Dynamische Datenstrukturen: Listen und Bäume

Martin Wirsing

in Zusammenarbeit mit
Moritz Hammer und Axel Rauschmayer

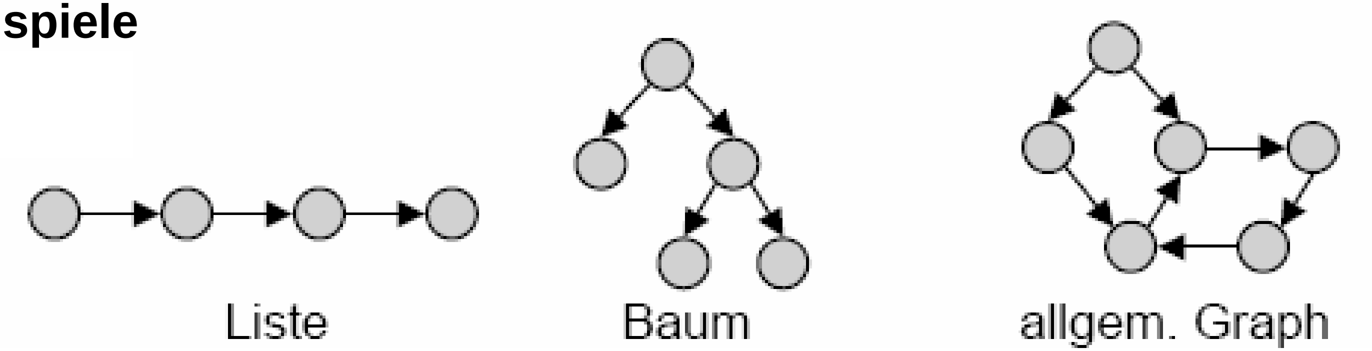
http://www.pst.ifi.lmu.de/lehre/SS06/infoII/

Ziele

- Standardimplementierungen für Listen kennenlernen
- Standardimplementierungen für Bäume kennen lernen
- Noch mehr Entwurfsmuster: Composite

Dynamische Datenstrukturen

- **Motivation**
  - Länge eines Arrays ist nach der Erzeugung festgelegt.
  - Hilfreich wären unbeschränkt große Datenstrukturen.
  - Lösungsidee: Verkettung einzelner Objekte zu größeren Strukturen
- **Beispiele**



- **Charakterisierung**
  - Knoten werden zur Laufzeit (also dynamisch) erzeugt und verkettet.
  - Strukturen können dynamisch wachsen und schrumpfen.
  - Größe einer Struktur ist nur durch verfügbaren Speicherplatz beschränkt und muss nicht im vorhinein bestimmt werden.

Beispiele: Dynamische Datenstrukturen

- **Liste**
  - Jeder Knoten (außer dem letzten) hat **genau einen Nachfolger**.
  - Jeder Knoten (außer dem ersten) hat **genau einen Vorgänger**.

- **Baum**
  - Ein Knoten kann **mehrere Nachfolger** haben („Verzweigungsgrad“).
  - Jeder Knoten (außer der Wurzel) hat **genau einen Vorgänger**.
  - Modellierung von Hierarchien (Bsp. Verzeichnisstruktur, Teilestruktur eines Fahrzeugs.
  - **Binärbaum**: Jeder Knoten hat **höchstens zwei Nachfolger**.
  - **Effiziente Implementierung von Mengen**

- **Allgemeiner Graph**
  - Knoten können **beliebige Vorgänger und Nachfolger** haben.
  - Folge: Es können **Zyklen** gebildet werden.
  - **Modellierung von Netzen** (Bsp. Straßennetz, Schienennetz).

Die Rechenstruktur der Listen

- Eine **Liste** ist eine **endliche Sequenz von Elementen**, deren Länge (im Gegensatz zu Reihungen) durch Hinzufügen und Wegnehmen von Elementen geändert werden kann.
- **Standardoperationen für Listen** sind:
  - Löschen aller Elemente der Liste (`clear`)
  - Zugriff auf und Änderung des letzten Elements
  - Einfügen und Löschen des ersten Elements
  - Prüfen auf leere Liste, Suche nach einem Element
  - Berechnen der Länge der Liste,
  - Listendurchlauf [siehe Aufgabe 9-2]
- Die **Javabibliothek** bietet Standardschnittstellen und -Klassen für Listen an:

  interface List<E>, class LinkedList<E>, ArrayList<E>

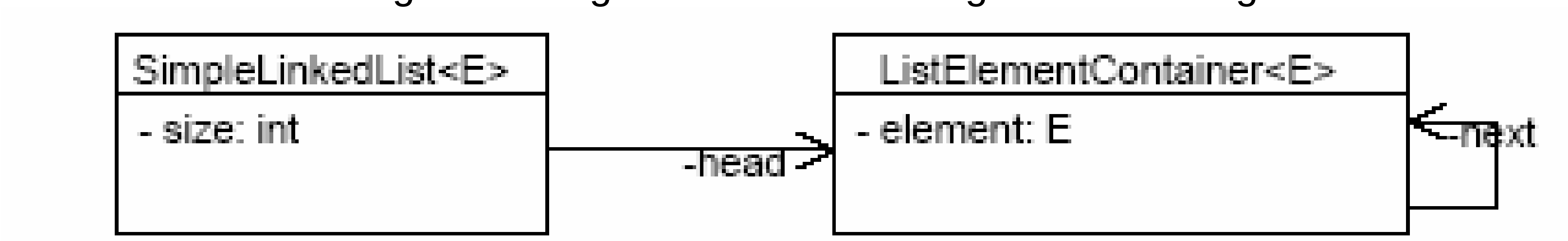
  die weitere Operationen enthalten, insbesondere den direkten Zugriff auf Elemente durch Indizes wie bei Reihungen

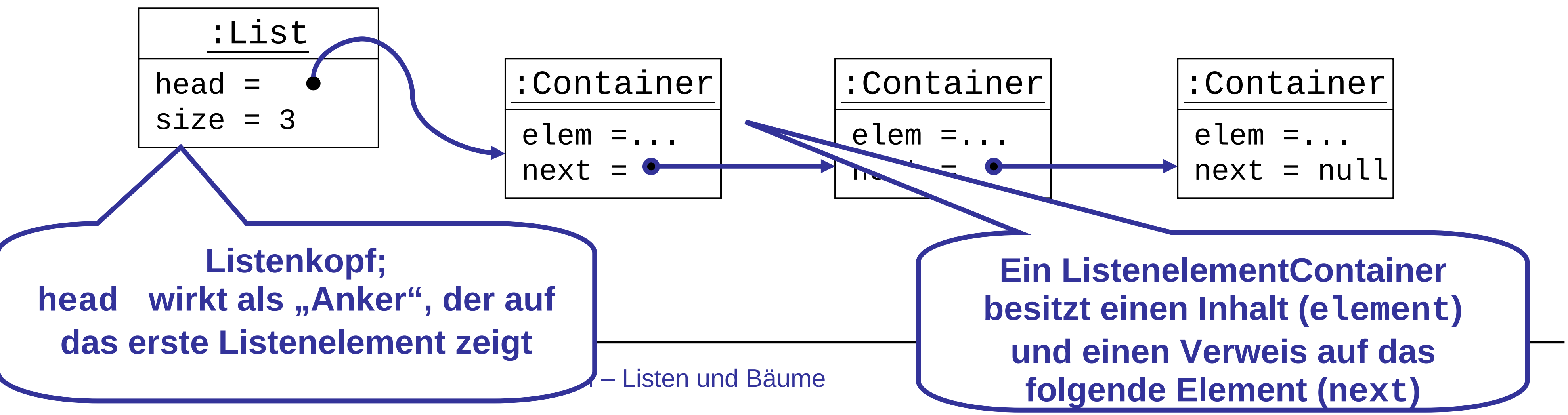
  ➔ ! Problematisch: Führt zur Vermischung von Reihung und Liste

Listenimplementierung: Einfach verkettete Listen

- Eine einfach verkettete Liste ist eine Sequenz von Objekten, wobei jedes Element auf seinen Nachfolger in der Liste zeigt [siehe Aufgabe 9-2].
- **Unterschiedliche Implementierungen:**

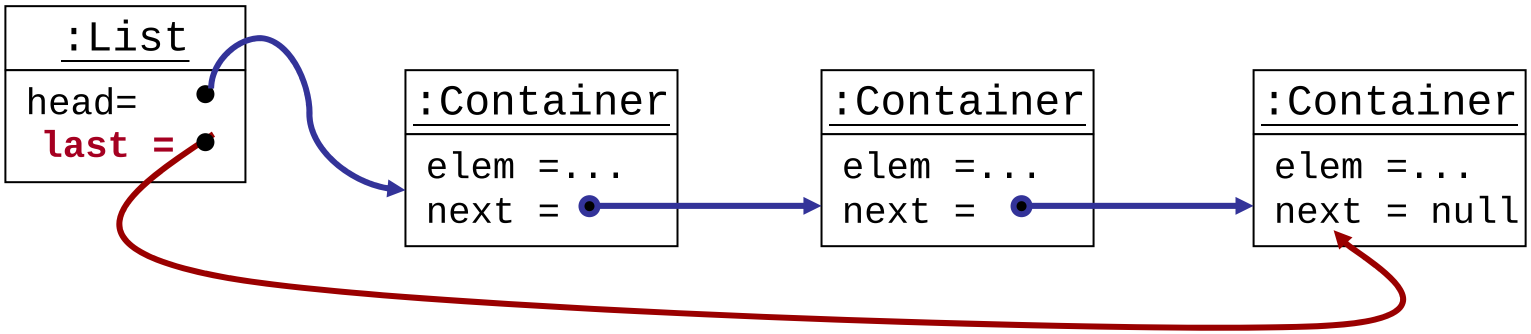
1. Realisierung des Anfügens vorne und Längenbestimmung in konstanter Zeit:



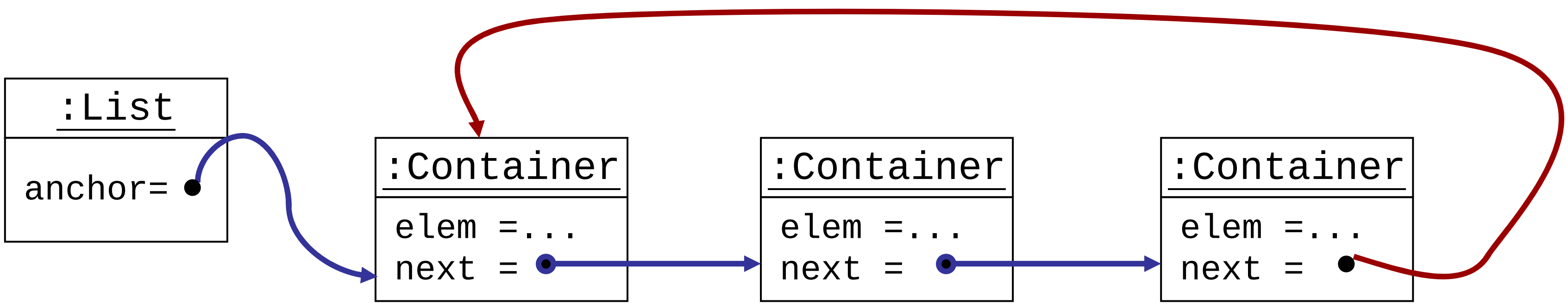


Einfach verkettete Listen

2. Realisierung des Anfügens vorne und hinten in konstanter Zeit:

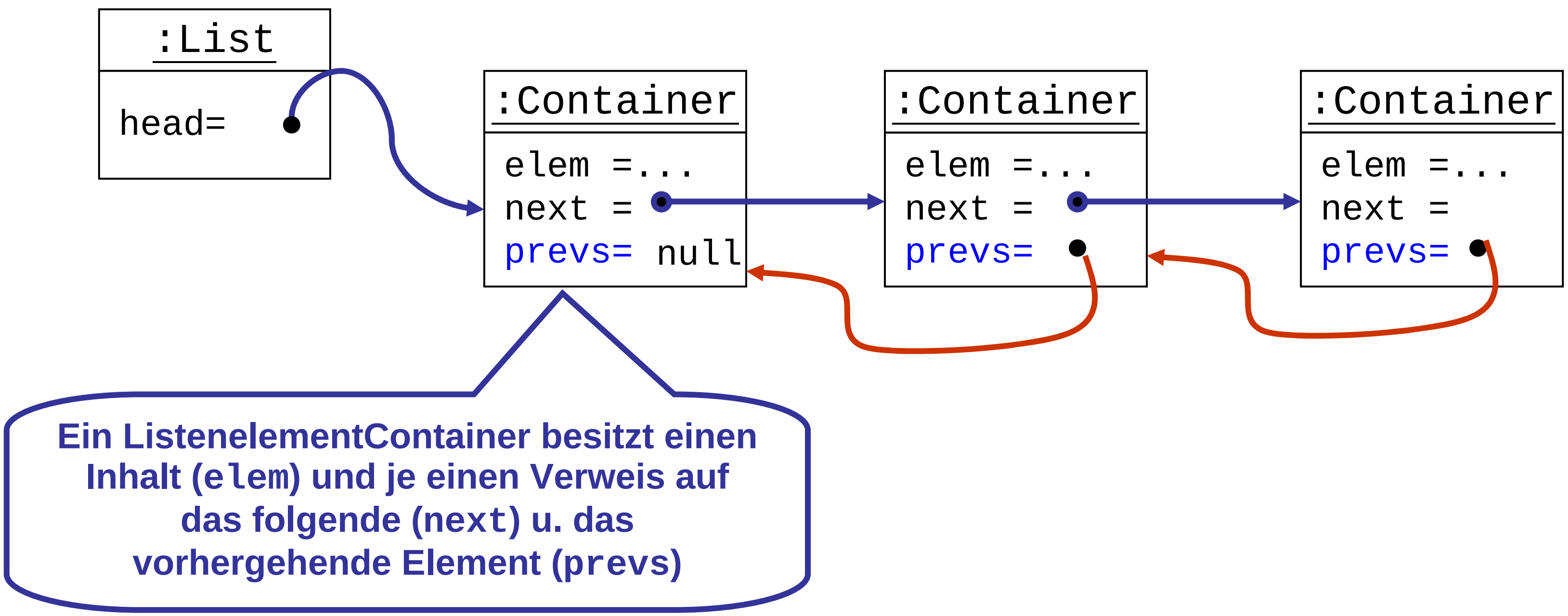


3. Zirkuläre Liste:



Verfeinerung: Doppelt verkettete Listen

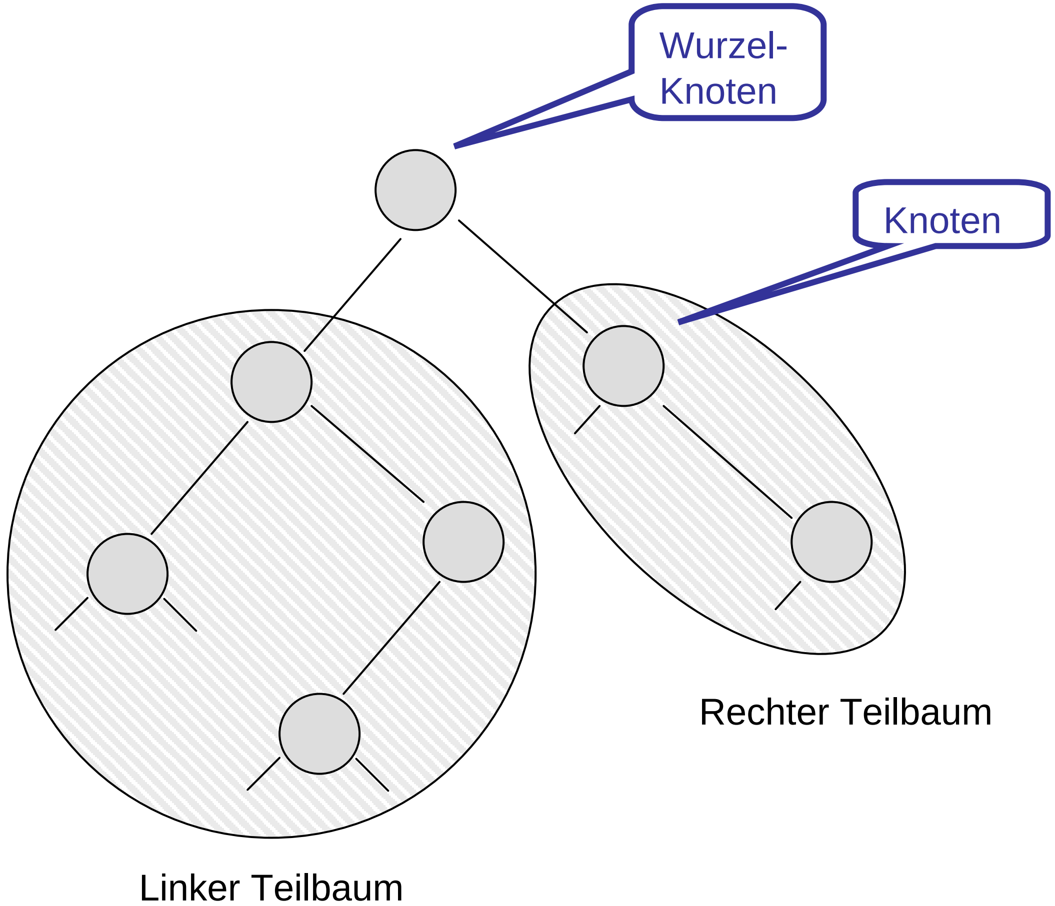
- Doppelt verkettete Listen können auch von rechts nach links durchlaufen werden.



- Die Standardlistenklasse von Java ist doppelt verkettet implementiert.

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 Teilbaum,
  - den rechten Teilbaum.
- In den Knoten kann Information gespeichert werden.



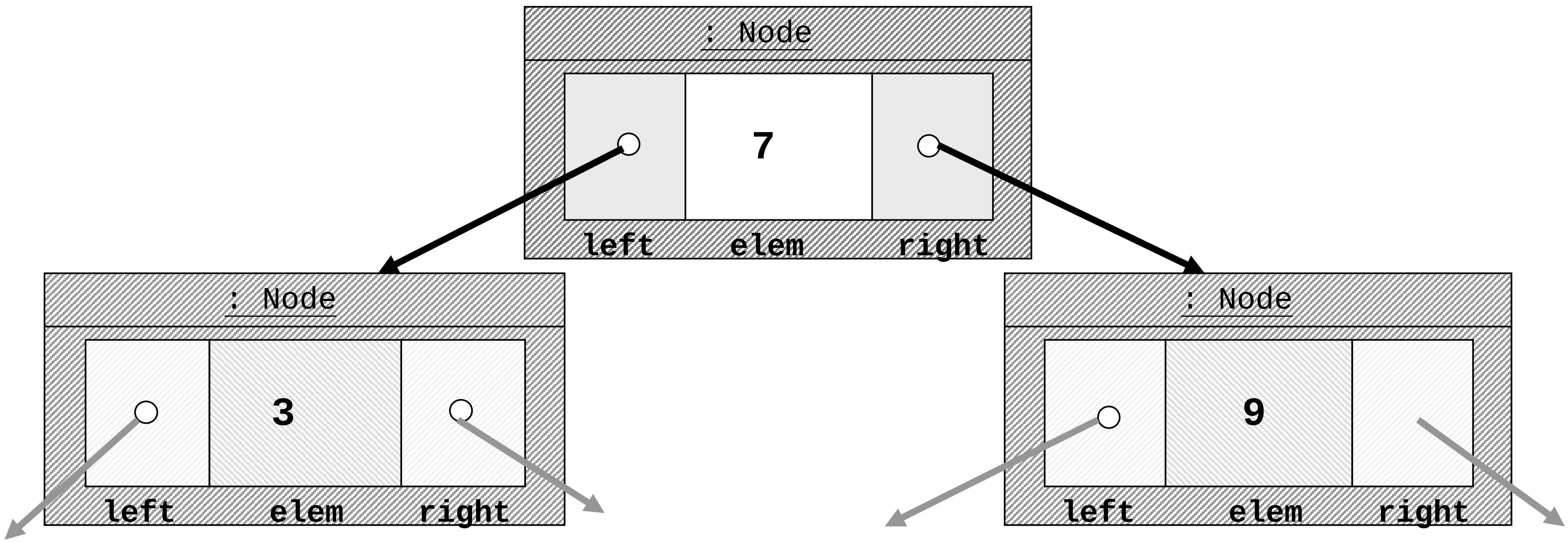
Realisierungen von Binären Bäume

- Mehrere Möglichkeiten:
  - Verallgemeinerung von Linked List auf Container mit zwei Nachfolgerknoten (left, right)
  - Analog rekursiven Datenstrukturen in SML: Eine Unterklasse für jeden Konstruktor (im Sinne von SML).

Implementierung mit Nachfolgerknoten

```
class Node<E extends Comparable<E>>
{  Node<E> left;
   E elem;
   Node<E> right; . . .
}
```

- Ein Knoten wird implementiert als **Objekt mit zwei Zeigern** auf Knoten und **einem (Schlüssel-) Element**
- Häufig wird auch noch ein Datenobjekt gespeichert (auf das wir hier verzichten).



Implementierung von Bäumen

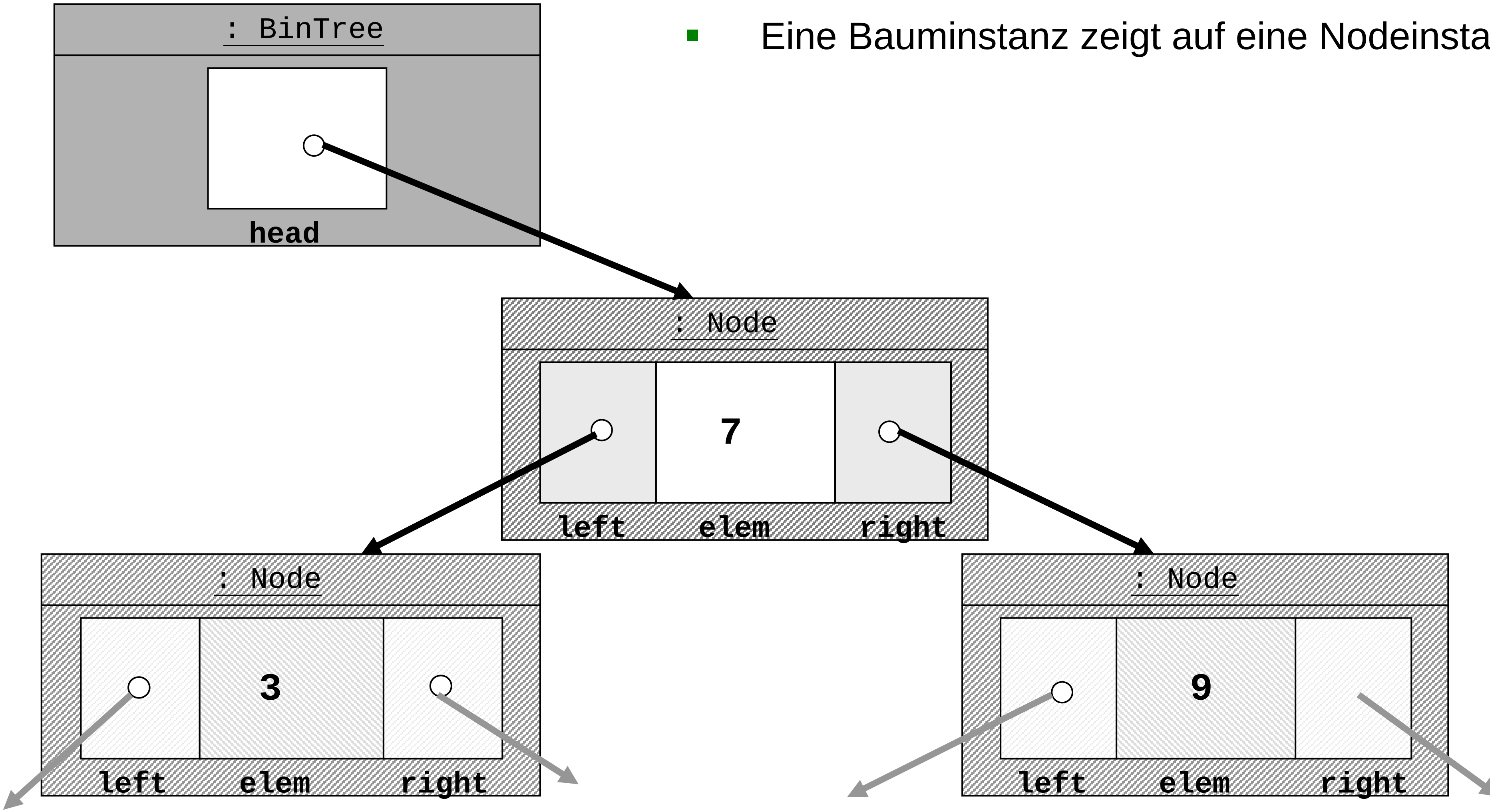
```
public class BinTree<E extends Comparable<E>>
{       Node<E> head;
        . . .
}

class Node<E extends
              Comparable<E>>
{       Node<E> left;
        E elem;
        Node<E> right;

        // Konstruktor
        Node(Node<E> b1, E x, Node<E> b2)
        {  left = b1; elem = x; right = b2;
        }
 . . .
}
```

- 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`.

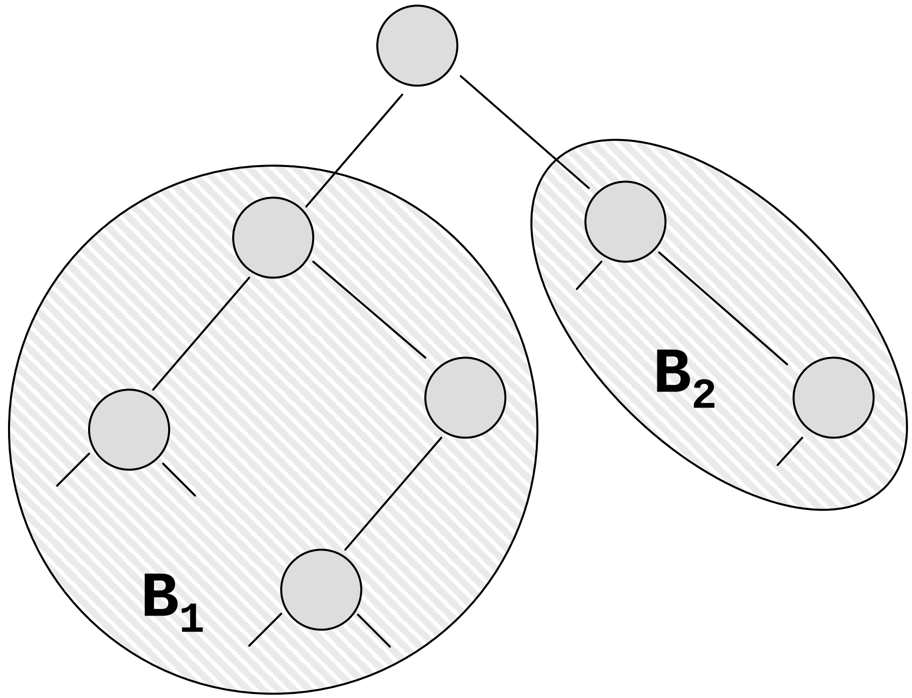
```
class BinTree
{  Node anchor;…
}
```

Ein Beispiel für eine Instanz von BinTree

- Eine Bauminstanz zeigt auf eine Nodeinstanz

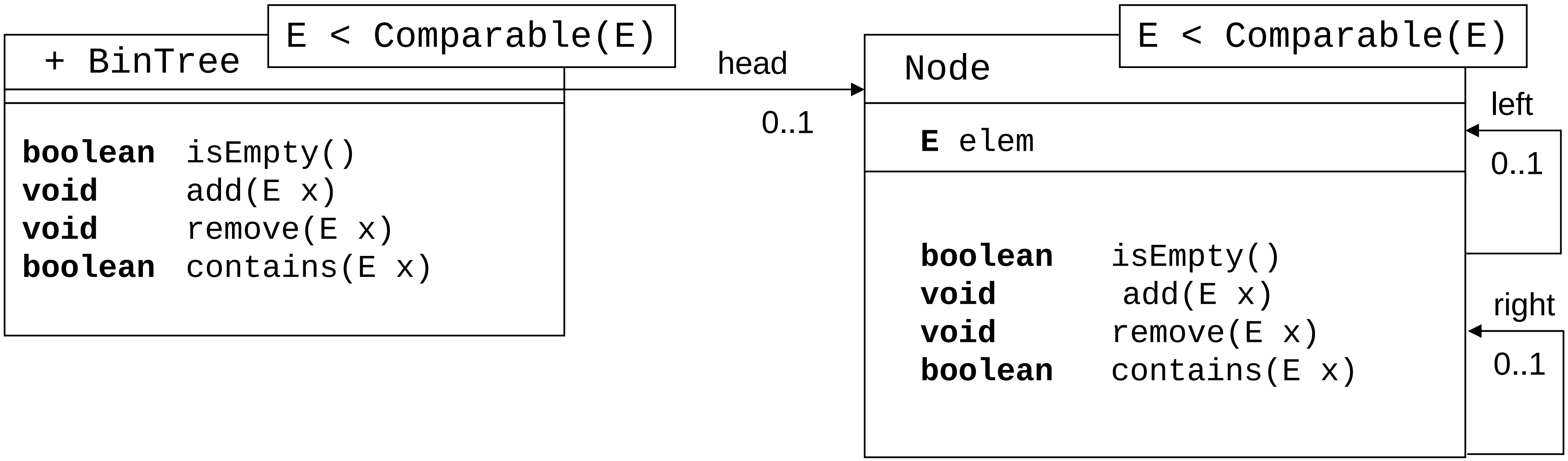


Operationen auf BinTree



- Konstruktoren
  - `BinTree()`
    - der leere Baum
  - `BinTree(x)`
    neuer einelementiger `BinTree`
- Prädikat `isEmpty`
  - Testen, ob ein `BinTree` leer ist
- Operationen
  - `add, remove, contains` wobei
    - `add(x)` x **geordnet** einfügt
    - `remove(x)` x entfernt
    - `contains(x)` prüft, ob x im aktuellen Baum enthalten ist

BinTree in UML



Bemerkung: Anstelle des Rechtecks zur Kennzeichnung der Parametrisierung schreiben wir häufig direkt den Parametertyp zum Namen der Klasse.

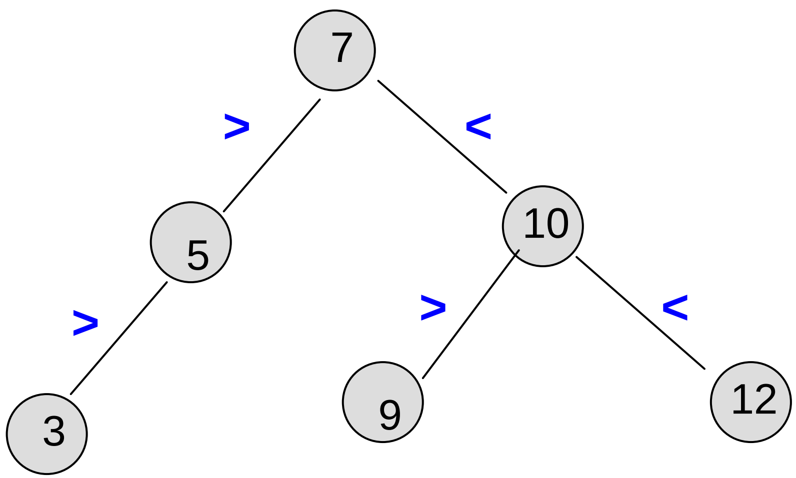
Realisierung in Java

```
public  class BinTree <E extends Comparable<E>> {
        private Node<E> head;

        public BinTree(E x){
                head = new Node<E>(x);
        }

        public  boolean isEmpty(){
                return head == null;
}

class Node<E extends Comparable<E>> {
        private E elem;
        private Node<E> left, right ;


        Node(E x) {
                elem = x;
        }
. . .
```

Geordnete Binärbäume (Suchbäume)

- Ein Binärbaum b heißt **geordnet**, wenn
  - b **leer** ist oder 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



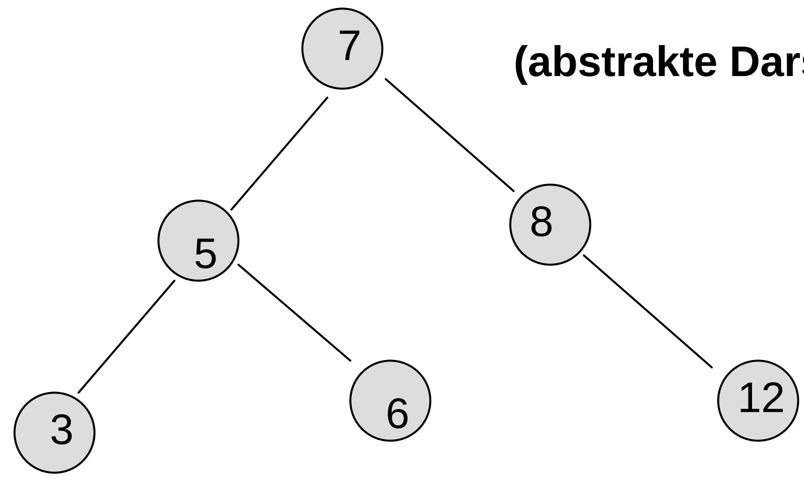
Geordnete Binärbäume (Suchbäume)

- **Beispiel: Geordnet** sind:

  Der leere Baum und

  der Baum t:

`t =`

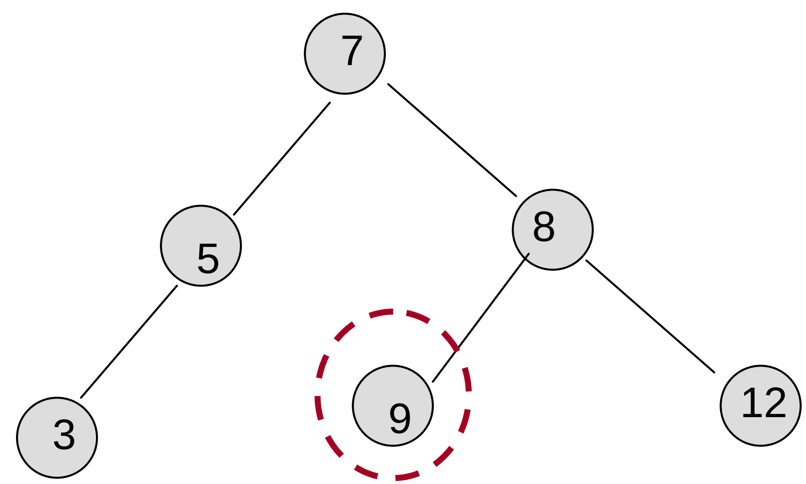


(abstrakte Darstellung)

- **Nicht geordnet** ist

  der Baum t1:

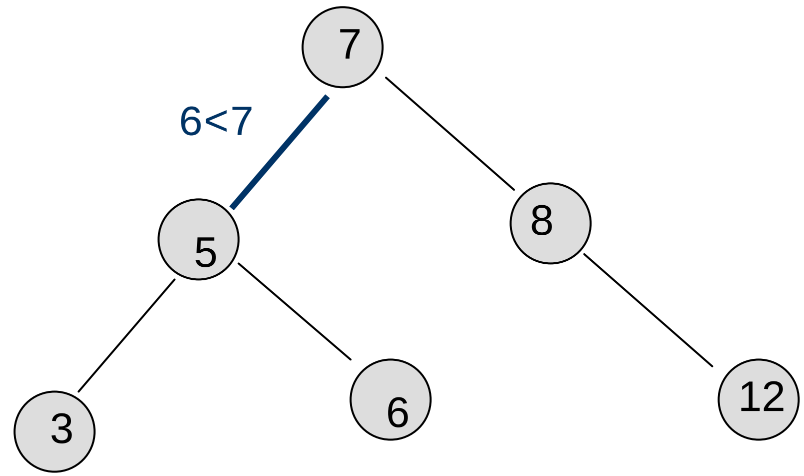
`t1 =`



Suche im geordneten Binärbaum

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

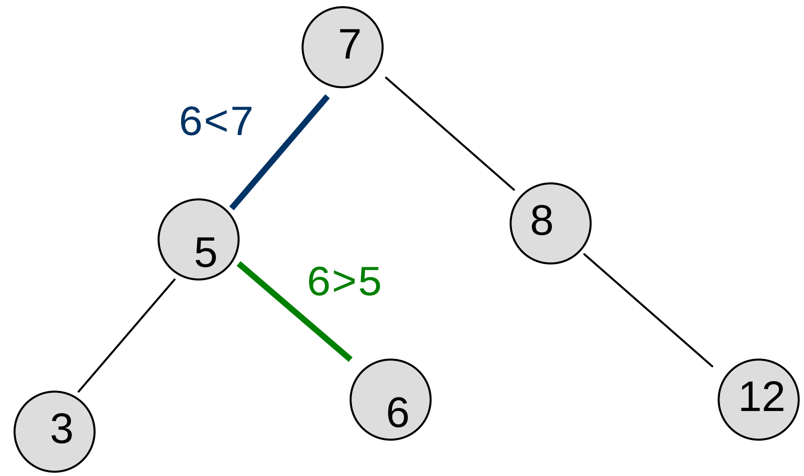
1. Vergleiche 6 mit dem Wert der Wurzel;
2. Da 6<7, gehe zum linken Kindknoten;



Suche im geordneten Binärbaum

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

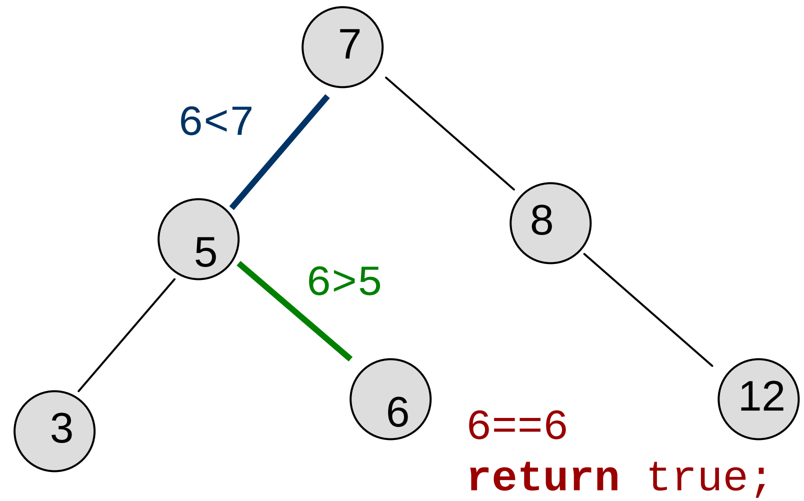
1. Vergleiche 6 mit dem Wert der Wurzel;
2. Da 6<7， gehe zum linken Kindknoten;
3. Vergleiche 6 mit dem Wert dieses Knotens;
4. Da 6>5, gehe zum rechten Kindknoten dieses Knotens;



Suche im geordneten Binärbaum

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

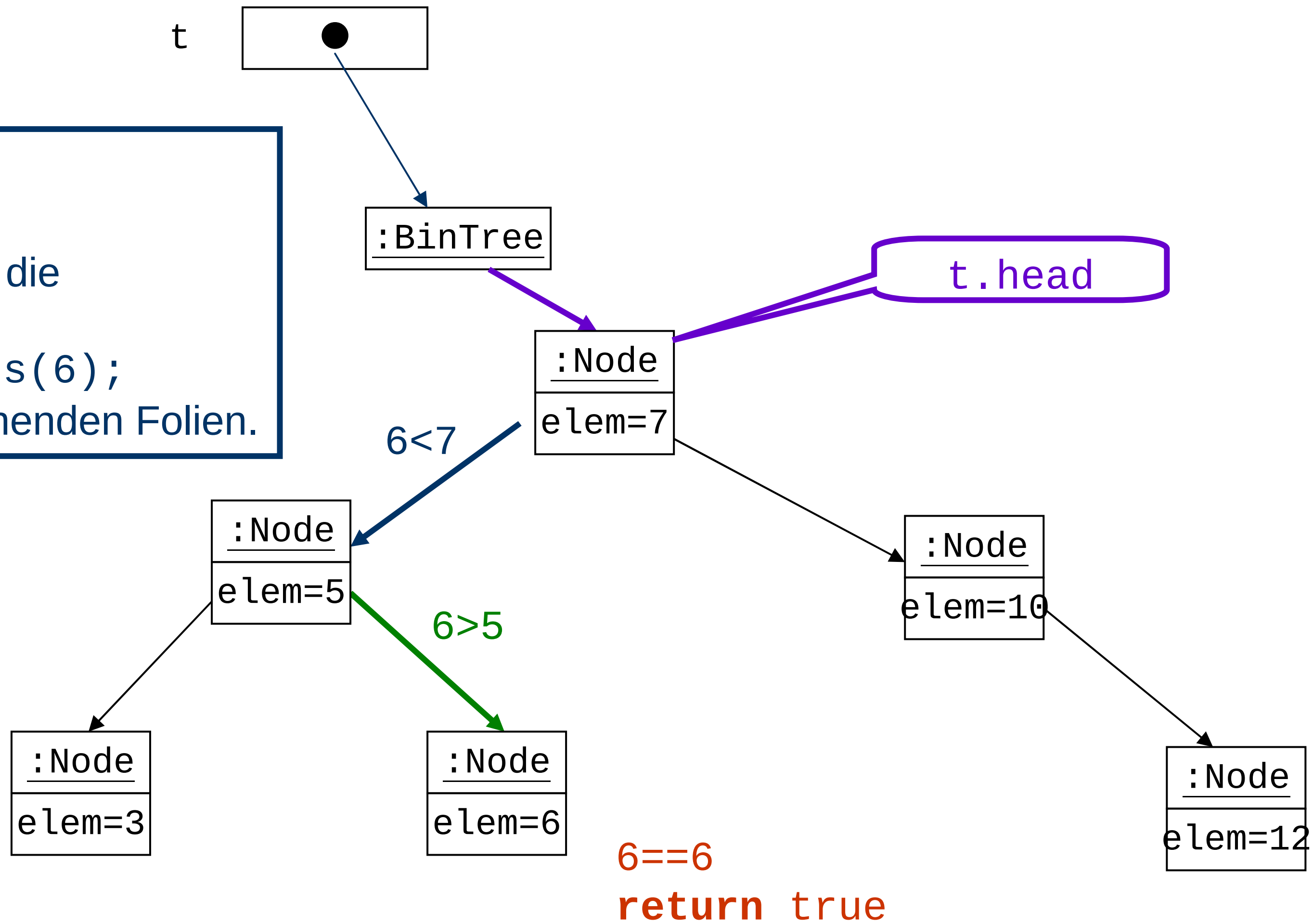
1. Vergleiche 6 mit dem Wert der Wurzel;
2. Da `6<7`, gehe zum linken Kindknoten;
3. Vergleiche 6 mit dem Wert dieses Knotens;
4. Da 6>5, gehe zum rechten Kindknoten dieses Knotens;
5. Vergleiche 6 mit dem Wert dieses Knotens;
6. Da 6==6, gebe Ergebnis true zurück.



Suche im geordneten Binärbaum (Implementierung)

`t.contains(6):`

1. Suche zunächst in BinTree.
2. Wenn t nicht leer, delegiere die Aufgabe an Node durch Aufruf von `head.contains(6);`
3. Verfahre wie auf den vorgehenden Folien.



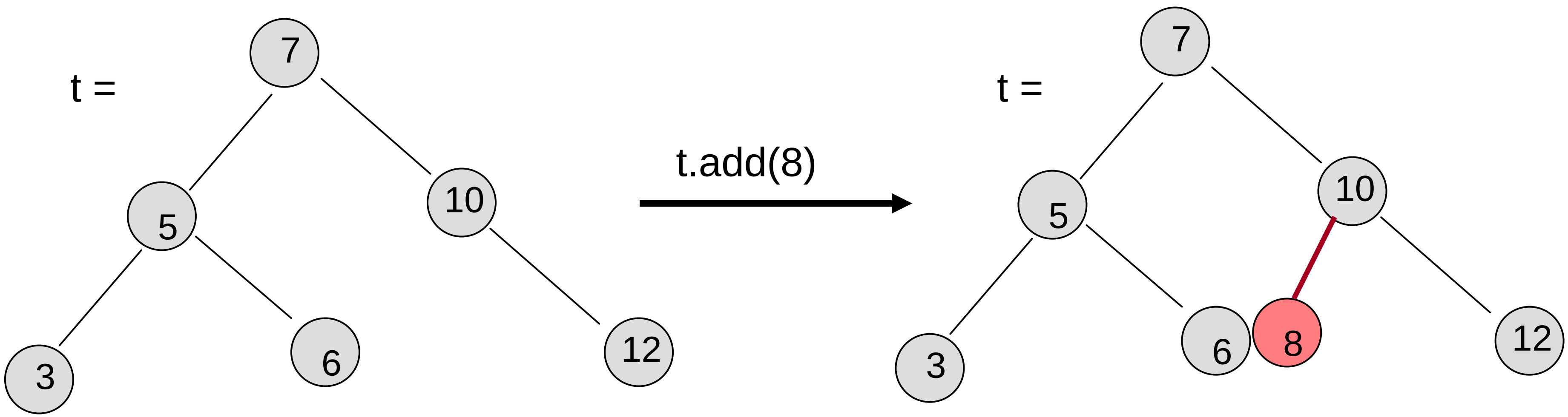
Suche im geordneten Binärbaum

```
 public boolean contains(E x)
   { if (head == null) return false;
     else return head.contains(x);
   }
wobei

class Node<E extends Comparable<E>>
{ . . .
     boolean contains (E x)
     {  Node current = this;
        while(current.elem != x)        // solange nicht gefunden,
        { if (x < current.elem)         // 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
     }
```

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

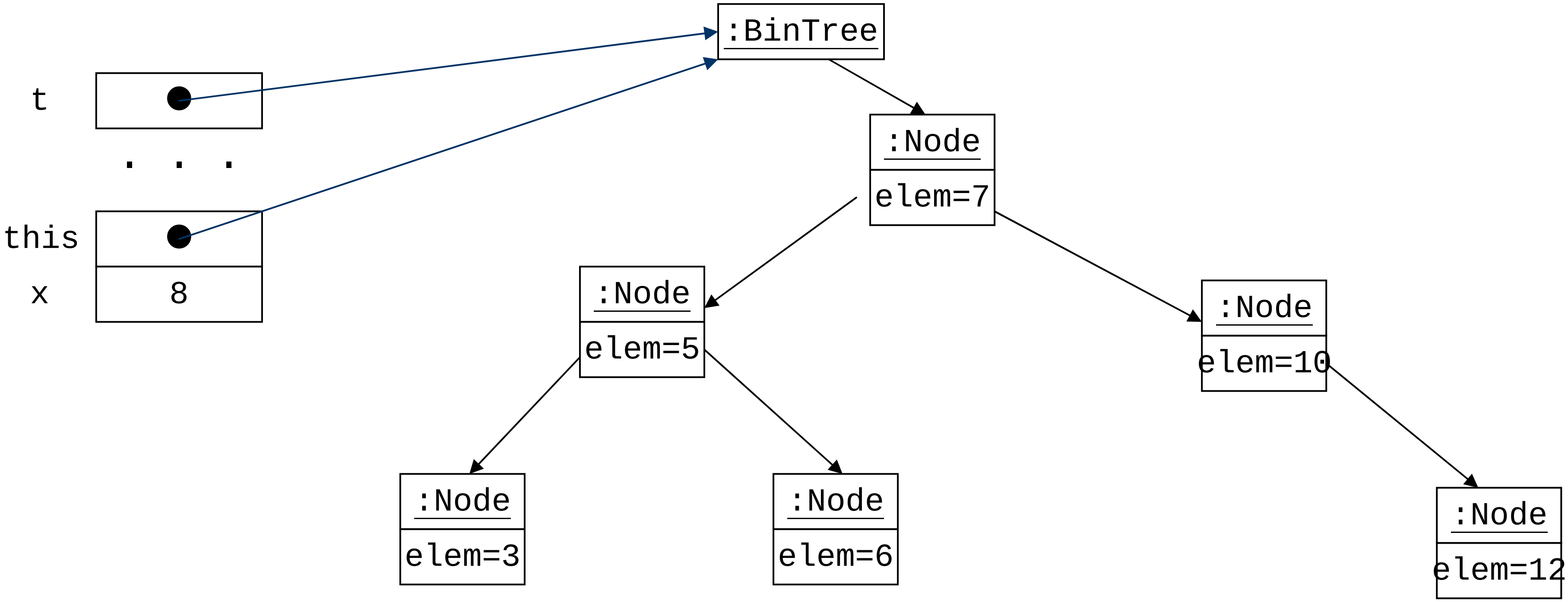
Einfügen in geordneten Binärbaum

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



Einfügen in geordneten Binärbaum (Implementierung)

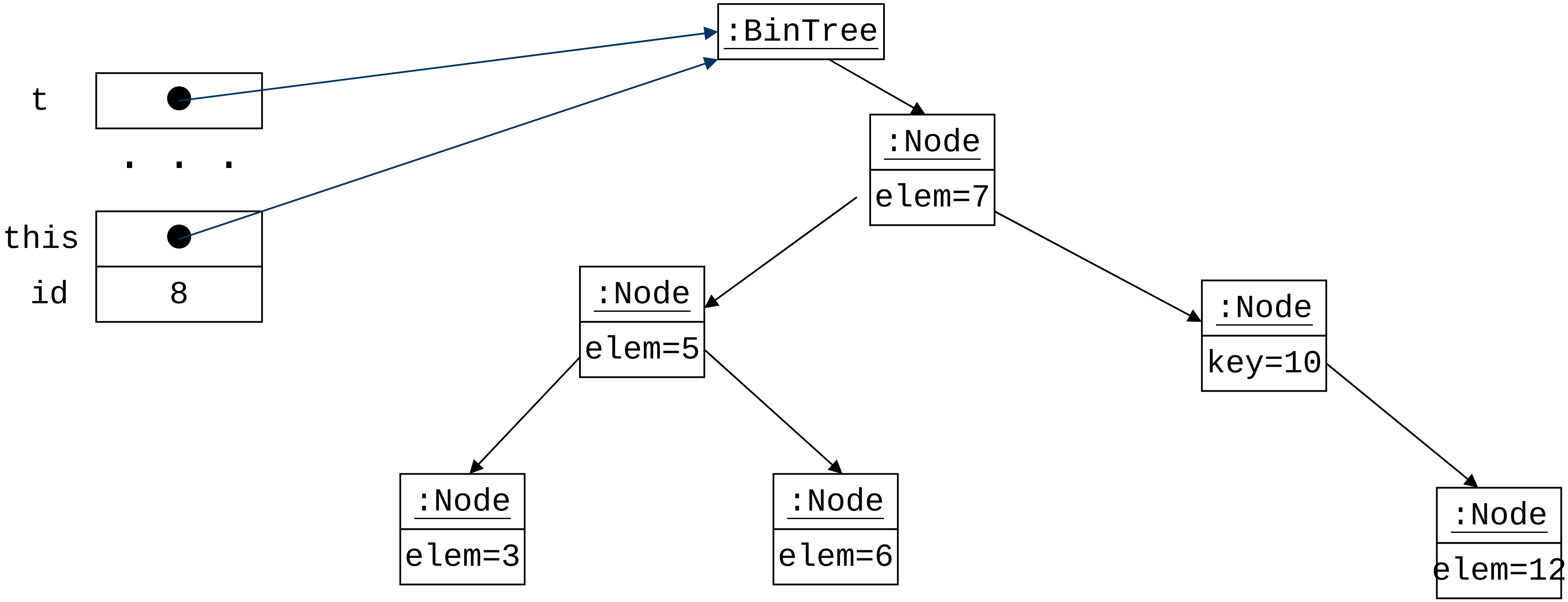
t.insert(8)



Aufruf von `t.add(x):`

Einfügen in geordneten Binärbaum (Implementierung)

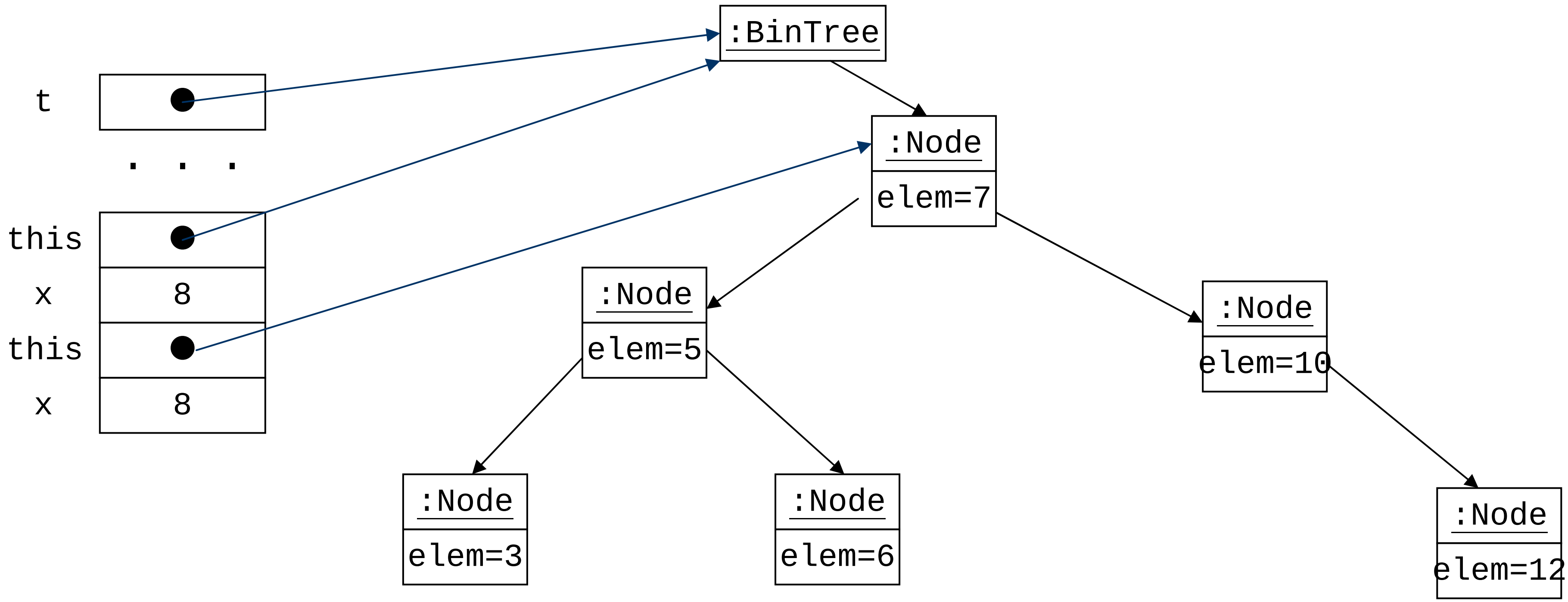
t.insert(8)



Delegieren der Aufgabe durch Aufruf von `head.add(x):`

Einfügen in geordneten Binärbaum (Implementierung)

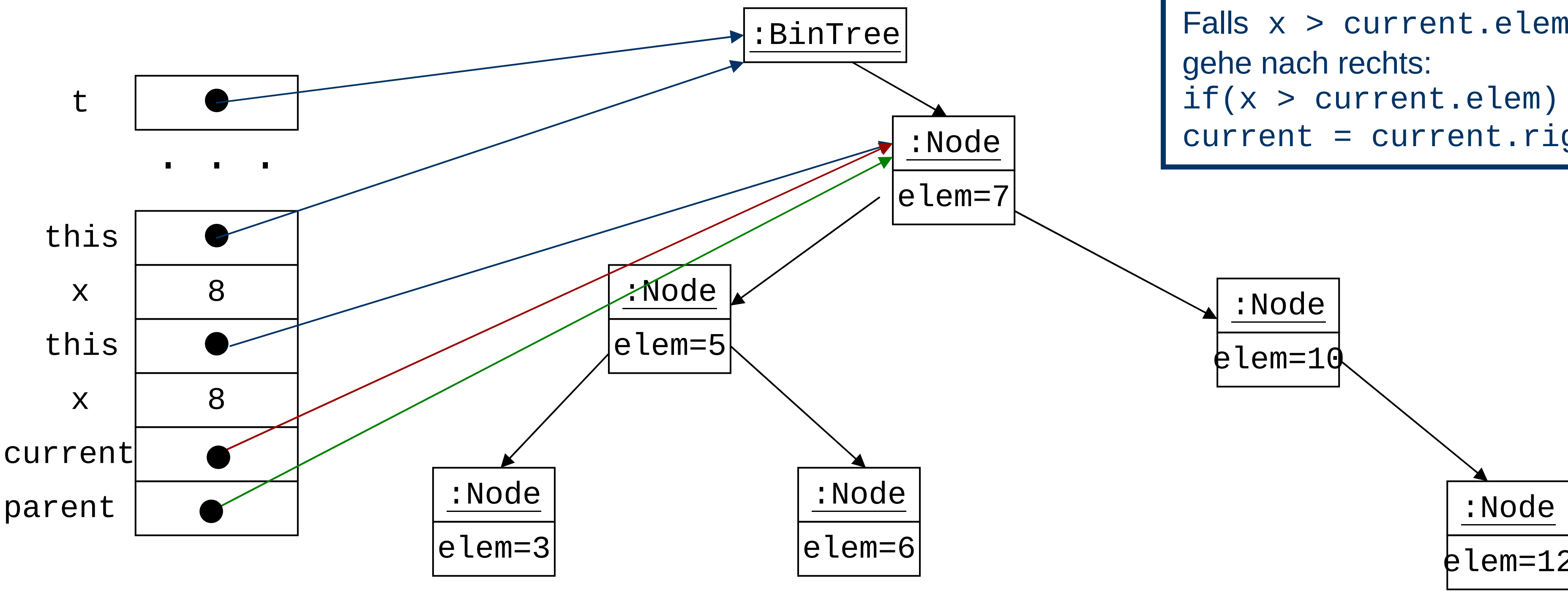
anchor.insertKey(8):



Delegieren der Aufgabe durch
Aufruf von `head.add(x)`:
Durchlauf durch das Node-Geflecht mit
zwei Hilfsvariablen `current` und `parent`

Einfügen in geordneten Binärbaum (Implementierung)

anchor.insert(8):



Falls `x > current.elem`,
gehe nach rechts:

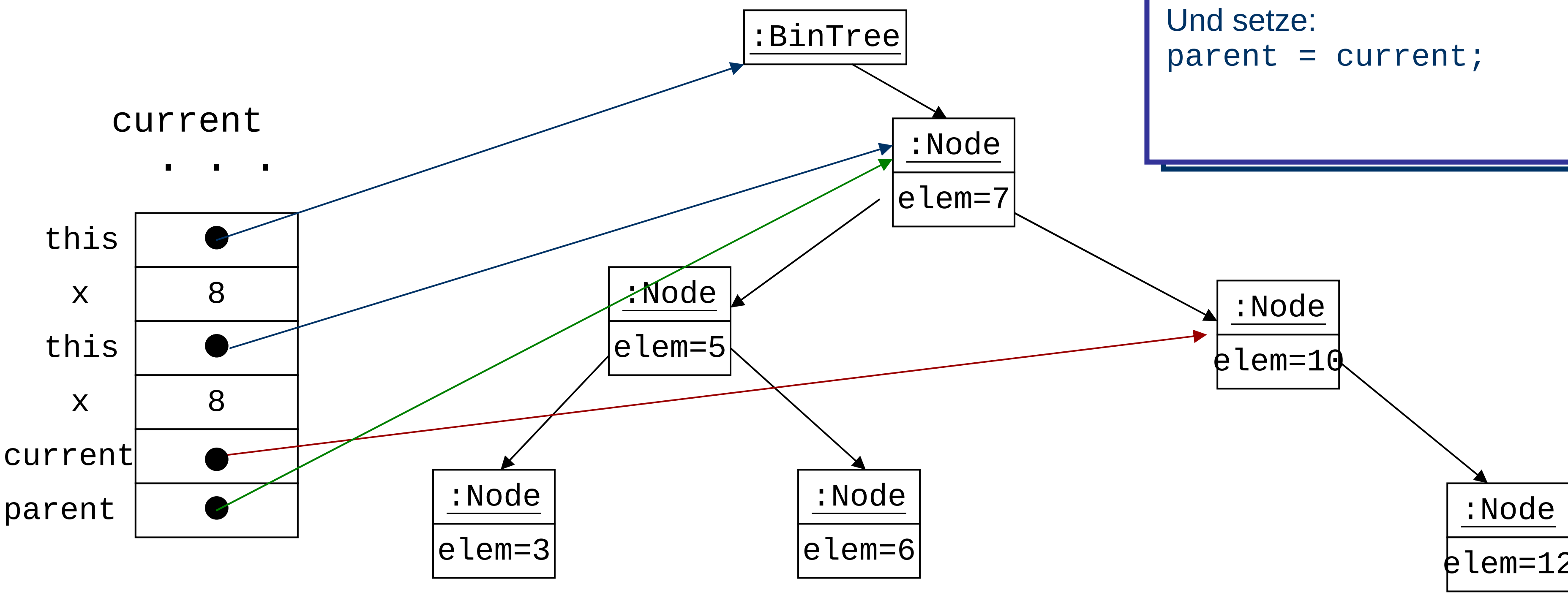
```
if(x > current.elem)
current = current.right;
```

Delegieren der Aufgabe durch Aufruf von `head.add(x)`:
Durchlauf durch das Node-Geflecht mit zwei Hilfsvariablen `current` und `parent`

Einfügen in geordneten Binärbaum (Implementierung)

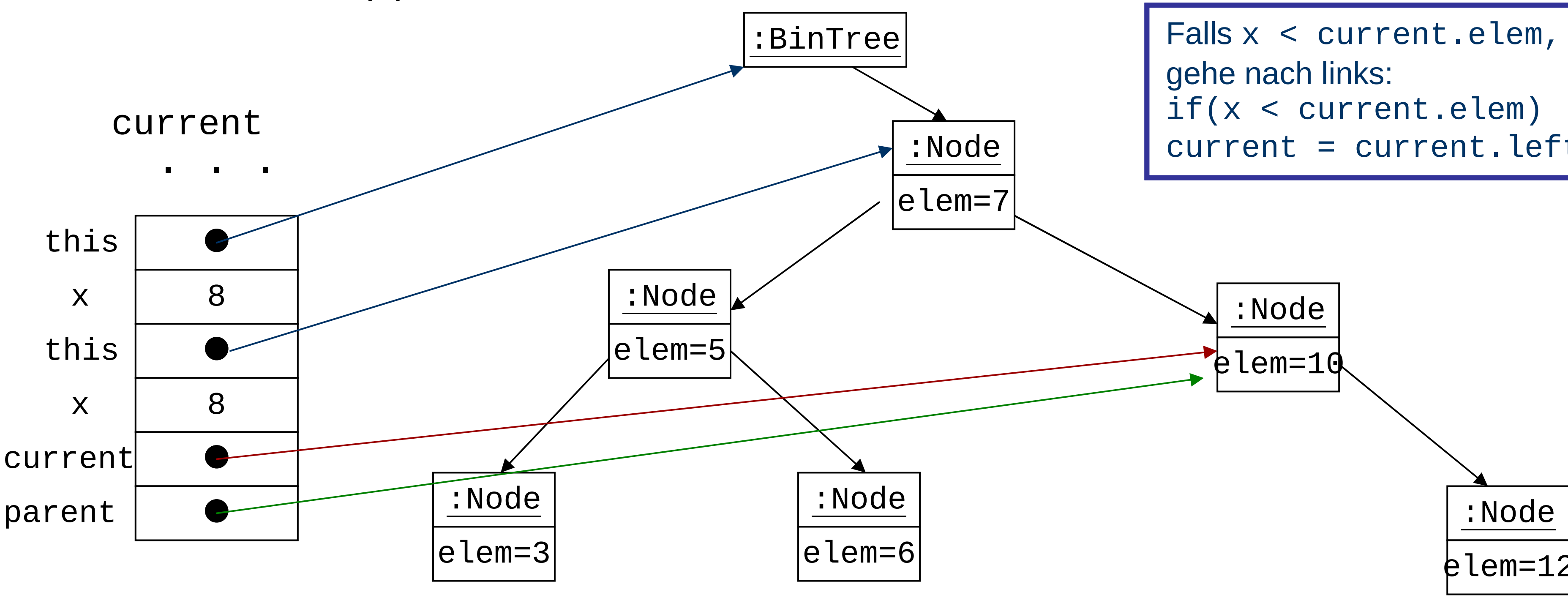
anchor.insert(8):

Und setze:
`parent = current;`



Einfügen in geordneten Binärbaum (Implementierung)

anchor.insert(8):



Falls `x < current.elem`,
gehe nach links:

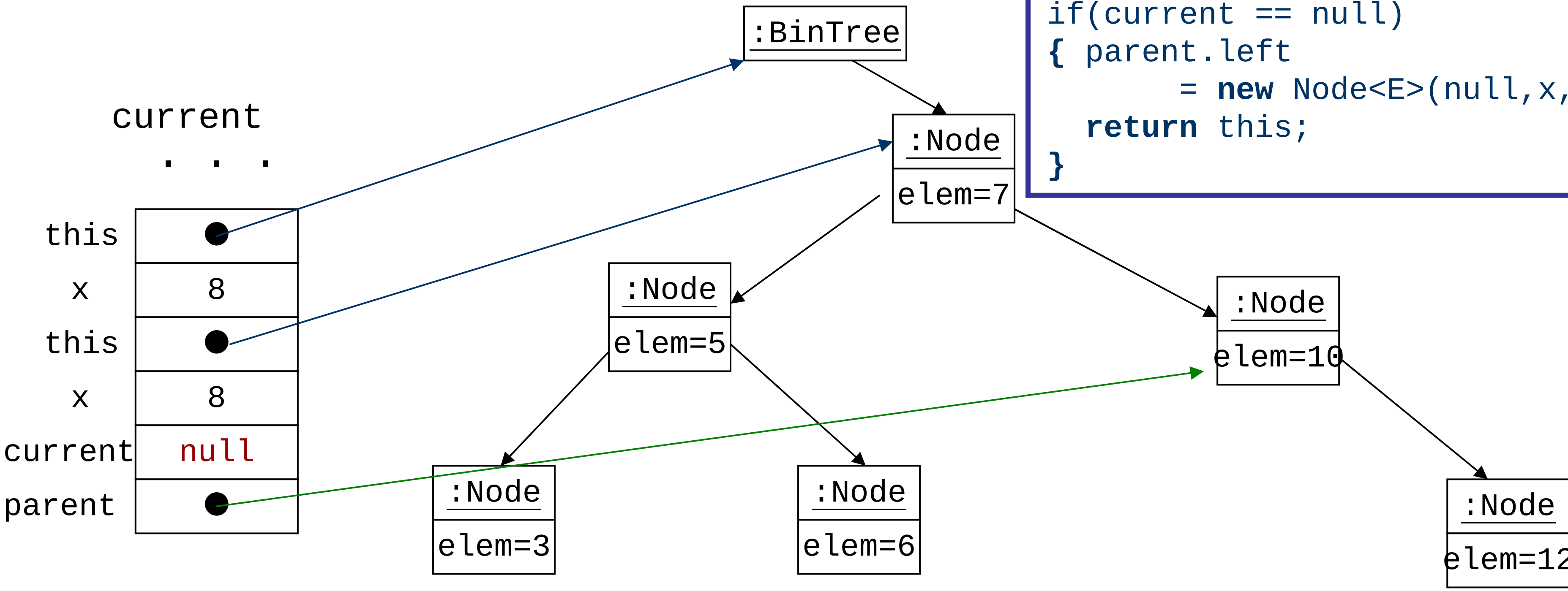
```
if(x < current.elem)
current = current.left;
```

Einfügen in geordneten Binärbaum (Implementierung)

anchor.insert(8):

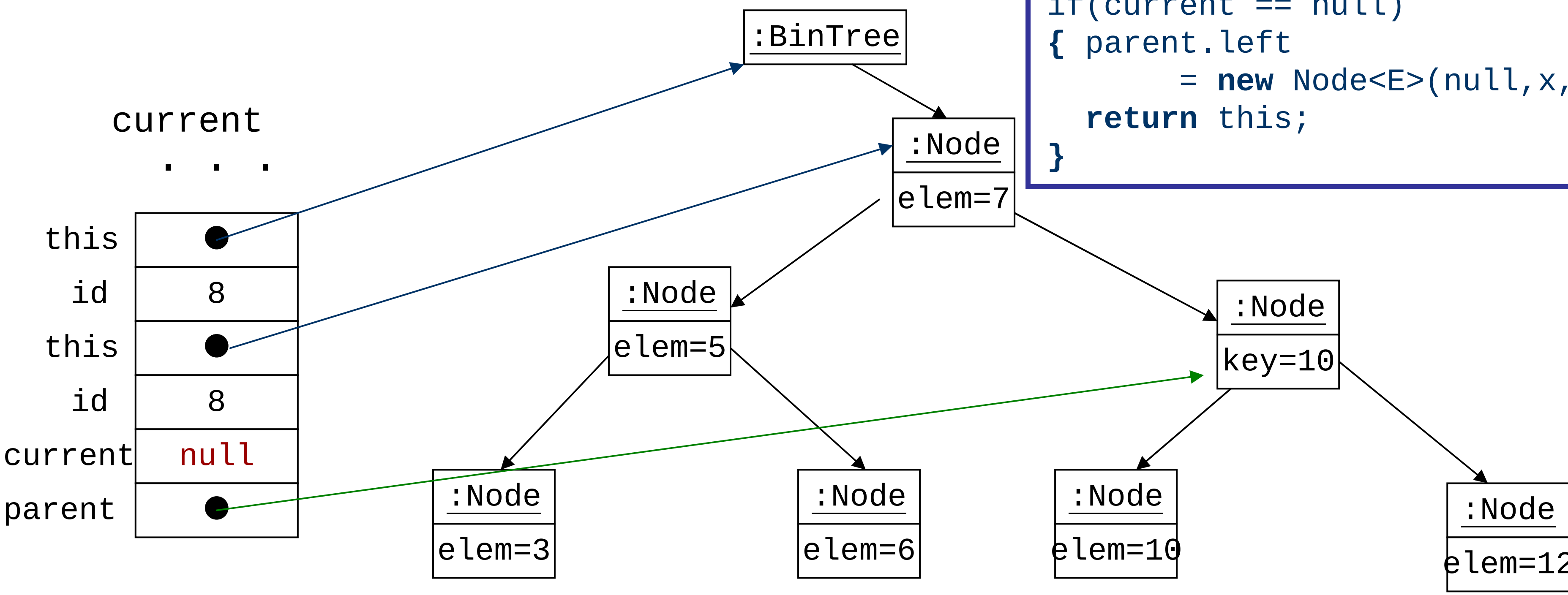
Wenn `current= null`, füge neuen Knoten ein:

```
if(current == null)
{ parent.left
        = new Node<E>(null,x,null);
  return this;
}
```



Einfügen in geordneten Binärbaum (Implementierung)

anchor.insert(8):



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

```
if(current == null)
{ parent.left
        = new Node<E>(null,x,null);
  return this;
}
```

Einfügen in geordneten Binärbaum

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

```
public void add(E x)
{   if(head==null)                      // falls kein Knoten im head
        head = new Node<E>(null, x, null);    // neuer Knoten
    else  head = head.add(x);
}
```

wobei `add` in **`class`** `Node<E>` folgendermaßen definiert wird:

Einfügen in geordneten Binärbaum (Implementierung)

```
Node<E> add(E x)
{    Node<E> current = this;        // starte bei this
     Node<E> parent;
     while(true)                    // terminiert intern
     {   parent = current;
         if(x.compareTo(current.elem)<0)  // falls x < current.elem ,
        { current = current.left;         // gehe nach links
         if(current == null)  // am Ende füge links ein
            {   parent.left = new Node<E>(null, x, null);
                return this;
            }
         }  // end if go left
         else                 // falls x > current.elem, gehe nach rechts
         {  current = current.right;
            if(current == null)  // am Ende füge rechts ein
            {   parent.right = new Node<E>(null, x, null);
                return this;
            }
         }  // end else go right
     }  // end while
}
```

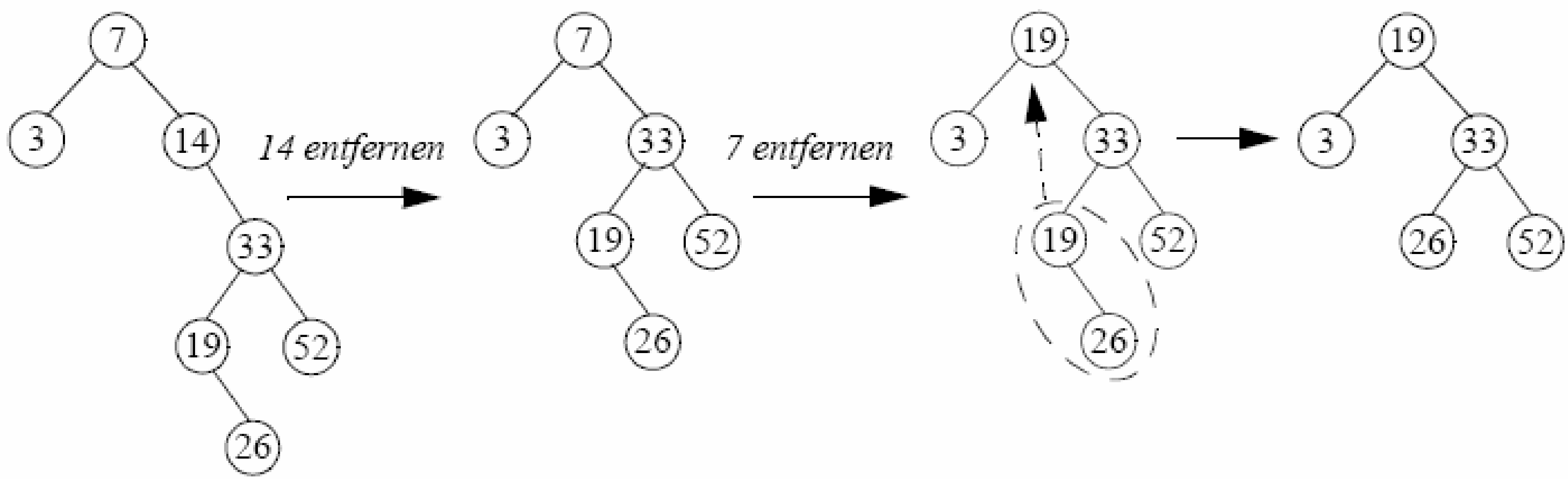
Fügt einen neuen Knoten passend ein

Achtung:x darf nicht im Baum vorkommen!

Löschen in geordneten Binärbaum

Beim Entfernen eines Schlüssels muss die Suchbaumstruktur aufrecht erhalten werden. Das ist etwas schwieriger bei inneren Knoten.

Beispiel:

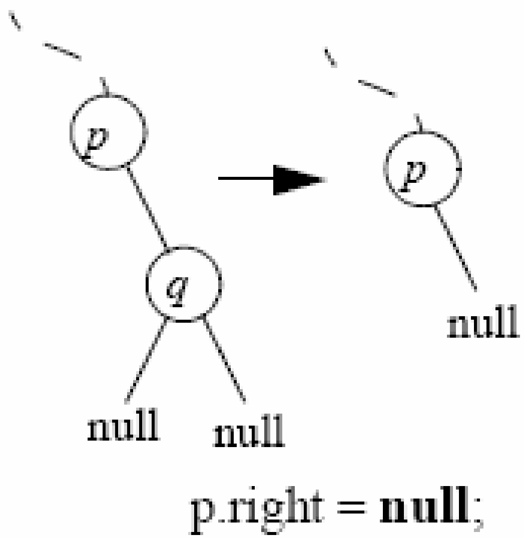


Löschen in geordneten Binärbaum

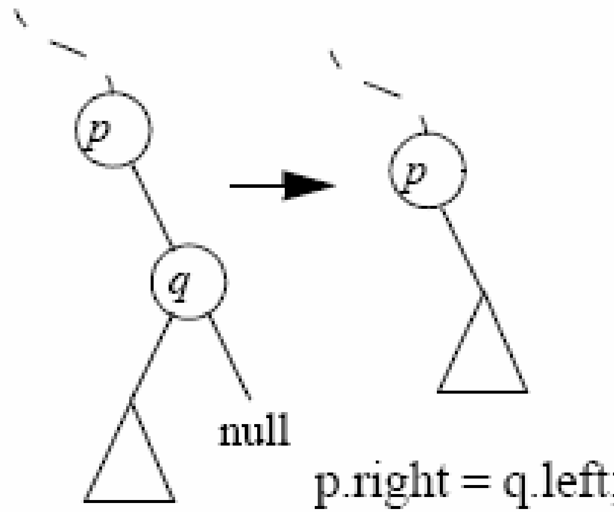
Allgemein treten zwei grundsätzlich verschiedene Situationen auf:

Fall 1: der Knoten *q* mit dem zu entfernenden Schlüssel besitzt höchstens einen Sohn (Blatt oder Halbblatt):

a) der Knoten *q* besitzt keinen Sohn



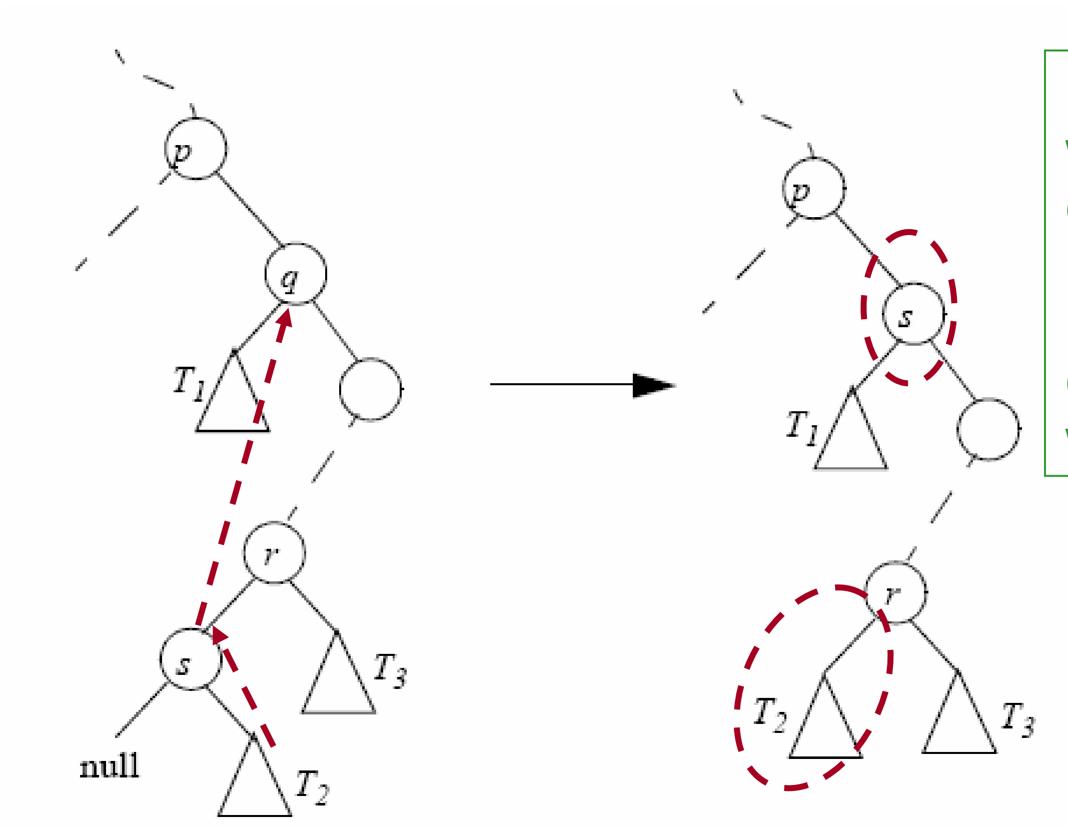
b) der Knoten *q* besitzt einen linken Sohn (rechts: *symmetrisch*)



Löschen in geordneten Binärbaum

Fall 2:

Der Knoten *q* mit dem zu entfernenden Schlüssel besitzt zwei Söhne (innerer Knoten):



- Der zu löschende Schlüssel von q wird durch **den kleinsten der größeren Schlüssel** ersetzt.
- Dieser Schlüssel befindet sich stets in einem Blatt oder einem Halbblatt, das daraufhin aus dem Baum entfernt wird.

Löschen in geordneten Binärbaum (Implementierung)

```
 public void remove(E x) {
       head = head.remove(x, head);}

public Node<E> remove(E x, Node<E> t) {
   if (t == null)
       throw new RuntimeException("x does not exist(remove)");
   if (t.getElem().compareTo(x) < 0) {       // falls t.elem<x,
       t.setRight(remove(x, t.getRight())); //lösche x  in rechtem TBaum
   } else if (x.compareTo(t.getElem()) < 0) {// falls x<t.elem,
         t.setLeft(remove(x, t.getLeft())); //lösche x  in linkem TBaum
   } else if (t.getLeft()!=null && t.getRight()!=null) {//x in innerem Knoten
         t.setElem(findMin(t.getRight()).getElem());//ersetze x durch kleinsten
                                                    //Schlüssel des rechten TBaums und
         t.setRight(deleteMin(t.getRight())); //entferne diesen Knoten
   } else  // falls x ein Blatt oder  Halbblatt, setze neue Wurzel
         t = (t.getLeft() != null) ? t.getLeft() : t.getRight();
   return t;
}
```

Löschen in geordneten Binärbaum (Implementierung)

```
public Node<E> findMin(Node<E> t) {
   if (t == null)
       throw new RuntimeException("elem not found in findMin");
   while (t.getLeft() != null)
       t = t.getLeft();
   return t;
}
```

findMin(t) sucht den kleinsten Knoten im Suchbaum t

```
public Node<E> deleteMin(Node<E> t) {
   if (t == null)
       throw new RuntimeException("elem not found in deleteMin");
   if (t.getLeft() != null)
       t.setLeft(deleteMin(t.getLeft()));
   else
       t = t.getRight();
   return t;
}
```

deleteMin(t) entfernt den kleinsten Knoten im Suchbaum t und setzt dessen rechten Teilbaum um.

Laufzeitanalyse von add, remove, contains

- Alle drei Methoden

  `add, remove, contains`

  sind **linear in der Höhe h des Suchbaums**:

  Im schlechtesten Fall ist der Aufwand O(h) und damit O(n),

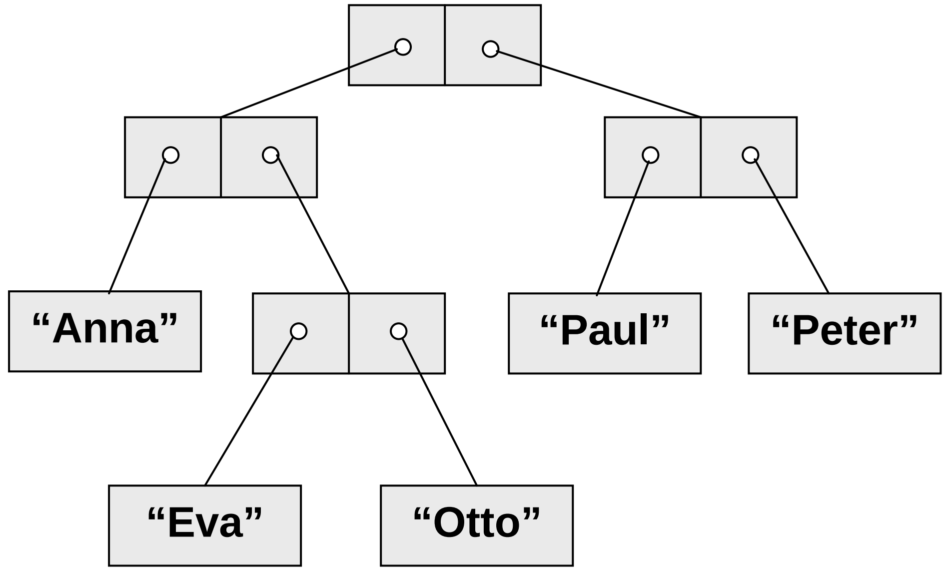
  wenn n die Anzahl der Knoten bezeichnet.

- Im durchschnittlichen Fall
  - wenn der Baum nur durch Einfügungen entstanden ist und alle möglichen Permutationen der Eingabereihenfolge sind gleichwahrscheinlich

  ist die Höhe logarithmisch, d.h der durchschnittliche Zeitbedarf ist O(log n).

Bäume mit Blättern

- Jeder Zweig soll in einem Blatt enden
- Die Information speichern wir in Blättern
- Jeder Knoten hat zwei Unterbäume




```
class Knoten
{       Knoten links;
        Knoten rechts;
}
```

```
class Blatt
{       String elem;
}
```

Wir haben ein Problem: Wir müssen auch zulassen:

```
Blatt links;
Blatt rechts;
```

... aber ein Blatt ist kein Knoten!

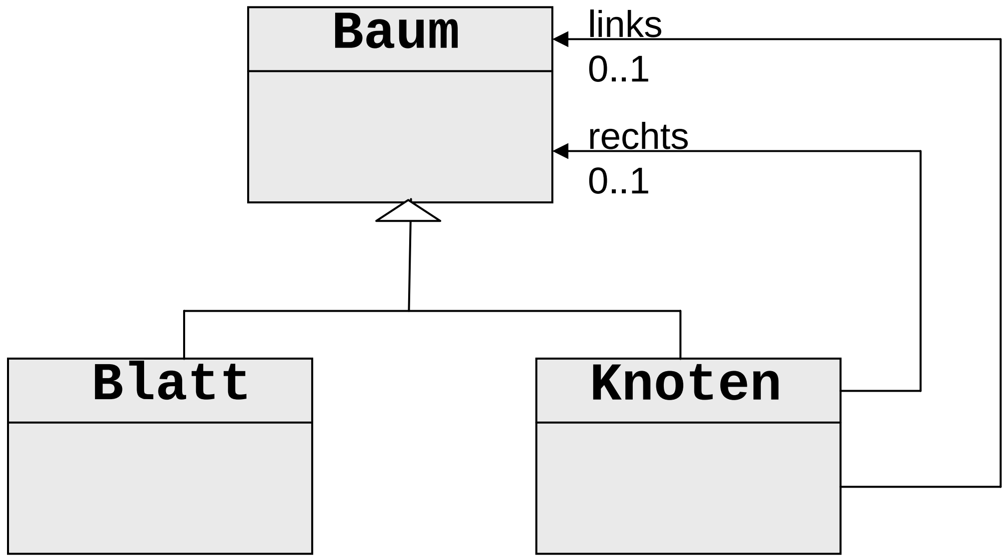
Baum in UML

- Wir wollen Blatt und Knoten zu einer Klasse Baum zusammenfassen.

- Blatt wird Unterklasse von Baum
- Knoten wird Unterklasse von Baum

```
class Blatt extends Baum

class Knoten extends Baum
```

- Viele Methoden müssen für alle Bäume funktionieren
  - istBlatt()
  - istKnoten
  - contains()
  - toString()



Default-Methoden redefinieren

- In **Baum** definieren wir die Methoden irgendwie:

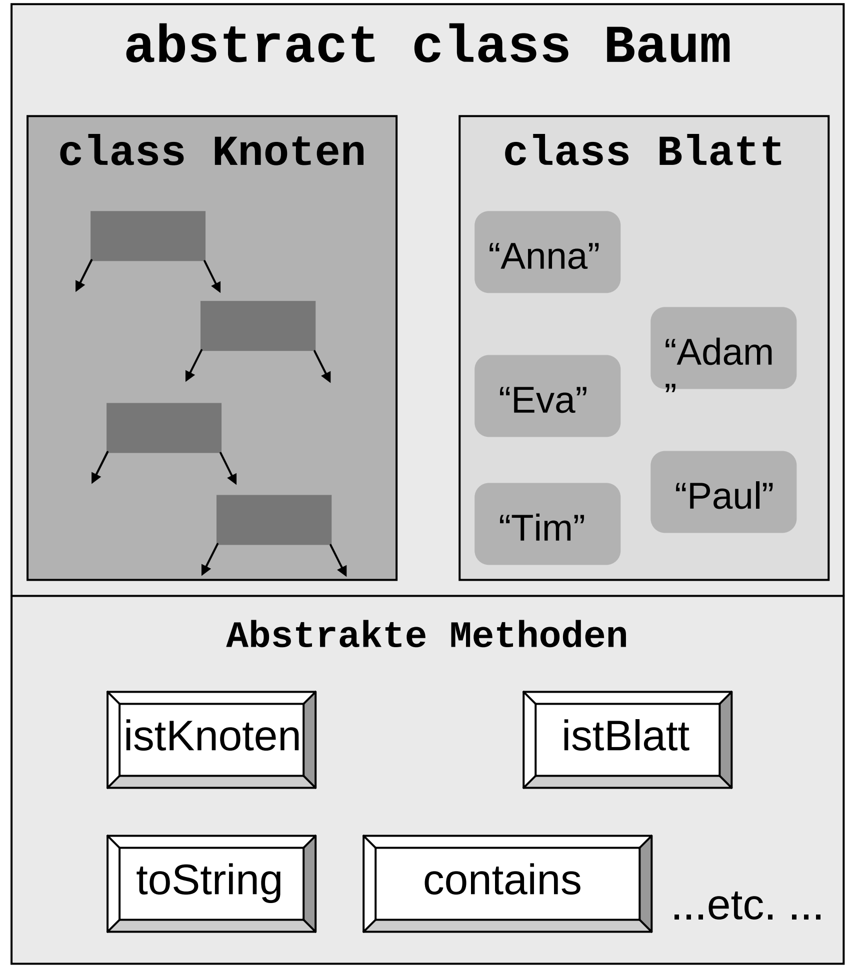
```
boolean isBlatt(){
{   return false;    // äähm na ja...
}
void toString(){}           // tu nix
```

- In den Unterklassen redefinieren wir sie wieder

```
// z.B. In Blatt:
boolean isBlatt ()
{   return true;
}
void toString()
{   System.out.println(info);
}
```

Besser: Abstrakte Klassen

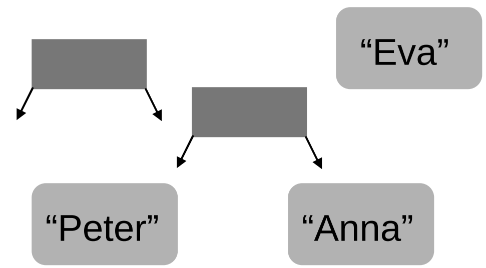
- Vereinigung von Unterklassen
- Gemeinsame Methoden
  - In der Oberklasse abstrakt erklärt
    - Nur die Signatur wird aufgeführt
  - In jeder nicht abstrakten Unterklasse implementiert
- Beispiel
  - Jedes Blatt ist ein Baum
  - Jeder Knoten ist ein Baum
  - Definiere Baum als abstrakte Klasse, die Blatt und Knoten umfasst



Abstrakte Klasse Baum

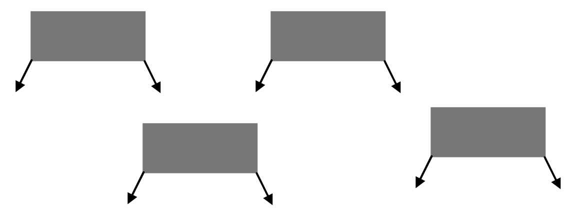
- Klassen werden wechselseitig rekursiv, vgl. die Implementierung von Bäumen in SML.

```
abstract class Baum
{   ...
}
```



Jeder Knoten ist ein Baum

```
class Knoten extends Baum
{  Baum links;
   Baum rechts;
       ...
}
```



Jedes Blatt ist ein Baum

```
class Blatt extends Baum
{   String elem;
}
```

Implementierung

- Abstrakte Methoden müssen in (konkreten) Unterklassen implementiert werden
- Wird vom Computer geprüft

```
abstract class Baum<E>{
   abstract boolean istBlatt();
   abstract boolean contains(E x);
       ...
}
```

```
class Knoten<E> extends Baum<E>{
   Baum<E> links, rechts ;
   boolean istBlatt()
   {   return false;}
   boolean contains(E x){
   {   return
          links.contains(x) ||
          rechts.contains(x);
   }
}
```

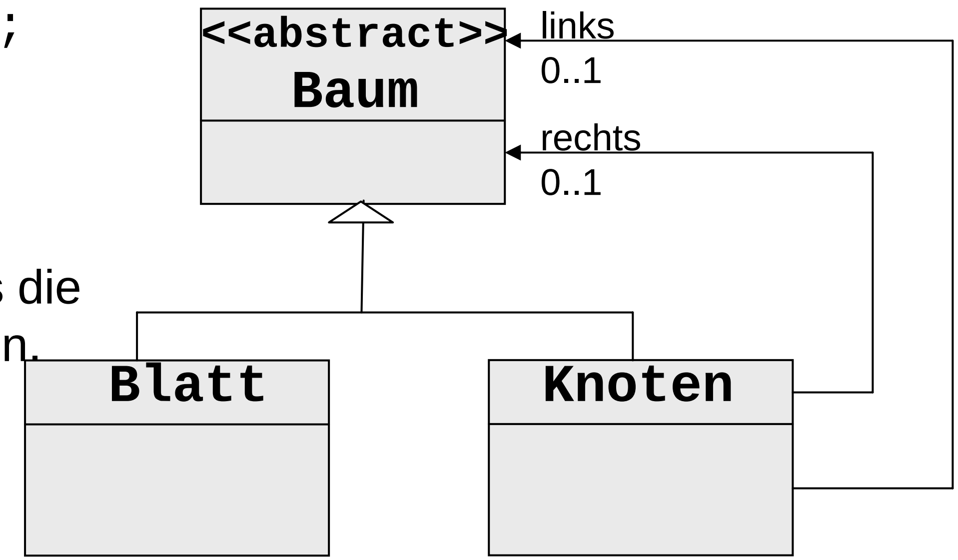
```
class Blatt<E> extends Baum<E>
{  E elem;
   boolean istBlatt(){
   {   return true;}
   boolean contains(E x)
   { return
     (elem.compareTo(x) == 0); }
}
```

Abstrakte Klassen

- Haben **keine** eigenen Objekte
  - Was sollte auch new Baum() liefern:
  - ein Blatt oder einen Knoten?
- Können abstrakte und konkrete Methoden enthalten:

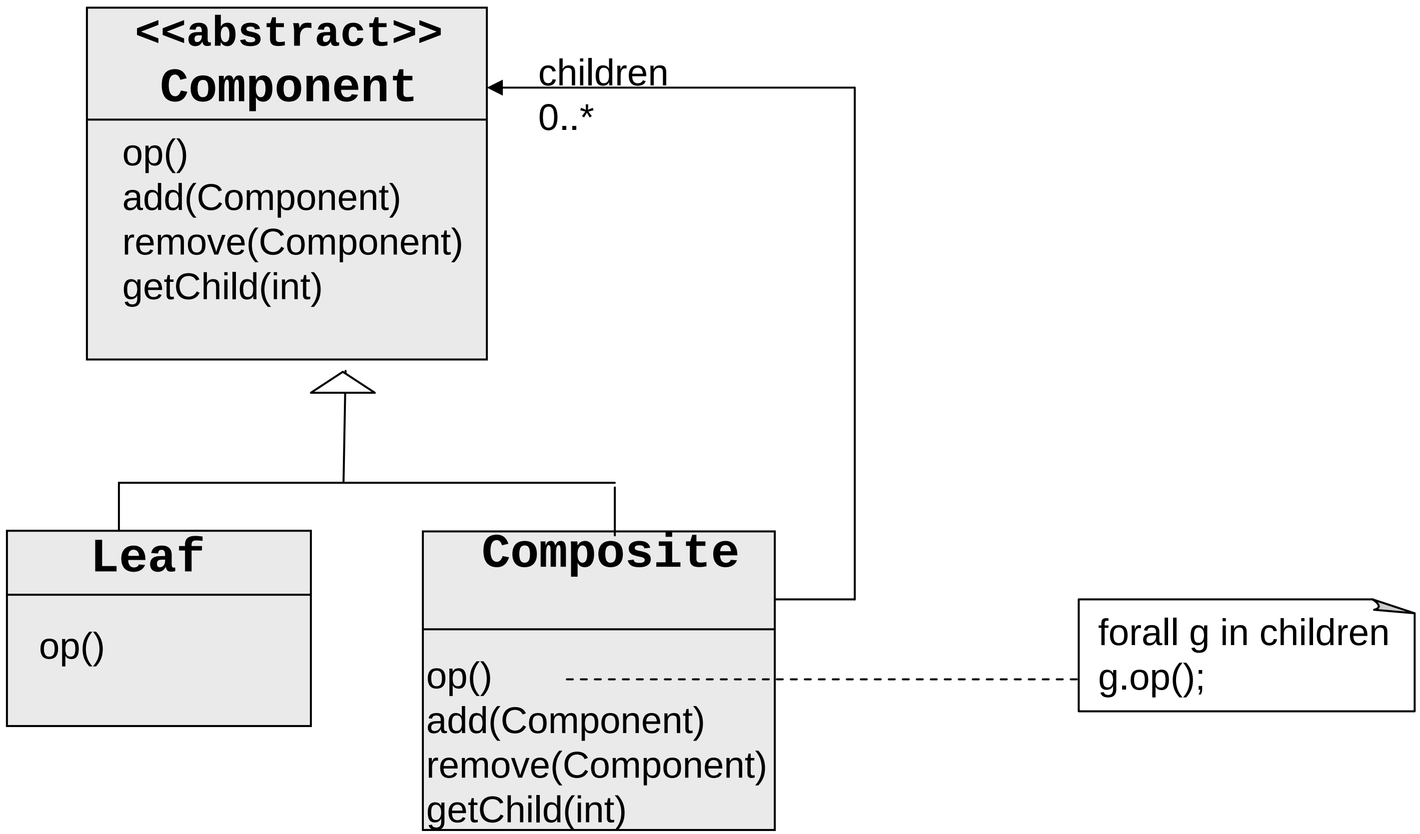
```
abstract boolean istBlatt();
boolean istKnoten()
{return !istBlatt(); }
```

- Sobald eine Methode abstrakt ist, muss die ganze Klasse als abstrakt erklärt werden.
- Der Compiler achtet darauf, dass jede abstrakte Methode in jeder Unterklasse implementiert wird.



Das Composite-Muster

- Das Composite-Muster dient zum Entwurf allgemeiner Baumstrukturen; es verallgemeinert das Muster für Binärbaume.



Das Composite-Muster

Das Composite-Muster wird verwendet zur Implementierung von Objekthierarchien wie etwa hierarchischen Benutzeroberflächen, verschachtelten Diagrammen oder Parsebäumen in Übersetzern.

- **Component**

  Bildet die Schnittstelle und implementiert das Standardverhalten für alle Klassen des Musters.

- **Leaf**
  - repräsentiert die Blattobjekte. Ein Blatt hat keine Kinder.
  - Definiert das Verhalten der primitiven Objekte.
- **Composite**
  - Definiert das Verhalten für Komponenten mit Kindern
  - Speichert die Kindkomponenten

Zusammenfassung

- Listen werden in Java als einfach oder doppelt verkettete oder auch als zirkuläre und Ringlisten realisiert.

- Binäre Bäume werden in Java implementiert:
  - als Verallgemeinerung der einfach verketteten Listen mit zwei Nachfolgerverweisen oder
  - durch eine abstrakte Oberklasse und zwei Unterklassen, einer Blatt- und eine Knotenklasse
- Eine Operation auf binären Bäume mit Knoten wird definiert:
  - durch Delegation der Operation an die Knotenklasse oder
- Eine Operation auf beblätterten binären Bäume werden definiert:

  durch Definition der Operation in beiden Unterklassen.

Zusammenfassung

- Das Composite-Muster dient zur Beschreibung hierarchischer Objektstrukturen; es verallgemeinert die zweite Implementierung binärer Bäume auf Bäume mit endlich vielen Kindbäumen.
- Suchbäume bieten effiziente Implementierungen für Mengen. Einfügen. Löschen und suchen besitzt lineare Komplexität in Höhe des Suchbaums.