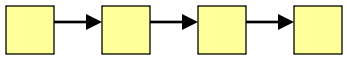
12. Dynamische Datenstrukturen

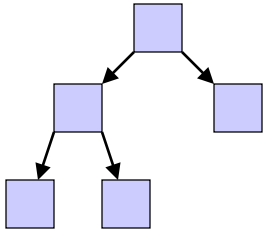
Warum "dynamisch"

- Elemente werden zur Laufzeit (dynamisch) mit *new* angelegt
- Datenstruktur kann dynamisch wachsen und schrumpfen

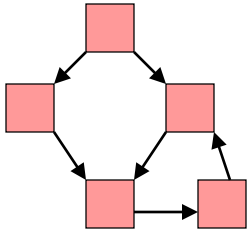
Wichtigste dynamische Datenstrukturen



Liste



Baum



Graph

Bestehen aus "Knoten", die über "Kanten" miteinander verbunden sind.

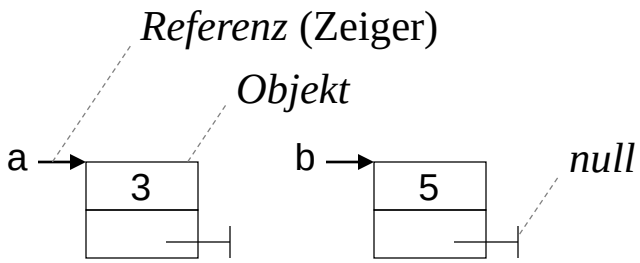
Knoten ... Objekte
Kanten ... Zeiger

Verknüpfen von Knoten

```
class Node {
    int       val;
    Node  next;
    Node(int v) {val = v;}
}
```

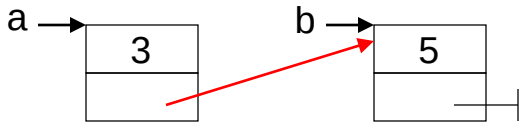
Erzeugen

```
Node a = new Node(3);
Node b = new Node(5);
```



Verknüpfen

```
a.next = b;
```

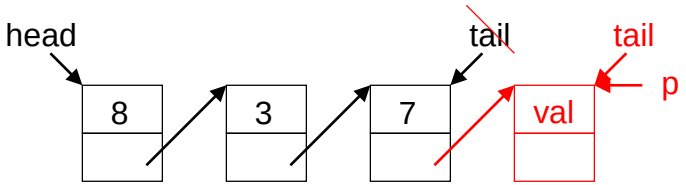


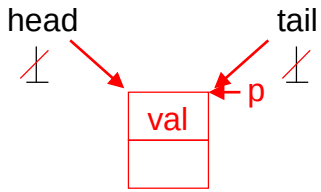
12. Dynamische Datenstrukturen

Unsortierte Listen

Einfügen am Listenende

```
class List {
    Node head = null;
    Node tail = null;
    ...
    void append (int val) {
        Node p = new Node(val);
        if (head == null) {
            head = p;
        } else {
            tail.next = p;
        }
        tail = p;
    }
}
```





Benutzung

```
List list = new List();
list.append(8);
list.append(3);
list.append(7);
...
```

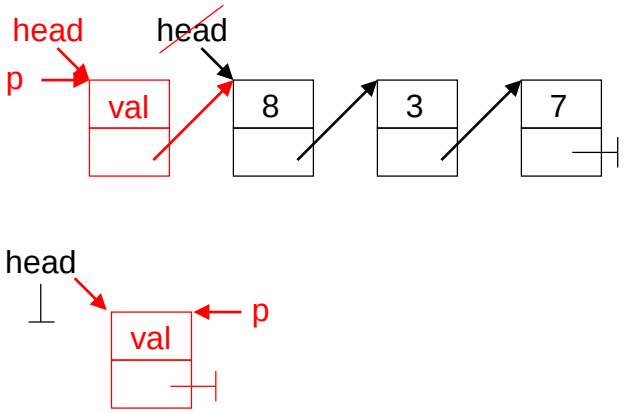
Unterschied zu einem Array

\+ Liste kann beliebig lang werden
\- Kein direkter Zugriff über Index möglich

Unsortierte Listen (Einfügen am Anfang)

Einfacher ist es, am Listenanfang einzufügen

```
class List {
    Node head = null;
    ...
    void prepend (int val) {
        Node p = new Node(val);
        p.next = head;
        head = p;
    }
}
```



Benutzung

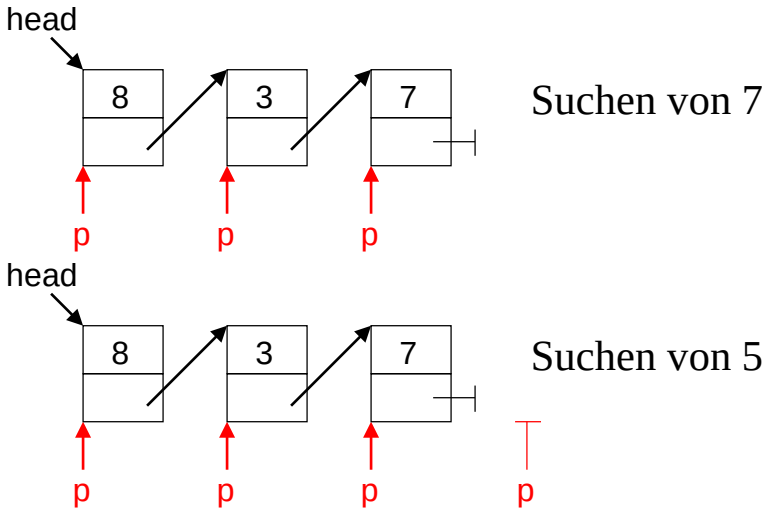
```
List list = new List();
list.prepend(5);
list.prepend(2);
...
```

- Man braucht kein *tail*
- Keine Sonderbehandlung bei leerer Liste

Unsortierte Listen (Suchen)

Suchen eines Werts

```
class List {
    Node head = null;
    ...
    boolean contains (int val) {  // Suchen
        Node p = head;
        while (p != null && p.val != val) p = p.next;
        // p == null || p.val == val
        return p != null;
    }
}
```



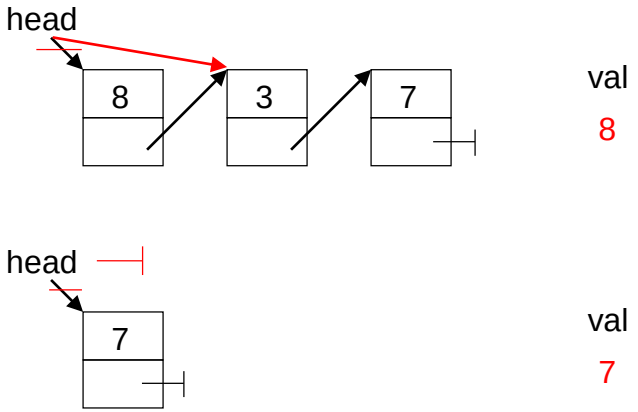
Benutzung

```
List list = new List();
list.append(8);
list.append(3);
list.append(7);
...
if (list.contains(7)) ...
```

Unsortierte Listen (Löschen am Anfang)

Entnehmen des ersten Elements

```
class List {
    Node head = null;
    ...
    int removeFirst () {
        if (head == null) {
            return -1;
        } else {  // head != null
            int val = head.val;
            head = head.next;
            return val;
    }
}
```



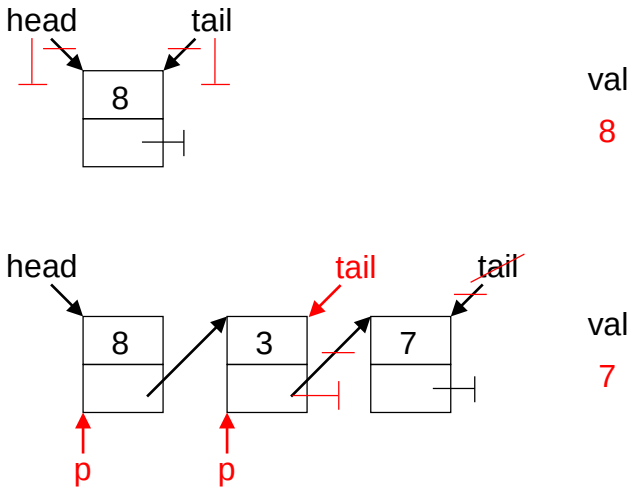
Benutzung

```
...
int val = list.removeFirst();
```

Unsortierte Listen (Löschen am Ende)

Entnehmen des letzten Elements

```
class List {
    Node head = null;
    Node tail = null;
    ...
    int removeLast () {
        if (head == null) {
            return -1;
        } else if (head == tail) { // just 1 elem
            int val = tail.val;
            head = tail = null;
            return val;
        } else { // more than 1 elem
            Node p = head;
            while (p.next != tail) p = p.next;
            // p.next == tail
            int val = tail.val;
            p.next = null;
            tail = p;
            return val;
        }
    }
}
```



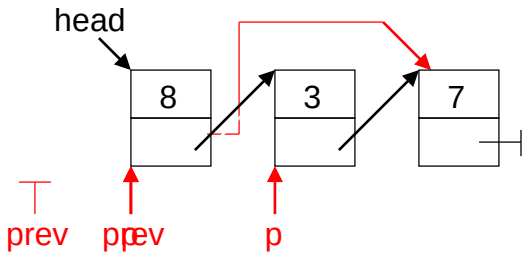
Benutzung

```
...
int val = list.removeLast();
```

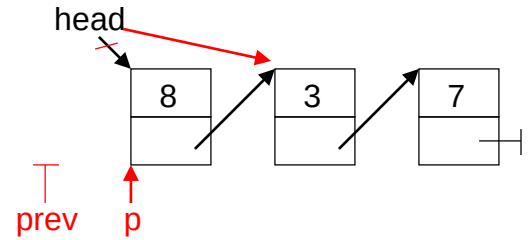
Unsortierte Listen (Löschen eines Werts)

Löschen eines Werts

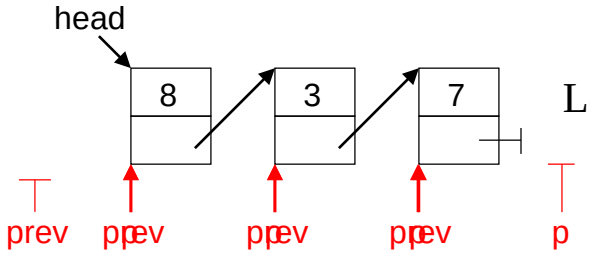
```
class List {
    Node head = null;
    Node tail = null;
    ...

    void delete (int val) {
        Node p = head, prev = null;
        while (p != null && p.val != val) {
            prev = p; p = p.next;
        }
        // p == null || p.val == val
        if (p != null) { // p.val == val
            if (p == head) {
                head = p.next
            } else {
                prev.next = p.next;
            }
            if (p == tail) tail = prev;
        }
    }
}
```



Löschen von 3



Löschen von 8



Löschen von 5

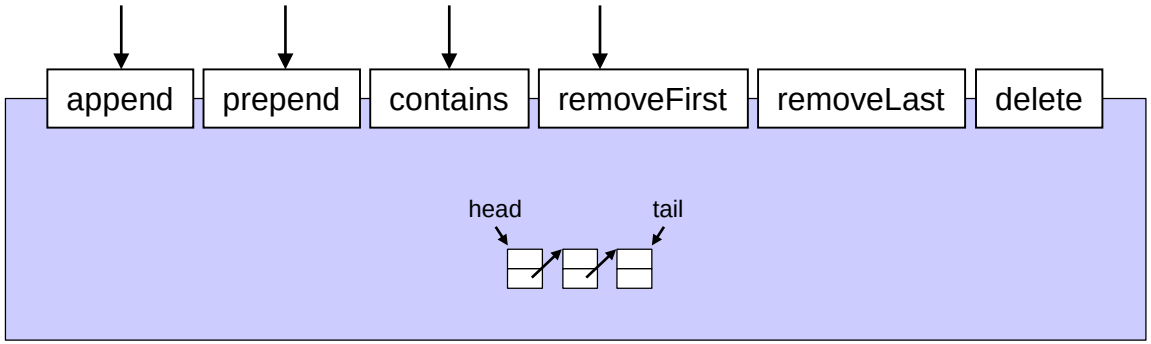
Benutzung

```
...
list.delete(3);
```

Im Gegensatz zu einem Array muss beim Löschen nichts verschoben werden!

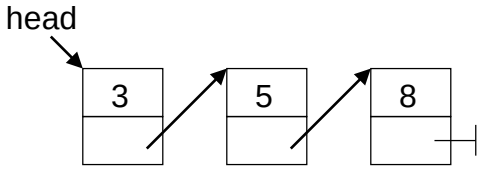
Unsortierte Listen (Zusammenfassung)

```
class List {
    Node head = null;
    Node tail = null;

    void append(int val) {...}
    void prepend(int val) {...}
    boolean contains(int val) {...}
    int removeFirst() {...}
    int removeLast() {...}
    void delete(int val) {...}
}
```



12. Dynamische Datenstrukturen

Sortierte Listen (Einfügen)

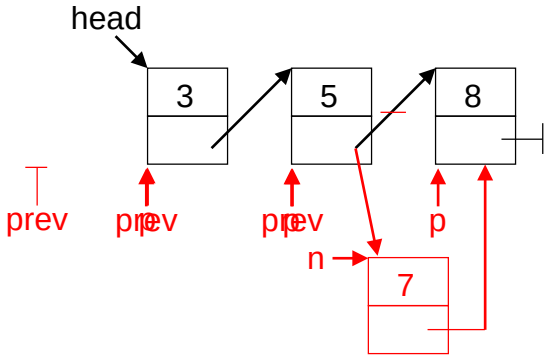


Elemente (z.B. aufsteigend) sortiert

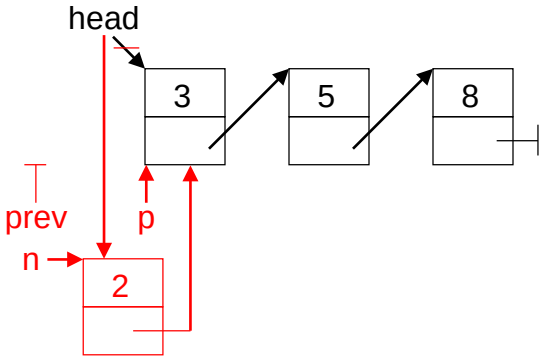
- Einfügen º Einsortieren
- Suchen im Mittel nur bis zur Hälfte der Liste

Einfügen (Einsortieren)

```
class SortedList {
   Node head = null;
   ...
   void insert (int val) {
      Node p = head, prev = null;
      while (p ! = null && p.val < val) {
         prev = p; p = p.next;
      }
      // p == null || p.val >= val
      Node n = new Node(val);
      n.next = p;
      if (prev == null) {
         head = n;
      } else {
         prev.next = n;
      }
   }
}
```



Einfügen von 7

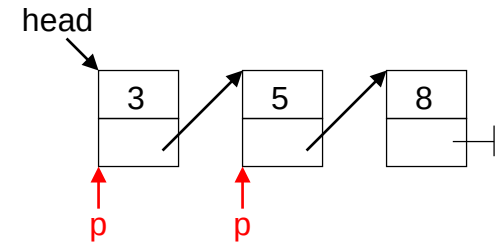


Einfügen von 2

Sortierte Listen (Suchen)

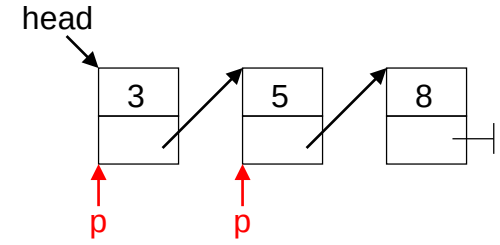
Suchen eines Werts

```
class SortedList {
    Node head = null;
    ...
    int indexOf (int val) {
        Node p = head;
        int pos = 0;
        while (p ! = null && p.val < val) {
            p = p.next; pos++;
        }
        // p == null || p.val >= val
        if (p != null && p.val == val) {
            return pos;
        } else { // p == null || p.val != val
            return -1;
        }
    }
}
```



Suchen von 5

=> return 1;



Suchen von 4

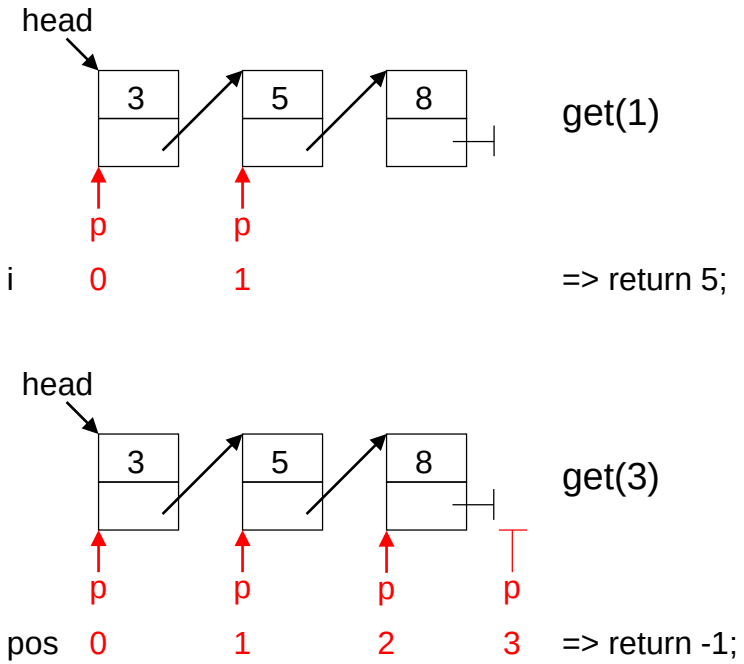
=> return -1;

Muss nicht bis ans Ende durchsucht werden

Sortierte Listen (Zugriff über Index)

Zugriff auf das i. Listenelement

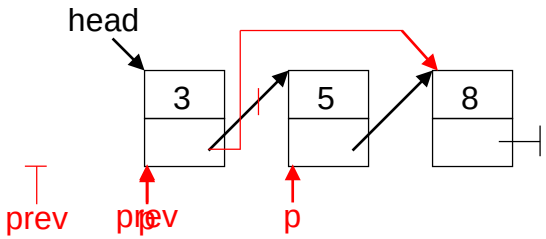
```
class SortedList {
    Node head = null;
    ...
    int get (int pos) {
        if (pos < 0) return -1;
        Node p = head;
        int i = 0;
        while (p != null && i < pos) {
            p = p.next; i++;
        }
        // p == null || i == pos
        if (p != null) { // i == pos
            return p.val;
        } else { // p == null
            return -1;
        }
    }
}
```



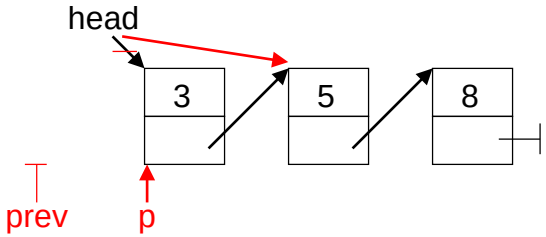
Sortierte Listen (Löschen)

Löschen eines Werts

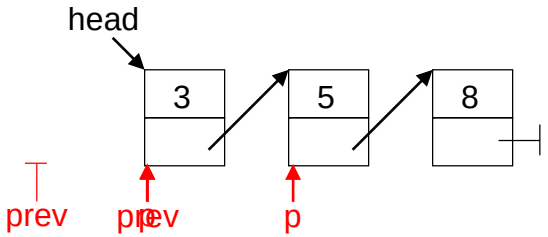
```
class SortedList {
    Node head = null;
    ...
    void delete (int val) {
        Node p = head, prev = null;
        while (p ! = null && p.val < val) {
            prev = p; p = p.next;
        }
        // p == null || p.val >= val
        if (p != null && p.val == val) {
            if (prev == null) {
                head = p.next;
            } else {
                prev.next = p.next;
            }
        }
    }
}
```



Löschen von 5



Löschen von 3



Löschen von 4

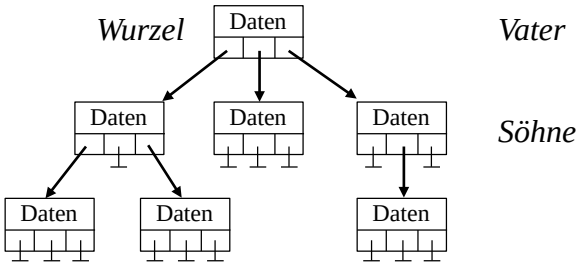
Muss nicht bis ans Ende durchsucht werden

12. Dynamische Datenstrukturen

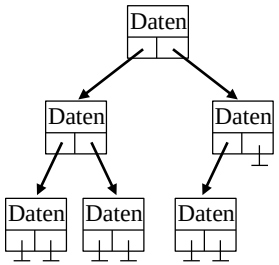
Bäume

Zur Darstellung von Hierarchien

- Abteilungen und Unterabteilungen
- Komponenten und Unterkomponenten
- Strukturen (z.B. von Programmen; Syntaxbaum)
- Datenstruktur zum schnellen Suchen



Mehrwegbaum



Binärbaum

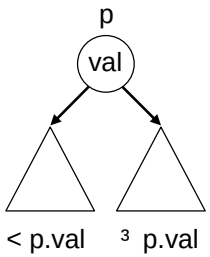
jeder Knoten hat max. 2 Söhne

Unterbäume sind wieder Bäume
(rekursive Datenstruktur)

```
class Node {
   int val; // data
   Node left, right;

   Node(int val) {
      this.val = val;
   }
}
```

Binäre Suchbäume

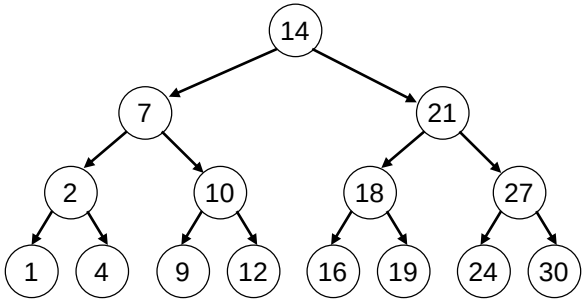
Binärbäume mit folgender Eigenschaft



Für jeden Knoten *p* gilt

- alle Werte im Baum *p.left* < *p.val*
- alle Werte im Baum *p.right* ³ *p.val*

Beispiel



Ermöglicht binäres Suchen

z.B.:

Suchen von 10: 14 nach links 7 nach rechts 10

Suchen von 16: 14 nach rechts 21 nach links 18 nach links 16

Suchen von 3: 14 nach links 7 nach links 2 nach rechts 4 nach links ✗

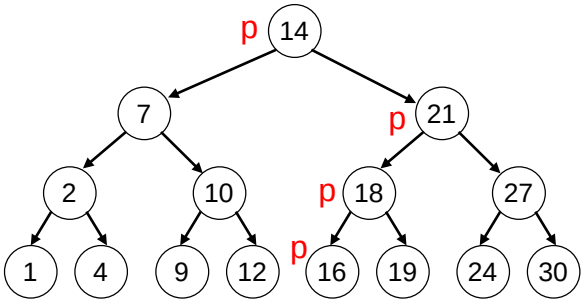
$O(\log_2 n)$

Binäres Suchen (iterativ)

```
class BinTree {
   Node root = null;

   Node search (int val) {
      Node p = root;
      while (p != null) {
         if (val == p.val) return p;
         else if (val < p.val) p = p.left;
         else /* val > p.val */ p = p.right;
      }
      return null;
   }
...
}
```

```
class Node {
   int val; // data
   Node left, right;

   Node(int val) {
      this.val = val;
   }
}
```

Beispiel Suchen von 16

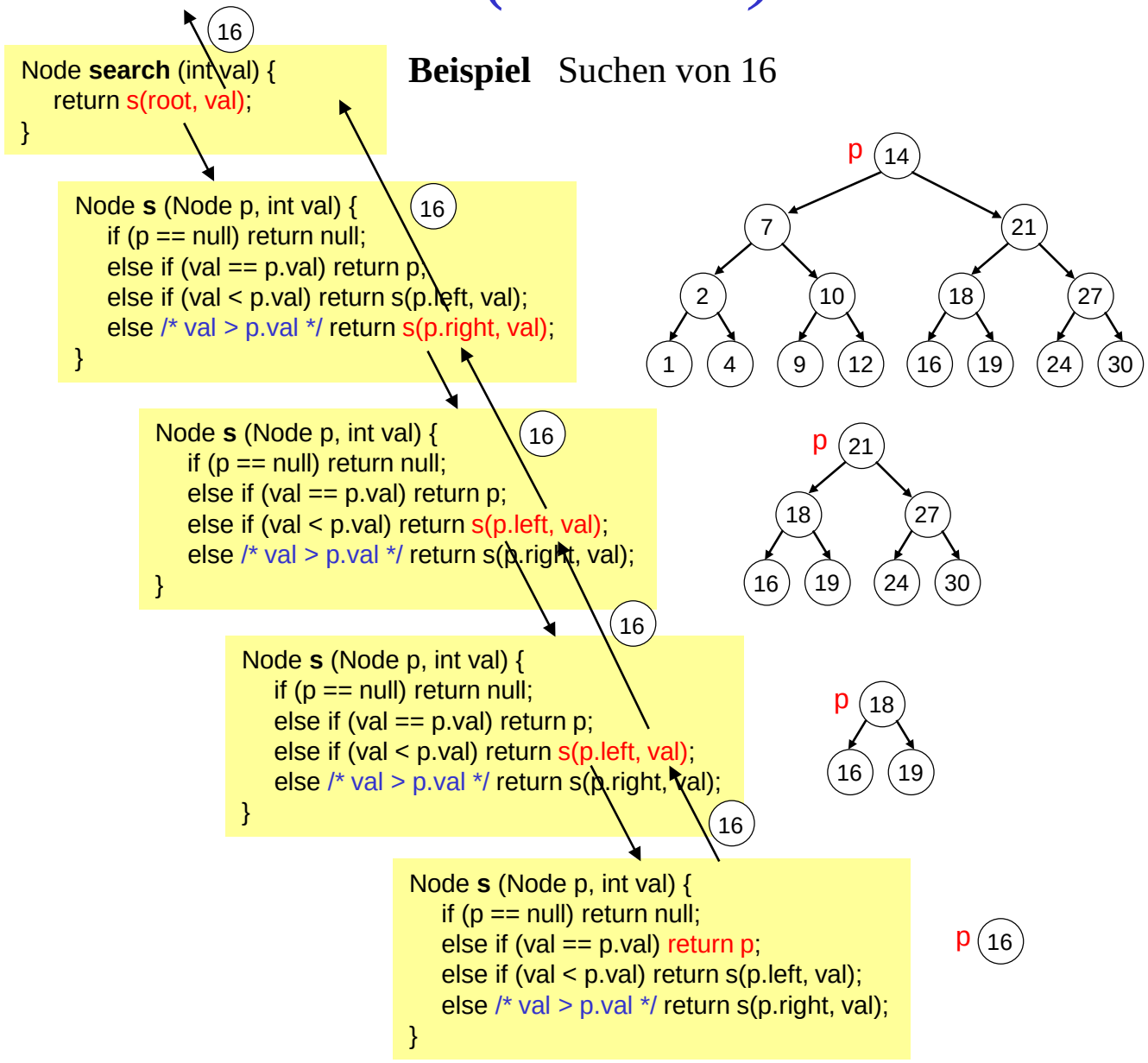


Binäres Suchen (rekursiv)

```
class BinTree {
   Node root = null;

   Node search (int val) {
      return s(root, val);
   }

   Node s (Node p, int val) {
      if (p == null) return null;
      else if (val == p.val) return p;
      else if (val < p.val) return s(p.left, val);
      else /* val > p.val */ return s(p.right, val);
   }
...
}
```

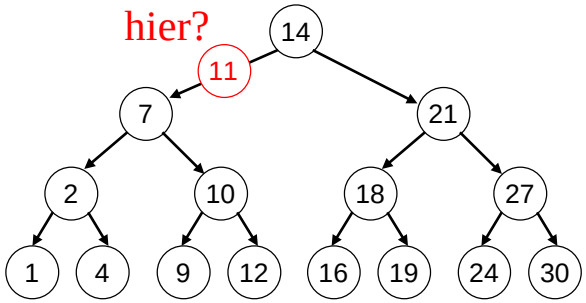
```
class Node {
   int val; // data
   Node left, right;

   Node(int val) {
      this.val = val;
   }
}
```

Binäres Suchen (rekursiv)



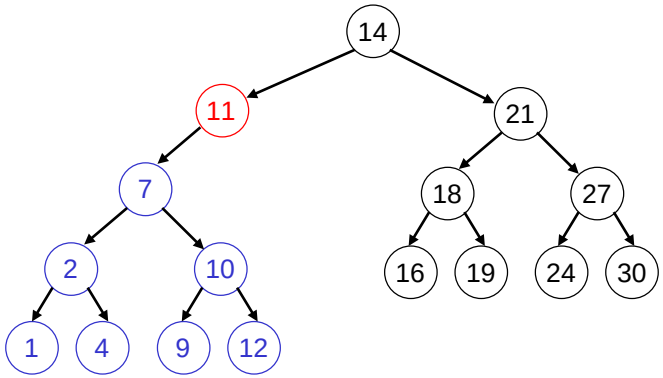
Einfügen in Binären Suchbäumen

Angenommen, wir wollen den Wert 11 in folgenden Baum einfügen

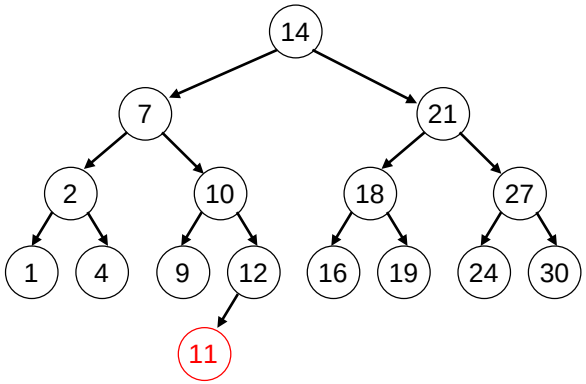


Wo sollen wir ihn einfügen?

Möglich - würde aber die Suchzeit von 7 anderen Knoten verlängern



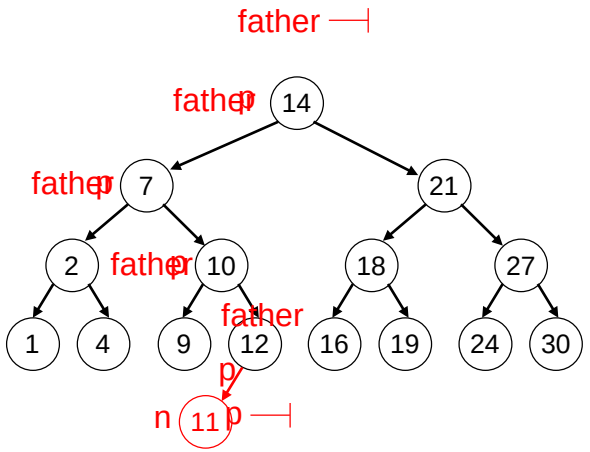
Daher wird ein neuer Knoten *immer als Blatt* eingefügt



Einfügen in Binären Suchbäumen

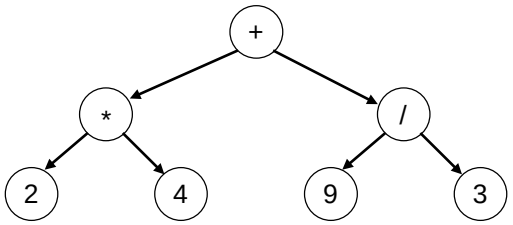
```
class BinTree {
   Node root = null;
   ...
   Node insert (int val) {
      Node p = root, father = null;
      while (p != null) {
         father = p;
         if (val < p.val) p = p.left; else p = p.right;
      }
      // p == null; father zeigt auf Blatt,
      // unter dem eingefügt werden muss
      Node n = new Node(val);
      if (root == null) root = n;
      else if (val < father.val) father.left = n;
      else father.right = n;

   }
...
}
```

Beispiel: Einfügen von 11



Durchlaufen von (Binär-) Bäumen

Syntaxbaum



Kein binärer Suchbaum
(funktioniert aber auch für binäre Suchbäume)

entspricht

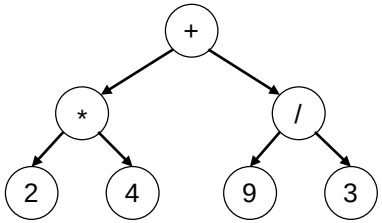
`2 * 4 + 9 / 3`

3 Arten des Durchlaufs (rekursiv)

Preorder	Wurzel, linker Teilbaum, rechter Teilbaum	`+ * 2 4 / 9 3`
Postorder	linker Teilbaum, rechter Teilbaum, Wurzel	`2 4 * 9 3 / +`
Inorder	linker Teilbaum, Wurzel, rechter Teilbaum	`2 * 4 + 9 / 3`

Durchlaufen von (Binär-) Bäumen

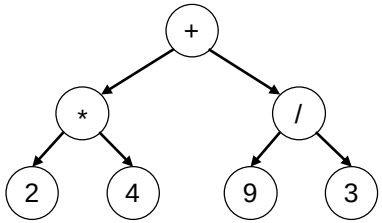
```
class BinTree {
   Node root = null;
   ...

   Node preOrder (Node p) {
      ... process p.val ...
      if (p.left != null) preOrder(p.left);
      if (p.right != null) preOrder(p.right);
   }

...
}
```



+ * 2 4 / 9 3

```
class Node {
   String val; // data
   Node left, right;

   Node(int val) {
      this.val = val;
   }
}
```

Durchlaufen von (Binär-) Bäumen




```
Node preOrder (Node p) {
  ... process p.val ...
  if (p.left != null) preOrder(p.left);
  if (p.right != null) preOrder(p.right);
}
```

+

+ * 2 4 / 9 3

```
Node preOrder (Node p) {
  ... process p.val ...
  if (p.left != null) preOrder(p.left);
  if (p.right != null) preOrder(p.right);
}
```

*

```
Node preOrder (Node p) {
  ... process p.val ...
  if (p.left != null) preOrder(p.left);
  if (p.right != null) preOrder(p.right);
}
```

/

```
Node preOrder (Node p) {
  ... process p.val ...
  if (p.left != null) preOrder(p.left);
  if (p.right != null) preOrder(p.right);
}
```

2

```
Node preOrder (Node p) {
  ... process p.val ...
  if (p.left != null) preOrder(p.left);
  if (p.right != null) preOrder(p.right);
}
```

4

```
Node preOrder (Node p) {
  ... process p.val ...
  if (p.left != null) preOrder(p.left);
  if (p.right != null) preOrder(p.right);
}
```

9

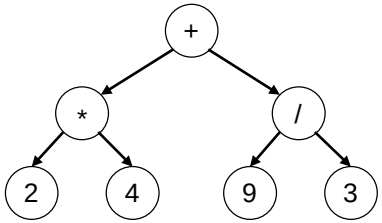
```
Node preOrder (Node p) {
  ... process p.val ...
  if (p.left != null) preOrder(p.left);
  if (p.right != null) preOrder(p.right);
}
```

3

```
Node preOrder (Node p) {
  ... process p.val ...
  if (p.left != null) preOrder(p.left);
  if (p.right != null) preOrder(p.right);
}
```

Durchlaufen von (Binär-) Bäumen

```
class BinTree {
   Node root = null;
   ...

   Node preOrder (Node p) {
      ... process p.val ...
      if (p.left != null) preOrder(p.left);
      if (p.right != null) preOrder(p.right);
   }

   Node postOrder (Node p) {
      if (p.left != null) postOrder(p.left);
      if (p.right != null) postOrder(p.right);
      ... process p.val ...
   }

   Node inOrder (Node p) {
      if (p.left != null) inOrder(p.left);
      ... process p.val ...
      if (p.right != null) inOrder(p.right);
   }
...
}
```



+ * 2 4 / 9 3

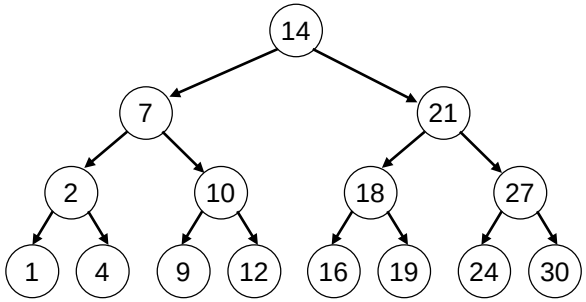
2 4 * 9 3 / +

2 * 4 + 9 / 3

```
class Node {
   String val; // data
   Node left, right;

   Node(int val) {
      this.val = val;
   }
}
```

Durchlaufen von (Binär-) Bäumen

Inorder-Durchlauf eines binären Suchbaums ergibt Werte in sortierter Reihenfolge




```
Node preOrder (Node p) {
    ... process p.val ...
    if (p.left != null) preOrder(p.left);
    if (p.right != null) preOrder(p.right);
}
```

1 2 4 7 9 10 12 14 16 18 19 21 24 27 30