Bäume

Martin Wirsing

in Zusammenarbeit mit
Michael Barth, Philipp Meier und Gefei Zhang

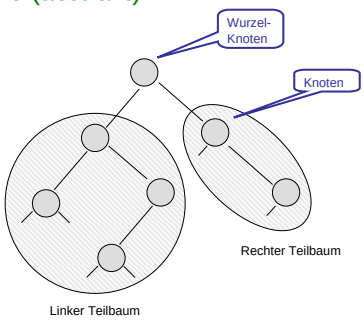
02/05

Ziele

- Standardimplementierungen für Bäume kennen lernen

Bäume (abstrakt)

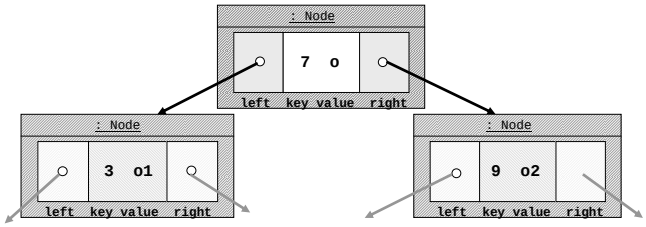
- Bäume sind hierarchische Strukturen.
- Bäume bestehen aus
  - Knoten und
  - Teilbäumen.
- Der oberste Knoten heißt Wurzel.
- Bei Binärbäumen hat jeder Knoten zwei Unterbäume:
  - den linken Unterbaum,
  - den rechten Unterbaum.
- In den Knoten kann Information gespeichert werden.



Implementierung von Knoten

```
class Node
{  Node left;
   int key; Object value
   Node right; …
}
```

- Ein Knoten wird implementiert als **Objekt mit zwei Zeigern**.
- Außerdem wird in jedem Knoten **ein Schlüssel und ein Wert** gespeichert.



Implementierung von Bäumen

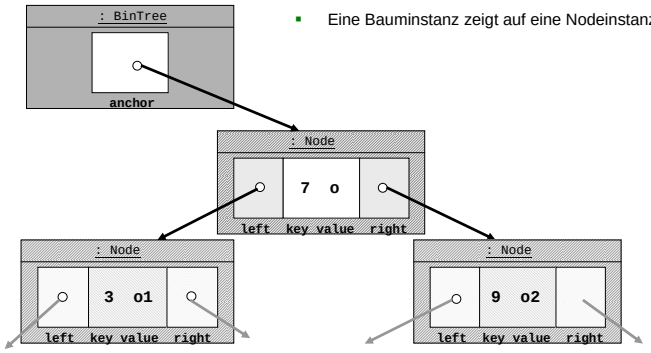
- Ein Baum wird implementiert durch einen Zeiger auf ein Geflecht von Knoten:
  - Die Klasse `BinTree` hat wie `List` einen Anker, der auf `Node` zeigt.
  - Die Klasse `Node` ist eine Hilfsklasse für die Klasse `BinTree`.

```
public class BinTree
{      Node anchor;  . . .
}
class Node
{      Node left;
       int key; Object value;
       Node right;

       // Konstruktor
       Node(Node b1, int k, Object o, Node b2)
       {  left = b1; key = k; value = o; right = b2;
       }
  . . .
}
```


Ein Beispiel für eine Instanz von BinTree

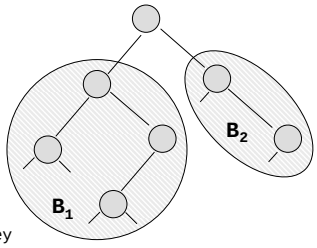
```
class BinTree
{  Node anchor;…
}
```

- Eine Bauminstanz zeigt auf eine Nodeinstanz

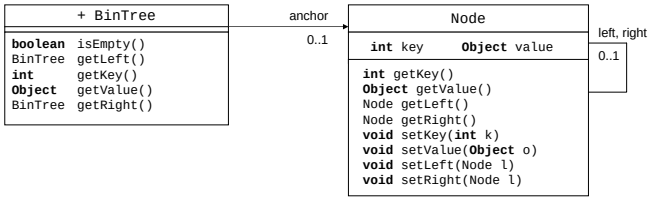


Operationen auf BinTree

- Konstruktoren
  - `BinTree()`
    - der leere Baum
  - `BinTree(`B_1`, k,o, `B_2`)` neuer `BinTree` mit
    - linkem Teilbaum B_1
    - rechtem Teilbaum B_2
    - Inhalt der Wurzel: `k,o`
- Prädikat `isEmpty`
  - Testen, ob ein `BinTree` leer ist
- Selektoren
  - `getLeft, getRight, getKey`
    Falls der BinTree nicht leer ist, liefert
    - `getLeft()` den linken Teilbaum
    - `getRight()` den rechten Teilbaum
    - `getKey()` den Inhalt der Wurzel



BinTree in UML



Implementierung in Java

Die Implementierung von BinTree verläuft analog zu Listen:

- `BinTree` repräsentiert Binärbäume über einem Integer-Schlüssel und Werten vom Typ `Object`.
  - `BinTree` selbst speichert nur einen Verweis auf die Wurzel des Baumes.
  - Die eigentliche Funktionalität wird von der Klasse `Node` realisiert;

- Um Funktionen auf `BinTree` zu definieren, verwenden wir folgende Fallunterscheidung:
  - leerer Baum: Berechnung des Resultats direkt in `BinTree`
  - nicht-leerer Baum: Weitergeben ("delegieren") der Funktion an `Node`

Implementierung BinTree: isEmpty & Zugriff auf linken Teilbaum

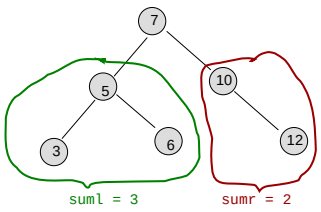
```
public class BinTree
{       private Node anchor;

        BinTree(){};   // der leere Baum
        BinTree(BinTree b1, int k, Object o, BinTree b2)
        {       anchor = new Node(b1.anchor, k, o, b2.anchor);
        }

        boolean isEmpty(){return anchor==null;}

        BinTree getLeft() throws NoSuchElementException
        {       if (anchor == null) throw new NoSuchElementException();
                else
                {   BinTree l = new BinTree();
                    l.anchor = anchor.getLeft();
                    return l;
                }
        }
. . .
```

- Implementierung verläuft analog zu Listen

Anzahl der Knoten

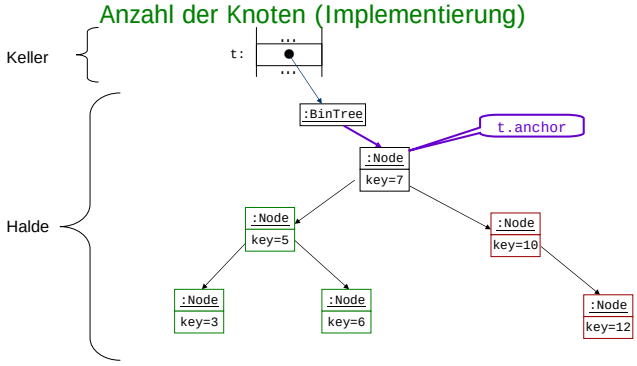
Prinzipieller Ablauf der (rekursiven) Berechnung von t.sumNodes():

1. Berechne die Anzahl der Knoten suml
   des linken Teilbaums;
2. Berechne die Anzahl der Knoten sumr
   des rechten Teilbaums;
3. Gesamtanzahl der Knoten:
   1 + suml +sumr .

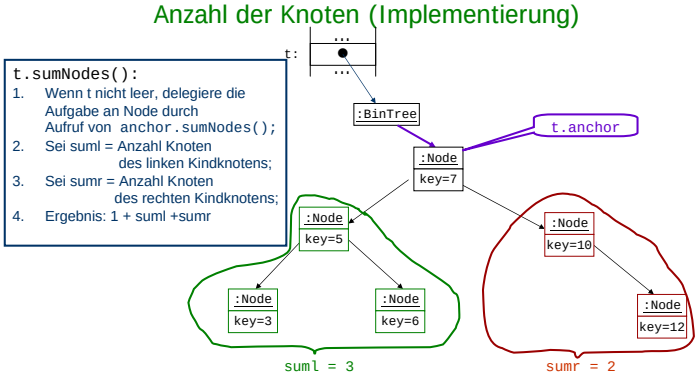


Ergebnis = 1 + suml +sumr = 6

Anzahl der Knoten (Implementierung)



Anzahl der Knoten (Implementierung)



Ergebnis = 1 + suml +sumr = 6

Implementierung BinTree: Anzahl der Knoten

```
int sumNodes()
{       if (anchor == null) return 0;
        else return anchor.sumNodes();
}
where

class Node
{ . . .
        int sumNodes()
        {       int suml = 0, sumr = 0;
                if (left != null) suml = left.sumNodes();
                if (right != null) sumr = right.sumNodes();
                return 1 + suml + sumr;
        }
 . . .
```

Wenn der Baum leer ist, gibt es keinen Knoten; d.h. Anzahl = 0

Delegieren der Aufgabe an sumNodes() (aus der Klasse Node)

Geordnete Binärbäume

- Ein Binärbaum b heißt geordnet, wenn folgendes für alle nichtleeren Teilbäume t von b gilt:

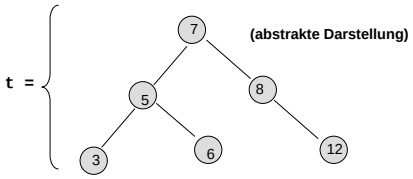
  Der Schlüssel von t ist
  - größer (oder gleich) als alle Schlüssel des linken Teilbaums von t und
  - kleiner (oder gleich) als alle Schlüssel des rechten Teilbaums von t

Geordnete Binärbäume

- Beispiel: Geordnet sind:

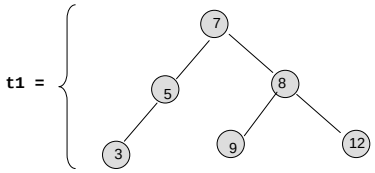
  Der leere Baum und

  der Baum t:



- Nicht geordnet ist

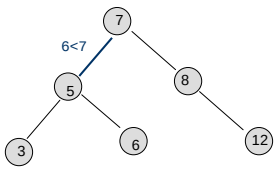
  der Baum t1:



Suche im geordneten Binärbaum

Prinzipieller Ablauf der Berechnung von t.find(6):

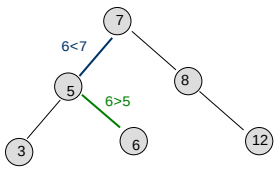
1. Vergleiche 6 mit dem Wert der Wurzel;
2. Da 6<7, gehe zum linken Kindknoten;



Suche im geordneten Binärbaum

Prinzipieller Ablauf der Berechnung von t.find(6):

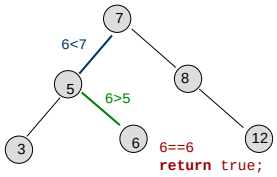
1. Vergleiche 6 mit dem Wert der Wurzel;
2. Da 6<7, gehe zum linken Kindknoten;
3. Vergleiche 6 mit dem Wert dieses Knotens;
4. Da 6>5, gehe zum rechten Kindknoten dieses Knotens;



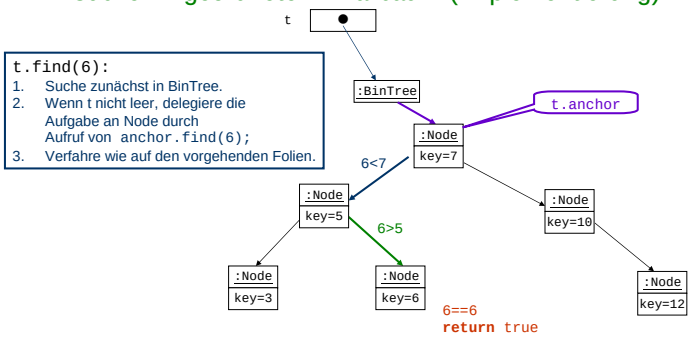
Suche im geordneten Binärbaum

Prinzipieller Ablauf der Berechnung von t.find(6):

1. Vergleiche 6 mit dem Wert der Wurzel;
2. Da 6<7, gehe zum linken Kindknoten;
3. Vergleiche 6 mit dem Wert dieses Knotens;
4. Da 6>5, gehe zum rechten Kindknoten dieses Knotens;
5. Vergleiche 6 mit dem Wert dieses Knotens;
6. Da 6==6, gebe Ergebnis true zurück.



Suche im geordneten Binärbaum (Implementierung)



Suche im geordneten Binärbaum

Gibt true zurück, wenn key im Baum; sonst wird false zurückgegeben

```
public Object find(int key)
  { if (anchor == null) return false;
    else return anchor.find(key);
  }
wobei

class Node
{ . . .
    Object find (int key)
    {  Node current = this;
       while(current.key != key)        // solange nicht gefunden,
       { if(key < current.key)          // gehe nach links?
           current = current.left;
         else                           // sonst gehe nach rechts
           current = current.right;
         if(current == null) return false; //nicht gefunden!
       }
       return true;                 //gefunden; gib true zurück
    }
```


Suche im geordneten Binärbaum

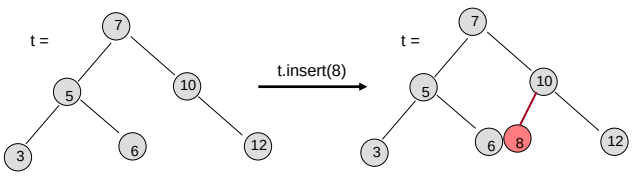
Gibt value zurück, wenn key im Baum; sonst wird null zurückgegeben

```
public Object findValue(int key)
  { if (anchor == null) return null;
    else return anchor.find(key);
  }
wobei

class Node
{ . . .
    Object findValue(int key)
    {  Node current = this;
       while(current.key != key)        // solange nicht gefunden,
       { if(key < current.key)          // gehe nach links?
           current = current.left;
         else                           // sonst gehe nach rechts
           current = current.right;
         if(current == null) return null; //nicht gefunden!
       }
       return current.value;        //gefunden; gib value zurück
    }
```


Einfügen in geordneten Binärbaum

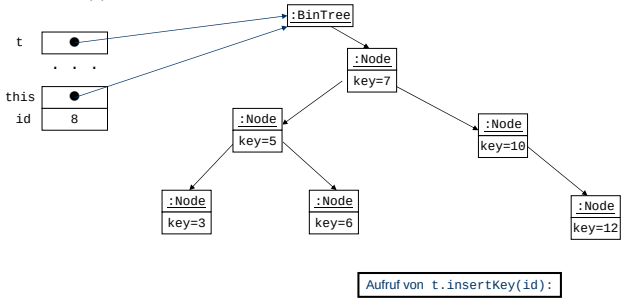
- Beim Einfügen in einen geordneten Binärbaum wird rekursiv die "richtige" Stelle gesucht, so dass wieder eine geordneter Binärbaum entsteht.
- Beispiel: t.insert(8) ergibt:

(Zur Vereinfachung der Darstellung wird hier nur ein Schlüssel und kein Wert eingefügt.)



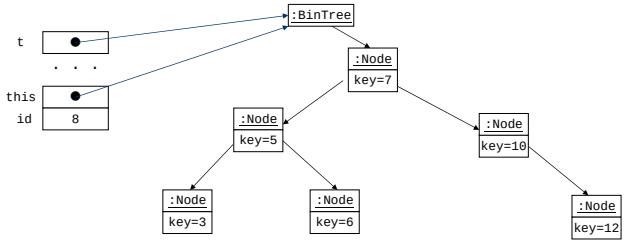
Einfügen in geordneten Binärbaum (Implementierung)

t.insert(8)



Einfügen in geordneten Binärbaum (Implementierung)

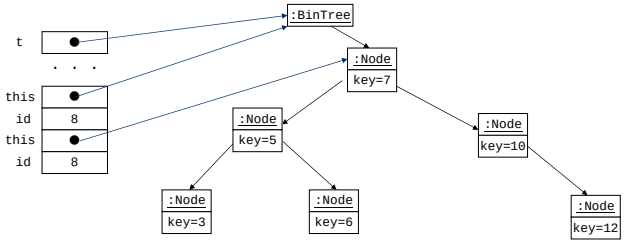
t.insert(8)



Delegieren der Aufgabe durch
Aufruf von `anchor.insertKey(id):`

Einfügen in geordneten Binärbaum (Implementierung)

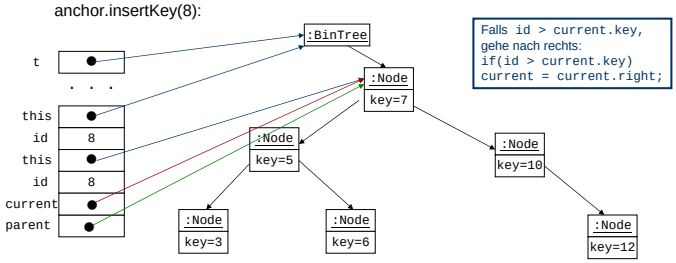
anchor.insertKey(8):



Delegieren der Aufgabe durch
Aufruf von `anchor.insertKey(id):`
Durchlauf durch das Node-Geflecht mit
zwei Hilfsvariablen `current` und `parent`

Einfügen in geordneten Binärbaum (Implementierung)

anchor.insertKey(8):

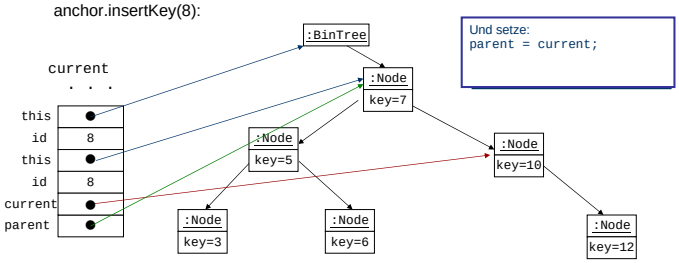


Falls `id > current.key`,
gehe nach rechts:
`if(id > current.key)`
`current = current.right;`

Delegieren der Aufgabe durch
Aufruf von `anchor.insertKey(id):`
Durchlauf durch das Node-Geflecht mit
zwei Hilfsvariablen `current` und `parent`

Einfügen in geordneten Binärbaum (Implementierung)

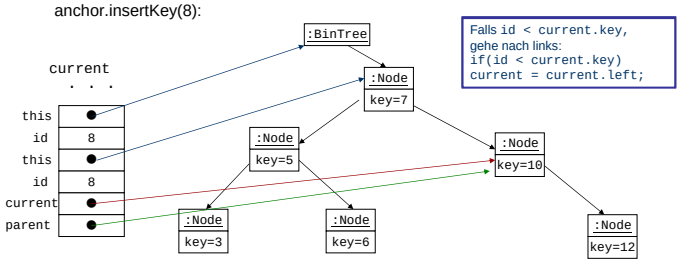
anchor.insertKey(8):



Und setze:
`parent = current;`

Einfügen in geordneten Binärbaum (Implementierung)

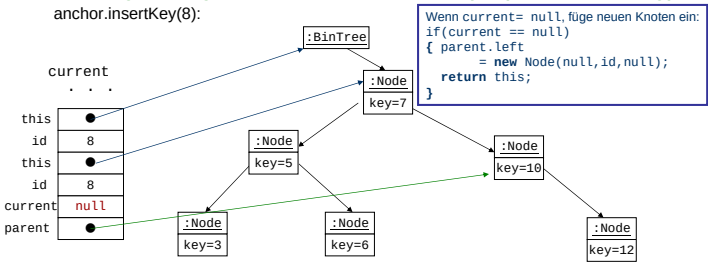
anchor.insertKey(8):



Falls `id < current.key`,
gehe nach links:
`if(id < current.key)`
`current = current.left;`

Einfügen in geordneten Binärbaum (Implementierung)

anchor.insertKey(8):

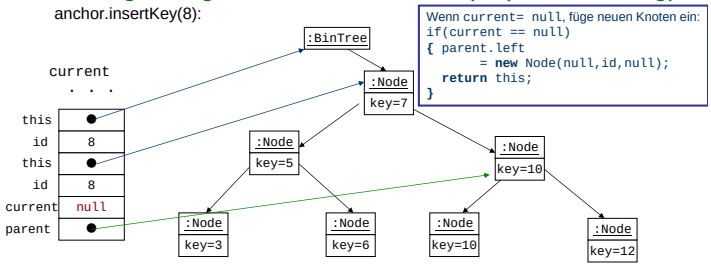


Wenn `current= null`, füge neuen Knoten ein:

```
if(current == null)
{ parent.left
        = new Node(null,id,null);
  return this;
}
```

Einfügen in geordneten Binärbaum (Implementierung)

anchor.insertKey(8):



Wenn `current= null`, füge neuen Knoten ein:

```
if(current == null)
{ parent.left
        = new Node(null,id,null);
  return this;
}
```


Einfügen in geordneten Binärbaum

Fügt einen neuen Knoten mit Schlüssel `id` an der richtigen Stelle im geordneten Baum ein

```
public void insert(int id, Object o)
{   if(anchor==null)                // falls kein Knoten im anchor
       anchor = new Node(null, id,o, null);    // neuer Knoten
    else  anchor = anchor.insertKeyObj(id, o);
}
```

wobei `insertKeyObj` in **`class`** `Node` folgendermaßen definiert wird:

Einfügen in geordneten Binärbaum (Implementierung)

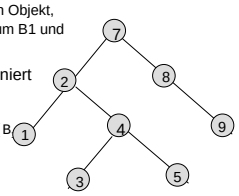
Fügt einen neuen Knoten passend ein Achtung:`id` darf nicht im Baum vorkommen!

```
Node insertKeyObj(int id, Object o)
{    Node current = this;       // starte bei this
     Node parent;
     while(true)                // terminiert intern
     {   parent = current;
         if(id < current.key)  // gehe nach links?
         {  current = current.left;
            if(current == null)  // am Ende füge links ein
            {   parent.left = new Node(null, id, o, null);
                return this;
            }
         }  // end if go left
         else            // falls id > current.key, gehe nach rechts
         {  current = current.right;
            if(current == null)  // am Ende füge rechts ein
            {   parent.right = new Node(null, id, o, null);
                return this;
            }
         }  // end else go right
     }  // end while
}
```


Rekursion auf Binärbäumen

- Binärbäume sind induktiv definiert
  - Der Leere Baum ist ein Binärbaum
  - Sind B_1 und B_2 Binärbäume, id ein Schlüssel und o ein Objekt, dann ist der Baum mit Wurzel id und o, linkem Teilbaum B1 und rechtem Teilbaum B_2 ein Binärbaum

BinTree(B_1, k,o, B_2)



sumNodes: 8

- Operationen f auf Binärbäumen können rekursiv definiert werden
  - Falls `isEmpty(B):` gibt `f(B)` direkt an
  - Ansonsten beschreibe wie sich der Wert von `f` aus `f(B`$_1$`)` `f(B`$_2$`)` und `id,o` ergibt.
- Beispiel: sumNodes1()
  - Falls `isEmpty(B): 0`
  - Ansonsten `getLeft().sumNodes1()+getRight().sumNodes1()+1`
- Beispiel: exist(int k) (ist k vorhanden ?)
  - Falls `isEmpty(B) : false`
  - Ansonsten: `getLeft().exists(k)`
    `|| getKey()==k ||getRight().exists(k)`

Einfache Baumoperationen

```
public class XBinTree extends BinTree
{
   public int sumNodes()
   {   if(isEmpty()) return 0;
       else return
           ((XBinTree)getLeft()).sumNodes() +
           ((XBinTree)getRight()).sumNodes() +1;
   }

   public boolean exists(int k)
   {       if(isEmpty()) return false;
           else return
                ((XBinTree)getLeft()).exists(k)
                || getKey()==k
                || ((XBinTree)getRight()).exists(k);
   }
```

- `isEmpty(), left(), right()` werden aus der Oberklasse geerbt
- `left()` liefert einen `BinTree`
- `sumNodes()` ist nur in der Unterklasse definiert - für `XBinTrees`
- Wir brauchen casts um die `BinTrees` in `XBinTrees` zu verwandeln

Zusammenfassung

- Binäre Bäume werden in Java implementiert:
  - als Verallgemeinerung der einfach verketteten Listen mit zwei Nachfolgerverweisen
- Eine Operation auf binären Bäume mit Knoten wird definiert:
  - durch Weitergeben der Operation an die Knotenklasse **oder**
  - durch Fallunterscheidung bzgl. des leeren Baums und rekursiven Aufruf der Selektoren getLeft() und getRight() von BinTree.