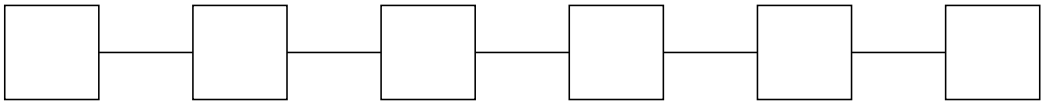
Listen und Bäume

D. Pattinson, LMU München

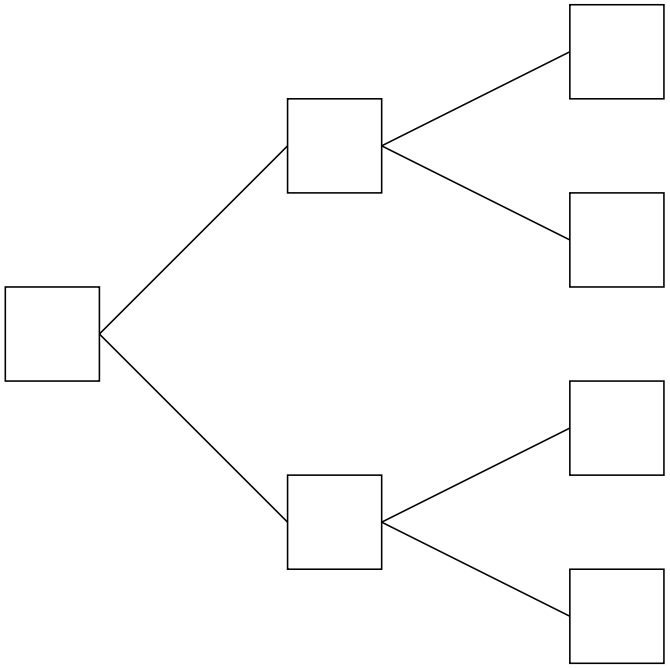
Listen und Bäume

Abstrakt: **“Bäume sind Listen mit mehr als einem Nachfolger”**

Listen



Bäume



Operationen auf Listen und Bäumen

Operationen auf Listen:

Erzeugen von Listen

```
public LinkedObjectList()
public void addFirst(Object o)
```

Traversieren von Listen

```
public boolean isEmpty()
public Object removeFirst()
```

Operationen auf Bäumen:

Erzeugen von Bäumen:

```
public LinkedBinTree()
public void insert(int id, Object o)
```

Traversieren von Bäumen:

```
public boolean isEmpty()
public LinkedBinTree getLeft()
public LinkedBinTree getRight()
```

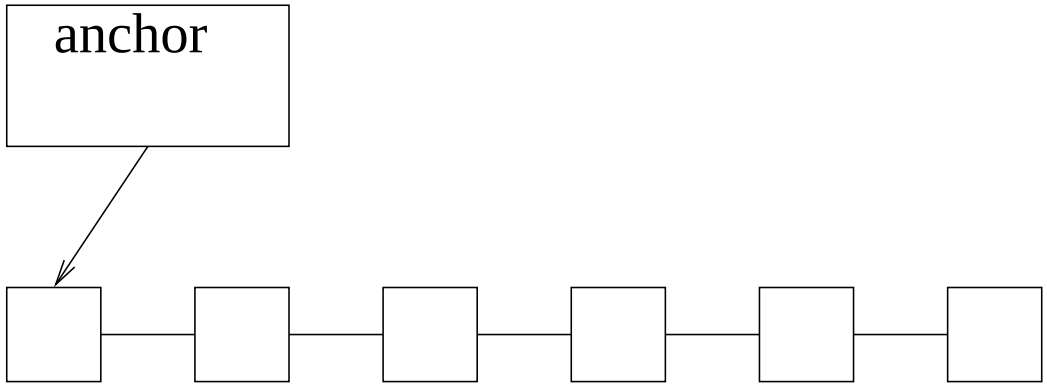
NB: Traversieren von Listen auch per *Iterator* möglich.

Traversieren per Iterator

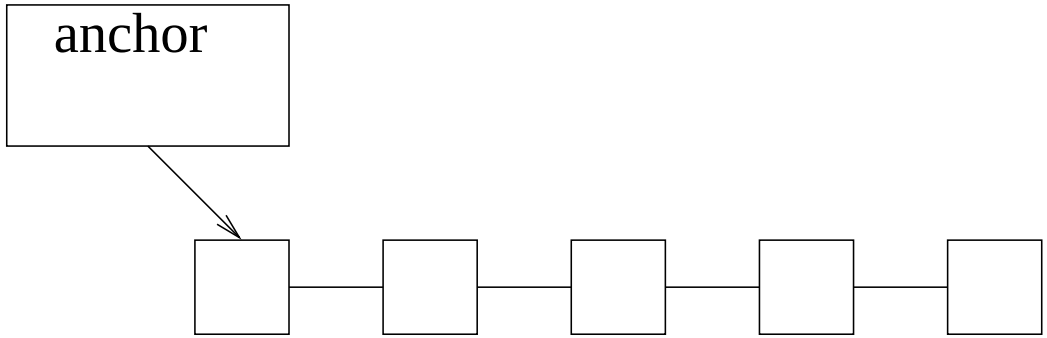
```
/* l hat Typ AbstractObjectList */

  Iterator iter = l.iterator();
  while (iter.hasNext())
  { Object curr = iter.next();

    /* ... */
    /* hier wird mit curr gearbeitet */
    /* ... */

  }
```

Der Variablen `curr` werden sukzessive alle Datenelemente der Liste zugewiesen.

Traversieren per Hand



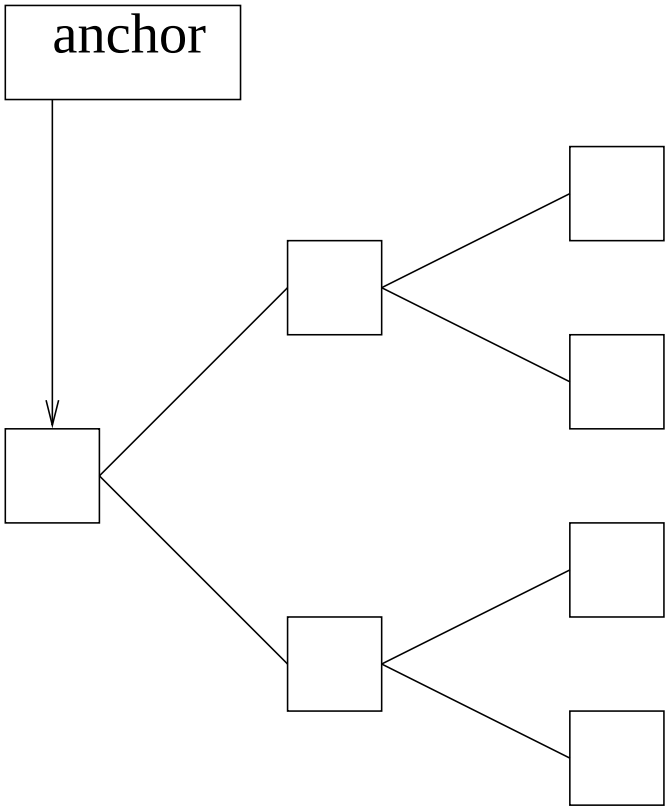
Nach `removeFirst()`:



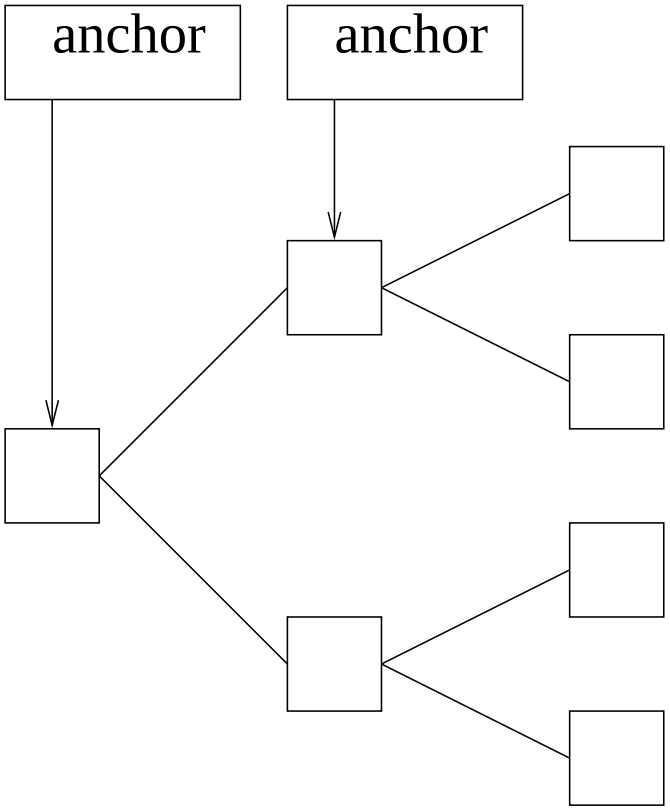
Man nennt `removeFirst` auch *destruktiv*.

Traversieren von Bäumen

Vor `getLeft`:



Nach `getLeft`:



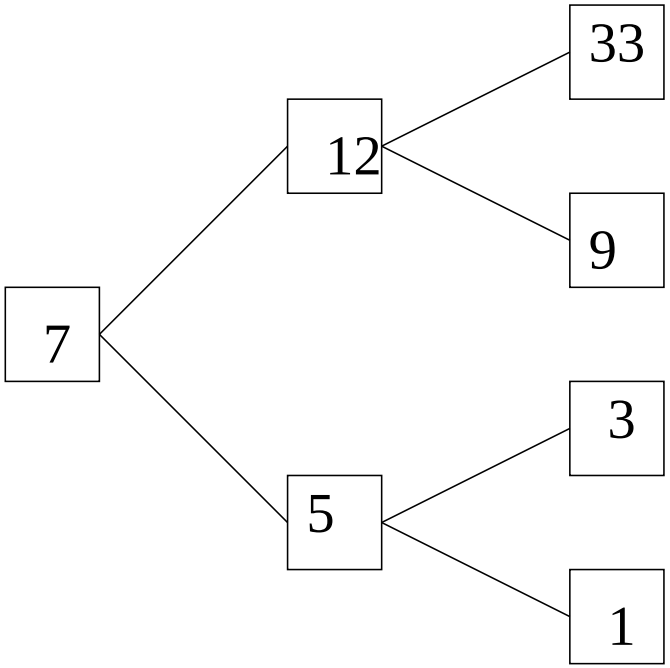
Die Funktion `getLeft` ist *nicht* destruktiv.

Geordnete Bäume, oder: Einfügen, aber wo?

Invariante: alle linken Knoten $\leq$ allen rechten Knoten.

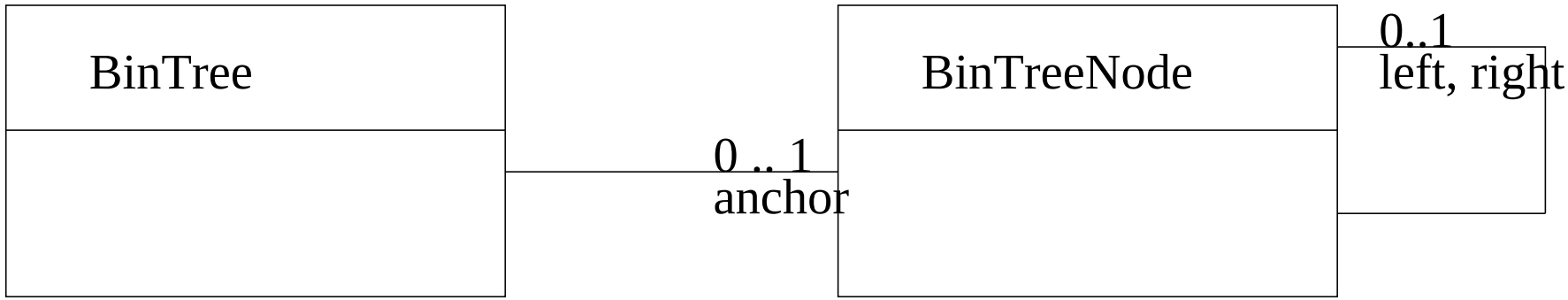
Beachte:

```
public void insert(int id, Object o)
```



NB: Bäume der Vorlesung sind immer sortiert.

Remainder: Wie Funktionieren Bäume?



Warum geordnete Bäume?

Ordnung erlaubt schnelles Suchen:

```
Object find(int key)
{ LinkedBinTreeNode cur = this;
  while(cur.key != key)
  { if(key < cur.key)
    cur = cur.left;
    else cur = cur.right;
    if(cur == null) return null;
  }
  return cur.value;
}
```

Komplexität: Höhe des Baumes
($\approx$ Anzahl der Schleifendurchläufe)

```
Object findrec (int key)
{ if (key == this.key)
  return value;
  if (key < this.key && left != null)
  return left.findrec(key);
  if (right != null)
  return right.findrec(key);
  return null;
}
```

Komplexität: Höhe des Baumes
($\approx$ Anzahl der rek. Aufrufe)

In beiden Fällen: $\mathcal{O}(n)$ wobei n = Höhe des Baumes.

Für “balancierte” Bäume also $\mathcal{O}(\log n)$ mit n = Anzahl der Knoten.

Warum so Kompliziert?

Listen in SML:

```
datatype 'a list = Empty | Node of 'a * 'a list;
```

“Eine Liste *ist* leer oder *ist ein* Element, zusammen mit einer Liste.”

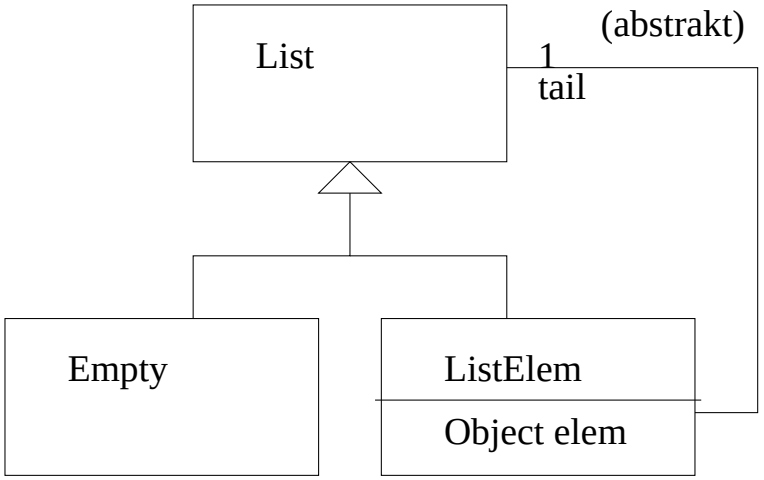
Bäume in SML:

```
datatype 'a BT = Empty | Node of 'a BT * 'a * 'a BT;
```

“Ein Baum *ist* leer oder *ist ein* Element, zusammen mit zwei Bäumen.”.

Die “*ist ein*” - Beziehung: $\rightsquigarrow$ Vererbung.

Listen als Composites



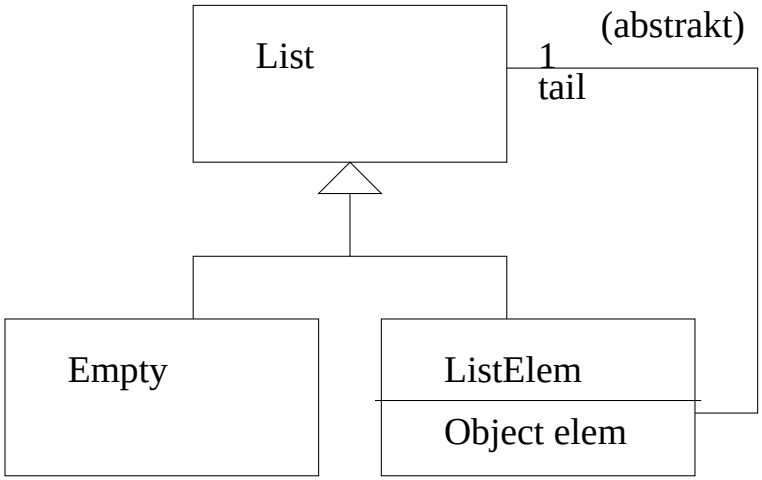
Erzeugen von Listen in SML:

```
l = [];
l = a::s;
```

Erzeugen von Listen in Java:

```
l = new Empty()
l = new ListElem (a, l)
```

Funktionen auf Listen



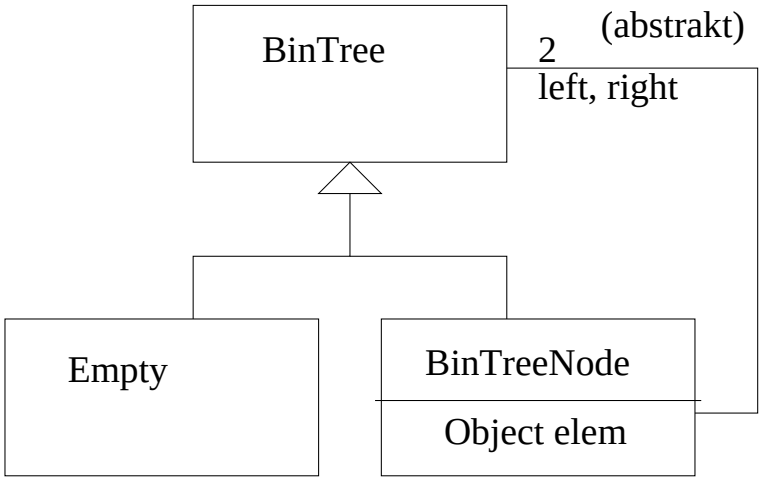
Länge in SML:

```
fun length([]) = 0
 |  length(a::l) = 1+(length l);
```

Länge in Java:

```
/* In Empty */
public int length() { return 0; }

/* in ListElem */
public int length()
{ return 1 + tail.length(); }
```

Bäume als Composites



Erzeugen von Bäumen in SML:

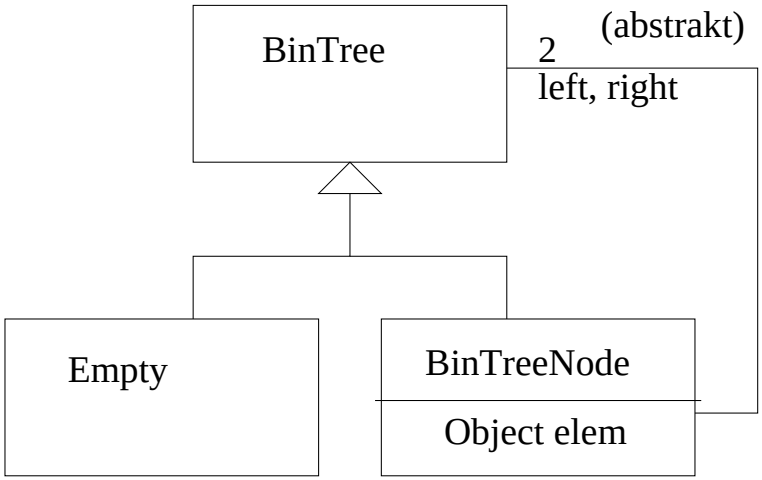
```
t = Empty
t = Node (t1, x, t2)
```

Erzeugen von Bäumen in Java:

```
t = new Empty()
t = new BinTreeNode(t1, x, t2)
```

NB: Bäume nicht notwendigerweise sortiert bzw. mit Schlüssel.

Funktionen auf Bäumen



Knoten in SML

```
fun nodes (Empty) = 0
 |  nodes (Node (t1, x, t2)) =
      1+(nodes t1)+(nodes t2);
```

Knoten in Java:

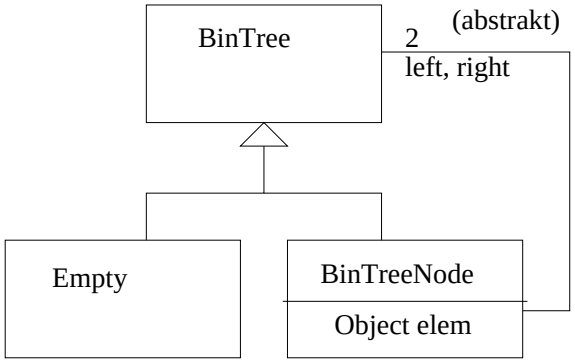
```
/* in Empty */
int nodes()  { return 0; }

/* in BinTreeNode */
int nodes ()
{ return 1+left.nodes()+right.nodes();
}
```

Beispiel 2: Suchen mit Prädikat

In SML:

```
datatype 'a BinTree = Empty | Node of 'a BinTree * 'a * 'a BinTree;

fun contains p Empty = false
 |  contains p (Node (left, elt, right)) =
     p(elt) orelse contains p left orelse contains p right;

val contains = fn : ('a -> bool) -> 'a BinTree -> bool
```

Suchen mit Prädikiat in Java




```
public interface Predicate {
        boolean isTrue(Object s);
}

/* In Empty */
public boolean contains (Predicate p)
{ return false; }

/* In BinTreeNode */
public boolean contains (Predicate P)
{ return p.isTrue(elem) || left.contains(p) || right.contains(p); }
```