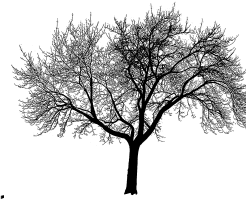


# Informatik II Bäume



G. Zachmann  
Clausthal University, Germany  
[zach@tu-clausthal.de](mailto:zach@tu-clausthal.de)



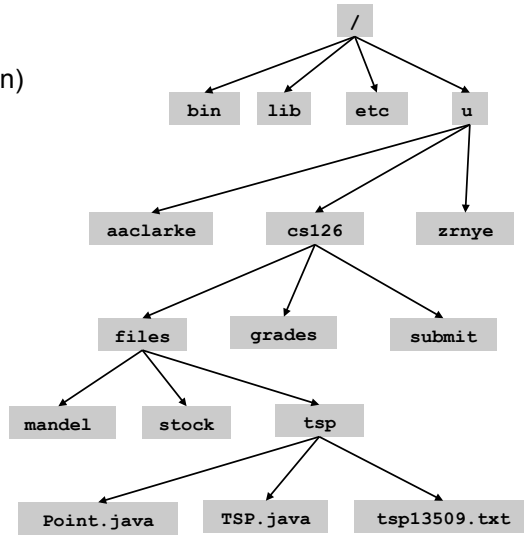
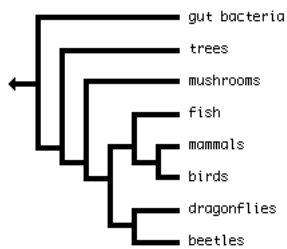
## Beispiele



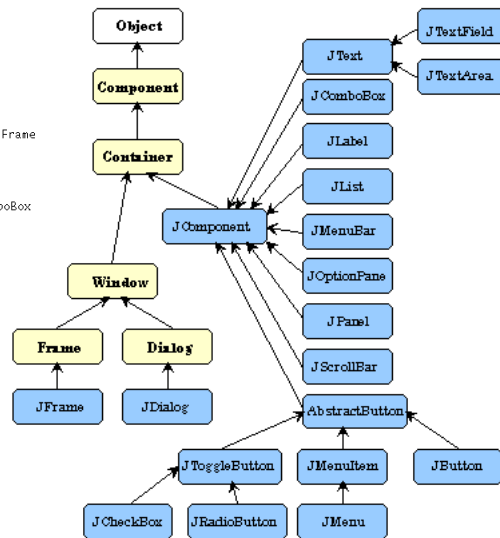
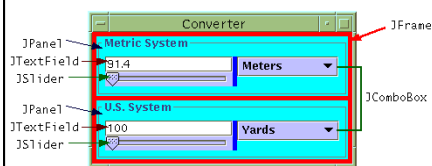
- Stammbaum



- Unix file hierarchy
- Stammbaum (Evolution)

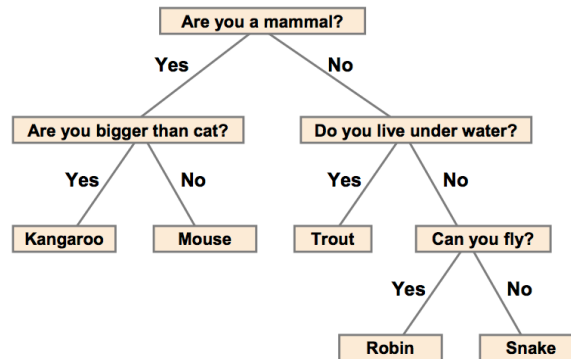


- Klassenhierarchie (z.B. in GUI-Libraries)



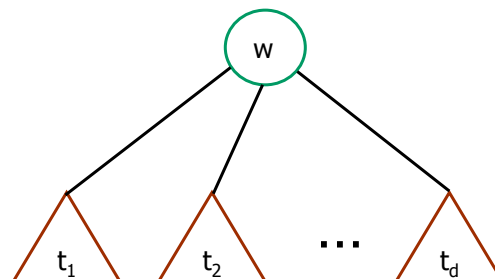
- Beispiel für Entscheidungsbaum:

**Example 1** [ binary tree for decision process ]



## Definition

- Rekursive Definition:  
Ein **Baum** ist entweder
  - ein **einzelner Knoten**, oder
  - ein als **Wurzel** dienender Knoten  $w$ , der mit einer Menge von Bäumen  $t_1, \dots, t_d$  verbunden ist.



## Terminologie bei Bäumen

- **Baum** = Menge von **Knoten** und **Kanten**
- **Knoten** = repräsentiert beliebiges Objekt
- **Kante** = Verbindung zwischen zwei Knoten
- **Pfad** = Folge unterschiedlicher, durch Kanten verbundener Knoten
- **Wurzel** = ausgezeichnete Knoten, der keine Vorgänger hat
- **Blatt** = Knoten ohne Nachfolger
- **Vater** = Vorgänger eines Knotens
- **Kind** = Nachfolger eines Knotens
- **Innerer Knoten** = Nicht-Blatt
- **Geschwister** = Knoten mit gleichem Vater

G. Zachmann Informatik 2 — SS 11 Bäume 7

- **Grad eines Knotens** = Anzahl von direkten Söhnen
- **Ordnung** = maximaler Grad aller Knoten ("Baum der Ordnung n" = "n-ary tree")
- **geordnet** → Reihenfolge unter Geschwistern (gemäß irgend einer Ordnungsrelation)
- **Teilbaum** = Knoten mit all seinen Nachfolgern (direkte & indirekte)
- **Level** eines Baumes:

G. Zachmann Informatik 2 — SS 11 Bäume 8



## Baumtiefe



- Definition: **Tiefe eines Knotens**
  - Länge des Pfades von der Wurzel zu dem Knoten
  - Ist eindeutig, da es nur einen Pfad bei Bäumen gibt
  - Dabei zählt man die Knoten entlang des Pfades
    - Wurzel = Tiefe 1 (manchmal auch Tiefe 0)
    - 1. Schicht = Tiefe 2, etc.
- Definition: **Tiefe / Höhe eines Baumes**
  - leerer Baum: Tiefe 0
  - ansonsten: Maximum der Tiefe seiner Knoten



## Eigenschaften

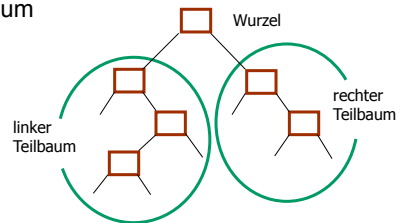


1. Von der Wurzel gibt es zu jedem Knoten genau einen Pfad
2. Für je zwei verschiedene Knoten existiert genau ein Pfad, der sie verbindet.
  - ⇒ jeder beliebige Knoten kann Wurzel sei
3. Ein Baum mit  $n$  Knoten hat  $n-1$  Kanten
  - Beweis von Eigenschaft 3 durch Induktion:
    - Induktionsanfang:  $n=1 \rightarrow 0$  Kanten
    - Induktionsschritt:  $n>1$ 
      - Wurzel hat  $k$  Teilbäume, mit je  $n_j$  Knoten, wobei  $n_1 + \dots + n_k = n-1$
      - Per Induktionsannahme hat jeder Teilbaum  $n_j-1$  Kanten
      - Zusammen also  $n-1-k$  Kanten
      - Dazu  $k$  Kanten von den Wurzeln der Teilbäume zur Wurzel  $\rightarrow$  Behauptung



## Binärbäume

- Wichtiger Spezialfall: jeder Knoten hat höchstens zwei Kinder
  - = Baum der Ordnung 2 = Binärbaum



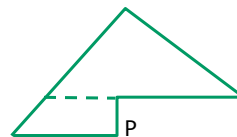
- Wichtige (einfache) Eigenschaft:  
In einem Baum, in dem jeder Knoten entweder genau 2 Kinder (= innerer Knoten) hat oder keines (= Blatt), gilt:  
 $\# \text{ Blätter} = \# \text{ innerer Knoten} + 1$



## Vollständige Bäume

- Definition:  
Ein **vollständiger Baum** ist ein binärer Baum B mit folgenden Eigenschaften:
  - für jedes  $k$  mit  $k < \text{Tiefe}(B)$  gilt, die  $k$ -te Schicht ist voll besetzt; und,
  - die letzte Schicht ist von links nach rechts bis zu einem Knoten P besetzt



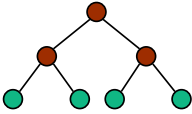
- **Achtung:** manchmal abweichende Def., wonach **jede** Schicht voll besetzt sein muß!



- Die Höhe eines vollständigen binären Baumes mit  $n$  Knoten ist  
 $\lceil \log_2(n + 1) \rceil$

## Maximale Anzahl Knoten

Wie groß ist die maximale Anzahl der Knoten eines vollständigen Baumes gegebener Höhe?

Baum	Höhe	Anzahl innere Knoten	Anzahl Blätter
	1	0	1
	2	1	2
	3	3	4
	4	7	8
	h	$2^{h-1}-1$	$2^{h-1}$
		$\underbrace{\hspace{10em}}_{\Sigma = 2^h - 1}$	

G. Zachmann Informatik 2 — SS 11
Bäume 13

- **Satz:** Ein maximal vollständiger binärer Baum der Höhe  $h$  enthält  $2^{h-1}$  Blätter und  $2^{h-1}-1$  inneren Knoten und  $2^h-1$  Knoten. ( $h \geq 1$ )
- **Beweis :**
  1. Induktionsanfang:  $h=1$   
 Der Baum besteht nur aus der Wurzel, die auch das einzige Blatt ist:  
 $2^{1-1} = 2^0 = 1$  Blatt  
 $2^{1-1} = 2 - 1 = 1$  Knoten
  2. Induktionsschritt:  $h \rightarrow h' = h + 1$   

Höhe $h$	Höhe $h' = h + 1$
$2^{h-1}$ Blätter	$2 \cdot 2^{h-1} = 2^h = 2^{h'-1}$ Blätter $\rightarrow$ Beh.
$2^{h-1}$ Knoten	$2^{h-1}$ innere Knoten + $2^h$ Blätter = $2^{h+1}-1 = 2^{h'-1}$

G. Zachmann Informatik 2 — SS 11
Bäume 14

## Nummerierung der Knoten

- Von oben nach unten, von links nach rechts, beginnend bei 1
- Beobachtung:
  - ein Knoten  $i$  hat immer die Nachfolger  $2i$  und  $2i+1$
  - Vater hat immer die Nummer  $\lfloor i/2 \rfloor$
- Fazit: Knoten können in einem Array abgelegt werden
- Achtung:
  - Array-Indizierung beginnt bei 0, aber Knoten-Nummerierung bei 1!
  - Knoten  $i$  = Array-Element  $A[i-1]$
- Frage: funktioniert ein ähnliches Schema auch, wenn man die Knoten selbst mit 0 beginnend nummeriert?

G. Zachmann Informatik 2 — SS 11 Bäume 15

## Speichern eines vollständigen Baumes im Array

- Aus Sicht des Knotens  $i$  (Adresse ist **nicht** als Referenz im Knoten gespeichert)
 

Knoten  $i$ : **A[i-1]**

Nummerierung beginnt bei 1

bzw. mit  $j=i-1$

**A[j]**

Nummerierung beginnt bei 0
- Alternative:  $A[0]$  frei lassen,  $A[1]$  speichert Knoten Nummer 1

G. Zachmann Informatik 2 — SS 11 Bäume 16



## Implementierung eines Baumes (in Python)

- Ein Knoten hat (mind.) 3 Instanzvariablen:
  - Eine Referenz zu item (*payload data*)
  - Eine Referenz zum linken Unterbaum
  - Eine Referenz zum rechten Unterbaum

```

class Tree:
    def __init__( self, item,
                  left = None,
                  right = None ):
        self.item = item
        self.left = left
        self.right = right
  
```

G. Zachmann Informatik 2 — SS 11 Bäume 18

## Anwendung: Parse-Tree von Ausdrücken

- Parse-Tree = Abstrakte Repräsentation von Ausdrücken
- Anwendung: Compiler, Computerlinguistik, ...
- Beispiel für den Aufbau:

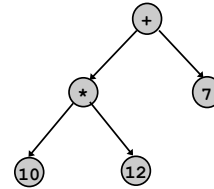
```

a = Tree(10)
b = Tree(12)
c = Tree(" * ", a, b)
d = Tree(7)
e = Tree(" + ", c, d)
  
```

G. Zachmann Informatik 2 — SS 11 Bäume 19

## Auswertung eines Ausdruckes mit Hilfe von Parse-Trees

- Auswertung eines Parse-Tree:
  - Wenn Knoten ein Integer ist → Wert = Integer
  - Sonst, werte rekursiv beide Unterbäume aus und gebe die Summe oder das Produkt aus (je nach Operator, der im Knoten gespeichert ist)



$((10 * 12) + (7)) = 127$

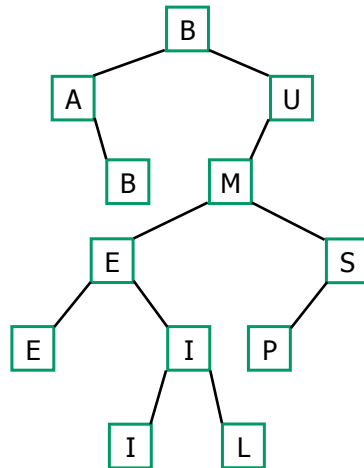
```
class Tree:
    def eval(self):
        if self.item == "+":
            return self.left.eval() + self.right.eval()
        elif self.item == "*":
            return self.left.eval() * self.right.eval()
        else:
            return self.item
```

## Postorder-Traversierung

- Reihenfolge:
  - Traversiere den linken Teilbaum (in Postorder)
  - Traversiere den rechten Teilbaum (in Postorder)
  - "Besuche" den Knoten selbst = führe die Operation durch
- Implementierung :

```
class Tree (cont'd) ...
    def postorder( self, params):
        if self.left:
            self.left.postorder(params)
        if self.right:
            self.right.postorder(params)
        do something with node self
```

- Beispiel:



Postorder Traversierung:

BAEILIEPSMUB

## Baumtraversierungen: Preorder-Traversierung

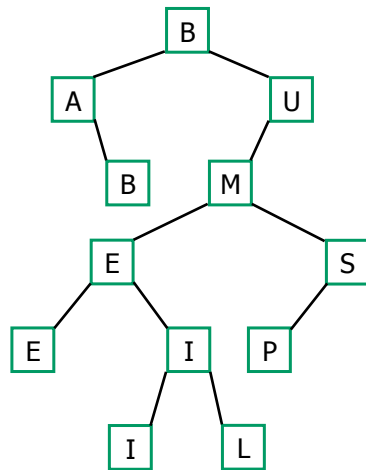
- Reihenfolge:

1. Besuche die Wurzel, führe Operation an Wurzel durch
2. Traversiere den linken Teilbaum (in Preorder-Reihenfolge)
3. Traversiere den rechten Teilbaum (in Preorder)

- Implementierung :

```
class Tree (cont'd) ...  
  
def preorder( self, params ) :  
    do something with node self  
    if self.left != None:  
        self.left.preorder(params)  
    if self.right != None :  
        self.right.preorder(params)  
  
tree.preorder( params )
```

▪ Beispiel



Preorder Traversierung:

BABUMEEIILSP

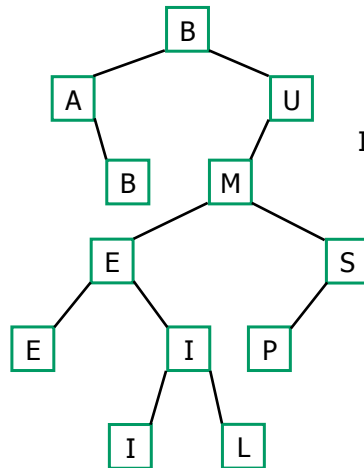
## Inorder-Traversierung

▪ Reihenfolge:

1. Traversiere den linken Teilbaum (in Inorder)
2. "Besuche" den Knoten, **Operation darauf durchführen**
3. Traversiere den rechten Teilbaum (in Inorder)

```
def inorder(self, params):  
    if self.left:  
        self.left.inorder(params)  
    do something with node self  
    if self.right:  
        self.right.inorder(params)
```

▪ Beispiel

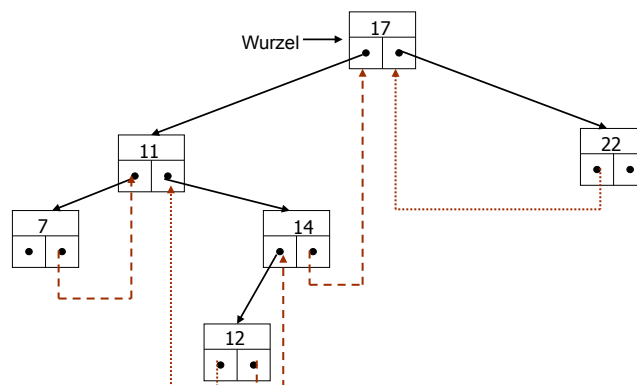


Inorder Traversierung:

ABBEEIILMPSU

▪ Nicht-rekursive Varianten mit *threaded trees*

- Rekursion kann vermieden werden, wenn man anstelle der Null-Referenzen sogenannte *thread pointer* auf den in-order Vorgänger bzw. den in-order Nachfolger verwendet:

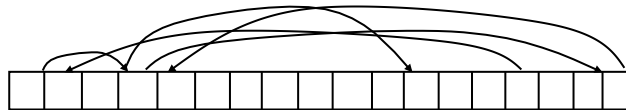




## Lokalität und Bäume



- Einfügen, Löschen und Umhängen von Knoten führen zu Adressfolgen, die keinerlei Lokalität aufweisen



- relativ harmlos, falls sich alle Daten im Hauptspeicher befinden
  - aber: schlechte Ausnutzung des Caches
- Katastrophe, falls Daten auf Festplatte oder Magnetband
  - siehe 2-3-4-Bäume, B-Bäume, Rot-Schwarz-Bäume



## Depth-First Search allgemein



## Levelorder-Traversierung (aka breadth-first search, BFS)

### Reihenfolge:

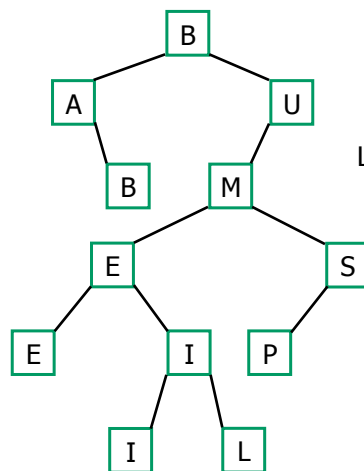
- besuche die Knoten schichtweise
  - zuerst die Wurzel
  - dann die Wurzeln des linken und rechten Teilbaums
  - etc.

### Algorithmus

- kann nicht rekursiv angegeben werden!
- erfordert eine Zwischenspeicherung der Knoten in einer Queue

```
def levelorder(self, params):  
    q = Queue()  
    q.enqueue(self)  
    while not q.empty():  
        n = q.dequeue()  
        if n != None:  
            do something with node n  
            q.enqueue(n.left)  
            q.enqueue(n.right)
```

### Beispiel



Levelorder Traversierung:

BAUBMESEIPIIL



## Exkurs: Visitor Pattern



- Oftmals muß man verschiedene Operationen (= *do something with node* im Code) auf dem Baum ausführen
- Bisherige (naive) Implementierung würde jedesmal eine neue `preorder`-Methode benötigen
- **Auslagerung** der Operationen außerhalb der Traversierungsmethoden ist die einfachste Form des sog. **Visitor Patterns**
- Klasse `DoSomethingWithANode` heißt **Visitor**, weil diese jeden Knoten "besucht"
- Methode `preorder/postorder` heißt **Mapper**, weil diese die Operation (`DoSomethingWithANode.visit`) auf jeden Knoten anwenden (mappen), und wissen, in welcher Reihenfolge dies geschehen soll



## Beispiel



- Ansatz: diese Operation in eine sog. **Visitor-Klasse** verpacken, z.B.

```
class DoSomethingWithANode(object):
    def __init__( self, params ):
        . . .
    def visit( self, treenode ):
        do something with treenode.item
```





## Beispiel

- Instanz davon kann dann der allgemeinen Traversierungsmethode als Parameter übergeben werden:

```
class Tree (object):  
    ...  
    def preorder( self, visitor ):  
        visitor.visit(self)  
        if self.left != None:  
            self.left.preorder(visitor)  
        if self.right != None :  
            self.right.preorder(visitor)  
  
doIt = DoSomethingWithANode( params )  
tree.preorder( doIt )
```



- Vorteil von Visitor-Klasse im Gegensatz zu einer Visitor-Funktion:  
man kann damit die Operationen sehr einfach parametrisieren,  
z.B.

```
class PrintNode(object):  
    def __init__(self, tolower):  
        self.tolower = tolower  
    def visit(node):  
        s = str( node.getData() ) # make sure we  
        if self.tolower:          # get a string  
            s = s.lower()  
        print s
```

```
r = root of tree  
v = PrintNode(false)  
r.preorder(v) # print all nodes in preorder  
v = PrintNode(true)  
r.preorder(v) # again, but all in lowercase
```

- Vorteil der Trennung in Visitor-Klasse und Baum-Traversierungsmethode:
  - man muß Traversierungsroutine nur 1x schreiben
  - man kann trotzdem beliebige Operationen ausführen lassen
- Beispiel: andere Operation, z.B. alle Knoten in eine Liste sammeln

```
class CollectNodes(object):  
    def __init__(self):  
        self.nodes = []  
    def visit(self, node):  
        self.nodes.append( node.getData() )
```

```
v = CollectNodes()  
root.preorder(v)  
print v.nodes
```