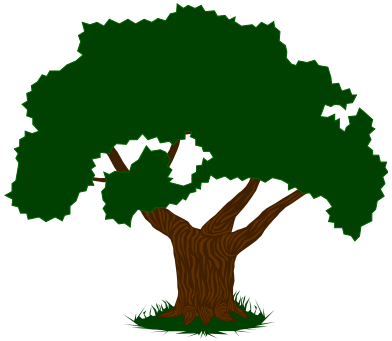
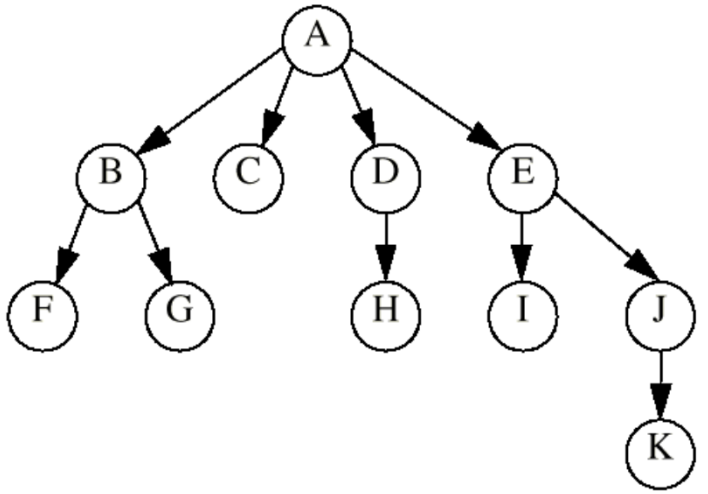
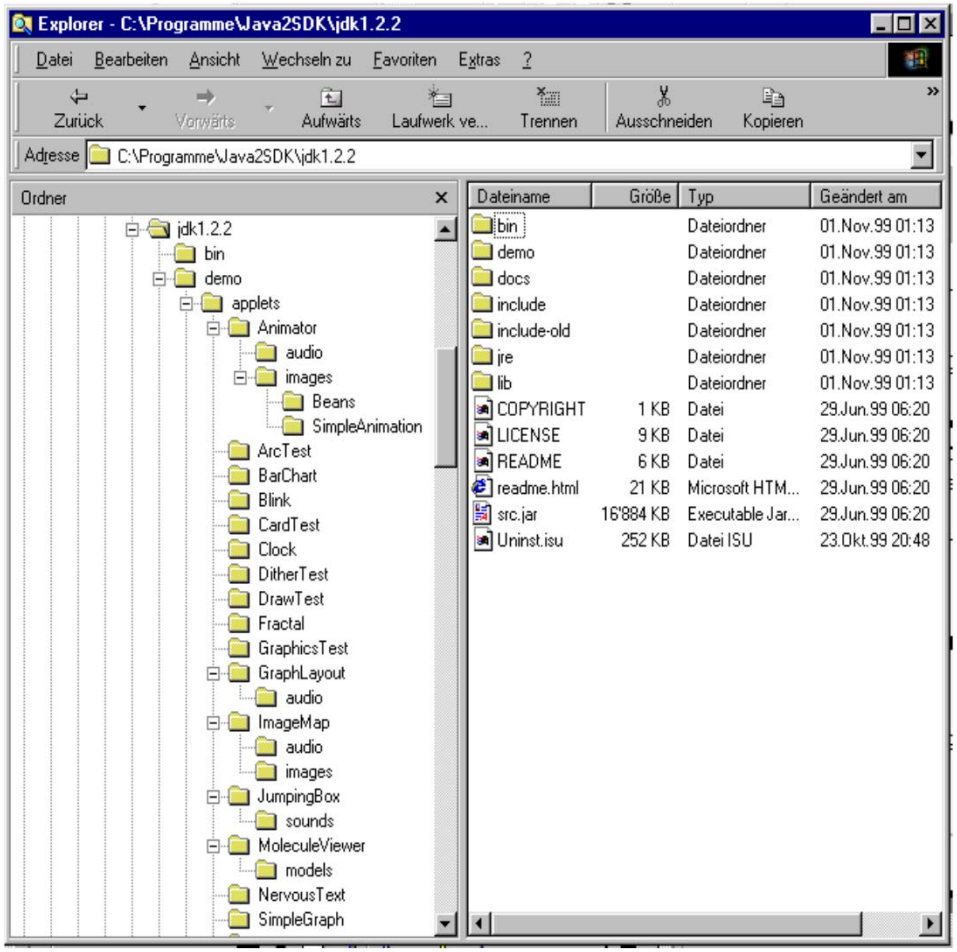
Bäume





- Sie wissen, was Bäume (in der Informatik) sind
- Sie kennen das Visitor Pattern
- Sie kennen Binärbäume
- Sie können die Bäume auf unterschiedliche Arten traversieren
- Sie wissen, wie man in Binärbäumen Elemente einfügt und löscht

Bedeutung & Anwendung

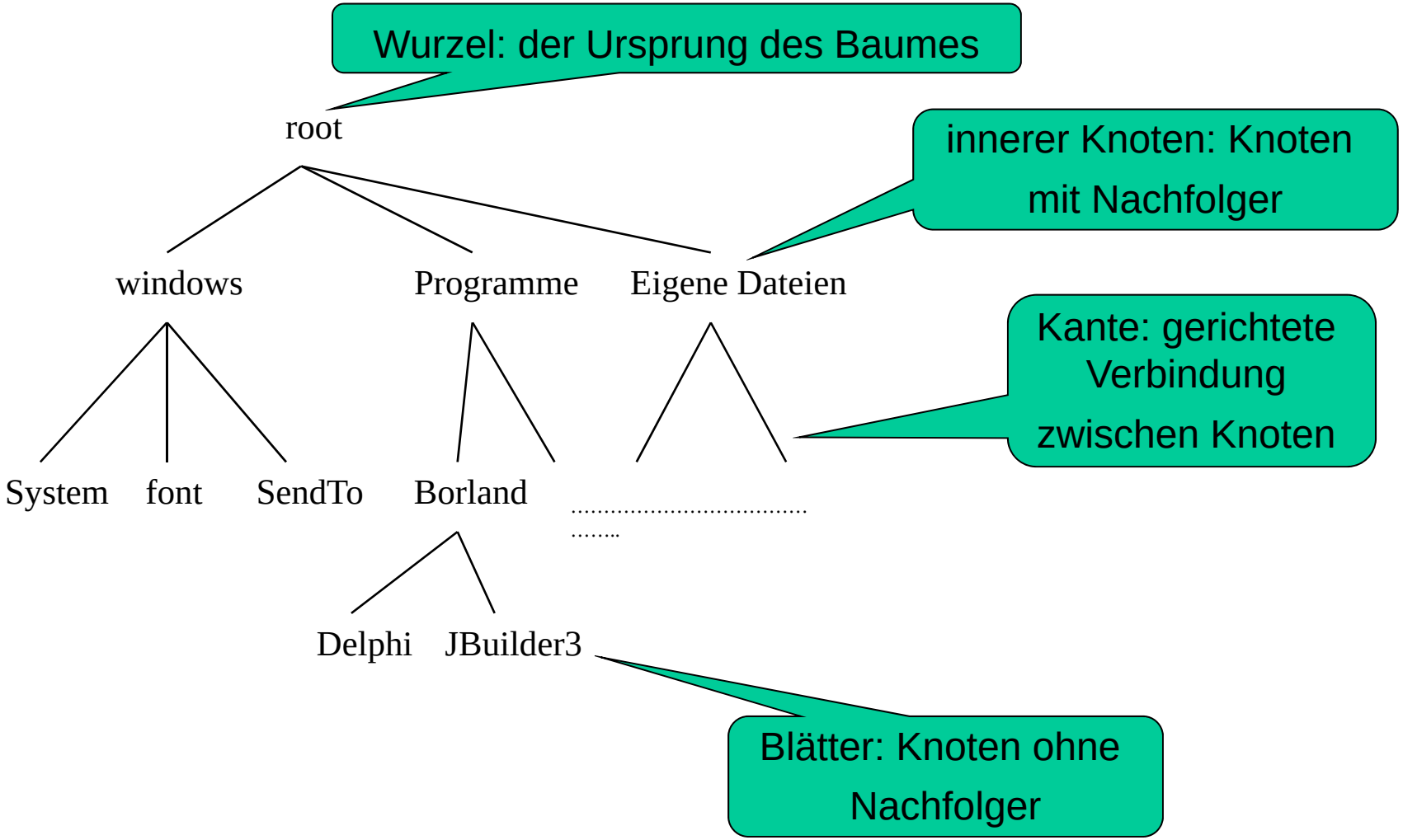


Der Baum ist eine **fundamentale Datenstruktur in der Informatik**:

z.B. Betriebssysteme:

- Dateien werden in Bäumen oder baum-ähnlichen Strukturen abgelegt:
  - es werden Verzeichnisse (Ordner) als die innere **Knoten** abgelegt,
  - alle anderen Dateien in einem **Blatt** abgelegt werden.
  - Das Ursprungsverzeichnis wird auch als **Wurzel** oder **Root** bezeichnet.
- Einer **Kante** im Baum zu folgen entspricht dem Hinabsteigen zu einem Unterverzeichnis.

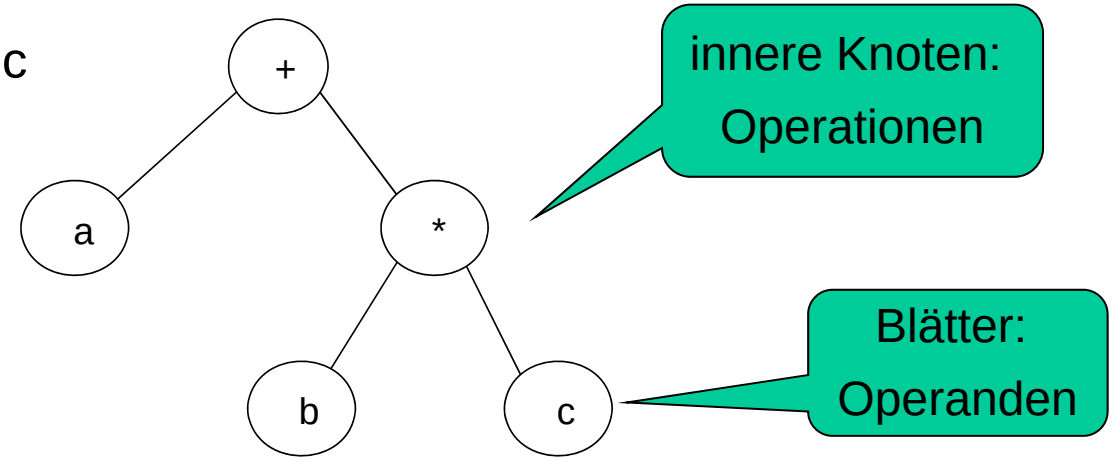
Beispiel 1: Dateisystem



Beispiel 2: Ausdruck-Baum

- Der Ausdruck-Baum (expression tree) wird eingesetzt um arithmetische Ausdrücke auszuwerten: der Ausdruck wird zuerst in einen Baum umgeformt und dann ausgewertet.

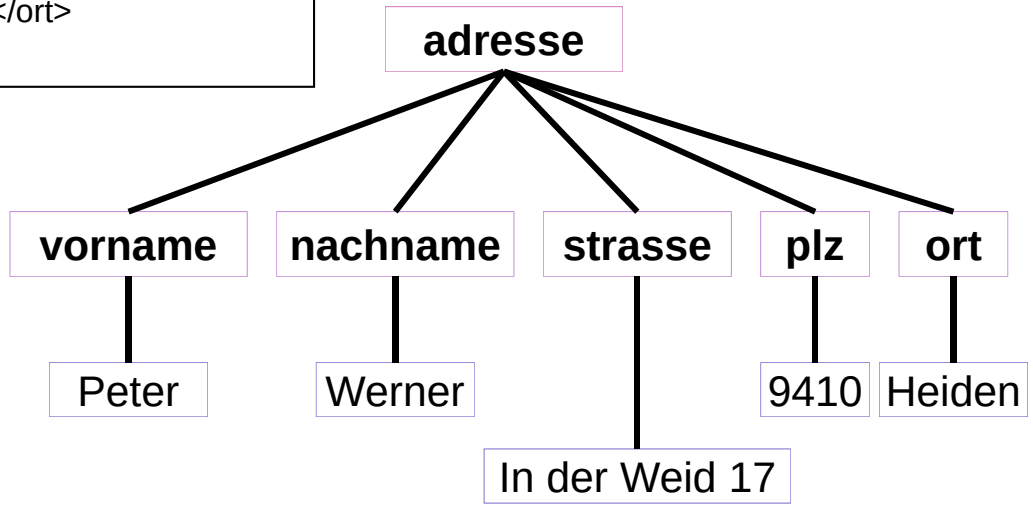
a + b * c



Beispiel 3: XML- Dokument

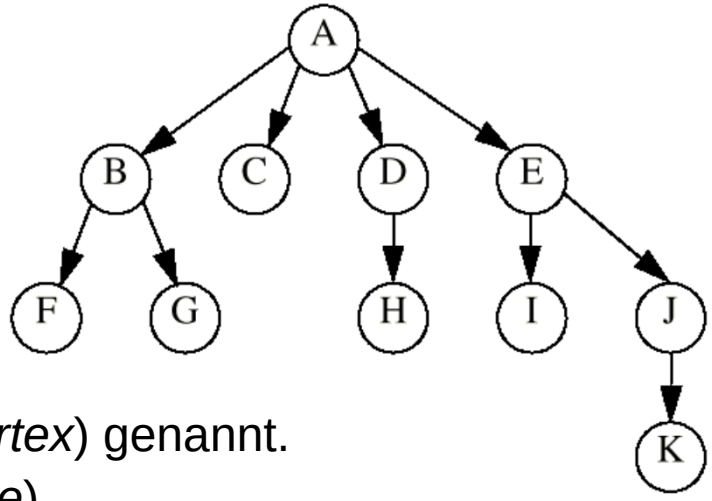
- Ein XML Dokument besteht aus einem Wurzelelement an dem beliebig viele Nachfolgeelemente angehängt sind, an denen wiederum Nachfolgeelemente hängen können.

```
<adresse>
    <anrede>Herr</anrede>
    <vorname>Peter<vorname>
    <nachname>Werner</nachname>
    <strasse>In der Weid 17</strasse>
    <plz>9410</plz>
    <ort>Heiden</ort>
</adresse >
```



Definition Baum (nicht rekursiv)

Ein (gerichteter) Baum *T=(V,E)* besteht aus einer Menge von ***Knoten V*** und einer Menge von ***gerichteten Kanten E***. Der *root*-Knoten $r \in V$ hat nur Ausgangskanten. Alle anderen Knoten $n \in V \setminus r$ haben genau eine Eingangskante, wobei für alle Kanten gilt: e= (v1 ,v2) und v1 != v2 .

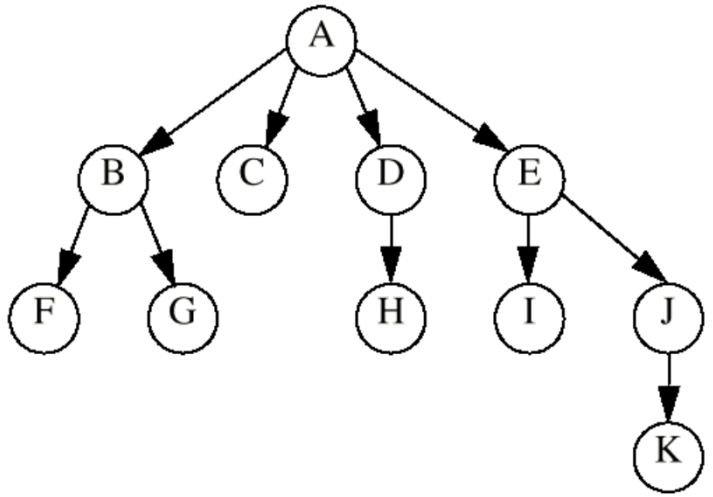


Hinweis:

- Knoten werden auch *vertices* (sing. *vertex*) genannt.
- Kanten heissen auch *edges* (sing. *edge*).

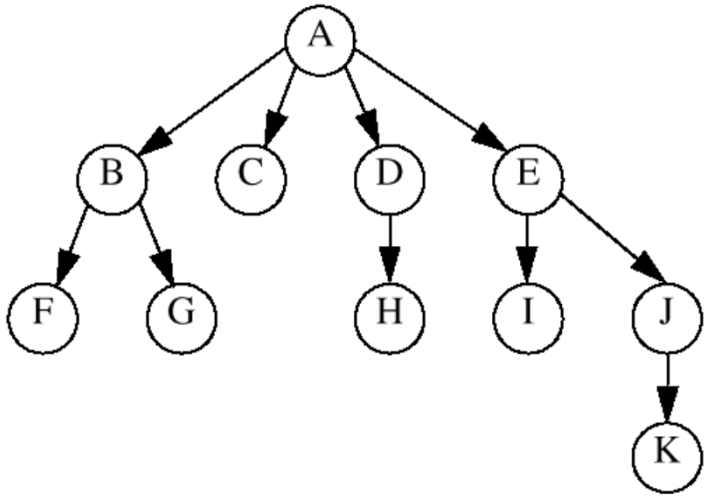
Definitionen 1

- Die Nachfolger (*descendants*) eines Knotens sind alle Knoten, die von diesem erreichbar sind, wenn man zu den Blättern absteigt.
- Die direkten Nachfolger werden als Kinder (*children*) bezeichnet.
- Die Vorgänger (*ancestor*)-Knoten eines Knotens, sind alle Knoten, die auf dem Weg von der Wurzel bis zu diesem Knoten traversiert werden.
- Der direkte Vorgänger eines Knoten wird mit Vater (*parent*) bezeichnet.
- Knoten mit Nachfolger werden als innere Knoten bezeichnet
- Knoten ohne Nachfolger sind Blattknoten (*leaves*).
- Knoten mit dem gleichen Vater-Knoten sind Geschwisterknoten (*sibling*).
- Es gibt genau einen Pfad vom *Wurzel*-Knoten zu jedem anderen Knoten.
- Die Anzahl der Kanten, denen wir folgen müssen, um von einem Knoten zum andern zu gelangen, ist die Weglänge (*path length*).



Definitionen 2

- Ein Baum mit N Knoten hat ..... Kanten, da jeder Knoten (ausser root) eine Eingangskante hat.
- Eine gerichtete Kante verbindet Vater-Knoten und Kind-Knoten.
- Die *Tiefe/Höhe* eines Baumes gibt an, wie weit die "tiefsten" Blätter von der Wurzel entfernt sind, d.h. die Länge des Pfades von der Wurzel bis zu den tiefsten Blättern.
- Das *Gewicht* ist die Anzahl der Knoten des Baumes



Übung

root-Node = ...

Tiefe = ….

Gewicht = ….

Nachfolger von B = ...

Nachfolger von A = ...

Vorgänger von K = ...

Vater von K =

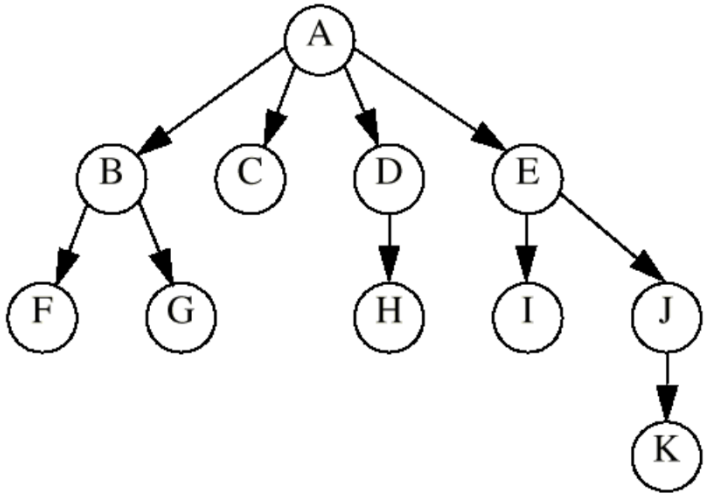
Blattknoten = ...

Geschwister von C = ...

Geschwister von H = ...

Ein Knoten hat ..... *parent*-Knoten.

Ein Knoten kann ..... Kinder haben.



Definition Baum (rekursiv)

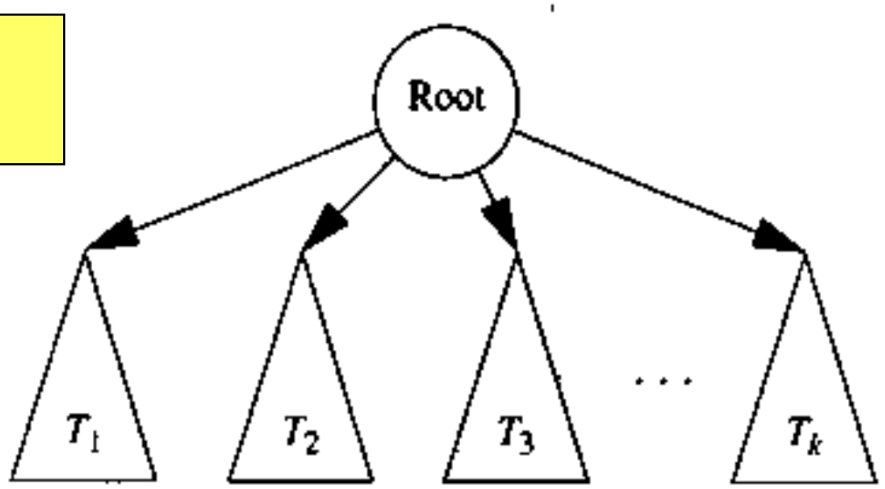
ein Baum ist leer

oder

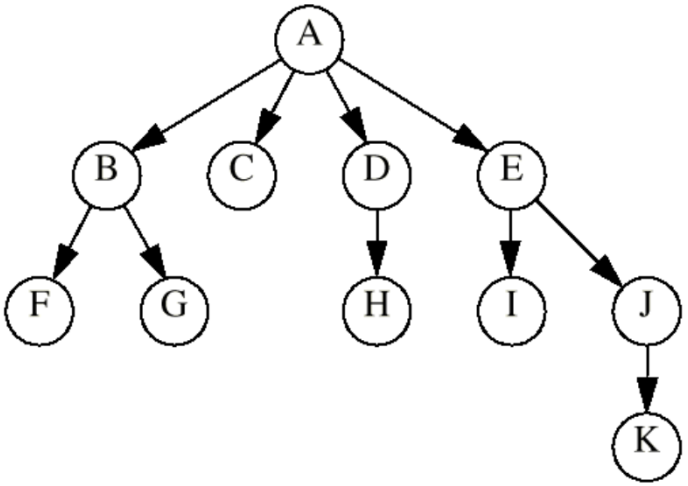
er besteht aus einem Element (root Knoten) mit Null oder mehr Teilbäumen T1 , T2 , ... Tk .

Baum = leer

Baum = Element (Baum)*



Implementation




```
class Tree<T> {
   T element;
   Tree[] edges;

   Tree(T theElement){
     element = theElement;
   }
}
```

Jeder Knoten hat Zeiger auf jedes Kind im Array gespeichert

Die Zahl der Kinder pro Knoten kann *stark variieren* und ist meist *nicht zum Voraus bekannt*

-> nicht effizient.

mögliche bessere Lösung: Zeiger in *Linked List* verwalten.

Der Binärbaum

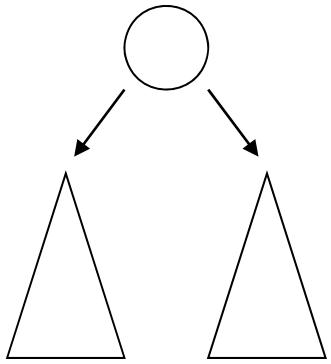
Die am häufigsten verwendete Art von Bäumen: beim Binärbaum hat ein Knoten maximal **2 Kinder.**

Definition (rekursiv):

Ein Binärbaum ist entweder leer, oder besteht aus einem Wurzel-Knoten und aus einem linken und einem rechten Teilbaum.

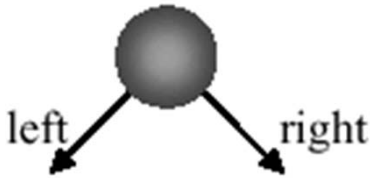
Baum ::= leer

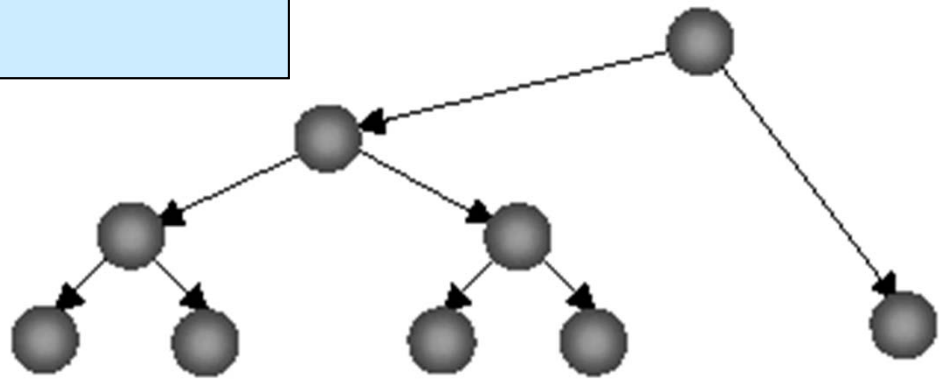
Baum ::= Element (Baum) (Baum)



Die Datenstruktur des Binärbaums

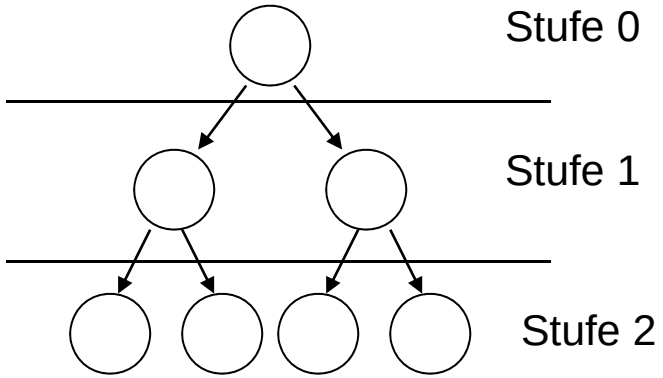
```
public class Tree<T> {
   T element;
   Tree<T> left;
   Tree<T> right;

   Tree(T theElement) {
      element = theElement;
   }
}
```





Eigenschaften

- Tiefe: *k*
- auf jeder Stufe *s* max. 2^s Knoten
- Maximal 2^{k+1}- 1 Knoten



- Ein Binärbaum heisst **voll**, wenn ausser der letzten alle seine Ebenen voll besetzt sind.
- Frage: wie tief ist ein voller Binärbaum mit 37 Knoten?
  … mit *N* Knoten?

Übung: Auflisten aller Elemente

Baum = leer
Baum = Element (Baum) (Baum)

```
public class Tree<T> {
   T element;
   Tree<T> left;
   Tree<T> right;
}
```

- Schreiben Sie eine rekursive Methode `traverseTree`, die die Elemente eines Baumes ausgibt

Hinweis: Rekursives Traversieren einer Liste:

```
void traverseList (ListNode<T> node) {
    if (node!=null) {
        System.out.println(node.element);
        traverseList(node.next);
    }
}
```

oder

```
void traverse () {
    System.out.println(element);
    if (next != null) next.traverse();
}
```

Traversieren

Das (rekursive) Besuchen aller Knoten in einem Baum wird als durchlaufen oder traversieren bezeichnet.

Die Art der Traversierung bestimmt die Reihenfolge, in welcher die Knoten besucht werden.

Dadurch sind die Knoten *“linear geordnet”*

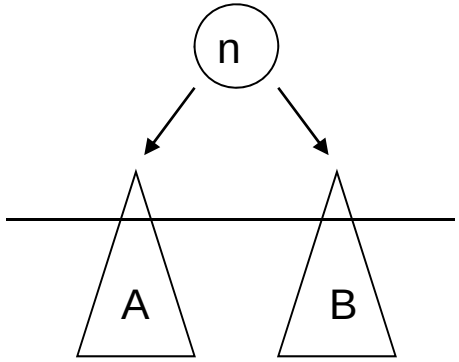
Die möglichen Arten von Traversierung (beim Binärbaum) sind:

Preorder, Knoten zuerst: n, A, B

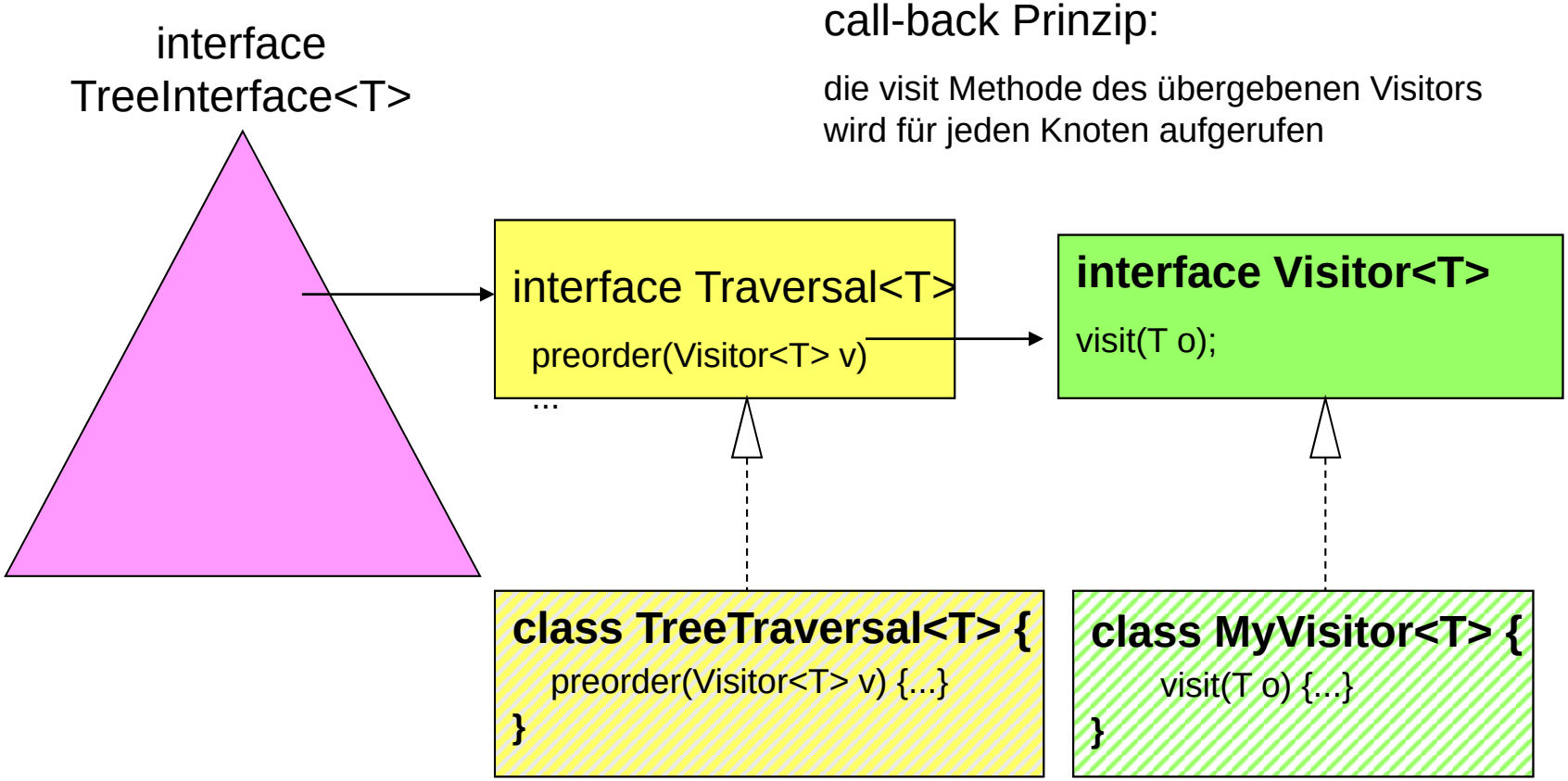
Inorder, Knoten in der Mitte: A, n, B

Postorder, Knoten am Schluss: A, B, n

Levelorder: n, a_0, b_0, a_1, a_2, b_1, b_2, ..



Klassen zur rekursiven Traversierung



Klassen zur rekursiven Traversierung

```
interface TreeInterface<T> {
     Traversal<T> traversal();
     void add(T o);
     void remove(T o);
}
```

Schnittstelle mit den grundlegenden Operationen

```
interface Traversal<T> {
     void preorder(Visitor<T> visitor);
     void inorder(Visitor<T> visitor);
     void postorder(Visitor<T> visitor);
}
```

Interface mit traversal Methode(n)

```
interface Visitor<T> {
   void visit(T o);
}
```

Interface des "Besuchers"

```
class Tree<T> implements TreeInterface<T>{
      T element;
      Tree<T> left,right;
      ...
}
```

Implementation des Baums

```
class MyVisitor<T> implements Visitor<T>{
     visit(T o) {
          System.out.println(o);
     }
}
```

Implementation des "Besuchers"

Klassen zur rekursiven Traversierung

```
interface Traversal<T> {
    public void preorder (Visitor<T> visitor);
    public void inorder  (Visitor<T> visitor);
    public void postorder(Visitor<T> visitor);
}
```

```
class TreeTraversal<T> implements Traversal<T> {
   Tree<T> root;
   TreeTraversal(Tree<T> tree) {
      root = tree;
   }
   private void preorder(Tree<T> tree, Visitor<T> visitor) {
      if (tree != null) {
         visitor.visit(tree.element);
         preorder(tree.left, visitor);
         preorder(tree.right, visitor);
      }
   }
   public void preorder(Visitor<T> visitor) {
      preorder(root,visitor);
   }
   ...
}
```

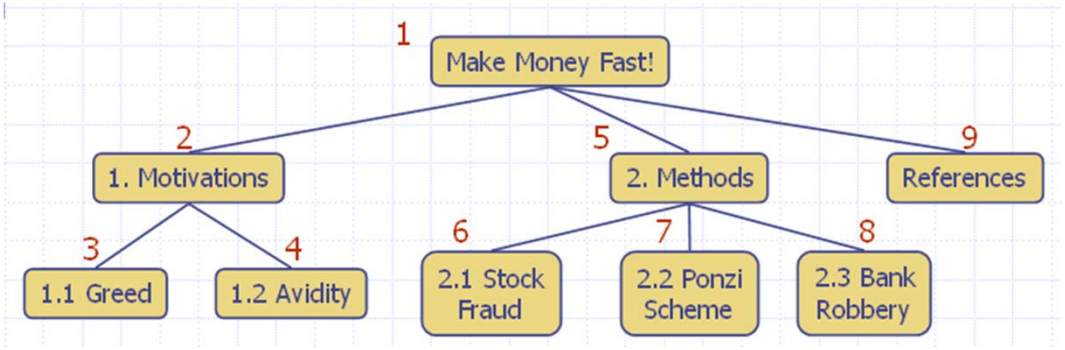
Implementation Preorder (Verarbeitung am Anfang)

- **Besuche die Wurzel.**
- **Traversiere den linken Teilbaum (in Preorder).**
- **Traversiere den rechten Teilbaum (in Preorder).**

```
class TreeTraversal implements Traversal {
  void preorder(Tree tree, Visitor visitor) {
    if (tree != null) {
      visitor.visit(tree.element);
      preorder(tree.left, visitor);
      preorder(tree.right, visitor);
    }
  }
}
```

Bemerkung:
in dieser und den folgenden Folien sind die generischen Parameter weggelassen

Anwendung:
Strukturierte Ausgabe



Implementation Postorder (Verarbeitung am Schluss)

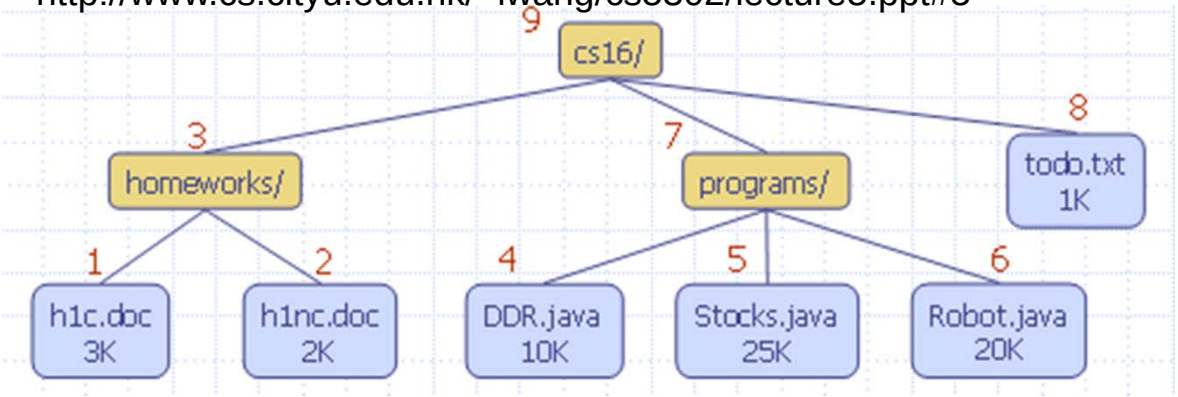
- **Traversiere den linken Teilbaum (in Postorder).**
- **Traversiere den rechten Teilbaum (in Postorder).**
- **Besuche die Wurzel.**

```
void postorder(Tree tree, Visitor visitor) {
  if (tree != null) {
    postorder(tree.left,visitor);
    postorder(tree.right,visitor);
    visitor.visit(tree.element);
  }
}
```

Anwendung:

- Elementweises Löschen eines Baumes.
- Berechnung des Speicherplatzes von Verzeichnissen.

http://www.cs.cityu.edu.hk/~lwang/cs5302/lecture5.ppt#5



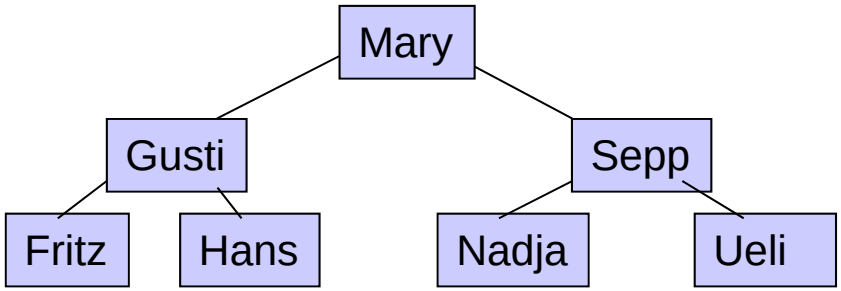
Implementation Inorder (Verarbeitung in der Mitte)

- **Traversiere den linken Teilbaum (in Inorder).**
- **Besuche die Wurzel.**
- **Traversiere den rechten Teilbaum (in Inorder).**

```
void inorder(Tree tree, Visitor visitor) {
  if (tree != null) {
    inorder(tree.left,visitor);
    visitor.visit(tree.element);
    inorder(tree.right,visitor);
  }
}
```

Der Baum wird quasi von links nach rechts abgearbeitet.

Anwendung: Bei einem sortierten Baum werden die Elemente in der richtigen Reihenfolge traversiert.



Implementation Levelorder

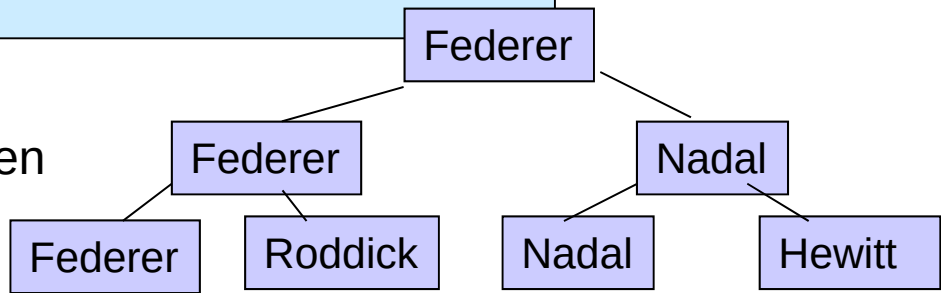
Besuche die Knoten schichtenweise:

- **zuerst die Wurzel,**
- **dann die Wurzel des linken und rechten Teilbaumes,**
- **dann die nächste Etage, usw. ...**

```
void levelorder(Tree<T> tree, Visitor<T> visitor)
{  Queue<T> q = new Queue<T>();
   if (tree != null) q.put(tree);
   while (!q.empty()) {
      tree = q.get();
      visitor.visit(tree.element);
      if (tree.left  !=null) q.put(tree.left);
      if (tree.right !=null) q.put(tree.right);
   }
}
```

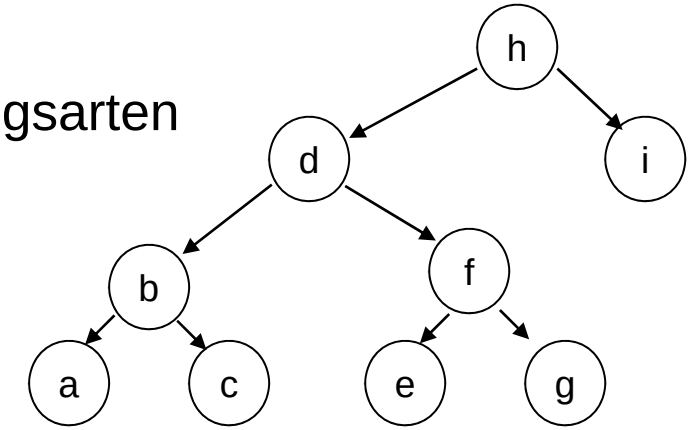
Statt eines (impliziten) Stacks wird eine Queue verwendet

Anwendung: Viertels-, Halb-, Finalpaarungen



Übung

Zeigen Sie die Reihenfolge bei
den verschiedenen Traversierungsarten
auf



Preorder: (n, L, R) ……

Inorder: (L, n, R) ……..

Postorder: (L, R, n) ……..

Levelorder: ……..

Sortierte Binärbäume

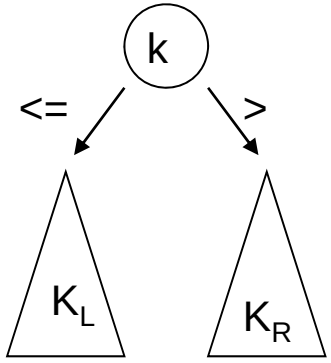
Beim binären Suchbaum werden die Objekte anhand ihres (Schlüssel-) Werts geordnet eingefügt:

In Java: Interface `Comparable`

Definition:

Für jeden Knoten gilt*:

- im linken Unterbaum sind alle kleineren Elemente K_L <= k
- im rechten Unterbaum sind alle grösseren Elemente: K_R >* k



* manchmal auch < und >=

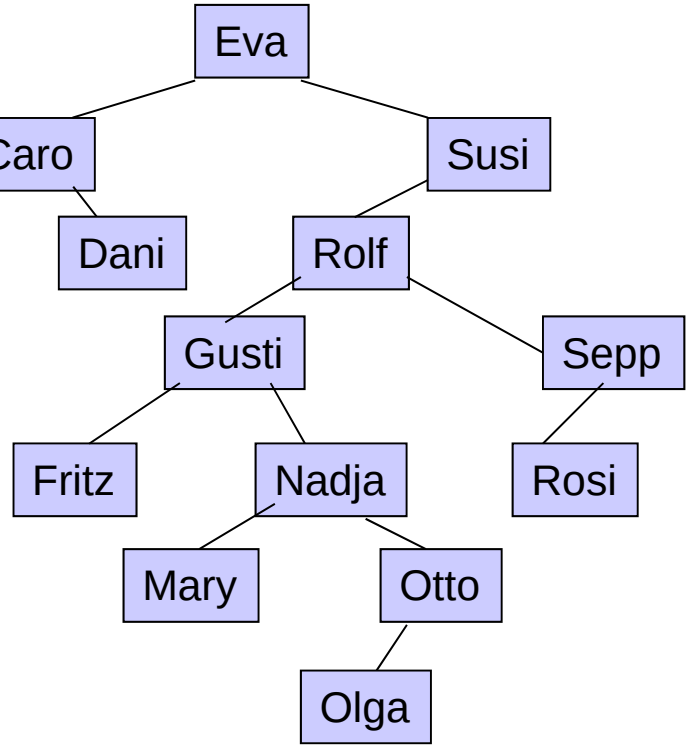
Einfügen

- Beim Einfügen muss links eingefügt werden, wenn das neue Element kleiner oder gleich ist, sonst rechts

```
class BinaryTree<T extends Comparable<T>> {

    private T element = null;
    private BinaryTree<T> left  = null;
    private BinaryTree<T> right = null;

    BinaryTree(T element) { this.element = element; }
    BinaryTree() {}

    public void insert (T c) {
        if (element==null) element = c;      // only necessary for root
        else if (c.compareTo(element) <= 0) {
            if (left==null) left = new BinaryTree<T>(c);
            else left.insert(c);
        }
        else {
            if (right==null) right = new BinaryTree<T>(c);
            else right.insert(c);
        }
    }
}
```

Beispiel eines sortierten Binärbaums

- Der sortierte Binärbaum hat nach dem Einfügen von *Eva, Caro, Susi, Rolf, Gusti, Fritz, Nadja, Sepp, Rosi, Otto, Mary, Dani, Olga* folgende Gestalt:




```
class BinaryTree<T extends Comparable<T>> {

     private T element = null;
     private BinaryTree<T> left  = null;
     private BinaryTree<T> right = null;

     BinaryTree(T element) { this.element = element; }
     BinaryTree() {}

     public void insert (T c) {
          if (element==null) element = c;
          // only necessary for root
          else if (c.compareTo(element) <= 0) {
               if (left==null) left = new BinaryTree<T>(c);
               else left.insert(c);
          }
          else {
               if (right==null) right = new BinaryTree<T>(c);
               else right.insert(c);
          }
     }
}
```

Übung

- Nummerieren Sie die Knoten entsprechend ihrer Reihenfolge beim Einfügen

- Geben Sie an, welche Namen bei einer Inorder-Traversierung ausgegeben werden

- Zeichnen Sie den Baum auf, bei dem man die Namen in umgekehrter Reihenfolge eingefügt hat, d.h. Olga, Dani, Mary, Otto, ...

Löschen: einfache Fälle

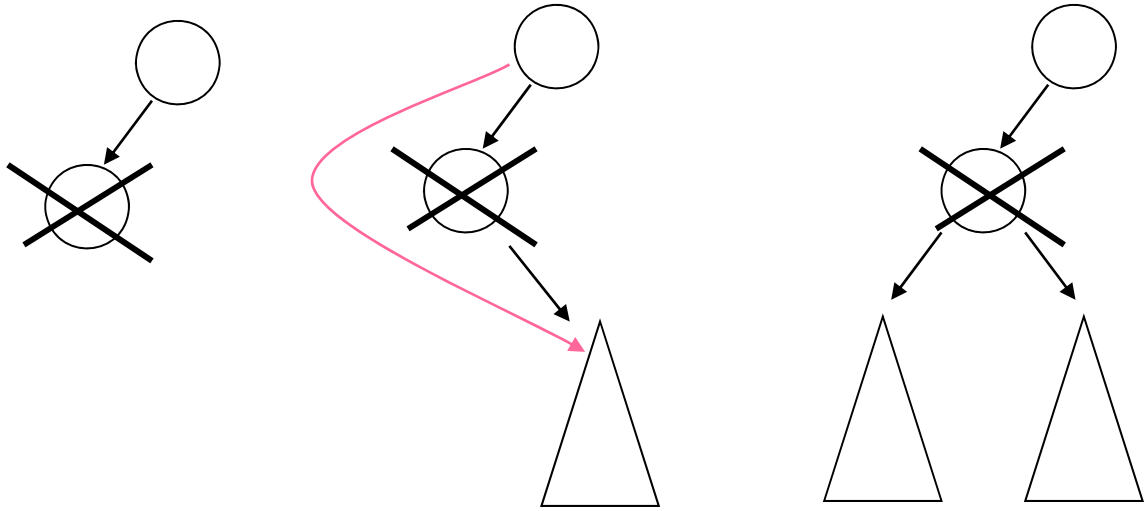
a) den zu entfernenden Knoten suchen

b) Knoten löschen. Dabei gibt es 3 Fälle:

1) der Knoten hat keine Kinder $\Rightarrow$ Knoten löschen

2) der Knoten hat ein Kind

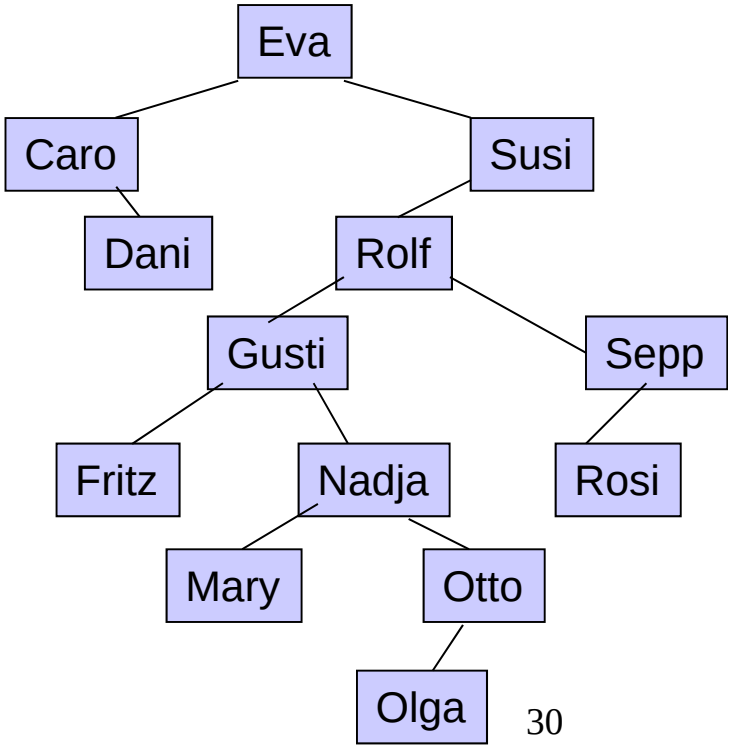
3) der Knoten hat zwei Teilbäume (später)



Löschen: einfache Fälle

```
class BinaryTree<T extends Comparable<T>> {
      T element = null;
      BinaryTree<T> left = null;
      BinaryTree<T> right = null;

      BinaryTree() {}

      BinaryTree(T element) {...}

      public void insert (T c) {...}

      public BinaryTree remove (T c) throws NoSuchElementException {
            if (element == null) return this;     // an empty tree.
            else {
                  if(c.compareTo(element) < 0) {
                        if (left != null)  left = left.remove(c);
                        else throw new NoSuchElementException(…);
                  }
                  else if (c.compareTo(element) > 0) {
                        if (right ! = null) right = right.remove(c);
                        else throw new NoSuchElementException(…);
                  }
                  else if (c.compareTo(element)==0) {          // found
                        if (left == null) return right;
                        if (right == null) return left;
                        else {...}                          // komplizierter Fall
                  }
            }
            return this;
      }
}
```

Aufruf:
myTree = myTree.remove(element);



Löschen, komplizierter Fall

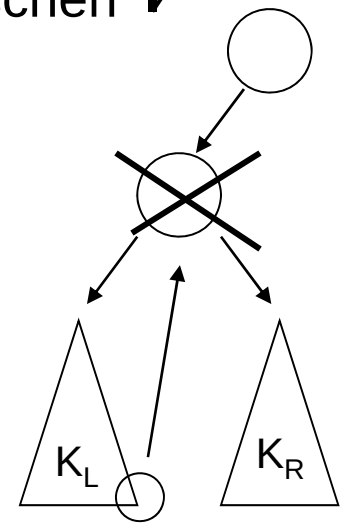
a) den zu entfernenden Knoten suchen

b) Knoten löschen. Dabei gibt es 3 Fälle:

1) der Knoten hat keinen Nachfolger $\Rightarrow$ Knoten löschen ✓

2) der Knoten hat einen Nachfolger ✓

3) der Knoten hat zwei Nachfolger



Fall 3: Es muss ein Ersatzknoten mit Schlüssel k gefunden werden, so dass gilt: $K_L <= k$ und $K_R > k$

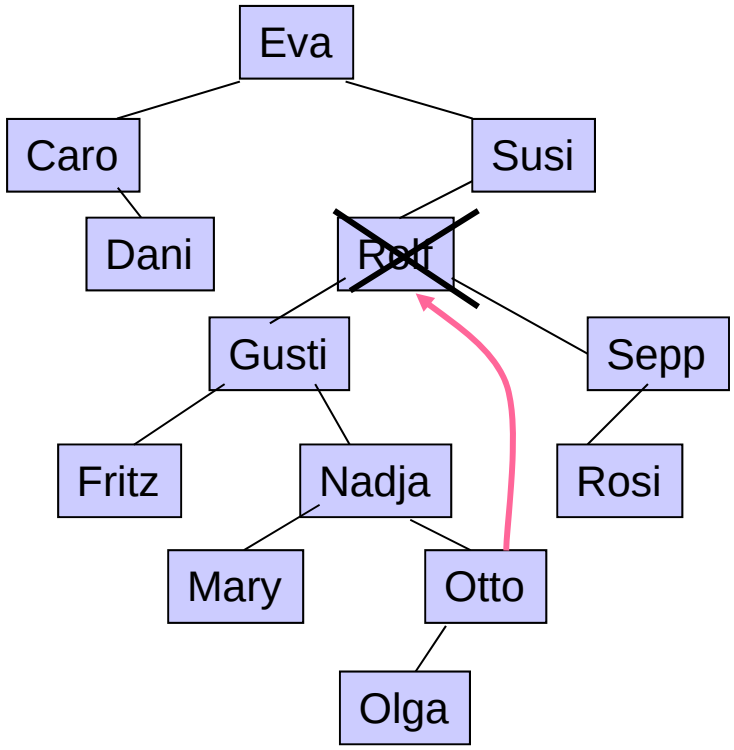
Lösung: der Knoten, der im linkenTeilbaum ganz rechts liegt, bzw. im rechten Teilbaum ganz links liegt

Löschen Beispiel

Es soll *Rolf* gelöscht werden.

Vom linken Teilbaum wird das Element ganz rechts als Ersatz genommen, also *Otto*.

Löschen den Knoten *Otto* (das ist einfach, denn er hat höchstens einen Nachfolger) und ersetzen den Inhalt von *Rolf* durch den Inhalt von *Otto.*



Löschen Code

```
public BinaryTree<T> remove(T c) throws Exception {
   if (element == null) return this;       // only necessary for root

   else {
      if(c.compareTo(element) < 0) {
         if (left!=null) left = left.remove(c);
         else throw new NoSuchElementException(...);
      }
      else if (c.compareTo(element) > 0) {
         if (right!=null) right = right.remove(c);
         else throw new NoSuchElementException(...);
      }
      else if (c.compareTo(element)==0) {  // found
         if (left  == null) return right;
         if (right == null) return left;
         else {
            Wrapper<T> w = new Wrapper<T>();
            left = left.removeLargest(w);
            element = w.c;
         }
      }
   }
   return this;
}

private BinaryTree<T> removeLargest(Wrapper<T> w) {
   if (right != null) right = right.removeLargest(w);
   else {  // found
      w.c = element;
      return left;
   }
   return this;
}
```

class Wrapper<T> { T c=null; }
Used to return the value of a non-mutable element (like a String)

Aufgabe: Verstehen Sie diesen Code

Suchen (Ausblick)

Suche x im Baum B:

- **Wenn x == Wurzelelement gilt, haben wir x gefunden.**
- **Wenn x > Wurzelelement gilt, wird die Suche im rechten Teilbaum von B fortgesetzt, sonst im linken Teilbaum.**

```
public T search(T x) {
  if (element == null) return null;
  else if (x.compareTo(element) == 0)
    return element;
  else if (x.compareTo(element) <= 0)
    return left.search(x);
  else
    return right.search(x);
}
```

- Bei einem vollen Binärbaum müssen lediglich $\log_2$ Schritte durchgeführt werden bis Element gefunden wird.
- Binäres Suchen
- sehr effizient Bsp: 1000 Elemente → 10 Schritte

Zusammenfassung

- Allgemeine Bäume
  - rekursive Definition
  - Knoten (Vertex) und Kanten (Edge)
  - Eigenschaften von Bäumen
- Binärbäume: Bäume mit maximal zwei Kindern
  - Traversal, Visitor
  - verschiedene Traversierungsarten
  - Inorder, Preorder, Postorder, Levelorder
- sortierte Binärbäume Einführung
  - Einfügen
  - Löschen
  - Suchen