6 Binäre Suchbäume

6.1 Was sind binäre Suchbäume?

Suchbäume (*binary search trees*) sind Datenstrukturen, die viele Operationen auf dynamischen Mengen effizient unterstützen. Beispiele sind:

- *Search(K, k)*: Suche ein Element x von K mit Schlüssel k
- *Insert(K, x)*: Füge das Element x in K ein
- *Delete(K, x)*: Entferne das Element x aus K
- *Minimum(K)*: Suche das (ein) Element von K mit minimalem Schlüssel
- *Maximum(K)*: Suche das (ein) Element von K mit maximalem Schlüssel
- *Successor(K, x)*: Suche das Element, dessen Schlüssel in der Ordnung von K dem Schlüssel von x folgt (der nächstgrößere Schlüssel) dem Schlüssel von x vorangeht (der nächstkleinere Schlüssel)

Ein Suchbaum kann also als zum Beispiel als *Dictionary* oder *Priority Queue* verwendet werden.
Ein binärer Suchbaum ist – wie der Name schon sagt – ein binärer Baum. Ein solcher Baum kann durch eine Pointer-Datenstruktur repräsentiert werden, in der jeder Knoten ein Objekt darstellt. Neben dem Schlüssel (*key*) hat jeder Knoten zwei Pointer *left_child* und *right_child*, die auf die Nachfolgerknoten zeigen, sowie einen Pointer *parent*, der auf den direkten Vorgänger des Knotens verweist. Gibt es keine *child*- oder *parent*-Knoten, ist der entsprechende Pointer auf *NULL* gesetzt. Abbildung 6.1 zeigt die typische Struktur eines Baumknotens, die im folgenden in der Klasse `TreeNode` implementiert ist. Abbildung 6.2 zeigt eine Beispielstruktur eines binären Baums. In der vereinfachten Abbildung 6.3(a) der Baumstruktur aus Abbildung 6.2 sind die Blätter (*leafs*) des Baums, also die Knoten ohne Kinder, rot unterlegt. In einem binären Suchbaum werden die Schlüssel wie folgt gespeichert (*binary-search-tree property* oder *Ordnungseigenschaft*):

> Sei x ein Knoten des Baums. Für jeden Knoten y im *linken* Unterbaum von x gilt $y.key \leq x.key$. Für jeden Knoten y im *rechten* Unterbaum von x gilt $y.key \geq x.key$.

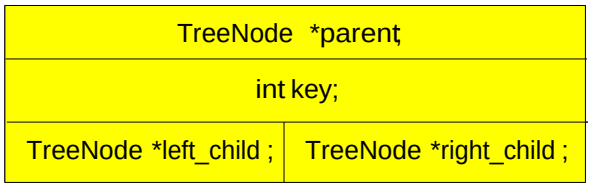


Abb. 6.1: Knoten in einem binären Baum

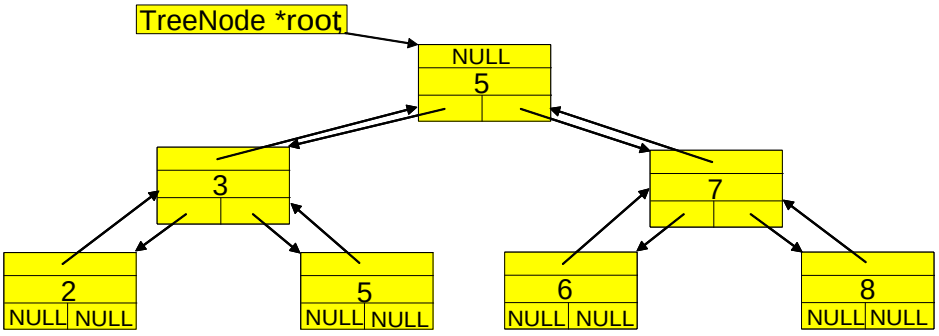


Abb. 6.2: Beispiel für einen binären Baum

Für den Baum in Abbildung 6.3(a) ist der Knoten 5 die Wurzel, die Knoten 2, 3 und 5 befinden sich entsprechend im linken Unterbaum, die Knoten 6, 7 und 8 im rechten Unterbaum. Knoten 3 ist zum Beispiel nicht kleiner als Knoten 2 in seinem linken Unterbaum und nicht größer als Knoten 5 in seinem rechten Unterbaum.

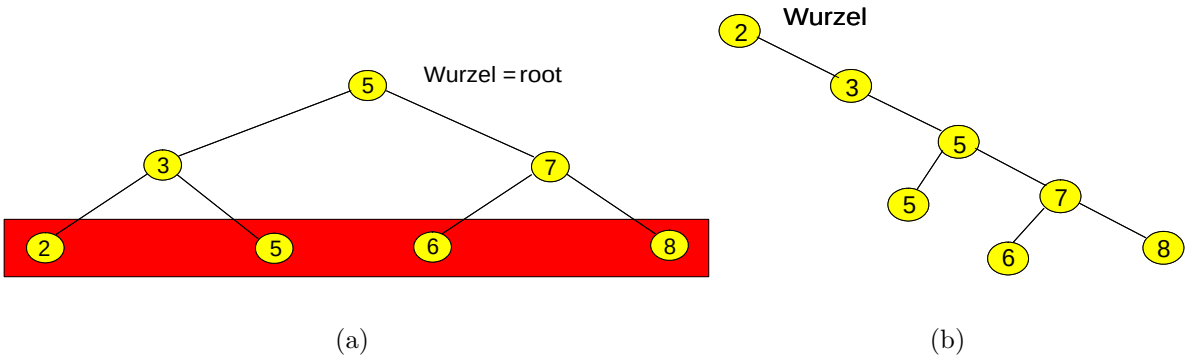
Definition 6.1. Die *Höhe* eines Knoten v in einem Baum ist gleich der Zahl der Kanten auf dem längsten Weg vom Knoten zu einem Blatt im Unterbaum des Knotens. Die *Höhe eines Baums* ist gleich der Höhe seiner Wurzel.

6.2 Algorithmen auf binären Suchbäumen

6.2.1 Tree walk

Die Ordnungseigenschaft ermöglichst die Ausgabe aller im Baum gespeicherten Schlüssel in sortierter Reihenfolge. Der entsprechende rekursive Algorithmus wird als *tree_walk* bezeichnet:

```
void tree_walk(TreeNode *x)
{
  if (x != NULL)
  {
    tree_walk(x->left_child);
    cout << x->key << endl;
    tree_walk(x->right_child);
```



(a) (b)

Abb. 6.3: 6.3(a) Der binäre Baum aus Abb. 6.2 in vereinfachter Darstellung. 6.3(b) Ein weniger effizienter binärer Suchbaum der Höhe 4, der die gleichen Knoten enthält.

```
    }
}
```

Ein Aufruf der Funktion *tree_walk(x)* für die Bäume in Abbildung 6.3 führt zur Ausgabe 2, 3, 5, 5, 6, 7, 8. Der Algorithmus benötigt $\Theta(n)$ Zeit, um einen binären Baum mit n Knoten zu durchlaufen, da die Funktion genau zweimal rekursiv für jeden Knoten des Baums aufgerufen wird: Einmal für das linke Kind des Knotens und einmal für das rechte Kind.

Satz 6.1. *Starten wir die Funktion tree_walk mit der Wurzel x eines Baums mit n Knoten, so benötigt tree_walk $\Theta(n)$ Zeit.*

Beweis. Sei $T(n)$ die Laufzeit für einen Baum mit n Knoten. Für einen leeren Baum benötigt *tree_walk* konstante Zeit $T(0) = c$.
Für $n > 0$ nehmen wir an, daß *tree_walk* für einen Knoten x mit einem linken Unterbaum der Größe k und einen rechten Unterbaum der Größe $n - k - 1$ (rekursiv) aufgerufen wird:

$$T(n) = T(k) + T(n - k = 1) + d \quad d \text{ ist eine Konstante}$$

Wir wenden nun die Substitutionsmethode an und zeigen, daß die Laufzeit von *tree_walk* von der folgenden Form ist:

$T(n) = (c + d)n + c$, wobei c und d die (oben definierten) Konstanten sind.

Induktionsstart: $n = 0 \quad T(0) = c = (c + d)0 + c$
Induktionsannahme: Wir nehmen an, daß die obige Aussage für alle ganzen Zahlen kleiner als n gilt.
Induktionsschluß:

$$T(n) = T(k) + T(n - k - 1) + d$$
$$T(n) = (c + d)k + c + (c + d)(n - k - 1) + c + d$$
$$T(n) = (c + d)n + c$$

■

6.2.2 Suche auf binären Bäumen

Eine gängige Operation in einem binären Suchbaum ist die Suche nach einem Schlüssel k. Die Funktion *tree_search* wird mit einem Pointer auf den Wurzelknoten des betreffenden Baums sowie mit dem Schlüssel k aufgerufen. Wird k gefunden, liefert *tree_search* einen Pointer auf den gefundenen Knoten zurück, andernfalls *NULL*.

```
TreeNode* tree_search(TreeNode *x, int k)
{
  TreeNode *result = NULL;
  if (x != NULL)
  {
    if (x->key == k)     result = x;
    else if(k < x->key) result = tree_search(x->left_child , k);
    else                 result = tree_search(x->right_child ,k);
  }
  return (result);
}
```

Die Funktion beginnt die Suche an der Wurzel und läuft im Baum nach unten. Für jeden Knoten x wird der gesuchte Schlüssel k mit dem Schlüssel $x \to k$ verglichen. Sind die Schlüssel gleich, bricht die Suche ab. Ist k kleiner als $x \to k$, fährt die Suche im linken Unterbaum von x fort, da die Ordnungseigenschaft impliziert, daß k dann nicht im rechten Unterbaum gespeichert sein kann. Entsprechend wird die Suche im rechten Unterbaum fortgesetzt, wenn k größer ist als $x \to k$. Die auf diese Weise durchlaufenen Knoten formen einen Pfad von der Wurzel zu einem Blatt. Die Laufzeit von *tree_search* ist also $\mathcal{O}(h)$, wobei h die Höhe des Baums angibt.
Die gleiche Funktion kann auch iterativ implementiert werden, indem die Rekursion in Form einer *while*-Schleife nachgebildet wird. Auf den meisten Maschinen ist diese Version effizienter als die rekursive Implementierung.

```
TreeNode* iterative_tree_search(TreeNode *x, int k)
{
  while (x != NULL && x->key != k)
  {
    if(k < x->key) x = x->left_child;
    else           x = x->right_child;
    }
  return x;
}
```

Die Funktionen *tree_search*, *iterative_tree_search*, *tree_predecessor*, *tree_minimum* und *tree_maximum* sind sehr ähnlich zu den angegebenen Implementierungen.

Satz 6.2. *Die Funktionen tree_search, iterative_tree_search, tree_predecessor, tree_minimum und tree_maximum haben eine Laufzeit von $\mathcal{O}(h)$, wobei h die Höhe des binären Baums ist.*

6.2.3 Einfügen und Löschen

Einfügen und Löschen bewirken eine Änderung der Baumstruktur. Die Datenstruktur muß entsprechend geändert werden, ohne dabei gegen die Ordnungseigenschaft zu verstoßen. Wir werden sehen, daß das Einfügen relativ einfach ist, während das Löschen deutlich komplizierter ist.

Einfügen

Die Funktion *tree_insert* fügt einen neuen Knoten z mit dem Schlüssel k in den Baum ein. Wir nehmen an, daß $z \rightarrow$*left_child* und $z \rightarrow$*right_child* schon auf *NULL* gesetzt wurden.

```
void tree_insert(Tree *T, TreeNode *z)
{
  TreeNode *x=T->root; // Pointer zum Iterieren ueber den Baum
  TreeNode *p=NULL;    // Pointer auf Vater von Knoten x

  while(x!=NULL) {      // Durchmustern des Baums
    p = x;              // p speichert den Vater von x
    if(x->key < z->key)
      x=x->right_child;    // Die Suche geht rechts weiter
    else
      x=x->left_child;     // Die Suche geht links weiter
  }
  z->parent = p;
  if(p==NULL) T->root  =z;  // z ist die neue Wurzel
  else if(z->key < p->key)
         p->left_child =z;  // Links von p einfuegen
       else
         p->right_child=z;  // Rechts von p einfuegen
}
```

Wie die Such-Funktionen hat *tree_insert* eine Laufzeit von $\mathcal{O}(h)$ (h: Höhe des Baums).

Löschen

Beim Löschen eines Baumknotens ist eine Fallunterscheidung erforderlich:

1. Der zu löschende Knoten z hat keine Kinder. Dieser Fall ist der einfachste: Der entsprechende Zeiger des Vaterknotens von z wird auf *NULL* gesetzt, z wird gelöscht. Die Laufzeit ist $\mathcal{O}(1)$. Fall 1 ist in Abbildung 6.4(a) dargestellt.

2. Der zu löschende Knoten z hat genau ein Kind, das dann mit dem Vater von z verbunden wird (*splice out*, Abb. 6.4(b)). Diese Operation hat eine Laufzeit $\mathcal{O}(1)$.

3. Der zu löschende Knoten hat zwei Kinder. Dieser Fall ist der aufwendigste: Wir wenden das *splice out* aus Schritt 2 auf den Successor von Knoten z an:

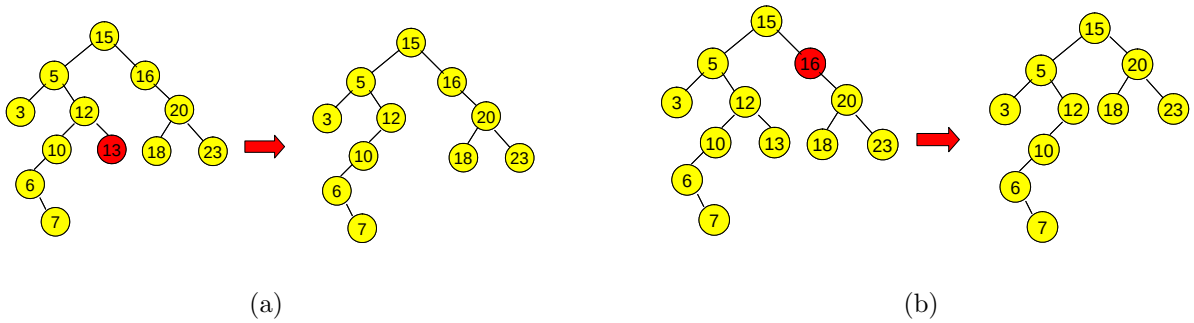


Abb. 6.4: Löschen eines Knotens in einem binären Baum - Fall 1 und 2

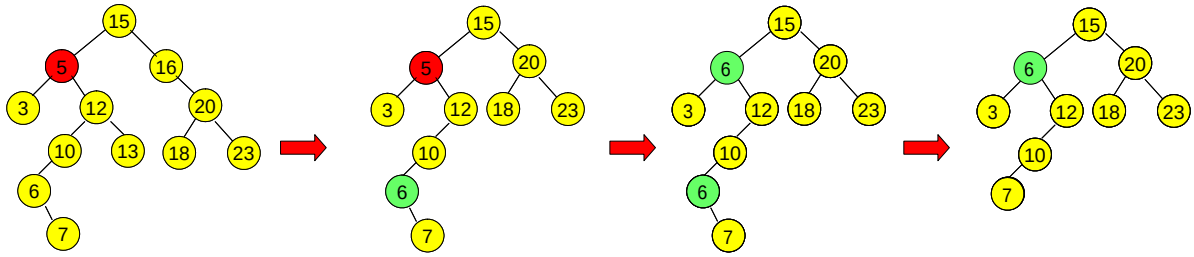


Abb. 6.5: Löschen eines Knotens in einem binären Baum - Fall 3

a) Suche den Successor von z in seinem rechten Unterbaum

b) Übergebe Daten (den Schlüssel) des Successors an z

c) Lösche z mit der Prozedur für Knoten mit einem Kind

Dieser Fall ist in Abb. 6.5 gezeigt. Die Laufzeit ist $\mathcal{O}(h)$ im *worst case*.

Satz 6.3. *In einem binären Suchbaum der Höhe h kann in Zeit $\mathcal{O}(h)$ ein Element eingefügt oder gelöscht werden.*

6.3 Rot-Schwarz-Bäume

Im vorhergehenden Abschnitt haben wir gelernt, daß die Standard-Operationen dynamischer Mengen auf binären Bäumen in Zeit proportional zur Höhe des Baums ausgeführt werden können.
Deshalb sollte man versuchen, die Höhe eines Baums so klein wie möglich zu halten, und man sollte dafür sorgen, daß der Baum immer gut *balanciert* bleibt.
Ein *Rot-Schwarz-Baum* ist ein binärer Suchbaum, der pro Knoten über ein zusätzliches Blatt verfügt, mit dem die Farbe des Knotens (rot oder schwarz) gespeichert wird. Wir verschwenden hier Speicherplatz und verwenden im folgenden eine ganze Zahl `int color` mit `RED=0` und `BLACK=1`.
Jeder Knoten (*TreeNode*) enthält die folgenden Komponenten (Daten):

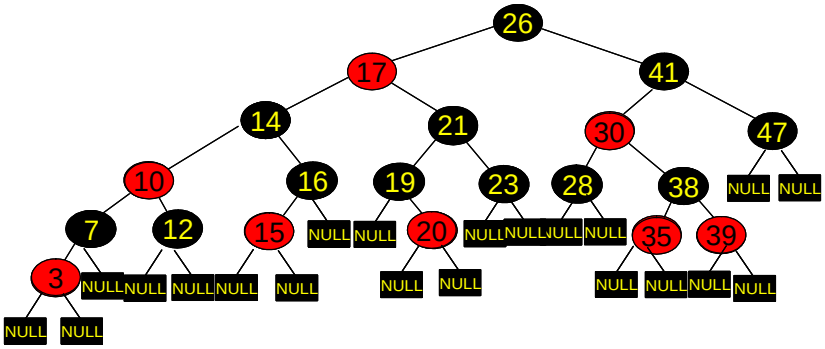


Abb. 6.6: Rot-Schwarz-Baum

```
int key;
TreeNode *parent;
TreeNode *left_child;
TreeNode *right_child;
int color;
```

Definition 6.2. 1. Jeder Knoten ist entweder rot oder schwarz.

2. Die Wurzel ist schwarz.

3. Jedes Blatt [NULL] ist schwarz.

4. Wenn ein Knoten rot ist, dann sind seine Kinder schwarz.

5. Für jeden Knoten gilt: Alle Pfade vom Knoten zu Blättern im Unterbaum enthalten die gleichen Zahl von schwarzen Knoten.

In Abbildung 6.6 ist ein Beispiel für einen Rot-Schwarz-Baum dargestellt. Die NULL-Zeiger können alle auf die Adresse eines besonderen Knoten `TNULL` umgesetzt werden (Abb. 6.7). Mit diesem Knoten können wir wie mit jedem anderen Knoten arbeiten:
TNULL−>color=BLACK;
`parent`, `left_child`, `key` und `right_child` können beliebig gesetzt werden.
Da die (inneren) Knoten mit den Schlüsseln im Mittelpunkt unseres Interesses stehen, lassen wir in den folgenden Abbildungen die Zeiger auf `TNULL` weg.

Definition 6.3. Die *schwarze Höhe* $sh(x)$ *eines Knoten* x ist die Zahl der schwarzen Knoten von x zu einem Blatt, wobei der Knoten x nicht mitgezählt wird.

Lemma 6.1. *Ein Rot-Schwarz-Baum mit* n *inneren Knoten hat höchstens die Höhe* $2log(n+1)$.

Beweis. Wir zeigen zunächst, daß jeder Unterbaum mit Wurzel x mindestens $2^{sh(x)} - 1$ innere Knoten enthält. Dies zeigen wir mittels vollständiger Induktion über die Höhe von x.

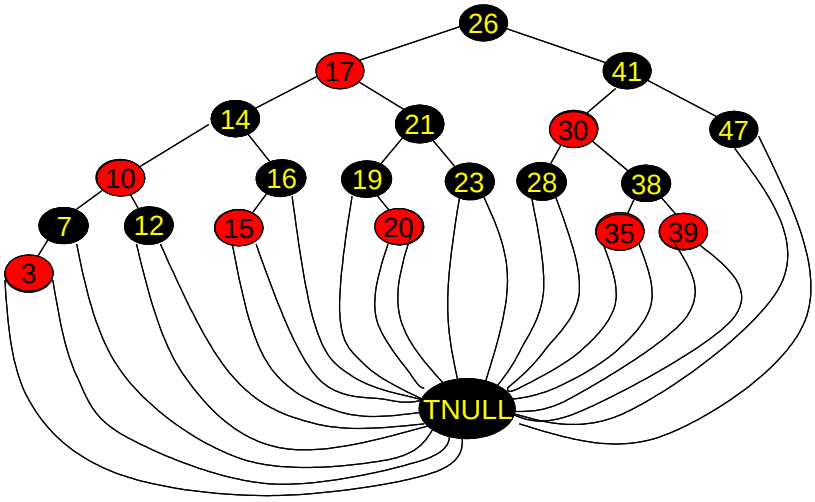


Abb. 6.7: Der Knoten `TNULL`

Höhe(x)=0: x ist `TNULL` und enthält daher $2^{sh(x)} - 1 = 2^0 - 1 = 0$ innere Knoten.
Wir betrachten nun für den Induktionsschritt einen Knoten x, der eine positive Höhe hat und der ein innerer Knoten mit zwei Kindern ist. Jedes Kind hat entweder eine schwarze Höhe von $sh(x)$ oder $sh(x) - 1$, je nachdem, ob es ein roter oder ein schwarzer Knoten ist. Da die Kinder von x eine um eins kleinere Höhe als x haben, folgt aus der Induktionsannahme, daß die Zahl der inneren Knoten mindestens
$(2^{sh(x)-1} - 1) + (2^{sh(x)-1} - 1) + 1 = 2^{sh(x)} - 1$ ist.
Sei h die Höhe des Baums. Mindestens die Hälfte der Knoten von der Wurzel zu irgend einem Blatt sind schwarz. Hieraus folgt:
$sh(root) \geq h/2 \rightarrow n \geq 2^{h/2} - 1 \rightarrow 2log(n+1) \geq h$ ■

Aus dem obigen Lemma folgt direkt:

> Die Suche nach einem Element, dem Minimum, dem Maximum, dem Successor oder Predecessor eines Elements kann in einem Rot-Schwarz-Baum mit n Knoten in Zeit $\mathcal{O}(logn)$ ausgeführt werden.

Auch die im vorhergehenden Abschnitt beschriebenen Funktionen *tree_insert* und *tree_delete* haben eine Laufzeit von $\mathcal{O}(logn)$. Nach der entsprechenden Operation liegt jedoch eventuell kein Rot-Schwarz-Baum mehr vor.
Wir werden im folgenden Abschnitt lernen, daß man mittels sogenannter Rotationen von Teilbäumen und mittels Umfärben der Knoten die Eigenschaften des Rot-Schwarz-Baums wiederherstellen kann.

6.3.1 Rotationen

Um nach Einfügungen und Löschungen im Baum die Eigenschaften des Rot-Schwarz-Baums wiederherzustellen, müssen gewisse Knoten umgefärbt werden, und manche Zeiger müssen geändert werden. Diese sogenannten *Rotationen* sind lokale Änderungen der

Baumstruktur, die die Erhaltung der Ordnungseigenschaft garantieren. Abbildung 6.8 zeigt die Rotation von Knoten x im Baum T nach links und rechts. Wir nehmen an, daß das rechte Kind y von x nicht `NULL` (`TNULL`) ist. Die Links-Rotation dreht den Baum um die Kante von x nach y herum. Die neue Wurzel des Unterbaums ist damit y, x wird zum linken Kind von y, das bisherige linke Kind von y wird rechtes Kind von x. Die Rechts-Rotation verläuft symmetrisch. Beide Operationen haben eine Laufzeit $\mathcal{O}(1)$. Während der Rotation werden ausschließlich Pointer verändert, die übrigen Knotenfelder bleiben unberührt.

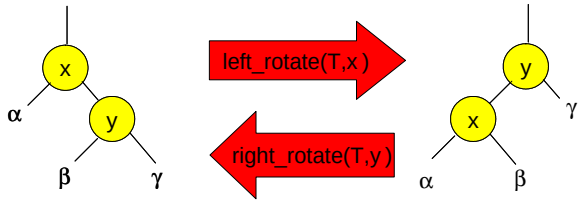


Abb. 6.8: Rotation im Baum T

6.3.2 Einfügen in Rot-Schwarz-Bäume

Wir verwenden eine leicht modifizierte Version von *tree_insert*:

```
void RB_insert(Tree T, TreeNode *z) {
  TreeNode *x=T->root;      // Zeiger zum Durchwandern des Baums
  TreeNode *p=TNULL;        // Zeiger auf Vater
  while(x!=TNULL) {         // Durchmustern des Baums
    p=x;                    // Man merke sich den Vater von x
    if(x->key<z->key)
      x=x->right_child;     // Die Suche geht rechts weiter
    else
      x=x->left_child;      // Die Suche geht links weiter
  }
  z->parent=p;
  if(p==TNULL) T->root=z; // z ist die neue Wurzel
  else
    if(z->key<p->key)
      p->left_child=z;      // Links von p einfuegen
    else
      p->right_child=z;     // Rechts von p einfuegen
    z->color=RED;           // Der neue Knoten z ist rot
    z->left_child=z->right_child=TNULL; // z hat keine Kinder
    RB_insert_fixup(T,z); // Hier erfolgt die Reparatur
                          // des Rot-Schwarz-Baums
}
```

Welche Eigenschaften können durch das Einfügen verletzt werden?

1. Jeder Knoten ist entweder rot oder schwarz.
2. Die Wurzel ist schwarz.
3. Jedes Blatt [NULL] ist schwarz.
4. Wenn ein Knoten rot ist, dann sind seine Kinder schwarz.
5. Für jeden Knoten gilt: Alle Pfade vom Knoten zu Blättern im Unterbaum enthalten die gleichen Zahl von schwarzen Knoten.

Die Eigenschaften 2 und 4 können durch Einfüge-Operationen verletzt werden. Eigenschaft 2 können wir sehr einfach garantieren, indem wir am Ende der Reparatur-Prozedur *RB_insert_fixup* der Wurzel immer die Farbe *schwarz* zuordnen:
T->root->color=BLACK.
Da z nur schwarze Blätter [TNULL] als Kinder hat, kann Eigenschaft 4 nur verletzt werden, wenn der Vater $z \rightarrow parent$ von z auch rot ist.
Beide möglichen Verletzungen entstehen also dadurch, daß z rot gefärbt wird. Wir können nun drei Fälle unterscheiden, für die wir jeweils die entsprechenden Teile der Prozedur *RB_insert_fixup* angeben.

Fall 1: Der Onkel y von z ist auch rot (Abb. 6.9).

```
<WHILE-LOOP>:
while(z->parent!=NULL && z->parent->color==RED) {
  if(z->parent==z->parent->parent->left_child) {
    y=z->parent->parent->right_child;
  <FALL 1>:
    if(y->color==RED) {
      y->color=BLACK;
      z->parent->color=BLACK;
      z->parent->parent->color==RED;
      z=z->parent->parent;
    }
    else { <FALL 2>}
  }
else {...} // symmetrisch zur ersten if-Schleife
}
```

Fall 2: Der Onkel y von z ist schwarz und z ist ein *rechtes* Kind (Abb. 6.10).

```
<FALL 2>:
if(z==z->parent->right_child) {
  z=z->parent;
  left_rotate(T,z);
}
```

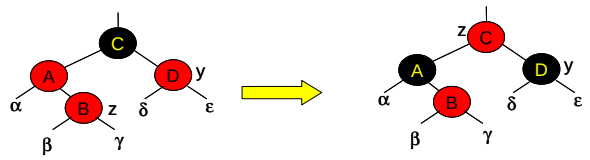


Abb. 6.9: Reparatur-Prozedur Rot-Schwarz-Baum, Fall 1

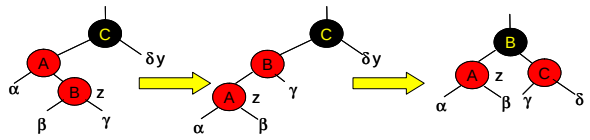


Abb. 6.10: Reparatur-Prozedur Rot-Schwarz-Baum, Fall 2

Fall 3: Der Onkel y von z ist schwarz und z ist ein *linkes* Kind.

```
<FALL 3>:
else {
  z->parent->color=BLACK;
  z->parent->parent->color=RED;
  right_rotate(T,z->parent);
}
```

Die Prozedur *RB_insert_fixup* hat damit folgende Form:

```
void RB_insert_fixup(Tree T, TreeNode *z) {
  TreeNode *y;      // Hilfspointer fuer Onkel

  <WHILE-LOOP>

  T->root->color=BLACK;
}
```

Die Laufzeit von *RB_insert* ist $\mathcal{O}(logn)$.

6.3.3 Löschen in Rot-Schwarz-Bäumen

Das Löschen eines Knotens aus einem Rot-Schwarz-Baum ist sehr ähnlich zum Löschen in einfachen binären Suchbäumen. Die wesentliche Änderung besteht darin, daß man am Ende der Prozedur wieder eine Reparatur-Prozedur *RB_delete_fixup* startet. Falls ein roter Knoten gelöscht wird, bleiben die Eigenschaften des Rot-Schwarz-Baums erhalten. Nur beim Entfernen eines schwarzen Knotens können diese Eigenschaften verletzt werden. Drei Probleme können dabei auftreten:

- P1: Der entfernte Knoten y war die Wurzel, und ein rotes Kind x von y ist nach dem Löschen von y die neue Wurzel. Lösung: Färbe x schwarz

- P2: Falls das einzige Kind x des gelöschten Knotens y und der Vater von y rot sind, dann ist nach dem Löschen die Bedingung 4 verletzt.

- P3: Pfade, die y enthalten haben, haben nun einen schwarzen Knoten weniger. Deshalb ist die Bedingung 5 für alle Vorfahren von y verletzt.

Das erste Problem ist einfach zu lösen. Wir betrachten im folgenden, wie P2 und P3 behandelt werden können.

Sei x das Kind von y.

[a] Ist x rot, so färben wir x schwarz, und Bedingung 5 ist wieder erfüllt. Dies ist auch der einzige Fall, in dem zwei rote Knoten nach dem Löschen aufeinander folgen können, und somit ist auch Bedingung 4 wieder erfüllt.

[b] Ist x schwarz und könnten wir x einen weiteren „Schwarz-Wert“ zuordnen (also insgesamt zwei), so wäre die Bedingung 5 wiederhergestellt. Da ein Rot-Schwarz-Baum jedoch keine Zähler für Farbeinheiten besitzt, müssen wir weitere Reparaturen durchführen, um jeweils den Knoten x mit dem künstlichen Schwarz-Wert von „zwei“ zu reparieren. Dabei hangeln wir (mit x) solange den Pfad zur Wurzel entlang, bis die Verletzung von Bedingung 5 eliminiert ist. Der Knoten w sei immer der Bruder von x. Wir unterscheiden vier Fälle und beginnen mit dem letzten Fall. In den zugehörigen Abbildungen können die blau gezeichneten Knoten entweder rot *oder* schwarz gefärbt sein.

Fall 4: w ist schwarz, und sein rechtes Kind ist rot: Abbildung 6.11. D bekommt die Farbe von B. Wir führen eine Links-Rotation auf $x \rightarrow parent$ aus und entfernen die „zweite“ Schwarzmarkierung von x.

Fall 2: w und beide Kinder von w sind schwarz: Abbildung 6.12. Falls B rot ist, färben wir B schwarz und D rot und sind fertig.

Fall 3: w ist schwarz, und sein linkes Kind ist rot: Abbildung 6.13. Wir können die Farben von w und seinem linken Kind ändern und anschließend eine Rechts-Rotation auf w ausführen, ohne eine der Rot-Schwarz-Eigenschaften zu verletzen. Der neue Bruder w von x ist jetzt ein schwarzer Knoten mit einem roten Kind – der rechte Teil der Abbildung zeigt, daß wir damit bei Fall 4 landen.

Fall 1: w ist rot: Abbildung 6.14. Da w schwarze Kinder haben muß, können wir die Farbe von w und $x \rightarrow parent$ ändern und dann eine Links-Rotation auf $x \rightarrow parent$ ausführen, ohne die Rot-Schwarz-Eigenschaften zu verletzen. Der neue Bruder von x (eines von w's Kindern vor der Rotation) ist jetzt schwarz. Falls wir von Fall 1 aus in Fall 2 gelandet sind, ist der neue Knoten x rot gefärbt, und wir färben ihn schwarz. Damit ist die Reparatur beendet.

Zusammenfassung der Reparatur-Prozedur:

1. Falls wir die Möglichkeit haben, den Fehler direkt zu eliminieren (Fall 4), führen wir die entsprechende Transformation sofort aus.

2. Falls eine sofortige Reparatur nicht möglich ist, transformieren wir den Baum so, daß entweder die Reparatur im nächsten Schritt vollständig erfolgen kann (Fall 3)

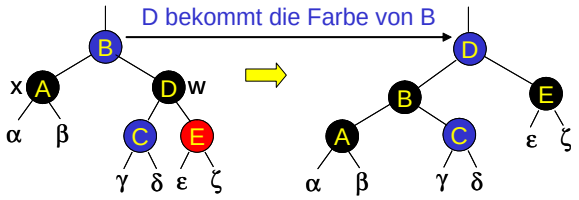


Abb. 6.11: Löschen in einem Rot-Schwarz-Baum, Fall 4

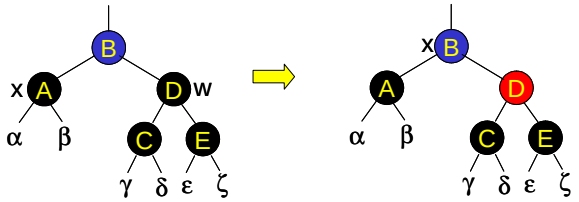


Abb. 6.12: Löschen in einem Rot-Schwarz-Baum, Fall 2

oder im übernächsten (Fall 1), oder daß zumindest x auf dem Pfad zur Wurzel eine Ebene hochgehoben werden kann (Fall 2).

Die Laufzeit der Reparatur-Prozedur und der gesamten Entfernen-Operation ist $\mathcal{O}(logn)$.

Satz 6.4. *Alle auf Seite 65 aufgeführten dynamischen Operationen haben auf Rot-Schwarz-Bäumen mit n Elementen Laufzeit* $\mathcal{O}(logn)$.

6.4 AVL-Bäume

Zur Erinnerung: Die Höhe eines Knotens v in einem Baum T ist gleich der Länge des längsten Pfades von v zu einem Nachkommen von v.
Die Höhe des Baums T entspricht der Höhe seiner Wurzel.

Definition 6.4. Ein Unterbaum eines Knotens v ist ein *maximaler Teilbaum*, dessen Wurzel ein Kind von v ist (siehe Abb. 6.15).

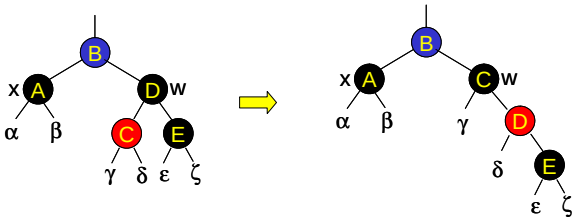


Abb. 6.13: Löschen in einem Rot-Schwarz-Baum, Fall 3

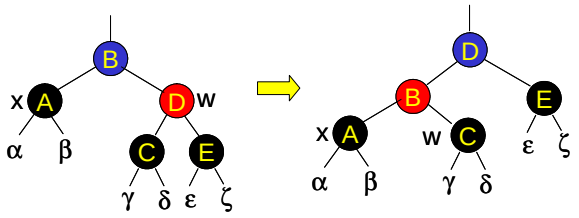


Abb. 6.14: Löschen in einem Rot-Schwarz-Baum, Fall 1

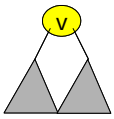


Abb. 6.15: Maximale Teilbäume

Definition 6.5. Wir definieren die *Balance* eines Knotens v als $bal(v) = h_l - h_r$, wobei h_l und h_r die Höhen des linken bzw. des rechten Unterbaums von v sind. Die Höhe eines leeren Baums wird als -1 definiert.

Definition 6.6. Wir nennen v *balanciert*, falls $bal(v) \in \{-1, 0, 1\}$. Sonst nennen wir v *unbalanciert*.

Definition 6.7. Ein *AVL-Baum* ist ein Baum, in dem alle Knoten balanciert sind.

Wie kann man nun Elemente in einen AVL-Baum einfügen und dabei die Balance-Eigenschaft aufrechterhalten? Zunächst benötigt jeder Baumknoten v einen weiteren Eintrag $h(v)$, in dem die Höhe des Knotens gespeichert wird.
Wir verwenden die Einfüge-Prozedur für binäre Suchbäume *tree_insert* (siehe S.69). Durch das Einfügen ändern sich eventuell die Höhen der Knoten auf dem Weg vom eingefügten Knoten zur Wurzel, und damit geht eventuell die Balance dieser Knoten verloren. Ein Beispiel dafür zeigt Abbildung 6.16. Um die Balance-Eigenschaft wiederherzustellen, wandern wir den (roten) Pfad vom neu eingefügten Element (hier Knoten 6) bis zur Wurzel des Baums nach oben und berechnen dabei die neuen Werte für Höhe und Balance. Gegebenenfalls führen wir eine Transformation durch, um die Balance wiederherzustellen.

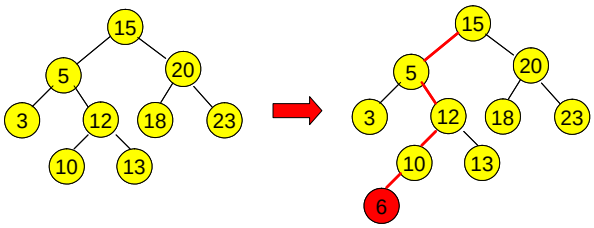


Abb. 6.16: Beim Einfügen eines Knotens in einen AVL-Baum kann sich die Balance der Knoten im Baum ändern

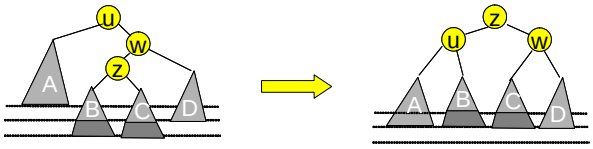


Abb. 6.17: Doppelrotation in einem AVL-Baum, Fall 1

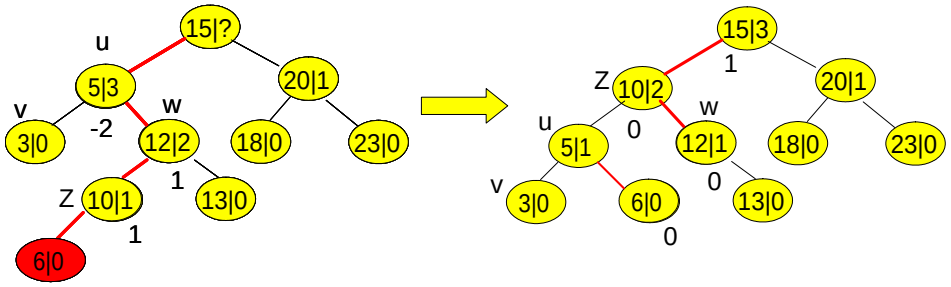


Abb. 6.18: Der AVL-Baum (Fall 1) nach der Doppelrotation

Die Höhe eines Knotens v berechnen wir wie folgt:

$$h(v) = max\{h(left\_child), h(right\_child)\} + 1$$

Sei u der erste Knoten (Knoten mit maximaler Tiefe), der nach der Einfüge-Operation nicht mehr balanciert ist, und sei v der linke Sohn von u und w der rechte Sohn von u. $bal(u)$ ist entweder $+2$ oder -2. Diese beiden Fälle sind symmetrisch. Deshalb betrachten wir hier nur den Fall $bal(u) = -2$. Da $bal(u) = -2$, muß folglich u ein „Übergewicht nach rechts“ haben.

Fall 1: $bal(w) = 1$: Sei z der linke Sohn von w. Wir führen eine *Doppel-Rotation* aus:

1. *right_rotate(T,w)*
2. *left_rotate(T,u)*

Der schematische Vorgang ist in Abbildung 6.17 dargestellt. Für unseren konkreten Baum ist das Ergebnis in Abbildung 6.18 zu sehen.

Fall 2: $bal(w) \in 0, 1$: Der schematische Vorgang für Fall 2 ist in Abbildung 6.19 dargestellt. Im Beispiel in Abbildung 6.20 wurde Knoten 14 neu eingefügt; die Abbildung zeigt das Ergebnis nach der erfolgten Rotation.
Sowohl die Doppelrotation im Fall 1 als auch die einfache Rotation im Fall 2 kann durch Umsetzen der entsprechenden Zeiger in konstanter Zeit ausgeführt werden. Das gleiche gilt auch für die Höhen- und Balance-Berechnungen.
Alle Knoten entlang des Pfades vom eingefügten Element zur Wurzel müssen eventuell repariert werden. Da man für jeden Knoten jedoch nur konstante Zeit benötigt, hat die Einfügeprozedur die Laufzeit $\mathcal{O}(logn)$, wobei n die Zahl der Elemente im Baum ist.

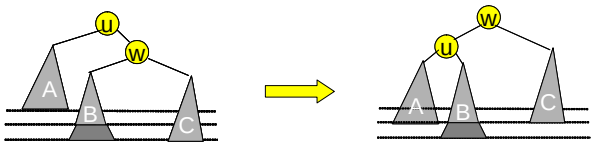


Abb. 6.19: Rotation in einem AVL-Baum, Fall 2

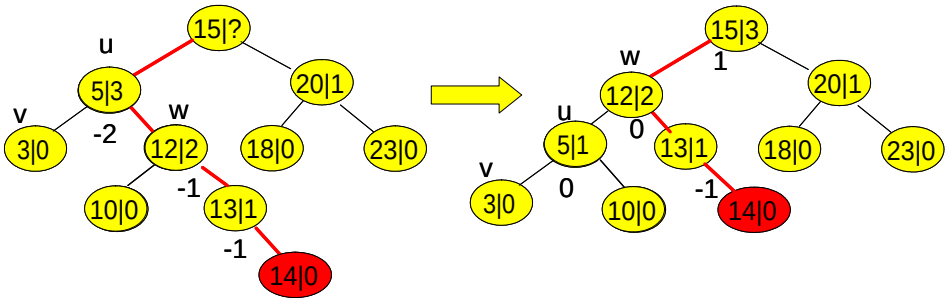


Abb. 6.20: AVL-Baum nach der Doppelrotation. Einfügt wurde Knoten 14.

Auch das Entfernen eines Elements beginnt mit der üblichen Prozedur für binäre Suchbäume, die zu Beginn dieses Kapitels diskutiert wurde. Dann wird wie beim Einfügen der Pfad zur Wurzel repariert. Daher kann auch das Löschen eines Elements in Zeit $\mathcal{O}(logn)$ durchgeführt werden.

Satz 6.5. *AVL-Bäume unterstützen alle Operationen eines Wörterbuchs in $\mathcal{O}(logn)$ Zeit, wobei n die Anzahl der Elemente im Baum ist.*

6.5 Anmerkungen

In Cormen et.al. [2] finden sich die Original-Kapitel *Binary Search Trees* und *Red-Black Trees*, die die Grundlage für diesen Abschnitt bildeten.