

Algorithmen und Datenstrukturen

Wintersemester 2021/22

14. Vorlesung

Rot-Schwarz-Bäume

Vorlesungsumfrage

Sehr geehrter Herr Prof. Dr. Wolff,
wir möchten Sie hiermit darüber informieren, dass die Teilnehmer soeben zur Evaluation der Veranstaltung "Algorithmen und Datenstrukturen" eingeladen wurden und die Umfrage nun startet. Folgender Befragungszeitraum ist vorgesehen:

Start der Umfrage: 01.12.2021 23:59:00

Ende der Umfrage: 15.12.2021 23:59:00

Mit freundlichen Grüßen

Ihr EvaSys Administrator

Vorlesungsumfrage

Sehr geehrter Herr Prof. Dr. Wolff,
wir möchten Sie hiermit darüber informieren, dass die Teilnehmer soeben zur Evaluation der Veranstaltung "Algorithmen und Datenstrukturen" eingeladen wurden und die Umfrage nun startet. Folgender Befragungszeitraum ist vorgesehen:

Start der Umfrage: 01.12.2021 23:59:00

Ende der Umfrage: 15.12.2021 23:59:00

Mit freundlichen Grüßen

Ihr EvaSys Administrator

Bitte teilnehmen!

Dynamische Menge

verwaltet Elemente einer
sich ändernden Menge M

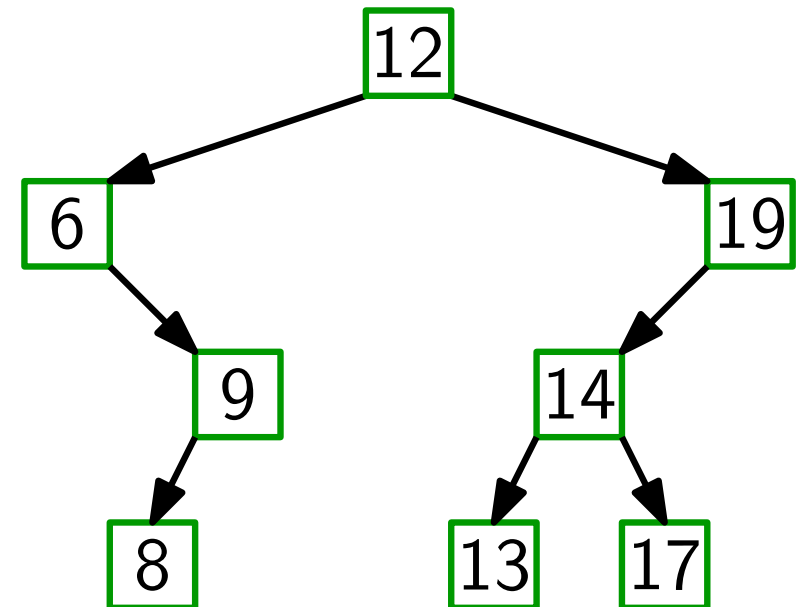


Abstrakter Datentyp	Funktionalität
<code>ptr Insert(key k, info i)</code> <code>Delete(ptr x)</code>	} Änderungen
<code>ptr Search(key k)</code> <code>ptr Minimum()</code> <code>ptr Maximum()</code> <code>ptr Predecessor(ptr x)</code> <code>ptr Successor(ptr x)</code>	

Implementierung: je nachdem...

Binäre Suchbäume

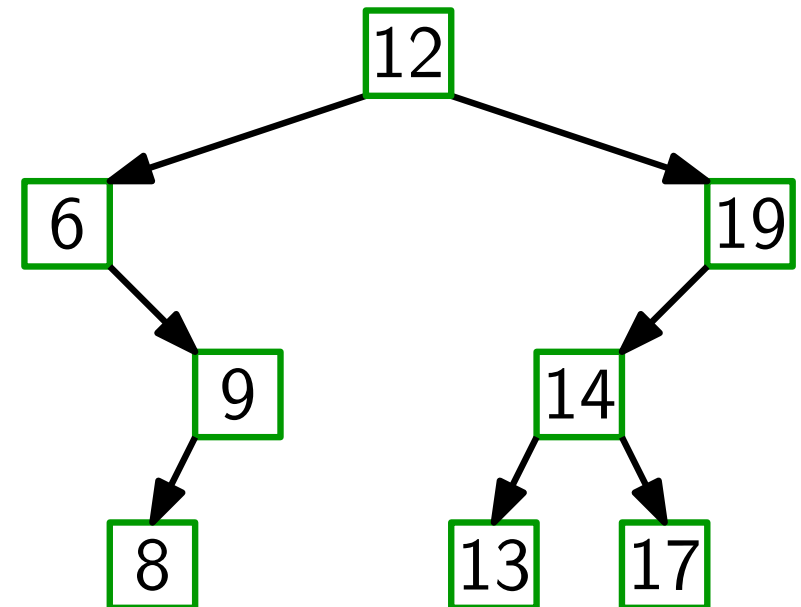
Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $O(h)$ Zeit, wobei h die momentane Höhe des Baums ist.



Binäre Suchbäume

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

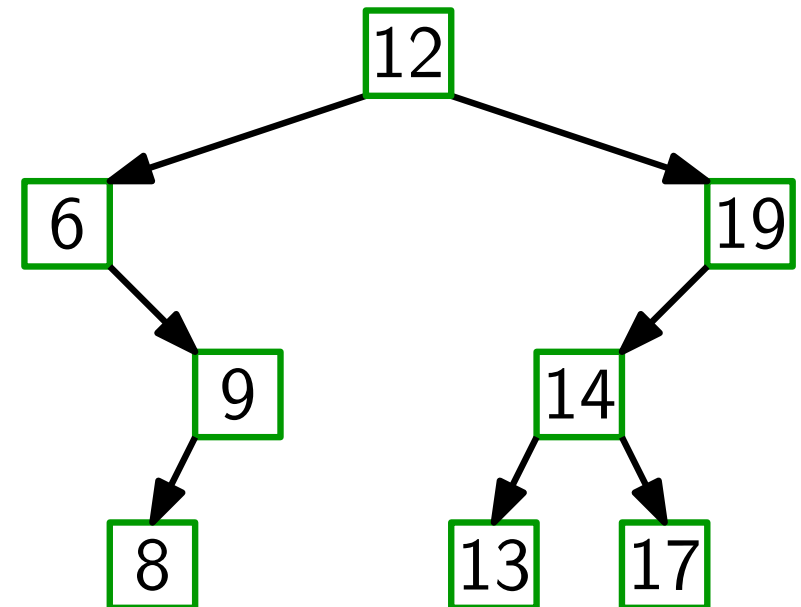
Aber:



Binäre Suchbäume

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

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

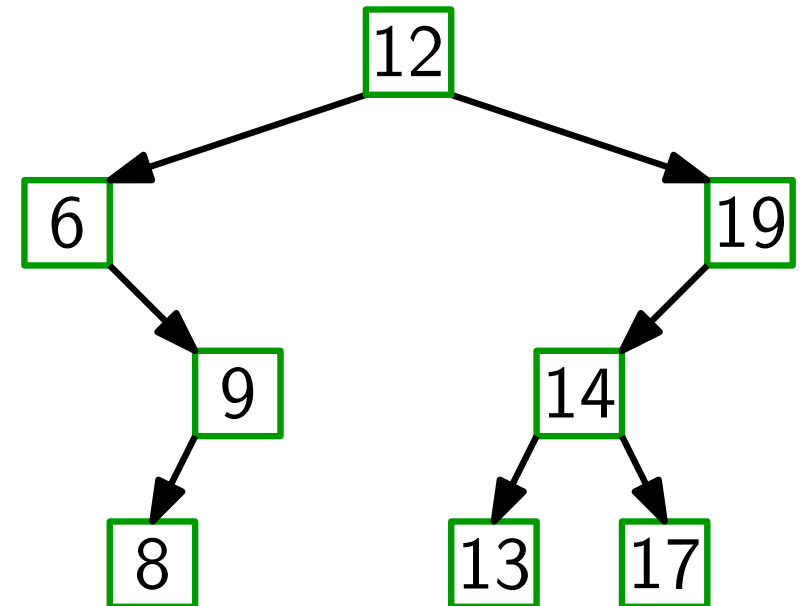


Binäre Suchbäume

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

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

Ziel:

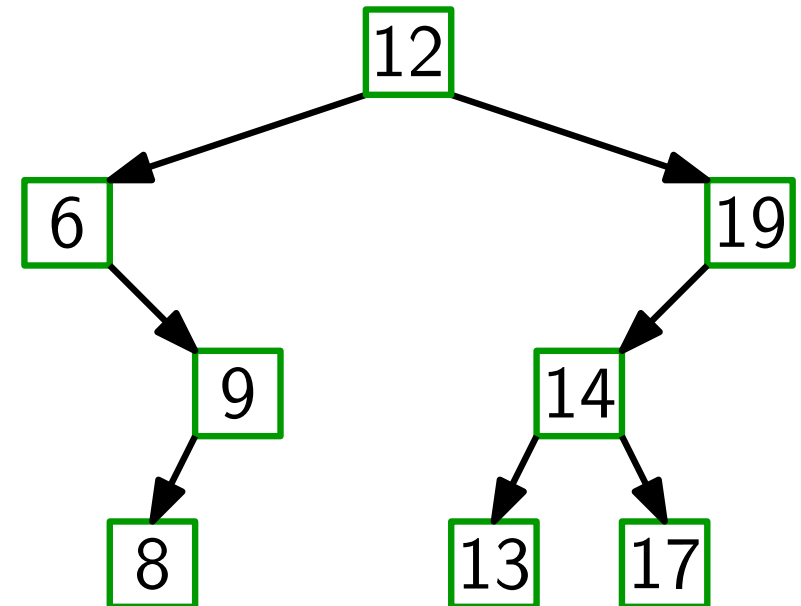


Binäre Suchbäume

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

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

Ziel: Suchbäume *balancieren*!

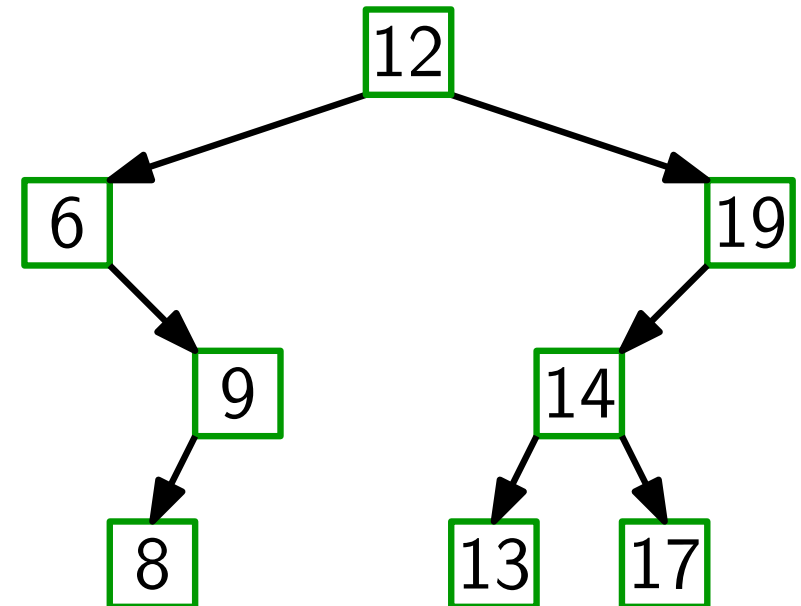


Binäre Suchbäume

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

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

Ziel: Suchbäume *balancieren!*
 $\Rightarrow h \in O(\log n)$



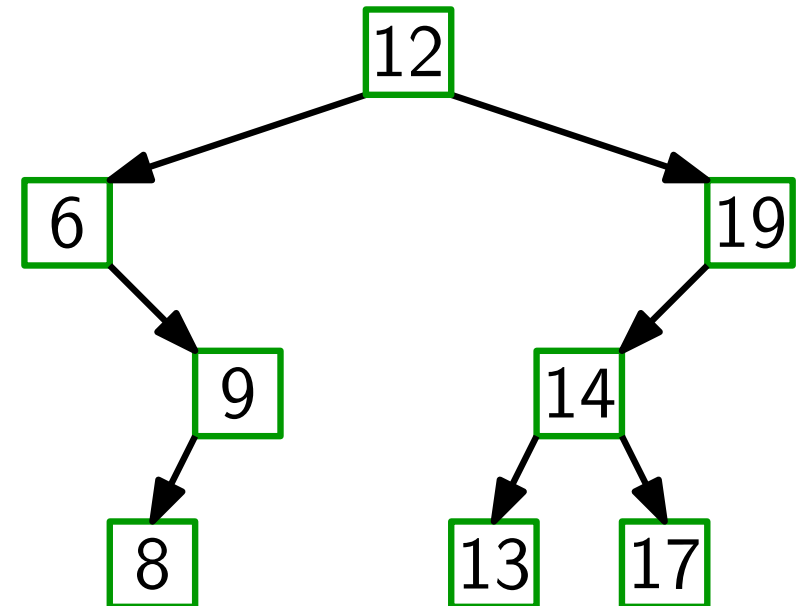
Binäre Suchbäume

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

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

Ziel: Suchbäume *balancieren!*
 $\Rightarrow h \in O(\log n)$

Binärer-Suchbaum-Eigenschaft:



Binäre Suchbäume

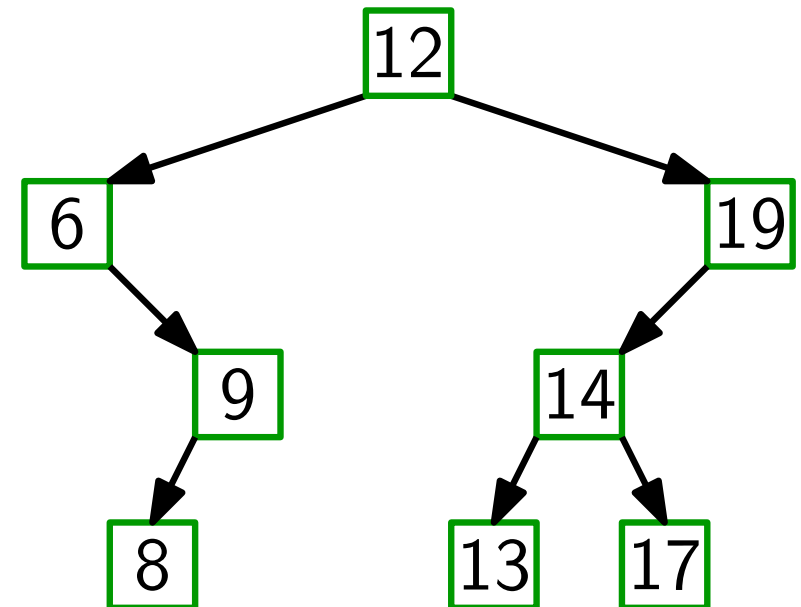
Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $O(h)$ Zeit, wobei h die momentane Höhe des Baums ist.

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

Ziel: Suchbäume *balancieren!*
 $\Rightarrow h \in O(\log n)$

Binärer-Suchbaum-Eigenschaft:

Für jeden Knoten v gilt:



Binäre Suchbäume

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

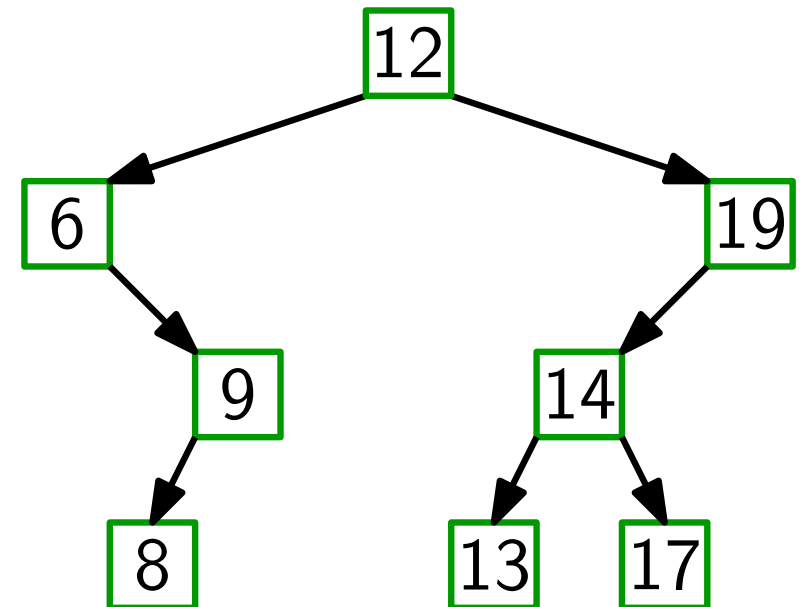
Aber: Im schlechtesten Fall gilt $h \in \Theta(n)$.

Ziel: Suchbäume *balancieren!*
 $\Rightarrow h \in O(\log n)$

Binärer-Suchbaum-Eigenschaft:

Für jeden Knoten v gilt:

alle Knoten im linken Teilbaum von v haben Schlüssel $\leq v.key$



Binäre Suchbäume

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

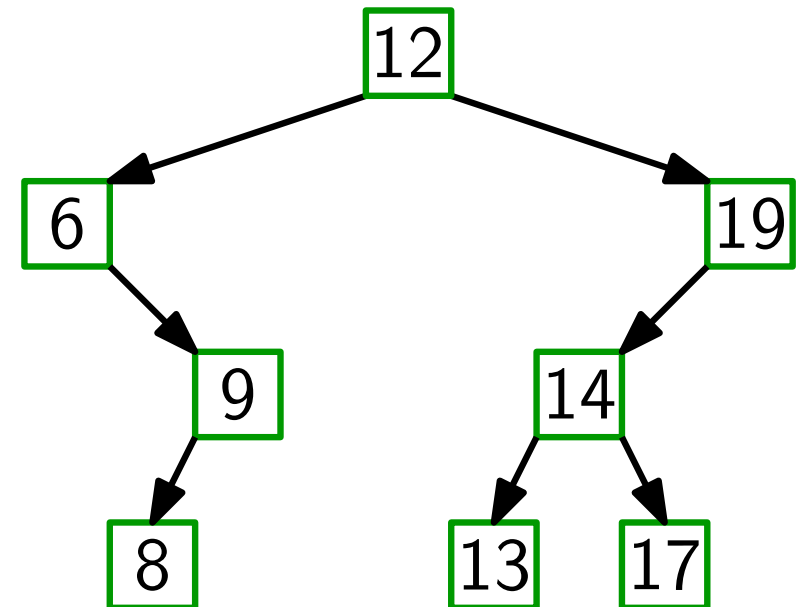
Aber: Im schlechtesten Fall gilt $h \in \Theta(n)$.

Ziel: Suchbäume *balancieren!*
 $\Rightarrow h \in O(\log n)$

Binärer-Suchbaum-Eigenschaft:

Für jeden Knoten v gilt:

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



Balaciermethoden

nach **Gewicht**

für jeden Knoten ist das Gewicht
(= Anzahl der Knoten) von linkem
u. rechtem Teilbaum ungefähr gleich.



Balaciermethoden

nach **Gewicht**

für jeden Knoten ist das Gewicht
(= Anzahl der Knoten) von linkem
u. rechtem Teilbaum ungefähr gleich.

nach **Höhe**

für jeden Knoten ist die Höhe
von linkem und rechtem Teilbaum ungefähr gleich.



Balaciermethoden

nach **Gewicht**

für jeden Knoten ist das Gewicht
(= Anzahl der Knoten) von linkem
u. rechtem Teilbaum ungefähr gleich.

nach **Höhe**

für jeden Knoten ist die Höhe
von linkem und rechtem Teilbaum ungefähr gleich.

nach **Grad**

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



Balaciermethoden

nach **Gewicht**

für jeden Knoten ist das Gewicht
(= Anzahl der Knoten) von linkem
u. rechtem Teilbaum ungefähr gleich.

nach **Höhe**

für jeden Knoten ist die Höhe
von linkem und rechtem Teilbaum ungefähr gleich.

nach **Grad**

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

nach **Knotenfarbe**

jeder Knoten ist entw. „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) von linkem
u. rechtem Teilbaum ungefähr gleich.

nach **Höhe**

AVL-Bäume^{*}

für jeden Knoten ist die Höhe
von linkem und rechtem Teilbaum ungefähr gleich.

nach **Grad**

$(2, 3)$ -Bäume

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

nach **Knotenfarbe**

Rot-Schwarz-Bäume

jeder Knoten ist entw. „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) von linkem
u. rechtem Teilbaum ungefähr gleich.

nach **Höhe**

AVL-Bäume^{*}

für jeden Knoten ist die Höhe
von linkem und rechtem Teilbaum ungefähr gleich.

^{*}) Georgi M. Adelson-Velski & Jewgeni M. Landis, Doklady Akademii Nauk SSSR, 1962

nach **Grad**

$(2, 3)$ -Bäume

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

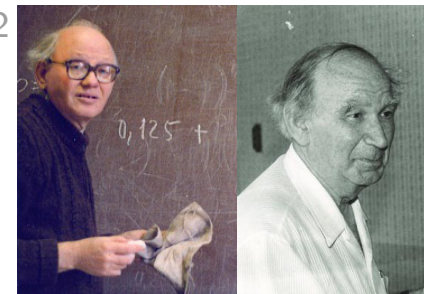
nach **Knotenfarbe**

Rot-Schwarz-Bäume

jeder Knoten ist entw. „gut“ oder „schlecht“; der Anteil
schlechter Knoten darf in keinem Teilbaum zu groß sein.



AV L



1922–2014 1921–1997

Balanciermethoden

Beispiele

nach **Gewicht**

$BB[\alpha]$ -Bäume

für jeden Knoten ist das Gewicht
(= Anzahl der Knoten) von linkem
u. rechtem Teilbaum ungefähr gleich.

nach **Höhe**

AVL-Bäume*

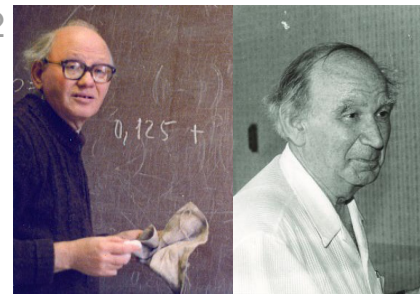
für jeden Knoten ist die Höhe
von linkem und rechtem Teilbaum ungefähr gleich.

*) Georgi M. Adelson-Velski & Jewgeni M. Landis, Doklady Akademii Nauk SSSR, 1962

nach **Grad**

$(2, 3)$ -Bäume

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



1922–2014 1921–1997

nach **Knotenfarbe**

Rot-Schwarz-Bäume

jeder Knoten ist entw. „gut“ oder „schlecht“; der Anteil
schlechter Knoten darf in keinem Teilbaum zu groß sein.

Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden *Rot-Schwarz-Eigenschaften*:

Rot-Schwarz-Bäume: Eigenschaften

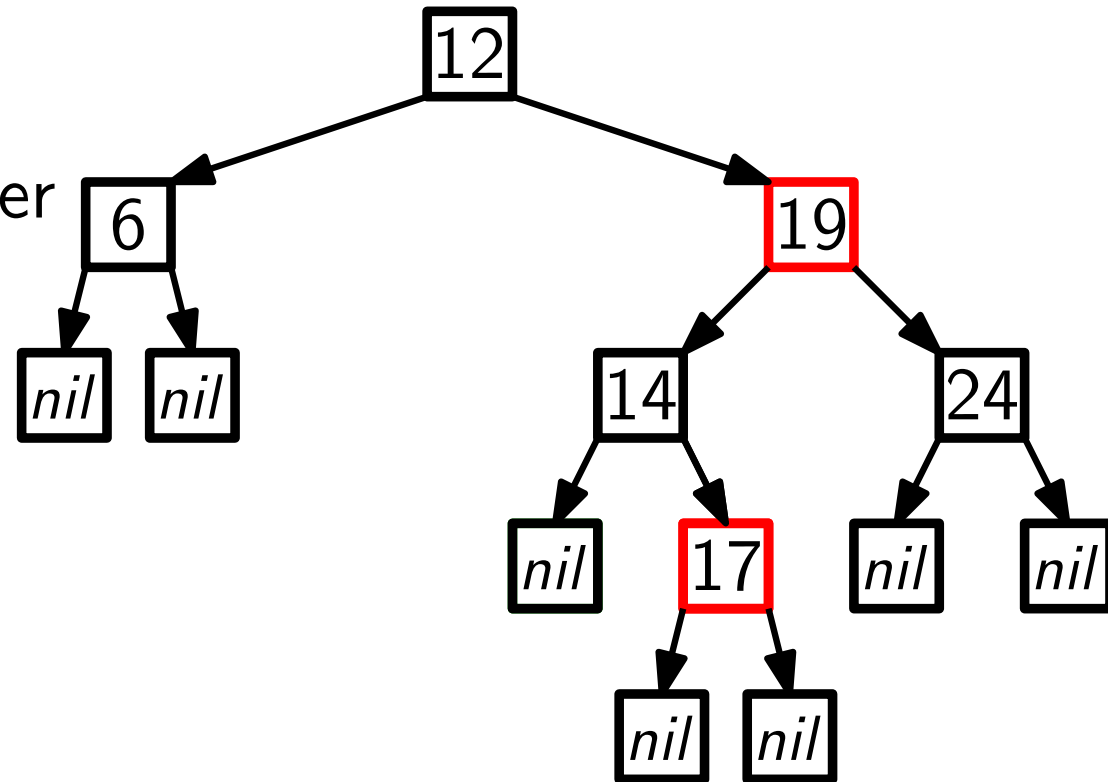
Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden *Rot-Schwarz-Eigenschaften*:

- (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.

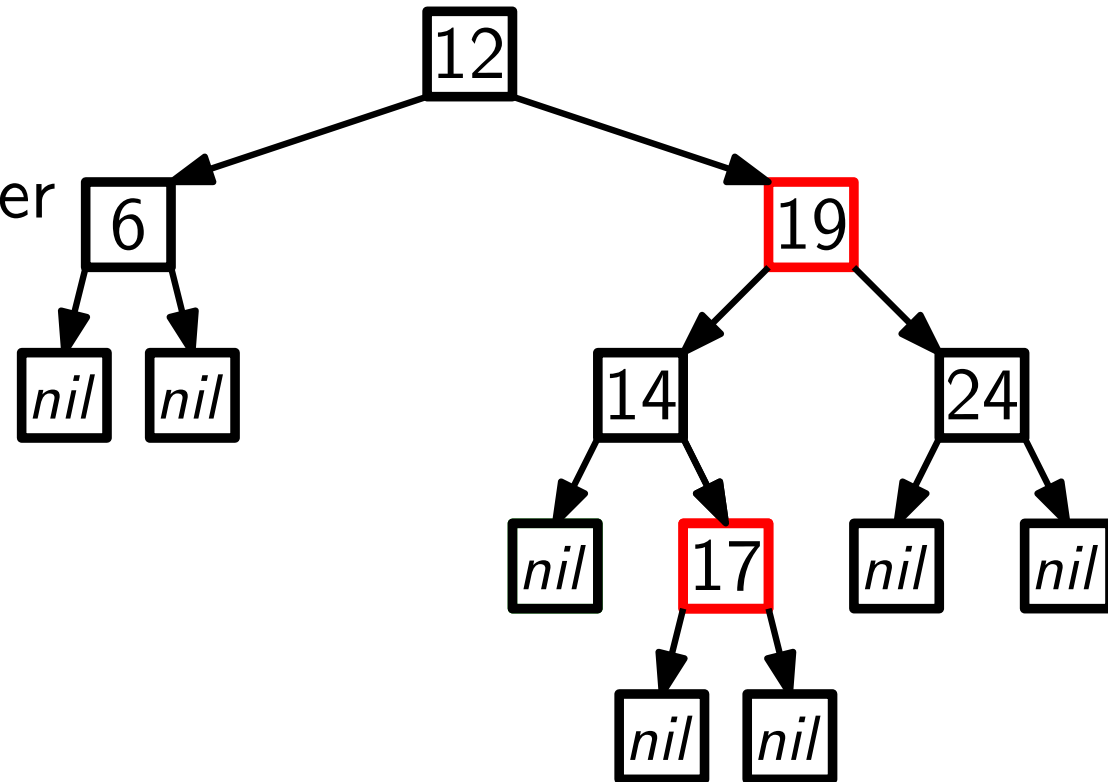


Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden *Rot-Schwarz-Eigenschaften*:

(E1) Jeder Knoten ist entweder rot oder schwarz.

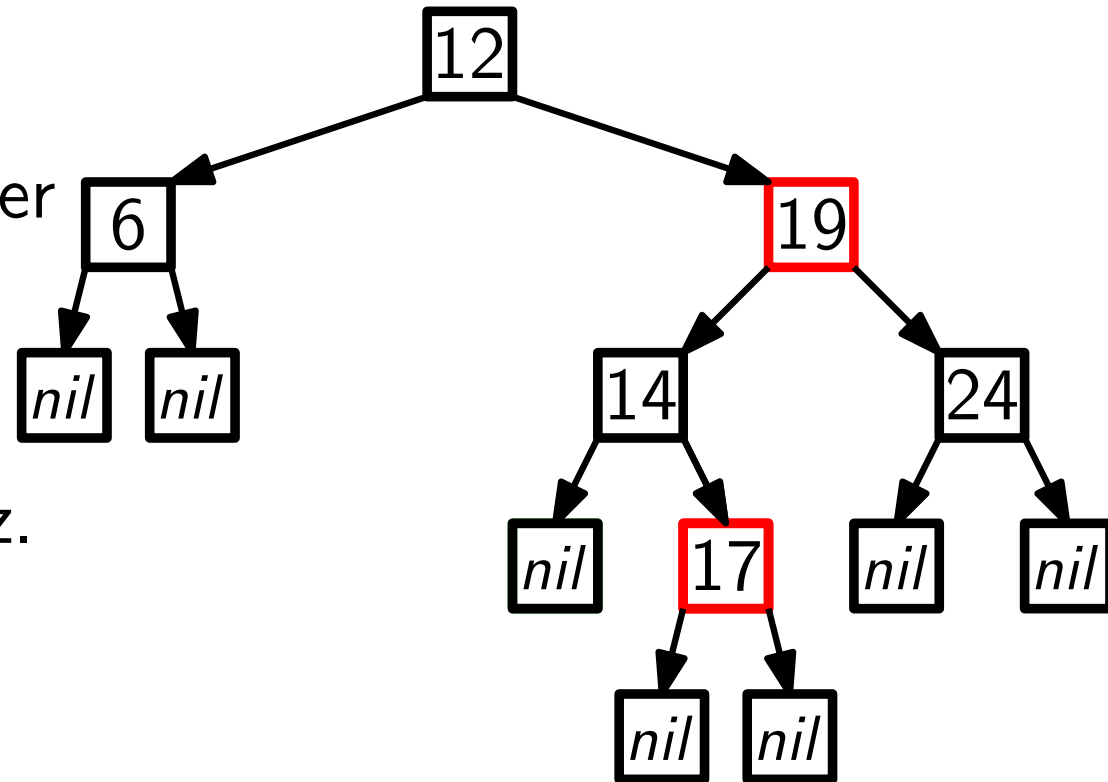
(E2) Die Wurzel ist schwarz.



Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden *Rot-Schwarz-Eigenschaften*:

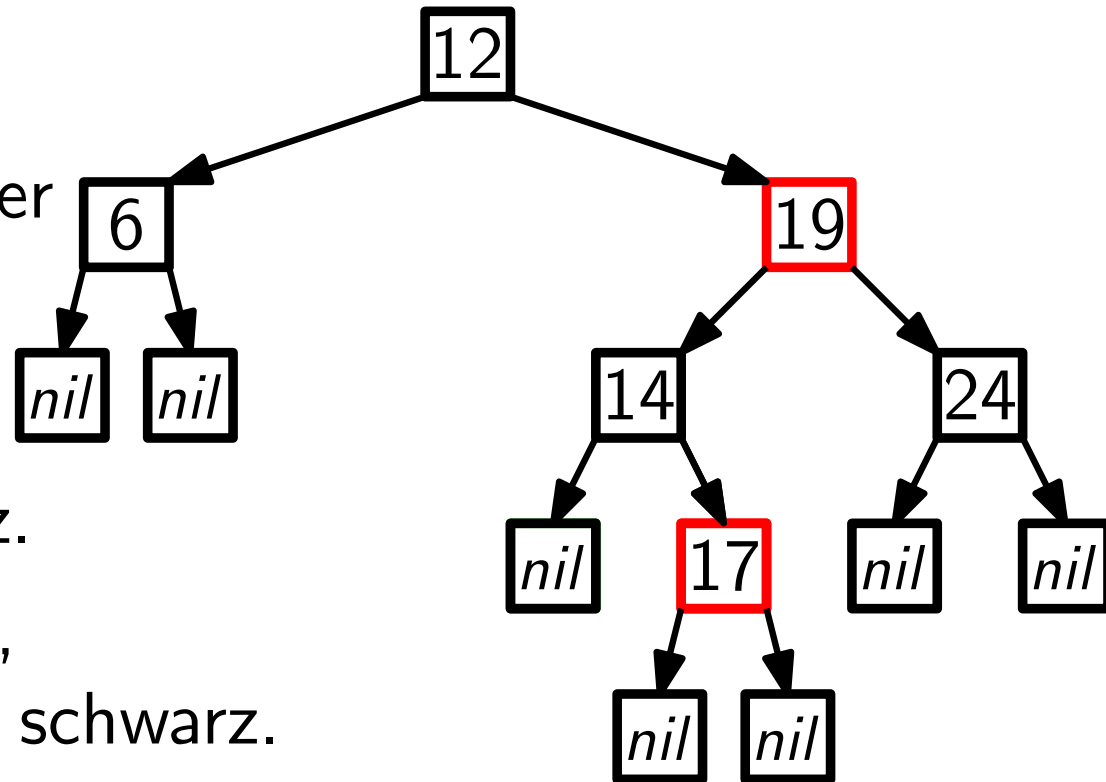
- (E1) Jeder Knoten ist entweder rot oder schwarz.
- (E2) Die Wurzel ist schwarz.
- (E3) Alle Blätter sind schwarz.



Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden *Rot-Schwarz-Eigenschaften*:

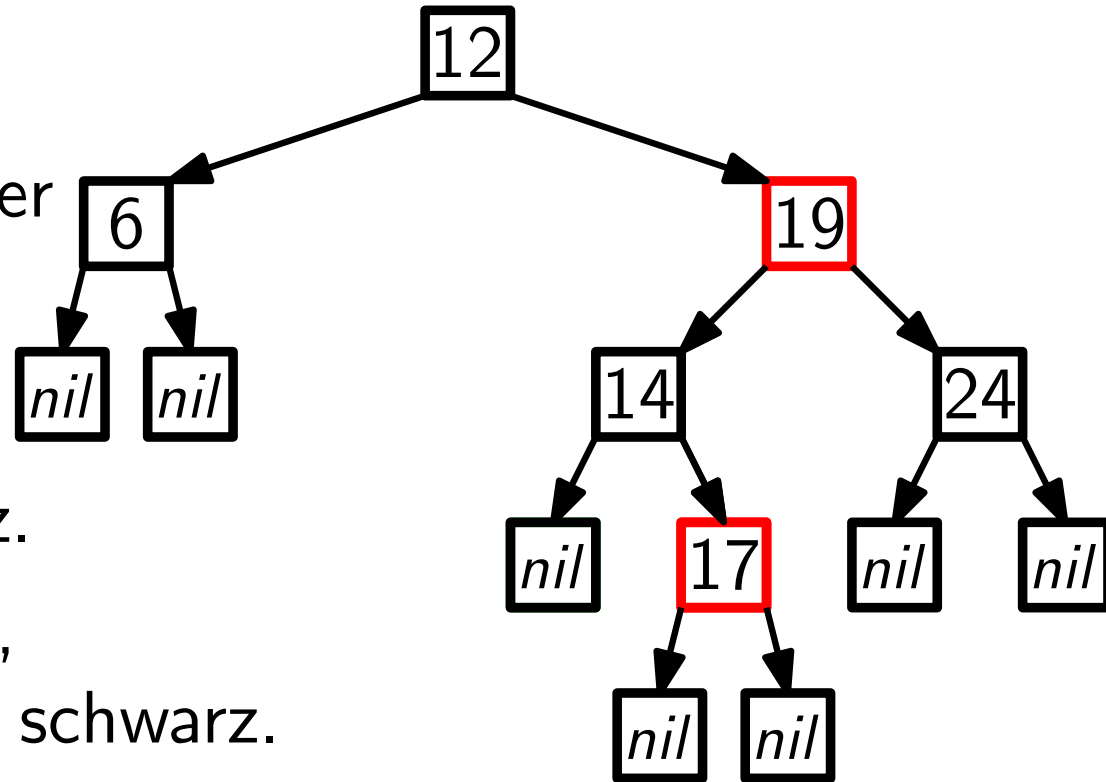
- (E1) Jeder Knoten ist entweder rot oder schwarz.
- (E2) Die Wurzel ist schwarz.
- (E3) Alle Blätter sind schwarz.
- (E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.



Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden *Rot-Schwarz-Eigenschaften*:

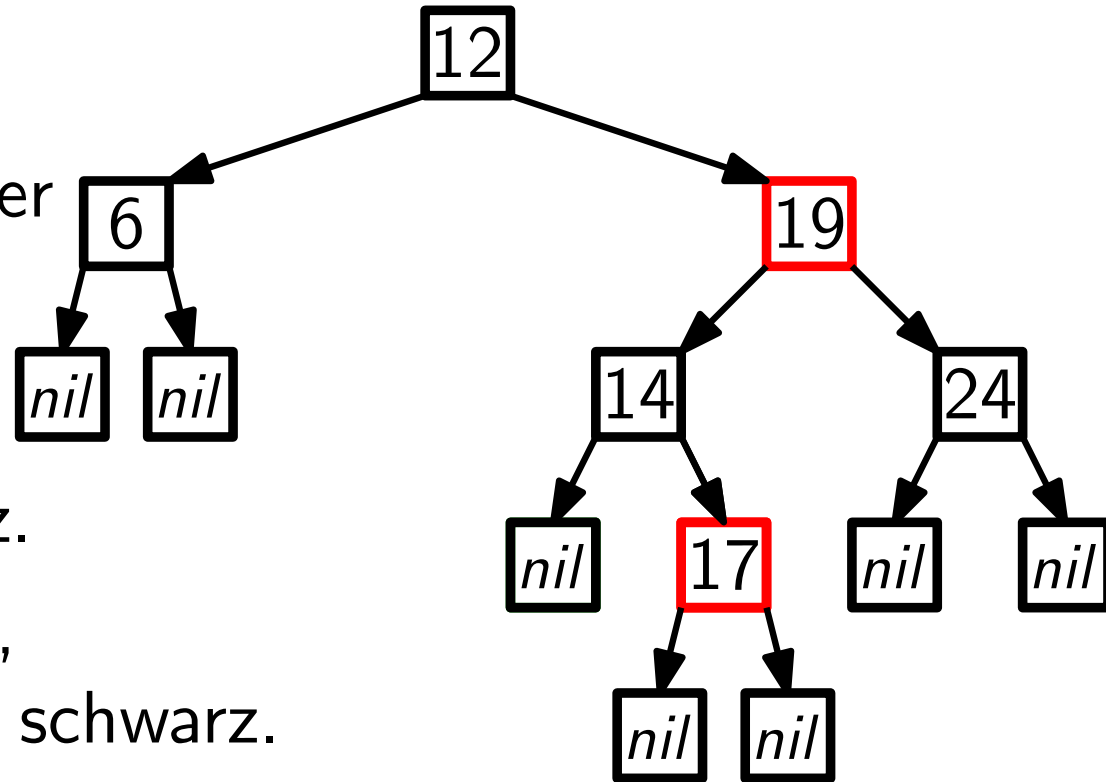
- (E1) Jeder Knoten ist entweder rot oder schwarz.
- (E2) Die Wurzel ist schwarz.
- (E3) Alle Blätter sind schwarz.
- (E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.



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 Blätter sind 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.

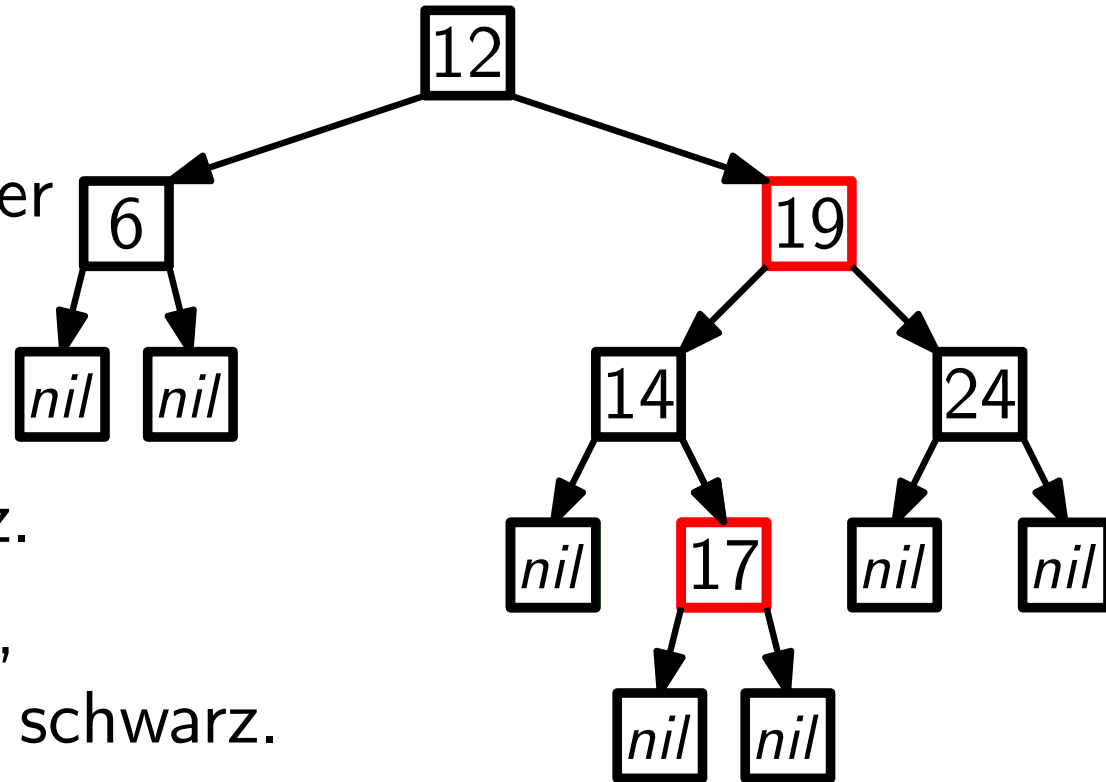


Aus (E4) folgt:

Rot-Schwarz-Bäume: Eigenschaften

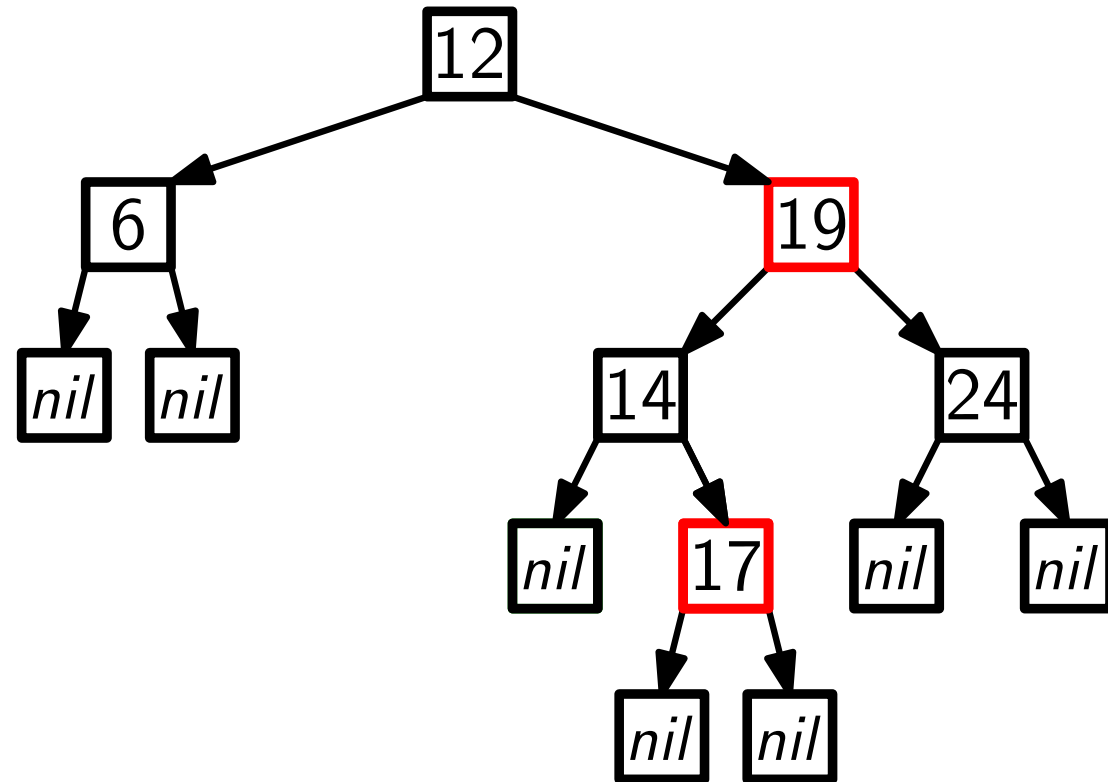
Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden *Rot-Schwarz-Eigenschaften*:

- (E1) Jeder Knoten ist entweder rot oder schwarz.
- (E2) Die Wurzel ist schwarz.
- (E3) Alle Blätter sind 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.



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

Technisches Detail



Technisches Detail

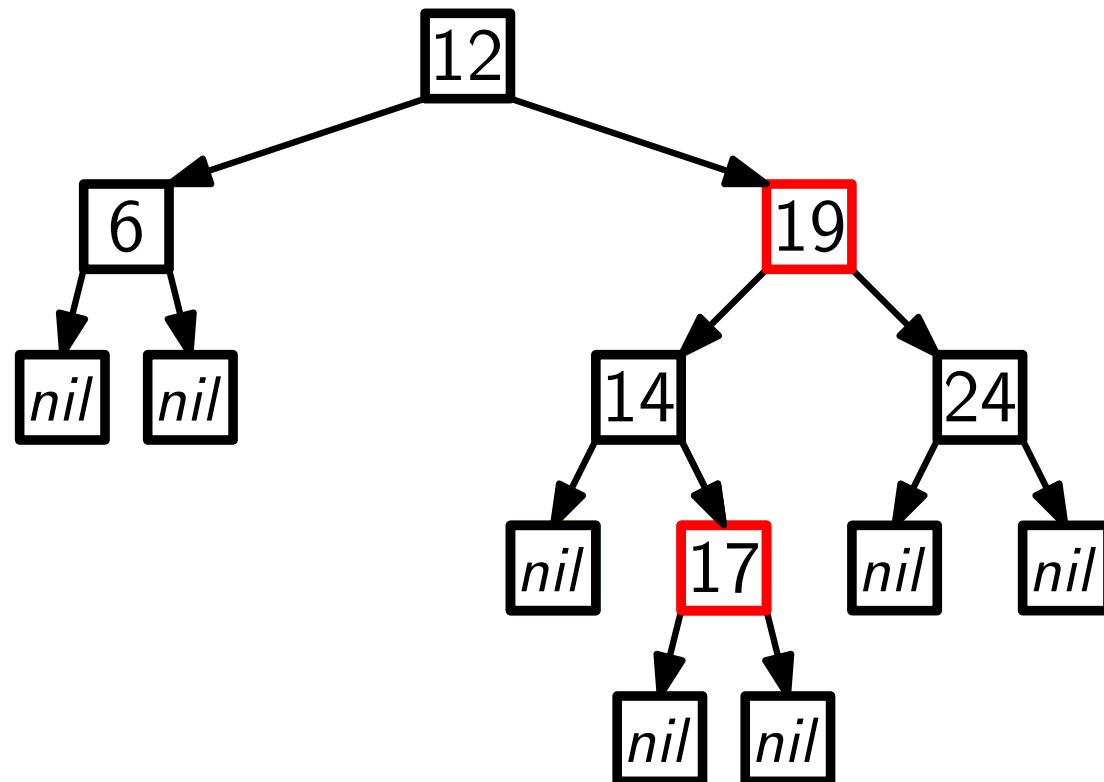
Node

Key *key*

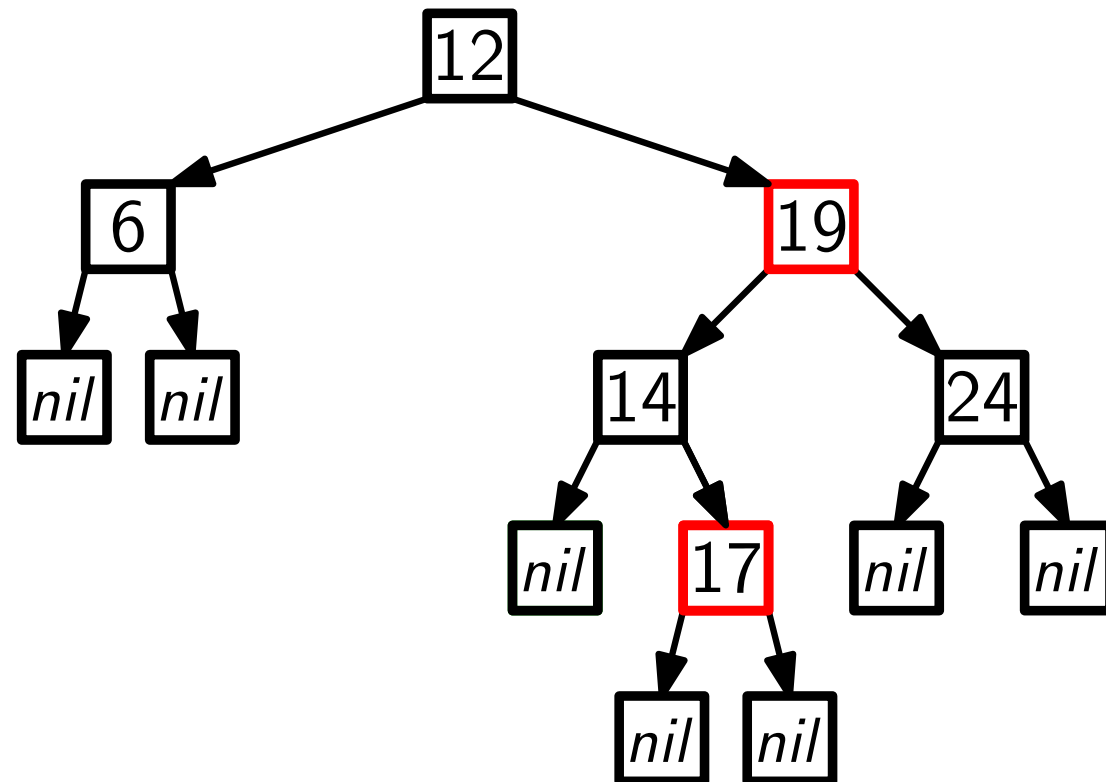
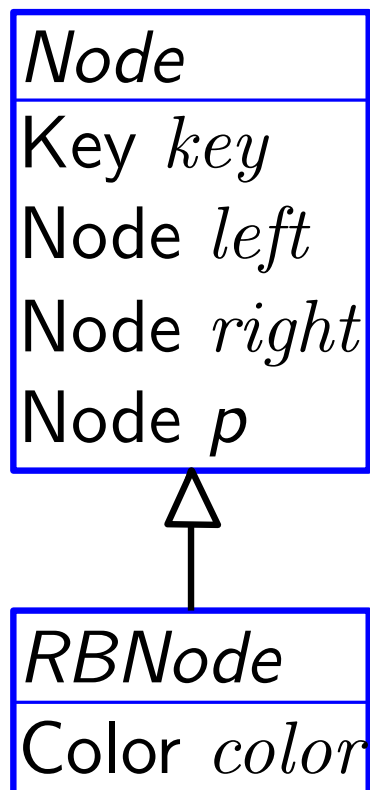
Node *left*

Node *right*

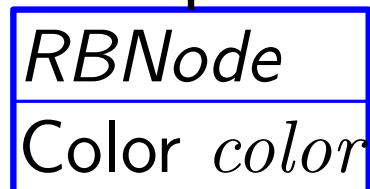
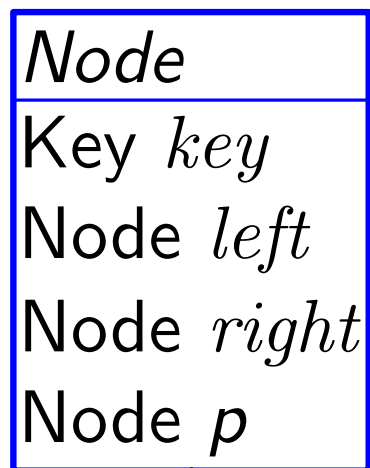
Node *p*



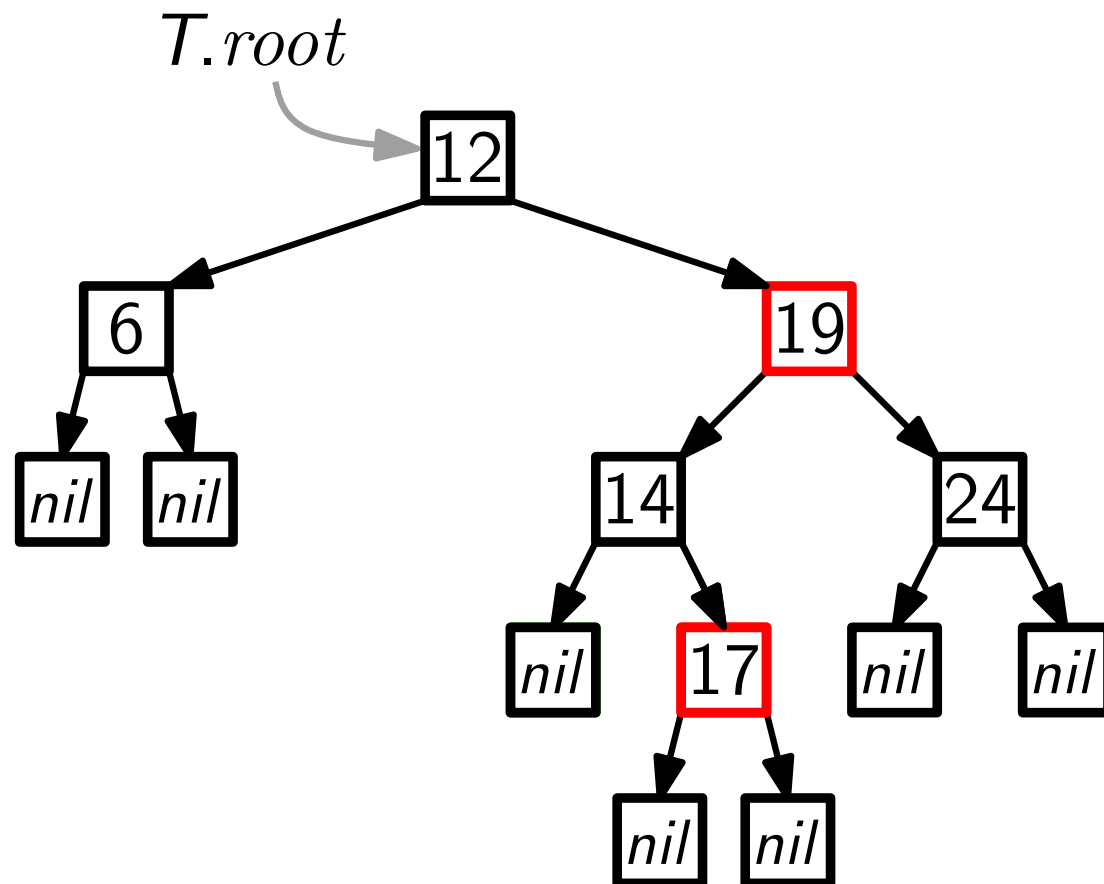
Technisches Detail



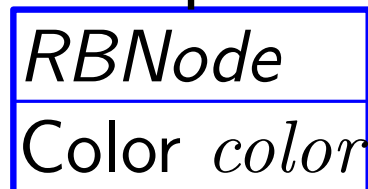
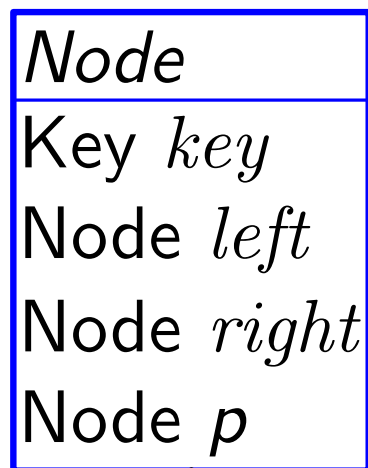
Technisches Detail



T.root

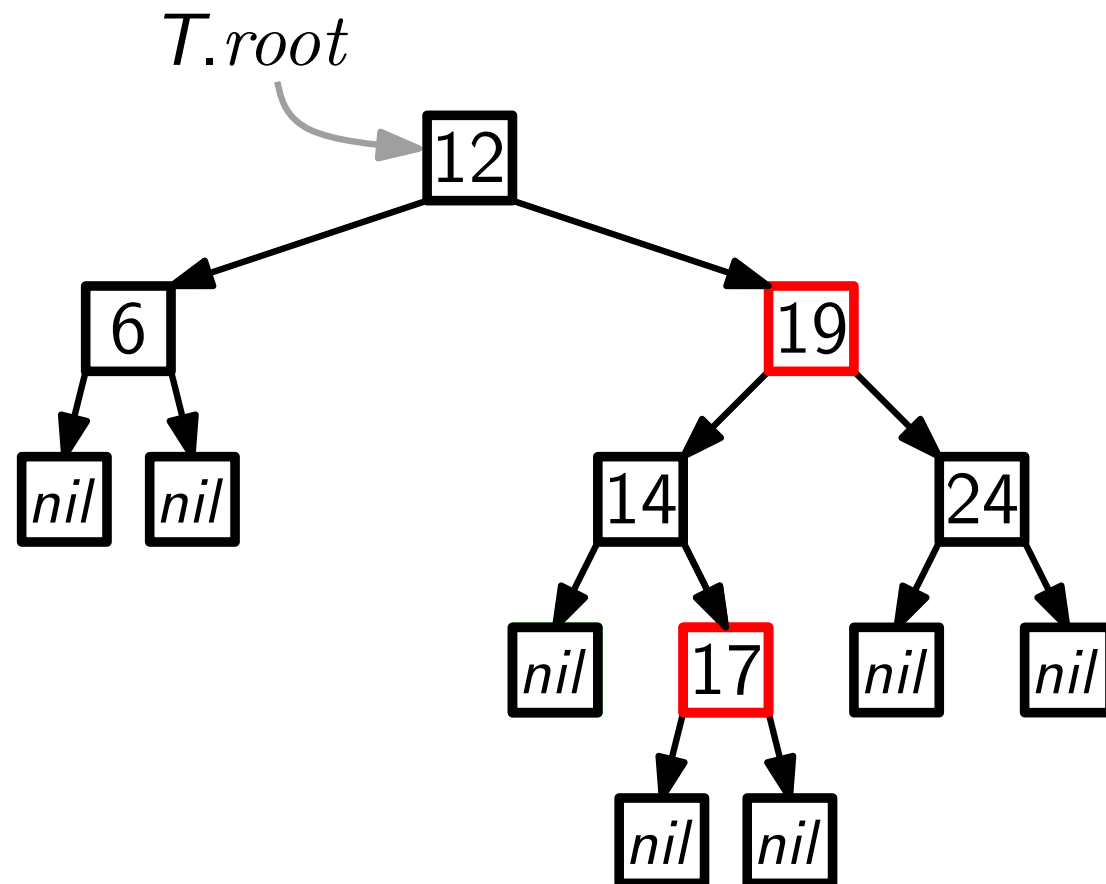


Technisches Detail



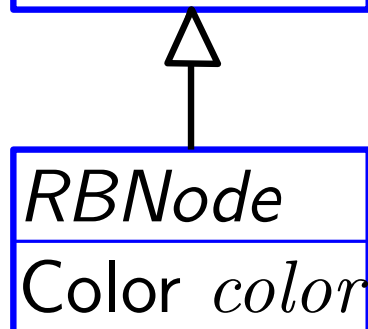
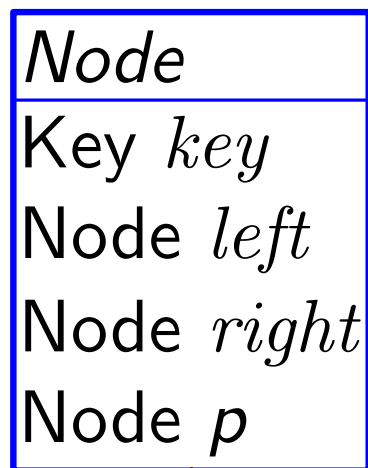
$T.root, T.nil$

Dummy-Knoten (engl. *sentinel*)



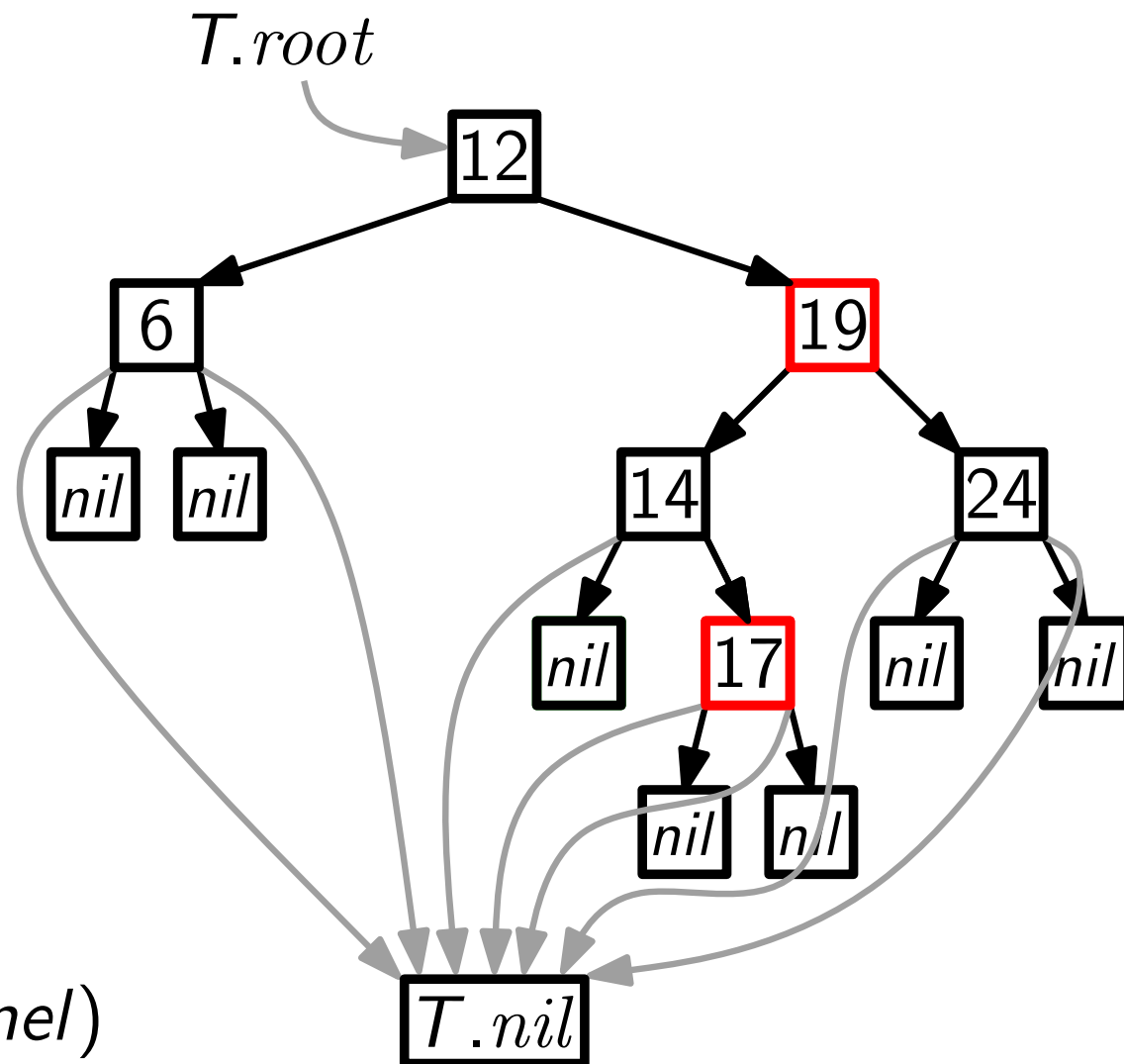
$T.nil$

Technisches Detail

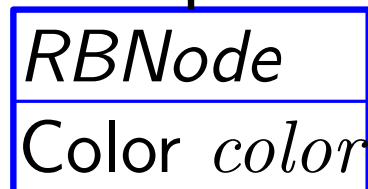
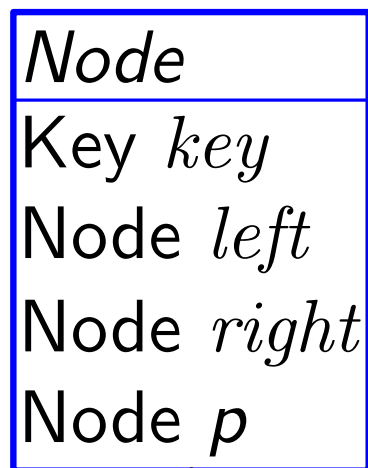


$T.root, T.nil$

Dummy-Knoten (engl. *sentinel*)

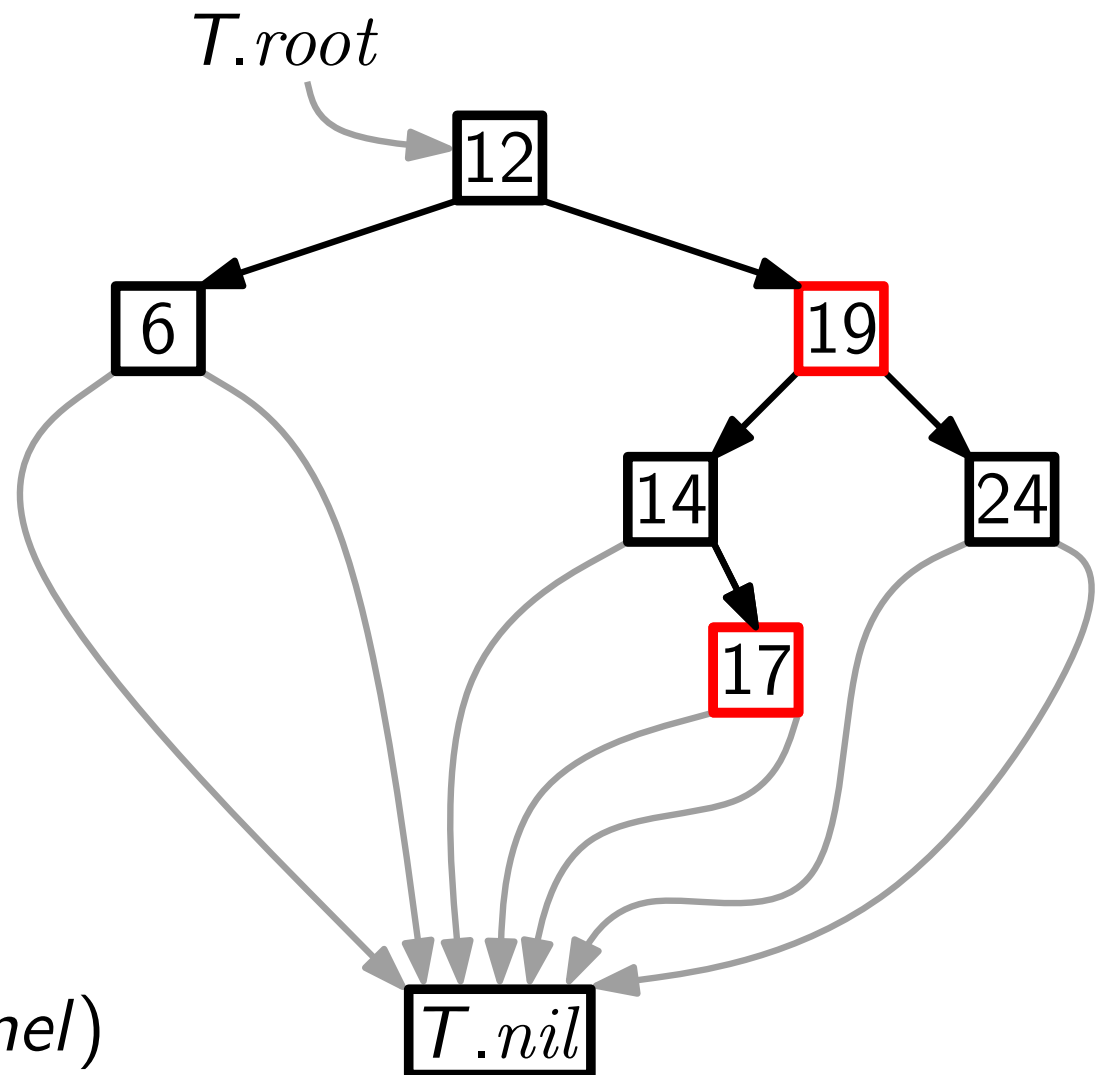


Technisches Detail



$T.root, T.nil$

Dummy-Knoten (engl. *sentinel*)



Technisches Detail

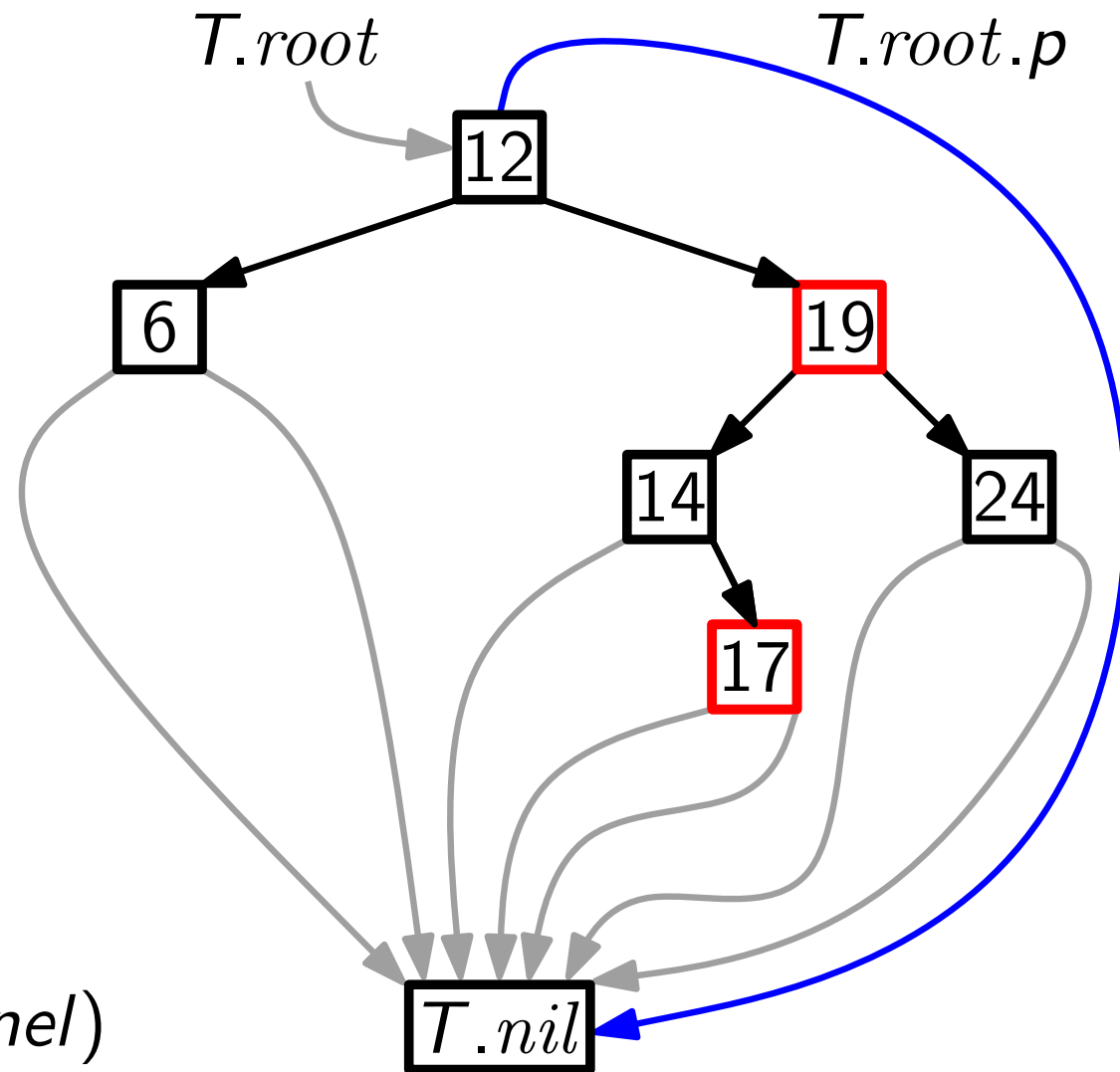
<i>Node</i>
Key <i>key</i>
Node <i>left</i>
Node <i>right</i>
Node <i>p</i>



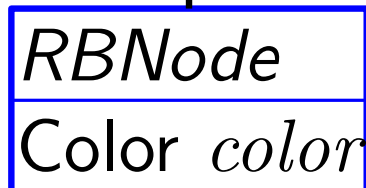
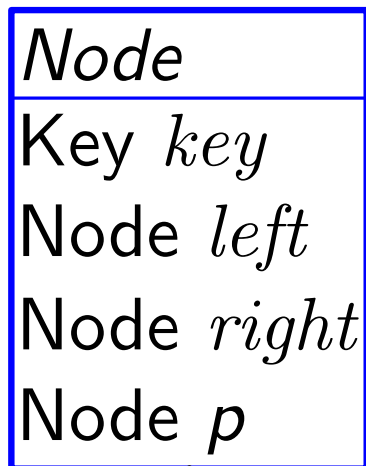
<i>RBNODE</i>
Color <i>color</i>

$T.root, T.nil$

Dummy-Knoten (engl. *sentinel*)



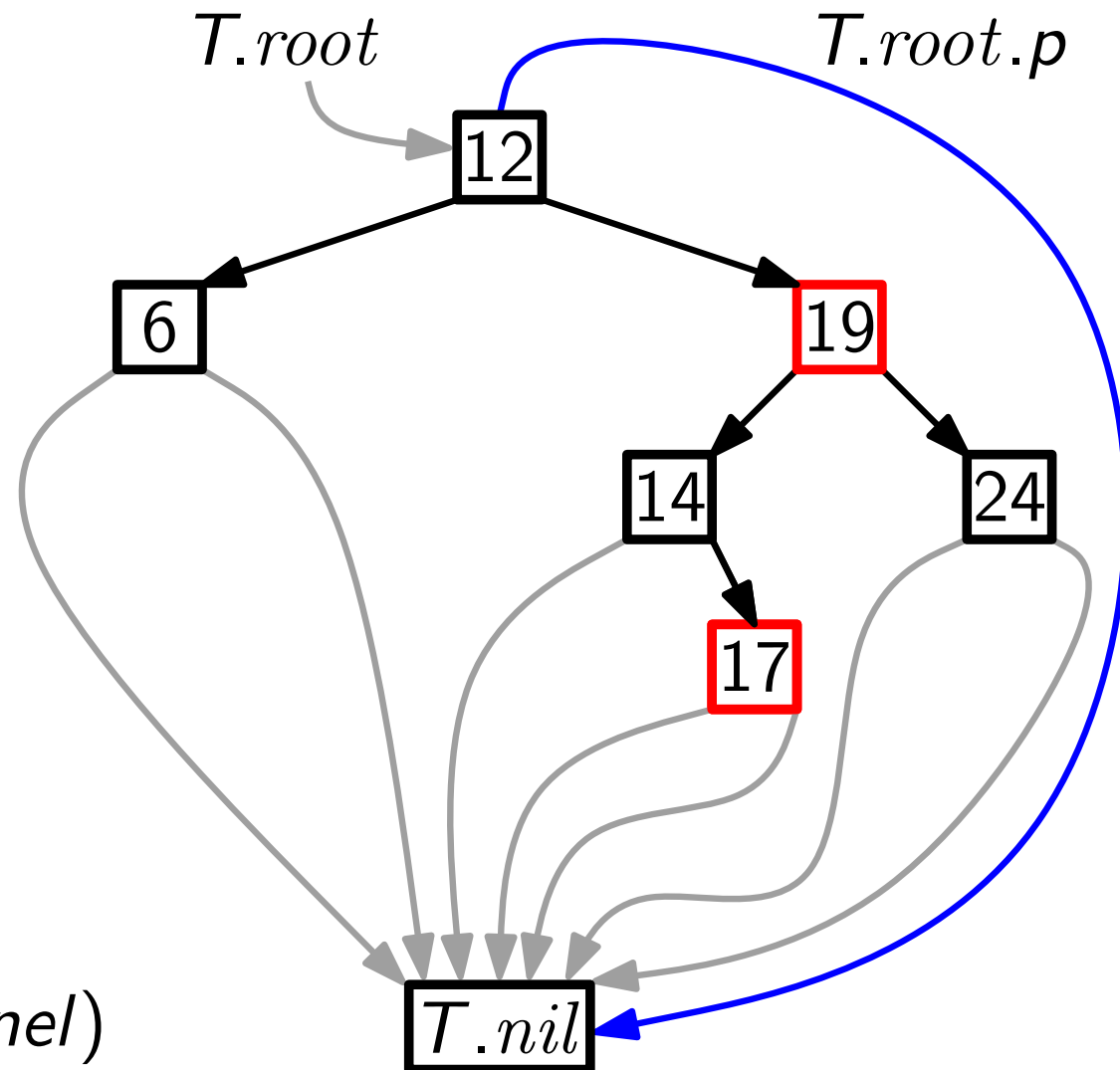
Technisches Detail



$T.root, T.nil$

Dummy-Knoten (engl. *sentinel*)

Zweck: um Baum-Operationen prägnanter aufschreiben zu können.



Technisches Detail

<i>Node</i>
Key <i>key</i>
Node <i>left</i>
Node <i>right</i>
Node <i>p</i>

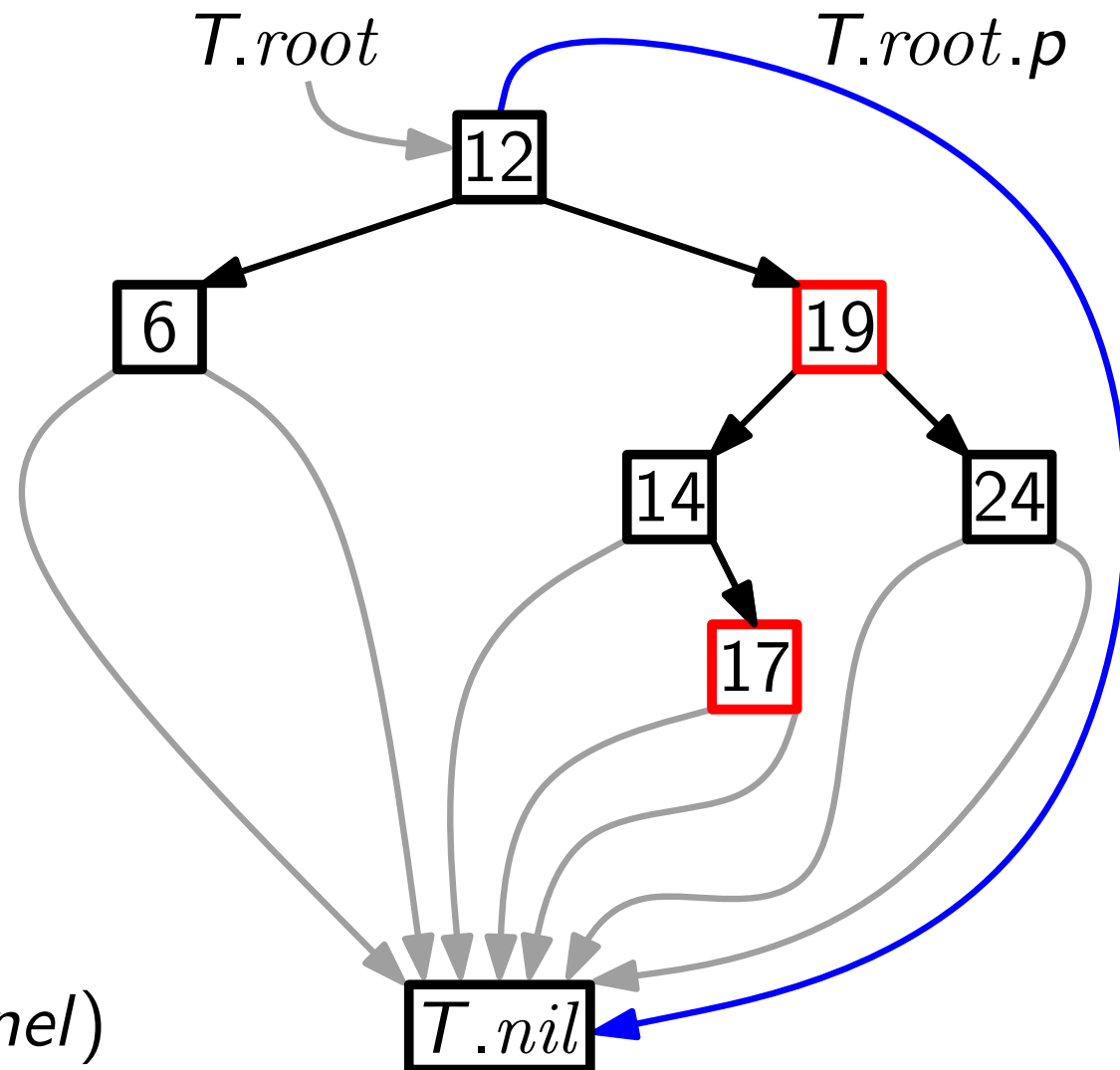


<i>RBNODE</i>
Color <i>color</i>

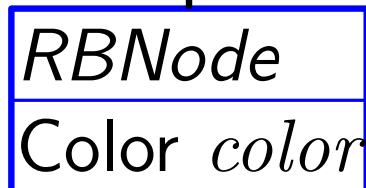
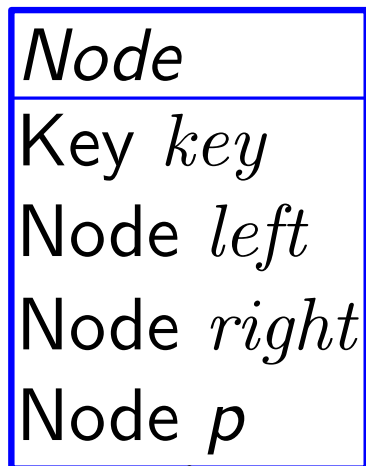
$T.root, T.nil$

Dummy-Knoten (engl. *sentinel*)

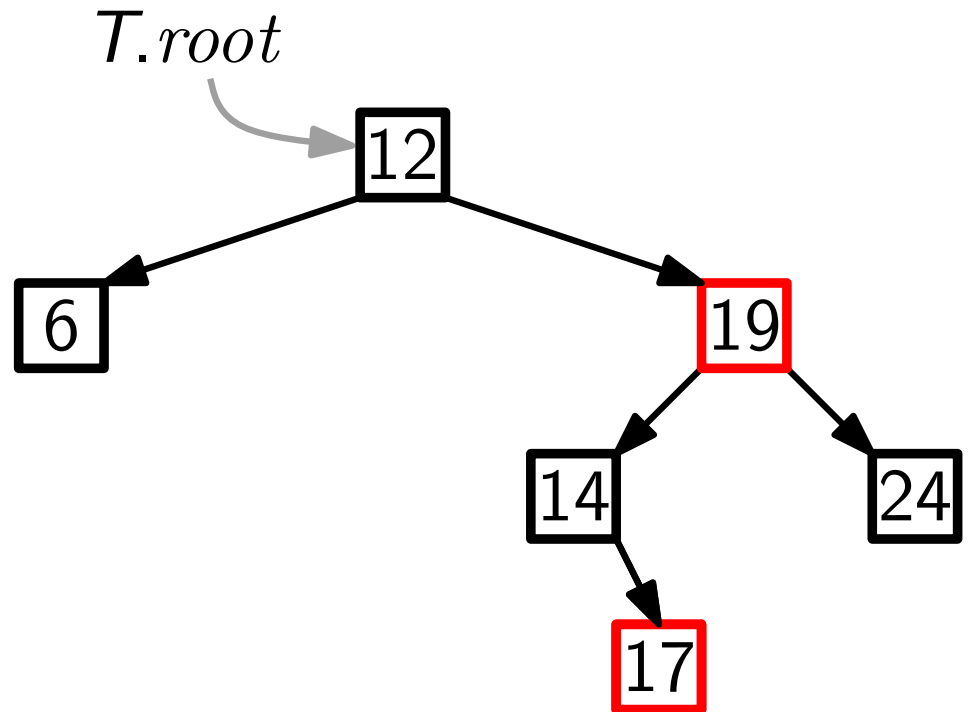
Zweck: um Baum-Operationen prägnanter aufschreiben zu können. (Wir zeichnen den Dummy-Knoten i.A. nicht.)



Technisches Detail



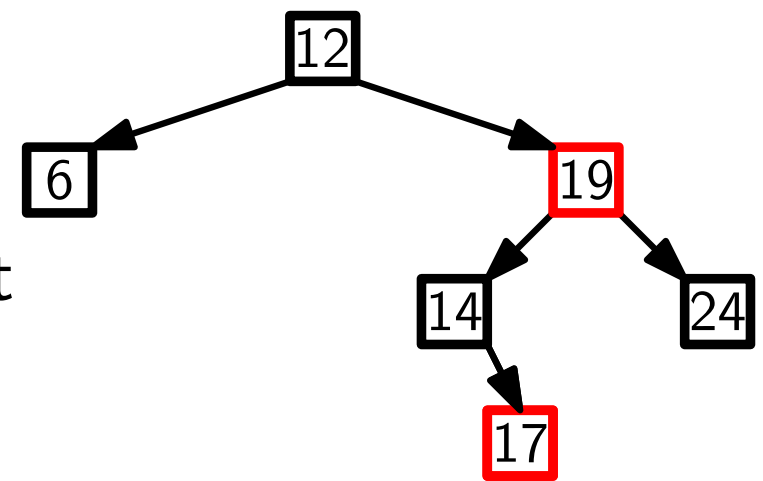
T.root, T.nil



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 Anz. seiner Kanten.

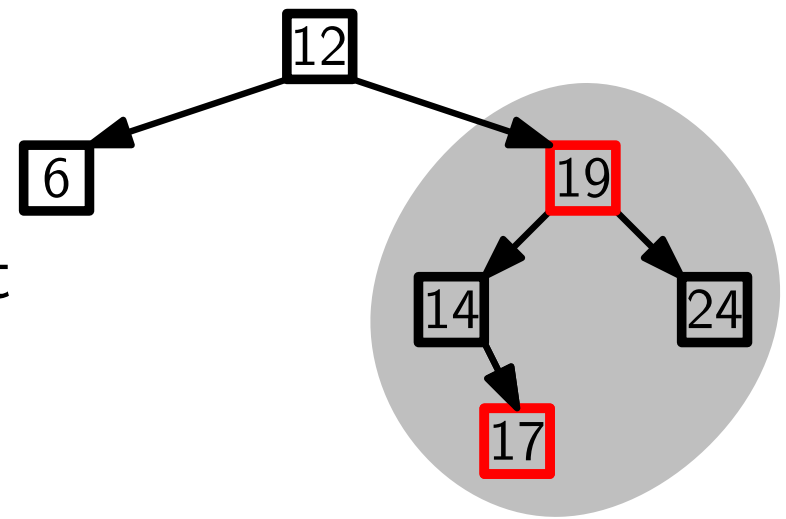


(Schwarz-) Höhe

Definition: Die *Länge* eines Pfades ist die Anz. seiner Kanten.

Definition: Sei B ein Baum.

Knoten u ist *unter* Knoten v , wenn u in dem Teilbaum B_v von B mit Wurzel v enthalten ist.



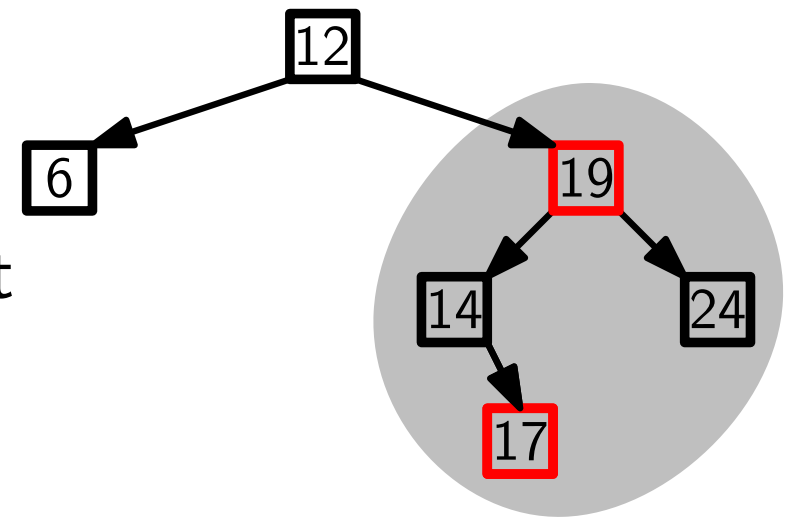
(Schwarz-) Höhe

Definition: Die *Länge* eines Pfades ist die Anz. seiner Kanten.

Definition: Sei B ein Baum.

Knoten u ist *unter* Knoten v , wenn u in dem Teilbaum B_v von B mit Wurzel v enthalten ist.

Beispiel: „17“ ist unter „19“



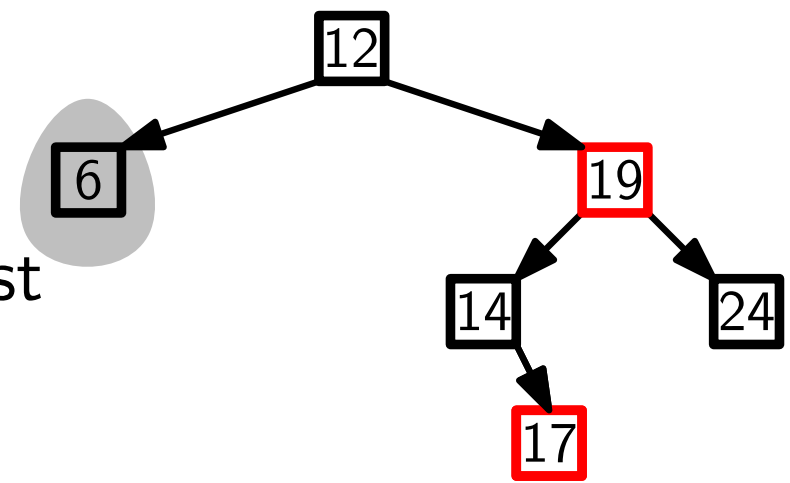
(Schwarz-) Höhe

Definition: Die *Länge* eines Pfades ist die Anz. seiner Kanten.

Definition: Sei B ein Baum.

Knoten u ist *unter* Knoten v , wenn u in dem Teilbaum B_v von B mit Wurzel v enthalten ist.

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



(Schwarz-) Höhe

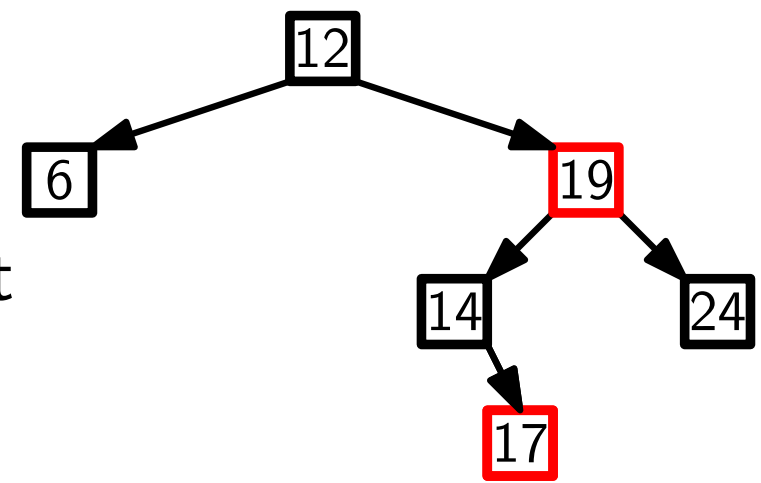
Definition: Die *Länge* eines Pfades ist die Anz. seiner Kanten.

Definition: Sei B ein Baum.

Knoten u ist *unter* Knoten v , wenn u in dem Teilbaum B_v von B 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 .



(Schwarz-) Höhe

Definition: Die *Länge* eines Pfades ist die Anz. seiner Kanten.

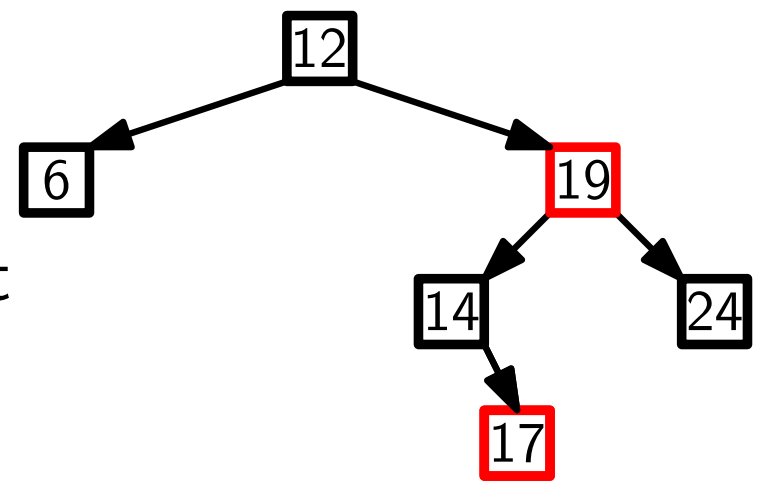
Definition: Sei B ein Baum.

Knoten u ist *unter* Knoten v , wenn u in dem Teilbaum B_v von B mit Wurzel v enthalten ist.

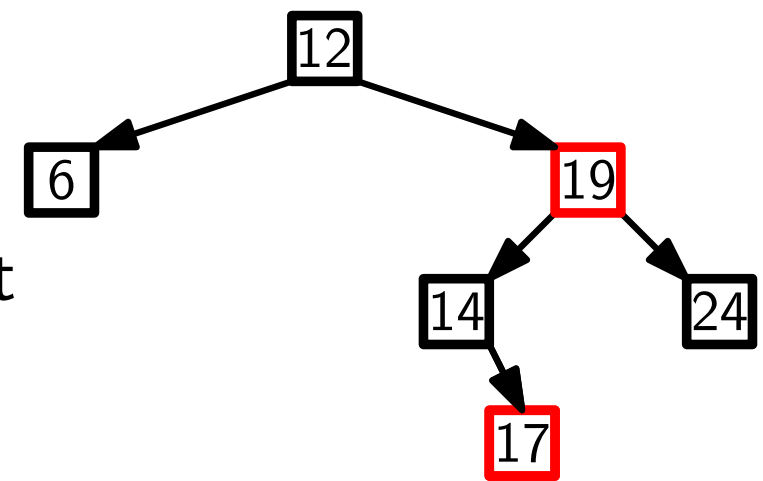
Beispiel: „17“ ist unter „19“, „14“ ist *nicht* unter „6“.

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

Definition! Die *Höhe* $\text{Höhe}(v)$ eines Knotens v ist die Anz. der Knoten (ohne v) auf dem längsten Pfad zu einem Blatt (inkl. Blatt) in B_v .



(Schwarz-) Höhe



Definition: Die *Länge* eines Pfades ist die Anz. seiner Kanten.

Definition: Sei B ein Baum.

Knoten u ist *unter* Knoten v , wenn u in dem Teilbaum B_v von B mit Wurzel v enthalten ist.

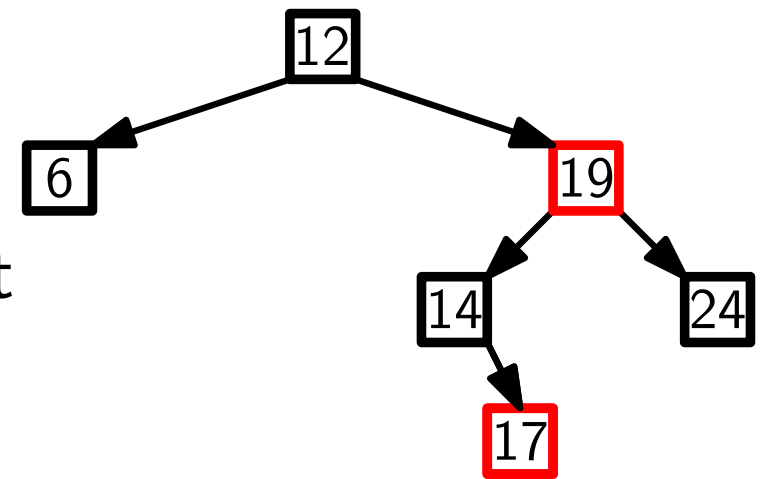
Beispiel: „17“ ist unter „19“, „14“ ist *nicht* unter „6“.

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

Definition! Die *Höhe* $\text{Höhe}(v)$ eines Knotens v ist die Anz. der Knoten (ohne v) auf dem längsten Pfad zu einem Blatt (inkl. Blatt) in B_v .

Beispiel: „12“ hat Höhe

(Schwarz-) Höhe



Definition: Die *Länge* eines Pfades ist die Anz. seiner Kanten.

Definition: Sei B ein Baum.

Knoten u ist *unter* Knoten v , wenn u in dem Teilbaum B_v von B mit Wurzel v enthalten ist.

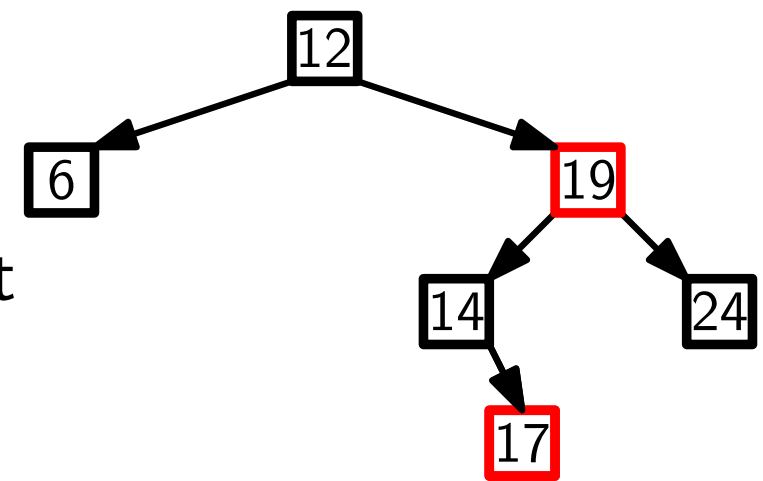
Beispiel: „17“ ist unter „19“, „14“ ist *nicht* unter „6“.

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

Definition! Die *Höhe* $\text{Höhe}(v)$ eines Knotens v ist die Anz. der Knoten (ohne v) auf dem längsten Pfad zu einem Blatt (inkl. Blatt) in B_v .

Beispiel: „12“ hat Höhe 4 (!)

(Schwarz-) Höhe



Definition: Die *Länge* eines Pfades ist die Anz. seiner Kanten.

Definition: Sei B ein Baum.

Knoten u ist *unter* Knoten v , wenn u in dem Teilbaum B_v von B mit Wurzel v enthalten ist.

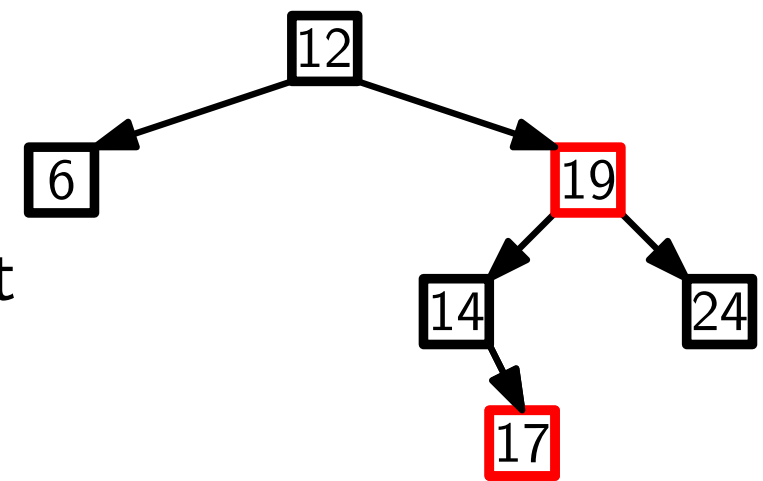
Beispiel: „17“ ist unter „19“, „14“ ist *nicht* unter „6“.

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

Definition: Die **Schwarz-Höhe** $shöhe(v)$ eines Knotens v ist die Anz. der **schwarzen** Knoten (ohne v) auf **jedem** ~~längsten~~ Pfad zu einem Blatt (inkl. Blatt) in B_v .

Beispiel: „12“ hat Höhe 4 (!)

(Schwarz-) Höhe



Definition: Die *Länge* eines Pfades ist die Anz. seiner Kanten.

Definition: Sei B ein Baum.

Knoten u ist *unter* Knoten v , wenn u in dem Teilbaum B_v von B mit Wurzel v enthalten ist.

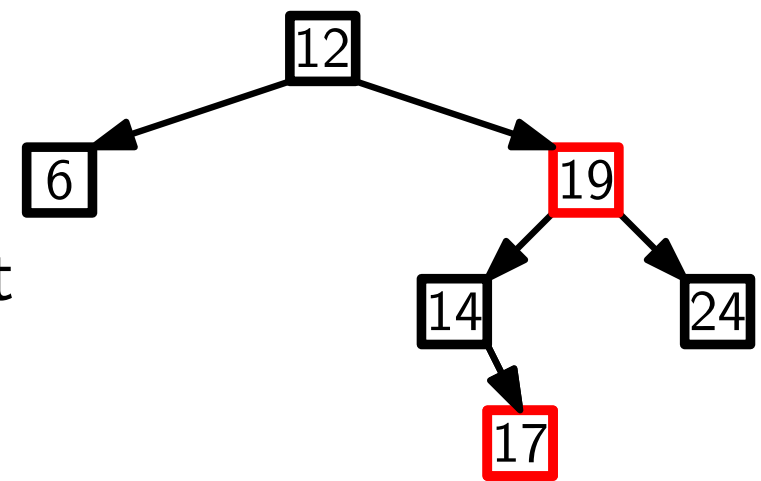
Beispiel: „17“ ist unter „19“, „14“ ist *nicht* unter „6“.

Definition: Die *Höhe* eines Knotens v ist die Länge eines längsten Pfades von v zu einem Blatt unter v . wohl-definiert wg. (E5)

Definition: Die **Schwarz-Höhe** $shöhe(v)$ eines Knotens v ist die Anz. der **schwarzen** Knoten (ohne v) auf **jedem** ~~längsten~~ Pfad zu einem Blatt (inkl. Blatt) in B_v .

Beispiel: „12“ hat Höhe 4 (!)

(Schwarz-) Höhe



Definition: Die *Länge* eines Pfades ist die Anz. seiner Kanten.

Definition: Sei B ein Baum.

Knoten u ist *unter* Knoten v , wenn u in dem Teilbaum B_v von B mit Wurzel v enthalten ist.

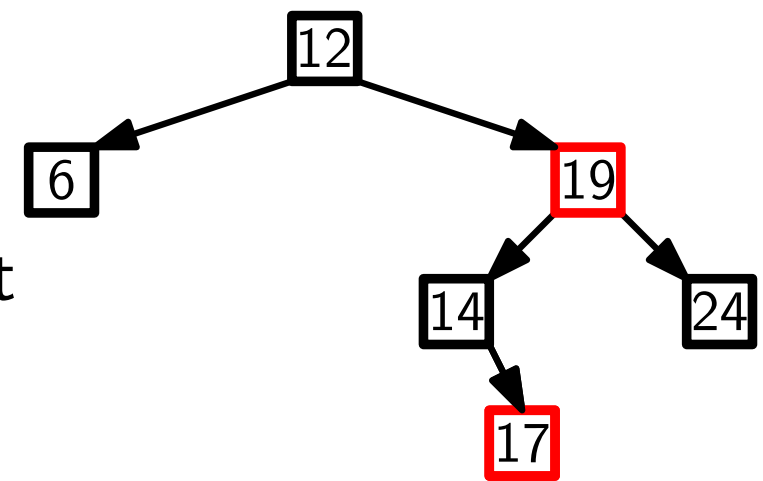
Beispiel: „17“ ist unter „19“, „14“ ist *nicht* unter „6“.

Definition: Die *Höhe* eines Knotens v ist die Länge eines längsten Pfades von v zu einem Blatt unter v . wohl-definiert wg. (E5)

Definition: Die **Schwarz-Höhe** $sHöhe(v)$ eines Knotens v ist die Anz. der **schwarzen** Knoten (ohne v) auf **jedem** ~~längsten~~ Pfad zu einem Blatt (inkl. Blatt) in B_v .

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

(Schwarz-) Höhe



Definition: Die *Länge* eines Pfades ist die Anz. seiner Kanten.

Definition: Sei B ein Baum.

Knoten u ist *unter* Knoten v , wenn u in dem Teilbaum B_v von B mit Wurzel v enthalten ist.

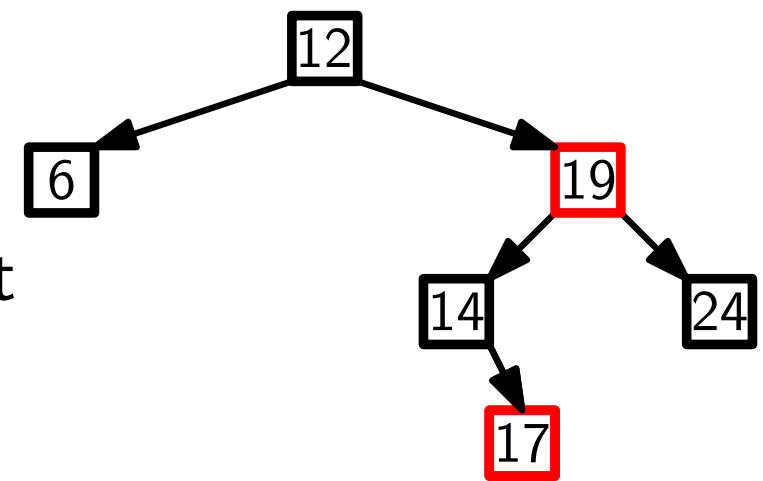
Beispiel: „17“ ist unter „19“, „14“ ist *nicht* unter „6“.

Definition: Die *Höhe* eines Knotens v ist die Länge eines längsten Pfads von v zu einem Blatt unter v . wohl-definiert wg. (E5)

Definition: Die **Schwarz-Höhe** $sHöhe(v)$ eines Knotens v ist die Anz. der **schwarzen** Knoten (ohne v) auf **jedem** ~~längsten~~ Pfad zu einem Blatt (inkl. Blatt) in B_v .

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

(Schwarz-) Höhe



Definition: Die *Länge* eines Pfades ist die Anz. seiner Kanten.

Definition: Sei B ein Baum.

Knoten u ist *unter* Knoten v , wenn u in dem Teilbaum B_v von B 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 .

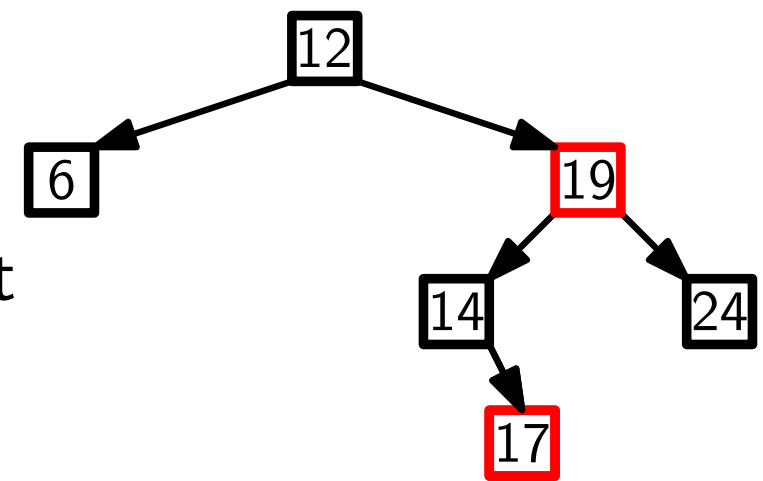
wohl-
definiert
wg. (E5)

Definition: Die *Schwarz-Höhe* $sHöhe(v)$ eines Knotens v ist die Anz. der *schwarzen* Knoten (ohne v) auf *jedem* ~~längsten~~ Pfad zu einem Blatt (inkl. Blatt) in B_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 Anz. seiner Kanten.

Definition: Sei B ein Baum.

Knoten u ist *unter* Knoten v , wenn u in dem Teilbaum B_v von B mit Wurzel v enthalten ist.

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

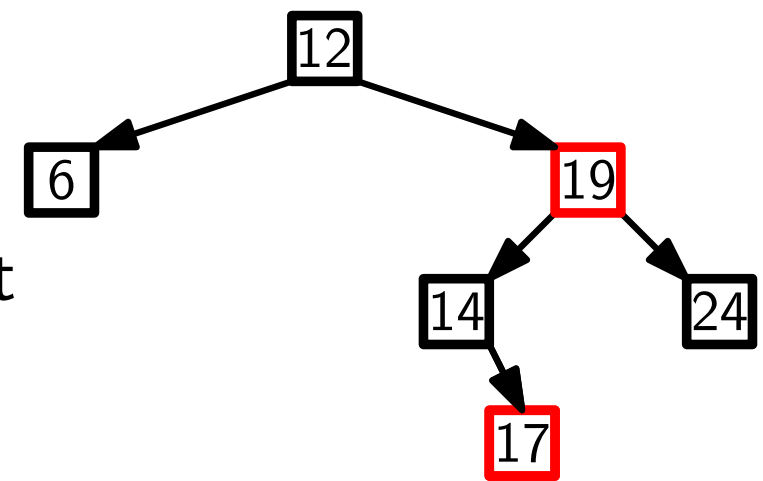
Definition: Die *Höhe* eines Knotens v ist die Länge eines längsten Pfades von v zu einem Blatt unter v . wohl-definiert wg. (E5)

Definition: Die **Schwarz-Höhe** $sHöhe(v)$ eines Knotens v ist die Anz. der **schwarzen** Knoten (ohne v) auf **jedem** ~~längsten~~ Pfad zu einem Blatt (inkl. Blatt) in B_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 Anz. seiner Kanten.

Definition: Sei B ein Baum.

Knoten u ist *unter* Knoten v , wenn u in dem Teilbaum B_v von B mit Wurzel v enthalten ist.

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

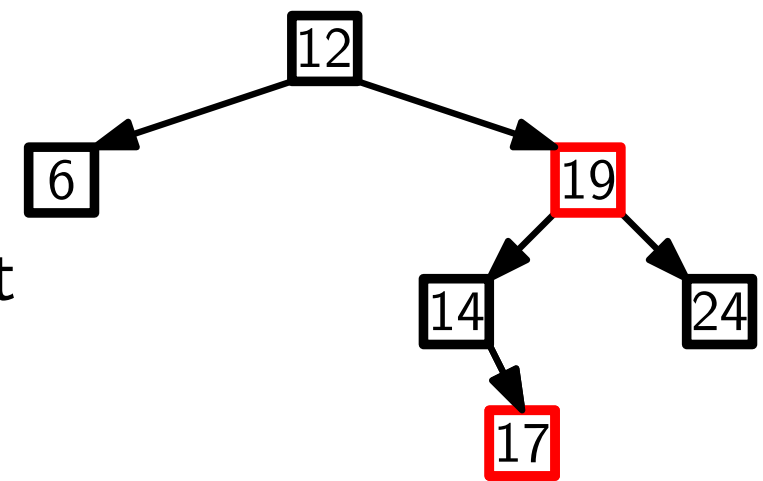
Definition: Die *Höhe* eines Knotens v ist die Länge eines längsten Pfades von v zu einem Blatt unter v . wohl-definiert wg. (E5)

Definition: Die **Schwarz-Höhe** $sHöhe(v)$ eines Knotens v ist die Anz. der **schwarzen** Knoten (ohne v) auf **jedem** ~~längsten~~ Pfad zu einem Blatt (inkl. Blatt) in B_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 Anz. seiner Kanten.

Definition: Sei B ein Baum.

Knoten u ist *unter* Knoten v , wenn u in dem Teilbaum B_v von B mit Wurzel v enthalten ist.

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

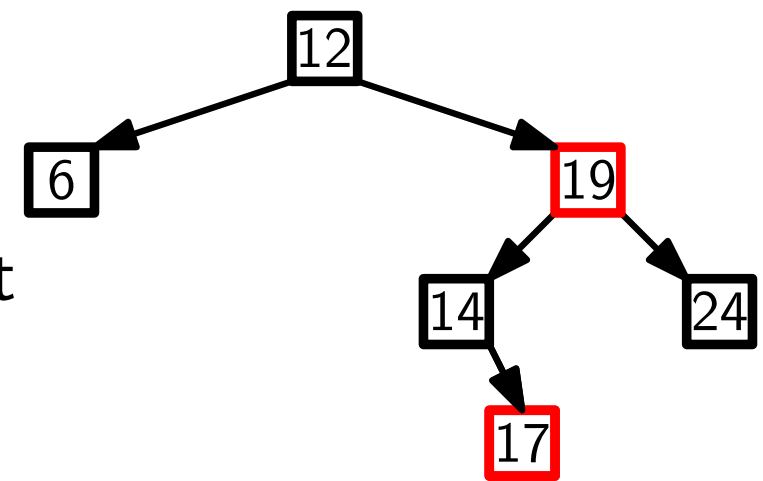
Definition: Die *Höhe* eines Knotens v ist die Länge eines längsten Pfades von v zu einem Blatt unter v . wohl-definiert wg. (E5)

Definition: Die **Schwarz-Höhe** $sHöhe(v)$ eines Knotens v ist die Anz. der **schwarzen** Knoten (ohne v) auf **jedem** ~~längsten~~ Pfad zu einem Blatt (inkl. Blatt) in B_v .

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

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

(Schwarz-) Höhe



Definition: Die *Länge* eines Pfades ist die Anz. seiner Kanten.

Definition: Sei B ein Baum.

Knoten u ist *unter* Knoten v , wenn u in dem Teilbaum B_v von B 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 . wohl-definiert wg. (E5)

Definition: Die **Schwarz-Höhe** $sHöhe(v)$ eines Knotens v ist die Anz. der **schwarzen** Knoten (ohne v) auf **jedem** ~~längsten~~ Pfad zu einem Blatt (inkl. Blatt) in B_v .

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

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

Höhe $\in \Theta(\log n)!!$

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

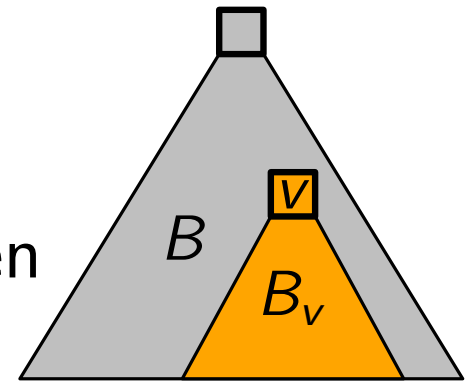
Höhe $\in \Theta(\log n)!!$

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

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

Höhe $\in \Theta(\log n)!!$

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



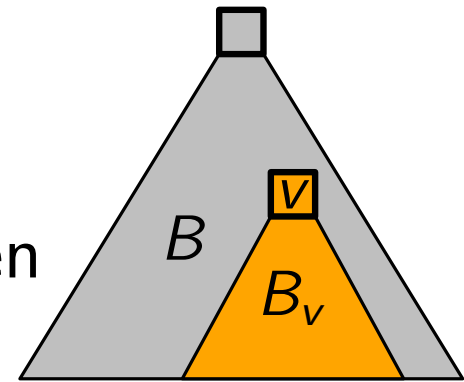
Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{Höhe}(v)} - 1$ innere Knoten.

Höhe $\in \Theta(\log n)!!$

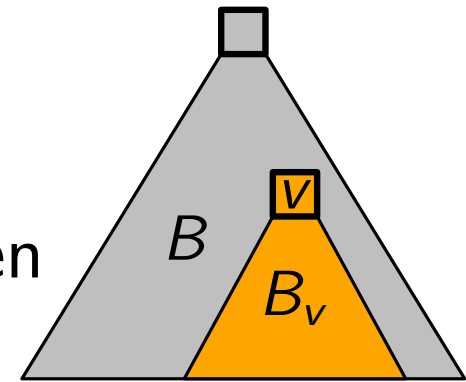
Lemma. Ein Rot-Schwarz-Baum B mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.



Beweis. Behauptung: Für jeden Knoten v von B gilt:
 B_v hat $\geq 2^{\text{Höhe}(v)} - 1$ innere Knoten.
Beweis durch vollständige Induktion über $\text{Höhe}(v)$.

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

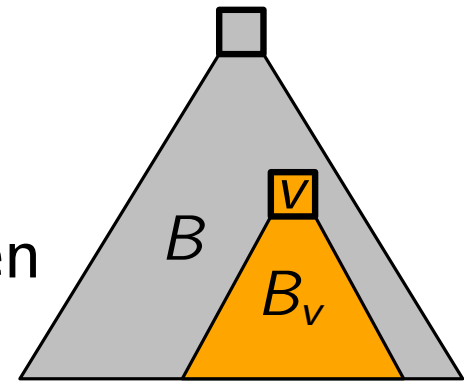
B_v hat $\geq 2^{\text{Höhe}(v)} - 1$ innere Knoten.

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

$\text{Höhe}(v) = 0$.

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

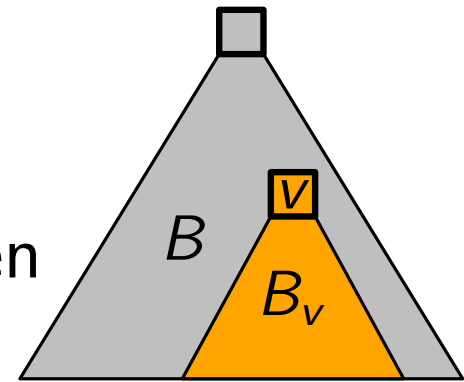
B_v hat $\geq 2^{\text{Höhe}(v)} - 1$ innere Knoten.

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

$\text{Höhe}(v) = 0$. Dann $B_v = B.\text{nil}$

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

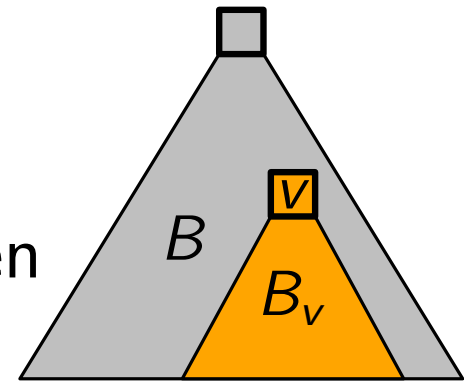
B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

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

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

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

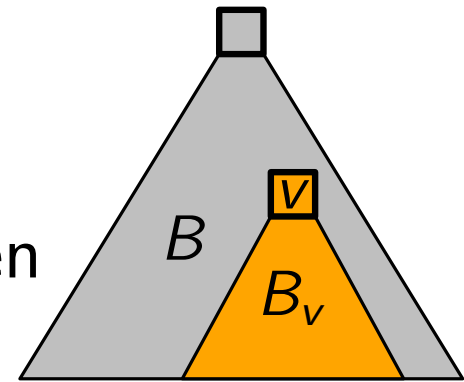
Beweis durch vollständige Induktion über $\text{Höhe}(v)$.

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

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

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

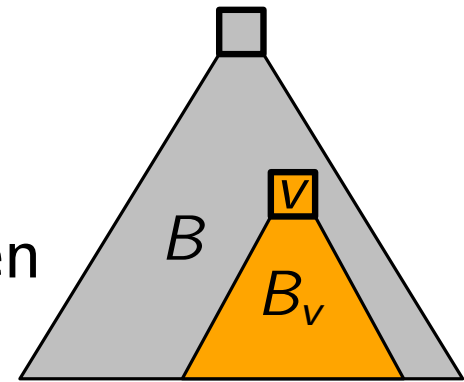
Beweis durch vollständige Induktion über $\text{Höhe}(v)$.

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

B_v hat $2^0 - 1 = 0$ innere Knoten. ✓

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

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

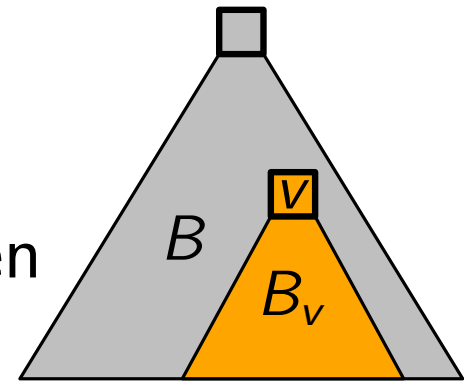
$\text{Höhe}(v) = 0$. Dann $B_v = B.\text{nil}$ und $\text{sHöhe}(v) = 0$.

B_v hat $2^0 - 1 = 0$ innere Knoten. ✓

$\text{Höhe}(v) > 0$.

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

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

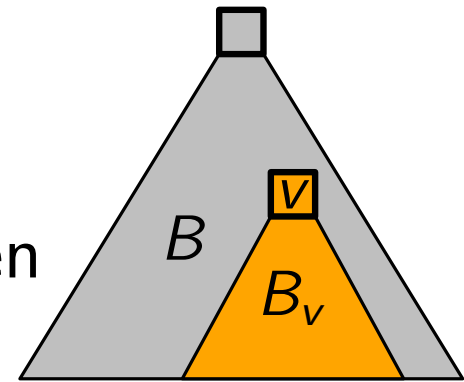
$\text{Höhe}(v) = 0$. Dann $B_v = B.\text{nil}$ und $\text{sHöhe}(v) = 0$.

B_v hat $2^0 - 1 = 0$ 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 B mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

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

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

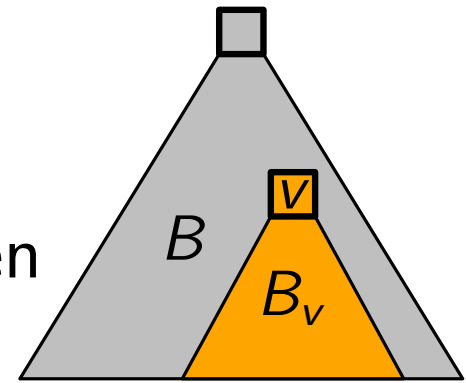
B_v hat $2^0 - 1 = 0$ innere Knoten. ✓

$\text{Höhe}(v) > 0$. 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 B mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

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

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

B_v hat $2^0 - 1 = 0$ innere Knoten. ✓

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

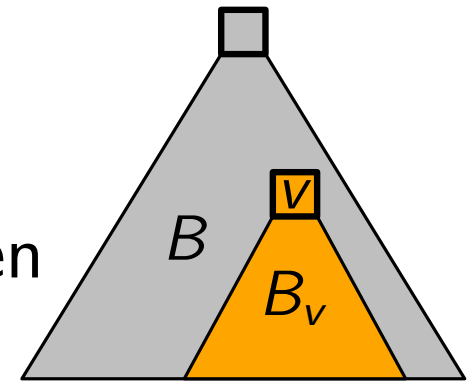
\Rightarrow können Ind.-Annahme anwenden.

\Rightarrow Anz. innere Knoten von B_v ist mind.

$2 \cdot (\quad) + 1 =$

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

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

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

B_v hat $2^0 - 1 = 0$ innere Knoten. ✓

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

\Rightarrow können Ind.-Annahme anwenden.

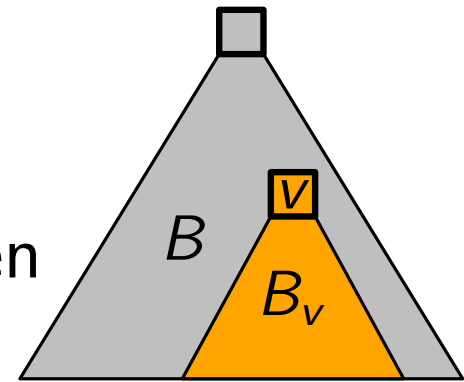
\Rightarrow Anz. innere Knoten von B_v ist mind.

$2 \cdot (\underbrace{\hspace{10em}}) + 1 =$

Anz. innerer Knoten unter
einem Kind von v

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

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

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

B_v hat $2^0 - 1 = 0$ innere Knoten. ✓

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

\Rightarrow können Ind.-Annahme anwenden.

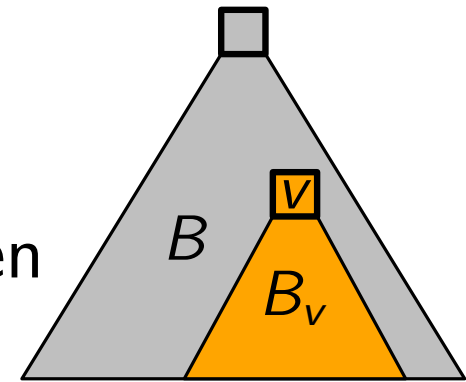
\Rightarrow Anz. innere Knoten von B_v ist mind.

$$2 \cdot \underbrace{(2^{\text{sHöhe}(v)-1} - 1)}_{\text{Anz. innerer Knoten unter einem Kind von } v} + 1 =$$

Anz. innerer Knoten unter
einem Kind von v

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

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

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

B_v hat $2^0 - 1 = 0$ innere Knoten. ✓

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

\Rightarrow können Ind.-Annahme anwenden.

\Rightarrow Anz. innere Knoten von B_v ist mind.

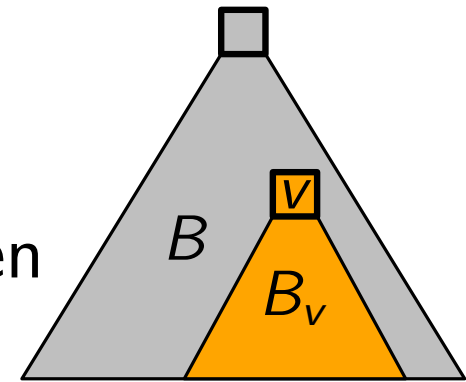
$$2 \cdot (2^{\text{sHöhe}(v)-1} - 1) + 1 =$$

sHöhe der Kinder von v ist mind.

Anz. innerer Knoten unter einem Kind von v

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

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

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

B_v hat $2^0 - 1 = 0$ innere Knoten. ✓

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

\Rightarrow können Ind.-Annahme anwenden.

\Rightarrow Anz. innere Knoten von B_v ist mind.

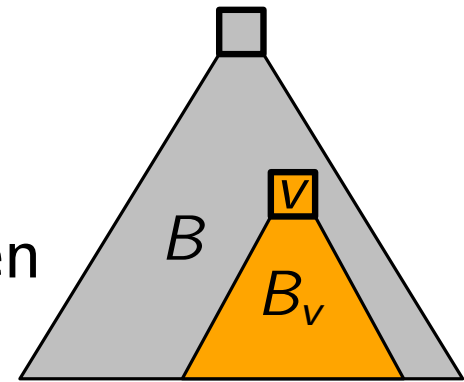
$2 \cdot (2^{\text{sHöhe}(v)-1} - 1) + 1 = 2^{\text{sHöhe}(v)} - 1$.

sHöhe der Kinder von v ist mind.

Anz. innerer Knoten unter einem Kind von v

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

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

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

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

\Rightarrow können Ind.-Annahme anwenden.

\Rightarrow Anz. innere Knoten von B_v ist mind.

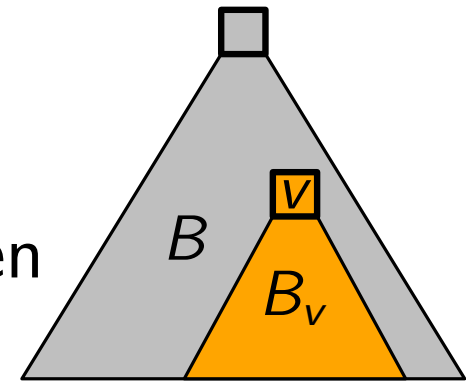
$2 \cdot (2^{\text{sHöhe}(v)-1} - 1) + 1 = 2^{\text{sHöhe}(v)} - 1$. ✓

sHöhe der Kinder von v ist mind.

Anz. innerer Knoten unter
einem Kind von v

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

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

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

B_v hat $2^0 - 1 = 0$ innere Knoten. ✓

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

\Rightarrow können Ind.-Annahme anwenden.

\Rightarrow Anz. innere Knoten von B_v ist mind.

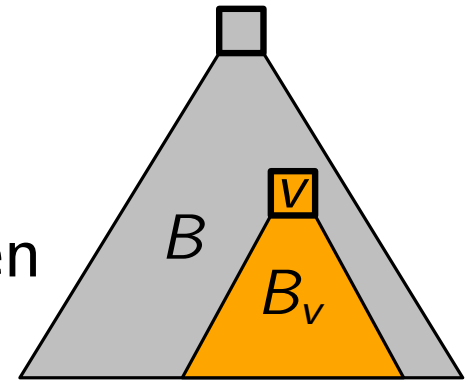
$2 \cdot (2^{\text{sHöhe}(v)-1} - 1) + 1 = 2^{\text{sHöhe}(v)} - 1$. ✓

sHöhe der Kinder von v ist mind.

Anz. innerer Knoten unter einem Kind von v

Höhe $\in \Theta(\log n)!!$

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



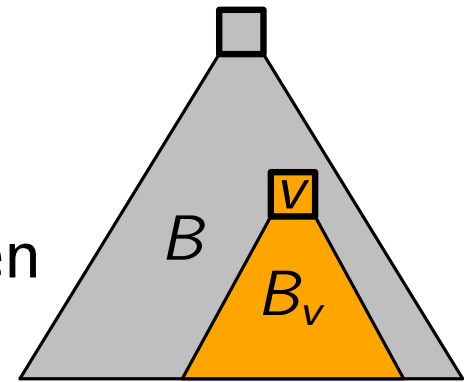
Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

Höhe $\in \Theta(\log n)!!$

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



Beweis.

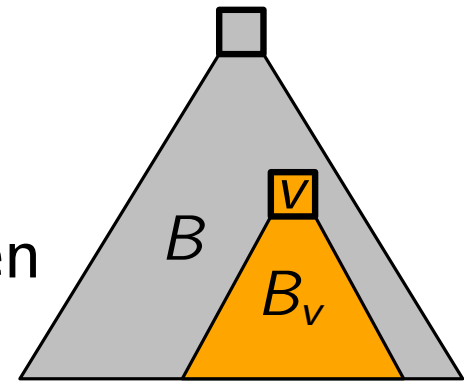
Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

$v := B.\text{root} \Rightarrow$

Höhe $\in \Theta(\log n)!!$

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



Beweis.

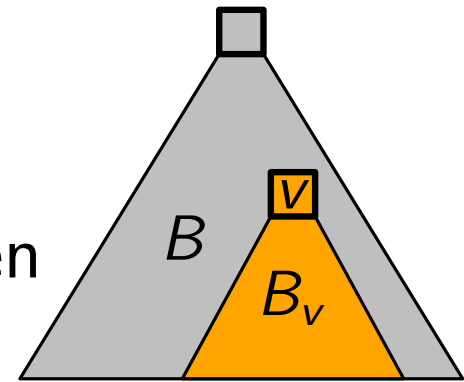
Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

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

Höhe $\in \Theta(\log n)!!$

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



Beweis.

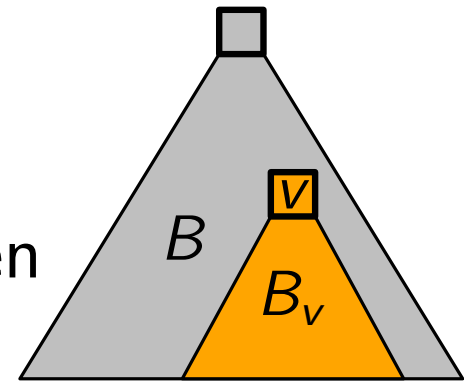
Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

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

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

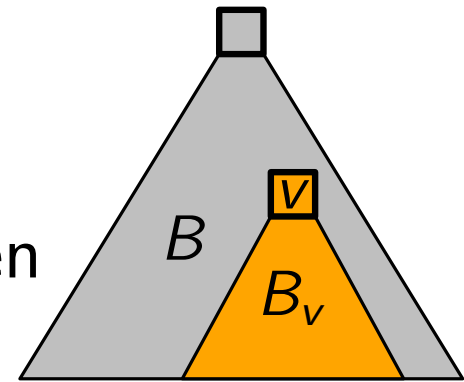
B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

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

$\Rightarrow \text{sHöhe}(B) \leq$

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

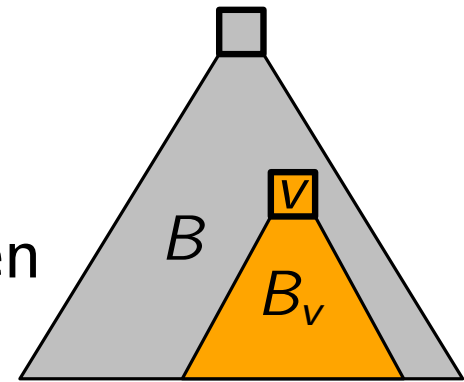
B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

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

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

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

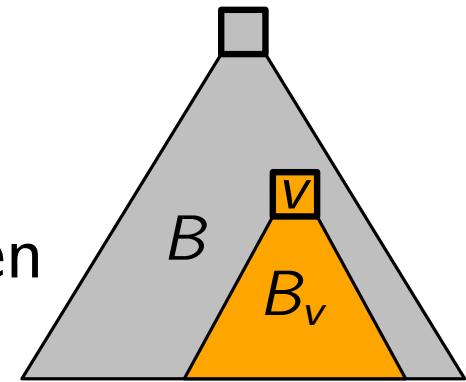
$v := B.\text{root} \Rightarrow \underbrace{\# \text{ innere Knoten}(B)}_n \geq 2^{\text{sHöhe}(B)} - 1.$

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

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

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

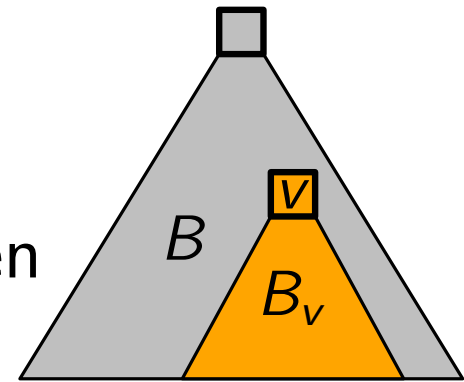
$$v := B.\text{root} \Rightarrow \underbrace{\# \text{ innere Knoten}(B)}_n \geq 2^{\text{sHöhe}(B)} - 1.$$

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

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

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

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

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

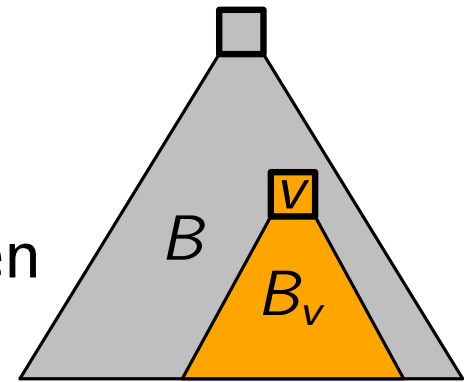
Wegen R-S-Eig. (E4) gilt: $\text{Höhe}(B) \leq 2 \cdot \text{sHöhe}(B)$.

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

□

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

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

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

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

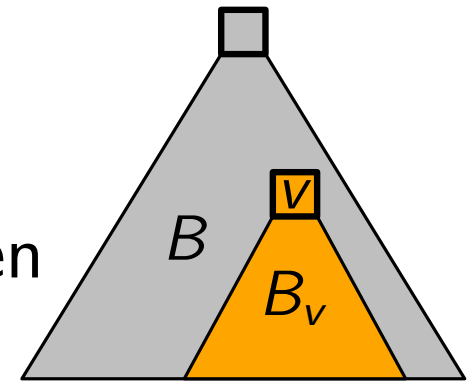
$$\Rightarrow \text{Höhe}(B) \leq 2 \log_2(n + 1)$$

□

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

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

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

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

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

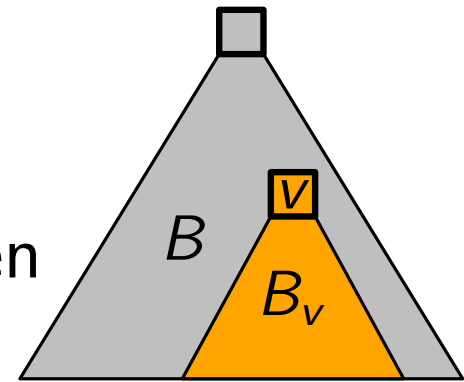
$$\Rightarrow \text{Höhe}(B) \leq 2 \log_2(n + 1)$$

□

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

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

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

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

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

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

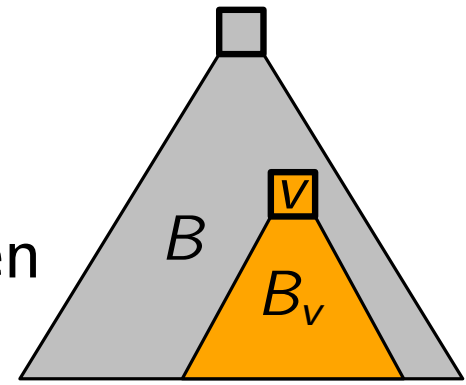
□

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

Nee:

Höhe $\in \Theta(\log n)!!$

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



Beweis.

Behauptung: Für jeden Knoten v von B gilt:

B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

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

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

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

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

□

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

Nee: **Insert & Delete können R-S-Eig. verletzen!**

Einfügen

Node Insert(key k)

$y = nil$

$x = root$

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

$y = x$

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

$x = x.left$

else $x = x.right$

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

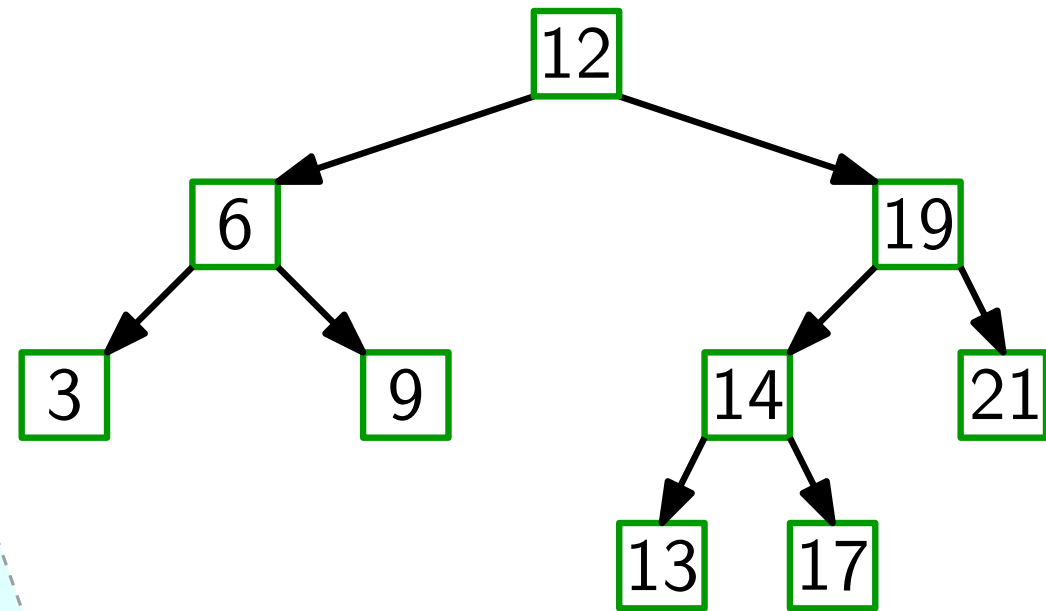
if $y == nil$ **then** $root = z$

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 $x \neq nil$ **do**

$y = x$

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

$x = x.left$

else $x = x.right$

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

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

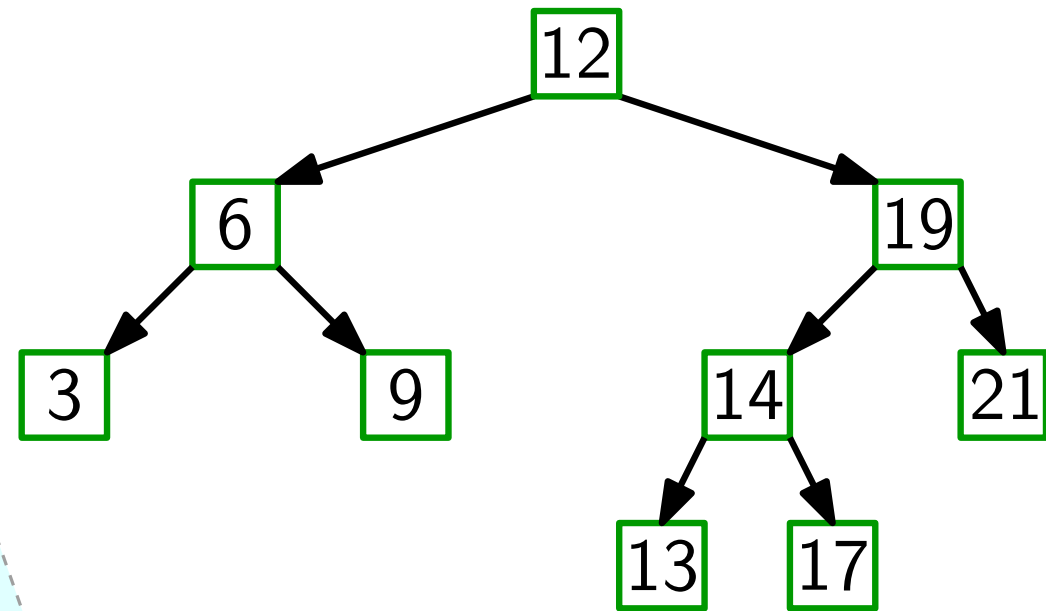
else

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 $x \neq nil$ **do**

$y = x$

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

$x = x.left$

else $x = x.right$

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

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

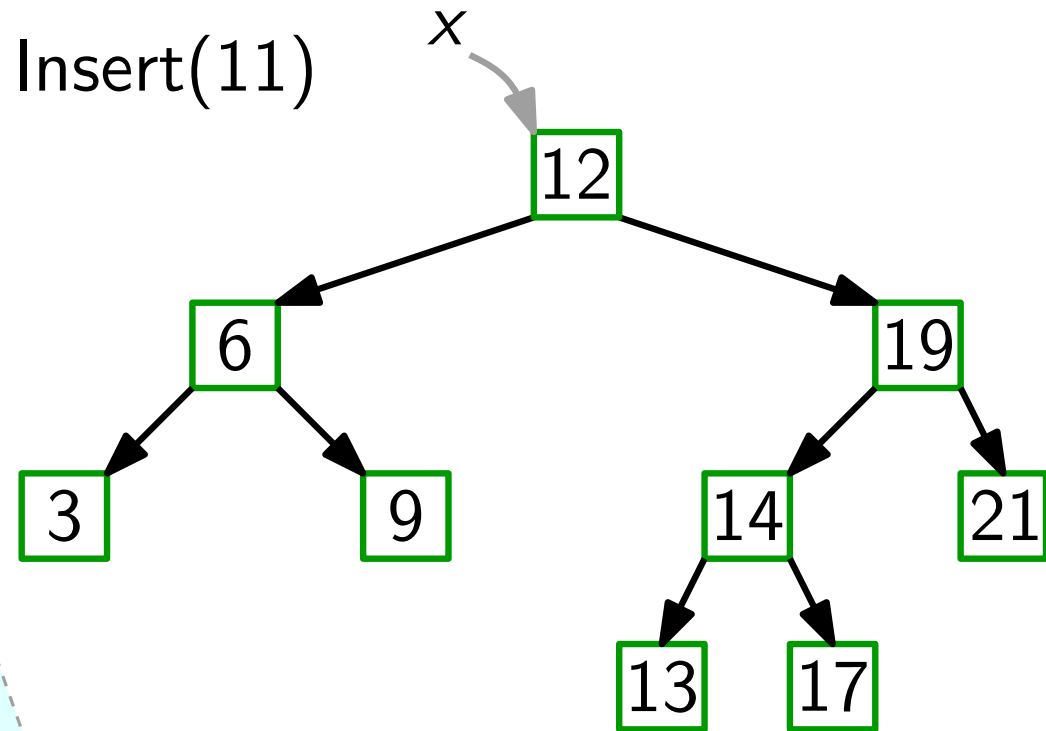
else

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 $x \neq nil$ **do**

$y = x$

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

$x = x.left$

else $x = x.right$

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

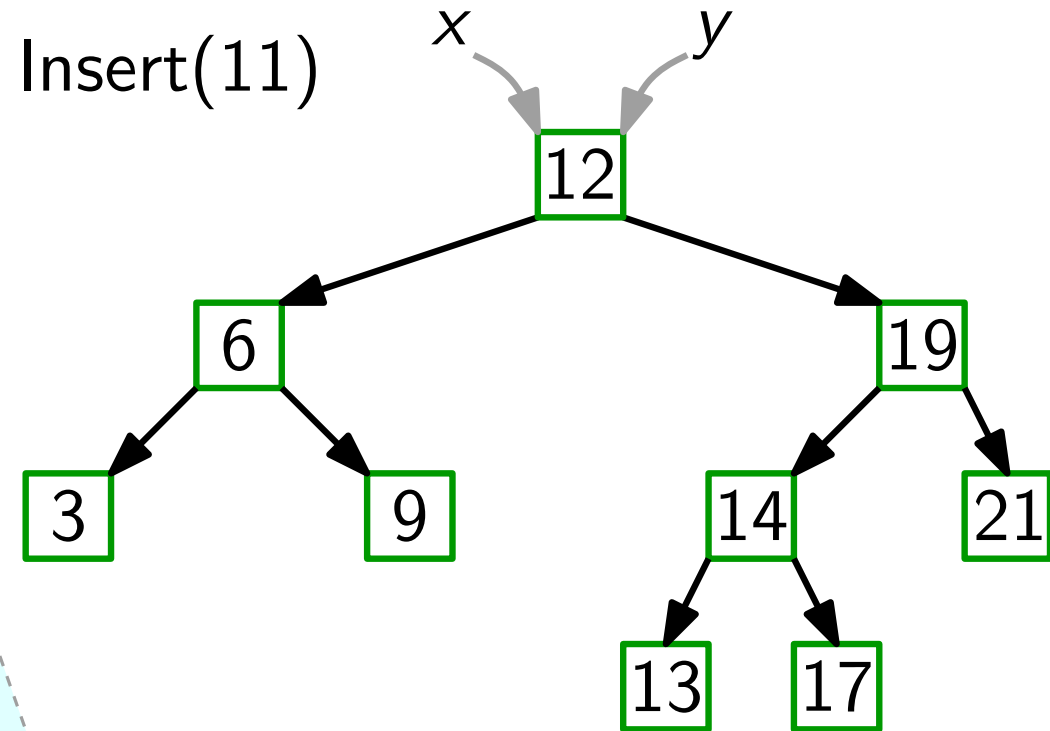
if $y == nil$ **then** $root = z$

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 $x \neq nil$ **do**

$y = x$

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

$x = x.left$

else $x = x.right$

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

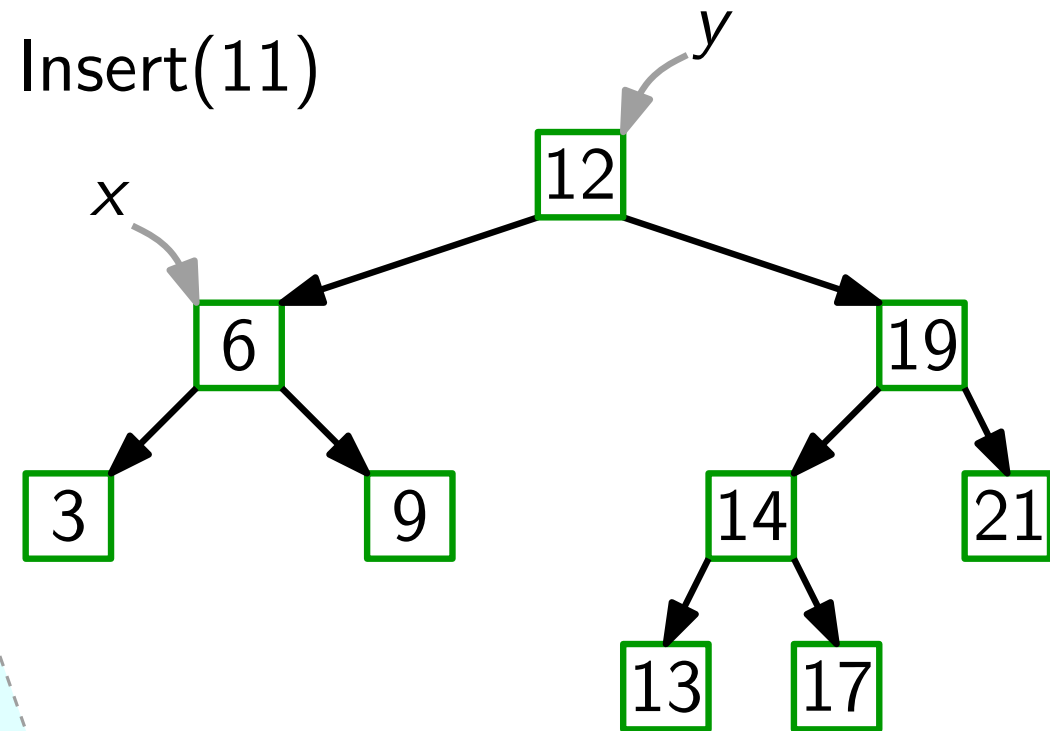
if $y == nil$ **then** $root = z$

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 $x \neq nil$ **do**

$y = x$

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

$x = x.left$

else $x = x.right$

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

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

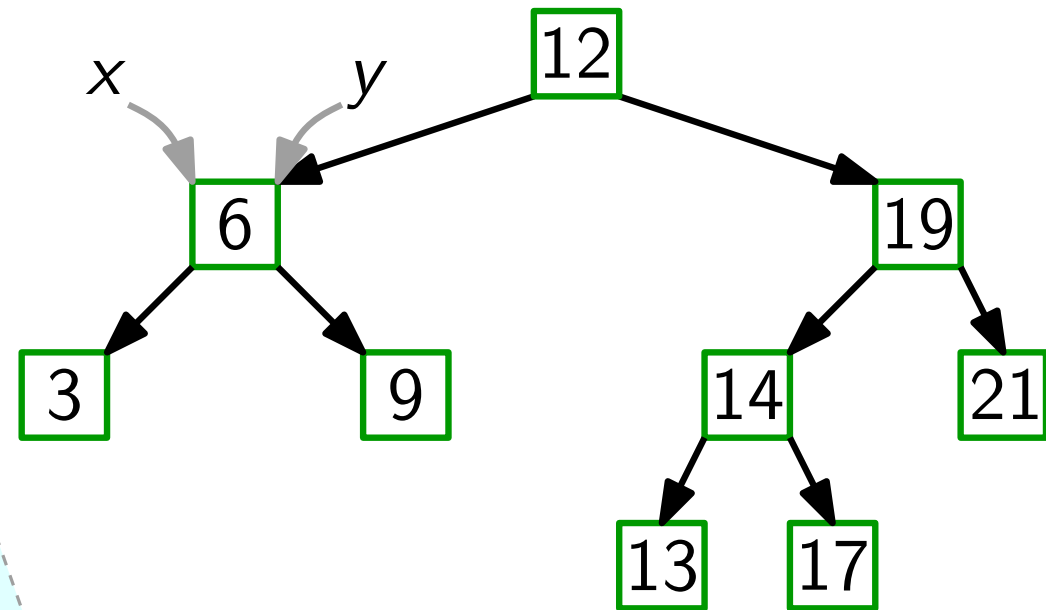
else

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 $x \neq nil$ **do**

$y = x$

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

$x = x.left$

else $x = x.right$

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

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

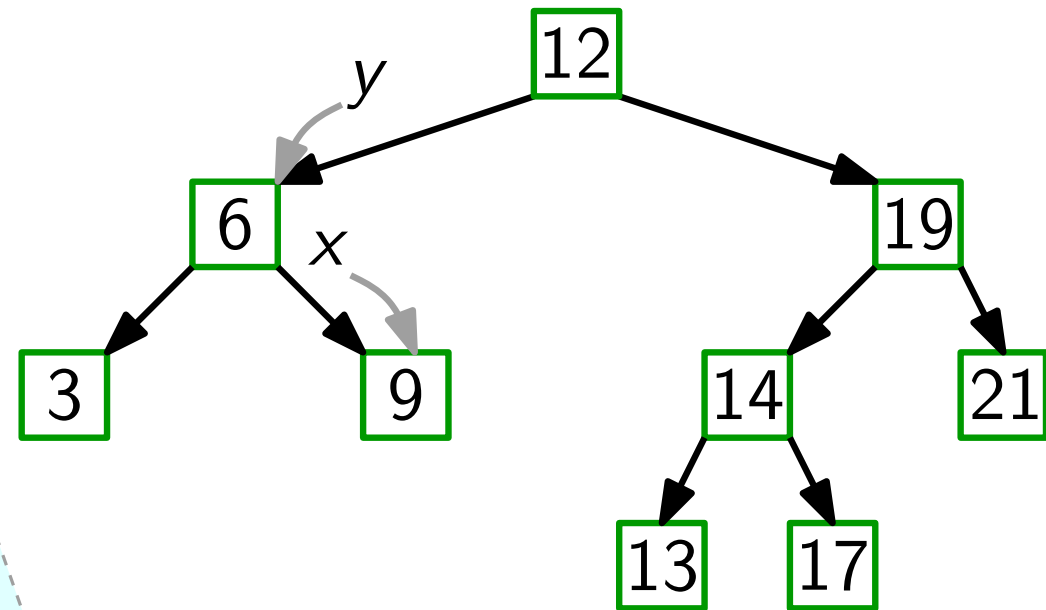
else

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 $x \neq nil$ **do**

$y = x$

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

$x = x.left$

else $x = x.right$

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

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

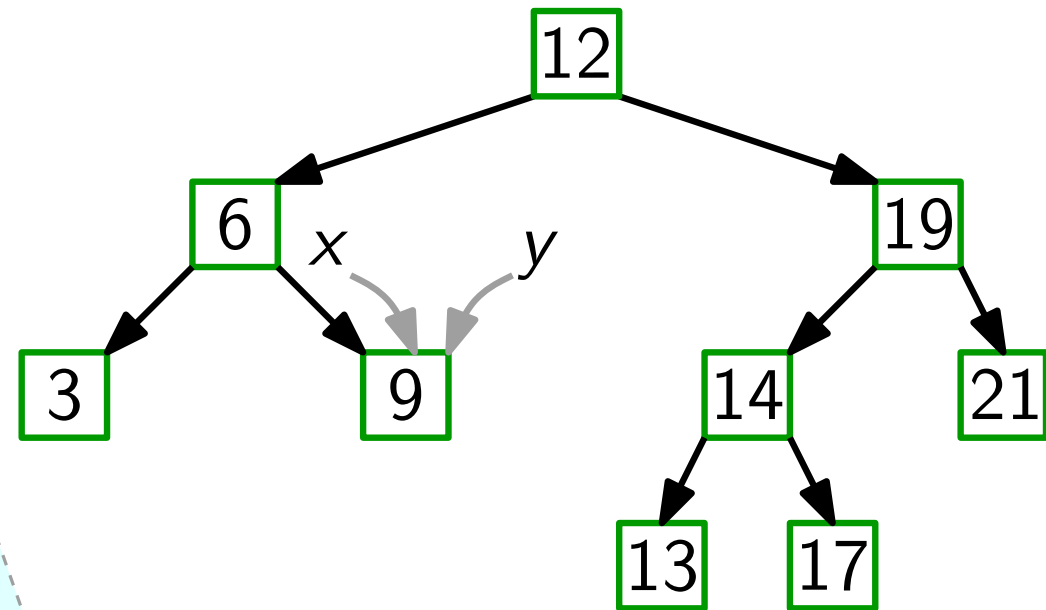
else

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 $x \neq nil$ **do**

$y = x$

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

$x = x.left$

else $x = x.right$

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

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

else

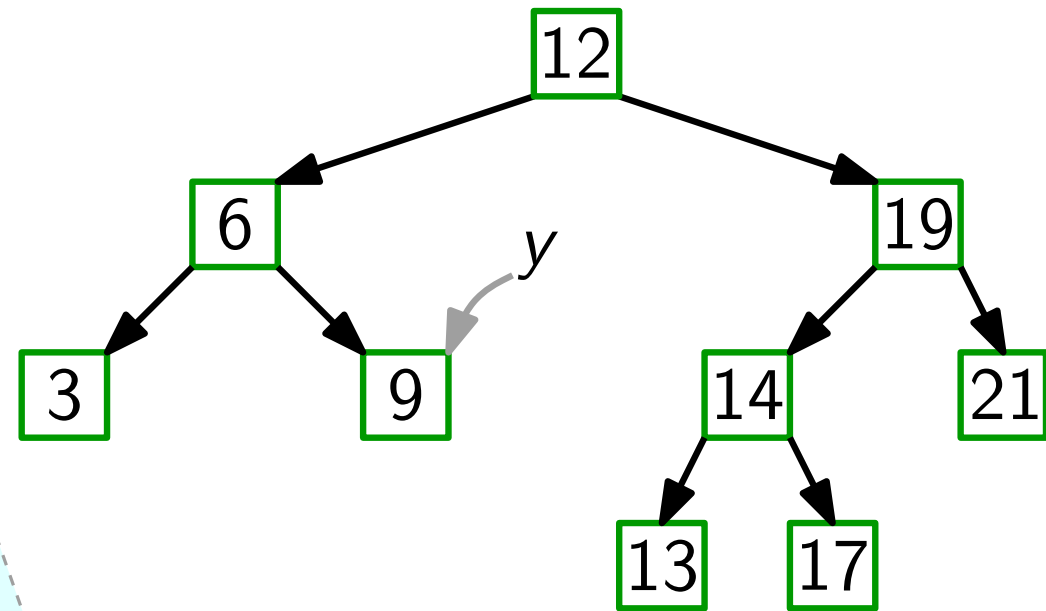
if $k < y.key$ **then** $y.left = z$

else $y.right = z$

return z

Insert(11)

$x == nil$



Einfügen

Node Insert(key k)

$y = nil$

$x = root$

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

$y = x$

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

$x = x.left$

else $x = x.right$

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

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

else

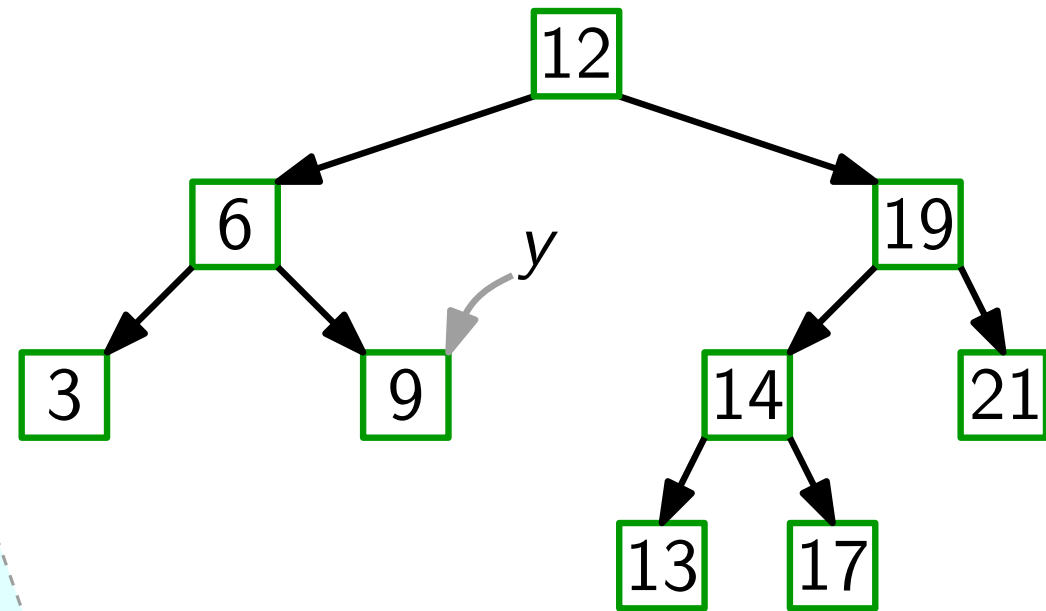
if $k < y.key$ **then** $y.left = z$

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 $x \neq nil$ **do**

$y = x$

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

$x = x.left$

else $x = x.right$

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

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

else

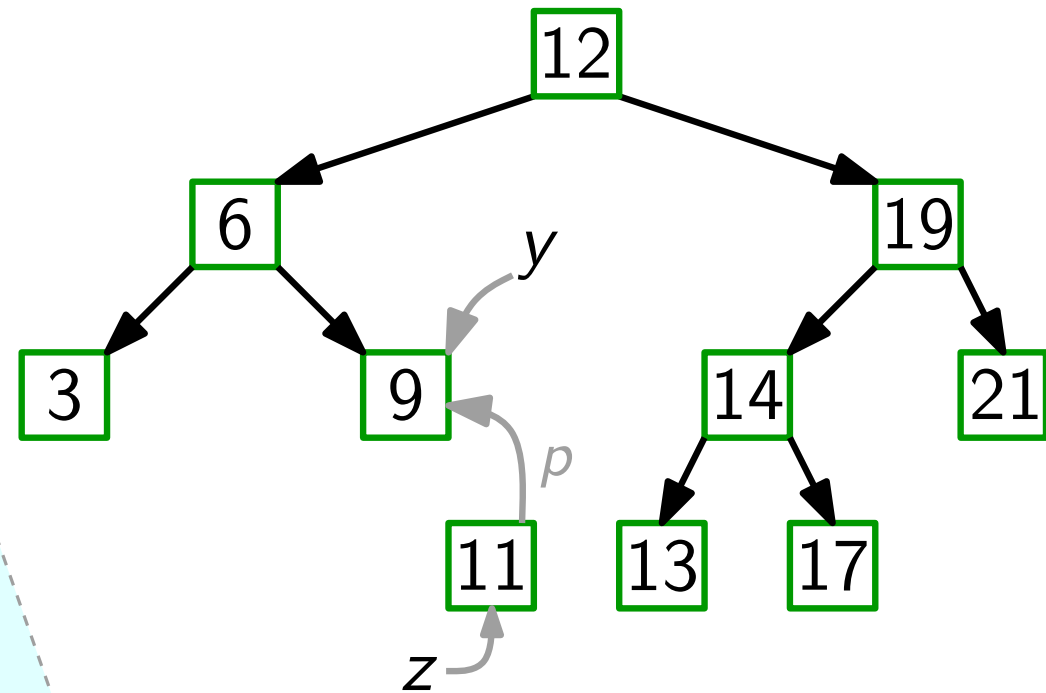
if $k < y.key$ **then** $y.left = z$

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 $x \neq nil$ **do**

$y = x$

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

$x = x.left$

else $x = x.right$

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

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

else

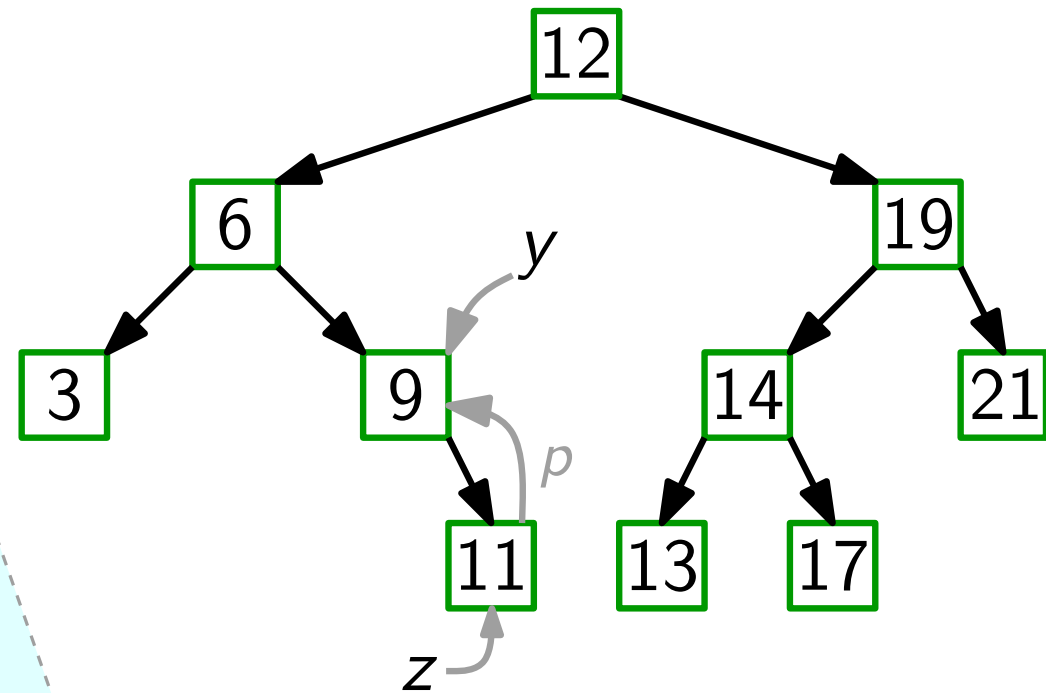
if $k < y.key$ **then** $y.left = z$

else $y.right = z$

return z

Insert(11)

$x == nil$



Node(Key k , Node par)

$key = k$

$p = par$

$right = left = nil$

Einfügen

RB

Node Insert(key k)

$y = nil$

$x = root$

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

$y = x$

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

$x = x.left$

else $x = x.right$

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

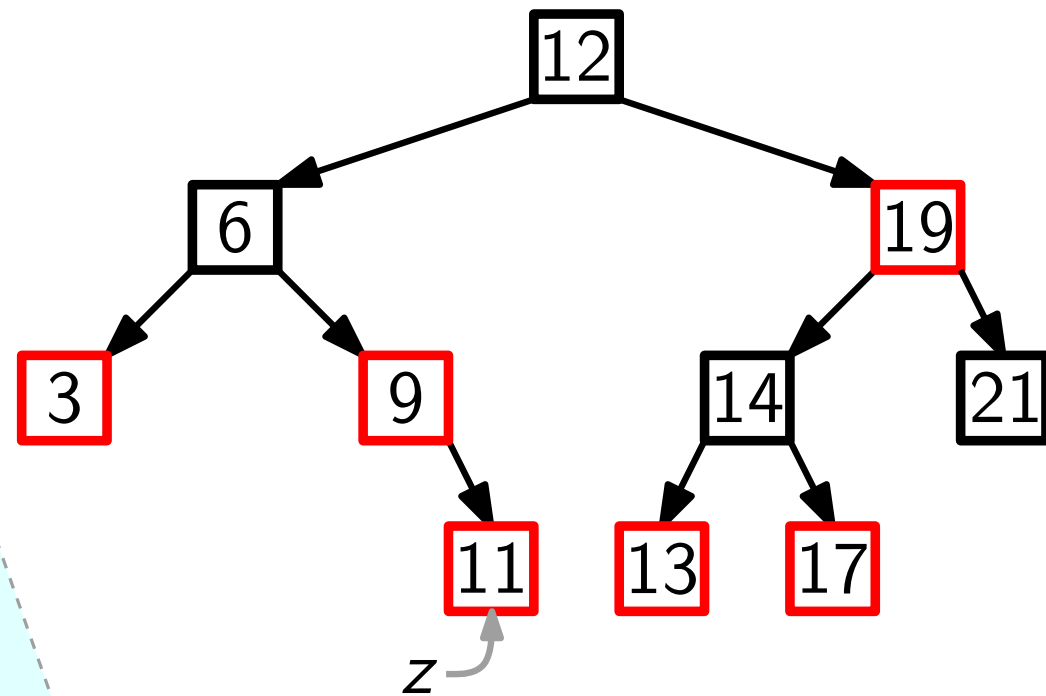
if $y == nil$ **then** $root = z$

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

RB

Node 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 $x = x.right$

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

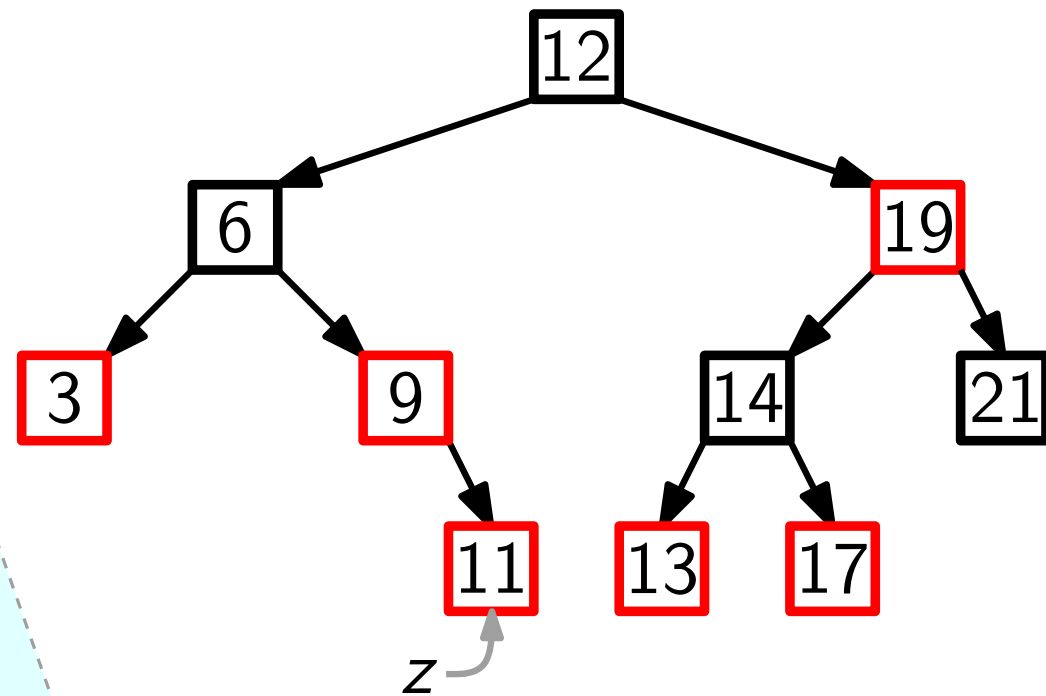
if $y == T.nil$ **then** $root = z$

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

RB RB

Node 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 $x = x.right$

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

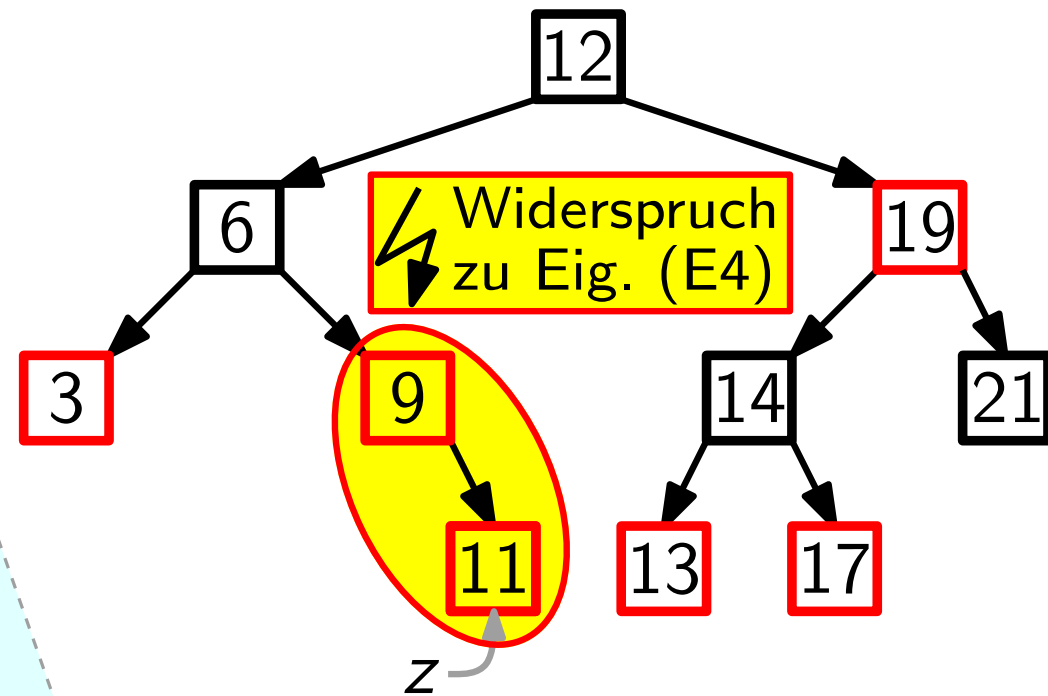
if $y == T.nil$ **then** $root = z$

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

RB RB

Node 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 $x = x.right$

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

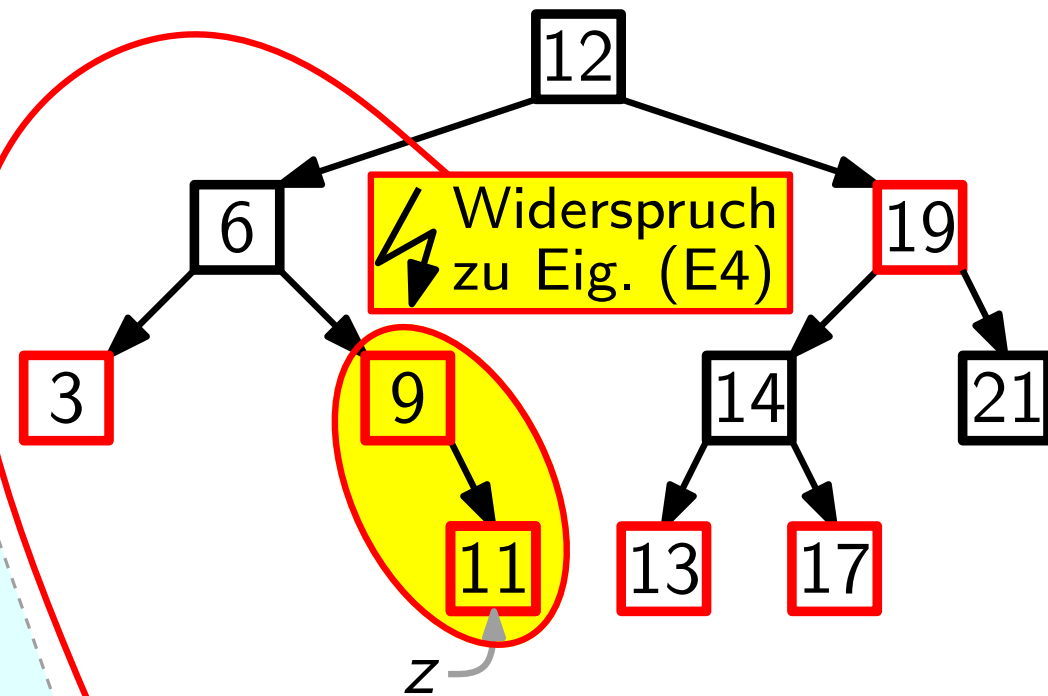
if $y == T.nil$ **then** $root = z$

else

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

else $y.right = z$

RBInsertFixup(z)
return z



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)

RB RB

Node 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 $x = x.right$

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

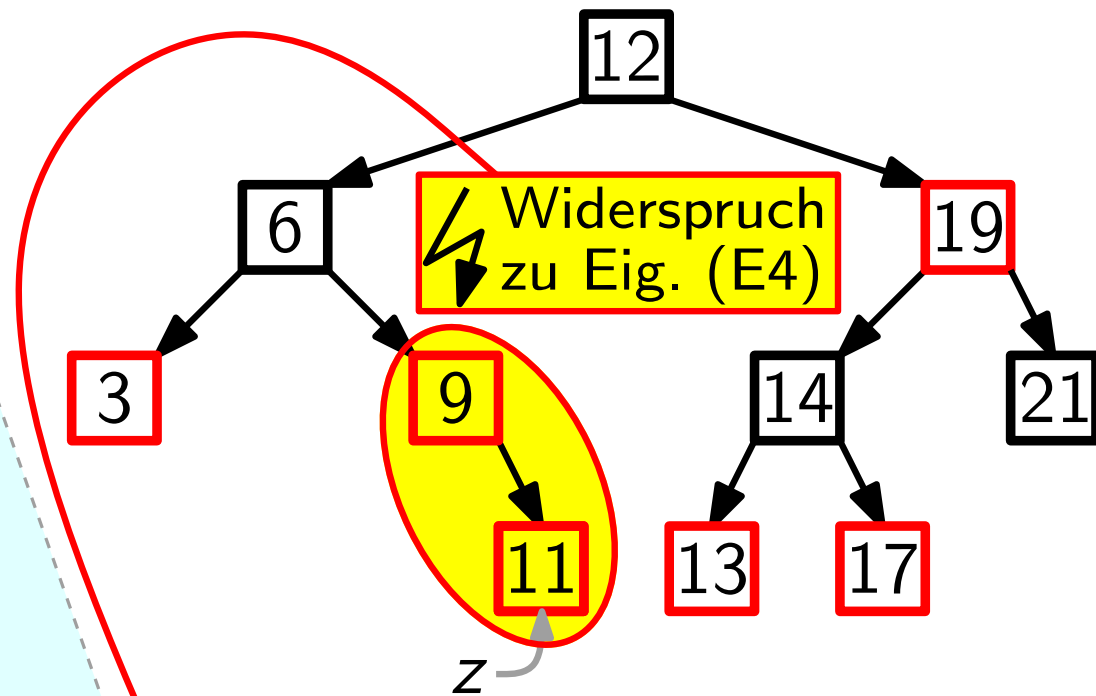
if $y == T.nil$ **then** $root = z$

else

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

else $y.right = z$

RBInsertFixup(z)
return z



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) $O(h)$

RB RB

Node 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 $x = x.right$

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

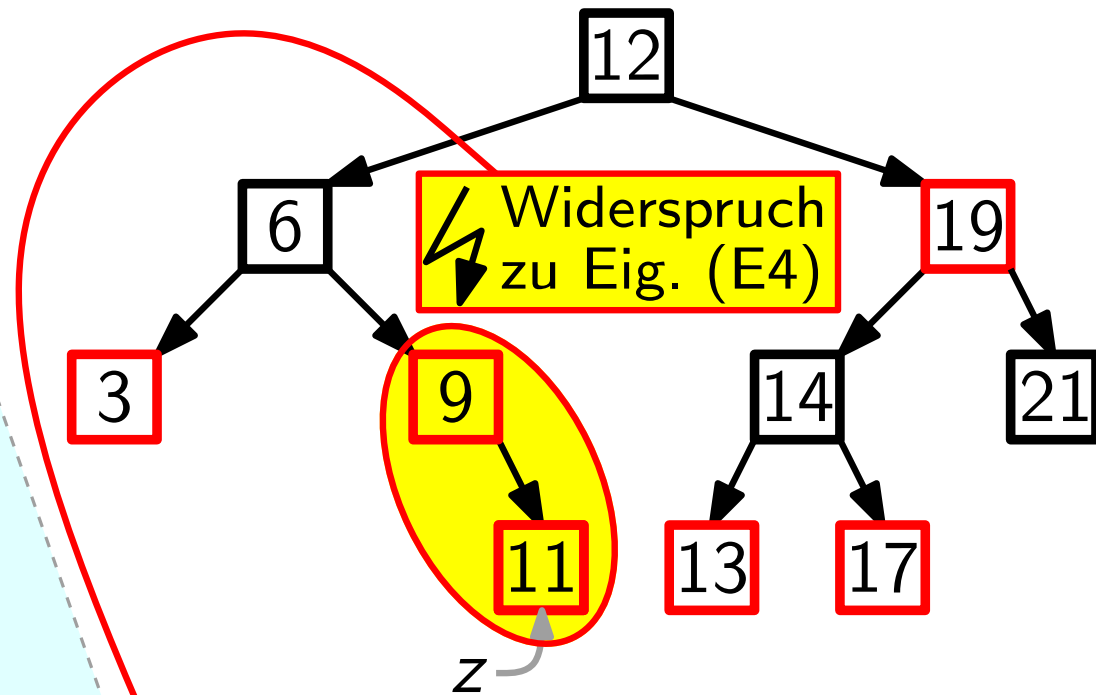
if $y == T.nil$ **then** $root = z$

else

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

else $y.right = z$

RBInsertFixup(z)
return z



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) $O(h) = O(\log n)$

RB RB

Node 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 $x = x.right$

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

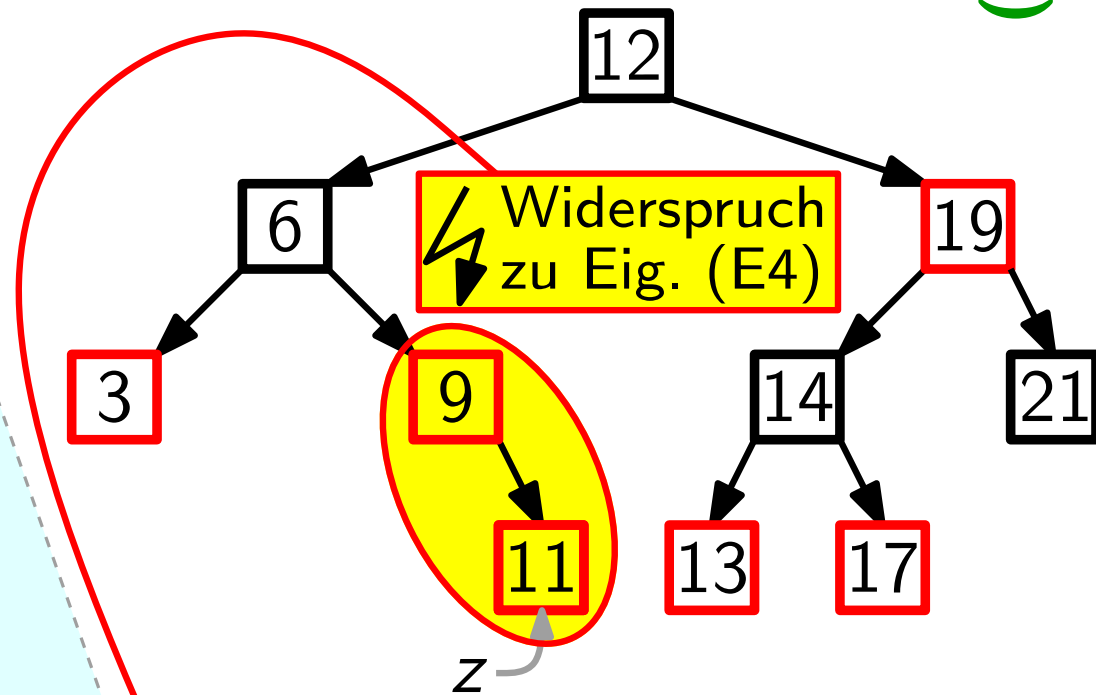
if $y == T.nil$ **then** $root = z$

else

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

else $y.right = z$

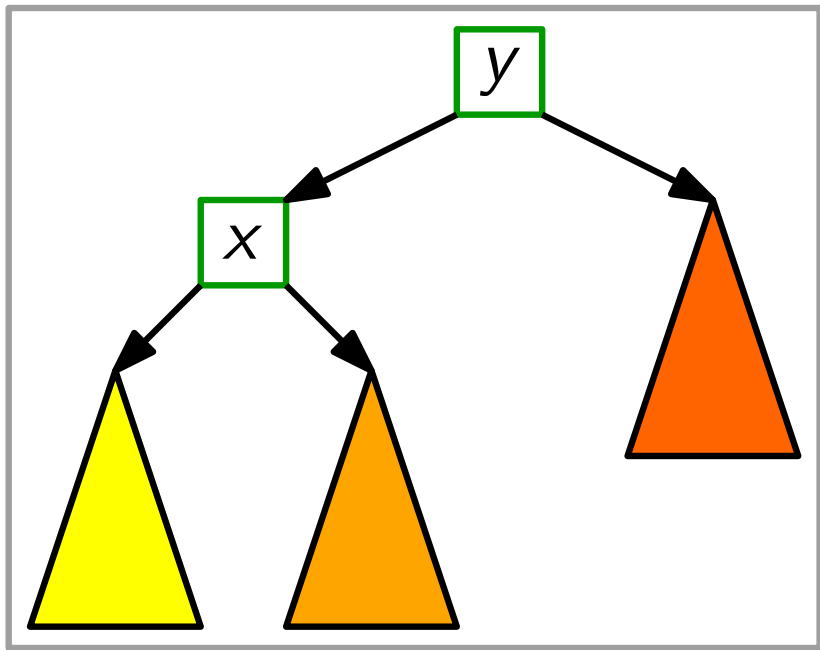
RBInsertFixup(z)
return z



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

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

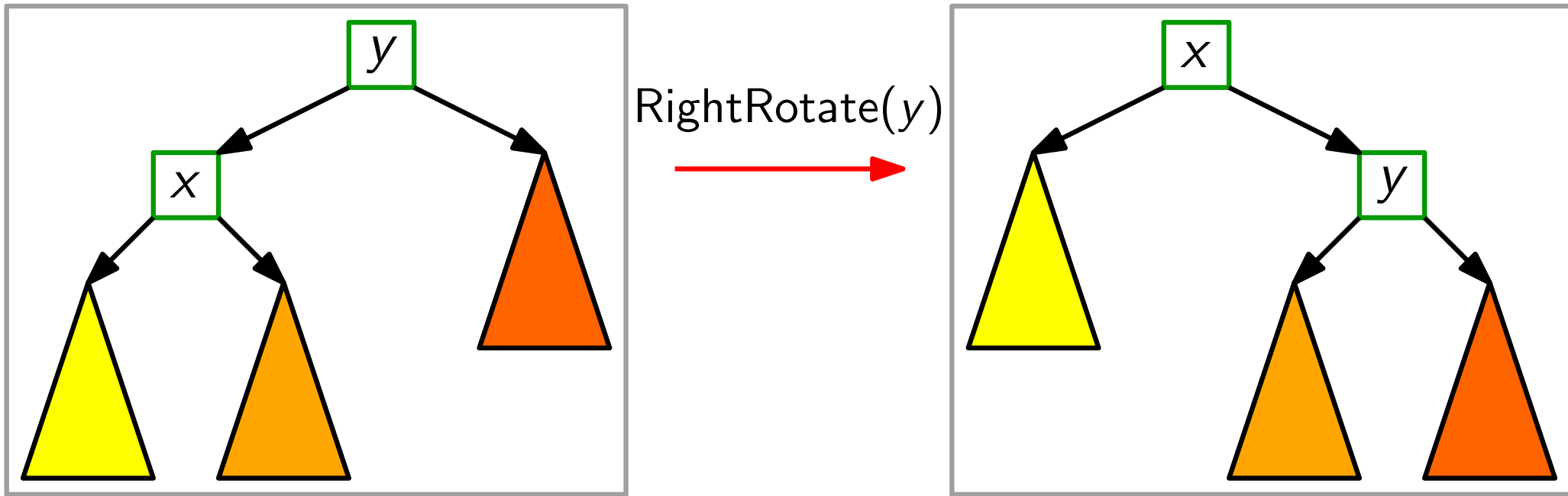
Exkurs: Rotationen



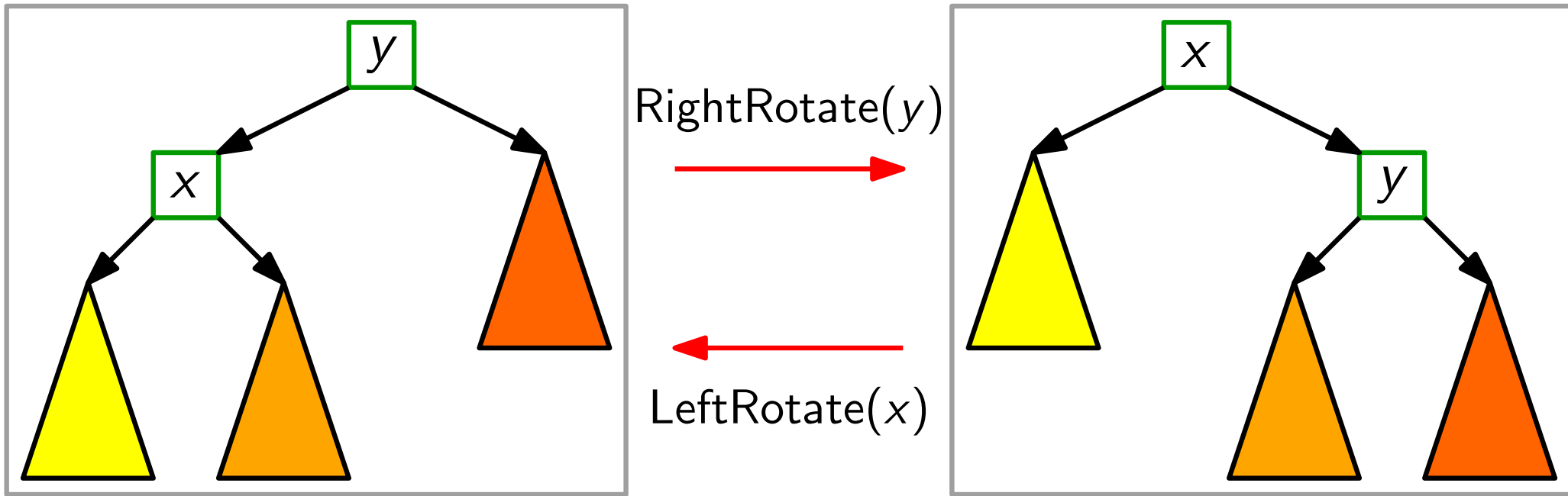
RightRotate(y)



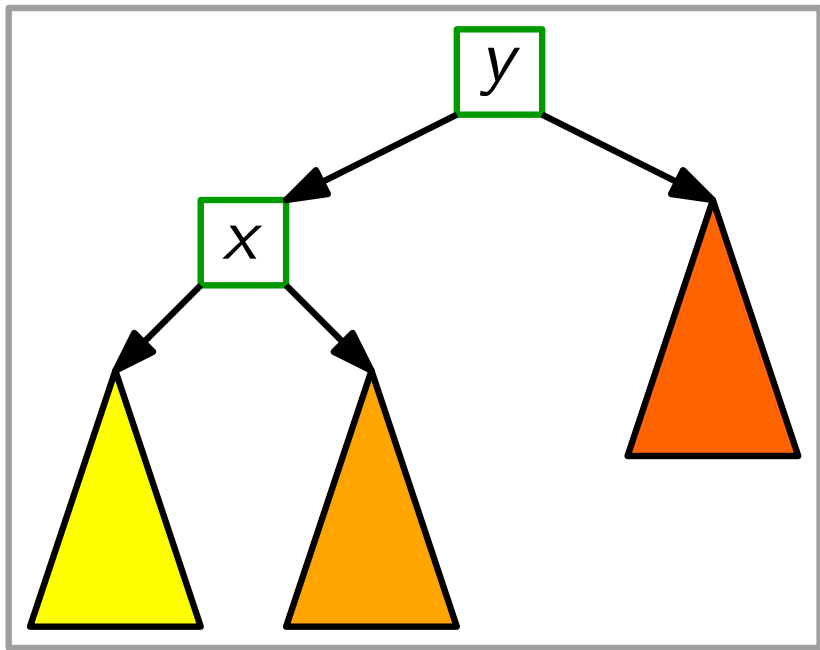
Exkurs: Rotationen



Exkurs: Rotationen



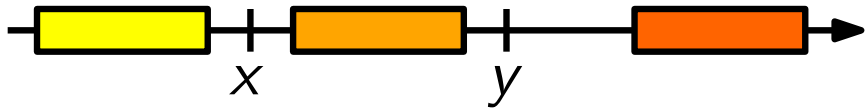
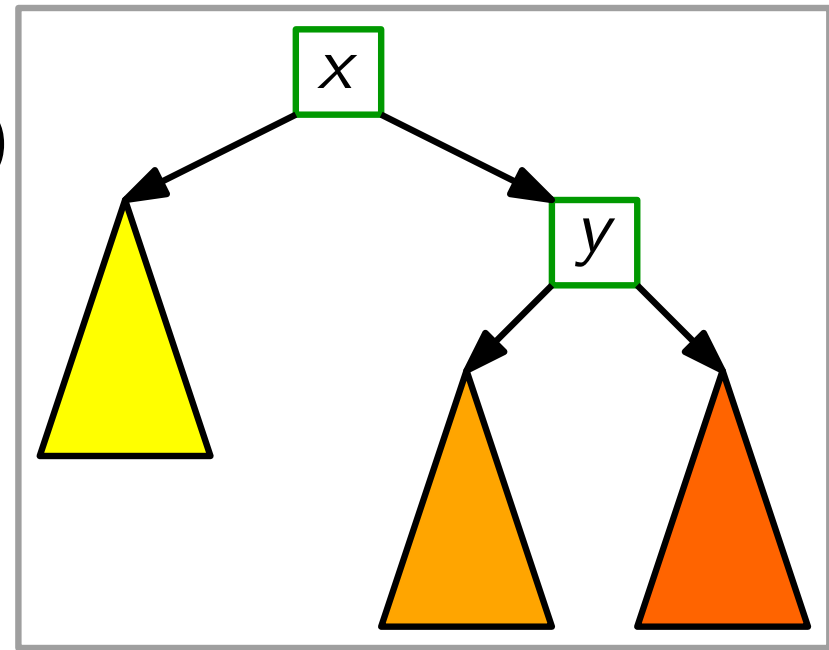
Exkurs: Rotationen



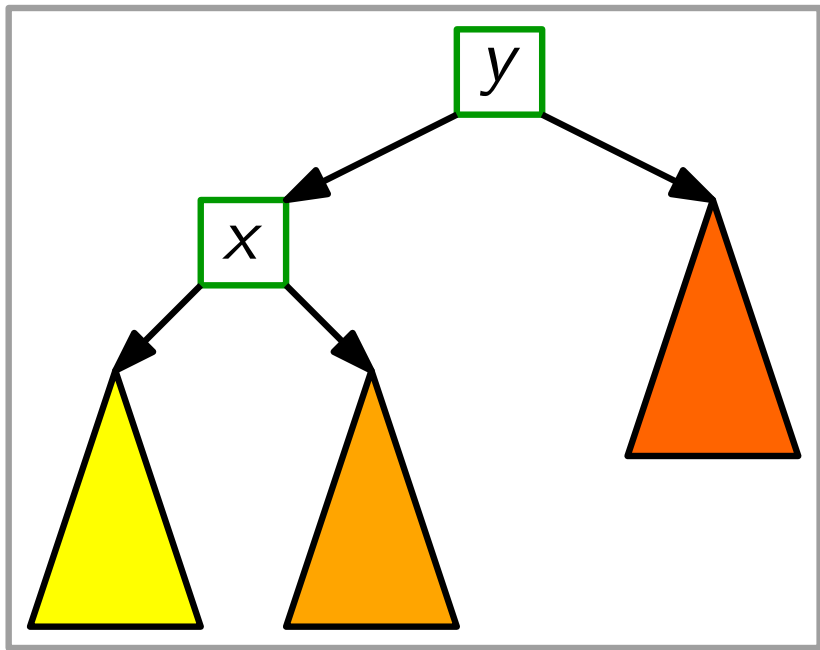
RightRotate(y)



LeftRotate(x)



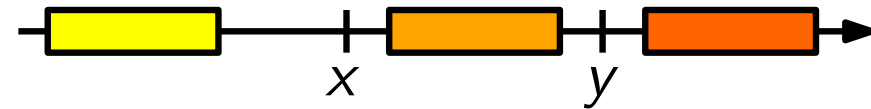
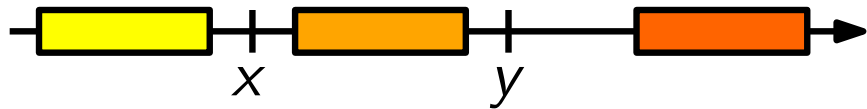
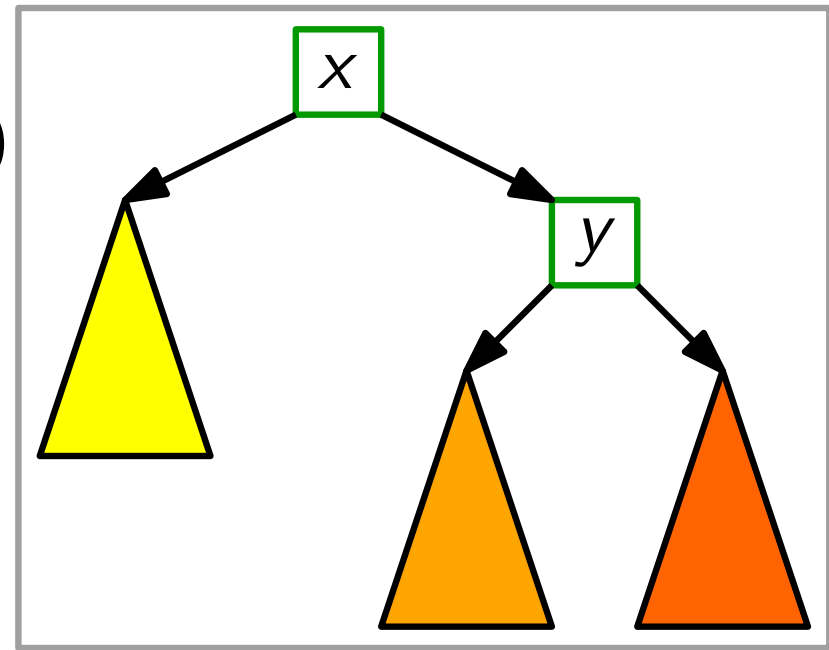
Exkurs: Rotationen



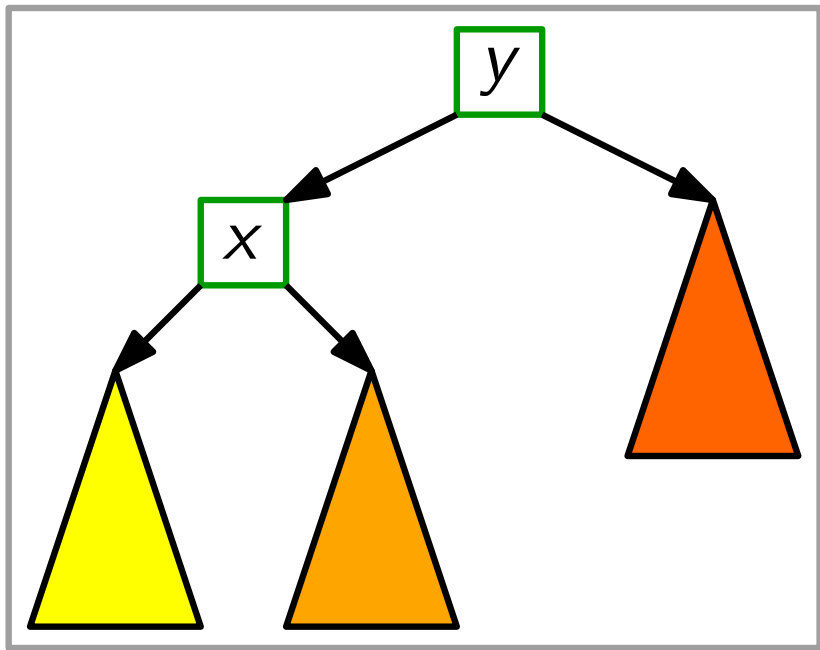
RightRotate(y)



LeftRotate(x)



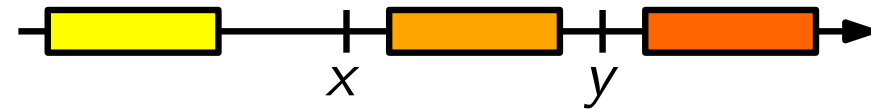
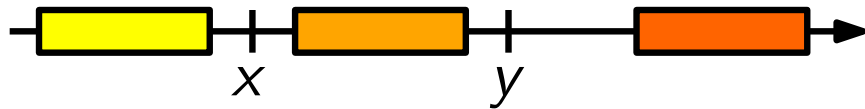
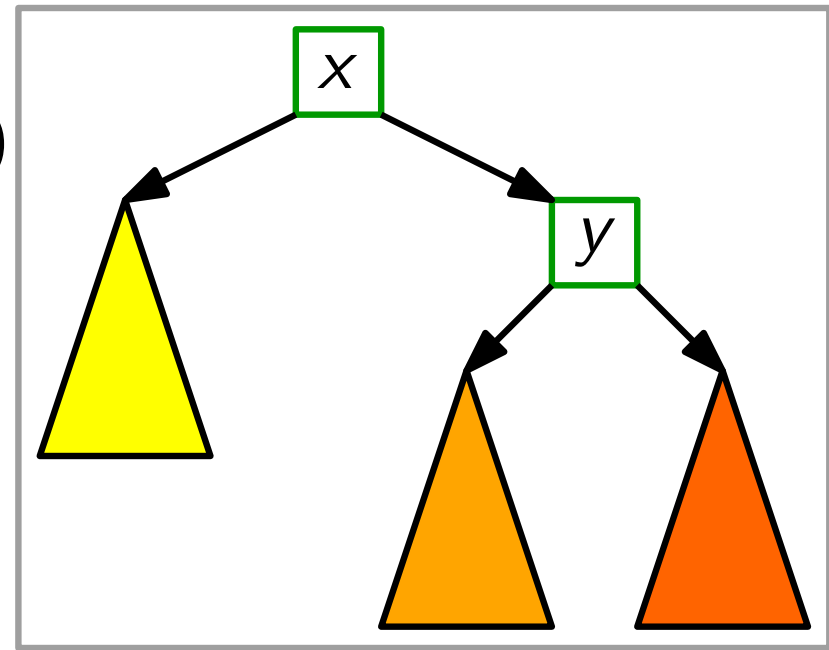
Exkurs: Rotationen



RightRotate(y)



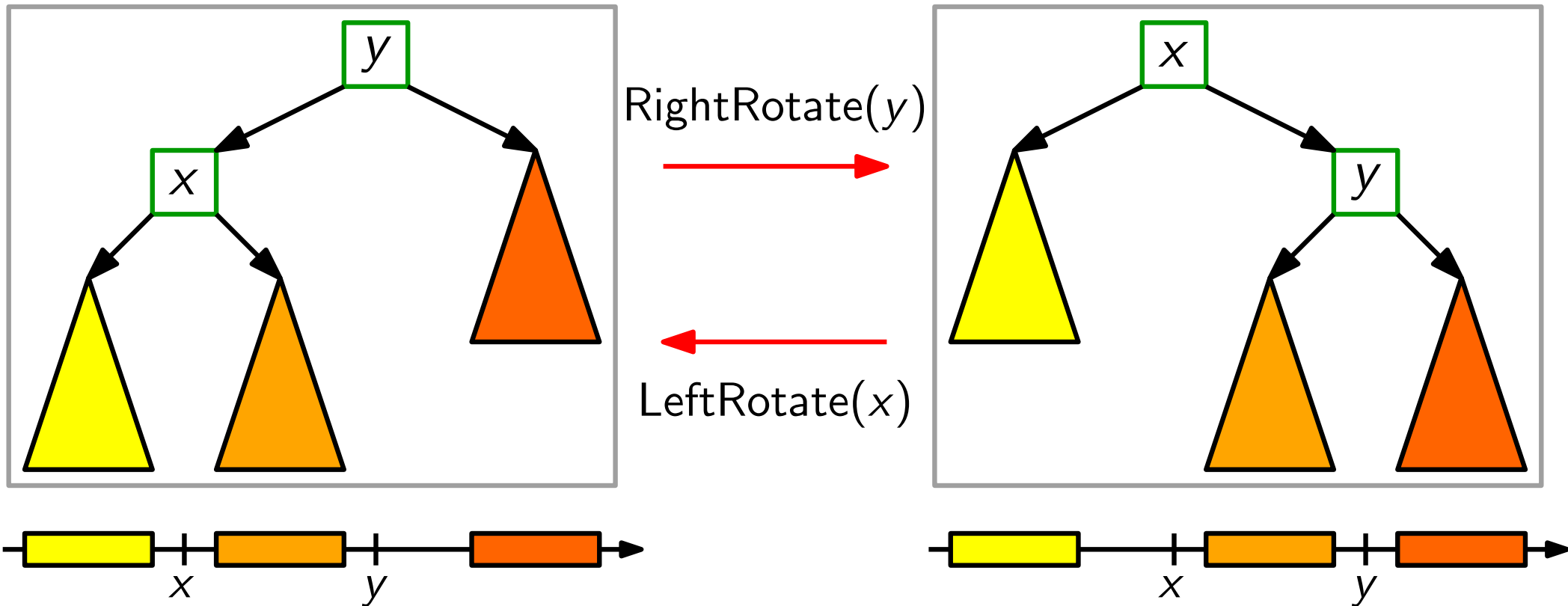
LeftRotate(x)



Also:

Binärer-Suchbaum-Eig. bleibt beim Rotieren erhalten!

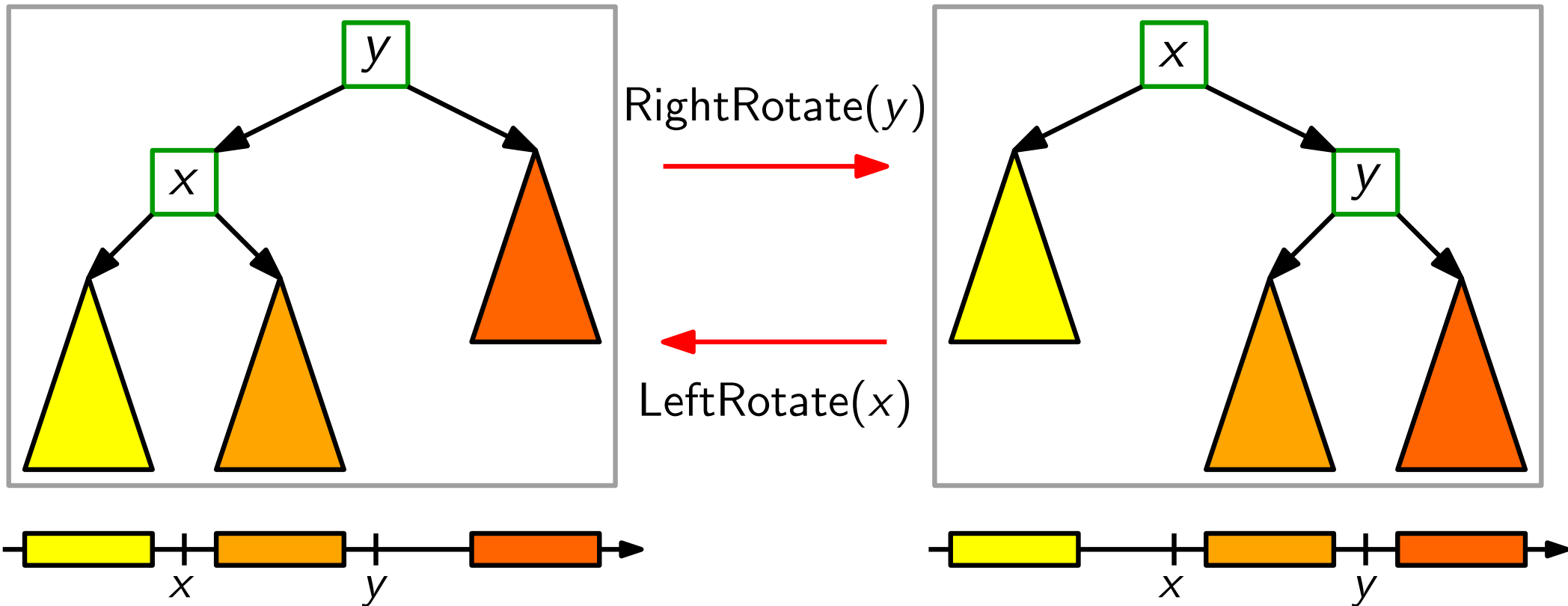
Exkurs: Rotationen



Also: Binärer-Suchbaum-Eig. bleibt beim Rotieren erhalten!

Aufgabe: Schreiben Sie Pseudocode für LeftRotate(x)!

Exkurs: Rotationen

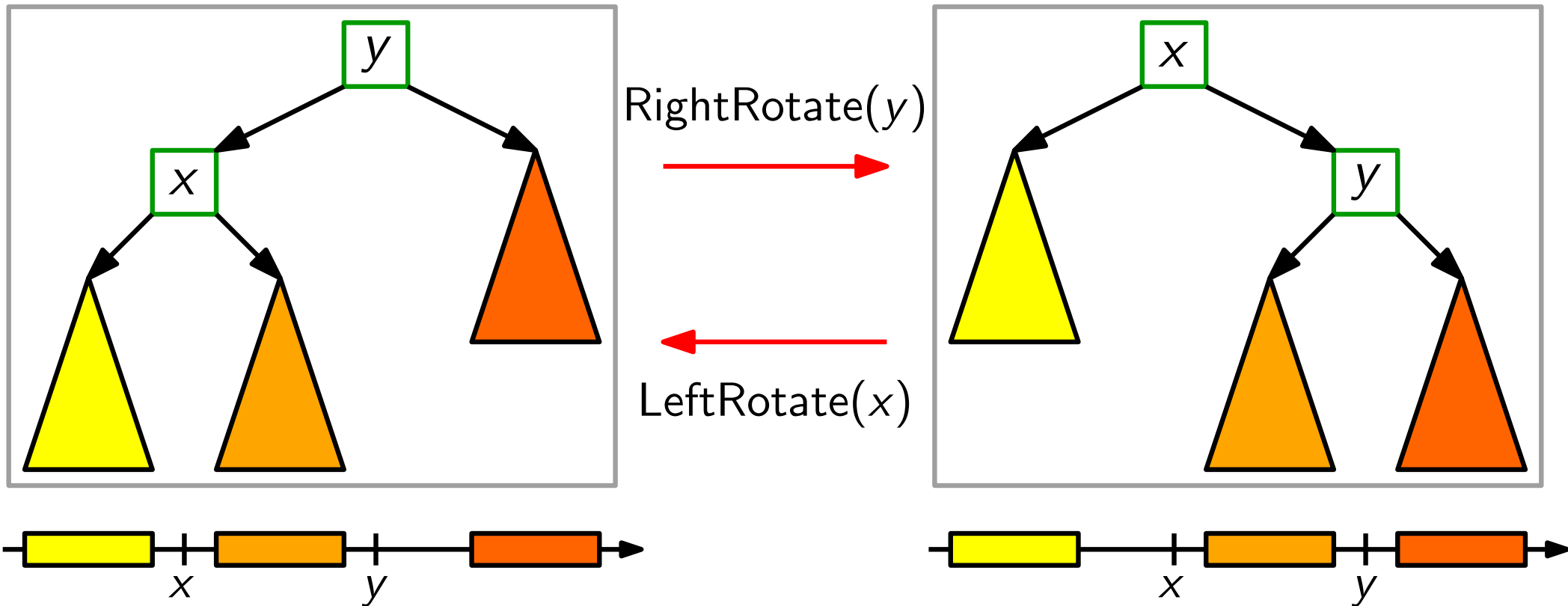


Also: Binärer-Suchbaum-Eig. bleibt beim Rotieren erhalten!

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

Laufzeit:

Exkurs: Rotationen



Also: Binärer-Suchbaum-Eig. bleibt beim Rotieren erhalten!

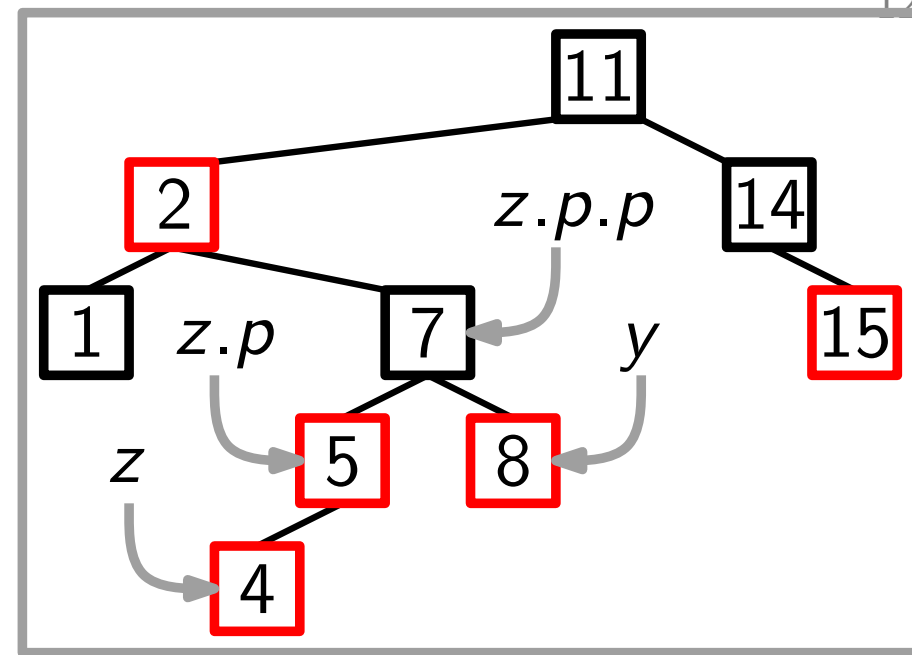
Aufgabe: Schreiben Sie Pseudocode für $\text{LeftRotate}(x)$!

Laufzeit: $O(1)$.



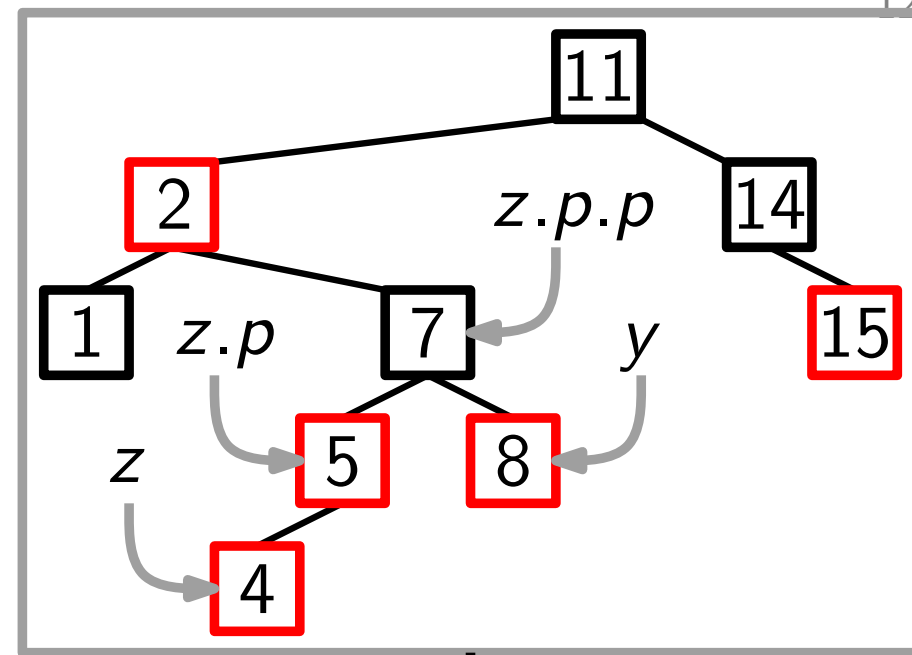
RBInsertFixup(Node z)

```
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
```



RBInsertFixup(Node z)

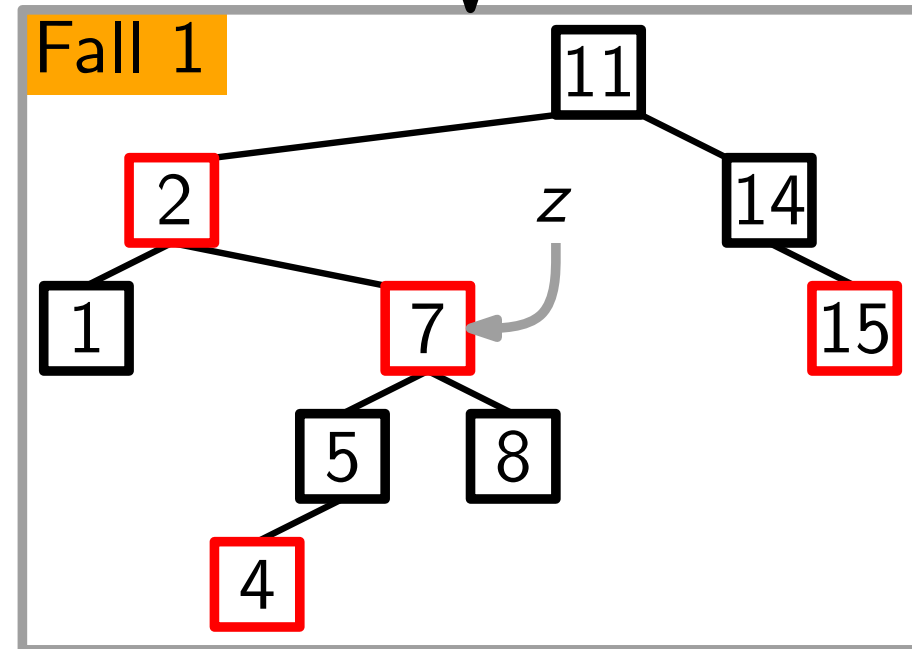
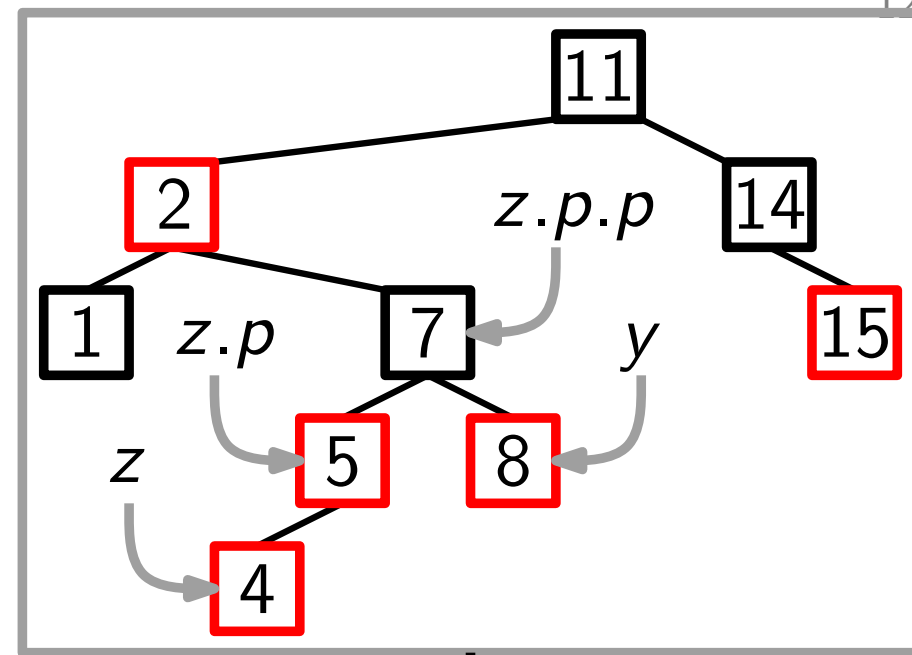
```
while z.p.color == red do  
  if z.p == z.p.p.left then  
    y = z.p.p.right // Tante von z  
    if y.color == red then  
      z.p.color = black  
      z.p.p.color = red  
      y.color = black  
      z = z.p.p  
    else  
      if z == z.p.right then  
        z = z.p  
        LeftRotate(z)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



Fall 1

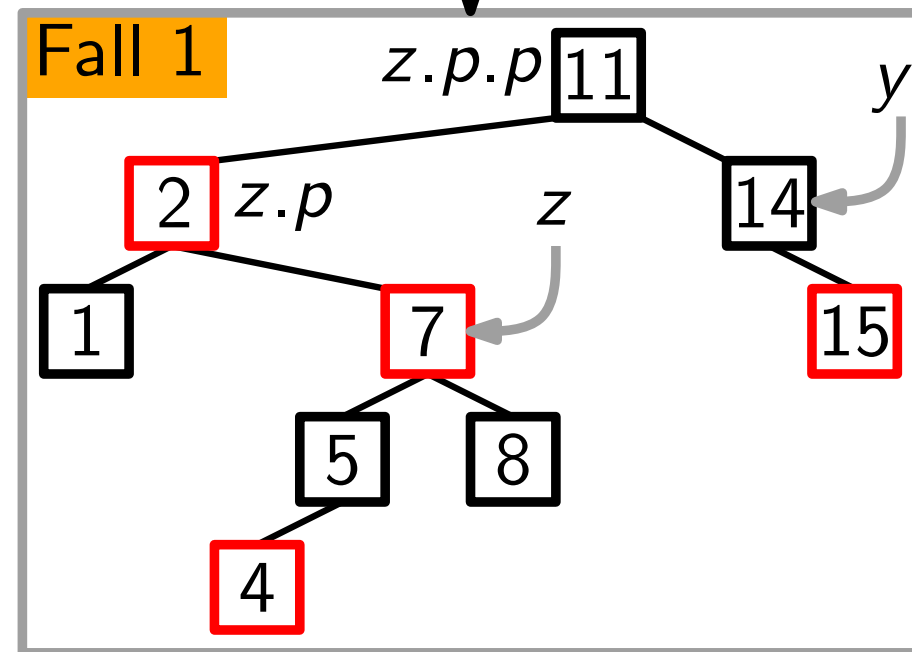
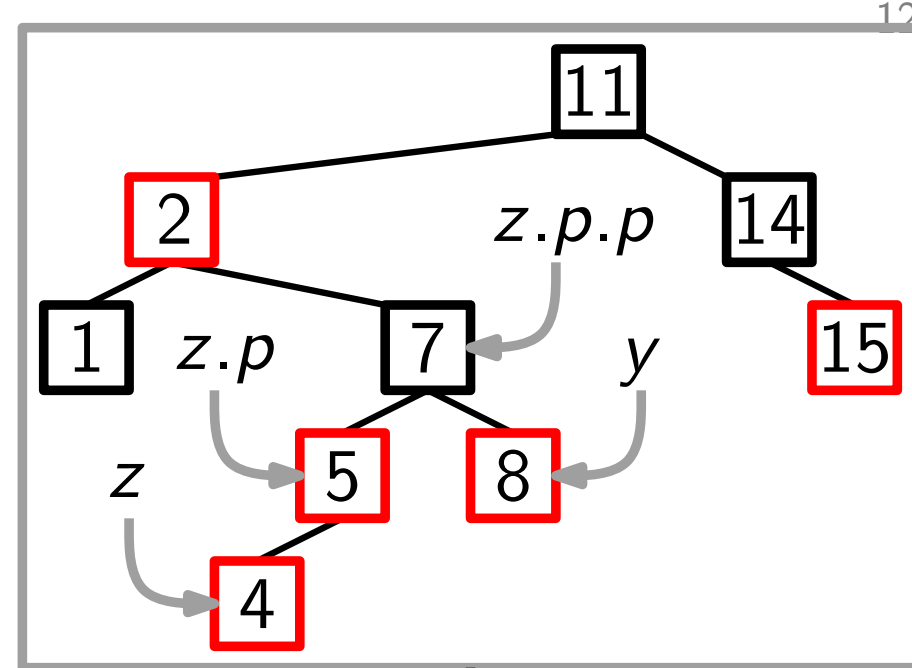
RBInsertFixup(Node z)

```
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
```



RBInsertFixup(Node z)

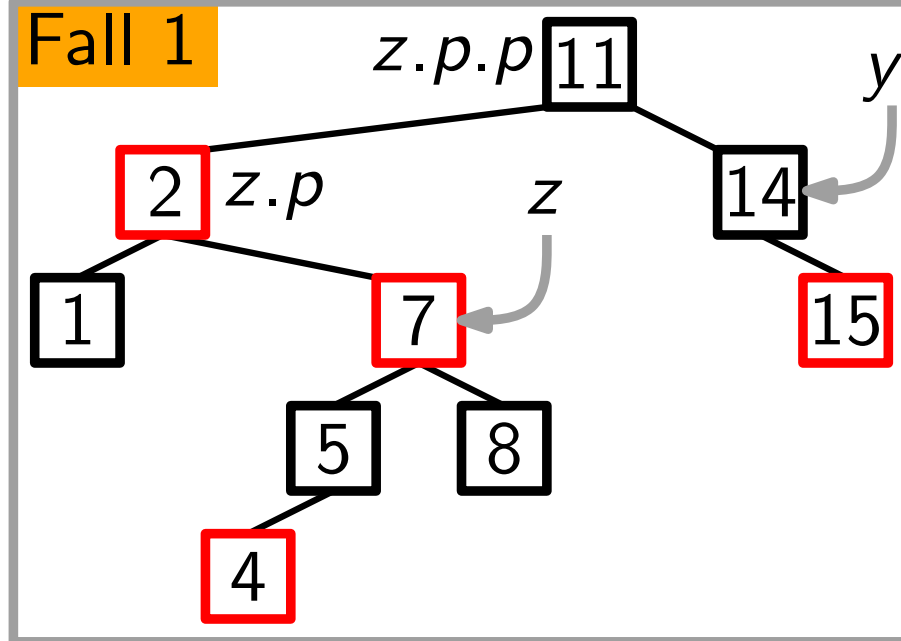
```
while z.p.color == red do
  if z.p == z.p.p.left then
    y = z.p.p.right // Tante von z
    if y.color == red then
      z.p.color = black
      z.p.p.color = red
      y.color = black
      z = z.p.p
    else
      if z == z.p.right then
        z = z.p
        LeftRotate(z)
      z.p.color = black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



RBInsertFixup(Node z)

```

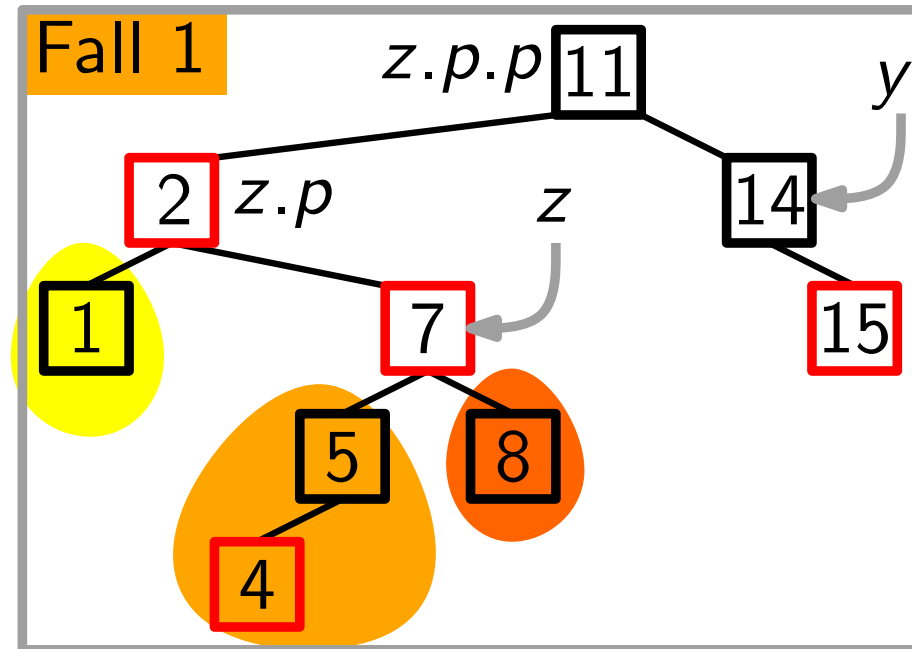
while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$  // Tante von z
    if  $y.color == red$  then
       $z.p.color = black$ 
       $z.p.p.color = red$ 
       $y.color = black$ 
       $z = z.p.p$ 
    else
      if  $z == z.p.right$  then
         $z = z.p$ 
    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



RBInsertFixup(Node z)

```

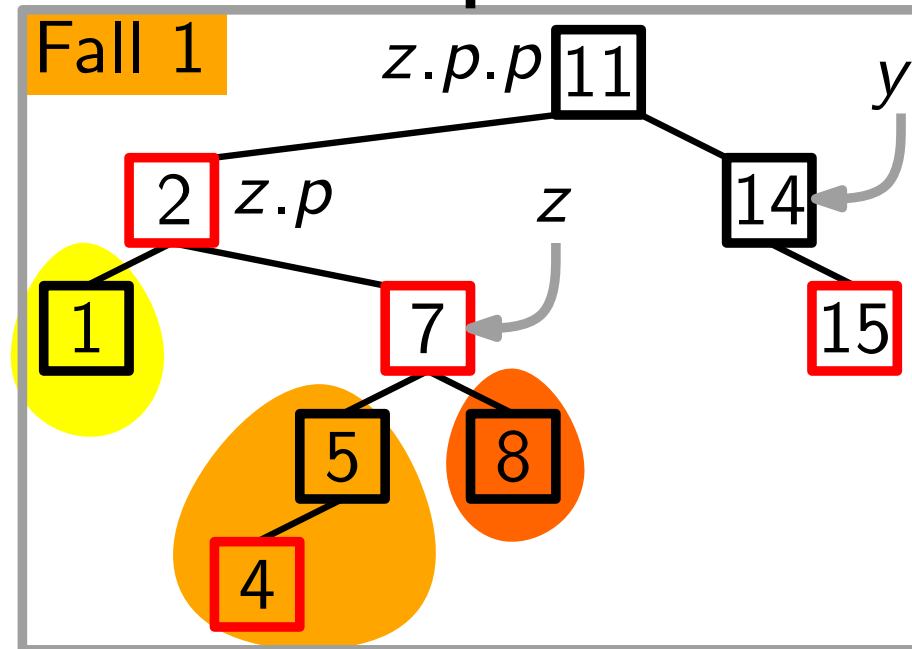
while z.p.color == red do
  if z.p == z.p.p.left then
    y = z.p.p.right    // Tante von z
    if y.color == red then
      z.p.color = black
      z.p.p.color = red
      y.color = black
      z = z.p.p
    else
      if z == z.p.right then
        z = z.p
        LeftRotate(z)
      z.p.color = black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



RBInsertFixup(Node z)

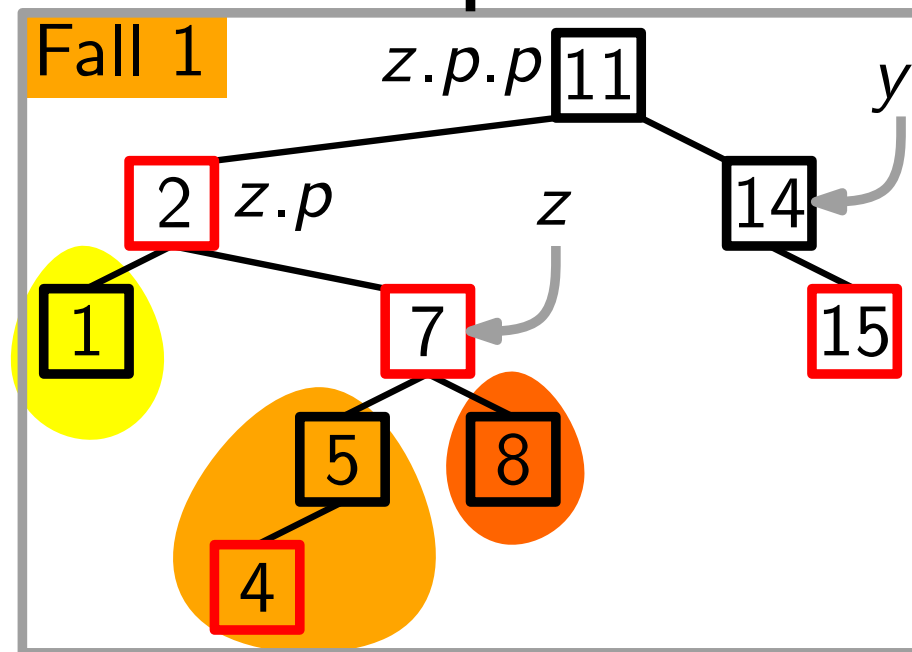
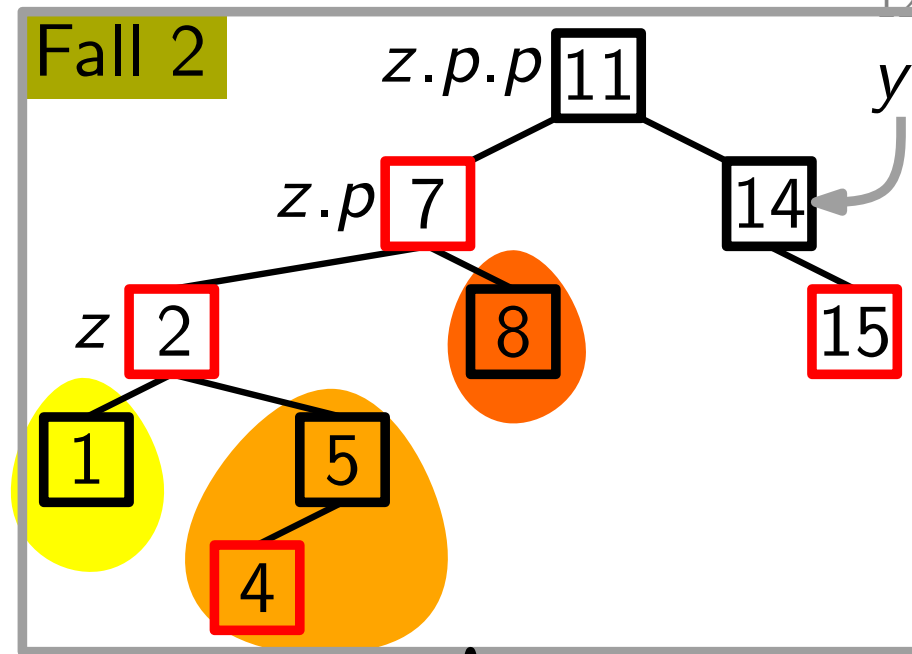
```
while z.p.color == red do  
  if z.p == z.p.p.left then  
    y = z.p.p.right // Tante von z  
    if y.color == red then  
      z.p.color = black  
      z.p.p.color = red  
      y.color = black  
      z = z.p.p  
    else  
      if z == z.p.right then  
        z = z.p  
        LeftRotate(z)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

Fall 2



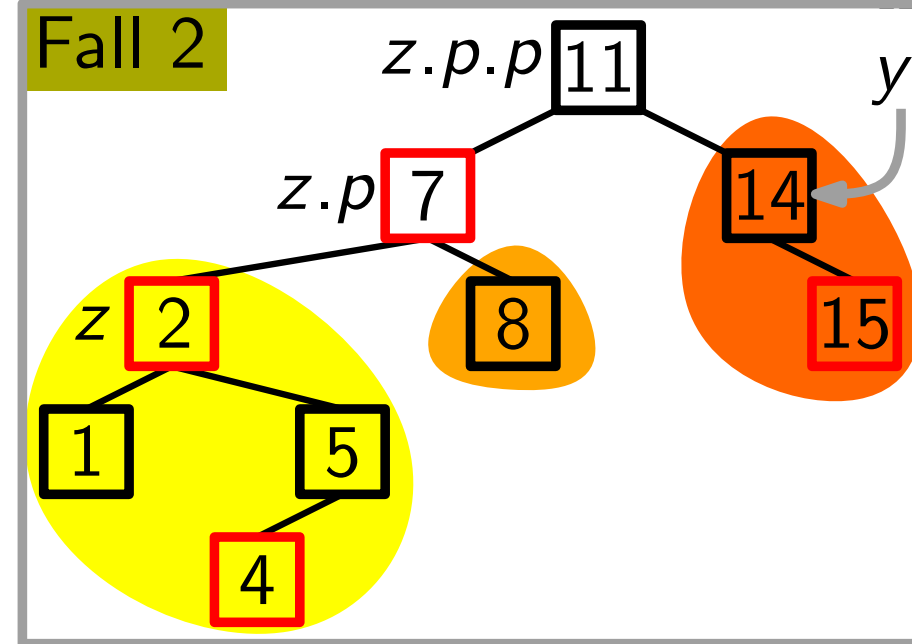
RBInsertFixup(Node z)

```
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
```



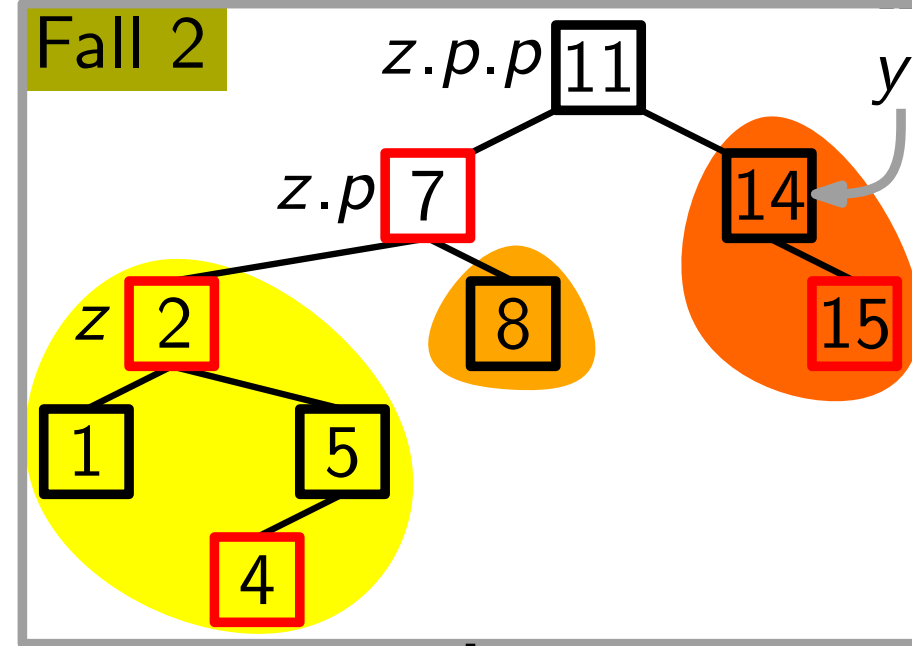
RBInsertFixup(Node z)

```
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
```



RBInsertFixup(Node 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
```

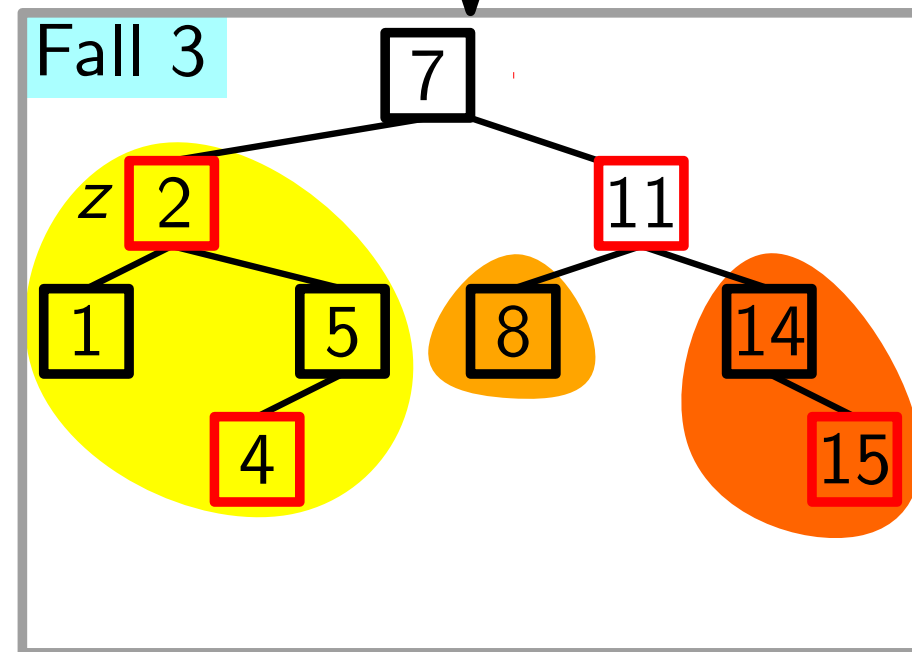
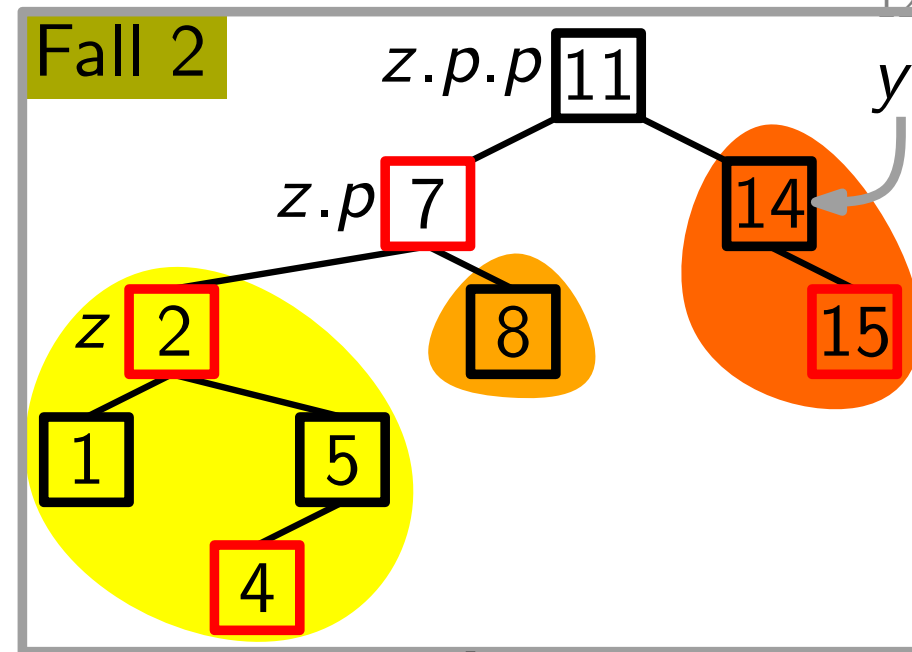


Fall 3

RBInsertFixup(Node 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.
- 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.

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:

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

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

Viel Arbeit! Siehe [CLRS, Kapitel 13.3].

Laufzeit RBInsertFixup

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$ 
    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
     $y = z.p.p.right$ 
    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
   $root.color = black$ 
  
```

$\left. \begin{array}{l} \text{if } y.color == red \text{ then} \\ \quad z.p.color = black \\ \quad z.p.p.color = red \\ \quad y.color = black \\ \quad z = z.p.p \end{array} \right\} O(1)$

$\left. \begin{array}{l} \text{if } z == z.p.right \text{ then} \\ \quad z = z.p \\ \quad \text{LeftRotate}(z) \end{array} \right\} O(1)$

$\left. \begin{array}{l} z.p.color = black \\ z.p.p.color = red \\ \text{RightRotate}(z.p.p) \end{array} \right\} O(1)$

Laufzeit RBInsertFixup

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$ 
    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
   $root.color = black$ 
  
```

} $O(1)$

Klettert im Baum
2 Ebenen nach oben.

} $O(1)$

} $O(1)$

Laufzeit RBInsertFixup

```

while  $z.p.color == red$  do
  if  $z.p == z.p.p.left$  then
     $y = z.p.p.right$ 
    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
   $root.color = black$ 
  
```

} $O(1)$

Klettert im Baum
2 Ebenen nach oben.

} $O(1)$

Führt zum Abbruch
der while-Schleife.

} $O(1)$

Laufzeit RBInsertFixup

Insgesamt:

- Fall 1 $O(h)$ mal
- Fall 2 ≤ 1 mal
- Fall 3 ≤ 1 mal

```

while z.p.color == red do
  if z.p == z.p.p.left then
    y = z.p.p.right
    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
  
```

} $O(1)$

Klettert im Baum
2 Ebenen nach oben.

} $O(1)$

Führt zum Abbruch
der while-Schleife.

} $O(1)$

Laufzeit RBInsertFixup

Insgesamt:

- Fall 1 $O(h)$ mal
- Fall 2 ≤ 1 mal
- Fall 3 ≤ 1 mal

```

while z.p.color == red do
  if z.p == z.p.p.left then
    y = z.p.p.right
    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
  
```

} $O(1)$

Klettert im Baum
2 Ebenen nach oben.

} $O(1)$

Führt zum Abbruch
der while-Schleife.

} $O(1)$

Laufzeit RBInsertFixup

```

while z.p.color == red do
  if z.p == z.p.p.left then
    y = z.p.p.right
    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 $O(h)$ mal
- Fall 2 ≤ 1 mal
- Fall 3 ≤ 1 mal

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

} $O(1)$

Klettert im Baum
2 Ebenen nach oben.

} $O(1)$

Führt zum Abbruch
der while-Schleife.

} $O(1)$

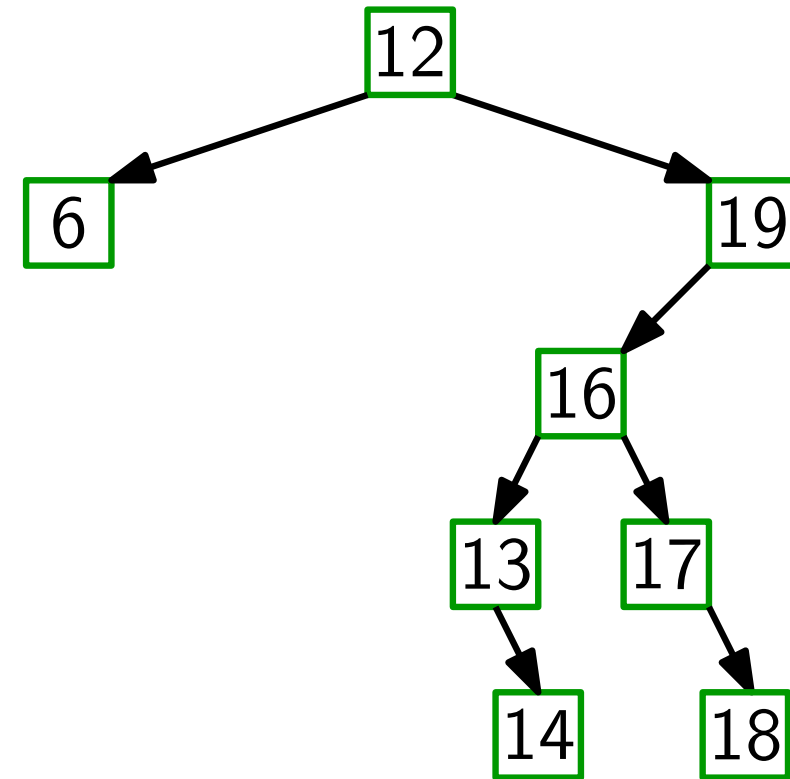
Löschen in (farblosen) binären Suchbäumen

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

1. z hat kein li. Kind.

2. z hat kein re. Kind.

3. z hat zwei Kinder.



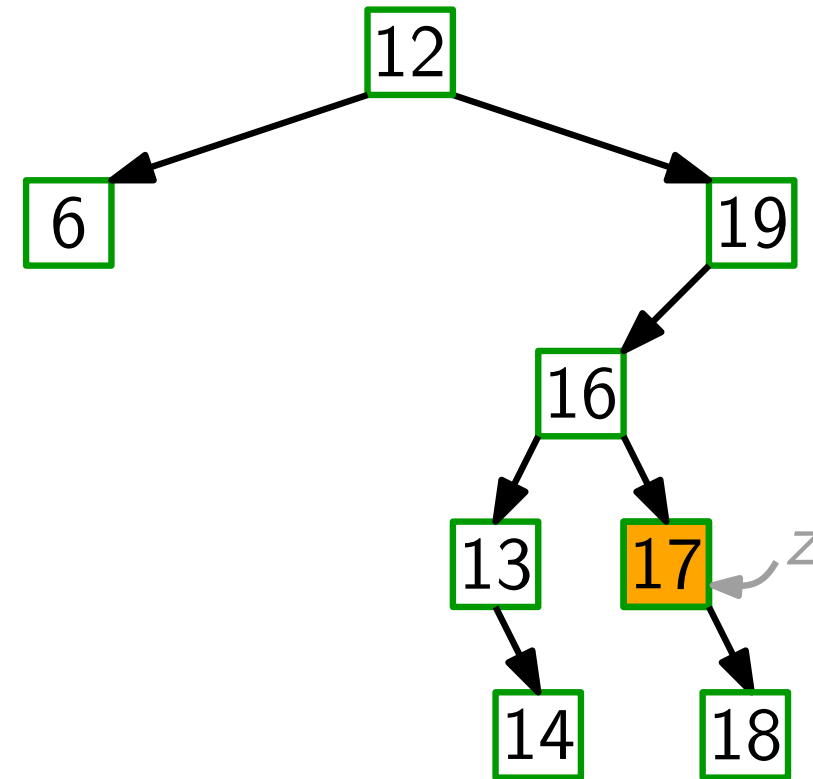
Löschen in (farblosen) binären Suchbäumen

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

1. z hat kein li. Kind.

2. z hat kein re. Kind.

3. z hat zwei Kinder.



Löschen in (farblosen) binären Suchbäumen

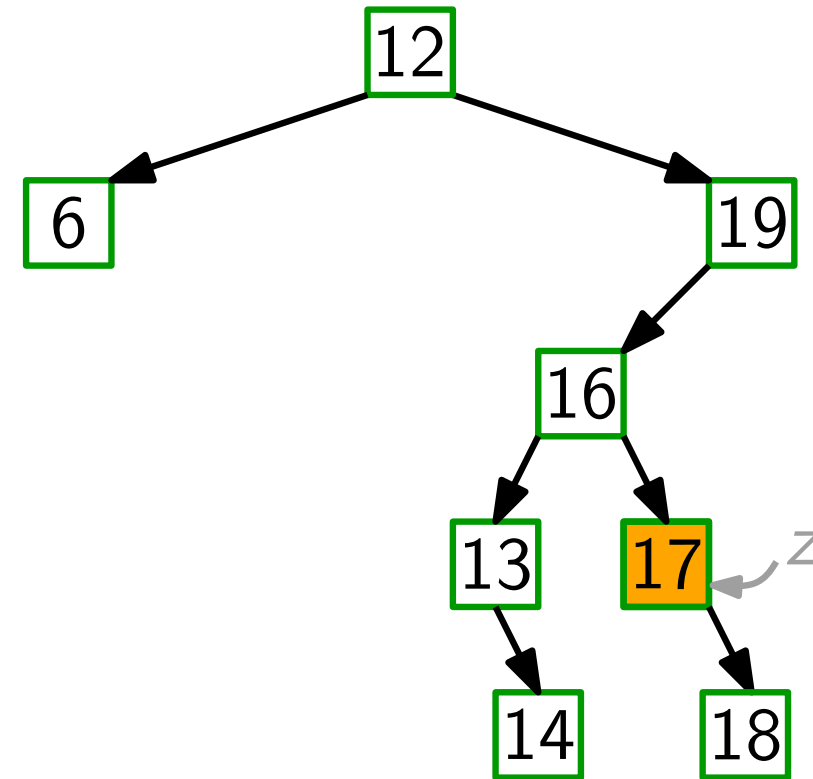
Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat kein li. Kind.

Setze $z.right$ an die Stelle von z .
Lösche z .

2. z hat kein re. Kind.

3. z hat zwei Kinder.



Löschen in (farblosen) binären Suchbäumen

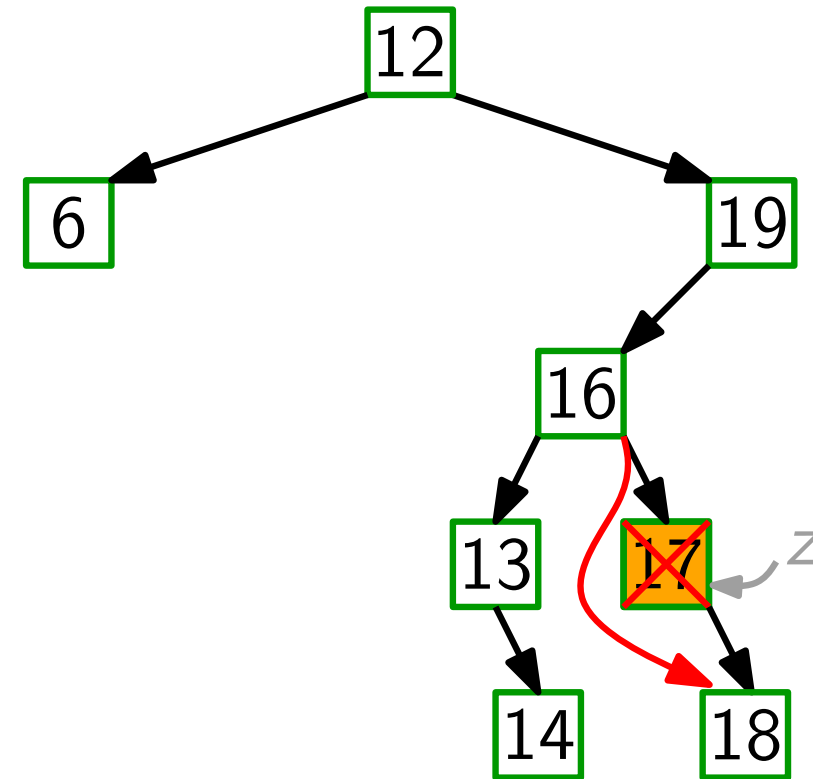
Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat kein li. Kind.

Setze $z.right$ an die Stelle von z .
Lösche z .

2. z hat kein re. Kind.

3. z hat zwei Kinder.



Löschen in (farblosen) binären Suchbäumen

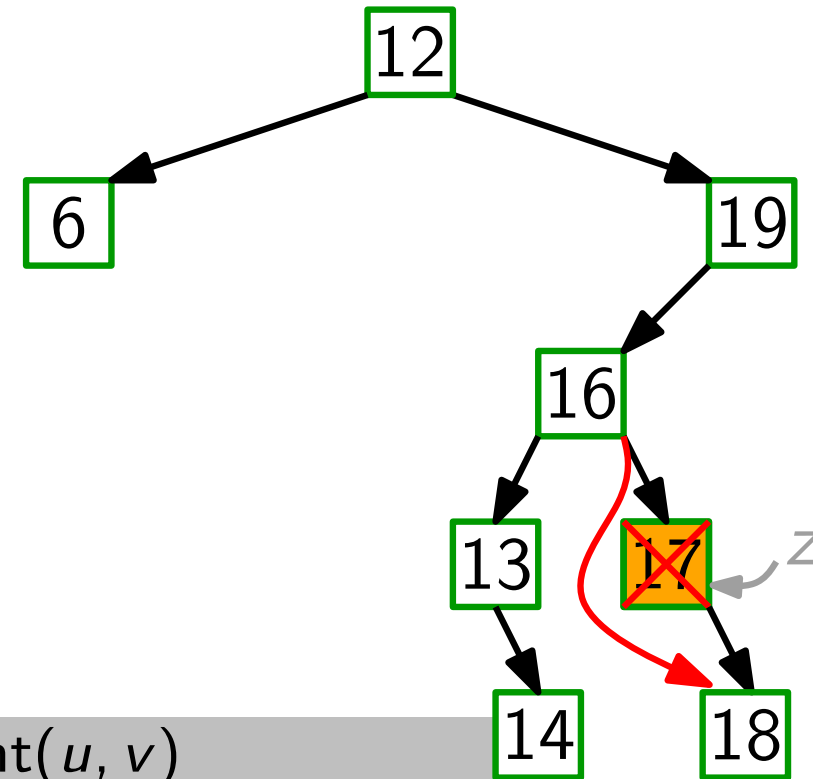
Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat kein li. Kind.

Setze $z.right$ an die Stelle von z .
Lösche z .

2. z hat kein re. Kind.

3. z hat zwei Kinder.



Transplant(u, v)

if $u.p == nil$ then $root = v$

else

if $u == u.p.left$ then

└ $u.p.left = v$

else $u.p.right = v$

if $v \neq nil$ then $v.p = u.p$

Löschen in (farblosen) binären Suchbäumen

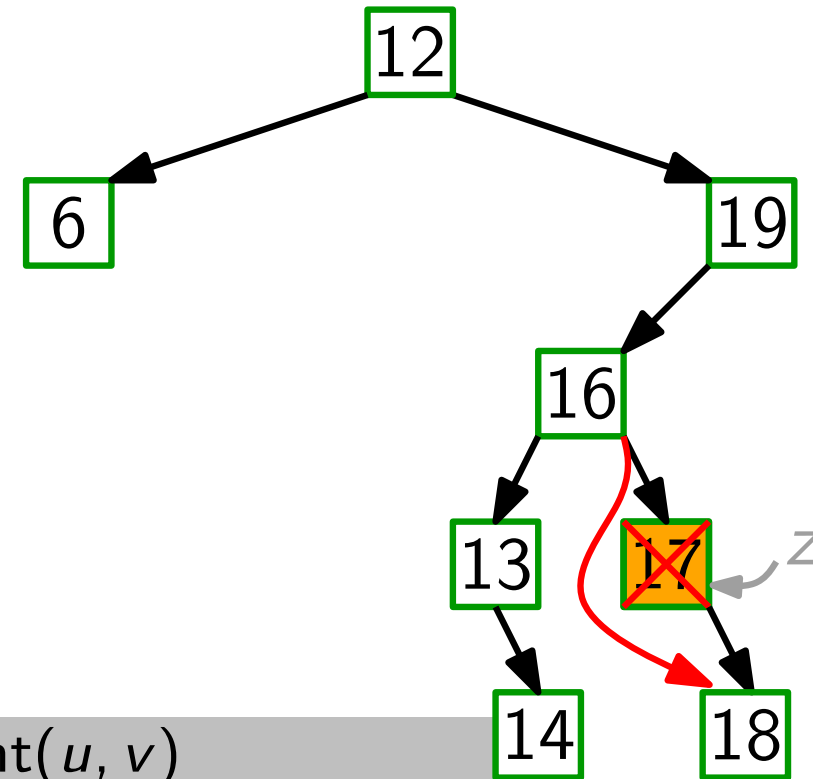
Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat kein li. Kind.

Setze $z.right$ an die Stelle von z .
Lösche z .

2. z hat kein re. Kind.

3. z hat zwei Kinder.



Transplant(u, v)

if $u.p == nil$ then $root = v$
else

if $u == u.p.left$ then

└ $u.p.left = v$

else $u.p.right = v$

if $v \neq nil$ then $v.p = u.p$

Setze v
an die
Stelle
von u .

Löschen in (farblosen) binären Suchbäumen

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

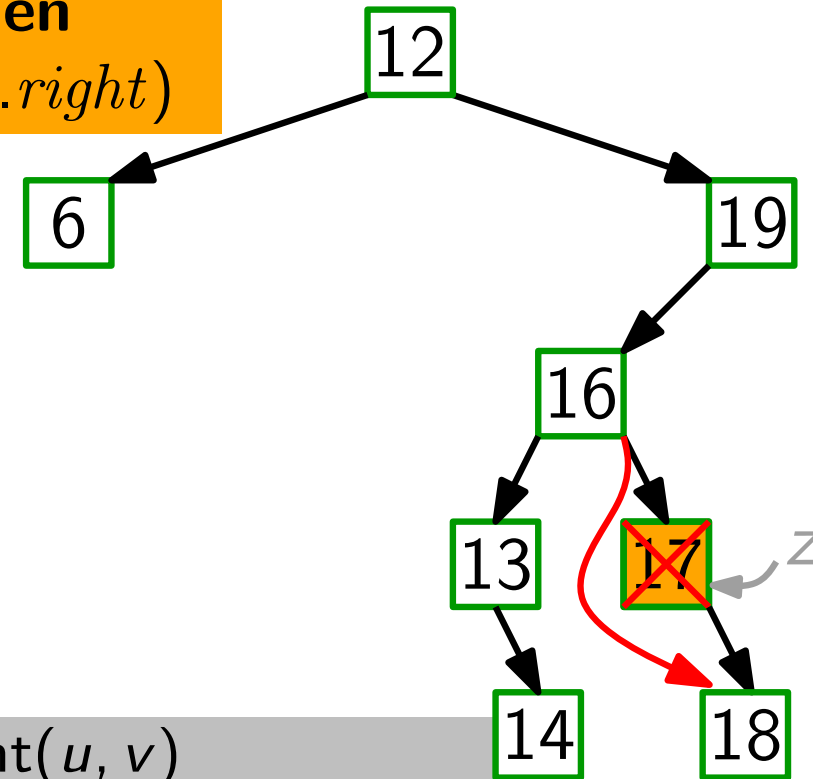
1. z hat kein li. Kind.

if $z.left == nil$ then
 Transplant($z, z.right$)

Setze $z.right$ an die Stelle von z .
 Lösche z .

2. z hat kein re. Kind.

3. z hat zwei Kinder.



Transplant(u, v)

if $u.p == nil$ then $root = v$
 else

 if $u == u.p.left$ then

$u.p.left = v$

 else $u.p.right = v$

if $v \neq nil$ then $v.p = u.p$

Setze v
 an die
 Stelle
 von u .

1. z hat kein li. Kind.

```

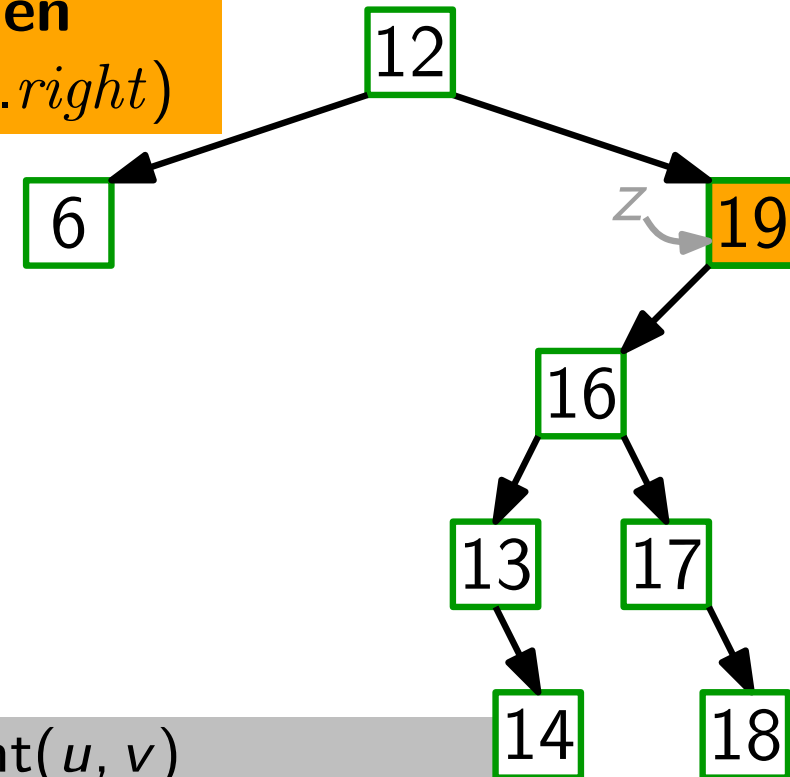
if  $z.left == nil$  then
    Transplant( $z, z.right$ )

```

Setze $z.right$ an die Stelle von z .
 Lösche z .

2. z hat kein re. Kind.

3. z hat zwei Kinder.



Transplant(u, v)

```
if u.p == nil then root = v
else
```

if $u == u.p.left$ then

$$| \quad u.p.left \quad = \quad v$$

```
else u.p.right = v
```

if $v \neq nil$ **then** $v.p = u.p$

Setze v
an die
Stelle
von u .


```

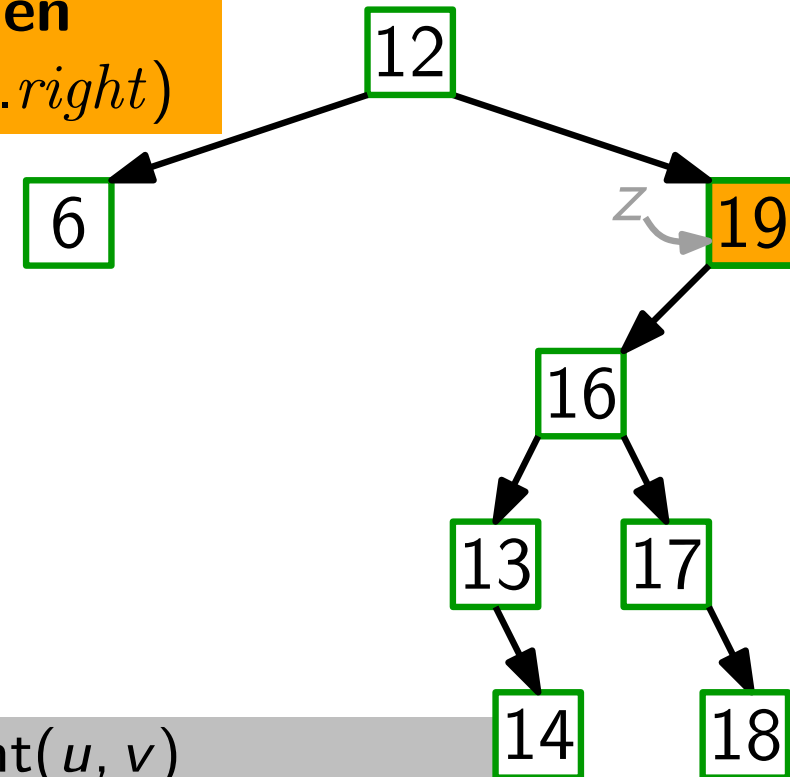
if  $z.left == nil$  then
    Transplant( $z, z.right$ )

```

- Setze $z.right$ an die Stelle von z .
Lösche z .

- 2.** z hat kein re. Kind.
symmetrisch!

- ### 3. z hat zwei Kinder.



Transplant(u, v)

```
if u.p == nil then root = v
else
```

if $u == u.p.left$ then

$$| \quad u.p.left \quad = \quad v$$

```
else u.p.right = v
```

if $v \neq nil$ **then** $v.p = u.p$

Setze v
an die
Stelle
von u .

Löschen in (farblosen) binären Suchbäumen

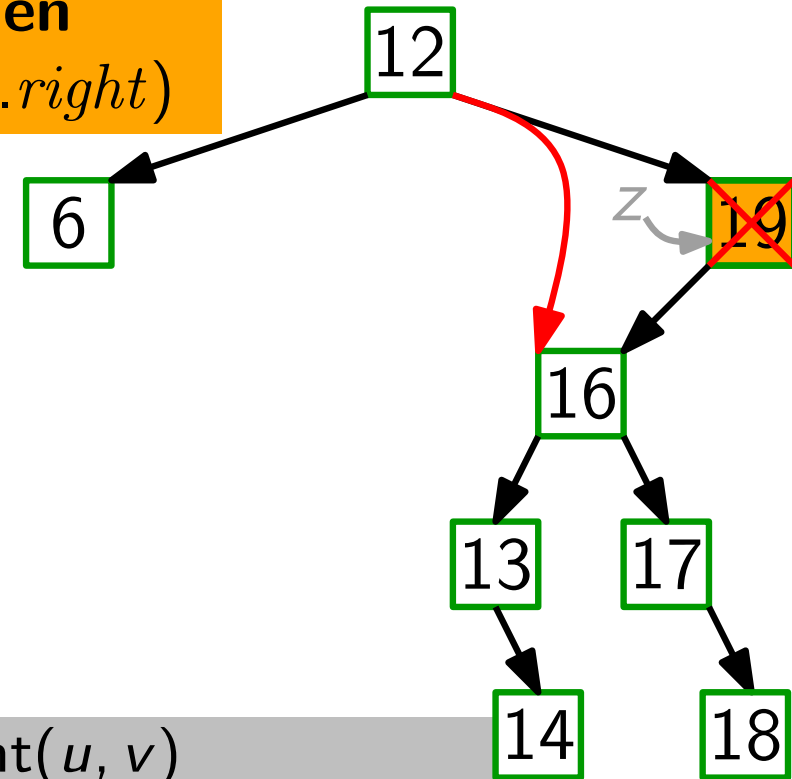
Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat kein li. Kind. **if $z.left == nil$ then**
 Transplant($z, z.right$)

Setze $z.right$ an die Stelle von z .
 Lösche z .

2. z hat kein re. Kind.
symmetrisch!

3. z hat zwei Kinder.



Transplant(u, v)

if $u.p == nil$ then $root = v$
 else

 if $u == u.p.left$ then

$u.p.left = v$

 else $u.p.right = v$

if $v \neq nil$ then $v.p = u.p$

Setze v
 an die
 Stelle
 von u .

Löschen in (farblosen) binären Suchbäumen

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

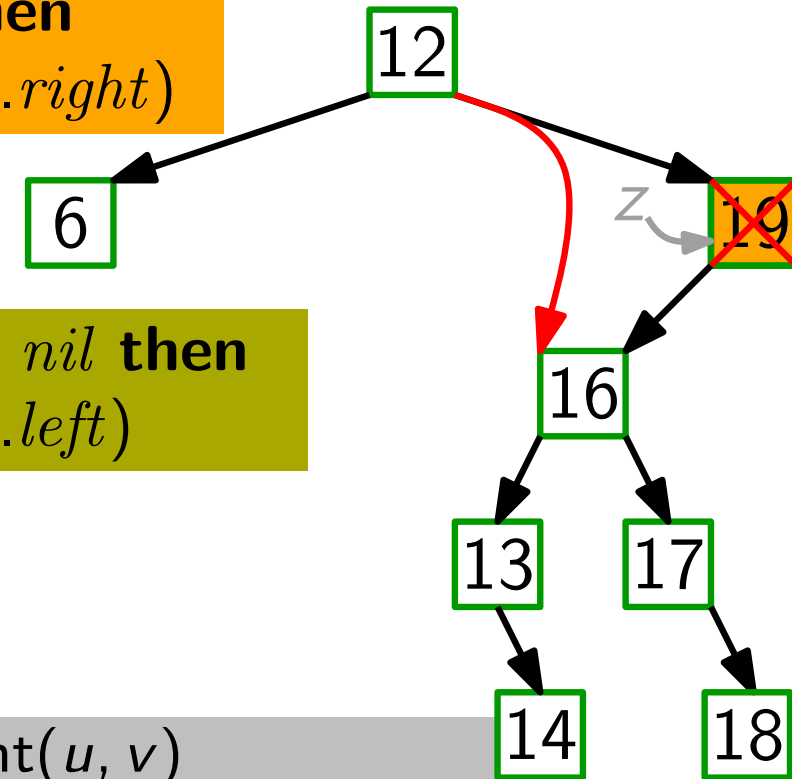
1. z hat kein li. Kind. **if $z.left == nil$ then**
 Transplant($z, z.right$)

Setze $z.right$ an die Stelle von z .
 Lösche z .

2. z hat kein re. Kind.
symmetrisch!

else if $z.right == nil$ then
 Transplant($z, z.left$)

3. z hat zwei Kinder.



Transplant(u, v)

if $u.p == nil$ then $root = v$
else

if $u == u.p.left$ then

$u.p.left = v$

else $u.p.right = v$

if $v \neq nil$ then $v.p = u.p$

Setze v
 an die
 Stelle
 von u .

Löschen in (farblosen) binären Suchbäumen

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

1. z hat kein li. Kind. **if $z.left == nil$ then**
 Transplant($z, z.right$)

Setze $z.right$ an die Stelle von z .
 Lösche z .

2. z hat kein re. Kind. **else if $z.right == nil$ then**
 Transplant($z, z.left$)

symmetrisch!

3. z hat zwei Kinder.

Transplant(u, v)

if $u.p == nil$ then $root = v$

else

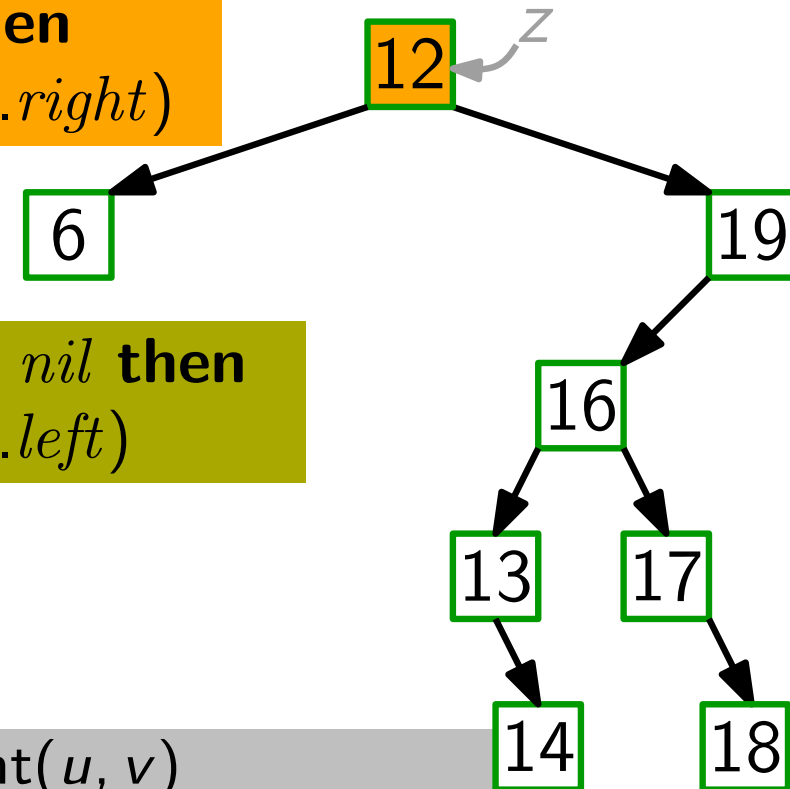
if $u == u.p.left$ then

$u.p.left = v$

else $u.p.right = v$

if $v \neq nil$ then $v.p = u.p$

Setze v
 an die
 Stelle
 von u .



1. z hat kein li. Kind.

```

if z.left == nil then
    Transplant(z, z.right)

```

Setze $z.right$ an die Stelle von z .
Lösche z .

2. z hat kein re. Kind.
symmetrisch!

```

else if  $z.right == nil$  then
    Transplant( $z, z.left$ )

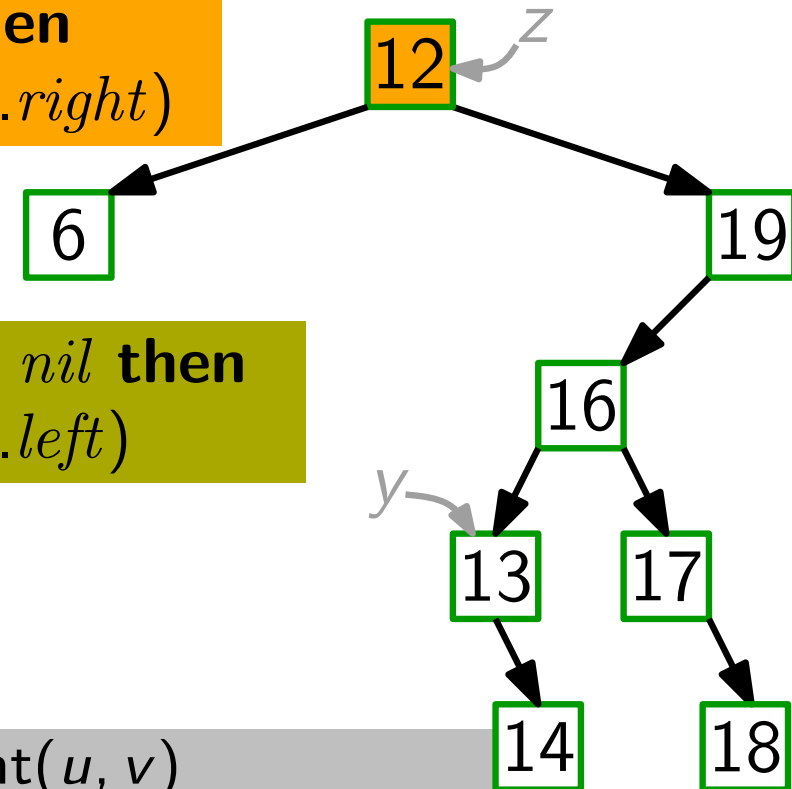
```

3. z hat zwei Kinder.

Setze $y = \text{Successor}(z)$

Falls $y.p \neq z$, setze $y.right$
an die Stelle von y .

Setze y an die Stelle von z



Transplant(u, v)

```
if u.p == nil then root = v
else
```

if $u == u.p.left$ then

$$| \quad u.p.left \quad = \quad v$$

```
else u.p.right = v
```

if $v \neq nil$ then $v.p = u.p$

Setze v
an die
Stelle
von u .

Löschen in (farblosen) binären Suchbäumen

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

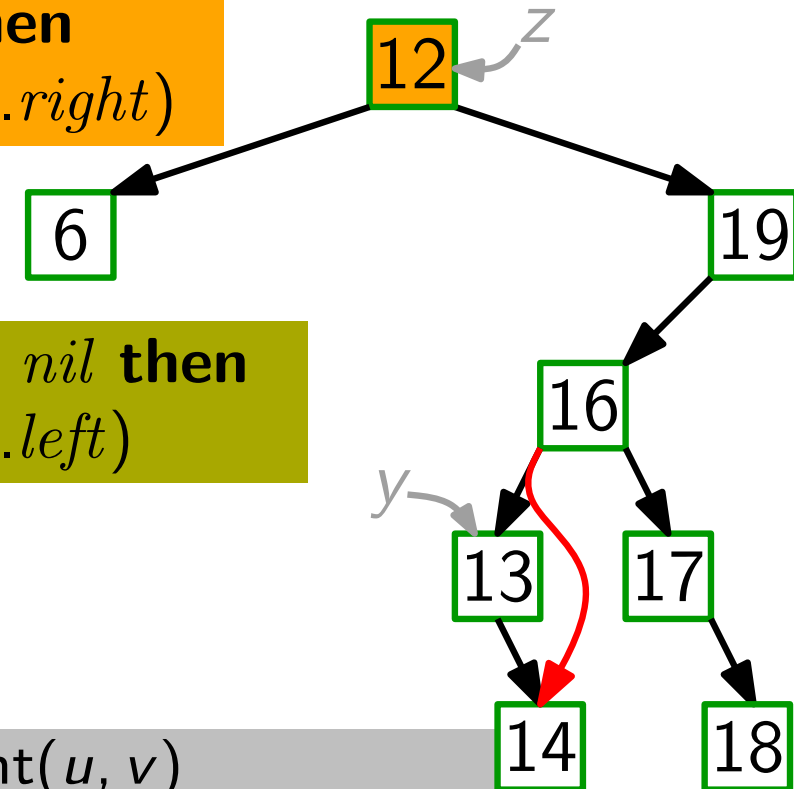
1. z hat kein li. Kind. **if $z.left == nil$ then**
 Transplant($z, z.right$)

Setze $z.right$ an die Stelle von z .
 Lösche z .

2. z hat kein re. Kind. **else if $z.right == nil$ then**
 Transplant($z, z.left$)
symmetrisch!

3. z hat zwei Kinder.

Setze $y = \text{Successor}(z)$
 Falls $y.p \neq z$, setze $y.right$
 an die Stelle von y .
 Setze y an die Stelle von z



Transplant(u, v)

if $u.p == nil$ then $root = v$
else

if $u == u.p.left$ then

$u.p.left = v$

else $u.p.right = v$

if $v \neq nil$ then $v.p = u.p$

Setze v
 an die
 Stelle
 von u .

1. z hat kein li. Kind.

```

if z.left == nil then
    Transplant(z, z.right)

```

Setze $z.right$ an die Stelle von z .
Lösche z .

2. z hat kein re. Kind.
symmetrisch!

```

else if  $z.right == nil$  then
    Transplant( $z, z.left$ )

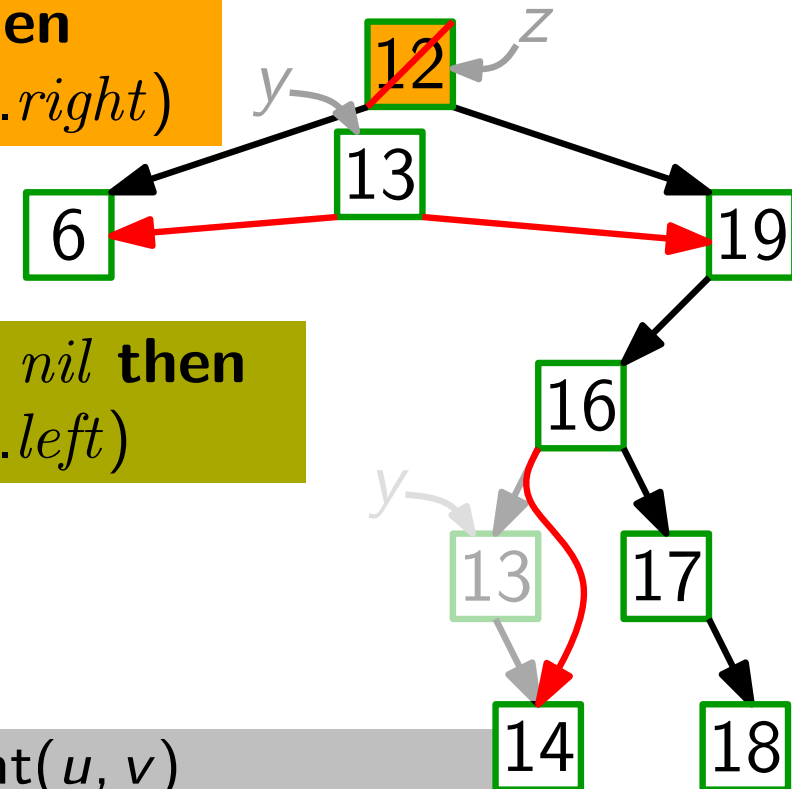
```

3. z hat zwei Kinder.

Setze $y = \text{Successor}(z)$

Falls $y.p \neq z$, setze $y.right$
an die Stelle von y .

Setze y an die Stelle von z



Transplant(u, v)

```
if u.p == nil then root = v
else
```

if $u == u.p.left$ then

$$| \quad u.p.left \quad = \quad v$$

```
else u.p.right = v
```

if $v \neq nil$ then $v.p = u.p$

Setze v
an die
Stelle
von u .

Löschen in (farblosen) binären Suchbäumen

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

1. z hat kein li. Kind. **if $z.left == nil$ then**
 Transplant($z, z.right$)

Setze $z.right$ an die Stelle von z .
 Lösche z .

2. z hat kein re. Kind. **else if $z.right == nil$ then**
 Transplant($z, z.left$)
symmetrisch!

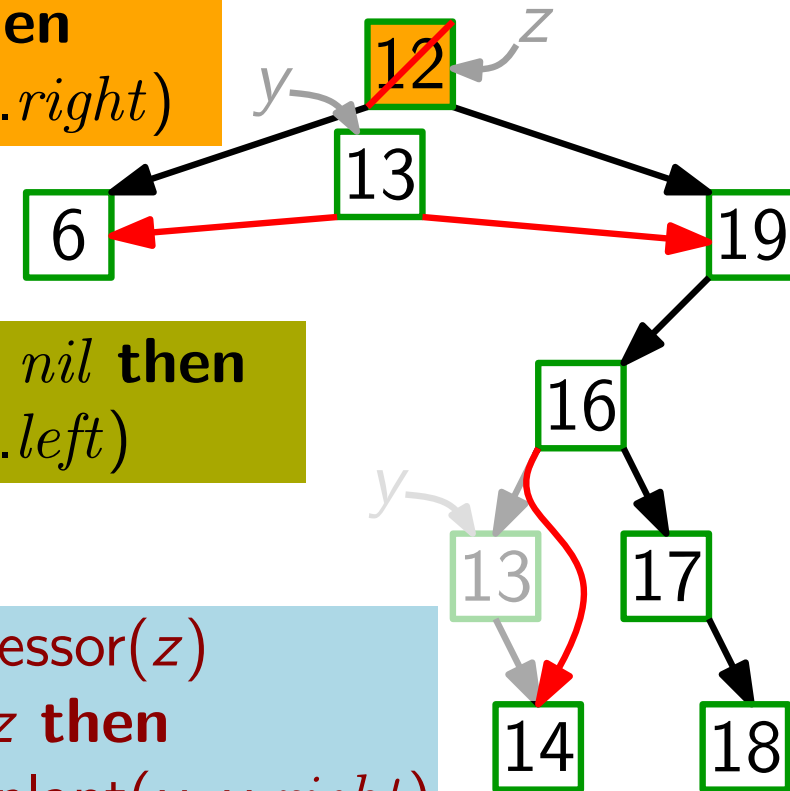
3. z hat zwei Kinder.

Setze $y = \text{Successor}(z)$

Falls $y.p \neq z$, setze $y.right$
 an die Stelle von y .

Setze y an die Stelle von z

$y = \text{Successor}(z)$
if $y.p \neq z$ then
 Transplant($y, y.right$)
 $y.right = z.right$
 $y.right.p = y$
Transplant(z, y)
 $y.left = z.left$
 $y.left.p = y$



Löschen (Übersicht)

Delete(Node *z*)

```
if z.left == nil then           // kein linkes Kind  
|   Transplant(z, z.right)
```

else

```
if z.right == nil then         // kein rechtes Kind  
|   Transplant(z, z.left)
```

```
else                             // zwei Kinder
```

```
    y = Successor(z)
```

```
    if y.p ≠ z then
```

```
        |   Transplant(y, y.right)
```

```
        |   y.right = z.right
```

```
        |   y.right.p = y
```

```
    Transplant(z, y)
```

```
    y.left = z.left
```

```
    |   y.left.p = y
```

RBDelete(Node z)

```
if z.left == T.nil then
    RBTransplant(z, z.right)
else
    if z.right == T.nil then
        RBTransplant(z, z.left)
    else
        y = Successor(z)

        if y.p == z then
            else
                RBTransplant(y, y.right)
                y.right = z.right
                y.right.p = y
            RBTransplant(z, y)
            y.left = z.left
            y.left.p = y
```

RBTransplant(*u*, *v*)

```
if u.p == T.nil then root = v
else
    if u == u.p.left then
        u.p.left = v
    else u.p.right = v
if v ≠ nil then v.p = u.p
```

RBDelete(Node z)

```
if  $z.left == T.nil$  then
    RBTransplant( $z, z.right$ )
else
    if  $z.right == T.nil$  then
        RBTransplant( $z, z.left$ )
    else
         $y = \text{Successor}(z)$ 

        if  $y.p == z$  then
            else
                RBTransplant( $y, y.right$ )
                 $y.right = z.right$ 
                 $y.right.p = y$ 
            RBTransplant( $z, y$ )
             $y.left = z.left$ 
             $y.left.p = y$ 
```

RBTransplant(u, v)⁷

```
if  $u.p == T.nil$  then  $root = v$ 
else
    if  $u == u.p.left$  then
         $u.p.left = v$ 
    else  $u.p.right = v$ 
if  $v \neq nil$  then  $v.p = u.p$ 
```

```

RBDelete(Node z)
  y = z; origcolor = y.color
  if z.left == T.nil then
    | x = z.right
    | RBTransplant(z, z.right)
  else
    if z.right == T.nil then
      | x = z.left
      | RBTransplant(z, z.left)
    else
      y = Successor(z)
      origcolor = y.color
      x = y.right
      if y.p == z then x.p = y
      else
        RBTransplant(y, y.right)
        y.right = z.right
        y.right.p = y
      RBTransplant(z, y)
      y.left = z.left
      y.left.p = y; y.color = z.color

  if origcolor == black then RBDeleteFixup(x)

```

y zeigt auf den Knoten, der entweder gelöscht oder verschoben wird.

x zeigt auf den Knoten, der die Stelle von *y* einnimmt – das ist entweder das einzige Kind von *y* oder *T.nil*.

RBDelete(Node z)

y = z; origcolor = y.color

if *z.left == T.nil* **then**

x = z.right

 RBTransplant(*z, z.right*)

else

if *z.right == T.nil* **then**

x = z.left

 RBTransplant(*z, z.left*)

else

y = Successor(z)

origcolor = y.color

x = y.right

if *y.p == z* **then** *x.p = y*

else

 RBTransplant(*y, y.right*)

y.right = z.right

y.right.p = y

 RBTransplant(*z, y*)

y.left = z.left

y.left.p = y; y.color = z.color

if *origcolor == black* **then** RBDeleteFixup(*x*)

y zeigt auf den Knoten, der entweder gelöscht oder verschoben wird.

x zeigt auf den Knoten, der die Stelle von *y* einnimmt – das ist entweder das einzige Kind von *y* oder *T.nil*.

Falls *y* ursprünglich *rot* war, bleiben alle R-S-Eig. erhalten:

- Keine Schwarzhöhe hat sich verändert.
- Keine zwei roten Knoten sind Nachbarn geworden.
- *y* rot $\Rightarrow y \neq \text{Wurzel} \Rightarrow \text{Wurzel bleibt schwarz}$.

RBDeleteFixup

Was kann schief gehen, wenn y schwarz war?

RBDeleteFixup

Was kann schief gehen, wenn y schwarz war?

(E2) y war Wurzel, und ein rotes Kind von y wurde Wurzel.

RBDeleteFixup

Was kann schief gehen, wenn y schwarz war?

- (E2) y war Wurzel, und ein rotes Kind von y wurde Wurzel.
- (E4) x und $x.p$ sind rot.

RBDeleteFixup

Was kann schief gehen, wenn y schwarz war?

- (E2) y war Wurzel, und ein rotes Kind von y wurde Wurzel.
- (E4) x und $x.p$ sind rot.
- (E5) Falls y verschoben wurde, haben jetzt alle Pfade, die vorher y enthielten, einen schwarzen Knoten zu wenig.

RBDeleteFixup

Was kann schief gehen, wenn y schwarz war?

- (E2) y war Wurzel, und ein rotes Kind von y wurde Wurzel.
- (E4) x und $x.p$ sind rot.
- (E5) Falls y verschoben wurde, haben jetzt alle Pfade, die vorher y enthielten, einen schwarzen Knoten zu wenig.

„Repariere“ Knoten x zählt eine schwarze Einheit extra
(E5): (ist also „rot-schwarz“ oder „doppelt schwarz“)

RBDeleteFixup

Was kann schief gehen, wenn y schwarz war?

- (E2) y war Wurzel, und ein rotes Kind von y wurde Wurzel.
- (E4) x und $x.p$ sind rot.
- (E5) Falls y verschoben wurde, haben jetzt alle Pfade, die vorher y enthielten, einen schwarzen Knoten zu wenig.

„Repariere“ Knoten x zählt eine schwarze Einheit extra
(E5): (ist also „rot-schwarz“ oder „doppelt schwarz“)

Ziel: Schiebe die überzählige schwarze Einheit nach oben, bis:

RBDeleteFixup

Was kann schief gehen, wenn y schwarz war?

- (E2) y war Wurzel, und ein rotes Kind von y wurde Wurzel.
- (E4) x und $x.p$ sind rot.
- (E5) Falls y verschoben wurde, haben jetzt alle Pfade, die vorher y enthielten, einen schwarzen Knoten zu wenig.

„Repariere“ Knoten x zählt eine schwarze Einheit extra
 (E5): (ist also „rot-schwarz“ oder „doppelt schwarz“)

Ziel: Schiebe die überzählige schwarze Einheit nach oben, bis:

- x ist rot-schwarz \Rightarrow mach x schwarz.

RBDeleteFixup

Was kann schief gehen, wenn y schwarz war?

- (E2) y war Wurzel, und ein rotes Kind von y wurde Wurzel.
- (E4) x und $x.p$ sind rot.
- (E5) Falls y verschoben wurde, haben jetzt alle Pfade, die vorher y enthielten, einen schwarzen Knoten zu wenig.

„Repariere“ Knoten x zählt eine schwarze Einheit extra
(E5): (ist also „rot-schwarz“ oder „doppelt schwarz“)

- Ziel:** Schiebe die überzählige schwarze Einheit nach oben, bis:
- x ist rot-schwarz \Rightarrow mach x schwarz.
 - x ist Wurzel \Rightarrow schwarze Extra-Einheit verfällt.

RBDeleteFixup

Was kann schief gehen, wenn y schwarz war?

- (E2) y war Wurzel, und ein rotes Kind von y wurde Wurzel.
- (E4) x und $x.p$ sind rot.
- (E5) Falls y verschoben wurde, haben jetzt alle Pfade, die vorher y enthielten, einen schwarzen Knoten zu wenig.

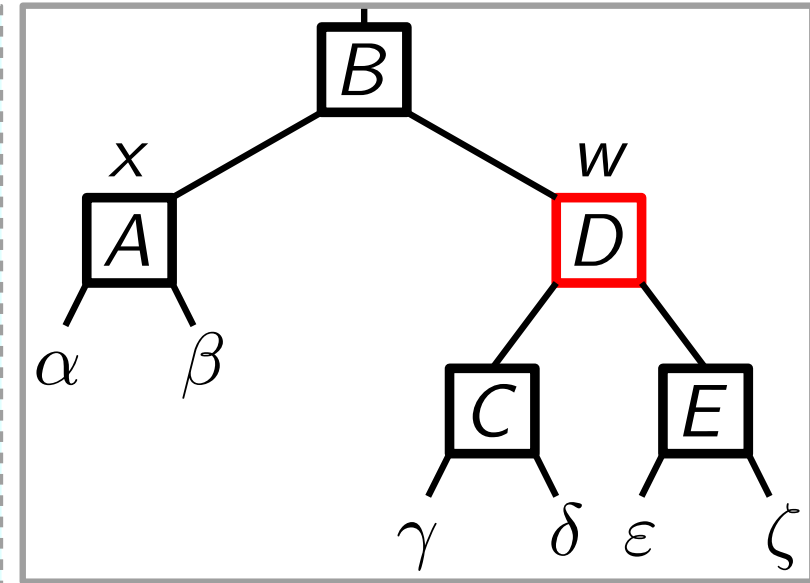
„Repariere“ Knoten x zählt eine schwarze Einheit extra
(E5): (ist also „rot-schwarz“ oder „doppelt schwarz“)

- Ziel:** Schiebe die überzählige schwarze Einheit nach oben, bis:
- x ist rot-schwarz \Rightarrow mach x schwarz.
 - x ist Wurzel \Rightarrow schwarze Extra-Einheit verfällt.
 - Problem wird lokal durch Umfärben & Rotieren gelöst.

RBDeleteFixup(RBNode x)

```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
     $w = x.p.\text{right}$  // Schwester von  $x$ 
    if  $w.\text{color} == \text{red}$  then
       $w.\text{color} = \text{black}$ 
       $x.p.\text{color} = \text{red}$ 
      LeftRotate( $x.p$ )
       $w = x.p.\text{right}$ 
    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 // kommt gleich!!
  else // wie oben; nur  $\text{left} \leftrightarrow \text{right}$ 
 $x.\text{color} = \text{black}$ 
  
```

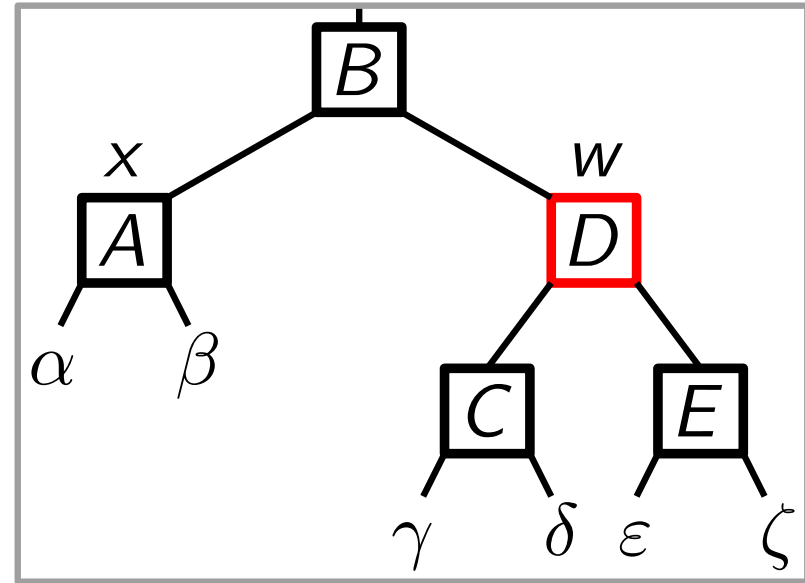


RBDeleteFixup(RBNode x)

```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
     $w = x.p.\text{right}$  // Schwester von x
    if  $w.\text{color} == \text{red}$  then
       $w.\text{color} = \text{black}$ 
       $x.p.\text{color} = \text{red}$ 
      LeftRotate( $x.p$ )
       $w = x.p.\text{right}$ 
      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 // kommt gleich!!
    else // wie oben; nur left  $\leftrightarrow$  right
   $x.\text{color} = \text{black}$ 
  
```

Ziel:
 $w \rightarrow$ schwarz
 ohne R-S-Eig.
 zu verletzen.



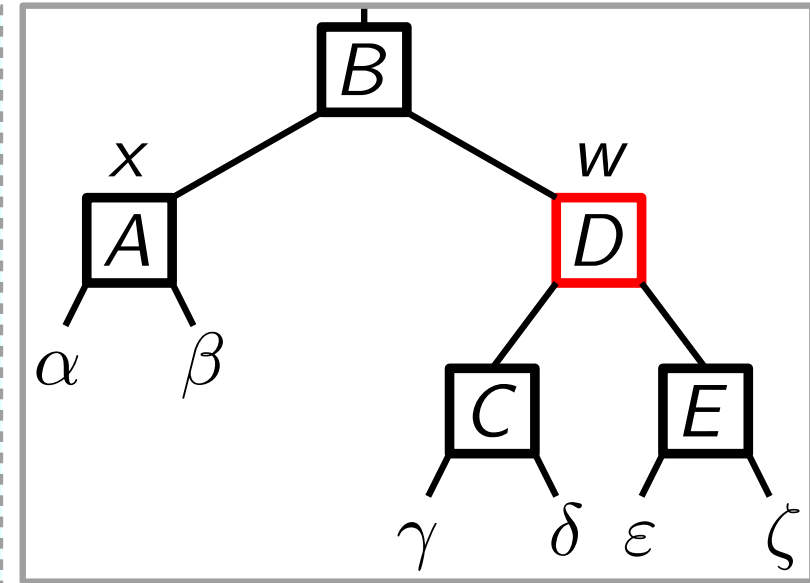
Fall 1

RBDeleteFixup(RBNode x)

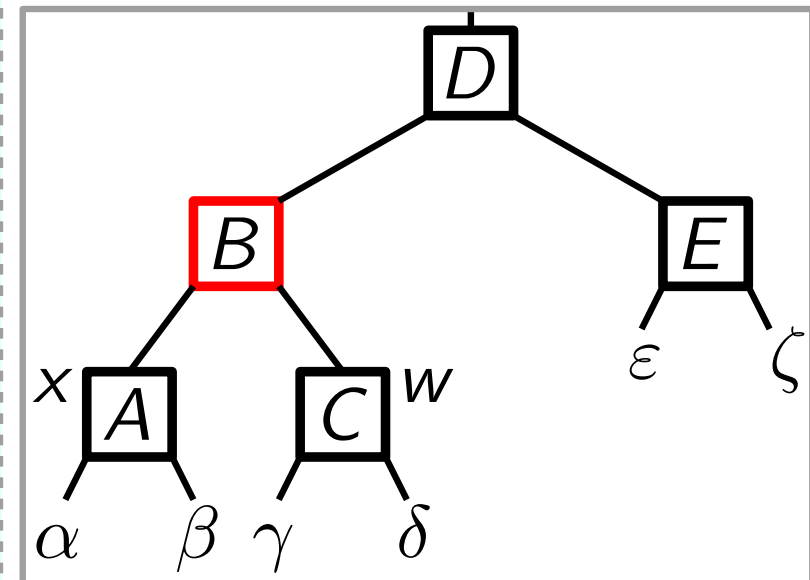
```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
     $w = x.p.\text{right}$  // Schwester von  $x$ 
    if  $w.\text{color} == \text{red}$  then
       $w.\text{color} = \text{black}$ 
       $x.p.\text{color} = \text{red}$ 
      LeftRotate( $x.p$ )
       $w = x.p.\text{right}$ 
      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 // kommt gleich!!
    else // wie oben; nur  $\text{left} \leftrightarrow \text{right}$ 
   $x.\text{color} = \text{black}$ 
  
```

Ziel:
 $w \rightarrow$ schwarz
 ohne R-S-Eig.
 zu verletzen.



↓ Fall 1



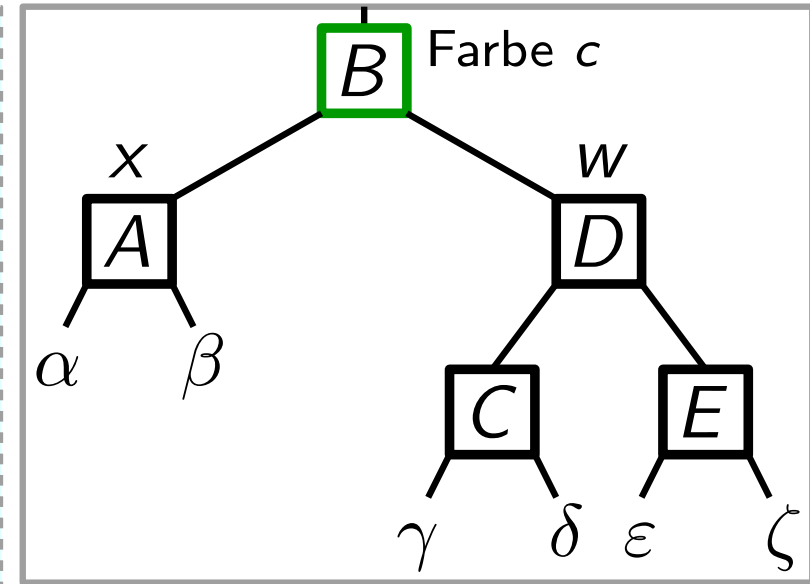
RBDeleteFixup(RBNode x)

```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
     $w = x.p.\text{right}$  // Schwester von x
    if  $w.\text{color} == \text{red}$  then
       $w.\text{color} = \text{black}$ 
       $x.p.\text{color} = \text{red}$ 
      LeftRotate( $x.p$ )
       $w = x.p.\text{right}$ 
      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 // kommt gleich!!
    else // wie oben; nur left  $\leftrightarrow$  right
   $x.\text{color} = \text{black}$ 
  
```

Ziel:
 $w \rightarrow$ schwarz
 ohne R-S-Eig.
 zu verletzen.

Schw. Einheit
 raufschieben.



Fall 2

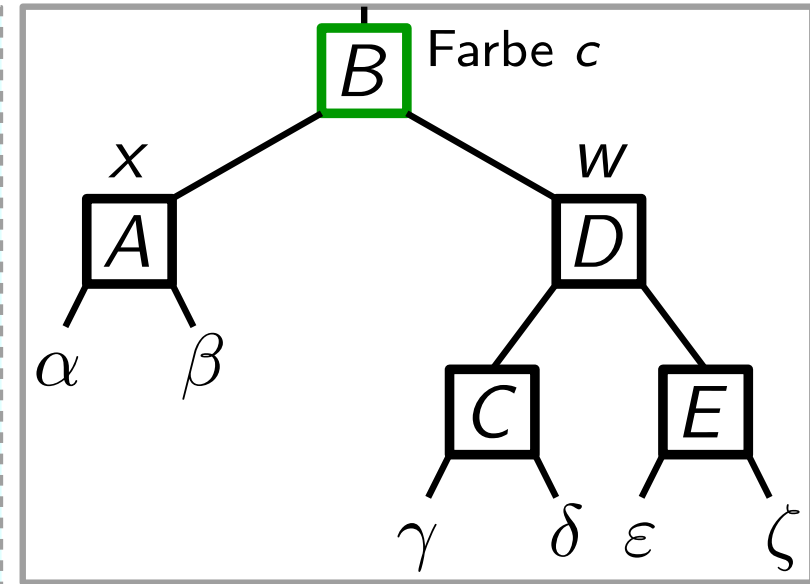
RBDeleteFixup(RBNode x)

```

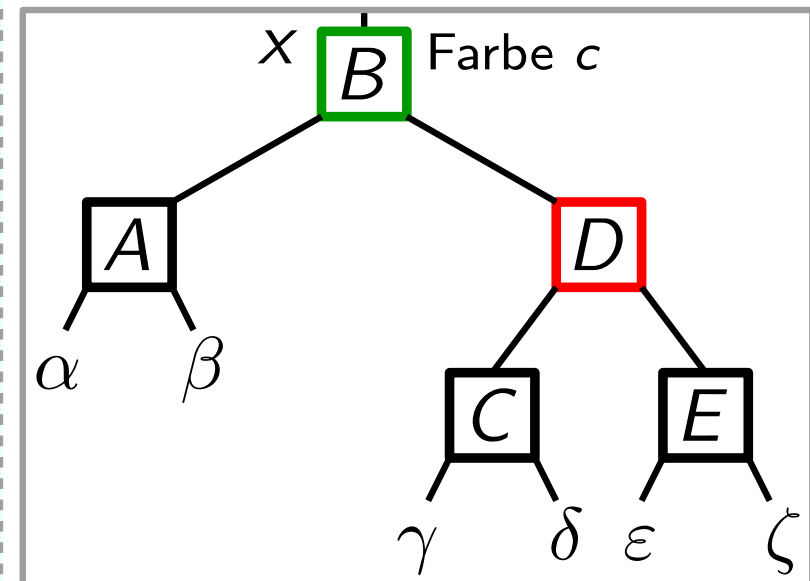
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
     $w = x.p.\text{right}$  // Schwester von x
    if  $w.\text{color} == \text{red}$  then
       $w.\text{color} = \text{black}$ 
       $x.p.\text{color} = \text{red}$ 
      LeftRotate( $x.p$ )
       $w = x.p.\text{right}$ 
      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 // kommt gleich!!
        else // wie oben; nur  $\text{left} \leftrightarrow \text{right}$ 
     $x.\text{color} = \text{black}$ 
  
```

Ziel:
 $w \rightarrow$ schwarz
 ohne R-S-Eig.
 zu verletzen.

Schw. Einheit
 raufschieben.



Fall 2



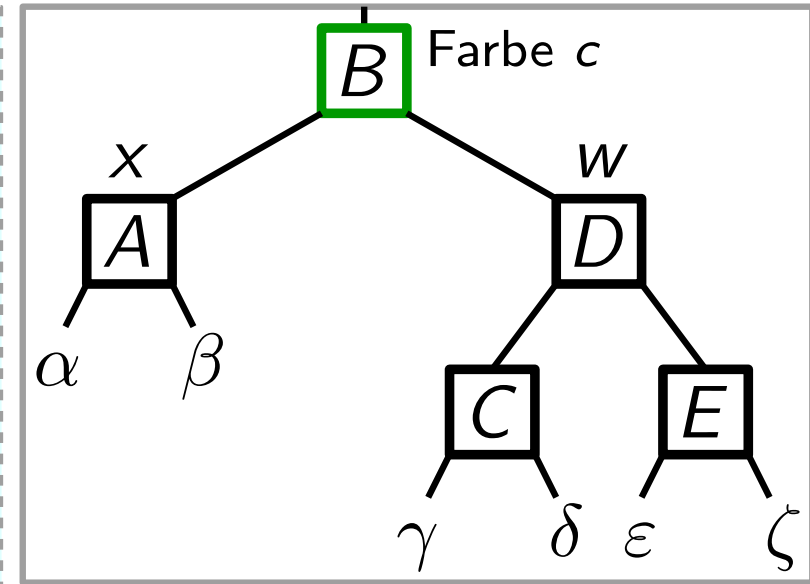
RBDeleteFixup(RBNode x)

```

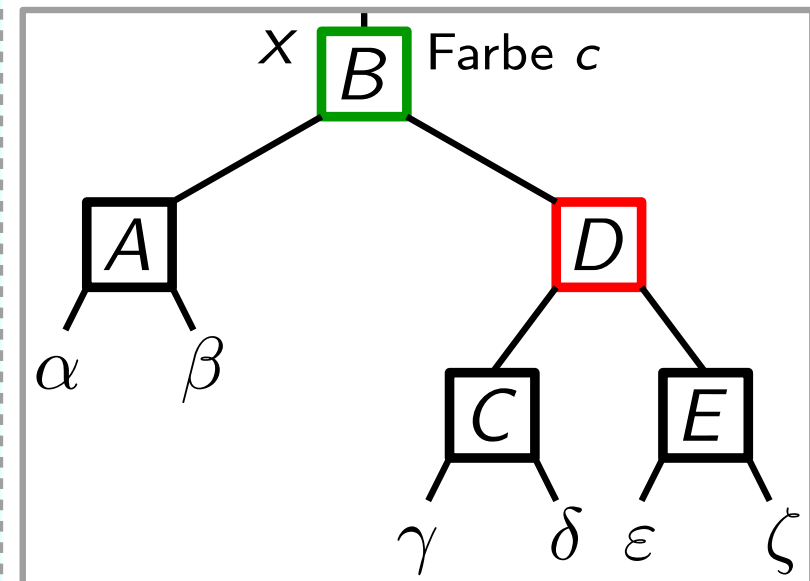
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
     $w = x.p.\text{right}$  // Schwester von x
    if  $w.\text{color} == \text{red}$  then
       $w.\text{color} = \text{black}$ 
       $x.p.\text{color} = \text{red}$ 
      LeftRotate( $x.p$ )
       $w = x.p.\text{right}$ 
      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 // kommt gleich!!
    else // wie oben; nur left  $\leftrightarrow$  right
   $x.\text{color} = \text{black}$ 
  
```

Ziel:
 $w \rightarrow$ schwarz
 ohne R-S-Eig.
 zu verletzen.

Schw. Einheit
 raufschieben.



Fall 2



Bem.: Anz. der schw. Knoten (inkl. Extra-Einh. bei x) bleibt auf allen Pfaden gleich!

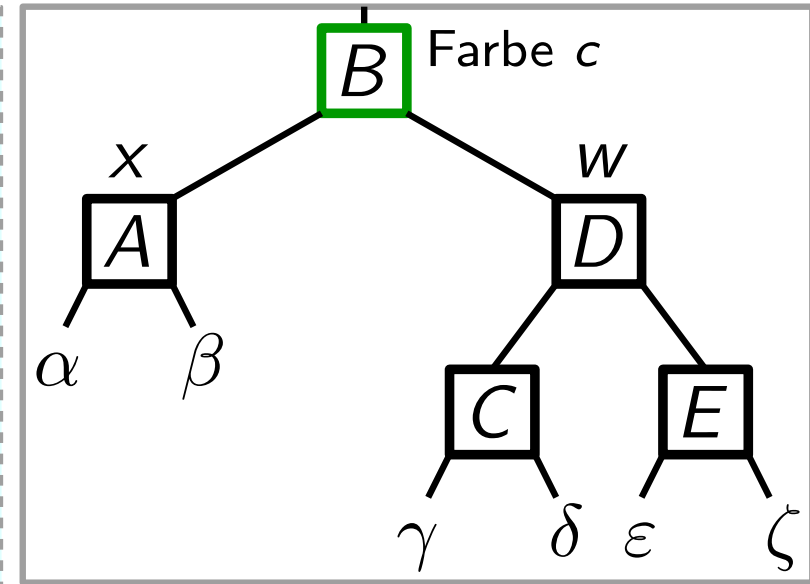
RBDeleteFixup(RBNode x)

```

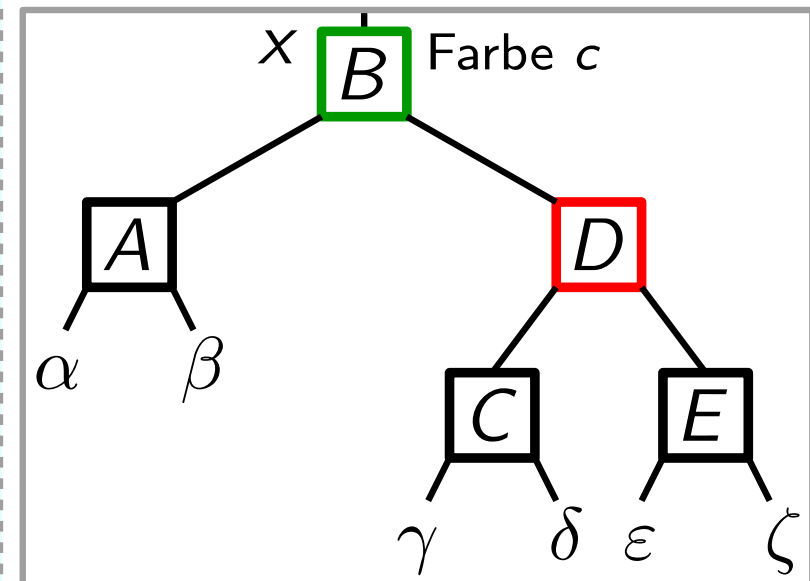
while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
  if  $x == x.p.\text{left}$  then
     $w = x.p.\text{right}$  // Schwester von x
    if  $w.\text{color} == \text{red}$  then
       $w.\text{color} = \text{black}$ 
       $x.p.\text{color} = \text{red}$ 
      LeftRotate( $x.p$ )
       $w = x.p.\text{right}$ 
      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 // kommt gleich!!
    else // wie oben; nur left  $\leftrightarrow$  right
   $x.\text{color} = \text{black}$ 
  
```

Ziel:
 $w \rightarrow$ schwarz
 ohne R-S-Eig.
 zu verletzen.

Schw. Einheit
 raufschieben.



↓ Fall 2



Bem.: Anz. der schw. Knoten (inkl. Extra-Einh. bei x) bleibt auf allen Pfaden gleich!

RBDeleteFixup (Forts.)

else

if $w.right.color == black$ **then**

$w.left.color = black$

$w.color = red$

RightRotate(w)

$w = x.p.right$

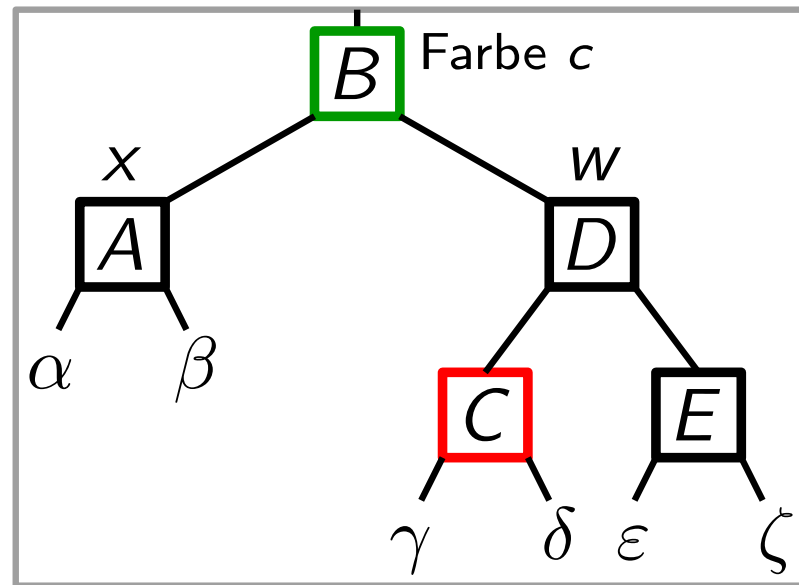
$w.color = x.p.color$

$x.p.color = black$

$w.right.color = black$

LeftRotate($x.p$)

$x = root$



Fall 3



RBDeleteFixup (Forts.)

else

if $w.right.color == black$ **then**

$w.left.color = black$

$w.color = red$

RightRotate(w)

$w = x.p.right$

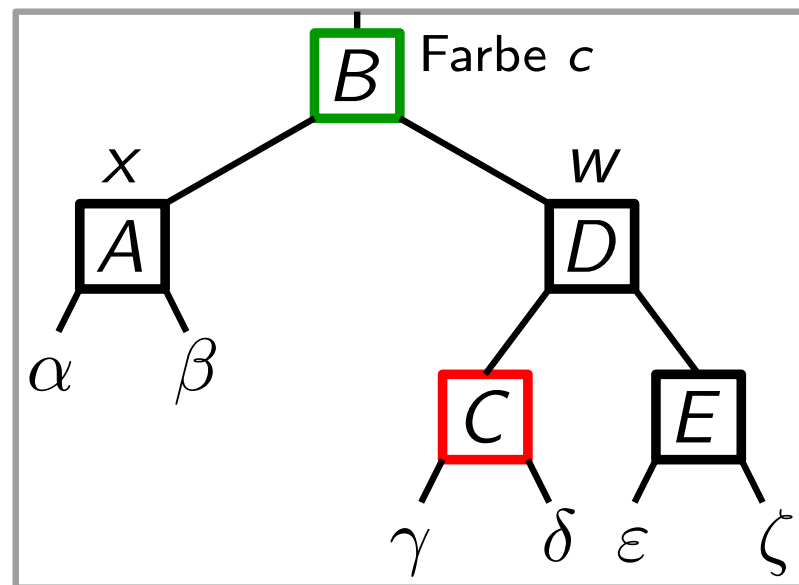
$w.color = x.p.color$

$x.p.color = black$

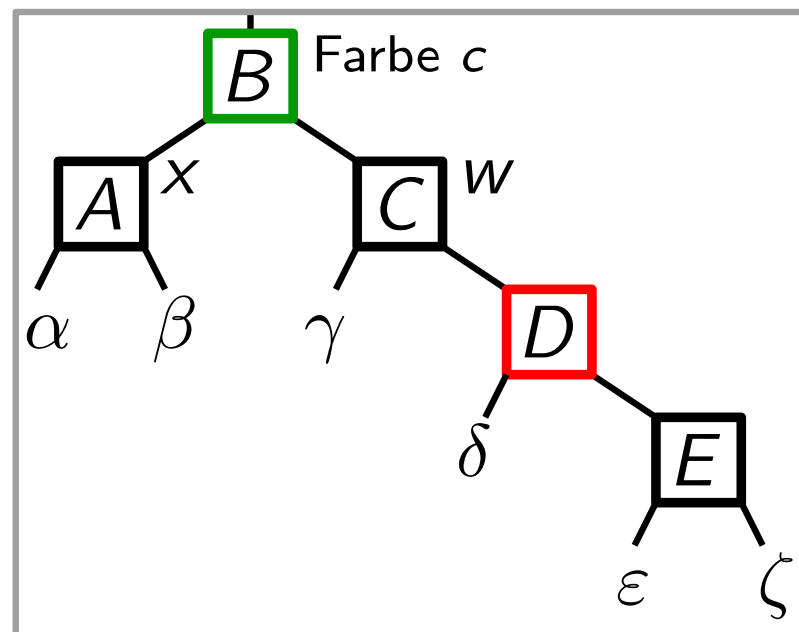
$w.right.color = black$

LeftRotate($x.p$)

$x = root$



Fall 3



RBDeleteFixup (Forts.)

else

if $w.right.color == black$ **then**

$w.left.color = black$

$w.color = red$

RightRotate(w)

$w = x.p.right$

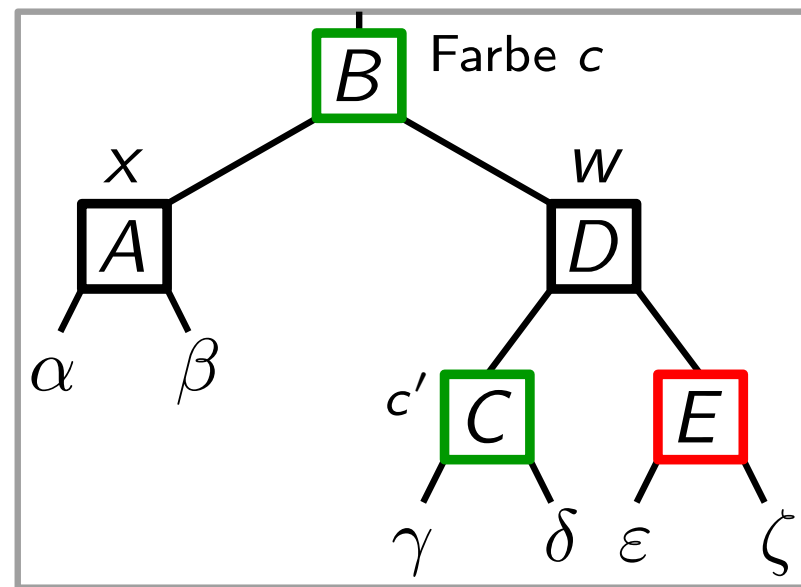
$w.color = x.p.color$

$x.p.color = black$

$w.right.color = black$

LeftRotate($x.p$)

$x = root$



Fall 4

RBDeleteFixup (Forts.)

else

if $w.right.color == black$ **then**

$w.left.color = black$

$w.color = red$

RightRotate(w)

$w = x.p.right$

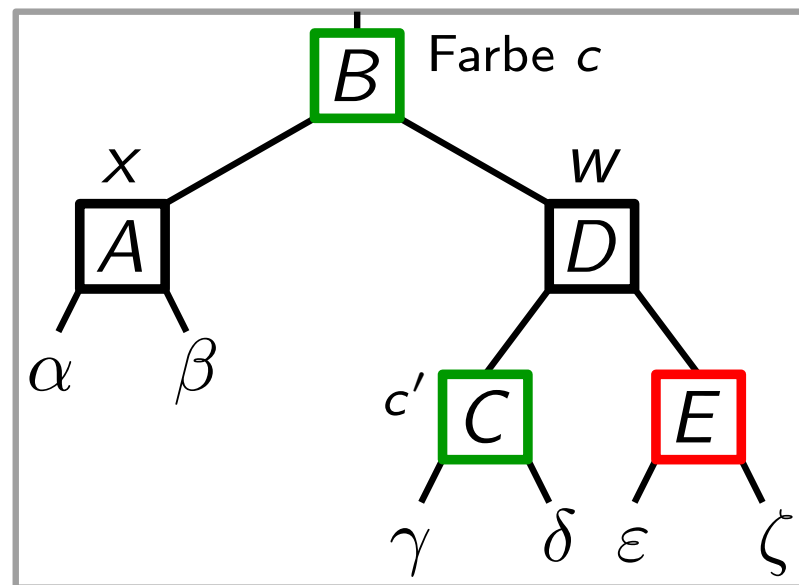
$w.color = x.p.color$

$x.p.color = black$

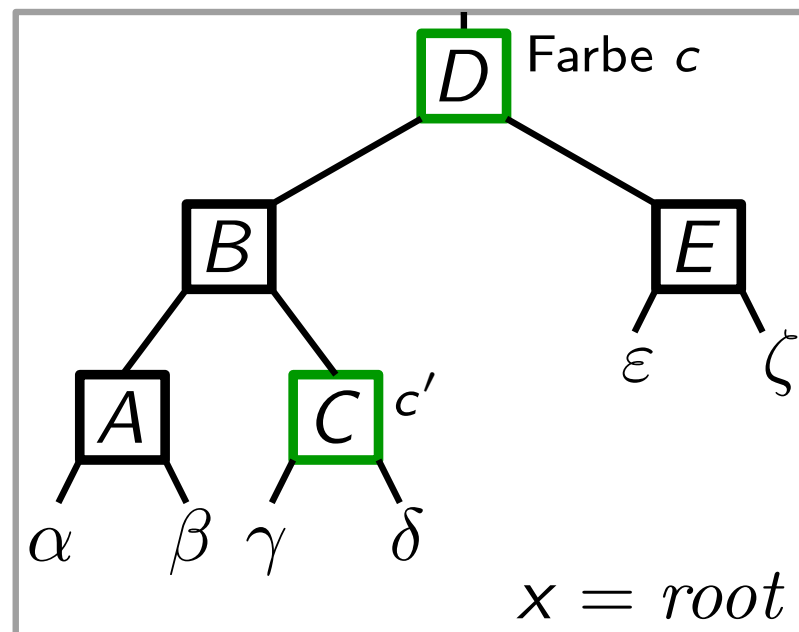
$w.right.color = black$

LeftRotate($x.p$)

$x = root$



Fall 4



RBDeleteFixup (Forts.)

else

if $w.\text{right}.color == \text{black}$ **then**

$w.\text{left}.color = \text{black}$

$w.\text{color} = \text{red}$

RightRotate(w)

$w = x.p.\text{right}$

$w.\text{color} = x.p.\text{color}$

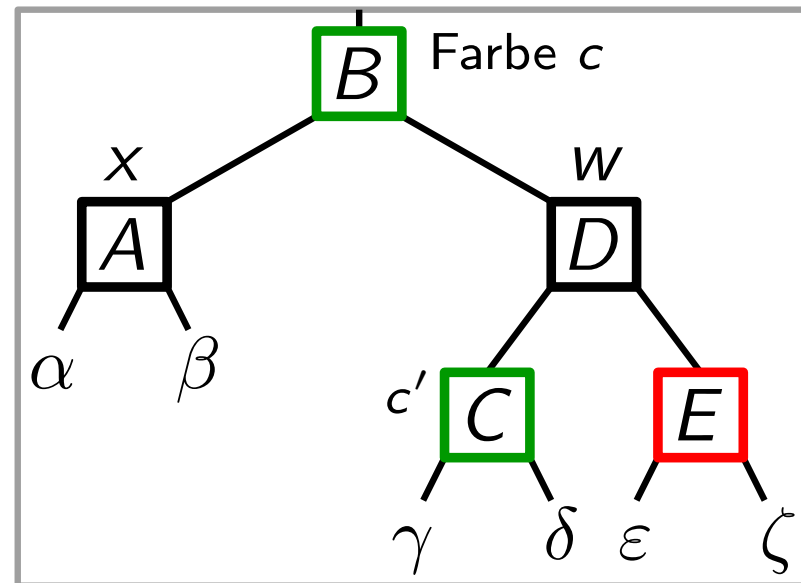
$x.p.\text{color} = \text{black}$

$w.\text{right}.color = \text{black}$

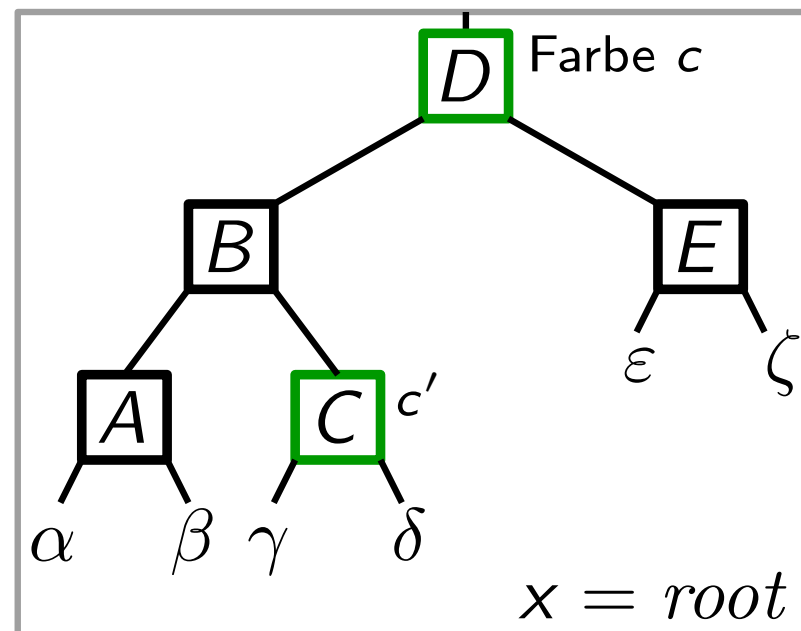
LeftRotate($x.p$)

$x = \text{root}$

Bem.: Anz. der schwarzen Knoten
(inkl. der Extra-Einheit bei x)
bleibt auf allen Pfaden gleich!



Fall 4



RBDeleteFixup (Forts.)

else

if $w.\text{right}.color == \text{black}$ **then**

$w.\text{left}.color = \text{black}$

$w.\text{color} = \text{red}$

$\text{RightRotate}(w)$

$w = x.p.\text{right}$

$w.\text{color} = x.p.\text{color}$

$x.p.\text{color} = \text{black}$

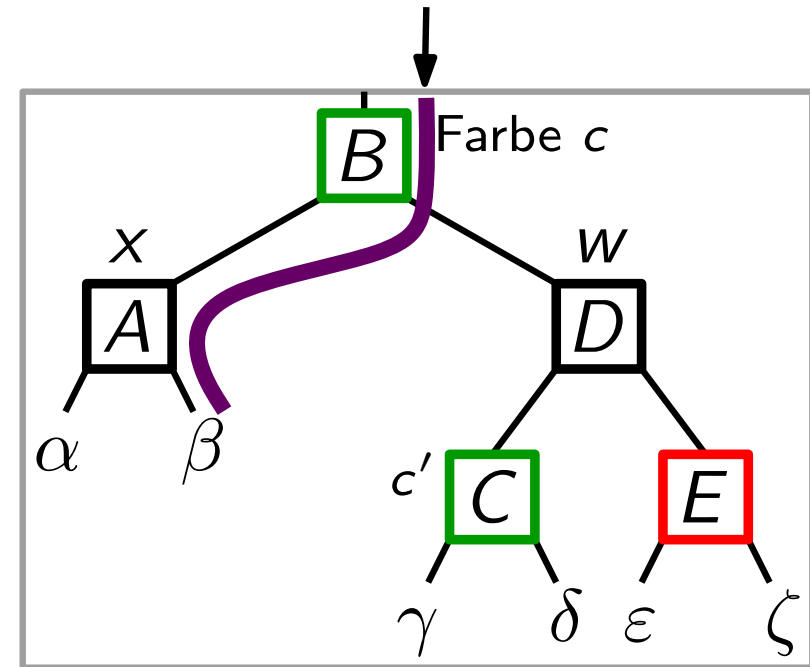
$w.\text{right}.color = \text{black}$

$\text{LeftRotate}(x.p)$

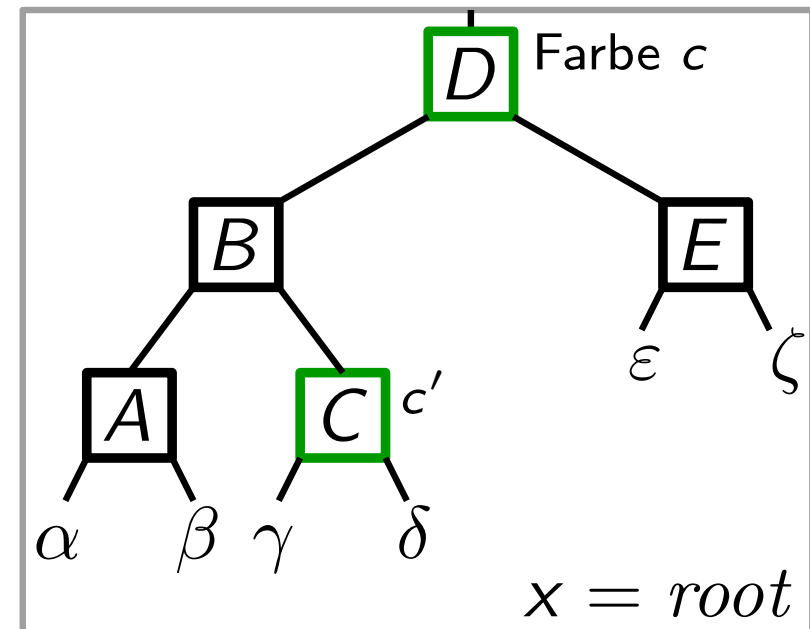
$x = \text{root}$

Bem.: Anz. der schwarzen Knoten
(inkl. der Extra-Einheit bei x)
bleibt auf allen Pfaden gleich!

vorher:
schwarze Einheiten = $c + 2$



Fall 4



RBDeleteFixup (Forts.)

else

if $w.\text{right}.color == \text{black}$ **then**

$w.\text{left}.color = \text{black}$

$w.\text{color} = \text{red}$

$\text{RightRotate}(w)$

$w = x.p.\text{right}$

$w.\text{color} = x.p.\text{color}$

$x.p.\text{color} = \text{black}$

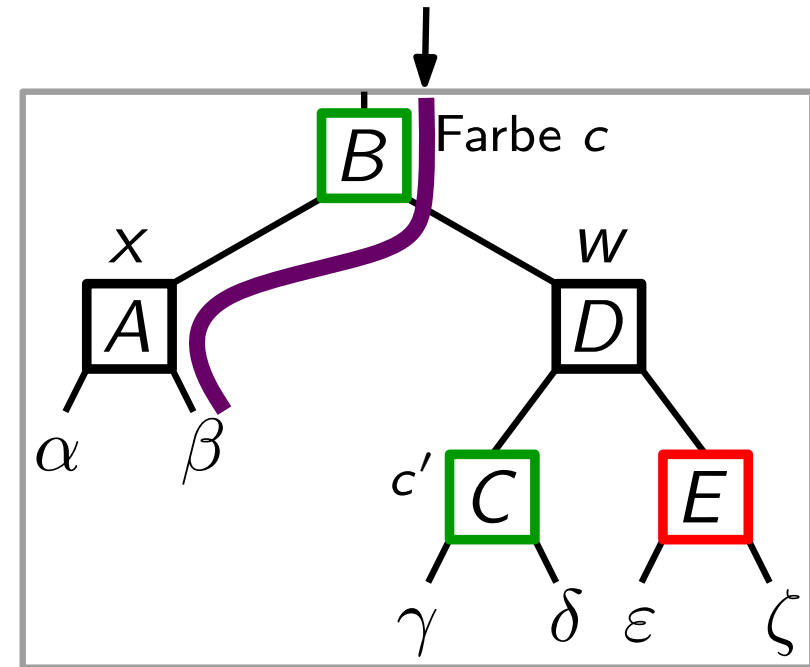
$w.\text{right}.color = \text{black}$

$\text{LeftRotate}(x.p)$

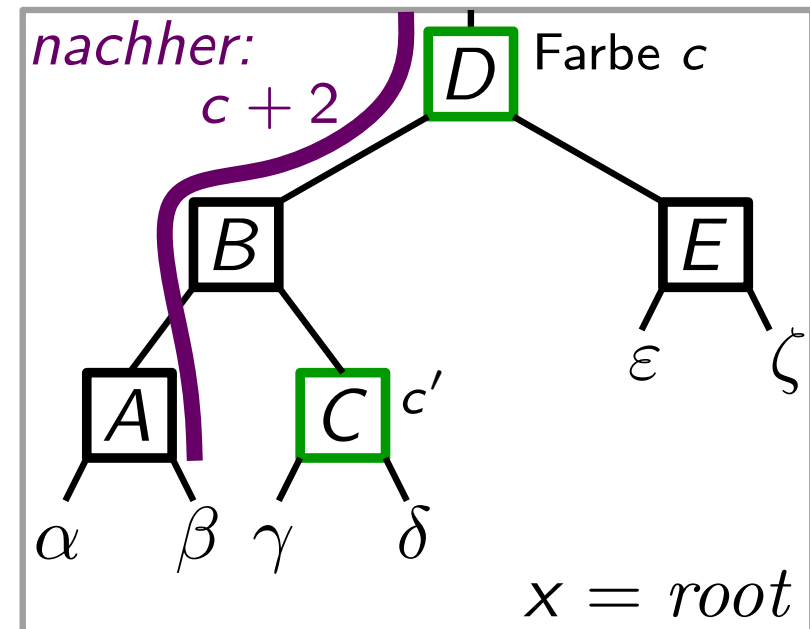
$x = \text{root}$

Bem.: Anz. der schwarzen Knoten
(inkl. der Extra-Einheit bei x)
bleibt auf allen Pfaden gleich!

vorher:
schwarze Einheiten = $c + 2$



Fall 4



RBDeleteFixup (Forts.)

else

if $w.\text{right}.color == \text{black}$ **then**

$w.\text{left}.color = \text{black}$

$w.\text{color} = \text{red}$

RightRotate(w)

$w = x.p.\text{right}$

$w.\text{color} = x.p.\text{color}$

$x.p.\text{color} = \text{black}$

$w.\text{right}.color = \text{black}$

LeftRotate($x.p$)

$x = \text{root}$

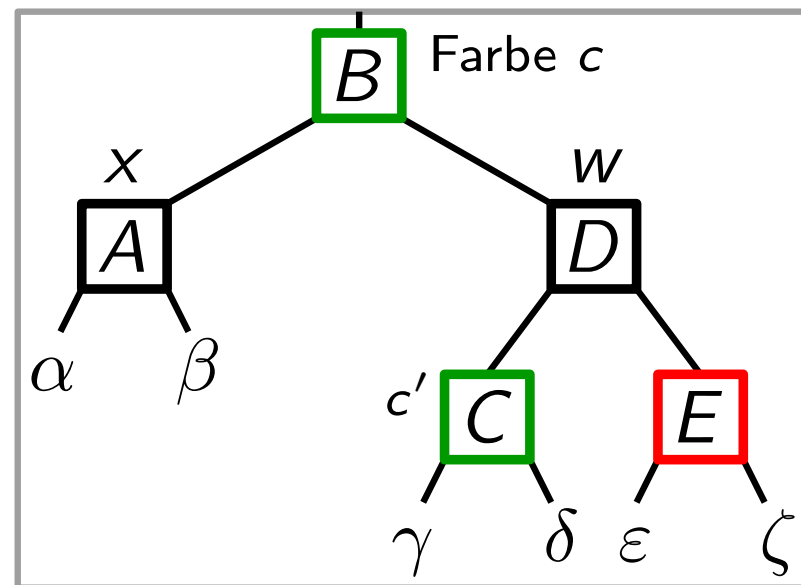
Laufzeit?

Fall 1:

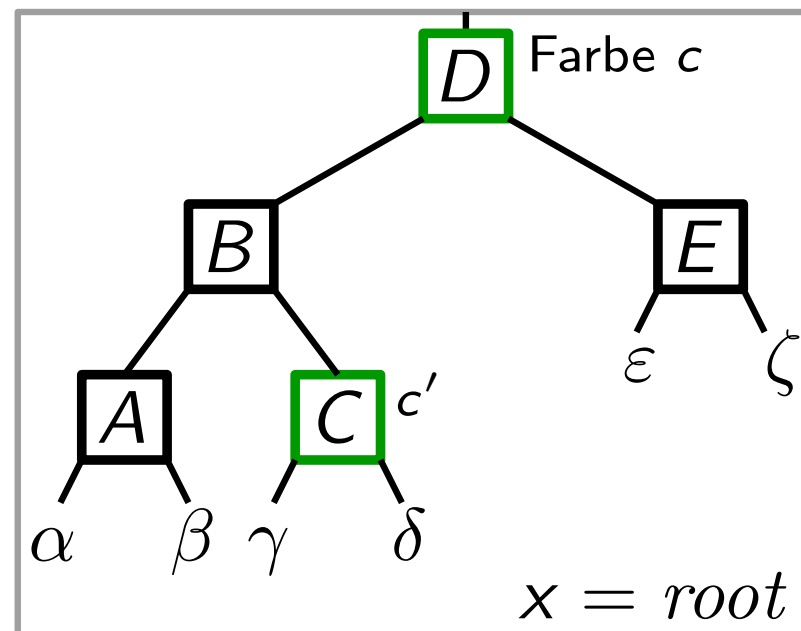
Fall 2:

Fall 3:

Fall 4:



Fall 4



RBDeleteFixup (Forts.)

else

if $w.right.color == black$ **then**

$w.left.color = black$

$w.color = red$

RightRotate(w)

$w = x.p.right$

$w.color = x.p.color$

$x.p.color = black$

$w.right.color = black$

LeftRotate($x.p$)

$x = root$

Laufzeit?

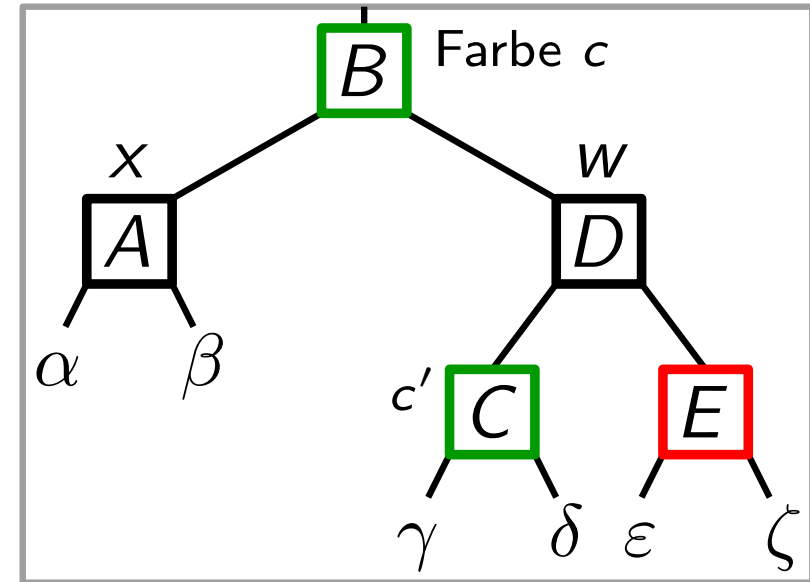
Fall 1:

Fall 2:

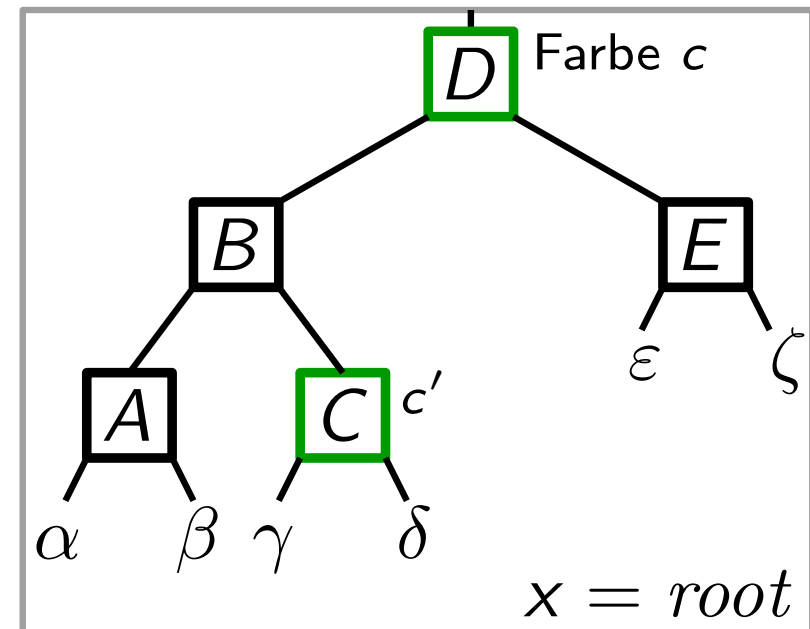
Fall 3:

Fall 4:

$O(h)$ Umfärbungen



Fall 4



RBDeleteFixup (Forts.)

else

if $w.right.color == black$ **then**

$w.left.color = black$

$w.color = red$

RightRotate(w)

$w = x.p.right$

$w.color = x.p.color$

$x.p.color = black$

$w.right.color = black$

LeftRotate($x.p$)

$x = root$

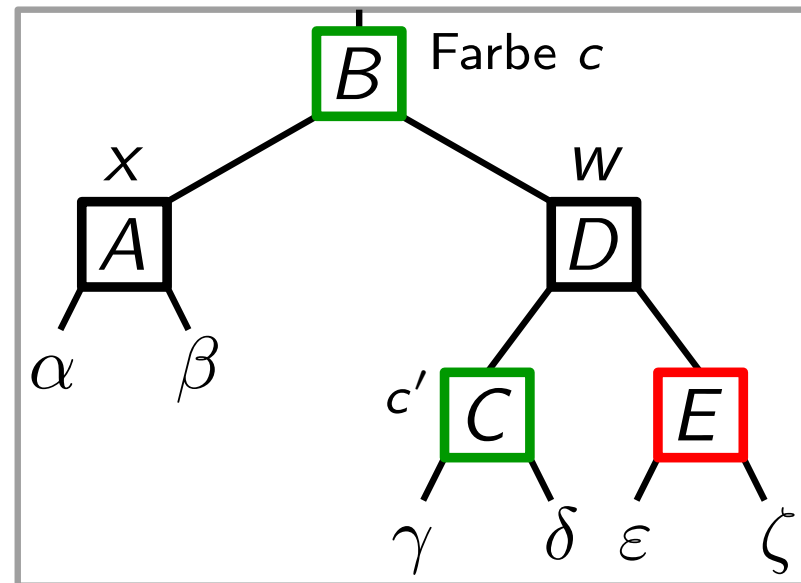
Laufzeit?

Fall 1: $O(h)$ Umfärbungen

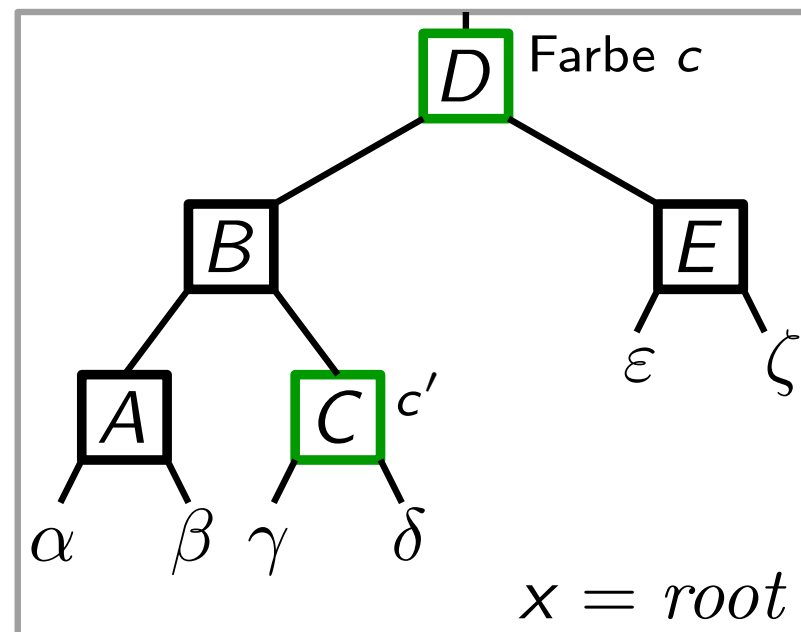
Fall 2: 1 Rotation + $O(1)$

Fall 3:

Fall 4:



Fall 4



RBDeleteFixup (Forts.)

else

if $w.right.color == black$ **then**

$w.left.color = black$

$w.color = red$

RightRotate(w)

$w = x.p.right$

$w.color = x.p.color$

$x.p.color = black$

$w.right.color = black$

LeftRotate($x.p$)

$x = root$

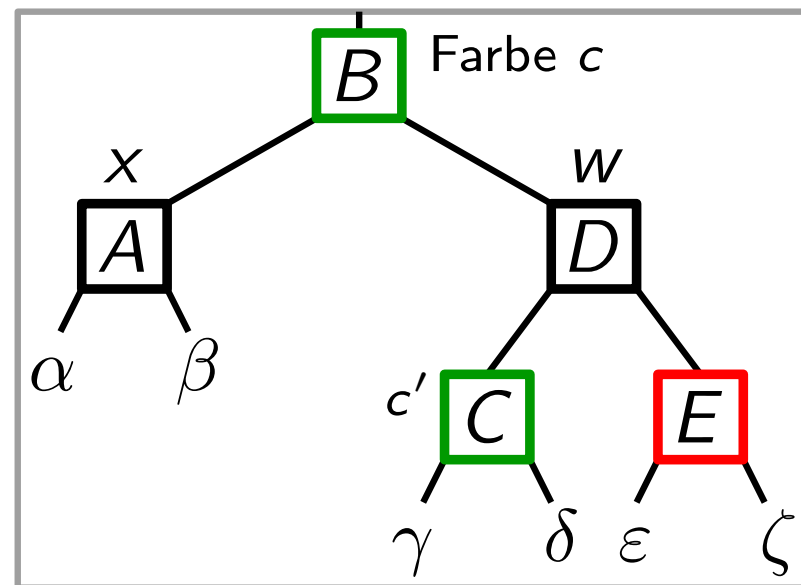
Laufzeit?

Fall 1: $O(h)$ Umfärbungen

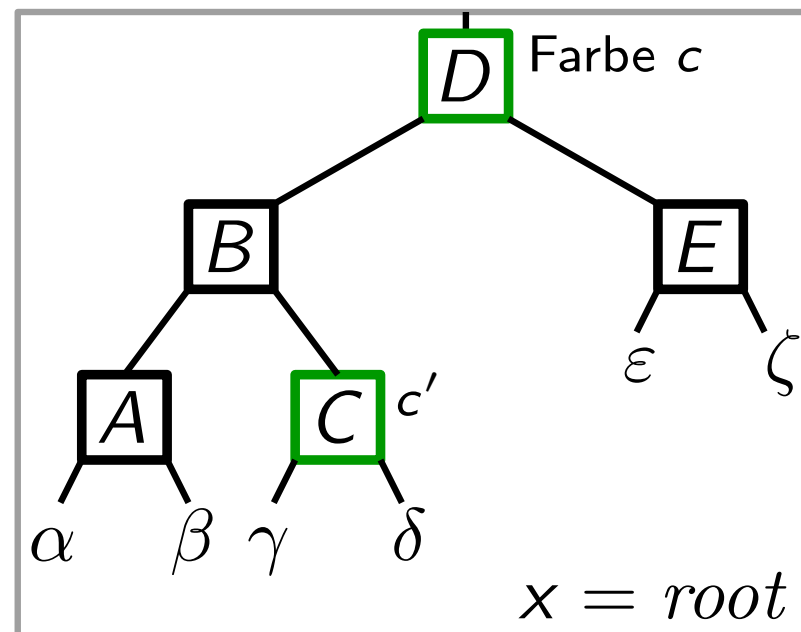
Fall 2: 1 Rotation + $O(1)$

Fall 3: 1 Rotation + $O(1)$

Fall 4:



Fall 4



RBDeleteFixup (Forts.)

else

if $w.right.color == black$ **then**

$w.left.color = black$

$w.color = red$

RightRotate(w)

$w = x.p.right$

$w.color = x.p.color$

$x.p.color = black$

$w.right.color = black$

LeftRotate($x.p$)

$x = root$

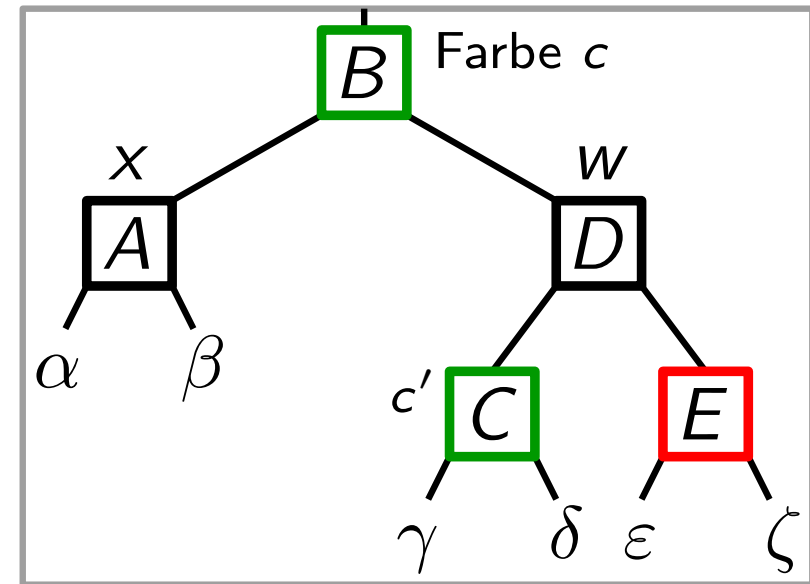
Laufzeit?

Fall 1: $O(h)$ Umfärbungen

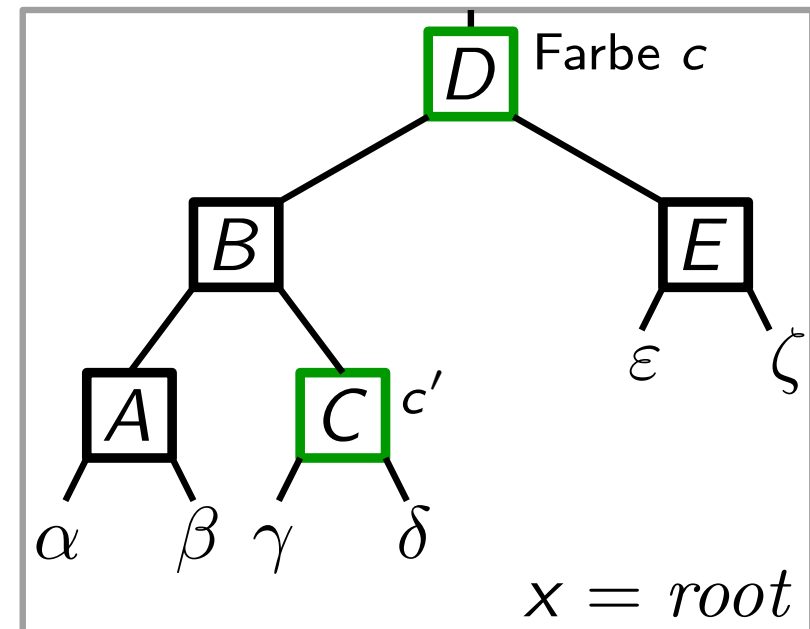
Fall 2: 1 Rotation + $O(1)$

Fall 3: 1 Rotation + $O(1)$

Fall 4: 1 Rotation + $O(1)$



Fall 4



Zusammenfassung

Laufzeit RBDelete \in

Zusammenfassung

Laufzeit RBDelete $\in O(h)$ + Laufzeit RBDeleteFixup

Zusammenfassung

$$\text{Laufzeit RBDelete} \in O(h) + \underbrace{\text{Laufzeit RBDeleteFixup}}_{O(h)}$$

Zusammenfassung

$$\text{Laufzeit RBDelete} \in O(h) + \underbrace{\text{Laufzeit RBDeleteFixup}}_{O(h)} = O(h)$$

Zusammenfassung

$$\text{Laufzeit RBDelete} \in O(h) + \underbrace{\text{Laufzeit RBDeleteFixup}}_{O(h)} = O(h)$$

RBDelete erhält die Rot-Schwarz-Eigenschaften.

Zusammenfassung

$$\text{Laufzeit RBDelete} \in O(h) + \underbrace{\text{Laufzeit RBDeleteFixup}}_{O(h)} = O(h)$$

RBDelete erhält die Rot-Schwarz-Eigenschaften.

Also gilt (siehe Lemma): $h \in O(\log n)$

Zusammenfassung

$$\text{Laufzeit RBDelete} \in O(h) + \underbrace{\text{Laufzeit RBDeleteFixup}}_{O(h)} = O(h)$$

RBDelete erhält die Rot-Schwarz-Eigenschaften.

Also gilt (siehe Lemma): $h \in O(\log n)$



Zusammenfassung

$$\text{Laufzeit RBDelete} \in O(h) + \underbrace{\text{Laufzeit RBDeleteFixup}}_{O(h)} = O(h)$$

RBDelete erhält die Rot-Schwarz-Eigenschaften.

Also gilt (siehe Lemma): $h \in O(\log n)$



$$\text{Laufzeit RBDelete} \in O(\log n)$$

Zusammenfassung

$$\text{Laufzeit RBDelete} \in O(h) + \underbrace{\text{Laufzeit RBDeleteFixup}}_{O(h)} = O(h)$$

RBDelete erhält die Rot-Schwarz-Eigenschaften.

Also gilt (siehe Lemma): $h \in O(\log n)$



$$\text{Laufzeit RBDelete} \in O(\log n)$$

Satz.

Rot-Schwarz-Bäume implementieren alle dynamische-Menge-Operationen in $O(\log n)$ Zeit, wobei n die momentane Anz. der Schlüssel ist.

Zusammenfassung

$$\text{Laufzeit RBDelete} \in O(h) + \underbrace{\text{Laufzeit RBDeleteFixup}}_{O(h)} = O(h)$$

RBDelete erhält die Rot-Schwarz-Eigenschaften.

Also gilt (siehe Lemma): $h \in O(\log n)$



$$\text{Laufzeit RBDelete} \in O(\log n)$$

Satz.

Rot-Schwarz-Bäume implementieren alle dynamische-Menge-Operationen in $O(\log n)$ Zeit, wobei n die momentane Anz. der Schlüssel ist.