Informatik I: Einführung in die Programmierung

9. Bäume

Albert-Ludwigs-Universität Freiburg

Peter Thiemann

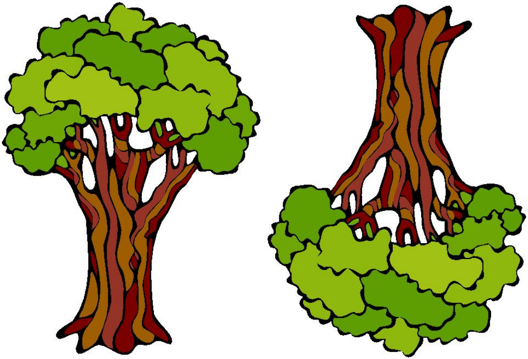
27. November 2018

1 Der Baum

- Definition
- Terminologie
- Beispiele

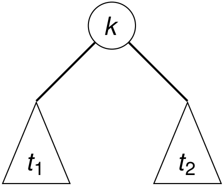
Bäume in der Informatik

- Bäume sind in der Informatik allgegenwärtig.
- Gezeichnet werden sie meistens mit der Wurzel nach oben!



Bäume in der Informatik - Definition

- Induktive Definition:
  - Der leere Baum ist ein Baum.
  - Wenn $t_1, \dots, t_n$, $n \geq 0$ disjunkte Bäume sind und k ein Knoten, der nicht in $t_1, \dots, t_n$ vorkommt, dann ist auch die Struktur bestehend aus der Wurzel k mit zugeordneten Teilbäumen $t_1, \dots, t_n$ ein Baum.
  - Nichts sonst ist ein Baum.
  - Beispiel:

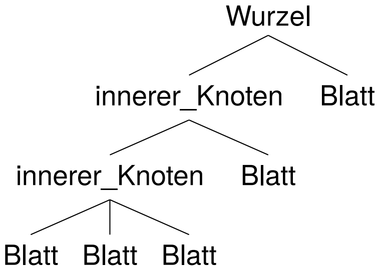


□

- Beachte: Bäume können auch anders definiert werden und können auch eine andere Gestalt haben (z.B. ungewurzelt).

Terminologie I

- Alle Knoten, denen keine Teilbäume zugeordnet sind, heißen Blätter.
- Knoten, die keine Blätter sind, heißen innere Knoten.



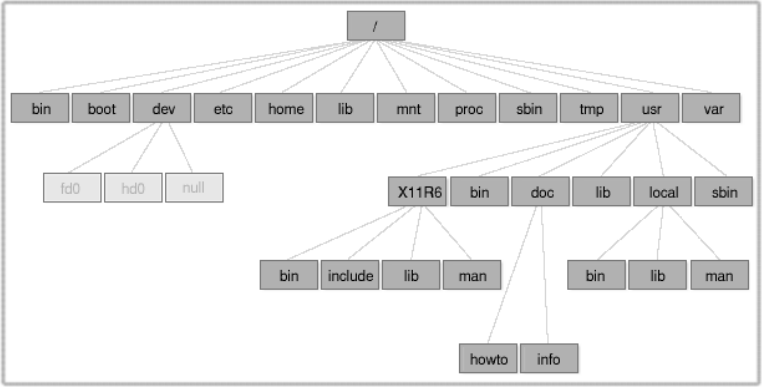
- Die Wurzel kann also ein Blatt sein (keine weiteren Teilbäume) oder ein innerer Knoten.

Terminologie II

- Wenn k_1 ein Knoten und k_2 die Wurzel eines zugeordneten Teilbaums ist, dann gilt:
  - k_1 ist Elternknoten von k_2,
  - k_1 sowie der Elternknoten von k_1 sowie dessen Elternknoten usw. sind Vorgänger von k_2.
  - k_2 ist Kind von k_1.
  - Alle Kinder von k_1, deren Kinder, usw. sind Nachfolger von k_1.
- Bäume sind oft markiert. Die Markierung weist jedem Knoten eine Marke zu.
- Formal: Wenn K die Knotenmenge eines Baums ist und M eine Menge von Marken, dann ist die Markierung eine Abbildung $\mu : K \to M$.

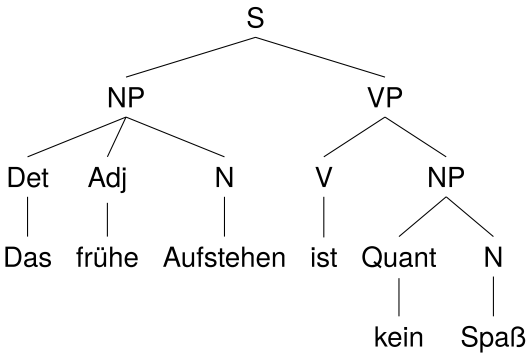
Beispiel: Verzeichnisbaum

In Linux (und anderen Betriebssystemen) ist die Verzeichnisstruktur im Wesentlichen baumartig.



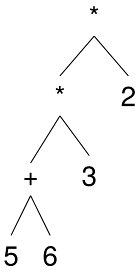
Beispiel: Syntaxbaum

Wenn die Struktur einer Sprache mit Hilfe einer formalen Grammatiken spezifiziert ist, dann kann der Satzaufbau durch sogenannte Syntaxbäume beschrieben werden.



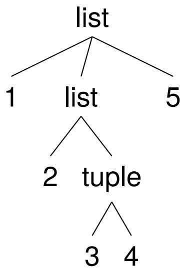
Beispiel: Ausdrucksbaum

- Bäume können arithmetische (und andere) Ausdrücke so darstellen, dass ihre Auswertung eindeutig (und einfach durchführbar) ist, ohne dass Klammern notwendig sind.
- Beispiel: $(5+6)*3*2$
- Entspricht: $((5+6)*3)*2$
- Operatoren als Markierung innerer Knoten, Zahlen als Markierung der Blätter:



Beispiel: Listen und Tupel als Bäume

- Jede Liste und jedes Tupel kann als Baum angesehen werden, bei dem der Typ die Knotenmarkierung ist und die Elemente die Teilbäume sind.
- Beispiel: `[1, [2, (3, 4)], 5]`



2 Binärbäume

- Repräsentation
- Beispiel
- Funktionen auf Bäumen
- Baumeigenschaften
- Traversierung

Der Binärbaum

- Der Binärbaum ist ein Spezialfall eines Baumes.
- Ein Binärbaum ist entweder leer oder besteht aus einem (Wurzel-) Knoten und zwei Teilbäumen.
- Für viele Anwendungsfälle angemessen.
- Funktionen über solchen Bäumen sind einfach definierbar.

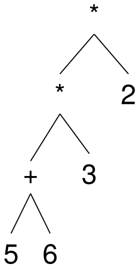
Binärbäume durch Objekte repräsentieren

- Der leere Baum wird durch `None` repräsentiert.
- Jeder andere Knoten wird durch ein `Node`-Objekt repräsentiert.
- Das Attribut `mark` enthält die Markierung.
- Das Attribut `left` enthält den linken Teilbaum.
- Das Attribut `right` enthält den rechten Teilbaum.
- Beispiele:
  - Der Baum bestehend aus dem einzigen Knoten mit der Markierung 8: `Node (8, None, None)`
  - Der Baum mit Wurzel '+', linkem Teilbaum mit Blatt 5, rechtem Teilbaum mit Blatt 6:
    `Node('+', Node(5, None, None), Node(6, None, None))`

Baumobjekte

```
class Node:
    def __init__(self, mark, left, right):
        self.mark = mark
        self.left = left
        self.right = right
```

Beispiel: Der Ausdrucksbaum



wird folgendermaßen mit `Node` Objekten dargestellt:

```
Node ('*", Node ('*', Node ('+', Node (5, None, None),
                                 Node (6, None, None)),
                      Node (3, None, None)),
           Node (2, None, None))
```

Funktionsgerüst

```
def tree_str (tree : Node) -> string:
    if tree is None:
        return "fill␣in"
    else:
        l_str = tree_str (tree.left)
        r_str = tree_str (tree.right)
        return "fill␣in"
```

- Node Objekte enthalten selbst wieder Node Objekte (oder None) in den Attributen left und right.
- Zum Ausdrucken eines Node Objekts müssen auch die enhaltenen Node Objekte ausgedruckt werden.
- Daher ist tree_str rekursiv, d.h. es wird in seiner eigenen Definition aufgerufen!

Drucken von Bäumen erfolgt rekursiv

UNI FREIBURG

- Die rekursiven Aufrufe `tree_str (tree.left)` und `tree_str (tree.left)` erfolgen auf den Kindern des Knoten.
- Ergibt sich zwangsläufig aus der induktiven Definition!
- Rekursive Aufrufe auf den Teilbäumen sind Teil des Funktionsgerüsts, sobald eine baumartige Struktur bearbeitet werden soll.

- Die Alternative "`tree is None`" ergibt sich daraus, dass ein `tree` entweder `None` oder ein `Node`-Objekt ist.

- Alle Funktionen auf Binärbäumen verwenden dieses Gerüst.

Drucken von Bäumen

Funktionsdefinition


```
def tree_str (tree : Node) -> string:
    if tree is None:
        return "None"
    else:
        return ("Node("
            + repr(tree.mark) + ",␣"
            + tree_str (tree.left) + ",␣"
            + tree_str (tree.right) + ")")
```

Tiefe von Knoten, Höhe und Größe von (Binär-)Bäumen

- Die Tiefe eines Knotens k (Abstand zur Wurzel) ist
  - 0, falls k die Wurzel ist,
  - $i+1$, wenn i die Tiefe des Elternknotens ist.
- Die Höhe eines Baumes ist die maximale Tiefe über alle Blätter:
  - -1 für den leeren Baum,
  - $m+1$, wenn m die maximale Höhe aller der Wurzel zugeordneten Teilbäume ist.
- Die Größe eines Baumes ist die Anzahl seiner Knoten.
  - 0 für den leeren Baum,
  - $s+1$, wenn s die Summe der Größen der Teilbäume ist.

Induktive Definition von Höhe und Größe von Binärbäumen

$$
\mathit{height}(\mathit{tree}) = \begin{cases} -1, & \text{if } \mathit{tree} \text{ is empty} \\ 1 + \max(\quad \mathit{height}(\mathit{tree.left}), & \\ \qquad\qquad \mathit{height}(\mathit{tree.right})), & \text{otherwise.} \end{cases}
$$

$$
\mathit{size}(\mathit{tree}) = \begin{cases} 0, & \text{if } \mathit{tree} \text{ is empty;} \\ 1 \quad + \mathit{size}(\mathit{tree.left}) & \\ \qquad + \mathit{size}(\mathit{tree.right})), & \text{otherwise.} \end{cases}
$$

Funktionen für Höhe und Größe

Höhe und Größe von Binärbäumen

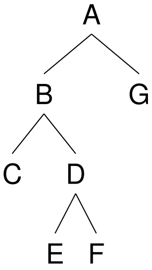
```
def height(tree):
    if (tree is None):
        return -1
    else:
        return(max(height(tree.left),
                   height(tree.right)) + 1)

def size(tree):
    if (tree is None):
        return 0
    else:
        return(size(tree.left)
            + size(tree.right) + 1)

tree = Node('*', Node('+', Node(6, None, None), Node(5,
            Node(1, None, None))
```

Traversierung von Bäumen

- Oft sollen alle Knoten eines Baumes besucht und bearbeitet werden.
- 3 Vorgehensweisen (Traversierungen) sind üblich:
  - Pre-Order (Hauptreihenfolge): Zuerst der Knoten selbst, dann der linke, danach der rechte Teilbaum
  - Post-Order (Nebenreihenfolge): Zuerst der linke, danach der rechte Teilbaum, zum Schluss der Knoten selbst
  - In-Order (symmetrische Reihenfolge): Zuerst der linke Teilbaum, dann der Knoten selbst, danach der rechte Teilbaum
- Manchmal auch Reverse In-Order (anti-symmetrische Reihenfolge): Rechter Teilbaum, Knoten, dann linker Teilbaum
- Auch das Besuchen nach Tiefenlevel von links nach rechts (level-order) ist denkbar

Pre-Order Ausgabe eines Baums

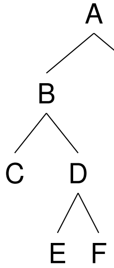
- Gebe Baum *pre-order* aus



- Ausgabe: A B C D E F G

Post-Order Ausgabe eines Baums

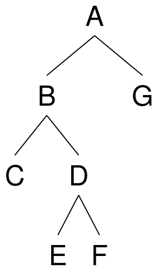
- Gebe Baum *post-order* aus



- Ausgabe: C E F D B G A

In-Order Ausgabe eines Baums

- Gebe Baum *in-order* aus.



- Ausgabe: C B E D F A G

Post-order Programm

Post-Order Printing

```
def postorder(tree):
    if tree is None:
        pass
    else:
        postorder(tree.left)
        postorder(tree.right)
        print(tree.mark)
def leaf (m):
    return Node (m, None, None)
tree = Node('*', Node('+', leaf(6), leaf(5)),
                 leaf(1))
postorder(tree)
```

Die *post-order* Ausgabe eines arithmetischen Ausdrucks heißt auch umgekehrt polnische oder Postfix-Notation (HP-Taschenrechner, Programmiersprachen *Forth* und *PostScript*)

3 Suchbäume

- Definition
- Suche
- Aufbau

Suchbäume

- *Suchbäume* realisieren Wörterbücher und dienen dazu, Items schnell wieder zu finden.
- Ein Suchbaum ist ein binärer Baum, der die Suchbaumeigenschaften erfüllt:
  - Alle Markierungen im linken Teilbaum sind *kleiner* als die aktuelle Knotenmarkierung, alle Markierungen im rechten Teilbaum sind *größer*.
- Suchen nach einem Item *m*: Vergleiche mit Markierung im aktuellem Knoten,
  - wenn gleich, stoppe und gebe True zurück,
  - wenn *m* kleiner ist, suche im linken Teilbaum,
  - wenn *m* größer ist, such im rechten Teilbaum.
- Suchzeit ist proportional zur Höhe des Baums! Im besten Fall *logarithmisch in der Größe des Baums*.

Suche im Suchbaum

Search in search tree

```
def search(tree, item):
    if tree is None:
        return False
    elif tree.mark == item:
        return True
    elif tree.mark > item:
        return search(tree.left, item)
    else:
        return search(tree.right, item)

# smaller values left, bigger values in right subtree
nums = Node(10, Node(5, leaf(1), None),
                Node (15, leaf (12), leaf (20)))
print(search(nums, 12))
```

- Aufruf `insert(tree, item)` für das Einsortieren von `item` in `tree`
- Ist `tree` leer, so wird der Knoten `leaf(item)` zurückgegeben.
- Wenn die Markierung `tree.mark` größer als `item` ist, wird `item` in den linken Teilbaum eingesetzt und der Baum rekonstruiert (das erhält die Suchbaumeigenschaft!).
- Falls `tree.mark` kleiner als `item` ist, entsprechend.
- Falls `tree.mark == item` müssen wir nichts machen.

Creating a search tree

```
def insert(tree, item):
    if tree is None:
        return leaf(item)
    elif tree.mark > item:
        return Node (tree.mark,
                     insert (tree.left, item),
                     tree.right)
    elif tree.mark < item:
        return Node (tree.mark,
                     tree.left,
                     insert(tree.right, item))
    else:
        return tree
```

Suchbaumaufbau

UNI FREIBURG


```
def insertall (tree, lst):
    for key in lst
        tree = insert (tree, key)
    return tree

bst = insertall (None, [10, 15, 20, 12, 5, 1])
```

Creating a mutable search tree

```
def insertm(tree, item):
    if tree is None:
        return leaf(item)
    if tree.mark > item:
        tree.left = insertm(tree.left, item)
    elif tree.mark < item:
        tree.right = insertm(tree.right, item)
    return tree
```

Suchbaumaufbau

Mutable — veränderlich


```
def insertmall (tree, lst):
    for key in lst
        tree = insertm (tree, key)
    return tree

bst = insertmall (None, [10, 15, 20, 12, 5, 1])
```

4 Zusammenfassung

Zusammenfassung

- Der Baum ist eine Struktur, die in der Informatik allgegenwärtig ist.
- Binärbäume sind Bäume, bei denen jeder Knoten genau zwei Teilbäume besitzt.
- Operationen über (Binär-)Bäumen lassen sich einfach als rekursive Funktionen implementieren.
- Es gibt drei Hauptarten der Traversierung von Binärbäumen.
- Suchbäume sind Binärbäume, die die Suchbaumeigenschaft besitzen, d.h. in linken Teilbaum sind nur kleinere, im rechten nur größere Markierungen.
- Das Suchen und Einfügen kann durch einfache rekursive Funktionen realisiert werden. Sortierte Ausgabe ist auch sehr einfach!