Universität Osnabrück

VL 10 | 1

Informatik B - Objektorientierte Programmierung in Java

Vorlesung 10: Collections 4

Inhalt

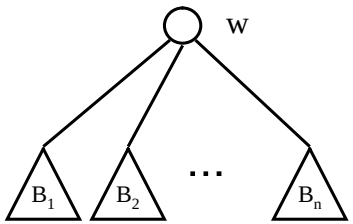
- Bäume
- Suchbäume
- Implementierung binärer Suchbäume
- Exkurs: Visitor/Visitable

Bäume

- Einführung
  - Bäume sind verallgemeinerte Listenstrukturen
  - Lineare Liste
    - Jedes Element hat höchstens einen Nachfolger.
  - Baum:
    - Jedes Element (Knoten) hat eine endliche, begrenzte Anzahl von Nachfolgern (Kindknoten).
    - Ein Knoten ohne Vorgänger (Elternknoten) heißt *Wurzel*.

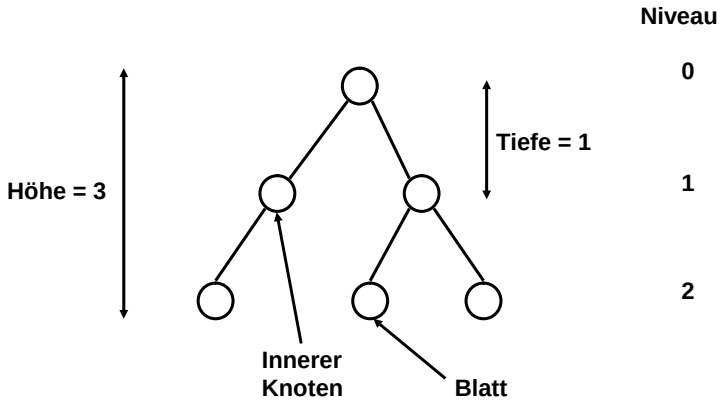
Bäume

- Definition
  - Ein einzelner Knoten ist ein Baum.
  - Sind B_1, ..., B_n für $n \geq 1$ Bäume, so erhält man einen (weiteren) Baum, indem man die Wurzeln von B_1,...,B_n zu Kindern einer neu geschaffenen Wurzel w macht.



Bäume

- Wichtige Begriffe



Bäume

- Wichtige Begriffe
  - Pfad:

    Folge von Knoten $k_0,...,k_n$ eines Baumes mit k_{i+1} ist Kind von k_i, $0 \leq i < n$
  - Binärbaum:

    Jeder Knoten besitzt maximal 2 Kinder.
  - geordneter Baum:

    Unter den Kindern eines jeden Knotens ist eine Anordnung (z.B. linkes und rechtes Kind) definiert.
  - vollständiger Baum:

    Auf jedem Niveau existiert die maximal mögliche Knotenzahl.

Suchbäume

- Definition
  - Ein binärer Suchbaum ist ein Binärbaum, der in jedem Knoten ein Paar (Schlüssel, Wert) speichert.
  - Die Schlüssel müssen aus einer linear geordneten Menge stammen (Operation $\leq$ muss definiert sein!).
  - Für einen binären Suchbaum gilt:
    - Jeder Knoten k enthält einen Schlüssel $S(k)$
    - Für alle Knoten u im linken Teilbaum des Knotens k gilt: $S(u) < S(k)$
    - Für alle Knoten u im rechten Teilbaum des Knotens k gilt: $S(u) > S(k)$

Implementierung binärer Suchbäume

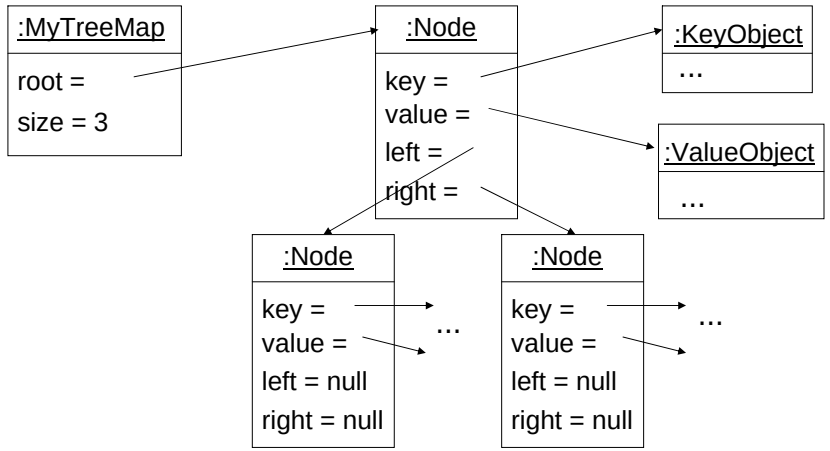
- Im folgenden wird eine Möglichkeit zur Implementierung von Suchbäumen in Java vorgestellt.
- Dies entspricht einem „Blick hinter die Kulissen“ der Java-Klassenbibliothek (Klasse `TreeMap`).
- Um die Darstellung überschaubar zu halten, wird eine Implementierung binärer Suchbäume ohne Balancierung vorgestellt.

Implementierung binärer Suchbäume

- An der Realisierung eines binären Suchbaumes in Java sind Objekte verschiedener Klassen beteiligt:
  - ein Objekt der eigentlichen Baumklasse (`MyTreeMap`)
  - ein oder mehrere Objekte der Klasse, die die Knoten des Baumes darstellen (`Node`)
  - die Objekte, die als Schlüssel im Baum gespeichert sind, und einer beliebigen Klasse angehören, die das Interface `Comparable` implementiert (hier als `KeyObject` bezeichnet)
  - die Objekte, die als Werte im Baum gespeichert sind, und einer beliebigen von `Object` abgeleiteten Klasse angehören (hier als `ValueObject` bezeichnet)

Implementierung binärer Suchbäume

- Aufbau



Implementierung binärer Suchbäume

- Klasse `MyTreeMap`
  - verwaltet einen Verweis auf den Wurzelknoten des Binärbaumes

```
public class MyTreeMap
{
  // Attribute
  private Node root; // Verweis auf Wurzelknoten
  private int size;  // Anzahl der Knoten

  public MyTreeMap() {
       this.root = null;
       this.size = 0;
  }
  ...
}
```


Implementierung binärer Suchbäume

- Klasse `Node`
  - Innere Klasse der Klasse `MyTreeMap`, da die Knoten nur im Kontext des Binärbaumes einen Sinn ergeben:

```
public class MyTreeMap
{
  class Node
  {
    // Attribute
    Comparable key; // Verweis auf Schlüssel-Objekt
    Object value;   // Verweis auf Wert-Objekt
    Node left;      // Verweis auf linken Kindknoten
    Node right;     // Verweis auf rechten Kindknoten

    Node(Comparable key, Object value)
    {...}
    public String toString()
    {...}
  }
...}
```

Implementierung binärer Suchbäume

- Suchen
  - Aufgabe
    - Suche den Knoten im Suchbaum, der einen vorgegebenen Schlüssel enthält
  - Lösungsansatz
    - Einfache Fälle werden direkt gelöst.
    - Komplizierte Fälle werden auf einfache Fälle reduziert.
  - Fallunterscheidung:
    - Ist der Baum leer, so ist der Schlüssel nicht enthalten.
    - Ist der Schlüssel gleich dem Schlüssel der Wurzel, ist der gesuchte Knoten gefunden.
    - Ist der Schlüssel kleiner als der Schlüssel der Wurzel, suche nach dem gleichen Prinzip im linken Teilbaum
    - Ist der Schlüssel größer als der Schlüssel der Wurzel, suche nach dem gleichen Prinzip im rechten Teilbaum

MyTreeMap.java

```
private Node getNodeRec(Comparable key, Node tree)
{
   Node result;
   if (tree==null)    // Baum ist leer
      result = null;
   else {
      int cmp = key.compareTo(tree.key);
      if (cmp==0)     // Knoten ist gefunden
         result = tree;
       else if(cmp<0)  // Suchen im linken Teilbaum
         result = getNodeRec(key, tree.left);
      else            // Suchen im rechten Teilbaum
         result = getNodeRec(key, tree.right);
      }
   return result;
}
```


Implementierung binärer Suchbäume

- Suchen
  - Mit Hilfe der Suche nach einem Knoten, der einen vorgegebenen Schlüssel enthält, kann die Suche nach dem zugehörigen Wert realisiert werden:

```
Object get(Object key) {
   Object result = null;

   Node p = getNodeRec(key, this.root);
   if (p != null)
      result = p.value;

   return result;
}
```


Implementierung binärer Suchbäume

- Einfügen
  - Aufgabe
    - Füge ein Paar (Schlüssel, Wert) in den Suchbaum ein
  - Fallunterscheidung:
    - Ist der Baum leer, so erzeuge einen neuen Baum mit einem Knoten, der das Paar (Schlüssel,Wert) enthält.
    - Ist der Schlüssel gleich dem Schlüssel der Wurzel, ersetze den alten durch den neuen Wert.
    - Ist der Schlüssel kleiner als der Schlüssel der Wurzel, füge das Paar in den linken Teilbaum ein.
    - Ist der Schlüssel größer als der Schlüssel der Wurzel, füge das Paar in den rechten Teilbaum ein.

Implementierung binärer Suchbäume

```
Node insertNode(Comparable key, Object value, Node tree)
{
  if (tree==null) { // Neuen Knoten erzeugen
    tree = new Node(key, value);
    this.size++;
  }
  else {
    int cmp = key.compareTo(tree.key);
    if (cmp==0) tree.value = value; // Austauschen
    else if(cmp<0) // Einfügen im linken Teilbaum
      tree.left = insertNode(key,value, tree.left);
    else          // Einfügen im rechten Teilbaum
      tree.right= insertNode(key,value,tree.right);
  }
  return tree;
}
```


Implementierung binärer Suchbäume

- Einfügen
  - Mit Hilfe der Operation `insertNode` kann das Einfügen in den Suchbaum realisiert werden:

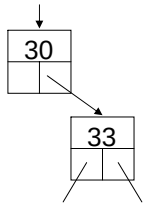
```
void put(Comparable key, Object value)
{
  if (key!=null && value!=null)
   this.root = insertNode(key, value, this.root);
}
```

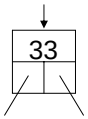

Implementierung binärer Suchbäume

- Entfernen
  - Aufgabe:
    - Entfernen eines Knotens zu einem vorgegebenen Schlüssel, falls dieser vorhanden ist.
  - Fallunterscheidung:
    - Ist der Baum leer, so kann kein Knoten mit dem Schlüssel entfernt werden.
    - Ist der Schlüssel gleich dem Schlüssel der Wurzel, entferne die Wurzel aus dem Baum.
      → weitere Fallunterscheidung notwendig!
    - Ist der Schlüssel kleiner als der Schlüssel der Wurzel, entferne den Schlüssel im linken Teilbaum.
    - Ist der Schlüssel größer als der Schlüssel der Wurzel, entferne den Schlüssel im rechten Teilbaum.

Implementierung binärer Suchbäume

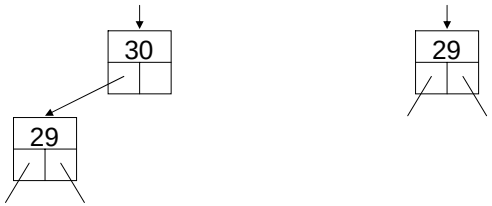
- Weitere Fallunterscheidung:
  - Fall 1: Besitzt die Wurzel keinen linken Kindknoten, so ersetze die Wurzel durch den rechten Kindknoten





Implementierung binärer Suchbäume

- Weitere Fallunterscheidung:
  - Fall 2: Besitzt die Wurzel keinen rechten Kindknoten, so ersetze die Wurzel durch den linken Kindknoten

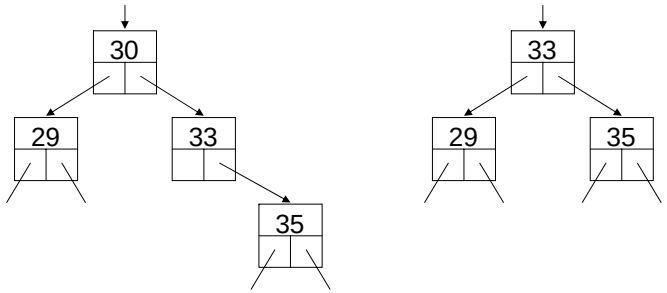


Implementierung binärer Suchbäume

- Weitere Fallunterscheidung:
  - Besitzt die Wurzel w einen linken und rechten Kindknoten, so ersetze w durch den symmetrischen Nachfolger.
  - Der symmetrische Nachfolger ist der Knoten mit dem kleinsten Schlüssel, der größer als der Schlüssel von w ist.
  - Der symmetrische Nachfolger kann keinen linken Kindknoten besitzen und kann somit leicht entfernt werden.
  - Der symmetrisch Nachfolger ist der am weitesten links stehende Knoten im rechten Teilbaum von w.

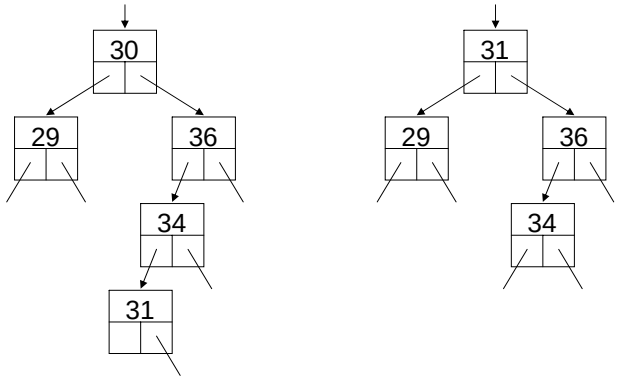
Implementierung binärer Suchbäume

- Weitere Fallunterscheidung:
  - Fall 3: Der rechte Kindknoten der Wurzel besitzt keinen linken Kindknoten und ist somit der symmetrische Nachfolger.



Implementierung binärer Suchbäume

- Weitere Fallunterscheidung:
  - Fall 4: Der rechte Kindknoten der Wurzel besitzt einen linken Kindknoten



Implementierung binärer Suchbäume

- Bestimmung des Elternknotens des symmetrischen Nachfolgers:

```
private Node parentSymSucc(Node tree)
{
   Node result;

   result = tree;
   if (result.right.left!=null)
   {
      result = result.right;
      while (result.left.left!=null)
        result = result.left;
   }

   return result;
}
```


Implementierung binärer Suchbäume

- Entfernen des Knotens:

```
private Node removeNode(Comparable key, Node tree)
{
  if (tree!=null) {
     int cmp = key.compareTo(tree.key);
     if (cmp<0)      // Entf. im linken Teilbaum
       tree.left = removeNode(key, tree.left);
     else if (cmp>0) // Entf. im rechten Teilbaum
       tree.right = removeNode(key, tree.right);
     else
     { // zu entfernender Knoten gefunden
       this.size--;
       if (tree.left==null)       // Fall 1
         tree = tree.right;
       else if (tree.right==null) // Fall 2
         tree = tree.left;
```


Implementierung binärer Suchbäume

```
       else {
         Node p = parentSymSucc(tree);
         if (p==tree) { // Fall 3
           tree.key = p.right.key;
           tree.value = p.right.value;
           p.right = p.right.right;
         }
         else {         // Fall 4
           tree.key = p.left.key;
           tree.value = p.left.value;
           p.left = p.left.right;
         }
       }
     }
   }
   return tree;
}
```


Implementierung binärer Suchbäume

- Baumdurchlauf
  - Um die Einträge eines Suchbaumes aufsteigend sortiert auszugeben, müssen die Knoten des Baum in einer geeigneten Reihenfolge durchlaufen werden.
  - Da die Knoten u im linken Teilbaum, kleinere Schlüssel als der Knoten k besitzen, müssen diese vor k ausgegeben werden.
  - Da die Knoten u im rechten Teilbaum, größere Schlüssel als der Knoten k besitzen, müssen diese nach k ausgegeben werden.
  - Diese Strategie wird auch als Inorder-Durchlauf bezeichnet.

Implementierung binärer Suchbäume

```
private void inorderPrint(Node tree)
{
  if (tree!=null) {
     inorderPrint(tree.left);
     System.out.println(tree);
     inorderPrint(tree.right);
  }
}
void print()
{
     System.out.println("Liste der Einträge");
     System.out.println("------------------");
     inorderPrint(this.root);
     System.out.println();
}
```


Exkurs: Visitor-Pattern

- Konzept eines Visitors
  - Ganz allgemein kann man von einer Collection verlangen, dass einem Besucher (Visitor) alle Elemente genau einmal vorgestellt werden.
  - Interface **`Visitable`**: Methode zum Anliefern eines Visitors bei der Collection.
    Ein beliebiges Objekt ist ein *Visitor*, wenn ihm Objekte vorgeführt werden können.
  - Interface **`Visitor`**: Methode, mit der die Collection dem Visitor ihre Insassen vorstellt. Eine Collection ist *Visitable*, wenn sie Besucher empfangen kann.

Interfaces Vistable und Visitor

```
public interface Visitable {
  /** receive a visitor, manage the visit.
      @return true if the visitor always replies true.
    */
  boolean visit (Visitor v);
}


public interface Visitor {
  /** visit an object.
      @return true to continue visiting.
    */
  boolean visit (Object x);
}
```


Exkurs: Visitor-Pattern

- Konzept eines Visitors (2)
  - Ausbaumöglichkeit: **`visit()`** in **`Visitor`** könnte mit verschiedenen Signaturen verschiedene Arten von Insassen unterscheiden.
  - Beim Suchbaum hat man die schöne Möglichkeit, die Einträge (Knoten) sortiert auszugeben. Beispielsweise kann man Wörter in beliebiger Reihenfolge einlesen und sortiert ausgeben.
  - Ein Suchbaum sollte *Visitable* sein, um die Sortierung ausnutzen zu können.
  - Implementiert der Suchbaum selber einen *Visitor*, kann man den Baum sehr elegant mit der Methode **`toString`** darstellen lassen („dumpen“).

Exkurs: Visitor-Pattern

- Konzept eines Visitors (3)
  - Die **`visit()`**-Methoden haben als Resultat-Wert **`boolean`**. Über das Resultat der Methode in **`Visitor`** kann der Besucher steuern, ob die Traverse fortgesetzt werden soll (Besucher sagt: “weiter” oder “ich will nicht mehr”).

Exkurs: Visitor-Pattern

- Suchbaum mit Visitor
  - In **`MyVisitableTreeMap`** wird eine inorder-Traversierung realisiert.
  - **`visit()`**-Methode in **`MyVisitableTreeMap`** schickt *Visitor* zur Wurzel des Baums
  - **`inorderVisit()`** schickt *Visitor* zum übergebenen Knoten und realisiert Inorder-Traversierung.
  - In **`inorderVisit()`** wird geprüft, ob an der aktuellen Position noch ein Unterbaum existiert. Wenn ja, wird die **`inorderVisit()`**-Methode für beide Unterbäume aufgerufen: Hier wird die Inorder-Traverse realisiert. Das Objekt **`tree`** (der aktuelle Knoten) wird an die **`visit()`**-Methode des *Visitors* übergeben.

Beispiel: Suchbaum mit Visitor

```
public class MyVisitableTreeMap implements Visitable
{
   ...
   public boolean visit(Visitor v)
   {
        return inorderVisit(this.root, v);
   }

   private boolean inorderVisit(Node tree, Visitor v)
   {
        if (tree!=null)
        {
               return inorderVisit(tree.left, v) &&
                           v.visit(tree) &&
                           inorderVisit(tree.right, v);
        }
        return true;
   }
...}
```


Beispiel: Test des Suchbaums

```
public class VisitableTreeTest
{
   public static class PrintVisitor implements Visitor
   {
        public boolean visit (Object o)
        {
               System.out.println(o);
               return true;
        }
   }
   public static void main(String[] args)
   {
        MyVisitableTreeMap dictionary =
                                    new MyVisitableTreeMap();
        ...
        dictionary.visit(new PrintVisitor());
        ...
   }
}
```

Exkurs: Visitor-Pattern

- Suchbaum mit Dumper
  - Für **`MyVisitableTreeMap`** implementiert man **`toString()`** mit einem **`Dumper`**-Objekt vom Typ **`Visitor`**, das jeden Insassen in seinen **`StringBuffer`** einträgt und diesen bei **`toString()`** (von **`Dumper`**) als seine eigene Darstellung abliefert.
  - **`Dumper`** ist ein Beispiel für eine lokale Klasse.
  - In **`toString()`** wird die **`visit()`**-Methode von **`MyVisitableTreeMap`** aufgerufen, der ein **`Dumper`**-Objekt übergeben wird.
  - Damit ist dann der folgende Aufruf aus dem Testprogramm möglich, der den Suchbaum „dumpt“:

```
System.out.print(dictionary.toString());
```


Beispiel: Suchbaum mit Dumper

```
public class MyVisitableTreeMap implements Visitable
{  ...
   public String toString () {
       class Dumper implements Visitor {
              protected StringBuffer buf = new StringBuffer();
              public boolean visit (Object o)
              {
                     buf.append('\n').append(o);
                     return true;
              }
              public String toString ()
              {
                     return buf.toString();
              }
       }             Dumper d = new Dumper();
       visit(d);
       return d.toString();
   }
}
```