

Bäume

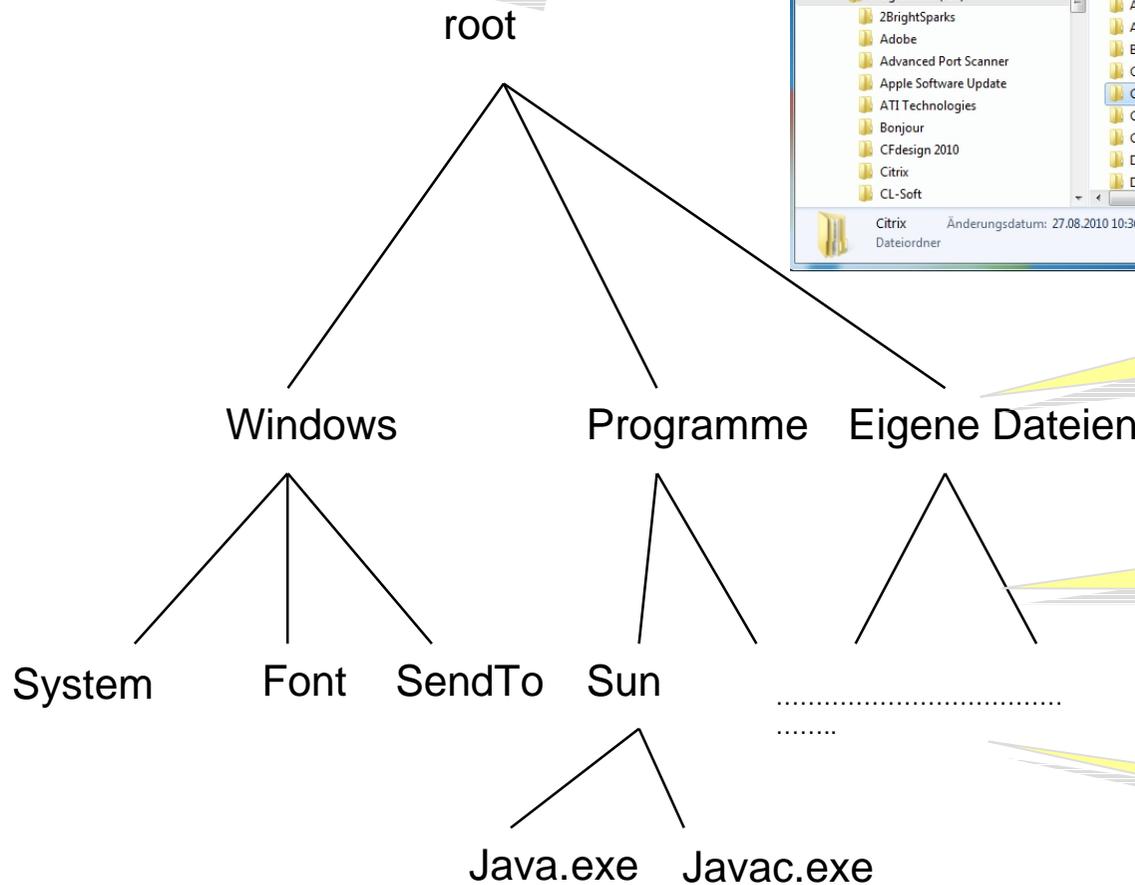
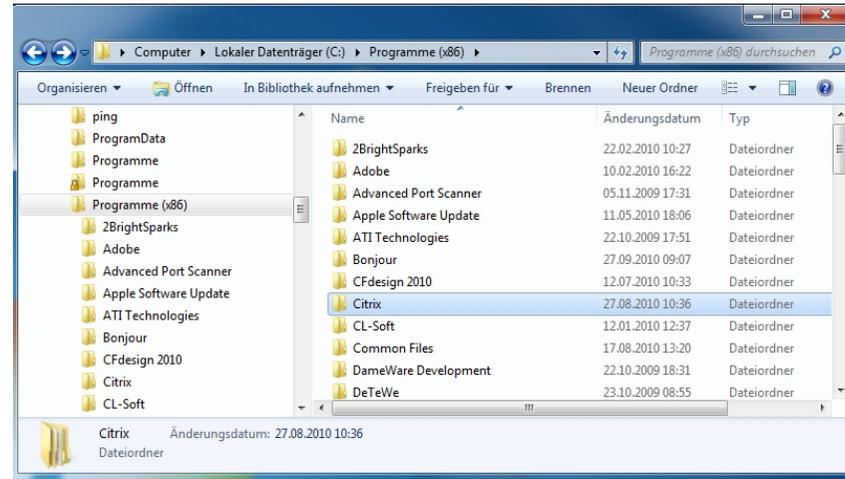


- Sie wissen, was Bäume in der Informatik sind
- Sie kennen das Besucher-Entwurfsmuster
- Sie kennen Binärbäume
- Sie können die Bäume auf unterschiedliche Arten traversieren
- Sie wissen, wie man in Binärbäumen Elemente löscht

Bäume, Anwendung und Begriffe

Beispiel 1: Dateisystem

Wurzel: der Ursprung des Baumes



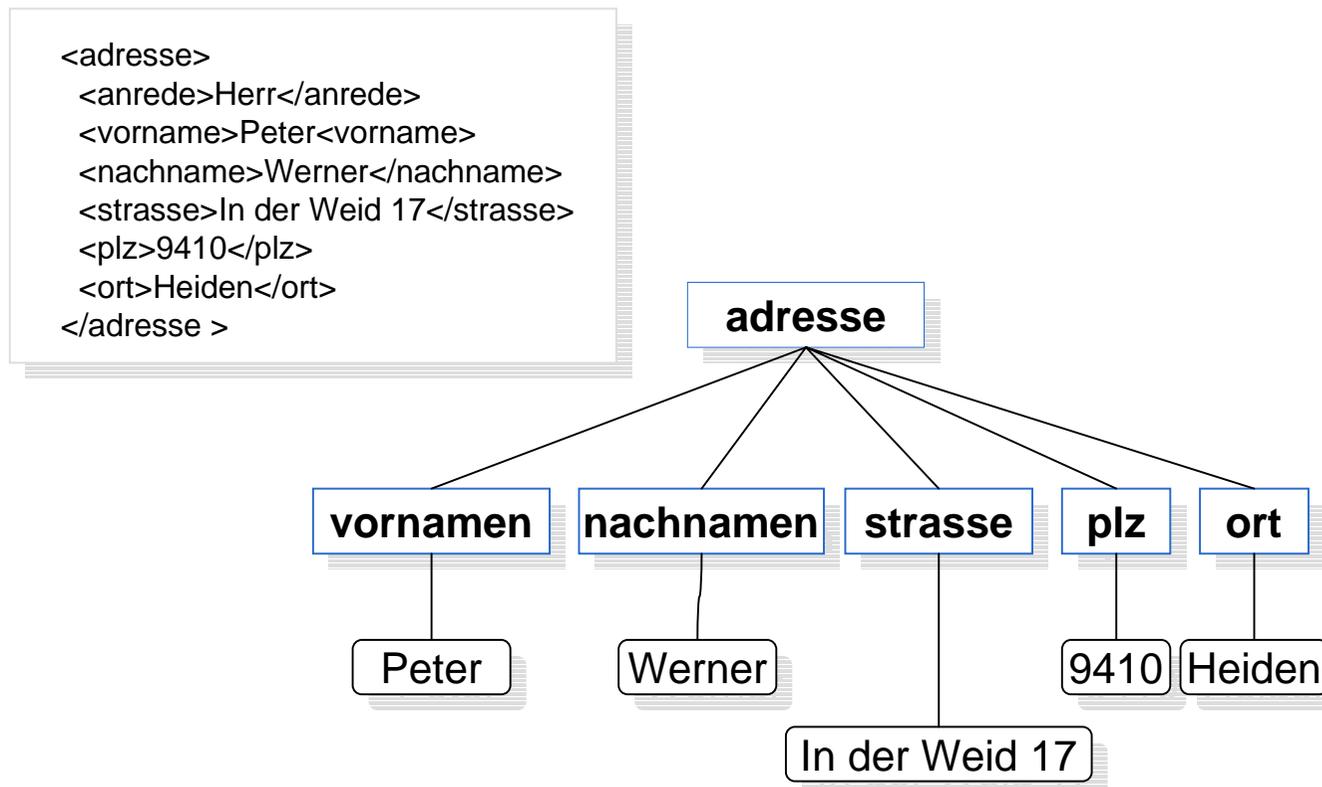
innerer Knoten: Knoten mit Nachfolger

Kante: gerichtete Verbindung zwischen Knoten

Blätter: Knoten ohne Nachfolger

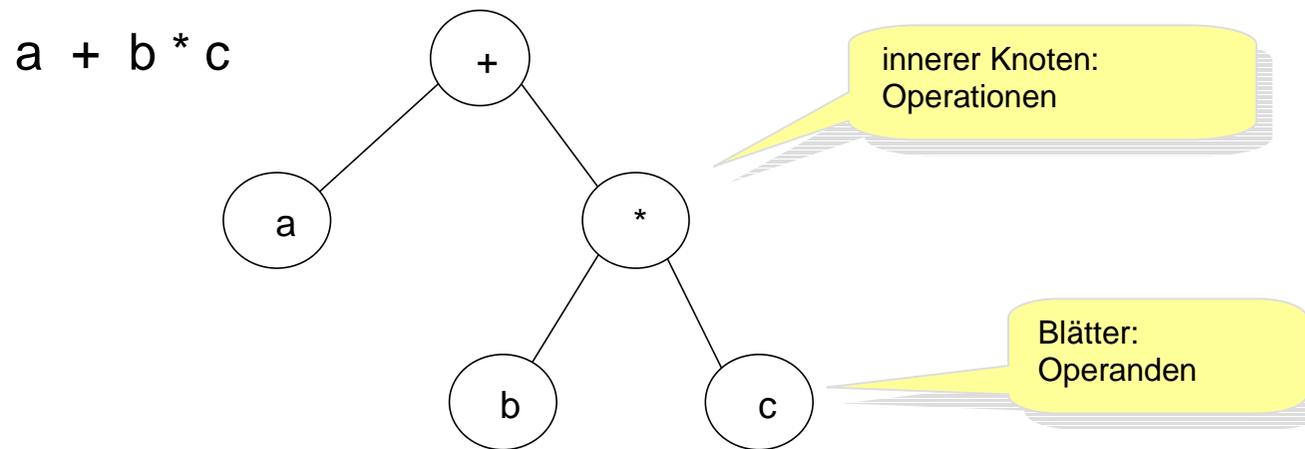
Beispiel 2: XML- Dokument

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



Beispiel 3: Ausdruck-Baum

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



Definition Baum (rekursiv)

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

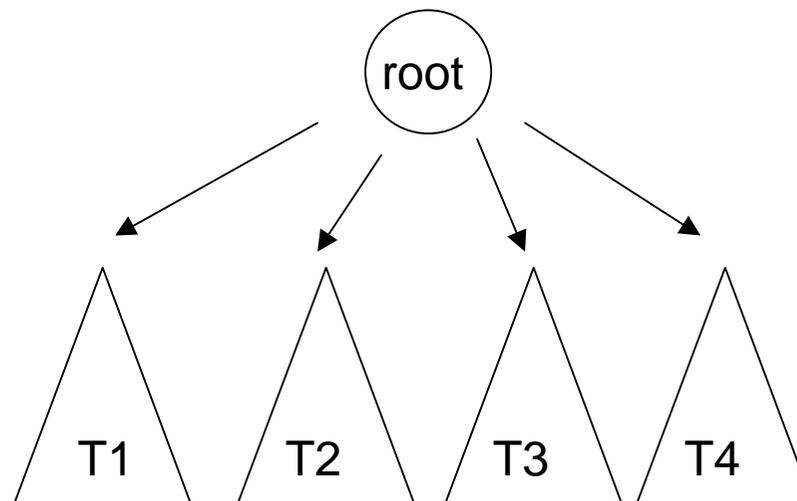
ein Baum ist **leer**

oder

er besteht aus einem Knoten mit keinem, einem oder mehreren disjunkten Teilbäumen T_1 , T_2 , ... T_k .

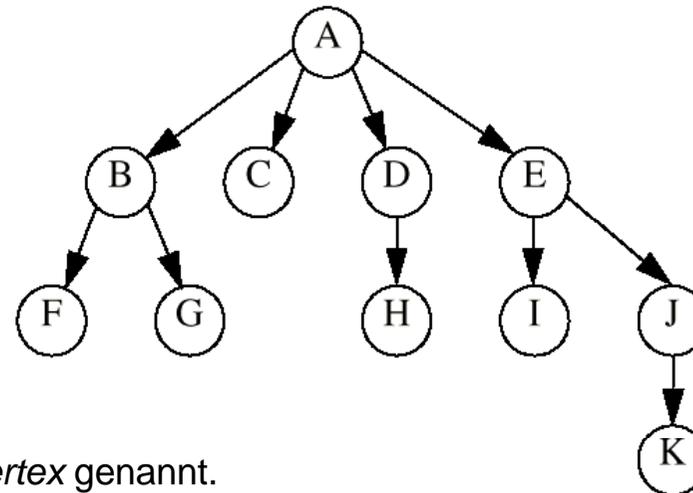
Baum = leer

Baum = Knoten (Baum)*



Definition Baum (nicht rekursiv)

Ein Baum $T=(V,E)$ besteht aus eine 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: $e = (v_1, v_2)$ und $v_1 \neq v_2$.



■ Hinweis:

- Knoten werden auch *vertices* bzw. *vertex* genannt.
- Kanten heissen auch *edges* bzw. *edge*.

Eigenschaften und Begriffe

- Alle Knoten ausser Wurzel (*root*) sind **Nachfolger** (*descendant, child*) genau eines **Vorgänger-Knotens** (*ancestor, parent*).
- Knoten mit Nachfolger werden **innerer Knoten** bezeichnet
- Knoten ohne Nachfolger sind **Blattknoten**.
- Knoten mit dem gleichen Vorgänger-Knoten sind **Geschwisterknoten** (*sibling*).
- Es gibt **genau einen Pfad** vom *Wurzel*-Knoten zu jedem anderen Knoten.
- Die Anzahl der Kanten, denen wir folgen müssen, ist die **Weglänge** (*path length*).
- Die **Tiefe (oder Höhe)** eines Baumes gibt an, wie weit die "tiefsten" Blätter von der Wurzel entfernt sind: Anzahl Kanten + 1.
- Die **Gewicht** ist die Anzahl der Knoten des (Teil-)Baumes

Übung

root-Node =

Tiefe =

Gewicht =

Nachfolger von B =

Nachfolger von A =

Vorgänger von K =

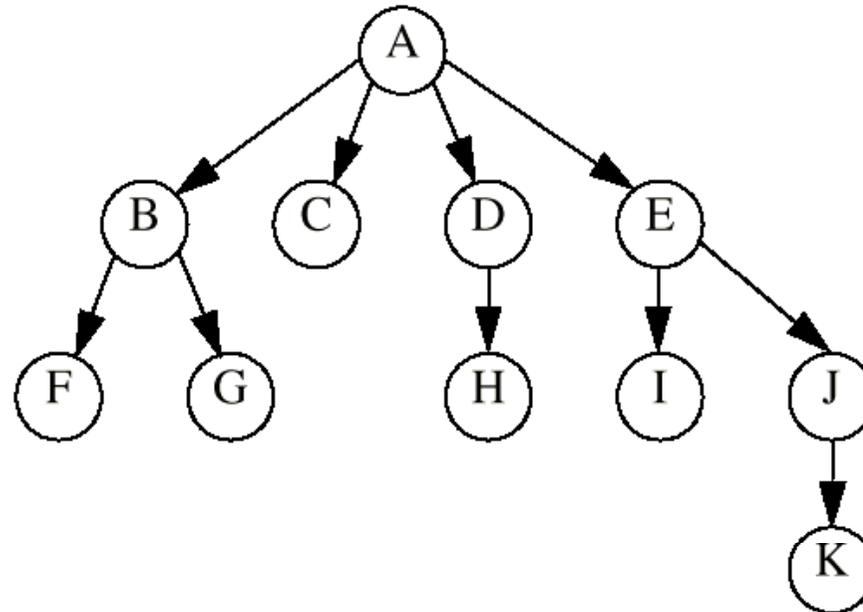
Blattknoten =

Geschwister von C =

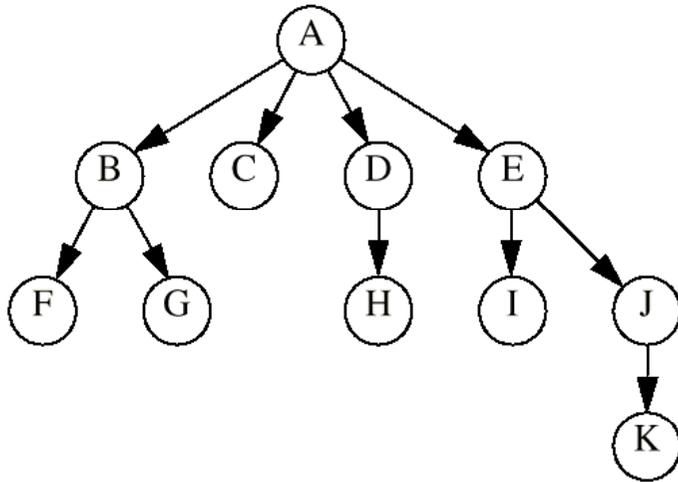
Geschwister von H =

Ein Knoten hat direkte Vorgänger-Knoten.

Ein Knoten kann direkte Nachfolger haben.



Knoten Implementation



```
class TreeNode<T> {  
    T element;  
    List<TreeNode<T>> edges;  
  
    TreeNode(T theElement) {  
        element = theElement;  
    }  
}
```

- Jeder Knoten hat Zeiger auf jeden Nachfolger in Array gespeichert
- Die Zahl der Nachfolger pro Knoten kann *stark variieren* und ist meist *nicht zum voraus bekannt* -> nicht effizient.
- mögliche bessere Lösung: Zeiger in *Liste* verwalten.

Binärbaum

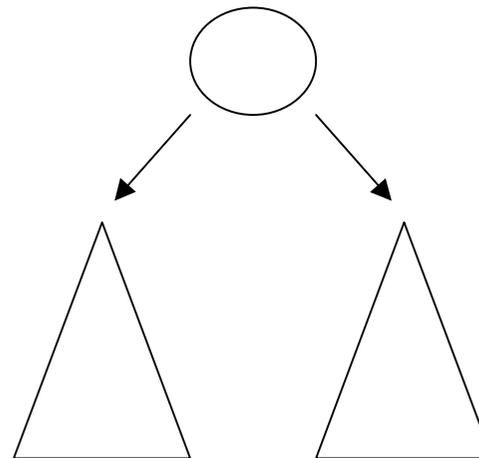
Der Binärbaum

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

■ Definition (rekursiv):

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

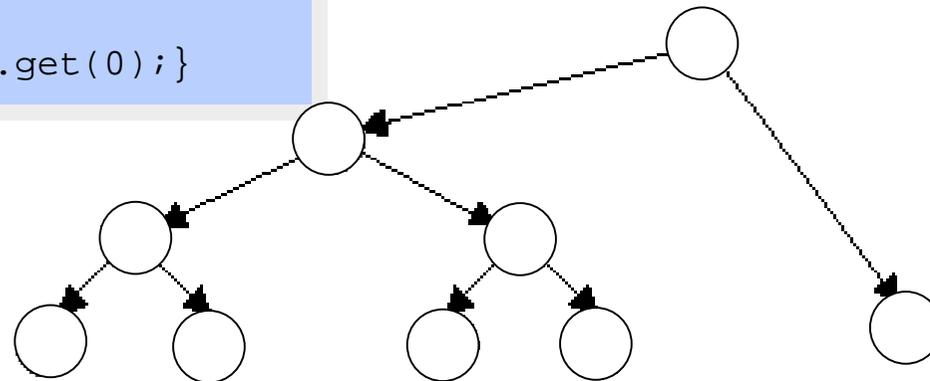
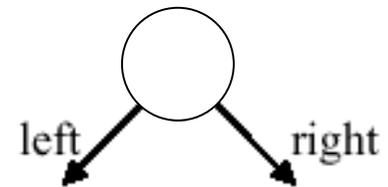
Baum = leer
Baum = Knoten (Baum Baum)



Die Datenstruktur des Binärbaums

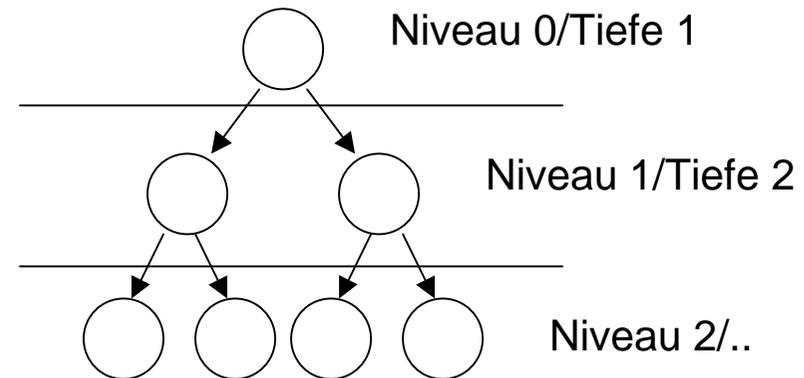
```
class TreeNode<T> {  
    List<T> elements;  
    TreeNode<T> left;  
    TreeNode<T> right;  
  
    TreeNode(T theElement) {  
        elements = new LinkedList<>();  
        elements.add(theElement);  
    }  
    T getValue(){return elements.get(0);}  
}
```

als Liste wegen
Doubletten



Eigenschaften

- Tiefe/Höhe: k
- auf jedem Niveau 2^n Knoten oder $2^{(k-1)}$
- Maximal Anzahl $2^k - 1$ Knoten



- Ein Binärbaum heisst **voll (oder vollständig)**, wenn ausser der letzten alle seinen Ebenen vollständig besetzt sind.
- Aufgabe: Leiten Sie eine Formel für die Höhe des vollen Baumes her und bestimmen Sie die Tiefe eines vollen Binärbaums mit 37 Knoten?
- Hinweis $\log_a n = \ln n / \ln a$

Traversierungen

Ausgeben aller Elemente

Baum = leer
Baum = Element (Baum Baum)

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

- Übung: Schreiben Sie eine rekursive Methode `printTree`, die alle Elemente eines Baumes ausgibt

Hinweis: Ausgeben einer Liste mit einer rekursiven Methode:

```
void printList(ListNode node) {  
    if (node != null)  
        System.out.println(node.getValue());  
        printList(node.next)  
    }  
}
```

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”*

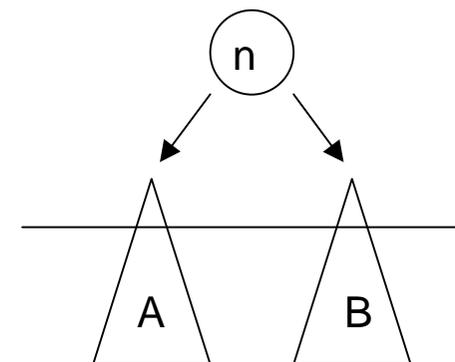
Die möglichen Arten von Traversierung (beim Binärbaum) sind:

Preorder Knoten zuerst: n, A, B

Inorder Knoten in der Mitte: A, n, B

Postorder Knoten am Schluss: A, B, n

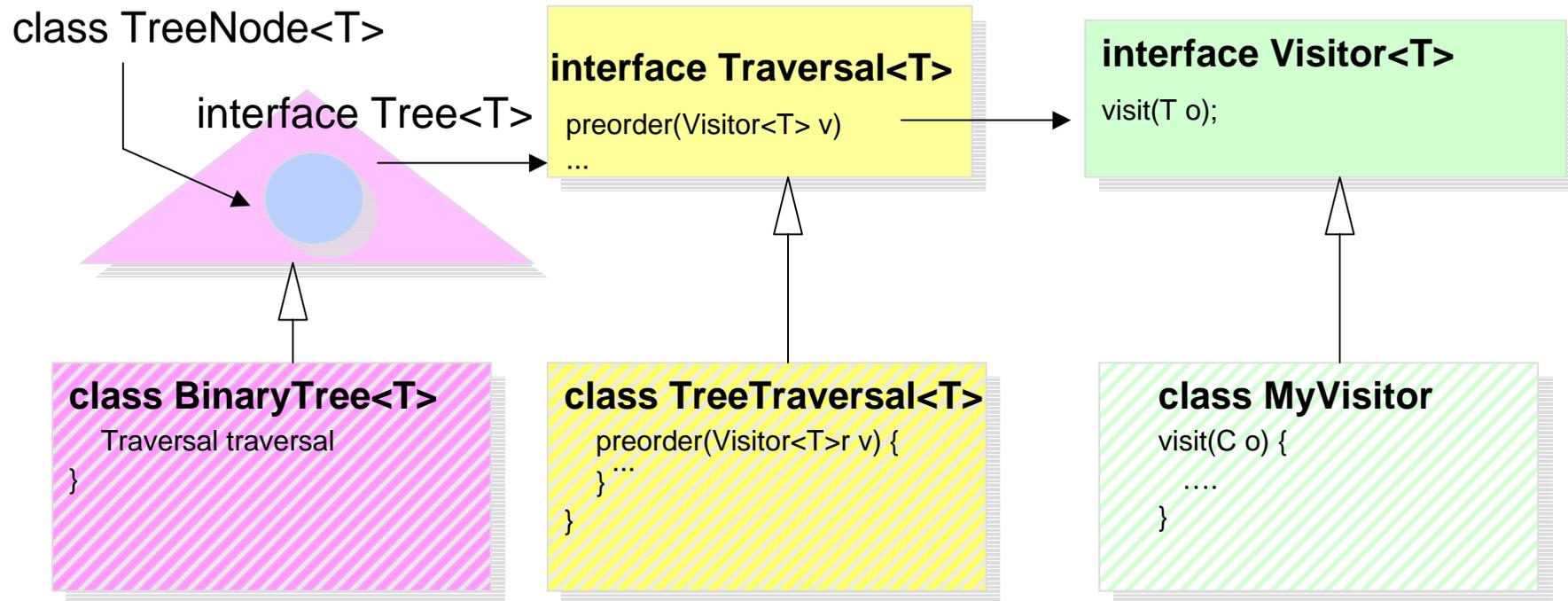
Levelorder: n, a₀, b₀, a₁, a₂, b₁, b₂, ..



Klassen zur rekursiven Traversierung

Rückruf Prinzip & Besucher Entwurfsmuster:

die visit Methode des übergebenen Visitors wird für jeden Knoten aufgerufen



Klassen zur Traversierung eines Baumes

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

Schnittstelle mit den
grundlegenden Operationen

```
class TreeNode<T> {  
    T element;  
    TreeNode<T> left, right;  
}
```

Knotenelement

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

Interface mit traversal Methode(n)

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

Interface des "Besuchers"

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

Implementation des "Besuchers"

Implementation Preorder

■ Verarbeitung am Anfang

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

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

```
class MyCVisitor implements Visitor<T> {  
    public void visit (T obj) {System.out.println(obj);}  
}
```

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



call-back (Hollywood) Prinzip

Implementation Postorder

■ Verarbeitung am Schluss

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

```
private void postorder(TreeNode<T> node, Visitor<T> visitor) {  
    if (node != null) {  
        postorder(node.left, visitor);  
        postorder(node.right, visitor);  
        visitor.visit(node.getValue());  
    }  
}
```

■ Zuerst werden die Nachfolger abgearbeitet und dann der Knoten selber (von unten nach oben)

■ somit sind left und right-Teilbaum verarbeitet und die Verweise können "verändert" bzw. anderweitig gesetzt werden.

- kann z.B. ausgenutzt werden, wenn der Baum umgespeichert werden muss
 - z.B bei *Expression-Tree: Umwandlung von Infix -> Prefix Notation*

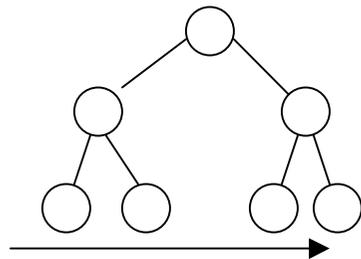
Implementation Inorder

■ Verarbeitung in der Mitte

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

```
private void inorder(TreeNode<T> node, Visitor<T> visitor) {  
    if (node != null) {  
        inorder(node.left, visitor);  
        visitor.visit(node.getValue());  
        inorder(node.right, visitor);  
    }  
}
```

■ Der Baum wird quasi von links nach rechts abgearbeitet



Implementation Preorder mit explizitem Stack

- Es wird nicht der Aufrufstack verwendet
- Kindelemente werden auf den Stack abgelegt und im nächsten Durchgang verarbeitet

```
void preorder(TreeNode<T> node, Visitor<T> visitor) {  
    Stack s = new Stack();  
    if (node != null) s.push(node);  
    while (!s.isEmpty()){  
        node = s.pop();  
        visitor.visit(node.getValue());  
        if (node.right != null) s.push(node.right);  
        if (node.left != null) s.push(node.left);  
    }  
}
```

Implementation Levelorder

■ Besuche die Knoten schichtenweise:

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

```
void levelorder(TreeNode<T> node, Visitor<T> visitor) {  
    Queue q = new Queue();  
    if (node != null) q.enqueue(node);  
    while (!q.isEmpty()){  
        node = q.dequeue();  
        visitor.visit(node.getValue());  
        if (node.left != null) q.enqueue(node.left);  
        if (node.right != null) q.enqueue(node.right);  
    }  
}
```

Aufgerufene Methode (Lambda Ausdruck)

- Methode ist in einer Klasse die das Interface implementiert

```
Traversal {
    preorder(Visitor<T> visitor);
    ...
}

class MyCVisitor implements Visitor<T> {
    public void visit (T obj) {System.out.println(obj);}
}

tree.traversal().preorder(new MyCVisitor());
```

- Kann auch als Anonyme Klasse "inline" implementiert werden

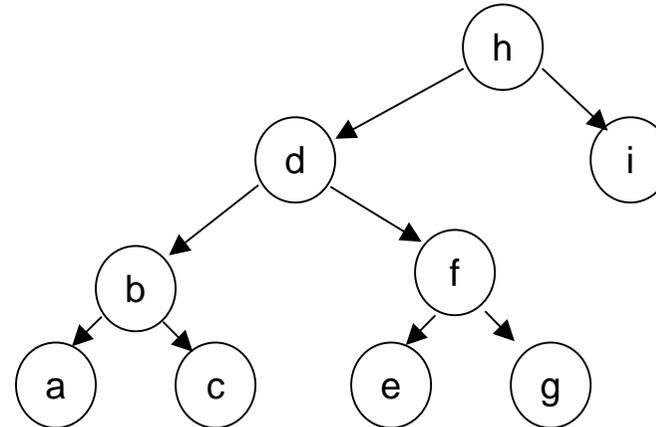
```
tree.traversal().preorder(new Visitor()
    {public void visit (T obj) {System.out.println(obj);}}
);
```

- Noch eleganter als Java 8 Lambda Ausdruck

```
tree.traversal().preorder(obj -> {System.out.println(obj);})
```

Übung

Zeigen Sie die Reihenfolge bei den verschiedenen Traversierungsarten auf



Preorder: (n, L, R)

Inorder: (L, n, R)

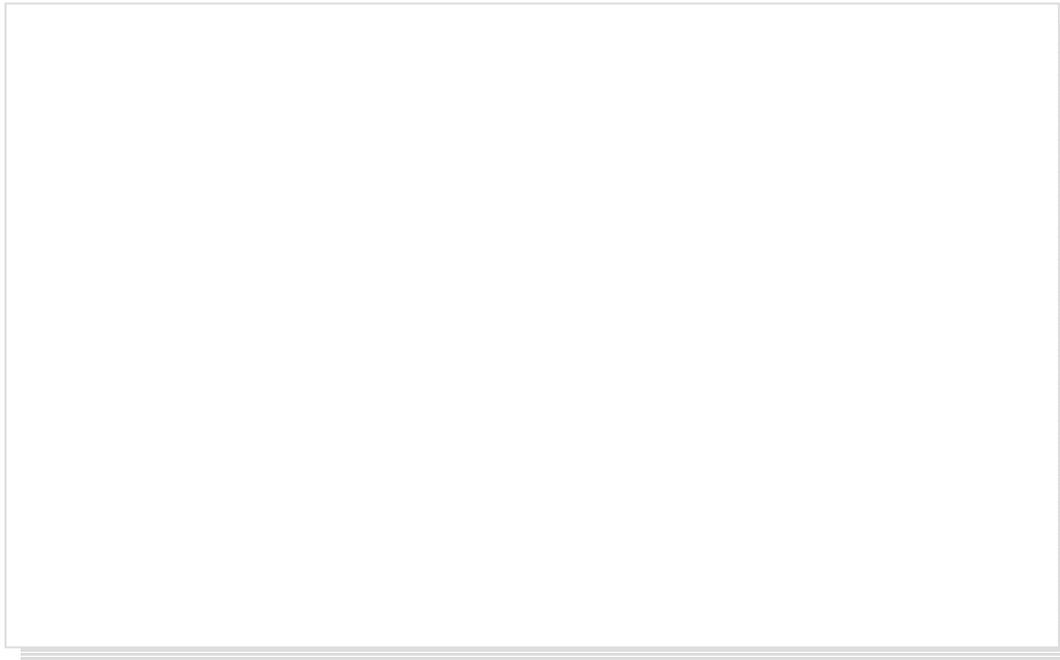
Postorder: (L, R, n)

Levelorder:

Übung

- Preorder-Traversierung: 10, 3, 1, 4, 2, 9, 7, 5, 8
- Inorder-Traversierung: 3, 4, 1, 10, 9, 7, 2, 8, 5

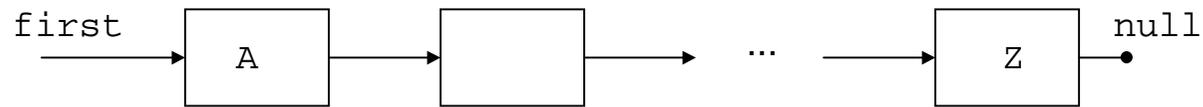
- Zeichnen und stellen Sie den Baum anhand dieser Informationen wieder her.



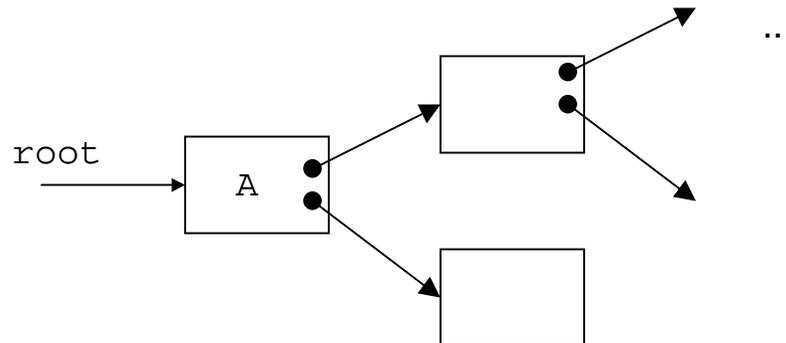
Mutationen von (sortieren) Bäumen

Übung Anfügen an Liste

- Zum Einfügen stellt man sich Bäume am einfachsten als erweiterte Listen vor



- **Übung:** Schreiben Sie eine rekursive Methode `insertAt`, die ein neues Element am Schluss einer Liste anhängt.



Einfügen unsortiert

- a) freie Stelle finden
- b) neuen Knoten einfügen

```
class BinaryTree<T> implements Tree<T>{
    private TreeNode<T> root;

    private TreeNode insertAt(TreeNode node, T x) {
        if (node == null) {
            return new TreeNode(x);
        }
        else {
            node.right = insertAt(node.right, x);
            // or
            // node.left = insertAt(node.left, x);
            return node;
        }
    }

    public void insert (T x) {
        root = insertAt(root, x);
    }
}
```

links oder rechts
einfügen

Sortierte Binärbäume

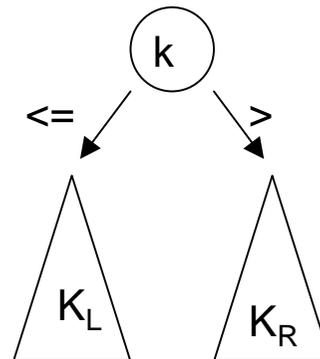
- Beim binären Suchbaum werden die Objekte anhand ihres (Schlüssel-) Werts geordnet eingefügt:
- in Java: Interface `Comparable<T>`

Definition:

Für jeden Knoten im Baum gilt die **Invariante**

im linken Unterbaum sind alle kleineren Elemente $K_L \leq^* k$

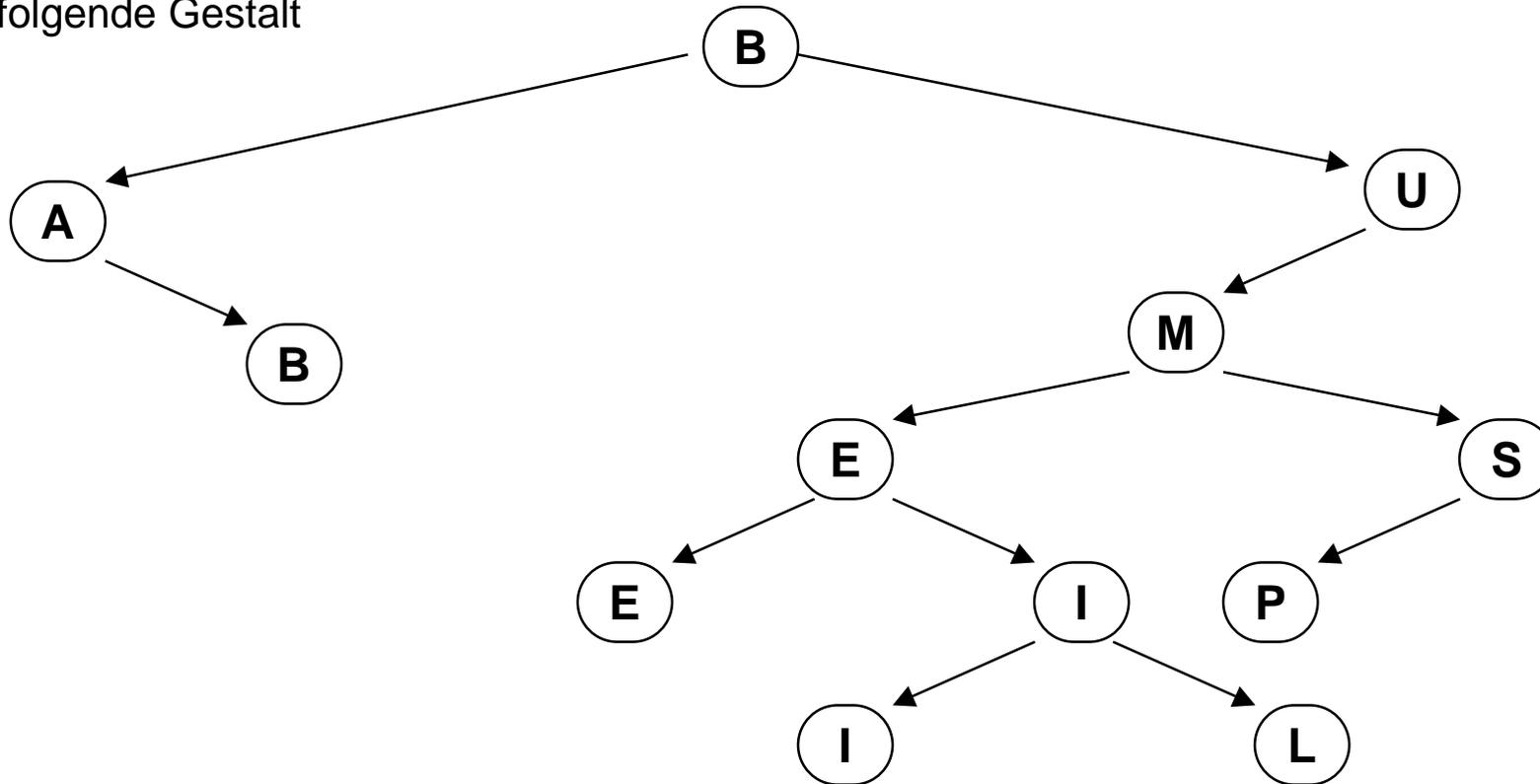
im rechten Unterbaum sind alle grösseren Elemente: $K_R >^* k$



* manchmal
auch < und >=
man kann auch
gleiche Werte im
Knoten (als
Liste) speichern

Beispiel eines sortierten Binärbaums

- Der sortierte Binärbaum hat nach dem Einfügen von "BAUMBEISPIEL" folgende Gestalt



- Numerieren Sie die Knoten entsprechend ihrer Reihenfolge beim Einfügen
- Geben Sie an, welche Zeichenkette bei einer Inorder Traversierung ausgegeben wird
- Welche Traversierung muss angewendet werden, damit die Knoten in alphabetischer Reihenfolge ausgegeben werden?
- Zeichnen Sie den Baum auf, bei dem man die Zeichenkette rückwärts eingefügt hat, d.h. L,E,I, ...
- Zeichnen sie den Baum auf, der beim Einfügen von A,B,B,E,E,I,I,L,M,P,S,U entsteht

Einfügen sortiert

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

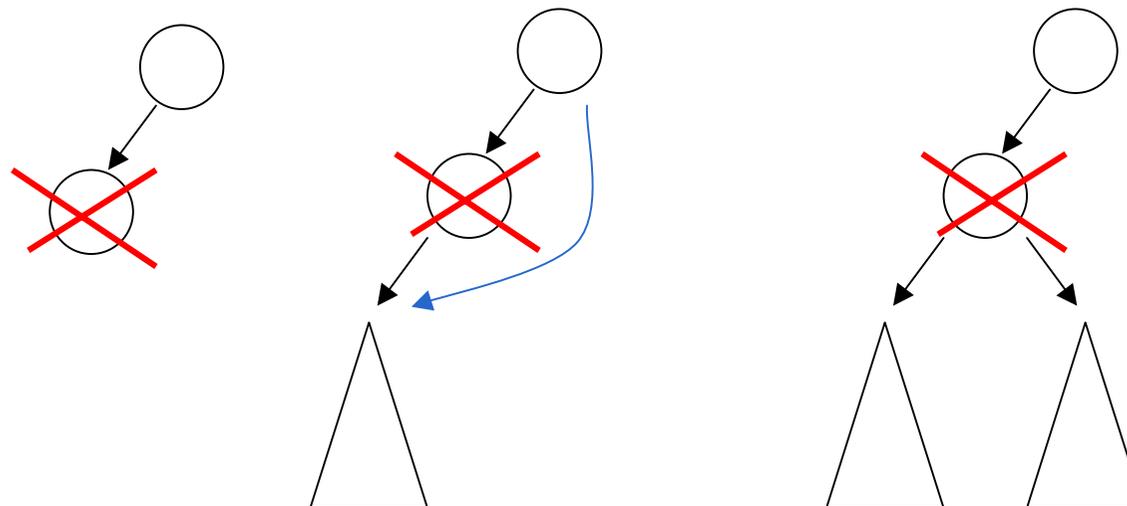
```
class BinaryTree<T extends Comparable<T>> implements Tree<T> {
    private TreeNode<T> root;

    private TreeNode<T> insertAt(TreeNode<T> node, T x) {
        if (node == null) {
            return new TreeNode(x);
        }
        else {
            if (x.compareTo(element) <= 0)
                node.left = insertAt(node.left, x);
            else
                node.right = insertAt(node.right, x);
            return node;
        }
    }

    public void add (T x) {
        root = insertAt(root, x);
    }
}
```

Löschen: einfache Fälle

- a) den zu entfernenden Knoten suchen
- b) Knoten löschen. Dabei gibt es 3 Fälle:
 - 1) der Knoten hat keinen Teilbaum \Rightarrow Knoten löschen
 - 2) der Knoten hat einen Teilbaum
 - 3) der Knoten hat zwei Teilbäume (später)



Löschen: komplizierter Fall

a) den zu entfernenden Knoten suchen

b) Knoten löschen. Dabei gibt es 3 Fälle:

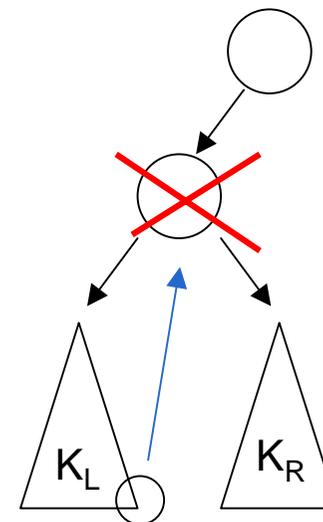
1) der Knoten hat keinen Teilbaum \Rightarrow Knoten löschen ✓

2) der Knoten hat einen Teilbaum ✓

3) der Knoten hat zwei Teilbäume

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

Lösung: der Knoten, der im linken Teilbaum ganz rechts liegt.



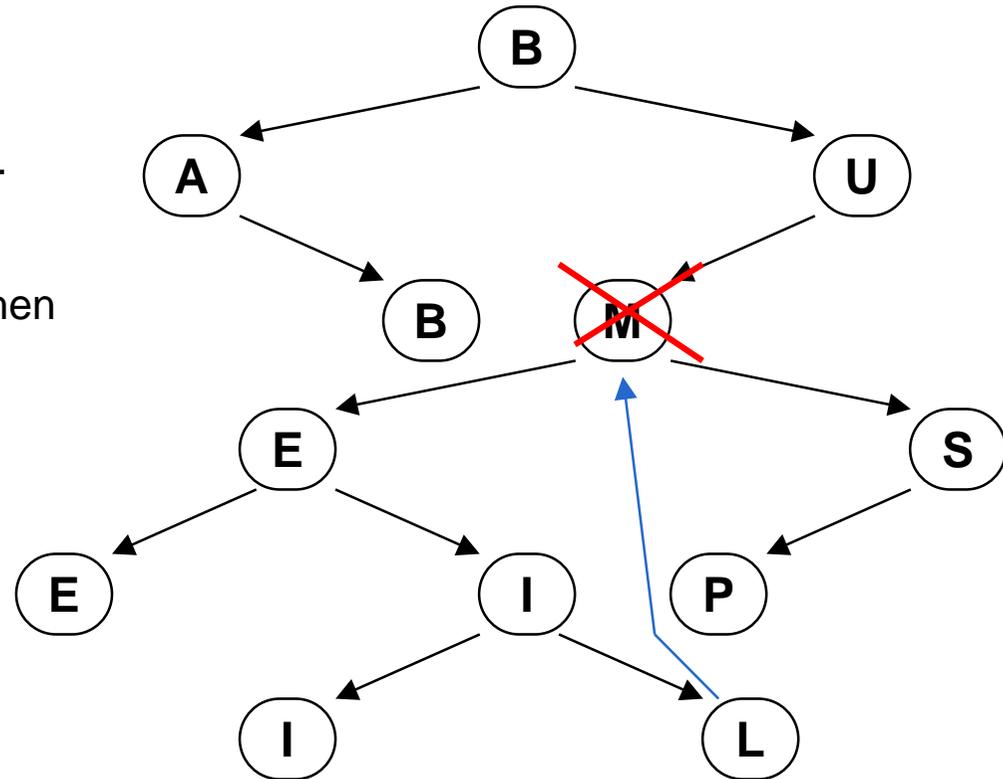
Frage: wieso dürfen wir diesen Knoten entfernen?

Löschen Beispiel

■ Es soll M gelöscht werden.

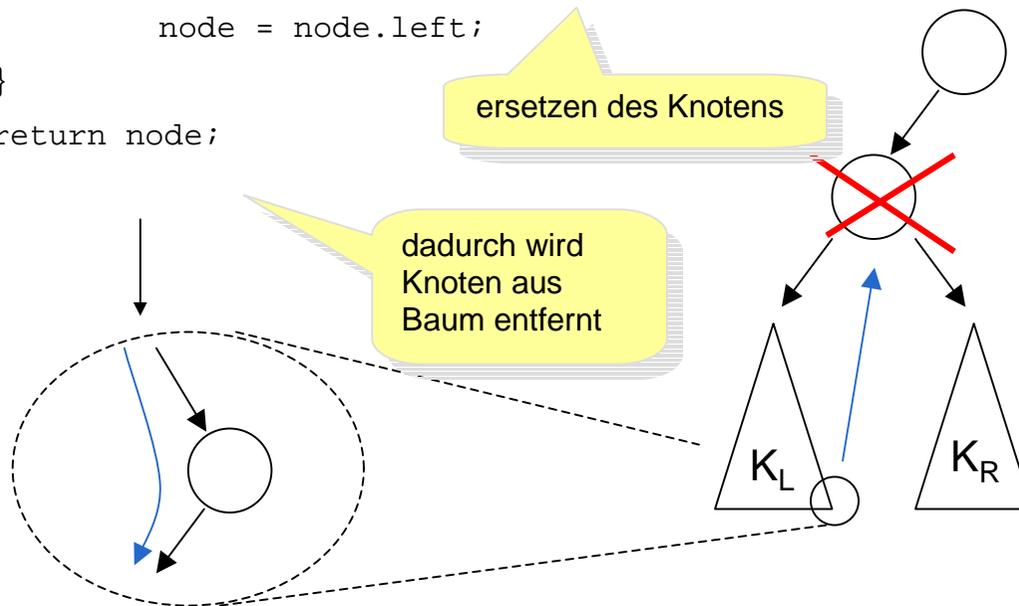
■ vom linken Teilbaum wir das Element ganz rechts als Ersatz genommen, i.e. L.

■ L kann einfach aus seiner ursprünglichen Position herausgelöst werden, da es maximal einen Nachfolger hat.



Finden von Ersatzknoten

```
// find node to replace
private TreeNode<T> findRepAt(TreeNode<T> node, TreeNode<T> rep) {
    if (node.right != null) {
        node.right = findRepAt(node.right, rep);
    } else {
        rep.elements = node.elements;
        node = node.left;
    }
    return node;
}
```



Löschen Code

```
// remove node
private TreeNode<T> removeAt(TreeNode<T> node,
                             T x,TreeNode<T> removed ) {
    if (node == null) {
        return null;
    } else {
        if (x.compareTo(node.getValue()) == 0) {
            // found
            removed.elements = node.elements;
            if (node.left == null) {
                node = node.right;
            } else if (node.right == null) {
                node = node.left;
            } else {
                node.left = findRepAt(node.left,node);
            }
        } else if (x.compareTo(node.getValue()) < 0) {
            // search left
            node.left = removeAt(node.left, x, removed);
        } else {
            // search right
            node.right = removeAt(node.right, x, removed);
        }
    }
    return node;
}
```

```
// find node to replace
private TreeNode<T> findRepAt(TreeNode<T> node,
                               TreeNode<T> rep) {
    if (node.right != null) {
        node.right = findRepAt(node.right,rep);
    } else {
        rep.elements = node.elements;
        node = node.left;
    }
    return node;
}

public T remove(T x) {
    TreeNode<T> removed = new TreeNode<T>(null);
    root = removeAt(root, x, removed);
    return removed.element;
}
```

Suche x im Baum B:

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

```
public Object search(TreeNode<T> node, T x) {  
    if (node == null) return null;  
    else if (x.compareTo(node.getValue()) == 0)  
        return node;  
    else if (x.compareTo(node.getValue()) < 0)  
        return search(node.left, x);  
    else  
        return search(node.right, x);  
}
```

- Bei einem vollen Binärbaum müssen lediglich \log_2 Schritte durchgeführt werden bis Element gefunden wird.
- Entspricht Aufwand des Binären Suchens
- sehr effizient Bsp: 1000 Elemente -> 10 Schritte

■ Allgemeine Bäume

- rekursive Definition
- Knoten (Vertex) und Kanten (Edge)
- Eigenschaften von Bäumen

■ Binärbäume: Bäume mit maximal zwei Nachfolgern

- Traversal, Visitor
- verschiedene Traversierungsarten
- Inorder, Preorder, Postorder, Levelorder

■ sortierte Binärbäume Einführung

- Einfügen
- Löschen
- Suchen