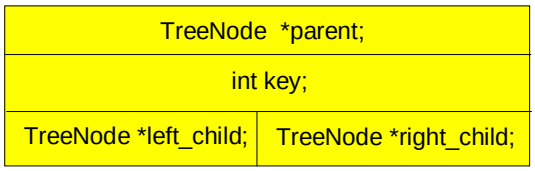
Binäre Suchbäume

Suchbäume unterstützen alle unten angegebenen Operationen für dynamische Mengen K effizient:

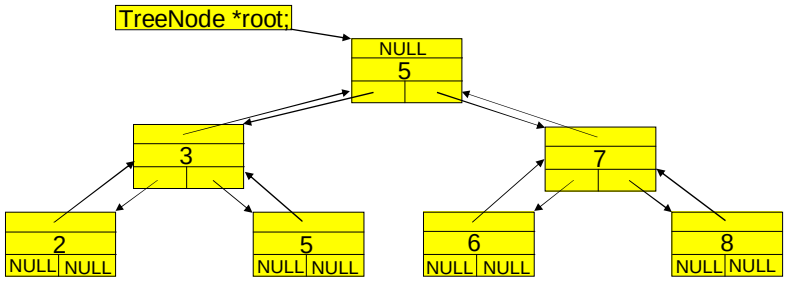
Search(K ,k)	----	Suche ein Element x von K mit Schlüssel k.
Insert(K ,x)	----	Füge das Element x in K ein.
Delete(K ,x)	----	Entferne das Element x aus K.
Minimum(K)	----	Suche das (ein) Element von K mit minimalem Schlüssel.
Maximum(K)	----	Suche das (ein) Element von K mit maximalem Schlüssel.
Successor(K,x)	----	Suche das Element, dessen Schlüssel in der Ordnung von K dem Schlüssel von x folgt (der nächst größere Schlüssel).
Predecessor(K,x)	----	Suche das Element, dessen Schlüssel in der Ordnung von K dem Schlüssel von x vorangeht (der nächst kleinere Schlüssel).

Binäre Suchbäume

Typische Struktur eines Baumknotens („Class TreeNode“):



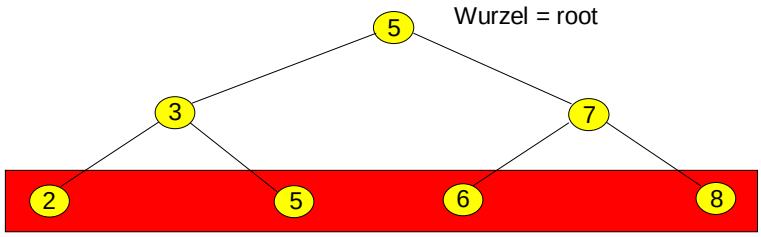
Beispielstruktur eines binären Baumes:



Typische Struktur eines Baumknotens („Class TreeNode“):

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

Vereinfachte Darstellung der Baumstruktur:



Blätter (leafs) des Baumes: Die Knoten ohne Kinder!

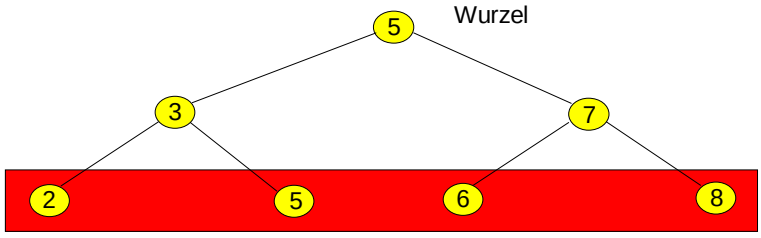
In einem binären Suchbaum werden die Schlüssel wie folgt gespeichert:

Sei x ein Knoten des Baumes:

Für jeden Knoten y im linken Unterbaum von x gilt: $y.key \leq x.key$

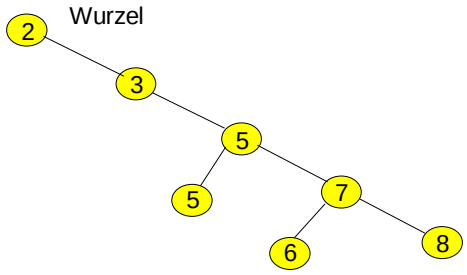
Für jeden Knoten y im rechten Unterbaum von x gilt: $y.key \geq x.key$

Vereinfachte Darstellung der Baumstruktur:



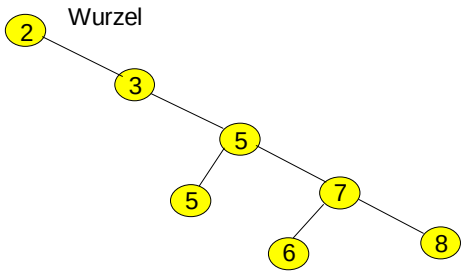
Blätter des Baumes: Die Knoten ohne Kinder!

Auch der folgende Baum ist ein binärer Suchbaum mit der obigen Ordnungs-eigenschaft:



Die Ordnungseigenschaft ermöglicht die Ausgabe aller im Baum gespeicherten Schlüssel, sortiert nach der Größe des Schlüssels.


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



Satz [1]:

Starten wir die Funktion „tree_walk“ mit der Wurzel x eines Baumes 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:

$T(n) = T(k) + T(n-k-1) + d$ d 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 $T(0) = c = (c+d)0 + c$
Induktionsannahme: Wir nehmen an, dass die obige Aussage für alle ganzen Zahlen kleiner als n gilt:

Induktionsschluss:

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

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

$T(n) = (c+d)n + c$ ■

Wie sucht man in einem binären Suchbaum nach einem Schlüssel k?

```
// Rekusive Suche

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);
}
```

Wie sucht man in einem binären Suchbaum nach einem Schlüssel k?

```
// Iterative Suche

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

```
TreeNode* tree_successor(TreeNode *x) {???}
TreeNode* tree_predeccessor(TreeNode *x) {???}
TreeNode* tree_minimum(TreeNode *x){???}
TreeNode* tree_maximum(TreeNode *x){???}
```

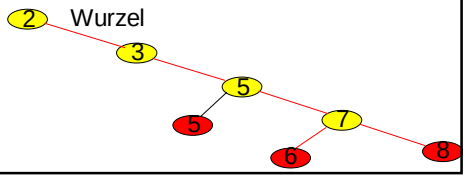
sind sehr ähnlich zu den angebenen Implementierungen und daher sind sie auch einfach zu implementieren.

Satz [2]:

Die Funktionen „tree_search“, „iterative_tree_search“, „tree_successor“, „tree_predecessor“, „tree_minimum“ und „tree_maximum“ haben eine Laufzeit von O(h), wobei h die Höhe des binären Baumes ist.

Definition [1]: Höhe eines Knoten v in einem Baum ist gleich der Zahl der Kanten auf dem längsten Weg von dem Knoten zu einem Blatt im Unterbaum des Knotens.

Höhe des Baumes = Höhe der Wurzel



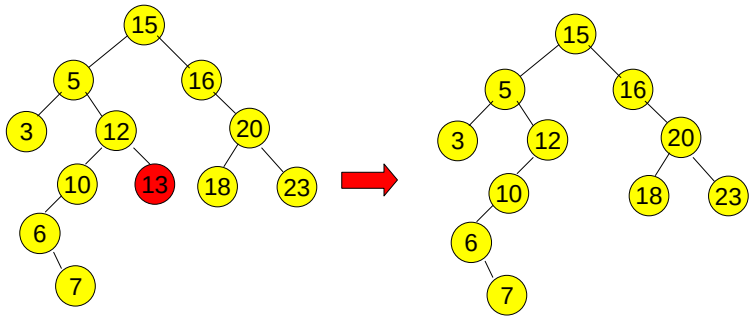
Wir betrachten nun die Funktion „tree_insert“, die einen neuen Knoten z mit Schlüssel k einfügt. Wir nehmen an, dass „z->left_child“ und „z->right_child“ schon auf NULL gesetzt wurden.

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

  while ( x != NULL) {              // Durchmustern des Baumes
     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 einfügen
        else                   p->right_child = z;   // Rechts von p einfügen

}
```

Zeit für Einfügung im worst case: O(h) , wobei h die Höhe des Baumes ist.

Beim Löschen von Baumknoten ist eine Fallunterscheidung erforderlich:

[1] Der zu löschende Knoten z hat keine Kinder:

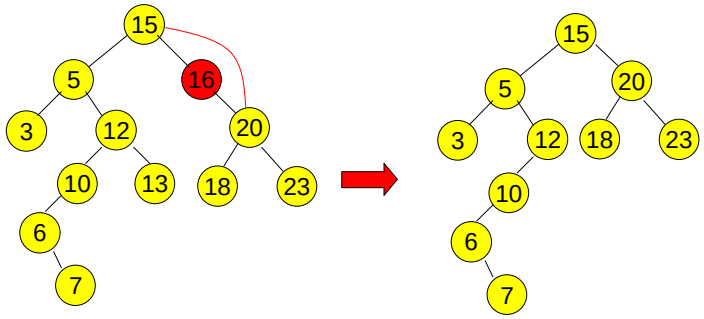


Man setze den entsprechenden Zeiger des Vaters auf NULL und lösche den Knoten z.

O(1)

Beim Löschen von Baumknoten ist eine Fallunterscheidung erforderlich:

[2] Der zu löschende Knoten z hat genau ein Kind:



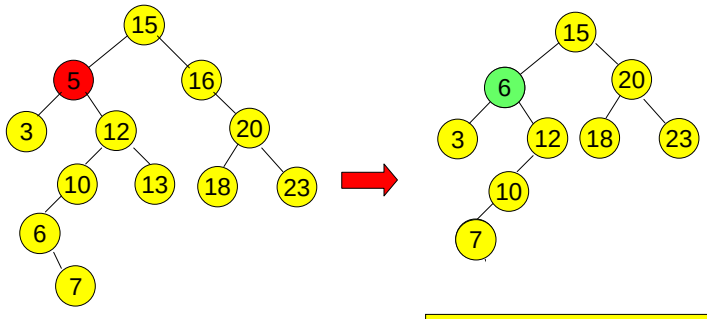
„Splice out z“:

O(1)

Verbinde das Kind von z mit dem Vater von z.

Beim Löschen von Baumknoten ist eine Fallunterscheidung erforderlich:

[3] Der zu löschende Knoten z hat zwei Kinder:



„Splice out z‘s Successor“:

Zeit O(h) im worst case

[1] Suche den „Successor“ von z im rechten Unterbaum von z.

[2] Übergibt die Daten (den Schlüssel) des Successors an z.

[3] Lösche z mit der Prozedur für Knoten mit einem Kind.

Satz [3]:

In einem binären Suchbaum der Höhe h kann in Zeit O(h) ein Element eingefügt oder gelöscht werden.

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 Bit 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):

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


Definition [2]: (Rot-Schwarz-Baum):

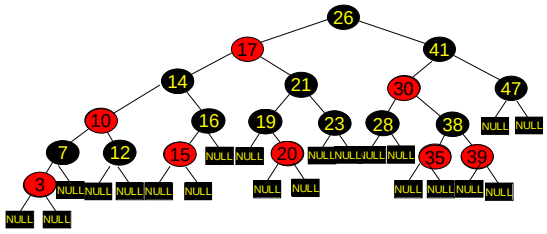
[1] Jeder Knoten ist entweder rot oder schwarz.

[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 gleiche Zahl von schwarzen Knoten.

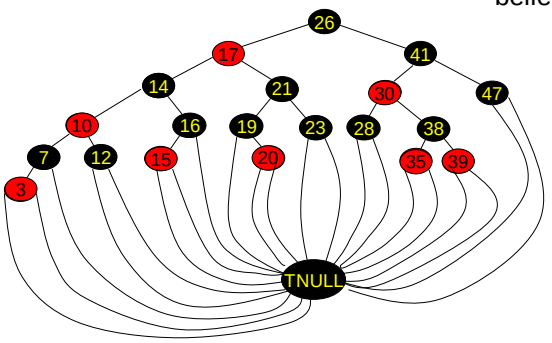


Mit diesem Knoten können wir wie mit jedem anderen Knoten arbeiten.

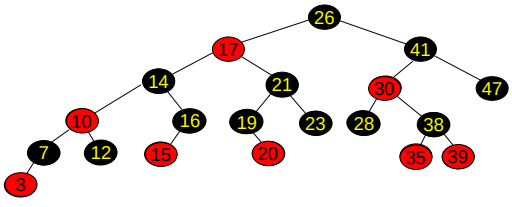
TNULL->color = BLACK;

„parent“, „left_child“, „key“, und „right_child“ können beliebig gesetzt werden.

Die NULL Zeiger können alle auf die Adresse eines besonderen Knoten TNULL umgesetzt 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 [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 [1]:

Ein Rot-Schwarz-Baum mit n inneren Knoten hat höchstens Höhe 2 log(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.

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

$$(2^{sh(x)-1} - 1) + (2^{sh(x)-1} - 1) + 1 = 2^{sh(x)} - 1 \quad \text{ist.}$$

Sei h die Höhe des Baumes. Mindestens die Hälfte der Knoten von der Wurzel zu irgend einem Blatt sind schwarz. Hieraus folgt:

$$sh(root) \geq h/2 \Rightarrow n \geq 2^{h/2} - 1 \Rightarrow 2\log(n+1) \geq h$$ ■

Aus dem obigen Lemma folgt direkt:

Die Suche nach einem Element, dem Minimum, dem Maximum, dem Successor oder Predecessor eines Elementes kann in Zeit O(log n) in einem Rot-Schwarz-Baum mit n Knoten ausgeführt werden.

Auch die im vorhergehenden Abschnitt beschriebenen Funktionen „tree_insert“ und „tree_delete“ haben eine Laufzeit von O(log n). Nach der entsprechenden Operation liegt jedoch eventuell kein Rot-Schwarz-Baum mehr vor.

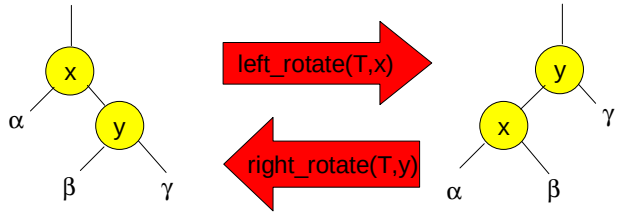
Wir werden im folgenden Abschnitt lernen, dass man mittels sogenannter Rotationen von Teilbäumen und mittels Umfärben der Knoten die Eigenschaften des Rot-Schwarz-Baums wiederherstellen kann.

Rotationen:

Um nach Einfügungen und Löschungen im Baum die Eigenschaften des Rot-Schwarz-Baumes 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.

[1] Rotation im Baum T von x nach links (rechts)



Wir nehmen an, dass das rechte Kind y von x nicht NULL (TNULL) ist.

Übung: Implementieren Sie die Rotation nach rechts!

Die Einfüge-Operation für 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 Baumes
     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 einfügen
        else                  p->right_child = z; // Rechts von p einfügen
  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 gleiche Zahl von schwarzen Knoten.

Eigenschaft [2] und [4] können durch Einfüge-Operationen verletzt werden.

Eigenschaft [2] können wir sehr einfach garantieren, in dem wir am Ende der Reparatur-Prozedur „RB_insert_fixup“ immer der Wurzel die Farbe „schwarz“ zuordnen:

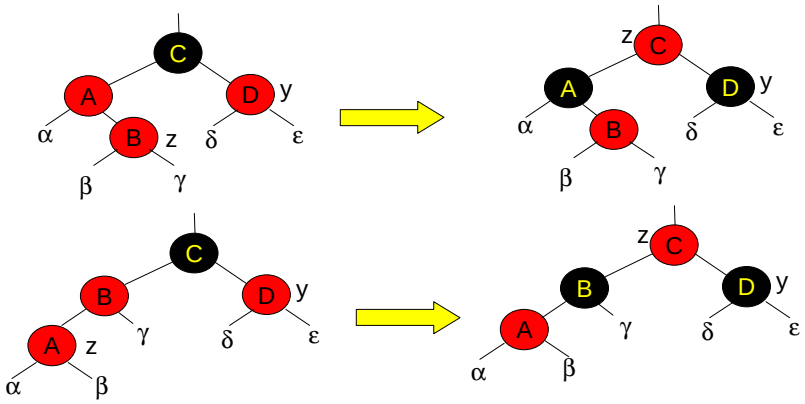
```
T->root->color = BLACK;
```

Da z nur schwarze Blätter [TNULL] als Kinder hat, kann Eigenschaft [4] nur verletzt werden, wenn der Vater z->parent von z auch rot ist.

Da z nur schwarze Blätter [TNULL] als Kinder hat, kann Eigenschaft [4] nur verletzt sein, wenn der Vater z->parent von z auch rot ist.

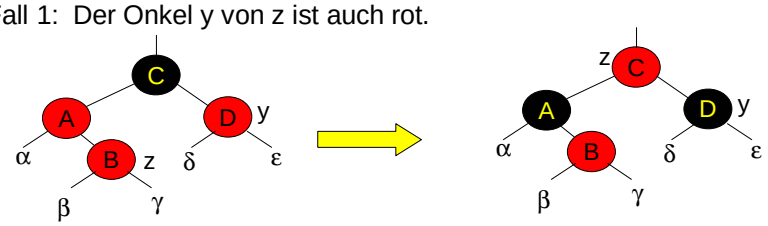
Es folgt eine Fallunterscheidung mit drei Fällen:

Fall 1: Der Onkel y von z ist auch rot.



Da z nur schwarze Blätter [TNULL] als Kinder hat, kann Eigenschaft [4] nur verletzt sein, wenn der Vater z->parent von z auch rot ist.

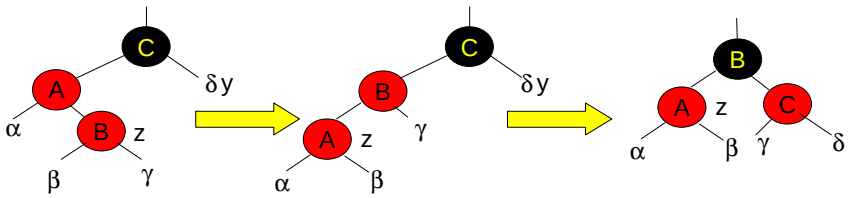
Fall 1: Der Onkel y von z ist auch rot.




```
<WHILE-LOOP>: while (z->parent != NULL &&  z->parent->color == RED) {
                  if ( z->parent == z->parent->parent->left_child) {
                     y = z->parent->parent->right_child;
          <FALL1>:   if ( y->color == RED) {
                        y->color = BLACK;
                        z->parent->color = BLACK;
                        z->parent->parent->color = RED;
                        z = z->parent->parent;
                     } else {   <FALL2>     }
                  } else {  ??? } //symmetrisch zu ersten if Schleife
               }
```


Da z nur schwarze Blätter [TNULL] als Kinder hat, kann Eigenschaft [4] nur verletzt sein, wenn der Vater z->parent von z auch rot ist.

Fall 2: Der Onkel y von z ist schwarz und z ist ein „rechtes“ Kind.




```
<FALL2>: if ( z == z->parent->right_child) {
            z = z->parent;
            left_rotate(T,z);
         }
```

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

```
<FALL3>: else {   z->parent->color = BLACK;
                  z->parent->parent->color = RED;
                  right_rotate(T, z->parent);   }
```

Da z nur schwarze Blätter [TNULL] als Kinder hat, kann Eigenschaft [4] nur verletzt sein, wenn der Vater z->parent von z auch rot ist.

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

   <WHILE-LOOP>

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

Die Laufzeit von RB_insert_fixup ist O(log n).

Um die Korrektheit der obigen Prozedur „RB_insert_fixup“ zu beweisen, muss man zeigen, dass die folgenden Invarianten vor, während und nach der Prozedur erfüllt sind:

[1] Der Knoten z ist rot.

[2] Falls der Vater von z gleich der Wurzel ist, dann ist der Vater schwarz.

[3] Falls eine der Bedingungen des Rot-Schwarz-Baums verletzt ist, dann ist es entweder Bedingung 2 oder Bedingung 4. Falls 2 verletzt ist, dann ist z rot und zugleich die Wurzel des Baums. Falls 4 verletzt ist, dann sind z und der Vater von z rot.

Wie entfernt man einen Knoten aus einem Rot-Schwarz-Baum?

Die Prozedur ist sehr ähnlich zur Entferne-Operation auf 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 hierbei 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 Knoten y verletzt.

Wie kann man die beiden Probleme [P2] und [P3] lösen bzw. die Verletzungen der Bedingungen eliminieren ?

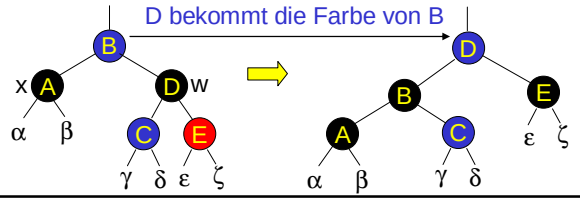
Wie kann man das Problem [P3] ([P2]) lösen?

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.

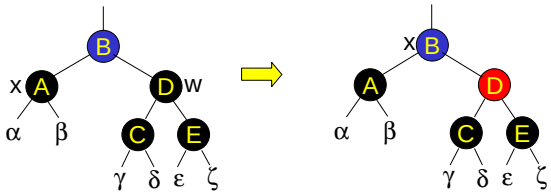
[b] Ist x schwarz und könnten wir x einen weiteren „Schwarz-Wert“ zuordnen (also insgesamt 2), so wäre die Bedingung 5 wiederhergestellt. Da ein RS-Baum jedoch keine Zähler für Farbeinheiten besitzt, müssen wir gewisse weitere Reparaturen durchführen, um jeweils den Knoten x mit dem künstlichen Schwarz-Wert von „zwei“ zu reparieren. Dabei hangeln wir (mit x) solange den Pfad zur Wurzel hoch, 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:

Fall 4: w ist schwarz und w‘s rechtes Kind ist rot



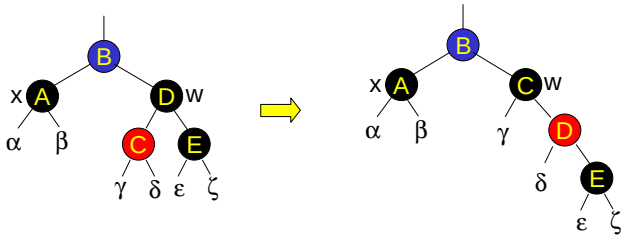
Die blau gezeichneten Knoten können entweder rot oder schwarz gefärbt sein.

Fall 2: w ist schwarz und beide Kinder von w sind auch schwarz.



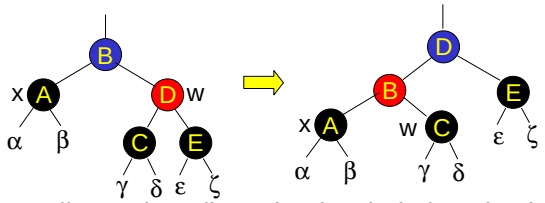
Falls B rot ist, dann färben wir B schwarz und D rot und sind fertig.

Fall 3: w ist schwarz und w‘s linkes Kind ist rot.



Dies ist Fall 4!!!

Fall 1: w ist rot.



Falls wir von Fall1 aus in Fall 2 gelandet sind, dann ist der neue Knoten „x“ rot gefärbt und wir färben ihn dann schwarz und die Reparatur ist 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) 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 Entferne-Operation ist O(log n).

Satz [4]:

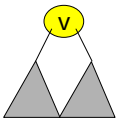
Die auf der ersten Folie erwähnten dynamischen Operationen haben auf Rot-Schwarz-Bäumen mit n Elementen Laufzeit O(log n).

AVL-Bäume:

Zur Erinnerung: Die Höhe eines Knotens v eines Baumes T ist gleich der Länge des längsten Pfades von v zu einem Nachkommen von v.

Die Höhe des Baumes T ist gleich der Höhe der Wurzel.

Definition 4: [a] Ein Unterbaum eines Knotens v ist ein maximaler Teilbaum, dessen Wurzel ein Kind von v ist.



[b] Wir definieren die Balance eines Knoten 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.

[c] Wir nennen v balanciert, falls $bal(v) \in \{ -1, 0, 1 \}$, und unbalanciert sonst.

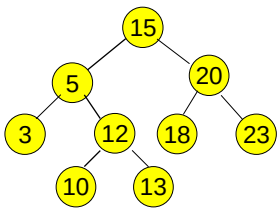
[d] Ein AVL-Baum ist ein Baum, in dem alle Knoten balanciert sind.

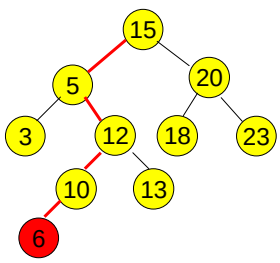
Übungsaufgabe: Weisen Sie nach, dass AVL-Bäume logarithmische Höhe haben (n = Zahl der Elemente in T => h(T) = O(log n)).

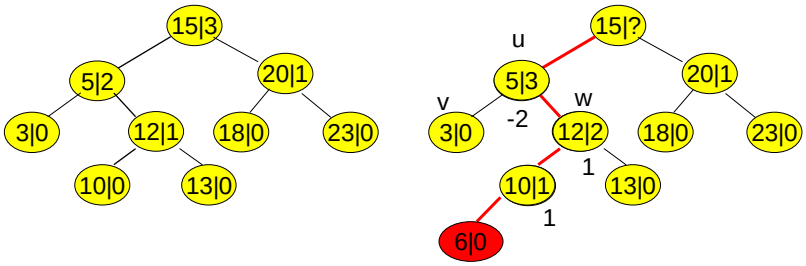
Wie kann man Elemente in einen AVL-Baum einfügen und dabei die Balance-Eigenschaft aufrechterhalten?

Zunächst benötigt jeder Baumknoten v ein weiteren Eintrag h(v), in dem die Höhe des Knotens gespeichert wird.

Wir verwenden die auf Seite 11 vorgestellte Einfüge-Prozedur für binäre Suchbäume. 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.



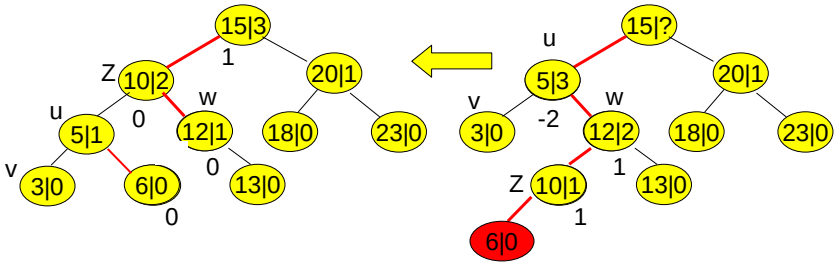




Wir gehen nun den (roten) Pfad von dem neuen Element zur Wurzel hoch, berechnen die aktuellen Höhen und die Balance-Werte und führen gegebenfalls eine Transformation durch, die die Balance wiederherstellt.
Die Höhe eines Knoten v berechnen hierbei wie folgt:

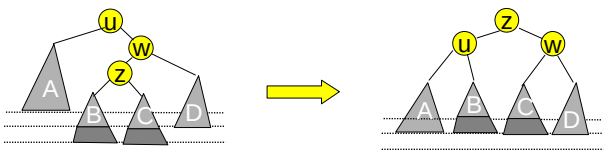
h(v) = max { h(left_child), h(right_child)} + 1

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 sei w der rechte Sohn von v .



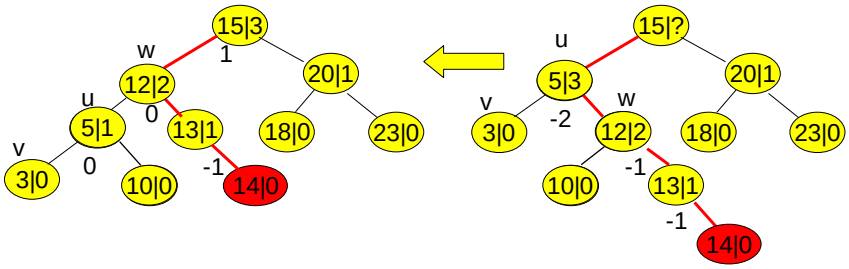
bal(u) ist entweder +2 oder –2. Diese beiden Fälle sind symmetrisch.
Deshalb betrachten wir hier nur den Fall bal(u) = -2.
Da bal(u) = -2, muss folglich u „Übergewicht nach rechts“ haben.

Fall 1: bal(w) = 1 : Sei z der linke Sohn von w.

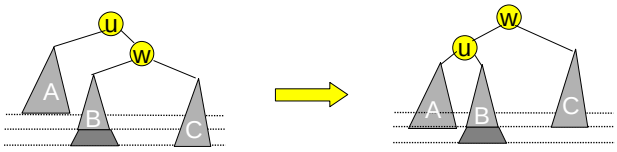


Doppelrotation:

[1] right_rotate(T,w)
[2] left_rotate(T,u)



Fall 2: bal(w) ∈ {0,-1}:



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 (natürlich auch die Höhen- und Balanceberechnungen).

Alle Knoten entlang des Pfades von dem eingefügten Element zur Wurzel müssen eventuell repariert werden. Da man für jeden Knoten jedoch nur konstante Zeit benötigt, hat die Einfügeprozedur Laufzeit O(log n), wobei n die Zahl der Elemente im Baum ist.

Auch das Entfernen eines Elementes beginnt mit der üblichen Prozedur für binäre Suchbäume, die zu Beginn dieser Vorlesung diskutiert wurde. Dann wird wie beim Einfügen der Pfad zur Wurzel repariert. Daher kann auch das Löschen eines Elements in Zeit O(log n) durchgeführt werden.

Satz [5]:

AVL-Bäume unterstützen alle Operationen eines Wörterbuchs in O(log n) Zeit, wobei n die Anzahl der Elemente im Baum ist.