Einführung in die Informatik 2

– Listen & Bäume –

Sven Kosub

AG Algorithmik/Theorie komplexer Systeme
Universität Konstanz

E 202 | Sven.Kosub@uni-konstanz.de | Sprechstunde: Freitag, 14:00-15:00 Uhr, o.n.V.

Sommersemester 2009

Referenzen (Wiederholung)

Verbundhierarchien:

- Verbünde dürfen rekursiv geschachtelt sein, d.h. Verbund hält Referenz auf Verbünde mit den gleichen Komponenten
- typische Verwendung: Verwaltung dynamischer Datensammlungen, Navigation

Spezialfälle der rekursiven Referenzierung:

- **Listen**
- **Bäume**

Listen

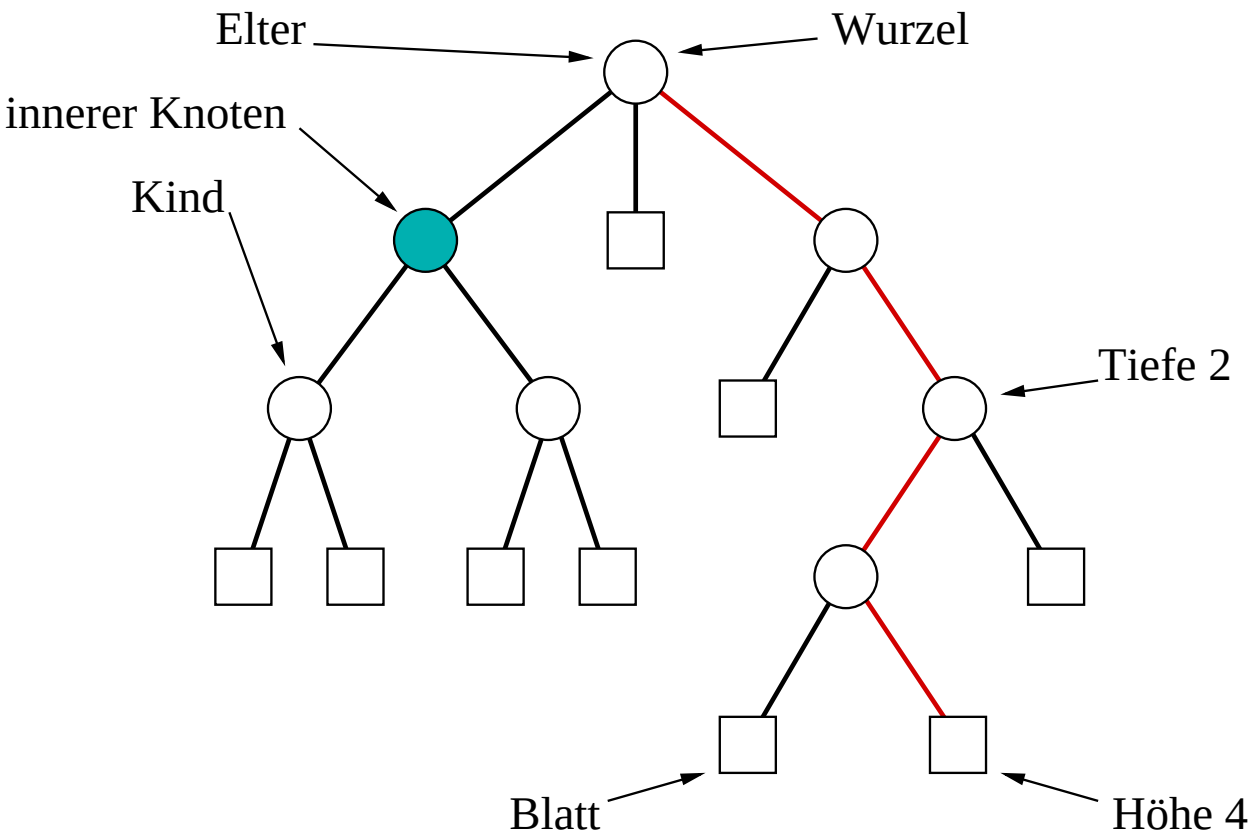
Liste:

- lineare Form der Datenorganisation
- erster Verbund der Liste heißt Kopf (`head`)
- jeder Verbund der Liste mit Referenz auf einen Nachfolgerverbund (`next`)
- jeder Verbund hat höchstens einen Verbund, der auf ihn referenziert
- letzter Verbund der Liste referenziert auf sich selbst oder auf speziellen Bezeichner für den leeren Verbund (`null`)

Bäume

- Bäume (engl. *trees*) Erweiterung von Listen:
  - in Liste hat jeder Knoten höchstens einen Nachfolger
  - in Bäumen hat jeder Knoten mehrere Nachfolger aber höchstens einen Vorgänger
- Nachfolgerknoten heißt Kind (engl. *child*)
- Vorgängerknoten heißt Elter (engl. *parent*)
- Knoten ohne Kinder heißt Blatt (engl. *leaf*)
- Knoten mit Kindern heißt innerer Knoten (engl. *internal node/vertex*)
- Baum enthält nur einen Knoten ohne Elter; Knoten heißt Wurzel (engl. *root*)
- Tiefe eines Knotens ist die Länge des Pfades bis zur Wurzel
- Höhe eines Baumes ist die maximale Tiefe eines seiner Knoten

Bäume



Bäume

- Grad eines Knotens in Baum ist die Anzahl seiner Kinder
- Baum ist Binärbaum (engl. *binary tree*) $\Longleftrightarrow_{\text{def}}$ alle Knoten haben Grad ≤ 2
- Binärbaum heißt voll $\Longleftrightarrow_{\text{def}}$ alle inneren Knoten haben Grad 2
- voller Binärbaum heißt vollständig $\Longleftrightarrow_{\text{def}}$ alle Blätter haben gleiche Tiefe

Fakten:

- Baum mit n Knoten besitzt $n-1$ Kanten
- Ein voller Binärbaum mit n Blättern besitzt $n-1$ innere Knoten
- Ein vollständiger Binärbaum der Höhe h besitzt 2^h Blätter

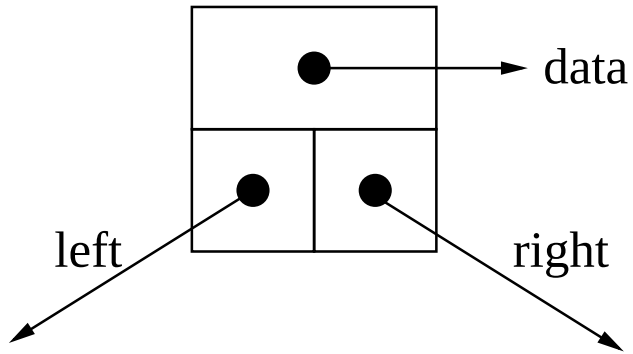
Implementierung von Bäumen: Binärbäume

```
package tree;

class Node {
  Object data;
  Node left;
  Node right;

  // Konstruktor
  Node(Object obj) {
    data=obj;
    left=right=null;
  }
}
⋮
```

- Paket `tree` kapselt Klasse `Node`; deshalb: kein Widerspruch zur Knotenklasse bei Listen



Implementierung von Bäumen: Binärbäume

```
⋮
public class Tree {
  protected Node root;

  // Konstruktoren
  Tree() { root=null; }
  Tree(Node r) { root=r; }

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

- Generische Binärbäume durch Referenz auf Knoten
- Ein leerer Baum t wird nicht durch `t=null` repräsentiert, sondern durch existierendes Objekt `t.root=null`

Rekursive Traversierung von Binärbäumen

```
package tree;
⋮
interface NodeActionInterface {
  public void action(Node n);
}
⋮
```

- Schnittstellenklasse für generische Durchläufe (Traversierungen)

Aktionsobjekt der Klasse `NodeActionInterface` als Parameter

```
public void preorder/inorder/postorder(NodeActionInterface p) {
  if (this.isEmpty()) return;
  Tree leftTree=new Tree(root.left);
  Tree rightTree=new Tree(root.right);
```

Präorder

```
p.action(root);
leftTree.preorder(p);
rightTree.preorder(p);
```

Inorder

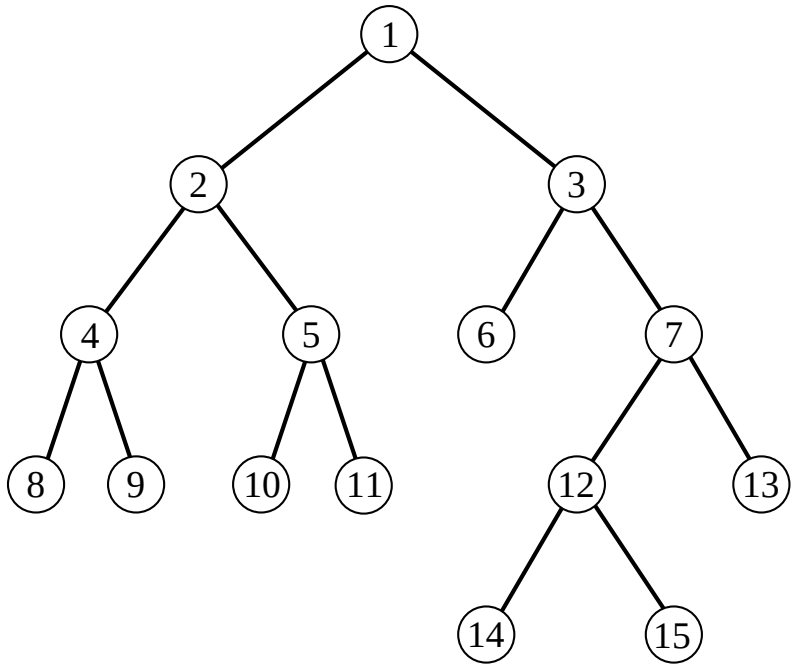
```
leftTree.inorder(p);
p.action(root);
rightTree.inorder(p);
```

Postorder

```
leftTree.postorder(p);
rightTree.postorder(p);
p.action(root);
```

Rekursive Traversierung von Binärbäumen: Beispiel

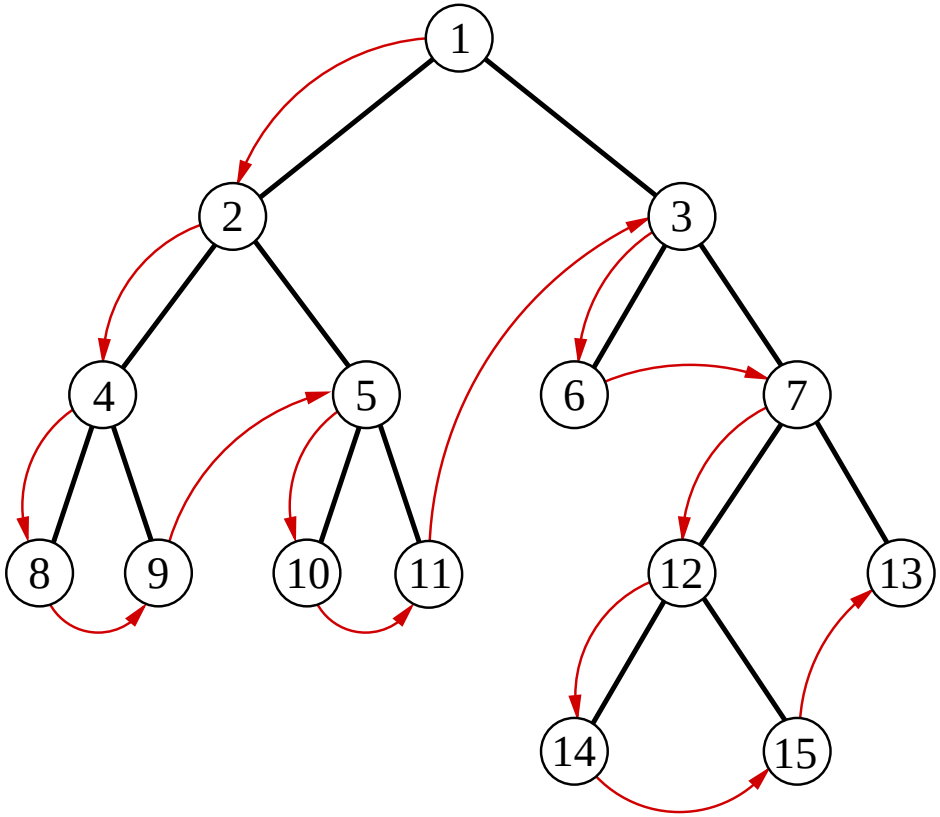
- betrachten Binärbäume mit Integer-Zahlen als Datenobjekten
- Aktionsobjekt mit Methode `action(Node n)` gibt die im Knoten abgespeicherte Zahl aus



In welcher Reihenfolge werden die gespeicherten Zahlen bei Präorder-, Inorder- und Postorder-Traversierung ausgegeben?

Rekursive Traversierung von Binärbäumen: Beispiel

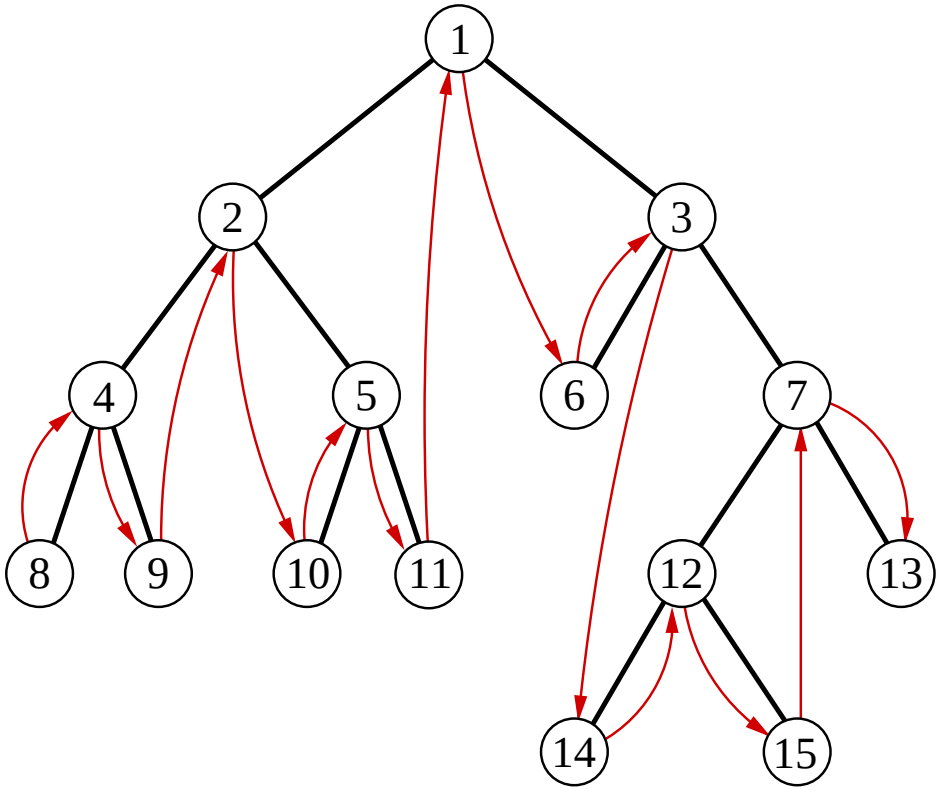
Präorder-Traversierung



Ausgabe: $(1, 2, 4, 8, 9, 5, 10, 11, 3, 6, 7, 12, 14, 15, 13)$

Rekursive Traversierung von Binärbäumen: Beispiel

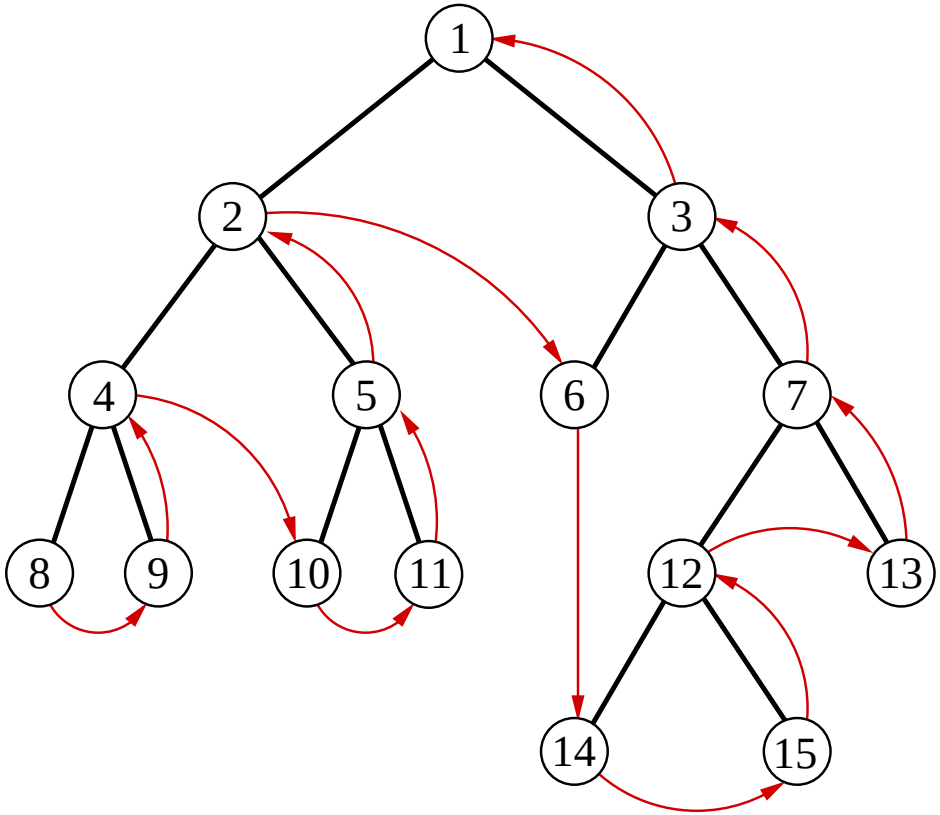
Inorder-Traversierung



Ausgabe: $(8, 4, 9, 2, 10, 5, 11, 1, 6, 3, 14, 12, 15, 7, 13)$

Rekursive Traversierung von Binärbäumen: Beispiel

Postorder-Traversierung



Ausgabe: $(8, 9, 4, 10, 11, 5, 2, 6, 14, 15, 12, 13, 7, 3, 1)$

Rekursive Traversierung von Binärbäumen: Beurteilung

Laufzeit:

- jede Baumkante wird zweimal durchlaufen: einmal beim Absteigen im Rekursionsbaum, einmal beim Wiederaufstieg
- damit: $2n - 2$ Kantentraversierungen, d.h., $O(n)$ Schritte, für Baum mit n Knoten

(zusätzlicher) Speicherplatz:

- betrachten Baumerzeugungen beim Rekursionsaufruf:

```
Tree leftTree=new Tree(root.left);
Tree leftTree=new Tree(root.right);
```

- für jeden Knoten werden 2 Bäume erzeugt und wieder zerstört
- damit: im schlechtesten Fall $2n$ zusätzliche Referenzen auf Objekte (Bäume) und $2n$ zusätzliche Variablen (Wurzeln)

Nicht-rekursive Traversierung von Binärbäumen

Reduzierung des Rekursionsaufwandes bei der Traversierung:

- Stacks: Simuliere Rekursionsstack innerhalb der Klasse
  - Stack ist Datenstruktur mit Operationen `push` und `pop`
  - `push` legt ein Objekt als oberstes Element auf den Stack
  - `pop` nimmt das oberste Objekt vom Stack
  - Realisierung des LIFO-Prinzips („last-in first-out“)
- Jackets: Benutze eine Mantelprozedur
  - benutze zwei Traversierungsmethoden: eine `public` und eine `private`
  - private Methode greift intern explizit auf die Knoten zu

Traversierung von Binärbäumen: Stacks

```
package tree;
public class Tree {
  protected Node root;
  ⋮
  public void preorderNonRecursive(NodeActionInterface p) {
    java.util.Stack stack=new java.util.Stack();
    stack.push(root);
    while (!stack.isEmpty()) {
      Object tmp=stack.pop();
      if (tmp != null && tmp instanceof Node) {
        Node tmpn = (Node) tmp;
        p.action(tmpn);
        stack.push(tmpn.right);
        stack.push(tmpn.left);
      }
    }
  }
  ⋮
}
```

Traversierung von Binärbäumen: Jackets

```
package tree;
public class Tree {
  protected Node root;
  :
  public void preorderNonRecursive(NodeActionInterface p) {
    traversePreorder(root,p);
  }
  private void traversePreorder(Node n,NodeActionInterface p) {
    if (n == null) return;
    p.action();
    traversePreorder(n.left,p);
    traversePreorder(n.right,p);
  }
  :
}
```

Problem: stets $n + 1$ triviale Aufrufe von `traversePreOrder`

Traversierung von Binärbäumen: Jackets

Ausweg: späterer Emptiness-Test

```
package tree;
public class Tree {
  protected Node root;
  ⋮
  public void preorder(NodeActionInterface p) {
    if (!isEmpty()) traversePreorderNonEmpty(root,p);
  }
  private void traversePreorderNonEmpty(Node n,NodeActionInterface p) {
    p.action();
    if (left != null) traversePreorder(n.left,p);
    if (right != null) traversePreorder(n.right,p);
  }
  ⋮
}
```