Einführung in die Informatik I
Bäume

Prof. Bernd Brügge, Ph.D
Technische Universität München

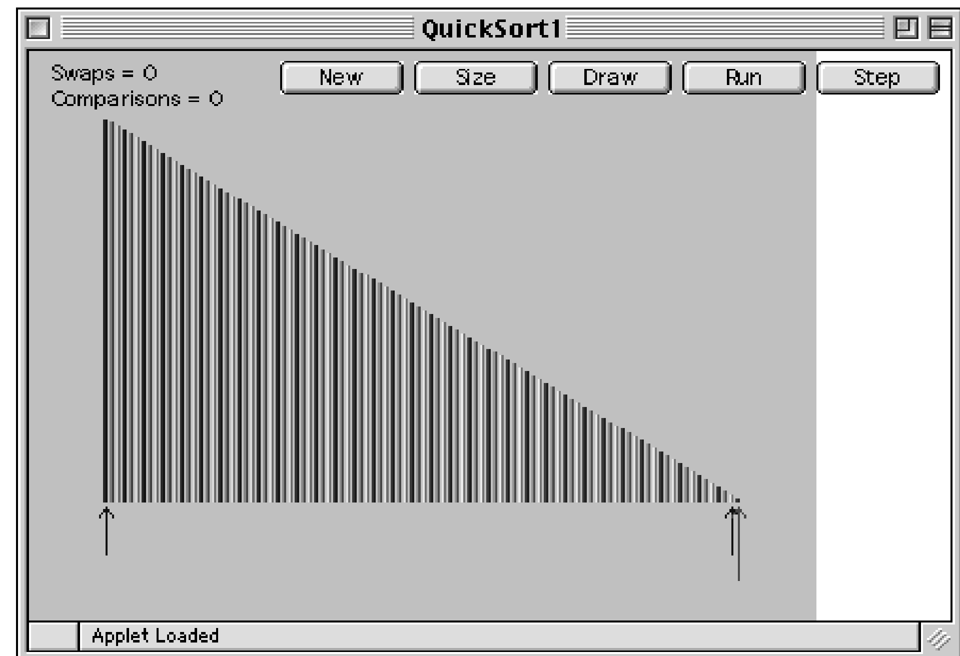
Wintersemester 2000/2001

22 und 23. Januar 2001

Zusätzliche Bemerkungen zu Quicksort

- Die Komplexität im durchschnittlichen Fall ist O(n*log n).
  - Dieser Fall nimmt an, dass die Reihung mehr oder weniger zufällig geordnet ist.
  - Wenn die Reihung fast vollständig geordnet ist, wird das Laufzeitverhalten von Quicksort schlecht. Im schlechtesten Fall,wenn die Reihung invers sortiert ist, ist die Komplexität O(n^2) wie bei den einfachen Sortierverfahren (Bubblesort, Insertionsort,...)

quicksort_invers.html



Warum ist Quicksort bei invers sortierten Reihungen so schlecht?

- Im idealen Fall ist der Pivotwert immer der Median aller Werte in der Reihung, d.h. die eine Hälfte der Werte ist kleiner, die andere größer.
  - Dann partitioniert `partitions` die Reihung immer in 2 Teilreihungen gleicher Größe .
- In der invers sortierten Reihung dividiert `partitions` (bei Verwendung des letzten bzw. kleinsten Elements als Pivotwert) die Reihung in eine Teilreihung mit 1 Element und in eine Teilreihung mit n-1 Elementen.
  - In diesem Fall erfordert jedes Element einen separaten `partitions`-Aufruf: $n * O(n) = O(n^2)$.
  - In diesem Fall ist Quicksort sogar schlechter als Bubblesort, da jeder Partitionschritt noch 2 rekursive Methodenaufrufe erfordert.

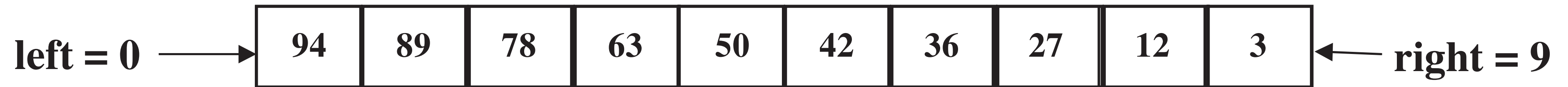
Alternative Pivotauswahl

- Das Problem ist die Auswahl des Pivotwertes (**Pivotauswahlstrategie).**
- Wenn wir immer den rechtesten Wert in einer Reihung als Pivot nehmen, dann ergibt eine invers sortierte Reihung $O(n^2)$.
- Viele Strategien sind versucht worden, einen besseren Pivotwert zu finden:
  - Die Strategie muss schnell und einfach sein, und verhindern, dass der größte oder kleinste Wert in der Reihung gewählt wird.
    - Wir könnten z.B. alle Werte vergleichen und den Median berechnen, aber eine solche Strategie braucht fast genauso viel Zeit wie die Sortierung selber.
    - Ein Kompromiss ist es, den Median des ersten, mittleren und letzten Elementes in der Reihung zu berechnen.

Copyright 2001 Bernd Brügge

3-Median-Pivotauswahl

Invers sortierte Reihung

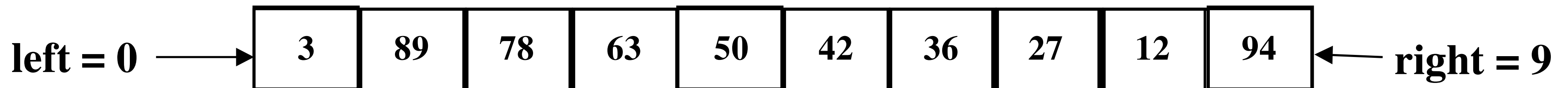


center = (left+right) / 2 = 4

Median von {94, 50, 3} = 50

pivot = 50

Sortieren der 3 Elemente, die zur Pivotberechnung verwendet wurden:



center = 4

Copyright 2001 Bernd Brügge

Medianberechung und Vorsortierung

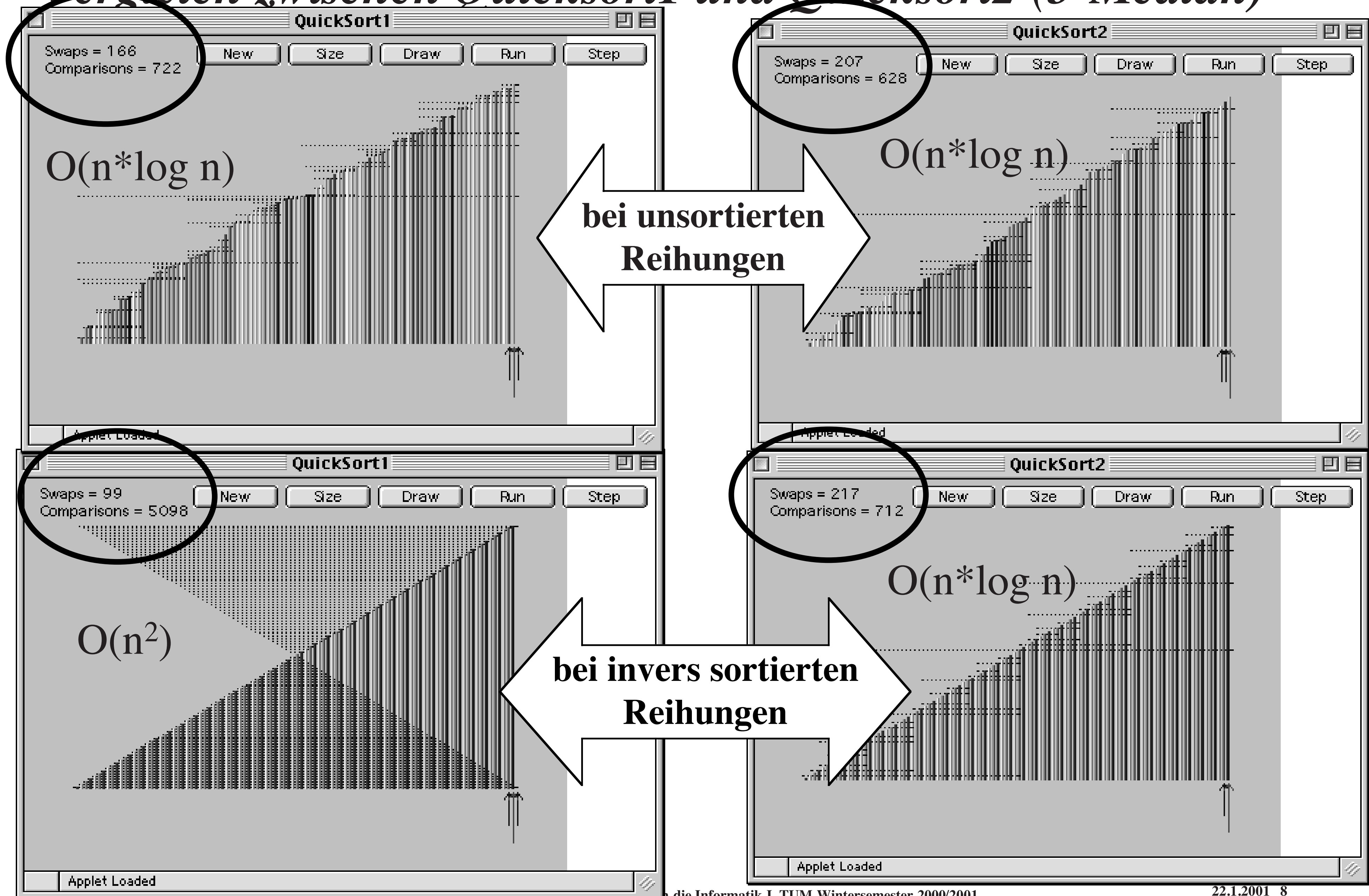
```
public double medianOf3(int left, int right) {
   int center = (left+right)/2;
   if( m[left] > m[center] )                // ordne left und center
        swap(left, center);
   if( m[left] > m[right] )                 // ordne left und right
        swap(left, right);

   if( m[center] > m[right] )               // ordne center und right
        swap(center, right);
   return m[center];                        // Wert des Medians
}
```

Eigenschaften der 3-Median-Pivotstrategie

- Wir fragen beim Eintritt von Quicksort ab, ob die Teilreihung 3 Elemente oder weniger hat:
  - In diesem Fall müssen wir "manuell" sortieren (z.B. durch Fallunterscheidung), da die Pivotberechnung mindestens 3 Elemente benötigt
- Partition kann bei left+1 und right-1 anfangen:
  - Wir wissen, dass left bereits in der korrekten Partition ist.
  - Wir wissen, dass right bereits in der korrekten Partition ist.
- Insgesamt:
  - Die 3-Median-Pivotstrategie vermeidet die $O(n^2)$-Komplexität für invers sortierte Reihungen
  - Leichte Reduktion in der Anzahl der Elemente, die partitioniert werden müssen.

Copyright 2001 Bernd Brügge

Vergleich zwischen Quicksort1 und Quicksort2 (3-Median)



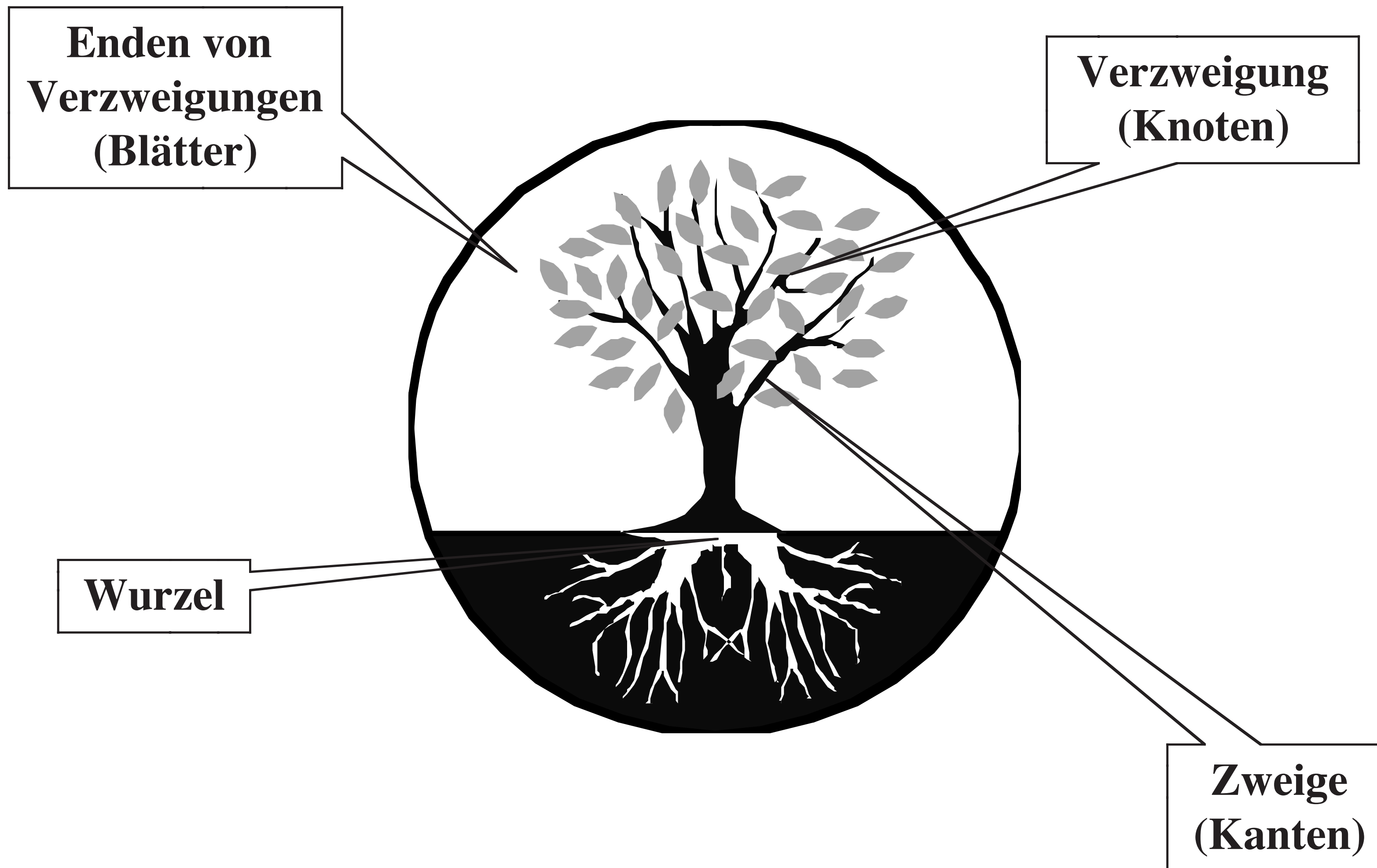
Überblick über diesen Vorlesungsblock

- Themen:
  - Bäume und Binäre Suchbäume
  - Einfügen, Suchen, Durchwandern
  - Weitere Baumalgorithmen (Inordnung, Vordnung, Nachordnung)
- Ziele:
  - Sie können aktiv mit Bäumen umgehen, insbesondere mit Baum-Operationen.
    - Sie beherrschen die Grundalgorithmen der wichtigsten Binärbaumoperationen
    - Sie kennen die Komplexität von Baumoperationen und können sie mit der Komplexität von Reihungs- und Listen-Operationen vergleichen.

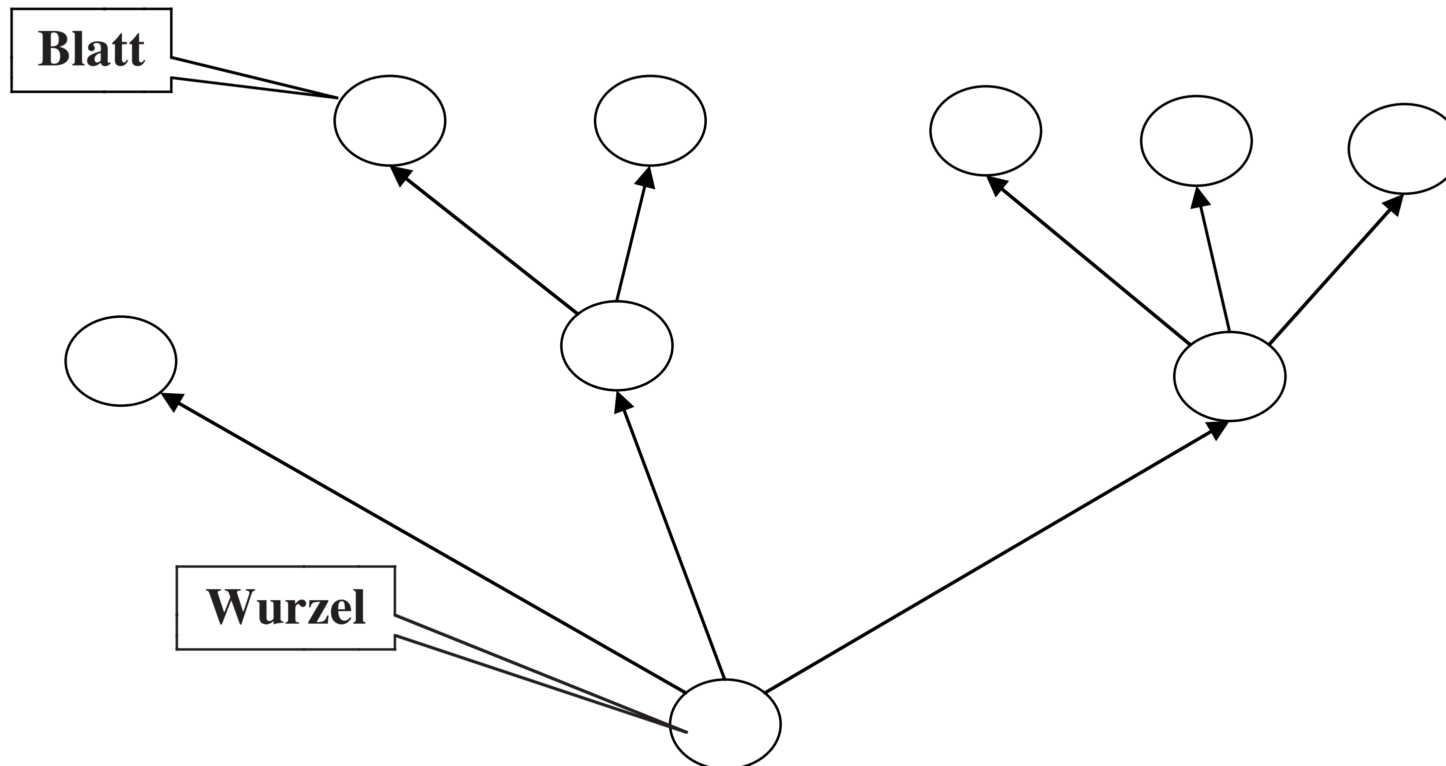
Warum Bäume?

- Sortierte Reihungen:
  - **Suchen** eines Elements ist O(log(n)) (**Schnell**)
  - **Einfügen** eines Elements ist O(n) (**Langsam**)
- Verkettete Liste (nicht sortiert):
  - **Suchen** eines Elements ist O(n) (**Langsam**)
  - **Einfügen** eines Elements ist O(1) (**Schnell**)
- **Binäre Suchbäume** (auch **Sortierte Binärbäume** genannt) vereinen bei geschickter Nutzung die Vorteile von sortierten Reihungen und von verketteten Listen:
  - **Suchen** von Elementen: **O(log(n))**
  - **Einfügen** von Elementen**:** **O(log(n))**

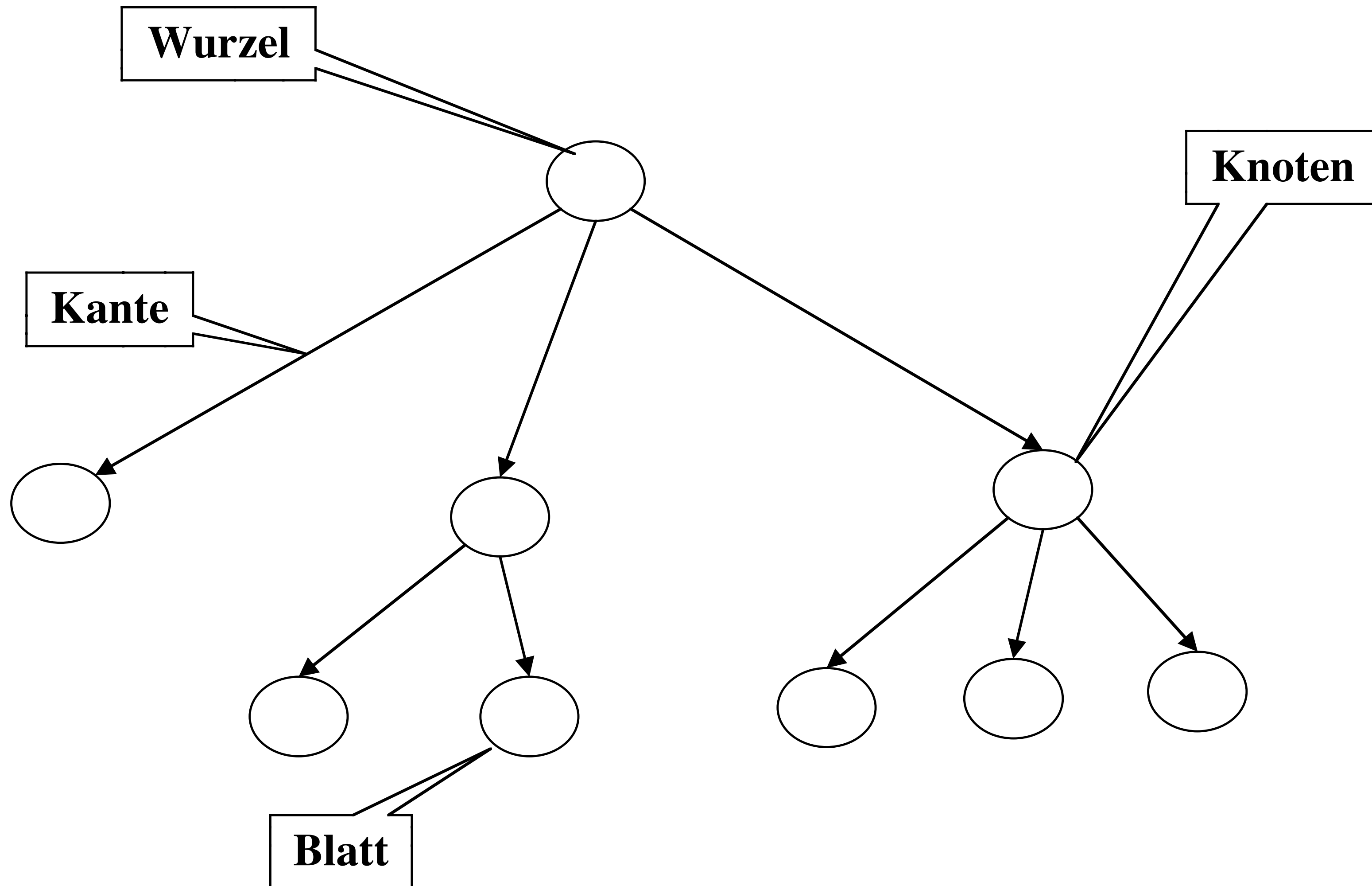
Was ist ein Baum?



Abstrakte Sicht eines Baums



Informatiker haben die Wurzel gern oben



Bäume sind Graphen

- Der Informatiker ist vor allem an **binären Bäumen** interessiert.
- Ein binärer Baum ist ein Spezialfall eines Baumes. Ein Baum wiederum ist ein Graph, deshalb fangen wir mit Graphen an.

- Ein Graph G = {V, E} besteht aus einer Menge V von **Knoten** (*nodes*) bzw. **Ecken** (*vertices*) und einer Menge K von **Kanten** (*edges*), wobei gilt:
  - $E \subseteq V \times V$
- Ein Graph ist endlich, wenn die Anzahl der Knoten endlich ist.
- Die Elemente $(v, v') \in E$ heißen Kanten. Für (v, v') schreiben wir oft $v \rightarrow v'$, wenn Kantenmenge und Graph klar sind.
- Ein Graph ist die bildliche Darstellung einer Relation. Jede Relation kann als Kantenmenge eines gerichteten Graphen aufgefasst werden.
  - Wir benutzen deshalb die Begriffe **Relation** und **gerichteter Graph** weitgehend synonym.

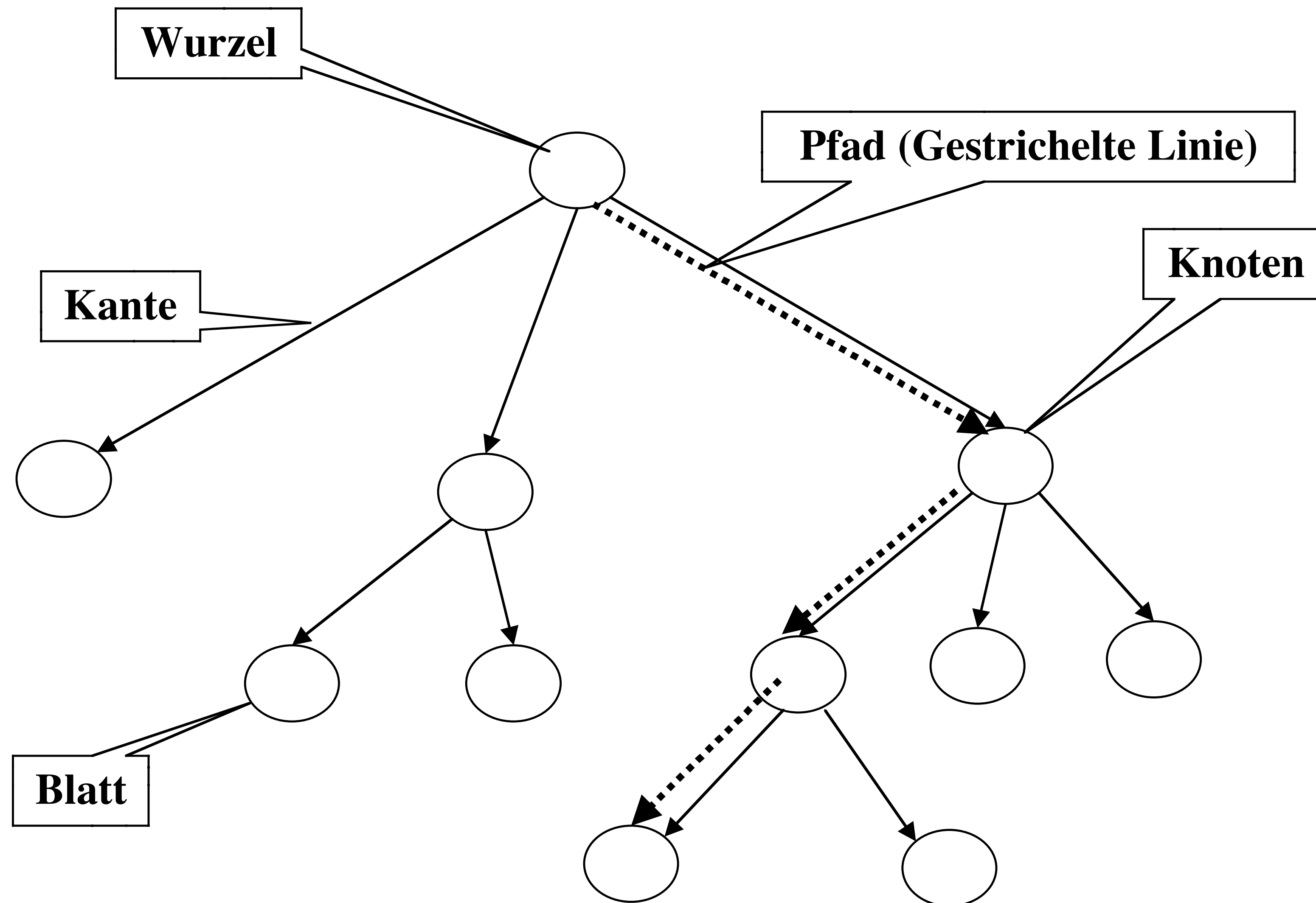
Von Graphen zu Bäumen

- Ist die Relation symmetrisch, d.h. gehört für jede Kante $v \rightarrow v'$ auch die Kante $v' \rightarrow v$ zur Kantenmenge, dann haben wir einen **ungerichteten Graphen.**
- Die Menge v* der Kanten, die von einem Knoten v eines gerichteten Graphen ausgehen, heißt die **Ausgangsmenge** des Knotens v.
- Die Menge *v der Kanten, die in einem Knoten v enden, heißt die **Eingangsmenge** des Knotens v.
- Eine Folge $(e_1, e_2, \ldots e_n)$ von Kanten $e_i = (v_{i-1}, v_i) \in E$ heißt ein **Pfad** der Länge $n \geq 0$ vom Knoten v_o zum Knoten v_n. Wir schreiben dafür auch $v_o \xrightarrow{*} v_n$.
- Ein Pfad der Länge $n \geq 1$ heißt ein **Kreis** oder **Zyklus**, wenn $v_o = v_n$.
- Ein ungerichteter Graph ohne Kreise, in dem je zwei Knoten durch einen Pfad verbunden sind, heißt **ungerichteter Baum**.

Copyright 2001 Bernd Brügge

Bäume

- **Definition (gerichteter) Baum (tree):**
  – Ein gerichteter Baum, der genau einen Knoten v mit $|{}^*v| = 0$ hat, d.h. der nur einen Knoten ohne Eingangsmenge hat.
    – Dieser Knoten heißt die **Wurzel** des Baumes
- **Definition Blatt (leaf):**
  – Ein Knoten eines Baumes mit Ausgangsgrad $|v^*| = 0$.
- **Definition Innerer Knoten (inner node):**
  – Ein Knoten eines Baumes mit Ausgangsgrad $|v^*| > 0$.
- **Definition Höhe eines Knotens (height of a node) :**
  – Die Länge des Pfades von der Wurzel bis zu einem Knoten v heißt die Höhe von v.
- **Definition Höhe eines Baumes (height of a tree):**
  – Das Maximum der Höhen aller Blätter.



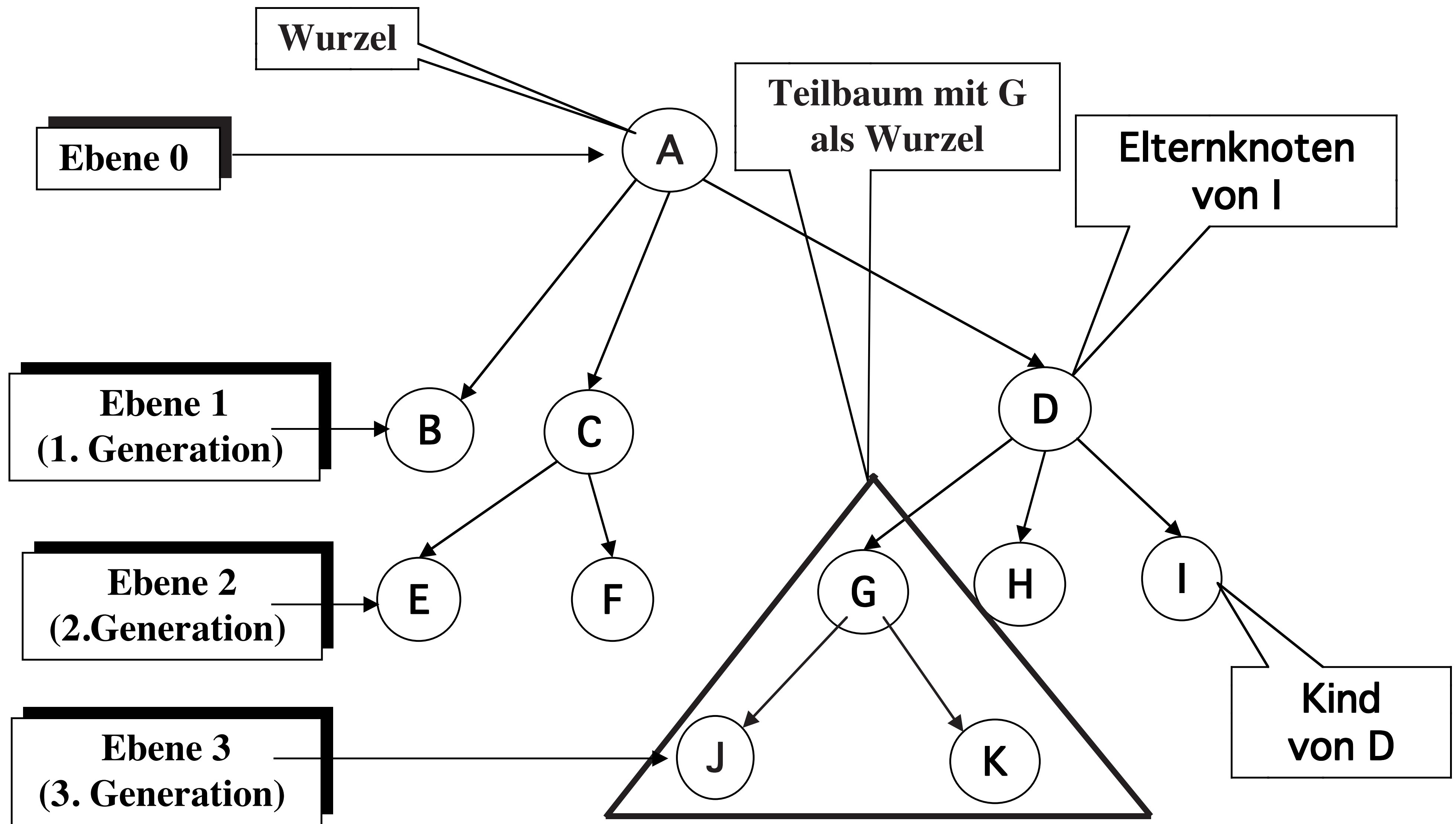
Für einen Baum G = (V, E) mit endlich vielen Knoten gilt |V| = |E| + 1.

Copyright 2001 Bernd Brügge

Noch ein paar Definitionen aus der Genealogie

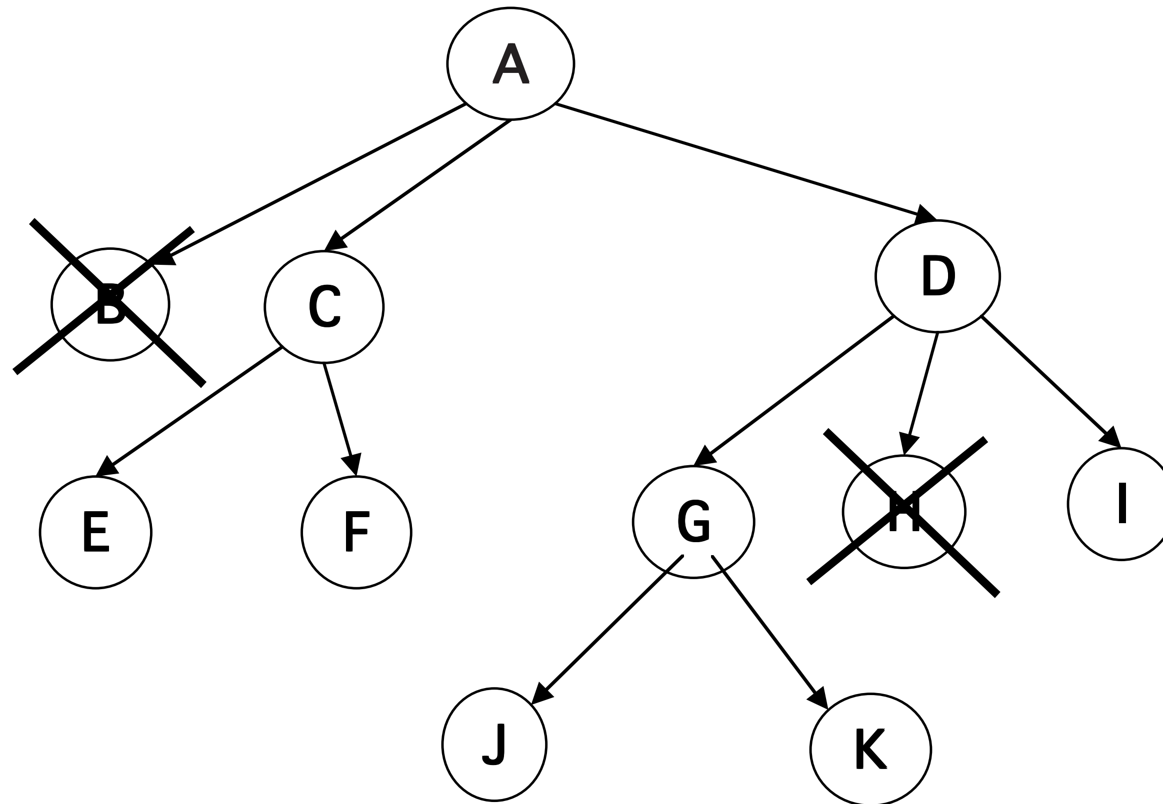
- **Elternknoten** (parent node)
  - Jeder Knoten (mit Ausnahme der Wurzel) hat genau eine Kante, die zu einem anderem Knoten hochwandert. Dieser Knoten heißt **Elternknoten**.
- **Kindknoten** (child node)
  - Jeder Knoten (außer Blättern) hat eine oder mehr Kanten, die zu anderen Knoten führen. Diese Knoten heißen die **Kindknoten** bzw. **Kinder** des Knotens.
- **Teilbaum** (subtree)
  - Jeder Knoten kann als Wurzel eines Teilbaums angesehen werden. Ein **Teilbaum** eines Knotens v enthält also alle Kinder und Kindeskinder, d.h. alle Nachfahren von v.
- **Ebene** (level):
  - Die Generation eines Knotens, von der Wurzel aus gerechnet.

Copyright 2001 Bernd Brügge


Wurzel
Ebene 0
A
Teilbaum mit G
als Wurzel
Elternknoten
von I
Ebene 1
(1. Generation)
B
C
D
Ebene 2
(2.Generation)
E
F
G
H
I
Kind
von D
Ebene 3
(3. Generation)
J
K

Binärbaum

- **Binärbaum:** Ein Baum, in dem jeder Knoten höchstens 2 Kinder haben kann.



Beispiel eines Baumes

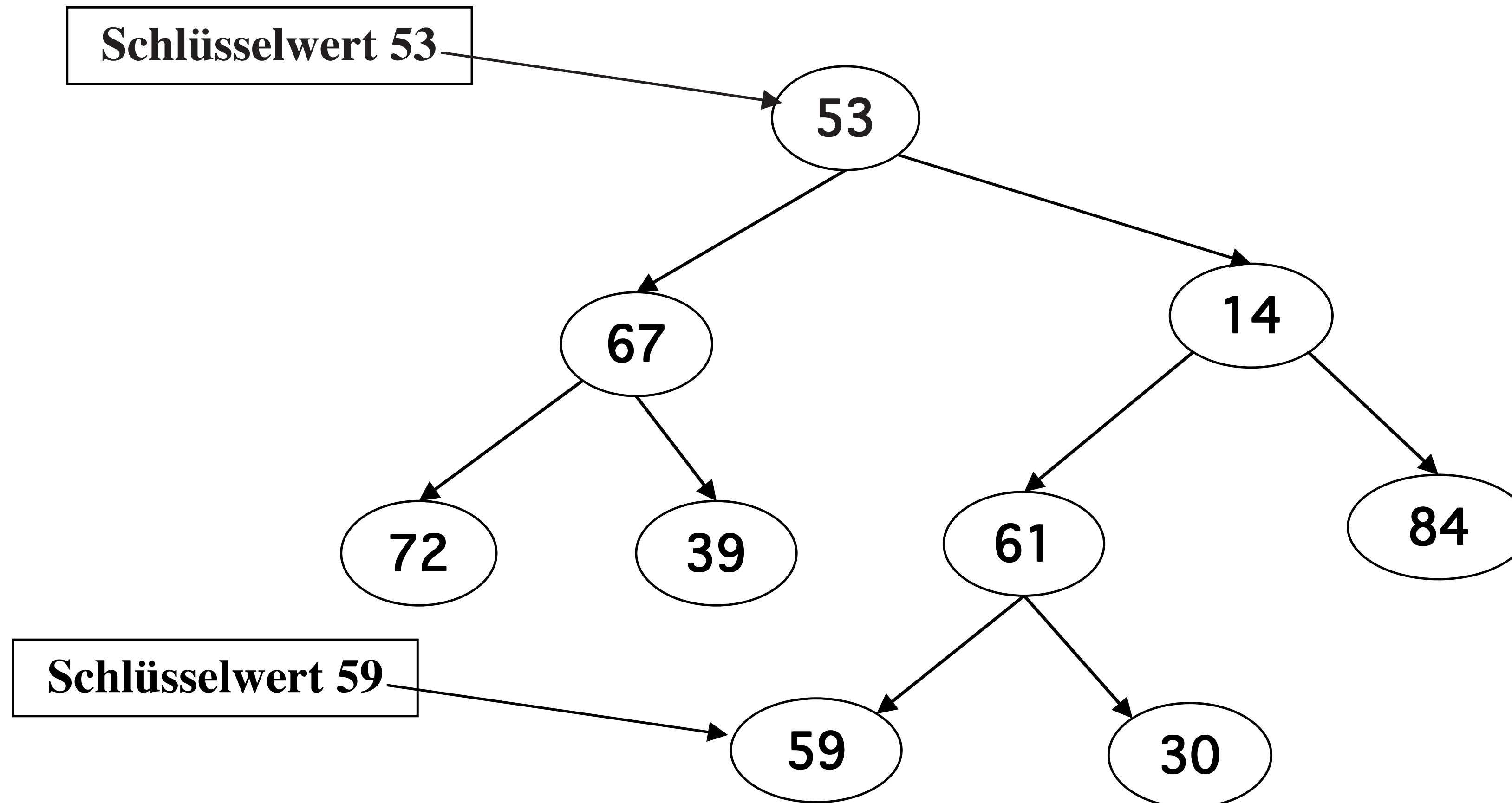
- Beispiel: Hierarchisches Dateisystem von Unix
  - Jedes Verzeichnis (directory) kann viele Kinder, d.h. Unterverzeichnisse und Dateien, haben.
  - Eine Datei (file) ist ein Blatt; sie kann kein Unterverzeichnis mehr haben.
  - Ein vollständiger Dateiname, z. B. **`/usr/bin/Test.java`** korrespondiert zu dem Pfad von der Wurzel (**`/`**) bis zum Dateinamen (**`Test.java`**).
- Das Unix-Dateisystem ist ein Baum, aber kein Binärbaum, da ein Verzeichnis mehr als 2 Dateien enthalten kann.

Copyright 2001 Bernd Brügge

Schlüsselwert

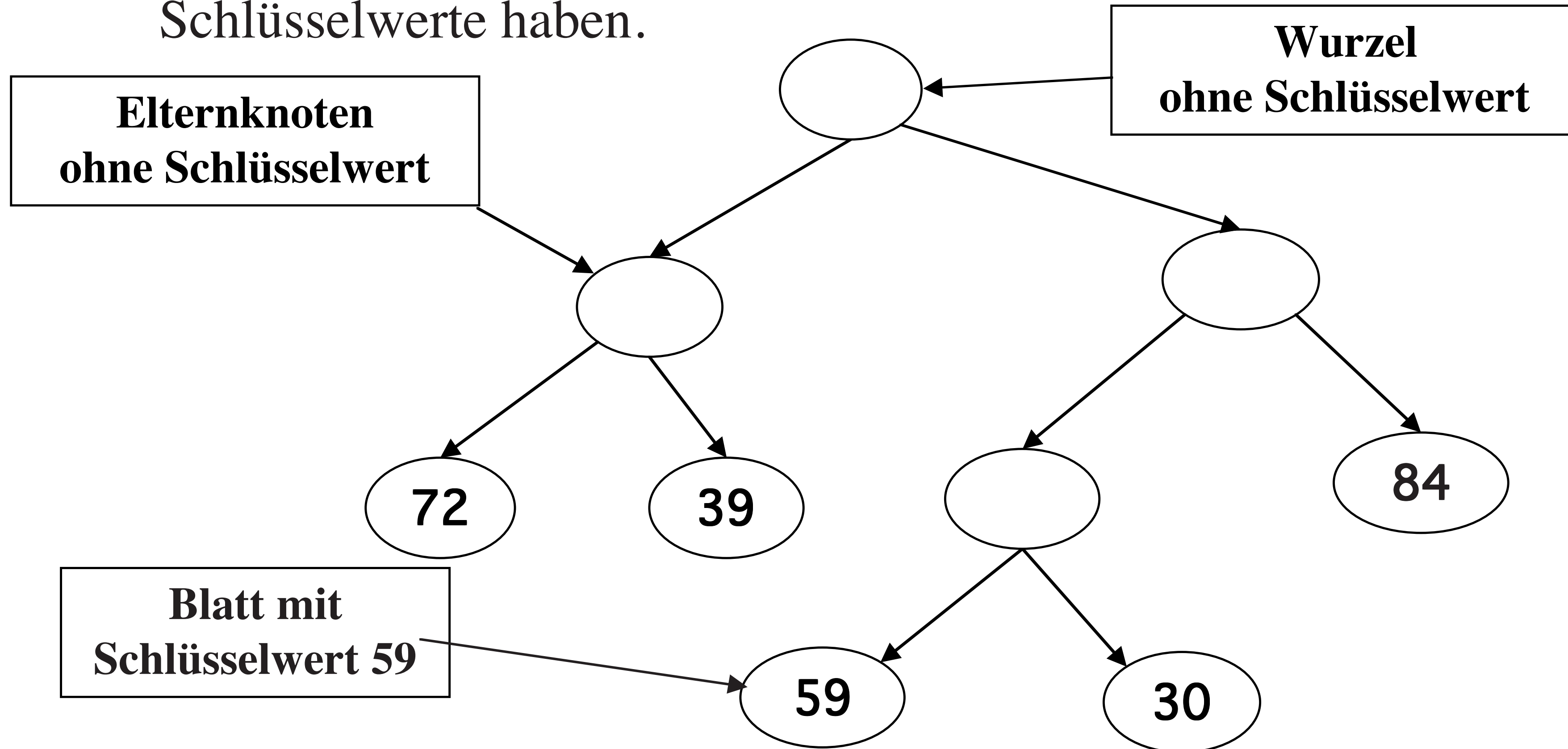
- In der Informatik sind die Knoten in einem Baum vor allem dann interessant, wenn sie Objekte aus der Wirklichkeit darstellen.
  - Menschen, Autoteile, Flugzeugreservierungen
- Die verschiedenen Objekte unterscheiden wir gewöhnlich durch ihren sogenannten Schlüsselwert.
- **Schlüsselwert (key):** Der Wert eines Objektes, der benutzt wird, um nach dem Objekt zu suchen oder andere Operationen auf dem Objekt auszuführen.
- In dieser Vorlesung verwenden wir keine Objekte, sondern nur Schlüsselwerte vom Typ **int.** Diese Zahl wird im Knoten angegeben.
  - In einer späteren Vorlesung (5-6. Februar) werden wir zeigen, wir man ein Studentenverzeichnis als Baum realisieren kann, dessen Knoten vom Typ **`Student`** sind.

Beispiel eines Baumes mit Schlüsselwerten



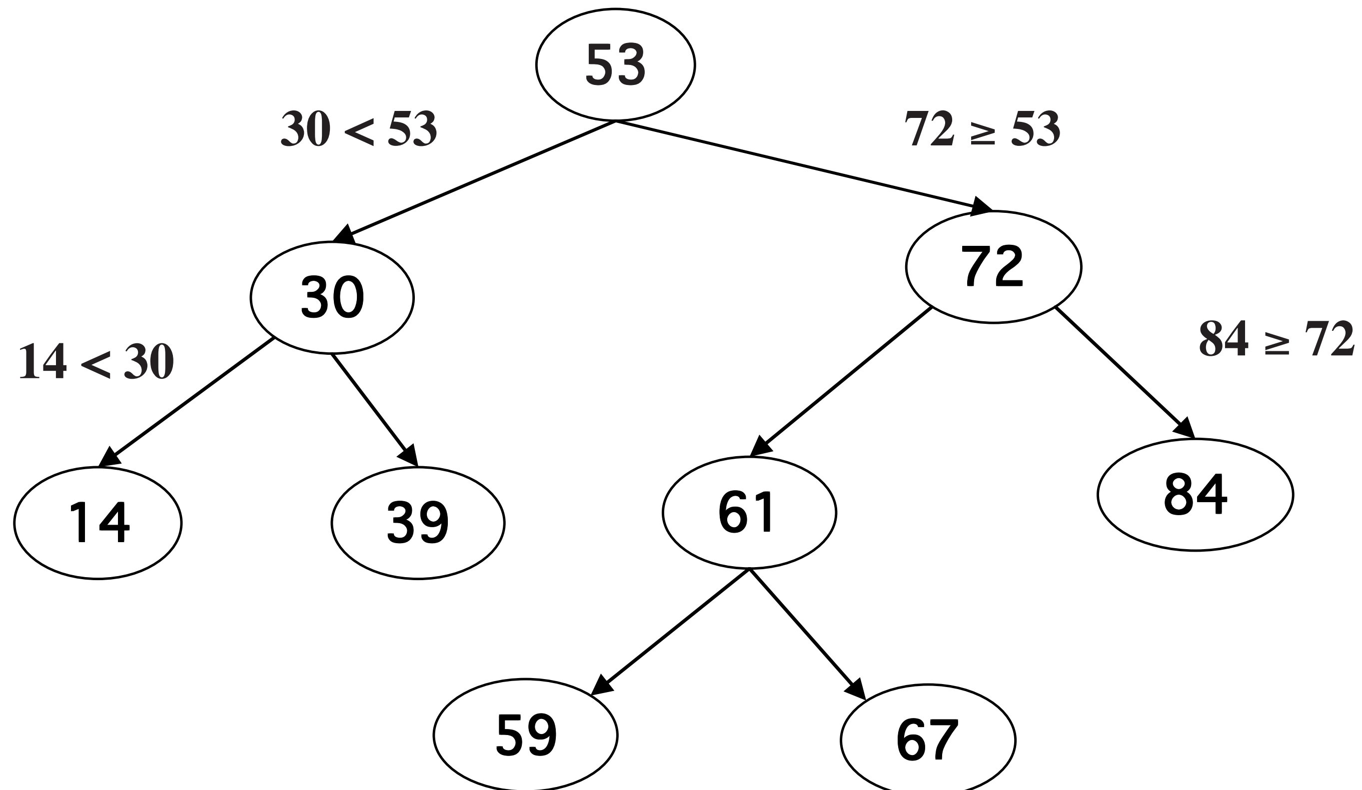
Beispiel eines Blattbaums

- **Blattbaum:** Ein spezieller Baum, in dem nur die Blätter Schlüsselwerte haben.



Binärer Suchbaum

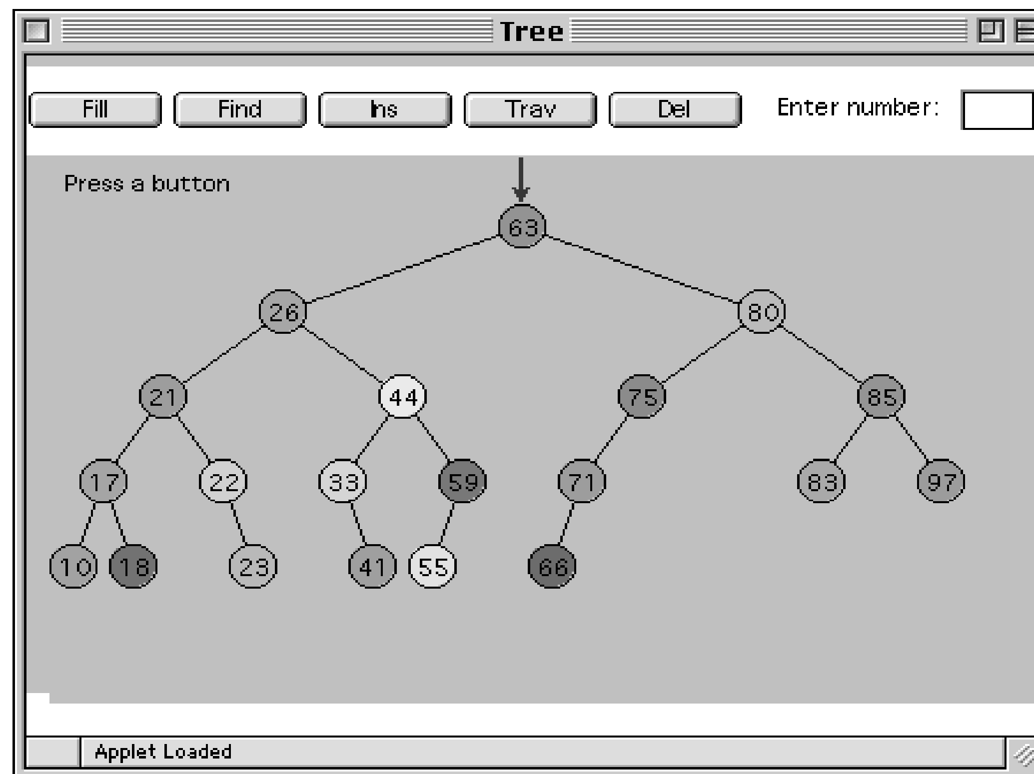
- In einem **binären Suchbaum** gilt für alle Knoten v:
  - Das linke Kind v_L hat einen Schlüsselwert, der kleiner ist als der Schlüsselwert von v.
  - Das rechte Kind v_R hat einen Schlüsselwert, der gleich oder größer ist als der Schlüsselwert von v.



Typische Operationen auf binären Suchbäumen

- Kreieren des Baums (Fill)
- Finden eines Knotens mit gegebenem Schlüssel (Find)
- Einfügen eines neuen Knotens (Ins)
- Besuchen aller Knoten des Baumes (Trav)
- Löschen eines Knotens (Del)

Tree.html

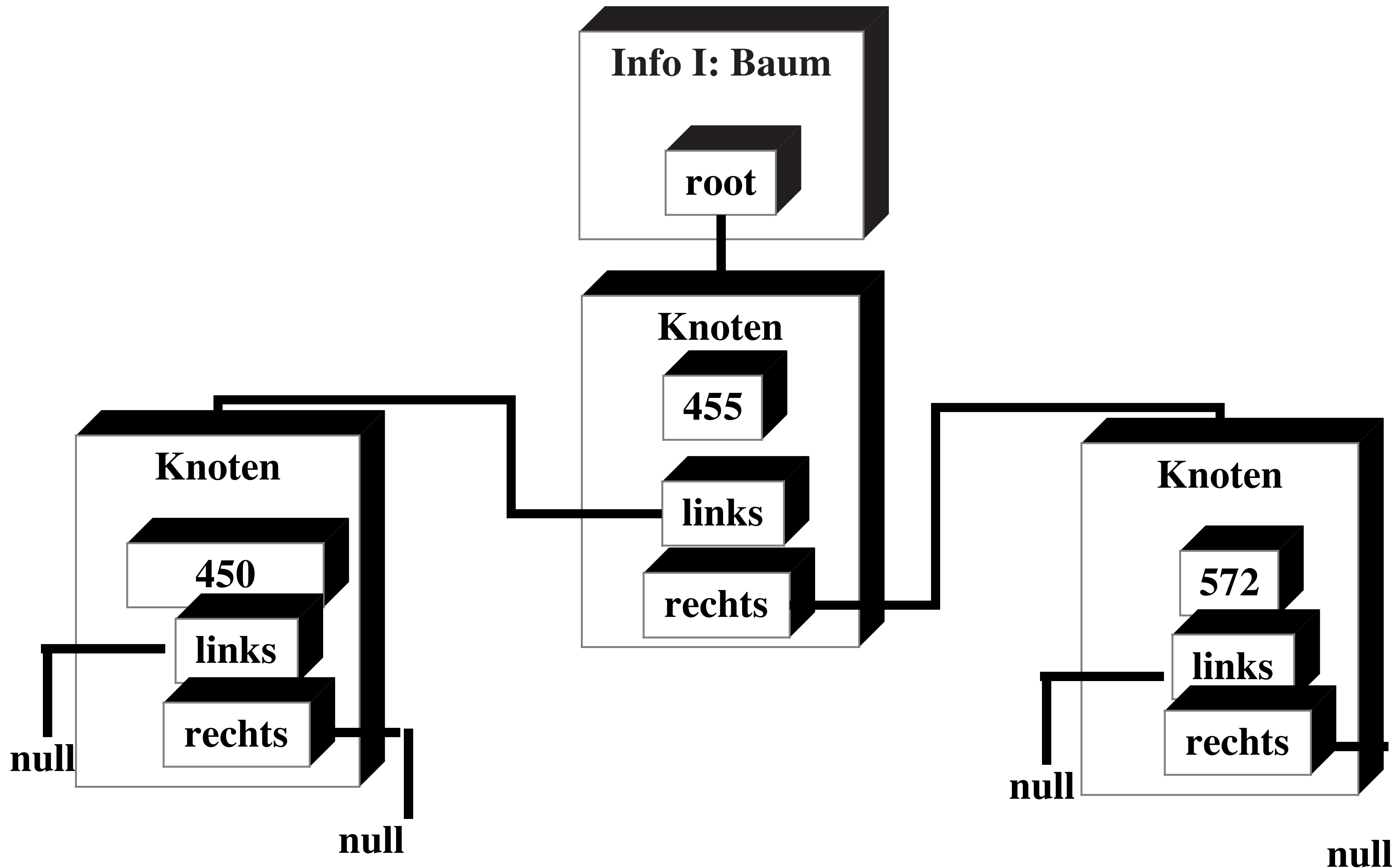


Modellierung von Bäumen

- Beispiel:
  - Das Studentenverzeichnis aller Studenten in Info I
  - Die Immatrikulation hat gerade begonnen.
    - Das Studentenverzeichnis besteht aus 3 Studenten:
    - Andreas (455), Selma (450), Alexis (572)
- Wir können das Verzeichnis als Baum modellieren.
  - Die Matrikelnummern von Andreas, Selma und Alexis sind dann die applikationsspezifischen Daten (vom Typ `int`).

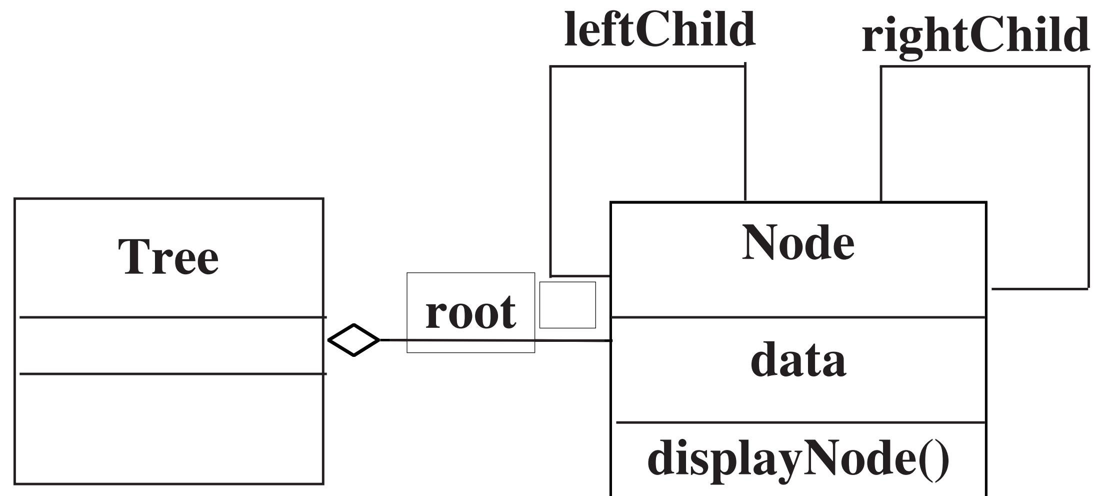
Copyright 2001 Bernd Brügge

Ein binärer Baum für das Studentenverzeichnis



Modellierung eines Binären Suchbaums: Problembeschreibung und Analyse

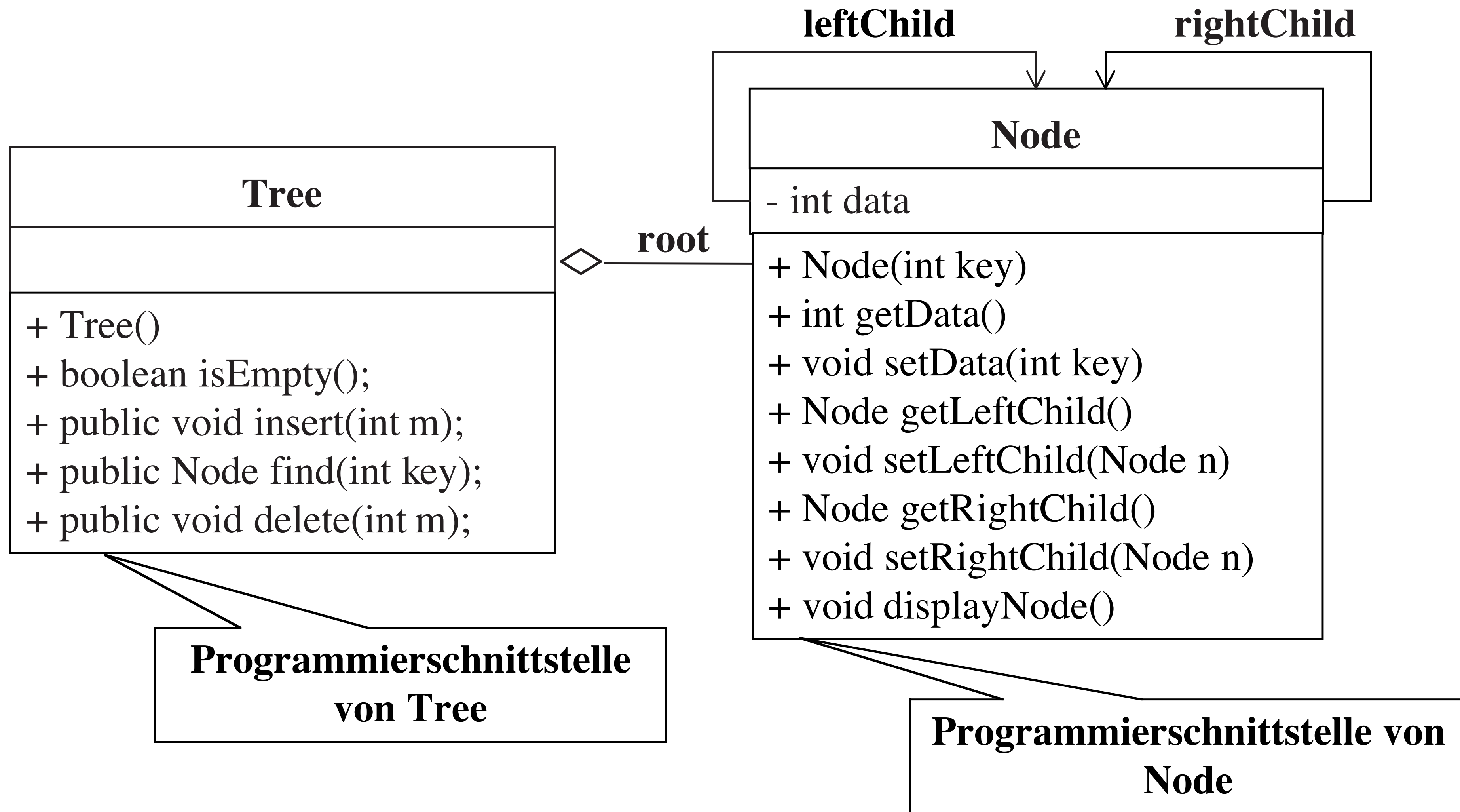
- **Problembeschreibung:** Ein Baum besteht aus 0 oder mehr Knoten
- **Analyse-Phase:**
  - ***Klassenidentifizierung***: Die Komponenten sind Baum (**Tree**) und Knoten (**Node**)
  - ***Assoziationen:*** „besteht aus 0 oder mehr" übersetzen wir als „1-viele" Aggregation
  - ***Attribute:*** Ein Knoten eines Binärbaums enthält immer drei Attribute:
    - applikationsspezifische Daten (**data**)
    - eine Referenz auf das linke Kind (**leftChild**)
    - eine Referenz auf das rechte Kind (**rightChild**)
  - ***Operationen:*** **displayNode()** druckt die applikationsspezifischen Daten.



Benennung von Assoziationen: Die Aggregation zwischen **Tree** und **Node** modelliert die Wurzel, wir bezeichnen sie deshalb mit **root**.

Copyright 2001 Bernd Brügge

Modellierung eines binären Suchbaums: Objektentwurf



Copyright 2001 Bernd Brügge

Modellierung eines binären Suchbaums: Java-Implementation eines `int`-Knotens

```
class Node {
   private int data;
   private Node leftChild;
   private Node rightChild;

public Node(int key)   {
  data = key;            // Initialisiere Datum
  leftChild= null;       // Initialisiere die
  rightChild = null;     // Kinder vom Knoten
   }
public int getData() {
   return data;
}
public void setData(int key) {
  data = key;
}
```

```
public Node getLeftChild() {
    return leftChild;
}
public void setLeftChild(Node n) {
   leftChild= n;
}
public Node getRightChild() {
    return rightChild;
}
public void setRightChild(Node n) {
   rightChild= n;
}
public void displayNode()     {
   System.out.print("<" + data + "> ");
}
}  // end class Node
```

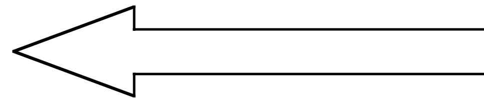
Java-Spezifikation eines binären Suchbaums

```
class Tree {
  private  Node root;
✓ public boolean isEmpty();
  public void insert(int key);
  public Node find(int key);
  public void delete(int key);
} // End class Tree

 public boolean isEmpty() {
   return root == null;
}
```

Copyright 2001 Bernd Brügge

Java-Spezifikation eines binären Suchbaums

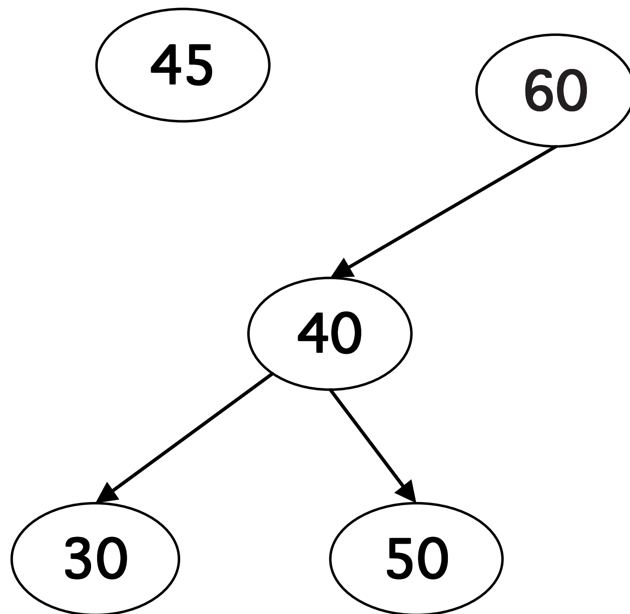
```
class Tree {
  private  Node root;
✓ public boolean isEmpty();
  public void insert(int key);
  public Node find(int key);
  public void delete(int key);
} // End class Tree
```



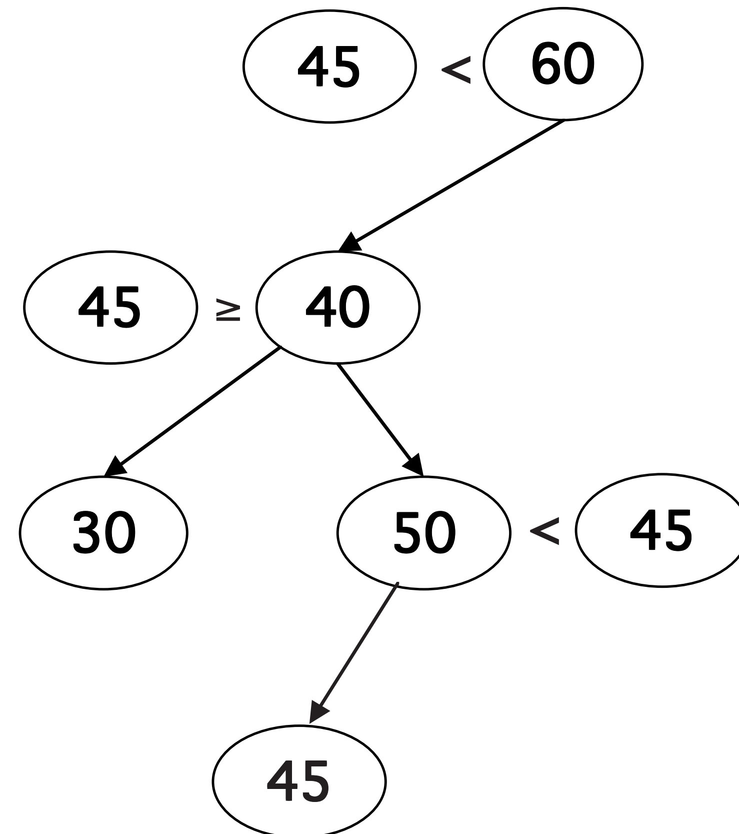
```
 public boolean isEmpty() {
   return root == null;
}
```

Einfügen eines Knotens

Vorher



Nachher



Copyright 2001 Bernd Brügge

Einfügen eines Knotens: Java-Implementation von insert()

```
public void insert (int key) {                        // Einfügen eines Knoten mit Schlüssel key
    Node newNode = new Node(key);                     // neuen Knoten mit Schlüssel key erzeugen
    if (root == null)                                 // wenn der Baum bisher leer war,
        root = newNode;                               // newNode als Wurzel einfügen
    Node current = root;
    while (current != newNode) {                      // solange newNode nicht eingefügt wurde
        if (key < current.getData()) {                // wenn key < aktueller Schlüssel, nach links
          if (current.getLeftChild() == null)         // wenn kein linker Kindknoten vorhanden,
            current.setLeftChild(newNode);            // newNode als linken Kindknoten einfügen
          current = current.getLeftChild();           // weiter mit linkem Kindknoten
        } // Ende von links einfügen
        else {                                        // wenn key ≥ aktueller Schlüssel, nach rechts
          if (current.getRightChild() == null)        // wenn kein rechter Kindknoten vorhanden,
            current.setRightChild(newNode);           // newNode als rechten Kindknoten einfügen
          current = current.getRightChild();          // weiter mit rechtem Kindknoten
        } // Ende von rechts einfügen
    } // end while
} // end insert ()
```

Komplexität von Einfügen in einen binären Suchbaum

- Wir finden den Knoten, an den das Element zu hängen ist, mit $V(n)= \log_2(n)$ Vergleichen und $Z(n) = \log_2(n)$ Zuweisungen, d.h. die Komplexität von **Einfügen** in einem **binären Suchbaum** ist **$O(\log_2(n))$**.
- Dies gilt allerdings nur für **balancierte** Bäume, d.h. Bäume, bei denen sich der linke und der rechte Teilbaum jedes inneren Knotens bzgl. Höhe und Knotenzahl wenig unterscheiden.
- Bei unbalancierten Bäumen kann die Komplexität im ungünstigsten Fall wie bei Listen linear ($O(n)$) werden.
- Ein **vollständiger Baum** ist ein Baum, bei dem alle Ebenen maximal gefüllt sind. Ein Binärbaum mit Höhe k hat demnach insgesamt $2^{k+1}-1$ Knoten.

Copyright 2001 Bernd Brügge

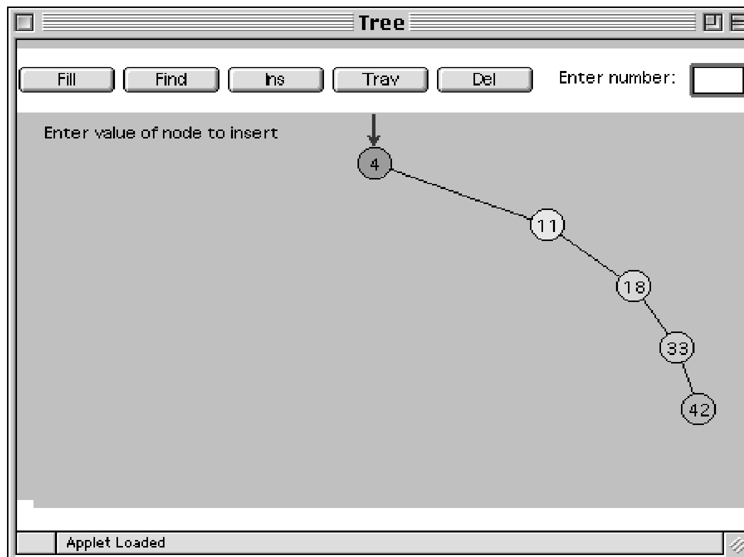
Unbalancierte Bäume

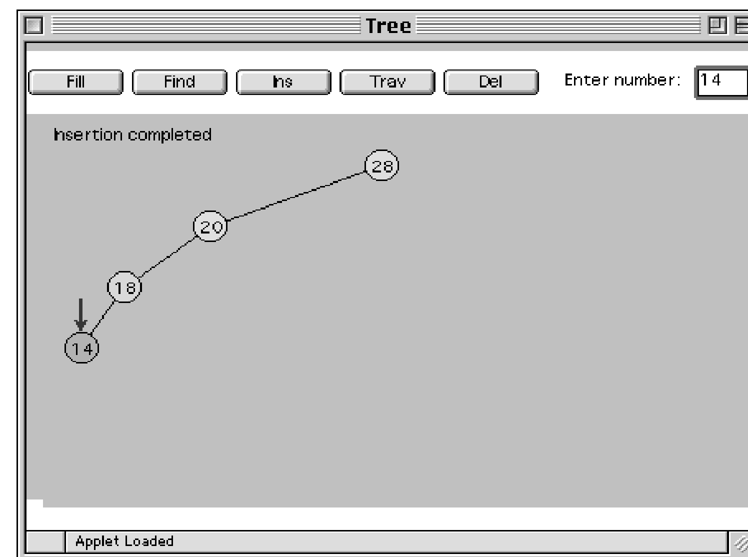
❖ Bäume können aus der Balance kommen, wenn man die Daten in einer bestimmten Weise einfügt:

– Wenn man Werte in aufsteigender Reihenfolge (z.B. 11, 18, 33, 42, 65, usw) einfügt, dann werde alle diese Werte als rechte Kinder eingefügt.

– Wenn man Werte in absteigender Reihenfolge einfügt, dann werde alle diese Werte als linke Kinder eingefügt.

unbalanced.html



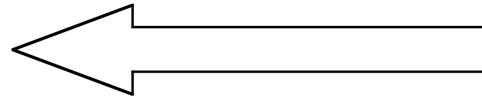


Wann entstehen unbalancierte Bäume?

- Wenn Daten in sortierter Reihenfolge in einen binären Suchbaum eingetragen werden, entsteht ein unbalancierter Baum.
  - Beispiel: Es soll eine Datei für das Studentenverzeichnis der TU München erstellt werden. Ein Datentypist trägt die Studenten nach aufsteigenden Matrikelnummern ein.
- Wenn Daten in unsortierter Form in einen binären Suchbaum eingetragen werden, entsteht ein besser balancierter Baum (aber nicht unbedingt optimal balanciert).
- Die Komplexität von Operationen auf unbalancierten Bäumen ist wesentlich schlechter als auf balancierten Bäumen.

Java-Spezifikation eines binären Suchbaums

```
class Tree {
   private Node root;
✔ public boolean isEmpty();
✔ public void insert(int key);
   public Node find(int key);
   public void delete(int key);
} // end class Tree
```



Copyright 2001 Bernd Brügge

Finden eines Knotens: Java-Implementation von find()

```
public Node find(int key) {                   // Finde Knoten mit gegebenem Schlüssel
    Node current = root;                      // wir starten an der Wurzel
    while (current != null) {                 // solange wir nichts gefunden haben,
                                              // gehen wir weiter nach unten
        if (key == current.getData())         // ist der gesuchte Knoten gefunden?
            return current;                   // gefundenen Knoten zurückgeben
        else if (key < current.getData())     // müssen wir nach links gehen?
            current = current.getLeftChild(); // dann linken Teilbaum untersuchen
        else                                  // oder nach rechts?
            current = current.getRightChild(); // dann rechten Teilbaum untersuchen
    }
    return null;                              // null heißt: Wir haben nichts gefunden
}  // end find()
```

Komplexität der Suche in einem binären Suchbaum

- Wir schauen uns wieder die Anzahl der Vergleiche V(n) und Zuweisungen Z(n) an, um einen Knoten zu finden.
  - die Werte hängen davon ab, auf welcher Ebene der Knoten angesiedelt ist; in einem **vollständig balancierten binären Suchbaum** mit $n= 2^k$ Elementen gibt es k Ebenen.
    - Knoten auf Ebene 0: 1 Zuweisung und 1 Vergleich
    - Knoten auf Ebene 1: 2 Zuweisungen und 2 Vergleiche
    - Knoten auf Ebene k-1: k Zuweisungen und k Vergleiche
- Wir finden also jeden Knoten mit maximal $V(n)= \log_2(n)$ Vergleichen und $Z(n) = \log_2(n)$ Zuweisungen, d.h. die **Komplexität der Suche** in einem **vollständig balancierten binären Suchbaum** ist $\mathbf{O(\log_2(n))}$.

Durchlaufen von Bäumen

- Bevor wir uns dem Löschen von Knoten zuwenden, schauen wir uns ein ganz wichtiges Konzept an: Das Durchlaufen (**Traversieren**) von Bäumen.
- Bei der **Traversierung** eines Baums (traversal of a tree) können wir die Knoten des Baums in einer genau spezifizierten Reihenfolge besuchen:
  - **Inordnung (inorder):**
    - Reihenfolge der Traversierung:
      linker Teilbaum - Wurzel - rechter Teilbaum
    - Besucht die Knoten eines sortierten Baumes in aufsteigender Reihenfolge.
  - **Vorordnung (preorder):**
    - Reihenfolge der Traversierung:
      Wurzel - linker Teilbaum - rechter Teilbaum
  - **Nachordnung (postorder):**
    - Reihenfolge der Traversierung:
      linker Teilbaum - rechter Teilbaum - Wurzel

Baum-Traversierung mit Inordnung

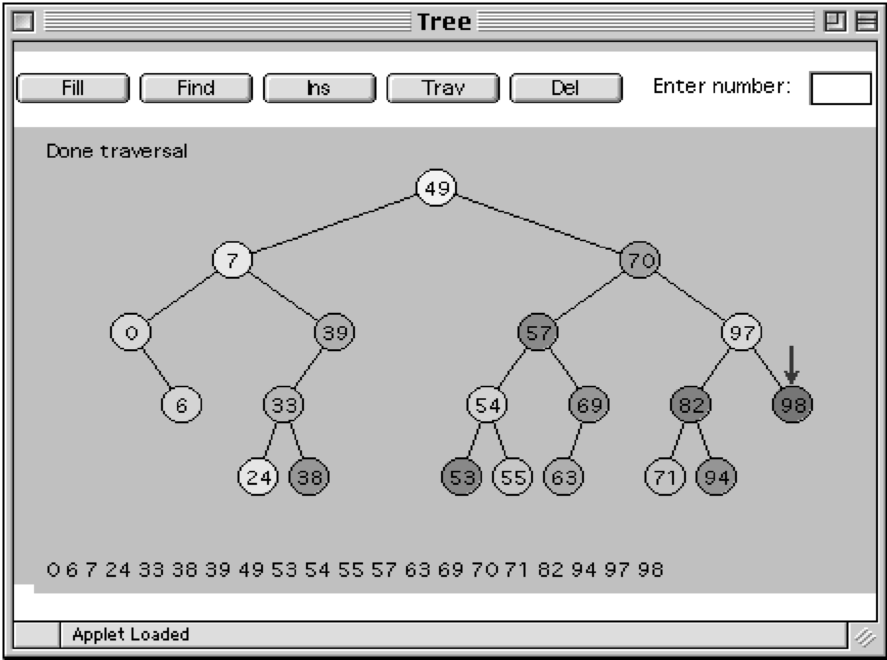
- Die Inordnung eines binären Suchbaumes besucht alle Knoten des Baumes in aufsteigender Reihenfolge.
  - Bevorzugte Methode, wenn man eine sortierte Liste aus den Daten eines binären Suchbaumes bekommen will.
- Rekursive Definition der Traversierung mit Inordnung:
  - Nehmen wir an, wir haben eine Methode **inOrder(node N)**, wobei N ein Knoten im Baum ist.
  - Als ersten aktuellen Parameter übergeben wir **root.**
- Wenn N nicht null ist, führt **inOrder** 3 Anweisungen aus:
  1. Aufruf von **inOrder** mit dem linken Teilbaum von N: **inOrder(getLeftChild(N))**
  2. **"Besuch" von N**: z.B. Ausdrucken des Schlüssels von N
  3. Aufruf von **inOrder** mit dem rechten Teilbaum von N: **inOrder(getLeftChild(N))**

Copyright 2001 Bernd Brügge

Visualisierung von Inordnung

trav.html

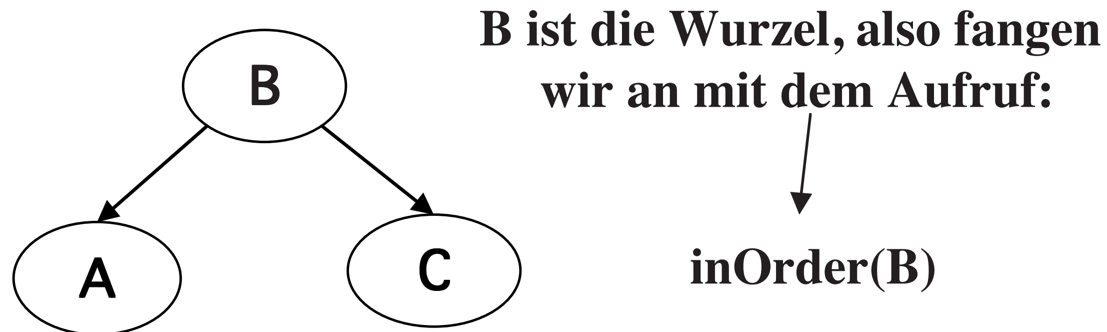
Sortierte Liste



Copyright 2001 Bernd Brügge

Inordnung: Java-Implementation von inOrder()

```
private void inOrder(Node localRoot) {
    if(localRoot != null)   {
      inOrder(localRoot.getLeftChild());
      localRoot.displayNode();
      inOrder(localRoot.getRightChild());
    }
 }
```

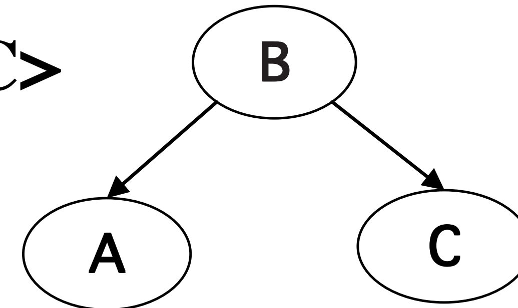


Copyright 2001 Bernd Brügge

Inordnung: Java-Implementation von inOrder()

```
private void inOrder(Node localRoot) {
   if(localRoot != null)   {
     inOrder(localRoot.getLeftChild());
     localRoot.displayNode();
     inOrder(localRoot.getRightChild());
   }
}
```

<A> <B> <C>



inOrder(B)

1. inOrder(A)
2. displayNode(B)
3. inOrder(C)

inOrder(A)

1. inOrder(null)
2. displayNode(A)
3. inOrder(null)

inOrder(C)

1. inOrder(null)
2. displayNode(C)
3. inOrder(null)

inOrder(null) — tut nichts

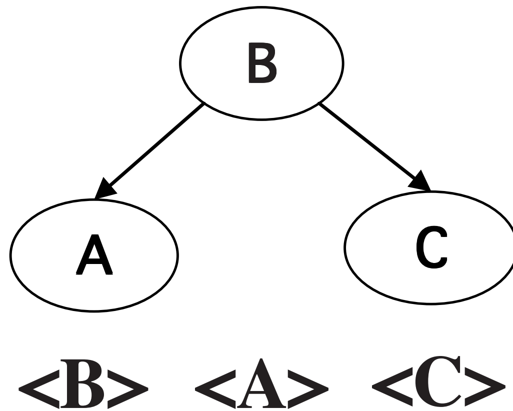
inOrder(null) — tut nichts

inOrder(null) — tut nichts

inOrder(null) — tut nichts

Vorordnung: Java-Implementation von preOrder()

```
private void preOrder(Node localRoot) {
    if(localRoot != null)   {
      localRoot.displayNode();
      preOrder(localRoot.getLeftChild());
      preOrder(localRoot.getRightChild());
    }
 }
```

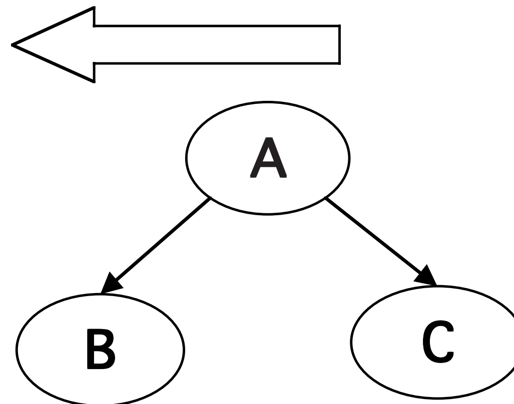


<B> <A> <C>

Präfix-Notation

Nachordnung: Java-Implementation von postOrder()

```
private void postOrder(Node localRoot) {
    if(localRoot != null)   {
      postOrder(localRoot.getLeftChild());
      postOrder(localRoot.getRightChild());
      localRoot.displayNode();
    }
 }
```



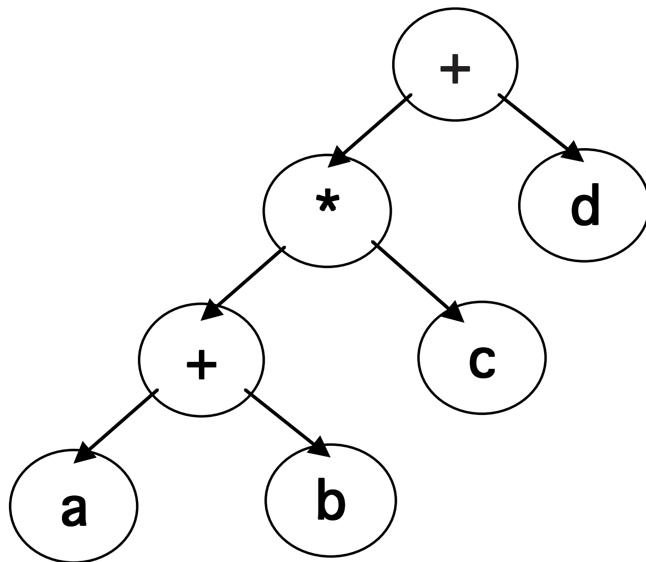
Postfix-Notation <B> <C> <A>

Wozu braucht man Präfix- und Postfix-Notationen?

- Beispiel eines Ausdruckes: **(a+b)*c + d**
  - Diese Notation heißt **Infix-Notation,** weil jeder Operator immer zwischen zwei arithmetischen (Teil-)Ausdrücken steht. Ohne Klammern ist der Ausdruck nicht eindeutig
- Prä-und Postfix-Notation sind bei der Erstellung von Compilern (z.B. einem Java-Compiler) sehr populär:
  - Man kann nämlich syntaktisch korrekte algebraische Ausdrücke *klammerfrei* darstellen.
- Der Compiler erzeugt einen **Syntaxbaum** für einen Ausdruck.
  - Die Wurzel enthält immer einen Operator
  - Jeder Teilbaum stellt entweder den Namen einer Variablen dar oder einen arithmetischen Teilausdruck.
- Ein derartiger Baum heißt **Kantorowitsch-Baum**
  - Beispiel: **(a+b)*c + d**

Kantorowitsch-Baum

- Ein Kantorowitsch-Baum stellt die syntaktisch korrekte Ableitung eines arithmetischen Ausdrucks dar:



Kantorowitsch-Baum des algebraischen Ausdrucks (a+b)*c + d

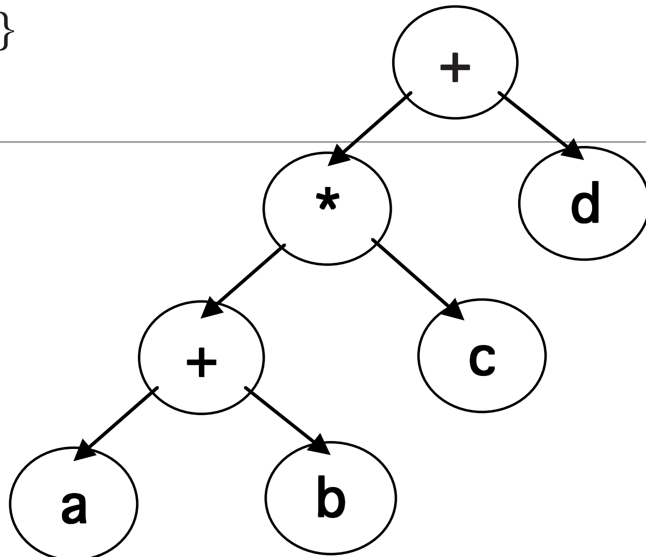
- Die **Preorder-Traversierung** eines Kantorowitsch-Baumes ergibt die klammerfreie **Präfix-Notation** des entsprechenden arithmetischen Ausdrucks
  – Die Notaton heißt Präfix, weil alle Operatoren am Anfang stehen:

+ * + a b c d

Präfix-Notation des algebraischen Ausdrucks (a+b)*c + d

Postfix-Notation

```
private void postOrder(Node localRoot) {
    if(localRoot != null)   {
      postOrder(localRoot.getLeftChild());
      postOrder(localRoot.getRightChild());
      localRoot.displayNode();
    }
}
```



Kantorowitsch-Baum des algebraischen Ausdrucks (a+b)*c + d

- Die **Postorder-Traversierung** eines Kantorowitsch-Baumes ergibt die klammerfreie **Postfix-Notation** (auch polnische Notation genannt) des entsprechenden arithmetischen Ausdrucks
  - Die Notaton heißt Postfix, weil der Operator immer hinter den Operanden steht:

Was kommt als nächstes?

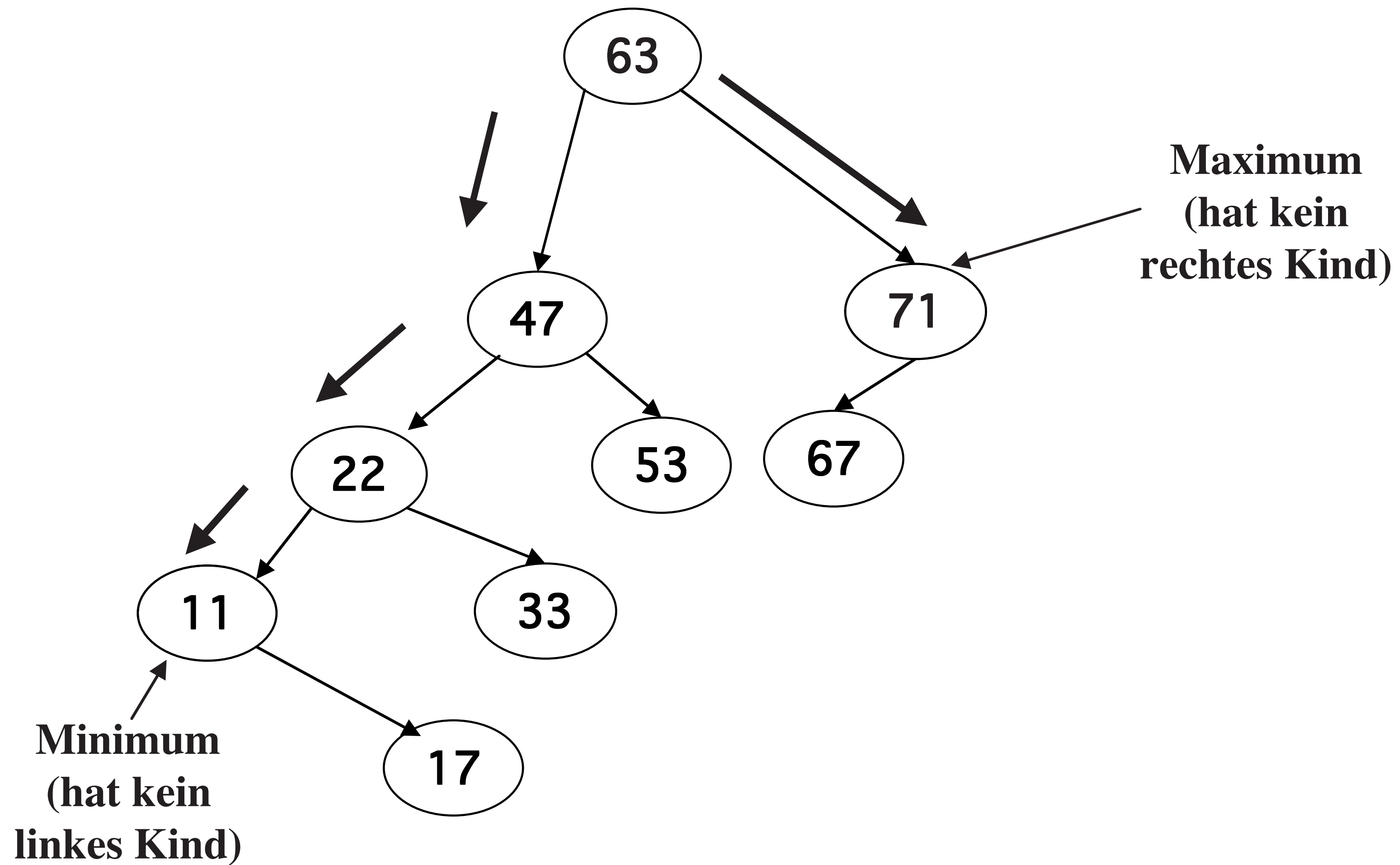
a b + c * d +

Postfix-Notation des algebraischen Ausdrucks (a+b)*c + d

Minimum und Maximum in einem binären Suchbaum

- Das Finden des Minimums (oder Maximums) in einer unsortierten Reihung hat die Komplexität O(n).
- In einem vollständig balancierten binären Suchbaum ist die Komplexität für diese Operationen $O(\log_2(n))$.
- Hier sind die Algorithmen:
  - **Finden des Minimums:**
    Wir starten bei der Wurzel und gehen immer zum linken Kind, und zwar solange, bis wir bei einem Knoten ankommen, *der kein linkes Kind mehr hat*. Dieser Knoten ist das Minimum.
  - **Finden des Maximums:**
    Wir starten bei der Wurzel und gehen immer zum rechten Kind, und zwar solange, bis wir bei einem Knoten ankommen, *der kein rechtes Kind mehr hat*. Dieser Knoten ist das Maximum.
- Die Implementation ist trivial.

Minimum und Maximum in einem binären Suchbaum

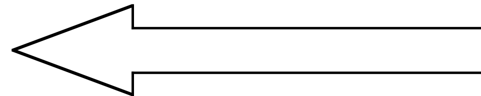


"*Neueste*" *Java-Spezifikation eines binären Suchbaums*

```
class Tree {
  private  Node root;
```

- ✓ private void inOrder(Node localRoot);
- ✓ private void preOrder(Node localRoot);
- ✓ private void postOrder(Node localRoot);
- ✓ public Node minimum();
- ✓ public Node maximum();
- ✓ boolean isEmpty();
- ✓ public void insert(int key);
- ✓ public Node find (int key);

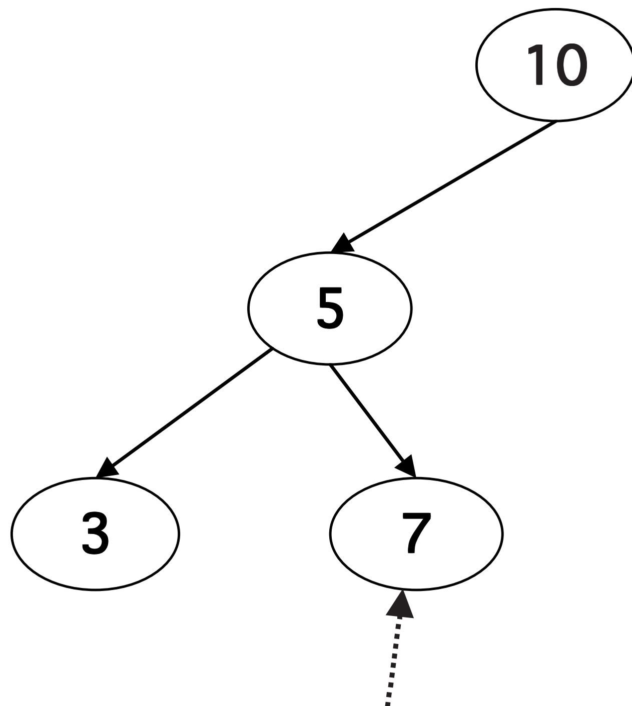
public void delete(int key);



```
} // End class Tree
```

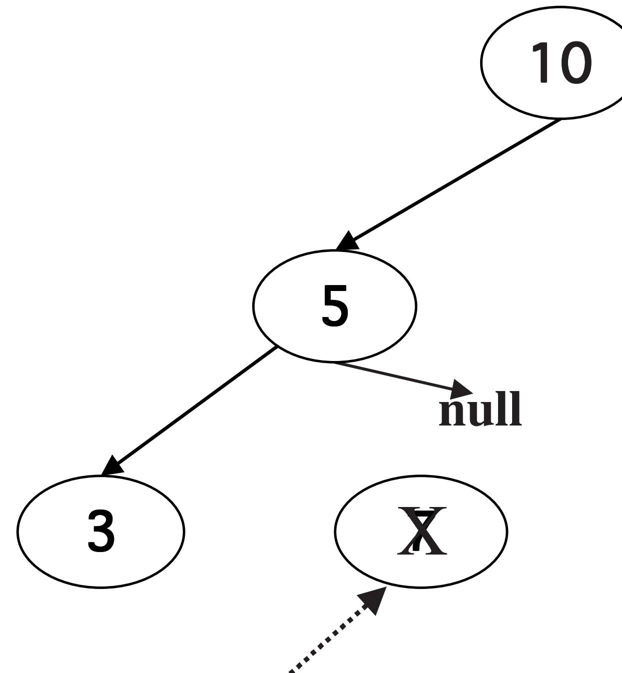
Löschen eines Knotens

Vorher



zu löschender Knoten

Nachher



Dieser Knoten wartet auf die Müllabfuhr (garbage collection)

Copyright 2001 Bernd Brügge

Löschen eines Knotens

- Das Löschen eines Knotens ist die kompliziertеste Baumoperation.
  - Viele Programmierer vermeiden deshalb die Implementation und benutzen ein boolesches Attribut **deleted**, das normalerweise **false** ist, und auf **true** gesetzt wird, wenn der Knoten gelöscht wird:

```
class Node {
    private int data;
    private bool deleted;
    private Node leftChild;
    private Node rightChild;
}
```

- Diese Lösung ist z.B. bei langlebigen Datenbanken mit vielen Knoten nicht sehr gut, da die Suche immer langwieriger wird..
- Außerdem ist das Studium des Löschens eine interessante Übung in Fallunterscheidungen….

Löschen eines Knotens ….

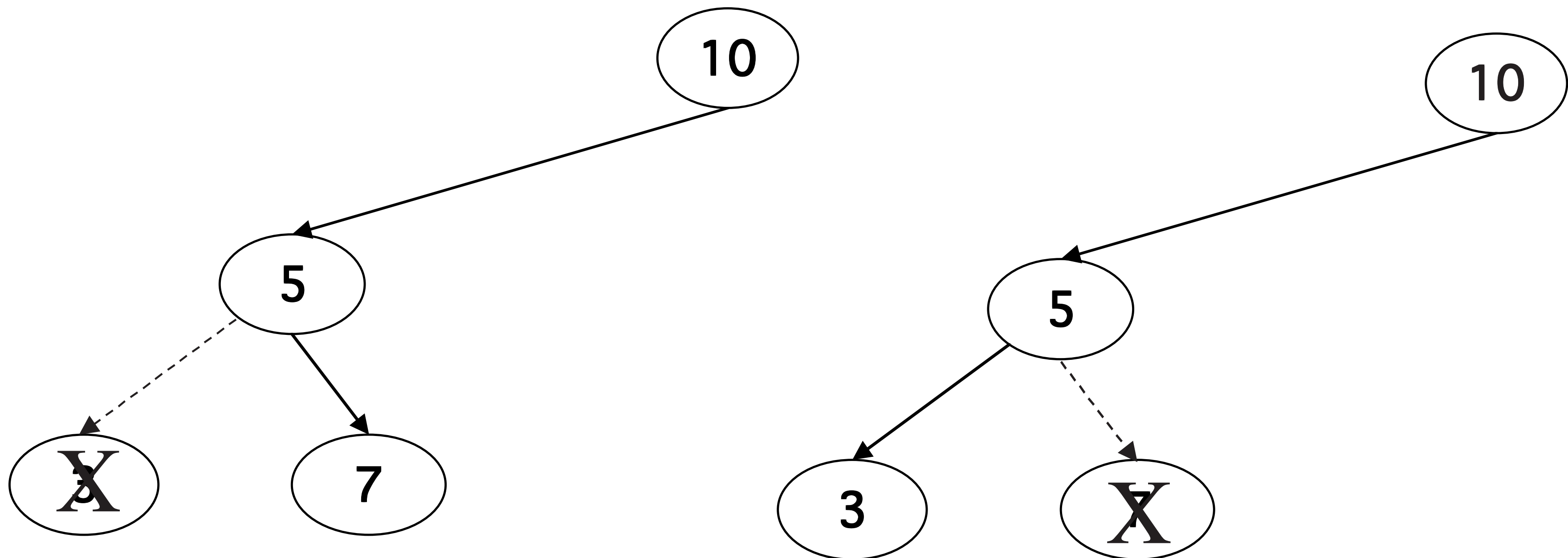
- Wir unterscheiden 3 Fälle**:**
  - – Der zu löschende Knoten ist ein Blatt **(sehr einfach)**
  - – Der zu löschende Knoten hat ein Kind **(immer noch einfach)**
  - – Der zu löschende Knoten hat 2 Kinder **(kompliziert)**.

Löschen eines Knoten: Erst einmal müssen wir den Knoten finden (x)

```
public boolean delete (int key) {          // Lösche Knoten mit gegebenen
                                           // Schlüssel in einem nicht-leeren Baum
    Node current = root;
    Node parent = root;                    // Zunächst müssen wir den Knoten suchen…
    boolean isLeftChild = true;
    while (current.getData() != key) {     // Auf geht die Suche...
      parent = current;                    // wir merken uns den Elternknoten
      if (key < current.getData() ) {      // müssen wir links gehen?
        isLeftChild = true;
        current = current.getLeftChild();
      }
      else {                               // oder rechts?
        isLeftChild = false;
        current = current.getRightChild();
      }
      if (current == null)                 // Wir haben nichts gefunden
        return false;
    } // end while
    // An diese Stelle kommen wir nur, wenn wir den Knoten gefunden haben.
```

1. Fall: Der zu löschende Knoten ist ein Blatt

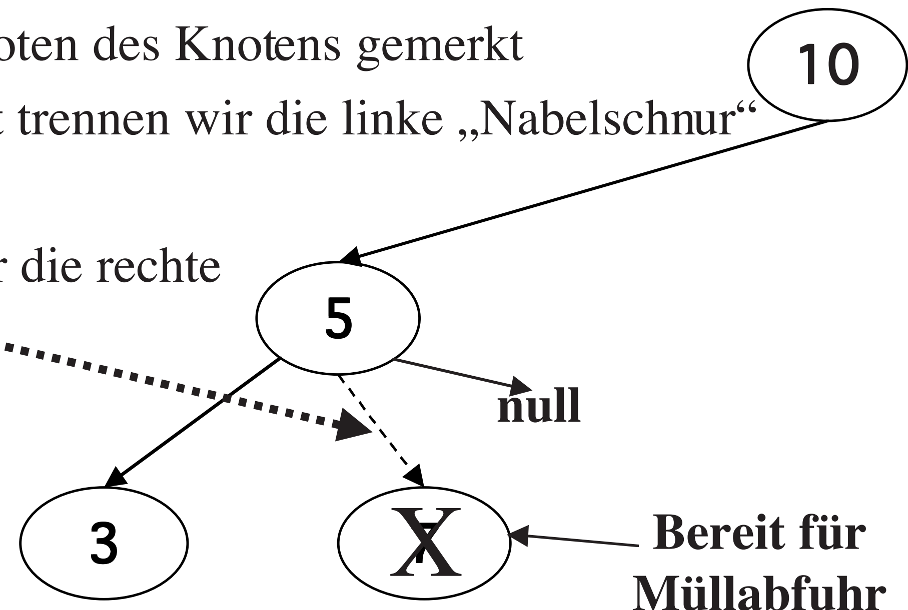
- 2 Unterfälle:
  - 1. Der zu löschende Knoten ist linkes Kind vom Elternknoten
  - 2. Der zu löschende Knoten ist rechtes Kind vom Elternknoten



Copyright 2001 Bernd Brügge

1. Fall: Löschen eines Blattes

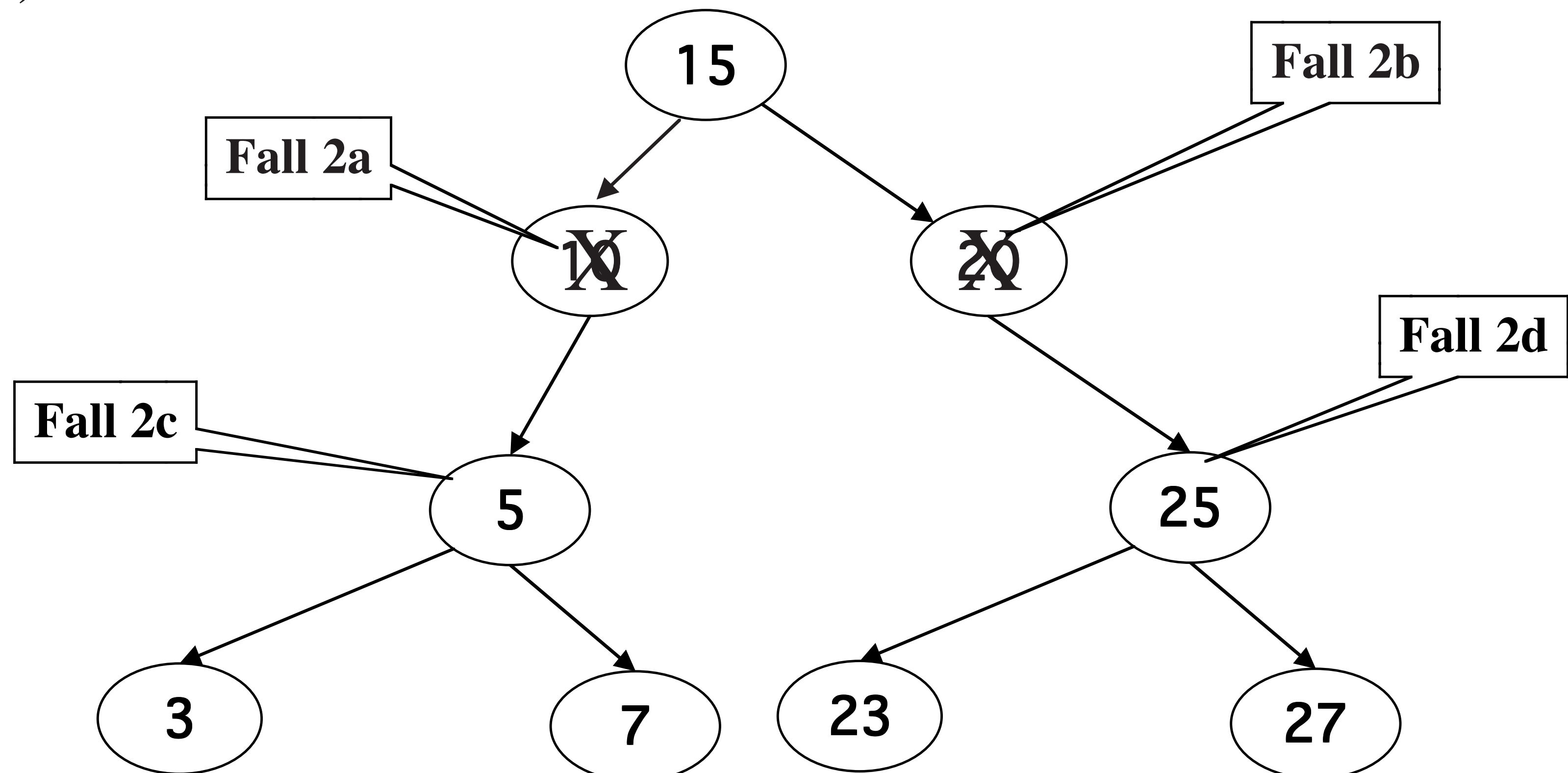
```
// Fortsetzung der delete-Methode: 1 Fall
if ((current.getLeftChild() == null) &&
   (current.rightChild==null))  {          // wir haben ein Blatt zu löschen!
    if (current == root)                   // wenn das Blatt auch die Wurzel ist,
      root = null;                         // dann ist der Baum jetzt leer
    else
    if (isLeftChild)
      // Wir haben uns ja in parent den Elternknoten des Knotens gemerkt
      parent.setLeftChild(null);           // jetzt trennen wir die linke „Nabelschnur“
    else
      parent.setRightChild(null);          // oder die rechte

} // Ende des 1. Falls
```



Copyright 2001 Bernd Brügge

2. Fall: Der zu löschende Knoten hat nur ein Kind

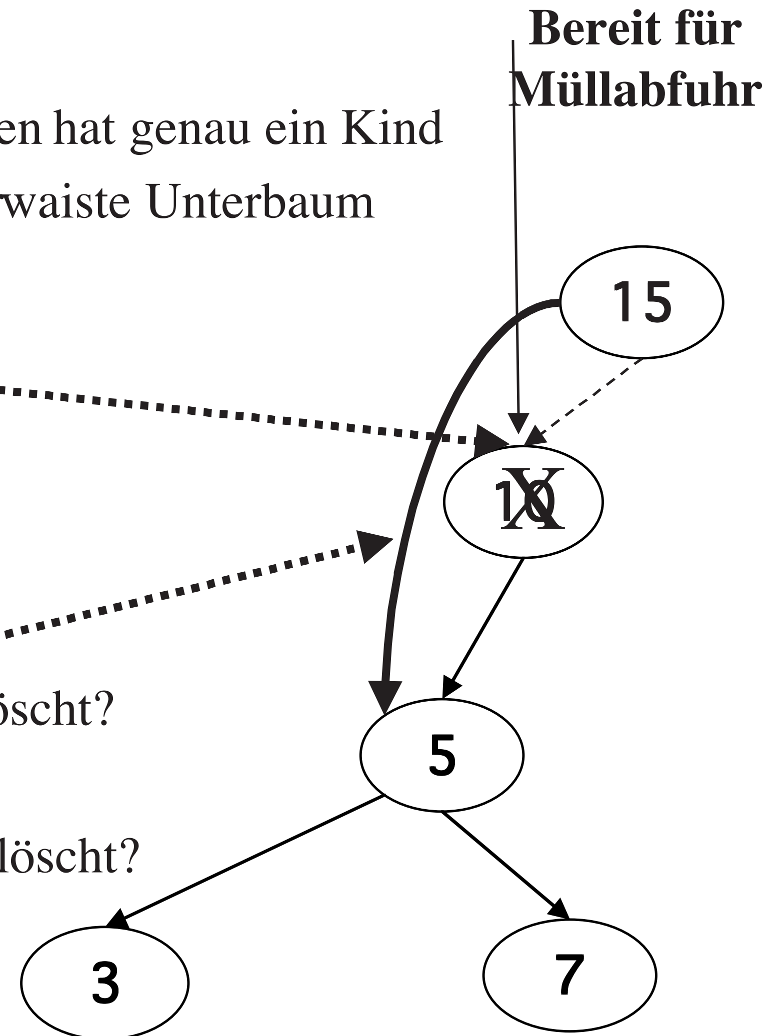
- Fallunterscheidungen:
  - 2a) Der zu löschende Knoten ist linkes Kind vom Elternknoten
  - 2b) Der zu löschende Knoten ist rechtes Kind vom Elternknoten
  - 2c) Das Kind vom zu löschenden Knoten ist links
  - 2d) Das Kind vom zu löschenden Knoten ist rechts



Copyright 2001 Bernd Brügge

2. Fall: Der zu löschende Knoten hat nur ein Kind

```
// Fortsetzung der delete-Methode:
else if  ((current.getLeftChild() == null) ||
          (current.getRightChild() == null)) { // der Knoten hat genau ein Kind
    Node child;                  // der durch das Löschen verwaiste Unterbaum
    if (current.getLeftChild() != null)  // Fall 2c
       child = current.getLeftChild();
    else
       child = current.getRightChild();  //  Fall 2d
    if (current == root)         // der verwaiste Unterbaum
       root = child;             // wird zur Wurzel
    else if (isLeftChild)        // Fall 2a: linker Knoten gelöscht?
       parent.setLeftChild(child);      // dann nach links
    else                         // Fall 2b: rechter Knoten gelöscht?
       parent.setRightChild(child);    // dann nach rechts
} // Ende des 2. Falls
```

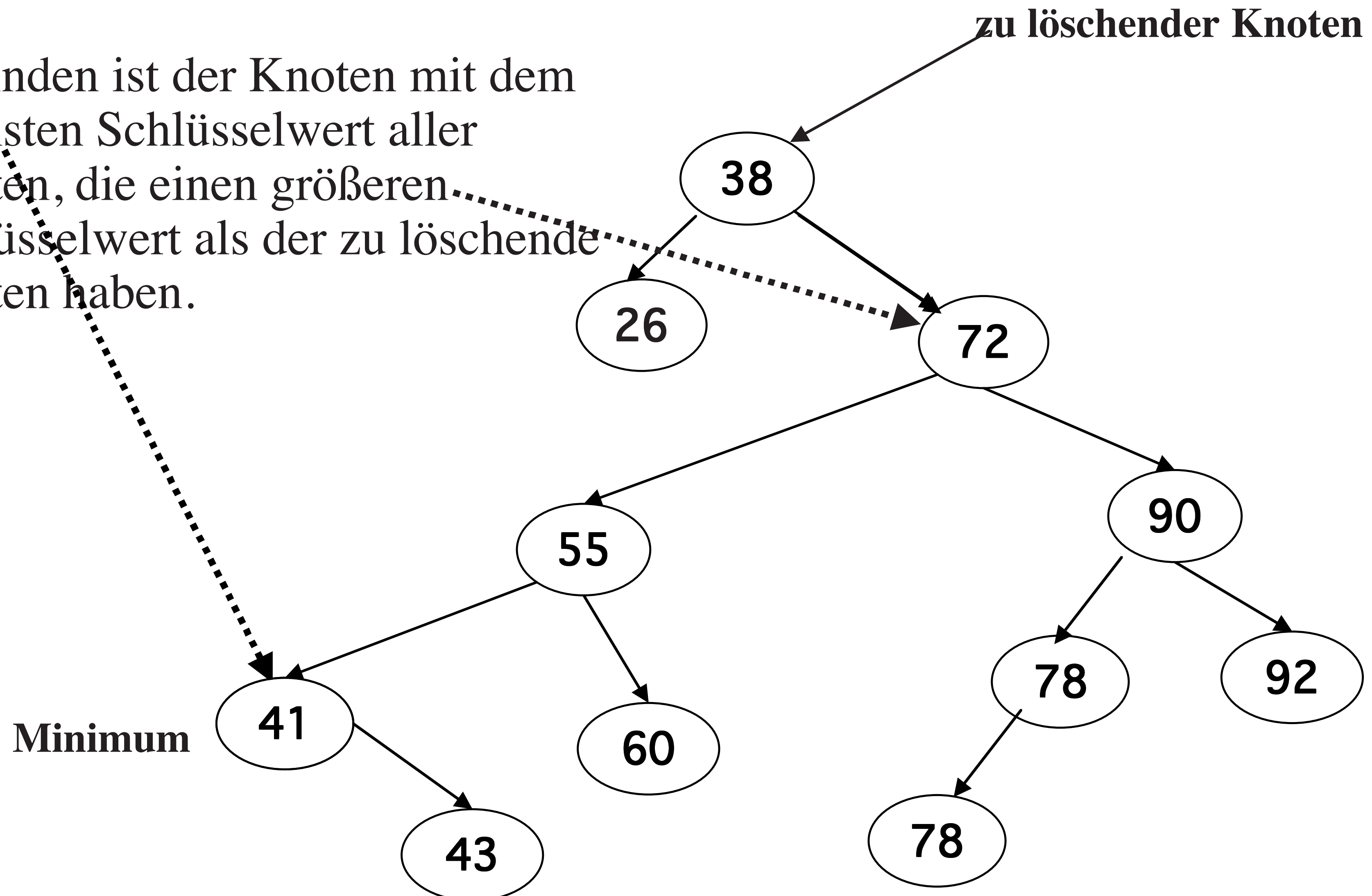


3. Fall: Der zu löschende Knoten hat 2 Kinder

```
// Fortsetzung der delete-Methode: 3. Fall
else {          // hier kommen wir an, wenn der zu löschende Knoten
                // zwei Kinder hat.
                // Diesen Fall machen wir als Übung!

  } Ende des 3. Falls
} // end delete()
```

Hinweis auf einen Algorithmus für den 3. Fall

❖ Zu finden ist der Knoten mit dem kleinsten Schlüsselwert aller Knoten, die einen größeren Schlüsselwert als der zu löschende Knoten haben.



Copyright 2001 Bernd Brügge

Abschließende Betrachtung der Komplexität von Baumoperationen für (vollständige) binäre Suchbäume

- Baumoperationen erfordern immer den Abstieg von der Wurzel und den Besuch von Knoten.
- **Analyse des durchschnittlichen Falls:**
  - In einem vollständigen binären Suchbaum ist ca. die Hälfte der Knoten auf der untersten Ebene (Blattebene).
  - Das bedeutet, dass ungefähr die Hälfte aller Suchoperationen, Einfügeoperationen und Löschoperationen immer bis auf die unterste Ebene müssen, d.h. $V(n) = O(\log_2(n))$
- Die meisten Baumoperationen haben also im durchschnittlichen Fall die Komplexität $O(\log_2(n))$.

Anzahl der Knoten	Anzahl der Ebenen
1023	10
…	...
32,767	15
….	…
1.048.575	20
…	…
33.554.432	25
…	…
1.073.741.824	30

Vergleich der Komplexität von Operationen auf Reihungen, Listen und Bäumen

Daten Struktur	Suchen	Einfügen	Löschen	Sortieren
Unsortierte Reihung	O(n)	O(1)	O(n)	$O(n \log_2(n))$
Sortierte Reihung	$O(\log_2(n))$	O(n)	O(n)	---
Verkettete Liste	O(n)	O(1)	O(n)	$O(n \log_2(n))$
Sortierte Liste	O(n)	O(n)	O(n)	---
Binärbaum	O(n)	O(1)	O(n)	$O(n \log_2(n))$
Binärer Suchbaum	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$	---

- Die meisten typischen Datenoperationen sind also effizient mit einem binären Suchbaum (d.h. einem *sortierten* Binärbaum) als Datenstruktur zu realisieren.