

# Bachelorarbeit

Andreas Windorfer

30. Mai 2020

## Zusammenfassung

# Inhaltsverzeichnis

<b>1 Fazit</b>	<b>4</b>
<b>2 Einleitung</b>	<b>5</b>
<b>3 Binäre Suchbäume</b>	<b>6</b>
3.1 Definition binärer Suchbaum . . . . .	6
3.2 Weitere Begriffe und Eigenschaften zum binären Suchbaum . . .	8
<b>4 Rot-Schwarz-Baum</b>	<b>16</b>
4.1 Grundoperationen . . . . .	18
4.2 Tango-Baum konformes vereinigen . . . . .	28
4.3 Tango-Baum konformes aufteilen . . . . .	32

# 1 Fazit

## 2 Einleitung

### 3 Binäre Suchbäume

Es gibt viele Varianten von binären Suchbäumen mit unterschiedlichen Eigenschaften und Leistungsdaten. In diesem Kapitel werden binäre Suchbäume im Allgemeinen beschrieben. Außerdem werden Begriffe definiert, die in den nachfolgenden Kapiteln verwendet werden.

#### 3.1 Definition binärer Suchbaum

Ein **Baum**  $T$  ist ein zusammenhängender, gerichteter Graph, der keine Zyklen enthält. Ein Baum ohne Knoten ist ein **leerer Baum**. In einem nicht leerem Baum gibt es genau einen Knoten ohne eingehende Kante, diesen bezeichnet man als **Wurzel**. Alle anderen Knoten haben genau eine eingehende Kante. Jeder Knoten  $v$  in  $T$  ist Wurzel eines **Teilbaumes**  $T(v)$ . Knoten ohne ausgehende Kante nennt man **Blatt**, alle anderen Knoten werden als **innere Knoten** bezeichnet. Enthält der Baum eine Kante von Knoten  $v_1$  zu Knoten  $v_2$  so nennt man  $v_2$  ein **Kind** von  $v_1$  und  $v_1$  bezeichnet man als den **Vater** von  $v_2$ . Die Wurzel hat also keinen Vater, alle anderen Knoten genau einen. Bei einem **binären Baum** kommt folgende Einschränkung hinzu:

*Ein Knoten hat maximal zwei Kinder.*

Entsprechend ihrer Zeichnung benennt man die Kinder in Binärbäumen als **linkes Kind** oder **rechtes Kind**. Sei  $w$  das linke bzw. rechte Kind von  $v$ , dann bezeichnet man den Teilbaum mit Wurzel  $w$  als **linken Teilbaum** bzw. **rechten Teilbaum** von  $v$ .

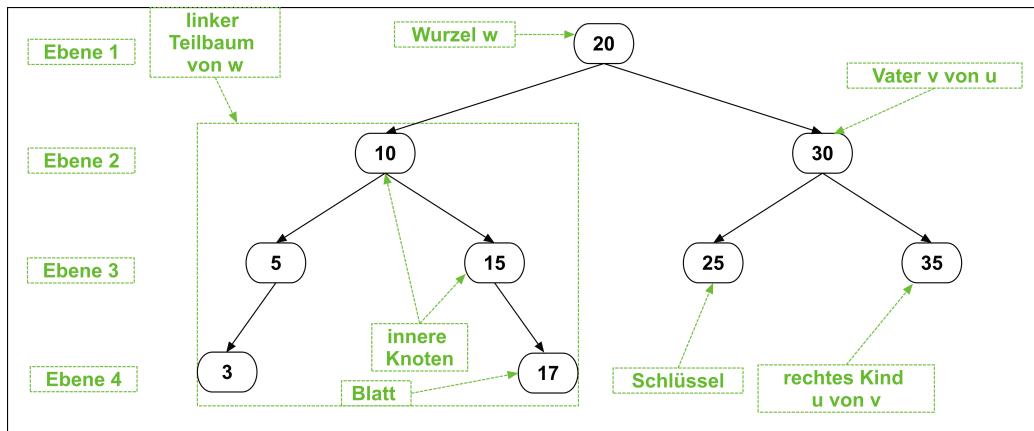
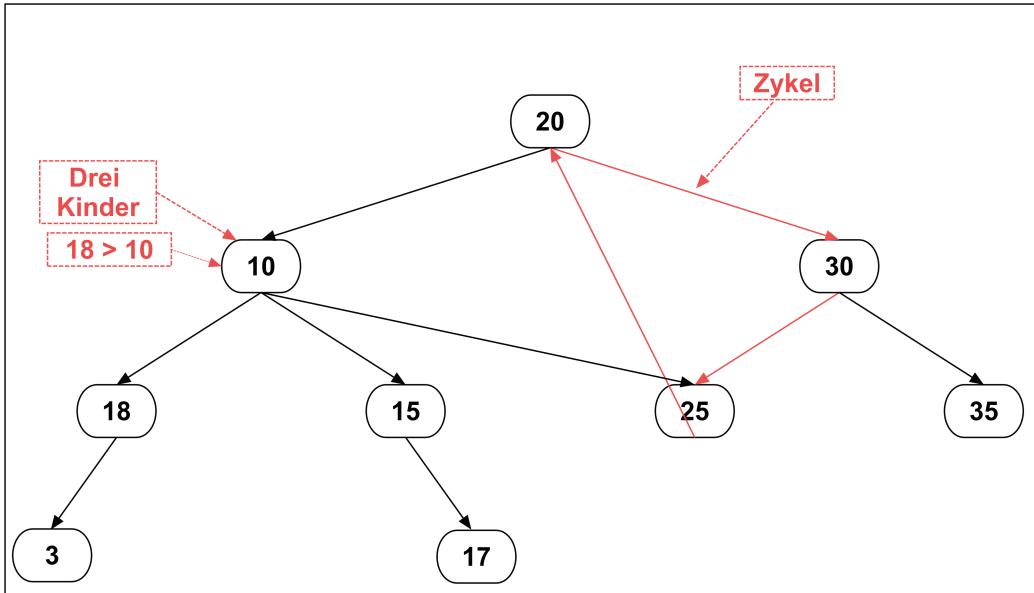


Abbildung 1: Ein binärer Suchbaum

Bei einem **binären Suchbaum** ist jedem Knoten ein innerhalb der Baumstruktur ein eindeutiger **Schlüssel** aus einem **Universum** zugeordnet. Als



**Abbildung 2:** Kein binärer Suchbaum

Universum kann jede Menge  $M$  verwendet werden, auf der eine totale Ordnung definiert ist. Auf totale Ordnungen wird in diesen Kapitel noch eingegangen. Hier und in den folgenden Kapiteln wird als Universum immer  $\mathbb{N}$  verwendet, wobei die 0 enthalten ist. Die in einem binären Suchbaum enthaltenen Schlüssel bezeichnen wir als seine **Schlüsselmenge**. Damit aus dem binären Baum ein binärer Suchbaum wird, benötigt man noch folgende Eigenschaft:

*Für jeden Knoten im binären Suchbaum gilt, dass alle in seinem linken Teilbaum enthaltenen Schlüssel kleiner sind als der eigene Schlüssel. Alle im rechten Teilbaum enthaltenen Schlüssel sind größer als der eigene Schlüssel.*

Es gibt eine rekursive Definition für binäre Suchbäume, aus der die gerade geforderten Eigenschaften direkt ersichtlich sind. Diese soll auch hier verwendet werden.

### **Definition 3.1. Binärer Suchbaum**

1. Der leere Baum ohne Knoten ist ein binärer Suchbaum.
2. Der Baum mit dem einzigen Knoten  $v$  der Schlüssel  $k_v$  enthält ist ein binärer Suchbaum.

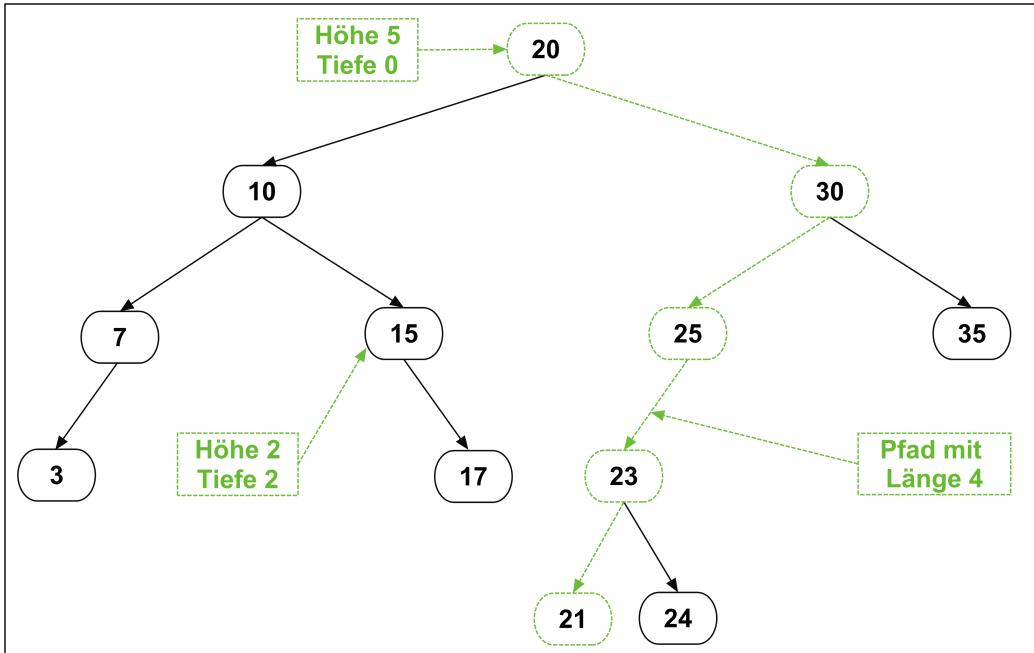
3. Es seien  $T_1$  und  $T_2$  binäre Suchbäume mit Schlüsselmenge  $K_1$  bzw.  $K_2$ . Sei  $i \in \mathbb{N}$ , mit  $\max(K_1) < i < \min(K_2)$ . Erzeuge einen neuen Knoten  $w$  mit Schlüssel  $i$ . Setze  $T_1$  als linken Teilbaum von  $w$  und  $T_2$  als rechten Teilbaum von  $w$ . Die so entstandenen Struktur ist ein binärer Suchbaum mit Wurzel  $w$ .
4. Eine Struktur die sich nicht durch Anwenden von Punkt 1, 2 und 3 erzeugen lässt, ist kein binärer Suchbaum.

Anstatt binärer Suchbaum schreibt man häufig **BST** für Binary Search Tree. Diese Abkürzung wird hier ab jetzt auch verwendet. In Implementierungen enthält jeder Knoten für das linke und rechte Kind jeweils einen Zeiger. Anstatt von entfernten oder hinzugefügten Kanten wird im folgenden häufig von umgesetzten Zeigern gesprochen.

### 3.2 Weitere Begriffe und Eigenschaften zum binären Suchbaum

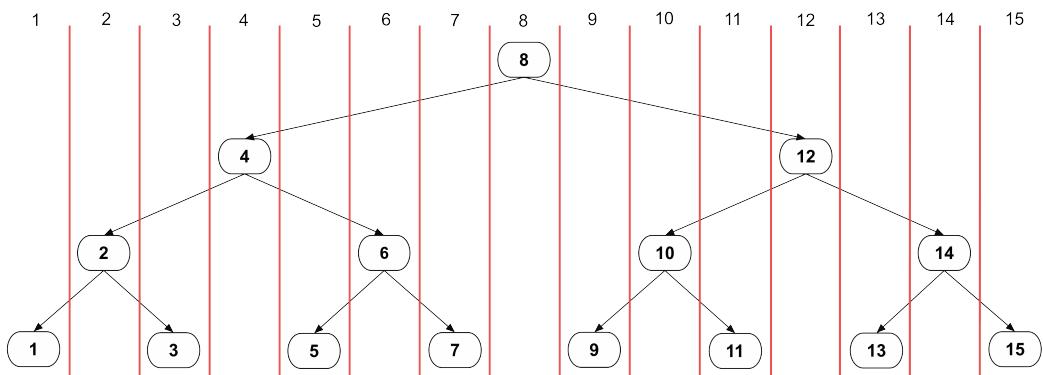
Zwei verschiedene Knoten mit dem selben Vater nennt man **Brüder**. Ein **Pfad**  $P_{jk}$  ist eine Folge von Knoten  $(v_0, v_1, \dots, v_n)$ , mit  $v_0 = v_j$ ,  $v_n = v_k$  und  $\forall i \in \{1, 2, \dots, n\}: v_{i-1} \text{ ist Vater von } v_i$ .  $n$  ist die **Länge des Pfades**. Die Knoten  $v_0$  bis  $v_{n-1}$  sind **Vorfahren** von  $v_n$ . Den Knoten in einem BST wird auch eine **Tiefe** und eine **Höhe** zugeteilt. Für einen Knoten  $v$  gilt, dass die Länge des Pfades von der Wurzel zu ihm seiner Tiefe entspricht. Sei  $l$  die maximale Länge eines von  $v$  aus startenden Pfades. Die Höhe  $h(v)$  von  $v$  ist dann  $l + 1$ . Die Höhe der Wurzel entspricht der **Höhe des Baumes**  $h(T)$ , wobei ein leerer Baum Höhe 0 hat. Einen BST  $T$  mit Höhe  $h_T$  unterteilt man von oben nach unten in die **Ebenen**  $1, 2, \dots, h_T$ . Die Wurzel liegt in der Ebene eins, deren Kinder in der Ebene zwei usw. Enthält eine Ebene ihre maximale Anzahl an Knoten nennt man sie **vollständig besetzt**.

Da im linken Teilbaum nur kleinere Schlüssel vorhanden sein dürfen und im rechten Teilbaum nur größere, kann man die Schlüsselmenge eines binären Suchbaumes, von links nach rechts, in aufsteigend sortierter Form ablesen. Denn angenommen es gibt zwei Knoten  $v_l, v_r$  mit den Schlüsseln  $k_l$  bzw.  $k_r$ , so dass  $k_l > k_r$  gilt und  $v_l$  liegt weiter links im Baum als  $v_r$ . Ist ein  $v_l$  Vorfahre von  $v_r$ , so enthält der rechte Teilbaum von  $v_l$  einen Schlüssel der kleiner ist als  $k_l$ . Ist  $v_r$  Vorfahre von  $v_l$ , so enthält der linke Teilbaum von  $v_r$  einen Schlüssel der größer ist als  $k_r$ . Ist keiner der Knoten Vorfahre des anderen, muss es zumindest einen gemeinsamen Vorfahren geben, denn dann kann weder  $v_r$  noch  $v_l$  die Wurzel des BST sein. Sei  $v_v$  der gemeinsame Vorfahre mit der größten Tiefe. Der linke Teilbaum von  $v_v$  enthält dann einen größeren Schlüssel, als



**Abbildung 3:** Ein weiterer binärer Suchbaum

der rechte Teilbaum dieses Knotens. In jedem Fall erhält man einen Widerspruch zu der von BSTs geforderten Eigenschaft. Aus Platzgründen passiert es bei Zeichnungen von BSTs manchmal, dass ein Knoten in einem linken Teilbaum weiter rechts steht als die Wurzel des Teilbaumes, oder umgekehrt, weshalb man bei der Betrachtung solcher Zeichnungen etwas vorsichtig sein muss. Abbildung 4 enthält keine solche Konstellation.



**Abbildung 4:** Schlüssel sind aufsteigend sortiert ablesbar.

Algorithmisch kann man sich die im BST enthaltenen Schlüssel aufsteigend sortiert durch eine **Inorder-Traversierung** ausgeben lassen. Es ist ein re-

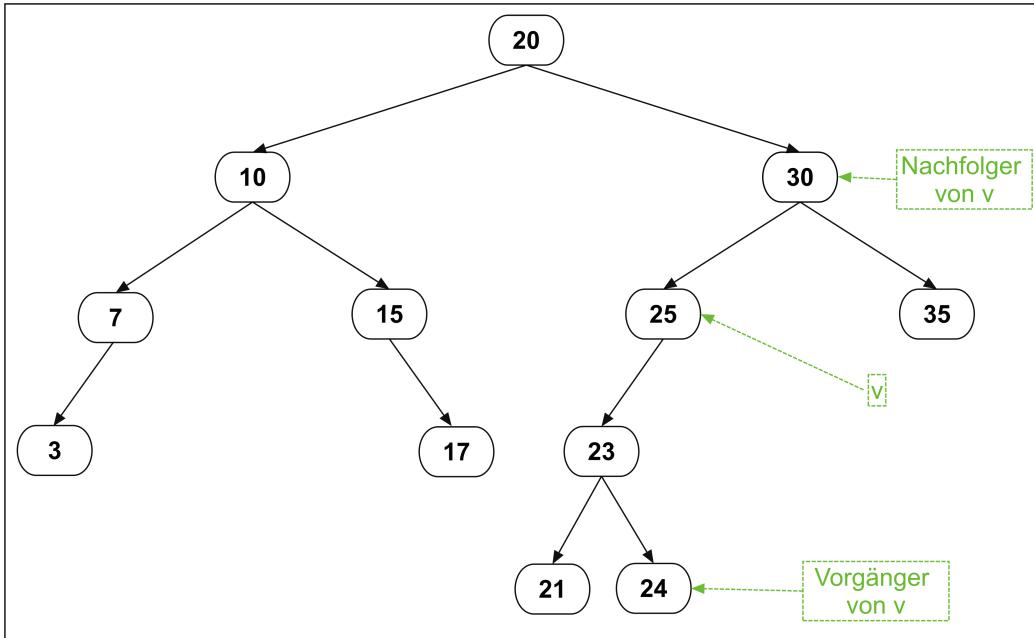
kursives Verfahren, dass an der Wurzel startet und pro Aufruf drei Schritte ausführt.

#### Algorithmus *inorder* (**Knoten** $v$ )

1. Existiert ein linkes Kind  $vl$  von  $v$ , rufe *inorder*( $vl$ ) auf.
2. Gib den Schlüssel von  $v$  aus.
3. Existiert ein rechtes Kind  $vr$  von  $v$ , rufe *inorder*( $vr$ ) auf.

Dass das Verfahren funktioniert sieht man leicht, durch Induktion über die Anzahl der Knoten  $n$ . Für  $n = 1$  funktioniert es, da der einzige im BST enthaltene Schlüssel ausgegeben wird. Wir nehmen nun an, dass die Ausgabe für BSTs mit Knotenzahl  $\leq n$  korrekt ist. Sei  $T_1$  ein BST mit Knotenzahl  $n+1$  und Wurzel  $w$ . Sowohl für den linken als auch für den rechten Teilbaum von  $w$  gilt, dass die Anzahl enthaltener Knoten  $\leq n$  ist. Als erstes wird der linke Teilbaum von  $w$  korrekt ausgegeben, dann der Schlüssel von  $w$  selbst und zuletzt der rechte Teilbaum von  $w$ . Damit wurde auch für den Gesamtbaum die richtige Ausgabe erzeugt. Als **Vorgänger** eines Knoten  $v$ , mit Schlüssel  $k_v$  bezeichnet man den Knoten mit dem größten im BST enthaltenem Schlüssel  $k$  für den gilt  $k < k_v$ . Aus der Inorder-Traversierung kann man eine Anleitung zum Finden des Vorgängers ableiten. Wenn ein linker Teilbaum vorhanden ist, wird der größte Schlüssel in diesem, also der am weitesten rechts liegende, direkt vor  $k$  ausgegeben. Andernfalls wird der Schlüssel des tiefsten Knotens, auf dem Pfad von der Wurzel zu  $v$  ausgegeben, bei dem  $v$  im rechten Teilbaum liegt. Als **Nachfolger** von  $v$ , bezeichnet man den Knoten mit dem kleinsten im BST enthaltenem Schlüssel  $k$  für den gilt  $k > k_v$ . Da dieser Schlüssel bei der Inorder-Traversierung direkt nach  $v$  ausgegeben wird, findet man den zugehörigen Knoten ganz links im rechten Teilbaum von  $v$ , falls ein solcher vorhanden ist. Ansonsten ist es der tiefste Knoten, auf dem Pfad von der Wurzel zu  $v$ , bei dem  $v$  im linken Teilbaum liegt. Abbildung 5 zeigt Vorgänger und Nachfolger eines Knotens.

**Total geordnete Menge** Eine Menge  $M$  wird als **total geordnet** bezeichnet wenn auf ihr eine zweistellige Relation  $\leq$  definiert ist, die folgende Eigenschaften erfüllt.



**Abbildung 5:** Darstellung von Vorgänger und Nachfolger.

Für alle  $a,b,c \in M$  gilt:

1.  $(a,a) \in R$  (reflexiv)
2.  $(a,b) \in R \wedge (b,a) \in R \Rightarrow a = b$  (antisymmetrisch)
3.  $(a,b) \in R \wedge (b,c) \in R \Rightarrow (a,c) \in R$  (transitiv)
4.  $(a,b) \notin R \Rightarrow (b,a) \in R$  (total)

Die Eigenschaften 1,2 und 4 werden benötigt um für zwei beliebige Elemente aus der Menge feststellen zu können ob sie gleich sind, oder bei Ungleichheit, welches Element weiter Links bzw. Rechts im BST liegen muss. Dafür wird z.B getestet ob die Elemente  $(a,b)$  und  $(b,a)$  in der Relation liegen. Eigenschaft 3 ist notwendig, denn liegt  $b$  weiter rechts im BST als  $a$  und  $c$  liegt weiter rechts als  $b$ , dann liegt  $c$  natürlich auch weiter rechts als  $a$ .

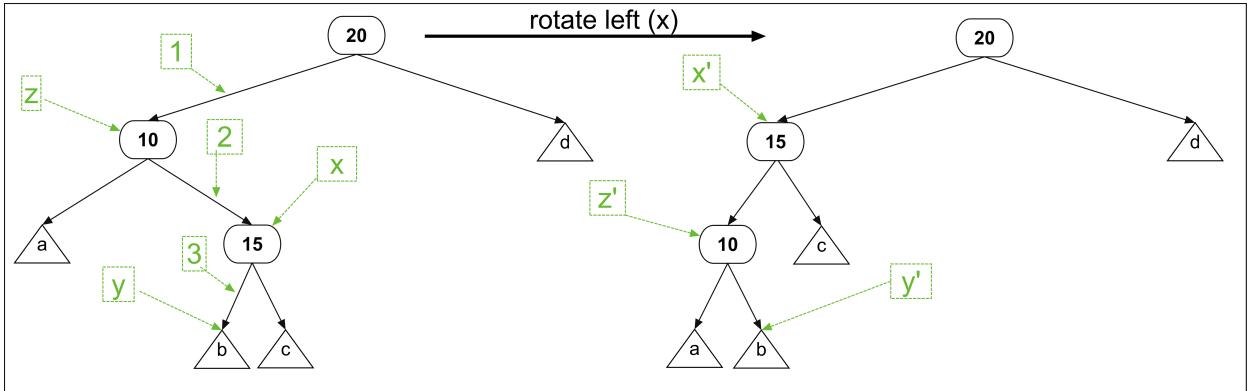
Die von uns verwendete „Kleiner-Gleich-Beziehung“ auf den natürlichen Zahlen erfüllt alle Eigenschaften.

**Verändern eines BST durch Rotationen.** Wird ein BST durch eine Veränderung in einen anderen BST überführt, kann es passieren dass sich

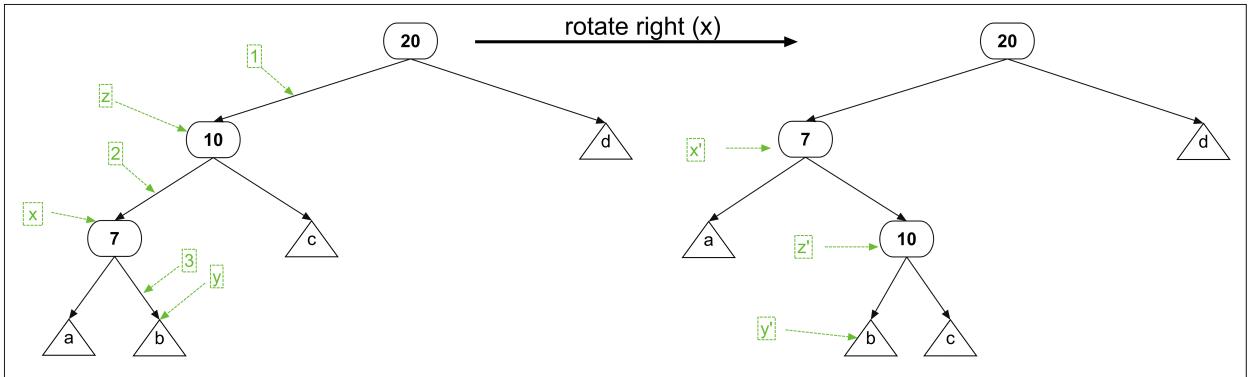
die Eigenschaften eines Knoten ändern. Um nicht immer erwähnen zu müssen auf welchen BST sich eine Aussage bezieht, wird es ab jetzt durchgängig so sein, dass sich ein Variablenname ohne angefügten Apostroph auf den BST vor der Änderung bezieht. Der gleiche Variablenname mit angefügtem Apostroph bezieht sich dann auf den selben Knoten nach der Änderung. Z.B. bezieht sich  $x$  auf den Knoten mit Schlüssel  $k$ , in der Ausgangssituation, dann bezieht sich  $x'$  auf den Knoten mit Schlüssel  $k$  nach dem Ausführen der Änderung.

**Rotationen** können verwendet werden um Änderungen an der Struktur eines BST durchzuführen, ohne eine der geforderten Eigenschaften zu verletzen. Es wird zwischen der Linksrotation und der Rechtsrotation unterschieden. Hier wird zunächst auf die in Abbildung 6 dargestellte Linksrotation eingegangen. Sei  $x$  der Knoten auf dem eine Linksrotation durchgeführt wird. Sei  $z$  der Vater von  $x$ .  $z$  muss existieren, ansonsten darf auf  $x$  keine Rotation durchgeführt werden. Sei  $y$  der linke Teilbaum von  $x$ . Nach der Rotation ist  $x'$  linkes bzw. rechtes Kind von dem Knoten, an dem  $z$  linkes bzw. rechtes Kind war.  $z'$  ist linkes Kind von  $x'$ .  $y'$  ist rechtes Kind von  $z'$ . Unabhängig von der Anzahl der im BST enthaltenen Knoten und der Ausführungsstelle im BST ist eine Linksrotation also mit dem Aufwand verbunden drei Zeiger umzusetzen. Zu beachten ist, dass die Höhen von  $x'$  und der Knoten in dessen, ansonsten unverändertem, rechtem Teilbaum jeweils um eins größer sind als die von  $x$  und den Knoten in dessen rechtem Teilbaum. Die Höhe der Knoten im Teilbaum mit Wurzel  $z'$  sind jeweils um eins kleiner als vor der Rotation. Abbildung 7 zeigt die symmetrische Rechtsrotation. Man muss im obigen Beschreibung lediglich links durch rechts ersetzen und umgekehrt. Dass es durch eine Rotation zu keiner Verletzung der BST Eigenschaften kommt, sieht man den Abbildungen direkt an. In Abbildung 8 erkennt man, dass sich die Wirkung einer Rotation auf  $x$  durch eine gegenläufige Rotation auf  $z'$  aufheben lässt.

**Grundoperationen Suchen, Einfügen und Löschen** Hier geht es nur um die Standardvariante eines BST. Später werden Varianten gezeigt die von diesem Verhalten zum Teil deutlich abweichen. Es sei ein BST  $T$  gegeben. Die Operation **suchen(Schlüssel  $k$ )** gibt eine Referenz auf den Knoten im BST zurück, dessen Schlüssel mit  $k$  übereinstimmt. Die Operation startet an der Wurzel und vergleicht den darin enthaltenen Schlüssel mit dem Gesuchten. Ist der gesuchte Schlüssel kleiner, muss er sich im linken Teilbaum des betrachteten Knoten befinden und die Suche wird bei dessen Wurzel fortgesetzt. Ist der Schlüssel größer, muss er sich im rechten Teilbaum befinden



**Abbildung 6:** Linkssrotation auf Knoten x.

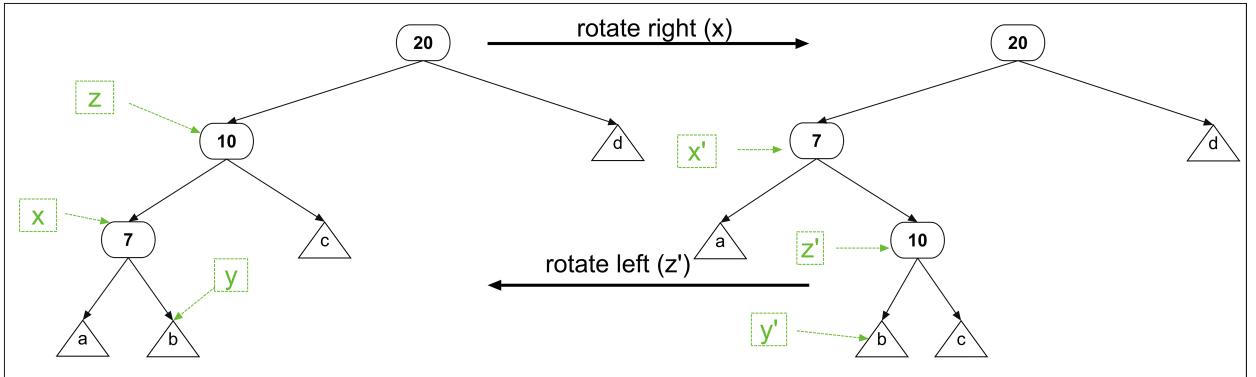


**Abbildung 7:** Rechtsrotation auf Knoten x.

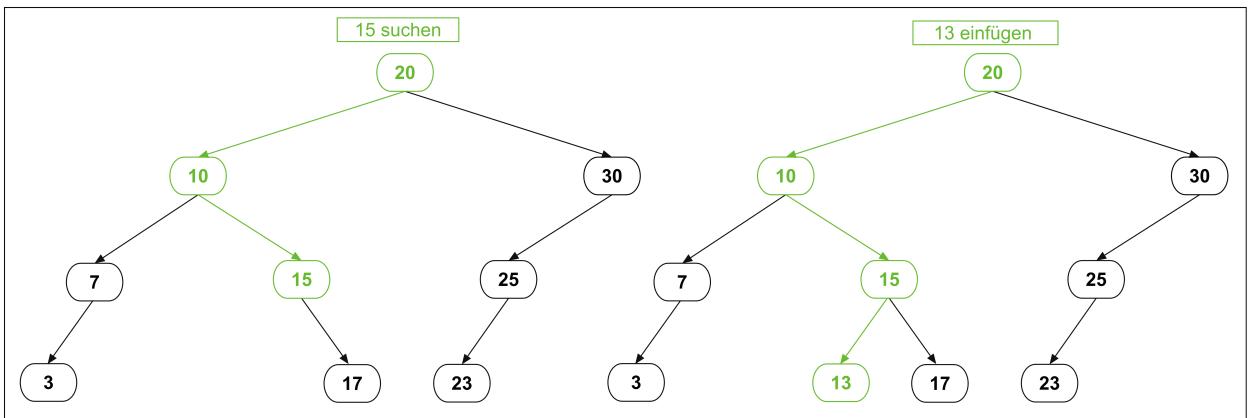
und die Suche wird bei dessen Wurzel fortgesetzt. Dieses Verhalten iteriert solange bis der gesuchte Schlüssel gefunden ist, oder der Teilbaum bei dem die Suche fortgesetzt werden müsste, leer ist. Ist das Letztere der Fall, ist der gesuchte Schlüssel im Baum nicht vorhanden und es wird eine leere Referenz zurückgegeben. In keinem Fall kommt es zu einer Veränderung des BST.

Bei **einfügen(Schlüssel k)** wird zunächst wie beim Suchen nach k verhalten. Findet man den Schlüssel wird das Einfügen abgebrochen und der BST bleibt unverändert. Wird ein leerer Teilbaum  $T_2$  erreicht, wird ein neu erzeugter Knoten mit Schlüssel  $k$  an der Position von  $T_2$  eingefügt. Durch den neuen Knoten wird keine BST Eigenschaft verletzt. Durch ersetzen eines leeren Teilbaumes, durch einen Knoten bleibt es bei einem binären Baum. Das Verhalten von Suchen stellt sicher, dass  $k$  nur in linken Teilbäumen von Knoten mit Schlüssel  $> k$  bzw. in rechten Teilbäumen von Knoten mit Schlüssel  $< k$  liegt.

Auch bei **löschen(Schlüssel k)** wird sich zunächst wie beim Suchen ver-



**Abbildung 8:** Gegenseitiges aufheben von Rotationen



**Abbildung 9:** Links zeigt eine Suche nach dem Schlüssel 15. Rechts das Einfügen des Schlüssels 13

halten. Ist  $k$  im BST nicht vorhanden wird abgebrochen und der BST bleibt unverändert. Ansonsten werden drei Fälle unterschieden. Sei  $v$  der Knoten mit Schlüssel  $k$ .

1.  $v$  ist ein Blatt:  
 $v$  kann ohne weiteres aus dem BST entfernt werden.
2.  $v$  hat genau ein Kind  $c$ :  
Ist  $v$  die Wurzel kann er entfernt werden und  $c$  wird zur neuen Wurzel. Ansonsten ist  $v$  entweder ein linkes oder ein rechtes Kind eines Knoten  $w$ .  $c$  nimmt nun den Platz von  $v$  im BST ein. Das bedeutet, dass die Kanten von  $w$  nach  $v$  und von  $v$  nach  $c$  entfernt werden. Außerdem wird eine Kante von  $w$  nach  $c$  so eingefügt, dass  $c$  wie zuvor  $v$  das linke bzw. rechte Kind von  $w$  wird.

3.  $v$  hat zwei Kinder:

Sei  $T_l$  der linke Teilbaum von  $v$  und  $T_r$  der Rechte. Sei  $z$  der Knoten mit dem kleinsten Schlüssel im rechten Teilbaum von  $v$ . Als Knoten mit dem kleinsten Schlüssel im rechten Teilbaum von  $v$ , kann  $z$  kein linkes Kind haben. Ist  $z$  ein Blatt wird seine eingehende Kante entfernt. Hat  $z$  ein rechtes Kind  $z_r$ , so nimmt dieses, analog zur Beschreibung im Fall 2, den Platz von  $z$  ein. In beiden Fällen ist  $z$  nun ein Knoten ohne Kante. Im nächsten Schritt nimmt nun  $z$  den Platz von  $v$  ein,  $T_l$  wird links an  $z$  angefügt und  $T_r$  rechts. War  $v$  zu Beginn die Wurzel, so wird  $z'$  zur neuen Wurzel.

In keinen Teilbäumen eines Knotens außer denen von  $z$  kommen Schlüssel hinzu. Um eventuelle Verletzungen von Eigenschaften festzustellen, kann sich also auf  $z'$  beschränkt werden. Der linke Teilbaum von  $z'$  war der linke Teilbaum von  $v$  und der Schlüssel von  $v$  ist kleiner als der von  $z$ . Der rechte Teilbaum von  $z$  enthält die Schlüssel des rechten Teilbaumes von  $v$  mit Ausnahme des Schlüssels von  $z$  selbst.  $z$  wurde gerade ausgewählt weil sein Schlüssel der Kleinste in diesem Teilbaum ist.

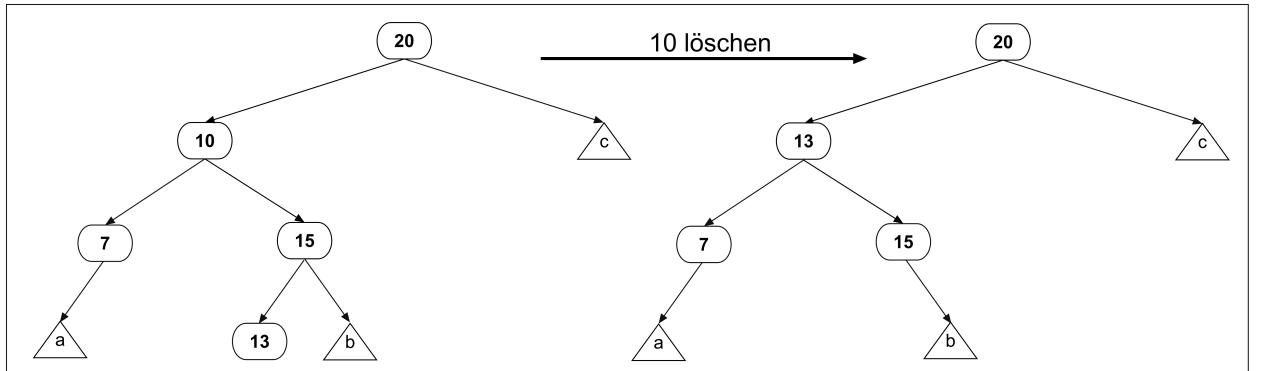


Abbildung 10: Löschen des Schlüssels 10

**Laufzeit** Die worst-case-Laufzeit der drei Operationen ist jeweils  $O(h)$ , wobei  $h$  die Höhe von  $T$  ist. Beim *suchen* werden maximal  $h$  Knoten aus  $T$  betrachtet. Beim Einfügen überlagern die Kosten von Suchen, die konstanten Kosten für das Anhängen des neuen Knotens. Bei *löschen* wird in Fall eins und zwei nach der Suche ebenfalls nur noch lokal beim gesuchten Knoten gearbeitet. Beim *löschen* mit Fall drei muss zunächst zum Knoten  $z$  erreicht werden, dafür sind maximal  $h$  Schritte notwendig. Danach muss  $v$  erreicht

werden, wozu ebenfalls maximal  $h$  Schritte notwendig sind. Die Kosten für das Entfernen und Hinzufügen von Kanten sind an beiden Stellen konstant.

**Unterschiedliche Baumhöhen** Da die Höhe  $h$  eines BST  $T$  mit  $n$  Knoten entscheidend für die Laufzeit der vorgestellten Operationen ist, wird hier auf diese eingegangen. Die maximale Höhe  $n$  erreicht ein BST wenn es ein Blatt im BST gibt und jeder andere Knoten ein Halbblatt ist. Die Baumstruktur geht in diesem Fall über zu einer Listenstruktur über, dies wird als **entarten** bezeichnet. Minimal wird  $h$  wenn  $T$  **vollständig balanciert** ist. Das ist der Fall wenn alle Ebenen über der Untersten vollständig besetzt sind.

**Lemma 3.1.** *Die Höhe eines vollständig balancierten BST  $T$  mit  $n$  Knoten ist  $\lfloor \log_2(n) \rfloor + 1$ .*

*Beweis.* Es sei  $N(h)$  die maximale Anzahl an Knoten in einem vollständig balancierten BST mit Höhe  $h$ .  $N(h)$  berechnet sich indem die maximale Anzahl an Knoten jeder Ebene aufaddiert wird.

$$N(h) = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

$h$  ist minimal wenn gilt:

$$\begin{aligned} N(h-1) &< n \leq N(h) \\ \Leftrightarrow N(h-1) + 1 &\leq n < N(h) + 1 \end{aligned}$$

Einsetzen:

$$\begin{aligned} 2^{h-1} &\leq n < 2^h \\ \Rightarrow h &= \lfloor \log_2(n) \rfloor + 1 \end{aligned}$$

□

## 4 Rot-Schwarz-Baum

Der Rot-Schwarz-Baum gehört zur Gruppe der **balancierten BST** und erfüllt alle Eigenschaften um ihn als Hilfsstruktur im Tango-Baum zu verwenden. Genau das ist auch die Rolle des Rot-Schwarz-Baumes in dieser

Ausarbeitung. Bei balancierten BST gilt für die Höhe  $h = O(\log n)$ , mit  $n =$  Anzahl der Knoten. Jeder Knoten benötigt ein zusätzliches Attribut, um eine Farbinformation zu speichern. Der Name der Datenstruktur kommt daher, dass die beiden durch das zusätzliche Attribut unterschiedenen Zustände als *Rot* und *Schwarz* bezeichnet werden. Die Farbe ist also eine Eigenschaft der Knoten und im folgenden wird einfach von roten bzw. schwarzen Knoten gesprochen. Innerhalb mancher Operationen wird von einem Knoten aus direkt auf dessen Vater zugegriffen, so dass man sich im Baum auch nach oben hin bewegen kann. In Implementierungen wird das so umgesetzt, dass es zusätzlich zu den beiden Zeigern auf die Kinder noch einen zum Vater gibt. Insbesondere kann man dadurch Pfade von tieferen Baumebenen zu höheren konstruieren. Als Blätter werden schwarze Sonderknoten verwendet, deren Schlüssel auf einen Wert außerhalb des Universums, hier *null*, gesetzt wird, um sie eindeutig erkennen zu können. *null* gehört nicht zur Schlüsselmenge des RBT. Fehlende Kinder von Knoten mit gewöhnlichem Schlüssel werden durch solche Blätter ersetzt.

Folgende zusätzliche Eigenschaften müssen bei einem Rot-Schwarz-Baum erfüllt sein.

1. Jeder Knoten ist entweder rot oder schwarz.
2. Die Wurzel ist schwarz.
3. Jedes Blatt (Sonderknoten) ist schwarz.
4. Der Vater eines roten Knotens ist schwarz.
5. Für jeden Knoten gilt, dass alle Pfade, die an ihm starten und an einem Blatt (Sonderknoten) enden, die gleiche Anzahl an schwarzen Knoten enthalten.

Sei  $(v_0, v_1, \dots, v_n)$  ein Pfad von einem Knoten  $v_0$  zu einem Blatt  $v_n$ . Die Anzahl der schwarzen Knoten innerhalb  $(v_1, \dots, v_n)$  wird als **Schwarz-Höhe**  $bh(v_0)$  von Knoten  $v_0$  bezeichnet. Die eigene Farbe des betrachteten Knotens bleibt dabei also außen vor. Dadurch hat ein Knoten die gleiche Schwarz-Höhe wie ein rotes Kind und eine um eins erhöhte Schwarz-Höhe gegenüber einem schwarzen Kind. Die Schwarz-Höhe der Wurzel entspricht der **Schwarz-Höhe des Baumes**  $bh(T)$ , wobei ein leerer Baum Schwarz-Höhe 0 hat. Die Schwarz-Höhe eines Knoten  $x$  ist genau dann eindeutig wenn er Eigenschaft 5 nicht verletzt. Hält  $x$  Eigenschaft 5 ein und sei  $i$  die Anzahl schwarzer Knoten in den entsprechenden Pfaden, so gilt  $bh(x) = i$  wenn  $x$  rot ist und  $bh(x) = i - 1$  wenn  $x$  schwarz ist. Ist  $bh(x)$  eindeutig, so enthält

jeder Pfad der mit  $x$  startet und an einem Blatt endet  $bh(x) + 1$  schwarze Knoten, wenn  $x$  schwarz ist und  $bh(x)$  schwarze Knoten wenn  $x$  rot ist.

Jeder Knoten speichert seine Schwarz-Höhe als weiteres Attribut, da wir dieses in Abschnitt 4.2 benötigen. Natürlich muss das Attribut, dann auch gesetzt und gepflegt werden, wobei es bei Sonderknoten fest mit 0 belegt ist. Die im folgenden wird **RBT** (Red-Black-Tree) als Abkürzung für Rot-Schwarz-Baum verwendet. Aufgrund der Sonderknoten gibt es eine etwas spezielle Situation, bei einem RBT mit Höhe 1. Diese Konstellation ist nur mit einem einzelnen Sonderknoten erreichbar, so dass man statt dessen auch einfach den leeren Baum verwenden könnte. Auch diese Konstellation erfüllt aber die Eigenschaften, so dass sie kein Problem darstellt.

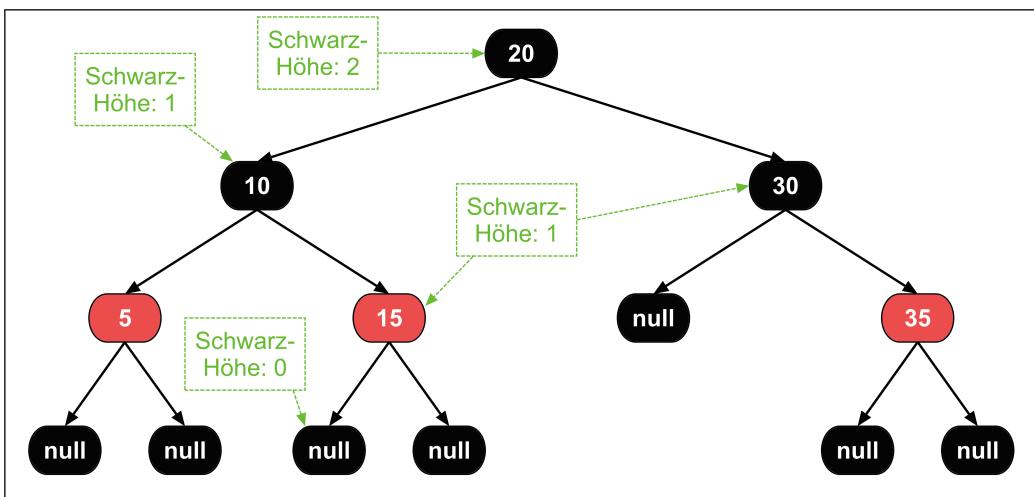
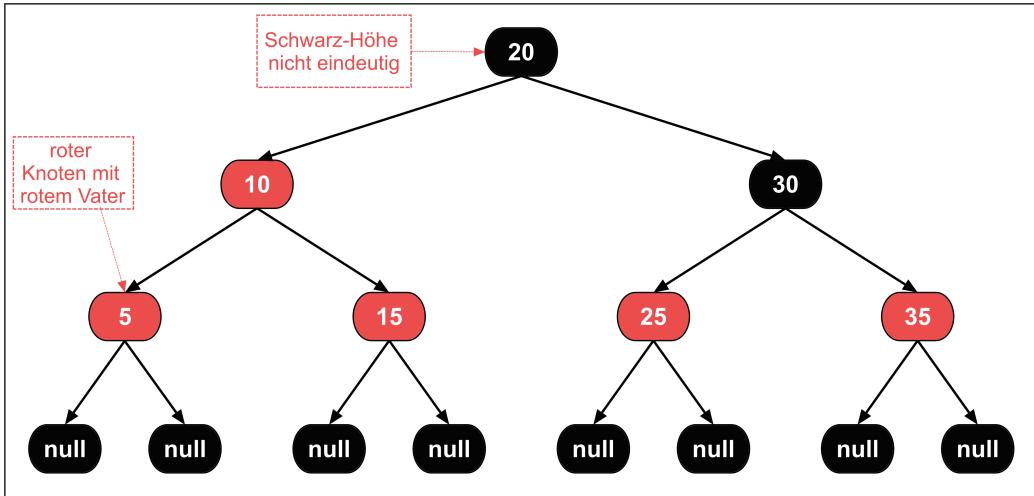


Abbildung 11: Rot-Schwarz-Baum ohne Verletzung von Eigenschaften.

## 4.1 Grundoperationen

**Suchen im Rot-Schwarz-Baum** Die Suche unterscheidet sich nur in einem Punkt von der in 3.2 vorgestellten. Wird nach einem Schlüssel gesucht, der im RBT nicht vorhanden ist, so wird einer der Sonderknoten erreicht. In diesem Fall wird die Suche abgebrochen und eine leere Referenz zurückzugeben. Die Operation verändert den RBT nicht.

**Einfügen in den Rot-Schwarz-Baum** Sei  $k$  der einzufügende Schlüssel. Zunächst wird wie beim Suchen vorgegangen. Wird  $k$  gefunden wird der RBT nicht verändert. Ansonsten wird ein Sonderknoten  $b$  erreicht. Ein neu

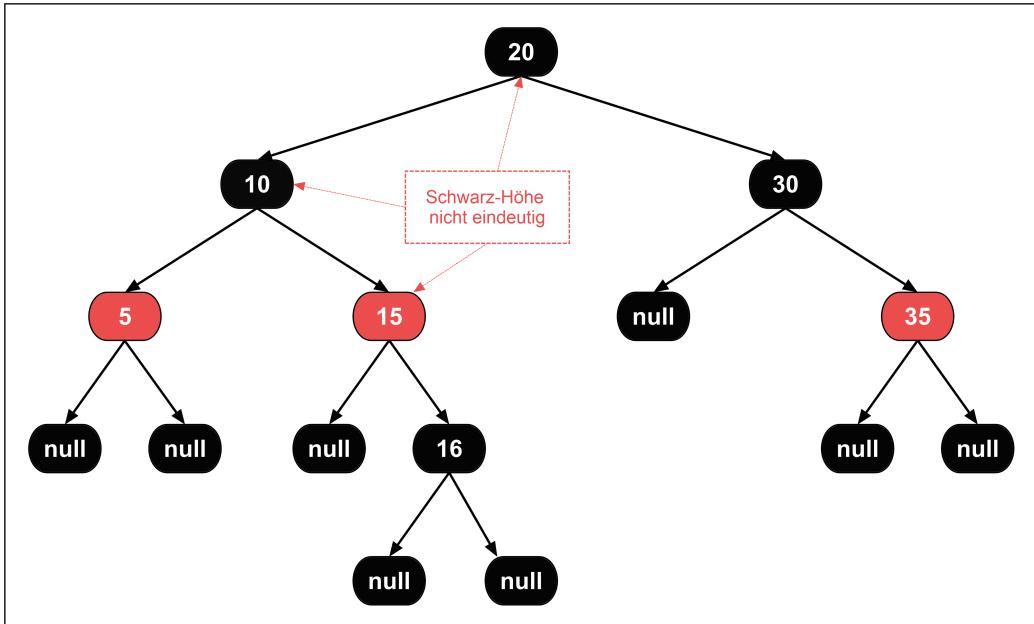


**Abbildung 12:** Rot-Schwarz-Baum bei dem Eigenschaft vier und fünf verletzt sind.

erzeugter roter Knoten  $v_k$  mit Schlüssel  $k$  und Schwarz-Höhe 1 nimmt den Platz von  $b$  ein.  $k$  werden Sonderknoten als linkes und rechtes Kind angefügt.  $k$  ist nun im Baum enthalten, es muss jedoch auf mögliche Verletzungen der fünf Eigenschaften geachtet werden. Welche können betroffen sein ?

1. Es ist immer noch jeder Knoten entweder rot oder schwarz.
2. Wurde in den leeren Baum eingefügt, so ist der neu eingefügte rote Knoten die Wurzel, was eine Verletzung darstellt. Waren bereits Knoten im Baum vorhanden blieb die Wurzel unverändert.
3. Aufgrund der Sonderknoten sind die Blätter immer noch schwarz.
4. Der Baum wird nur direkt an der Einfügestelle verändert. Der neue Knoten hat schwarze Kindknoten, er könnte jedoch einen roten Vater haben, so dass diese Eigenschaft verletzt wäre.
5. Die Schwarz-Höhe von  $v_k$  ist korrekt gesetzt. Die Schwarz-Höhe keines anderen Knotens hat sich verändert, denn den Platz eines schwarzen Knoten mit Schwarz-Höhe 0 nimmt nun ein roter Knoten mit Schwarz-Höhe 1 ein. Eigenschaft fünf bleibt also erhalten.

Es können also die Eigenschaften zwei und vier betroffen sein. Jedoch nur eine von ihnen, denn Eigenschaft zwei wird genau dann verletzt wenn der neue Knoten die Wurzel des Baumes ist, dann kann er aber keinen roten Vater haben.



**Abbildung 13:** Rot-Schwarz-Baum bei dem Eigenschaft fünf verletzt ist.

Zur Korrektur wird zum Ende von *einfügen* eine zusätzliche Operation, **einfügen-fixup(Knoten  $v_{in}$ )** aufgerufen. Diese Operation arbeitet sich von  $v_i$  startend, solange in einer Schleife nach oben im BST durch, bis alle Eigenschaften wieder erfüllt sind. Die Schleifenbedingung ist, dass eine Verletzung vorliegt. Dazu muss geprüft werden ob der betrachtete Knoten  $x$  die rote Wurzel des Gesamtbaumes ist, oder ob er und sein Vater beide rot sind. Vor dem ersten Durchlauf wird  $x = v_{in}$  gesetzt. Innerhalb der Schleife werden sechs Fälle unterschieden. Im folgenden wird auf vier Fälle detailliert eingegangen. Die restlichen zwei verhalten sich symmetrisch zu einem solchen. Jeder der Fälle verantwortet, dass zum Start der nächsten Iteration wieder nur maximal eine der beiden Eigenschaften zwei oder vier verletzt sein kann und Eigenschaft vier höchstens an einem Knoten verletzt ist. Die Fallauswertung geschieht in aufsteigender Reihenfolge. Deshalb kann man innerhalb einer Fallbehandlung verwenden, dass die vorherigen Fallbedingungen nicht erfüllt sind. Eigenschaft eins bleibt in der Beschreibung außen vor, da es während der gesamten Ausführungszeit der Operation nur Knoten gibt, die entweder rot oder schwarz sind.

**Fall 1:  $x$  ist die rote Wurzel des RBT:** Dieser Fall wird behandelt in dem man die Wurzel schwarz färbt. Man muss noch zeigen, dass es durch das Umfärben zu keiner anderen Verletzung gekommen ist.

Betrachtung der Eigenschaften:

1. -
2. Die Wurzel wurde schwarz gefärbt.
3. Die Blätter (Sonderknoten) sind unverändert.
4. Es wurden weder rote Knoten hinzugefügt, noch wurde die Kantenmenge verändert.
5. Das Umfärben der Wurzel kann hierauf keinen Einfluss haben, da sie in der Berechnung der Schwarz-Höhe jedes Knotens außen vor ist.

Es wird also keine Eigenschaft mehr verletzt und die Schleife wird keine weitere Iteration durchführen.

Die Fälle 2 - 6 behandeln nun die Situationen, in denen sowohl  $x$  als auch dessen Vater  $y$  rote Knoten sind. Da Eigenschaft fünf nach jeder Iteration erfüllt ist muss  $y$  einen Bruder haben. Denn da zu Beginn einer Iteration nur eine Eigenschaft verletzt sein kann, kann der rote  $y$  nicht die Wurzel sein, also muss auch  $y$  einen Vorgänger  $z$  haben. Da  $z$  kein Blatt(Sonderknoten) ist, müssen beide Kinder vorhanden sein.

Außerdem muss  $z$  schwarz sein, ansonsten wäre Eigenschaft vier an zwei Knoten verletzt.

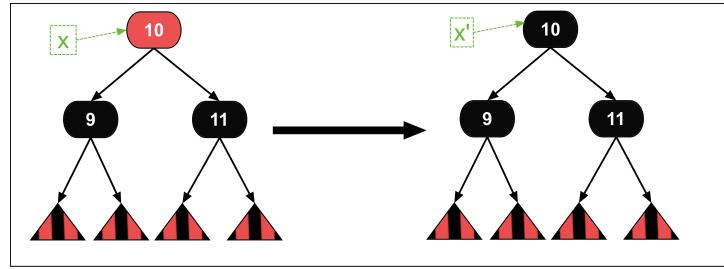
**Fall 2:  $y$  hat einen roten Bruder:** Diesen Fall veranschaulicht Abbildung 15. Es wird  $z$  rot gefärbt und beide Kinder von  $z$ , also  $y$  und dessen Bruder, schwarz. Die Schwarz-Höhe von  $z$  wird um eins erhöht. Somit ist der Vater von  $x$  nun schwarz und die Verletzung der Eigenschaft vier wurde an dieser Stelle behoben. Wie sieht es aber mit den Verletzungen insgesamt aus ?

Betrachtung der Eigenschaften:

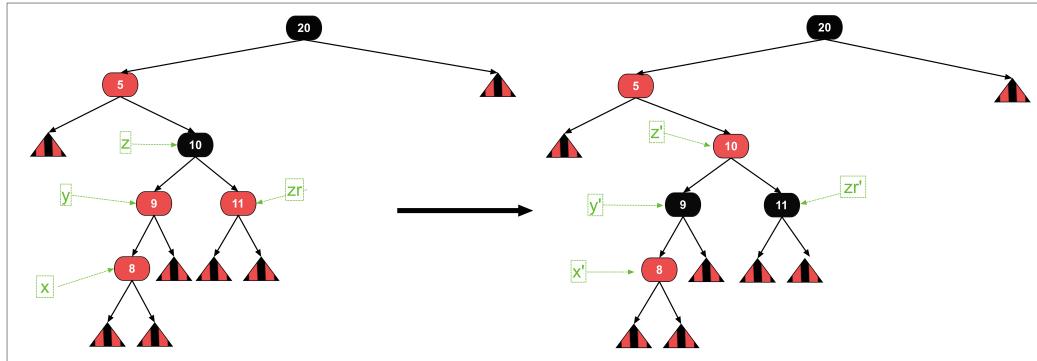
1. -
2. Wenn  $z$  die Wurzel des Baumes ist, wurde sie rot gefärbt und eine Verletzung liegt vor.
3. Der rot umgefärzte Knoten  $z'$  hat zwei Kinder, somit wurde kein Blatt rot gefärbt.
4. Wenn der rot gefärbte Knoten  $z'$  nicht die Wurzel ist, könnte er einen roten Vater haben und Eigenschaft vier ist weiterhin Verletzt. Das Problem liegt nun aber zwei Baumebenen höher.

5. Die Schwarz-Höhen der Vorfahren von  $z'$  bleiben unverändert, da jeder Pfad von ihnen zu einem Blatt auch entweder  $y'$  oder dessen Bruder enthält.  $z'$  Schwarz-Höhe steigt um eins gegenüber  $z$ , bleibt aber eindeutig. An keinem anderen Knoten ändert sich die Schwarz-Höhe.

Es kann also wieder nur entweder Eigenschaft zwei oder vier verletzt sein. Wenn das Problem noch nicht an der Wurzel ist, liegt es zumindest zwei Ebenen näher daran.  $x$  wird auf  $z'$  gesetzt.



**Abbildung 14:** einfügen-fixup. Dargestellt ist Fall 1



**Abbildung 15:** einfügen-fixup. Dargestellt ist Fall 2

### Fall 3: $x$ ist ein linkes Kind. $y$ ist ein linkes Kind:

Abbildung 16 zeigt eine entsprechende Situation. Es wird eine Rechtsrotation auf  $y$  ausgeführt. Anschließend wird  $z$  rot gefärbt und  $y$  schwarz.

Betrachtung der Eigenschaften:

Dazu werden vier weitere Variablen auf Knoten verwendet. Es zeigt  $xl$  auf das linke Kind von  $x$ ,  $xr$  entsprechend das rechte Kind.  $yr$  und  $zr$  bezeichnen die rechten Kinder von  $y$  bzw.  $z$ . Nachfolgend wird verwendet, dass die Teilbäume mit den Wurzeln  $xl$ ,  $xr$ ,  $yr$  und  $zr$  durch die Ausführung unverändert bleiben.

1. -
2. Wenn  $z$  zu Beginn nicht die Wurzel des Gesamtbaumes war, bleibt diese unverändert. Ansonsten wurde durch die Rotation  $y'$  zur neuen Wurzel und  $y'$  wurde schwarz gefärbt.
3. In der zweiten Ebene unter  $y'$  befinden sich ausschließlich die unveränderten Teilbäumen mit den Wurzeln  $xl'$ ,  $xr'$ ,  $yr'$  oder  $zr'$ . An den Blättern verändert sich also durch die Ausführung nichts.
4. Knoten  $x'$  ist linkes Kind des schwarzen  $y'$ . Die Teilbäume von  $x'$  blieben unverändert. Der linke Teilbaum von  $y'$  enthält somit keine aufeinanderfolgenden roten Knoten. Das rechte Kind von  $y'$  ist der rote Knoten  $z'$ . Rechts an  $z'$  hängt nun ein unveränderter Teilbaum, dessen Wurzel zuvor Bruder von  $y$  war. Dieser ist nach Fallunterscheidung ein schwarzer Knoten. Links hängt ebenfalls ein unveränderter Teilbaum, dessen Wurzel zuvor rechter Nachfolger von  $y$  war. Der rechte Nachfolger von  $y$  muss schwarz sein, ansonsten wäre Eigenschaft vier an zwei Knoten verletzt gewesen. Im Teilbaum mit Wurzel  $y$  gibt es also keine aufeinanderfolgenden roten Knoten. Da  $y'$  schwarz gefärbt wurde, kann auch außerhalb des Teilbaumes mit  $y'$  keine neue Verletzung entstanden sein.
5. Es gilt  $bh(xl) = bh(xr) = bh(yr) = bh(zr) = bh(z) - 1$ . Wie oben bereits erwähnt wird die zweite Ebene unter der Wurzel  $y'$  von den unveränderten Teilbäumen  $xl'$ ,  $xr'$ ,  $yr'$  und  $zr'$  gebildet. Es müssen also lediglich die Knoten  $x'$ ,  $y'$  und  $z'$  betrachtet werden. Die Kinder von  $x'$  und  $z'$  sind schwarze Knoten mit der Schwarz-Höhe  $bh(z) - 1$ . Die Schwarz-Höhen von  $x'$  und  $z'$  sind also eindeutig und es gilt  $bh(x') = bh(z') = bh(z)$ .  $y'$  Kinder sind die roten Knoten  $x'$  und  $z'$ . Da beide Kinder rot sind gilt  $bh(y') = bh(x') = bh(z)$ . Somit sind alle Schwarz-Höhen im betrachteten Teilbaum eindeutig. Die neue Wurzel der Teilbaumes  $y'$  hat die gleiche Schwarz-Höhe und die gleiche Farbe wie die vorherige Wurzel  $z$ . Damit kann es auch im Gesamtbaum zu keiner Verletzung der Eigenschaft gekommen sein.

Es ist keine der Eigenschaften verletzt, daher wird es zu keiner Iteration mehr kommen.

#### **Fall 4: $x$ ist ein rechtes Kind. $y$ ist ein linkes Kind.:**

Dieser in Abbildung 17 gezeigte Fall wird so umgeformt, dass eine Situation entsteht bei der Fall drei angewendet werden kann. Dazu wird eine Linksrotation an Knoten  $x$  durchgeführt.

Betrachtung der Eigenschaften:

Zu Veränderungen kommt es durch die Rotation lediglich im linken Teilbaum von  $z$ . Es sei  $xl$  das linke Kind von  $x$ , und  $xr$  das rechte Kind von  $x$ .  $yl$  ist das linke Kind von  $y$ .  $xl$ ,  $xr$  und  $yl$  müssen schwarz sein, ansonsten wäre Eigenschaft vier mehrfach verletzt gewesen.

1. -
2. Die Wurzel bleibt unverändert.
3. Die Teilbäume mit den Wurzeln  $xl$ ,  $xr$  und  $yl$  enthalten alle Blätter innerhalb des linken Teilbaumes von  $z$ . Die Teilbäume  $xl$ ,  $xr$  und  $yl$  bleiben durch die Rotation unverändert und  $xl'$ ,  $xr'$  und  $yl'$  enthalten auch alle Blätter des linken Teilbaumes von  $z'$ .
4. Da  $x$  und  $y$  rot sind müssen  $z$ ,  $xl$  und  $xr$  schwarz sein. Nach der Rotation ist  $y'$  linkes Kind von  $x'$ .  $x'$  ist Kind vom schwarzen  $z'$ . Alle verbleibenden Kinder von  $x'$  und  $y'$  werden durch die unveränderten Teilbäume  $xl'$ ,  $xr'$  und  $yl'$  gebildet. Deren Wurzeln müssen schwarz sein, ansonsten hätte es in ursprünglichen Baum an mehr als einem Knoten eine Verletzung von Eigenschaft vier gegeben. Durch die Rotation verbleibt es also bei einer Verletzung der Eigenschaft vier in der gleichen Baumebene. Die beiden beteiligten roten Knoten sind nun aber jeweils linke Kinder.
5.  $bh(yl) = bh(xl) = bh(xr) = bh(yl') = bh(xl') = bh(xr')$ . Die Schwarz-Höhen von  $x$  und  $y$  bleiben unverändert. Damit kommt es auch bei  $z$  zu keiner Veränderung der Schwarz-Höhe.

Es sind also weiterhin zwei rote aufeinanderfolgende rote Knoten in den gleichen Baumebenen vorhanden. Diese sind nun aber beides linke Kinder. Der Bruder des oberen roten Knotens ist der selbe schwarze Knoten wie vor der Ausführung von Fall 4. Damit kann direkt mit dem bearbeiten von Fall 3 begonnen werden. Es benötigt keine weitere Iteration.

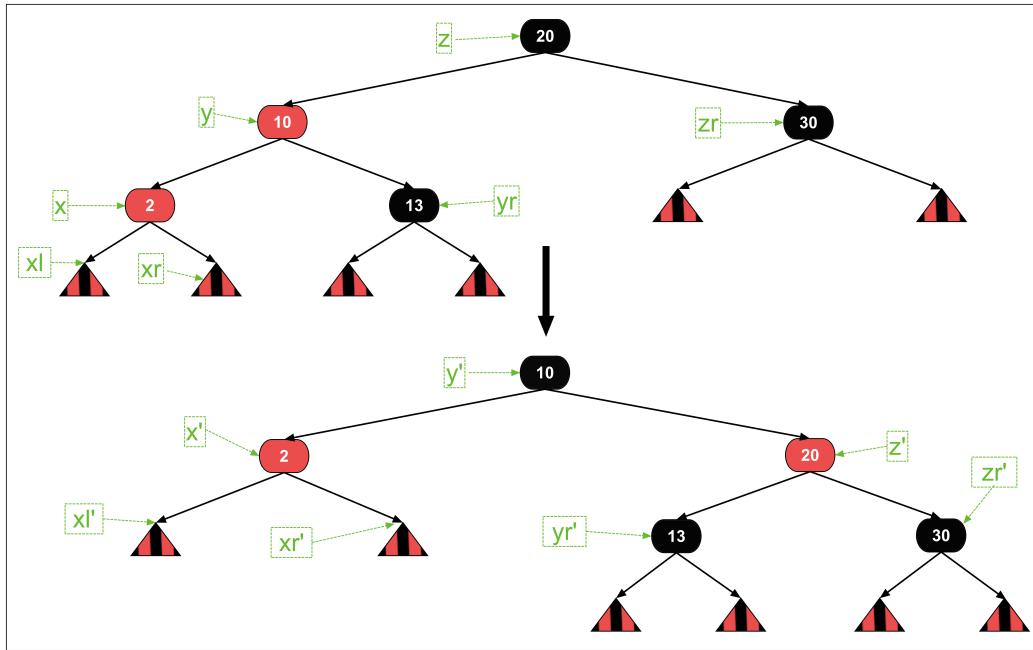
**Fall 5:  $x$  ist ein rechtes Kind.  $y$  ist ein rechtes Kind:**

Abbildung 18 zeigt den zu Fall 3 Links/Rechts-Symmetrischen Fall 5.

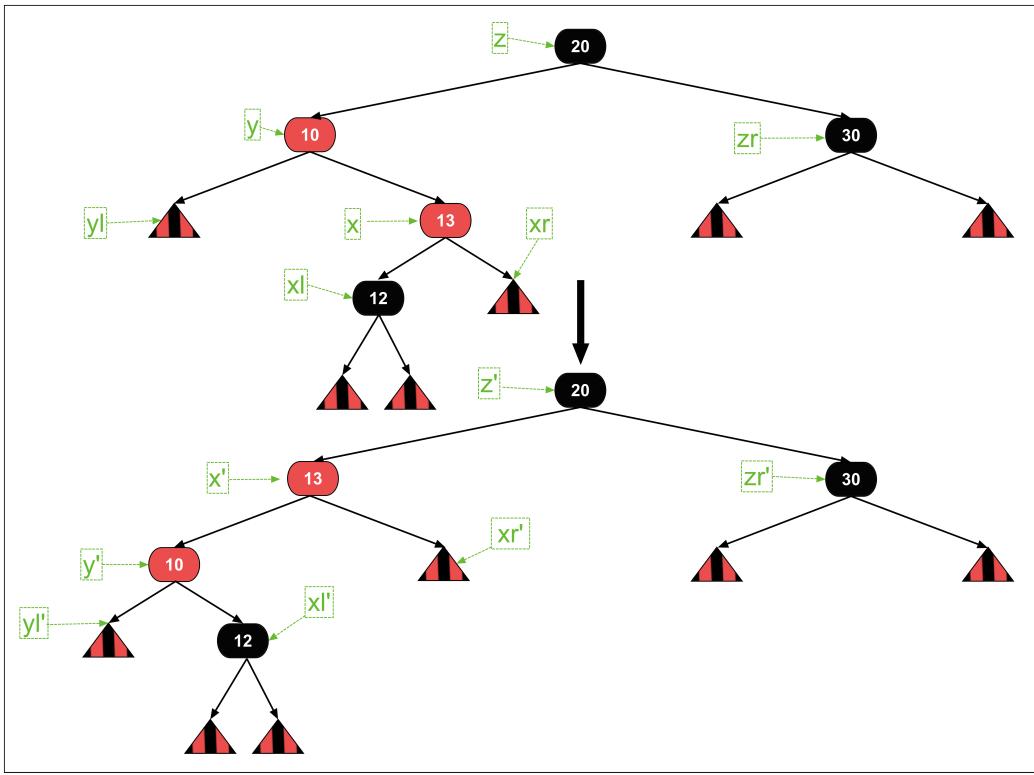
**Fall 6:  $x$  ist ein linkes Kind.  $y$  ist ein rechtes Kind:**

Abbildung 19 zeigt den zu Fall 4 Links/Rechts-Symmetrischen Fall 6.

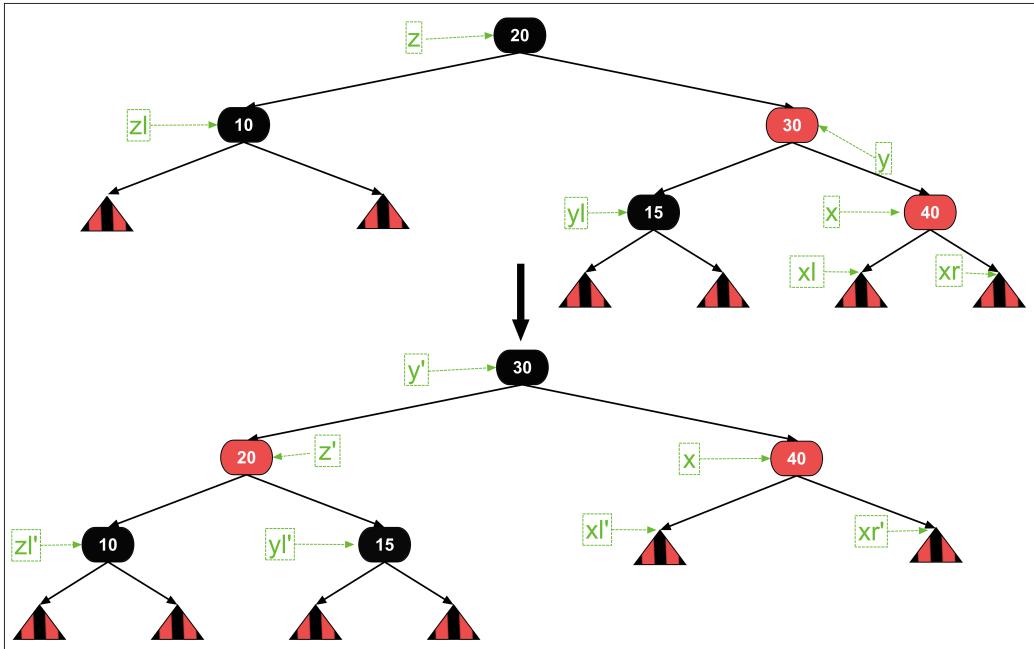
**Laufzeit** Sei  $h$  die Höhe des Gesamtbaumes vor Aufruf von einfügen-Fixup. Fall 2 kann maximal  $h/2$  mal ausgewählt werden, bevor  $x$  oder  $y$  an der Wurzel liegt. Nach einer Iteration bei der nicht Fall 2 ausgewählt wird, terminiert einfügen-fixup. Der Aufwand innerhalb jeder Fallbehandlung ist  $O(1)$ . Für die Gesamtaufzeit gilt deshalb  $O(h)$ .



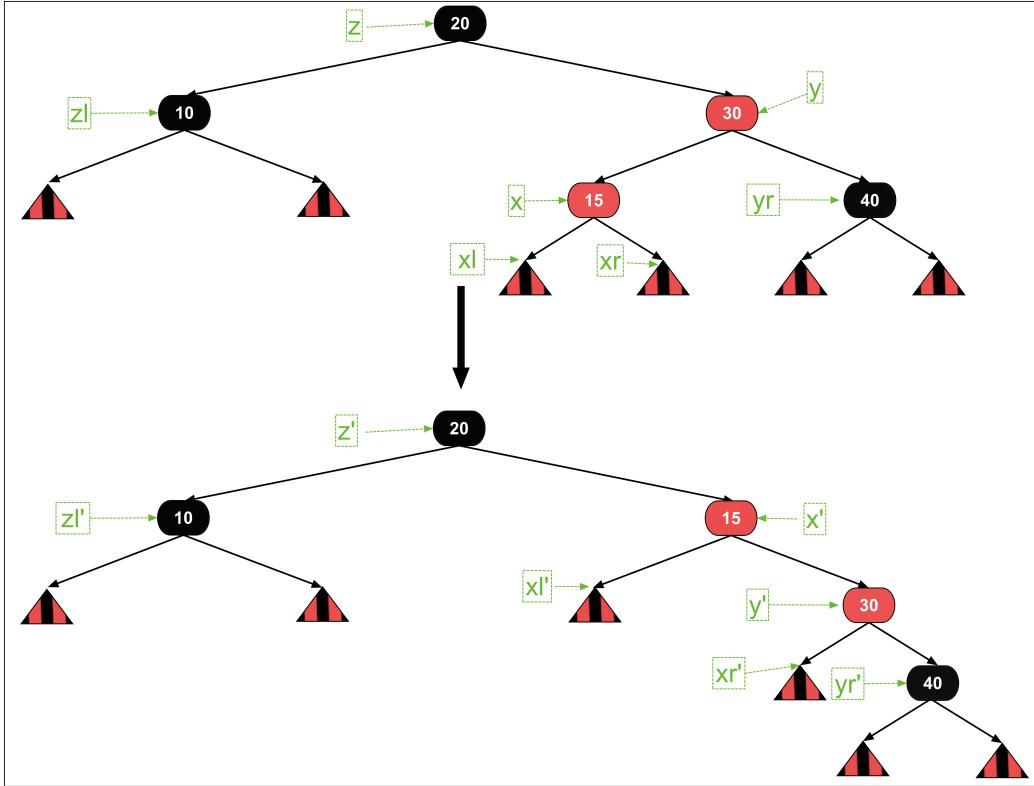
**Abbildung 16:** einfügen-fixup. Dargestellt ist Fall 3



**Abbildung 17:** einfügen-fixup. Dargestellt ist Fall 4



**Abbildung 18:** einfügen-fixup. Dargestellt ist Fall 5



**Abbildung 19:** einfügen-fixup. Dargestellt ist Fall 6

**Löschen aus dem Rot-Schwarz-Baum** Die Reparatur des RBT nach dem entfernen eines Knotens ist aufwendiger, als die nach dem einfügen. Da der RBT in der Rolle als Hilfsstruktur für den Tango-Baum keine solche Operation benötigt, entfällt die Beschreibung. In [1] ist eine detaillierte Beschreibung enthalten.

**Laufzeit der Grundoperationen** Zu Beginn des Kapitels wurde erwähnt, dass für die Höhe  $h$  eines RBT mit  $n$  Knoten  $h = O(\log n)$  gilt. Das wird nun gezeigt.

**Lemma 4.1.** *Für die Höhe  $h$  eines RBT  $T$  mit  $n$  Knoten gilt  $h = O(\log n)$ .*

*Beweis.* Sei  $w$  die Wurzel von  $T$  und  $m$  die Anzahl der inneren Knoten von  $T$ . Zunächst wird gezeigt, dass  $T$  mindestens  $2^{bh(w)} - 1$  innere Knoten enthält. Dies geschieht mit Induktion über  $h$ . Für  $h = 0$  und  $h = 1$  mit  $2^0 - 1 = 0$  stimmt die Behauptung, denn der Baum ist leer oder ein einzelner Sonderknoten. Induktionsschritt mit Höhe  $h + 1$ :

Sei  $tl$  der linke Teilbaum von  $w$  und  $tr$  der rechte Teilbaum von  $w$ . Im Induktionsschritt kann nun verwendet werden, dass  $h > 1$  gilt und  $w$  ein innerer Knoten sein muss.  $tl$  und  $tr$  haben Schwarz-Höhe  $bh(w) - 1$  wenn ihre Wurzel schwarz ist und Schwarz-Höhe  $bh(w)$  wenn ihre Wurzel rot ist. Ihre Höhe ist  $h - 1$  und somit enthalten sie nach Induktionsnahme mindestens  $2^{bh(w)-1} - 1$  innere Knoten. Aufzählen ergibt die Behauptung.

$$\begin{aligned} m &\geq 2^{bh(w)-1} - 1 + 1 + 2^{bh(w)-1} - 1 = 2^{bh(w)} - 1 \\ \Rightarrow \log_2(m+1) &\geq bh(w) \end{aligned}$$

Es gilt folgender Zusammenhang, da höchstens jeder zweite Knoten in einem Pfad rot sein kann

$$\begin{aligned} h(w) &\leq 2 \cdot bh(w) + 1 \\ \Rightarrow \frac{h(w) - 1}{2} &\leq bh(w) \end{aligned}$$

Einsetzen liefert:

$$\begin{aligned} \log_2(m+1) &\geq \frac{h(w) - 1}{2} \\ \Rightarrow 2 \cdot \log_2(m+1) + 1 &\geq h(w) \\ \Rightarrow h(w) &= O(\log m) \end{aligned}$$

Es kann nur maximal doppelt so viele Blätter wie innere Knoten geben. Daraus folgt.

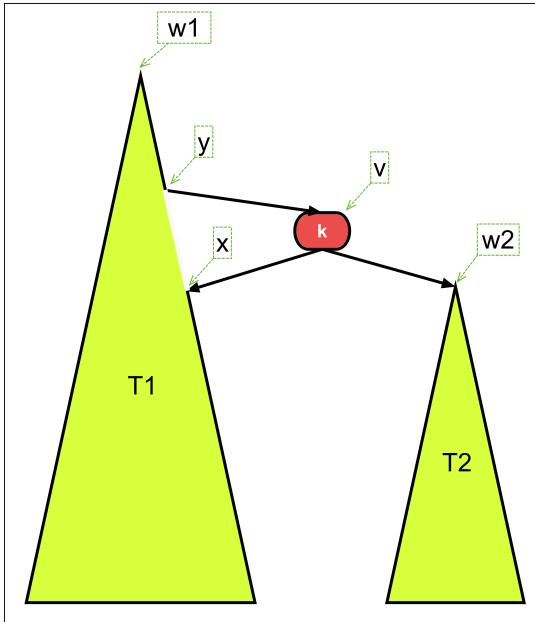
$$\begin{aligned} n &\leq 3m \\ \Rightarrow h(w) &= O(\log n) \end{aligned}$$

□

*suchen* hat also Laufzeit  $O(\log n)$ . *einfügenFixup* mit übergebenen Knoten  $v$  Einfügen hat Kosten  $O(\log n)$  für suchen und  $O(\log n)$  für *einfügenFixup*, insgesamt also  $O(\log n)$ .

## 4.2 Tango-Baum konformes vereinigen

Hier wird die *vereinigen(RBT T<sub>1</sub>, Schlüssel k, RBT T<sub>2</sub>)* Operation eingeführt, wie es ein Tango-Baum von seiner Hilfsstruktur verlangt. Sei  $K_1$  die Schlüsselmenge von  $T_1$  und  $K_2$  die von  $T_2$ . Die Operation gibt eine Referenz



**Abbildung 20:** Beispielhaftes vereinigen zweier RBT unterschiedlicher Schwarz-Höhe, nach Schritt 1.

auf die Wurzel eines vereinigten RBT  $T$  mit Schlüsselmenge  $K_1 \cup K_2 \cup \{k\}$  zurück, dabei werden  $T_1$  und  $T_2$  zerstört. An die Parameter wird die Vorbedingung  $(\forall i \in \mathbb{N} : i < k) \wedge (\forall j \in \mathbb{N} : k < j)$  gestellt.

Es werden im ersten Schritt der Ausführung drei Fälle unterschieden, wobei wieder der erste zutreffende Fall in aufsteigender Reihenfolge ausgewählt wird.

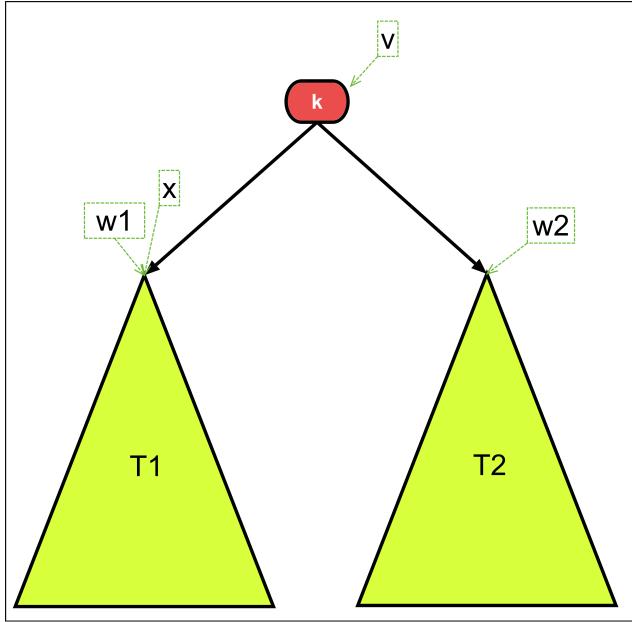
**Fall 1:**  $bh(T_1) = bh(T_2) = 0$

In diesem Fall wird ein roter Knoten  $v$  mit Schlüssel  $k$  und Schwarz-Höhe 1 erzeugt. An diesem werden zwei Sonderknoten angefügt.  $v$  ist die Wurzel von  $T$ .

In den restlichen Fällen ist nun immer zumindest ein Baum vorhanden, der über eine Wurzel verfügt. Der RBT mit der geringeren Schwarz-Höhe wird dabei an den mit der höheren „seitlich angefügt“. Abbildung 20 zeigt dies beispielhaft. Sind die Schwarz-Höhen gleich wird wie in Abbildung 21 vorgegangen. Nun werden die verbleibenden Fälle beschrieben.

**Fall 2:**  $bh(T_2) \leq bh(T_1)$

In diesem Fall wird  $T_2$  bei  $T_1$ , mit Hilfe von  $k$ , so angefügt, dass die Schwarz-Höhe jedes Knoten unverändert bleibt. Es sei  $w_1$  die Wurzel von  $T_1$ . Es sei  $p$  ein Pfad  $(r_0, r_1, \dots, r_l)$  in  $T_1$ , so dass  $r_0 = w_1$ ,  $r_l$  ein Blatt ist und  $\forall i \in \{1, 2, \dots, l\} : r_i$  ist rechtes Kind von  $r_{i-1}$ .  $p$  ist also der am weitesten



**Abbildung 21:** Beispielhaftes vereinigen zweier RBT gleicher Schwarz-Höhe, nach Schritt 1

rechts liegende Pfad von der Wurzel zu einem Blatt. Sei  $x$  der schwarze Knoten in  $p$ , mit  $bh(x) = bh(w_2)$ .  $x$  muss existieren denn  $bh(w_1) \geq bh(w_2)$  und  $bh(r_l) \leq bh(w_2)$ , außerdem sind  $w_1$  und  $r_l$  schwarz.

Nun wird ein neuer roter Knoten  $v$  mit Schlüssel  $k$  und Schwarz-Höhe  $bh(x) + 1$  erzeugt. Als linkes Kind von  $v$  wird  $x$  gesetzt, als rechtes Kind  $w_2$ . Ist  $x$  die Wurzel in  $T_1$ , so ist  $v$  die Wurzel von  $T$ . Ansonsten ist  $x$  rechtes Kind eines Knoten  $y$  und das rechte Kind von  $y$  wird auf  $v$  gesetzt. Außerdem ist dann  $w_1$  die Wurzel von  $T$ .

### Fall 3: $bh(T_1) < bh(T_2)$

Dieser Fall ist fast symmetrisch zu Fall 2, jedoch kann der neue Knoten nicht zur Wurzel von  $T$  werden, da  $bh(T_1) \neq bh(T_2)$ . Es wird  $T_1$  bei  $T_2$ , mit Hilfe von  $k$  angefügt. Es sei  $w_2$  die Wurzel von  $T_2$ . Es sei  $p$  ein Pfad  $(r_0, r_1, \dots, r_l)$  in  $T_2$ , so dass  $r_0 = w_2$ ,  $r_l$  ein Blatt ist und  $\forall i \in \{1, 2, \dots, l\}$ :  $r_i$  ist linkes Kind von  $r_{i-1}$ .  $p$  ist also der am weitesten links liegende Pfad von der Wurzel zu einem Blatt. Sei  $x$  der schwarze Knoten in  $p$ , mit  $bh(x) = bh(w_1)$ .  $x$  muss existieren denn  $bh(w_2) < bh(w_1)$  und  $bh(r_l) \leq bh(w_1)$ , außerdem sind  $w_2$  und  $r_l$  schwarz. Nun wird ein neuer roter Knoten  $v$  mit Schlüssel  $k$  und Schwarz-Höhe  $bh(x) + 1$  erzeugt. Als rechtes Kind von  $v$  wird  $x$  gesetzt, als linkes Kind  $w_2$ .  $x$  ist linkes Kind eines Knoten  $y$  und das linke Kind von  $y$  wird auf  $v$  gesetzt.  $w_2$  ist die Wurzel von  $T$ .

**Resultat nach der Fallbehandlung** Dass ein Baum mit Schlüsselmenge  $K_1 \cup K_2 \cup \{k\}$  entstanden ist, erkennt man direkt an den Abbildungen 20 und 21. Aufgrund der Vorbedingung an die Parameter, muss  $T$  auch ein BST sein. Es müssen aber wieder die fünf Eigenschaften eines RBT betrachtet werden:

1. Es ist immer noch jeder Knoten entweder rot oder schwarz.
2. Gilt  $bh(T_1) \neq bh(T_2)$  so wurde mit  $w_1$  oder  $w_2$  ein schwarzer Knoten zur Wurzel von  $T$ . Andernfalls ist  $v$  die rote Wurzel von  $T$  und diese Eigenschaft ist verletzt.
3. Aufgrund der Sonderknoten sind die Blätter immer noch schwarz.
4. Da  $T_1$  und  $T_2$  RBTs waren muss nur die Situation um  $v$  betrachtet werden.  $v$  hat in jedem Fall schwarze Kinder. Gilt  $bh(T_1) \neq bh(T_2)$  könnte  $v$  jedoch einen roten Vater  $y$  haben.
5. Die Schwarz-Höhe von  $v$  ist korrekt gesetzt. Existiert  $y$  so hat sich seine Schwarz-Höhe nicht verändert, denn  $v$  ist rot. Bei keinem anderen Knoten hat sich bezüglich bzgl. der Schwarz-Höhe etwas geändert.

Wir sind also in der Situation dass nur entweder Eigenschaft zwei oder vier verletzt sein kann. Wenn Eigenschaft vier verletzt ist dann nur an Knoten  $v$ . Das ist genau die Situation für die einfügenFixup entworfen wurde.

In Schritt zwei wird also einfügenFixup mit Parameter  $v$  aufgerufen und die Wurzel des resultierenden RBT zurückgegeben.

**Laufzeit** Sei  $n_1$  die Anzahl der Knoten von  $T_1$ , sei  $n_2$  die Anzahl der Knoten von  $T_2$  und  $n = n_1 + n_2$ . Der Tango-Baum fordert von seiner Hilfsstruktur eine Laufzeit von  $O(\log n)$  für die eben vorgestellte Operation. Das Ablaufen eines Pfades in  $T_1$  oder  $T_2$  zum Finden von  $x$  liegt in  $O(\log(n))$ .  $v$  erzeugen und in die Struktur einzubinden benötigt konstante Zeit und einfügenFixup benötigt  $O(\log(n))$  Zeit.

$$O(\log(n)) + O(1) + O(\log(n)) = O(\log(n))$$

Die Vorgabe des Tango-Baumes wird also eingehalten.

Für das nächste Kapitel wird noch eine genauere Betrachtung der Laufzeit benötigt. Es sei  $d = |bh(T_1) - bh(T_2)|$ . Die Suche nach  $x$  endet spätestens nach dem ein Pfad der Länge  $2d + 1$  betrachtet wurde. Dabei steht die 1 für den Zugriff auf  $x$  selbst. Zu jedem schwarzen Knoten über  $x$  könnte noch ein roter kommen.

Nun wird noch auf die Anzahl der Iterationen innerhalb von *einfügenFixup* eingegangen. Sei  $v$  der Parameter von *einfügenFixup*. Sei  $w$  die Wurzel von  $T$ .  $v$  ist ein roter Knoten mit  $bh(v) = bh(w) - d + 1$  und liegt in Ebene  $2d + 2$  oder höher. Deshalb führt *einfügenFixup* maximal  $d + 1$  Iterationen durch.

### 4.3 Tango-Baum konformes aufteilen

Auch *aufteilen* (*RBT T, Schlüssel k*) wird so vorgestellt, wie es als Hilfsstruktur für einen Tango-Baum benötigt wird. Voraussetzung an die Parameter ist, dass  $k$  in der Schlüsselmenge  $K$  von  $T$  vorhanden ist. Zurückgegeben wird eine Referenz auf den Knoten  $v$  mit Schlüssel  $k$ . Linkes Kind von  $v$  ist die Wurzel eines RBT  $T_L$  mit Schlüsselmenge  $K_L$ , wobei gilt  $K_L = \{i \mid i \in K \wedge i < k\}$ . Rechtes Kind von  $v$  ist die Wurzel eines RBT  $T_R$  mit Schlüsselmenge  $K_R$ , wobei gilt  $K_R = \{i \mid i \in K \wedge i > k\}$ . *aufteilen* gibt also in den meisten Fällen keinen RBT zurück. Die Operation setzt zunächst  $T_L$  auf den linken Teilbaum von  $v$  und  $T_R$  auf den rechten Teilbaum von  $v$ . Eventuell müssen die Wurzeln schwarz gefärbt werden. Sei  $(v_0, v_1, \dots, v_m)$  ein Pfad von  $v$  zu der Wurzel von  $T$ , mit  $\forall i \in 0, 1, \dots, m : v_{i+1}$  ist Vater von  $v_i$ . Es wird sich nun bei  $v_m$  startend Knoten für Knoten in dem Pfad nach oben gearbeitet. Ist der Schlüssel eines Knotens kleiner als  $k$ , so wird dieser Schlüssel und der linke Teilbaum des Knotens zu  $T_L$  hinzugefügt. Dies übernimmt unsere *vereinigen* Operation. Ist der Schlüssel größer als  $k$ , so wird dieser Schlüssel und der rechte Teilbaum des Knotens zu  $T_R$  hinzugefügt. Folgende Aufzählung beschreibt den Vorgang genauer.

1. Verwende die *suchen* Operation um den Knoten  $v_0$  mit Schlüssel  $k$  zu finden.
2. Setze den linken Teilbaum von  $v_0$  als  $T_L$ , den rechten als  $T_R$ . Löse beide Teilbäume aus  $T$  heraus.
3. Färbe die Wurzeln von  $T_L$  und  $T_R$  schwarz.
4.  $\forall i \in \{1, 2, \dots, m\}$  aufsteigend sortiert. Ist der Schlüssel  $k_i$  von  $v_i$  kleiner als  $k$ , vereinige  $T_L$  mit dem aus  $T$  herausgelösten linken Teilbaum von  $v_i$ , mit  $k_i$  als dritten Parameter. Ansonsten vereinige  $T_R$  mit dem aus  $T$  herausgelösten rechten Teilbaum von  $v_i$ , mit  $k_i$  als dritten Parameter. Evtl. muss die Wurzel des herausgelösten Teilbaumes vor dem *vereinigen* schwarz gefärbt werden.
5. Füge  $T_R$  rechts an  $v$  an,  $T_L$  links.
6. Gib  $v$  zurück.

Das  $T_L$  und  $T_R$  die gewünschten RBTs sind wird leichter erkannt, wenn man sich den Pfad von oben nach unten, also  $(v_m, \dots, v_0)$ , betrachtet.  $v_m$  und einer der beiden Teilbäume wird korrekt zugeordnet. Die Wurzel des anderen Teilbaumes von  $v_m$  ist  $v_{m-1}$ . Alle Schlüssel die nicht im Teilbaum mit Wurzel  $v_{m-1}$  liegen sind somit korrekt zugeordnet. Diese Betrachtung iteriert bis man auf  $v$  trifft. Die Schlüssel im Teilbaum mit Wurzel  $v$  werden korrekt zugeordnet.

**Laufzeit** Der Tango-Baum fordert eine Laufzeit von  $O(\log(n))$  von seiner Hilfsstruktur für *aufteilen*, mit  $n$  ist die Anzahl der Knoten. Punkt 1 kostet  $O(\log(n))$ . Durchführen von Punkt 2,3,5 und 6 kostet  $O(1)$ . Punkt 4 führt  $O(\log(n))$  Aufrufe von *vereinigen* durch. Das ergibt  $O(\log(n) \log(n))$ , was dann auch eine obere Schranke für die Gesamlaufzeit ist. Diese Schranke ist für unseren Einsatzzweck jedoch zu hoch.

Deshalb wird Punkt 4 nun genauer betrachtet, speziell die Konstruktion von  $T_L$ . Es sei  $l$  die Anzahl der Aufrufe von *vereinigen* nach denen  $T_L$  neu gesetzt wird. Sei  $\{t_1, t_2, \dots, t_l\}$  die Menge der aus  $T$  herausgelösten Teilbäume die für die  $l$  Aufrufe verwendet wurden, wobei  $t_1$  zum ersten Aufruf gehört,  $t_2$  zum zweiten usw.. Mit  $T_{Li}$  wird der Zustand von  $T_L$  bezeichnet nach dem  $t_i$  Parameter von *vereinigen* war.  $T_{L0}$  steht für den Zustand vor dem ersten Aufruf.

Dass die zweite Ungleichung gilt, erkennt man direkt.

$$bh(T_{Li}) \leq bh(v_{i+1}) \leq bh(t_{i+1}) + 1 \quad (1)$$

Die Erste wird nun durch Induktion gezeigt:

$T_{L0}$  war der linke oder rechte Teilbaum von  $v_1$ , der Induktionsanfang ist somit gemacht.

$bh(T_{Li})$  mit  $i > 0$  entsteht durch  $vereinigen(T_{Li-1}, k_{vi}, t_i)$ , mit  $k_{vi}$  ist der Schlüssel von  $v_i$ .

Fall  $v_i$  ist rot:

$$\begin{aligned} bh(t_i) &< bh(v_i) \wedge bh(T_{Li-1}) \stackrel{IA}{\leq} bh(v_i) \\ \Rightarrow bh(T_{Li}) &\leq bh(v_i) \leq bh(v_{i+1}) \end{aligned}$$

Fall  $v_i$  ist schwarz:

$$\begin{aligned} bh(t_i) &\leq bh(v_i) \wedge bh(T_{Li-1}) \stackrel{IA}{\leq} bh(v_i) \\ \Rightarrow bh(T_{Li}) &\leq bh(v_i) + 1 = bh(v_{i+1}) \end{aligned}$$

Dadurch gilt

$$\sum_{i=1}^l |bh(t_i) - bh(T_{Li-1})| = O(\log(n)) \quad (2)$$

denn

$$\begin{aligned} \sum_{i=1}^l |bh(t_i) - bh(T_{Li-1})| &\stackrel{l}{\leq} \sum_{i=1}^l (bh(t_i) - bh(T_{Li-1}) + 2) \\ &\leq \sum_{i=1}^l (bh(t_i) - bh(t_{i-1}) + 2) = bh(t_l) - bh(t_0) + 2 \cdot l = O(\log(n)) \end{aligned}$$

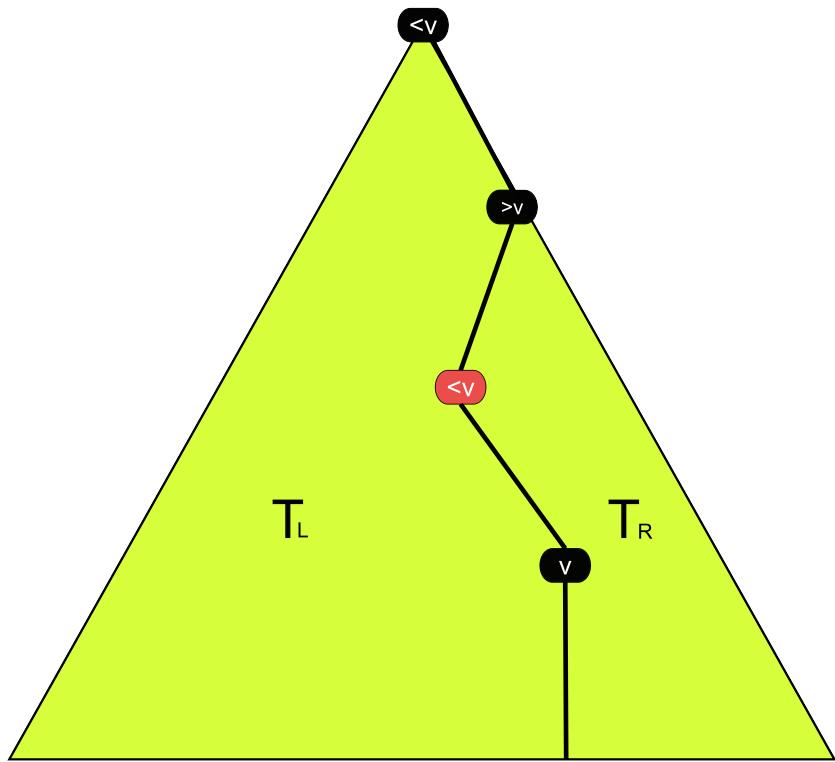
Der Gesamtaufwand für das Suchen von  $v$  in allen  $l$  Aufrufen berechnet sich mit:

$$\sum_{i=1}^l 2|bh(t_i) - bh(T_{Li-1})| + 1 = 2 \left( \sum_{i=1}^l |bh(t_i) - bh(T_{Li-1})| \right) + l = O(\log(n))$$

Für die Anzahl der Iterationen von *einfügenFixup* in allen  $l$  Aufrufen gilt:

$$\sum_{i=1}^l |bh(t_i) - bh(T_{Li-1})| + 1 = \sum_{i=1}^l (|bh(t_i) - bh(T_{Li-1})|) + l = O(\log(n))$$

Die Kosten innerhalb einer Iteration sind konstant, damit ist  $O(\log(n))$  eine obere Schranke für die Gesamtkosten zum konstruieren von  $T_L$ . Für  $T_R$  gilt analog das Gleiche. Für die Gesamtkosten von *vereinigen* innerhalb *aufteilen* gilt  $O(\log(n))$ , denn neben den Kosten für die Suche und *einfügenFixup* fallen pro Aufruf lediglich konstante Kosten an. Damit ist  $O(\log(n))$  eine obere Schranke für die Kosten von Punkt 4 und somit auch für *aufteilen*. Der hier vorgestellte RBT hält also die Anforderung des Tango-Baumes bezüglich der Laufzeit von *vereinigen* ein.



**Abbildung 22:** Beispielhaftes aufteilen eines RBT mit Parameter  $v$

## Literatur

- [1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.