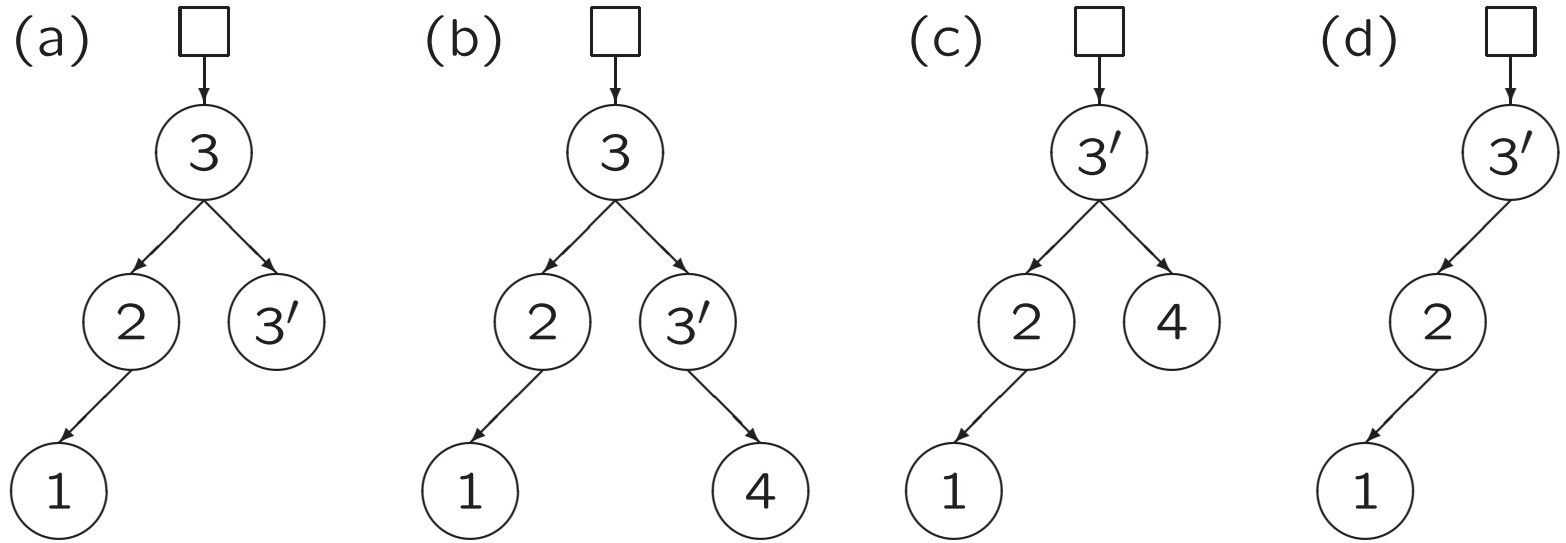
Binärer Baum



Kopf des binären Baums

```
public class IntTree {
    private IntTreeNode root = null;

    public void add (int e) {
        if (root == null) { root = new IntTreeNode(e); }
        else { root.add(e); }
    }
    public boolean contains (int elem) {
        return root != null && root.contains(elem);
    }
    public void remove (int elem) {
        if (root != null) { root = root.remove(elem); }
    }
}
```

Knoten des binären Baums

```
class IntTreeNode {
    private int elem;
    private IntTreeNode left = null;
    private IntTreeNode right = null;

    IntTreeNode (int e) {
        elem = e;
    }
    void add (int e)            { /* füge e in Baum ein */ }
    boolean contains (int e)    { /* suche e im Baum    */ }
    IntTreeNode remove (int e) { /* lösche e aus Baum  */ }
}
```

Rekursives Einfügen in binären Baum

```
void add (int e) {
    if (e < elem) {
        if (left != null) {
            left.add(e);
        } else {
            left = new IntTreeNode(e);
        }
    } else {  // e >= elem
        if (right != null) {
            right.add(e);
        } else {
            right = new IntTreeNode(e);
        }
    }
}
```

Rekursive Suche in binärem Baum

```
boolean contains (int e) {
    if (e < elem) {
        return left != null && left.contains(e);
    } else if (e == elem) {
        return true;
    } else {  // e > elem
        return right != null && right.contains(e);
    }
}
```

Iterative Suche in binärem Baum

```
boolean contains (int e) {
    IntTreeNode node = this;
    do {
        if (e < node.elem) {
            node = node.left;
        } else if (e == node.elem) {
            return true;
        } else {
            node = node.right;
        }
    } while (node != null);
    return false;
}
```

Rekursives Löschen aus binärem Baum

```
IntTreeNode remove (int e) {
    if (e < elem) {
        if (left != null) { left = left.remove(e); }
    } else if (e == elem) {
        if (right == null) { return left; }
        if (left != null) { right.addTree(left); }
        return right;
    } else {
        if (right != null) { right = right.remove(e); }
    }
    return this;
}
```

Verschieben eines Teilbaums

```
private void addTree (IntTreeNode t) {
    if (left != null) {
        left.addTree(t);
    } else {
        left = t;
    }
}
```

Algorithmische Kosten

Kostenabschätzungen

- grobe Abschätzung von Laufzeit und Speicherverbrauch
- konstante Faktoren vernachlässigt
  $\Rightarrow$ geschätzte Kosten unabhängig von Hardware, Compiler
- Anzahl der Elemente in Datenstruktur berücksichtigt
- Abschätzung maximaler und durchschnittlicher Kosten
- Kosten typisch für Algorithmen bzw. Datenstrukturen

Beispiele für Kosten (Laufzeit)

Operation	Schnitt	Max.
Einfügen in Liste:	$O(1)$	$O(1)$
Suche in Liste:	$O(n)$	$O(n)$
Löschen aus Liste:	$O(n)$	$O(n)$
Einfügen in Baum:	$O(\log(n))$	$O(n)$
Suche in Baum:	$O(\log(n))$	$O(n)$
Löschen aus Baum:	$O(\log(n))$	$O(n)$

$O(1)$ = konstant

$O(\log(n))$ = logarithmisch

$O(n)$ = linear

$O(n^2)$ = quadratisch

$O(n^k)$ = polynomial

$O(2^n)$ = exponentiell

Beispiele für Kosten (Speicher)

- Kosten für Liste und Baum: $O(n)$
- kosten für Operationen auf Liste und Baum:
  - iterativ: $O(1)$
  - rekursiv: gleich den Kosten für Laufzeit
  - Relevanz gering (höhere Kosten entscheidend)
- Ausgewogenheit zwischen Laufzeit und Speicher wichtig

Beispiel: Einlesen und sortiert ausgeben

Verkettete Liste: (Sortieren notwendig)

- Einlesen und Einfügen: $n \cdot O(1) = O(n)$
- Sortieren: $O(n^2)$ für Bubble Sort
- Ausgeben: $O(n)$
- insgesamt: $\max(O(n), O(n^2), O(n)) = O(n^2)$

Binärer Baum: (schon durch Einfügen sortiert)

- Einlesen und Einfügen: $n \cdot O(\log(n)) = O(n \cdot \log(n))$
- Ausgeben: $O(n)$
- insgesamt: $\max(O(n \cdot \log(n)), O(n)) = O(n \cdot \log(n))$

Bubble Sort in IntListNode: $O(n^2)$

```
void sort() {
    boolean changed;
    do { changed = false;
         IntListNode s = this, l = next;
         while (l != null) {
             if (l.elem < s.elem) {
                 int i = s.elem;
                 s.elem = l.elem;
                 l.elem = i;
                 changed = true;
             }
             s = l; l = l.next;
         }
    } while (changed);
}
```

Ausgabe in IntTreeNode: $O(n)$

```
void print() {
    if (left != null) {
        left.print();
    }
    System.out.println(elem);
    if (right != null) {
        right.print();
    }
}
```