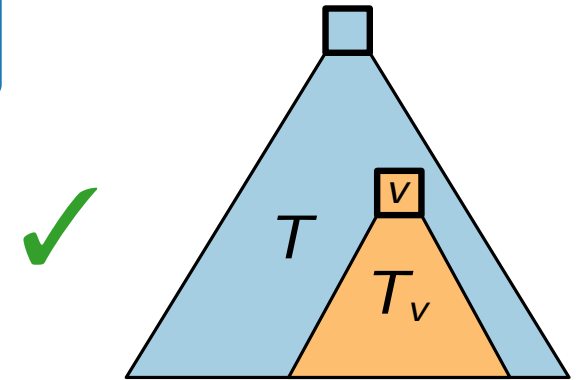
$$v = T.\text{root} \Rightarrow \underbrace{\# \text{ innere Knoten}(T)}_n \geq 2^{\text{sHöhe}(T)} - 1.$$

$$\Rightarrow \text{sHöhe}(T) \leq \log_2(n + 1)$$

Wegen R-S-Eig. **(E4)** gilt: $\text{Höhe}(T) \leq 2 \cdot \text{sHöhe}(T)$.

$$\Rightarrow \text{Höhe}(T) \leq 2 \log_2(n + 1)$$

□



Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

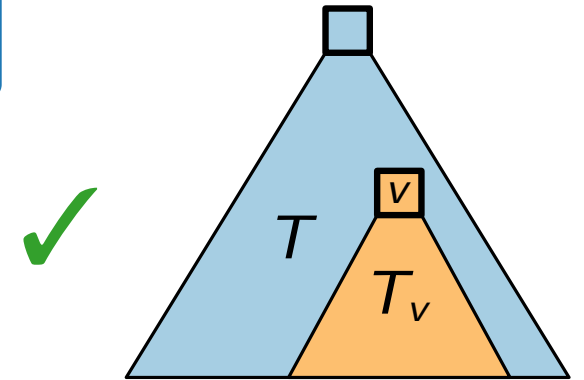
Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

$$v = T.\text{root} \Rightarrow \underbrace{\# \text{ innere Knoten}(T)}_n \geq 2^{\text{sHöhe}(T)} - 1.$$

$$\Rightarrow \text{sHöhe}(T) \leq \log_2(n + 1)$$

Wegen R-S-Eig. **(E4)** gilt: Höhe(T) $\leq 2 \cdot \text{sHöhe}(T)$.

$$\Rightarrow \text{Höhe}(T) \leq 2 \log_2(n + 1) \quad \square$$



Also: Rot-Schwarz-Bäume sind **balanciert!**

Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

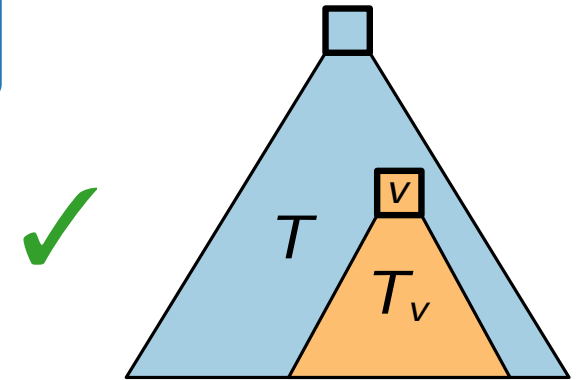
Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

$$v = T.\text{root} \Rightarrow \underbrace{\# \text{ innere Knoten}(T)}_n \geq 2^{\text{sHöhe}(T)} - 1.$$

$$\Rightarrow \text{sHöhe}(T) \leq \log_2(n + 1)$$

Wegen R-S-Eig. **(E4)** gilt: Höhe(T) $\leq 2 \cdot \text{sHöhe}(T)$.

$$\Rightarrow \text{Höhe}(T) \leq 2 \log_2(n + 1) \quad \square$$



Also: Rot-Schwarz-Bäume sind **balanciert!** Fertig?!

Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

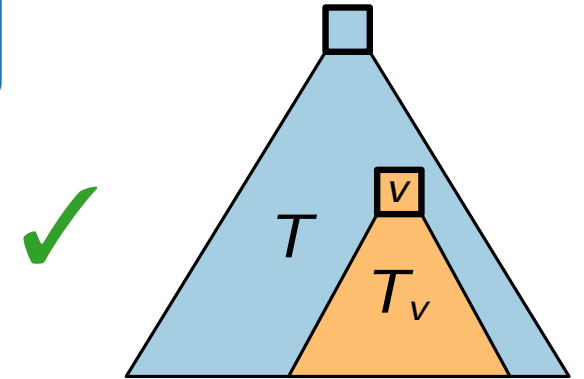
Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

$$v = T.\text{root} \Rightarrow \underbrace{\# \text{ innere Knoten}(T)}_n \geq 2^{\text{sHöhe}(T)} - 1.$$

$$\Rightarrow \text{sHöhe}(T) \leq \log_2(n + 1)$$

Wegen R-S-Eig. **(E4)** gilt: Höhe(T) $\leq 2 \cdot \text{sHöhe}(T)$.

$$\Rightarrow \text{Höhe}(T) \leq 2 \log_2(n + 1) \quad \square$$



Also: Rot-Schwarz-Bäume sind **balanciert!** Fertig?!

Nein:

Höhe $\in \Theta(\log n)$

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis.

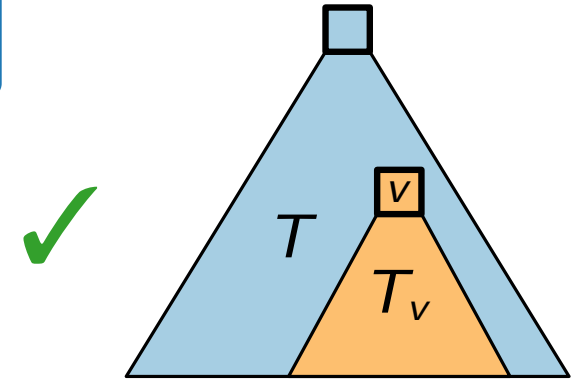
Behauptung: Für jeden Knoten v von T gilt:
 T_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

$$v = T.\text{root} \Rightarrow \underbrace{\# \text{ innere Knoten}(T)}_n \geq 2^{\text{sHöhe}(T)} - 1.$$

$$\Rightarrow \text{sHöhe}(T) \leq \log_2(n + 1)$$

Wegen R-S-Eig. **(E4)** gilt: Höhe(T) $\leq 2 \cdot \text{sHöhe}(T)$.

$$\Rightarrow \text{Höhe}(T) \leq 2 \log_2(n + 1) \quad \square$$



Also: Rot-Schwarz-Bäume sind **balanciert!** Fertig?!

Nein: INSERT & DELETE können Rot-Schwarz-Eigenschaft **verletzen!**

Vorlesungsumfrage – jetzt!

Bitte suchen Sie die E-Mail, die Sie von EvaSys bekommen haben, und klicken Sie dort auf den Link zur Umfrage.



Vorlesungsumfrage – jetzt!

Bitte suchen Sie die E-Mail, die Sie von EvaSys bekommen haben, und klicken Sie dort auf den Link zur Umfrage.



All questions are optional. Should a question not be applicable to a course, you can leave the answer open.

1 Course as a whole

1.1 Please rate the course as a whole.

very good ☐ ☐ ☐ ☐ ☐ insufficient

2 Lecture

2.1 The lecturer is well prepared and presents the material in a way that is easy to understand.

completely agree ☐ ☐ ☐ ☐ ☐ do not agree at all

2.2 The supplied lecture materials (writings on the blackboard, presentation slides, videos, additional literature) are well prepared and improve my understanding of the course contents.

completely agree ☐ ☐ ☐ ☐ ☐ do not agree at all

2.3 Portions of the lecture were held online (live stream, recorded lecture) and helped improve the lecture.

completely agree ☐ ☐ ☐ ☐ ☐ do not agree at all ☐ no online parts

3 Exercises

3.1 The tutor is well prepared and presents the material in a way that is easy to understand.

completely agree ☐ ☐ ☐ ☐ ☐ do not agree at all ☐ no exercises

3.2 The exercises are comprehensible and improve my understanding of the course contents.

completely agree ☐ ☐ ☐ ☐ ☐ do not agree at all ☐ no exercises

Vorlesungsumfrage – jetzt!

Bitte suchen Sie die E-Mail, die Sie von EvaSys bekommen haben, und klicken Sie dort auf den Link zur Umfrage.



All questions are optional. Should a question not be applicable to a course, you can leave the answer open.

1 Course as a whole

1.1 Please rate the course as a whole.

very good ☐ ☐ ☐ ☐ ☐ insufficient

2 Lecture

2.1 The lecturer is well prepared and presents the material in a way that is easy to understand.

completely agree ☐ ☐ ☐ ☐ ☐ do not agree at all

2.2 The supplied lecture materials (writings on the blackboard, presentation slides, videos, additional literature) are well prepared and improve my understanding of the course contents.

completely agree ☐ ☐ ☐ ☐ ☐ do not agree at all

2.3 Portions of the lecture were held online (live stream, recorded lecture) and helped improve the understanding of the course contents.

completely agree ☐ ☐ ☐ ☐ ☐ do not agree at all

3 Exercises

3.1 The tutor is well prepared and presents the material in a way that is easy to understand.

completely agree ☐ ☐ ☐ ☐ ☐ do not agree at all

3.2 The exercises are comprehensible and improve my understanding of the course contents.

completely agree ☐ ☐ ☐ ☐ ☐ do not agree at all

4. Additional remarks

4.1 What did you especially like in this course?

4.2 From your perspective, what could be improved? What do you criticize?

Zum Beispiel:

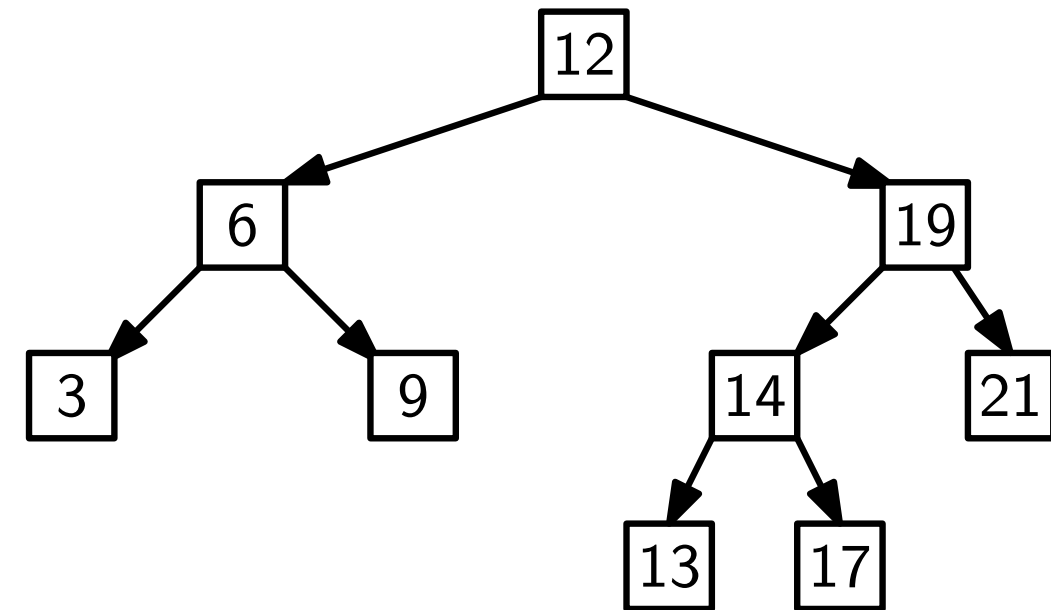
- Folien
- Übungsaufgaben
- Roter Faden?
- Beweise?
- Reaktion auf Fragen?
- Buch zur Vorlesung?
- Zwischentests
- Aufgaben in der VL
- ...

Einfügen

```

Node INSERT(key k)
  y = nil
  x = root
  while x ≠ nil do
    y = x
    if k < x.key then
      x = x.left
    else x = x.right
  z = new Node(k, y)
  if y == nil then root = z
  else
    if k < y.key then y.left = z
    else y.right = z
  return z

```



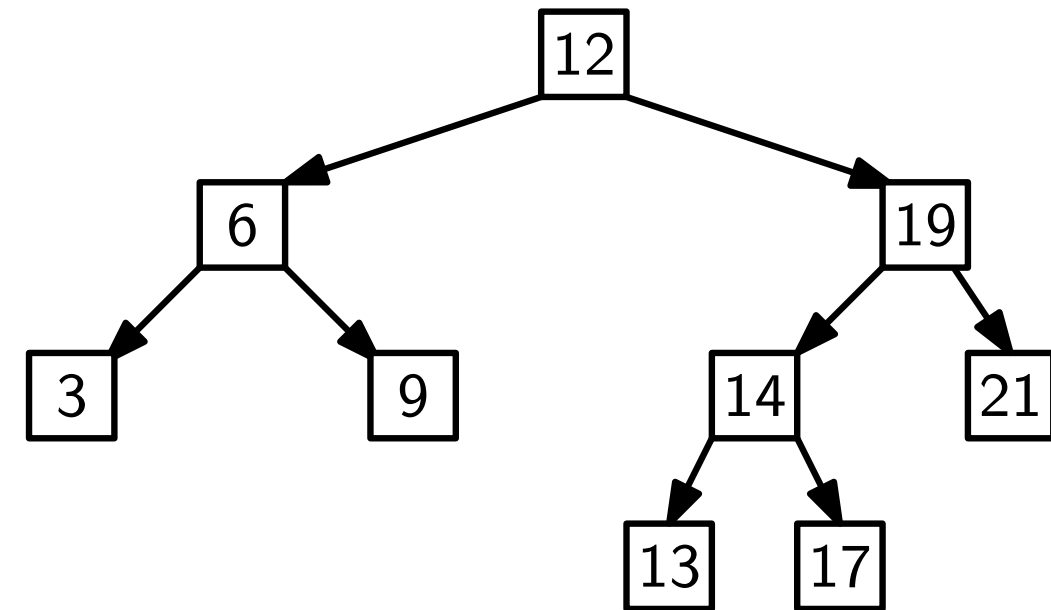
Einfügen

```

Node INSERT(key k)
  y = nil
  x = root
  while x ≠ nil do
    y = x
    if k < x.key then
      x = x.left
    else x = x.right
  z = new Node(k, y)
  if y == nil then root = z
  else
    if k < y.key then y.left = z
    else y.right = z
  return z

```

INSERT(11)

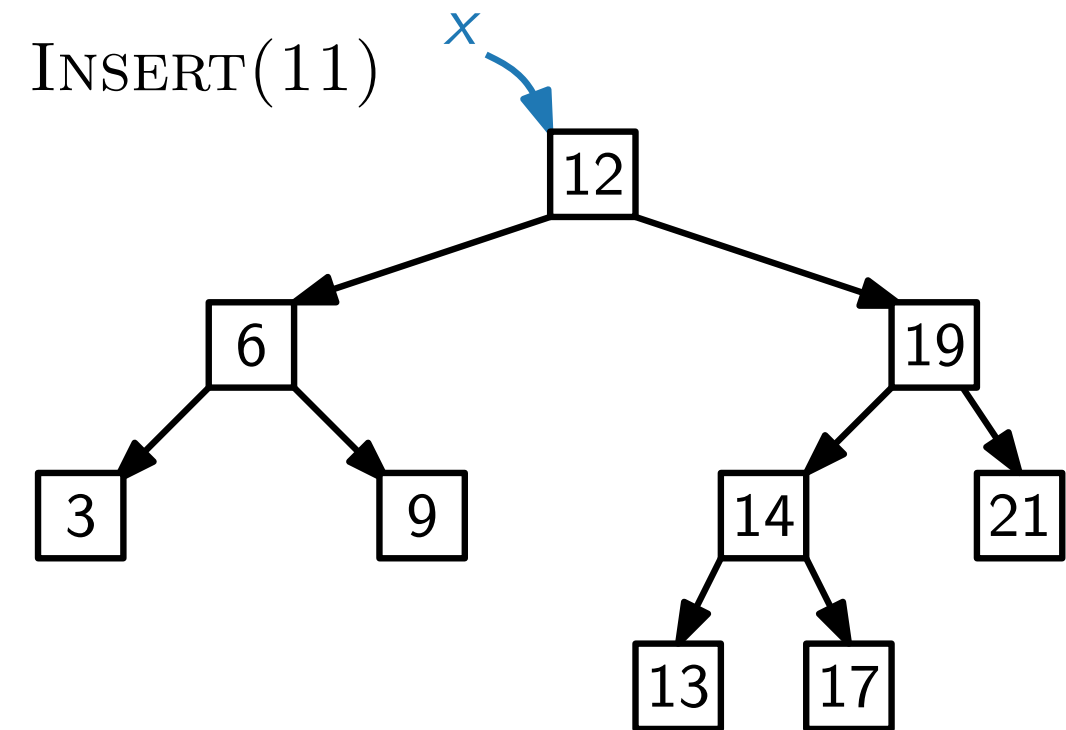


Einfügen

```

Node INSERT(key k)
  y = nil
  x = root
  while x ≠ nil do
    y = x
    if k < x.key then
      x = x.left
    else x = x.right
  z = new Node(k, y)
  if y == nil then root = z
  else
    if k < y.key then y.left = z
    else y.right = z
  return z

```



Einfügen

```
Node INSERT(key k)
```

```
y = nil
```

```
x = root
```

```
while x ≠ nil do
```

```
    y = x
```

```
    if k < x.key then
```

```
        x = x.left
```

```
    else x = x.right
```

```
z = new Node(k, y)
```

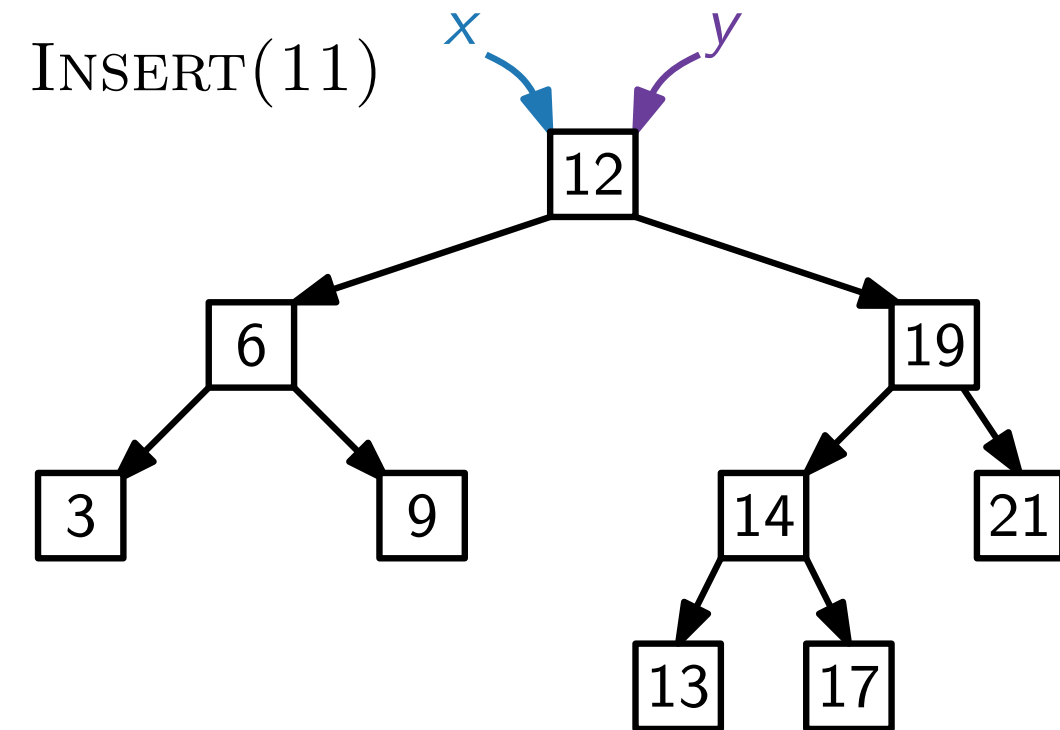
```
if y == nil then root = z
```

```
else
```

```
    if k < y.key then y.left = z
```

```
    else y.right = z
```

```
return z
```



Einfügen

```
Node INSERT(key k)
```

```
y = nil
```

```
x = root
```

```
while x ≠ nil do
```

```
    y = x
```

```
    if k < x.key then
```

```
        x = x.left
```

```
    else x = x.right
```

```
z = new Node(k, y)
```

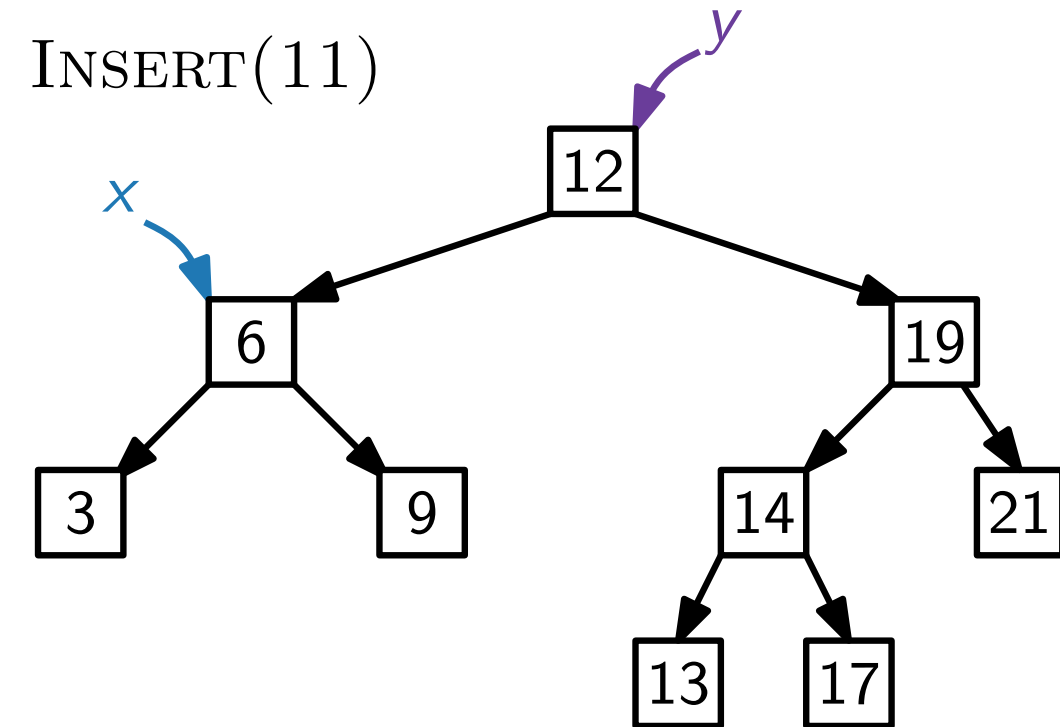
```
if y == nil then root = z
```

```
else
```

```
    if k < y.key then y.left = z
```

```
    else y.right = z
```

```
return z
```



Einfügen

```
Node INSERT(key k)
```

```
y = nil
```

```
x = root
```

```
while x ≠ nil do
```

```
    y = x
```

```
    if k < x.key then
```

```
        x = x.left
```

```
    else x = x.right
```

```
z = new Node(k, y)
```

```
if y == nil then root = z
```

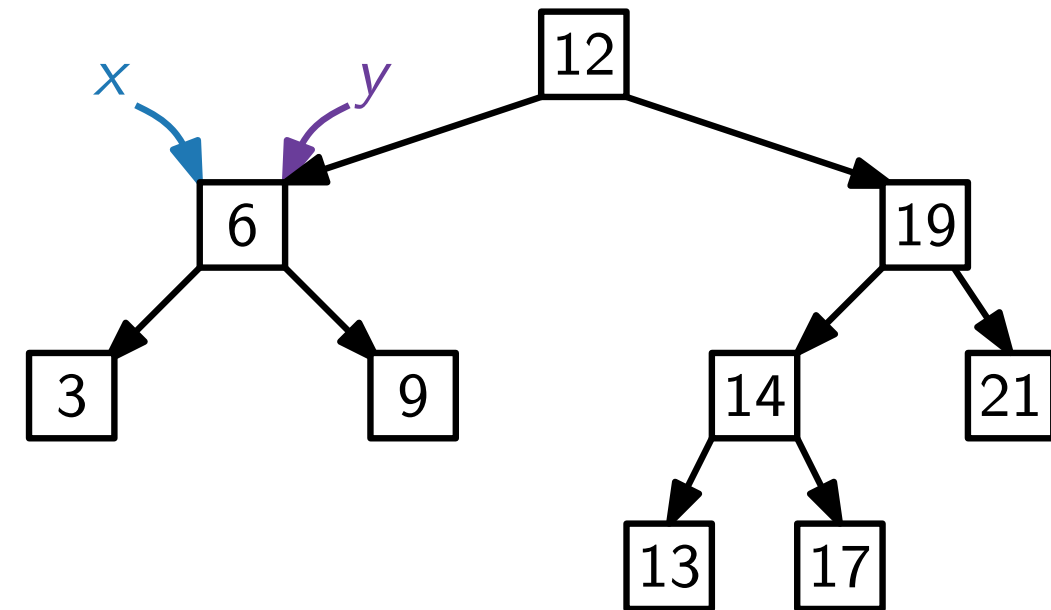
```
else
```

```
    if k < y.key then y.left = z
```

```
    else y.right = z
```

```
return z
```

INSERT(11)



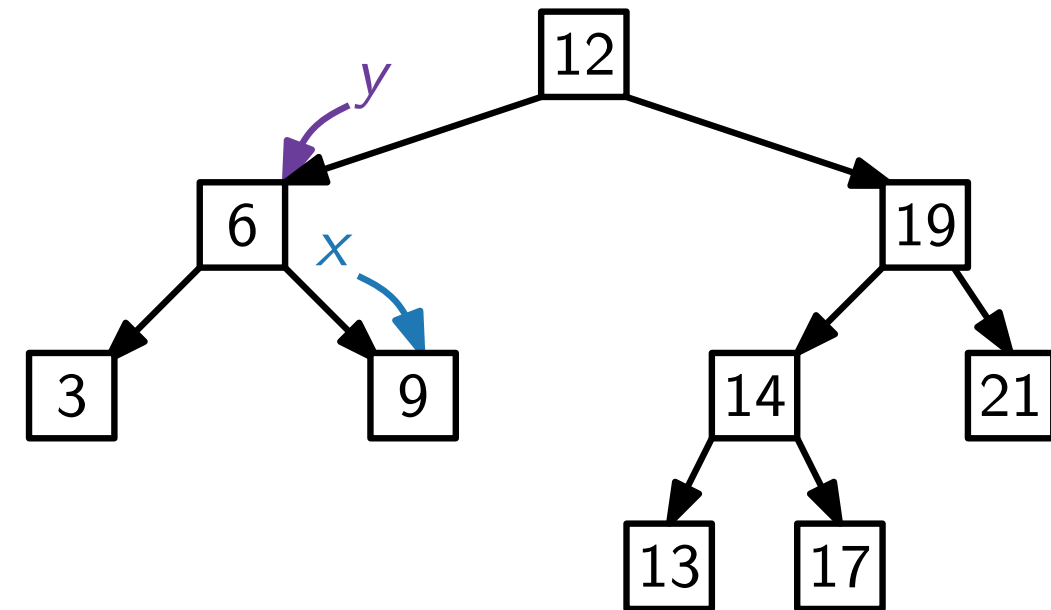
Einfügen

```

Node INSERT(key k)
  y = nil
  x = root
  while x ≠ nil do
    y = x
    if k < x.key then
      x = x.left
    else x = x.right
  z = new Node(k, y)
  if y == nil then root = z
  else
    if k < y.key then y.left = z
    else y.right = z
  return z

```

INSERT(11)



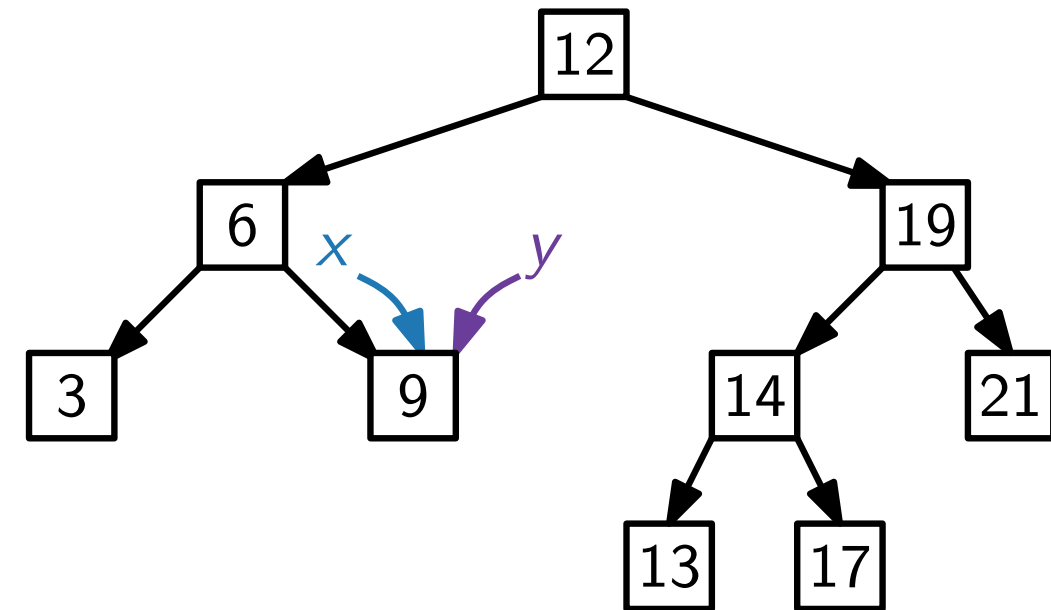
Einfügen

```

Node INSERT(key k)
  y = nil
  x = root
  while x ≠ nil do
    y = x
    if k < x.key then
      x = x.left
    else x = x.right
  z = new Node(k, y)
  if y == nil then root = z
  else
    if k < y.key then y.left = z
    else y.right = z
  return z

```

INSERT(11)



Einfügen

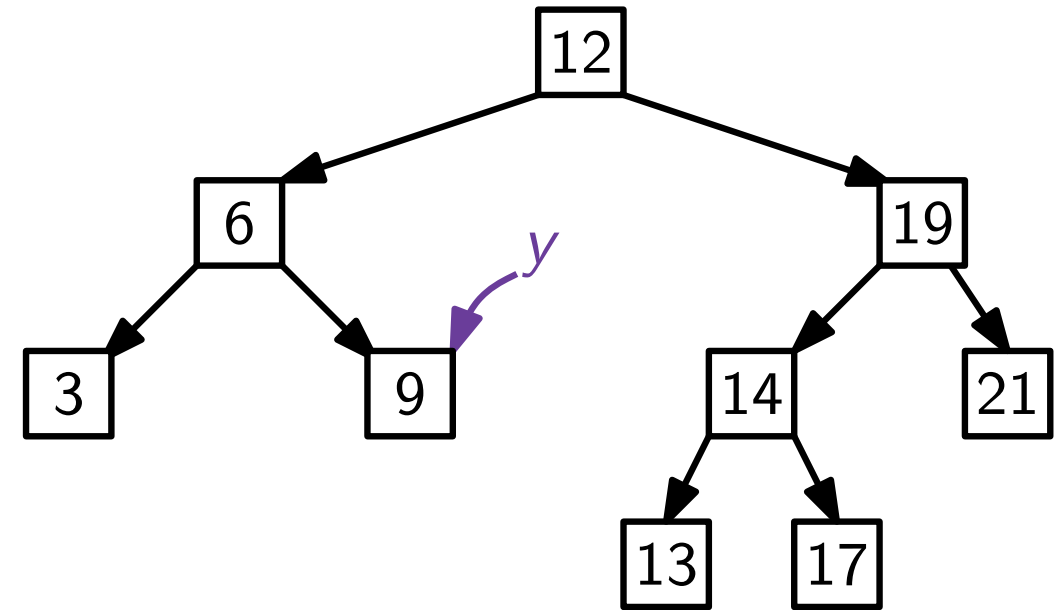
```

Node INSERT(key k)
  y = nil
  x = root
  while x ≠ nil do
    y = x
    if k < x.key then
      x = x.left
    else x = x.right
  z = new Node(k, y)
  if y == nil then root = z
  else
    if k < y.key then y.left = z
    else y.right = z
  return z

```

INSERT(11)

x == nil



Einfügen

```
Node INSERT(key k)
```

```
y = nil
```

```
x = root
```

```
while x ≠ nil do
```

```
    y = x
```

```
    if k < x.key then
```

```
        x = x.left
```

```
    else x = x.right
```

```
z = new Node(k, y)
```

```
if y == nil then root = z
```

```
else
```

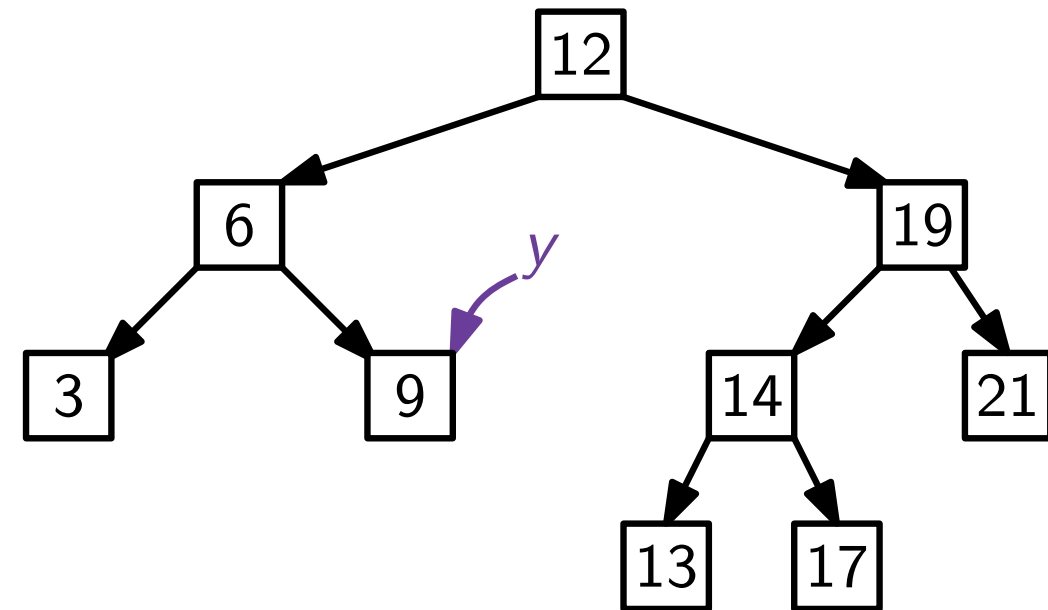
```
    if k < y.key then y.left = z
```

```
    else y.right = z
```

```
return z
```

INSERT(11)

x == nil



Einfügen

```
Node INSERT(key k)
```

```
y = nil
```

```
x = root
```

```
while x ≠ nil do
```

```
    y = x
```

```
    if k < x.key then
```

```
        x = x.left
```

```
    else x = x.right
```

```
z = new Node(k, y)
```

```
if y == nil then root = z
```

```
else
```

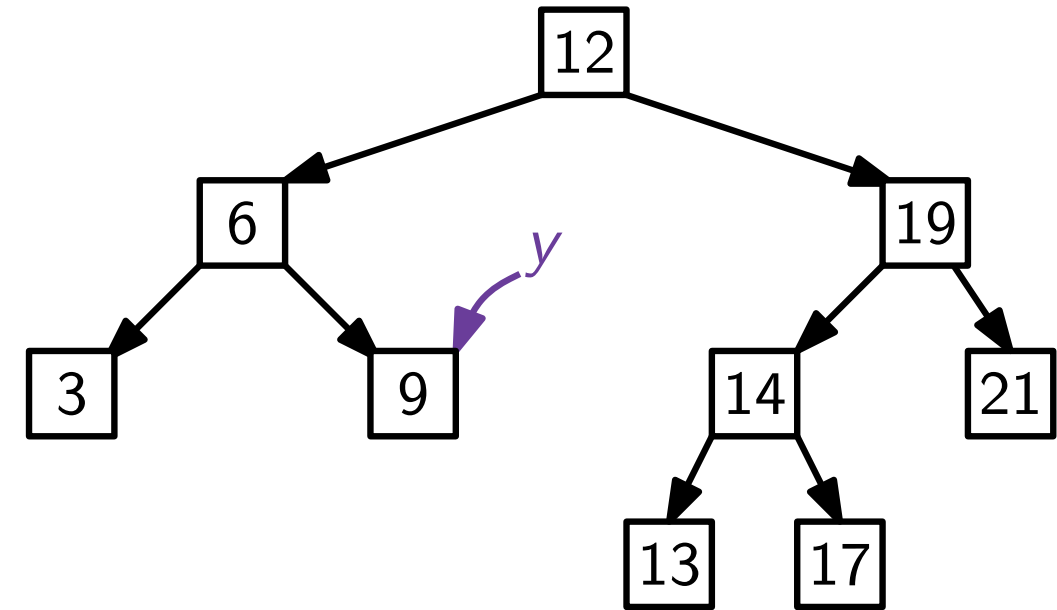
```
    if k < y.key then y.left = z
```

```
    else y.right = z
```

```
return z
```

INSERT(11)

x == nil



```
Node(Key k, Node par)
```

```
key = k
```

```
p = par
```

```
right = left = nil
```

Einfügen

```
Node INSERT(key k)
```

```
y = nil
```

```
x = root
```

```
while x ≠ nil do
```

```
    y = x
```

```
    if k < x.key then
```

```
        x = x.left
```

```
    else x = x.right
```

```
z = new Node(k, y)
```

```
if y == nil then root = z
```

```
else
```

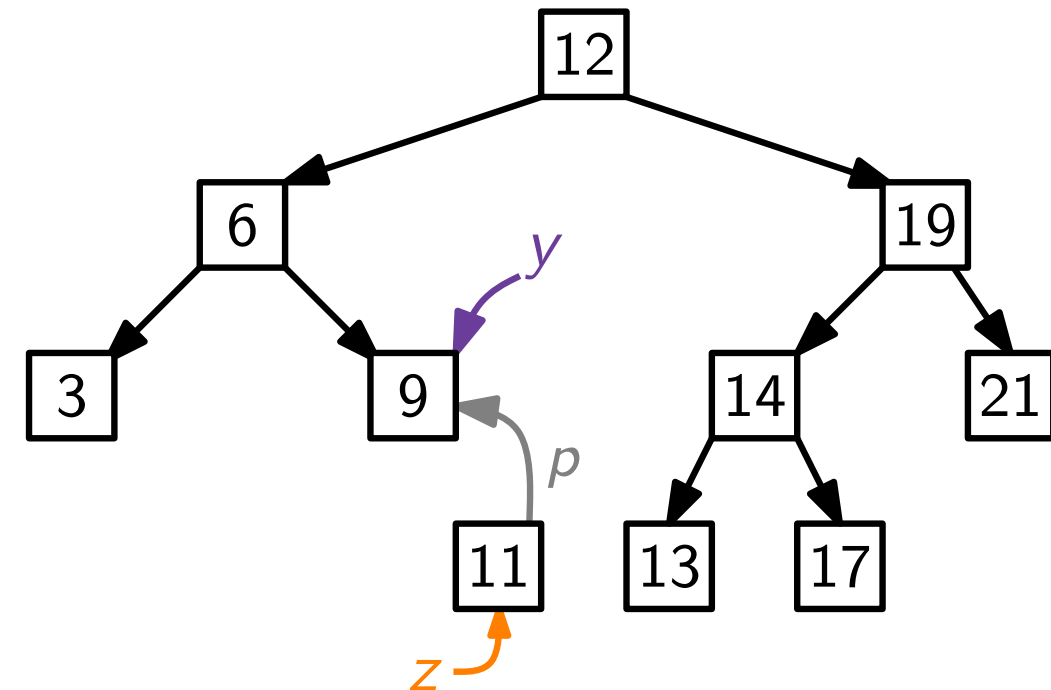
```
    if k < y.key then y.left = z
```

```
    else y.right = z
```

```
return z
```

INSERT(11)

x == nil



```
Node(Key k, Node par)
```

```
key = k
```

```
p = par
```

```
right = left = nil
```

Einfügen

```
Node INSERT(key k)
```

```
y = nil
```

```
x = root
```

```
while x ≠ nil do
```

```
    y = x
```

```
    if k < x.key then
```

```
        x = x.left
```

```
    else x = x.right
```

```
z = new Node(k, y)
```

```
if y == nil then root = z
```

```
else
```

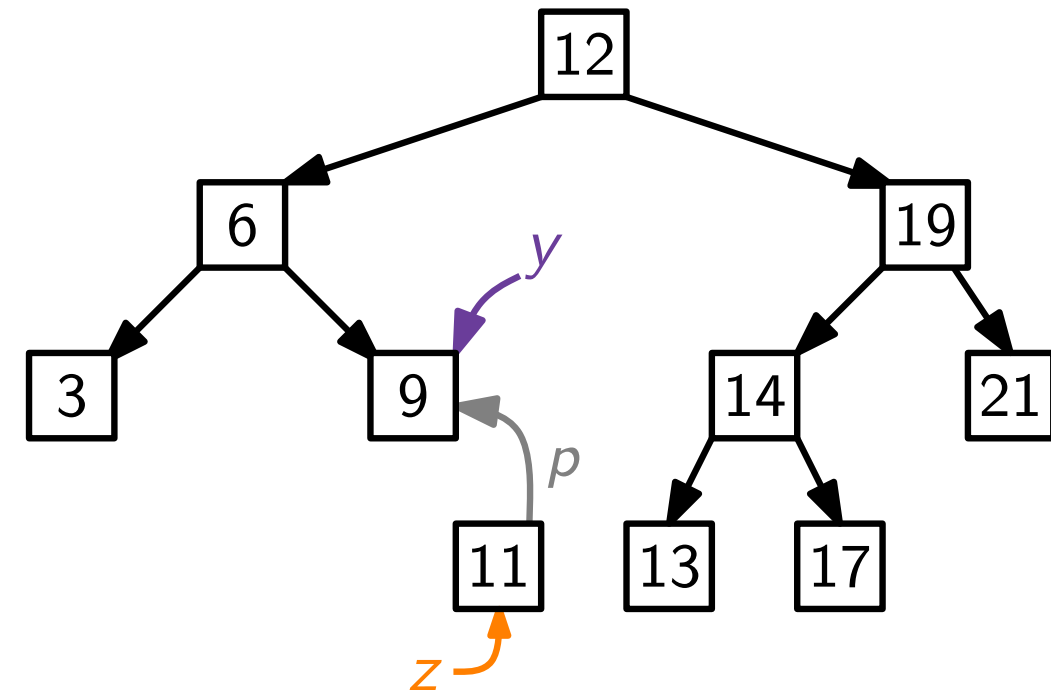
```
    if k < y.key then y.left = z
```

```
    else y.right = z
```

```
return z
```

INSERT(11)

x == nil



```
Node(Key k, Node par)
```

```
key = k
```

```
p = par
```

```
right = left = nil
```

Einfügen

```
Node INSERT(key k)
```

```
y = nil
```

```
x = root
```

```
while x ≠ nil do
```

```
    y = x
```

```
    if k < x.key then
```

```
        x = x.left
```

```
    else x = x.right
```

```
z = new Node(k, y)
```

```
if y == nil then root = z
```

```
else
```

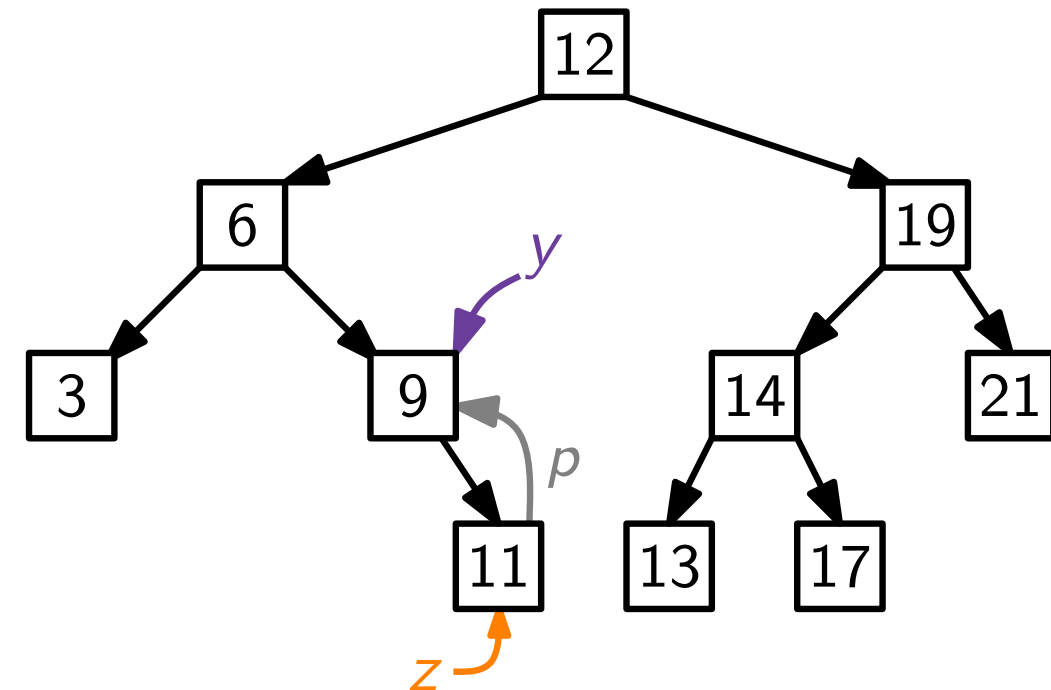
```
    if k < y.key then y.left = z
```

```
    else y.right = z
```

```
return z
```

INSERT(11)

x == nil



```
Node(Key k, Node par)
```

```
    key = k
```

```
    p = par
```

```
    right = left = nil
```

Einfügen

RBNode INSERT(key *k*)

y = *nil*

x = *root*

while *x* $\neq$ *nil* **do**

y = *x*

if *k* < *x*.key **then**

x = *x*.left

else *x* = *x*.right

z = **new** Node(*k*, *y*)

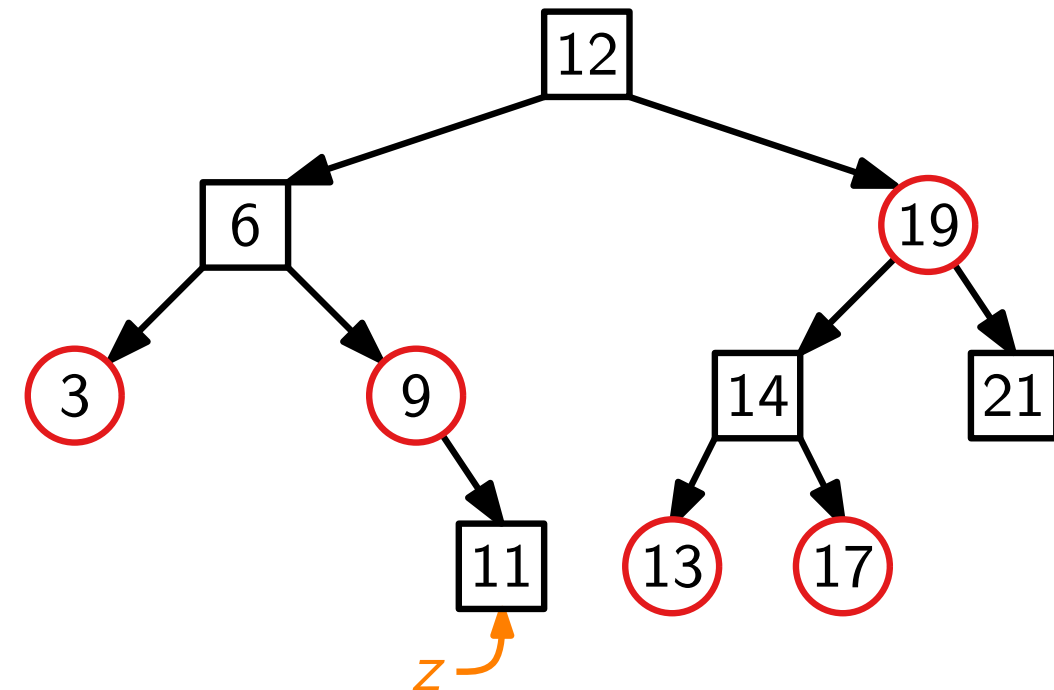
if *y* == *nil* **then** *root* = *z*

else

if *k* < *y*.key **then** *y*.left = *z*

else *y*.right = *z*

return *z*



Node(Key *k*, Node *par*)

key = *k*

p = *par*

right = *left* = *nil*

Einfügen

RBNode INSERT(key k)

$y = T.nil$

$x = root$

while $x \neq T.nil$ **do**

$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$

$z = \text{new Node}(k, y)$

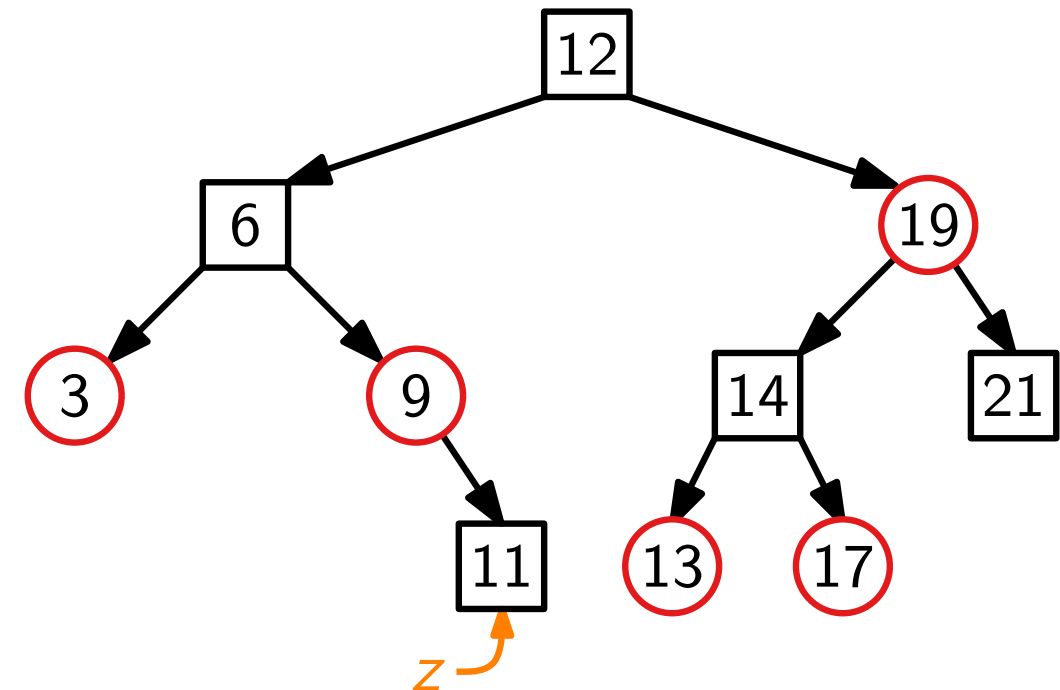
if $y == T.nil$ **then** $root = z$

else

if $k < y.key$ **then** $y.left = z$

else $y.right = z$

return z



Node(Key k , Node par)

$key = k$

$p = par$

$right = left = T.nil$

Einfügen

RBNode INSERT(key k)

$y = T.nil$

$x = root$

while $x \neq T.nil$ **do**

$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$

$z = \text{new RBNode}(k, y, \text{red})$

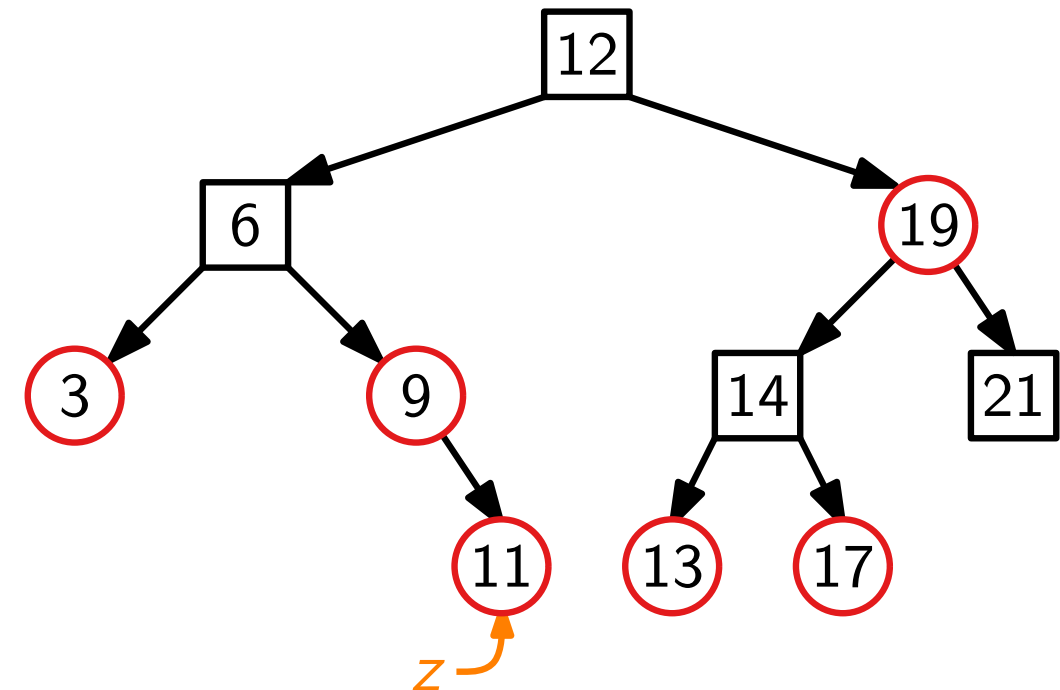
if $y == T.nil$ **then** $root = z$

else

if $k < y.key$ **then** $y.left = z$

else $y.right = z$

return z



Node(Key k , Node par)

$key = k$

$p = par$

$right = left = T.nil$

RBNode(..., Color c)

$\text{super}(k, par)$

$color = c$

Einfügen

RBNode INSERT(key k)

$y = T.nil$

$x = root$

while $x \neq T.nil$ **do**

$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$

$z = \text{new RBNode}(k, y, \text{red})$

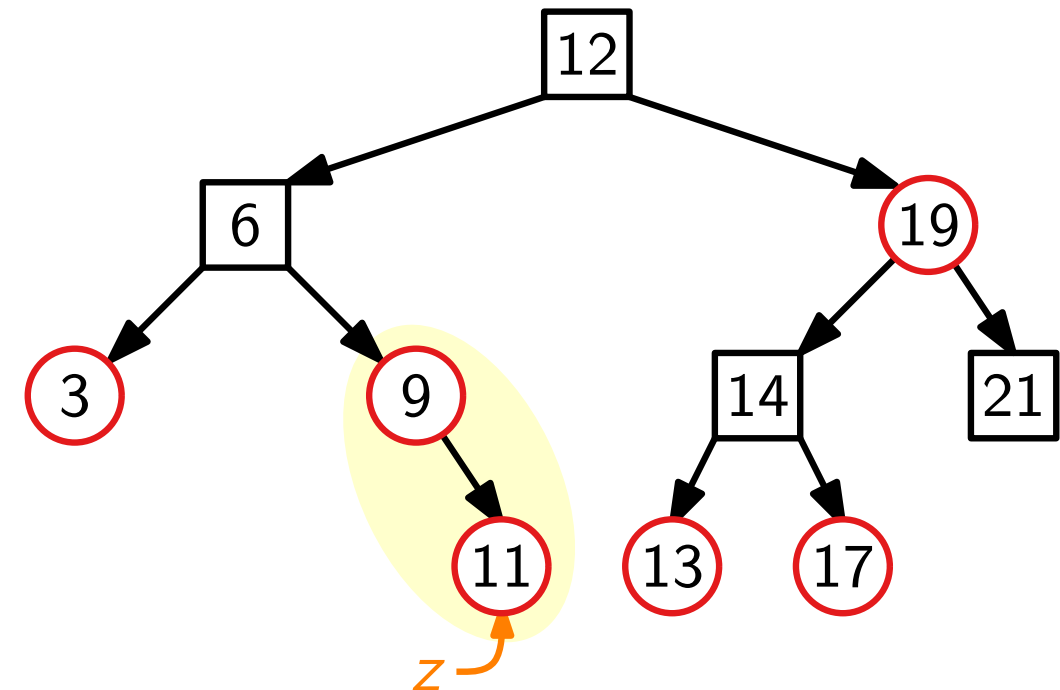
if $y == T.nil$ **then** $root = z$

else

if $k < y.key$ **then** $y.left = z$

else $y.right = z$

return z



Node(Key k , Node par)

$key = k$

$p = par$

$right = left = T.nil$

RBNode(..., Color c)

$super(k, par)$

$color = c$

Einfügen

RBNode INSERT(key k)

$y = T.nil$

$x = root$

while $x \neq T.nil$ **do**

$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$

$z = \text{new RBNode}(k, y, \text{red})$

if $y == T.nil$ **then** $root = z$

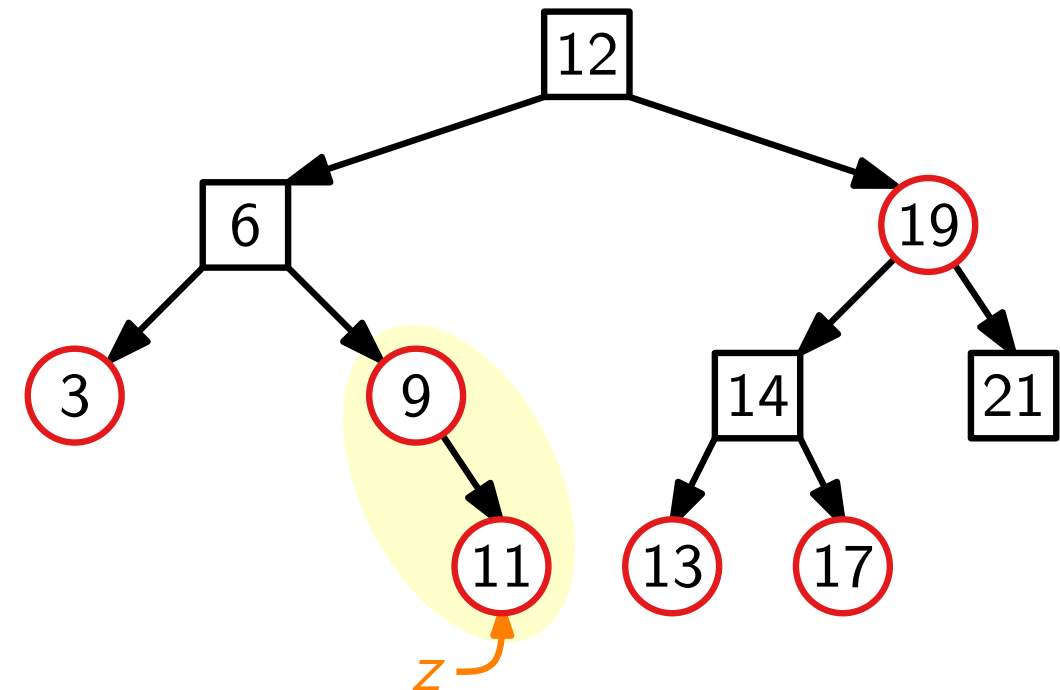
else

if $k < y.key$ **then** $y.left = z$

else $y.right = z$

return z

(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.



Node(Key k , Node par)

$key = k$

$p = par$

$right = left = T.nil$

RBNode(..., Color c)

$super(k, par)$

$color = c$

Einfügen

RBNode INSERT(key k)

$y = T.nil$

$x = root$

while $x \neq T.nil$ **do**

$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$

$z = \text{new RBNode}(k, y, red)$

if $y == T.nil$ **then** $root = z$

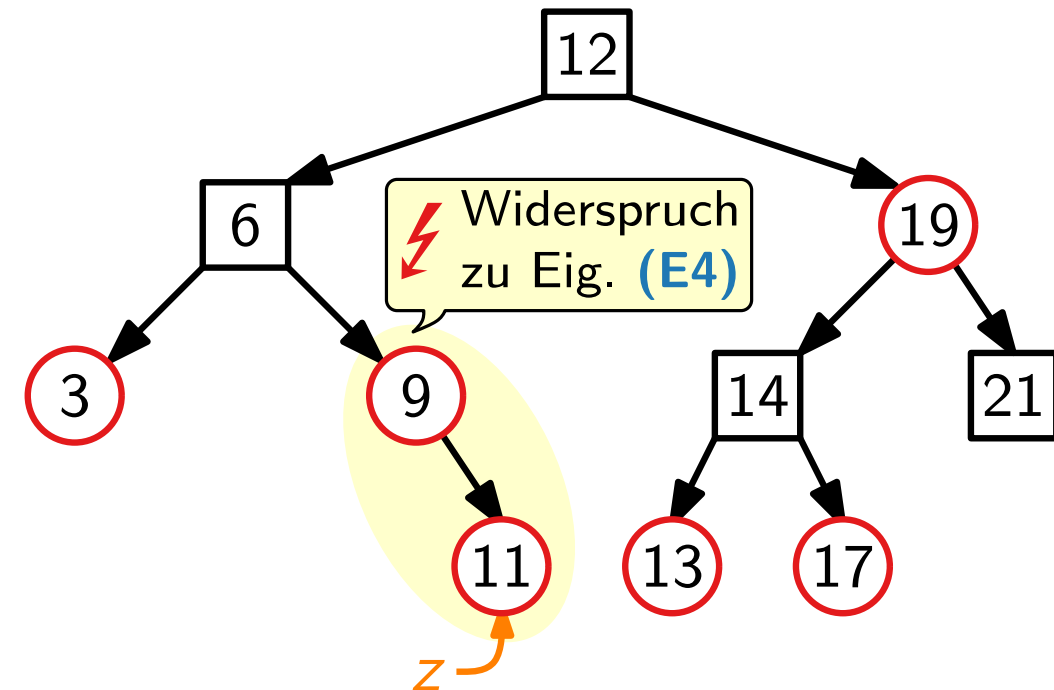
else

if $k < y.key$ **then** $y.left = z$

else $y.right = z$

return z

(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.



Node(Key k , Node par)

$key = k$

$p = par$

$right = left = T.nil$

RBNode(..., Color c)

$super(k, par)$

$color = c$

Einfügen

RBNode INSERT(key k)

$y = T.nil$

$x = root$

while $x \neq T.nil$ **do**

$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$

$z = \text{new RBNode}(k, y, \text{red})$

if $y == T.nil$ **then** $root = z$

else

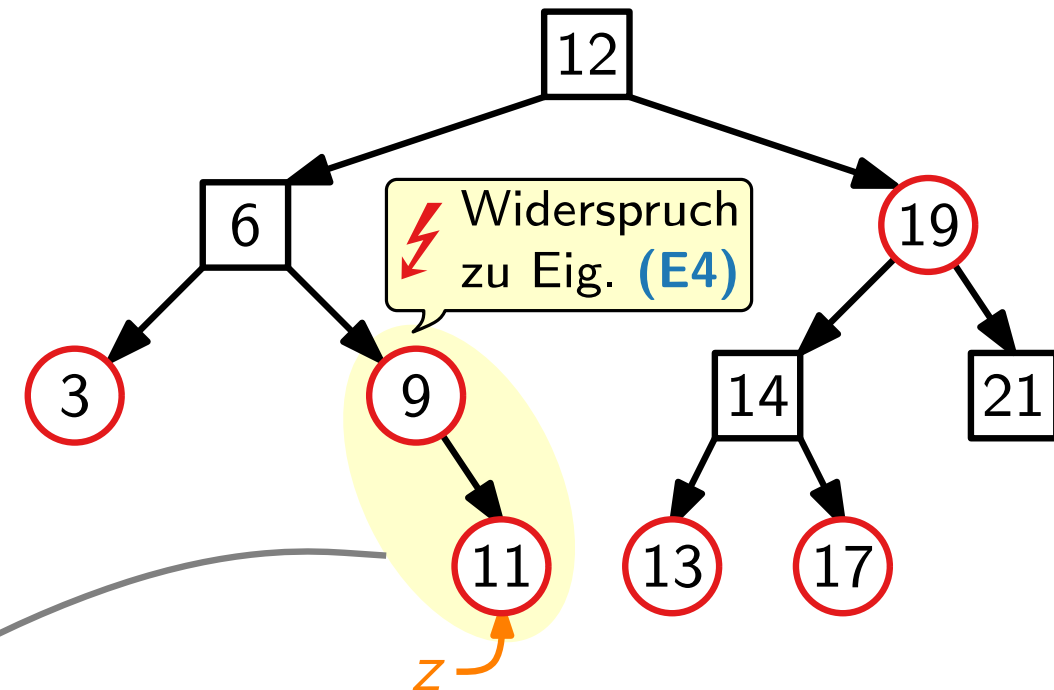
if $k < y.key$ **then** $y.left = z$

else $y.right = z$

RBINSERTFIXUP(z)

return z

(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.



Node(Key k , Node par)
 $key = k$
 $p = par$
 $right = left = T.nil$

RBNode(..., Color c)
 $super(k, par)$
 $color = c$

Einfügen

Laufzeit? (ohne RBINSERTFIXUP)

RBNode INSERT(key k)

$y = T.nil$

$x = root$

while $x \neq T.nil$ **do**

$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$

$z = \text{new RBNode}(k, y, \text{red})$

if $y == T.nil$ **then** $root = z$

else

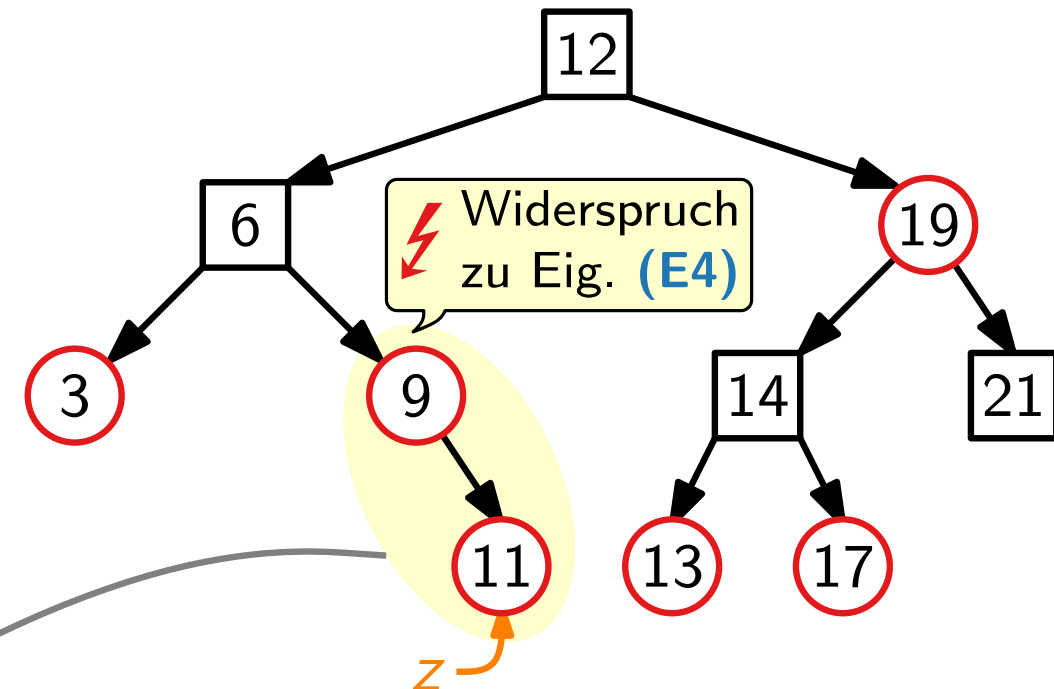
if $k < y.key$ **then** $y.left = z$

else $y.right = z$

RBINSERTFIXUP(z)

return z

(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.



Node(Key k , Node par)
 $key = k$
 $p = par$
 $right = left = T.nil$

RBNode(..., Color c)
 $super(k, par)$
 $color = c$

Einfügen

Laufzeit? (ohne RBINSERTFIXUP) $\mathcal{O}(h)$

RBNode INSERT(key k)

$y = T.nil$

$x = root$

while $x \neq T.nil$ **do**

$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$

$z = \text{new RBNode}(k, y, \text{red})$

if $y == T.nil$ **then** $root = z$

else

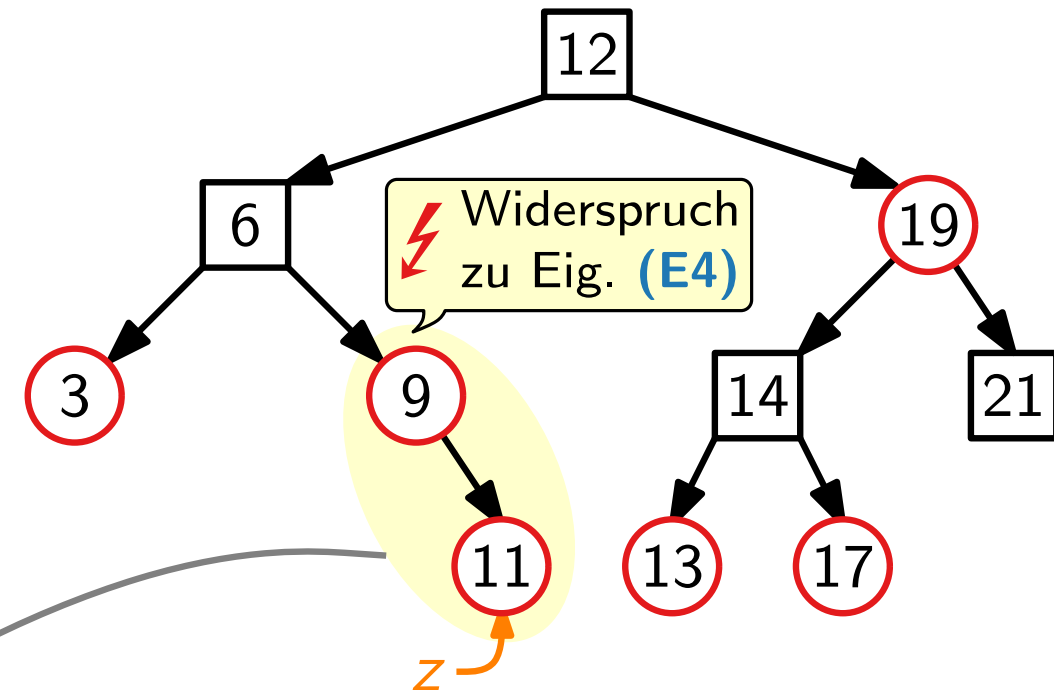
if $k < y.key$ **then** $y.left = z$

else $y.right = z$

RBINSERTFIXUP(z)

return z

(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.



Node(Key k , Node par)
 $key = k$
 $p = par$
 $right = left = T.nil$

RBNode(..., Color c)
 $super(k, par)$
 $color = c$

Einfügen

Laufzeit? (ohne RBINSERTFIXUP) $\mathcal{O}(h) = \mathcal{O}(\log n)$ 😊

RBNode INSERT(key k)

$y = T.nil$

$x = root$

while $x \neq T.nil$ **do**

$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$

$z = \text{new RBNode}(k, y, \text{red})$

if $y == T.nil$ **then** $root = z$

else

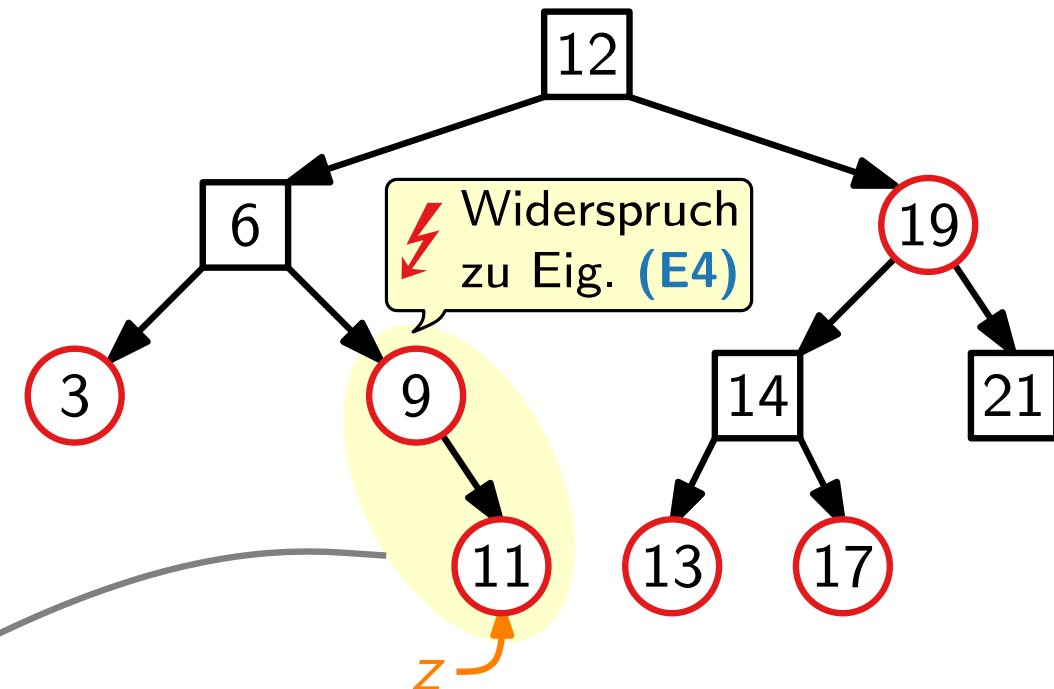
if $k < y.key$ **then** $y.left = z$

else $y.right = z$

RBINSERTFIXUP(z)

return z

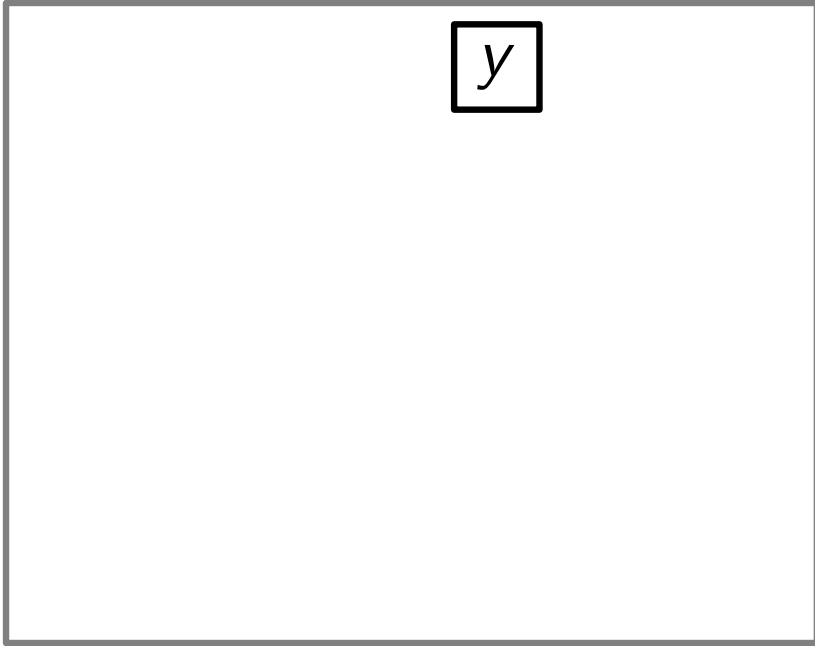
(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.



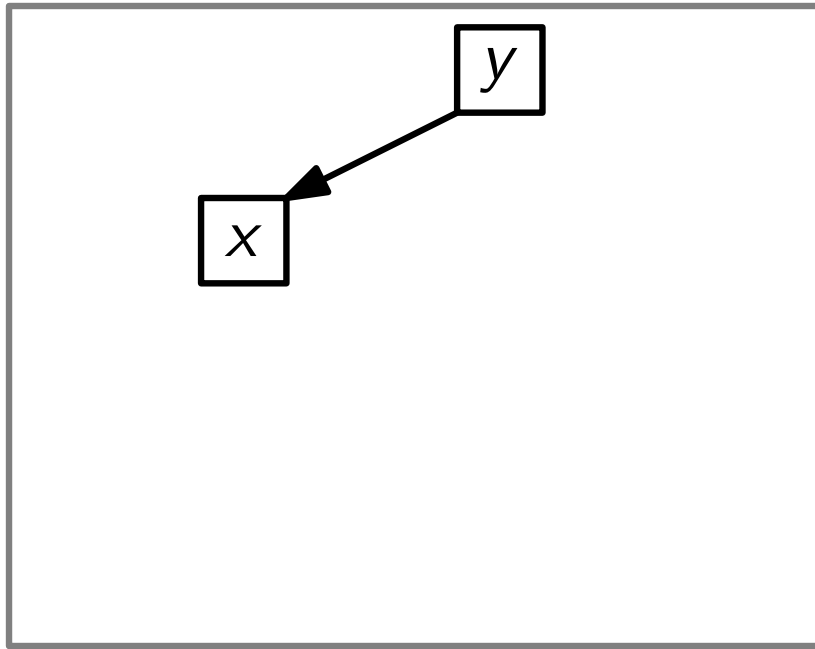
Node(Key k , Node par)
 $key = k$
 $p = par$
 $right = left = T.nil$

RBNode(..., Color c)
 $super(k, par)$
 $color = c$

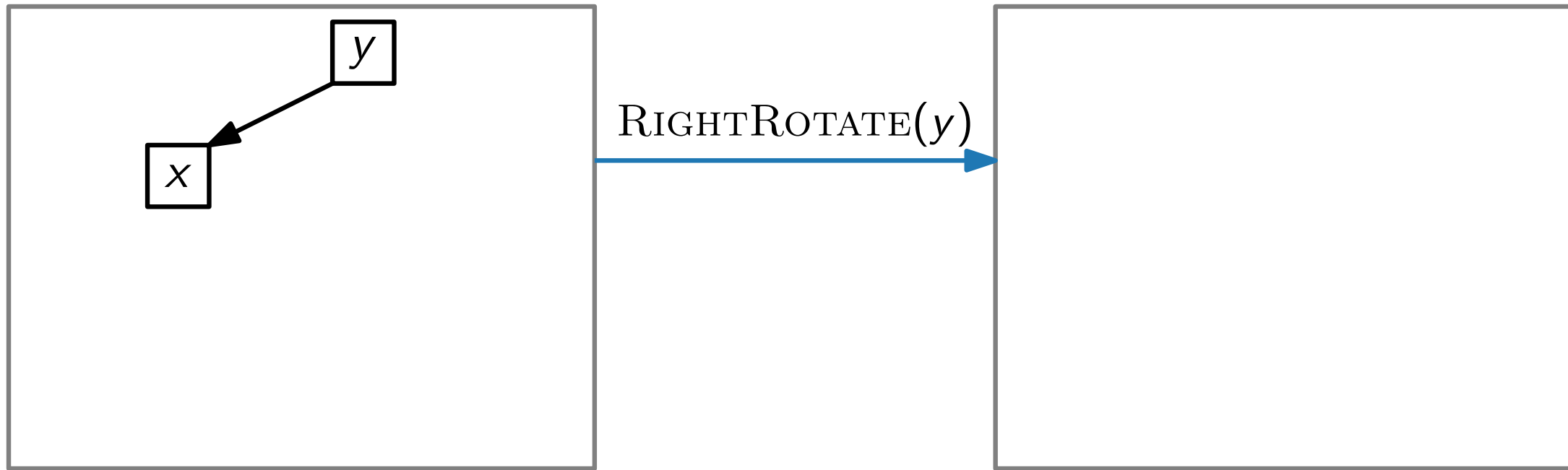
Exkurs: Rotationen



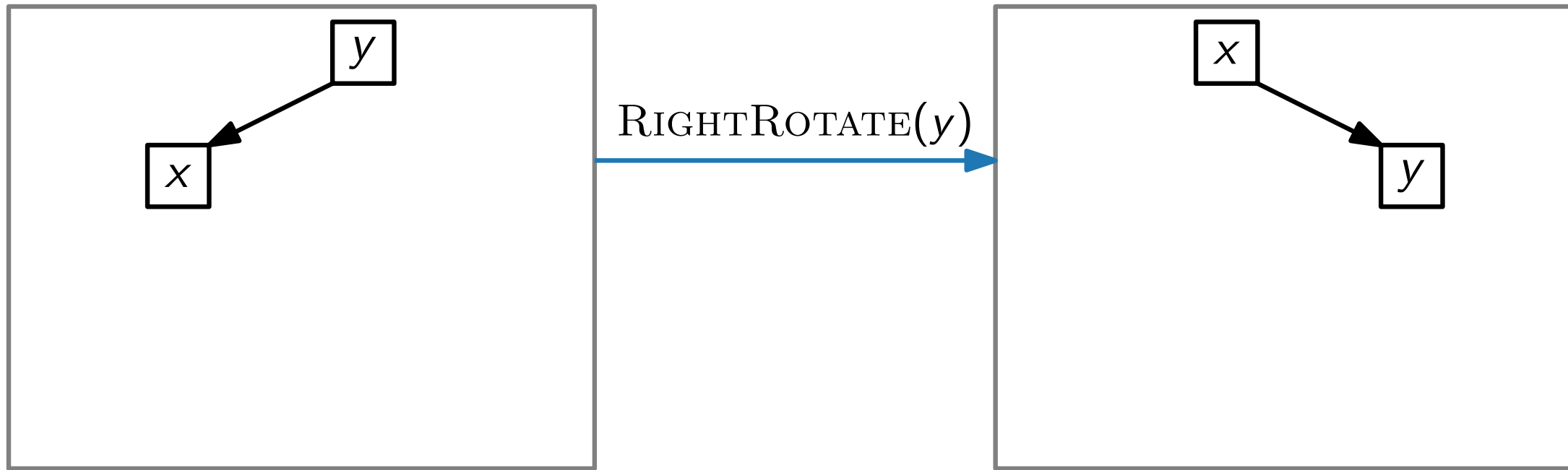
Exkurs: Rotationen



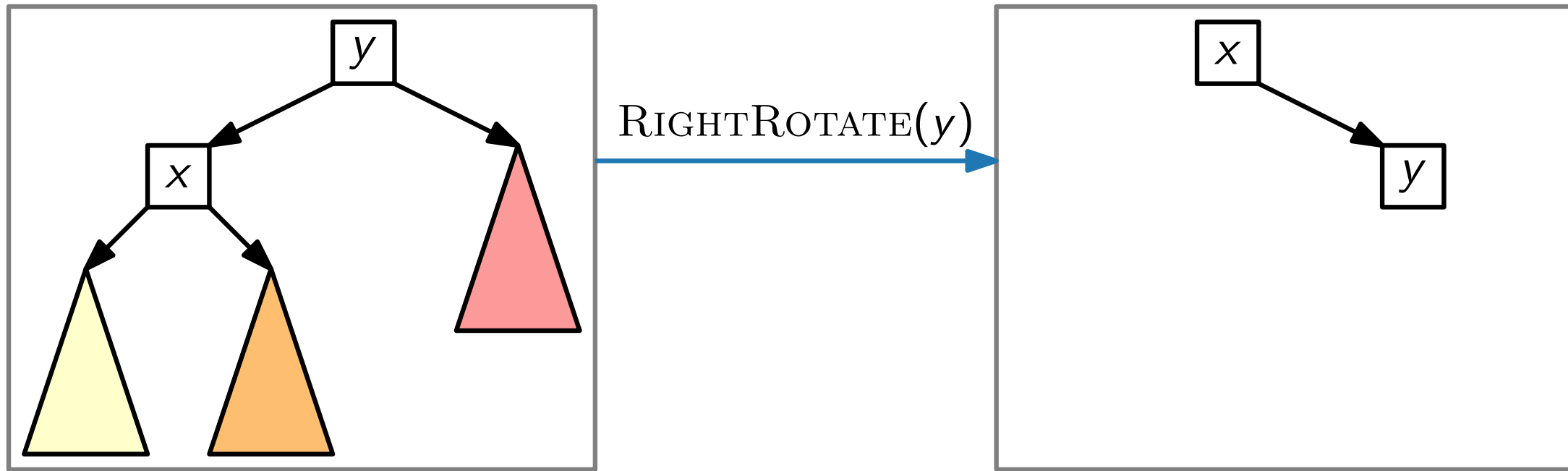
Exkurs: Rotationen



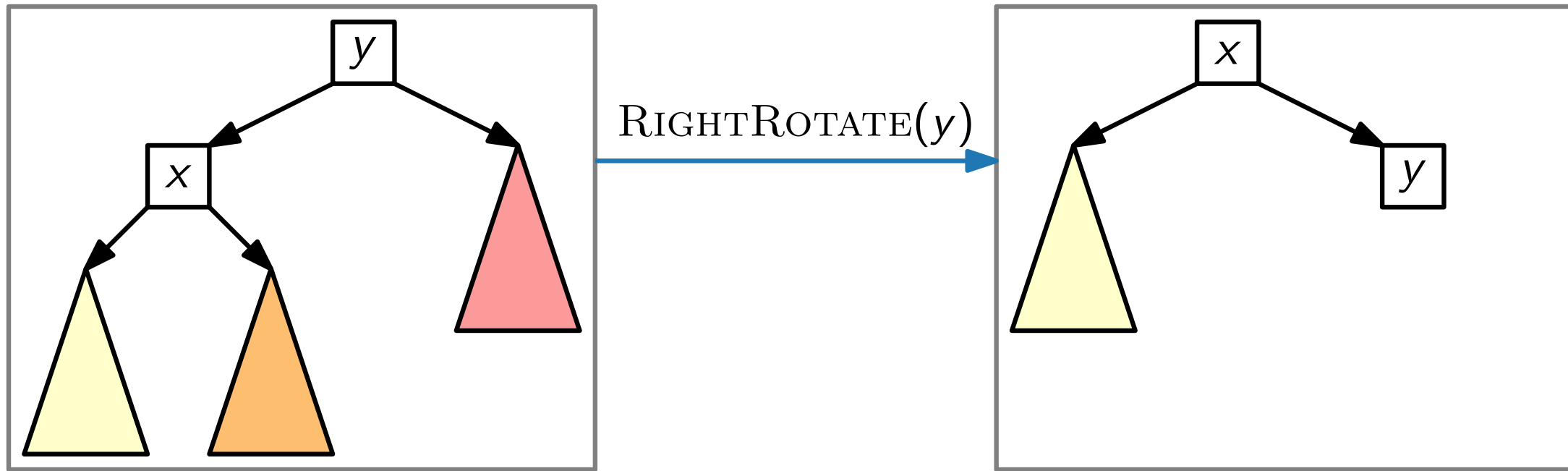
Exkurs: Rotationen



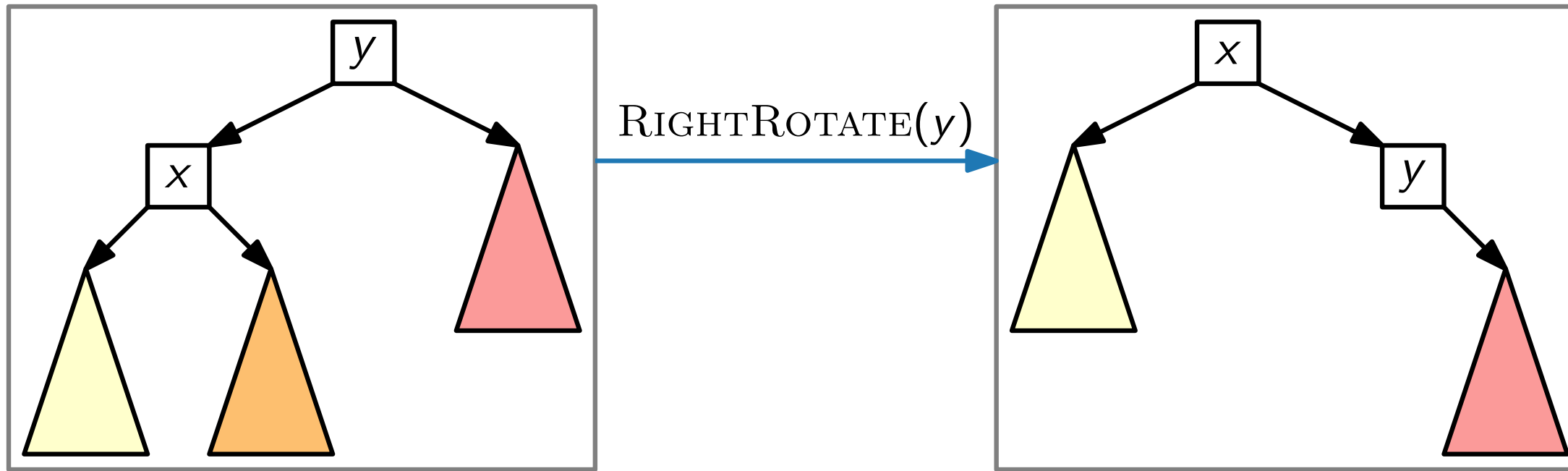
Exkurs: Rotationen



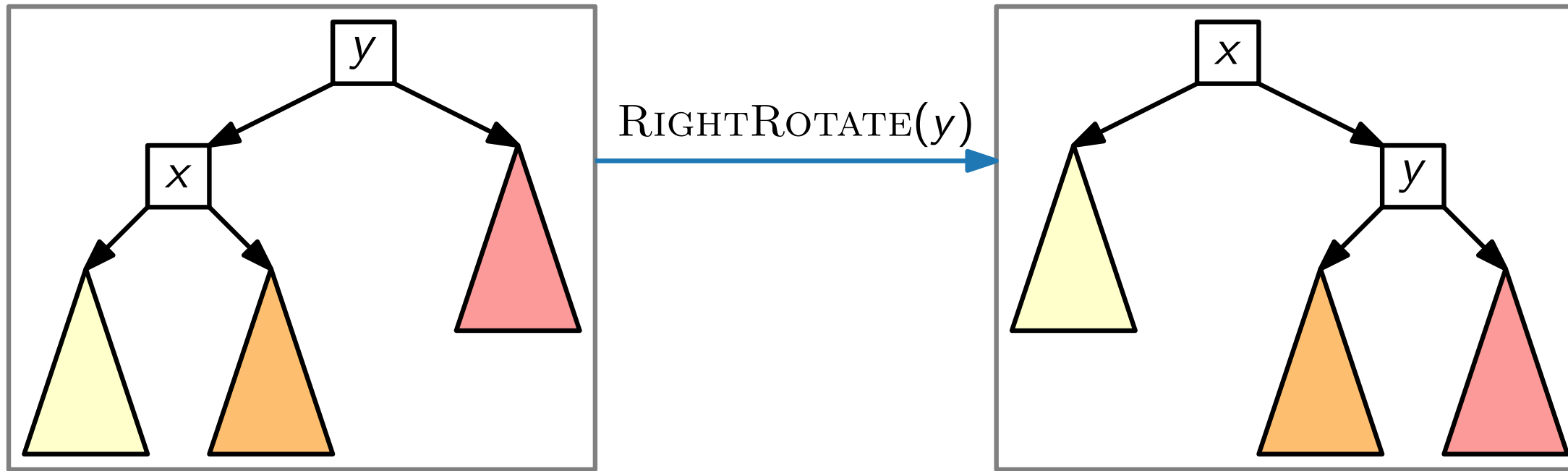
Exkurs: Rotationen



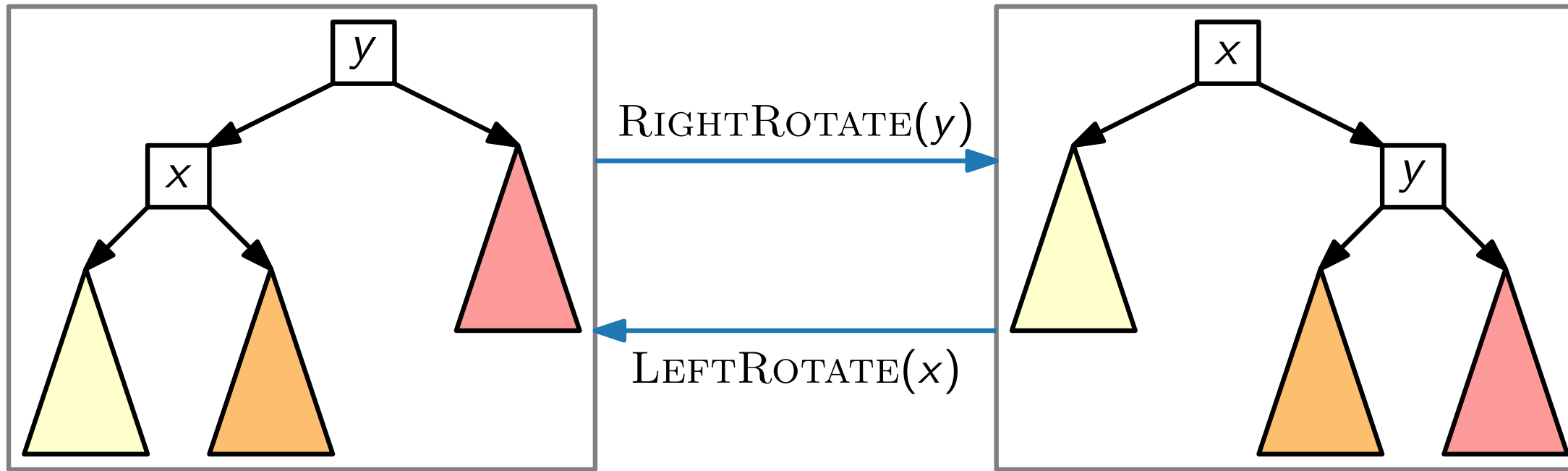
Exkurs: Rotationen



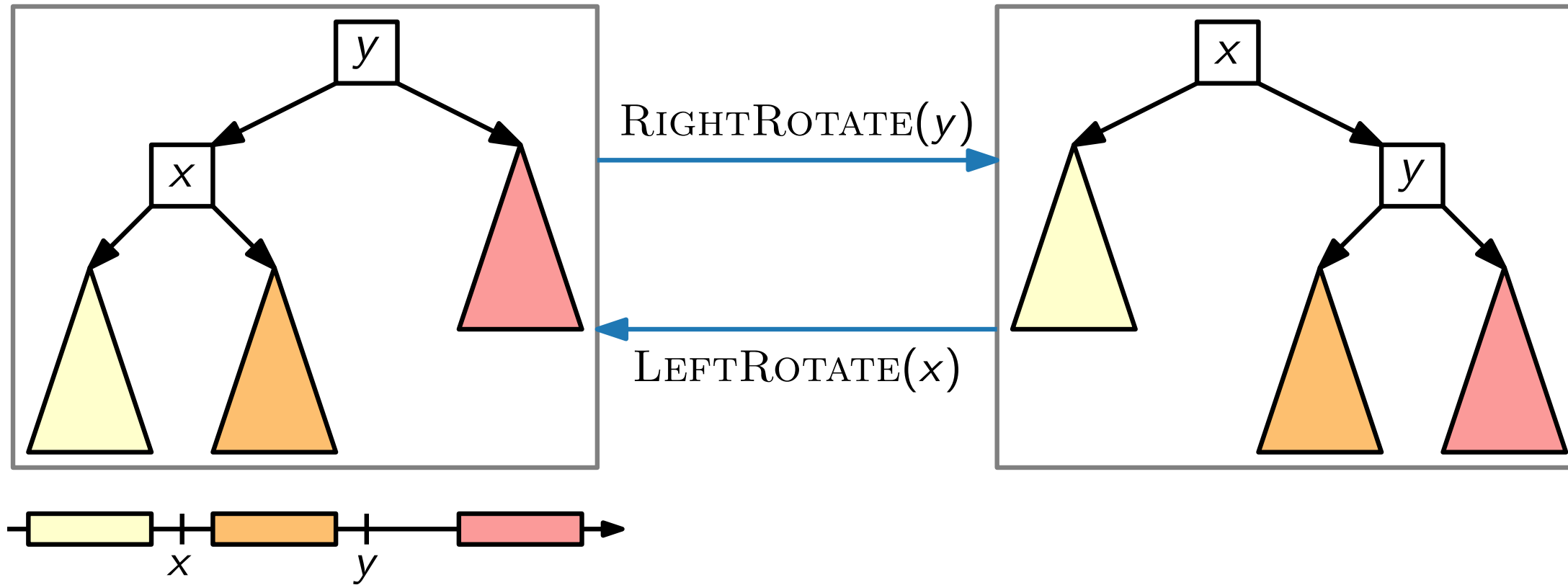
Exkurs: Rotationen



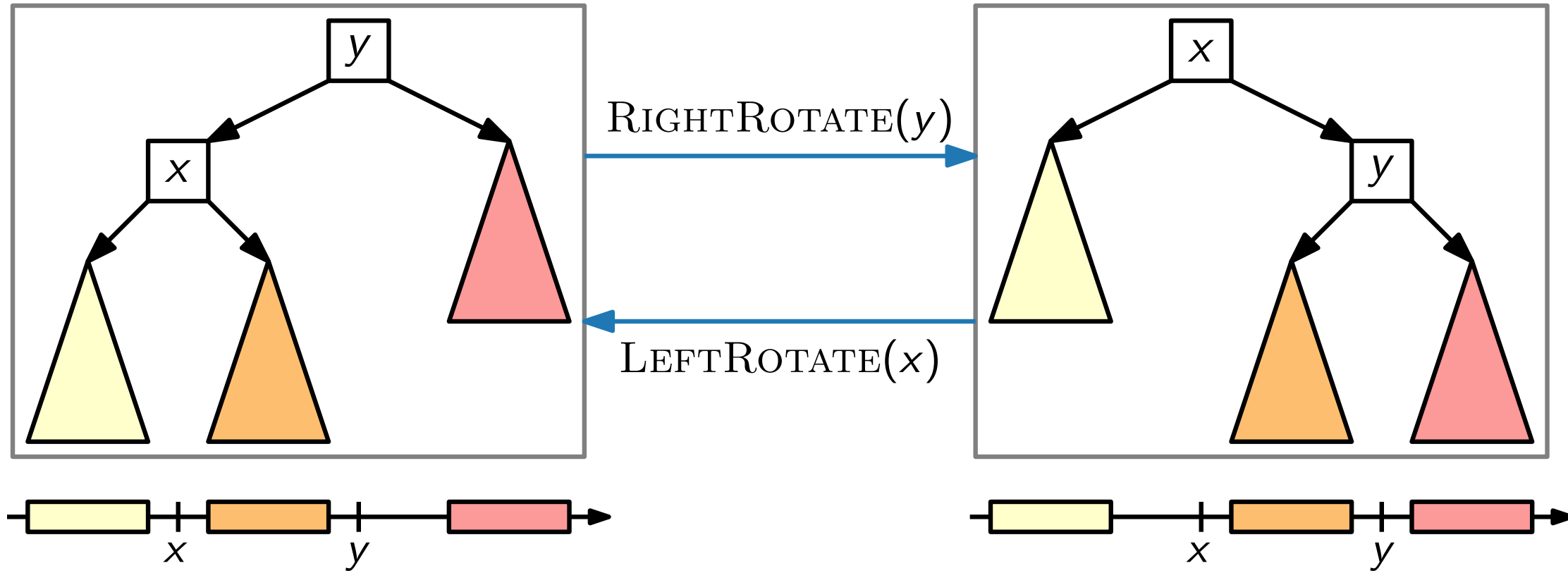
Exkurs: Rotationen



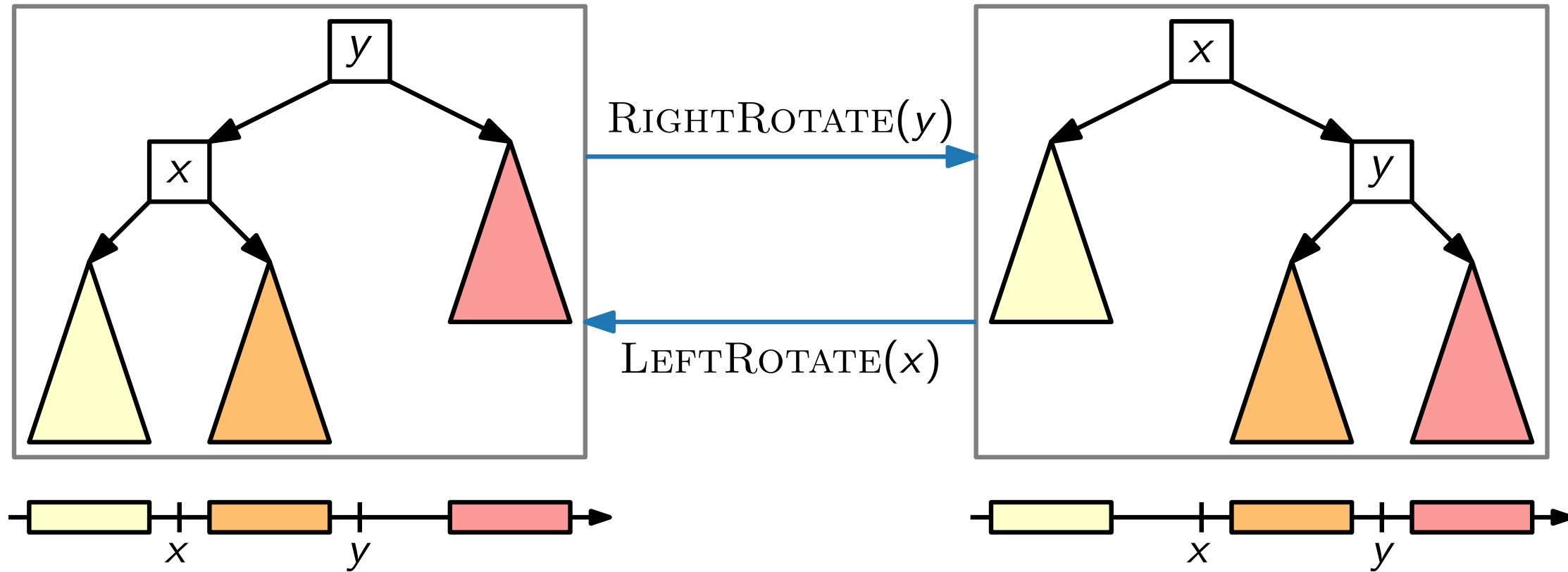
Exkurs: Rotationen



Exkurs: Rotationen

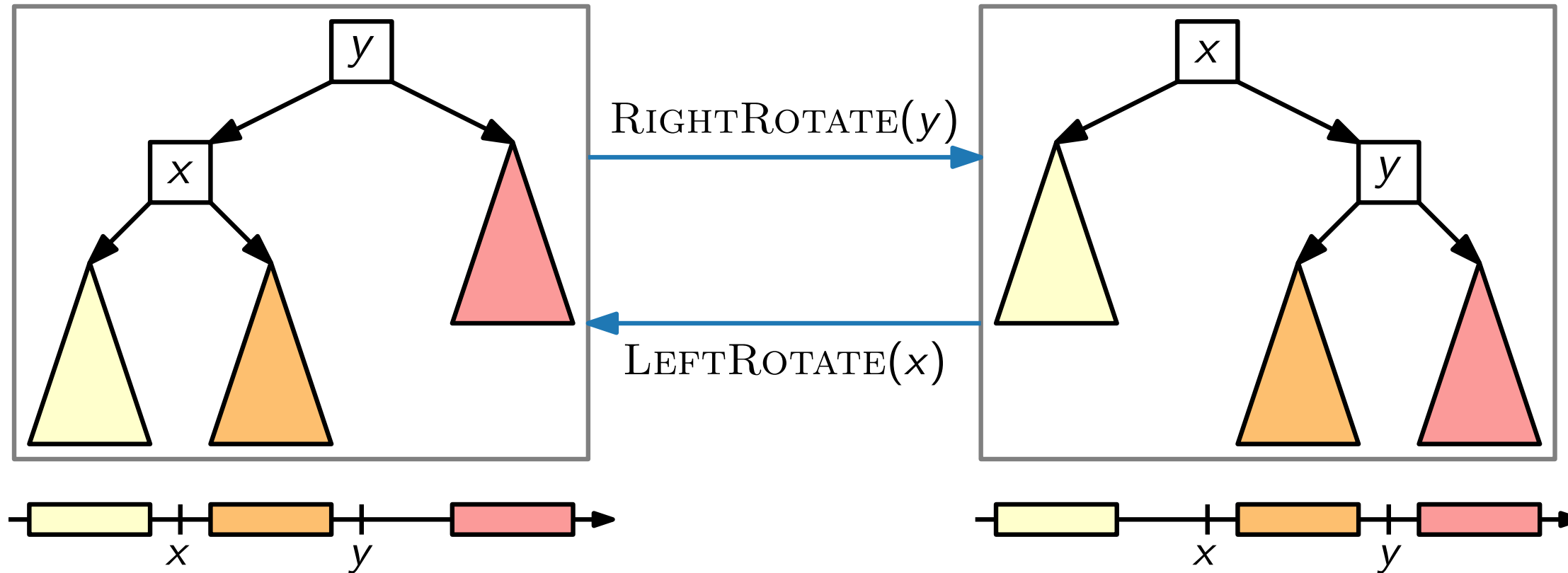


Exkurs: Rotationen



Also: **Binärer-Suchbaum-Eigenschaft** bleibt beim Rotieren erhalten!

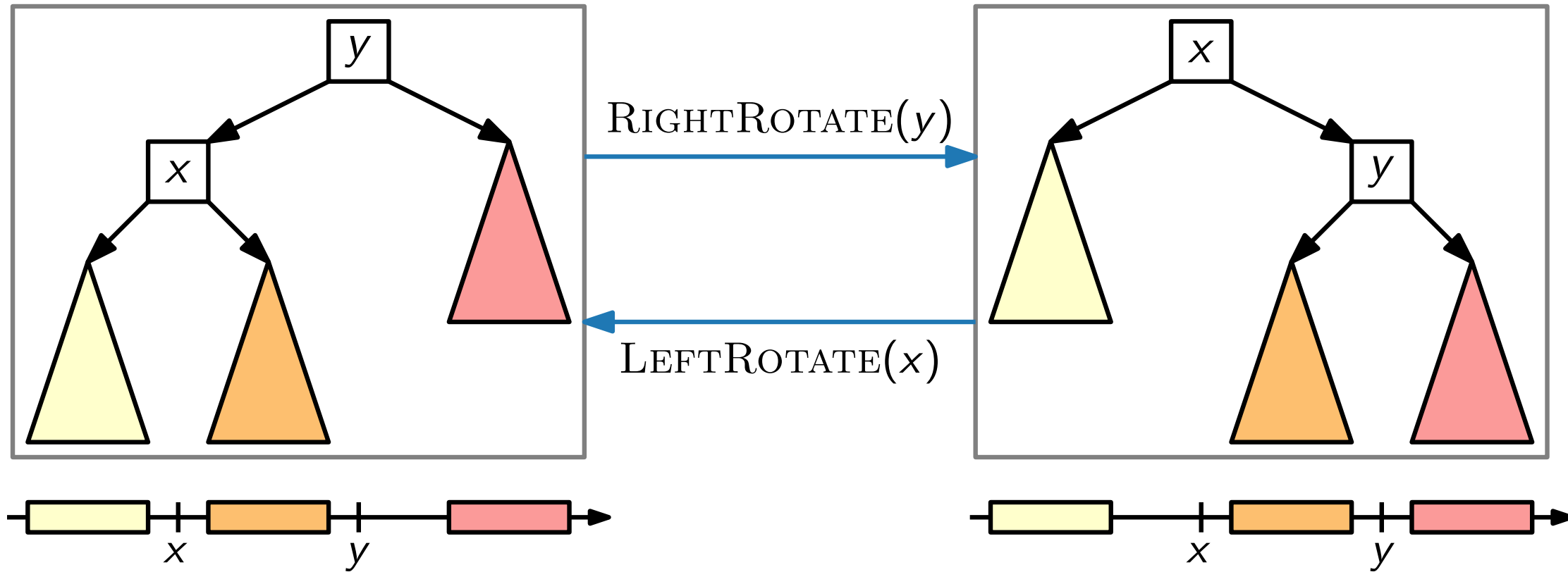
Exkurs: Rotationen



Also: **Binärer-Suchbaum-Eigenschaft** bleibt beim Rotieren erhalten!

Aufgabe: Schreiben Sie Pseudocode für $\text{LEFTROTATE}(x)$!

Exkurs: Rotationen

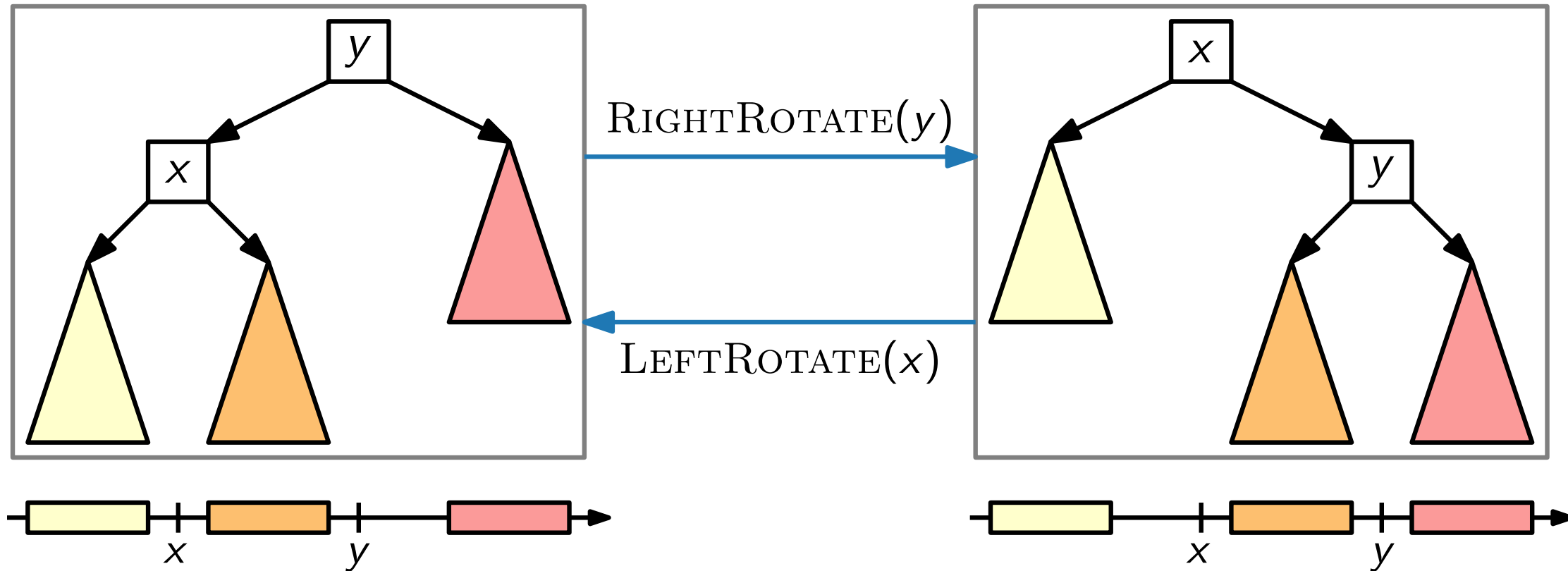


Also: **Binärer-Suchbaum-Eigenschaft** bleibt beim Rotieren erhalten!

Aufgabe: Schreiben Sie Pseudocode für **LEFTROTATE(x)**!

Laufzeit:

Exkurs: Rotationen

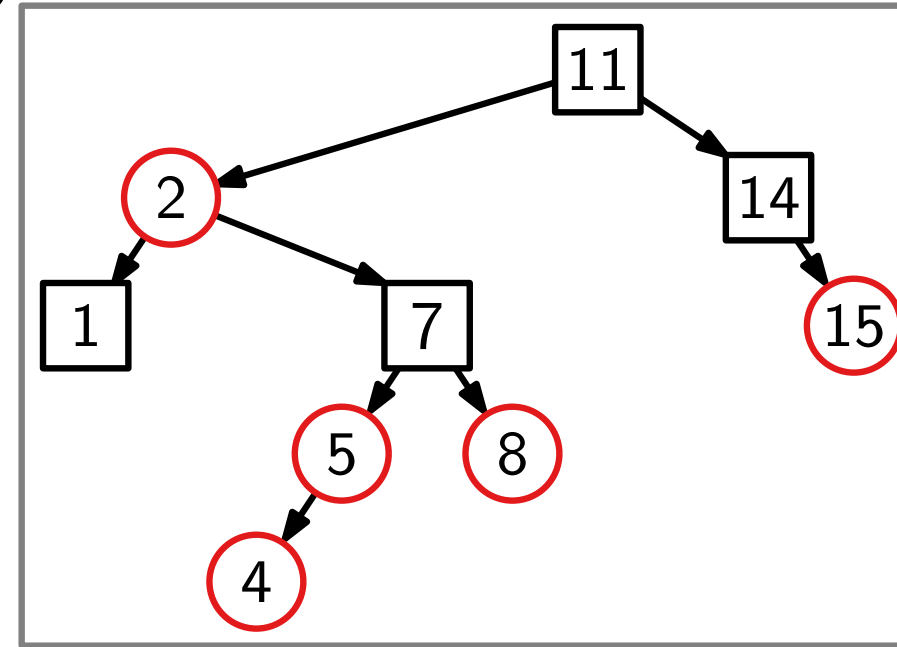


Also: **Binärer-Suchbaum-Eigenschaft** bleibt beim Rotieren erhalten!

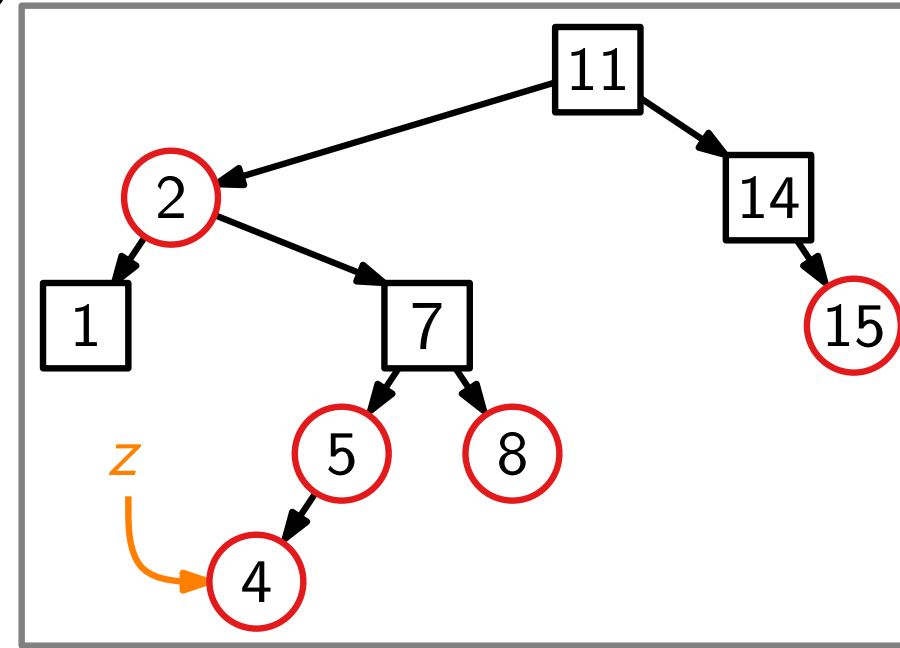
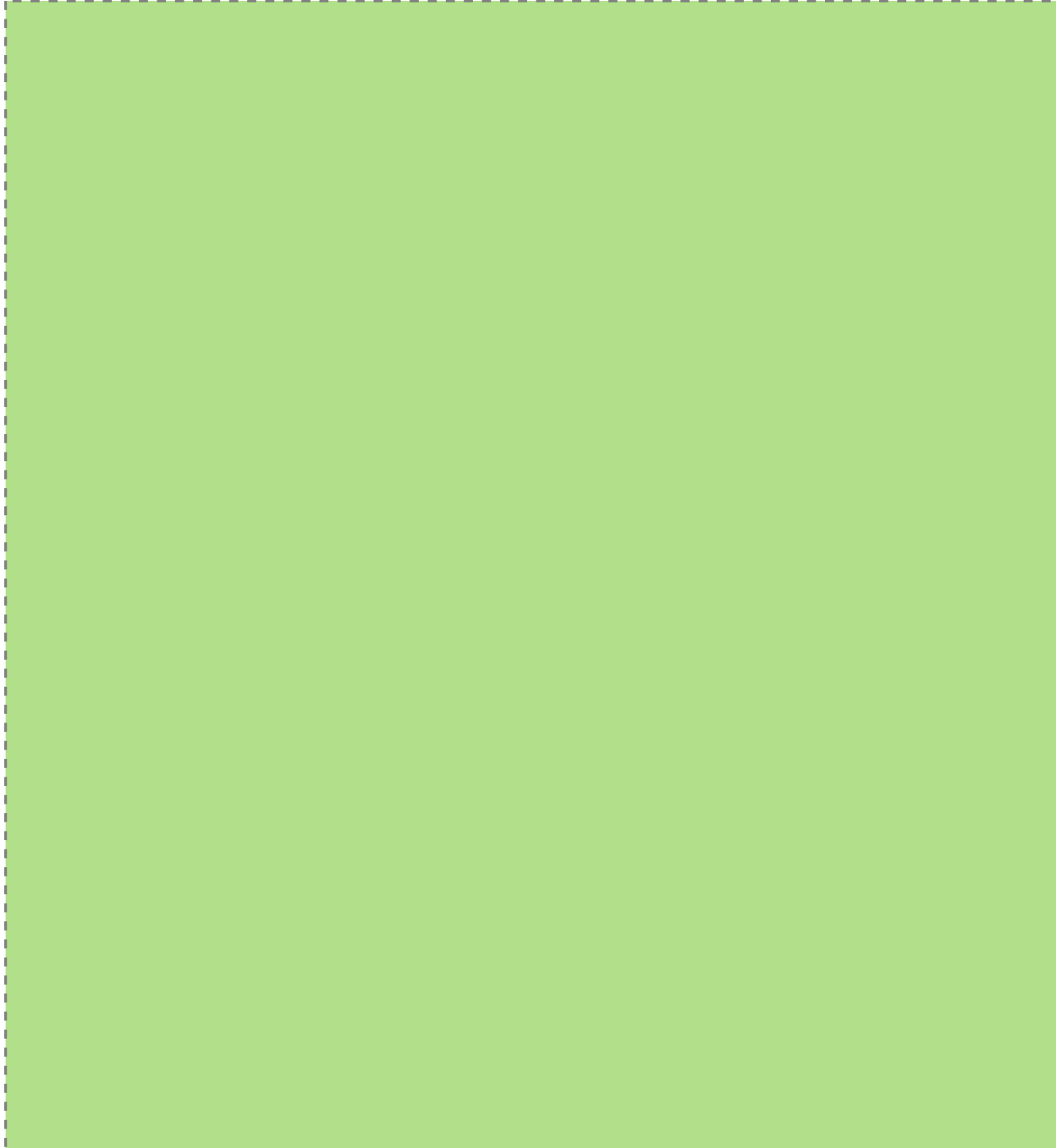
Aufgabe: Schreiben Sie Pseudocode für $\text{LEFTROTATE}(x)$!

Laufzeit: $\mathcal{O}(1)$. 😊

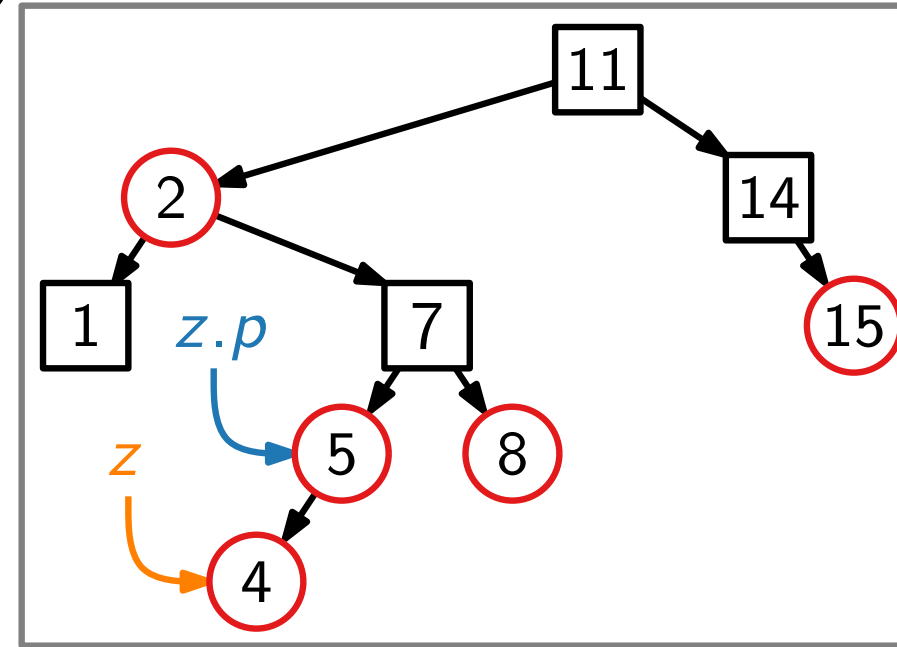
RBINSERTFIXUP(RBNode *z*)



RBINSERTFIXUP(RBNode z)

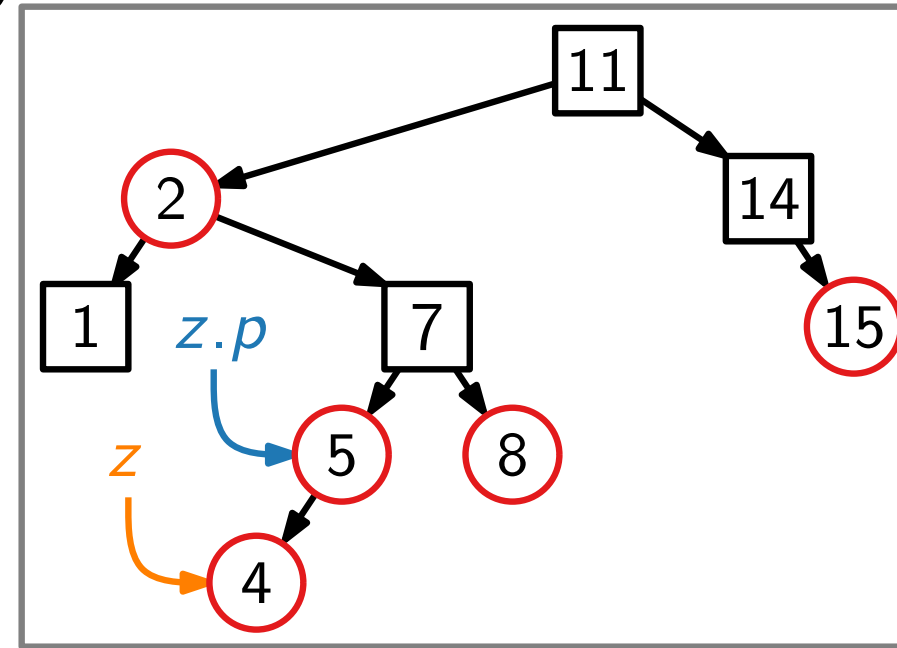


RBINSERTFIXUP(RBNode *z*)



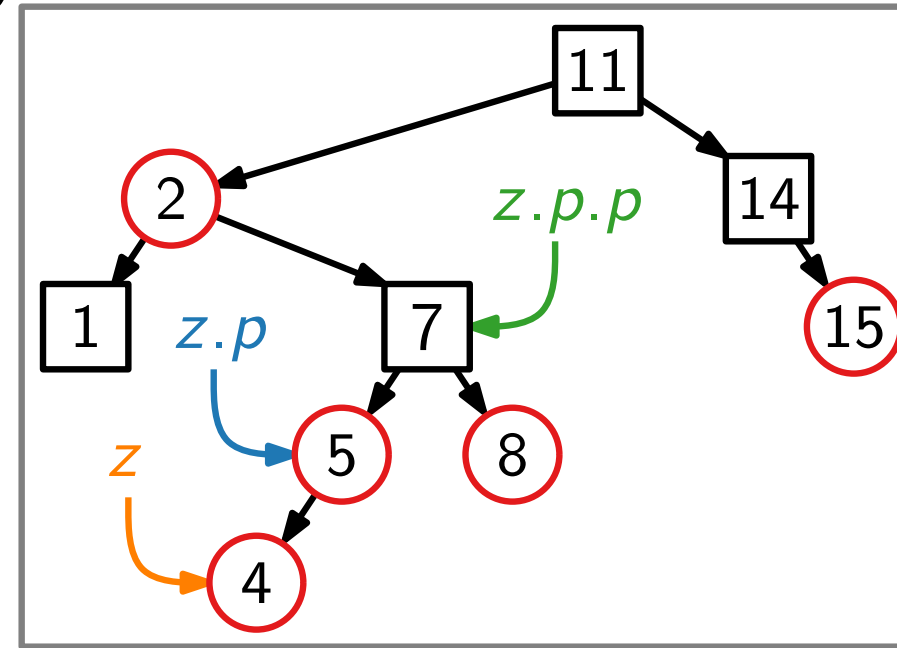
RBINSERTFIXUP(RBNode *z*)

```
while z.p.color == red do
```



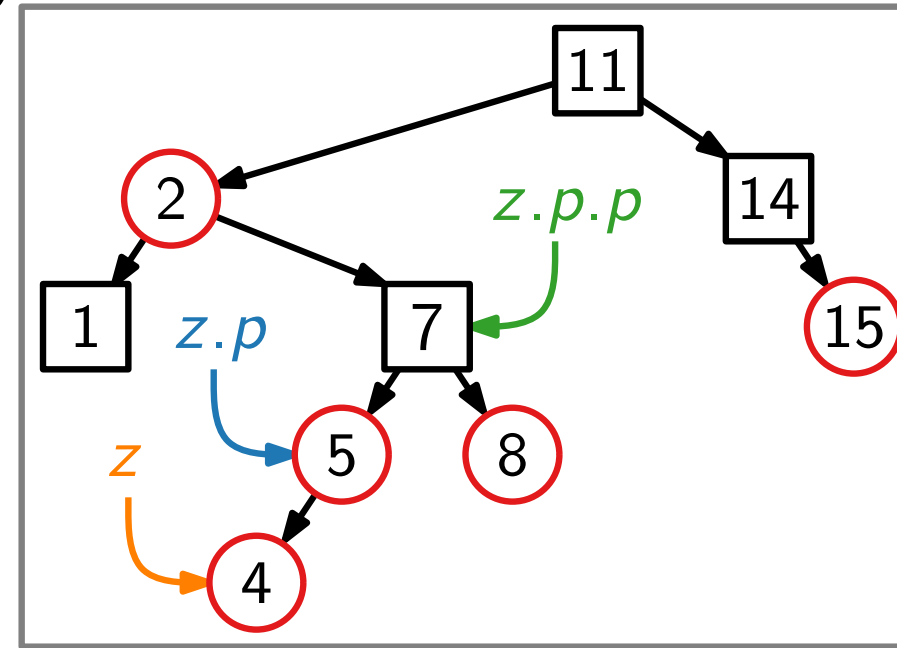
RBINSERTFIXUP(RBNode *z*)

```
while z.p.color == red do
```



RBINSERTFIXUP(RBNode *z*)

```
while z.p.color == red do
  if z.p == z.p.p.left then
    [ ]
  else
    [ ]
```

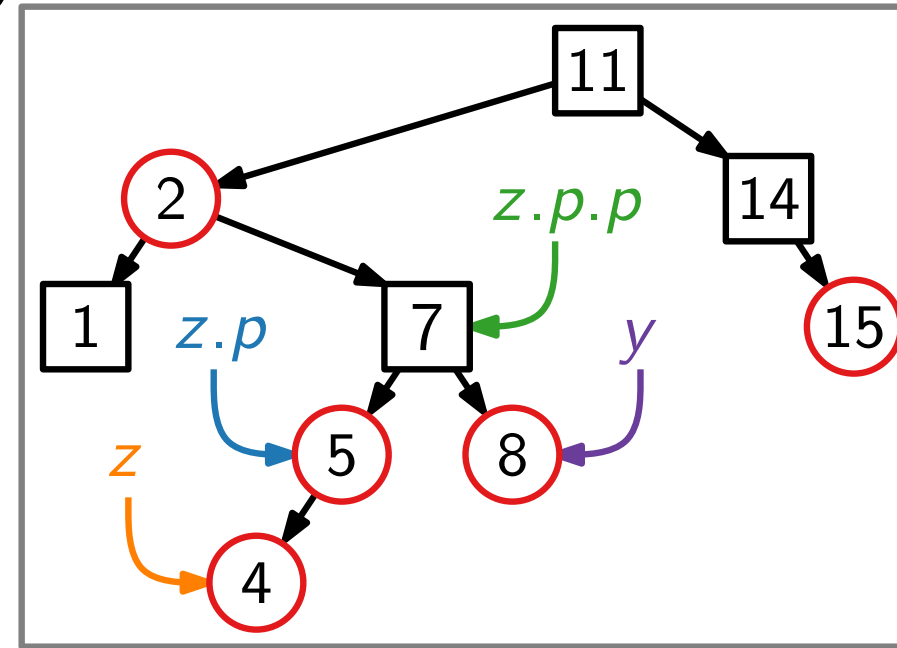


RBINSERTFIXUP(RBNode *z*)

```

while z.p.color == red do
  if z.p == z.p.p.left then
    y = z.p.p.right    // Tante von z
  else

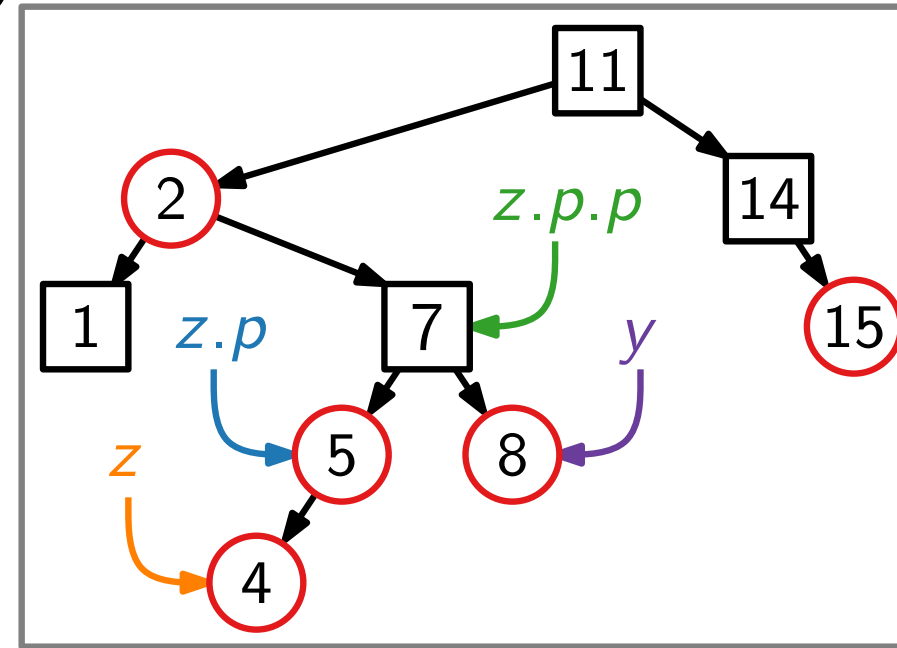
```



RBINSERTFIXUP(RBNode *z*)

```

while z.p.color == red do
  if z.p == z.p.p.left then
    y = z.p.p.right // Tante von z
    if y.color == red then
      // Case 1: Uncle is red
    else
      // Case 2: Uncle is black
  else
    // Case 3: z is the right child of its parent
  
```

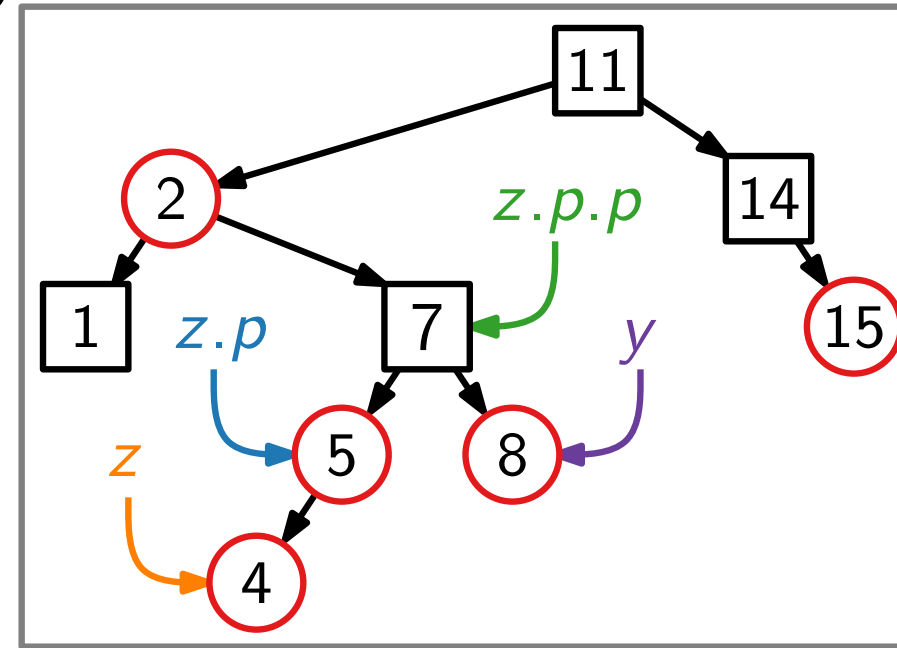


RBINSERTFIXUP(RBNode z)

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      // Left-Right or Right-Left case
      // (Implementation details for this case are not shown in the provided code snippet)
  else
    // z is the right child of its parent
    // (Implementation details for this case are not shown in the provided code snippet)

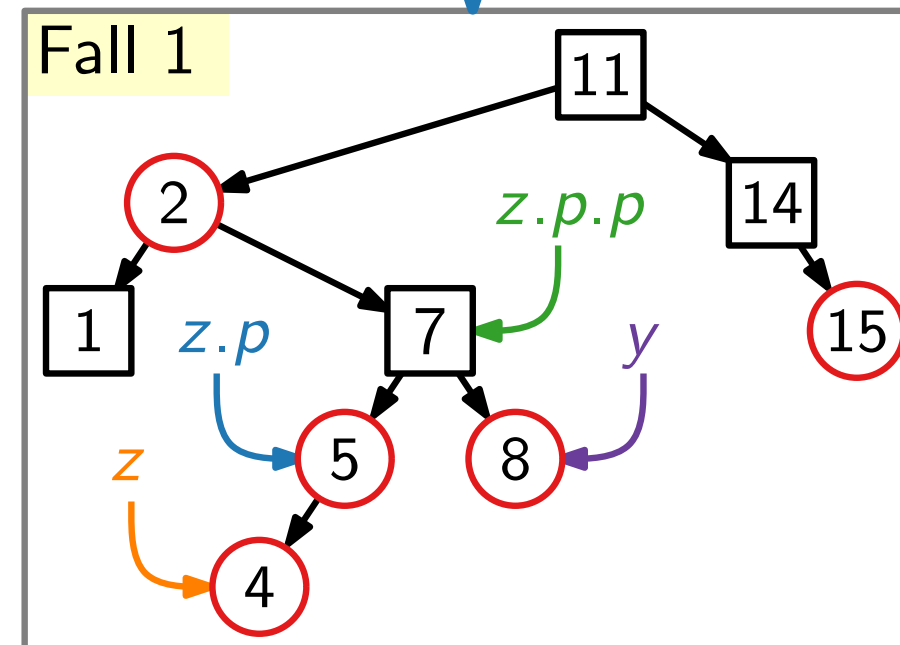
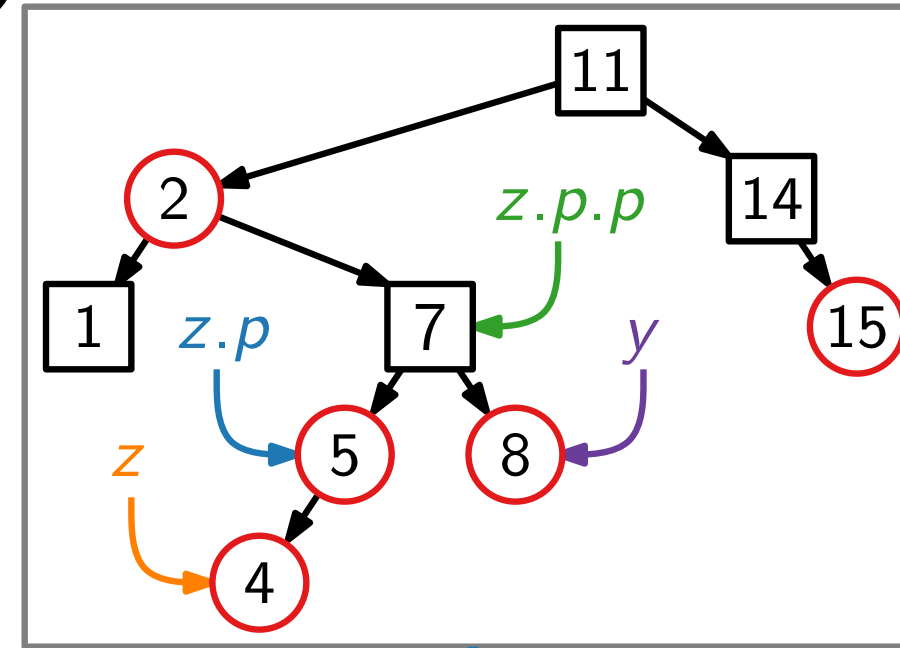
```



RBINSERTFIXUP(RBNode z)

```

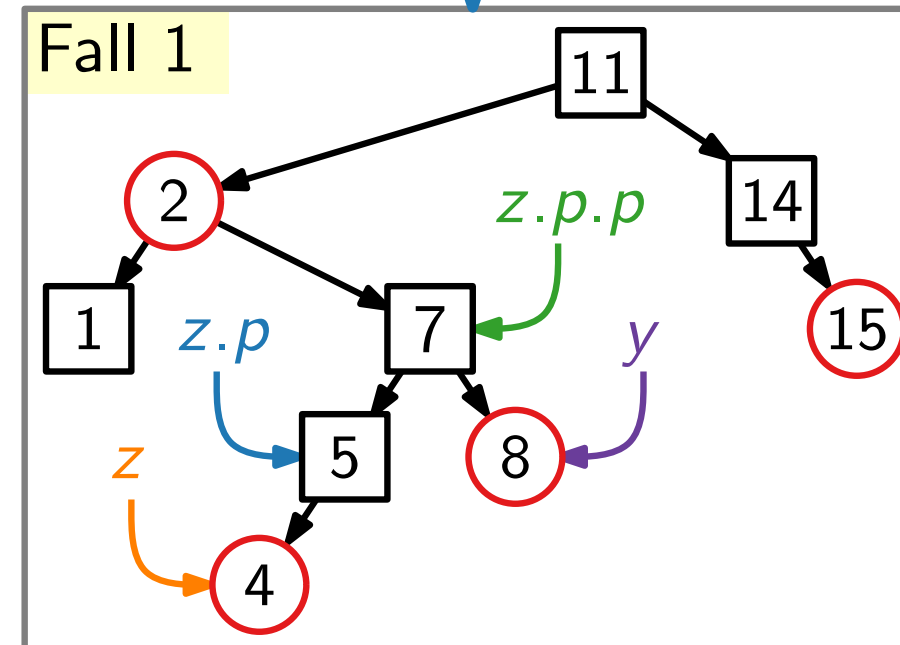
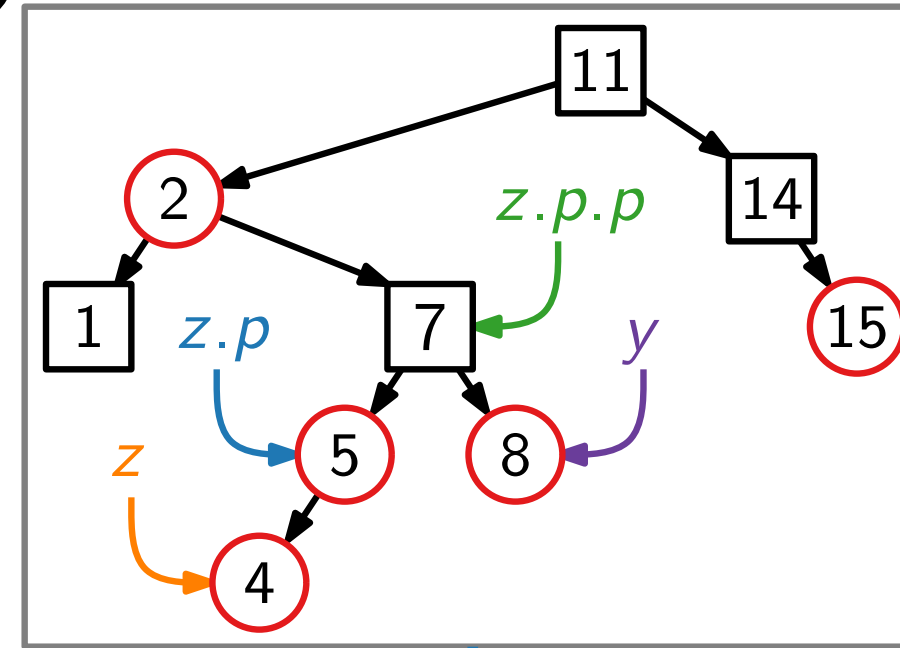
while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      // Case 1: Left-Right or Right-Left
      // Case 2: Left-Left or Right-Right
  else
    // Case 3: Left-Left or Right-Right
  
```



RBINSERTFIXUP(RBNode z)

```

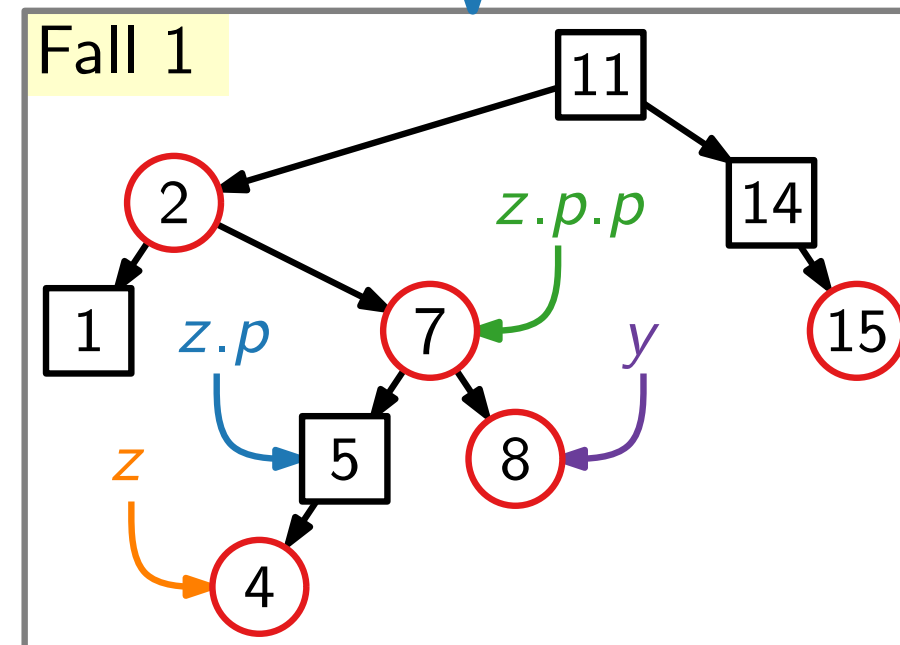
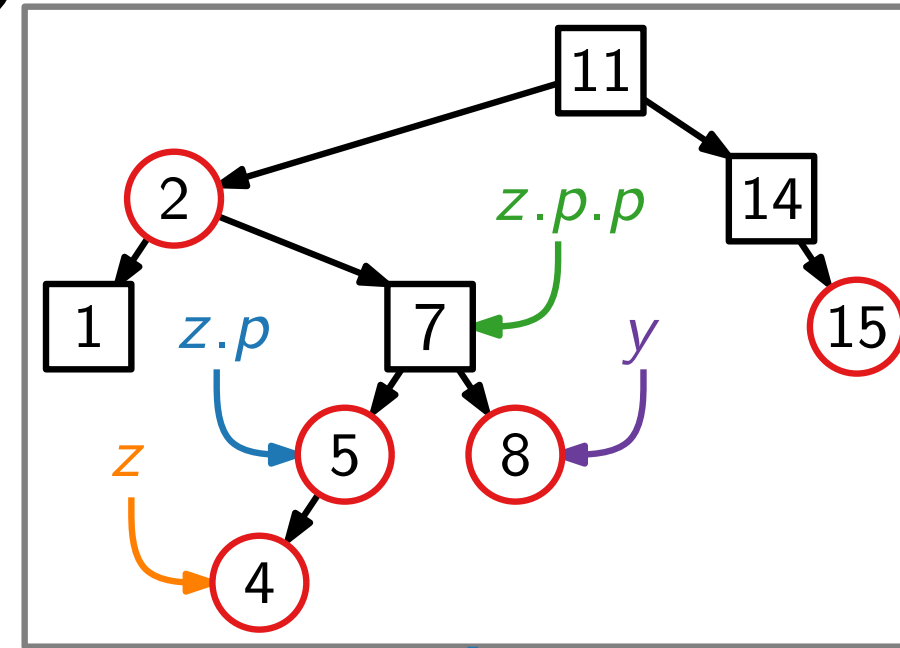
while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      // Case 1: Left-Right or Right-Left
      // Case 2: Left-Left or Right-Right
  else
    // Case 3: Left-Left or Right-Right
  
```



RBINSERTFIXUP(RBNode z)

```

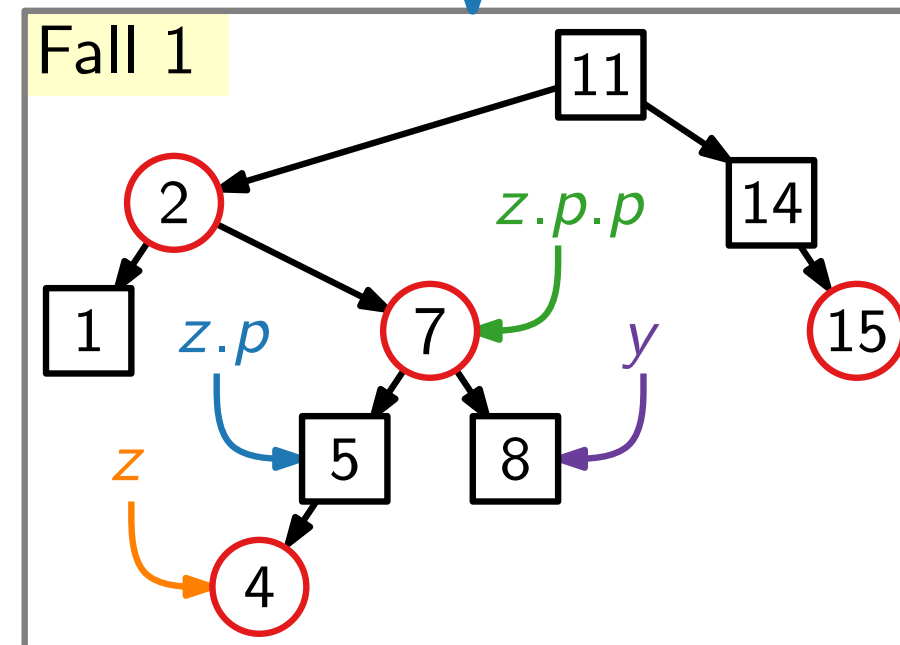
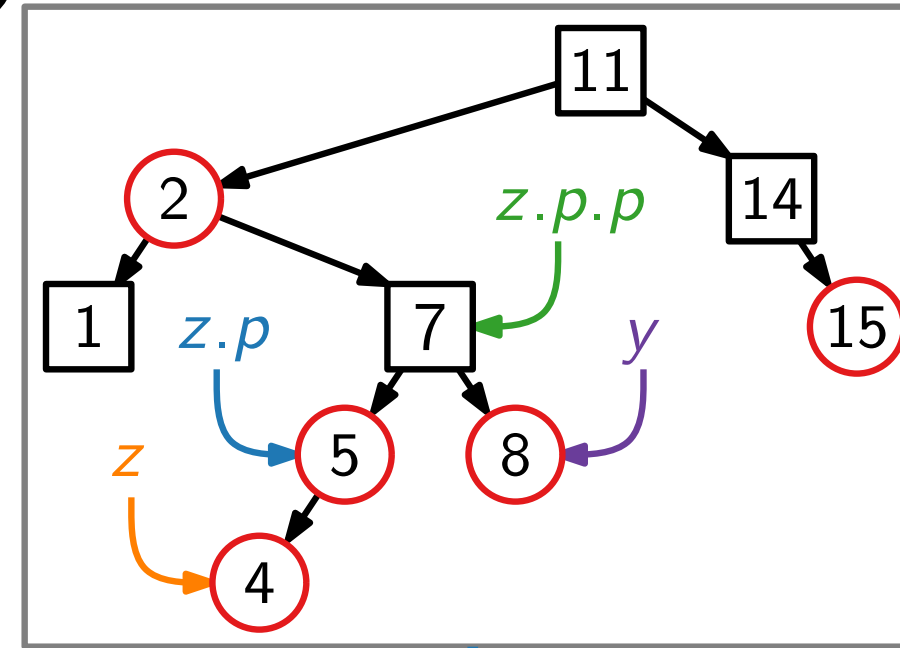
while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      // Case 1: Left-Right or Right-Left
      // Case 2: Left-Left or Right-Right
  else
    // Case 3: Left-Left or Right-Right
  
```



RBINSERTFIXUP(RBNode z)

```

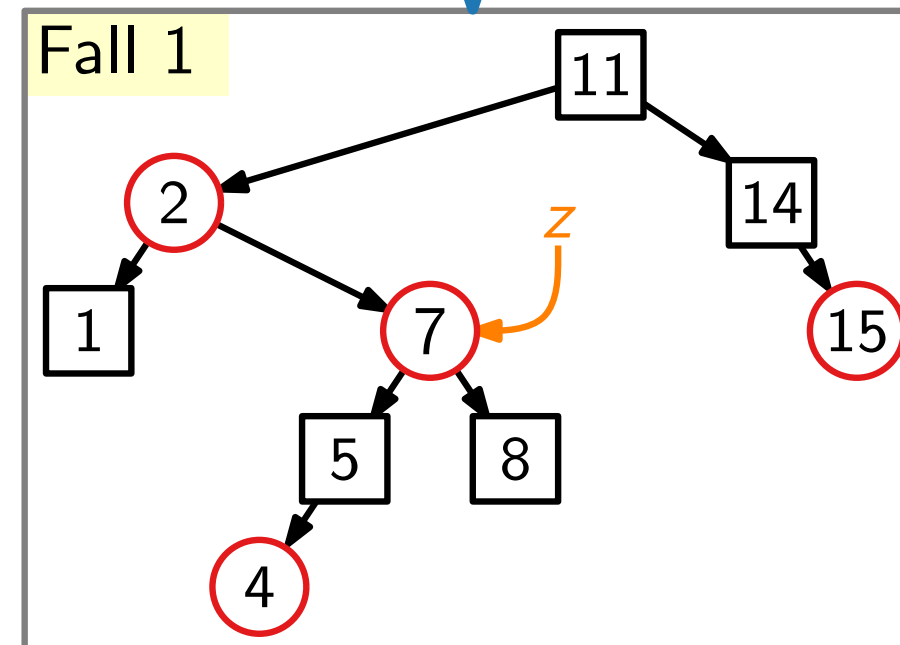
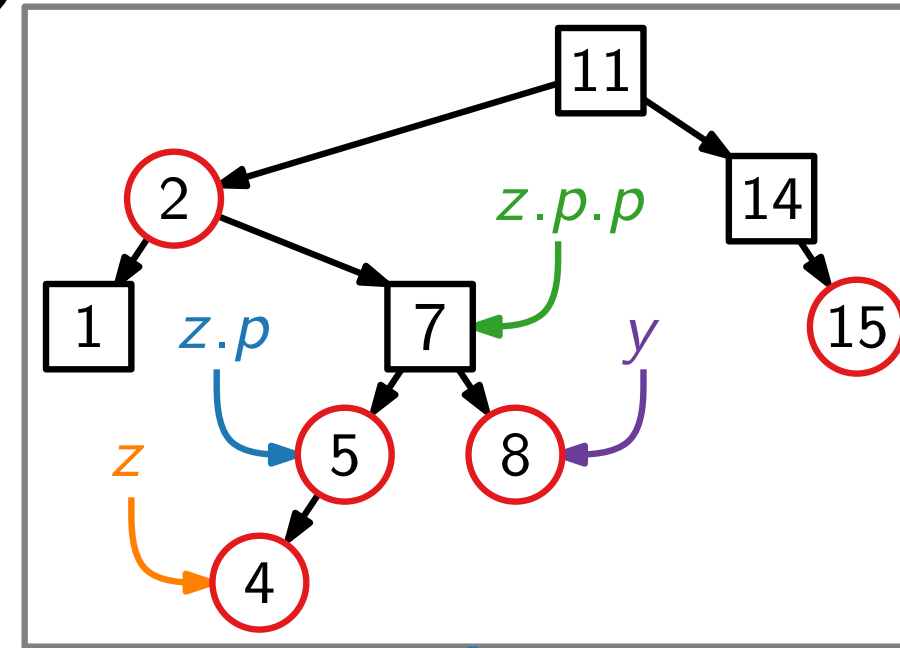
while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      // Case 1: Left-Right or Right-Left
      // Case 2: Left-Left or Right-Right
  else
    // Case 3: Left-Left or Right-Right
  
```



RBINSERTFIXUP(RBNode z)

```

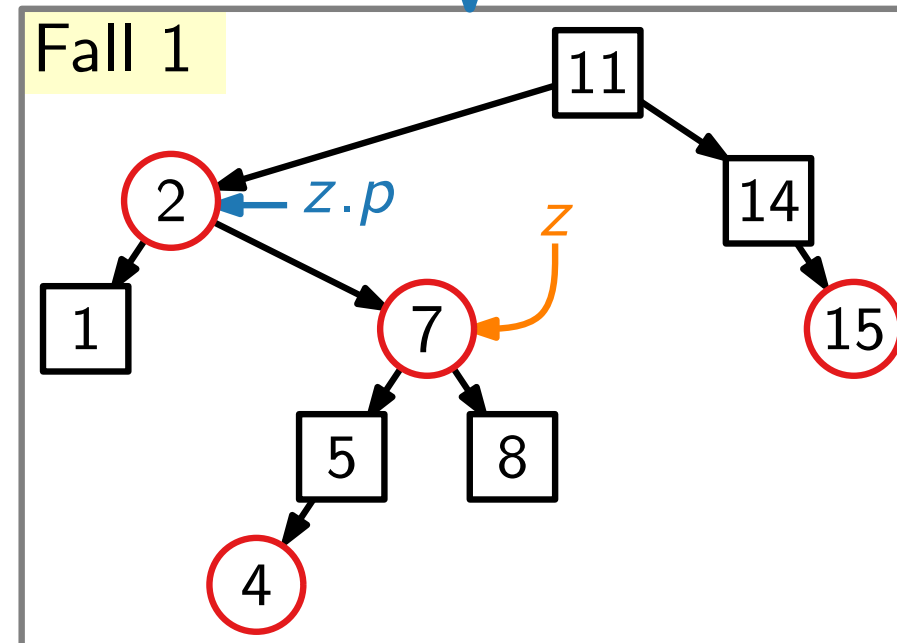
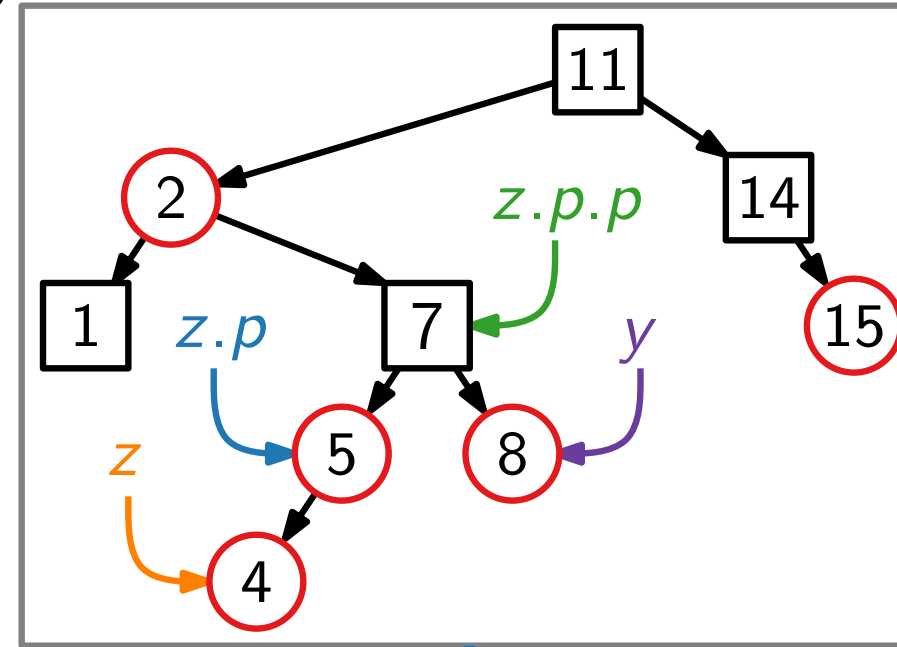
while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      // Case 1: Left-Right or Right-Left
      // Case 2: Left-Left or Right-Right
  else
    // Case 3: Left-Left or Right-Right
  
```



RBINSERTFIXUP(RBNode z)

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      // Case 1: Left-Right or Right-Left
      // Case 2: Left-Left or Right-Right
  else
    // Case 3: Left-Left or Right-Right
  
```

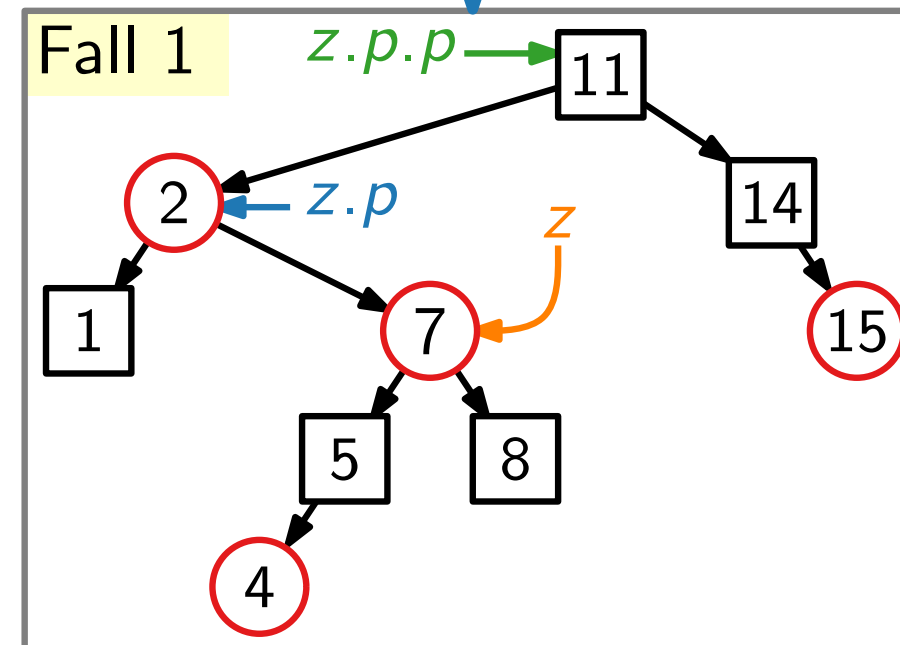
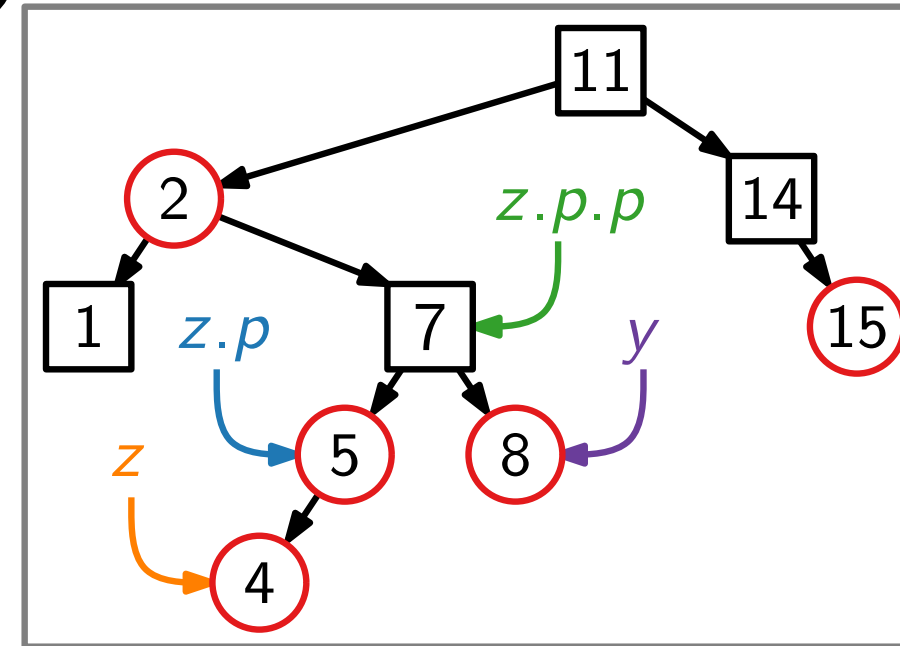


RBINSERTFIXUP(RBNode z)

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      // Left-Right or Right-Left case
      // ... (omitted) ...
    else
      // Left-Left or Right-Right case
      // ... (omitted) ...
  else
    // ... (omitted) ...
  end if
end while

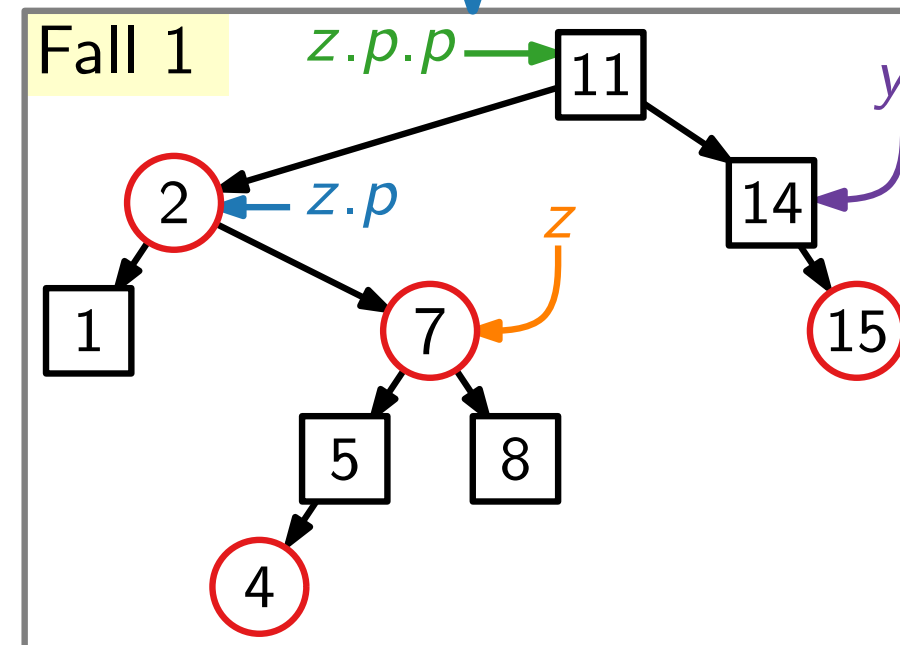
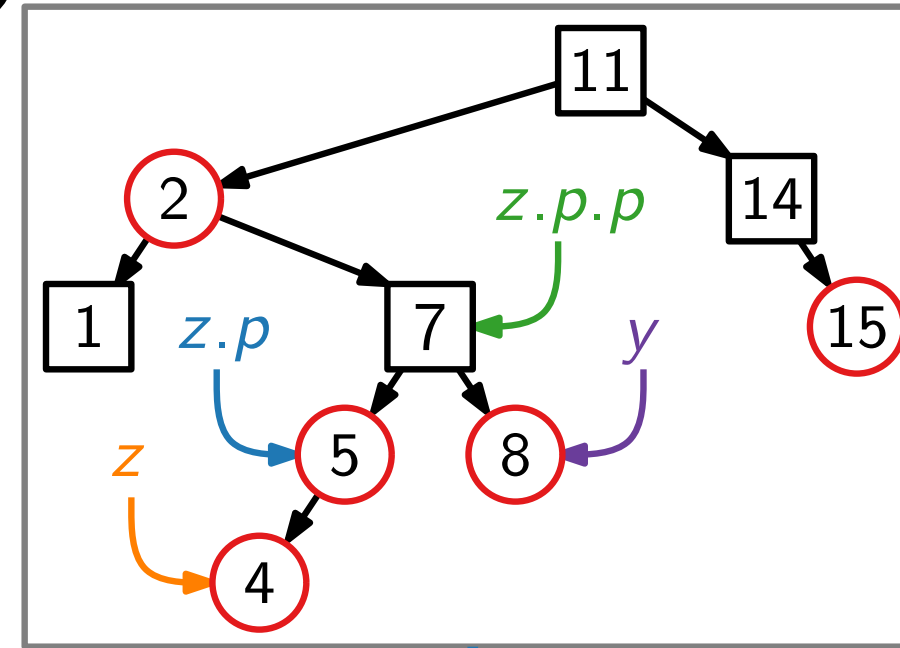
```



RBINSERTFIXUP(RBNode z)

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      // Case 1: Left-Right or Right-Left
      // Case 2: Left-Left or Right-Right
  else
    // Case 3: Left-Left or Right-Right
  
```

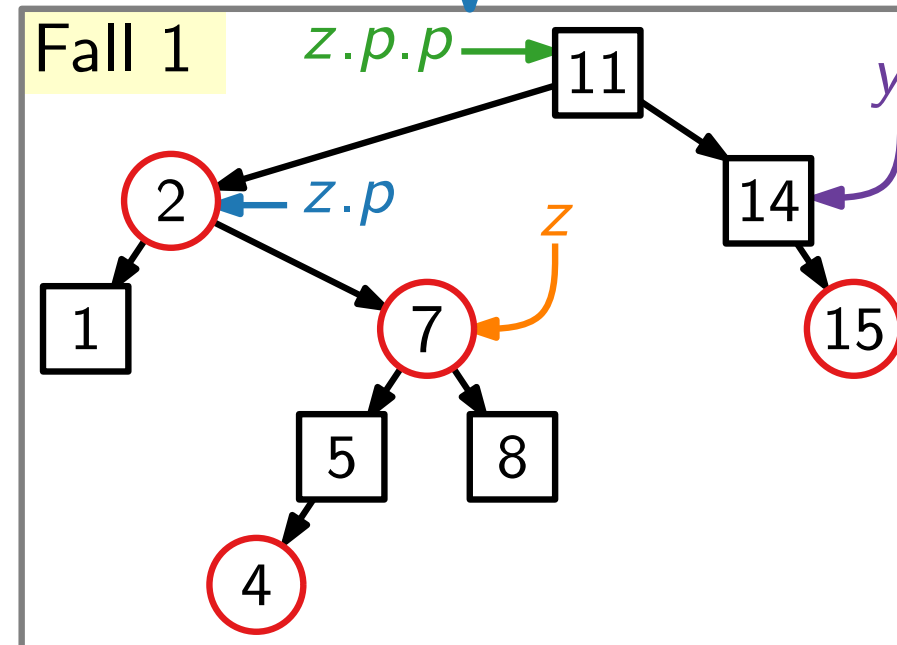
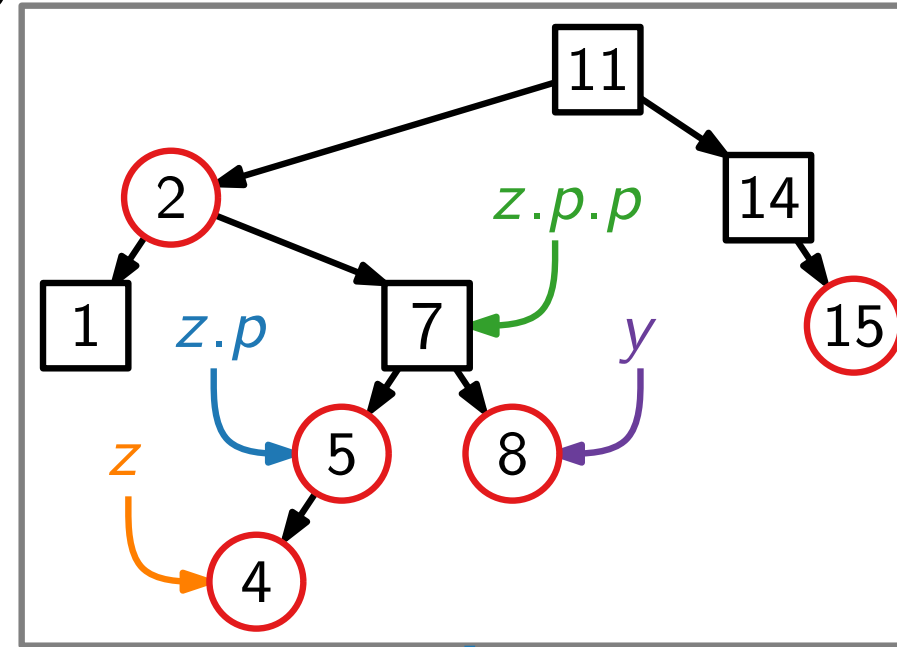


RBINSERTFIXUP(RBNode z)

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      if  $z == z.p.right$  then
        [Diagram showing a left-rotate operation around node 7]
      else
        [Diagram showing a right-rotate operation around node 2]
      end if
    end if
  else
    [Diagram showing a left-rotate operation around node 11]
  end if
end while

```

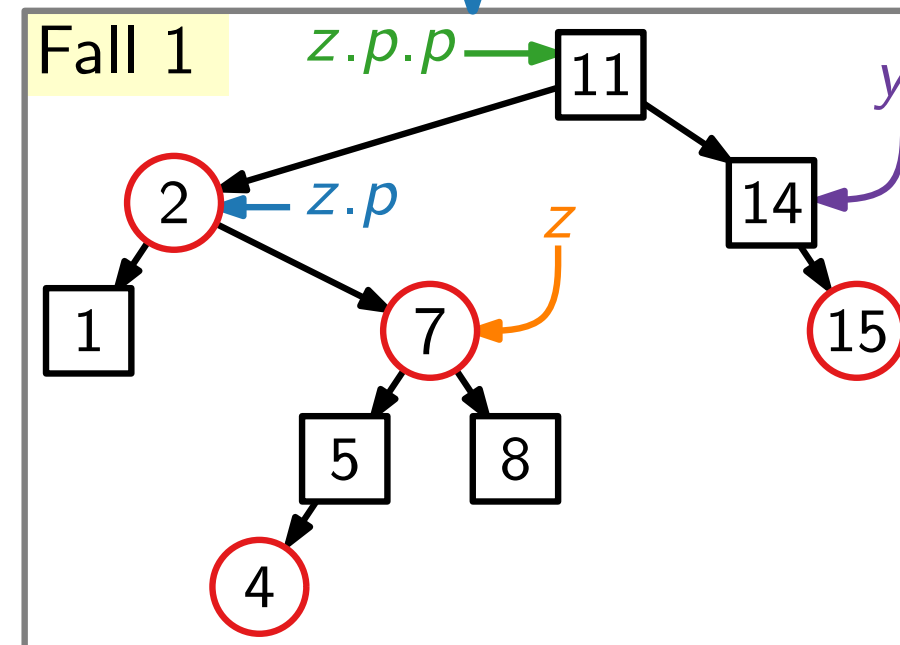
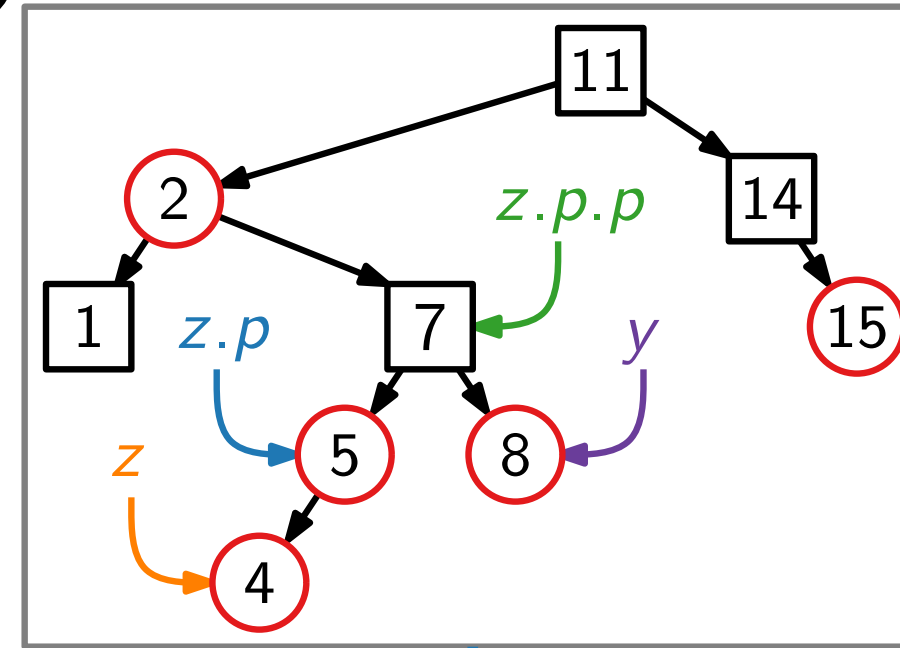


RBINSERTFIXUP(RBNode z)

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      if  $z == z.p.right$  then
         $z = z.p$ 
        LEFTROTATE( $z$ )
      else

```

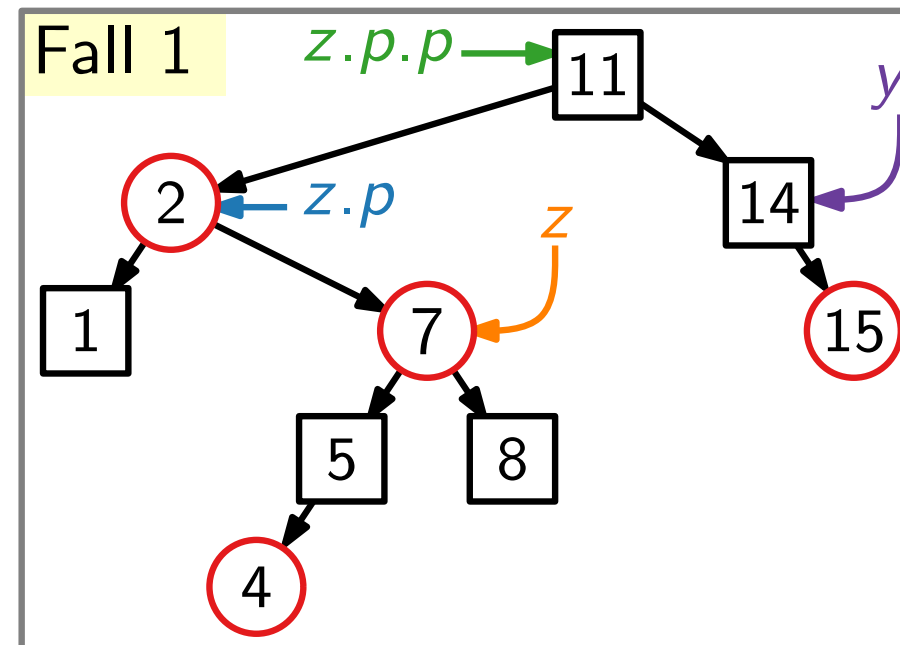


RBINSERTFIXUP(RBNode z)

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      if  $z == z.p.right$  then
         $z = z.p$ 
        LEFTROTATE( $z$ )
      else

```

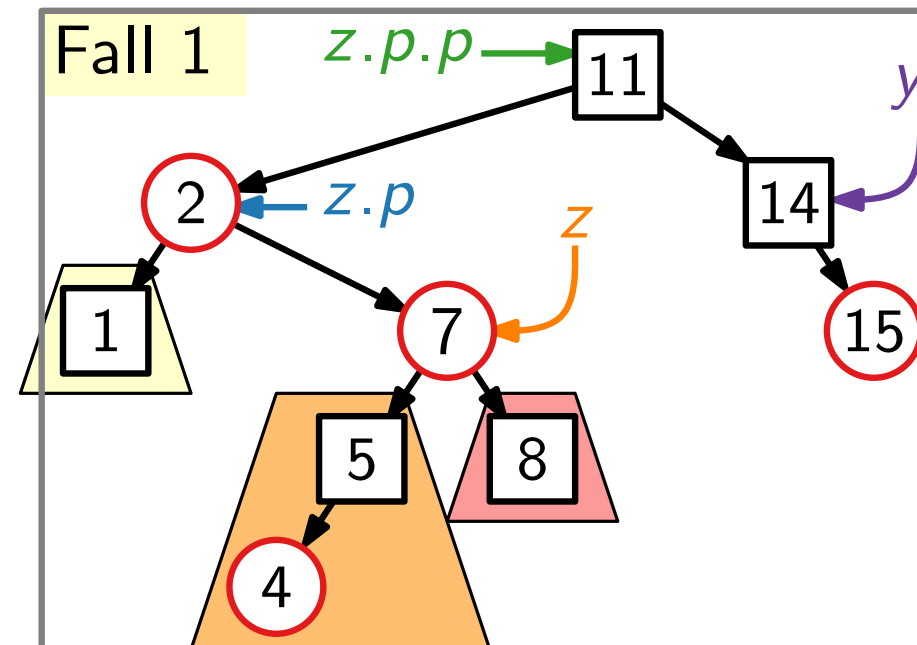


RBINSERTFIXUP(RBNode z)

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      if  $z == z.p.right$  then
         $z = z.p$ 
        LEFTROTATE( $z$ )
      else

```



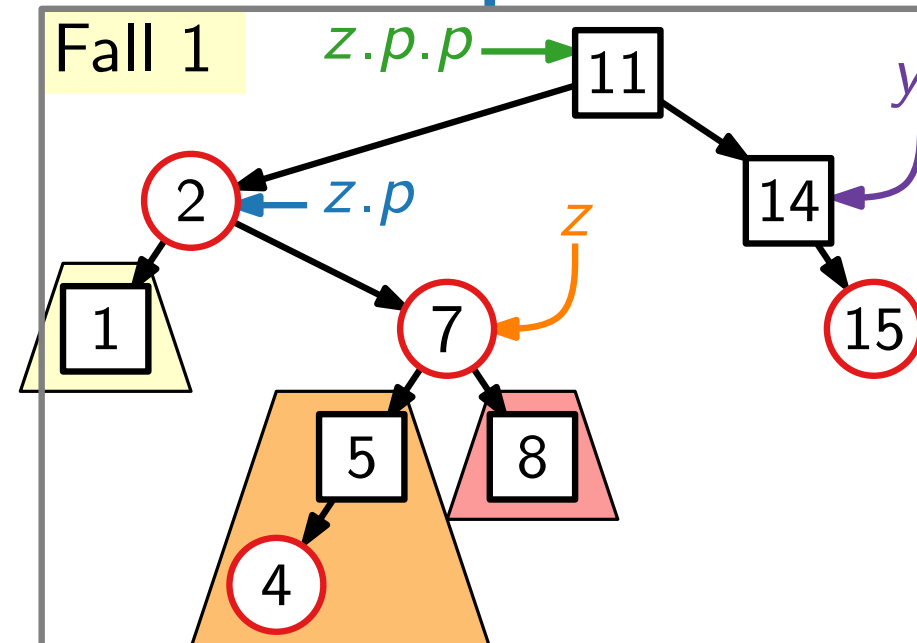
RBINSERTFIXUP(RBNode z)

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      if  $z == z.p.right$  then
         $z = z.p$ 
        LEFTROTATE( $z$ )
      else

```

Fall 2

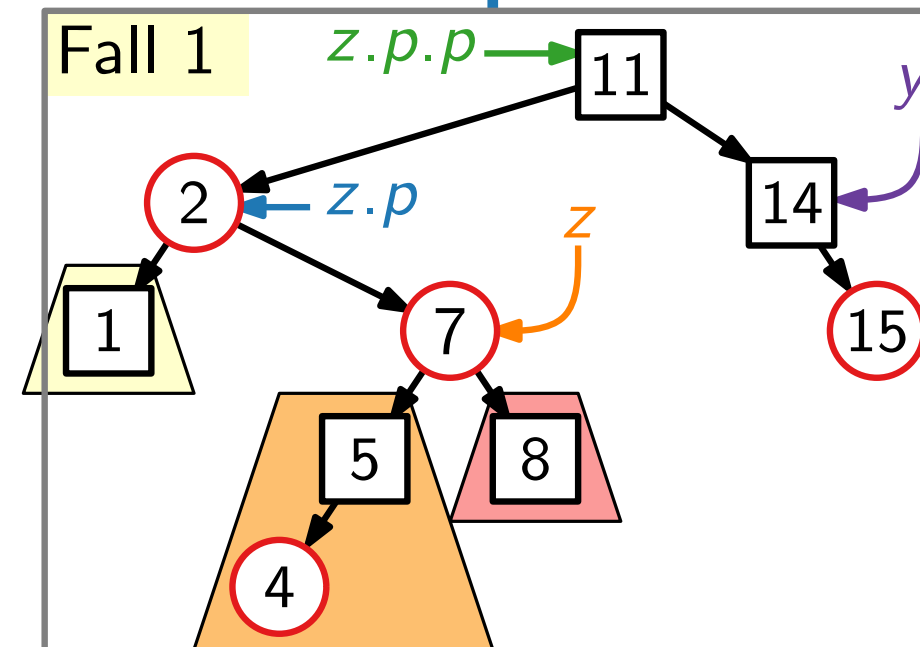
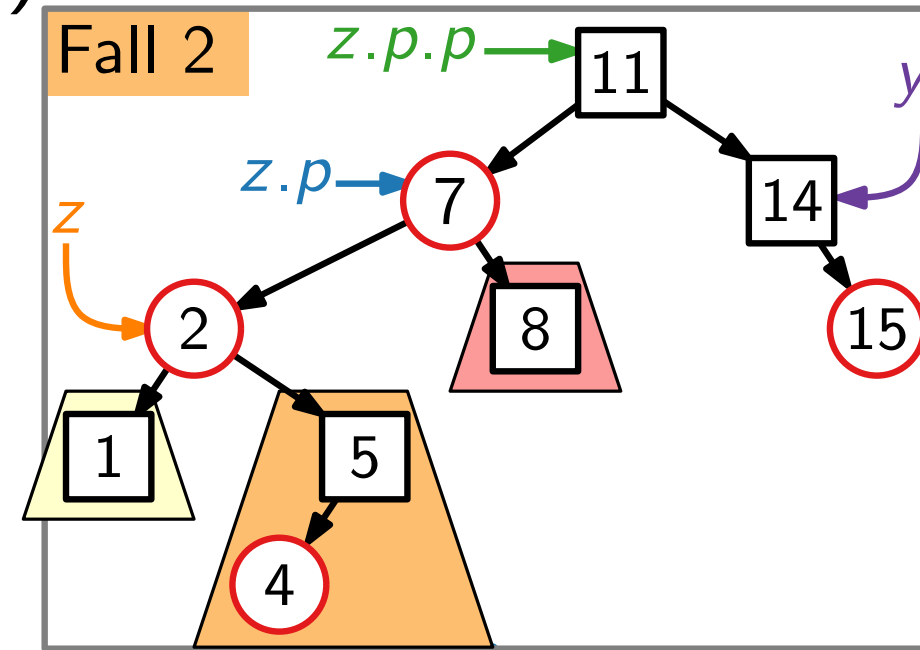


RBINSERTFIXUP(RBNode z)

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      if  $z == z.p.right$  then
         $z = z.p$ 
        LEFTROTATE( $z$ )
      else

```



The diagram illustrates the insertion of node 8 into a B-tree structure. The tree has three levels: a root node (11), an internal node (7), and leaf nodes (1, 2, 4, 5, 8, 14, 15). The root node (11) is a square. The internal node (7) is a circle. The leaf nodes (1, 2, 4, 5, 8, 14, 15) are squares. The leaf nodes are grouped into three sets: {1, 2}, {4, 5}, and {8, 14, 15}. The leaf nodes are represented by trapezoids. The root node (11) is a square. The internal node (7) is a circle. The leaf nodes (1, 2, 4, 5, 8, 14, 15) are squares. The leaf nodes are grouped into three sets: {1, 2}, {4, 5}, and {8, 14, 15}. The leaf nodes are represented by trapezoids. The root node (11) is a square. The internal node (7) is a circle. The leaf nodes (1, 2, 4, 5, 8, 14, 15) are squares. The leaf nodes are grouped into three sets: {1, 2}, {4, 5}, and {8, 14, 15}. The leaf nodes are represented by trapezoids.

Fall 2 (Top): Node 8 is merged into node 7. The merge is indicated by a green arrow labeled $z.p.p$ pointing to node 11, a blue arrow labeled $z.p$ pointing to node 7, and an orange arrow labeled z pointing to node 2. Node 8 is shown in a red trapezoid, and node 7 is in a red circle.

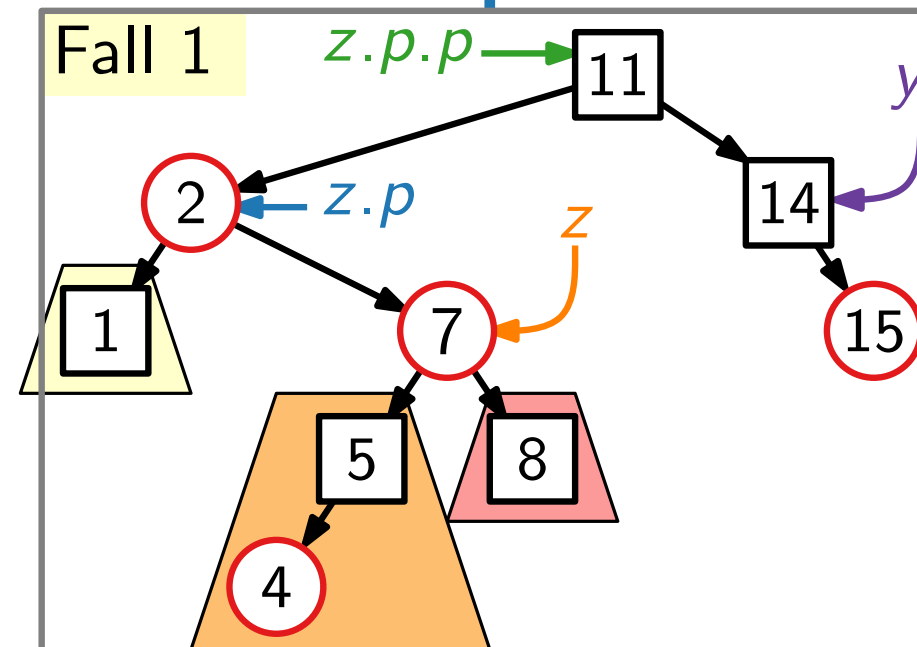
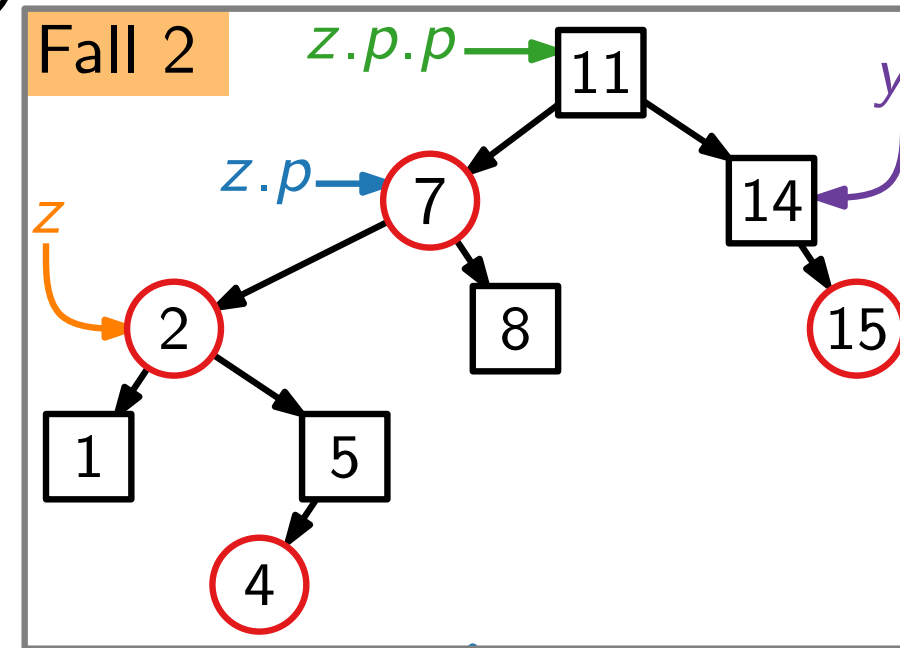
Fall 1 (Bottom): Node 8 is merged into node 2. The merge is indicated by a green arrow labeled $z.p.p$ pointing to node 11, a blue arrow labeled $z.p$ pointing to node 2, and an orange arrow labeled z pointing to node 7. Node 8 is shown in a red trapezoid, and node 2 is in a red circle.

RBINSERTFIXUP(RBNode z)

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      if  $z == z.p.right$  then
         $z = z.p$ 
        LEFTROTATE( $z$ )
       $z.p.color = black$ 
       $z.p.p.color = red$ 
      RIGHTROTATE( $z.p.p$ )
  else

```

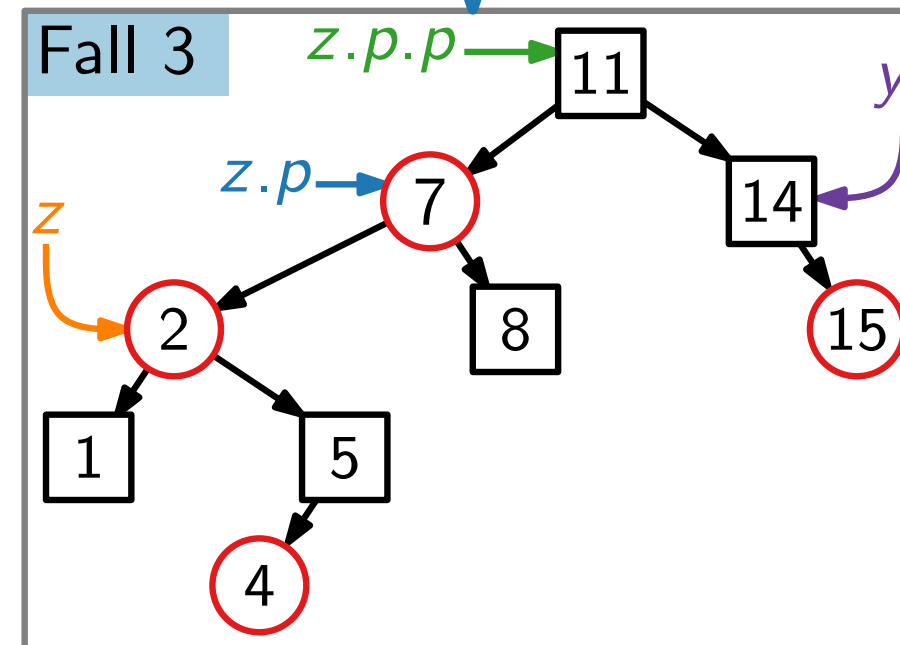
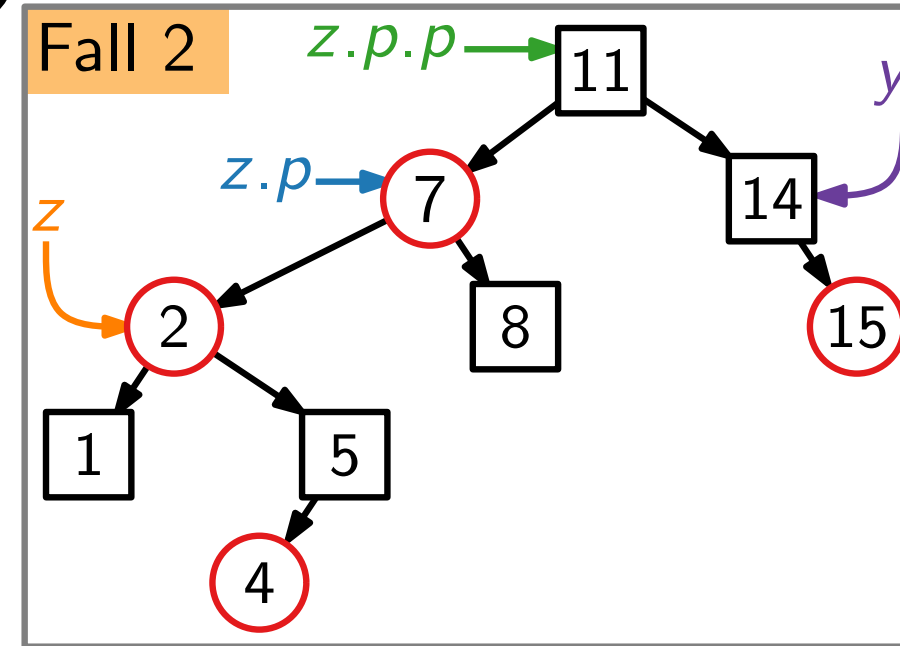


RBINSERTFIXUP(RBNode z)

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      if  $z == z.p.right$  then
         $z = z.p$ 
        LEFTROTATE( $z$ )
       $z.p.color = black$ 
       $z.p.p.color = red$ 
      RIGHTROTATE( $z.p.p$ )
  else

```

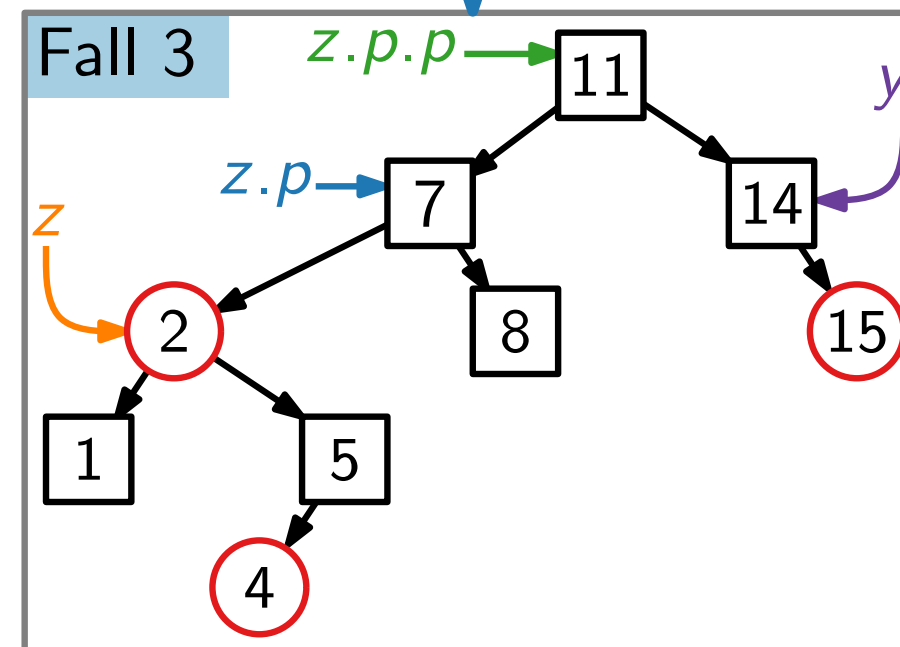
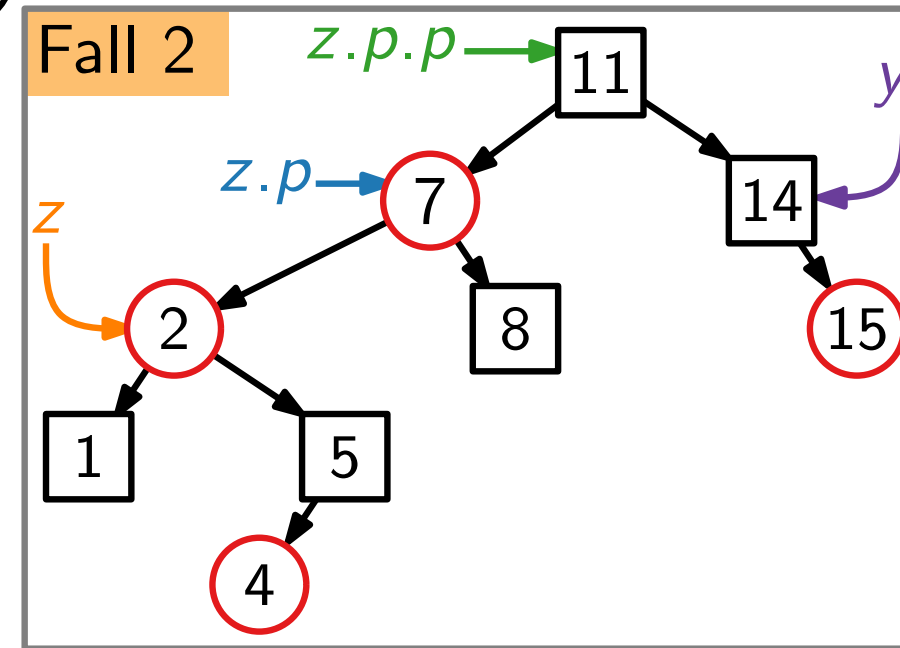


RBINSERTFIXUP(RBNode z)

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      if  $z == z.p.right$  then
         $z = z.p$ 
        LEFTROTATE( $z$ )
       $z.p.color = black$ 
       $z.p.p.color = red$ 
      RIGHTROTATE( $z.p.p$ )
  else

```

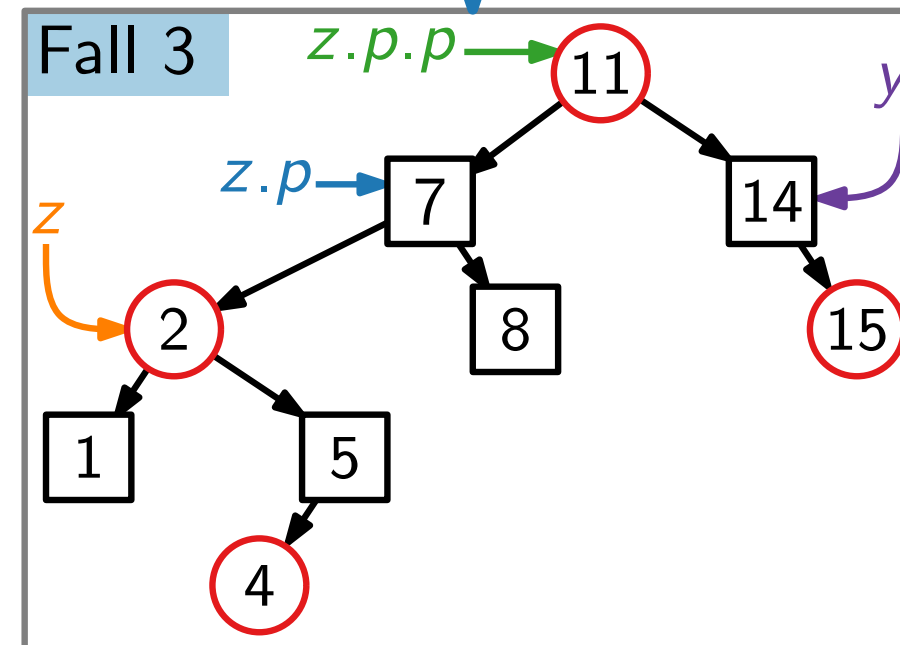
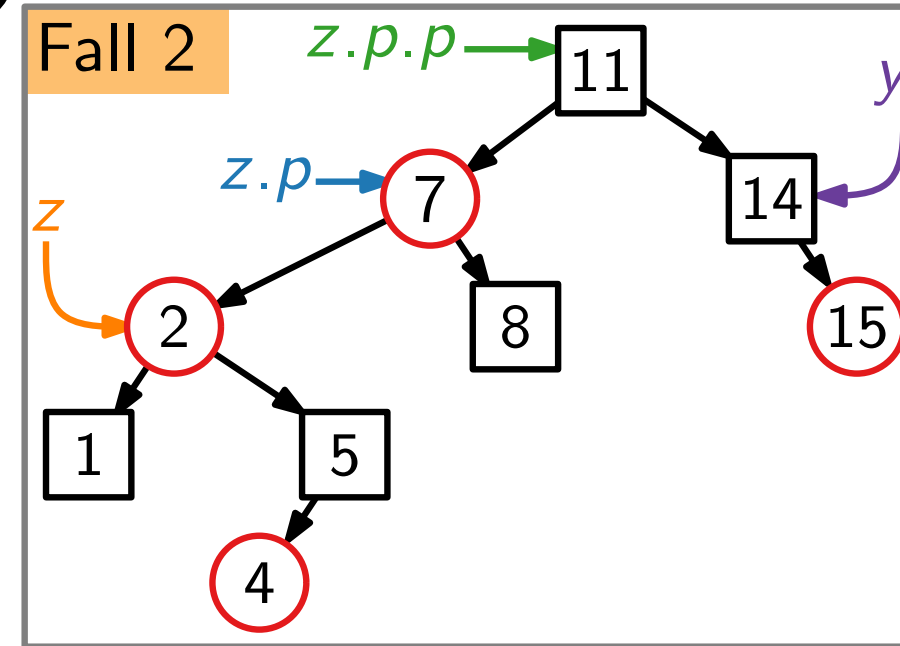


RBINSERTFIXUP(RBNode z)

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      if  $z == z.p.right$  then
         $z = z.p$ 
        LEFTROTATE( $z$ )
       $z.p.color = black$ 
       $z.p.p.color = red$ 
      RIGHTROTATE( $z.p.p$ )
  else

```

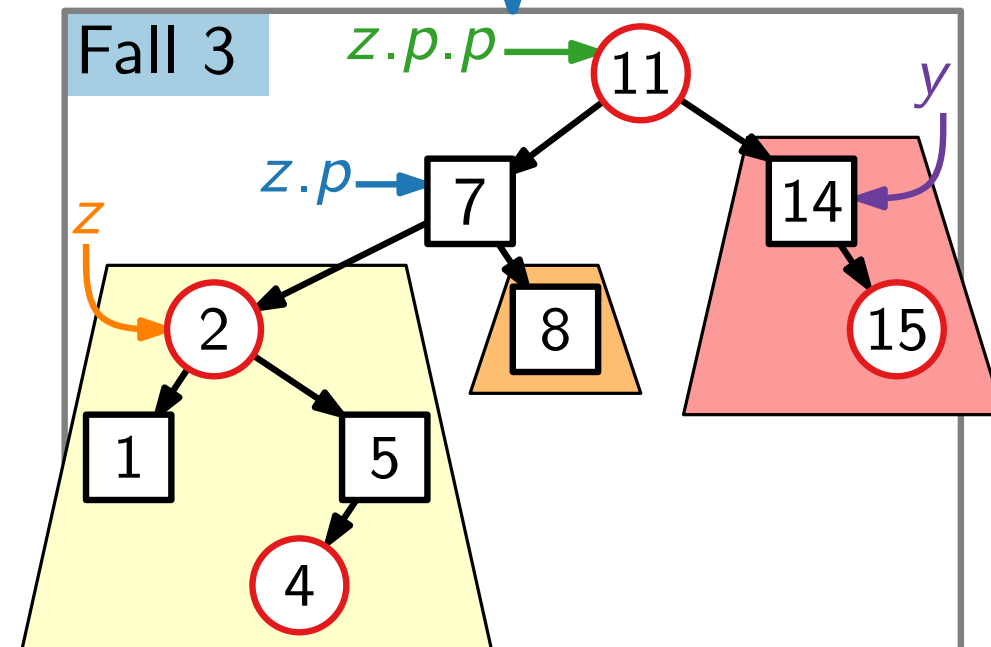
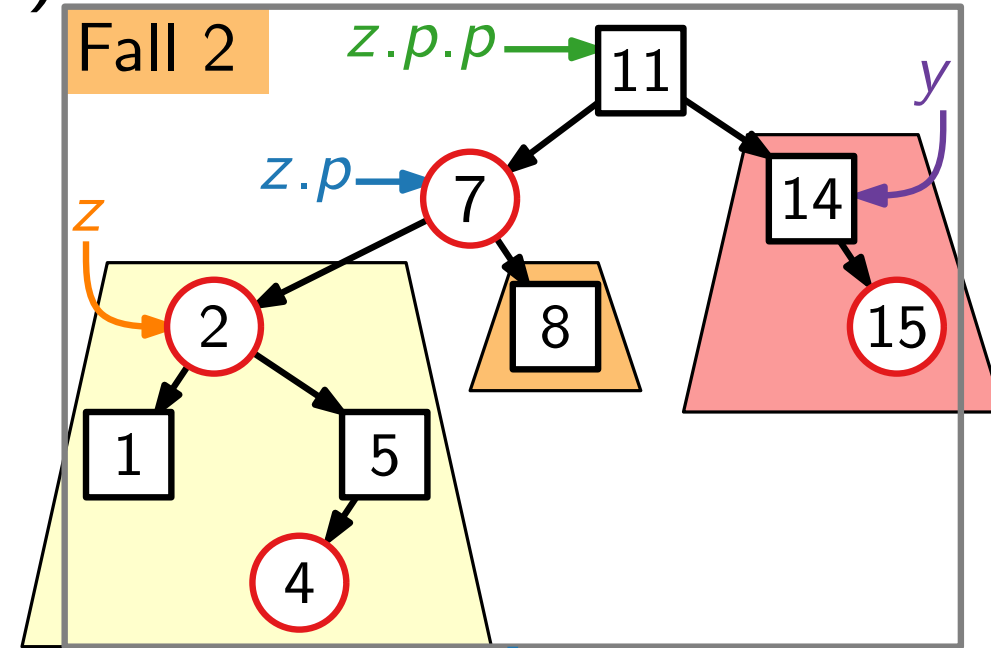


RBINSERTFIXUP(RBNode z)

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      if  $z == z.p.right$  then
         $z = z.p$ 
        LEFTROTATE( $z$ )
       $z.p.color = black$ 
       $z.p.p.color = red$ 
      RIGHTROTATE( $z.p.p$ )
  else

```

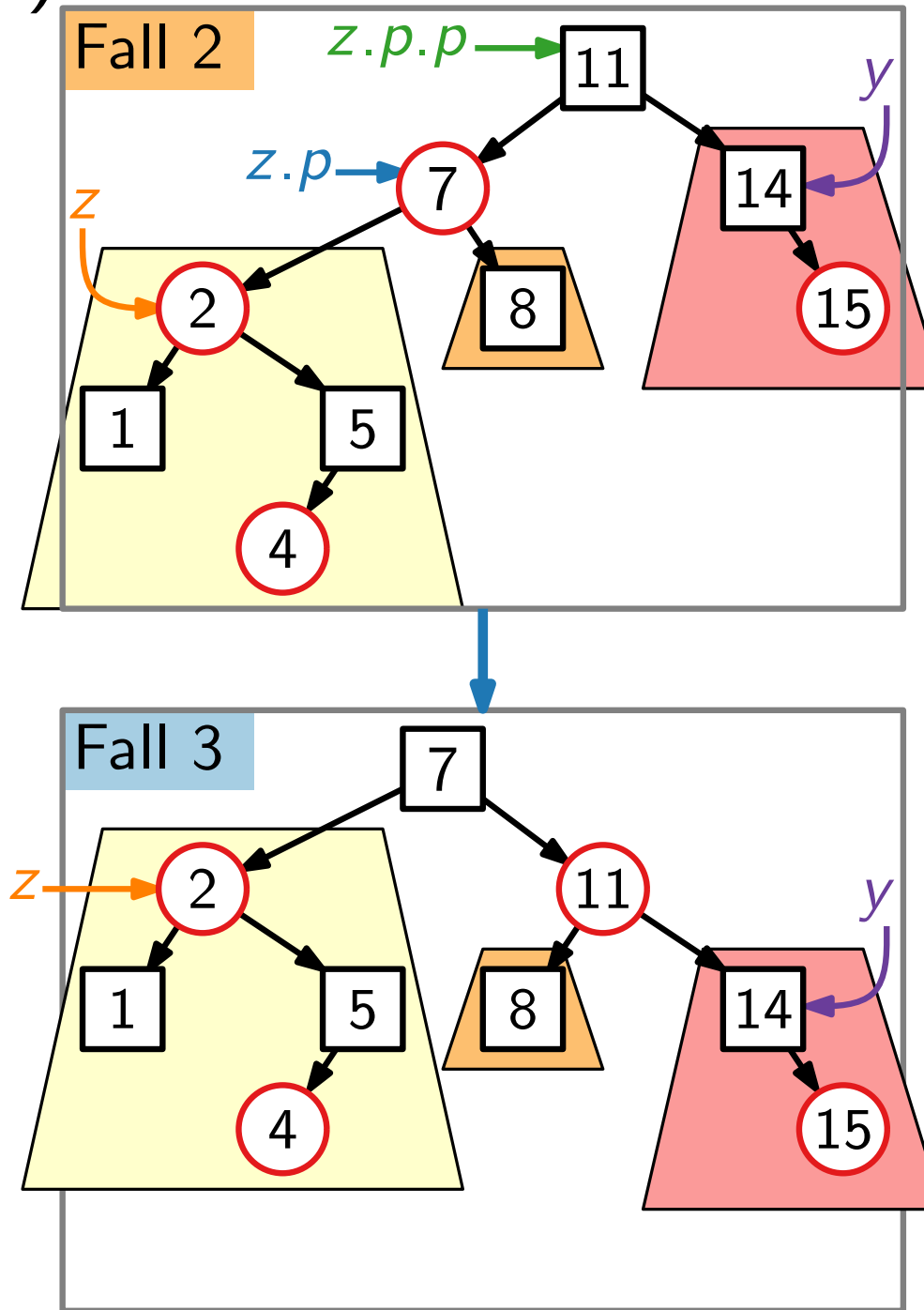


RBINSERTFIXUP(RBNode z)

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      if  $z == z.p.right$  then
         $z = z.p$ 
        LEFTROTATE( $z$ )
       $z.p.color = black$ 
       $z.p.p.color = red$ 
      RIGHTROTATE( $z.p.p$ )
  else

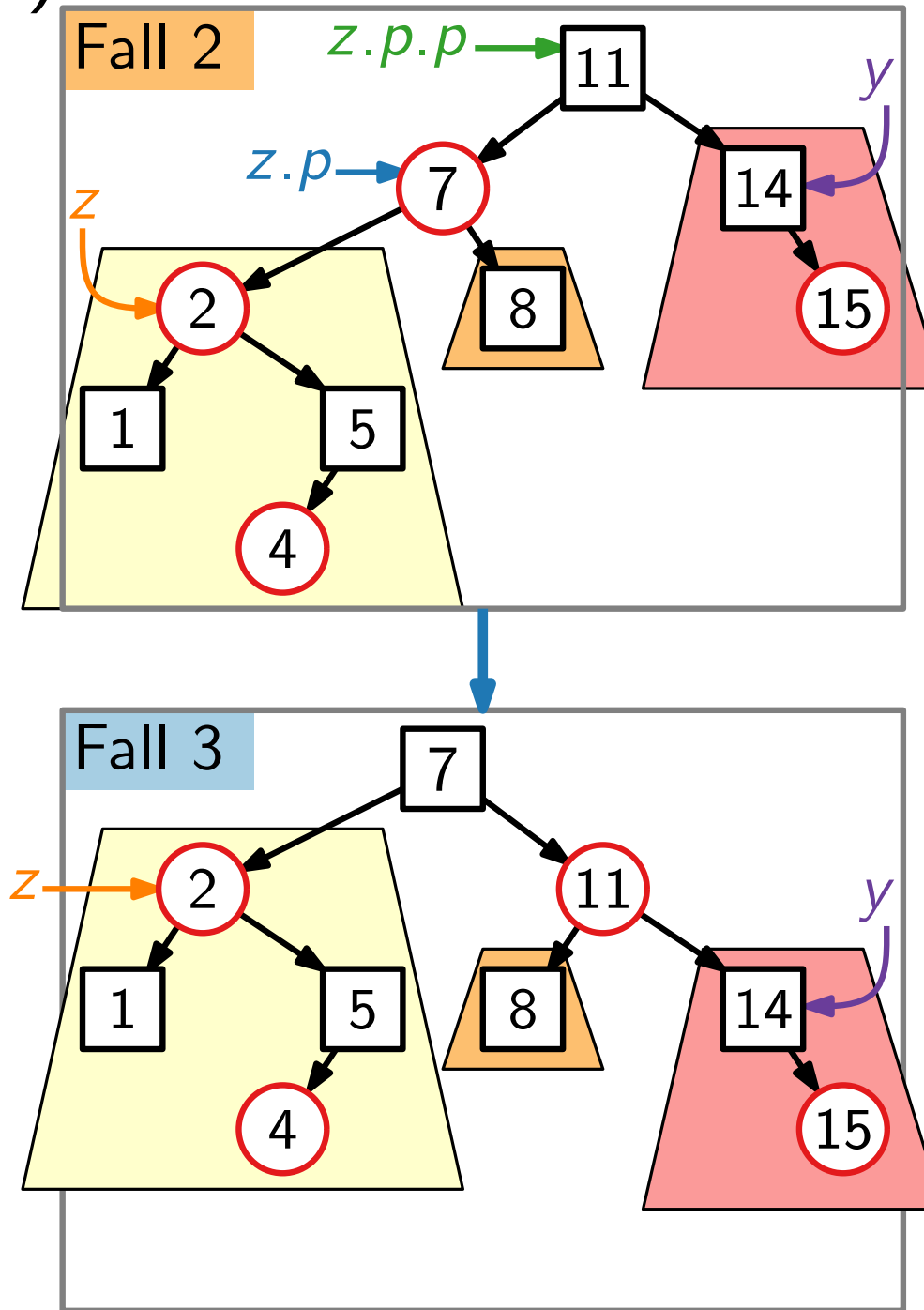
```



RBINSERTFIXUP(RBNode z)

```

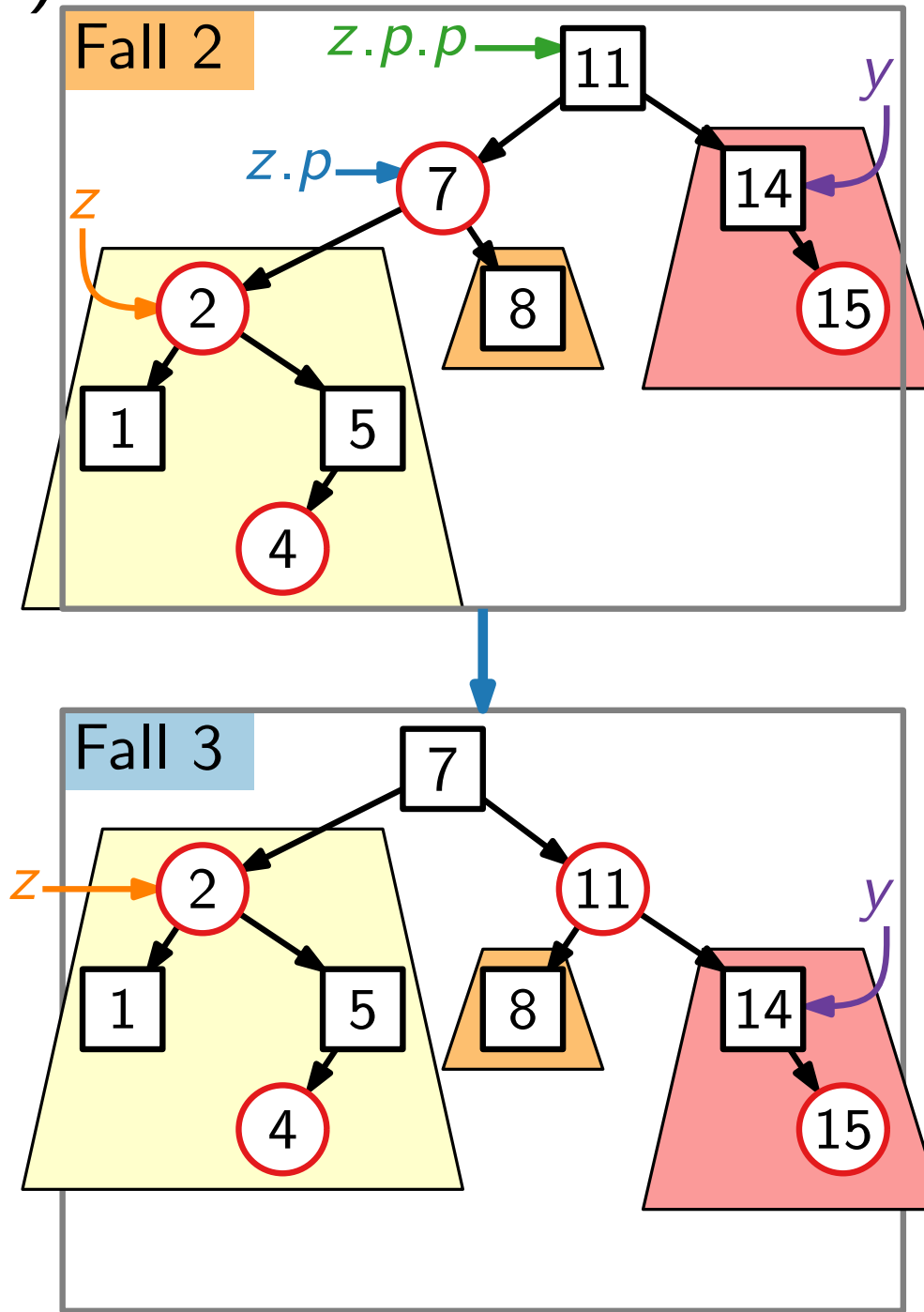
while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      if  $z == z.p.right$  then
         $z = z.p$ 
        LEFTROTATE( $z$ )
       $z.p.color = black$ 
       $z.p.p.color = red$ 
      RIGHTROTATE( $z.p.p$ )
  else ... // wie oben, aber re. & li. vertauscht
  
```



RBINSERTFIXUP(RBNode *z*)

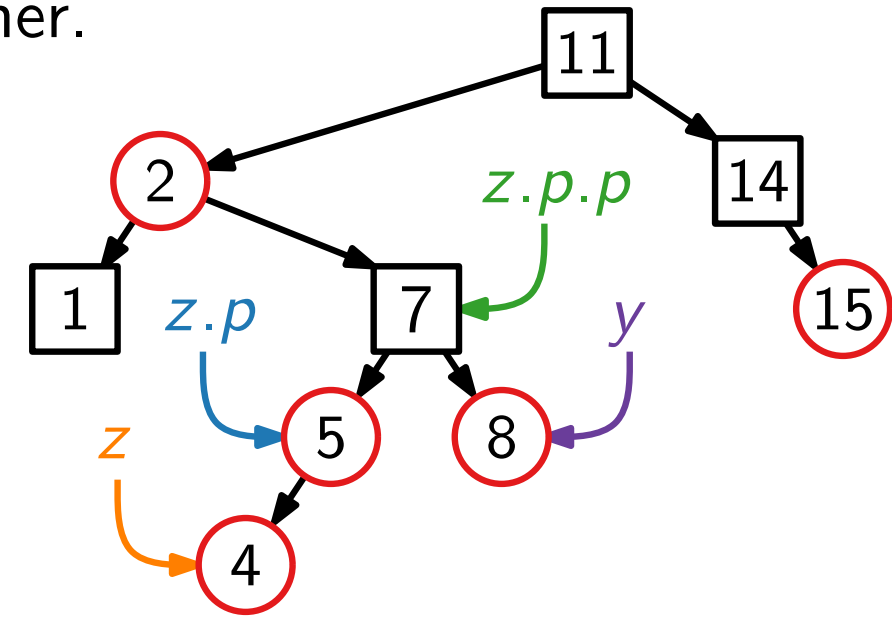
```

while z.p.color == red do
  if z.p == z.p.p.left then
    y = z.p.p.right // Tante von z
    if y.color == red then
      z.p.color = black
      z.p.p.color = red
      y.color = black
      z = z.p.p
    else
      if z == z.p.right then
        z = z.p
        LEFTROTATE(z)
      z.p.color = black
      z.p.p.color = red
      RIGHTROTATE(z.p.p)
    else ... // wie oben, aber re. & li. vertauscht
  root.color = black
  
```



Korrektheit

Zu zeigen: RBINSERTFIXUP stellt R-S-Eigenschaft wieder her.

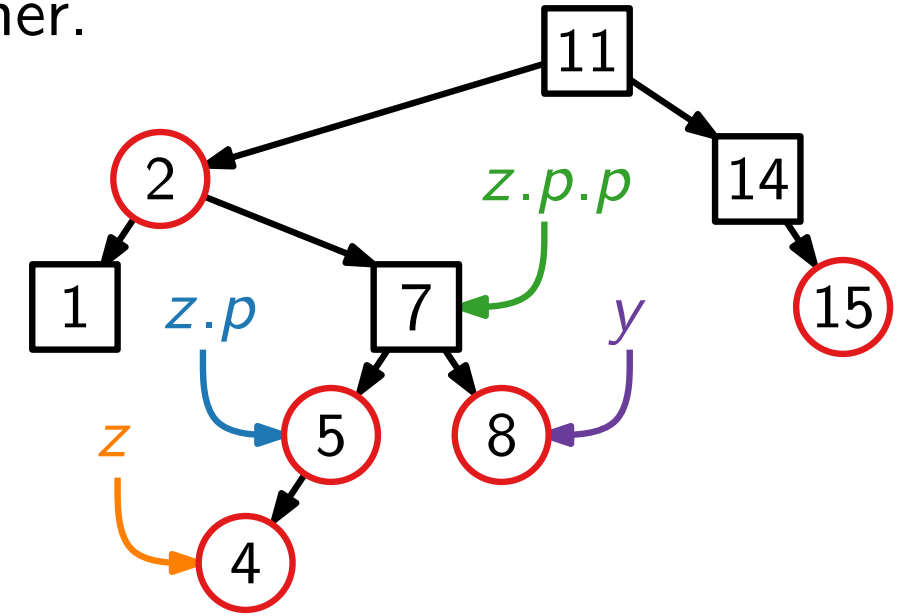


Korrektheit

Zu zeigen: RBINSERTFIXUP stellt R-S-Eigenschaft wieder her.

Schleifeninvariante (gültig am Anfang der while-Schleife)

■ *z* ist rot.

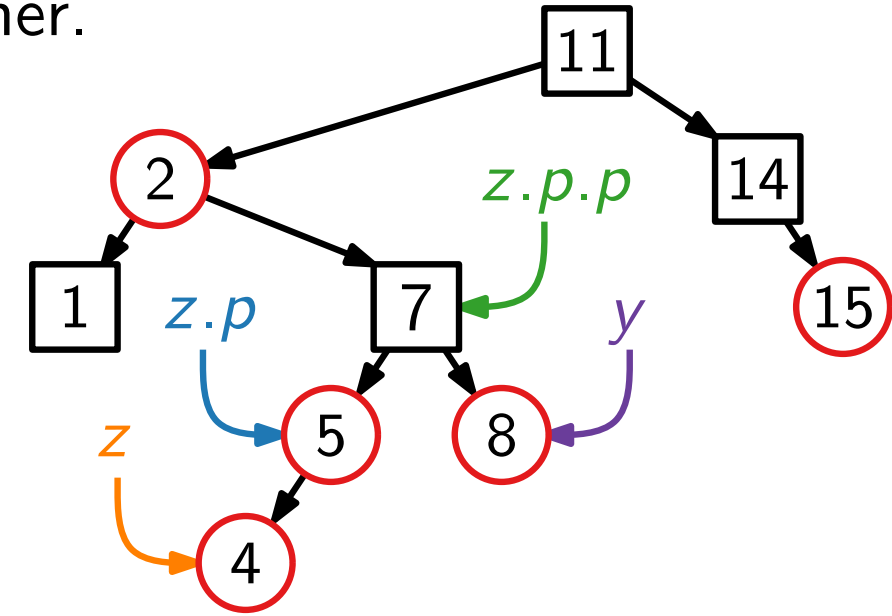


Korrektheit

Zu zeigen: RBINSERTFIXUP stellt R-S-Eigenschaft wieder her.

Schleifeninvariante (gültig am Anfang der while-Schleife)

- z ist rot.
- Falls $z.p$ die Wurzel ist, dann ist $z.p$ schwarz.

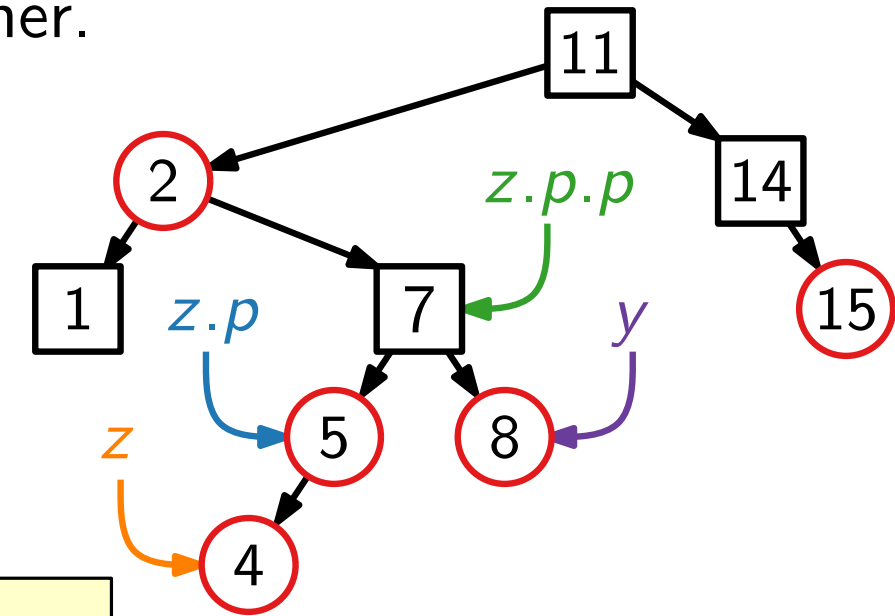


Korrektheit

Zu zeigen: RBINSERTFIXUP stellt R-S-Eigenschaft wieder her.

Schleifeninvariante (gültig am Anfang der while-Schleife)

- z ist rot.
- Falls $z.p$ die Wurzel ist, dann ist $z.p$ schwarz.



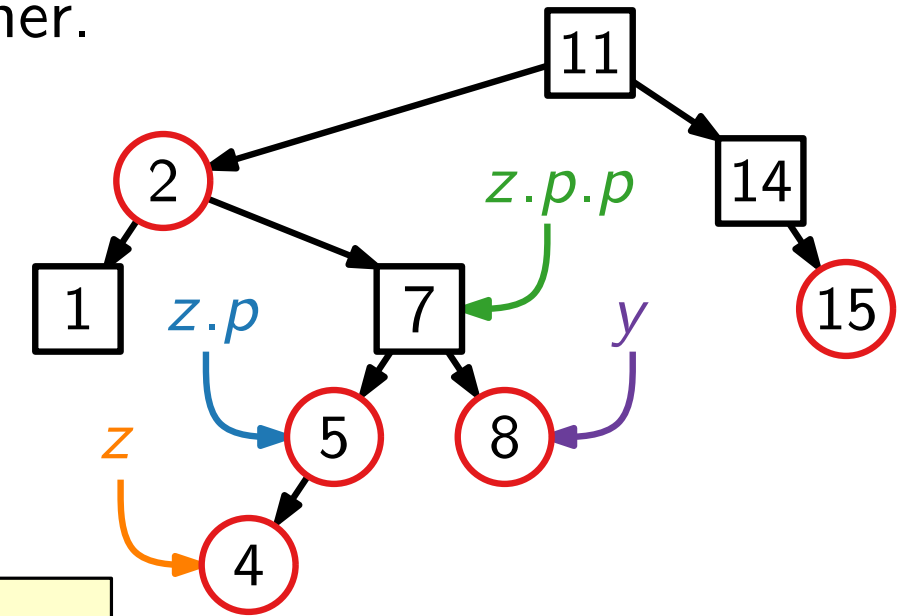
- (E1) Jeder Knoten ist entweder rot oder schwarz.
- (E2) Die Wurzel ist schwarz.
- (E3) Alle Blätter sind schwarz. nil
- (E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten
dieselbe Anzahl schwarzer Knoten.

Korrektheit

Zu zeigen: RBINSERTFIXUP stellt R-S-Eigenschaft wieder her.

Schleifeninvariante (gültig am Anfang der while-Schleife)

- z ist rot.
- Falls $z.p$ die Wurzel ist, dann ist $z.p$ schwarz.
- Falls R-S-Eig. verletzt sind, dann



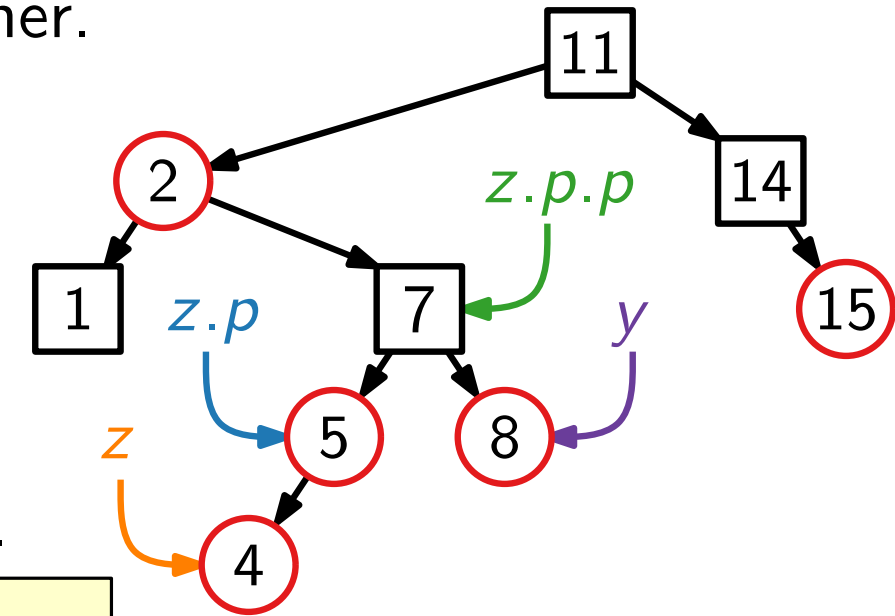
- (E1) Jeder Knoten ist entweder rot oder schwarz.
- (E2) Die Wurzel ist schwarz.
- (E3) Alle Blätter sind schwarz. nil
- (E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten
dieselbe Anzahl schwarzer Knoten.

Korrektheit

Zu zeigen: RBINSERTFIXUP stellt R-S-Eigenschaft wieder her.

Schleifeninvariante (gültig am Anfang der while-Schleife)

- z ist rot.
- Falls $z.p$ die Wurzel ist, dann ist $z.p$ schwarz.
- Falls R-S-Eig. verletzt sind, dann entweder (E2) oder (E4).



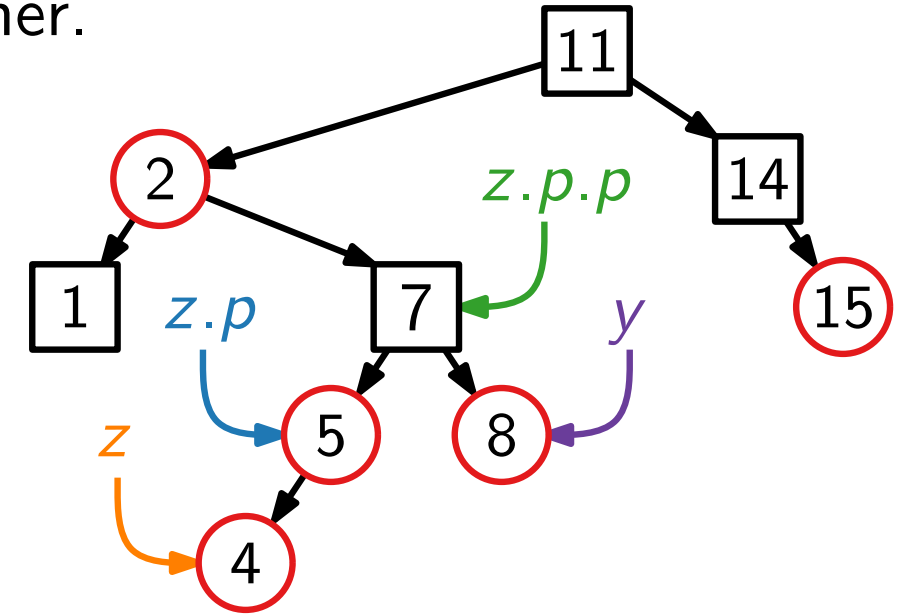
- (E1) Jeder Knoten ist entweder rot oder schwarz.
- (E2) Die Wurzel ist schwarz.
- (E3) Alle Blätter sind schwarz. nil
- (E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten
dieselbe Anzahl schwarzer Knoten.

Korrektheit

Zu zeigen: RBINSERTFIXUP stellt R-S-Eigenschaft wieder her.

Schleifeninvariante (gültig am Anfang der while-Schleife)

- z ist rot.
- Falls $z.p$ die Wurzel ist, dann ist $z.p$ schwarz.
- Falls R-S-Eig. verletzt sind, dann entweder **(E2)** oder **(E4)**.



(E2) Die Wurzel ist schwarz.

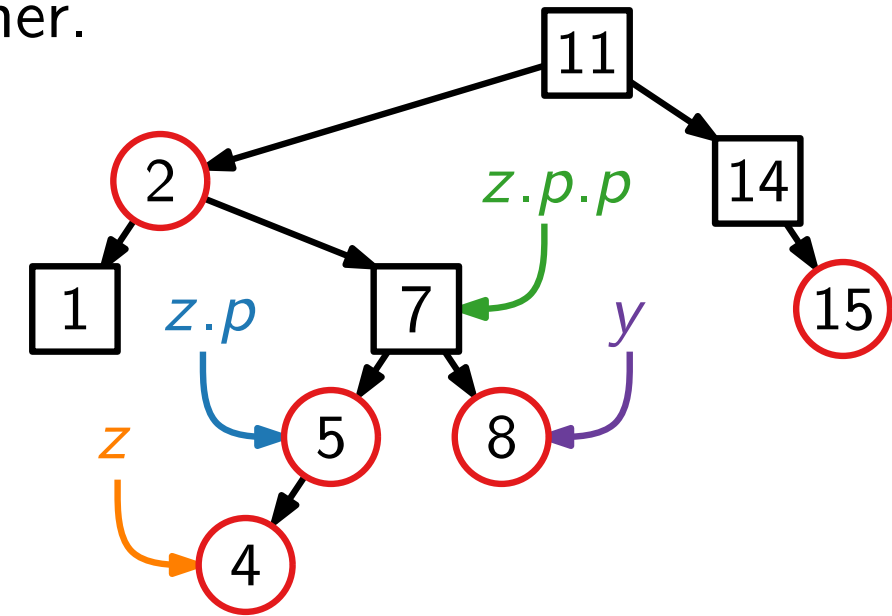
(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.

Korrektheit

Zu zeigen: RBINSERTFIXUP stellt R-S-Eigenschaft wieder her.

Schleifeninvariante (gültig am Anfang der while-Schleife)

- z ist rot.
- Falls $z.p$ die Wurzel ist, dann ist $z.p$ schwarz.
- Falls R-S-Eig. verletzt sind, dann entweder **(E2)** oder **(E4)**.
 - Falls **(E2)** verletzt ist, dann weil $z = \text{root}$ und z rot ist.



(E2) Die Wurzel ist schwarz.

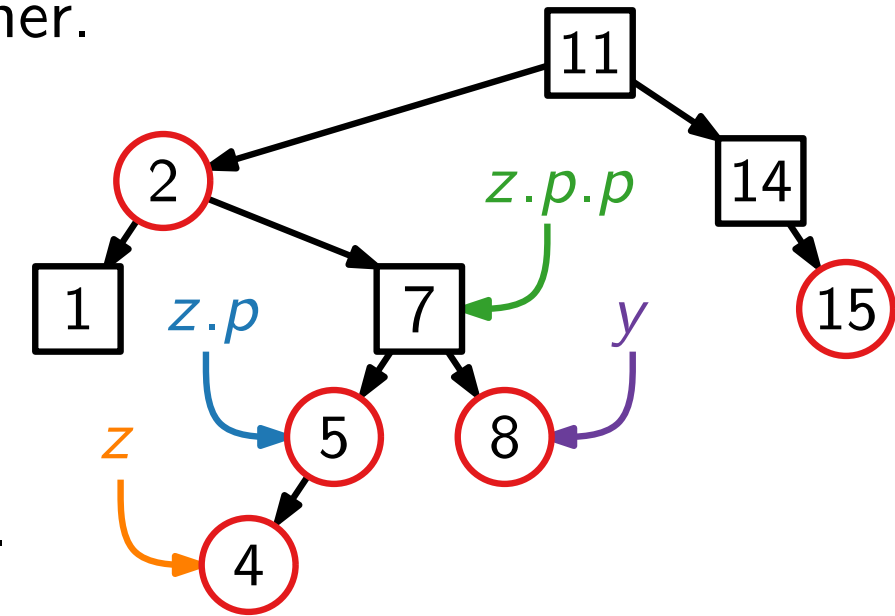
(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.

Korrektheit

Zu zeigen: RBINSERTFIXUP stellt R-S-Eigenschaft wieder her.

Schleifeninvariante (gültig am Anfang der while-Schleife)

- z ist rot.
- Falls $z.p$ die Wurzel ist, dann ist $z.p$ schwarz.
- Falls R-S-Eig. verletzt sind, dann entweder **(E2)** oder **(E4)**.
 - Falls **(E2)** verletzt ist, dann weil $z = root$ und z rot ist.
 - Falls **(E4)** verletzt ist, dann weil z und $z.p$ rot sind.



(E2) Die Wurzel ist schwarz.

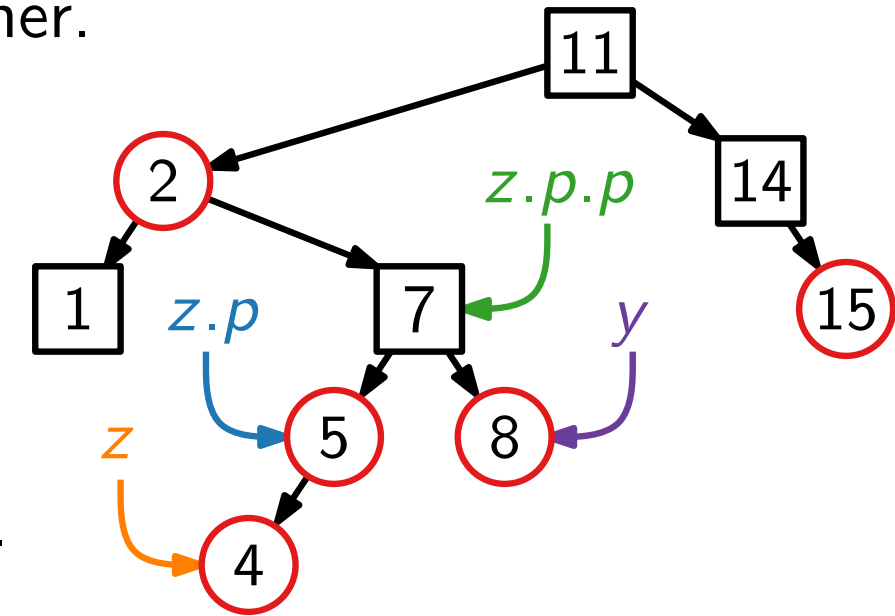
(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.

Korrektheit

Zu zeigen: RBINSERTFIXUP stellt R-S-Eigenschaft wieder her.

Schleifeninvariante (gültig am Anfang der while-Schleife)

- z ist rot.
- Falls $z.p$ die Wurzel ist, dann ist $z.p$ schwarz.
- Falls R-S-Eig. verletzt sind, dann entweder **(E2)** oder **(E4)**.
 - Falls **(E2)** verletzt ist, dann weil $z = root$ und z rot ist.
 - Falls **(E4)** verletzt ist, dann weil z und $z.p$ rot sind.



Zeige:

(E2) Die Wurzel ist schwarz.

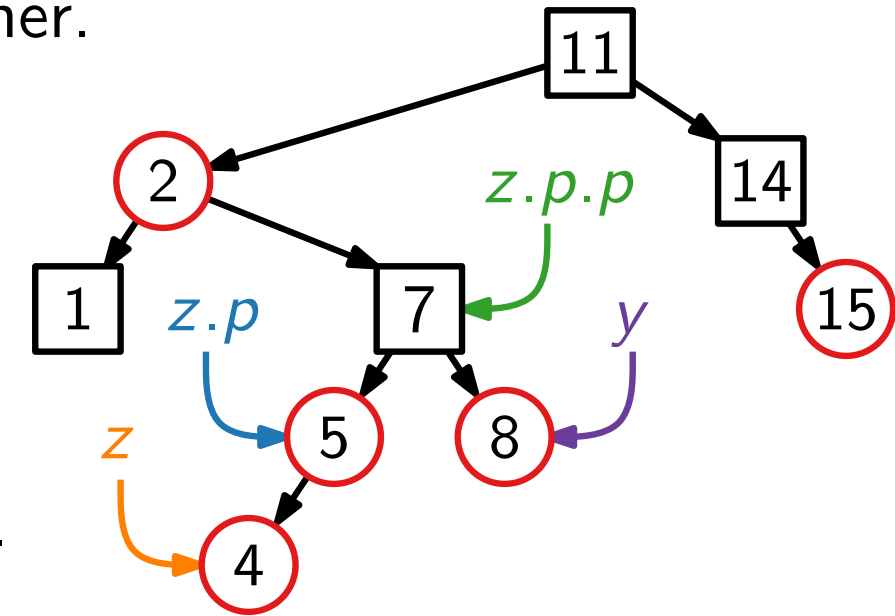
(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.

Korrektheit

Zu zeigen: RBINSERTFIXUP stellt R-S-Eigenschaft wieder her.

Schleifeninvariante (gültig am Anfang der while-Schleife)

- z ist rot.
- Falls $z.p$ die Wurzel ist, dann ist $z.p$ schwarz.
- Falls R-S-Eig. verletzt sind, dann entweder **(E2)** oder **(E4)**.
 - Falls **(E2)** verletzt ist, dann weil $z = root$ und z rot ist.
 - Falls **(E4)** verletzt ist, dann weil z und $z.p$ rot sind.



- Zeige:**
- Initialisierung
 - Aufrechterhaltung
 - Terminierung

(E2) Die Wurzel ist schwarz.

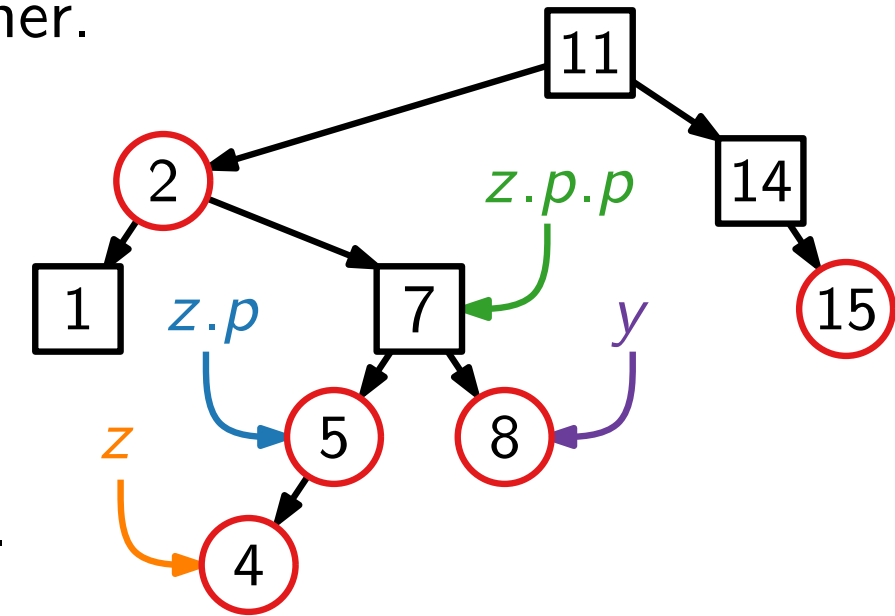
(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.

Korrektheit

Zu zeigen: RBINSERTFIXUP stellt R-S-Eigenschaft wieder her.

Schleifeninvariante (gültig am Anfang der while-Schleife)

- z ist rot.
- Falls $z.p$ die Wurzel ist, dann ist $z.p$ schwarz.
- Falls R-S-Eig. verletzt sind, dann entweder **(E2)** oder **(E4)**.
 - Falls **(E2)** verletzt ist, dann weil $z = root$ und z rot ist.
 - Falls **(E4)** verletzt ist, dann weil z und $z.p$ rot sind.



- Zeige:**
- Initialisierung
 - Aufrechterhaltung
 - Terminierung

- (E2)** Die Wurzel ist schwarz.
(E4) Wenn ein Knoten rot ist,
 sind seine beiden Kinder schwarz.

Viel Arbeit! Siehe [CLRS, Kapitel 13.3].

Laufzeit RBINSERTFIXUP

```

while z.p.color == red do
  if z.p == z.p.p.left then
    y = z.p.p.right    // Tante von z
    if y.color == red then
      z.p.color = black
      z.p.p.color = red
      y.color = black
      z = z.p.p
    else
      if z == z.p.right then
        z = z.p
        LEFTROTATE(z)
      z.p.color = black
      z.p.p.color = red
      RIGHTROTATE(z.p.p)
    else ... // wie oben, aber re. & li. vertauscht
  root.color = black

```

Laufzeit RBINSERTFIXUP

```

while z.p.color == red do
  if z.p == z.p.p.left then
    y = z.p.p.right    // Tante von z
    if y.color == red then
      z.p.color = black
      z.p.p.color = red
      y.color = black
      z = z.p.p
    }  $\mathcal{O}(1)$ 
    else
      if z == z.p.right then
        z = z.p
        LEFTROTATE(z)
      }  $\mathcal{O}(1)$ 
      z.p.color = black
      z.p.p.color = red
      RIGHTROTATE(z.p.p)
    }  $\mathcal{O}(1)$ 
  else ... // wie oben, aber re. & li. vertauscht
root.color = black

```

Laufzeit RBINSERTFIXUP

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      if  $z == z.p.right$  then
         $z = z.p$ 
        LEFTROTATE( $z$ )
       $z.p.color = black$ 
       $z.p.p.color = red$ 
      RIGHTROTATE( $z.p.p$ )
    else ... // wie oben, aber re. & li. vertauscht
  root.color = black
  
```

Klettert im Baum
2 Ebenen nach oben.

Laufzeit RBINSERTFIXUP

```

while z.p.color == red do
  if z.p == z.p.p.left then
    y = z.p.p.right    // Tante von z
    if y.color == red then
      z.p.color = black
      z.p.p.color = red
      y.color = black
      z = z.p.p ←
    else
      if z == z.p.right then
        z = z.p
        LEFTROTATE(z)
      z.p.color = black ←
      z.p.p.color = red
      RIGHTROTATE(z.p.p)
    else ... // wie oben, aber re. & li. vertauscht
  root.color = black
  
```

 $\mathcal{O}(1)$

Klettert im Baum
2 Ebenen nach oben.

 $\mathcal{O}(1)$
 $\mathcal{O}(1)$

Führt zum Abbruch
der **while**-Schleife.

Laufzeit RBINSERTFIXUP

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      if  $z == z.p.right$  then
         $z = z.p$ 
        LEFTROTATE( $z$ )
       $z.p.color = black$ 
       $z.p.p.color = red$ 
      RIGHTROTATE( $z.p.p$ )
    else ... // wie oben, aber re. & li. vertauscht
  root.color = black
  
```

Insgesamt:

- Fall 1 $\mathcal{O}(h)$ mal
- Fall 2 ≤ 1 mal
- Fall 3 ≤ 1 mal

Klettert im Baum
2 Ebenen nach oben.

Führt zum Abbruch
der **while**-Schleife.

Laufzeit RBINSERTFIXUP

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      if  $z == z.p.right$  then
         $z = z.p$ 
        LEFTROTATE( $z$ )
       $z.p.color = black$ 
       $z.p.p.color = red$ 
      RIGHTROTATE( $z.p.p$ )
    else ... // wie oben, aber re. & li. vertauscht
  root.color = black
  
```

Insgesamt:

- Fall 1 $\mathcal{O}(h)$ mal
- Fall 2 ≤ 1 mal
- Fall 3 ≤ 1 mal

Klettert im Baum
2 Ebenen nach oben.

Führt zum Abbruch
der **while**-Schleife.

Laufzeit RBINSERTFIXUP

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von  $z$ 
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      if  $z == z.p.right$  then
         $z = z.p$ 
        LEFTROTATE( $z$ )
       $z.p.color = black$ 
       $z.p.p.color = red$ 
      RIGHTROTATE( $z.p.p$ )
    else ... // wie oben, aber re. & li. vertauscht
  root.color = black
  
```

Insgesamt:

- Fall 1 $\mathcal{O}(h)$ mal
- Fall 2 ≤ 1 mal
- Fall 3 ≤ 1 mal

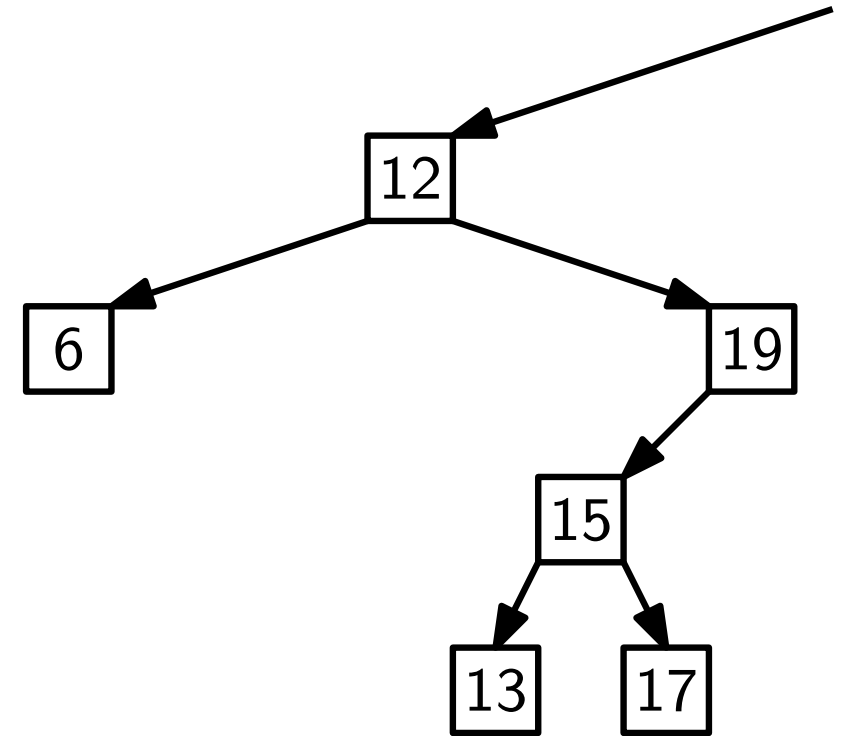
$\mathcal{O}(\log n)$ Umfärbungen
und ≤ 2 Rotationen

Klettert im Baum
2 Ebenen nach oben.

Führt zum Abbruch
der **while**-Schleife.

Löschen

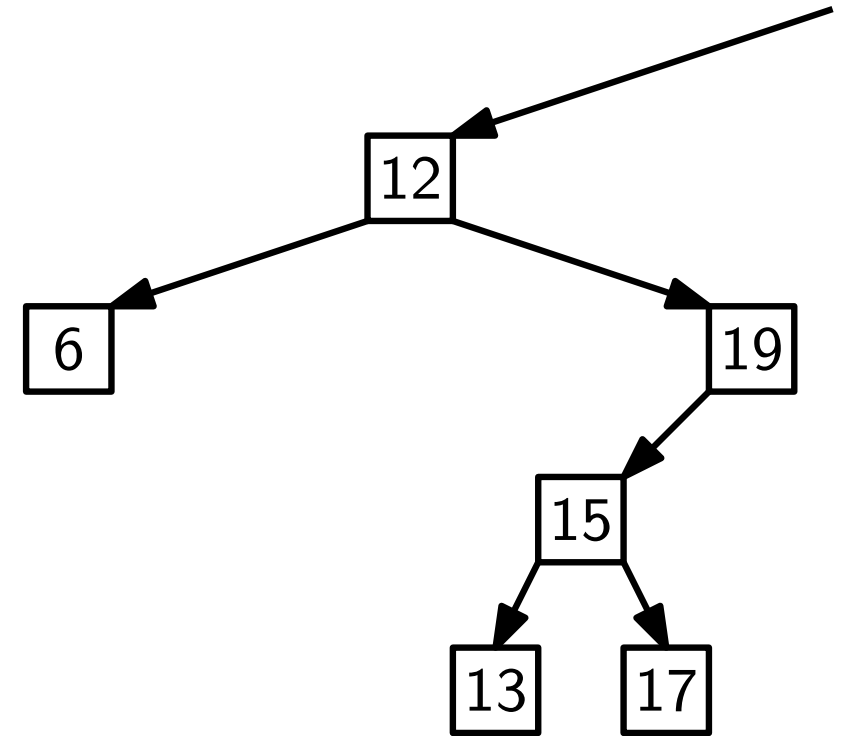
Sei **z** der zu löschende Knoten. Wir betrachten drei Fälle:



Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

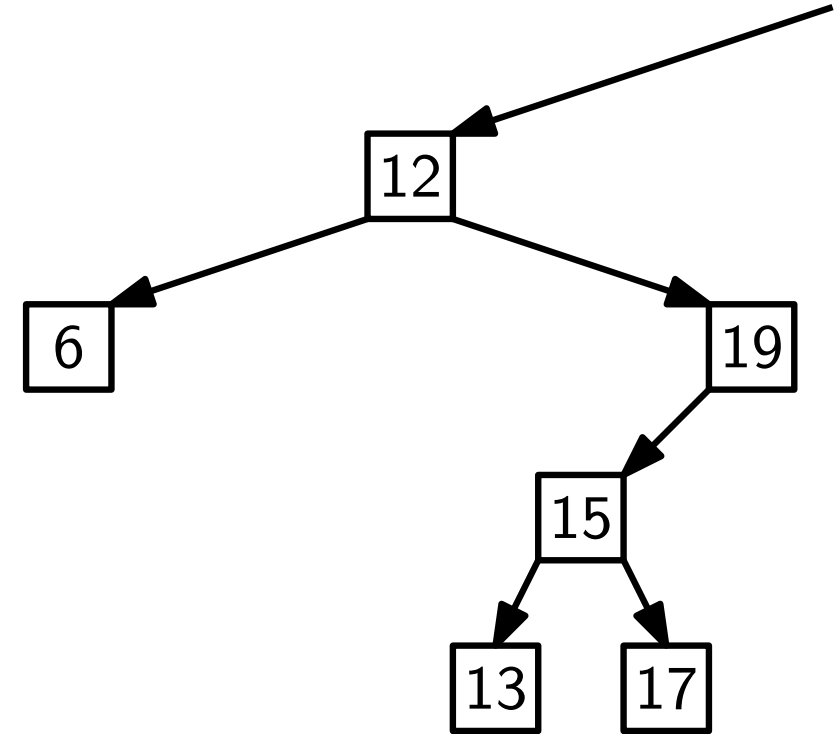


Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

2. z hat ein Kind x .



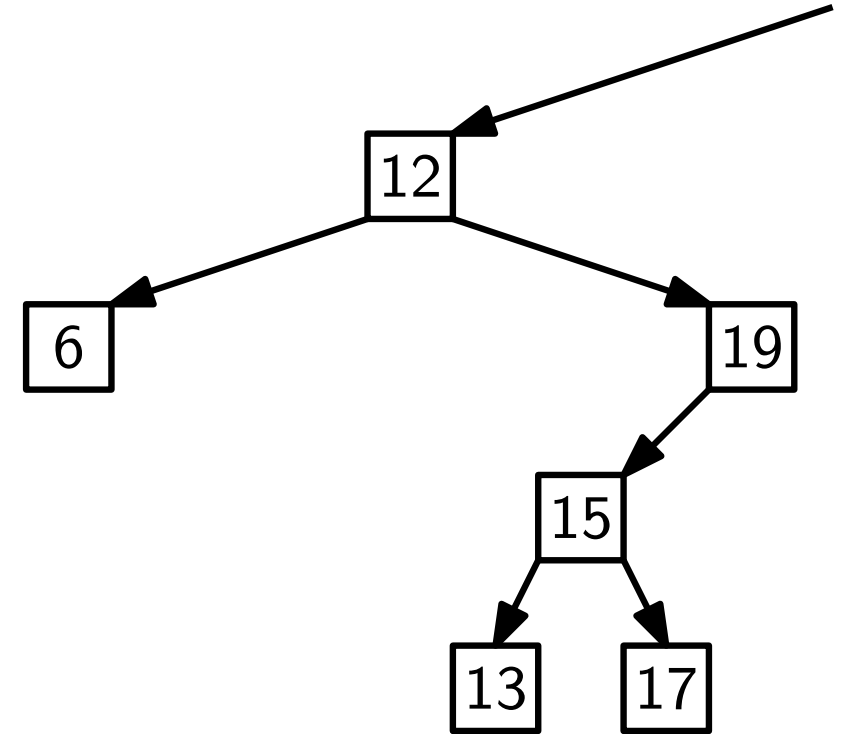
Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

2. z hat ein Kind x .

3. z hat zwei Kinder.



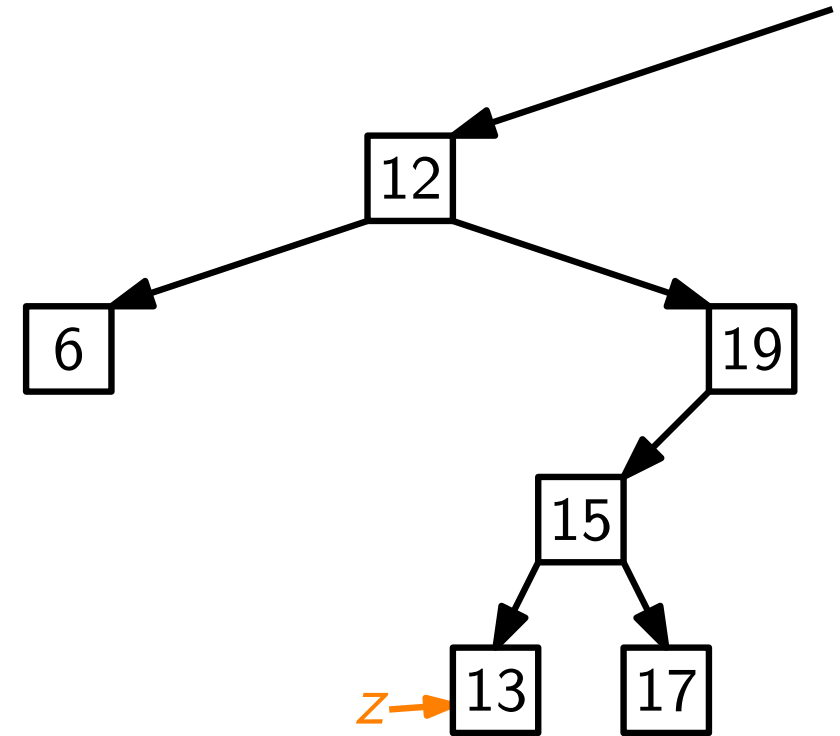
Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

2. z hat ein Kind x .

3. z hat zwei Kinder.



Löschen

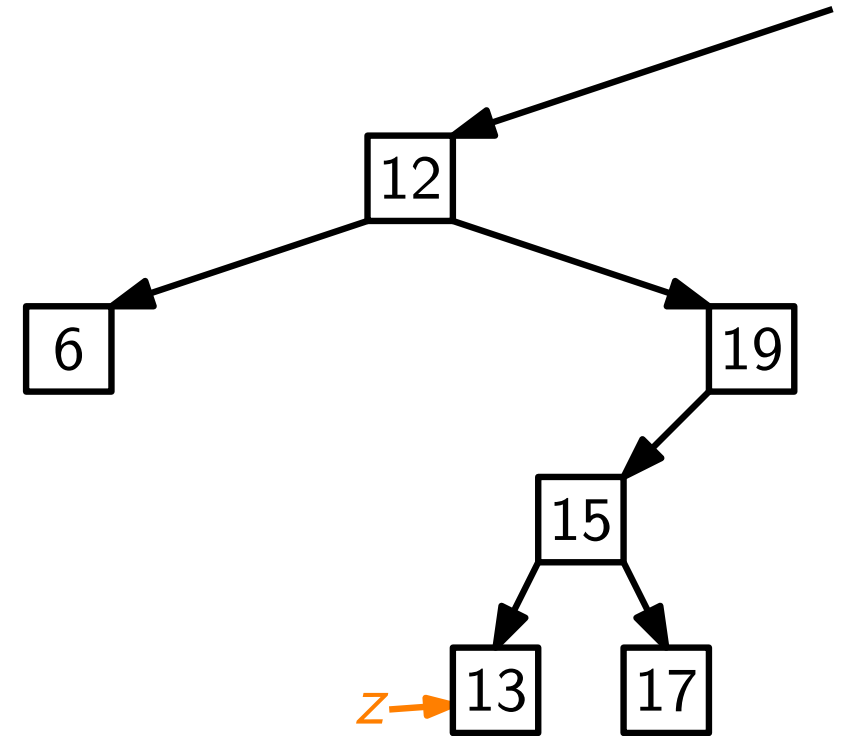
Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

2. z hat ein Kind x .

3. z hat zwei Kinder.



Löschen

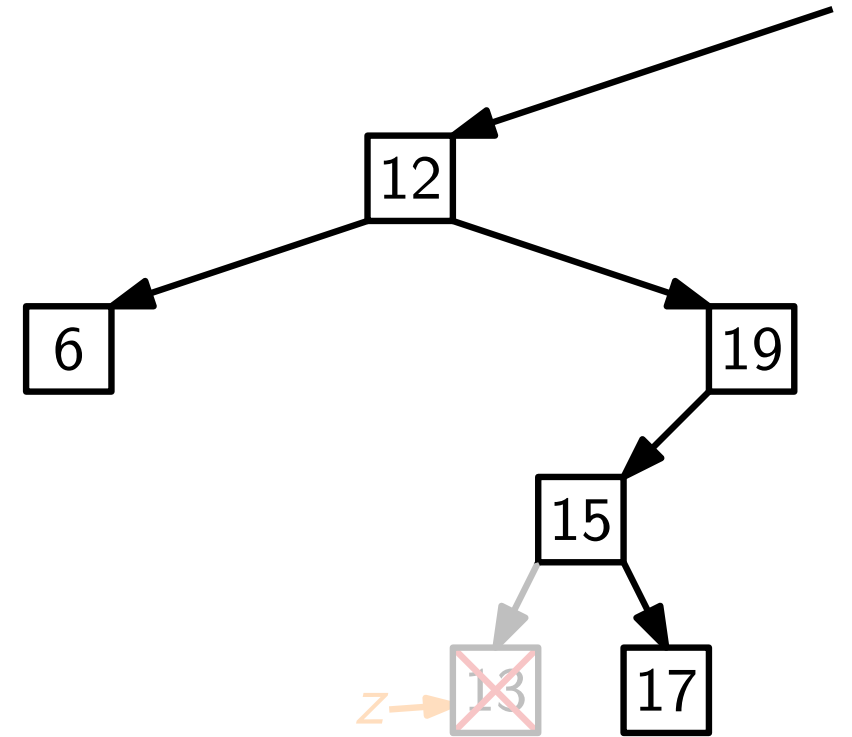
Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

2. z hat ein Kind x .

3. z hat zwei Kinder.



Löschen

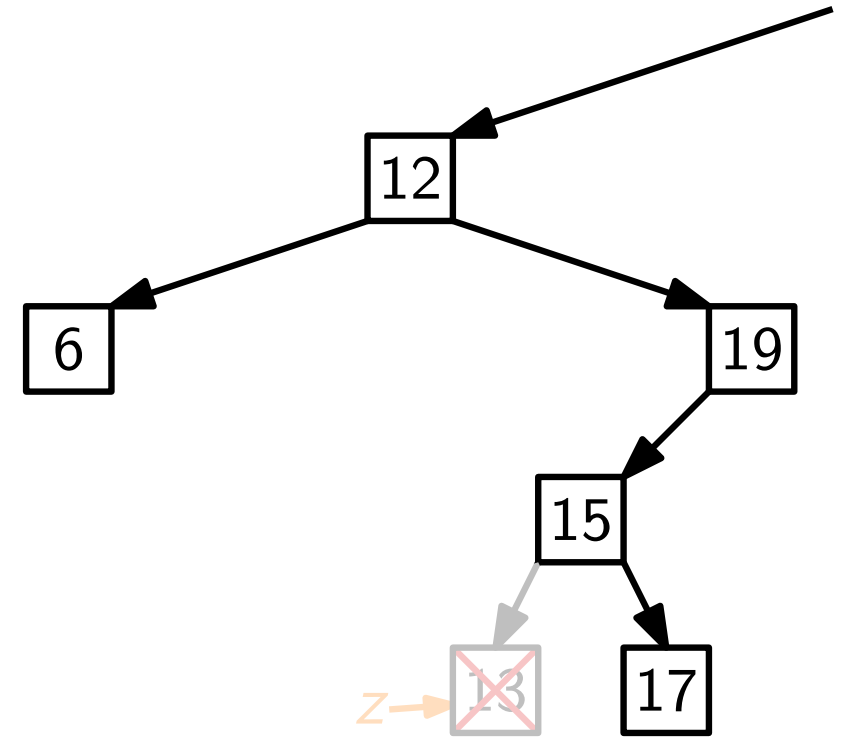
Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .
Was kann kaputt gehen?

2. z hat ein Kind x .

3. z hat zwei Kinder.



Löschen

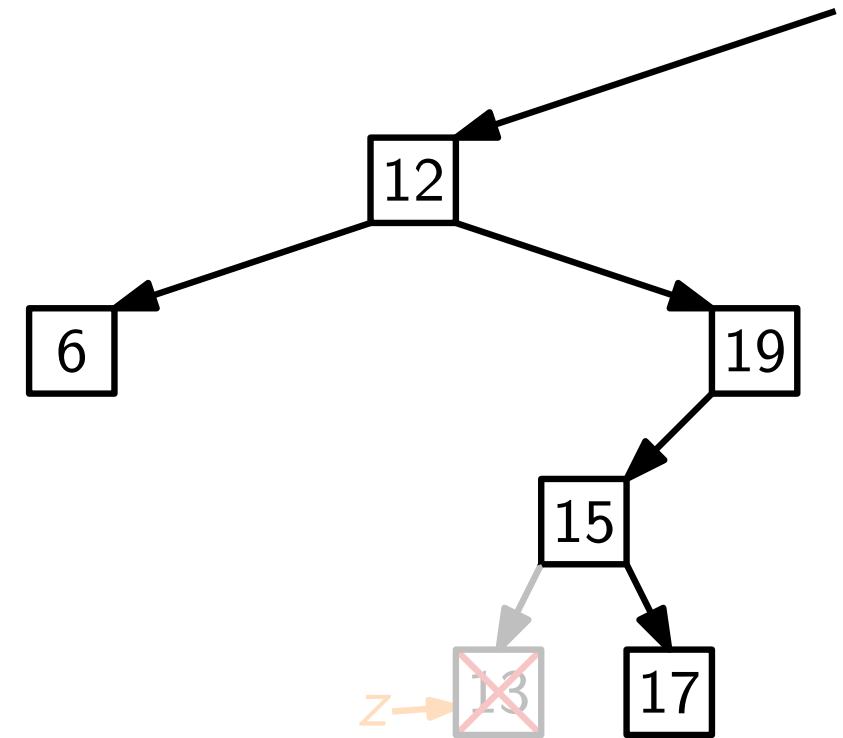
Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .
Was kann kaputt gehen?

2. z hat ein Kind x .

3. z hat zwei Kinder.



- (E1) Jeder Knoten ist entweder rot oder schwarz.
- (E2) Die Wurzel ist schwarz.
- (E3) Alle *nil*-Knoten sind schwarz. *nil*
- (E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten
dieselbe Anzahl schwarzer Knoten.

Löschen

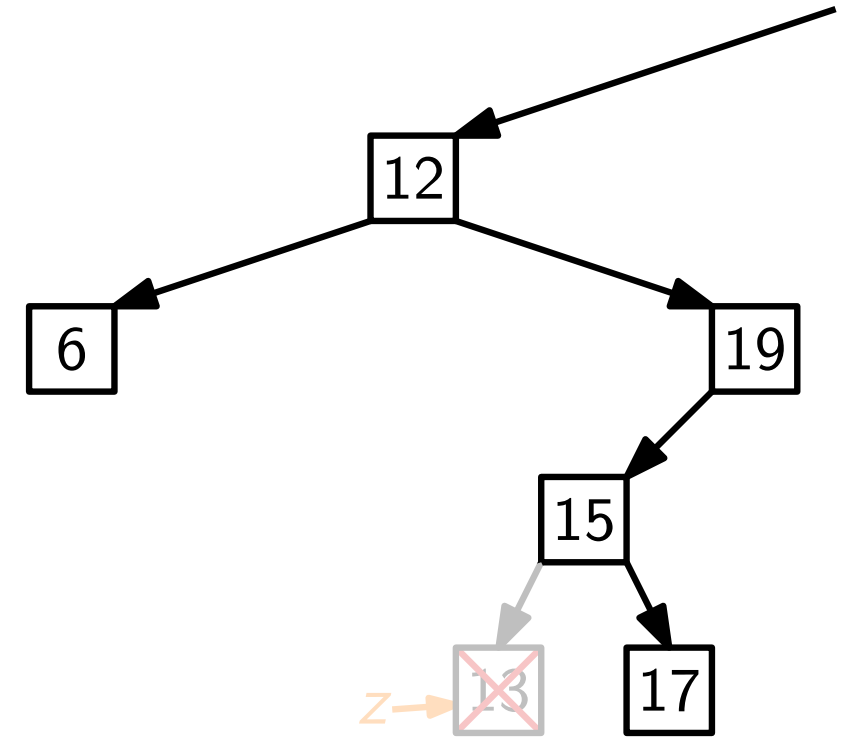
Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .
Was kann kaputt gehen?

2. z hat ein Kind x .

3. z hat zwei Kinder.



- (E2) Die Wurzel ist schwarz.
- (E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade
enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

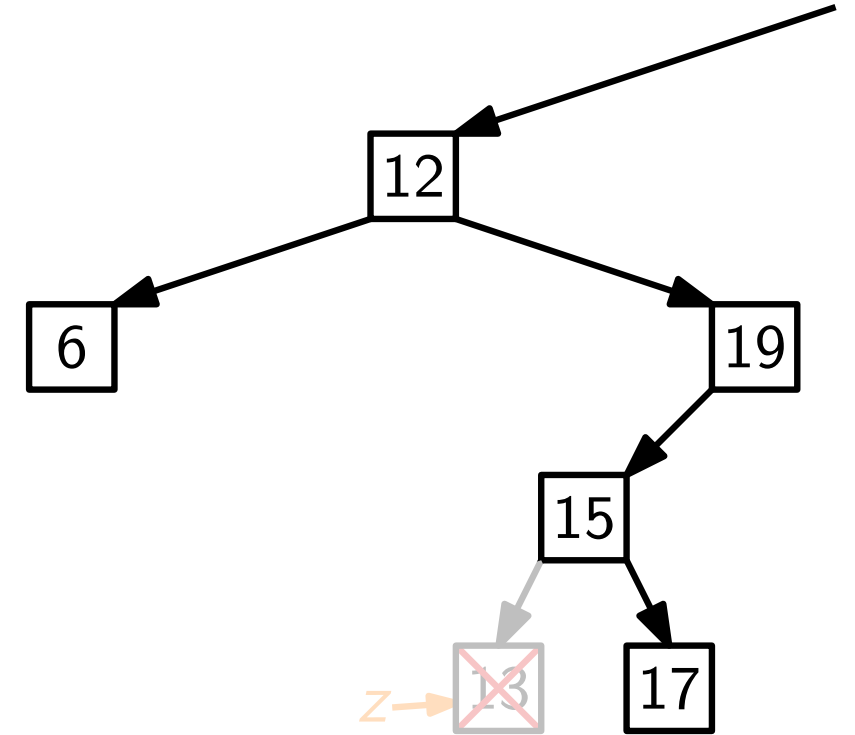
Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen? **(E5)** falls z schwarz war

2. z hat ein Kind x .

3. z hat zwei Kinder.



(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade
enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

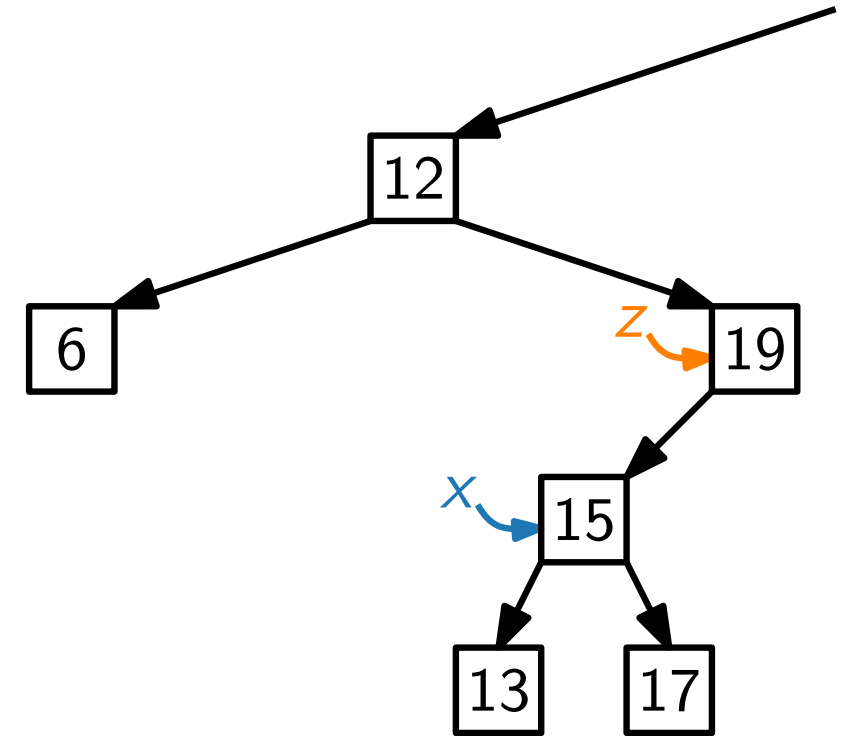
Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen? (E5) falls z schwarz war

2. z hat ein Kind x .

3. z hat zwei Kinder.



(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade
enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

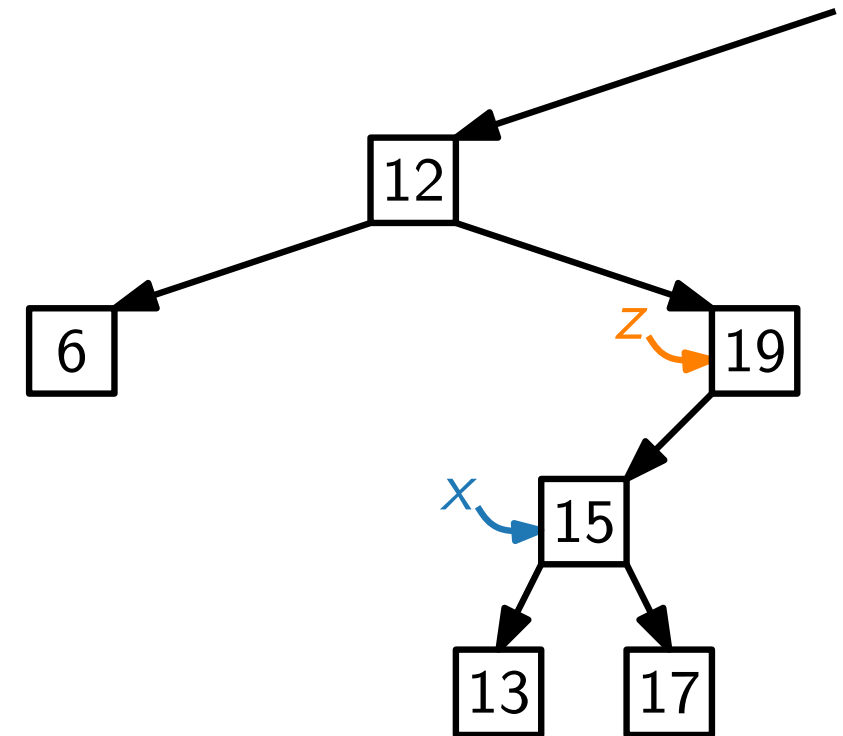
Was kann kaputt gehen? (E5) falls z schwarz war

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

3. z hat zwei Kinder.



(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade
enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

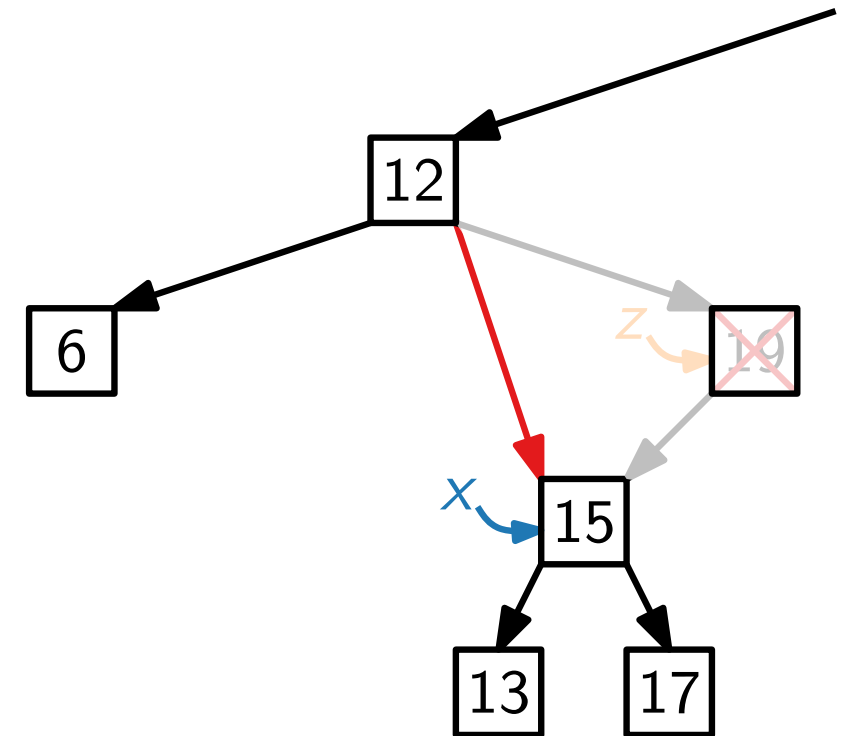
Was kann kaputt gehen? (E5) falls z schwarz war

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

3. z hat zwei Kinder.



(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade
enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen? (E5) falls z schwarz war

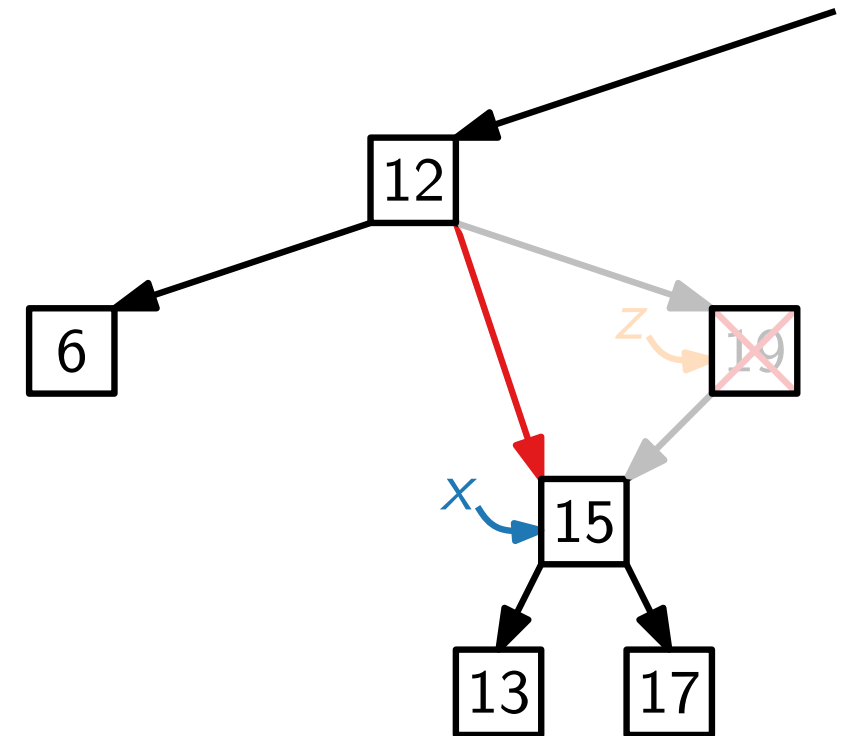
2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

Was kann kaputt gehen?

3. z hat zwei Kinder.



(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade
enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen? (E5) falls z schwarz war

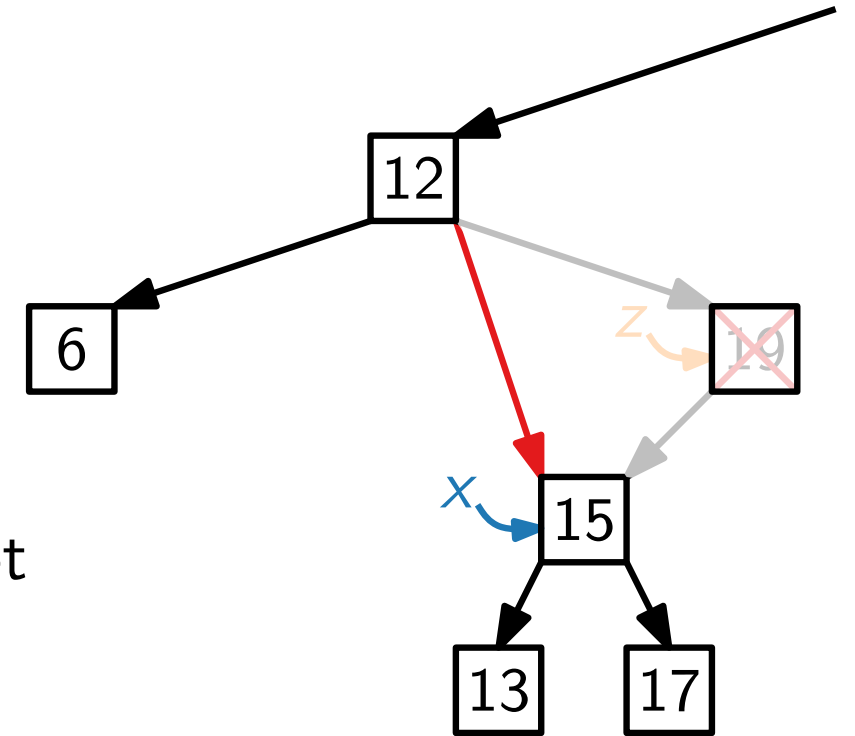
2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

Was kann kaputt gehen? (E2) falls z Wurzel und x rot

3. z hat zwei Kinder.



(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade
enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen? (E5) falls z schwarz war

2. z hat ein Kind x .

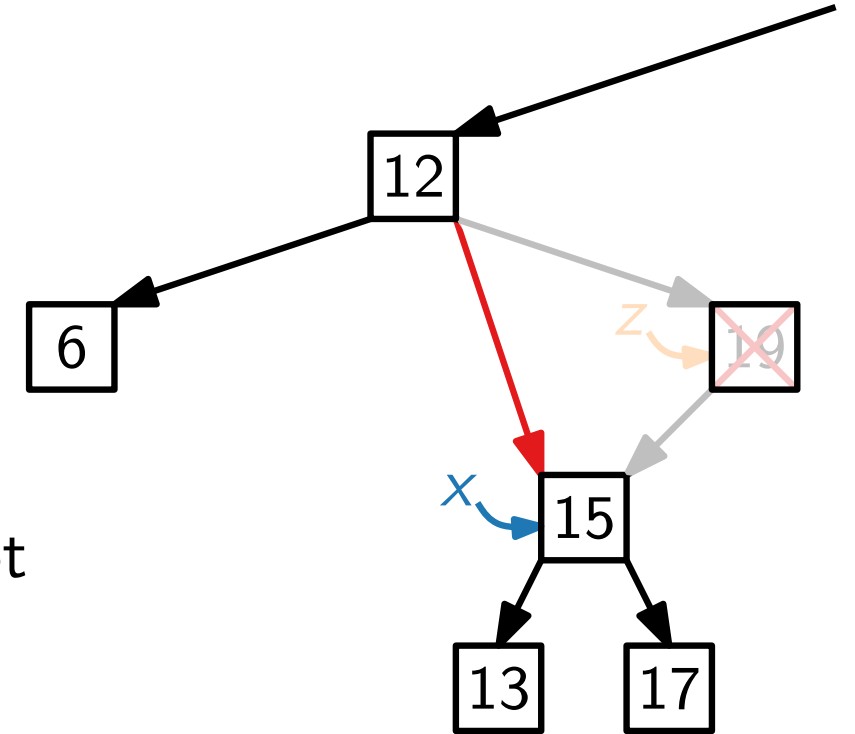
Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

Was kann kaputt gehen? (E2) falls z Wurzel und x rot

(E4) falls x und $z.p$ rot

3. z hat zwei Kinder.



(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade
enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen? (E5) falls z schwarz war

2. z hat ein Kind x .

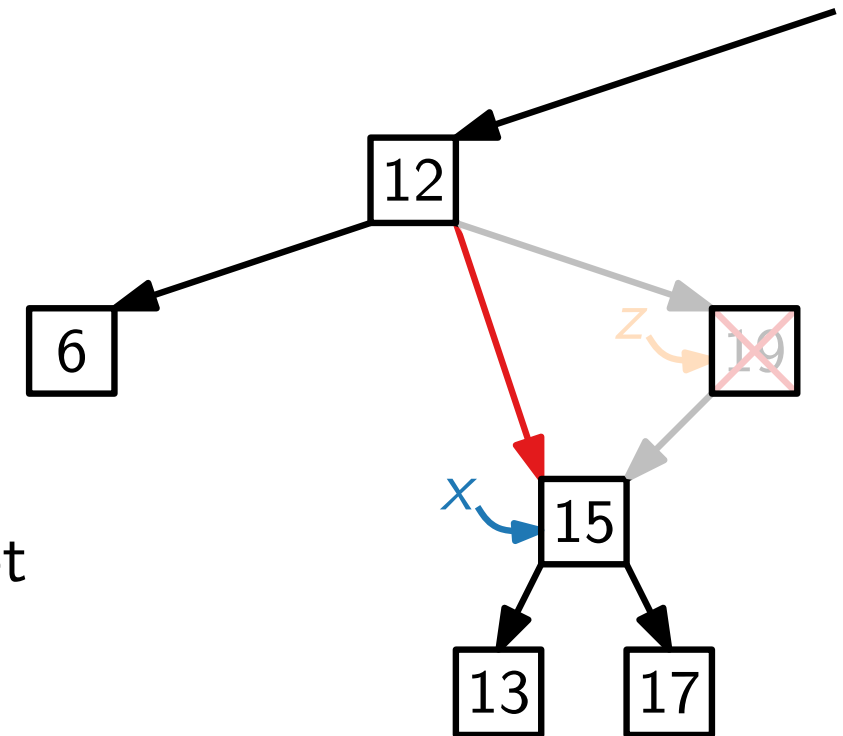
Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

Was kann kaputt gehen? (E2) falls z Wurzel und x rot

(E4) falls x und $z.p$ rot (E5) falls z schwarz war

3. z hat zwei Kinder.



(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade
enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen?

(E5) falls z schwarz war

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

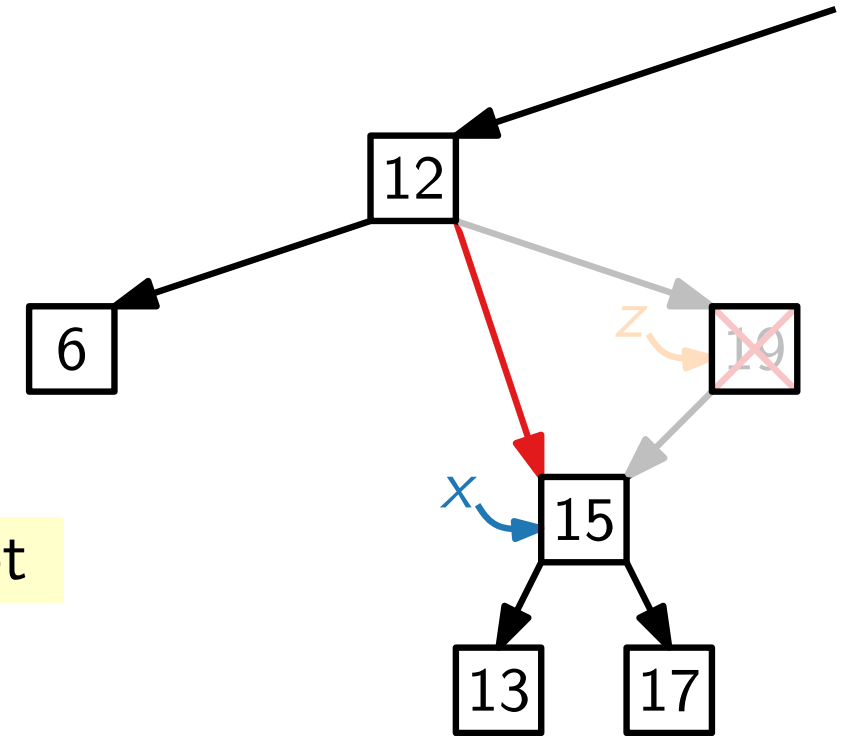
Was kann kaputt gehen?

(E2) falls z Wurzel und x rot

(E4) falls x und $z.p$ rot

(E5) falls z schwarz war

3. z hat zwei Kinder.



(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade
enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen? (E5) falls z schwarz war

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

Was kann kaputt gehen?

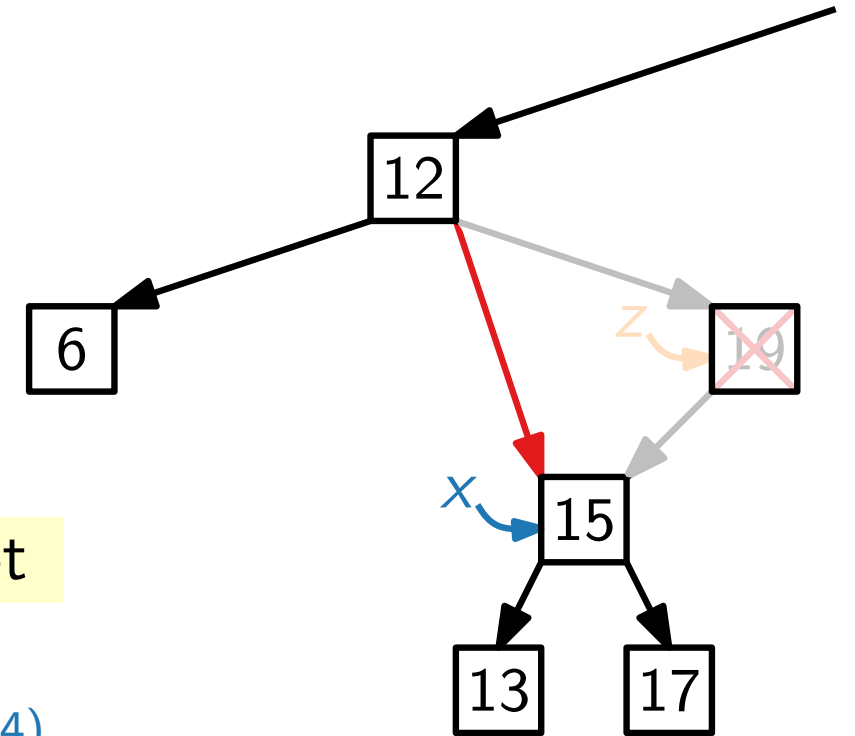
(E2) falls z Wurzel und x rot

(E4) falls x und $z.p$ rot

(E5) falls z schwarz war

3. z hat zwei Kinder.

(E4) $\Rightarrow z$ schwarz



(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade
enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen? (E5) falls z schwarz war

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

Was kann kaputt gehen?

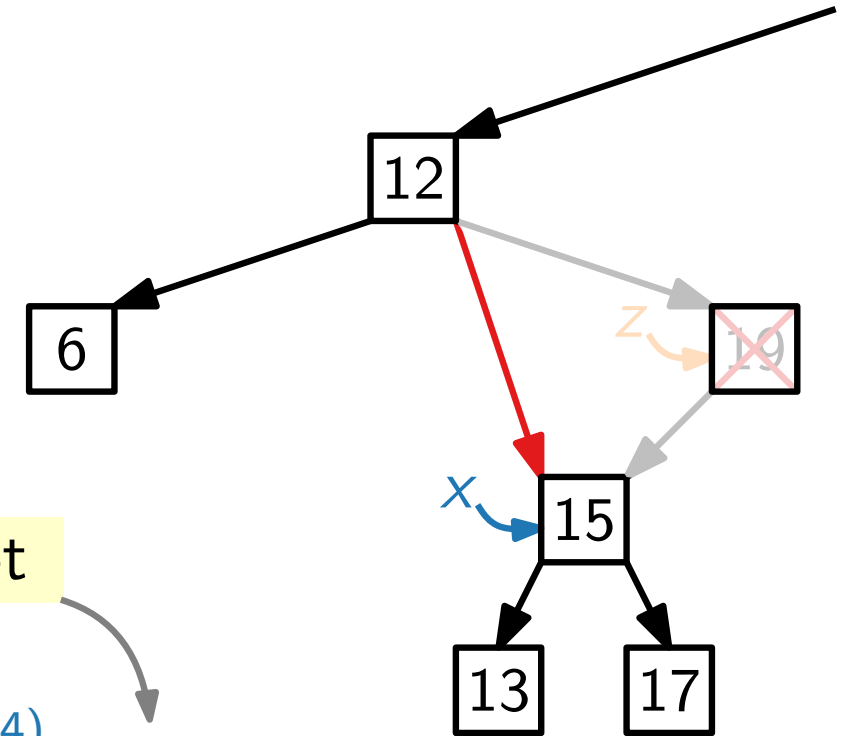
(E4) falls x und $z.p$ rot

(E2) falls z Wurzel und x rot

(E5) falls z schwarz war

3. z hat zwei Kinder.

(E4) $\Rightarrow z$ schwarz



(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade
enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen? (E5) falls z schwarz war

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

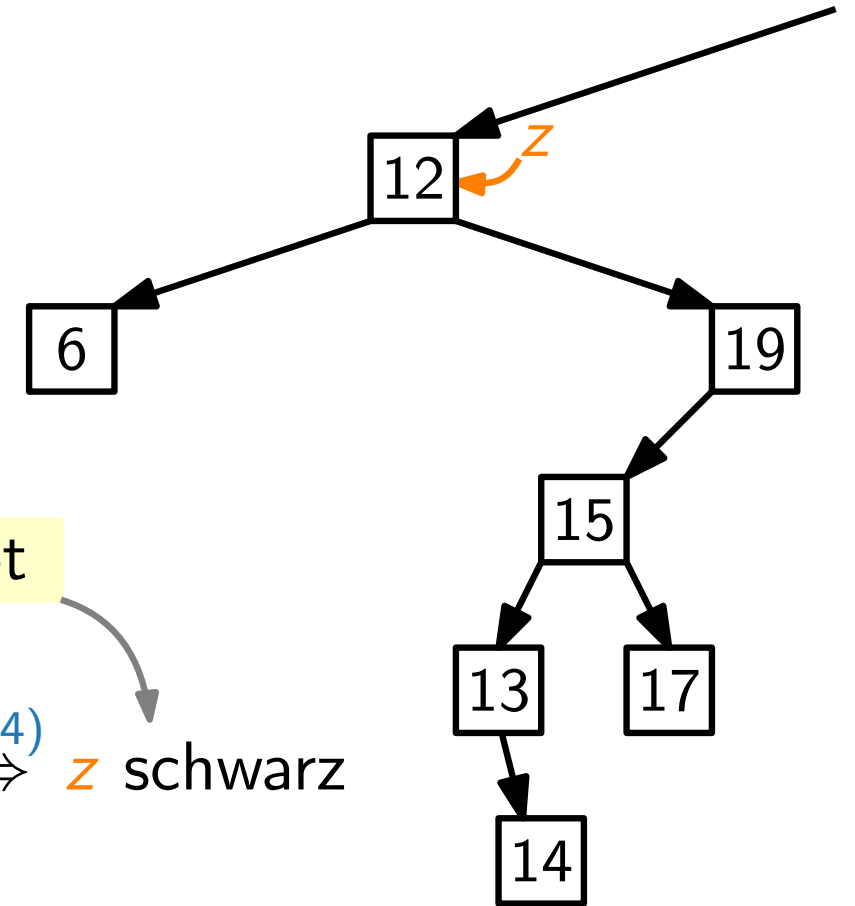
Was kann kaputt gehen?

(E4) falls x und $z.p$ rot

(E2) falls z Wurzel und x rot

(E5) falls z schwarz war

3. z hat zwei Kinder.



(E4) $\Rightarrow z$ schwarz

(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen? (E5) falls z schwarz war

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

Was kann kaputt gehen?

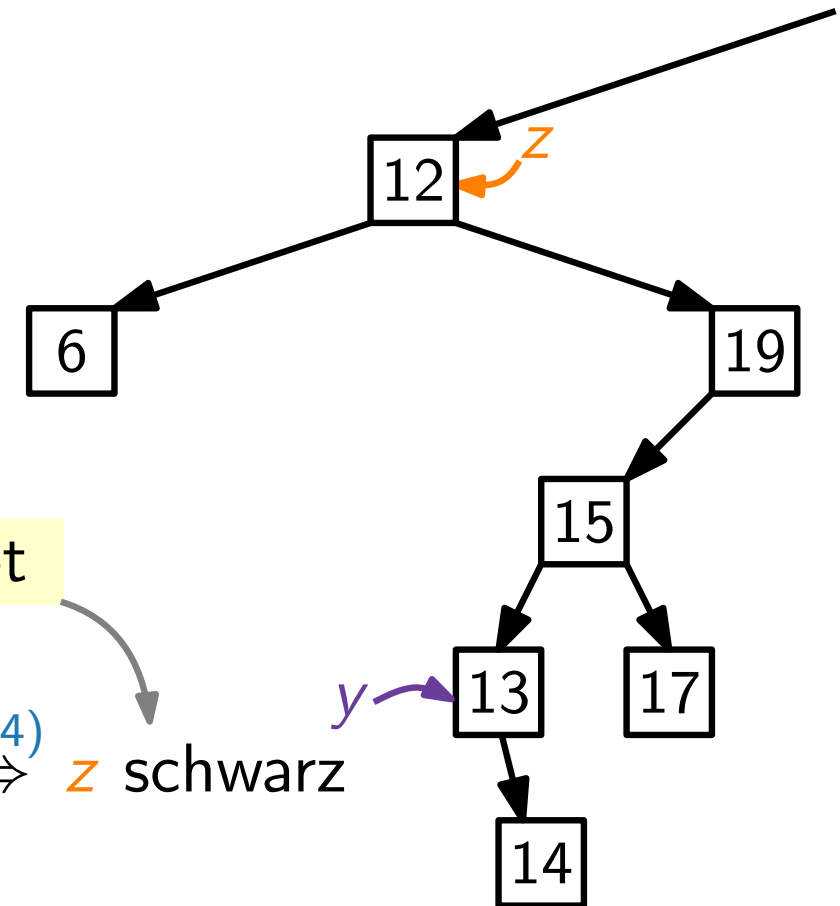
(E4) falls x und $z.p$ rot

(E2) falls z Wurzel und x rot

(E5) falls z schwarz war

3. z hat zwei Kinder.

Setze $y = \text{SUCCESSOR}(z)$.



(E4) $\Rightarrow$ z schwarz

(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen? (E5) falls z schwarz war

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

Was kann kaputt gehen?

(E4) falls x und $z.p$ rot

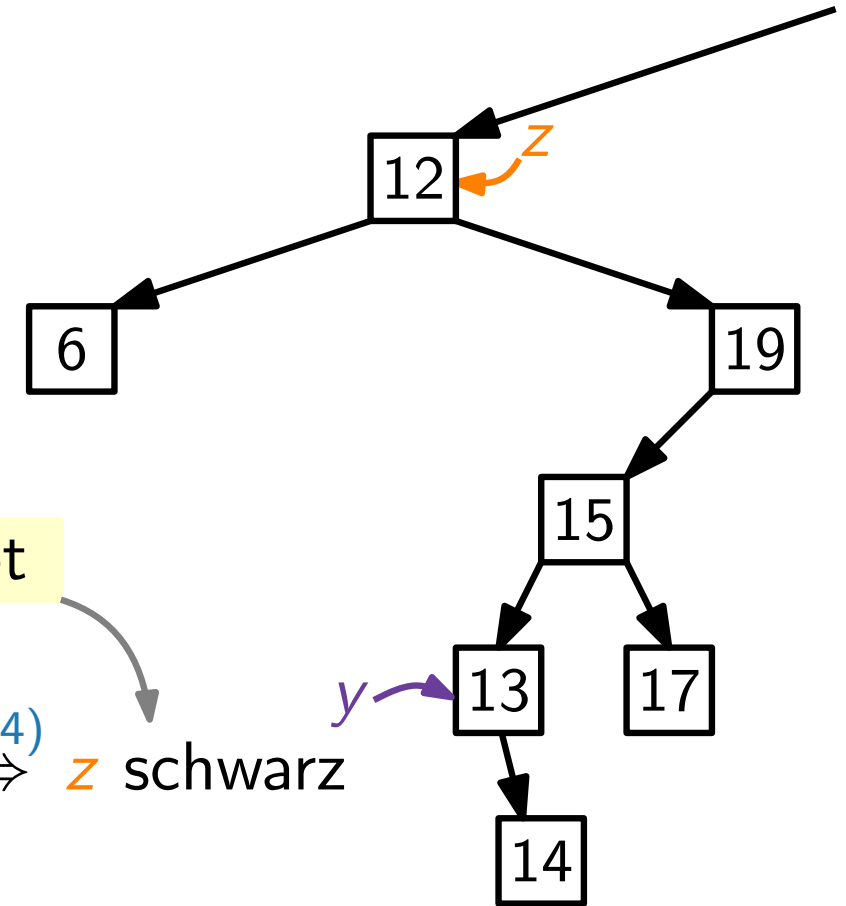
(E2) falls z Wurzel und x rot

(E5) falls z schwarz war

3. z hat zwei Kinder.

Setze $y = \text{SUCCESSOR}(z)$.

Setze $x = y.right$ and Stelle von y .



(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen? (E5) falls z schwarz war

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

Was kann kaputt gehen?

(E4) falls x und $z.p$ rot

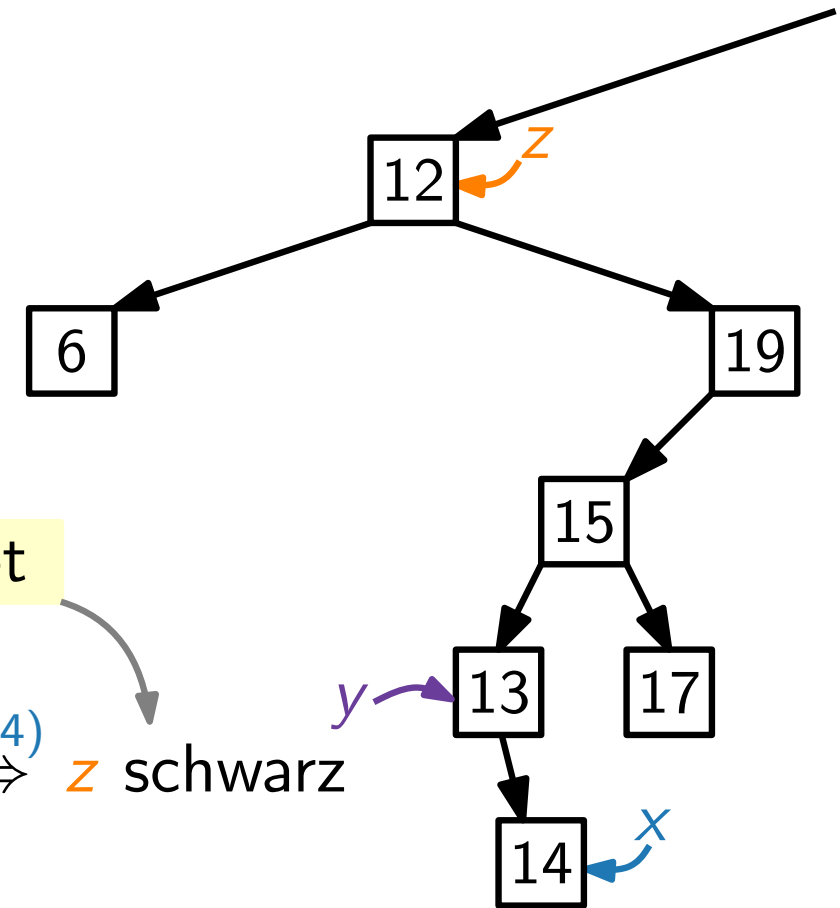
(E2) falls z Wurzel und x rot

(E5) falls z schwarz war

3. z hat zwei Kinder.

Setze $y = \text{SUCCESSOR}(z)$.

Setze $x = y.right$ and Stelle von y .



(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen? (E5) falls z schwarz war

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

Was kann kaputt gehen?

(E4) falls x und $z.p$ rot

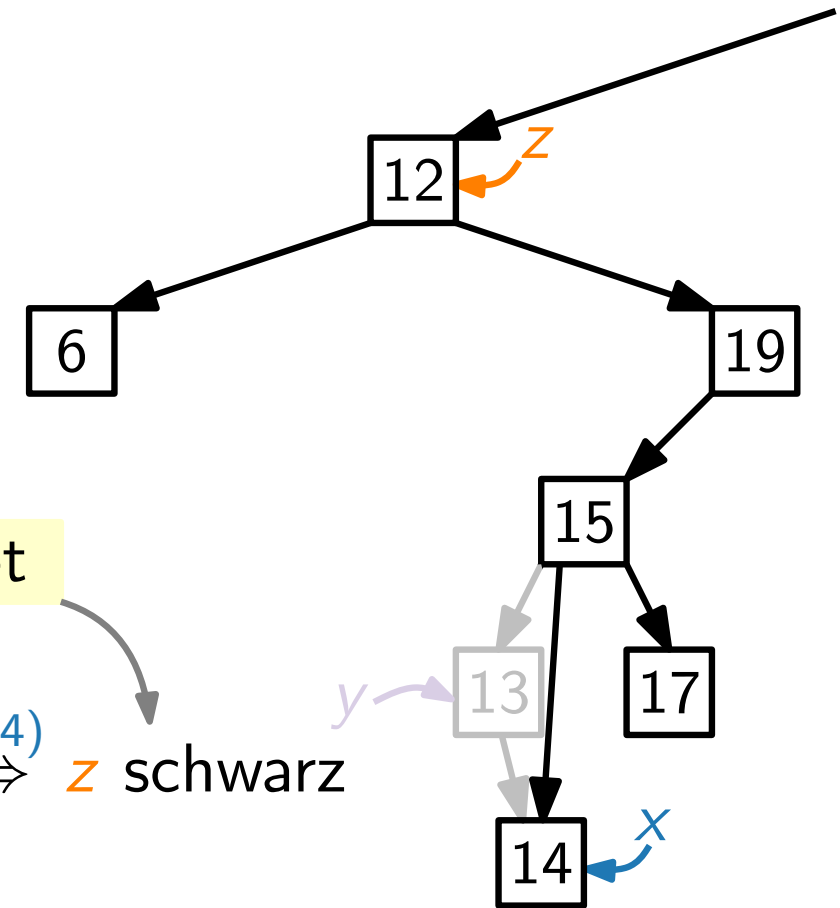
(E2) falls z Wurzel und x rot

(E5) falls z schwarz war

3. z hat zwei Kinder.

Setze $y = \text{SUCCESSOR}(z)$.

Setze $x = y.right$ and Stelle von y .



(E4) $\Rightarrow z$ schwarz

(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen? (E5) falls z schwarz war

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

Was kann kaputt gehen?

(E4) falls x und $z.p$ rot

(E2) falls z Wurzel und x rot

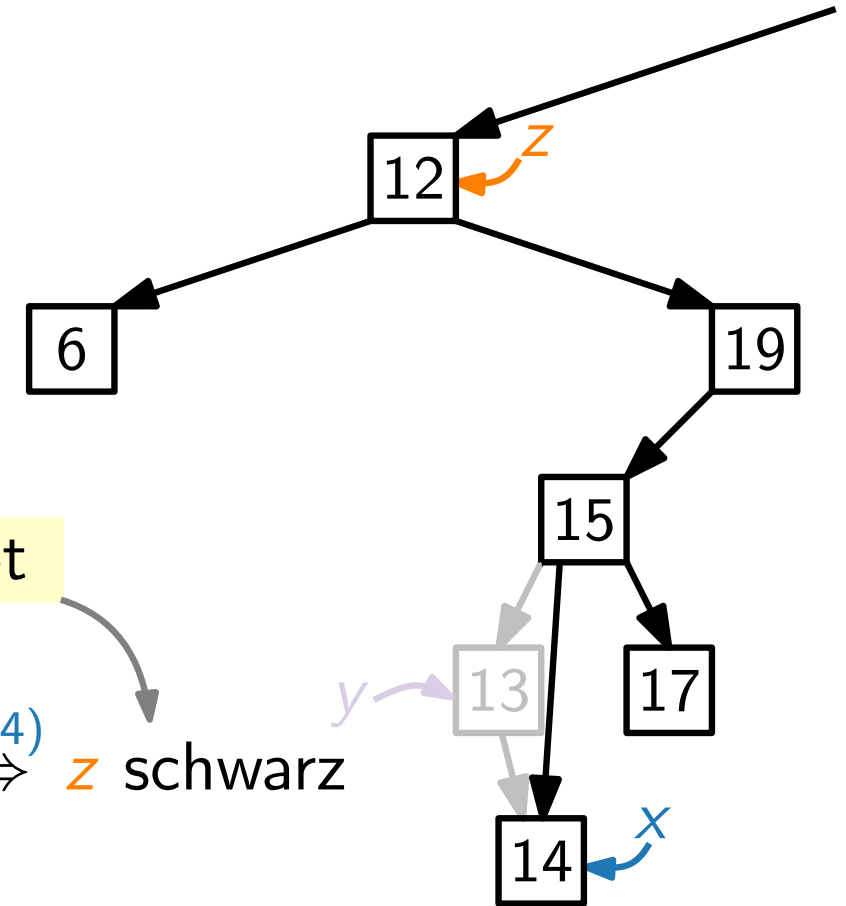
(E5) falls z schwarz war

3. z hat zwei Kinder.

Setze $y = \text{SUCCESSOR}(z)$.

Setze $x = y.right$ and Stelle von y .

Setze $z.key = y.key$.



(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen? (E5) falls z schwarz war

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

Was kann kaputt gehen?

(E2) falls z Wurzel und x rot

(E4) falls x und $z.p$ rot

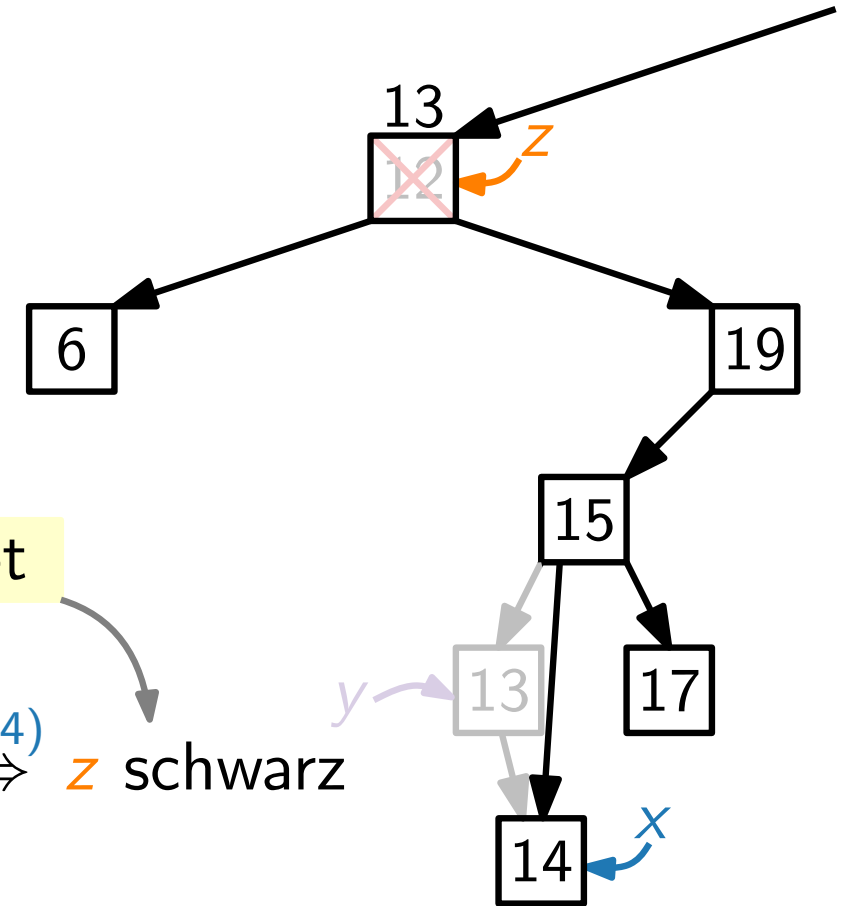
(E5) falls z schwarz war

3. z hat zwei Kinder.

Setze $y = \text{SUCCESSOR}(z)$.

Setze $x = y.right$ and Stelle von y .

Setze $z.key = y.key$.



(E4) $\Rightarrow z$ schwarz

- (E2) Die Wurzel ist schwarz.
- (E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen? (E5) falls z schwarz war

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

Was kann kaputt gehen?

(E4) falls x und $z.p$ rot

(E2) falls z Wurzel und x rot

(E5) falls z schwarz war

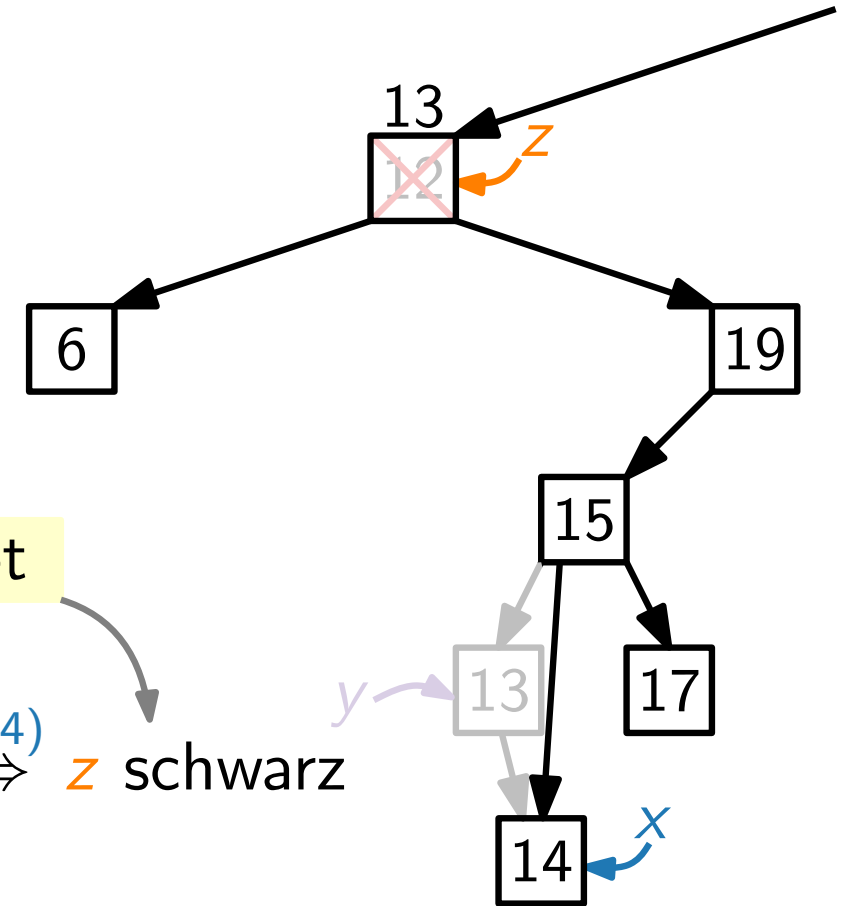
3. z hat zwei Kinder.

Setze $y = \text{SUCCESSOR}(z)$.

Setze $x = y.right$ and Stelle von y .

Setze $z.key = y.key$.

Was kann kaputt gehen?



(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen? (E5) falls z schwarz war

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

Was kann kaputt gehen?

(E4) falls x und $z.p$ rot

(E2) falls z Wurzel und x rot

(E5) falls z schwarz war

3. z hat zwei Kinder.

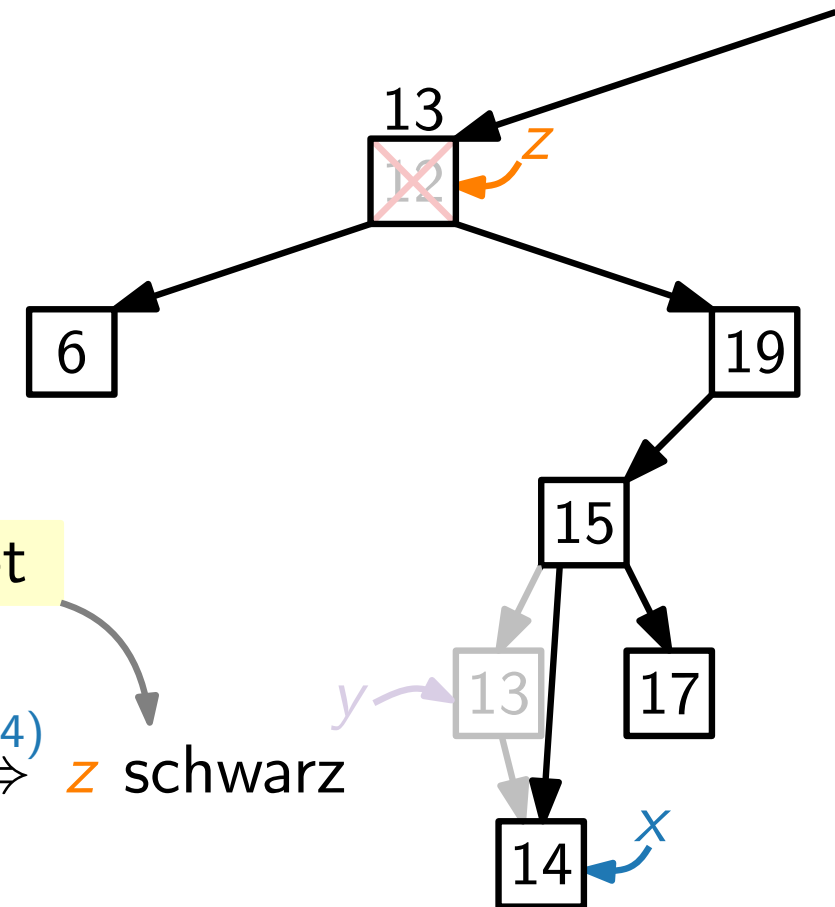
Setze $y = \text{SUCCESSOR}(z)$.

Setze $x = y.right$ and Stelle von y .

Setze $z.key = y.key$.

Was kann kaputt gehen?

(E4) falls x und $y.p$ rot



(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen? (E5) falls z schwarz war

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

Was kann kaputt gehen?

(E4) falls x und $z.p$ rot

(E2) falls z Wurzel und x rot

(E5) falls z schwarz war

3. z hat zwei Kinder.

Setze $y = \text{SUCCESSOR}(z)$.

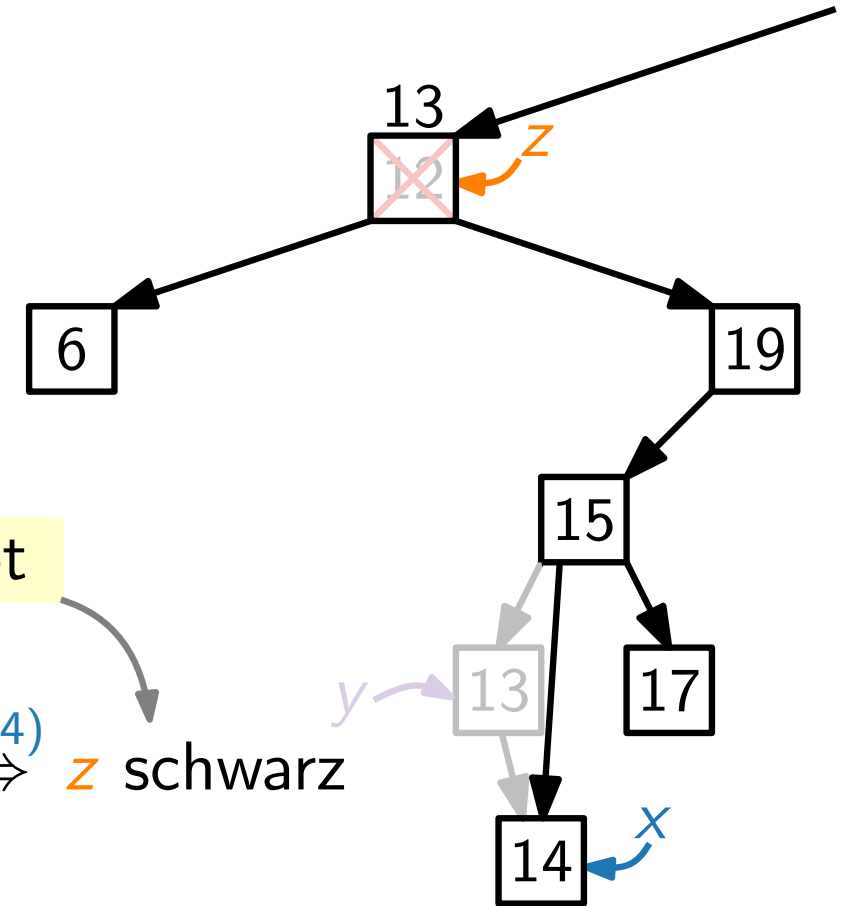
Setze $x = y.right$ and Stelle von y .

Setze $z.key = y.key$.

Was kann kaputt gehen?

(E4) falls x und $y.p$ rot

(E5) falls y schwarz war



(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen? (E5) falls z schwarz war

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

Was kann kaputt gehen?

(E4) falls x und $z.p$ rot

(E2) falls z Wurzel und x rot

(E5) falls z schwarz war

3. z hat zwei Kinder.

Setze $y = \text{SUCCESSOR}(z)$.

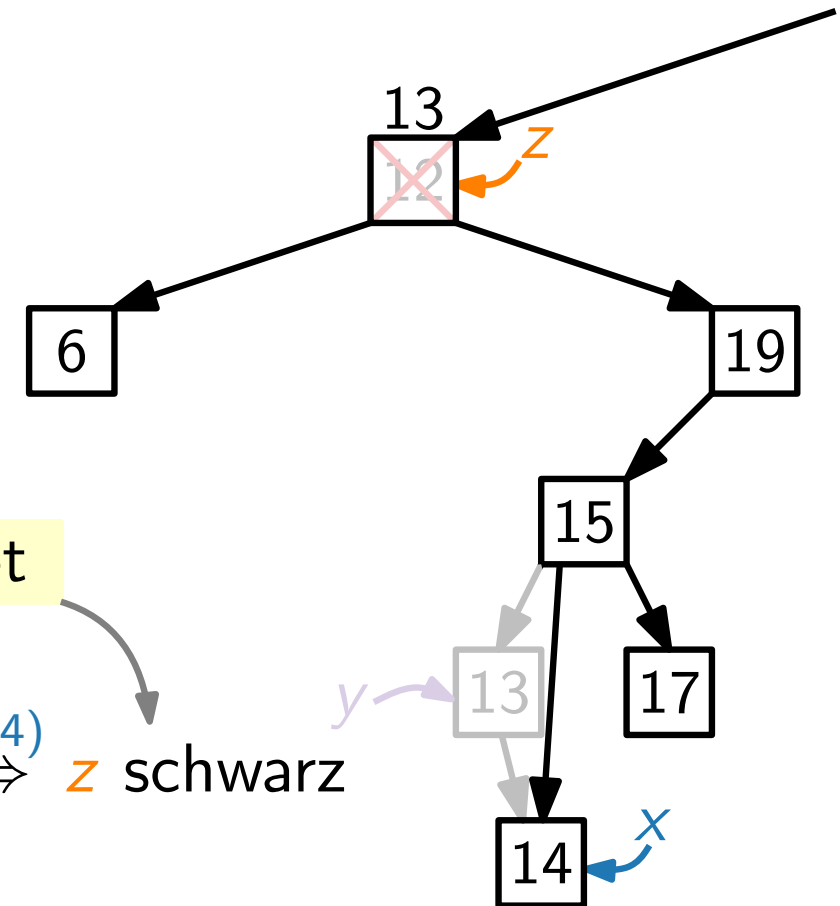
Setze $x = y.right$ and Stelle von y .

Setze $z.key = y.key$.

Was kann kaputt gehen?

(E4) falls x und $y.p$ rot

(E5) falls y schwarz war



(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen? (E5) falls z schwarz war

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

Was kann kaputt gehen? (E2) falls z Wurzel und x rot

(E4) falls x und $z.p$ rot (E5) falls z schwarz war

3. z hat zwei Kinder.

Setze $y = \text{SUCCESSOR}(z)$.

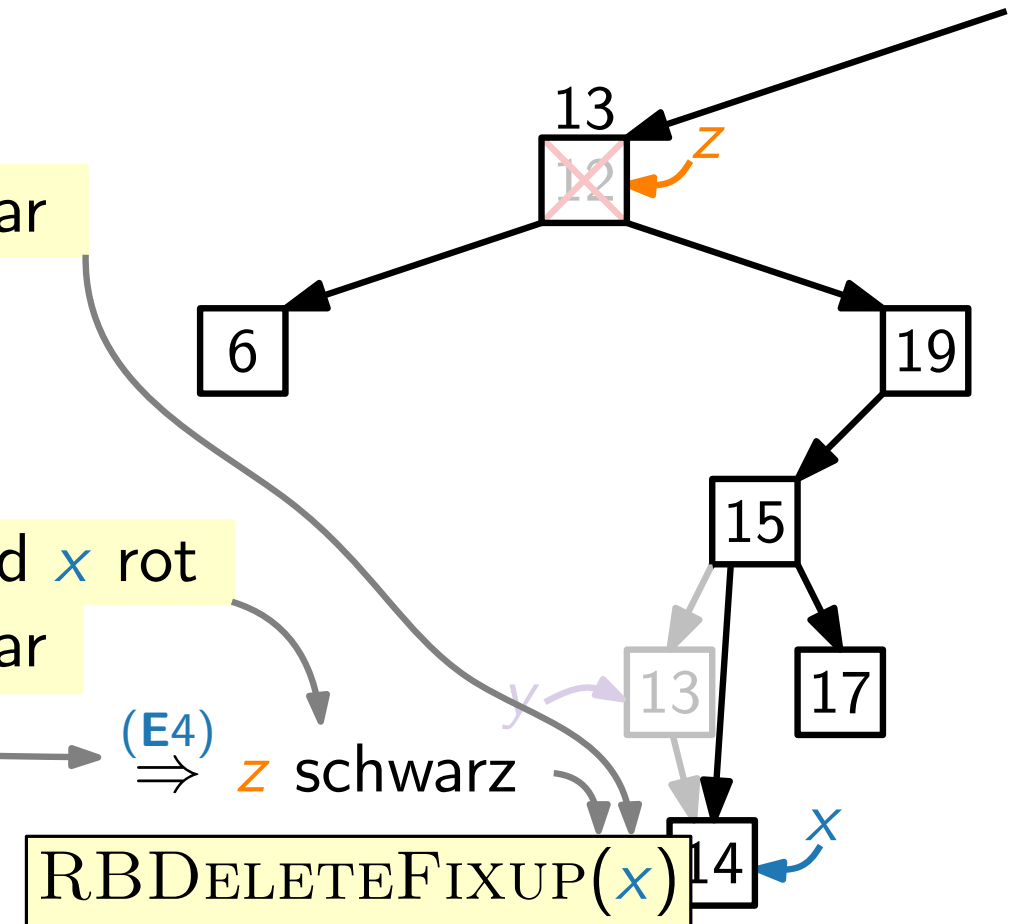
Setze $x = y.right$ and Stelle von y .

Setze $z.key = y.key$.

Was kann kaputt gehen?

(E4) falls x und $y.p$ rot

(E5) falls y schwarz war



(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade
enthalten dieselbe Anzahl schwarzer Knoten.

Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

Was kann kaputt gehen?

(E5) falls z schwarz war

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

Was kann kaputt gehen?

(E2) falls z Wurzel und x rot

(E4) falls x und $z.p$ rot

(E5) falls z schwarz war

3. z hat zwei Kinder.

Setze $y = \text{SUCCESSOR}(z)$.

Setze $x = y.right$ and Stelle von y .

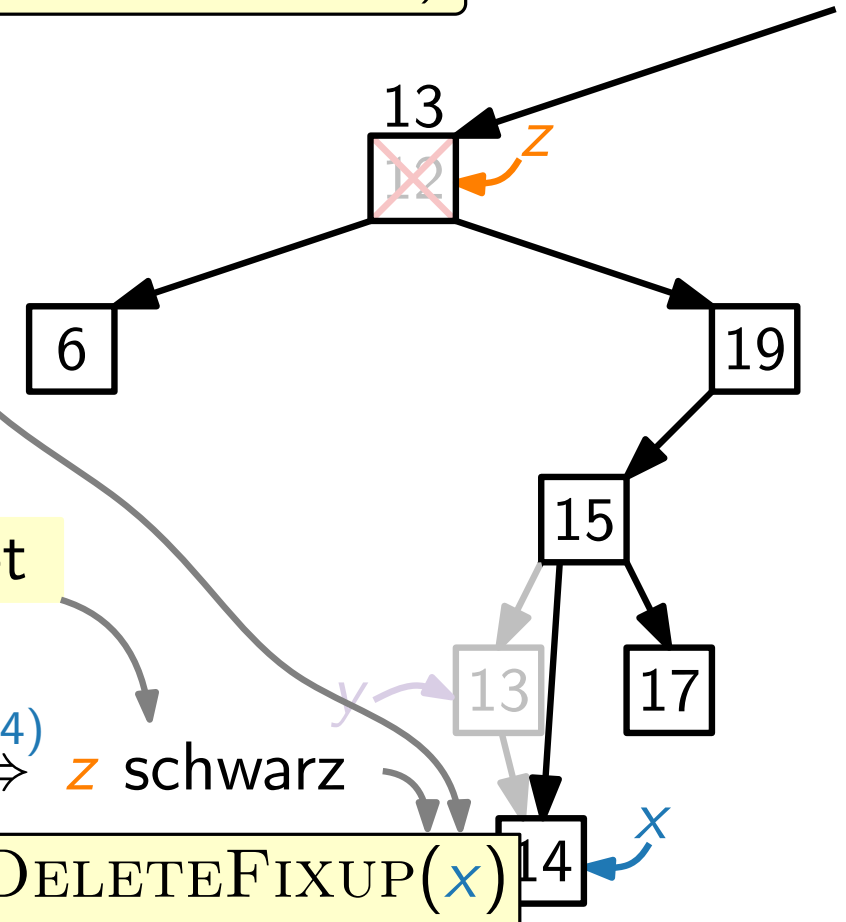
Setze $z.key = y.key$.

Was kann kaputt gehen?

(E4) falls x und $y.p$ rot

(E5) falls y schwarz war

Hier ist $x = z$ (obwohl z nicht mehr im Baum ist).



(E2) Die Wurzel ist schwarz.

(E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.

(E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade
enthalten dieselbe Anzahl schwarzer Knoten.

RBD~~E~~LETEFIXUP

Details siehe [CLRS, Kapitel 13.4]

- (E2) Die Wurzel ist schwarz.
- (E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten
dieselbe Anzahl schwarzer Knoten.

RBDELETEFIXUP

Details siehe [CLRS, Kapitel 13.4]

- (E2) Die Wurzel ist schwarz.
- (E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten
dieselbe Anzahl schwarzer Knoten.

Repariere (E5): Auf Knoten x fehlt eine schwarze Einheit.

RBD_{DELETEFIXUP}

Details siehe [CLRS, Kapitel 13.4]

- (E2) Die Wurzel ist schwarz.
- (E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten
dieselbe Anzahl schwarzer Knoten.

Repariere (E5): Auf Knoten x fehlt eine schwarze Einheit.

Ziel: „Hole“ eine schwarze Einheit von weiter oben bis:

RBD_{DELETEFIXUP}

Details siehe [CLRS, Kapitel 13.4]

- (E2) Die Wurzel ist schwarz.
- (E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten
dieselbe Anzahl schwarzer Knoten.

Repariere (E5): Auf Knoten x fehlt eine schwarze Einheit.

Ziel: „Hole“ eine schwarze Einheit von weiter oben bis:

- x ist rot $\Rightarrow$ mach x schwarz.

RBD_{DELETEFIXUP}

Details siehe [CLRS, Kapitel 13.4]

- (E2) Die Wurzel ist schwarz.
- (E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten
dieselbe Anzahl schwarzer Knoten.

Repariere (E5): Auf Knoten x fehlt eine schwarze Einheit.

Ziel: „Hole“ eine schwarze Einheit von weiter oben bis:

- x ist rot $\Rightarrow$ mach x schwarz.
- x ist Wurzel $\Rightarrow$ Schwarz-höhe überall 1 niedriger.

RBD_{DELETEFIXUP}

Details siehe [CLRS, Kapitel 13.4]

- (E2) Die Wurzel ist schwarz.
- (E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten
dieselbe Anzahl schwarzer Knoten.

Repariere (E5): Auf Knoten x fehlt eine schwarze Einheit.

Ziel: „Hole“ eine schwarze Einheit von weiter oben bis:

- x ist rot $\Rightarrow$ mach x schwarz.
- x ist Wurzel $\Rightarrow$ Schwarz-höhe überall 1 niedriger.

Problem wird lokal durch Umfärben & Rotieren gelöst.

RBDELETEFIXUP

Details siehe [CLRS, Kapitel 13.4]

- (E2) Die Wurzel ist schwarz.
- (E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten
dieselbe Anzahl schwarzer Knoten.

Repariere (E5): Auf Knoten x fehlt eine schwarze Einheit.

Ziel: „Hole“ eine schwarze Einheit von weiter oben bis:

- x ist rot $\Rightarrow$ mach x schwarz.
- x ist Wurzel $\Rightarrow$ Schwarz-höhe überall 1 niedriger.

Problem wird lokal durch Umfärben & Rotieren gelöst.

Laufzeit RBDELETE $\in$

RBDELETEFIXUP

Details siehe [CLRS, Kapitel 13.4]

- (E2) Die Wurzel ist schwarz.
- (E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten
dieselbe Anzahl schwarzer Knoten.

Repariere (E5): Auf Knoten x fehlt eine schwarze Einheit.

Ziel: „Hole“ eine schwarze Einheit von weiter oben bis:

- x ist rot $\Rightarrow$ mach x schwarz.
- x ist Wurzel $\Rightarrow$ Schwarz-höhe überall 1 niedriger.

Problem wird lokal durch Umfärben & Rotieren gelöst.

Laufzeit $\text{RBDELETE} \in \mathcal{O}(h) + \text{Laufzeit RBDELETEFIXUP}$

RBDELETEFIXUP

Details siehe [CLRS, Kapitel 13.4]

- (E2) Die Wurzel ist schwarz.
- (E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten
dieselbe Anzahl schwarzer Knoten.

Repariere (E5): Auf Knoten x fehlt eine schwarze Einheit.

Ziel: „Hole“ eine schwarze Einheit von weiter oben bis:

- x ist rot $\Rightarrow$ mach x schwarz.
- x ist Wurzel $\Rightarrow$ Schwarz-höhe überall 1 niedriger.

Problem wird lokal durch Umfärben & Rotieren gelöst.

Laufzeit $\text{RBDELETE} \in \mathcal{O}(h) + \underbrace{\text{Laufzeit RBDELETEFIXUP}}_{\mathcal{O}(h)}$

RBDELETEFIXUP

Details siehe [CLRS, Kapitel 13.4]

- (E2) Die Wurzel ist schwarz.
- (E4) Wenn ein Knoten rot ist,
sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten
dieselbe Anzahl schwarzer Knoten.

Repariere (E5): Auf Knoten x fehlt eine schwarze Einheit.

Ziel: „Hole“ eine schwarze Einheit von weiter oben bis:

- x ist rot $\Rightarrow$ mach x schwarz.
- x ist Wurzel $\Rightarrow$ Schwarz-höhe überall 1 niedriger.

Problem wird lokal durch Umfärben & Rotieren gelöst.

$$\text{Laufzeit RBDELETE} \in \mathcal{O}(h) + \underbrace{\text{Laufzeit RBDELETEFIXUP}}_{\mathcal{O}(h)} = \mathcal{O}(h)$$

RBDELETEFIXUP

Details siehe [CLRS, Kapitel 13.4]

- (E2) Die Wurzel ist schwarz.
- (E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Repariere (E5): Auf Knoten x fehlt eine schwarze Einheit.

Ziel: „Hole“ eine schwarze Einheit von weiter oben bis:

- x ist rot $\Rightarrow$ mach x schwarz.
- x ist Wurzel $\Rightarrow$ Schwarz-höhe überall 1 niedriger.

Problem wird lokal durch Umfärben & Rotieren gelöst.

$$\text{Laufzeit RBDELETE} \in \mathcal{O}(h) + \underbrace{\text{Laufzeit RBDELETEFIXUP}}_{\mathcal{O}(h)} = \mathcal{O}(h)$$

RBDELETEFIXUP

Details siehe [CLRS, Kapitel 13.4]

- (E2) Die Wurzel ist schwarz.
- (E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.

Lemma. Ein Rot-Schwarz-Baum T mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Repariere (E5): Auf Knoten x fehlt eine schwarze Einheit.

Ziel: „Hole“ eine schwarze Einheit von weiter oben bis:

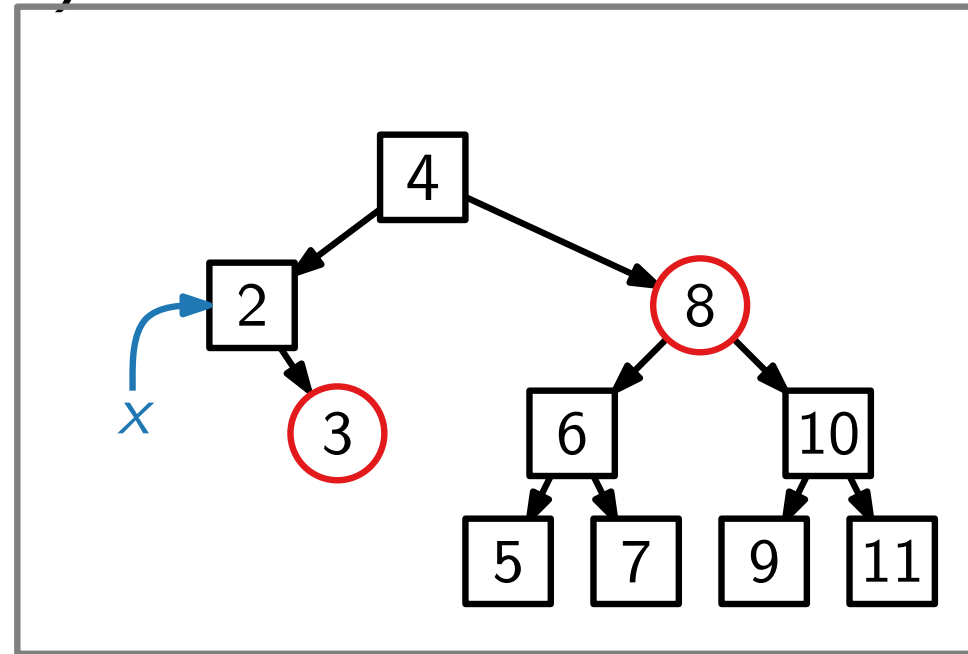
- x ist rot $\Rightarrow$ mach x schwarz.
- x ist Wurzel $\Rightarrow$ Schwarz-höhe überall 1 niedriger.

Problem wird lokal durch Umfärben & Rotieren gelöst.

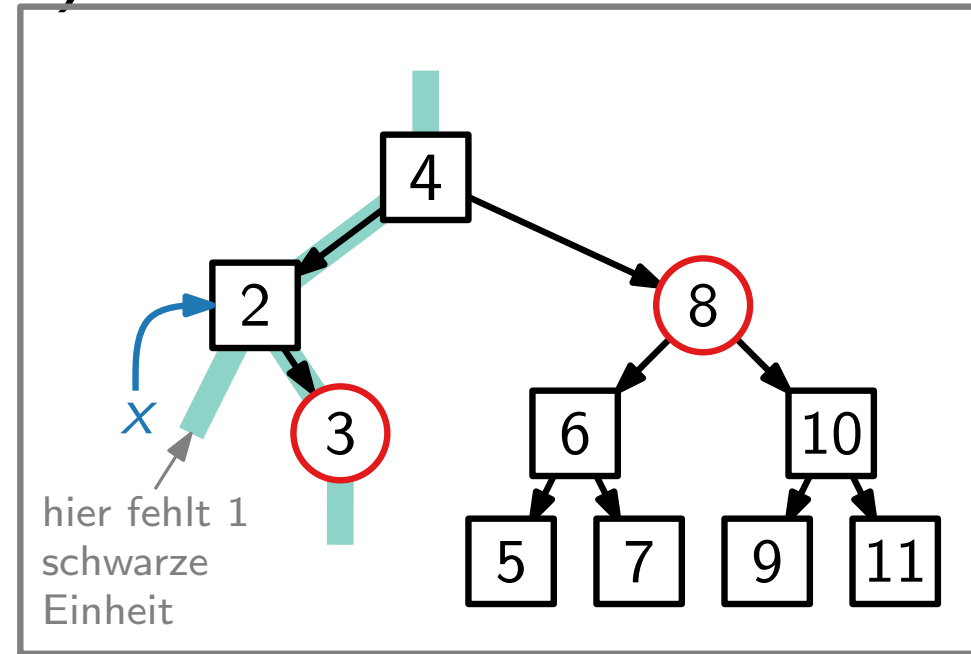
Laufzeit $\text{RBDELETE} \in \mathcal{O}(h) + \underbrace{\text{Laufzeit RBDELETEFIXUP}}_{\mathcal{O}(h)} = \mathcal{O}(h)$

Satz. Rot-Schwarz-Bäume implementieren alle dynamische-Menge-Operationen in $\mathcal{O}(\log n)$ Zeit, wobei n die momentane Anz. der Schlüssel ist.

RBDELETEFIXUP(RBNode x)



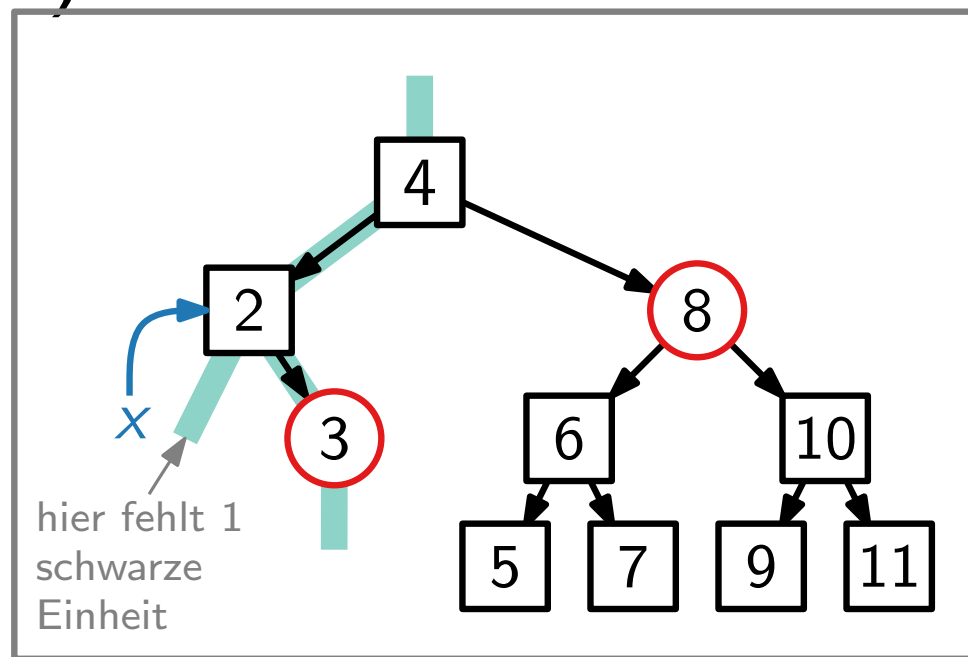
RBDELETEFIXUP(RBNode x)



RBDELETEFIXUP(RBNode x)

```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
```

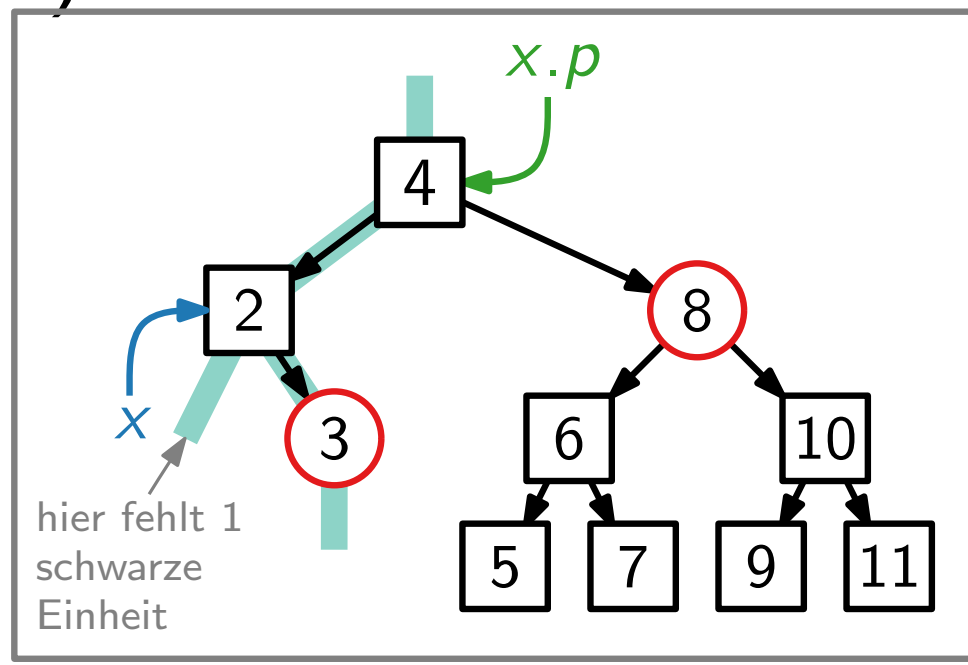
```
 $x.\text{color} = \text{black}$ 
```



RBDELETEFIXUP(RBNode x)

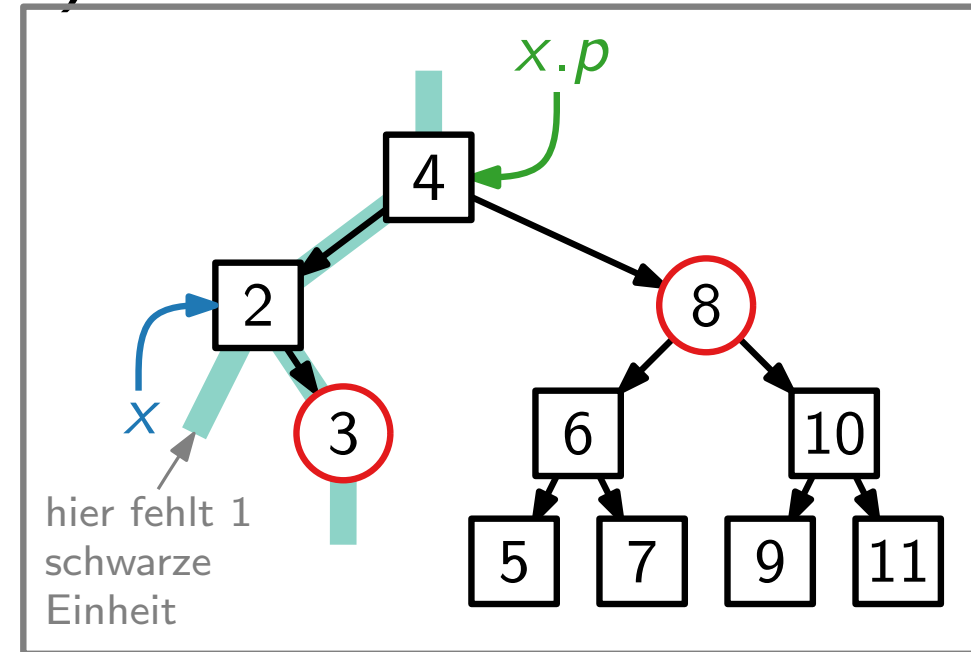
```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
```

```
 $x.\text{color} = \text{black}$ 
```



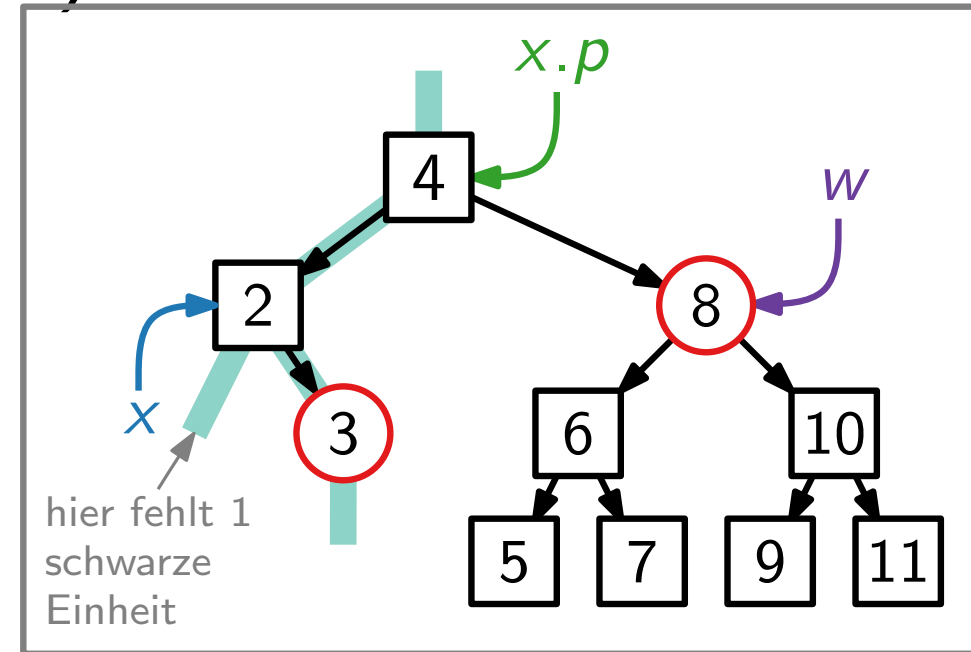
RBDELETEFIXUP(RBNode x)

```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do  
  if  $x == x.p.\text{left}$  then  
      
  else ... // wie oben, aber re. & li. vertauscht  
 $x.\text{color} = \text{black}$ 
```



RBDELETEFIXUP(RBNode x)

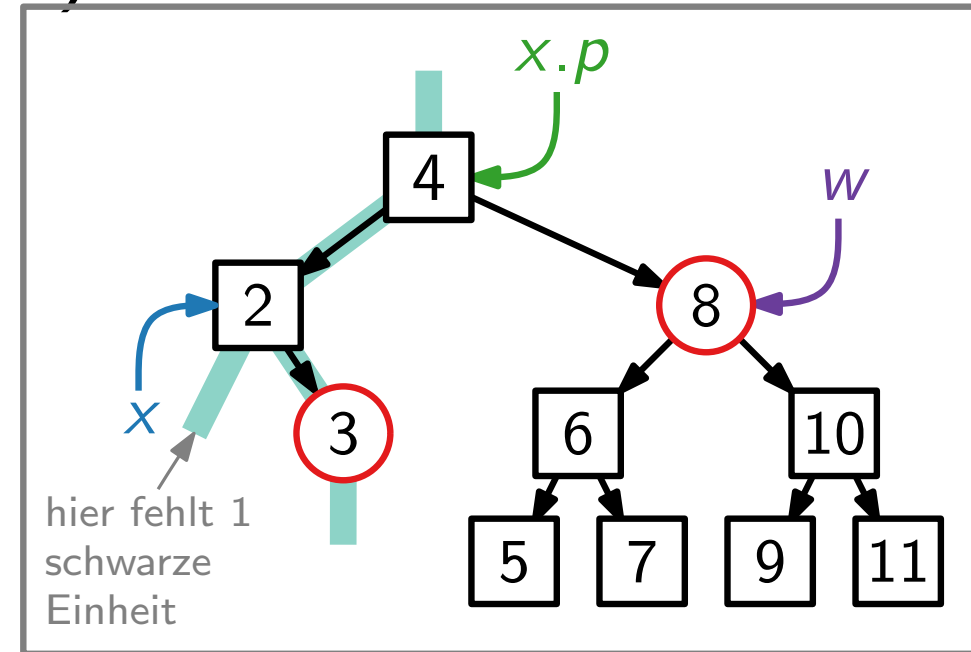
```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do  
  if  $x == x.p.\text{left}$  then  
     $w = x.p.\text{right}$  // Geschwister von  $x$   
  else ... // wie oben, aber re. & li. vertauscht  
 $x.\text{color} = \text{black}$ 
```



RBDELETEFIXUP(RBNode x)

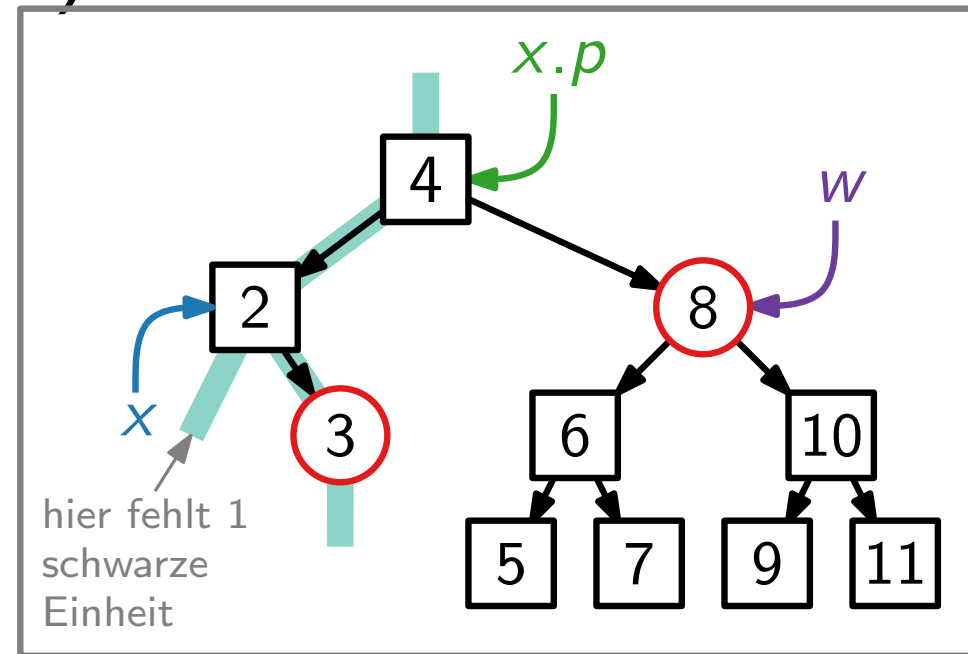
```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
     $w = x.p.\text{right}$  // Geschwister von  $x$ 

    // schwarze Einheit nach oben schieben
  else ... // wie oben, aber re. & li. vertauscht
 $x.\text{color} = \text{black}$ 
```



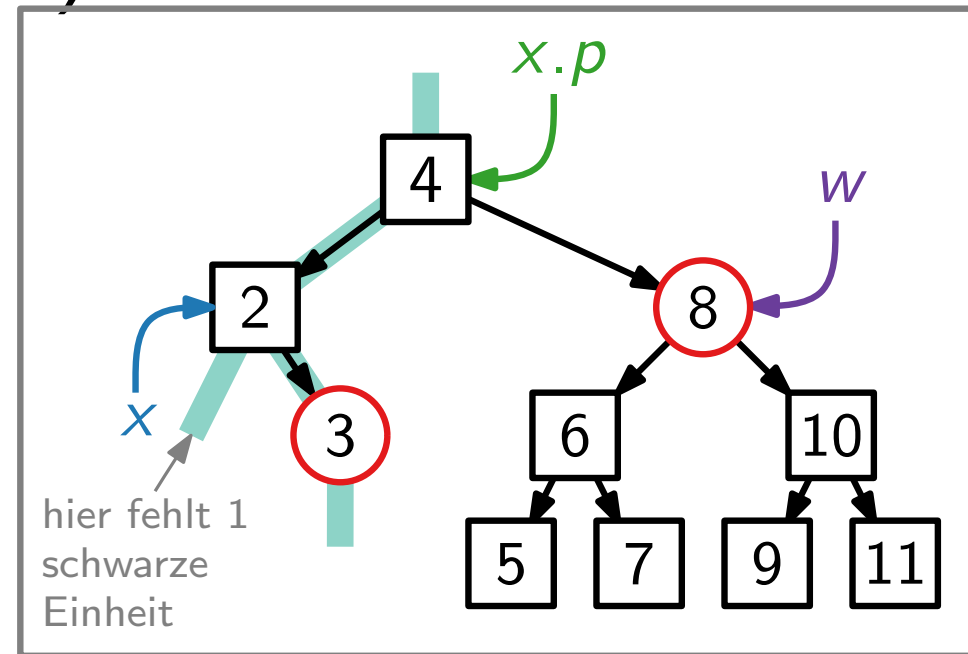
RBDELETEFIXUP(RBNode x)

```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
     $w = x.p.\text{right}$  // Geschwister von  $x$ 
    if  $w.\text{color} == \text{red}$  then
      // schwarze Einheit nach oben schieben
    else ... // wie oben, aber re. & li. vertauscht
   $x.\text{color} = \text{black}$ 
```



RBDELETEFIXUP(RBNode x)

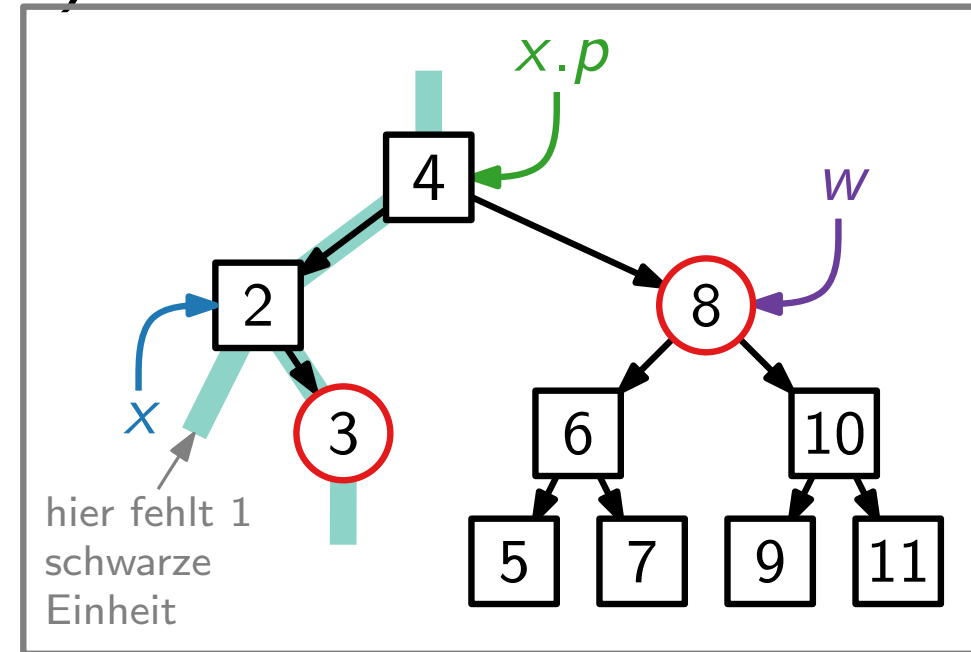
```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
     $w = x.p.\text{right}$  // Geschwister von  $x$ 
    if  $w.\text{color} == \text{red}$  then
      // färbe  $w$  schwarz
    else
      // schwarze Einheit nach oben schieben
    else ... // wie oben, aber re. & li. vertauscht
   $x.\text{color} = \text{black}$ 
```



RBDELETEFIXUP(RBNode x)

```

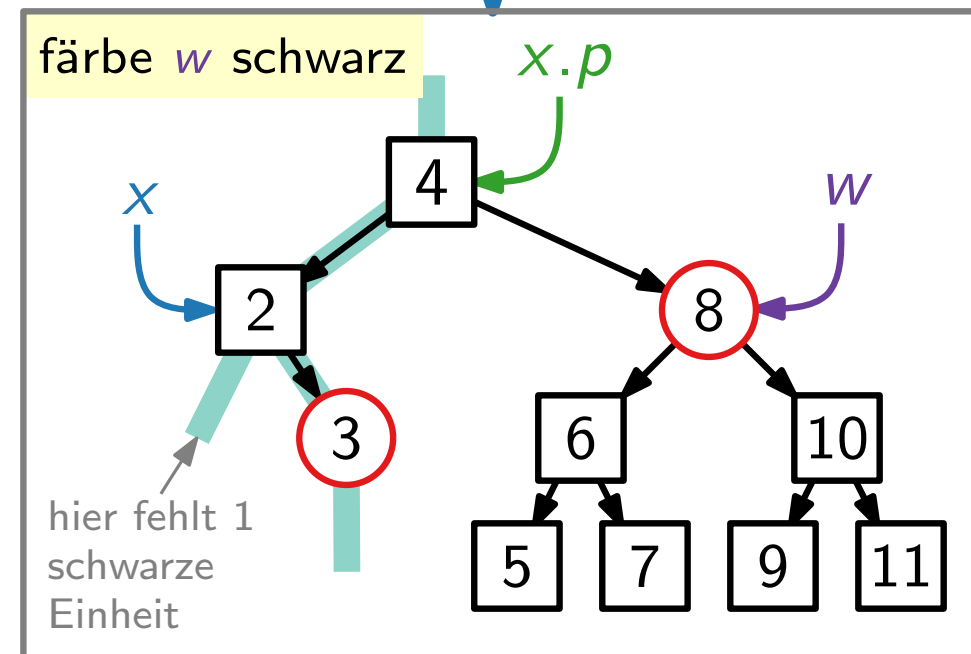
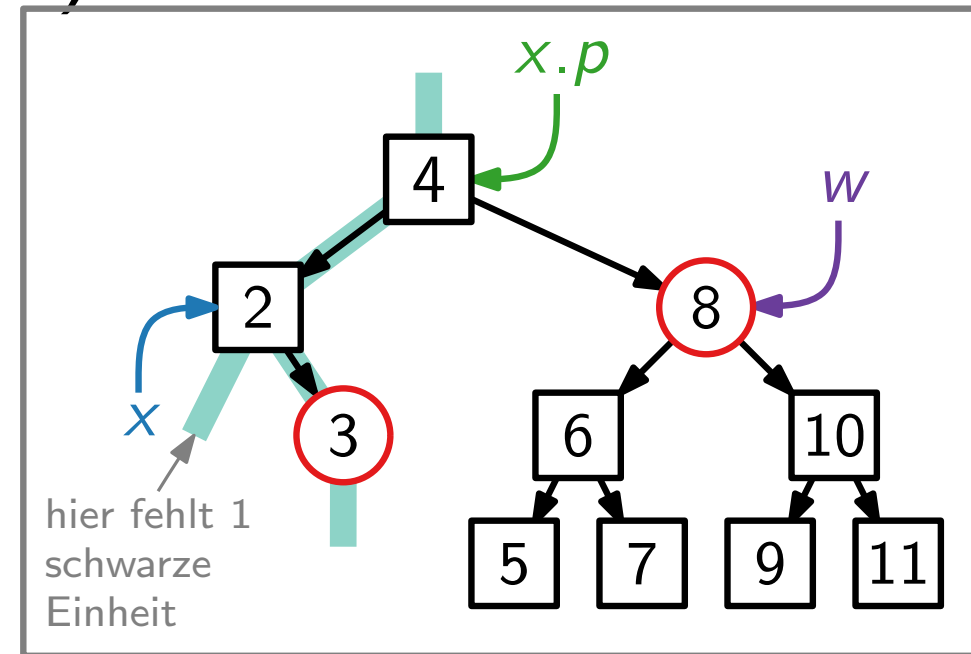
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
     $w = x.p.\text{right}$  // Geschwister von  $x$ 
    if  $w.\text{color} == \text{red}$  then
       $x.p.\text{color} = \text{red}$ 
       $w.\text{color} = \text{black}$  // färbe  $w$  schwarz
      LEFTROTATE( $x.p$ )
       $w = x.p.\text{right}$ 
    // schwarze Einheit nach oben schieben
  else ... // wie oben, aber re. & li. vertauscht
 $x.\text{color} = \text{black}$ 
  
```



RBDELETEFIXUP(RBNode x)

```

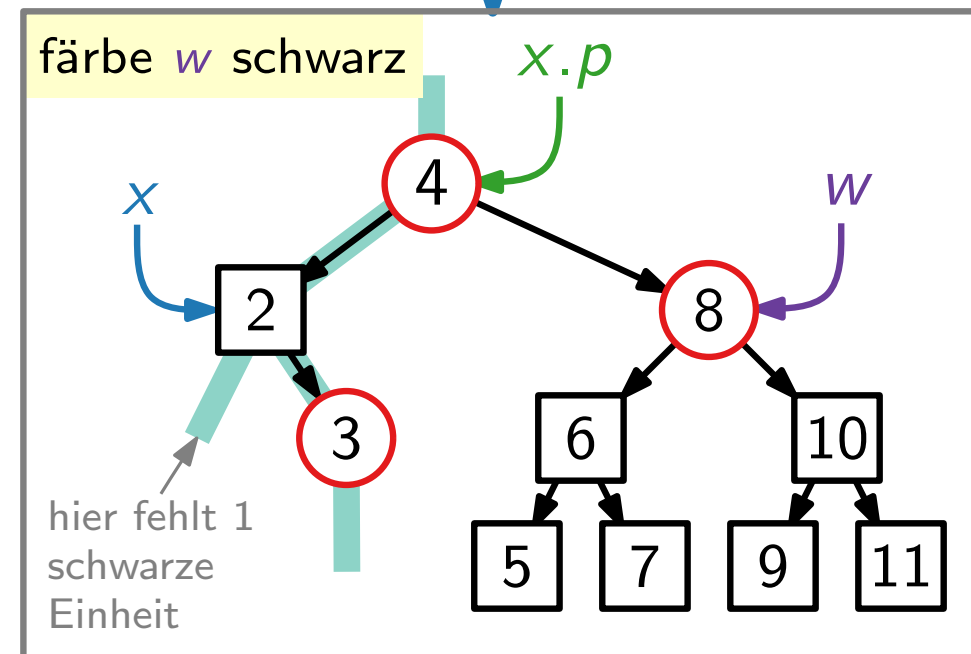
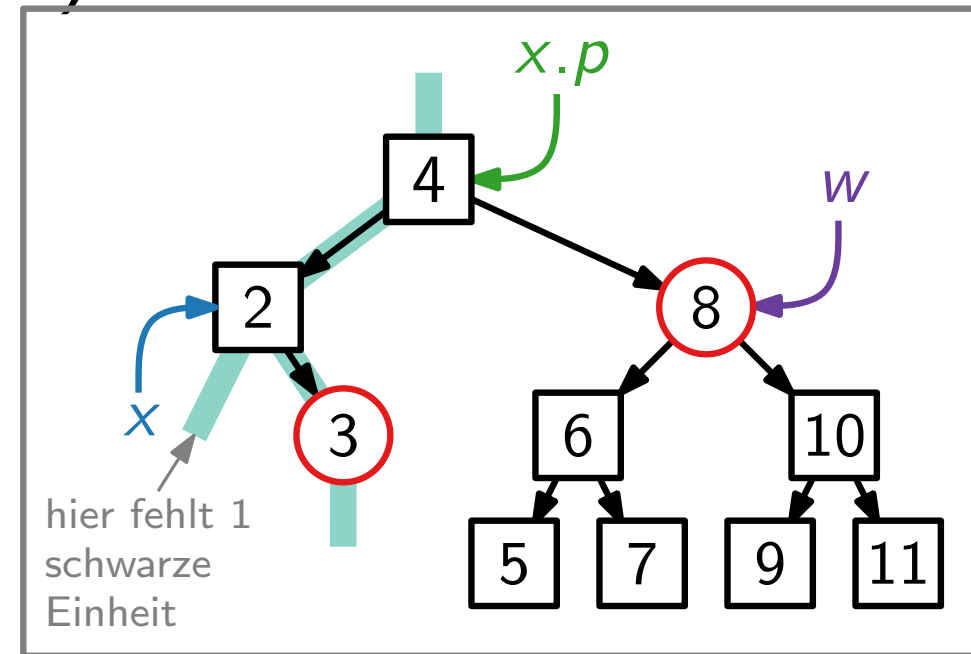
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
     $w = x.p.\text{right}$  // Geschwister von  $x$ 
    if  $w.\text{color} == \text{red}$  then
       $x.p.\text{color} = \text{red}$ 
       $w.\text{color} = \text{black}$  // färbe  $w$  schwarz
      LEFTROTATE( $x.p$ )
       $w = x.p.\text{right}$ 
      // schwarze Einheit nach oben schieben
    else ... // wie oben, aber re. & li. vertauscht
   $x.\text{color} = \text{black}$ 
  
```



RBDELETEFIXUP(RBNode x)

```

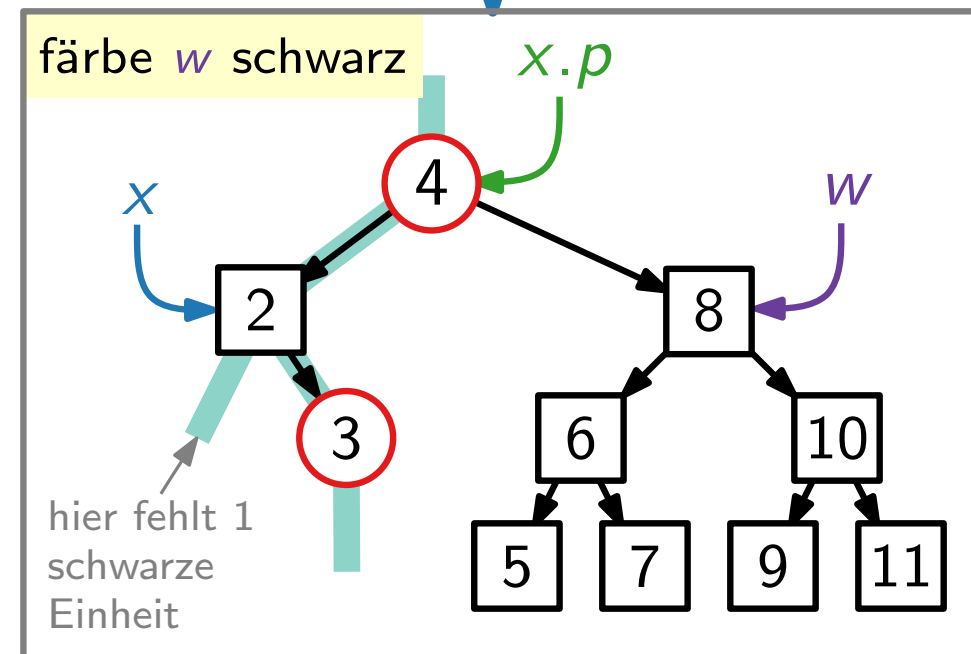
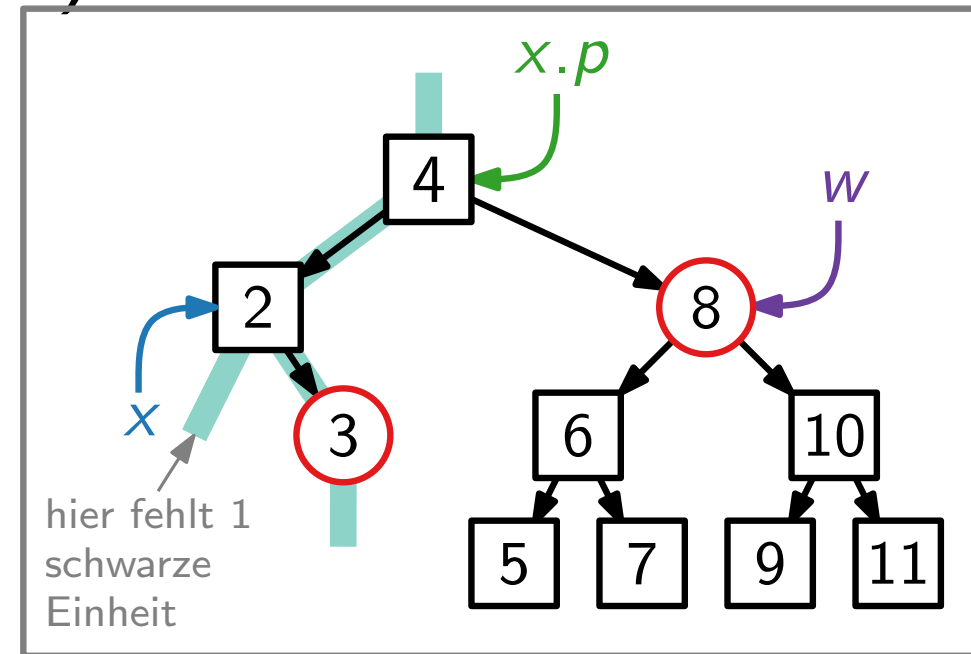
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
     $w = x.p.\text{right}$  // Geschwister von  $x$ 
    if  $w.\text{color} == \text{red}$  then
       $x.p.\text{color} = \text{red}$ 
       $w.\text{color} = \text{black}$  // färbe  $w$  schwarz
      LEFTROTATE( $x.p$ )
       $w = x.p.\text{right}$ 
      // schwarze Einheit nach oben schieben
    else ... // wie oben, aber re. & li. vertauscht
   $x.\text{color} = \text{black}$ 
  
```



RBDELETEFIXUP(RBNode x)

```

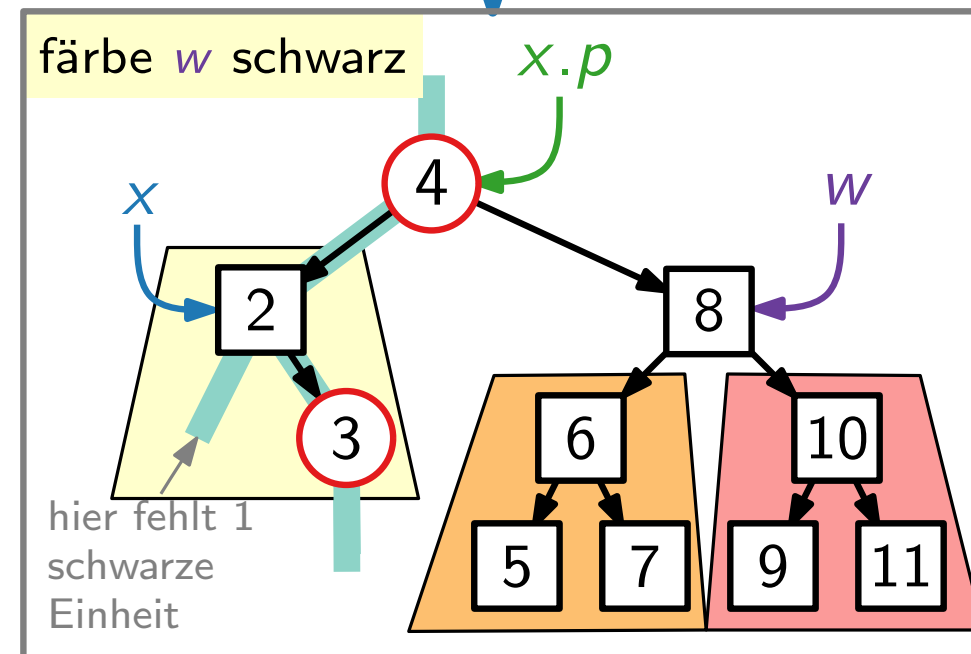
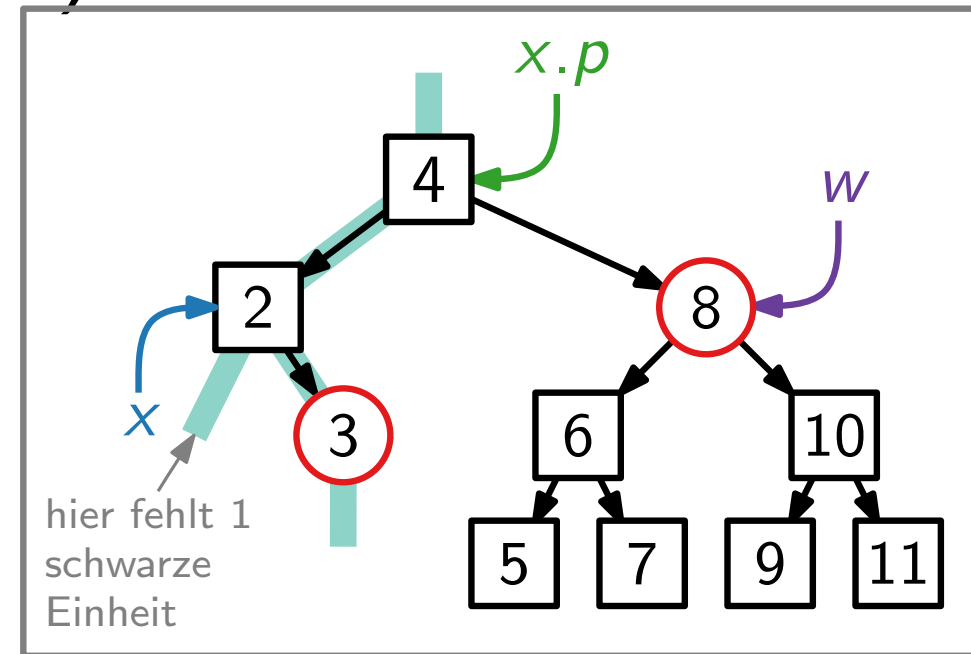
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
     $w = x.p.\text{right}$  // Geschwister von  $x$ 
    if  $w.\text{color} == \text{red}$  then
       $x.p.\text{color} = \text{red}$ 
       $w.\text{color} = \text{black}$  // färbe  $w$  schwarz
      LEFTROTATE( $x.p$ )
       $w = x.p.\text{right}$ 
      // schwarze Einheit nach oben schieben
    else ... // wie oben, aber re. & li. vertauscht
   $x.\text{color} = \text{black}$ 
  
```



RBDELETEFIXUP(RBNode x)

```

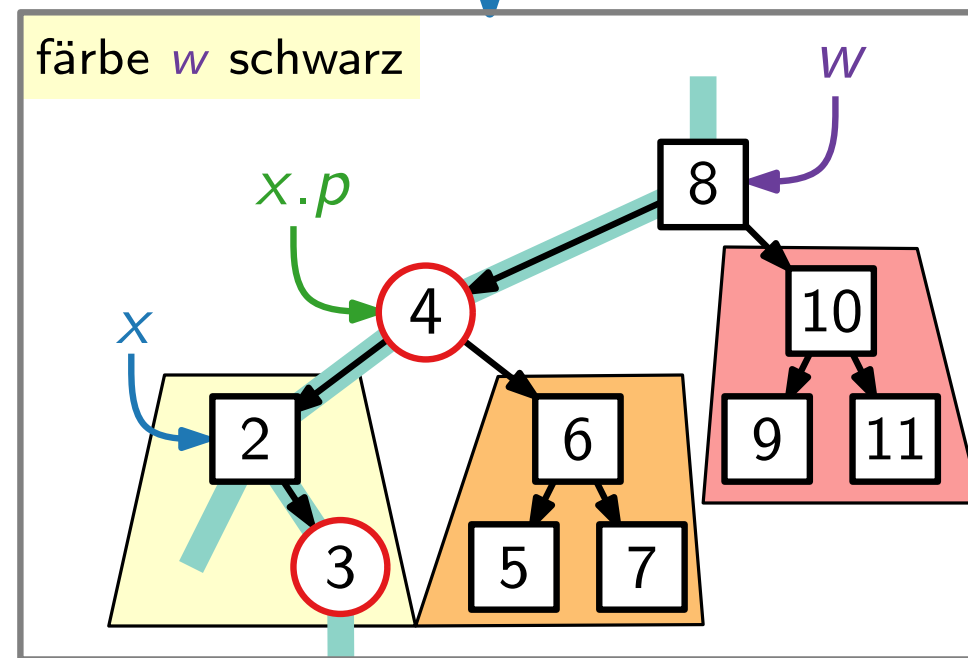
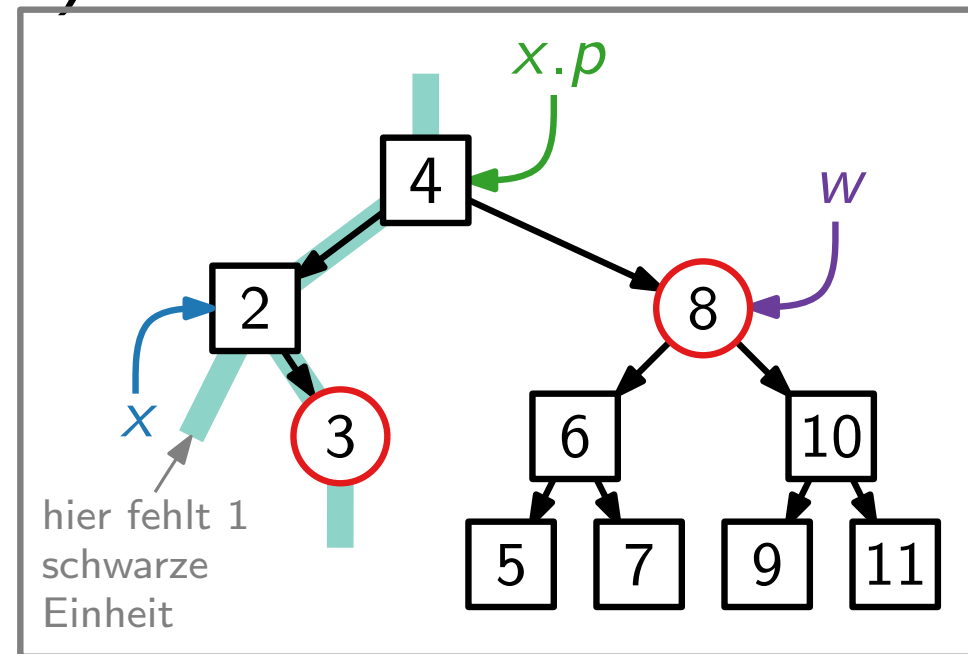
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
     $w = x.p.\text{right}$  // Geschwister von  $x$ 
    if  $w.\text{color} == \text{red}$  then
       $x.p.\text{color} = \text{red}$ 
       $w.\text{color} = \text{black}$  // färbe  $w$  schwarz
      LEFTROTATE( $x.p$ )
       $w = x.p.\text{right}$ 
      // schwarze Einheit nach oben schieben
    else ... // wie oben, aber re. & li. vertauscht
   $x.\text{color} = \text{black}$ 
  
```



RBDELETEFIXUP(RBNode x)

```

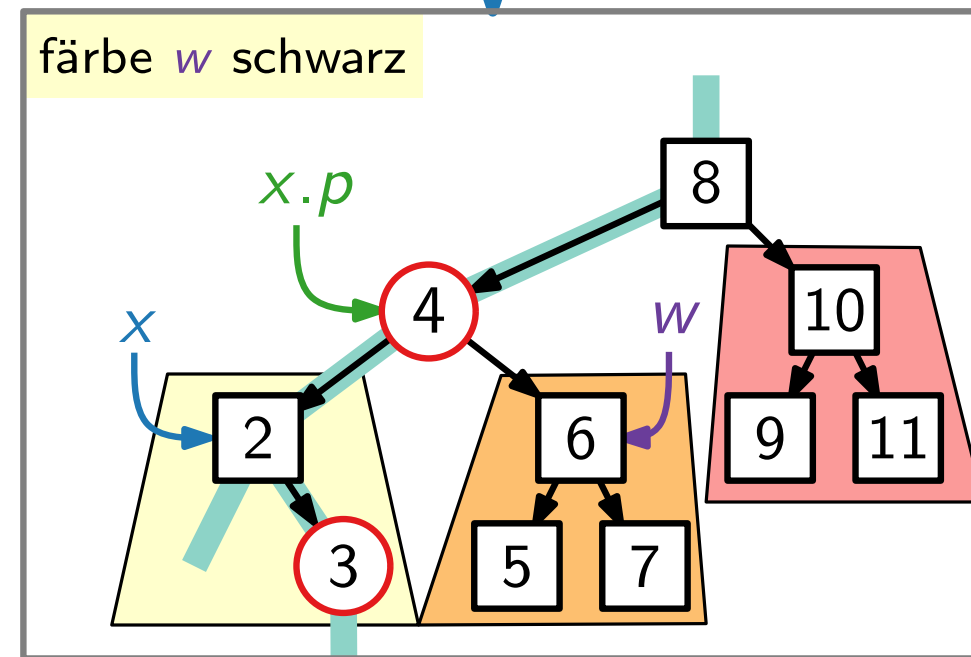
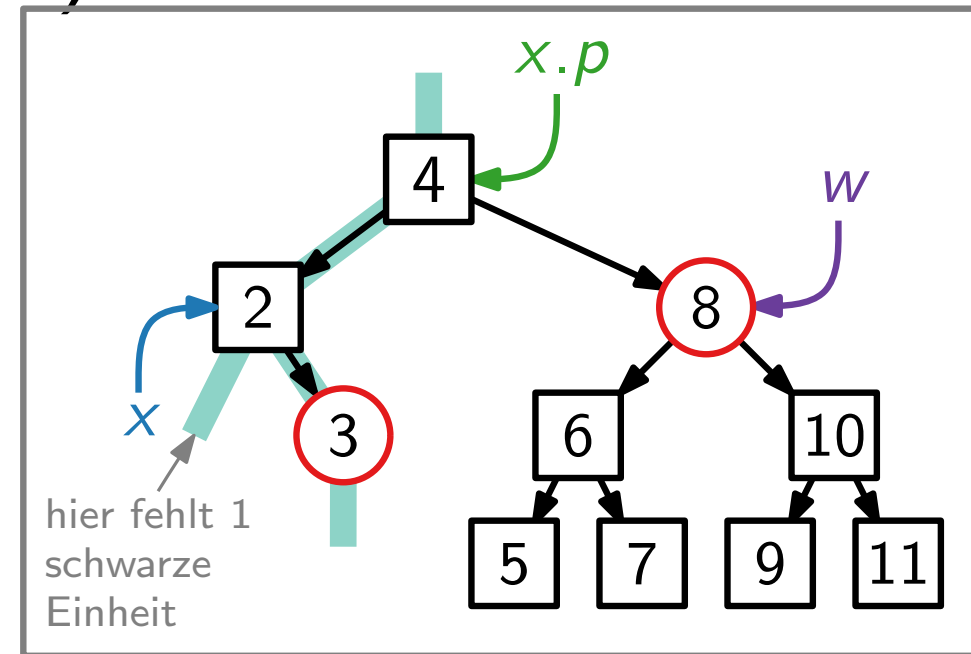
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
     $w = x.p.\text{right}$  // Geschwister von  $x$ 
    if  $w.\text{color} == \text{red}$  then
       $x.p.\text{color} = \text{red}$ 
       $w.\text{color} = \text{black}$  // färbe  $w$  schwarz
      LEFTROTATE( $x.p$ )
       $w = x.p.\text{right}$ 
      // schwarze Einheit nach oben schieben
    else ... // wie oben, aber re. & li. vertauscht
   $x.\text{color} = \text{black}$ 
  
```



RBDELETEFIXUP(RBNode x)

```

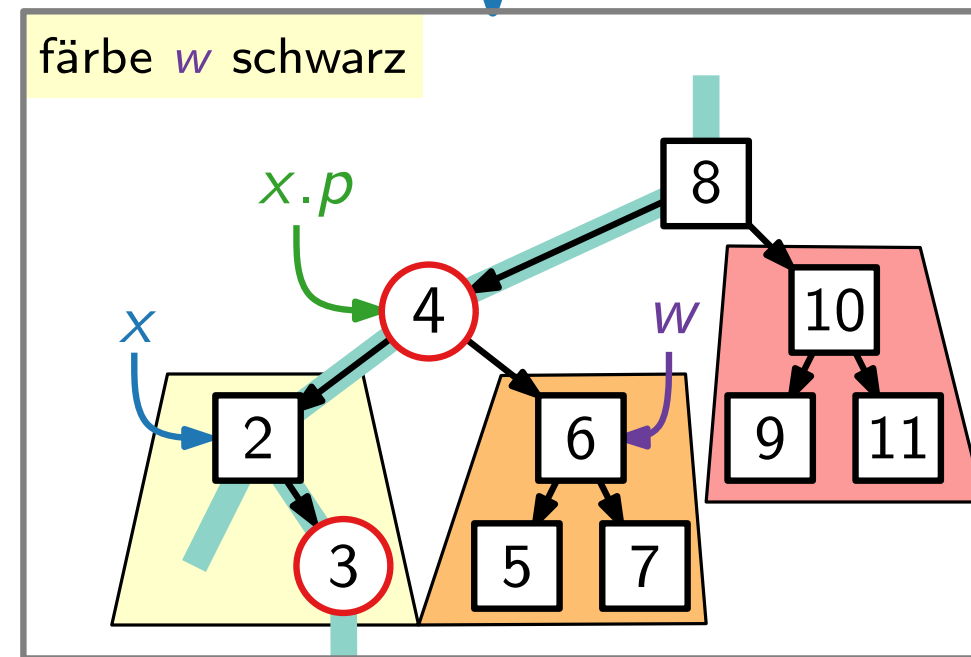
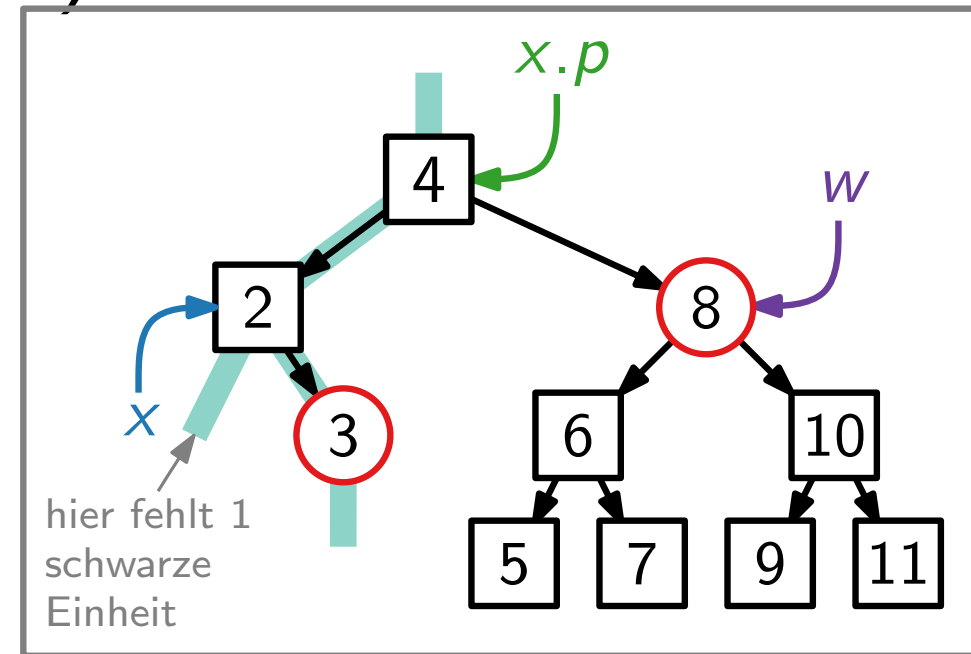
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
     $w = x.p.\text{right}$  // Geschwister von  $x$ 
    if  $w.\text{color} == \text{red}$  then
       $x.p.\text{color} = \text{red}$ 
       $w.\text{color} = \text{black}$  // färbe  $w$  schwarz
      LEFTROTATE( $x.p$ )
       $w = x.p.\text{right}$ 
      // schwarze Einheit nach oben schieben
    else ... // wie oben, aber re. & li. vertauscht
   $x.\text{color} = \text{black}$ 
  
```



RBDELETEFIXUP(RBNode x)

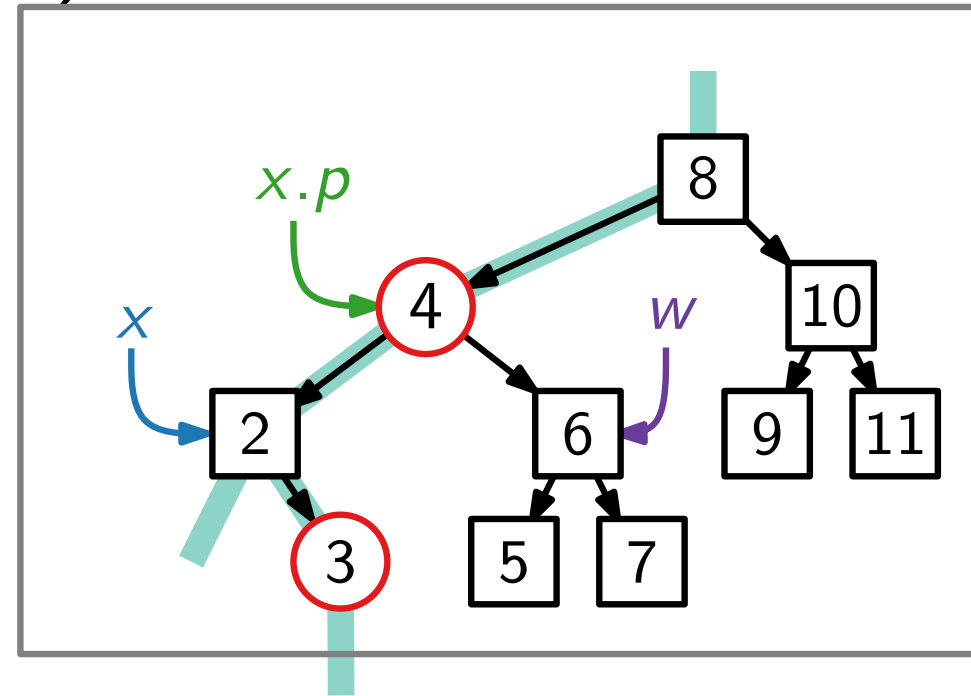
```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
     $w = x.p.\text{right}$  // Geschwister von  $x$ 
    if  $w.\text{color} == \text{red}$  then
       $x.p.\text{color} = \text{red}$ 
       $w.\text{color} = \text{black}$  // färbe  $w$  schwarz
      LEFTROTATE( $x.p$ )
       $w = x.p.\text{right}$ 
      // schwarze Einheit nach oben schieben
    else ... // wie oben, aber re. & li. vertauscht
   $x.\text{color} = \text{black}$ 
  
```



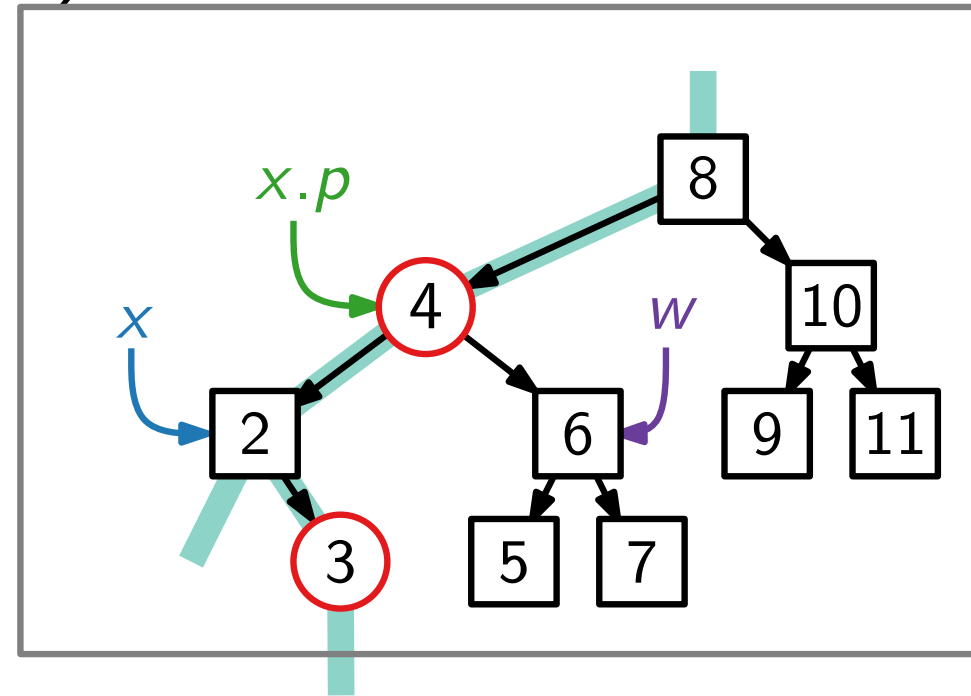
RBDELETEFIXUP(RBNode x)

```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do  
  if  $x == x.p.\text{left}$  then  
    ...  
  else ... // wie oben, aber re. & li. vertauscht  
 $x.\text{color} = \text{black}$ 
```



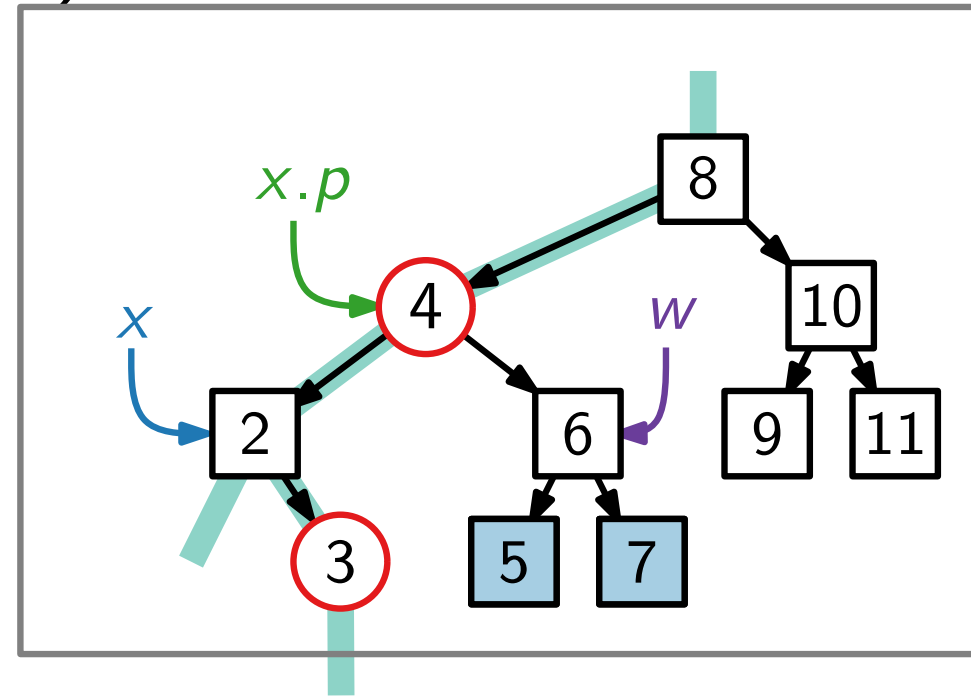
RBDELETEFIXUP(RBNode x)

```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
    ...
    if  $w.\text{left}.\text{color} == \text{black}$  and
        $w.\text{right}.\text{color} == \text{black}$  then
      ...
    else
      ...
  else ... // wie oben, aber re. & li. vertauscht
 $x.\text{color} = \text{black}$ 
```



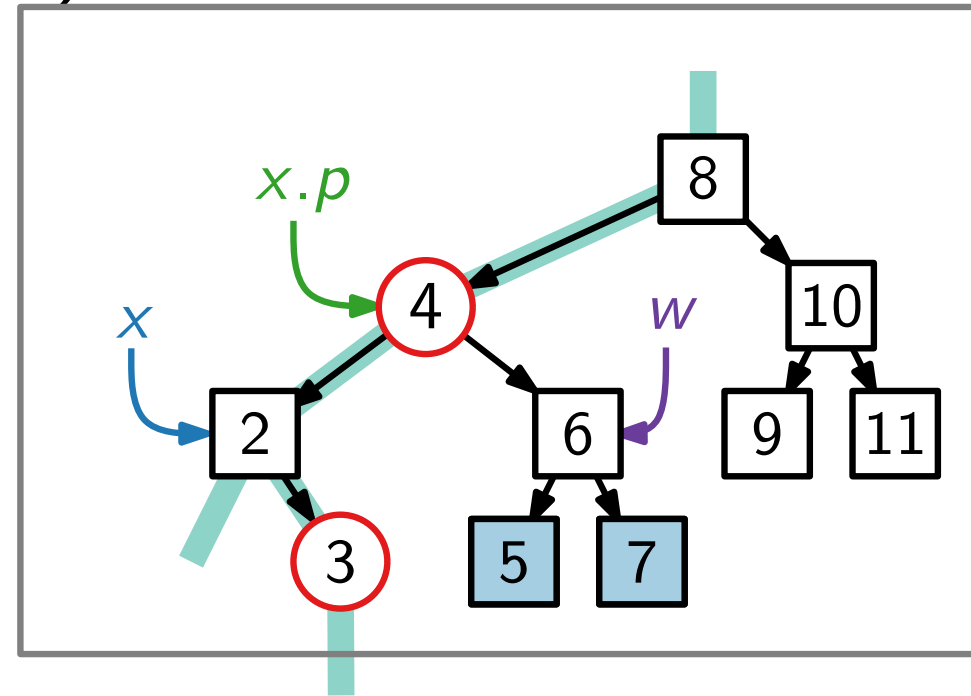
RBDELETEFIXUP(RBNode x)

```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
    ...
    if  $w.\text{left}.\text{color} == \text{black}$  and
        $w.\text{right}.\text{color} == \text{black}$  then
      ...
    else
      ...
  else ... // wie oben, aber re. & li. vertauscht
 $x.\text{color} = \text{black}$ 
```



RBDELETEFIXUP(RBNode x)

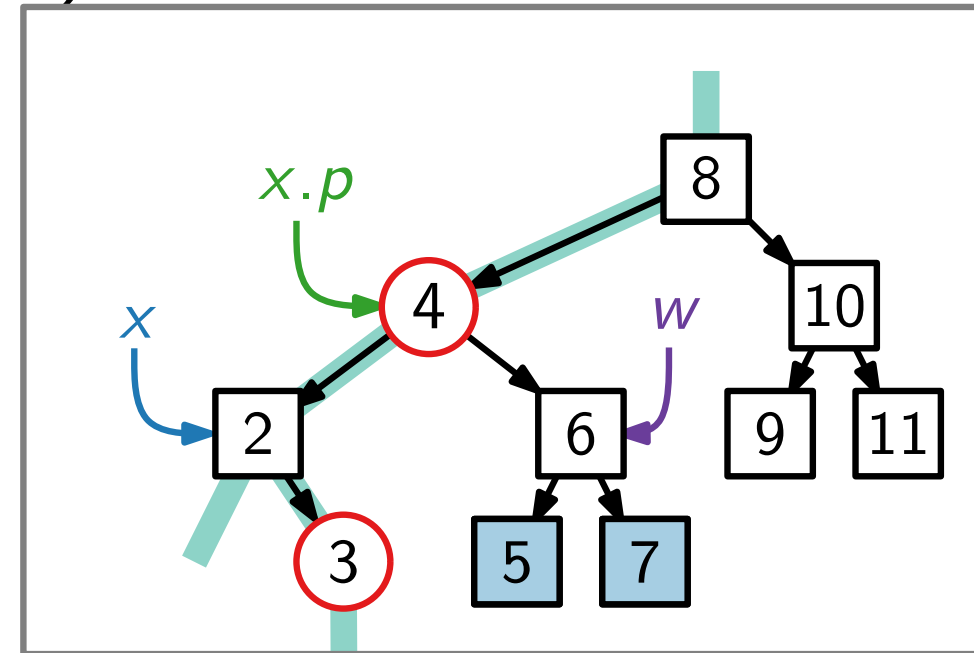
```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
    ...
    if  $w.\text{left}.\text{color} == \text{black}$  and
        $w.\text{right}.\text{color} == \text{black}$  then
       $w.\text{color} = \text{red}$ 
       $x = x.p$ 
    else
      ...
  else ... // wie oben, aber re. & li. vertauscht
 $x.\text{color} = \text{black}$ 
```



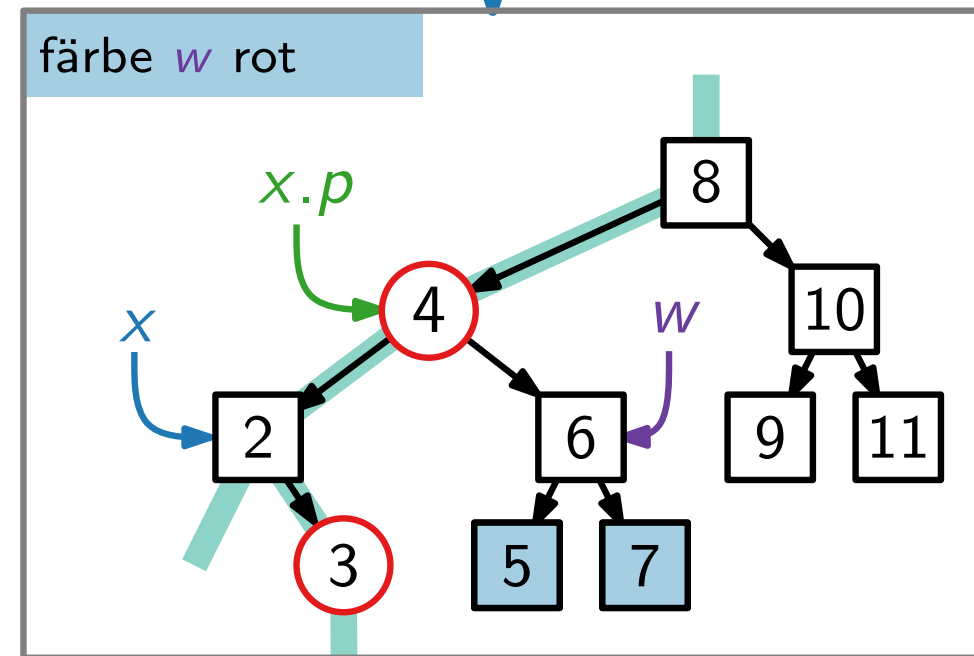
RBDELETEFIXUP(RBNode x)

```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
    ...
    if  $w.\text{left}.\text{color} == \text{black}$  and
        $w.\text{right}.\text{color} == \text{black}$  then
       $w.\text{color} = \text{red}$ 
       $x = x.p$ 
    else
      ...
  else ... // wie oben, aber re. & li. vertauscht
 $x.\text{color} = \text{black}$ 
  
```



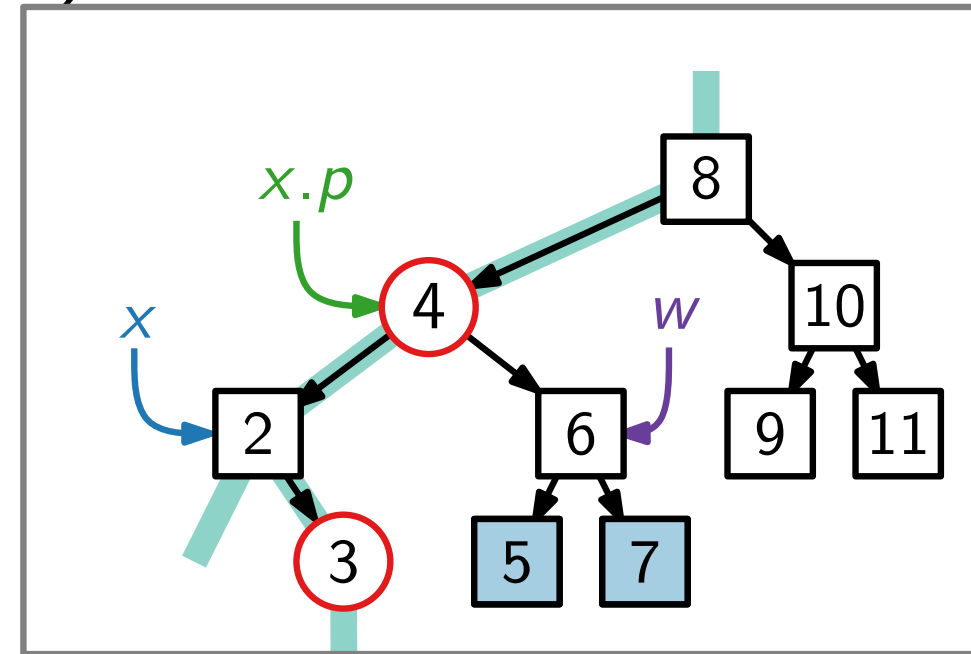
färbe w rot



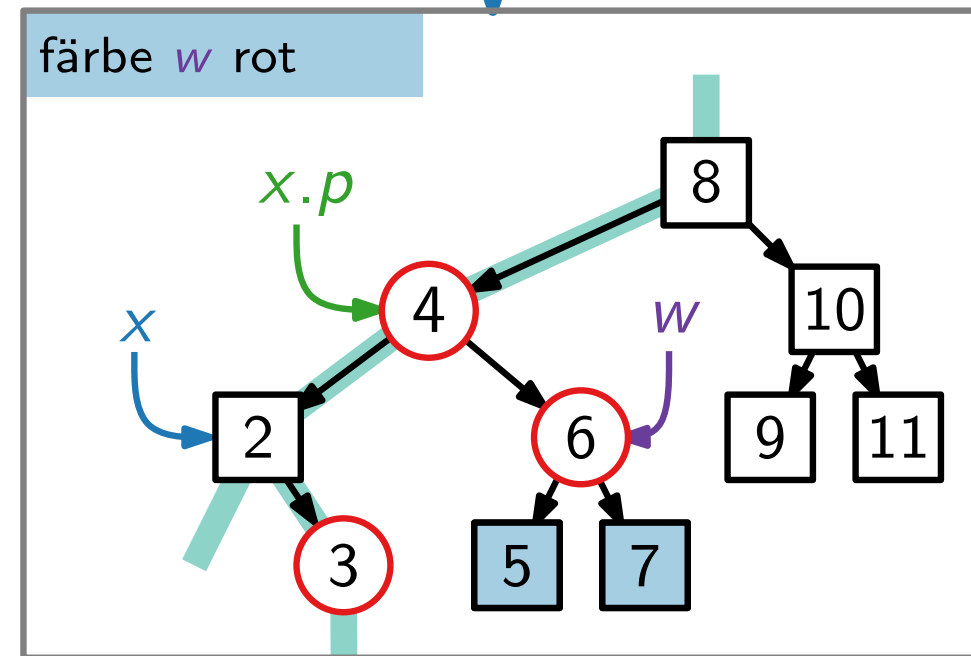
RBDELETEFIXUP(RBNode x)

```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
    ...
    if  $w.\text{left}.\text{color} == \text{black}$  and
        $w.\text{right}.\text{color} == \text{black}$  then
       $w.\text{color} = \text{red}$ 
       $x = x.p$ 
    else
      ...
  else ... // wie oben, aber re. & li. vertauscht
 $x.\text{color} = \text{black}$ 
  
```



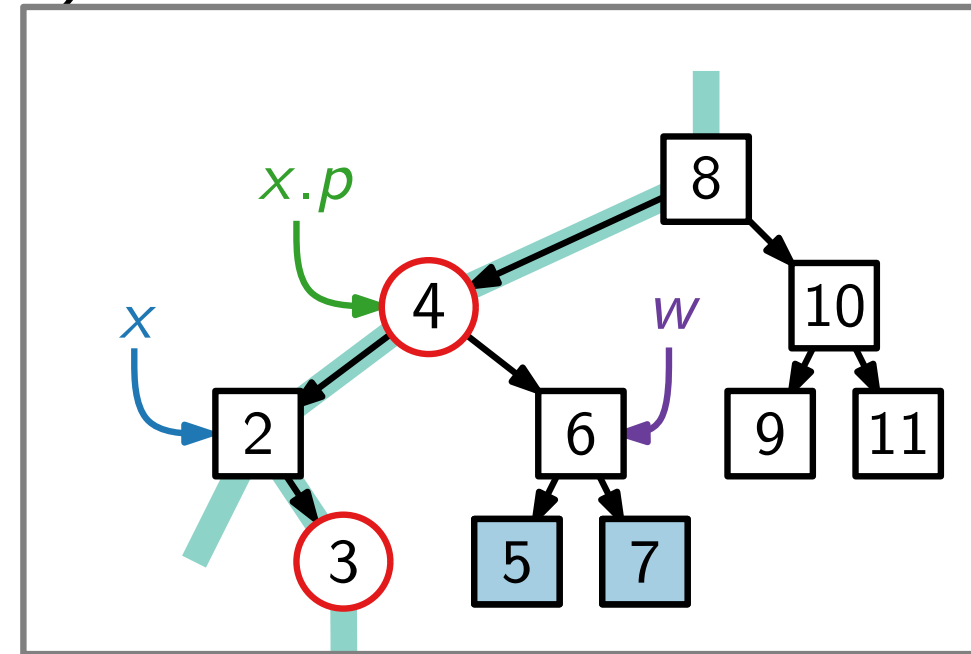
färbe w rot



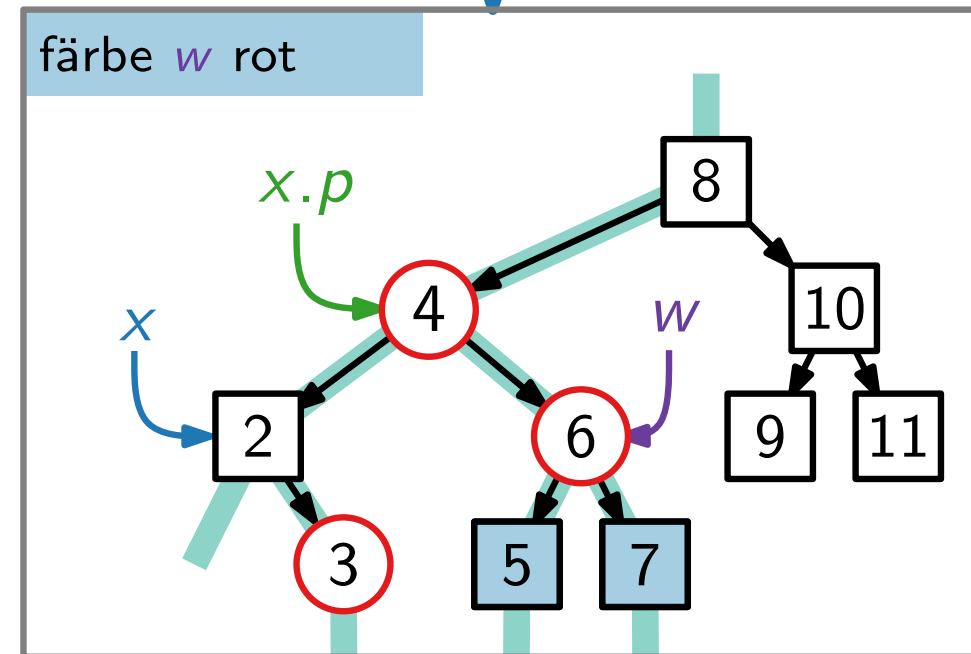
RBDELETEFIXUP(RBNode x)

```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
    ...
    if  $w.\text{left}.\text{color} == \text{black}$  and
        $w.\text{right}.\text{color} == \text{black}$  then
       $w.\text{color} = \text{red}$ 
       $x = x.p$ 
    else
      ...
  else ... // wie oben, aber re. & li. vertauscht
 $x.\text{color} = \text{black}$ 
  
```



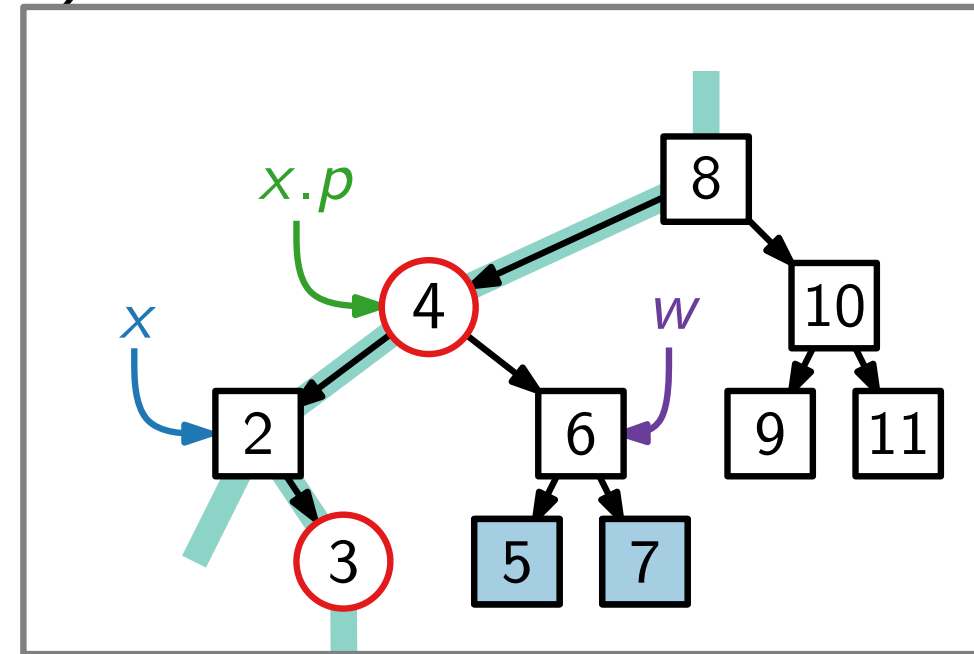
färbe w rot



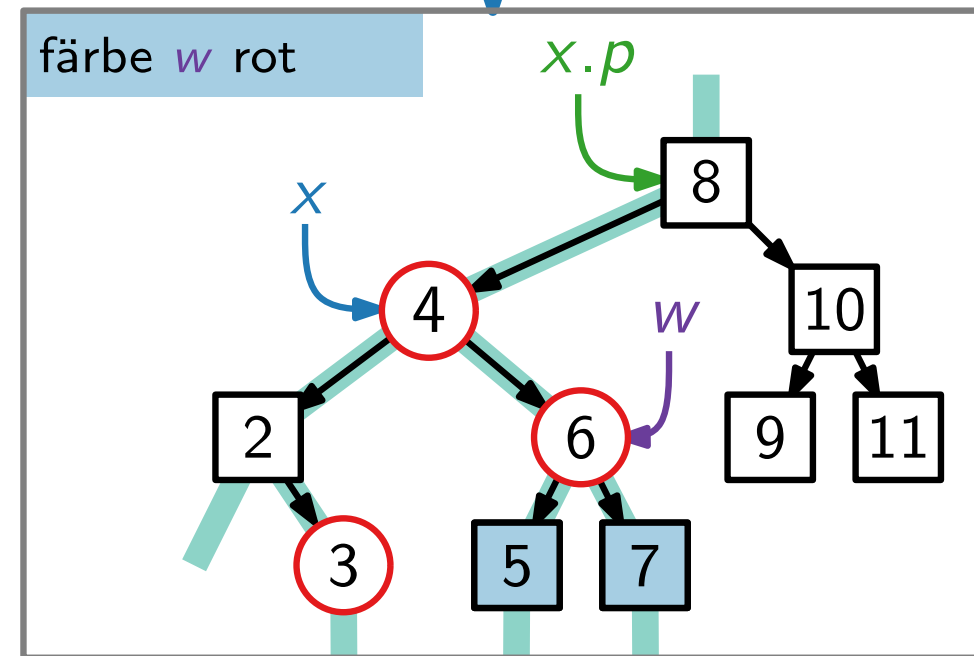
RBDELETEFIXUP(RBNode x)

```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
    ...
    if  $w.\text{left}.\text{color} == \text{black}$  and
        $w.\text{right}.\text{color} == \text{black}$  then
       $w.\text{color} = \text{red}$ 
       $x = x.p$ 
    else
      ...
  else ... // wie oben, aber re. & li. vertauscht
 $x.\text{color} = \text{black}$ 
  
```



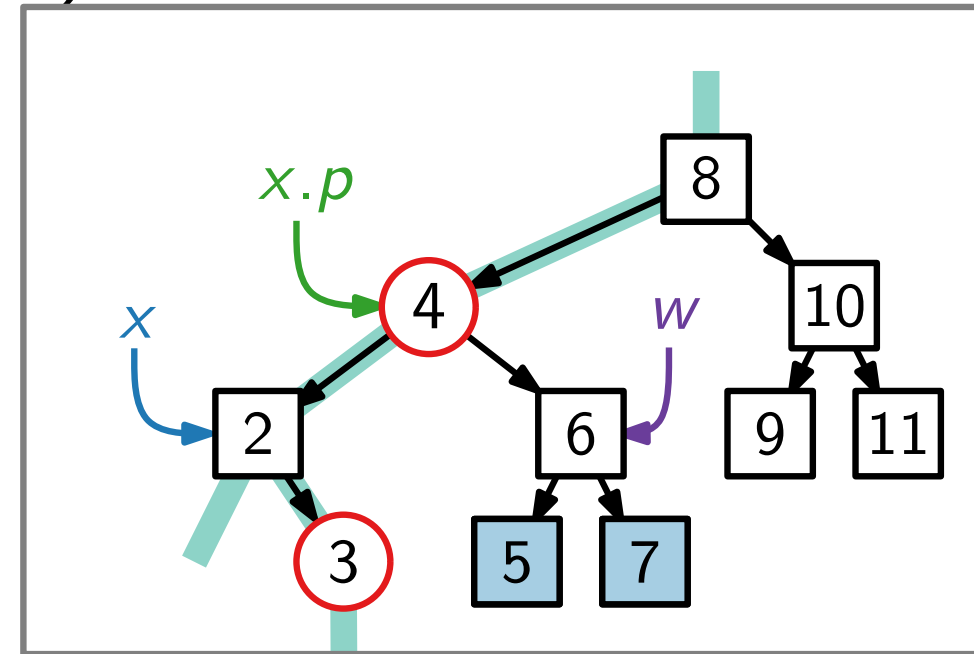
färbe w rot



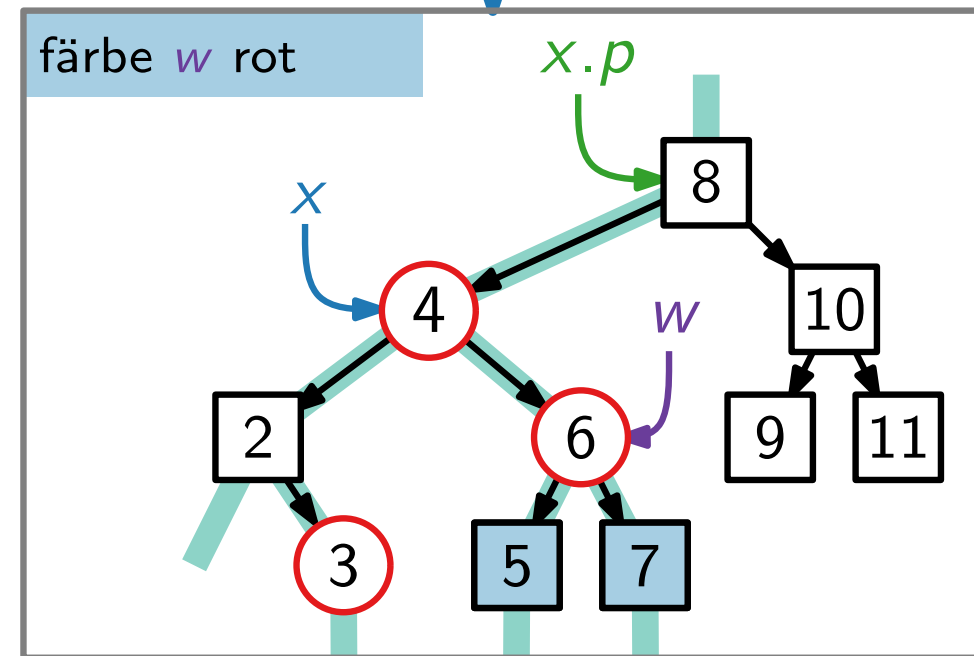
RBDELETEFIXUP(RBNode x)

```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
    ...
    if  $w.\text{left}.\text{color} == \text{black}$  and
        $w.\text{right}.\text{color} == \text{black}$  then
       $w.\text{color} = \text{red}$ 
       $x = x.p$ 
    else
      ...
  else ... // wie oben, aber re. & li. vertauscht
 $x.\text{color} = \text{black}$ 
  
```



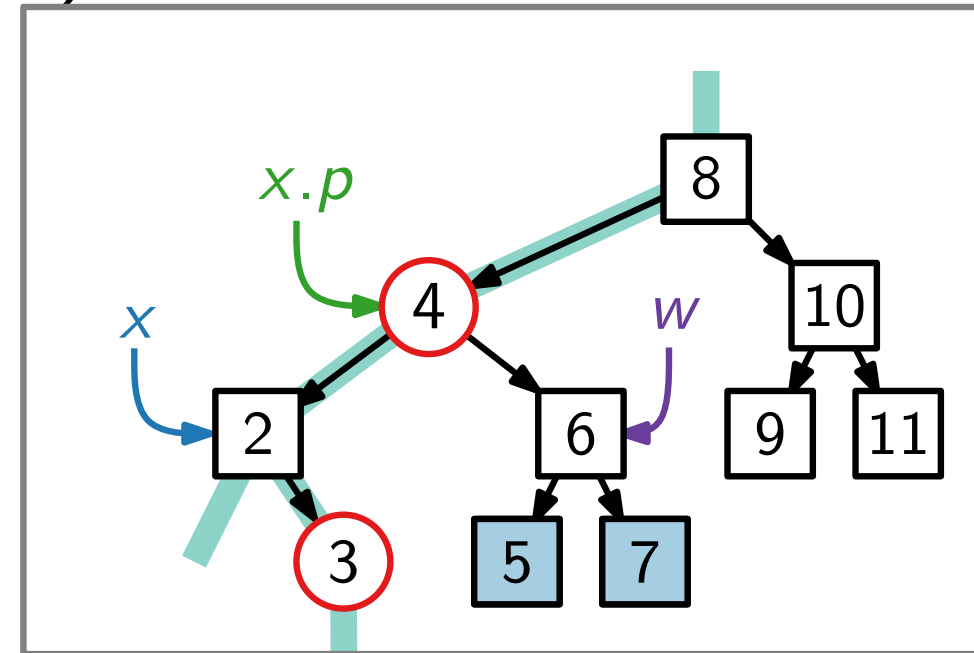
färbe w rot



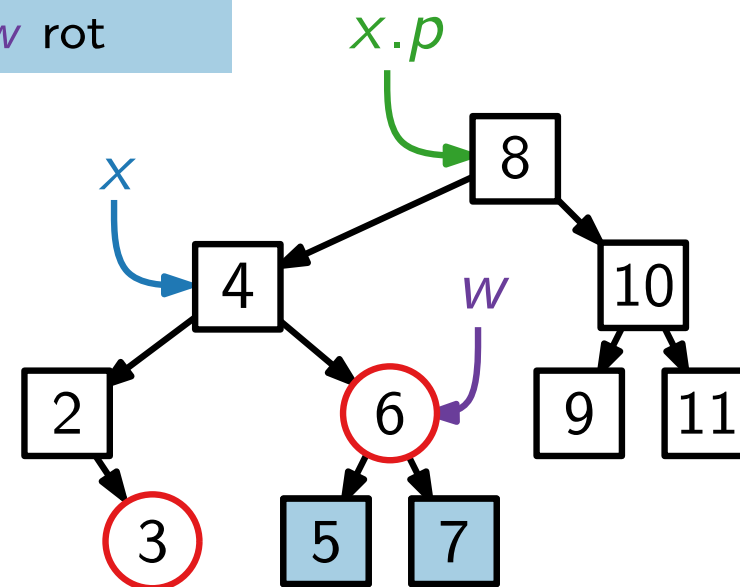
RBDELETEFIXUP(RBNode x)

```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
    ...
    if  $w.\text{left}.\text{color} == \text{black}$  and
        $w.\text{right}.\text{color} == \text{black}$  then
       $w.\text{color} = \text{red}$ 
       $x = x.p$ 
    else
      ...
  else ... // wie oben, aber re. & li. vertauscht
 $x.\text{color} = \text{black}$ 
  
```



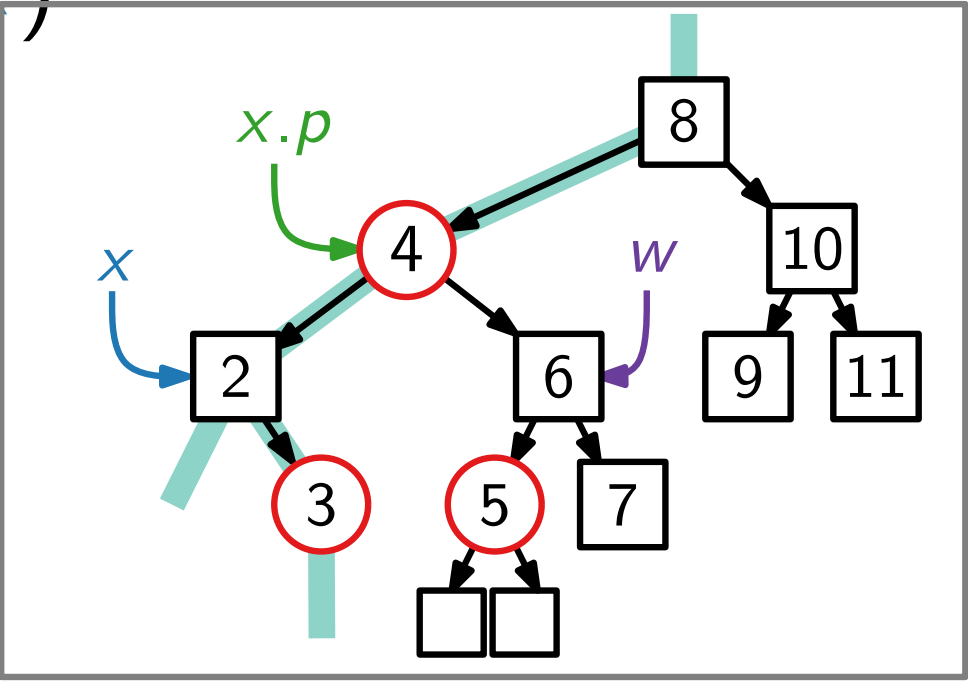
färbe w rot



RBDELETEFIXUP(RBNode x)

```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
    ...
  else
    ...
  else ... // wie oben, aber re. & li. vertauscht
 $x.\text{color} = \text{black}$ 
  
```

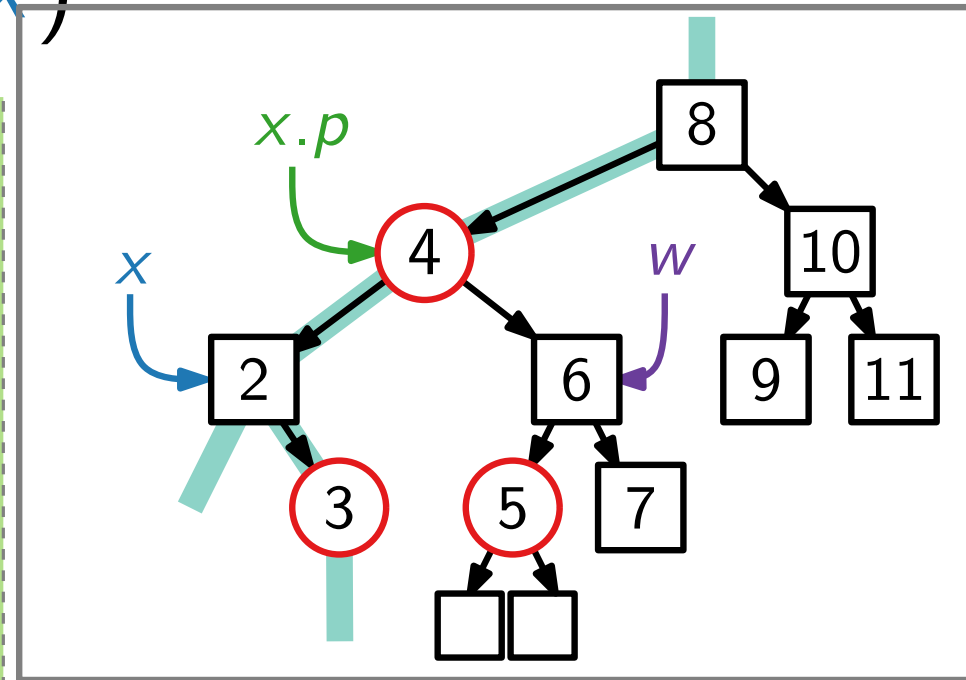


RBDELETEFIXUP(RBNode x)

```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
    ...
  else
    if  $w.\text{right}.\text{color} == \text{black}$  then
      ...
    else ... // wie oben, aber re. & li. vertauscht
   $x.\text{color} = \text{black}$ 

```

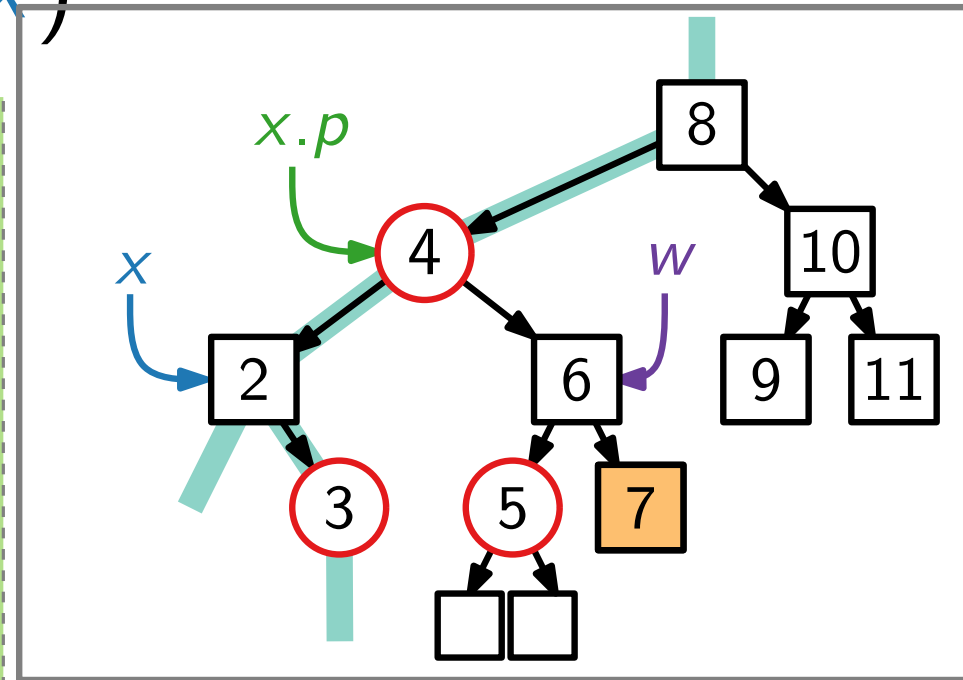


RBDELETEFIXUP(RBNode x)

```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
    ...
  else
    if  $w.\text{right}.\text{color} == \text{black}$  then
      ...
    else ... // wie oben, aber re. & li. vertauscht
   $x.\text{color} = \text{black}$ 

```



RBDELETEFIXUP(RBNode x)

```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
```

```
  if  $x == x.p.\text{left}$  then
```

```
    ...
```

```
  else
```

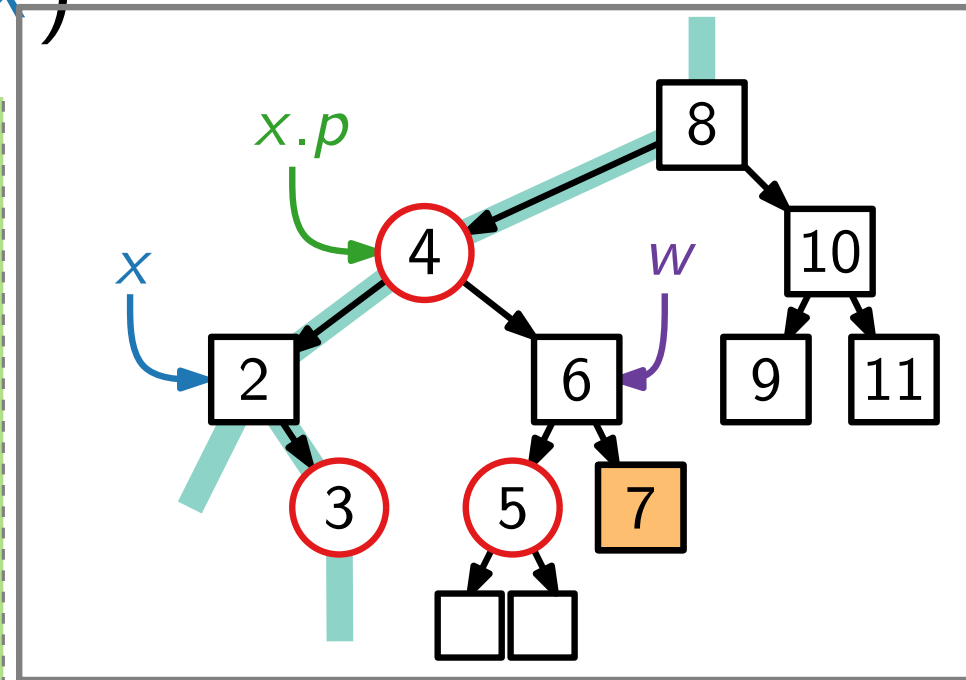
```
    if  $w.\text{right}.\text{color} == \text{black}$  then
```

```
       $w.\text{left}.\text{color} = \text{black}$ 
```

```
       $w.\text{color} = \text{red}$ 
```

```
  else ... // wie oben, aber re. & li. vertauscht
```

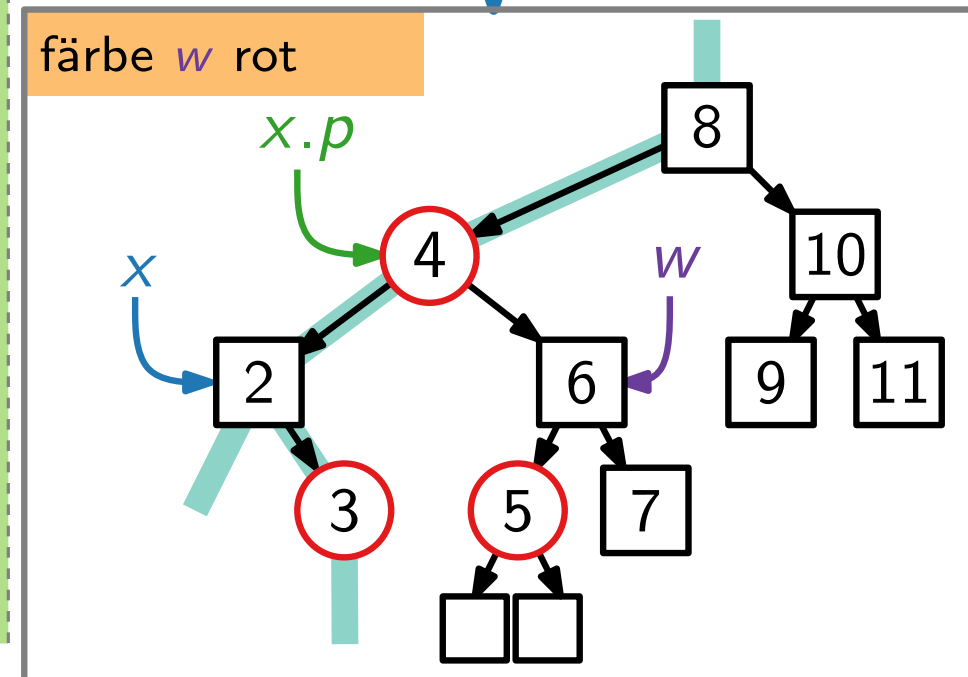
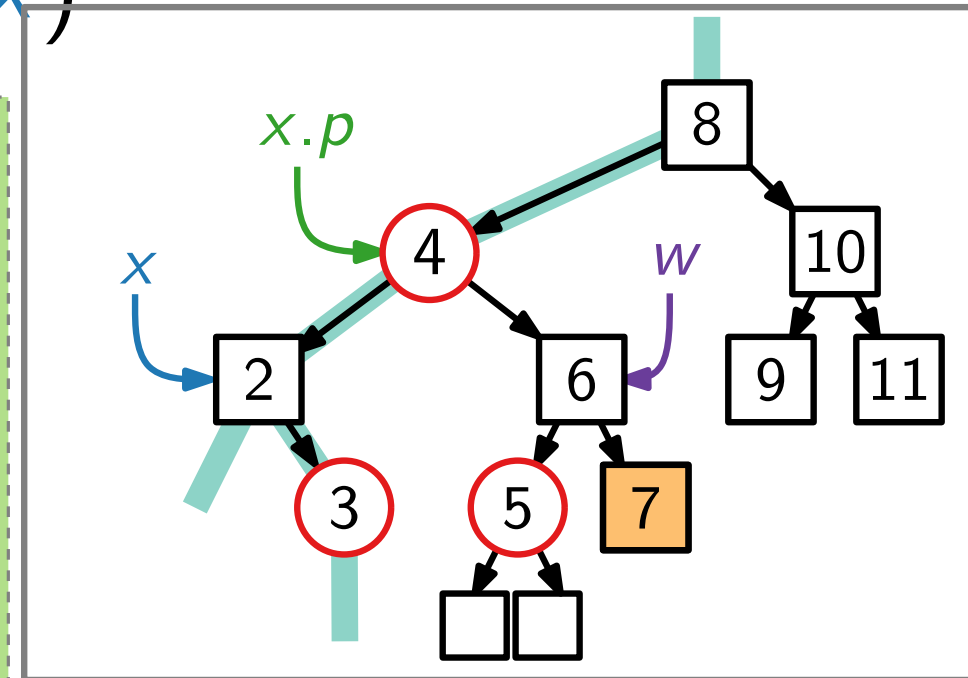
```
 $x.\text{color} = \text{black}$ 
```



RBDELETEFIXUP(RBNode x)

```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
    ...
  else
    if  $w.\text{right}.\text{color} == \text{black}$  then
       $w.\text{left}.\text{color} = \text{black}$ 
       $w.\text{color} = \text{red}$ 
    else ... // wie oben, aber re. & li. vertauscht
   $x.\text{color} = \text{black}$ 
  
```

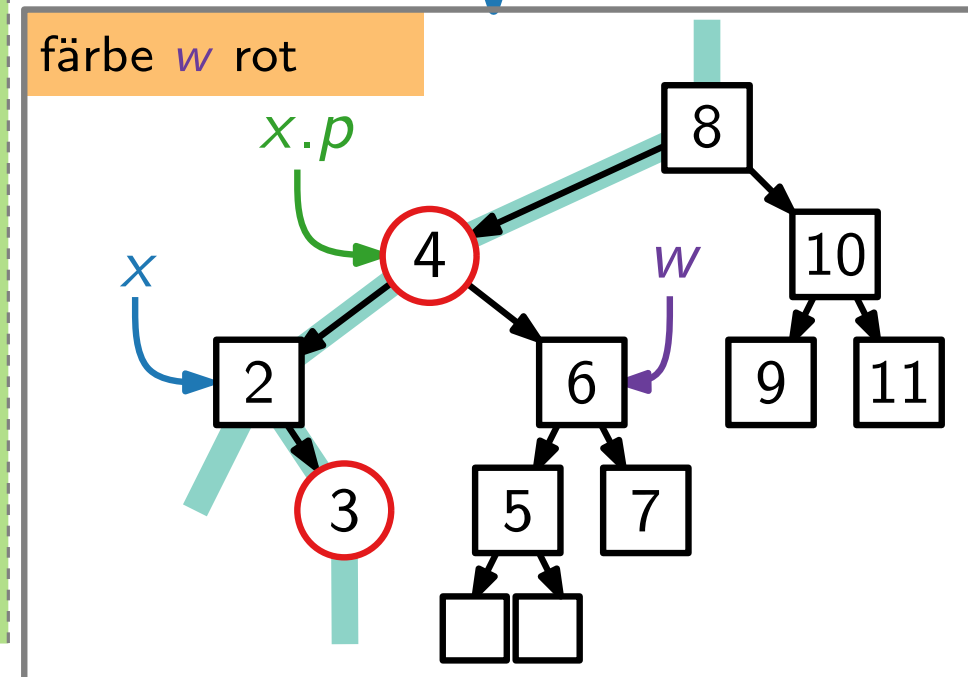
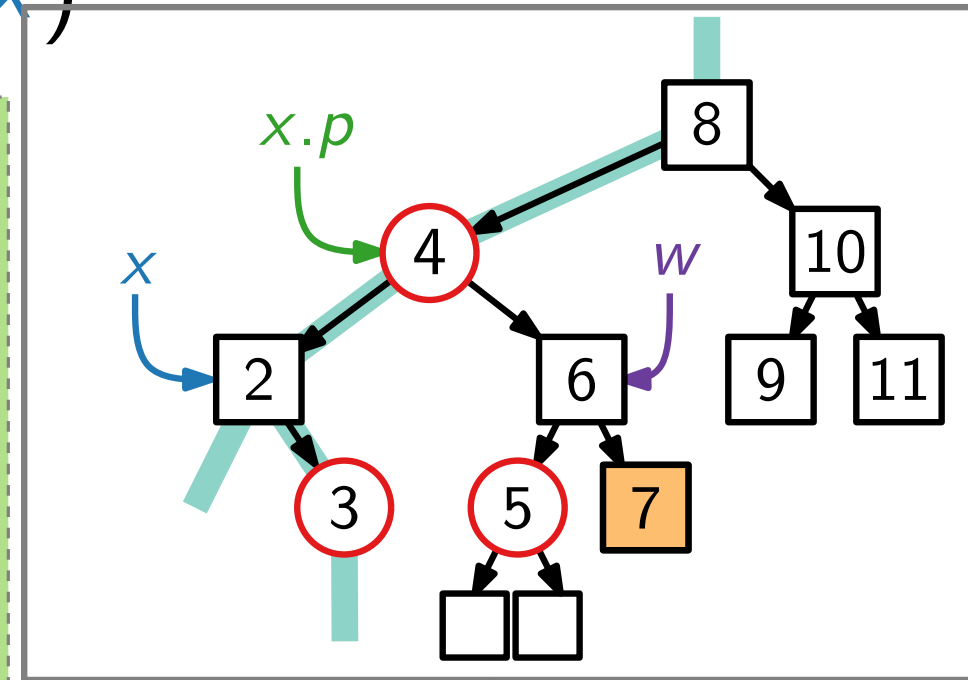


RBDELETEFIXUP(RBNode x)

```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
    ...
  else
    if  $w.\text{right}.\text{color} == \text{black}$  then
       $w.\text{left}.\text{color} = \text{black}$ 
       $w.\text{color} = \text{red}$ 
    else ... // wie oben, aber re. & li. vertauscht
   $x.\text{color} = \text{black}$ 

```



RBDELETEFIXUP(RBNode x)

```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
```

```
  if  $x == x.p.\text{left}$  then
```

```
    ...
```

```
  else
```

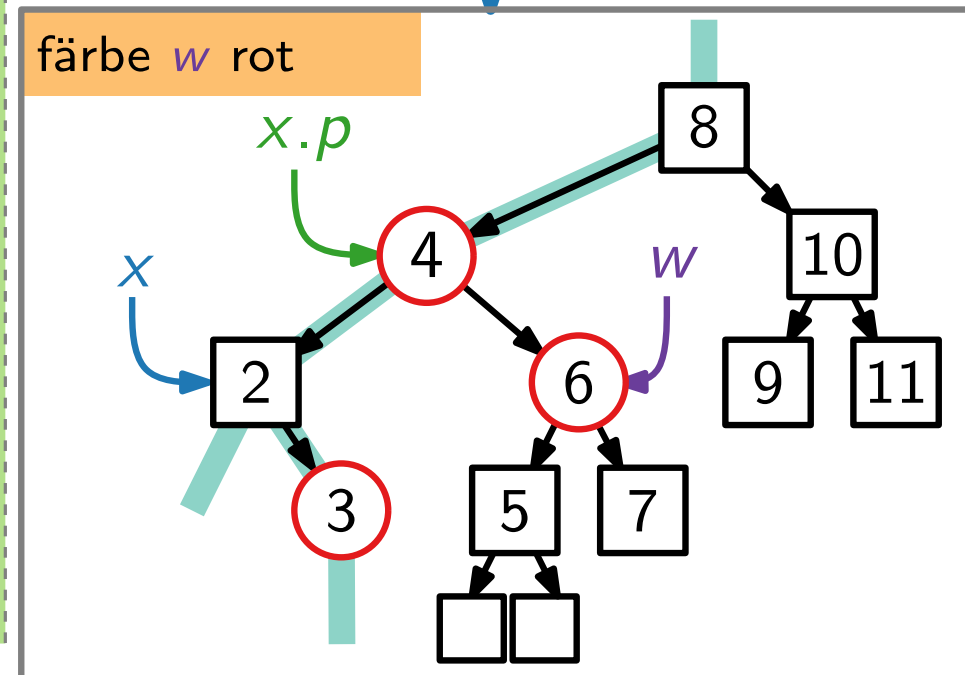
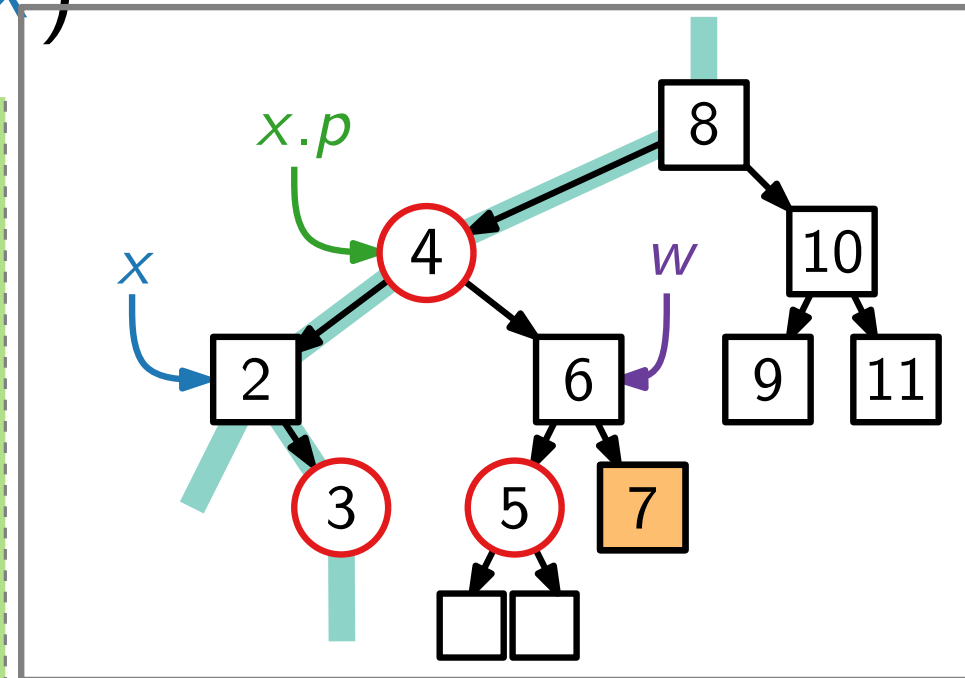
```
    if  $w.\text{right}.\text{color} == \text{black}$  then
```

```
       $w.\text{left}.\text{color} = \text{black}$ 
```

```
       $w.\text{color} = \text{red}$ 
```

```
    else ... // wie oben, aber re. & li. vertauscht
```

```
 $x.\text{color} = \text{black}$ 
```



RBDELETEFIXUP(RBNode x)

```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
```

```
  if  $x == x.p.\text{left}$  then
```

```
    ...
```

```
  else
```

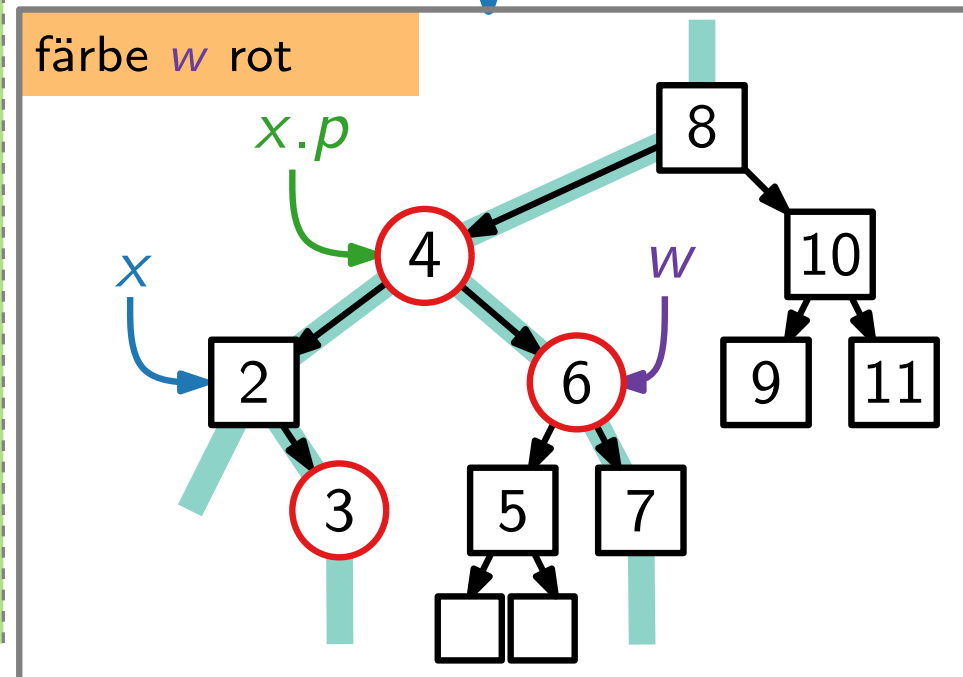
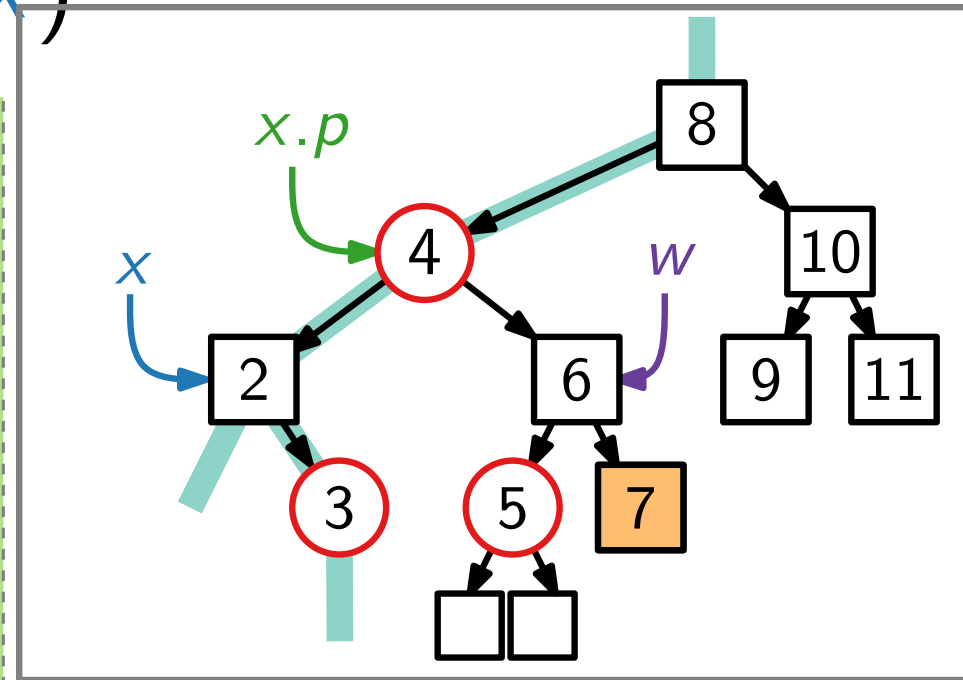
```
    if  $w.\text{right}.\text{color} == \text{black}$  then
```

```
       $w.\text{left}.\text{color} = \text{black}$ 
```

```
       $w.\text{color} = \text{red}$ 
```

```
    else ... // wie oben, aber re. & li. vertauscht
```

```
 $x.\text{color} = \text{black}$ 
```



RBDELETEFIXUP(RBNode x)

```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
```

```
  if  $x == x.p.\text{left}$  then
```

```
    ...
```

```
  else
```

```
    if  $w.\text{right}.\text{color} == \text{black}$  then
```

```
       $w.\text{left}.\text{color} = \text{black}$ 
```

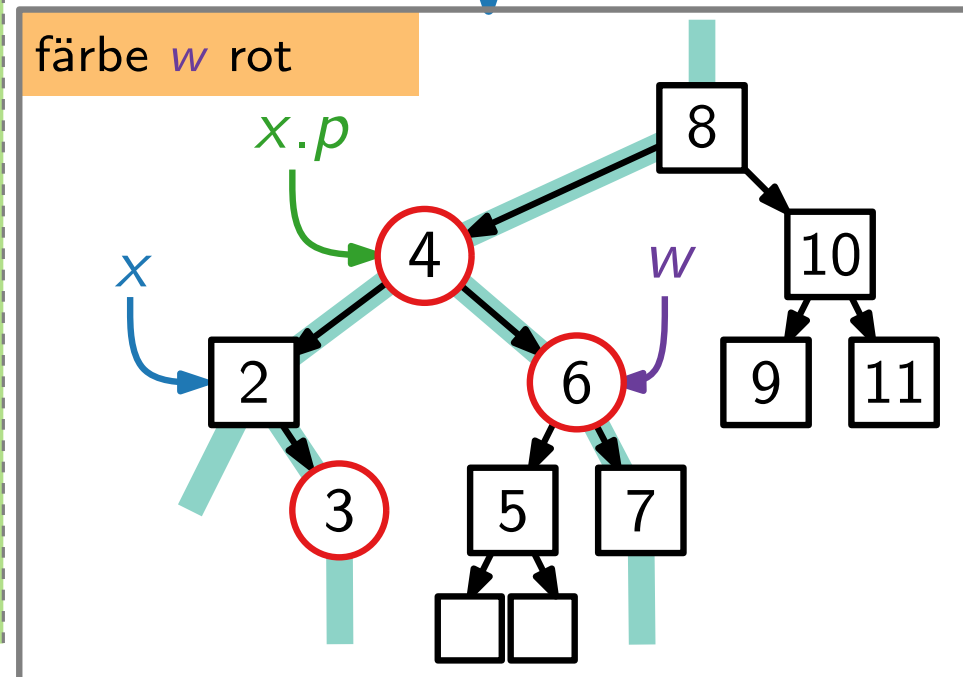
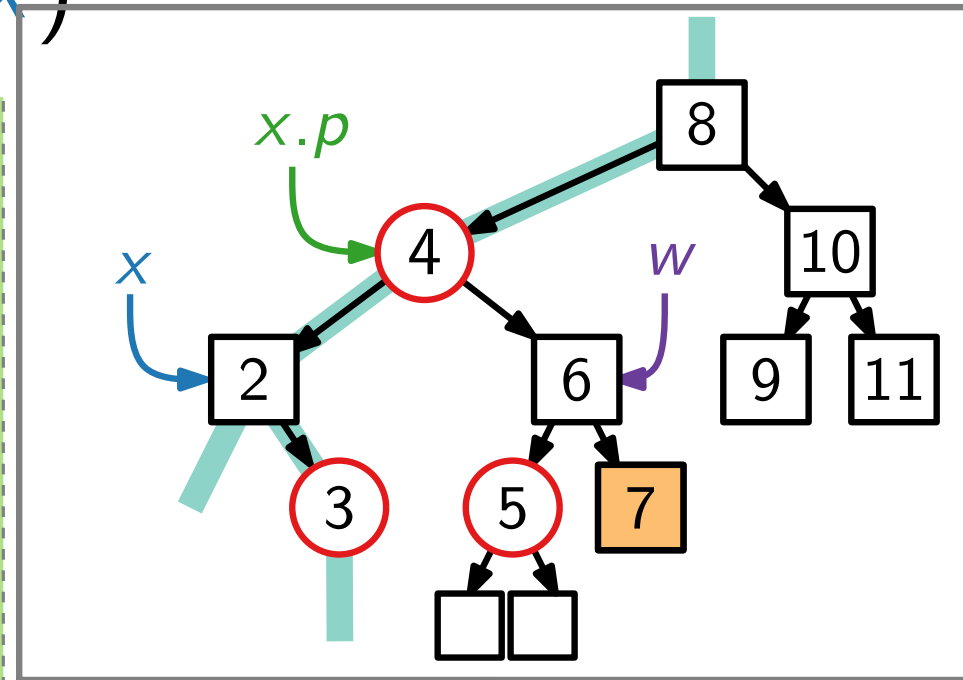
```
       $w.\text{color} = \text{red}$ 
```

```
      RIGHTROTATE( $w$ )
```

```
       $w = x.p.\text{right}$ 
```

```
  else ... // wie oben, aber re. & li. vertauscht
```

```
 $x.\text{color} = \text{black}$ 
```



RBDELETEFIXUP(RBNode x)

```
while  $x \neq root$  and  $x.color == black$  do
```

```
if  $x == x.p.left$  then
```

else

```
if w.right.color == black then
```

w.left.color = black

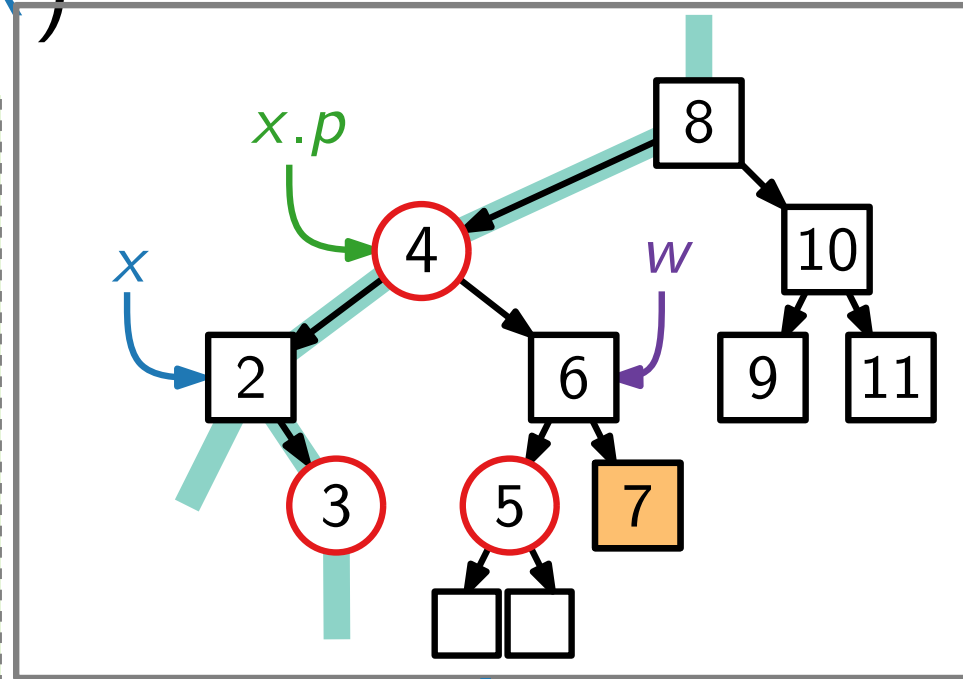
w.color = red

RIGHTROTATE(*w*)

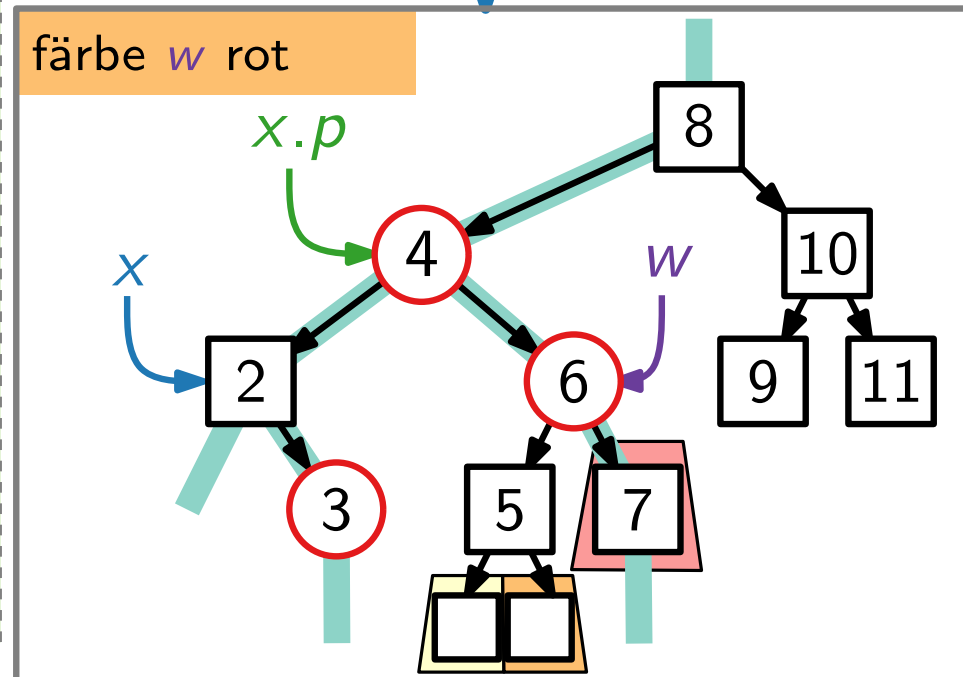
$w = x.p.right$

```
else ... // wie oben, aber re. & li. vertauscht
```

```
x.color = black
```



färbew rot



RBDELETEFIXUP(RBNode x)

```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
```

```
  if  $x == x.p.\text{left}$  then
```

```
    ...
```

```
  else
```

```
    if  $w.\text{right}.\text{color} == \text{black}$  then
```

```
       $w.\text{left}.\text{color} = \text{black}$ 
```

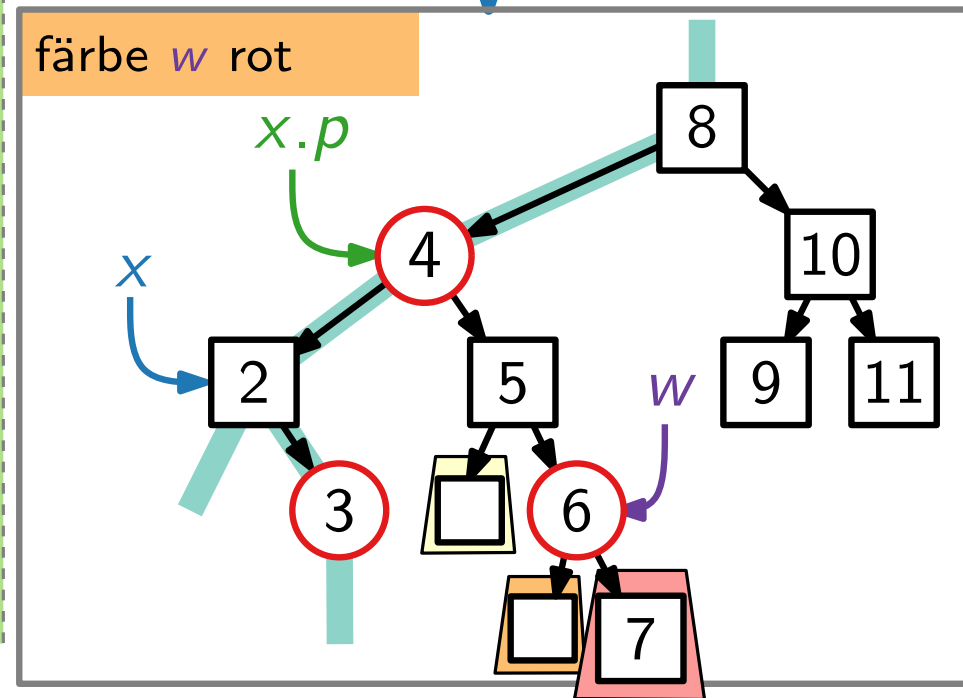
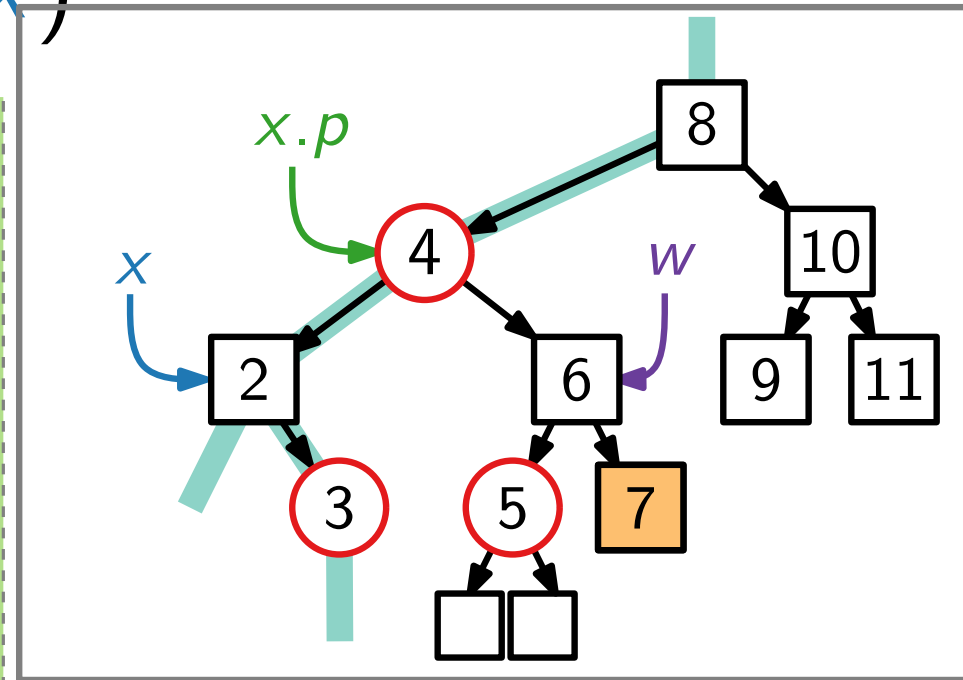
```
       $w.\text{color} = \text{red}$ 
```

```
      RIGHTROTATE( $w$ )
```

```
       $w = x.p.\text{right}$ 
```

```
  else ... // wie oben, aber re. & li. vertauscht
```

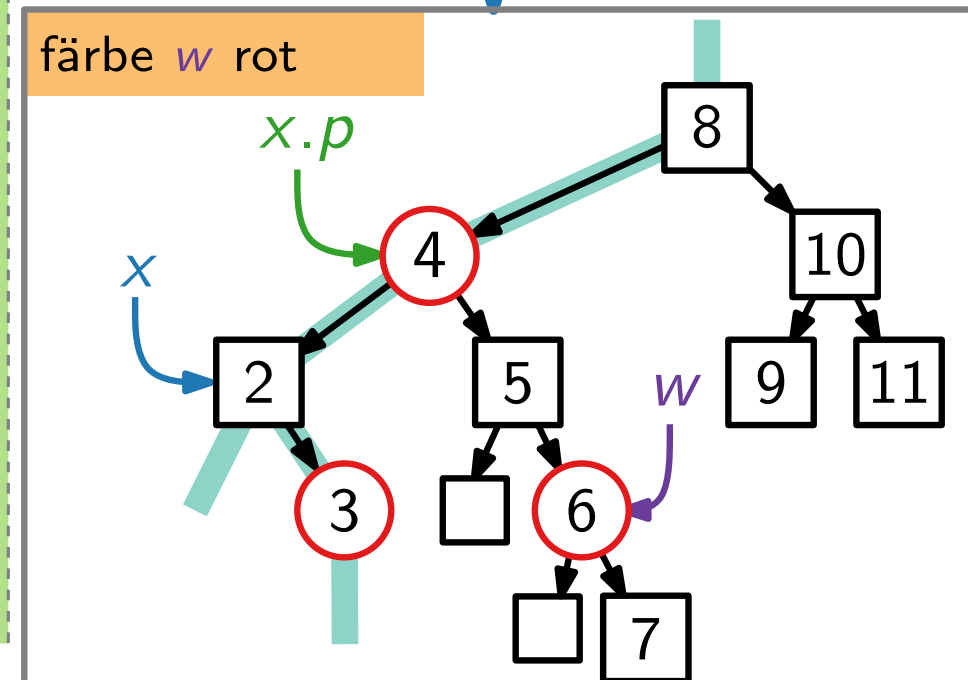
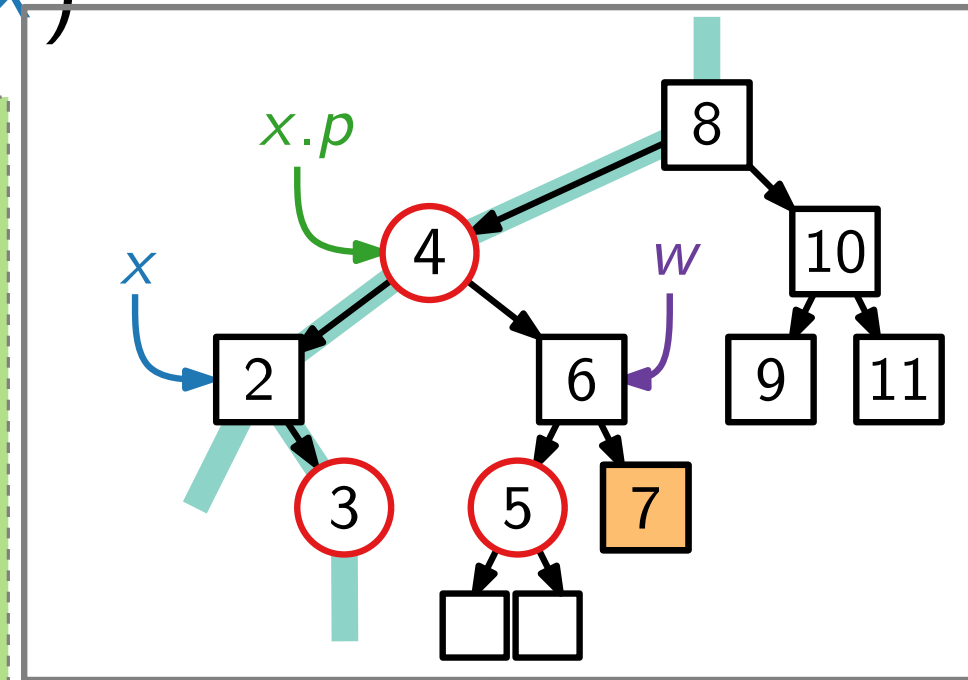
```
 $x.\text{color} = \text{black}$ 
```



RBDELETEFIXUP(RBNode x)

```

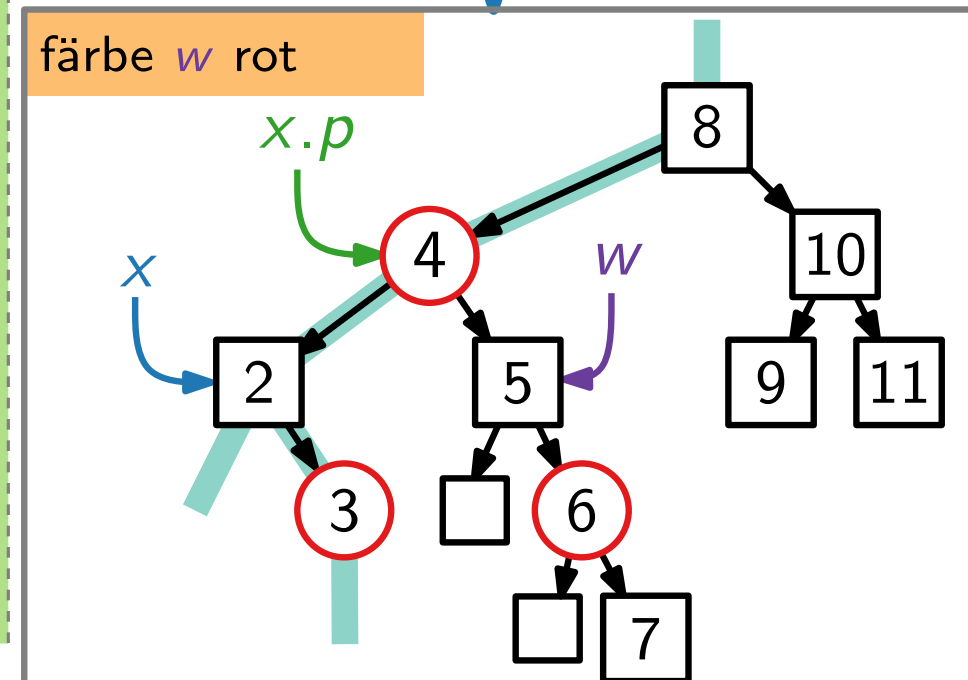
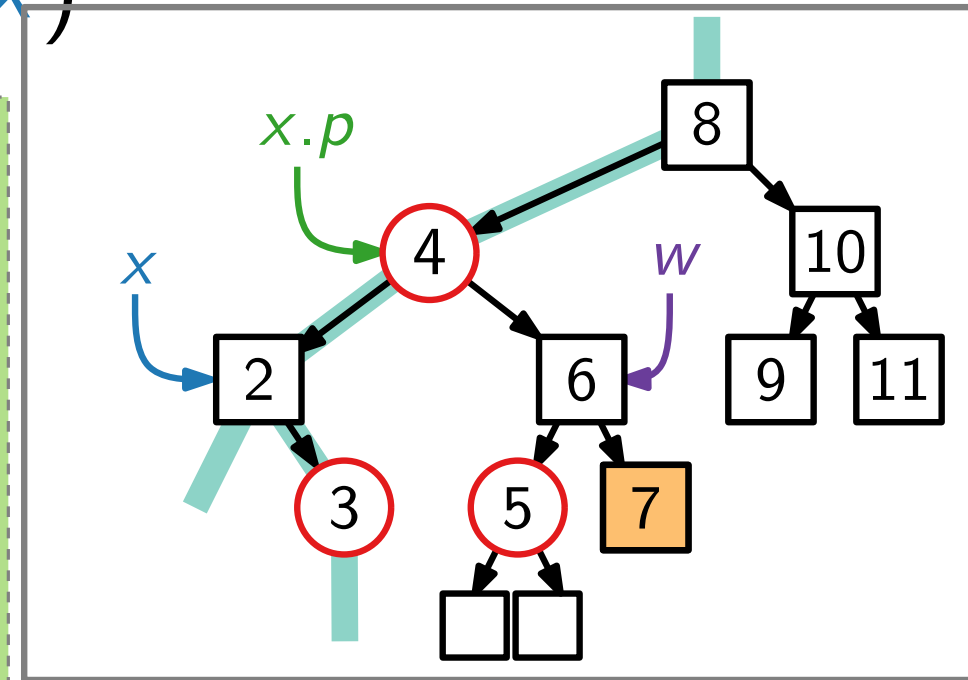
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
    ...
  else
    if  $w.\text{right}.\text{color} == \text{black}$  then
       $w.\text{left}.\text{color} = \text{black}$ 
       $w.\text{color} = \text{red}$ 
      RIGHTROTATE( $w$ )
       $w = x.p.\text{right}$ 
    else ... // wie oben, aber re. & li. vertauscht
   $x.\text{color} = \text{black}$ 
  
```



RBDELETEFIXUP(RBNode x)

```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
    ...
  else
    if  $w.\text{right}.\text{color} == \text{black}$  then
       $w.\text{left}.\text{color} = \text{black}$ 
       $w.\text{color} = \text{red}$ 
      RIGHTROTATE( $w$ )
       $w = x.p.\text{right}$ 
    else ... // wie oben, aber re. & li. vertauscht
   $x.\text{color} = \text{black}$ 
  
```



RBDELETEFIXUP(RBNode x)

while $x \neq \text{root}$ **and** $x.\text{color} == \text{black}$ **do**

if $x == x.p.\text{left}$ **then**

...

else

if $w.\text{right}.\text{color} == \text{black}$ **then**

$w.\text{left}.\text{color} = \text{black}$

$w.\text{color} = \text{red}$

RIGHTROTATE(w)

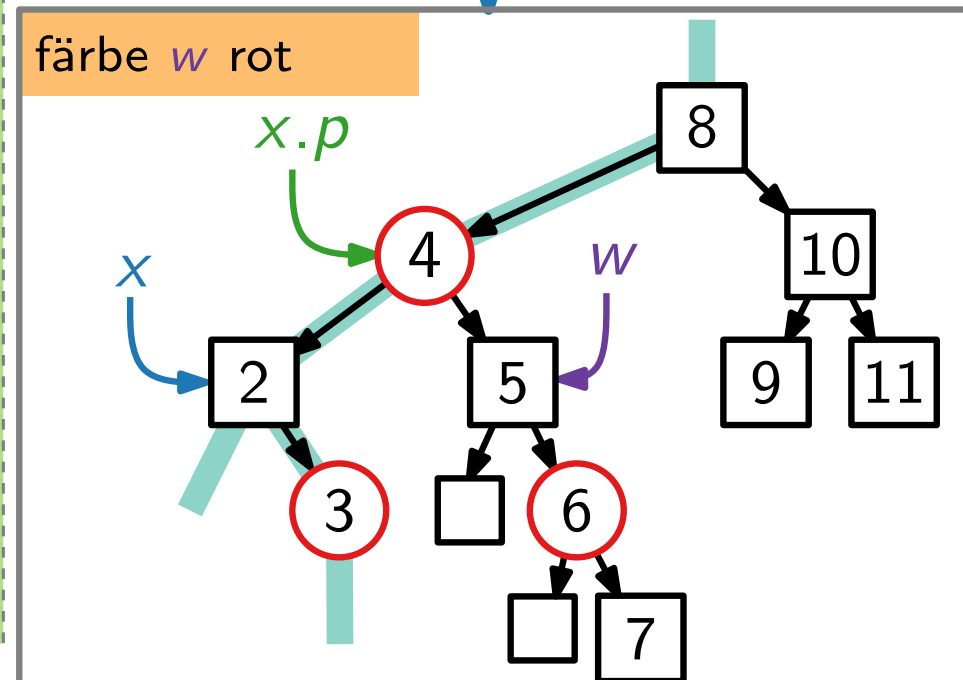
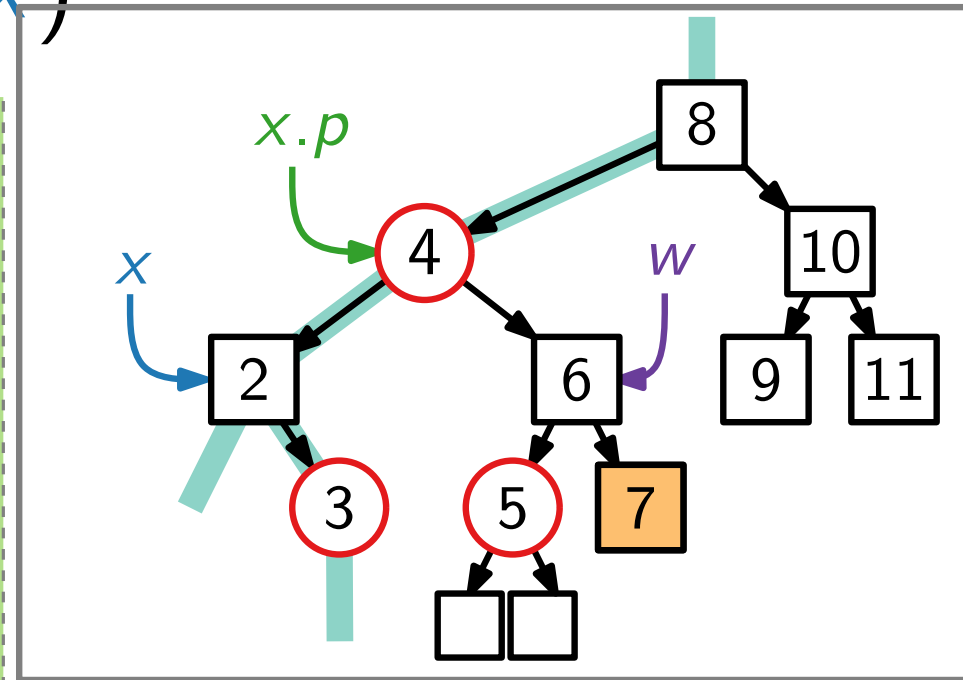
$w = x.p.\text{right}$

$w.\text{color} = x.p.\text{color}$

$x.p.\text{color} = \text{black}$

else ... // wie oben, aber re. & li. vertauscht

$x.\text{color} = \text{black}$



RBDELETEFIXUP(RBNode x)

```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
```

```
  if  $x == x.p.\text{left}$  then
```

```
    ...
```

```
  else
```

```
    if  $w.\text{right}.\text{color} == \text{black}$  then
```

```
       $w.\text{left}.\text{color} = \text{black}$ 
```

```
       $w.\text{color} = \text{red}$ 
```

```
      RIGHTROTATE( $w$ )
```

```
       $w = x.p.\text{right}$ 
```

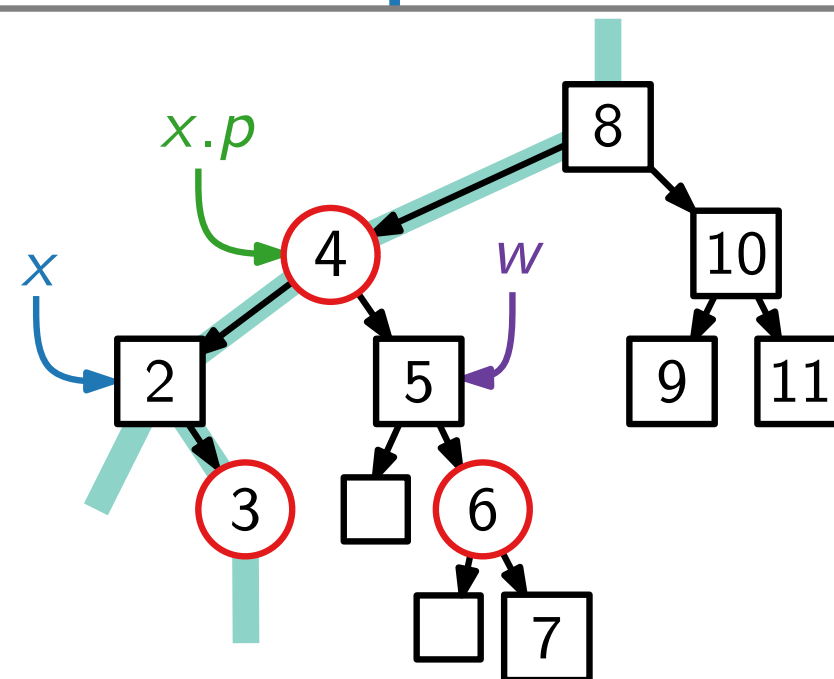
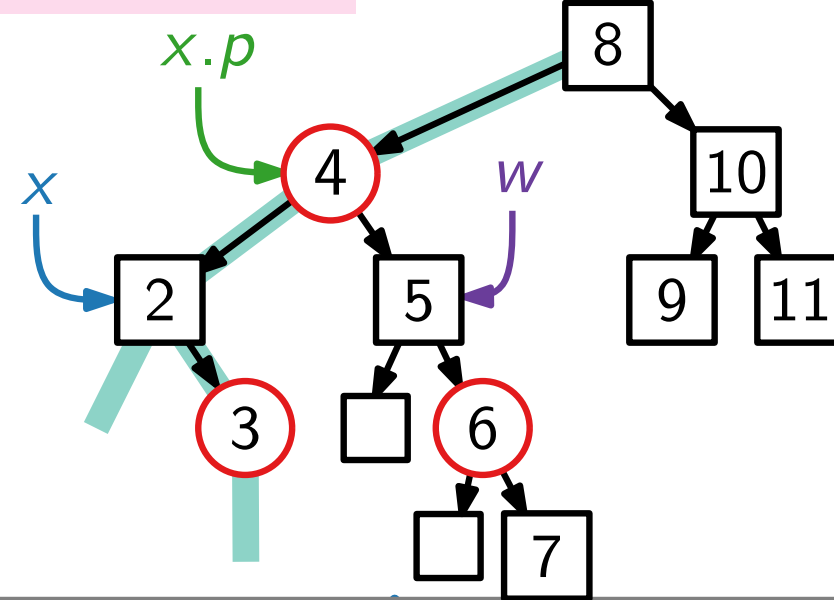
```
     $w.\text{color} = x.p.\text{color}$ 
```

```
     $x.p.\text{color} = \text{black}$ 
```

```
  else ... // wie oben, aber re. & li. vertauscht
```

```
 $x.\text{color} = \text{black}$ 
```

färbe $x.p$ schwarz



RBDELETEFIXUP(RBNode x)

```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
```

```
  if  $x == x.p.\text{left}$  then
```

```
    ...
```

```
  else
```

```
    if  $w.\text{right}.\text{color} == \text{black}$  then
```

```
       $w.\text{left}.\text{color} = \text{black}$ 
```

```
       $w.\text{color} = \text{red}$ 
```

```
      RIGHTROTATE( $w$ )
```

```
       $w = x.p.\text{right}$ 
```

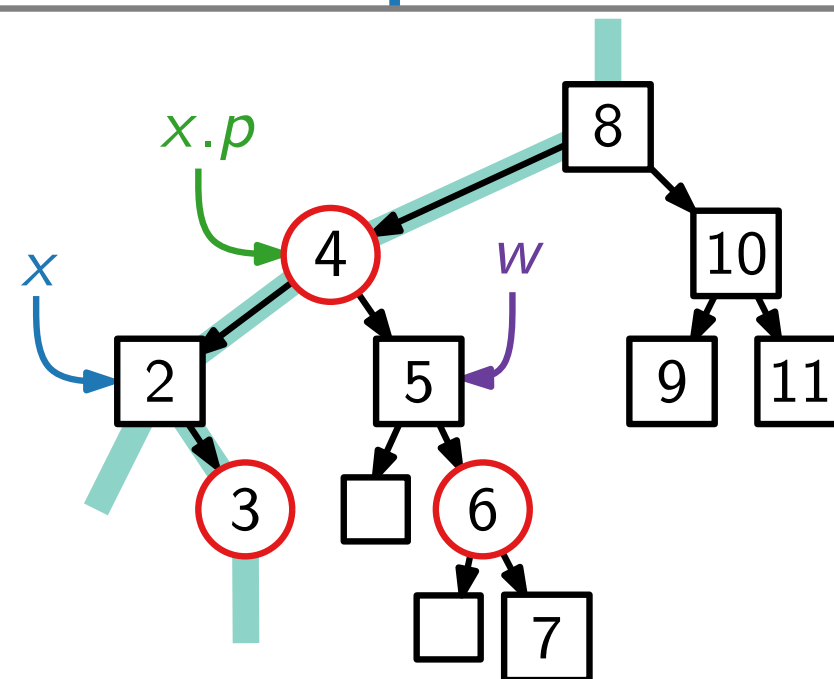
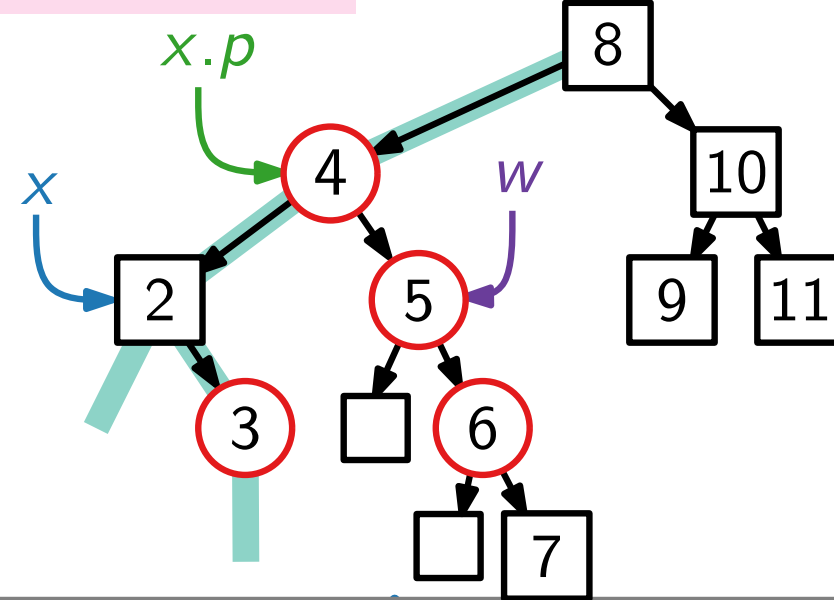
```
     $w.\text{color} = x.p.\text{color}$ 
```

```
     $x.p.\text{color} = \text{black}$ 
```

```
  else ... // wie oben, aber re. & li. vertauscht
```

```
 $x.\text{color} = \text{black}$ 
```

färbe $x.p$ schwarz



RBDELETEFIXUP(RBNode x)

```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
```

```
  if  $x == x.p.\text{left}$  then
```

```
    ...
```

```
  else
```

```
    if  $w.\text{right}.\text{color} == \text{black}$  then
```

```
       $w.\text{left}.\text{color} = \text{black}$ 
```

```
       $w.\text{color} = \text{red}$ 
```

```
      RIGHTROTATE( $w$ )
```

```
       $w = x.p.\text{right}$ 
```

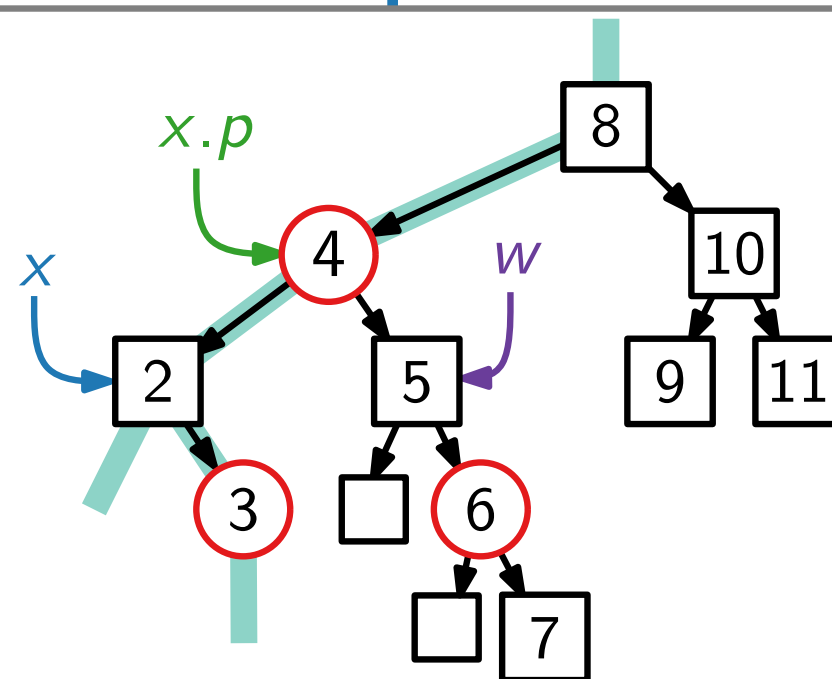
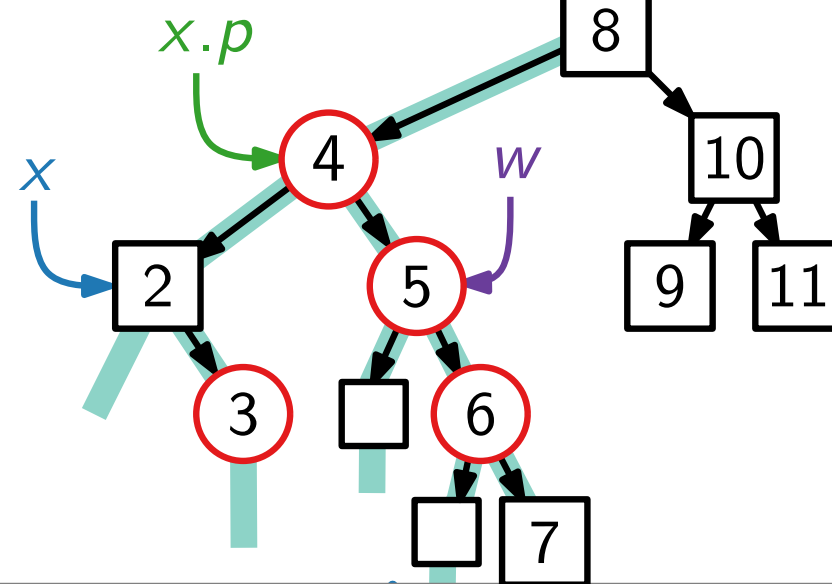
```
     $w.\text{color} = x.p.\text{color}$ 
```

```
     $x.p.\text{color} = \text{black}$ 
```

```
  else ... // wie oben, aber re. & li. vertauscht
```

```
 $x.\text{color} = \text{black}$ 
```

färbe $x.p$ schwarz



RBDELETEFIXUP(RBNode x)

```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
```

```
  if  $x == x.p.\text{left}$  then
```

```
    ...
```

```
  else
```

```
    if  $w.\text{right}.\text{color} == \text{black}$  then
```

```
       $w.\text{left}.\text{color} = \text{black}$ 
```

```
       $w.\text{color} = \text{red}$ 
```

```
      RIGHTROTATE( $w$ )
```

```
       $w = x.p.\text{right}$ 
```

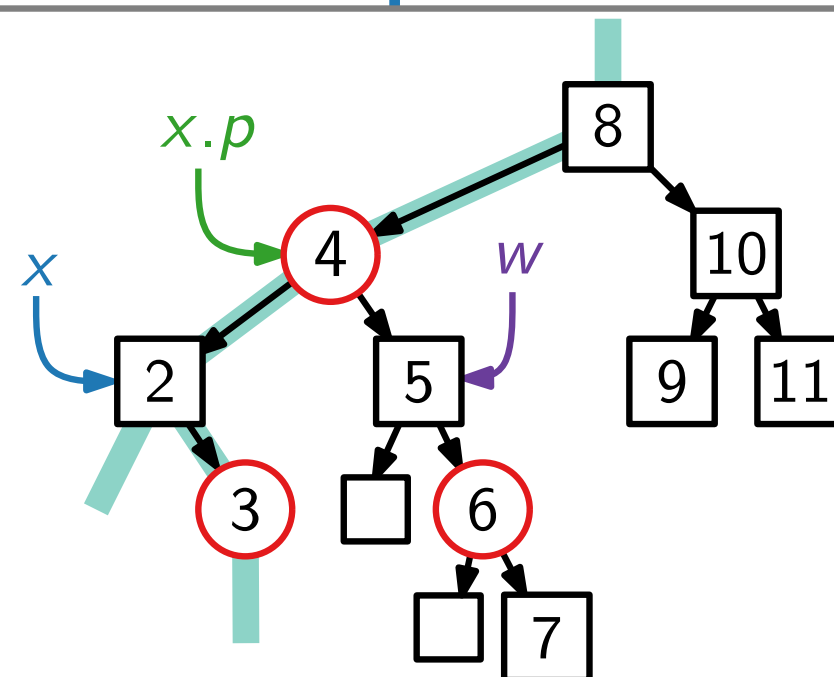
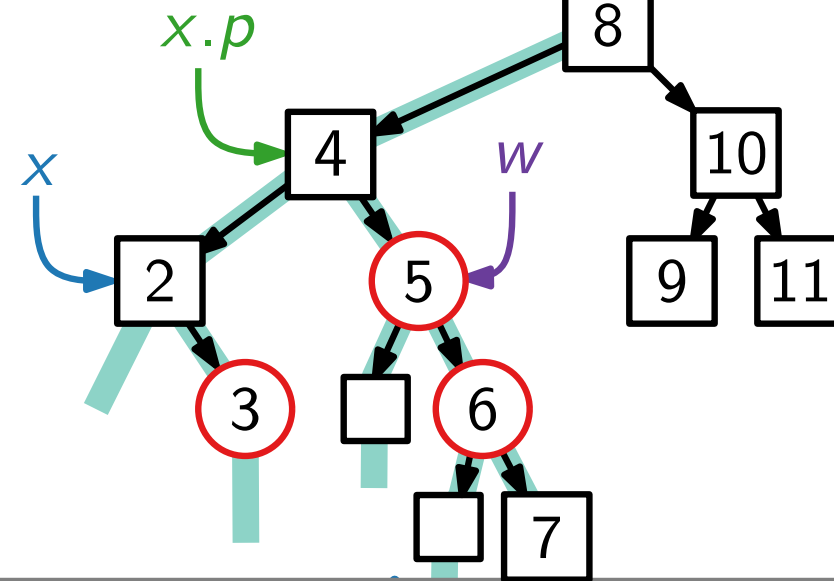
```
     $w.\text{color} = x.p.\text{color}$ 
```

```
     $x.p.\text{color} = \text{black}$ 
```

```
  else ... // wie oben, aber re. & li. vertauscht
```

```
 $x.\text{color} = \text{black}$ 
```

färbe $x.p$ schwarz



RBDELETEFIXUP(RBNode x)

while $x \neq \text{root}$ **and** $x.\text{color} == \text{black}$ **do**

if $x == x.p.\text{left}$ **then**

...

else

if $w.\text{right}.\text{color} == \text{black}$ **then**

$w.\text{left}.\text{color} = \text{black}$

$w.\text{color} = \text{red}$

RIGHTROTATE(w)

$w = x.p.\text{right}$

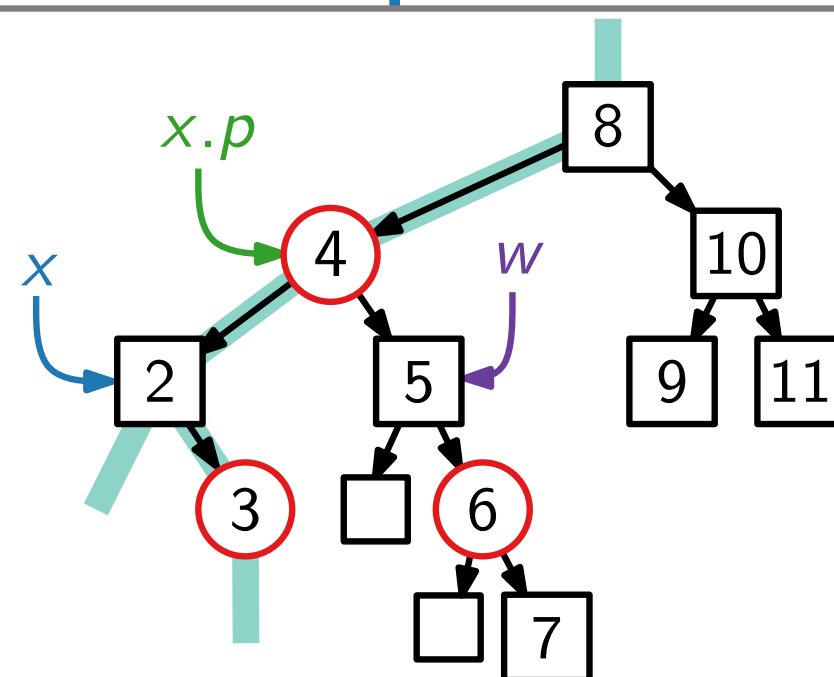
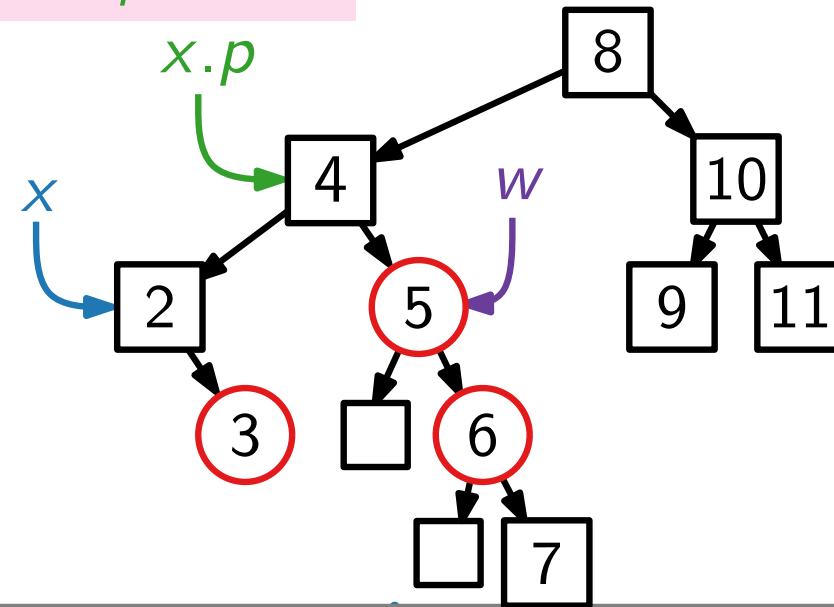
$w.\text{color} = x.p.\text{color}$

$x.p.\text{color} = \text{black}$

else ... // wie oben, aber re. & li. vertauscht

$x.\text{color} = \text{black}$

färbe $x.p$ schwarz



RBDELETEFIXUP(RBNode x)

while $x \neq \text{root}$ **and** $x.\text{color} == \text{black}$ **do**

if $x == x.p.\text{left}$ **then**

...

else

if $w.\text{right}.\text{color} == \text{black}$ **then**

$w.\text{left}.\text{color} = \text{black}$

$w.\text{color} = \text{red}$

RIGHTROTATE(w)

$w = x.p.\text{right}$

$w.\text{color} = x.p.\text{color}$

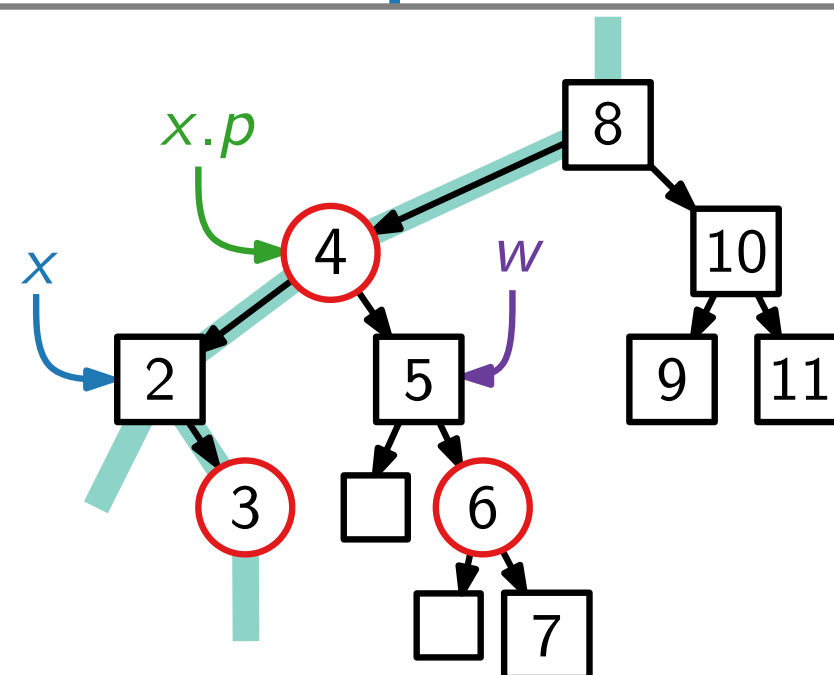
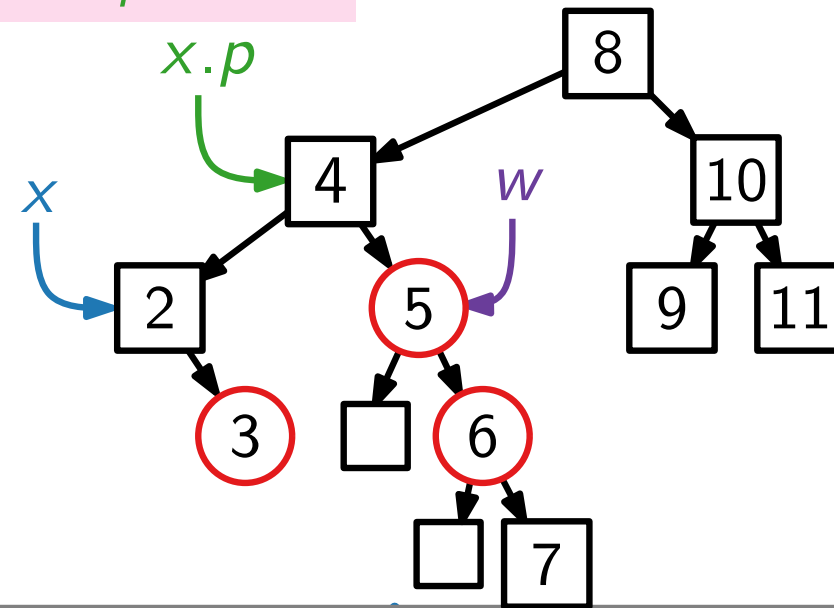
$x.p.\text{color} = \text{black}$

$w.\text{right}.\text{color} = \text{black}$

else ... // wie oben, aber re. & li. vertauscht

$x.\text{color} = \text{black}$

färbe $x.p$ schwarz



RBDELETEFIXUP(RBNode x)

while $x \neq \text{root}$ **and** $x.\text{color} == \text{black}$ **do**

if $x == x.p.\text{left}$ **then**

...

...

else

if $w.\text{right}.\text{color} == \text{black}$ **then**

$w.\text{left}.\text{color} = \text{black}$

$w.\text{color} = \text{red}$

RIGHTROTATE(w)

$w = x.p.\text{right}$

$w.\text{color} = x.p.\text{color}$

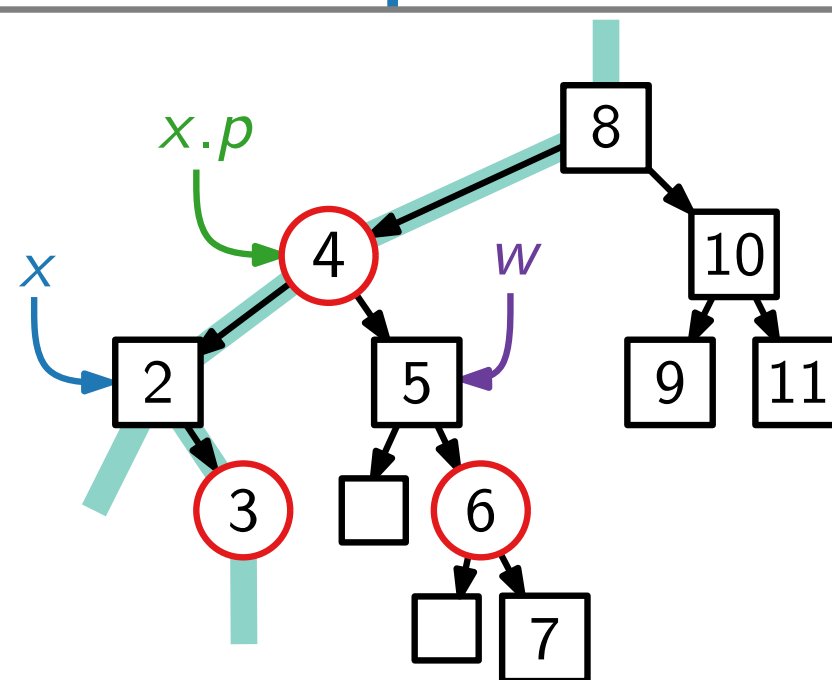
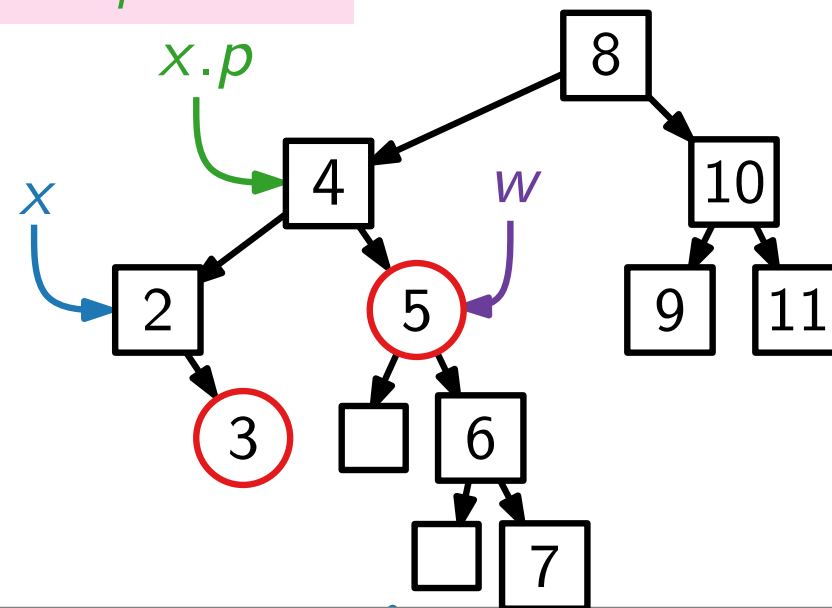
$x.p.\text{color} = \text{black}$

$w.\text{right}.\text{color} = \text{black}$

else ... // wie oben, aber re. & li. vertauscht

$x.\text{color} = \text{black}$

färbe $x.p$ schwarz



RBDELETEFIXUP(RBNode x)

while $x \neq \text{root}$ **and** $x.\text{color} == \text{black}$ **do**

if $x == x.p.\text{left}$ **then**

else

if $w.\text{right}.\text{color} == \text{black}$ **then**

$w.\text{left}.\text{color} = \text{black}$

$w.\text{color} = \text{red}$

 RIGHTROTATE(w)

$w = x.p.\text{right}$

$w.\text{color} = x.p.\text{color}$

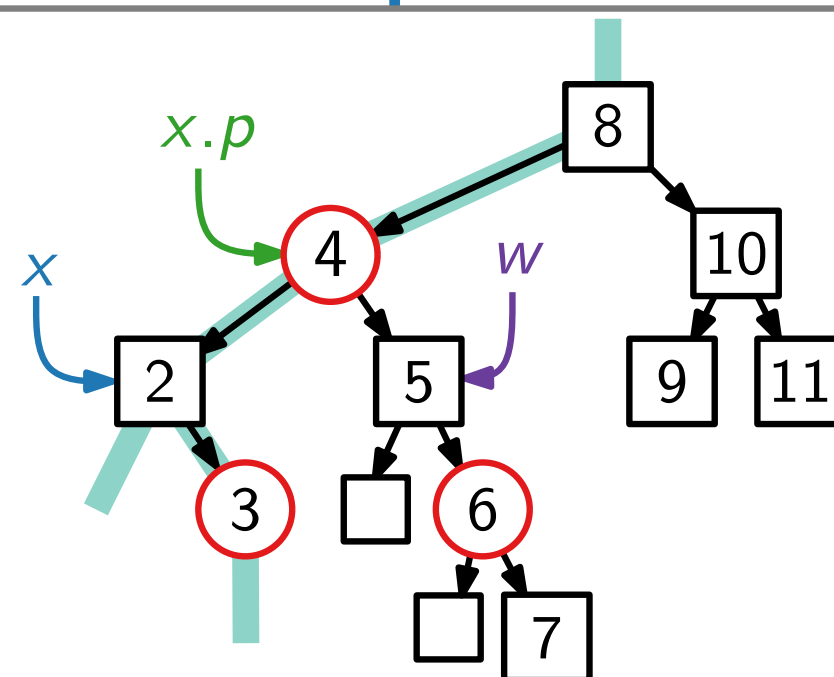
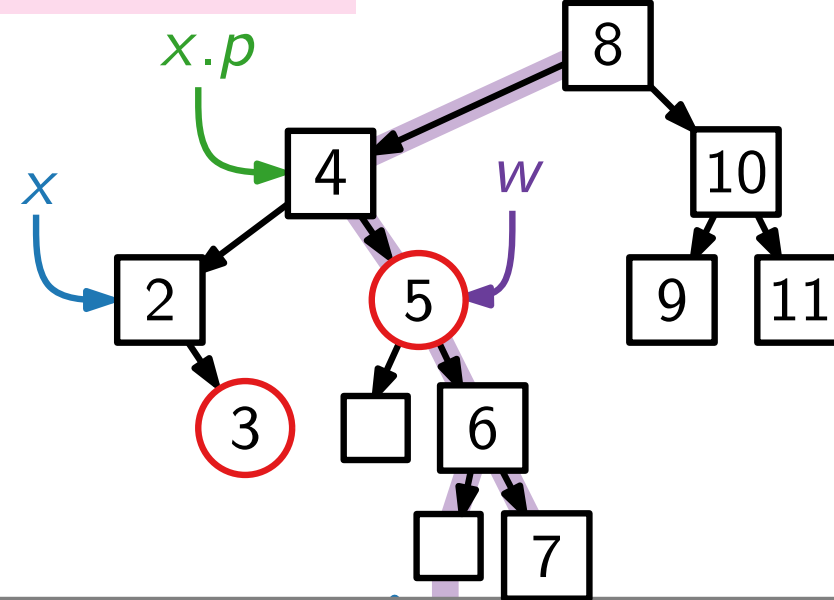
$x.p.\text{color} = \text{black}$

$w.\text{right}.\text{color} = \text{black}$

else ... // wie oben, aber re. & li. vertauscht

$x.\text{color} = \text{black}$

färbe $x.p$ schwarz



RBDELETEFIXUP(RBNode x)

```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
```

```
  if  $x == x.p.\text{left}$  then
```

```
    ...
```

```
    ...
```

```
  else
```

```
    if  $w.\text{right}.\text{color} == \text{black}$  then
```

```
       $w.\text{left}.\text{color} = \text{black}$ 
```

```
       $w.\text{color} = \text{red}$ 
```

```
      RIGHTROTATE( $w$ )
```

```
       $w = x.p.\text{right}$ 
```

```
       $w.\text{color} = x.p.\text{color}$ 
```

```
       $x.p.\text{color} = \text{black}$ 
```

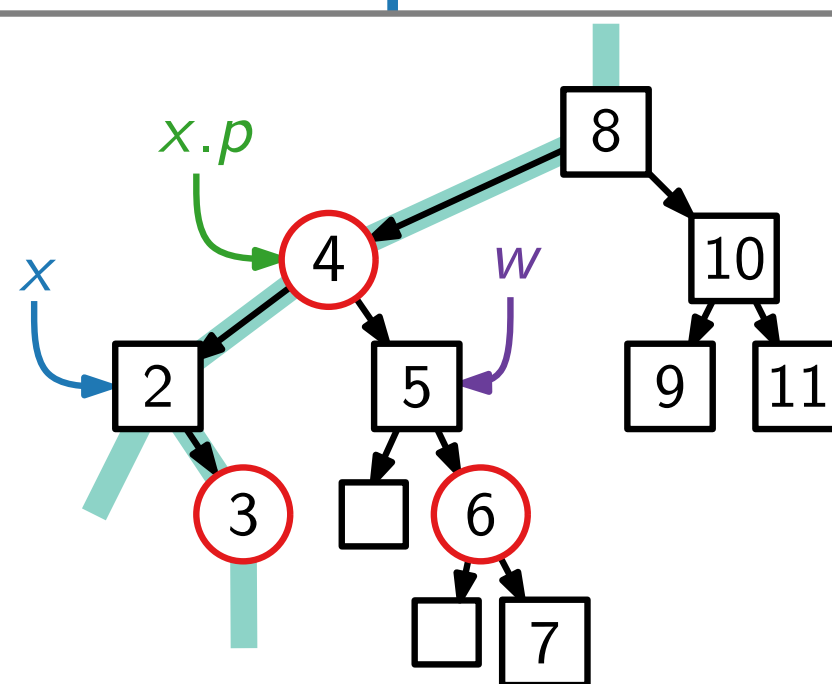
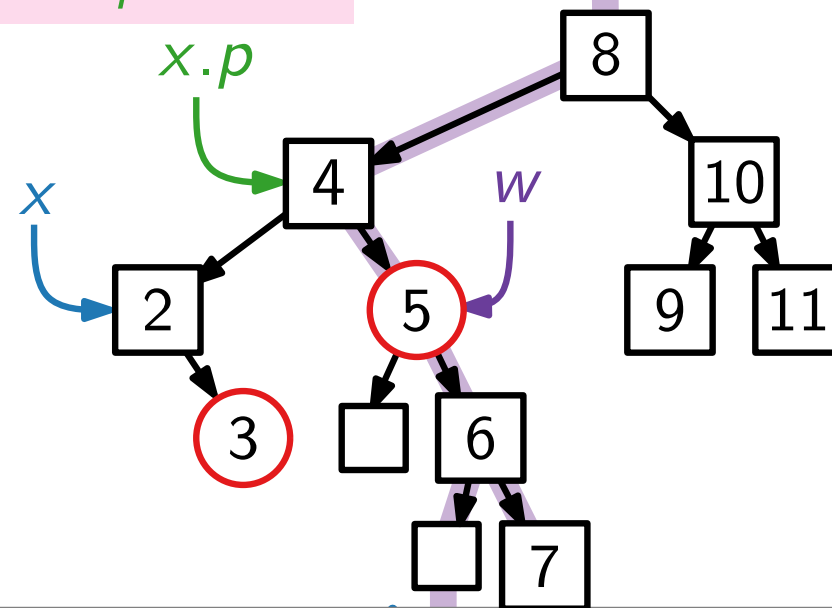
```
       $w.\text{right}.\text{color} = \text{black}$ 
```

```
      LEFTROTATE( $x.p$ )
```

```
    else ... // wie oben, aber re. & li. vertauscht
```

```
 $x.\text{color} = \text{black}$ 
```

färbe $x.p$ schwarz



RBDELETEFIXUP(RBNode x)

while $x \neq \text{root}$ **and** $x.\text{color} == \text{black}$ **do**

if $x == x.p.\text{left}$ **then**

...

else

if $w.\text{right}.\text{color} == \text{black}$ **then**

$w.\text{left}.\text{color} = \text{black}$

$w.\text{color} = \text{red}$

RIGHTROTATE(w)

$w = x.p.\text{right}$

$w.\text{color} = x.p.\text{color}$

$x.p.\text{color} = \text{black}$

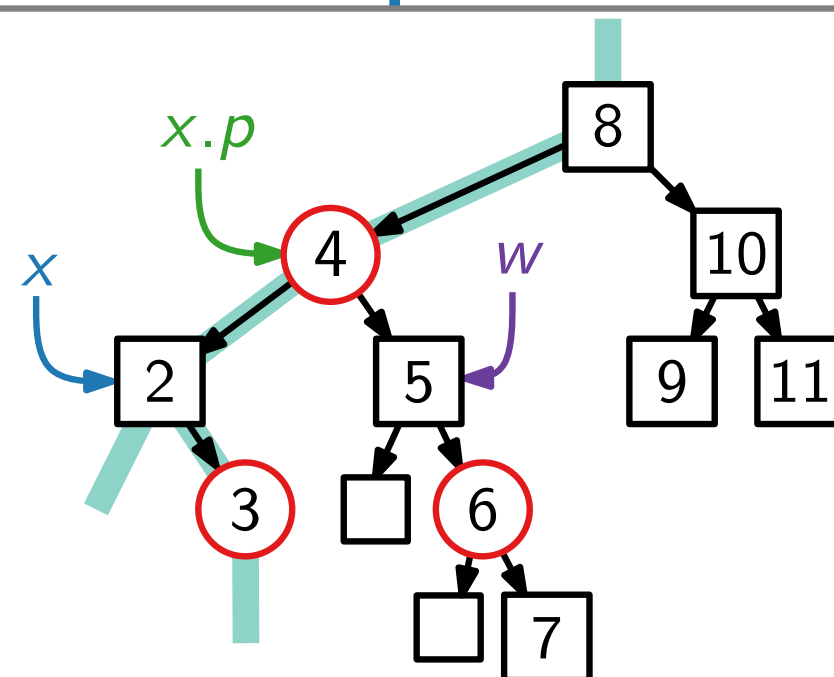
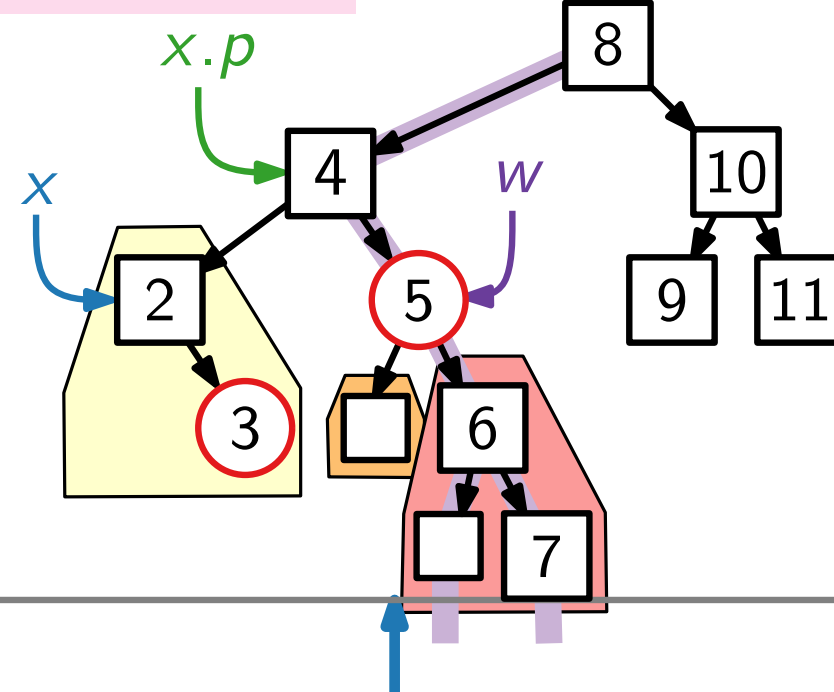
$w.\text{right}.\text{color} = \text{black}$

LEFTROTATE($x.p$)

else ... // wie oben, aber re. & li. vertauscht

$x.\text{color} = \text{black}$

färbe $x.p$ schwarz



RBDELETEFIXUP(RBNode x)

while $x \neq \text{root}$ **and** $x.\text{color} == \text{black}$ **do**

if $x == x.p.\text{left}$ **then**

else

if $w.\text{right}.\text{color} == \text{black}$ **then**

$w.\text{left}.\text{color} = \text{black}$

$w.\text{color} = \text{red}$

 RIGHTROTATE(w)

$w = x.p.\text{right}$

$w.\text{color} = x.p.\text{color}$

$x.p.\text{color} = \text{black}$

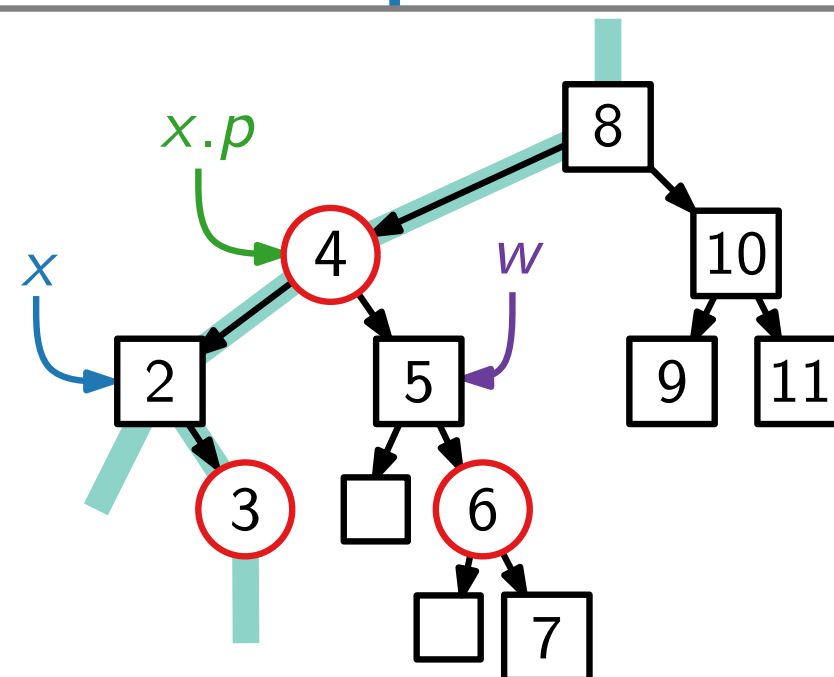
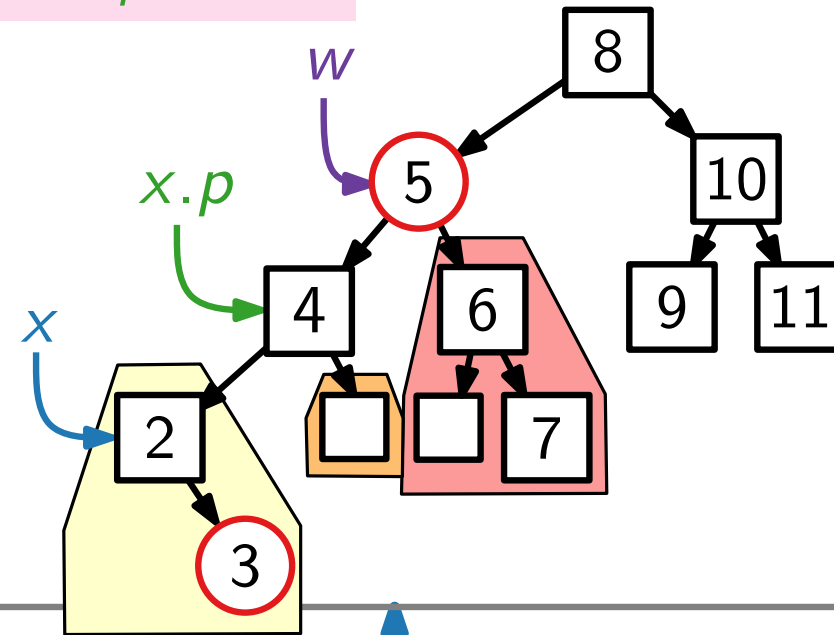
$w.\text{right}.\text{color} = \text{black}$

 LEFTROTATE($x.p$)

else ... // wie oben, aber re. & li. vertauscht

$x.\text{color} = \text{black}$

färbe $x.p$ schwarz



RBDELETEFIXUP(RBNode x)

while $x \neq \text{root}$ **and** $x.\text{color} == \text{black}$ **do**

if $x == x.p.\text{left}$ **then**

...

else

if $w.\text{right}.\text{color} == \text{black}$ **then**

$w.\text{left}.\text{color} = \text{black}$

$w.\text{color} = \text{red}$

RIGHTROTATE(w)

$w = x.p.\text{right}$

$w.\text{color} = x.p.\text{color}$

$x.p.\text{color} = \text{black}$

$w.\text{right}.\text{color} = \text{black}$

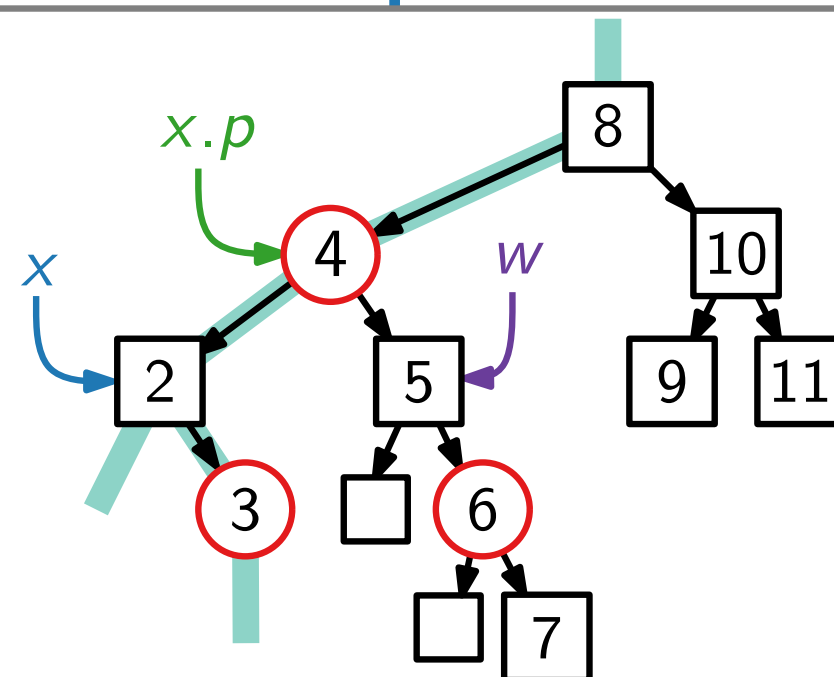
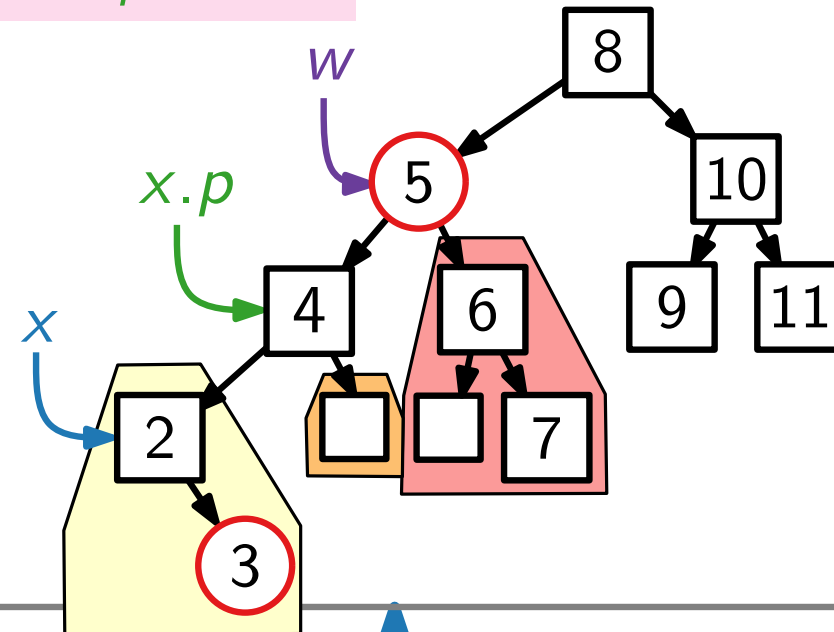
LEFTROTATE($x.p$)

$x = \text{root}$

else ... // wie oben, aber re. & li. vertauscht

$x.\text{color} = \text{black}$

färbe $x.p$ schwarz



RBDELETEFIXUP(RBNode x)

```
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
```

```
  if  $x == x.p.\text{left}$  then
```

```
    ...
```

```
  else
```

```
    if  $w.\text{right}.\text{color} == \text{black}$  then
```

```
       $w.\text{left}.\text{color} = \text{black}$ 
```

```
       $w.\text{color} = \text{red}$ 
```

```
      RIGHTROTATE( $w$ )
```

```
       $w = x.p.\text{right}$ 
```

```
     $w.\text{color} = x.p.\text{color}$ 
```

```
     $x.p.\text{color} = \text{black}$ 
```

```
     $w.\text{right}.\text{color} = \text{black}$ 
```

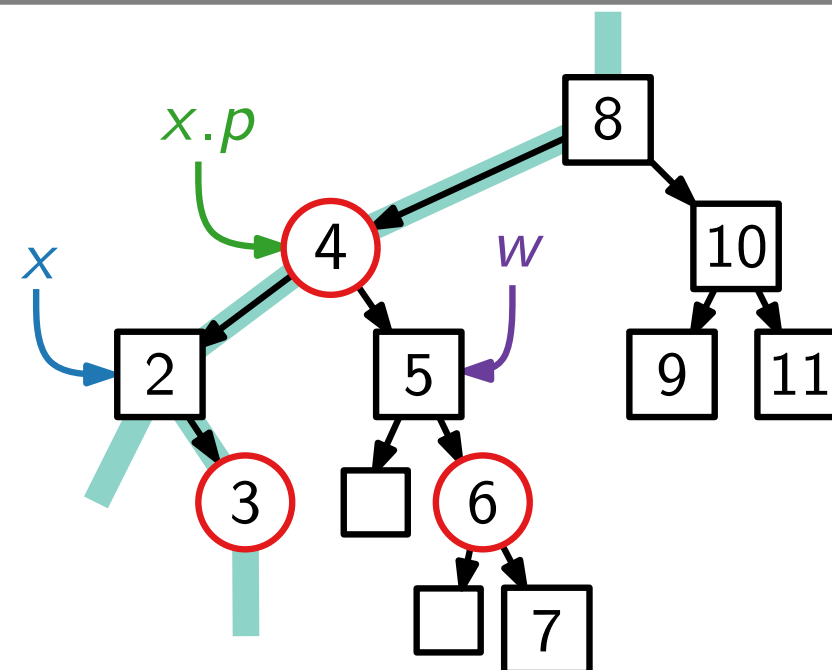
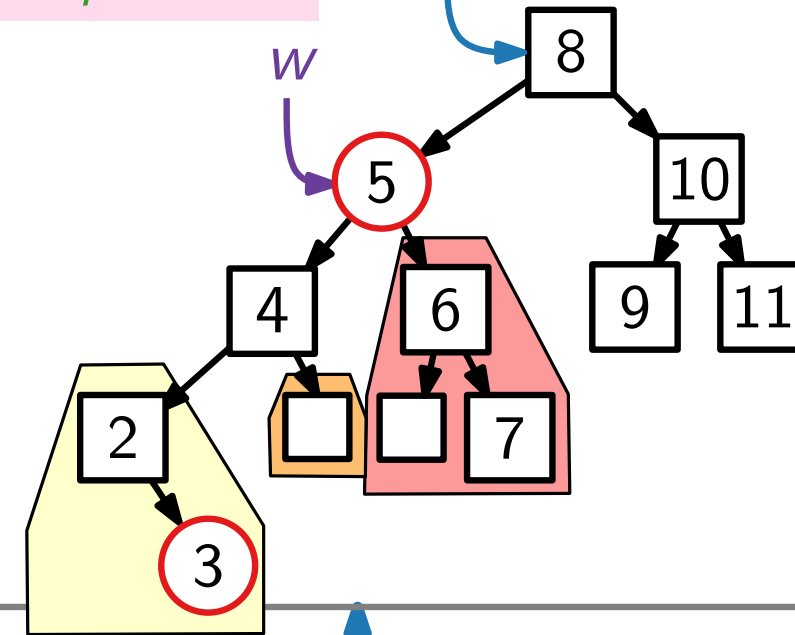
```
    LEFTROTATE( $x.p$ )
```

```
     $x = \text{root}$ 
```

```
  else ... // wie oben, aber re. & li. vertauscht
```

```
 $x.\text{color} = \text{black}$ 
```

färbe $x.p$ schwarz



Übersicht

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld		$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP	—	$\Theta(\log n)$	$\Theta(1)/—$	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

* unter bestimmten Annahmen.

Übersicht

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld		$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP	—	$\Theta(\log n)$	$\Theta(1)/—$	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$
Rot-Schwarz-Baum	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

* unter bestimmten Annahmen.