Programmierung 2
Studiengang MI / WI

Dipl.-Inf., Dipl.-Ing. (FH) Michael Wilhelm
Hochschule Harz
FB Automatisierung und Informatik
mwilhelm@hs-harz.de
Raum 2.202
Tel. 03943 / 659 338

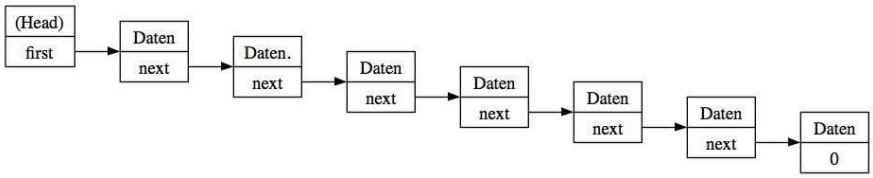
Inhalt der Vorlesung

Überblick:

- Objekte und Methoden
- Swing
- Exception
- I/O-Klassen / Methoden
- Threads
- **Algorithmen (Das Collections-Framework)**
- Design Pattern
- Graphentheorie
- JUnit

Verkettete Listen

- Definition (Verkettete Liste)
  - Eine verkettete Liste ist eine Menge von Objekten, bei der jedes Objekt Element in einem Node ist. Jeder Node enthält auch eine Referenz auf seinen folgenden Node (oder: auf den nächsten Node).
  - Zugriff ist O(N),
  - Einfügen ist O(1)
- Grafische Darstellung



Aufbau einer verketteten Liste: Implementierung

```
public class LinkedList<T> {

  private class Node {
  }
  private Node head;
}
```

```
private class Node {
        private T data;
        private Node next;

        public Node(T Daten)
        {       data = Daten;  }

        public Node getNachfolger()
        {       return next;        }

        public void setNachfolger(Node n)
        {       next = n;           }

        public T getDaten()
        {       return data;        }
} // class Node
```

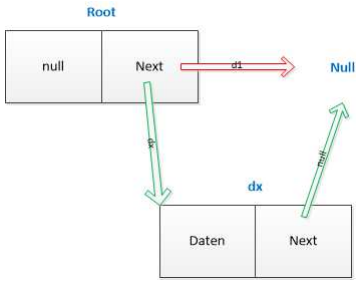
Operationen auf einer Liste

- **Initialisierung:**
  - Node<T> head = new Node<T>();
  - head.next = null;
- **Einfügen von Node t nach Node x:**
  - t.next = x.next ;
  - x.next = t ;
- **Löschen des auf x folgenden Node:**
  - x.next = x.next.next ;
- **Traversierung:**
  - for (Node<T> t=head; t != null; t = t.next)
- **Test auf leere Liste:**
  - if (head == null)

Java LinkedList

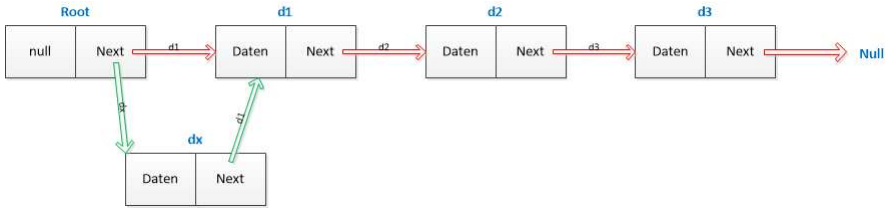
- **Initialisierung:**
  - List<Object> list = new LinkedList<Object>();
- **Einfügen vor Index i:**
  - Object o = . . . ;
  - list.add (i, o);
- **Löschen bei Index i**
  - list.remove (i);
- **Traversierung:**
  - Iterator<Object > t = list.iterator();
  - while (t.hasNext()) {
    - Object o = t.next ();
  - }
- **Teste leere Liste:**
  - if (list.isEmpty())

Grafische Darstellung einer verketteten Liste:



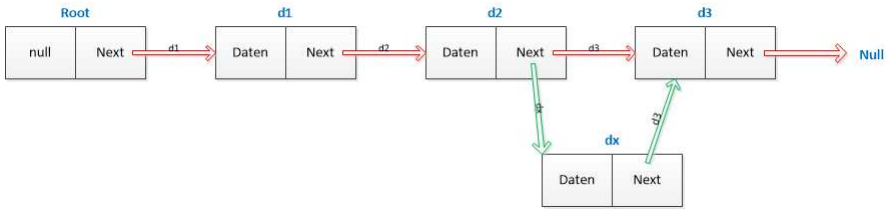
1) Einfügen eines neuen Knotens: Liste war leer

Grafische Darstellung einer verketteten Liste:



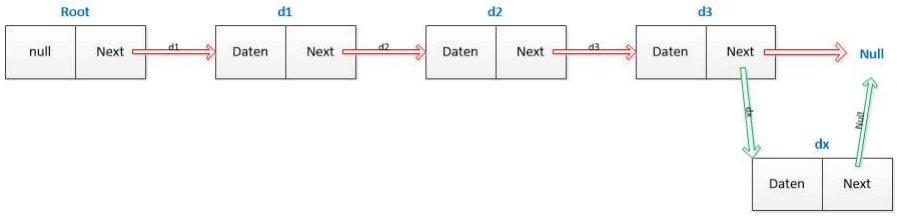
2) Einfügen eines neuen Knotens am Anfang

Grafische Darstellung einer verketteten Liste:



3) Einfügen eines neuen Knotens in der Mitte

Grafische Darstellung einer verketteten Liste:



4) Einfügen eines neuen Knotens am Ende

Grafische Darstellung einer verketteten Liste:

Löschen des Knotens d_3

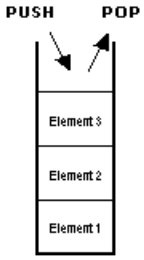
Löschen eines Knotens in einer verketteten Liste:

Unterschiedliche Fälle:

1. Die Liste ist leer.
2. Der erste Knoten der Liste soll gelöscht werden.
3. Der zu löschende Knoten liegt in der „Mitte“.
4. Der Knoten liegt am Ende der Liste.
5. Der Knoten ist nicht in der Liste

Stacks (Stapel)

- Ein Stack ist, wie der Name schon verrät, eine Art Stapel, d.h. man kann Elemente nur nach dem LIFO-Prinzip (Last in, First out) von oben wegnehmen (pop) bzw. drauflegen (push).
- Im Klartext heißt das: Ein Stack ist eine Liste, zu der man Elemente nur vorne einfügen kann und nur von vorne wegnehmen kann. Die einzigen Methoden sind:
  - push (drauflegen)
  - pop (wegnehmen)
  - get (das oberste Element anschauen)



Realisierung eines Stacks

- Verkettete Listen eignen sich gut zur Implementierung eines Stacks.
- Die beiden Methoden, push und pop sind einfach implementierbar:

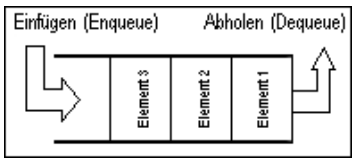
```
push(T element){
   Node<T> n = new Node<T>();
   n.data = element;
   n.next = head;
   head = n;
}

T pop() {
   T element = null;
   if ( head != null) {
      element = head.data;
      head = head.next;
   }
   return element;
}
```

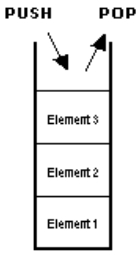
Warteschlange (engl. Queue)

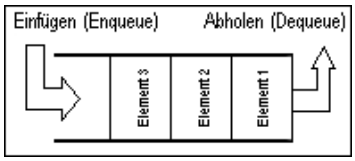
- Ein Stack ist eine LIFO (Last in, first out) Datenstruktur. Im Gegensatz hierzu handelt es sich bei einer Warteschlange, Queue, um eine FIFO (first in, first out) Datenstruktur. Auch diese kann effizient mittels einer verketteten Liste implementiert werden:

```
public void add (T element) {
   if (head != null) {
      Node<T> n = head;
      while ( n.next != null ) n = n.next;
      n.next = new Node<T>();
      n = n.next;
      n.data = element;
   } else {
      head = new Node<T>();
      head.data = element;
   }
}
public T remove() {
   T element = null;
   if ( head != null ) {
      element = head.data ;
      head = head.next;
   }
   return element;
}
```



Unterschied Stack vs. Queue





Eingetragen:	**Entfernt:**	**Eingetragen:**	**Entfernt:**
1 (1. Eintrag)	44	1 (1. Eintrag)	1
66	33	66	66
33	66	33	33
44	1	44	44

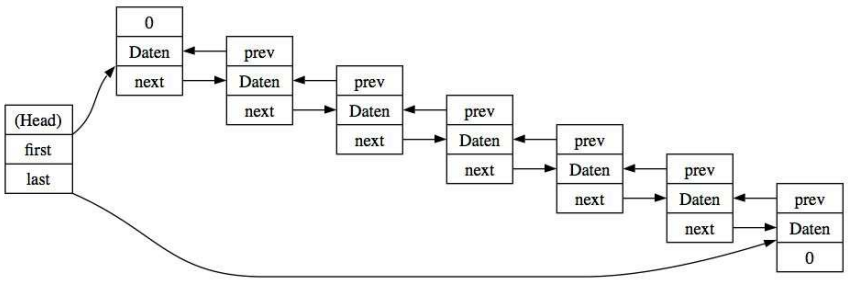
Spezielle Warteschlange: Deque

- Deque = (Stack und Queue)
  - Eine Deque ist eine Warteschlange bei der Elemente sowohl am Anfang wie am Ende entfernt und hinzugefügt werden können.
  - Sie wird am einfachsten mittels einer doppelt verketteten Liste implementiert.
- Java
  - LinkedList implementiert die Deque Schnittstelle.

Eingetragen:	**Entfernt:**	**Entfernt:**
• 1 (1. Eintrag)	• 1	• 44
• 66	• 66	• 1
• 33	• 33	• 66
• 44	• 44	• 33

Doppelt verkettete Listen

- Definition
  - Ein Node einer doppelt verketteten Liste kennt nicht nur seinen Nachfolger, sondern auch seinen Vorgänger.



Doppelt verkettete Listen

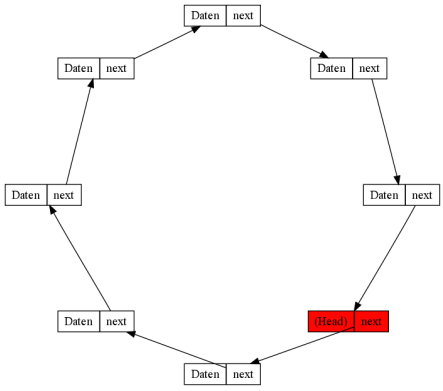
- Implementation

```
class DoubleLinkedList<T> {
   private class Node {
      public T data;
      public Node prev;
      public Node next;
   }
   Node root;
}
```

- Java
  - LinkedList implementiert eine doppelt verkettete Liste. Im ListIterator stehen daher auch folgende Methoden zur Verfügung:
  - boolean hasPrevious() stellt fest, ob ein Vorgänger vorhanden ist.
  - T getPrevious() positioniert den Cursor auf den Vorgänger und gibt eine Referenz auf dessen Dateninhalte zurück.

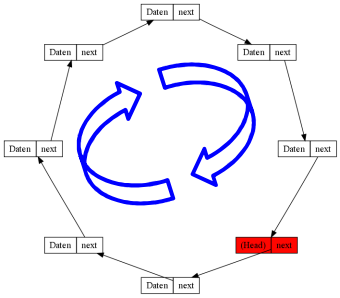
Spezielle Listen

- Zirkuläre Liste
  - Enthält die “letzte” Referenz auf einen Nachfolger anstatt der null eine Referenz auf das erste Element, so heißt die Liste zirkulär



Spezielle Listen

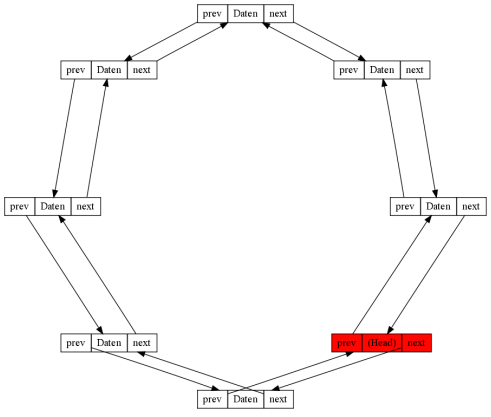
- Fragen
  - Wie erkennt ein Iterator in einer zirkulär verketteten Liste, wann alle Elemente abgearbeitet sind?
  - Also, dass es keine Endlos-Schleife gibt.
- Antwort:




```
if ( head != null ) {
  Node s = head;
  do {
        . . .
        s = s.next;
  } while ( s != head );
}
```

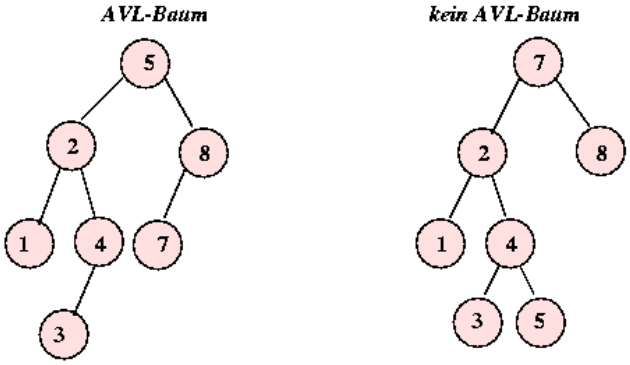

Spezielle Listen

- Doppelt-Zirkuläre Liste
  - Letzte Element hat nicht **null**, sondern zeigt auf das erste Elemente
  - Das erste Element hat für den Vorgänger nicht null, sondern es zeigt auf das letzte Element



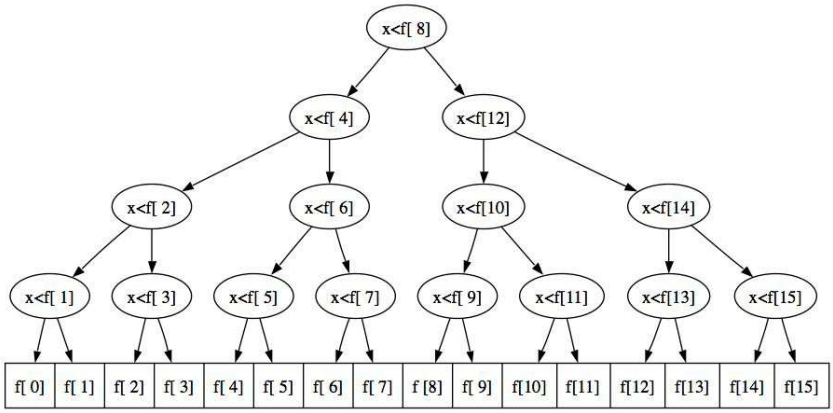
Bäume

- Ein Baum ist eine Datenstruktur, in dem ein Knoten seine „Kinder“ kennt.
- Ein Baum hat einen Wurzelknoten und „n“ Kinder



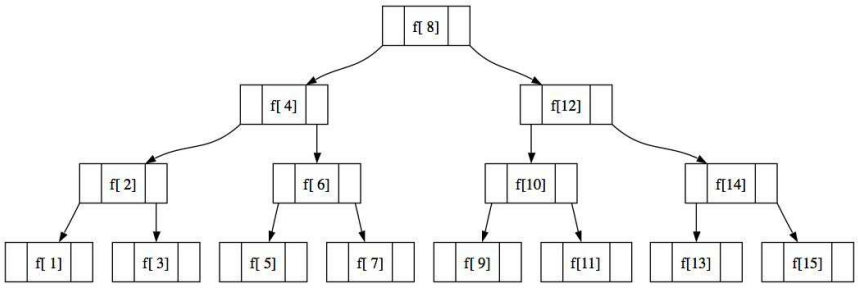
Binäre Suche

- Idee bereits bekannt durch binäre Suche



Binärbaum

- Kombination der binären Suche mit den Nodes einer verketteten Liste führt zu einem explizitem Binärbaum



Implementierung von Bäumen

- Allgemein:
  - Datenstruktur bestehend aus
    - Baum,
    - Baumknoten
    - und den vom Baum organisierten Datenobjekten
- Unterscheidungen nach:
  - Verzweigungsgrad des Baumes
  - geordnetem oder ungeordnetem Baum
  - Anzahl Knotentypen
  - Suchbäume
  - Verzweigung der Daten
  - Vorgehensweise für Einfügen, Löschen, etc.

Beispiel BTree

```
class BTree<T extends Comparable<T>> {
   private class Node<T> {
       public T content ;
       public Node<T> left, right;
   }

   private Node root = new Node();
   public BTree (T o) {
       root.content = o;
   }
   public void insert (T c) {
   // Implementation unten
   }
   public void traverse () {
   // Implementation unten
   }
```


Beispiel BTree

```
public void insert (T c) {
  for (Node node = root ; node != null; ) {
      if ( node.content == null) {
         node.content = c;
         break ;
      }
      int cmp = node.content.compareTo(c);
      if (cmp > 0) {
         if (node.left == null) node.left=new Node();
         node = node.left;
      }
      else if (cmp < 0) {
         if (node.right == null) node.right=new Node();
         node = node.right;
      }
      else break;
  } // for
}
```

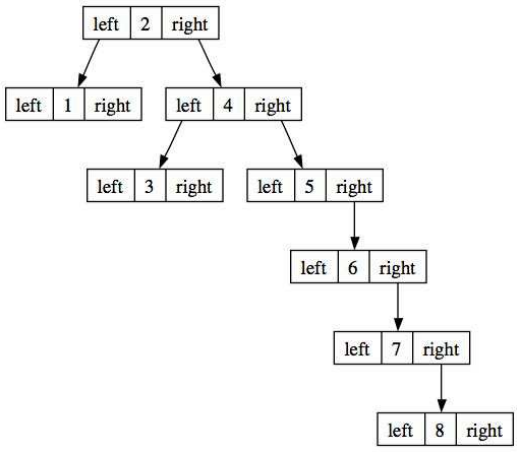
Beispiel BTree

```
class MyObject implements Comparable<MyObject> {
   private int i;
   public MyObject (int m) { i = m; }
   public int compareTo(MyObject o) { return i – o.i; }
   public String toString() { return Integer.toString(i);
     }
}

public class BTreeTest {
   public static void main ( String[ ] args ) {
     BTree<MyObject> btree =
          new BTree<MyObject>(new MyObject(2));
          btree.insert (new MyObject(1));
      for ( int i =4; i <9; ++ i )
          btree.insert (new MyObject(i));
     btree.insert (new MyObject(3));
     System.out.println (btree);
   }
}
```


BTree

- Beachte: die Form, und damit die Effizienz des binären Baumes hängt von der Reihen des Einfügens ab.



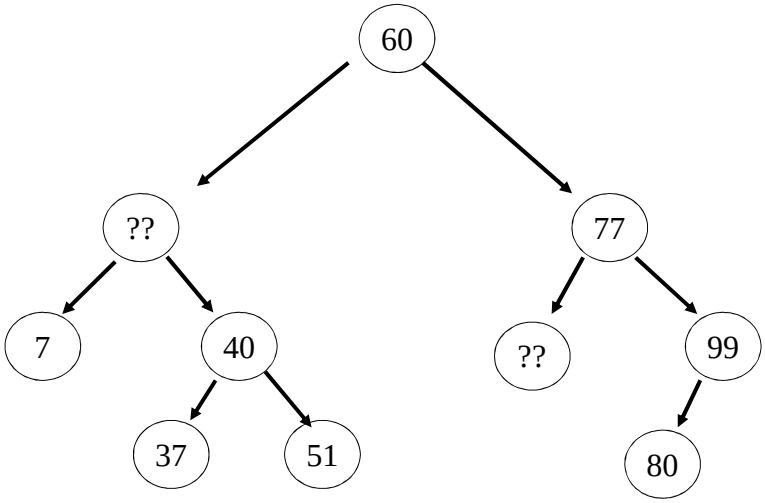
Baumoperationen

- **Einfügen (siehe später: AVL-Baum)**
- **Löschen eines Nodes. Es sind drei Fälle zu beachten:**
  - Der Node ist ein Blatt (keine Kinder): entferne Referenz in Elter.
  - Der Node besitzt nur einen Kind-Node: Lenke Verweis auf Node in Elter auf Kind-Node um.
  - Der Node besitzt zwei Kinder: Der Node muss durch den am weitesten links stehenden Node des rechten Subbaumes ersetzt werden.

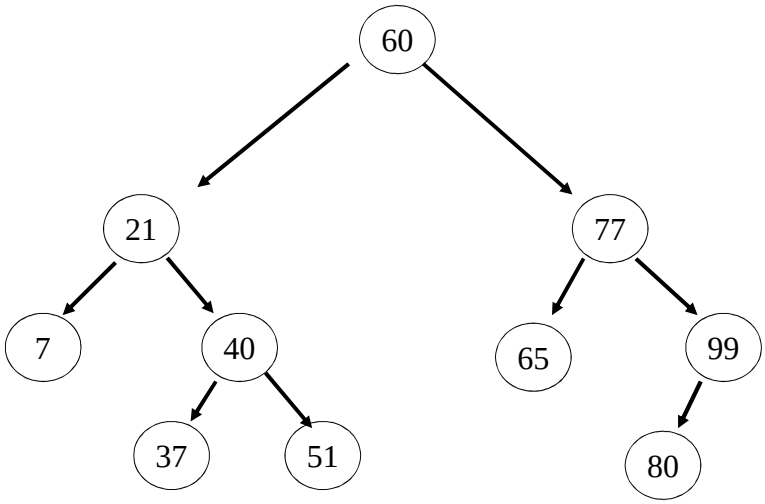
Baumoperationen

- **Traversierung: Mehrere Möglichkeiten:**
  - inorder: erst rekursiv der linke Subbaum, dann den Node bearbeiten, dann rekursiv den rechten Subbaum.
  - Preorder: erst den Node bearbeiten, dann jeweils rekursiv den linken und rechten Subbaum
  - Postorder: erst jeweils rekursiv die linken und rechten Subbäume, dann den Node bearbeiten.
  - Levelorder: Erst alle Nodes eines Levels bearbeiten bevor auf einen tieferen Level übergegangen wird. Benötigt extra Speicher in Form einer Warteschlange.

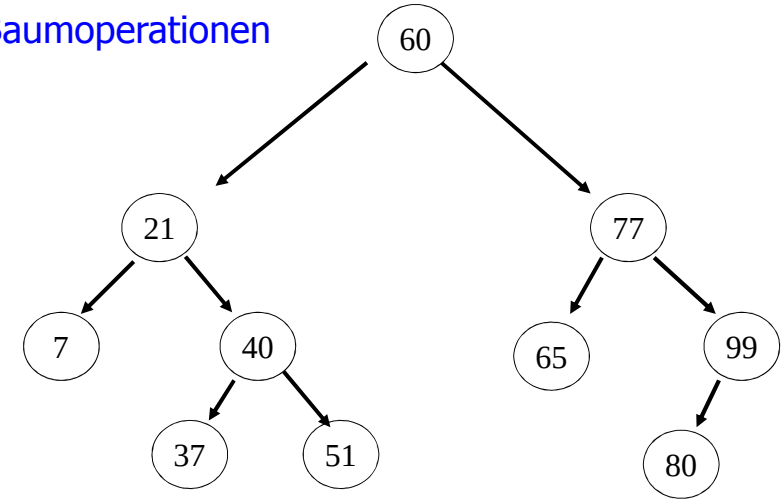
Baumoperationen: Fragen nach den Zahlenbereichen



Baumoperationen: Vorlesungsfolie



Baumoperationen



- Pre-order (root, left, right): 60, 21, 7, 40, 37, 51, 77,65, 99, 80
- In-order (left, root, right): 7,21,37, 40,51, 60, 65, 77, 80, 99
- Post-order (left, right, root): 7, 37, 51, 40, 21, 65, 80, 99, 77, 60
- Level-order: 60, 21, 77, 7, 40, 65, 99, 37, 51, 80

Einfacher Binarytree

```
public class BinaryNode  {
  public BinaryNode left=null;
  public  BinaryNode right=null;
  public int value;
  public byte orderMemory=0;

  public BinaryNode(int v) {
    value = v;
  }
  public BinaryNode(int v, BinaryNode left, BinaryNode right) {
    value = v;
    this.left = left;
    this.right=right;
  }
    public String toString() {
    return Integer.toString(value)+" ";
  }
```

Einfacher Binarytree

```
public void preOrder(JTextArea editor) {
   editor.append(value+"  ");
   if(left != null) left.preOrder(editor);
   if(right != null) right.preOrder(editor);
}
public void postOrder(JTextArea editor) {
   if(left != null) left.postOrder(editor);
   if(right != null) right.postOrder(editor);
   editor.append(value+"  ");
}
public void inOrder(JTextArea editor) {
 if (left != null) left.inOrder(editor);
 editor.append(value+"  ");
 if (right != null) right.inOrder(editor);
}
```

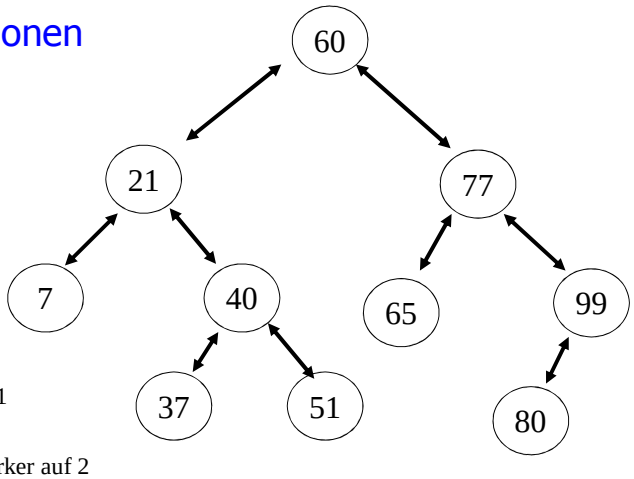
```
public void preOrder(IPrintOrder parent) {
   parent.printOrder(value);
   if(left != null) left.preOrder(parent);
   if(right != null) right.preOrder(parent);
}
public void postOrder(IPrintOrder parent) {
   if(left != null) left.postOrder(parent);
   if(right != null) right.postOrder(parent);
   parent.printOrder(value);
}
 public void inOrder(IPrintOrder parent) {
  if (left != null) left.inOrder(parent);
  parent.printOrder(value);
  if (right != null) right.inOrder(parent);
}
public interface IPrintOrder {
  public void printOrder(int value);
}
```


Einfacher Binarytree

```
private void test3() {
   BinaryNode b1 = new BinaryNode(40, new BinaryNode(37), new BinaryNode(51));
   BinaryNode bLeft = new BinaryNode(21, new BinaryNode(7), b1);
   BinaryNode b3 = new BinaryNode(99, new BinaryNode(80), null);
   BinaryNode bRight = new BinaryNode(77, new BinaryNode(65), b3);
   BinaryNode root = new BinaryNode(60, bLeft, bRight);

   editor.append("\n\nPre Order\n");
   root.preOrder(editor);
   editor.append("\n");
   editor.append("\n\nin Order\n");
   root.inOrder(editor);
   editor.append("\n");
   editor.append("\n\npostOrder\n");
   root.postOrder(editor);
  }
```

Baumoperationen mit parent



- Start mit root:
  - Marker auf 1
  - Nach links
  - Zurück: Marker auf 2
  - Nach links
  - Zurück
  - Marker auf 3
  - **Nach oben**
    - Pre-order (root, left, right): 60, 21, 7, 40, 37, 51, 77,65, 99, 80

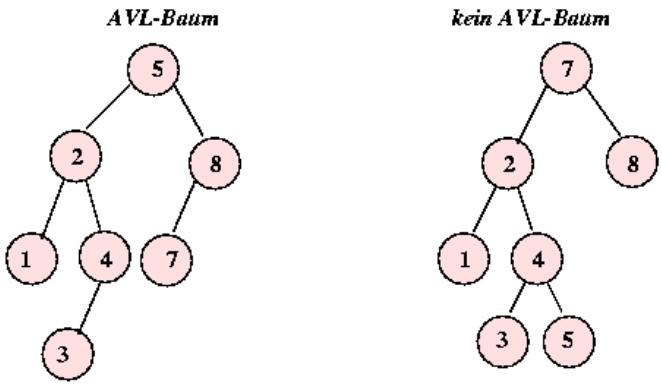
AVL Bäume

- Um stets einen möglichst optimalen Suchbaum zu erhalten, wäre es erforderlich, stets eine Transformation zu einem ausgeglichenen Baum vorzunehmen.
- Das kann ggf. eine Änderung aller Knotenwerte mit sich bringen.
- Um diesen Aufwand zu verringern, führten die russischen Mathematiker *Adelson-Velskii* und *Landis* eine abgeschwächte Form für einen ausgeglichenen binären Baum ein, der nach ihnen benannt AVL-Baum heißt (1962).
- Der AVL-Baum ist damit die älteste Datenstruktur für balancierte Bäume.

- Ein **AVL-Baum *(AVL Tree)*** ist ein binärer Baum, bei dem die Höhe der von jedem Knoten ausgehenden Teilbäume maximal um 1 differiert

Beispiele

- Das folgende Bild zeigt Beispiele für einen gütigen und einen ungültigen AVL-Baum.



AVL Eigenschaft

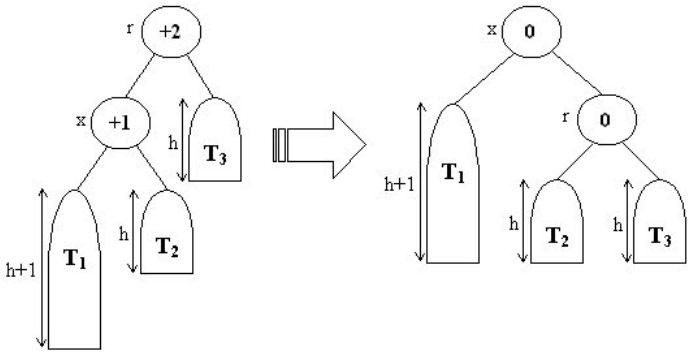
- Beim Einfügen eines Elementes muss der Baum gegebenenfalls
- umstrukturiert werden. Ziel ist: In keinem Teilbaum unterscheidet sich die Tiefe um mehr als eins,
  d.h. die Funktion b = tiefe(links) - tiefe(rechts) sollte nur die Werte (-1; 0; 1) annehmen.
- Falls beim Einfügen eine größere Differenz auftritt sind vier Fälle zu beachten die zu dieser Verletzung der AVL-Eigenschaft führen:

  (1) Einfügen im linken Subbaum des linken Kindes
  (2) Einfügen im rechten Subbaum des linken Kindes
  (3) Einfügen im rechten Subbaum des rechten Kindes
  (4)Einfügen im linken Subbaum des rechten Kindes

- Dabei sind (1) und (3) sowie (2) und (4) symmetrisch und können auf gleiche Weise behandelt werden.

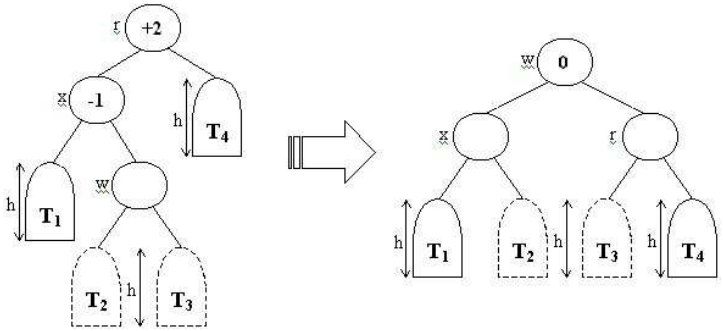
Einfache Rotation

- Falls im linken Subbaum des linken Kindes oder im rechten Subbaum des rechten Kindes eingefügt wurde, erfordert die Balancierung eine Einfachrotation:
  - x geht nach oben, r nach unten,
  - T2 wird umgehängt.



Doppel Rotation

- Falls im linken Subbaum des rechten Kindes oder im rechten Subbaum des linken Kindes eingefügt wurde, erfordert die Balancierung eine Doppelrotation:
  - im linken Subbaum von r geht x nach unten und w nach oben.
  - T2 wird umgehängt.
  - Dann geht w nach oben und r nach unten. T3 wird umgehängt.

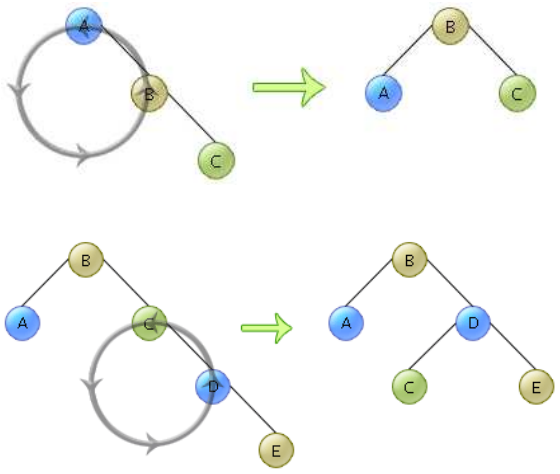


Vorteil AVL Baum

- Es kann nachgewiesen werden, dass in jedem Fall der Aufwand zum Einfügen eines neuen Knotens nur O(log n) ist.
- Gleiches gilt für das Lesen/Finden eines Knotens.

Beispiele

- Einfache Rotation



Beispiele

- Doppelte Rotation: