Bäume

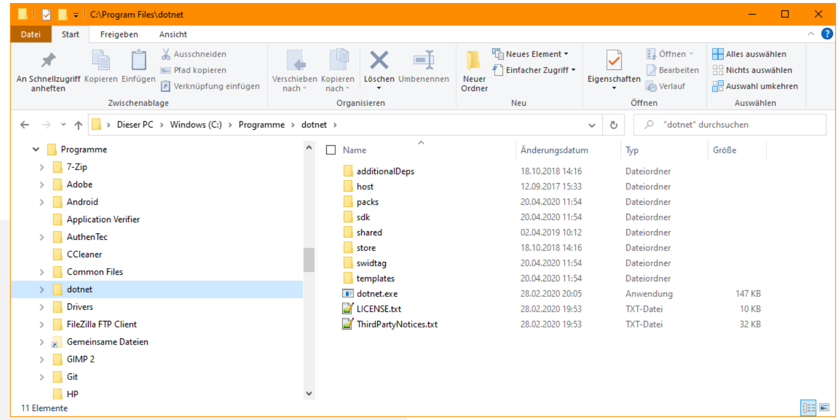
- Sie wissen, was Bäume in der Informatik sind
- Sie kennen das Besucher-Entwurfsmuster
- Sie kennen Binärbäume
- Sie können die Bäume auf unterschiedliche Arten traversieren
- Sie wissen, wie man in Binärbäumen Elemente löscht

Basiert auf Material von:

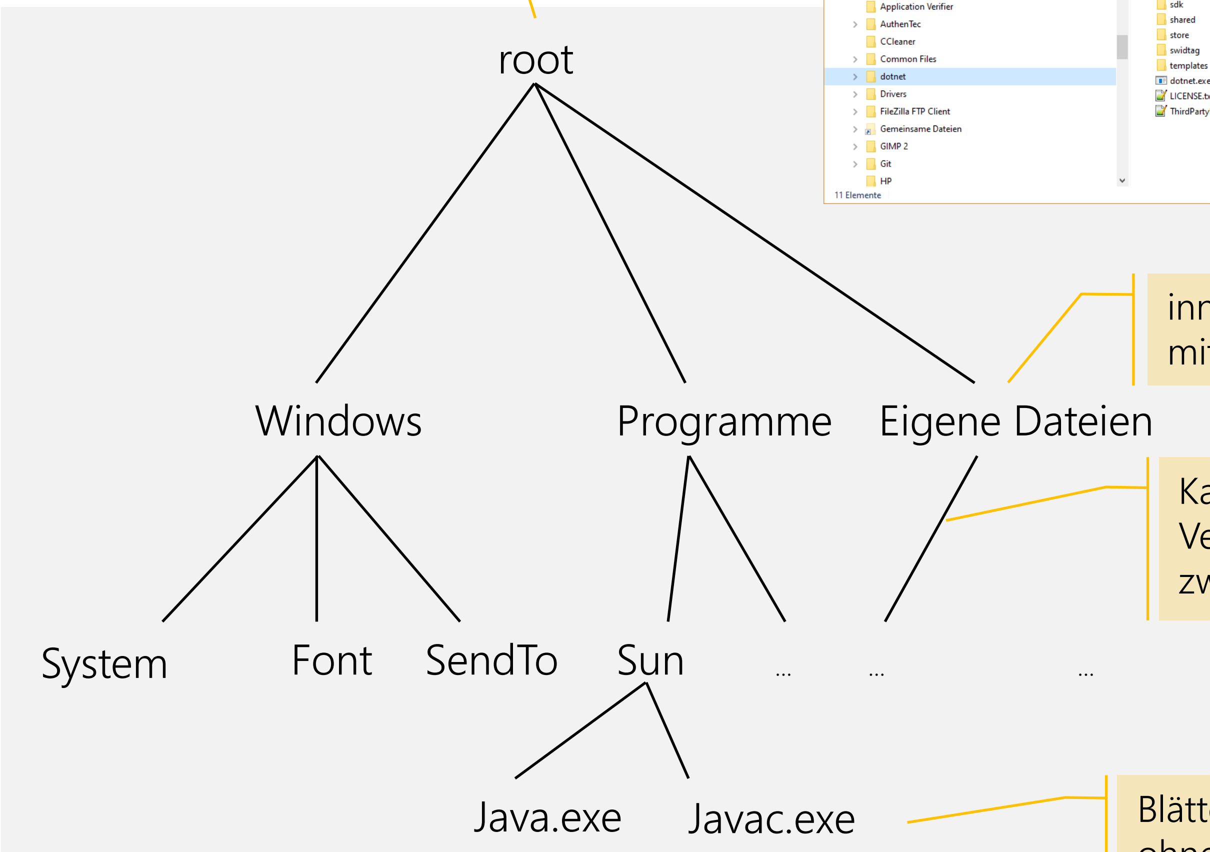
Kurt Bleisch
Stephan Neuhaus
Karl Rege
Marcela Ruiz
Jürgen Spielberger

Bäume, Anwendung und Begriffe

Beispiel 1: Dateisystem



Wurzel: der Ursprung des Baumes



innerer Knoten: Knoten mit Nachfolger

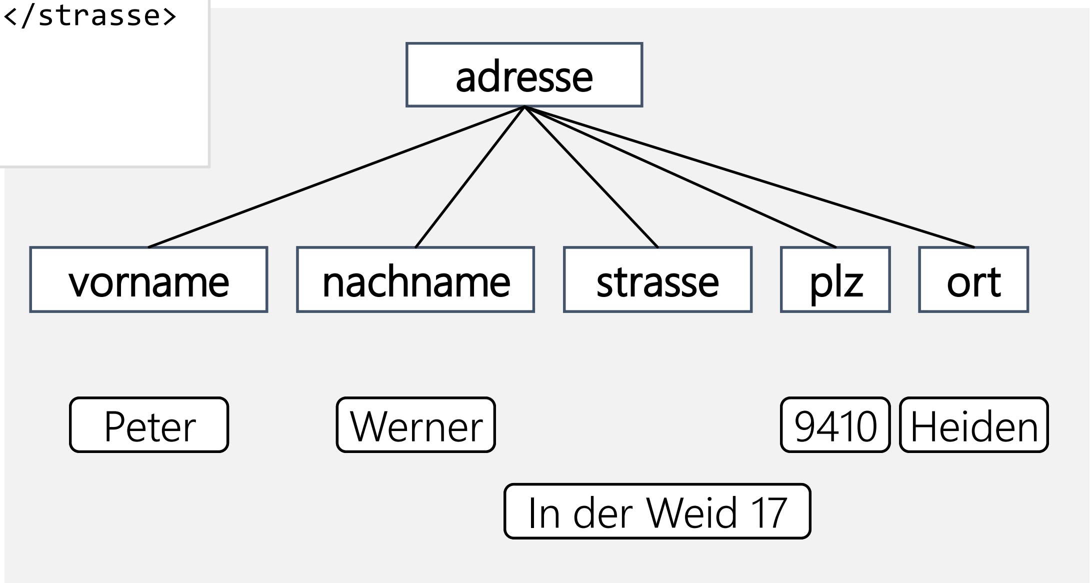
Kante: gerichtete Verbindung zwischen Knoten

Blätter: Knoten ohne Nachfolger

Beispiel 2: XML- Dokument

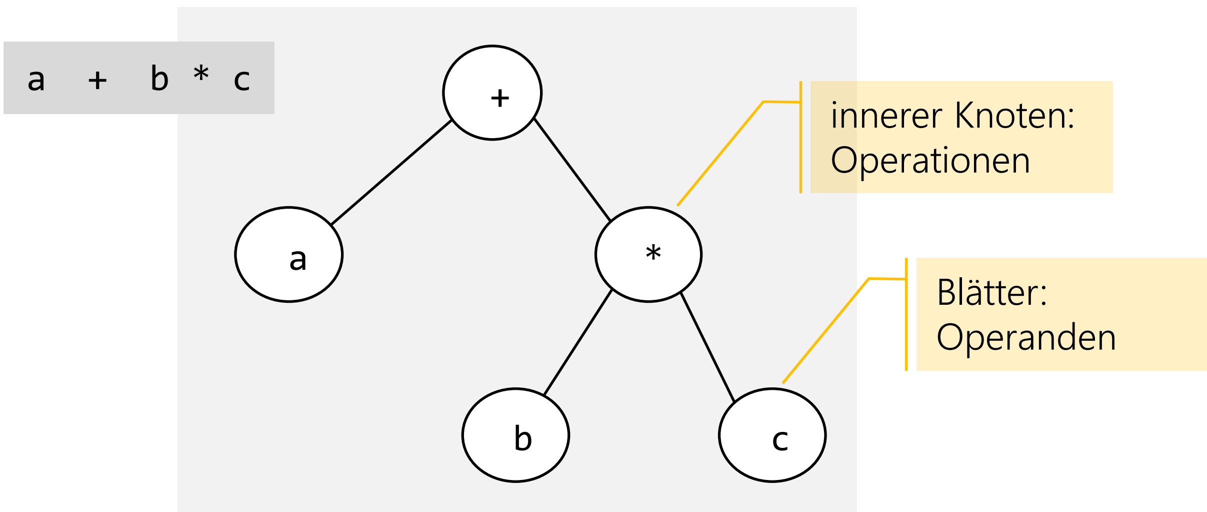
Ein XML Dokument besteht aus einem Wurzelelement, an dem beliebig viele Nachfolgeelemente angehängt sind, an denen wiederum Nachfolgeelemente hängen können.

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



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)

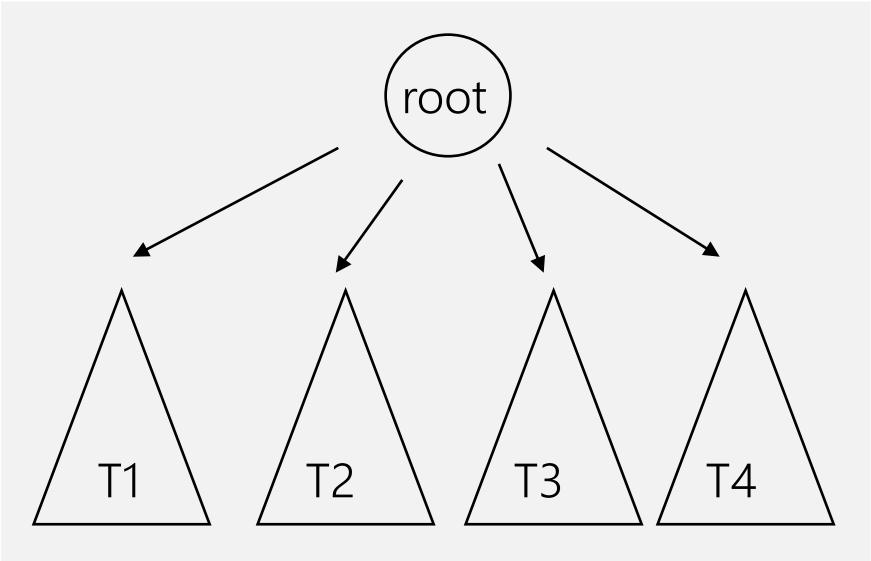
ein Baum ist **leer**

oder

er besteht aus einem Knoten mit keinem, einem oder mehreren disjunkten Teilbäumen T1 , T2 , ... Tk .

Baum = leer

Baum = Knoten (Baum)*

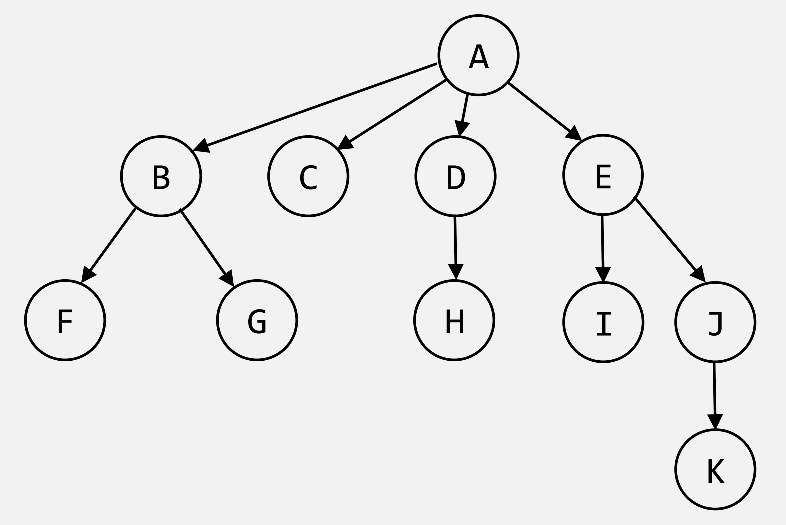


Definition Baum (nicht rekursiv)

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

Hinweis:

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



Eigenschaften und Begriffe

- Alle Knoten ausser der Wurzel (root) sind Nachfolger (descendant, child) genau eines Vorgänger-Knotens (ancestor, parent).
- Knoten mit Nachfolger werden als innere 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 Höhe (oder Tiefe) eines Baumes gibt an, wie viele «Ebenen» der Baum hat: Anzahl Kanten + 1.
- Das Gewicht ist die Anzahl der Knoten des (Teil-)Baumes.

Übung

Wurzelknoten =

Höhe/Tiefe =

Gewicht =

Nachfolger von B =

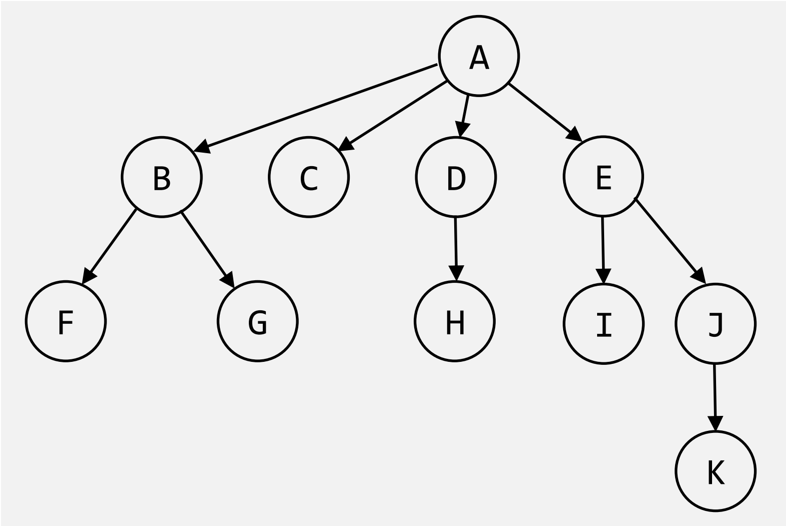
Nachfolger von A =

Vorgänger von K =

Blattknoten =

Geschwister von C =

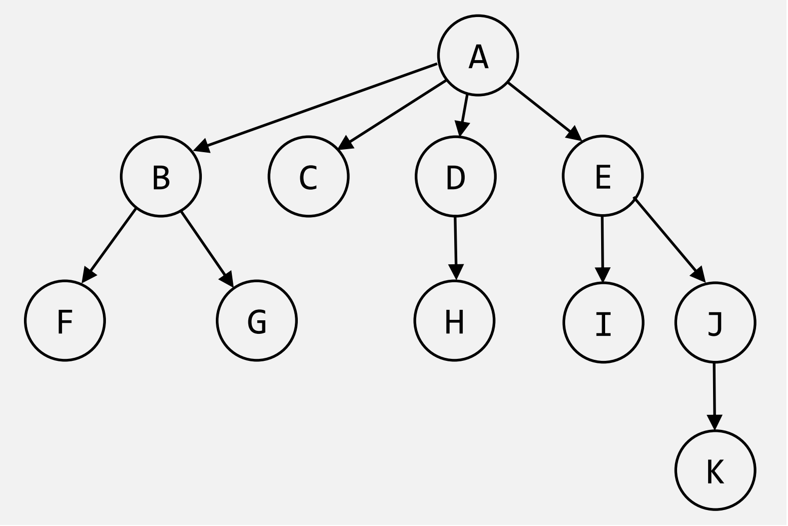
Geschwister von H =



Ein Knoten hat direkte Vorgänger-Knoten.

Ein Knoten kann viele direkte Nachfolger haben[1N)].

Knoten Implementation




```
class TreeNode<T> {
    T element;
    List<TreeNode<T>> nodes;

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

- Jeder Knoten hat Zeiger auf jeden Nachfolger z.B. in Array gespeichert. Falls es sehr wenige sind (z.B. immer max. 2), direkt in Attributen gespeichert.
- Allenfalls kann die Zahl der Nachfolger pro Knoten stark variieren und ist im Voraus nicht bekannt → Array nicht effizient.
  Mögliche bessere Lösung: Zeiger in Liste verwalten.

Zürcher Hochschule
für Angewandte Wissenschaften

School of Engineering

Binärbaum

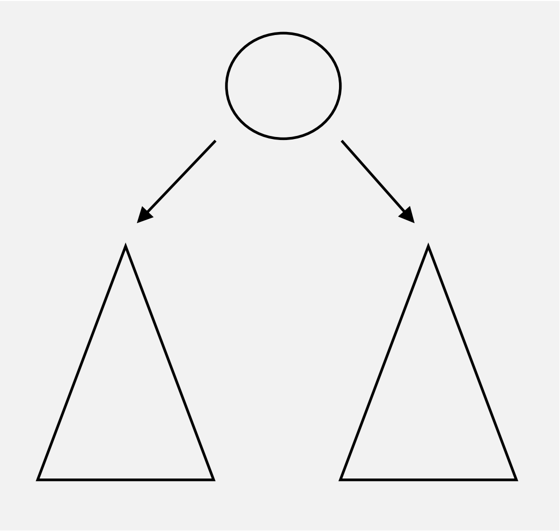
Binärbaum

Dies ist die am häufigsten verwendete Art von Bäumen. Beim Binärbaum hat ein Knoten maximal 2 Nachfolger.

Definition (rekursiv):

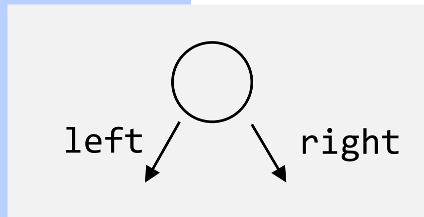
Ein Binärbaum ist entweder leer, oder besteht aus einem Wurzel-Knoten und aus einem linken und einem rechten, disjunkten Teilbaum.

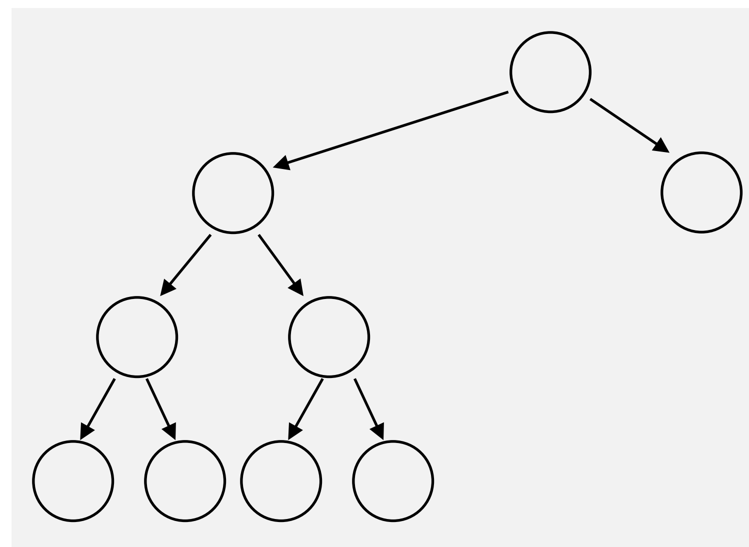
Baum = leer
Baum = Knoten (Baum Baum)



Binärbaum: Datenstruktur

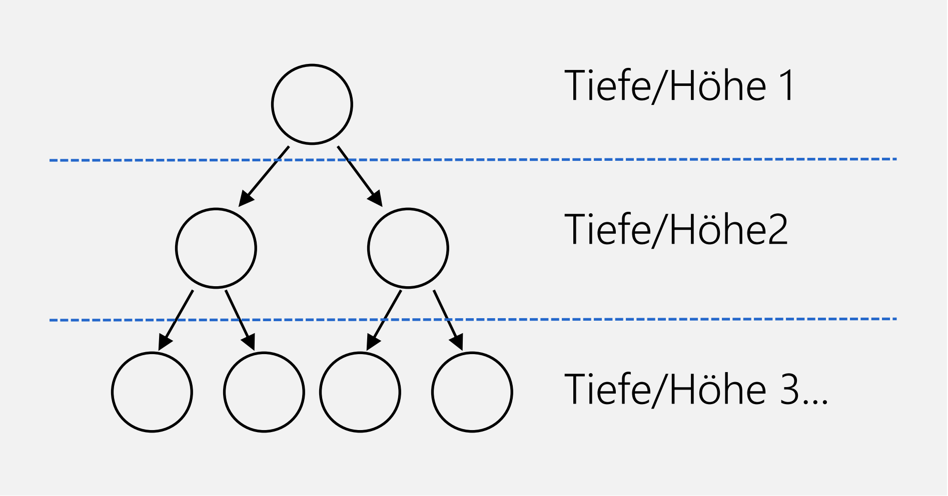
```
class TreeNode<T> {
    T element;
    TreeNode<T> left;
    TreeNode<T> right;

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





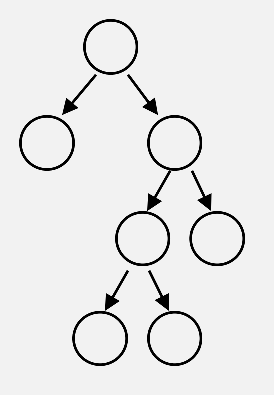
Binärbaum: Eigenschaften

- Tiefe/Höhe: k
- auf jeder Höhe: $2^{(k-1)}$ Knoten
- Maximal Anzahl Knoten: $2^k - 1$

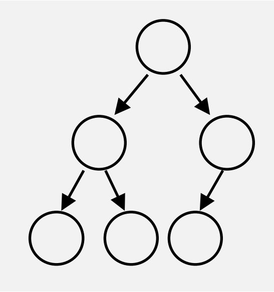


Binärbaum: Eigenschaften

- Ein Binärbaum heisst voll (Engl. full), wenn jeder Knoten entweder Blatt ist, oder zwei Kinder besitzt, es also keine «Halbblätter» gibt.

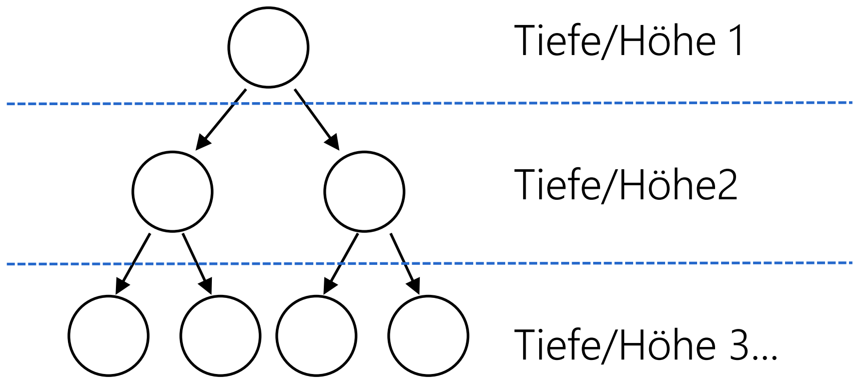


- Ein Binärbaum heisst vollständig (Engl. complete, daher manchmal auch als komplett bezeichnet), wenn alle Blätter dieselbe Höhe haben und die Blätter linksbündig angeordnet sind.



Binärbaum: Übung

Aufgabe: Leiten Sie eine Formel für die Höhe h für einen vollständigen Binärbaum her und bestimmen Sie die Höhe mit n = 37 Knoten (Hinweis: $\log_a n = \log n / \log a$).



Traversierung

Traversieren: Ausgeben aller Elemente

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering

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

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

Hinweis: Ausgeben einer Liste mit einer rekursiven Methode:

Rekursive Algorithmen eignen sich hervorragend für Bäume.

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

```
void printTree(TreeNode node) {
    if (node.left != null) printTree(node.left);
    System.out.println(node.element);
    if (node.right != null) printTree(node.right);
}
```

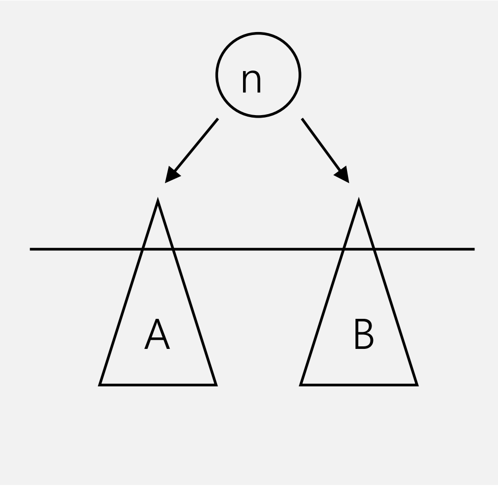
Traversieren

Das (ev. 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:

1. Preorder: n, A, B
2. Inorder: A, n, B
3. Postorder: A, B, n
4. Levelorder: n, a_0, b_0, a_1, a_2, b_1, b_2, ...



Traversieren: Visitor Pattern

Problem:

Wie kann ich den Baum traversieren (z.B. in Preorder) und während der Traversierung **verschiedene Aktionen je Knoten** ausführen, ohne dass ich die Klasse der Traversierung oder die Klasse des Baums erweitern muss?

Diese enthalten z.B. Kundenaufträge (Order), für welche ich Rechnungen erstellen möchte.

Idee:

Wir lagern die Methoden zum Verarbeiten der Knotendaten in eine eigene Klasse (die Visitor-Klasse) aus und rufen während der Traversierung diese jeweils auf.

Vorteile:

- Es braucht auch für verschiedenste Aktionen, die beim Traversieren mit einem Knoten gemacht werden sollen, nur eine Traversierungslogik und nur einen Baum.
- Muss die Verarbeitungslogik geändert werden, so geschieht das unabhängig vom Baum und der Traversierung (normalerweise ist die Logik des Besucher-Objekts «volatiler» als die Logik des besuchten Objekts).

Visitor-Pattern

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering

```java
interface Traversal<T> {
   void preorder(TreeNode<T> node,
      Visitor<T> visitor);
   ...
}
```

Eigentlicher Name im Pattern: Visitable

```java
class TreeTraversal<T> implements Traversal<T> {

   public void preorder(TreeNode<T> node,
      Visitor<T> visitor) {
      if (node != null) {
         visitor.visit(node.element);
         preorder(node.left,visitor);
         preorder(node.right,visitor);
      }
   }
   ...
}
```

```java
interface Visitor<T> {

   void visit(T element);

}
```

```java
class MyVisitor implements Visitor<Order>{

    @Override
    public void visit(Order element) {
       ... // konkrete Verarbeitung des Auftrags
    }

}
```

```java
interface Tree<T> {

   TreeTraversal<T> traversal();
}
```

```java
class BinaryTree<T> implements Tree<T> {

   public TreeNode<T> root;

   @Override
   public TreeTraversal<T> traversal() {
      return new TreeTraversal<T>();
   }

}
```

Element im Baum sind Aufträge.

```java
...
BinaryTree<Order> ordersTree =
   new BinaryTree<>();
... // hier wird der Baum gefüllt
MyVisitor myVisitor = new MyVisitor();
TreeTraversal<Order> treeTraversal =
   ordersTree.traversal();
treeTraversal.preorder(ordersTree.root,
   myVisitor);
...
```

Legt den Visitor für die Traversierung fest.

Traversieren: Implementation Preorder (n, A, B)

1. Besuche die Wurzel und verarbeite die Daten.
2. Traversiere den linken Teilbaum (in Preorder).
3. Traversiere den rechten Teilbaum (in Preorder).

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

```
...
treeTraversal.preorder(ordersTree.root, myVisitor);
...
```

Traversieren: Implementation Postorder (A, B, n)

1. Traversiere den linken Teilbaum (in Postorder).
2. Traversiere den rechten Teilbaum (in Postorder).
3. Besuche die Wurzel und verarbeite die Daten.

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

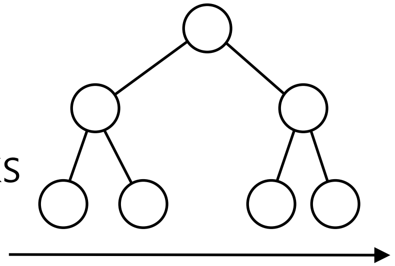
- Zuerst werden die Nachfolger abgearbeitet und dann der Knoten selbst (von unten nach oben).

Traversieren: Implementation Inorder (A, n, B)

1. Traversiere den linken Teilbaum (in Inorder).
2. Besuche die Wurzel und verarbeite die Daten.
3. Traversiere den rechten Teilbaum (in Inorder).

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

Der Baum wird quasi von links nach rechts abgearbeitet:



Traversieren: Implementation Preorder (n, A, B) mit Stack

- Implementierung der Preorder-Verarbeitung ohne Rekursion.
- 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.element);
      if (node.right != null) s.push(node.right);
      if (node.left !=null) s.push(node.left);
   }
}
```

Der rechte Teilbaum soll «unten» auf dem Stack liegen.

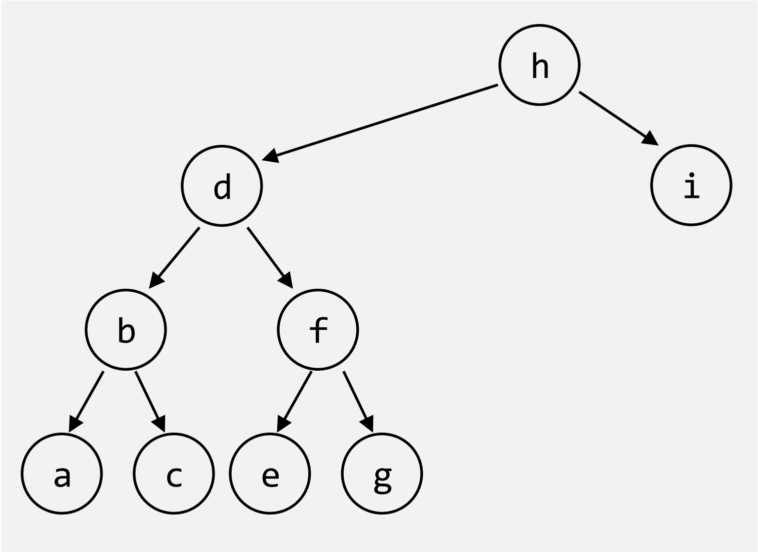
Traversieren: Implementation Levelorder (n, a_0, b_0, a_1, a_2, b_1, b_2, … mit Queue

1. zuerst die Wurzel,
2. dann die Wurzel des linken und rechten Teilbaumes,
3. 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.element);
      if (node.left !=null) q.enqueue(node.left);
      if (node.right != null) q.enqueue(node.right);
   }
}
```

Übung: Traversieren

Zeigen Sie die Reihenfolge bei den verschiedenen Traversierungsarten auf:



Preorder:	n, L, R	h, d, b, a, c, f, e, g, i
Inorder:	L, n, R	a, b, c, d, e, f, g, h, i
Postorder:	L, R , n	a, c, b, e, g, f, d, i, h
Levelorder:	h, d, i, b, f, a, c, e, g	

Übung: Traversieren

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

Stellen Sie den Baum anhand dieser Informationen wieder her:

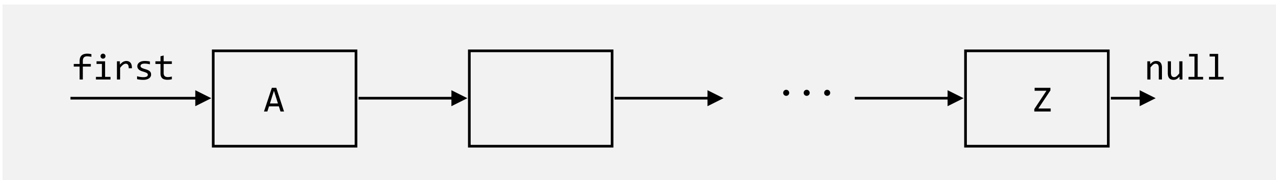
Zürcher Hochschule
für Angewandte Wissenschaften

School of Engineering

Mutationen von (sortierten) Bäumen

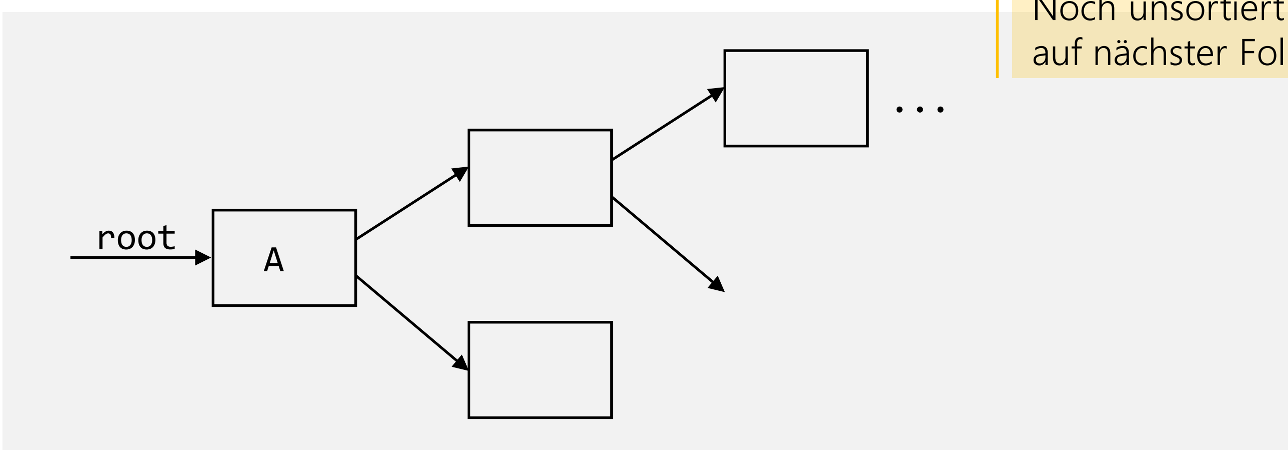
Übung: Einfügen unsortiert

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



- Schreiben Sie eine rekursive Methode `insertAt(T x)`, die ein neues Element am Schluss einer Liste anhängt.

Noch unsortiert (Lösung auf nächster Folie)



Übung: Einfügen unsortiert

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

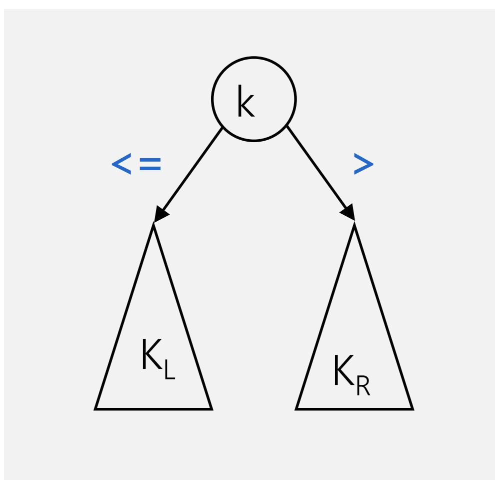
Sortierte Binärbäume

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

Für jeden Knoten gilt:

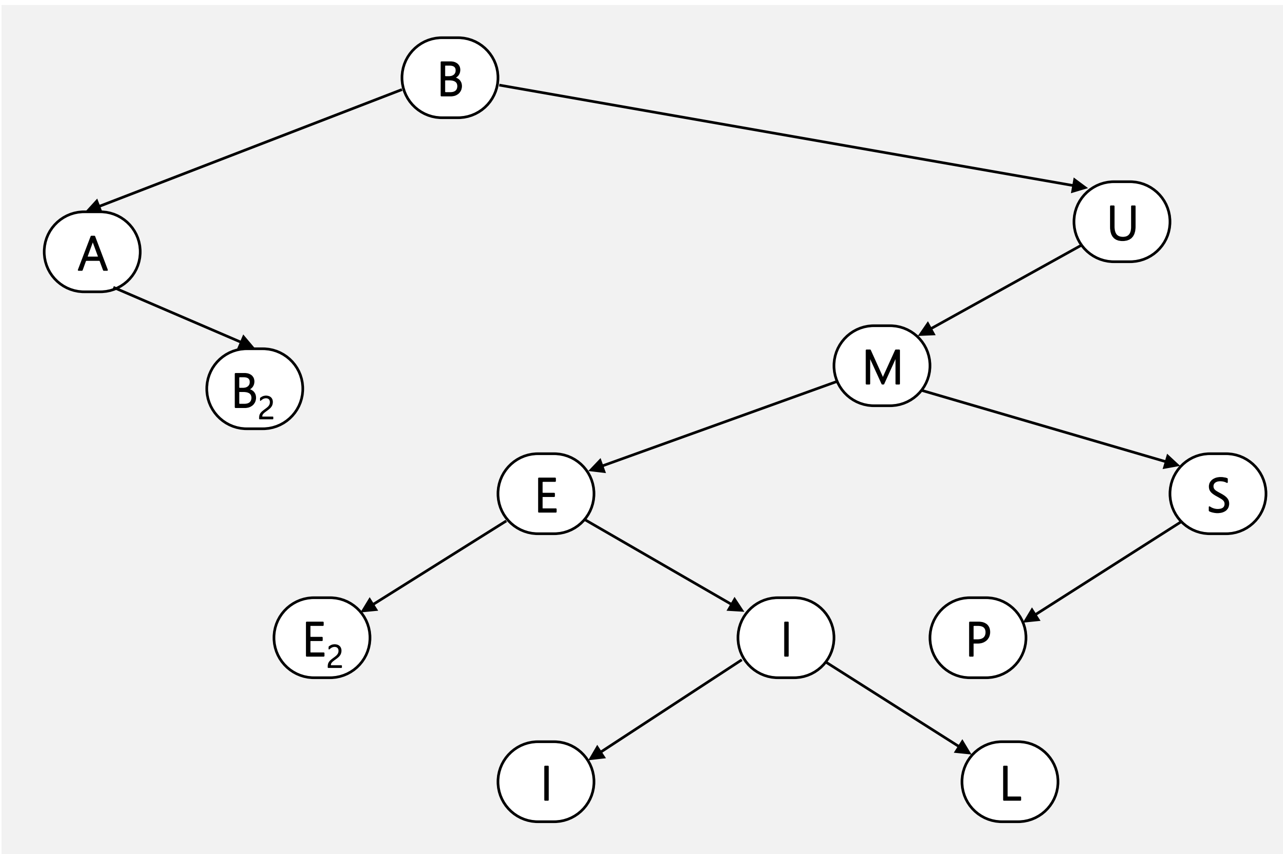
- im linken Unterbaum sind alle kleineren Elemente KL <=* k
- im rechten Unterbaum sind alle grösseren Elemente: KR >* k

* manchmal auch < und >=



Sortierte Binärbäume: Beispiel

Der sortierte Binärbaum hat nach dem Einfügen von «BAUMBEISPIEL» folgende Gestalt:



Sortierte Binärbäume: Suchen rekursiv

Zürcher Hochschule
für Angewandte Wissenschaften
zh
aw
School of
Engineering

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

- Bei einem vollständigen (resp. kompletten) Binärbaum müssen lediglich Log_2 Schritte durchgeführt werden bis Element gefunden wird.
- Entspricht dem Aufwand des binären Suchens.
- sehr effizient, Bsp.: 1000 Elemente → 10 Schritte.

Sortierte Binärbäume: Einfügen rekursiv

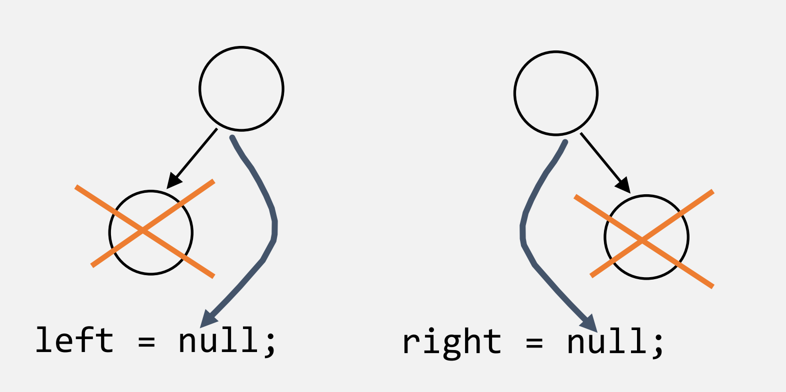
Beim Einfügen muss links eingefügt werden, wenn das neue Element kleiner oder gleich ist, sonst rechts.

```
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;
   }
}
```

Sortierte Binärbäume: Löschen

Einfacher Fall (2.1):

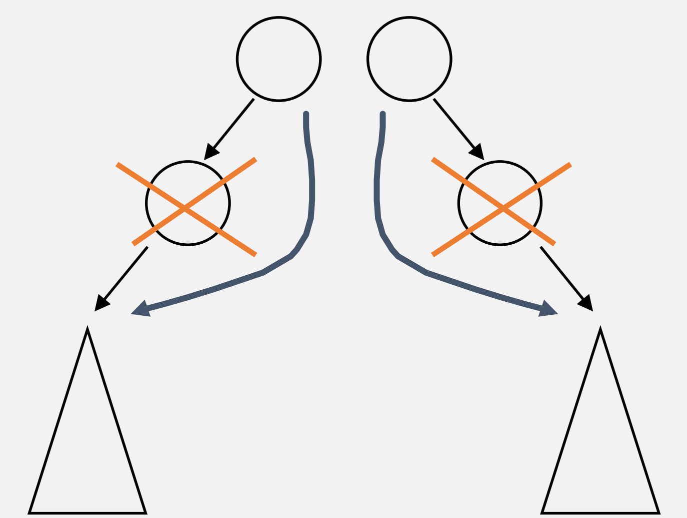
1. den zu entfernenden Knoten suchen
2. Knoten löschen. Dabei gibt es 3 Fälle:
   1. Fall: der zu löschende Knoten hat keinen Teilbaum → Knoten löschen



Sortierte Binärbäume: Löschen

Einfache Fälle (2.1 und 2.2):

1. den zu entfernenden Knoten suchen
2. Knoten löschen. Dabei gibt es 3 Fälle:
   1. Fall: der Knoten hat keinen Teilbaum → Knoten löschen
   2. Fall: der Knoten hat einen Teilbaum → Knoten löschen und Referenz neu setzen



```
left = left.left; right = right.right;
```

Der 1. Fall kann als Spezialfall des 2. Falls implementiert werden, der zu verknüpfende Teilbaum ist dann einfach leer (null-Pointer).

Sortierte Binärbäume: Löschen

Komplizierter Fall (2.3):

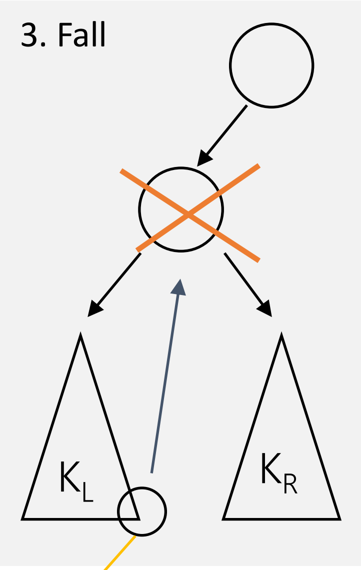
1. den zu entfernenden Knoten suchen
2. Knoten löschen. Dabei gibt es 3 Fälle:
   1. Fall: der Knoten hat keinen Teilbaum → Knoten löschen
   2. Fall: der Knoten hat einen Teilbaum → Knoten löschen und Referenz neu setzen
   3. Fall: der Knoten hat zwei Teilbäume → Es muss ein Ersatzknoten oder Ersatzwert gefunden werden (man kann den Knoten austauschen, oder den Inhalt des zu löschenden Knotens ersetzen).

Wir werden im Folgenden den Inhalt ersetzen.

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

Lösung 1: der Knoten, der im linken Teilbaum ganz rechts liegt.

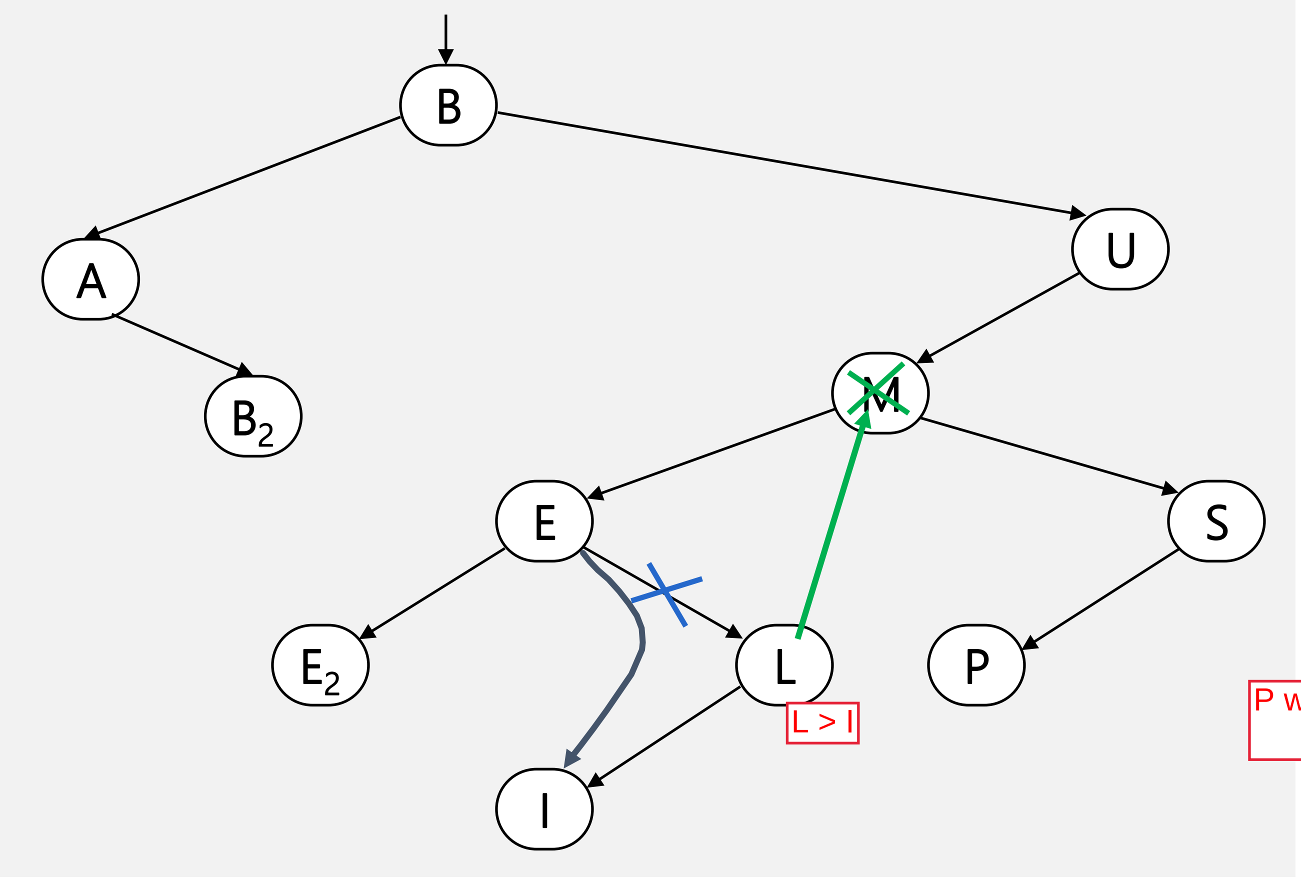
Lösung 2: der Knoten, der im rechten Teilbaum ganz links liegt.



Frage: wieso dürfen wir diesen Knoten entfernen?

Sortierte Binärbäume: Löschen Beispiel

- Es soll M gelöscht werden.
- Vom linken Teilbaum wird das Element ganz rechts als Ersatz genommen: L
- L kann einfach aus seiner ursprünglichen Position 'herausgelöst' werden, da es maximal einen Nachfolger hat.



P würde auch gehen

Ablauf:

1. Inhalt des Knotens M mit Inhalt des Knotens von L ersetzen.
2. `right` von E mit `left` von L ersetzen.

Sortierte Binärbäume: Löschen Algorithmus

Die rekursive Methode `removeAt` sucht den zu löschenden Knoten (mit Inhalt = x) und gibt dem aufrufenden Objekt als Antwort zurück, ob und wie die Referenz geändert werden muss (abhängig vom Fall 1, 2 oder 3).

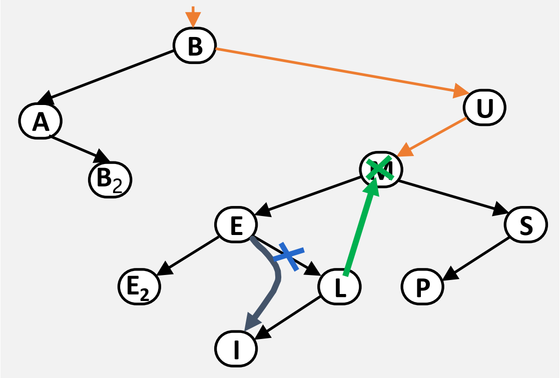
```
if (root != null) root = removeAt(root, x);
```

```
private TreeNode<T> removeAt(TreeNode<T> node, T x) {
   if ([Verankerung: zu löschender Knoten gefunden]) {
      ... // delete this node
   } else if (x.compareTo(node.element) < 0) {
      node.left = removeAt(node.left, x);  // search in left subtree
   } else {
      node.right = removeAt(node.right, x); // search in right subtree
   }
   return node;
}
```

Sortierte Binärbäume: Löschen Beispiel

```
if (root != null) root = removeAt(root, x);
```

```
private TreeNode<T> removeAt(TreeNode<T> node, T x) {
   if ([Verankerung: zu löschender Knoten gefunden]) {
      ... // delete this node
   } else if (x.compareTo(node.element) < 0) {
      node = removeAt(node.left, x);  // search in left subtree
   } else {
      node = removeAt(node.right, x); // search in right subtree
   }
   return node;
}
```



Call Stack:

```
removeAt(node: B (Root), x: M)
removeAt(node: U (B.right), x: M)
removeAt(node: M (U.left), x: M)
-> found M
-> case 3 for node M
```

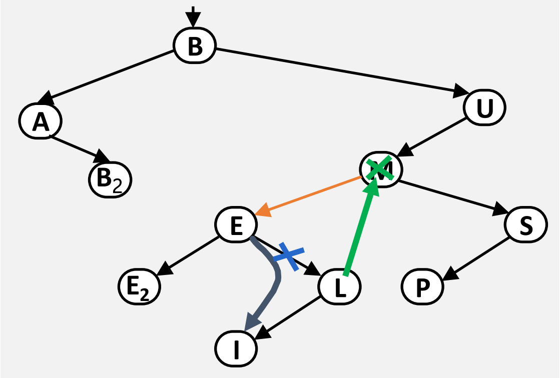
Sortierte Binärbäume: Löschen Algorithmus

```
private TreeNode<T> removeAt(TreeNode<T> node, T x) {
   if (x.compareTo(node.element) == 0) { // delete this node
      if (node.left == null) {
         node = node.right; // no left subtree -> case 1 or 2
      } else if (node.right == null) {
         node = node.left; // no right subtree -> case 2
      } else {
         // two subtrees -> case 3
         node.left = findRepAt(node.left, node); // node.left is root of left subtree
      }
   } else if (x.compareTo(node.element) < 0) {
      node.left = removeAt(node.left, x);  // search in left subtree
   } else {
      node.right = removeAt(node.right, x); // search in right subtree
   }
   return node;
}
```

Übernächste Folie

Sortierte Binärbäume: Löschen Beispiel

```
private TreeNode<T> removeAt(TreeNode<T> node, T x) {
   if (x.compareTo(node.element) == 0) { // delete this node
      if (node.left == null) {
         node = node.right; // no left subtree -> case 1 or 2
      } else if (node.right == null) {
         node = node.left; // no right subtree -> case 2
      } else {
         // two subtrees -> case 3
         // node.left is root of left subtree
         node.left = findRepAt(node.left, node);
      }
   } else if (x.compareTo(node.element) < 0) {
      ...
   }
   return node;
}
```



Call Stack:

```
removeAt(node: B (Root), x: M)
removeAt(node: U (B.right), x: M)
removeAt(node: M (U.left), x: M)
-> found M
-> case 3 for node M
findRepAt(node: E (M.left), rep: M)
```

Sortierte Binärbäume: Löschen Algorithmus

- Die rekursive Methode `findRepAt` sucht den Ersatzknoten (für den Fall 3) und entfernt diesen aus dem Baum.

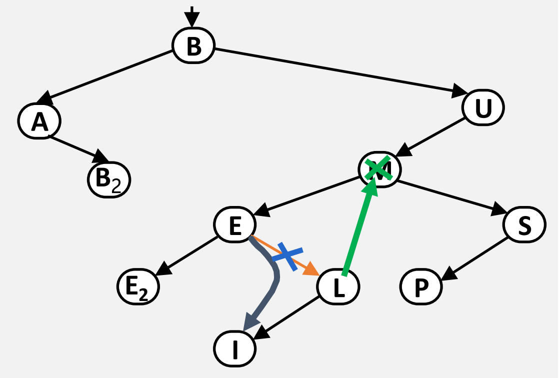
Zu durchsuchender Teilbaum

Zu 'löschender' Knoten

```
private  TreeNode<T> findRepAt(TreeNode<T> node, TreeNode<T> rep) {
   if ([Verankerung: Ersatznoten gefunden]) {
      ... // this is the node that replaces the node to delete
   )
   } else {
      // more nodes on the right side of left subtree
      node.right = findRepAt(node.right, rep);
   }
   return node;
}
```

Sortierte Binärbäume: Löschen Beispiel

```
private void findRepAt(TreeNode<T> node, TreeNode<T> rep) {
   if ([Verankerung: Ersatznoten gefunden]) {
      ... // this is the node that replaces the node to delete
   )
   } else {
      // more nodes on the right side of left subtree
      node.right = findRepAt(node.right, rep);
   }
   return node;
}
```



Call Stack:

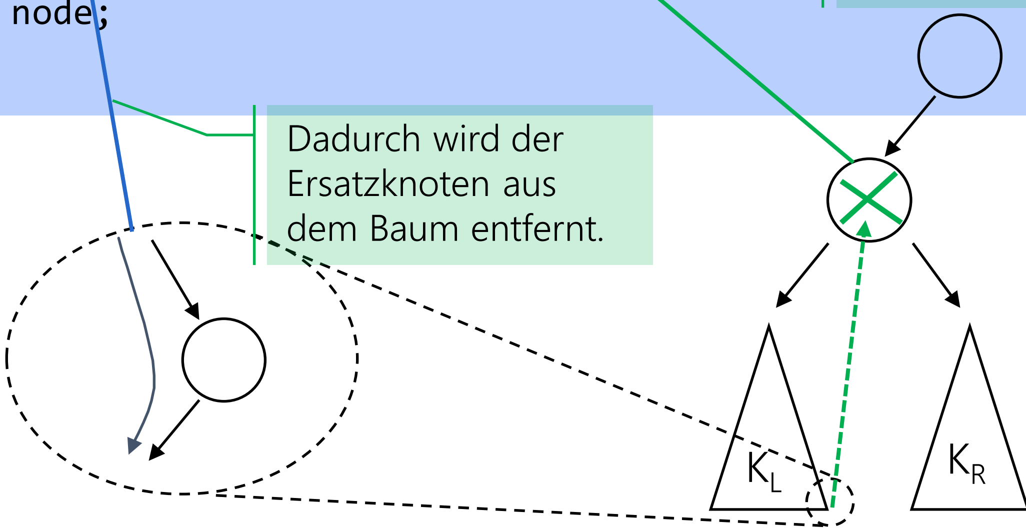
```
removeAt(node: B (Root), x: M)
removeAt(node: U (B.right), x: M)
removeAt(node: M (U.left), x: M)
-> found M
-> case 3 for node M
findRepAt(node: E (M.left), rep: M)
findRepAt(node: L (E.right), rep: M)
-> Rightmost node: L
```

Sortierte Binärbäume: Löschen Algorithmus

```
private TreeNode<T> findRepAt(TreeNode<T> node, TreeNode<T> rep) {
   if (node.right == null) {
      // node is the rightmost node, the node that should be replaced gets its element
      rep.element = node.element;
      // remove rightmost node of left subtree (return value is the 'new' node)
      node = node.left;
   } else {
      // more nodes on the right side of left subtree
      node.right = findRepAt(node.right, rep);
   }
   return node;
}
```

Der zu löschende Knoten erhält den Wert des Ersatzknotens.

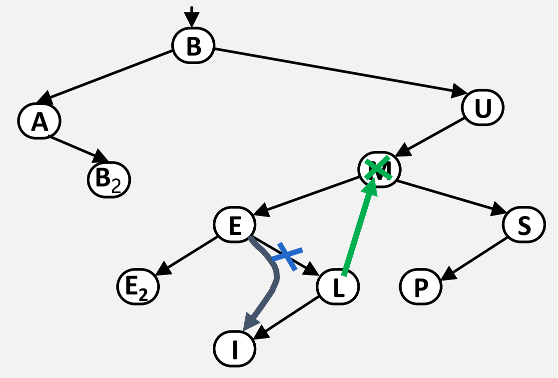
Dadurch wird der Ersatzknoten aus dem Baum entfernt.



Sucht den Ersatzknoten und 'löscht' den zu löschenden Knoten.

Sortierte Binärbäume: Löschen Beispiel

```
private void findRepAt(TreeNode<T> node, TreeNode<T> rep) {
   if (node.right == null) {
      // node is the rightmost node, the node that should be replaced
      // gets its element
      rep.element = node.element;
      // remove rightmost node of left subtree
      node = node.left;
   } else {
      // more nodes on the right side of left subtree
      node.right = findRepAt(node.right, rep);
   }
   return node;
}
```



Call Stack:

```
removeAt(node: B (Root), x: M)
removeAt(node: U (B.right), x: M)
removeAt(node: M (U.left), x: M)
-> found M
-> case 3 for node M
findRepAt(node: E (M.left), rep: M)
findRepAt(node: L (E.right), rep: M)
-> Rightmost node: L
Replace value of M with value of: L
--------------------------------------------
return from findRepAt: E.right = I
return from findRepAt: L.left = E // this is the moved L!
return from removeAt: U.left = L
return from removeAt: B.right = U
return from removeAt: root = B
```

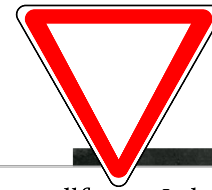
Jetzt gehen wir die Call-Hierarchie zurück

Zusammenfassung

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

- Binärbäume: Bäume mit maximal zwei Nachfo
  - Traversal, Visitor
  - verschiedene Traversierungsarten
  - Inorder, Preorder, Postorder, Levelorder

- sortierte Binärbäume Einführung
  - Suchen
  - Einfügen
  - Löschen



Kontrollfragen Lektion 5 nicht vergessen – heute mit Hans im Glück

Traversieren:
Aufruf als anonyme Klasse

```
Interface Traversal<T> {
   preorder(TreeNode<T> node, Visitor<T> visitor);
   ...
}
```

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

// start visit:
ordersTree.traversal().preorder(ordersTree.root, new MyVisitor());
```

Kann auch als anonyme Klasse 'inline' implementiert werden:

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

Traversieren:
Aufruf mittels Lambda-Ausdruck

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

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

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

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