Dynamische Datenstrukturen

Lennart Rosseburg

Lizenzhinweis

Unter Nutzung von

- Dem Kapitel "Dynamische Datenstrukturen" aus dem Kurs "Algorithmen und Datenstrukturen", von Wikiversity unter der Beteiligung folgender Autor:innen, unter der Lizenz CC-BY-SA (3.0).

  Diese Selbstlerneinheit konzentriert sich auf die Funktionsweise verschiedener Dynamischer Datenstrukturen und enthält interaktive Aufgaben um das Gelernte zu überprüfen und zu verinnerlichen.

Ziel des Kurses:

Am Ende dieser Selbstlerneinheit sollten Sie die vorgestellten Dynamischen Datenstrukturen kennen, unterscheiden und selbst anwenden können.

Einleitung

Unter dynamischen Datenstrukturen verstehen wir Datenstrukturen, bei denen man Elemente löschen und hinzufügen kann, eine interne Ordnung (z.B. Sortierung) vorliegt und diese Ordnung unter Änderungen aufrecht erhalten bleibt. Ein Beispiel sind Lineare Datenstrukturen und Sortierung. Bei einer **unsortierten** Liste sind Änderungen einfach, aber der Zugriff aufwändig. Bei einer **sortierten** Liste sind die Änderungen schwierig, aber dafür der Zugriff einfach. Für diesen Trade-off ist eine "intelligente Datenstruktur" gesucht, die Änderungen **und** Zugriffe einfach, sprich effizient, hält. Viele dynamische Datenstrukturen nutzen Bäume als Repräsentation.

> Die Inhalte dieses Kurses bauen auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java von Gunter Saake und Kai-Uwe Sattler auf. Daher empfiehlt sich dieses Buch, um das vorgestellte Wissen zu vertiefen.

Desweiteren erwarten Sie in diesem Kurs Beispiele zum Implementieren der vorgestellten dynamischen Datenstrukturen. Die benutzte Programmiersprache in diesen Beispielen ist **Java**.

Grundlagen

Bevor wir die einzelnen dynamischen Datenstrukturen vorstellen, ein kurzer Abschnitt zu den benötigten Grundlagen.

Bäume

In diesem Kapitel werden Bäume als kurzen Einschub behandelt. Ein Baumelement e ist ein Tupel $e = (v, \{e_1, ..., e_n\})$ mit v vom Wert e und $\{e_1, ..., e_n\}$ sind die Nachfolger, bzw. Kinder von e. Ein Baum T ist ein Tupel $T = (r, \{e_1, ..., e_n\})$ mit r als Wurzelknoten (ein Baumelement) und $\{e_1, ..., e_n\}$ als Knoten (Baumelemente) des Baumes mit $r \in \{e_1, ..., e_n\}$ und für alle $e_i = (v_i, K_i)$ und $e_j = (v_j, K_j) \in \{e_1, ..., e_n\}$ gilt $K_i \bigcap K_j = \emptyset$

Man spricht von einem geordneten Baum, wenn die Reihenfolge der Kinder $\{e_1, .., e_n\}$ eines jeden Elements $e = (v, \{e_1, ..., e_n\})$ festgelegt ist (schreibe dann $(e_1, ..., e_n)$ statt $\{e_1, ..., e_n\}$).

Beispiel:

$T = (v_4, \{v_1, v_2, v_3, v_4, v_5\})$

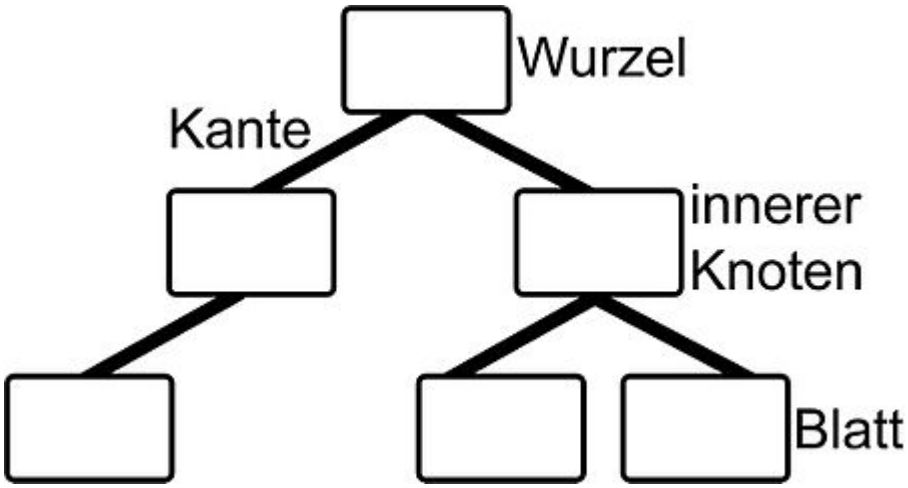
- $v_1 = (1, \{\})$
- $v_2 = (2, \{v_1, v_3\})$
- $v_3 = (3, \{\})$
- $v_4 = (4, \{v_2, v_5\})$
- $v_5 = (5, \{\})$

$T' = (v_4, \{v_2, v_5\})$

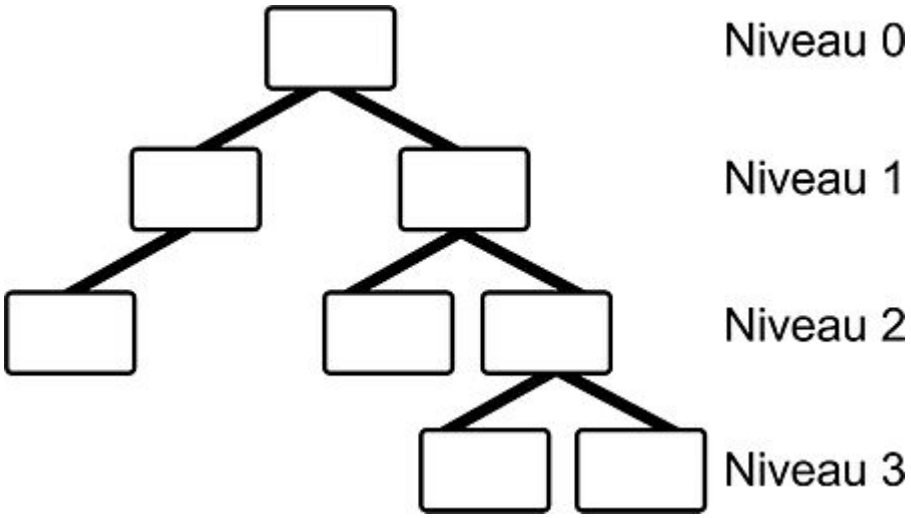
- $v_2 = (2, \{v_5\})$
- $v_4 = (4, \{v_2, v_5\})$
- $v_5 = (5, \{\})$

T' ist **kein** Baum, da v_4 und v_2 ein gemeinsames Kind haben.

Begriffe:



Ein **Pfad** folgt über **Kanten** zu verbundenen **Knoten**, dabei existiert zu jedem Knoten genau ein Pfad von der **Wurzel**. Ein Baum ist immer *zusammenhängend* und *zyklenfrei*.

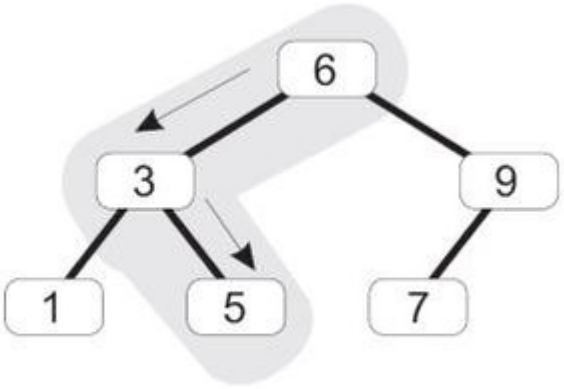


Das **Niveau** der jeweiligen Ebene entspricht immer der jeweiligen **Länge des Pfades**. Die Höhe eines Baumes entspricht dem größten Niveau + 1.

Anwendungen:

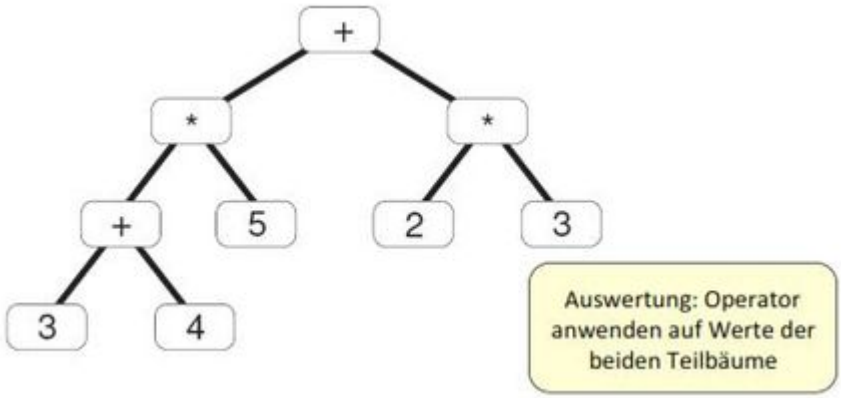
Man benutzt Bäume beispielsweise zur Darstellung von Hierarchien, wie Taxonomien, oder für Entscheidungsbäume. Bäume werden oft genutzt um sortierte, dynamische oder lineare Datenstrukturen zu repräsentieren, da Einfüge- und Löschoperationen leicht so definiert werden können, dass die Sortierung beibe-

halten wird. Ein Baum kann auch als Datenindex genutzt werden und stellt so eine Alternative zu Listen und Arrays dar.



Hier wird beispielsweise nach der 5 gesucht und der Baum wird als *Suchbaum* genutzt.

Man kann auch einen Baum aus *Termen* bilden. Der Term $(3 + 4) * 5 + 2 * 3$ ergibt folgenden Baum:



Atomare Operationen auf Bäumen:

Zu den Operationen zählen **lesen** mit

- *root*(): Wurzelknoten eines Baums
- *get*(e): Wert eines Baumelements e
- *children*(e): Kinderknoten eines Elements e
- *parent*(e): Elternknoten eines Elements e

und **schreiben** mit

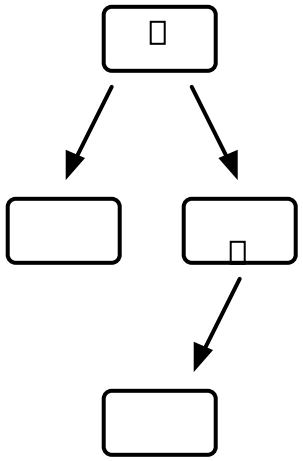
- $set(e, v)$: Wert des Elements e auf v setzen
- $addChild(e, e')$: Füge Element e' als Kind von e ein (falls geordneter Baum nutze $addChild(e, e', i)$ für Index i)
- $del(e)$: Lösche Element e (nur wenn e keine Kinder hat)

Spezialfall: Binärer Baum als Datentyp:

```
class TreeNode<K extends Comparable<K>> {

  K key;
  TreeNode<K> left = null;
  TreeNode<K> right = null;

  public TreeNode(K e) {key = e;}
  public TreeNode<K> getLeft() {return left;}
  public TreeNode<K> getRight()  {return right;}
  public K getKey() {return key;}

  public void setLeft(TreeNode<K> n) {left = n;}
  public void setRight(TreeNode<K> n) {right = n;}

  ...

}
```

Beispiel:

```
TreeNode<Character> root = new TreeNode<Character>('A');
TreeNode<Character> node1 = new TreeNode<Character>('B');
TreeNode<Character> node2 = new TreeNode<Character>('C');
TreeNode<Character> node3 = new TreeNode<Character>('D');

root.setLeft(node1);
root.setRight(node2);
node2.setLeft(node3);
```



Typische Problemstellungen

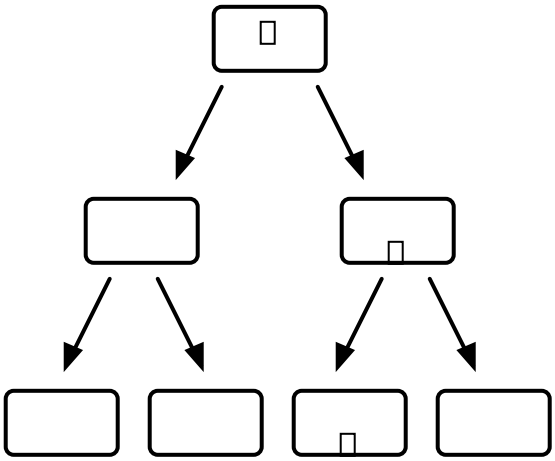
Als typische Problemstellungen haben wir zum einen die Traversierung, zum Anderen das Löschen eines inneren Knotens und die daraus folgende Re-strukturierung des Baumes und das Suchen in Bäumen.

Bäume in Java

In Java gibt es keine hauseigene Implementierung für allgemeine Bäume. Einige Klassen *(TreeMap, TreeSet)* benutzen Bäume zur Realisierung anderer Datenstrukturen. Andere Klassen *(JTree)* benutzen Bäume als Datenmodell zur Visualisierung.

Traversierung Bäume können visuell gut dargestellt werden. Manchmal ist jedoch eine Serialisierung der Elemente eines Baumes nötig. Man kann die Elemente eines Baumes durch *Preorder-Aufzählung*, *Inorder-Aufzählung*, *Postorder-Aufzählung* oder *Levelorder-Aufzählung* eindeutig aufzählen.

> Bei der **Traversierung** werden systematisch **alle Knoten** des Baumes durchlaufen.



Preorder (W-L-R): $A \to B \to D \to E \to C \to F \to G$

Inorder (L-W-R): $D \to B \to E \to A \to F \to C \to G$

Postorder (L-R-W): $D \to E \to B \to F \to G \to C \to A$

Levelorder: $A \to B \to C \to D \to E \to F \to G$

Traversierung mit Iteratoren

Bei der Traversierung sind Iteratoren erlaubt. Diese werden schrittweise abgearbeitet und es werden Standardschleifen für die Baumdurchläufe verwendet.

```
for (Integer i : tree) {
  System.out.print(i);
}
```

Dabei ist es allerdings notwendig, dass der Bearbeitungszustand zwischengespeichert wird.

```
public class BinarySearchTree<K extends Comparable<K>> implements Iterable<K> {

  public static final int INORDER = 1;
  public static final int PREORDER = 2;
  public static final int POSTORDER = 3;
  public static final int LEVELORDER = 4;

  private int iteratorOrder;
  ...

  public void setIterationOrder(int io) {
```

```
    if (io < i || io > 4) {
      return;
    }
    iteratorOrder = io;
  }

  public Iterator<K> iterator() {
    switch (iterationOrder) {
     case INORDER:
      return new InorderIterator<K>(this);
     case PRORDER:
      return new PreorderIterator<K>(this);
     case POSTORDER:
      return new PostorderIterator<K>(this);
     case LEVELORDER:
      return new LevelorderIterator<K>(this);
     default:
      return new InorderIterator<K>(this);
    }
  }
}
```

Preorder Traversierung

Bei der Preorder Traversierung wird der aktuelle Knoten zuerst behandelt und dann der linke oder rechte Teilbaum.

```
static class TreeNode<K extends Comparable<K>> {
...

  public void traverse() {
    if (key==null) {
      return;
    }
    System.out.print(" " + key);
    left.traverse();
    right.traverse();
  }
}
```

Preorder Iteratoren

Der Wurzelknoten wird auf den Stack gelegt, anschließend der rechte Knoten und dann der linke Knoten.

```
class PreorderIterator<K extends Comparable <K>> implements Iterator<K> {
```

```
  java.util.Stack<TreeNode<K>> st = new java.util.Stack<TreeNode<K>>();

  public PreorderIterator(BinarySearchTree<K> tree) {
    if (tree.head.getRight() != nullNode) {
      st.push(tree.head.getRight());
    }
  }

  public boolean hasNext() {
    return !st.isEmpty();
  }

  public K next() {
    TreeNode<K> node = st.pop();
    K obj = node.getKey();
    node = node.getRight();
    if(node != nullNode) {
       st.push(node);  //rechten Knoten auf den Stack
    }
    node = node.getLeft();
    if(node != nullNode) {
       st.push(node);  //linken Knoten auf den Stack
    }
    return obj;
  }
}
```

Inorder Traversierung

Bei der Inorder Traversierung wird zuerst der linke Teilbaum behandelt, dann der aktuelle Knoten und dann der rechte Teilbaum. Als Ergebnis erhält man den Baum in sortierter Reihenfolge.

```
static class TreeNode<K extends Comparable<K>> {
  ...
  public void traverse() {
    if (key==null) {
        return;
    }
    left.traverse();
    System.out.print(" " + key);
    right.traverse();
  }
}
```

Inorder Iteratoren

Der Knoten Head hat immer einen rechten Nachfolger. Es wird vom Wurzelknoten begonnen alle linken Knoten auf den Stack zu legen.

```
class InorderIterator<K extends Comparable <K>> implements Iterator<K> {

  java.util.Stack<TreeNode<K>> st = new java.util.Stack<TreeNode<K>>();

  public InorderIterator(BinarySearchTree<K> tree) {
    TreeNode<K> node = tree.head.getRight();
    while (node != nullNode) {
      st.push(node);
      node = node.getLeft();
    }
  }

  public boolean hasNext() {
    return !st.isEmpty();
  }

  public K next() {
    TreeNode<K> node = st.pop();
    K obj = node.getKey();
    node = node.getRight();  //rechten Knoten holen
    while (node != nullNode) {
      st.push(node);
      node = node.getLeft();  //linken Knoten auf den Stack
    }
    return obj;
  }
}
```

Postorder Traversierung

Bei der Postorder Traversierung wird zuerst der linke und der rechte Teilbaum behandelt und dann der aktuelle Knoten. Dies kann beispielsweise genutzt werden, um einen Baum aus Termen, entsprechend der Priorität der Operatoren, auszuwerten.

```
static class TreeNode<K extends Comparable<K>> {
  ...
  public void traverse() {
    if (key==null) {
        return;
```

```
    }
    left.traverse();
    right.traverse();
    System.out.print(" " + key);
  }
}
```

Levelorder Iteratoren

```
class LevelorderIterator<K extends Comparable <K>> implements Iterator<K> {

  //Wurzelknoten in die Warteschlange (queue) einfügen
  java.util.Queue<TreeNode<K>> q = new java.util.LinkedList<TreeNode<K>>();

  public LevelorderIterator(BinarySearchTree<K> tree) {
    TreeNode<K> node = tree.head.getRight();
    if (node != nullNode) {
      q.addLast(node);
    }
  }

  public K next() {
    TreeNode<K> node = q.getFirst();
    K obj = node.getKey();
    if (node.getLeft() != nullNode) {
      q.addLast(node.getLeft());
    }
    if (node.getRight() != nullNode) {
      q.addLast(node.getRight());
    }
    return obj;
  }
}
```

Grundlagenquiz

Bäume

In dem folgenden Kapitel werden wir Ihnen verschiedene Baumtypen vorstellen. Angefangen bei den **Binären Suchbäumen**, gefolgt von den **AVL-** und den **2-3-4-Bäumen**, bis hin zu den **Rot-Schwarz-Bäumen**.

Binäre Suchbäume

Auf dieser Seite werden die binären Suchbäume behandelt. Er ermöglicht einen schnellen Zugriff auf Daten mit dem Aufwand O(log n) unter geeigneten Voraus-

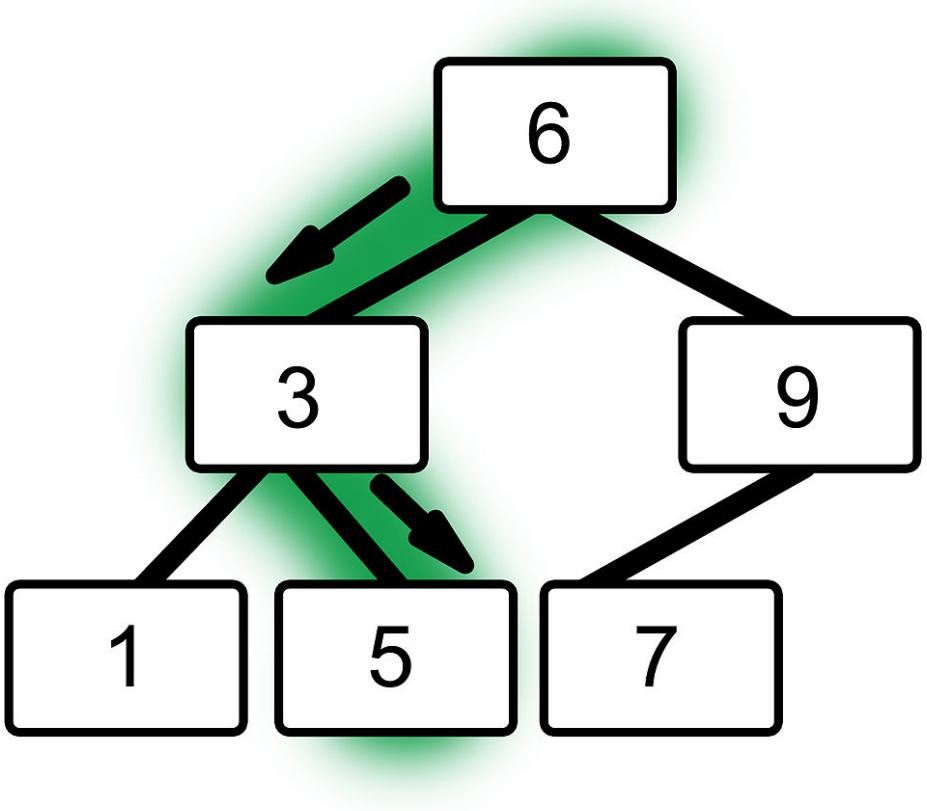
setzungen. Des Weiteren ermöglicht er effiziente Sortierung von Daten, durch Heapsort und effiziente Warteschlangen. Der binäre Suchbaum dient als Datenstruktur für kontextfreie Sprachen. In der Computergrafik sind Szenengraphen oft (Beinahe-)Bäume. Bei Informationssysteme dienen binäre Suchbäume zur Datenindizierung und Anfrageoptimierung.

Operationen

Auf Suchbäumen können die Operationen Suchen von Elementen, Einfügen von Elementen und Entfernen von Elementen angewandt werden, wobei letztere zwei voraussetzen, dass die Ordnung der Schlüssel erhalten bleibt.

Suchen Ein binärer Suchbaum kann für viele Anwendungen eingesetzt werden.

Hier ist der Baum ein Datenindex und eine Alternative zu Listen und Arrays. Beispielsweise kann dieser Baum als Suchbaum verwendet werden und nach 5 gesucht werden.



Bei der Anwendung von Bäumen zur effizienten Suche gibt es pro Knoten einen Schlüssel und ein Datenelement. Die Ordnung der Knoten erfolgt anhand der Schlüssel. Bei einem binären Suchbaum enthält der Knoten k einen Schlüsselwert $k.key$. Alle Schlüsselwerte im linken Teilbaum $k.left$ sind kleiner als $k.key$ und alle Schlüsselwerte im rechten Teilbaum $k.right$ sind größer als $k.key$. Die Auswertung eines Suchbaums sieht wie folgt aus:

1. Vergleich des Suchschlüssels mit Schlüssel der Wurzel
2. Wenn kleiner, dann in linken Teilbaum weiter suchen
3. Wenn größer, dann in rechtem Teilbaum weiter suchen
4. Sonst, gefunden oder nicht gefunden

```
static class  TreeNode<K extends Comparable<K>>{

  K key;
  TreeNode<K> left = null;
  TreeNode<K> right = null;

  public TreeNode(K e) {key = e;}
  public TreeNode<K> getLeft() {return left;}
  public TreeNode<K> getRight()  {return right;}
  public K getKey() {return key; }

  public void setLeft(TreeNode<K> n) {left = n;}
  public void setRight(TreeNode<K> n) {right = n;}

  ...
}
```

Knotenvergleich

```
class  TreeNode<...> {
  ...

  public int compareKeyTo(K k) {
    return (key == null ? -1 : key.compareTo(k));
  }

  ...
}
```

Rekursives Suchen

```
protected  TreeNode<K>recursiveFindNode(TreeNode<K>  n, k){
    /* k wird gesucht */
```

```
    if (n!= nullNode) {
      int cmp = n.compareKeyTo(k.key);
      if (cmp == 0) {
        return n;
      } else if (cmp > 0) {
        return recursiveFindNode(n.getLeft(),k);
      } else {
        return recursiveFindNode(n.getRight(),k);
      }
    }
    else {
      return null;
    }
}
```

Iteratives Suchen

```
protected  TreeNode<K> iterativeFindNode(TreeNode<K> k){
  /* k wird gesucht */

  TreeNode<K> n = head.getRight();
  while (n != nullNode) {
    int cmp = n.compareKeyTo(k.key);
    if (cmp == 0) {
      return n;
    } else {
      n = (cmp > 0 ? n.getLeft() : n.getRight());
    }
  }
  return null;
}
```

Suchen des kleinsten Elements

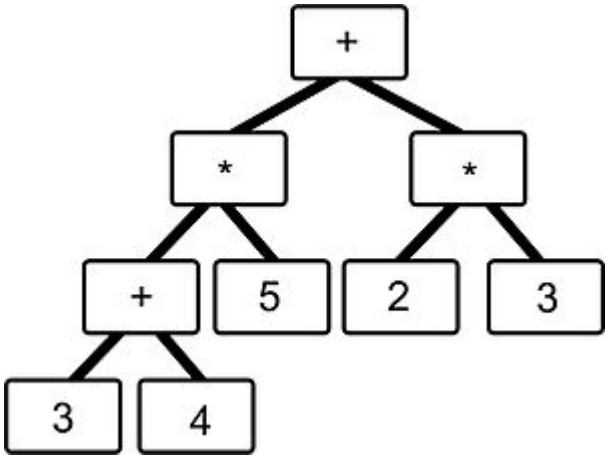
```
protected K findMinElement(){
  TreeNode<K> n = head.getRight();
  while (n.getLeft() != nullNode) {
    n = n.getLeft();
  }
  return n.getKey();
}
```

Suchen des größten Elements

```
protected K findMaxElement(){
  TreeNode<K> n = head.getRight();
  while (n.getRight() != nullNode) {
    n = n.getRight();
  }
  return n.getKey();
}
```

Der Baum aus Termen

Eine weitere Anwendungsmöglichkeit ist der Baum aus Termen. Wir haben den Term $(3 + 4) * 5 + 2 * 3$, als Baumdarstellung sieht es so aus:



Bei der Auswertung müssen die Operatoren auf die beiden Werte der Teilbäume angewandt werden.

Einfügen Das Finden der Einfügeposition erfolgt durch Suchen des Knotens, dessen Schlüsselwert größer als der einzufügende Schlüssel ist und der keinen linken Nachfolger hat oder durch Suchen des Knotens, dessen Schlüsselwert kleiner als der einzufügende Schlüssel ist und der keinen rechten Nachfolger hat. Das Einfügen erfolgt prinzipiell in 2 Schritten. Im **ersten Schritt** wird die Einfügeposition gesucht, sprich der Blattknoten mit dem nächstkleineren oder nächstgrößerem Schlüssel. Im **zweiten Schritt** wird ein neuer Knoten erzeugt und als Kindknoten des Knotens aus Schritt eins verlinkt. Wenn in Schritt eins der Schlüssel bereits existiert, dann wird nicht erneut eingefügt.

Programm in Java

```
/* Einfügeposition suchen */
public boolean  insert(K k) {
```

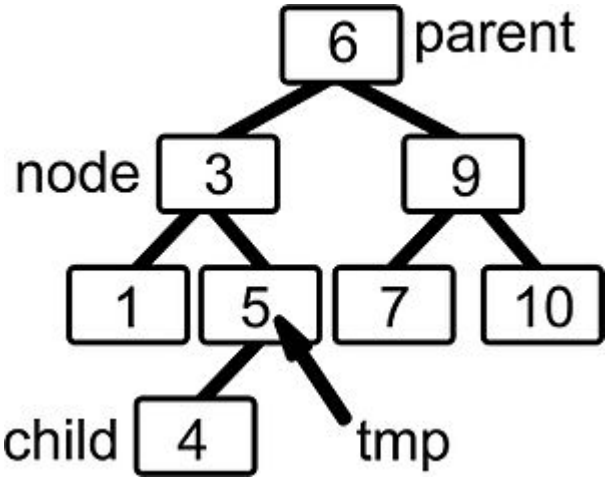
```
  TreeNode<K> parent = head;
  TreeNode<K> child = head.getRight();

  while (child != nullNode) {

    parent = child;
    int cmp = child.compareKeyTo(k);

    //Schlüssel bereits vorhanden
    if (cmp == 0) {
      return false;
    } else if (cmp > 0) {
      child = child.getLeft();
    } else {
      child = child.getRight();
    }

  }

  /* Neuen Knoten verlinken */
  TreeNode<K> node = new TreeNode<K>(k);
  node.setLeft(nullNode);
  node.setRight(nullNode);

  if (parent.compareKeyTo(k) > 0) {
    parent.setLeft(node);
  } else {
    parent.setRight(node);
  }

  return true;

}
```

Löschen Zuerst wird das zu löschendes Element gesucht, der Knoten k. Nun gibt es **drei** Fälle

1. k ist Blatt: löschen
2. k hat ein Kind: Kind „hochziehen"
3. k hat zwei Kinder: Tausche mit weitest links stehenden Kind des rechten Teilbaums, da dieser in der Sortierreihenfolge der nächste Knoten ist und entferne diesen nach den Regeln 1. oder 2.

Ein Schlüssel wird in drei Schritten gelöscht. Im ersten Schritt wird der zu löschende Knoten gefunden. Im zweiten Schritt wird der Nachrückknoten gefunden. Dafür gibt es mehrere Fälle. Im **Fall 1** handelt es sich um einen externen

Knoten, sprich ein Blatt, ohne Kinder. Dabei wird der Knoten durch einen NullNode ersetzt. Im **Fall 2a** gibt es nur einen rechten Kindknoten, dabei wird der gelöschte Knoten durch den rechten Kindknoten ersetzt. Im **Fall 2b** gibt es nur einen linken Kindknoten und der gelöschte Knoten wird durch diesen ersetzt. Im **Fall 3** gibt es einen internen Knoten mit Kindern rechts und links. Dabei wird der gelöschte Knoten durch den Knoten mit dem kleinstem (alternativ größtem) Schlüssel im rechten (alternativ linken) Teilbaum ersetzt. Im dritten und letzten Schritt wird nun der Baum **reorganisiert**. Während dem Löschen kann sich die Höhe von Teilbäumen ändern.



Programm in Java

```
/* Knoten suchen */
public boolean  remove(K k) {

  TreeNode<K> parent = head;
  TreeNode<K> node = head.getRight();
  TreeNode<K> child = null;
  TreeNode<K> tmp = null;

  while (node != nullNode) {

    int cmp = node.compareKeyTo(k);

    //Löschposition gefunden
    if (cmp == 0) {
      break;
    } else {
      parent = node;
      node = (cmp > 0 ? node.getLeft() : node.getRight());
    }
```

```
}

//Knoten k nicht im Baum
if (node == nullNode) {
  return false;
}

/* Nachrücker finden */
if (node.getLeft() == nullNode  && Node.getRight() == nullNode) { //Fall 1

  child = nullNode;

} else if (node.getLeft() == nullNode) { //Fall 2a

  child = node.getRight();

} else if (node.getRight() == nullNode) { //Fall 2b

  child = node.getLight();
  ...

} else { //Fall 3

  child = node.getRight();
  tmp = node;

  while (child.getLeft() != nullNode) {
    tmp = child;
    child = child.getLeft();
  }

  child.setLeft(node.getLeft());

  if (tmp != node) {
    tmp.setLeft(child.getRight());
    child.setRight(node.getRight());
  }

}

/* Baum reorganisieren */
if (parent.getLeft() == node) {
  parent.setLeft(child)
} else {
  parent.setRight(child);
```

```
  }

  return true;
  ...

}
```

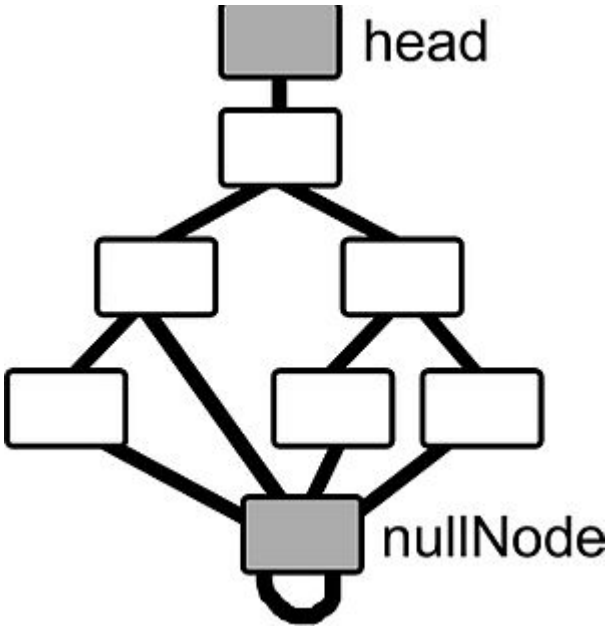
Implementierung Ein binärer Suchbaum ist eine häufig verwendete Hauptspeicherstruktur und ist besonders geeignet für Schlüssel fester Größe, z.B. numerische wie *int*, *float* und *char[n]*. Der Aufwand von $O(logn)$ für Suchen, Einfügen und Löschen ist garantiert, vorausgesetzt der Baum ist **balanciert**. Später werden wir lernen, dass die Gewährleistung der Balancierung durch spezielle Algorithmen gesichert wird. Des Weiteren sind größere, angepasste Knoten für Sekundärspeicher günstiger, diese nennt man B-Bäume. Für Zeichenketten benutzt man als Schlüssel variable Schlüsselgrößen, sogenannte Tries.

```
public class BinarySearchTree<K extends Comparable<K>> implements Iterable<K> {

  ...

  static class TreeNode<K extends Comparable<K>> {

    K key;
    TreeNode<K> left = null;
    TreeNode<K> right = null;

    ...

  }
}
```

Die Schlüssel müssen das **Comparable-Interface**, d.h. die *compareTo()-Methode*, implementieren, da der Suchbaum auf Vergleichen der Schlüssel basiert. Der Baum selbst implementiert das **Iterable-Interface**, d.h. die *iterator()-Methode*, um Traversierung des Baums über einen Iterator zu erlauben (später Baumtraversierung). *TreeNode* und alles Weitere werden als innere Klassen implementiert. Dadurch werden Zugriffe auf Attribute und Methoden der Baumklasse erlaubt. Eine Besonderheit der Implementierung sind die „leeren" **Pseudoknoten** *head* und *nullNode* zur Vereinfachung der Algorithmen (manchmal „Wächter" / „sentinel" genannt). Grundlegende Algorithmen sind:

- Suchen
- Einfügen
- Löschen

Implementierung mit Pseudoknoten



Wir vereinbaren an dieser Stelle, dass man auf dem *head* kein *getRight()* anwenden kann.

```
public class BinarySearchTree<K extends Comparable<K>> implements Iterable<K> {

  ...

  pulic BinarySearchTree() {

    head = new TreeNode<K>(null);
    nullNode = new TreeNode<K>(null);

    nullNode.setLeft(nullNode);
    nullNode.setRight(nullNode);
    head.setRight(nullNode);

  }

  ...
}
```

Das Ziel der Implementierung ist, die Reduzierung der Zahl an Sonderfällen. Im *Head* würde das Einfügen oder Löschen des Wurzelknotens spezielle Behandlung in der Baum-Klasse erfordern. Der *NullNode* erspart den Test, ob zum linken oder zum rechten Teilknoten navigiert werden kann. Des Weiteren ist im *NullNode* ein einfaches Beenden der Navigation (z.B. Beenden der Rekursion) möglich.

Weitere Aspekte

Komplexität

Die Komplexität der Operation hängt von der Höhe ab. Der Aufwand für die Höhe des Baumes beträgt $O(h)$. Die Höhe eines ausgeglichenen binären Baumes ist $h = ld(n)$ für Knoten. Bei einem ausgeglichenen oder balancierten Baum unterscheiden sich zum einen der rechte und linke Teilbaum eines jeden Knotens in der Höhe um höchstens 1 und zum anderen unterscheiden sich je zwei Wege von der Wurzel zu einem Blattknoten höchstens um 1 in der Länge. Rot-Schwarz Bäume und AVL Bäume benötigen einen Ausgleich nach dem Einfügen und Löschen.

Entartung von Bäumen

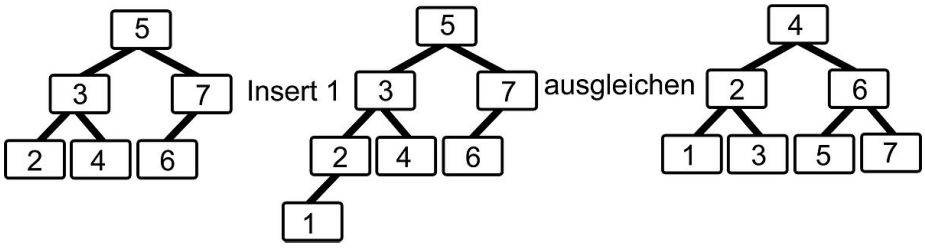
Eine ungünstige Einfüge- oder Löschreihenfolge führt zu extremer Unbalanciertheit im Baum. Im Extremfall wird der Baum zur Liste, dann haben die Operationen eine Komplexität von $O(n)$. Beispiel:

```
for (int i = 0; i < 10; i++) {
  tree.insert(i);
}
```

Vermeiden kann man dies durch spezielle Algorithmen zum Einfügen und Löschen, z.B. mit Rot-Schwarz-Bäumen und AVL-Bäumen.

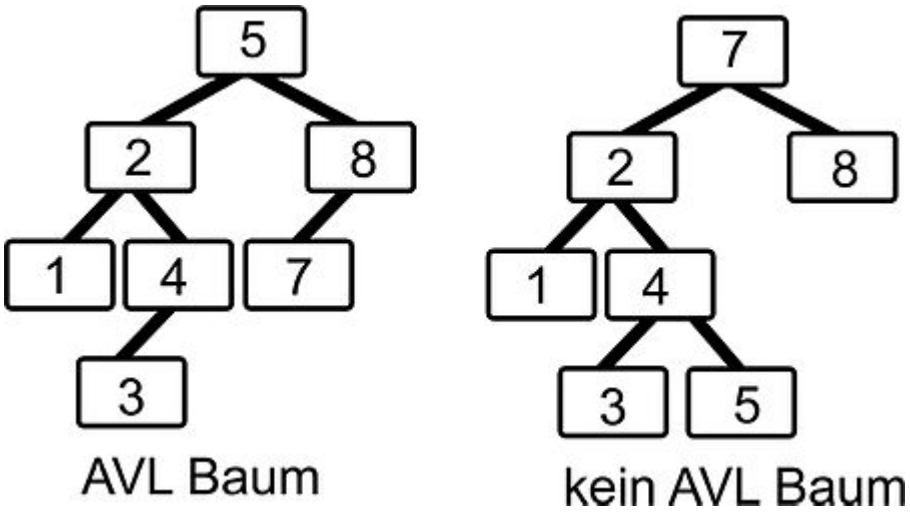
AVL-Bäume

Auf dieser Seite werden AVL Bäume behandelt. Ein Suchbaum erfordert nach Einfügen oder Löschen von Knoten einen Ausgleich, da sie sonst entarten. **AVL** steht für die russischen Mathematiker Adelson-Velskii und Landis. Liegt ein binärer Suchbaum mit AVL Kriterium vor, bedeutet das, dass für jeden inneren Knoten gilt: Die Höhe des linken und rechten Teilbaums differieren maximal um **1**. Es reicht nicht diese Bedingung nur für die Wurzel zu fordern, da beide Teilbäume der Wurzel entartet sein könnten.



Als Lösungsidee gibt es zwei Ansätze. Zum einen ein abgeschwächtes Kriterium für die ausgeglichene Höhe, beispielsweise AVL Bäume und zum anderen eine

ausgeglichene Höhe, aber ein unausgeglichener Verzweigungsgrad. Dabei können wir eine direkte Realisierung als Mehrwegbäume, beispielsweise B-Bäume, nutzen, oder eine Kodierung als binären Baum, beispielsweise Rot-Schwarz-Bäume.



Höhe von AVL-Bäumen

Wie viele Knoten k_{min} hat ein AVL Baum der Höhe h mindestens? Bei einer rekursiven Beziehung ist $k_{min}(0) = 1$, d.h der Baum hat nur eine Wurzel. Bei $k_{min}(1) = 2$, das heißt der Baum hat nur einen Zweig und bei $k_{min}(2) = 4$, das heißt der Baum hat mehr Zweige. Daraus lässt sich allgemein sagen, dass $k_{min}(n) = k_{min}(n-1) + k_{min}(n-2) + 1$. Damit wächst der Baum vergleichbar mit der Fibonacci Reihe.

$$k_{min}(h) := \begin{cases} 1 & h = 1 \\ 2 & h = 2 \\ k_{min}(h-1) + k_{min}(h-2) + 1 & h > 2 \end{cases}$$

$$Fib(n) := \begin{cases} 0 & h = 1 \\ 1 & h = 1 \\ Fib(h-1) + Fib(h-2) & h > 1 \end{cases}$$

Einfügen in AVL-Bäumen

Das Einfügen eines Schlüssels erfolgt mit den üblichen Algorithmen. Es kann aber passieren, dass danach die AVL Eigenschaft verletzt ist. Die Balance ist definiert als ***left.height-right.height***. Als AVL Eigenschaft muss die Balance $\in \{-1, 0, +1\}$ sein. Nach Einfügen ist die Balance aber $\in \{-2, -1, 0, 1, 2\}$ Reparieren kann man das Ganze mittels Rotation und Doppelrotation.

Fallunterscheidung

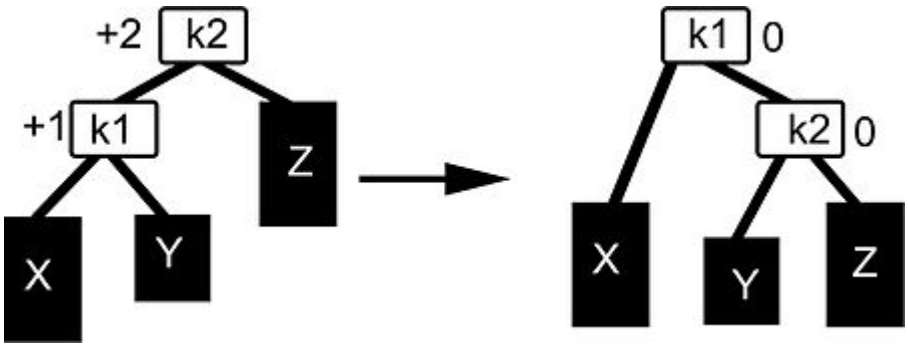
Beim Einfügen kann man in verschiedene Fälle unterteilen. Eine Verletzung der AVL Eigenschaft tritt ein bei

1. Einfügen in linken Teilbaum des linken Kindes
2. Einfügen in rechten Teilbaum des linken Kindes
3. Einfügen in linken Teilbaum des rechten Kindes
4. Einfügen in rechten Teilbaum des rechten Kindes

1 und **4** sowie **2** und **3** sind symmetrische Problemfälle.

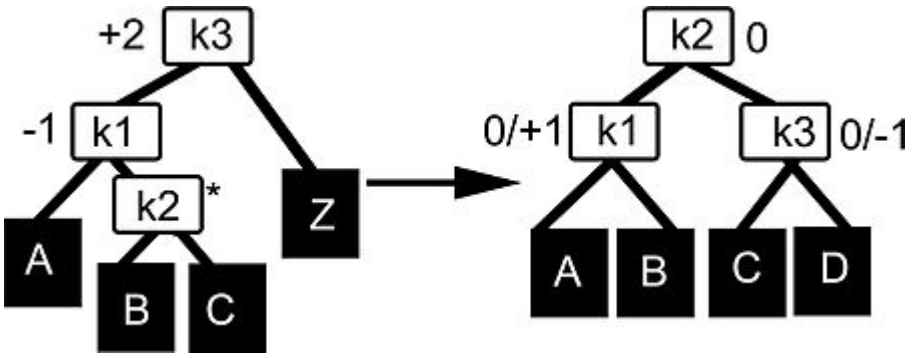
Einfache Rotation

Wir haben eine Rotation mit linkem Kind nach rechts oder rechtem Kind nach links.



Doppelrotation

Wir haben eine Doppelrotation mit linkem Kind nach rechts oder rechtem Kind nach links.



Beispiel

Im Folgenden werden wir die Rotation an einem Beispiel veranschaulichen:

- insert 3, 2, 1

  → einfache Rotation nach rechts (2,3)

- insert 4, 5

  → einfache Rotation nach links (4,3)

- insert 6

  → einfache Rotation (Wurzel) nach links (4,2)

- insert 7

  → einfache Rotation nach links (6,5)

- insert 16, 15

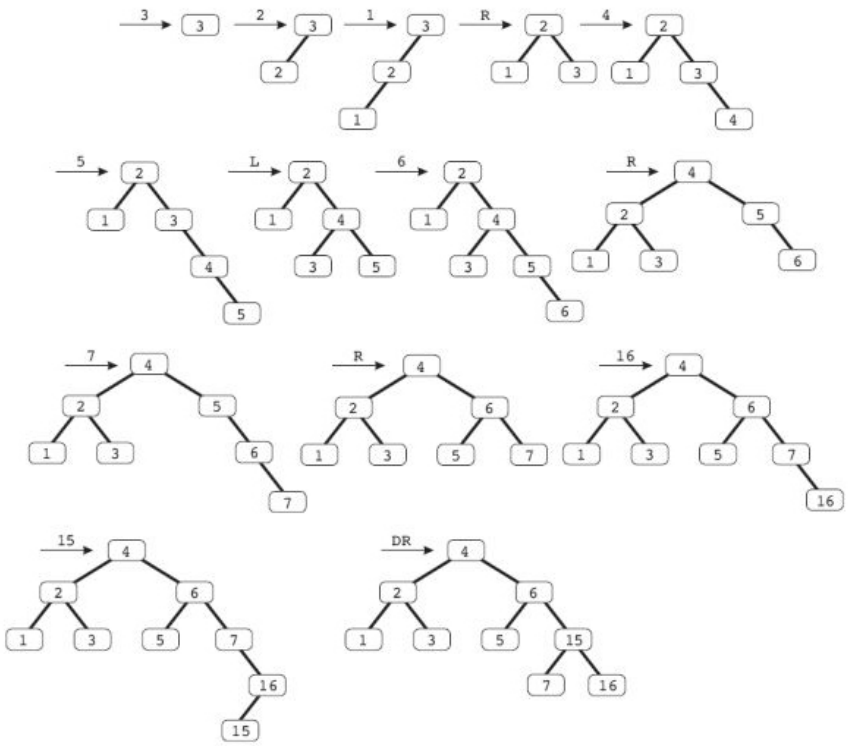
  → Doppelrotation nach links (7,15,15)

- insert 13+12+11+10

  → jeweils einfache Rotation

- insert 8, 9

  → Doppelrotation nach rechts



Eine **Verletzung der AVL-Eigenschaft** tritt ein

1. Beim Einfügen in linken Teilbaum des linken Kindes, dann muss eine Rotation mit dem linken Kind erfolgen
2. Beim Einfügen in rechten Teilbaum des linken Kindes, dann muss eine Doppelrotation mit dem linken Kind erfolgen
3. Beim Einfügen in linken Teilbaum des rechten Kindes, dann muss eine Doppelrotation mit dem rechten Kind erfolgen
4. Beim Einfügen in rechten Teilbaum des rechten Kindes, dann muss eine Rotation mit dem rechten Kind erfolgen

---

Implementation

Die Implementierung ist ähnlich des binären Suchbaums. Der Aufruf um einen neuen Knoten hinzuzufügen geschieht dabei mit ***AvlTree.insert(K k)***.

```
public class AvlTree<K extends Comparable<K>> {

  private AvlTreeNode<K> head;
  private AvlTreeNode<K> nullNode; //Neuer Knotentyp
```

```
  public AvlTree() {
    head = new AvlTreeNode<K>(null);
    nullNode = new AvlTreeNode<K>(null);

    nullNode.setLeft(nullNode);
    nullNode.setRight(nullNode);
    head.setRight(nullNode);
  }

  public boolean insert(K k){
    AvlTreeNode<K> parent = head;
    AvlTreeNode<K> child = head.getRight();

    while(child != nullNode) {
      parent = child;
      int cmp = child.getKey().compareTo(k);

      if(cmp == 0) {
        return false;
      } else if (cmp > 0) {
        child = child.getLeft();
      } else {
        child = child.getRight();
      }

    }

    AvlTreeNode<K> node = new AvlTreeNode<K>(k);
    node.setLeft(nullNode);
    node.setRight(nullNode);
    node.setParent(parent);

    if(parent.compareTo(node) > 0) {
       parent.setLeft(node);
    } else {
       parent.setRight(node);
    }

    //Nach der Einfügung die Balance ggfs. reparieren
    parent.repair(node, nullNode);
    return true;

  }
}
```

Die Knotenimplementierung ist zu Beginn genauso wie beim binären Suchbaum.

Die Methoden ***balance()*** und ***height()*** dienen als Hilfe für die Balance. Weiterhin wird die Methode ***repair()*** zur Reparatur der Balance benötigt. Sie testet, ob die Balance für den aktuellen Knoten ***(this)*** verletzt ist und repariert diese gegebenenfalls.

```
class AvlTreeNode<K extends Comparable<K>> {

  K key;
  private AvlTreeNode<K> left;
  private AvlTreeNode<K> right;
  private AvlTreeNode<K> parent;  //für jeden Knoten merken wir uns nun den Elternknoten

  public AvlTreeNode(K key){
    this.left = null;
    this.right = null;
    this.parents = null;
    this.key = key;
  }

  public AvlTreeNode<K> getLeft() { return left; }
  public AvlTreeNode<K> getRight() { return right; }
  public K getKey() { return key; }
  public void setLeft(AvlTreeNode<K> n) { left = n; }
  public void setRight(AvlTreeNode<K> n) { right = n; }
  public void setParent(AvlTreeNode<K> n) { parent = n; }

  public int compareTo(AvlTreeNode<K> other) {
    return (this.key == null) ? -1 : this.key.compareTo(other.key);
  }

  //Positive Balance, falls linker Teilbaum größer als der rechte Teilbaum
  //Negative Balance, falls rechter Teilbaum größer als der linke Teilbaum
  //Balance = 0, falls ausgeglichen
  public int balance() {

    if(this.key == null) { //Nullknoten sind stets ausgeglichen
       return 0;
    }

    return this.left.height() - this.right.height();

  }

  //Gibt die Länge des längsten Pfades zu einem Blatt wieder
  public int height() {
```

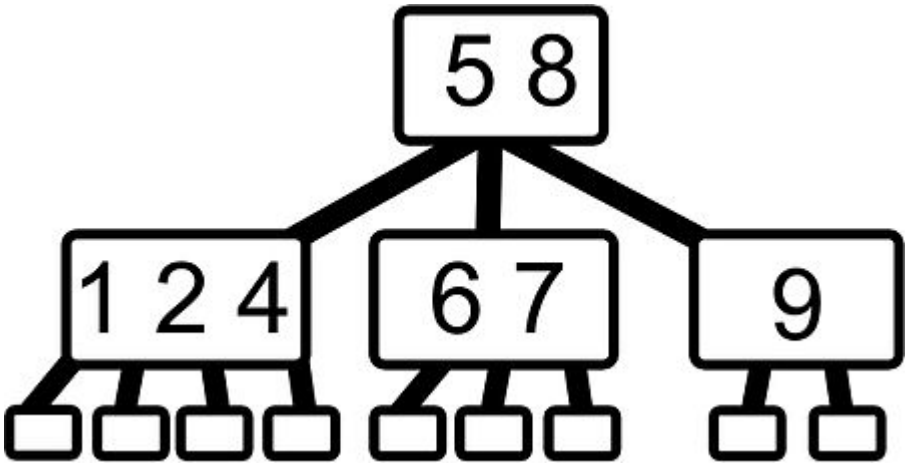
```
  if(this.key == null) { //Nullknoten haben die Höhe 0
    return 0;
  }

  return Math.max(this.left.height(), this.right.height()) + 1;
}

//Es werden die zwei nachfolgenden Knoten auf dem Pfad der Einfügung übergeben
//man startet vom Punkt der Einfügung, einem Blatt und arbeitet sich von unten nach oben
public void repair(AvlTreeNode<K> child, AvlTreeNode<K> grandchild){
  //Falls wie am head angekommen sind, terminieren wir
  if(this.key == null) {
    return;
  }

  //Fall 1: Einfügen im linken Teilbaum des linken Kindes
  if(this.balance() > 1 && child.balance() > 0) {
     child.parent = this.parent;
     this.parent = child;
     this.left = child.right;
     child.right = this;

     if(child.parent.left == this) {
       child.parent.left = child;
     } else {
       child.parent.right = child;
     }

  }

  //Fall 2: Einfügung im rechten Teilbaum des linken Kindes
  if(this.balance() > 1 && child.balance() < 0){
    grandchild.parent = this.parent;
    this.parent = grandchild;
    this.left = grandchild.right;
    this.left.parent = this;
    grandchild.right = this;
    child.right = grandchild.left;
    child.right.parent = child;
    grandchild.left = child;
    child.parent = grandchild;

    if(grandchild.parent.left == this) {
      grandchild.parent.left = grandchild;
    } else {
      grandchild.parent.right = grandchild;
```

```
      }
    }

    //Fall 3 und 4 wären analog
    //Fahre rekursiv mit Elternknoten fort
    this.parent.repair(this, child);
  }
}
```

2-3-4-Bäume

Auf dieser Seite werden die 2-3-4-Bäume behandelt. Die Idee ist ein **ausgeglichener Baum** mit **variablem Verzweigungsgrad**. Die Ausgeglichenheit wird bei der Einfügeoperation gewährleistet und die Implementierung erfolgt durch Binärbäume.

Neben binären Knoten (2-Knoten) haben wir nun auch 3-Knoten und 4-Knoten.



Operationen

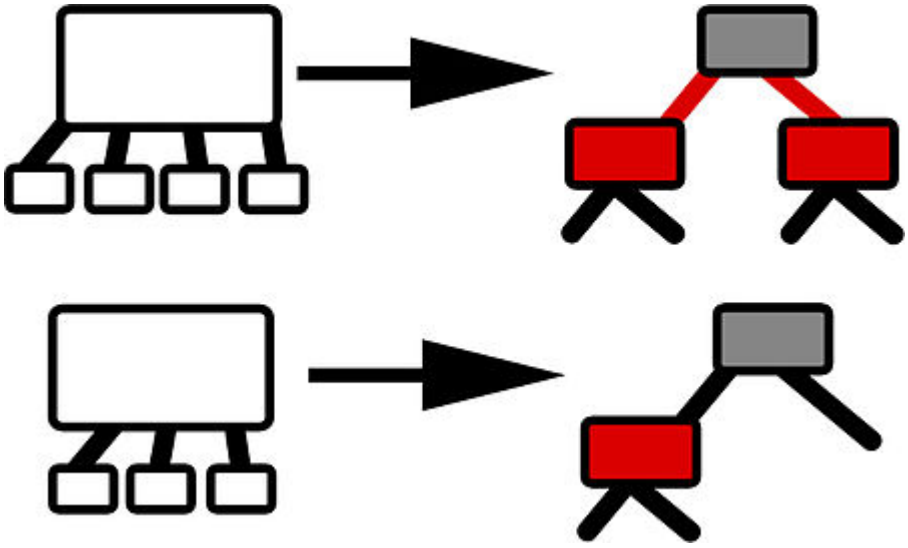
Die **Suche** in 2-3-4 Bäumen erfolgt analog zu binären Suchbäumen. Beim **Einfügen** liefert eine erfolglose Suche den Blattknoten b. Ist b ein **2-** oder **3-Knoten** wird eingefügt. Aber ist b ein **4-Knoten** dann wird aufgeteilt *(split)* und das mittlere Element nach oben gezogen. Das Splitten kann sich bis zur Wurzel fortpflanzen *(bottom-up)*.

Rot-Schwarz-Bäume

Auf dieser Seite werden die Rot-Schwarz Bäume (auch RS Bäume oder RB Bäume (englisch *"red-black tree"*) genannt) behandelt. Die Idee ist wie bei den 2-3-4

Bäumen ein ausgeglichener Baum mit variablem Verzweigungsgrad. Die Ausgeglichenheit wird bei der Einfügeoperation gewährleistet und die Implementierung erfolgt durch Binärbäume.

Binäre Repräsentation von 2-3-4 Bäumen als Rot-Schwarz Baum

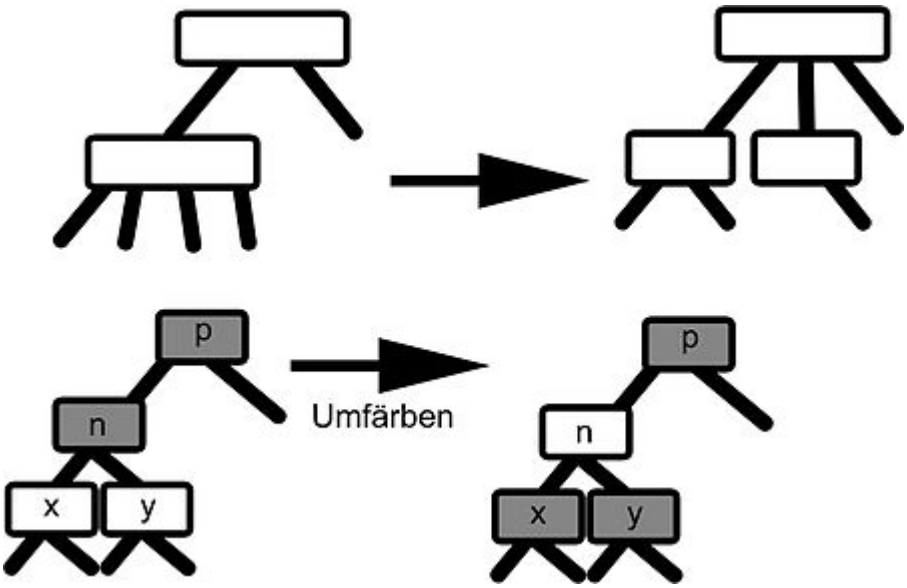


Rot-Schwarz-Bäume sind binäre Suchbäume. Jeder Knoten ist entweder rot oder schwarz. Der Wurzelknoten (Null-Knoten) ist per Definition schwarz. Die Kinder jedes roten Knotens sind schwarz. Für jeden Knoten k gilt, dass jeder Pfad von k zu einem Blatt die gleiche Anzahl schwarze Knoten enthält.

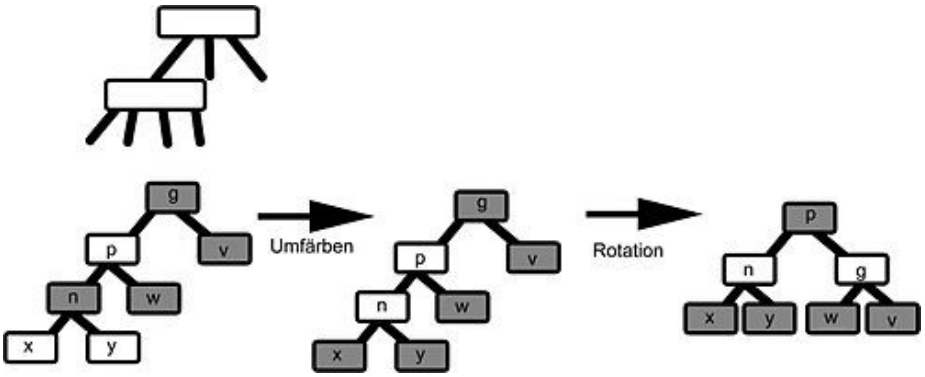
Einfügen in RB Bäume

Zuerst erfolgt **top-down** ein Splitten (nicht-rekursiv) der 4er-Knoten. Dann wird in zwei Hauptfälle unterschieden. Im **Fall 1** hängt der 4er-Knoten an einem 2er-Knoten und im **Fall 2** hängt der 4er-Knoten an einem 3er-Knoten. Hierbei gibt es noch Unterfälle. Bei **Fall 2A** hängt der Teilbaum links oder rechts. Bei **Fall 2B** hängt der Teilbaum in der Mitte.

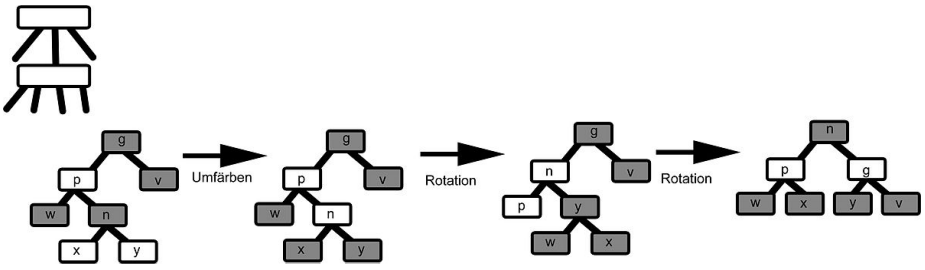
Das Splitten im RB Baum bei Fall 1



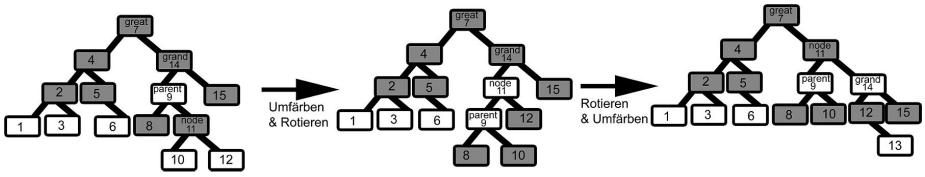
Das Splitten im RB Baum bei Fall 2a



Das Splitten im RB Baum bei Fall 2b



Beispiel Einfügen der Zahl 13



Zuerst wird die Einfügeposition gesucht. Die Knoten mit 2 roten Kindern (4-Knoten) werden auf dem Weg präventiv gesplittet. Anschließend wird ein neuer Knoten auf Blattebene erzeugt. Nun wird die Balance wieder hergestellt durch *Split* und *Rotation*.

Implementierung RB Baum

Die Datenstruktur ist weitgehend identisch mit einem „normalem" Binärbaum. Das Balance-Kriterium besagt, dass die Anzahl schwarzer Knoten auf dem Weg von jedem Blatt zur Wurzel gleich sein muss. Außerdem hat jeder schwarze Knoten 0, 1 oder 2 rote oder schwarze Kindknoten. Des Weiteren kann jeder rote Knoten nur schwarze Kindknoten haben.

Die im Folgenden vorgestellte Implementierung eines binären Suchbaumes wurde abgewandelt.

- Die Klasse **TreeNode** wird zu **RBNode** abgewandelt.
- Hinzufügen des booleschen Attributs **red**
- Keine generische Implementierung hier, d.h. **java.lang.Object** wird als Typ des Schlüssels verwendet
- Pseudoknoten **Head** und **NullNode**
- Innere Klasse für Knoten

```
public class RedBlackTree {

  static class RBNode {

    Object key;
    RBNode left = null, right = null;
    boolean red = false;
    ...
  }

  private RBNode head, nullNode;
  ...

}
```

Einfügeposition

Nach der Initialisierung wird *grand* ein Niveau tiefer gesetzt. Wenn der Knoten schon enthalten ist, wird ein *false* zurückgegeben. Ansonsten wird die Einfügeposition gesucht. In der **If** Anweisung werden präventiv die 4er Knoten aufgesplittet.

```
public boolean insert (Compareable c) {

  RBNode node, great, grand, parent;
  int cmp = 0;
  node = parent = grand = great = head;

  while (node != nullNode) {
    great = grand; grand = parent; parent = node;
    cmp = node.getKey.compareKeyTo (c);

    if (cmp == 0) {
      return false;
    } else {
      node = cmp > 0 ? node.getLeft () : node.getRight ();
    }

    if (node.getLeft().isRed() && node.getRight().isRed()) {
      split (c, node, parent, grand, great);
    }
  }
}
```

Neuen Knoten einfügen und Balancieren

Der neue Knoten wird als linker oder rechter Knoten eingefügt. Eingefügte Knoten werden als Teil eines 4er Knoten angenommen.

```
public boolean insert (Compareable c) {

  ...

  node = new RBNode (c);
  node.setLeft (nullNode); node.setRight(nullNode);

  if (parent.compareKeyTo(c) > 0) {
    parent.setLeft(node);
  } else {
    parent.setRight(node);
  }

  split(c, node, parent, grand, great);
```

```
  return true;
}
```

Der Splitvorgang

Zuerst werden die Knoten umgefärbt (Fall 1), dann wird überprüft, ob der Elternknoten ein 3er Knoten ist und ob der aktuelle Knoten und der Elternknoten gleich orientiert sind. Anschließend wird Fall 2b in Fall 2a überführt. Zum Schluss wird der geteilte Knoten umgefärbt.

```
private void split (Compare c, RBNode node, RBNode parent, RBNode grand, RBNode great) {
  node.setRed(true);
  node.getLeft().setRed(false);
  node.getRight().setRed(false);

  if (parent.isRed()) {
    grand.setRed(true);

    if (grand.compareKeyTo (c)!= parent.compareKeyTo (c)) {
      parent = rotate (c, grand);
    }

    node = rotate(c, great);
    node.setRed(false);
  }

  head.getRight().setRed(false);
}
```

Der Rotiervorgang

Zuerst wird der Nachfolgerknoten bestimmt, und eine Rotation findet rechts oder links herum statt. Zum Schluss wird die Rotation in node eingehängt.

```
private RBNode rotate(Comparable c, RBNode node) {
  RBNode child, gchild;
  child = node.compareKeyTo (c) > 0 ? node.getLeft (): node.getRight();

  if (child.getKey().compareKeyTo (c) > 0) {
    gchild = child, getLeft();
    child.setLeft(gchild.getRight());
    gchild.setRight(child);
  } else {
    gchild = child.getRight();
    child.setRight(gchild.getLeft());
    gchild.setLeft(child);
  }

  if (node.getKey().compareKeyTo(c) > 0) {
```

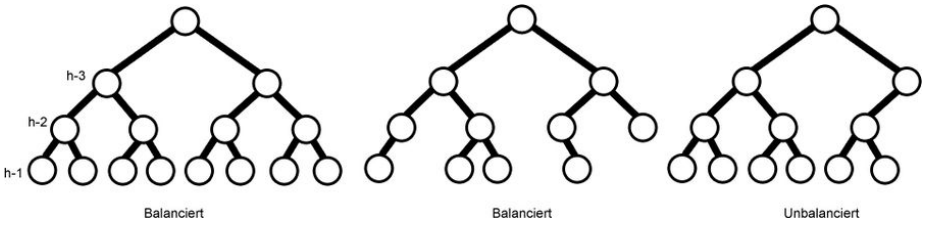
```
    node.setLeft(gchild);
  } else {
    node.setRight(gchild);
  }

  return gchild;
}
```

Heaps

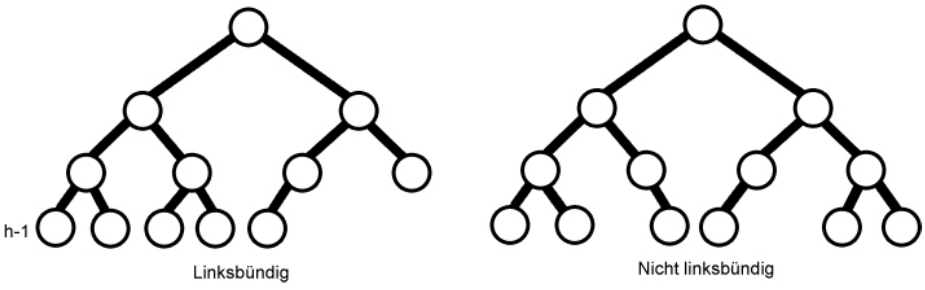
Auf dieser Seite wird das Thema Heap Sort behandelt. Von **“Heap”** gibt es zwei völlig verschiedene Definitionen. Zum einen ist es ein größeres Gebiet im Hauptspeicher, aus dem Programmierer Blöcke beanspruchen und wieder freigeben können und zum anderen ist es ein **balancierter**, **linksbündiger Binärbaum in dem kein Knoten einen Wert hat**, der größer ist als der Wert seines Elternknoten. Im Falle von Heapsort wird die zweite Definition benutzt.

Balancierter Binärbaum

Jeder Knoten ist in einer Ebene platziert, der Wurzelknoten in Ebene 0. Die Höhe eines Baumes ist die Distanz von seiner Wurzel zum weitest entfernten Knoten plus 1. Ein Knoten ist tiefer als ein anderer Knoten, wenn seine Ebene eine höhere Zahl hat. Ein Binärbaum der Höhe h ist balanciert, wenn alle Knoten der Ebenen 0 bis h-3 zwei Kinder haben.



Ein balancierter Binärbaum der Höhe h ist linksbündig, wenn er 2^k Knoten in der Ebene k hat für alle $k < h - 1$ und alle Blätter der Ebene $h - 1$ so weit wie möglich links sind.



Motivation

Der Vorteil von MergeSort gegenüber QuickSort ist, dass MergeSort einen garantierten Aufwand von $O(n*log*n)$ hat. Der Vorteil von QuickSort gegenüber MergeSort ist, dass QuickSort n viel Speicher benötigt und MergeSort $2n$ viel Speicher. Gibt es nun einen Sortieralgorithmus, der n viel Speicher benötigt und garantiert in $O(n * log * n)$ läuft? **Ja HeapSort!** Mit HeapSort lassen sich zudem die Warteschlangen mit Prioritäten effizient implementieren. Außerdem ist die Idee des Heaps sehr interessant. Eine komplexe Datenstruktur (Baum) wird in einer einfacheren Struktur (Array) abgebildet.

Heap Eigenschaft

Ein Knoten hat die Heap-Eigenschaft, wenn der Wert in dem Knoten so groß oder größer ist als die Werte seiner Kinder. Alle Blattknoten haben dann auch automatisch die Heap Eigenschaft. Ein Binärbaum ist nur dann ein Heap, wenn alle Knoten die Heap Eigenschaft besitzen.

Anmerkung

Ein Heap ist kein binärer Suchbaum. Das Sortierkriterium bei Suchbäumen war, dass der Wert eines Knotens stets größer ist, als die Werte der Knoten, die im linken Teilbaum liegen und, dass der Wert eines Knotens stets kleiner ist, als die Werte der Knoten, die im rechten Teilbaum liegen. Das **Sortierkriterium beim Heap** ist, dass die Werte eines Knotens stets größer oder gleich der Werte der Knoten sind, die in beiden Teilbäumen liegen.

Hashtabellen

Auf dieser Seite wird das Thema Hashtabellen behandelt. Gesucht ist eine dynamische Datenstruktur mit **sehr schnellem direktem Zugriff** auf Elemente. Die Idee der Hashfunktion ist, dass ein Feld von **0 bis N-1** benutzt wird, beispielsweise ein Array. Die einzelnen Positionen im Feld werden oft als **Buckets**

bezeichnet. Die Hashfunktion $h(e)$ bestimmt für Elemente e die Position im Feld. $H(e)$ ist sehr schnell berechenbar. Es gilt $h(e) \neq$ wenn $e \neq e'$.

Beispiel

Wir haben ein Array von 0 bis 9 und $h(i) = i * mod * 10$. Das Array sieht nach dem Einfügen der Zahlen 42 und 119 wie folgt aus:

Index	Eintrag
0	
1	
2	42
3	
4	
5	
6	
7	
8	
9	119

Der Vorteil von Hashing ist, dass Anfragen der Form *"Enthält die Datenstruktur das Element 42?"* schnell beantwortbar sind. Dazu verhalten sich Hashtabellen ähnlich zu binären Suchbäumen wie *BucketSort* zu vergleichsbasierten Sortierverfahren.

Hashfunktionen

Die Hashfunktionen hängen vom Datentyp der Elemente und der konkreten Anwendungen ab. Für den Datentyp *Integer* ist die Hashfunktion meist $h(i) = i * mod * N$. Das funktioniert im Allgemeinen sehr gut, wenn N eine Primzahl ist und hängt mit Methoden zur Erzeugung von Zufallszahlen zusammen. Für andere Datentypen führt man eine Rückführung auf Integer aus. Bei Fließpunkt-Zahlen werden Mantisse und Exponent einfach addiert.

Die Hashwerte sollten gut streuen. Das ist eventuell von den Besonderheiten der Eingabewerte abhängig. Beispielsweise tauchen Buchstaben des Alphabets in Namen unterschiedlich oft auf. Des Weiteren müssen die Hash-Werte effizient berechenbar sein. Ein konstanter Zeitbedarf ist erfordert, dieser ist nicht von der Anzahl der gespeicherten Werte abhängig.

Ungünstige Hashfunktionen

Als **erstes Beispiel** wählen wir $N = 2^i$ und eine generierte Artikelnummer mit den Kontrollziffern **1**,**3** oder **7** am Ende. Damit wäre die Abbildung nur auf ungeraden Adressen möglich.

Als **zweites Beispiel** wählen wir Matrikelnummern in einer Hashtabelle mit 100 Einträgen. In der ersten Variante nutzen wir die ersten beiden Stellen als Hashwert, damit kann eine Abbildung nur auf wenige Buckets erfolgen. In der zweiten Variante nutzen wir die beiden letzten Stellen und erhalten eine gleichmäßige Verteilung.

Abschlussquiz

Hinweis zur Nachnutzung