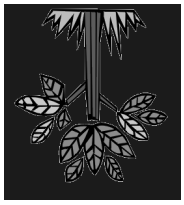
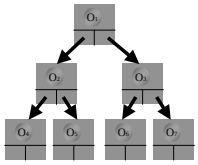
Bäume





Margarita Esponda
Institut für Informatik
Freie Universität Berlin

esponda@inf.fu-berlin.de

Inhalt

1. Einführung
2. Warum Bäume?
3. Listen and Arrays vs. Bäume
4. Einfach verkettete binäre Bäume
   - Baumtraversierung
   - Suchen
   - Einfügen
5. Doppelt verkettete binäre Bäume
   - Löschen
6. Iteratoren

Bäume

Bäume sind fundamentale Datenstrukturen für:

- Betriebssysteme
- Datenbanken
- Übersetzerbau
- Textverarbeitung
- Graphische Verarbeitung
- Datenkompression
- Spiele

usw.

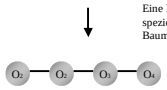
1. Wofür dynamische Datenstrukturen?

Für dynamische Datenmengen
(wachsen und schrumpfen zur Laufzeit)
ohne Vorkenntnisse der maximal erreichbaren Größe

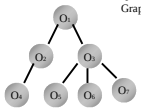
Beispiele:

Listen

Eine Liste ist ein spezieller Fall eines Baumes

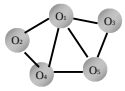


Bäume



Ein Baum ist ein spezieller Fall eines Graphen.

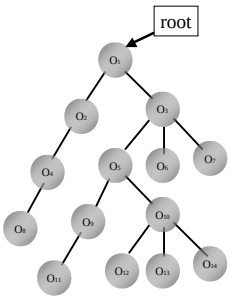
Graphen



(Dynamische Arrays)

Was ist ein Baum?

Eine Menge von Knoten und Kanten, die diese Knoten verbinden



1. Hat eine Wurzel
2. Alle Knoten außer der Wurzel haben genau eine Verbindung mit einem Vorfahren.
3. Es existiert genau ein Weg zwischen der Wurzel und jedem beliebigen Knoten.

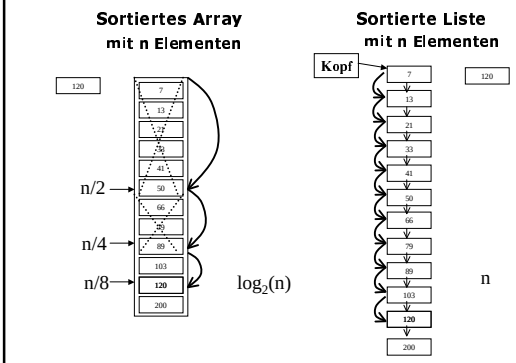
Warum Bäume?

Weil die Grundoperationen für dynamische Datenmengen effizienter realisiert werden können, wenn man Bäume verwendet.

Elementare Operationen für dynamische Mengen { Suchen, Einfügen, Löschen

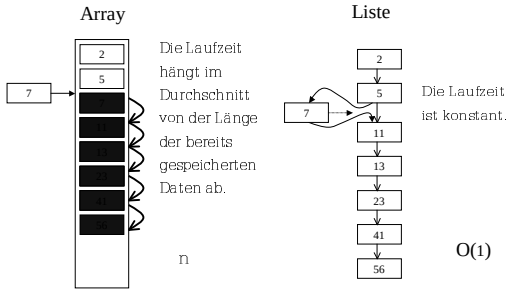
Bäume kombinieren die Vorteile von zwei Datenstrukturen, die wir schon diskutiert haben: Reihungen (arrays) und Listen.

Suchen

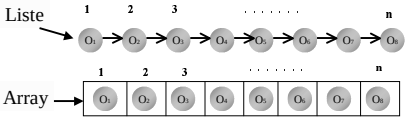


Einfüge- und Lösch-Operationen

(wenn die Position bereits bekannt ist)



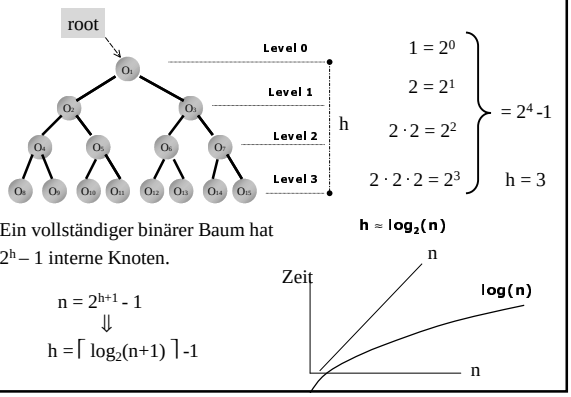
Liste vs. Array



Elementare Operationen für dynamische Mengen

	Liste Schlimmster Fall	Array Schlimmster Fall
Suchen	n	$\log_2(n)$
Einfügen	konstant + n	n + $\log_2(n)$
Löschen	konstant + n	n + $\log_2(n)$

Bäume vs. Listen



Ein vollständiger binärer Baum hat $2^h - 1$ interne Knoten.

$n = 2^{h+1} - 1$

$\Downarrow$

$h = \lceil \log_2(n+1) \rceil - 1$

Maximale Anzahl der besuchten Knoten bei einer Suchoperation

Liste	Binäre Bäume	
1	1	Minimale Höhe
3	2	
7	3	
15	4	
31	5	
...	...	
1,023	10	
...	...	
32,767	15	
...	...	
33,554,432	25	
...	...	
1,073,741,824	30	

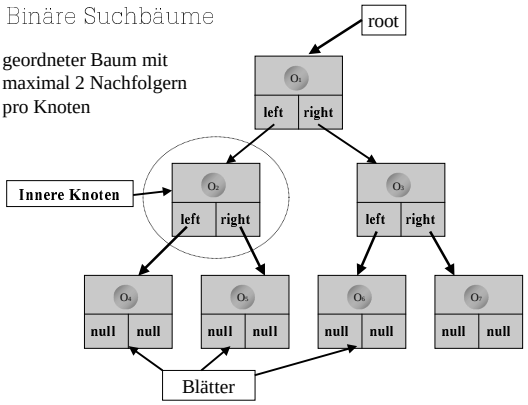
Binäre Bäume

- Baumstrukturen
  - binäre Bäume — einfachste
  - ausgeglichene Bäume
    - AVL-Bäume
    - Red-Black-Trees
    - B-Bäume

Ein binärer Baum ist eine endliche Menge von Knoten, die entweder leer ist oder aus einer Wurzel und den Elementen zweier disjunkter binärer Bäume besteht (linker und rechter Teilbaum).

Binäre Suchbäume

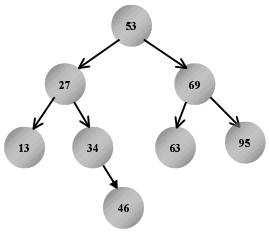
geordneter Baum mit
maximal 2 Nachfolgern
pro Knoten



Binäre Suchbäume

Binäre Suchbäume sind sortierte binäre Bäume,
d.h. die gespeicherten Elemente im Baum werden nach bestimmten Regeln einsortiert.

Beispiel:



Wie können wir binäre Bäume in Java implementieren?

Eine Klasse TreeNode

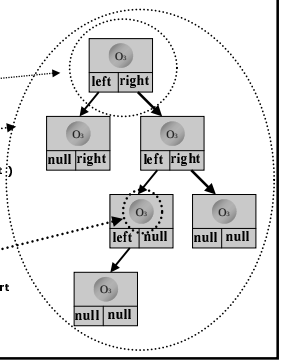
Eine Klasse BinaryTree

public TreeNode find (Comparable element)
public void insert (Comparable element)
public void delete (TreeNode node)

Eine Klasse ComparableObject,

die die Interface Comparable implementiert

public int compareTo (Object element)



Implementierung der java-interface Comparable

```
public interface Comparable {
      public int  compareTo( Object o );
}
```

```
public class Zahl implements Comparable {
  int num;
     . . .
  public int  compareTo( Object o ) {
     Zahl temp = (Zahlen) o;
     if ( this.num < temp.num ) return -1;
     else if ( this.num > temp.num ) return 1;
     else return 0;
  }
    . . .
}
```

TreeNode-Klasse

rekursive Definition

```
public class TreeNode {
    Object element;
    TreeNode left;
    TreeNode right;

   // Konstruktoren
   TreeNode( Object element ) {
         this.element = element;
         left = null;
         right = null;
   }
   TreeNode( ) {
         this( null );
   }
} // end of class TreeNode
```

Implementierung der Klasse BinaryTree

```
class BinaryTree {
    private TreeNode root;

    public BinaryTree() {
         root = null;
    }
   public TreeNode find ( Comparable element ) {...}
   public void insert( Comparable element ) {...}
   public void delete( TreeNode node ) {...}

   public void inOrder( TreeNode node ) {...}

   public TreeNode minimum( TreeNode node ) {...}
   public TreeNode successor( TreeNode node ) {...}
} // end of class BinaryTree
```

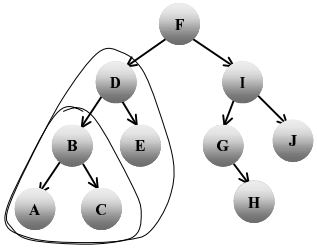
Traversierung binärer Bäume

Baumtraversierung bedeutet, alle Knoten des Baumes in einer bestimmten Reihenfolge zu besuchen.

	Preorder:	**Wurzel – linker Unterbaum – rechter Unterbaum**
→	**Inorder:**	**linker Unterbaum - Wurzel – rechter Unterbaum**
	Postorder:	**linker Unterbaum – rechter Unterbaum - Wurzel**
	Levelorder:	**von oben nach unten in jeder Ebene von links nach rechts**

Traversierung binärer Bäume

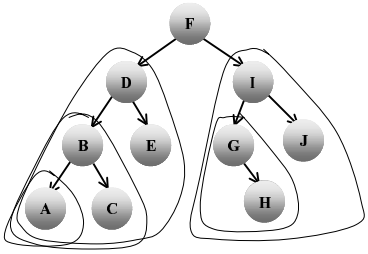
Preorder Wurzel - linker Unterbaum – rechter Unterbaum



F D B A C E I G H J

Traversierung binärer Bäume

Inorder Linker Unterbaum - Wurzel - Rechter Unterbaum



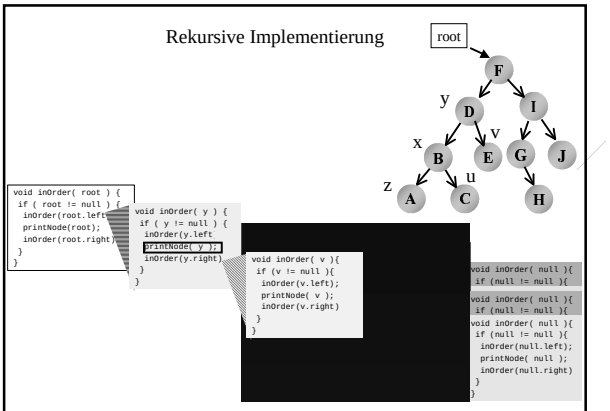
A B C D E F G H I J

Inorder

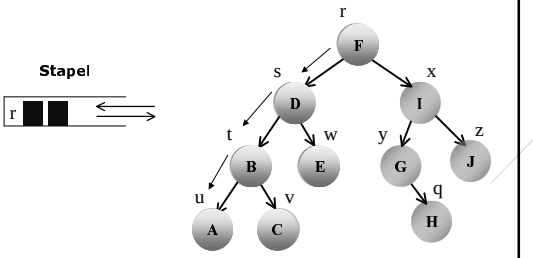
Rekursive Implementierung

```
public class Tree {
    ...
   public void inOrder( TreeNode node ) {
      if ( node != null ) {
         inOrder( node.left );
         System.out.println( node.element.toString() );
         inOrder( node.right );
      }
   }
    ...
}
```

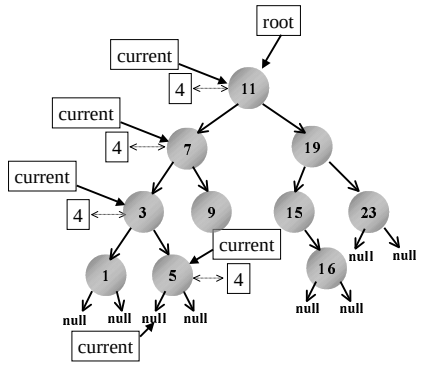
Rekursive Implementierung



Iterative Implementierung



Suchen



Suchen

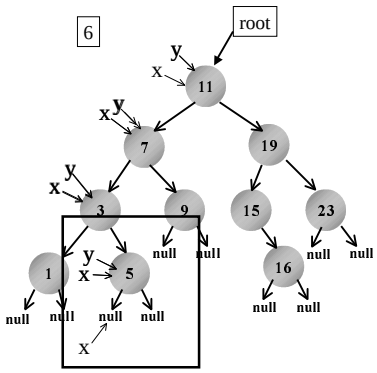
```
public TreeNode find ( Comparable element ) {

   TreeNode current = root;
   while ( element.compareTo( current.element )!= 0 ) {
        if ( element.compareTo( current.element ) < 0 )
           current = current.left;
        else
           current = current.right;
        if ( current == null )
           return null;       // didn't find it
   } // end of while
   return current;            // found it

}
```

Suchen

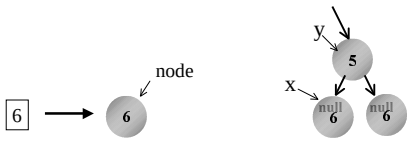
rekursive Implementierung

```
public TreeNode find ( TreeNode curr, Comparable elem ) {
   if ((curr==null)||(elem.compareTo(curr.element)==0))
        return curr;
   if ( elem.compareTo( curr.element ) < 0)
        curr = find( curr.left, elem );
   else
        curr = find( curr.right, elem );
   return curr;
}
```

Einfügen



Einfügen

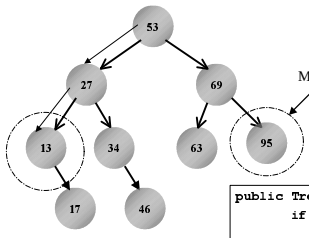



```
TreeNode node = new TreeNode( element );

if ( node.element.compareTo( y.element ) < 0 )
     y.left = node;
else
     y.right = node;
```

Einfügen

```
public void insert( Comparable element ) {
    TreeNode node = new TreeNode( element );
    TreeNode y = null;
    TreeNode x = root;
    while ( x != null ) {
        y = x;
        if ( node.element.compareTo( x.element ) < 0 )
            x = x.left;
        else
            x = x.right;
    }
    if ( y == null ) // Der Baum ist leer!
        root = node;
    else if ( node.element.compareTo( y.element ) < 0 )
            y.left = node;
        else
            y.right = node;
 } // end of insert
```

Minimum und Maximum



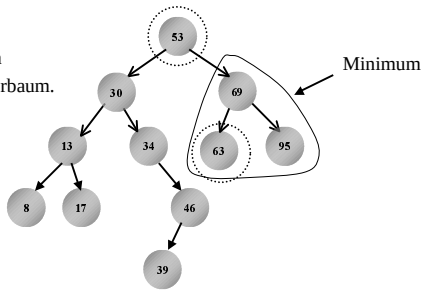
Der erste Knoten, der keine linken Kinder mehr hat, beinhaltet das kleinste Element.

```
public TreeNode minimum( TreeNode node ){
        if ( node == null )
            return null;
        else {
           while ( node.left != null ) {
               node = node.left;
           }
           return node;
        }
}
```

Nachfolger

1.Fall

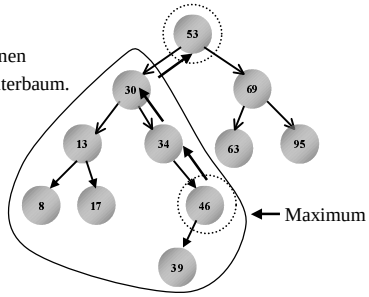
Es gibt einen rechten Unterbaum.



Nachfolger

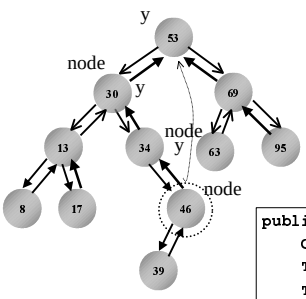
2.Fall

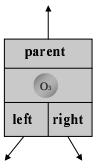
Es gibt keinen rechten Unterbaum.



Wie können wir nach oben laufen?

Doppelt verkettete Bäume

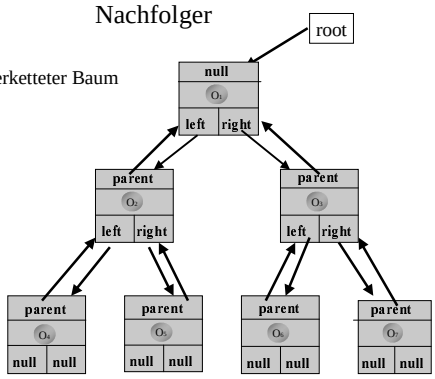





```
public class TreeNode {
    Object element;
    TreeNode parent;
    TreeNode left;
    TreeNode right;

} // end of class TreeNode
```

Nachfolger

Doppelt verketteter Baum



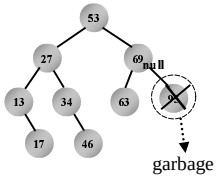
Nachfolger

```
public TreeNode successor( TreeNode node ) {
        if ( node.right != null )
            return minimum( node.right );
        TreeNode y = node.parent;
        while ( ( y != null )&&( node == y.right ) ) {
                node = y;
                y = y.parent;
        }
        return y;
}
```

Delete-Operation (Löschen)

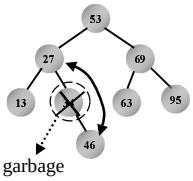
1. Fall

Löschen eines Knotens ohne Kinder



2. Fall

Löschen eines Knotens mit nur einem Kind

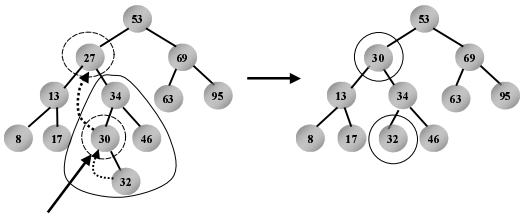


Löschen

3. Fall

Löschen eines Knotens mit zwei Kindern

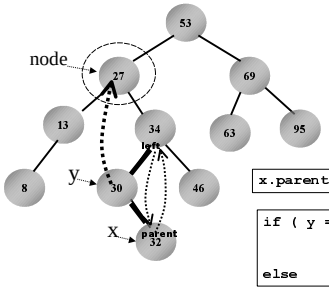
Der Knoten, den man löschen möchte, wird durch seinen Nachfolger ersetzt.



Der Nachfolger von 27 ist das Minimum des rechten Unterbaumes.

Das Minimum ist entweder ein Blatt oder hat maximal ein rechtes Kind.

Löschen




```
x.parent = y.parent;
```

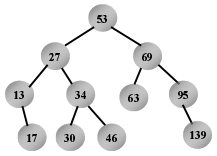
```
if ( y ==y.parent.left )
        y.parent.left = x;
else    y.parent.right = x;
```

```
if ( y != node )
   node.element = y.element;
```

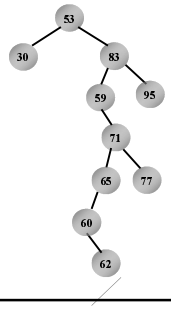
```
public void delete( TreeNode node ) {
   TreeNode y, x;
1.+2. Fall { if ( (node.left==null)||(node.right==null))
                y = node;
3. Fall  → else   y = successor( node );

x wird der Nachfolger des Nachfolgers von node.
         if ( y.left != null )
                x = y.left;
         else   x = y.right;
         if ( x != null )  x.parent = y.parent;

Der Baum hatte nur einen Knoten. { if ( y.parent == null )
                root = x;
         else   if ( y == y.parent.left )
                        y.parent.left = x;
                else    y.parent.right = x;

node wird durch seinen Nachfolger ersetzt. → if ( y != node )  node.element = y.element;
 }
```

Probleme mit einfachen binären Suchbäumen

balancierter Binärbaum



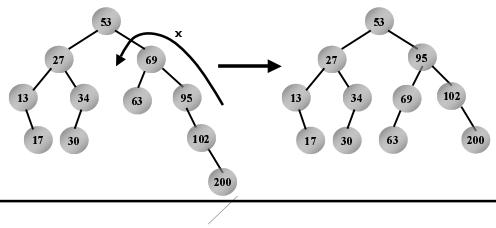
nicht balancierter Binärbaum



Lösungen

AVL-Bäume
Red-Black-Bäume
AA-Bäume
B-Bäume

Innerhalb der insert und delete-Operationen wird mit Hilfe von Rotationen die Balance des Baums ständig wiederhergestellt.



Iteratoren

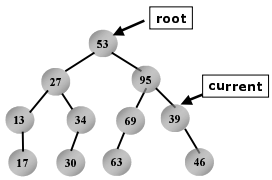
Unsere Baumobjekte werden durch eine Referenz zum Wurzelknoten identifiziert.

Auf jedes einzelne Element im Baum kann zugegriffen werden; dies erfolgt über eine Referenz zu dem jeweiligen Knoten, der dieses Element speichert.

Das Problem bei dieser Implementierungsstrategie ist, dass Fehler nur schwer identifiziert werden können.

Ein Benutzer kann sehr leicht Referenzen aus einem falschen Baum übergeben und einen anderen Baum dadurch korrumpieren.

Iteratoren




```
class BinaryTree {
  private TreeNode root;
  private TreeNode current;
  public BinaryTree() {
        root = null;
        current = root;
  }
  . . .
} //end of class BinaryTree
```

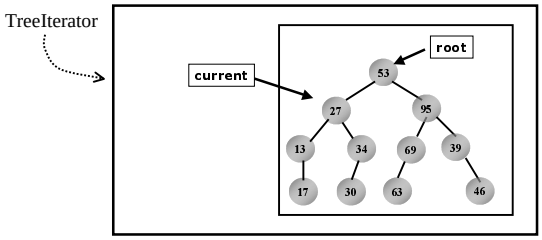
Weil der Zugriff auf **current** nur durch die Klassenmethoden möglich ist, garantieren wir damit, dass sich **current** nur innerhalb seines eigenen Baumes bewegt.

Die Einschränkung ist, dass nur eine aktuelle Position innerhalb des Baumes möglich ist.

Iterator-Klasse

Die Baumklasse wird dann nur Operationen zur Verfügung stellen, die den Baum als Ganzes betrachtet.

Methoden, die von einer bestimmten Position innerhalb des Baumes abhängig sind, werden in die Iterator-Klasse verlagert.



TreeIterator-Klasse

```
class BinaryTreeIterator {
    protected BinaryTree theTree;
    protected TreeNode current;

    public BinaryTreeIterator( BinaryTree theTree ) {
         this.theTree = theTree;
         current = null;
    }
   public void first ( ) {...}
   public void advance ( ) {...}
   public void next ( ) {...}
   public void previous ( ) {...}
   . . .

} // end of class BinaryTree
```

Elementare Operationen für dynamische Mengen

	Liste	Array	Baum
	Schlimmster Fall	Schlimmster Fall	Schlimmster Fall
Suchen	n	$\log_2(n)$	$\log_2(n)$
Einfügen	n	$n + \log_2(n)$	$\log_2(n)$
Löschen	n	$n + \log_2(n)$	$\log_2(n)$

Java-Klassen

java.util.*;

TreeMap-Klasse

Red-Black-Baum Implementierung von der SortedMap-interface. Garantiert log(n) Zeitkosten für die Grundoperationen

Suchen, Einfügen, Löschen.

TreeSet-Klasse

Diese Klasse garantiert log(n) Zeitkosten für die Grundoperationen (add, remove and contains).