Bäume

Listen und Bäume, Graphen und Bäume, elementare Eigenschaften von Binärbäumen, Implementierung, Generische Baumdurchläufe

Foliensatz von Prof. D. Saupe, Uni Konstanz, 2003.
Überarbeitet und ergänzt von A. Weber,
sowie von W. Küchlin zu Informatik II, Tübingen 2004

Grundkonzepte von Bäumen

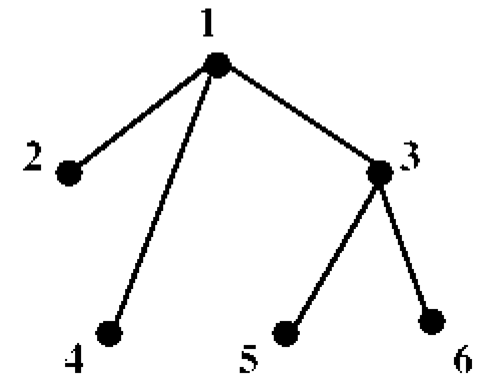
- Bäume (trees) können als eine Verallgemeinerung von Listen angesehen werden
  - Bei einer Liste hat jeder Knoten einen Nachfolger
  - Bei einem Baum hat jeder Knoten potentiell mehrere „Nachfolger“
  - Kinderknoten (child, children) genannt
  - Wie bei einer Liste hat jeder Knoten (bis auf den Kopfknoten) genau einen Vorgänger
  - Vorgänger eines Kindknotens wird Elternknoten (parent) genannt

Grundkonzepte von Bäumen

- Ein Knoten ohne Elternknoten wird Wurzel (root) genannt
  - Bei endlichen Bäumen gibt es stets Wurzeln
  - Als Baum wird i.A. Datenstruktur genannt, die genau eine Wurzel besitzt
  - Sonst wird von „Wald“ (forest) gesprochen
- Knoten ohne Kinderknoten heißen Blätter oder Terminalknoten (leaf, leaves)
- Knoten mit Kinderknoten heißen auch innere Knoten (inner nodes)
- Jedem Knoten ist eine **Ebene** (level) im Baum zugeordnet
  - Die Ebene eines Knotens ist die Länge des Pfades von diesem Knoten bis zur Wurzel
- Die **Höhe** (height) eines Baums ist die maximale Ebene, auf der sich Knoten befinden

Grundkonzepte von Bäumen

- Bäume kann man dadurch visualisieren, dass Verbindungen zwischen Eltern und ihren Kindern eingezeichnet werden
  - In der Informatik zeichnet man Bäume üblicherweise von der Wurzel abwärts
- Bei dem rechts dargestellten Baum ist der Knoten 3 ein Elternknoten der Knoten 5 und 6
  - Die Knoten 2, 3 und 4 sind Kinder von 1
  - Bei diesem Baum liegt der Knoten mit Etikett 1 auf Ebene 0, die Knoten 2, 3 und 4 auf Ebene 1 und die Knoten 5 und 6 auf Ebene 2
  - Die Höhe des Baumes ist 2



Grundkonzepte von Bäumen

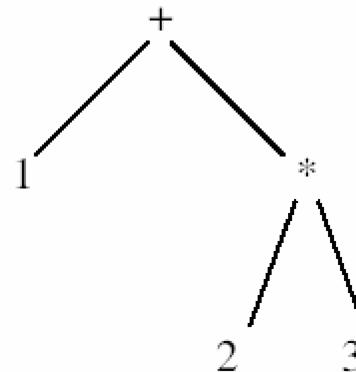
- Anwendungen sind vielfältig. Beispiele:
  - Organisationshierarchien in Unternehmen und Behörden
  - Aufrufstruktur von rekursiven Algorithmen wie etwa divide-and-conquer-Verfahren
  - Mögliche Züge in einem Zweipersonenspiel (z.B. Schach, Mühle)
  - Struktur eines mathematischen oder (programmiersprachlichen) Ausdrucks
  - Hierarchische Unterteilung von geometrischen Objekten oder vom Räumen
  - Struktur von Sequenzen von Entscheidungen in strategischen Untersuchungen
  - ...

Grundkonzepte von Bäumen

- Der **Verzweigungsgrad** (out degree) eines Knotens ist die Anzahl seiner Kinder
- Ein **Binärbaum** (binary tree) ist ein Baum, dessen Knoten höchstens den Verzweigungsgrad 2 haben
  - Der Baum des vorigen Beispiels ist kein Binärbaum, da die Wurzel Verzweigungsgrad 3 hat

Grundkonzepte von Bäumen

- Ausdrücke können durch Strukturbäume (parse trees) gut repräsentiert werden
  - Rekursive Definition spiegelt sich in Eltern-Kinderknoten-Verhältnis wieder
  - Beispiel: Strukturbaum von 1+2*3
    - Oder von 1+(2*3)

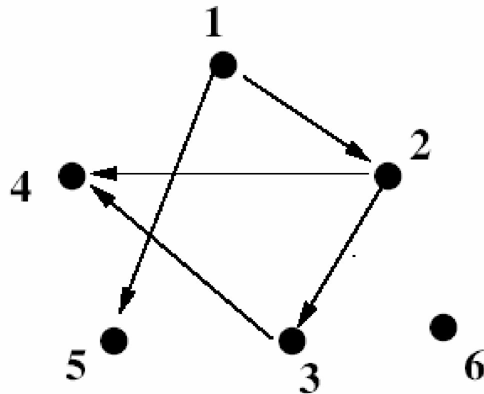


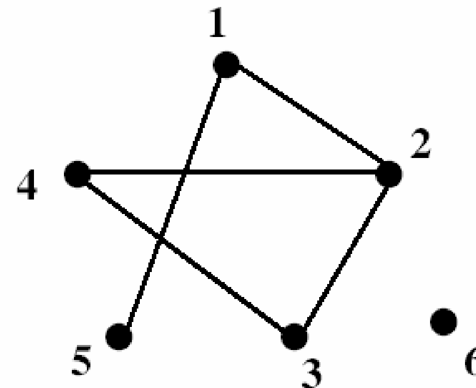
Grundkonzepte von Bäumen

- **Bemerkung:** Wir definieren hier Bäume als Verallgemeinerungen von Listen
  - Man kann Bäume aber auch als einen *Spezialfall* einer anderen für die Informatik sehr wichtigen Datenstruktur definieren, nämlich der eines Graphen
    - Ein (gerichteter) **Graph** (directed graph) ist ein Paar (V,E) bestehend aus einer nicht leeren Menge V von Ecken oder **Knoten** (vertices, nodes) und einer Menge E von **Kanten** (edges)
      - Dabei ist die Menge der Kanten E eine binäre Relation auf V, also $E \subseteq V \times V$
        - » Ein Knoten kann ein **Etikett** (label) und weitere Informationen enthalten
      - Die Kanten können als Verbindungen zwischen den als kleine Kreise dargestellten Ecken visualisiert werden
      - Die Richtung der Kanten wird i. a. durch einen Pfeil angegeben

Graphen und Bäume

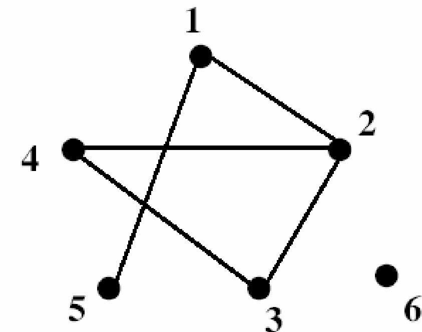
- Ungerichtete Graphen:
  - alle Kanten paarweise $((v_1,v_2) \in E \Leftrightarrow (v_2,v_1) \in E)$
  - es reicht nur eines der Paare anzugeben
- Beispiele gerichteter und ungerichteter Graphen
  - V = {1,2,3,4,5,6}, E = {(1,2), (2,3), (3,4), (2,4), (1,5)}
  - a) als gerichteter Graph, b) als ungerichteter Graph





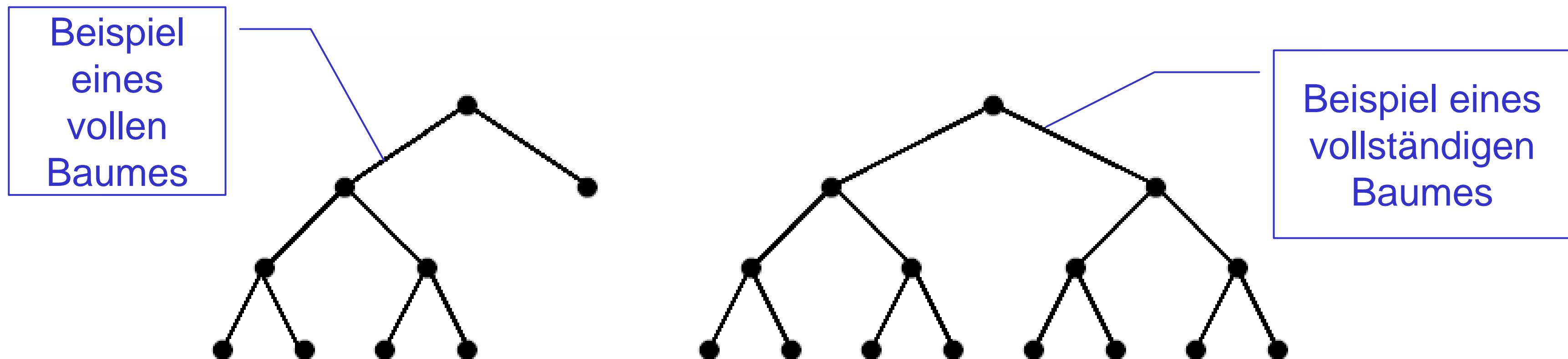
Graphen und Bäume

- Ein Pfad (path) ist eine Folge von verschiedenen Knoten, die durch Kanten verbunden sind
- Fakt: Ein (ungerichteter) Graph ist genau dann ein Baum, wenn es zwischen je zwei beliebigen Knoten genau einen Pfad gibt
  - Ist der Graph von letzter Folie ein Baum?
  - Beweis der Aussage (Übung)



Eigenschaften von Bäumen

- Ein Binärbaum heißt **voll**, falls alle inneren Knoten den Verzweigungsgrad 2 haben
- Ein voller Binärbaum heißt **vollständig**, falls alle Blätter den gleichen Level haben



Eigenschaften von Bäumen

Lemma: *Ein Baum mit n Knoten hat n-1 Kanten*

- **Beweis:** Jede Kante verbindet einen Knoten mit dem Elternknoten. Außer dem Wurzelknoten ist jeder Knoten durch eine Kante mit seinem Elternknoten verbunden. Also muss die Anzahl Kanten genau um eins kleiner sein als die Anzahl Knoten.

Eigenschaften von Bäumen

Lemma: *Ein vollständiger Binärbaum der Höhe n hat 2^n Blätter*

Beweis: Induktion über Höhe des Baumes

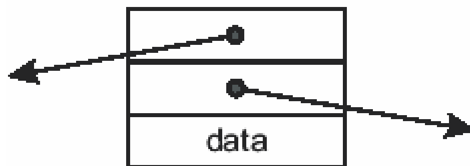
Lemma: *Ein vollständiger Binärbaum der Höhe n hat $2^{n+1} -1$ Knoten*

In einem vollständigen Baum ist die Anzahl der Blätter gleich der Anzahl der inneren Knoten minus 1.

Eigenschaften von Bäumen

- **Bemerkung**: Diese (und ähnliche) Zusammenhänge zwischen Höhe eines Baumes und Anzahl der Knoten bzw. Blätter ist ein Grund für die Bedeutung von Bäumen z.B. in Datenbanksystemen
  - Bei „Suchbäumen" wächst die Höhe nur „logarithmisch" mit der Anzahl der Blätter
    - Etwa Anfragen an Suchmaschinen, die Milliarden von Seiten im World Wide Web abdecken, liefern durch die Verwendung von Bäumen (zur Strukturierung der Daten) trotzdem sehr schnell eine Antwort
    - Auch wenn das WWW wächst, wird die Zeit für Suchanfragen nur „logarithmisch" wachsen

- Generische Binärbäume: Knotenklasse




```
package tree;

/**
 * Class for Nodes of a generic binary tree.
 */
class Node {
  Node left;
  Node right;
  Object data;
  // constructor
  Node(Object a) {
     data=a;
     left = right = null;}
}
```

Da wir im Paket tree sind, können wir den gleichen Namen wie bei Listenknoten für unsere Klasse nehmen

Paket tree kapselt die Klasse Node

Implementierung von Bäumen

- Generische Binärbäume durch Referenz auf Wurzelknoten
- Node kann für weitere Datenstrukturen verwendet werden
- Ein leerer Tree t wird nicht durch t=null repräsentiert, sondern durch ein existierendes Objekt t mit t.root=null

```
 * Class for a generic binary tree.
 */
public class Tree {
  protected Node root;

  /**
   * Constructor for empty tree.
   */
  Tree() { root=null;}
  /**
   * Constructs a tree with new root node rn.
   */
   Tree(Node rn) {
    root = rn;
   }
  /**
   * Checks whether this tree is empty.
   */
   public boolean isEmpty() {
     return (root==null);
   }
  // weitere Methoden siehe unten
}
```

Baumdurchläufe

- Schnittstellen-Klasse für generische Baumdurchläufe
- In folgenden Beispielen wählen wir ein Aktionsobjekt der Klasse NodeActionInterface als Parameter, das eine Funktion action(Node) kapselt

```
package tree;
 /**
  * Interface consisting of functions
  * which operate on the nodes of a tree.
  */
  interface NodeActionInterface {
   /**
    * Abstract function whose realizations
    * operate on tree nodes.
    */
    public void action(Node n);
  // evtl. weitere Funktionen
}
```

Wenn `p` ein formaler Parameter vom Typ
`NodeActionInterface`
ist und `n` ein Parameter oder eine Variable vom Typ `Node`, dann können wir also in den Methoden zum Baumdurchlauf Anweisungen der Form
`p.action(n);`
verwenden.

Baumdurchläufe

- Beispielklasse, die NodeActionInterface implementiert

```
package tree;

public class NodePrintAction
  implements NodeActionInterface {

  /**
   * Sample implementation of action.
   */
  public void action(Node n) {
    System.out.print(n.data.toString());
  }
}
```

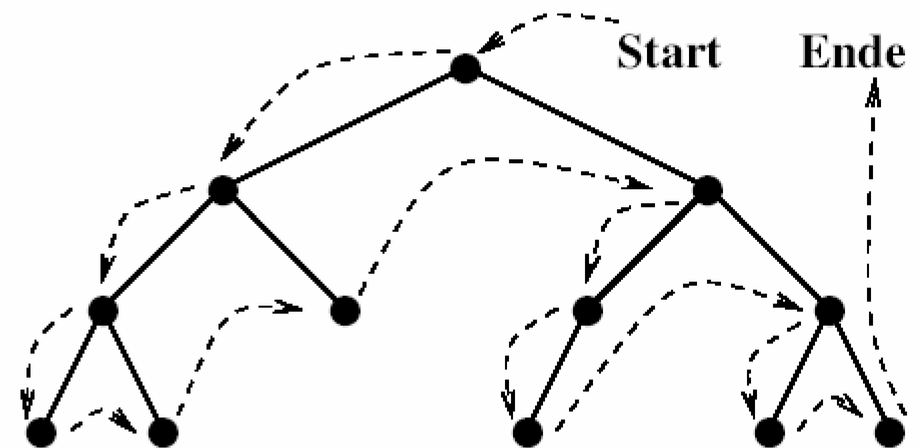
Baumdurchläufe

- Weiteres Beispiel einer Klasse, die `NodeActionInterface` implementiert
  - Zweck: die inneren Knoten sollen gezählt werden
  - Wieder ein fold-Operator

```
package tree;
public class CountInnerNodes
      implements NodeActionInterface {
  public int counter; // state field
                      // to count nodes
  /**
   * action: count inner nodes
   */
  public void action(Node n) {
    if (n.left!=null || n.right!=null) counter++;
  }
}
```

Baumdurchläufe: Präorder

- Das abstrakte Verfahren zum Durchlauf in Präorder lautet folgendermaßen:
    1. Betrachte die Wurzel des Baums (und führe eine Operation auf ihr aus)
    2. Durchlaufe den linken Teilbaum
    3. Durchlaufe den rechten Teilbaum



Baumdurchläufe: Präorder

```
package tree;

public class Tree {
  protected Node root;
  // ...

  /**
   * Applies the function p.action to any node
   * of the tree in preorder sequence.
   */
  public void preorder(NodeActionInterface p){
    if (root == null)           // empty tree?
      return;

    // init
    Tree leftTree =
      new Tree(root.left);    // left subtree
    Tree rightTree =
      new Tree(root.right);   // right subtree

    // work
    p.action(root);      // visit node: apply function
    leftTree.preorder(p);      // left recursion
    rightTree.preorder(p);     // right recursion
  }

}
```

Ein Baumdurchlauf für den oben gegebenen Strukturbaum (zu 1+2*3) in Präorder mit der Operation *Drucke Symbol* erzeugt folgende Ausgabe:

+ 1 * 2 3

Die Wiedergabe eines Strukturbaums für einen Ausdruck mit Präorder entspricht der **polnischen Notation** **(**Polish notation) für Ausdrücke.

Baumdurchläufe: Präorder (nicht-rekursive Version)

- Wollen wir keinen rekursiven Aufruf für die Tiefensuche verwenden, brauchen wir einen Stack, auf dem wir uns die zu behandelnden Knoten für später merken
  - Der rekursive Aufruf verwendet den Laufzeitstapel
- Wir könnten hierfür analog zur Klasse TStack eine spezielle Klasse NodeStack erstellen
- Wir wollen an dieser Stelle aber einmal die generische Klasse Stack aus dem Paket java.util heranziehen
  - Den generischen Stack von Elementen vom Typ Object benutzen wir also für Elemente vom Typ Node
  - Da die Methode pop ein Objekt vom Typ Object zurückliefert, überprüfen wir mittels des instanceof-Operators, ob es sich tatsächlich um einen Node handelt und spezialisieren es dann auch syntaktisch zu einem Node
  - Da wir vor dem Aufruf der Methode pop testen, ob der Stack leer ist, kann die Ausnahme EmptyStackException nicht vorkommen
    - Die üblichen Java-Compiler können eine solche Analyse jedoch nicht durchführen und verlangen auch in solchen Fällen eine explizite Ausnahmebehandlung - es sei denn, es handelt sich wie hier um eine ungeprüfte Ausnahme (unchecked exception), die ohnehin nicht unbedingt behandelt werden muss
      - Dazu gehört auch die Klasse EmptyStackException, da sie eine Unterklasse von RuntimeException ist

Baumdurchläufe: Präorder (nicht-rekursive Version)

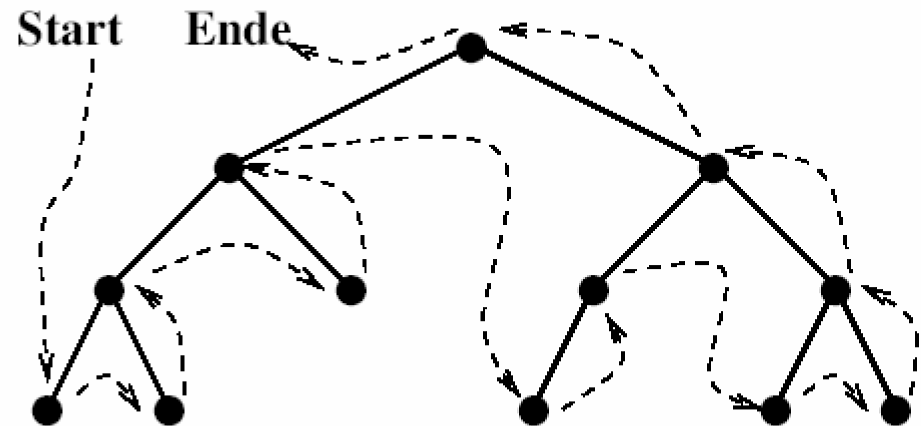
```
package tree;

public class Tree {
  protected Node root;
  // ...
```

```
/**
 * Applies the function p.action to any node
 * of the tree in preorder sequence.
 * Non-recursive version of the method.
 */
public void preorderNonRecursive(NodeActionInterface p){
  java.util.Stack stack =
    new java.util.Stack();
  stack.push(root);           // Initialize
  while (!stack.isEmpty()) {
    Object tmp =  stack.pop();
    if (tmp != null && tmp instanceof Node)
      // empty tree?
      // instanceof error must not happen
      {
        Node tmpn = (Node) tmp;
        p.action(tmpn);       // visit node:
                              // apply function
        stack.push(tmpn.right); // right subtree
        stack.push(tmpn.left);  // left subtree
      }
    }
  }
}
```

Baumdurchläufe: Postorder

- Das abstrakte Verfahren zum Durchlauf in Postorder lautet folgendermaßen:
  1. Durchlaufe den linken Teilbaum
  2. Durchlaufe den rechten Teilbaum
  3. Betrachte die Wurzel des Baums (und führe eine Operation auf ihr aus)



Baumdurchläufe: Postorder

```
package tree;
public class Tree {
  protected Node root;
```

Ein Baumdurchlauf für den oben gegebenen Strukturbaum in Postorder mit der Operation *Drucke Symbol* erzeugt folgende Ausgabe:

1 2 3 * +

Die Wiedergabe eines Strukturbaums für einen Ausdruck mit Postorder entspricht der **umgekehrten polnischen Notation** (reverse Polish notation) für Ausdrücke.

```
   // ...

  /**
   * Applies the function p.action
   * to any node of the tree in postorder
   * sequence.
   */
  public void postorder(NodeActionInterface p){
    if (root == null)          // empty tree?
      return;

    // init
    Tree leftTree =
      new Tree(root.left);    // left subtree
    Tree rightTree =
      new Tree(root.right);   // right subtree

    // work
    leftTree.postorder(p);    // left recursion
    rightTree.postorder(p);   // right recursion
    p.action(root);  // visit node: apply function
  }
}
```

Baumdurchläufe: Inorder

- Das abstrakte Verfahren zum Durchlauf in Inorder lautet folgendermaßen:
  1. Durchlaufe den linken Teilbaum
  2. Betrachte die Wurzel des Baums (und führe eine Operation auf ihr aus)
  3. Durchlaufe den rechten Teilbaum

Start

Ende

Baumdurchläufe: Inorder

```
package tree;
public class Tree {
  protected Node root;
  // ...

 /**
   * Applies the function p.action
   * to any node of the tree in inorder
   * sequence.
   */
  public void inorder(NodeActionInterface p){
    if (root == null)        // empty tree?
      return;

    // init
    Tree leftTree =
      new Tree(root.left);  // left subtree
    Tree rightTree =
      new Tree(root.right); // right subtree

    // work
    leftTree.inorder(p);  // left recursion
    p.action(root);     // visit node:
                        // apply function
    rightTree.inorder(p) // right recursion

  }
}
```

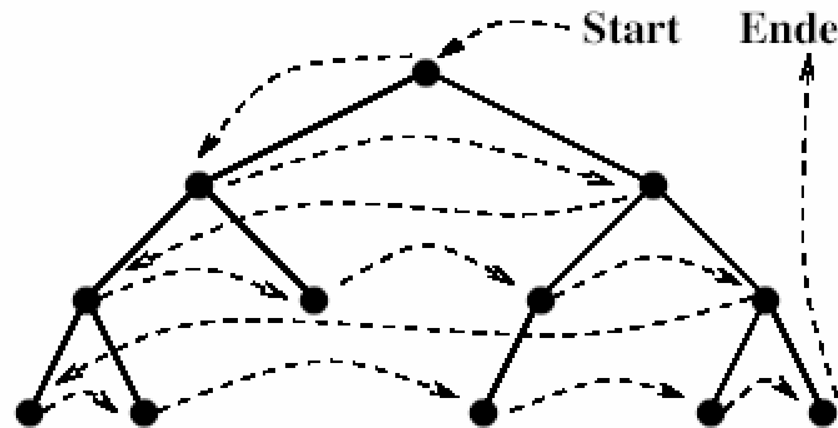
Ein Baumdurchlauf für den oben gegebenen Strukturbaum in Inorder mit der Operation *Drucke Symbol* erzeugt folgende Ausgabe:

1 + 2 * 3

Die Wiedergabe eines Strukturbaums für einen Ausdruck mit Inorder entspricht der normalen Operator-Schreibweise (wenn zusätzlich die Subausdrücke noch geklammert werden)

Baumdurchläufe: Levelorder

- Beim Durchlaufen eines Baumes Schicht für Schicht (levelorder) geht man wie folgt vor
  - Starte bei der Wurzel (Ebene 0)
  - Bis die Höhe des Baumes erreicht ist, setze die Ebene um eins höher und gehe von links nach rechts durch alle Knoten dieser Ebene



Baumdurchläufe: Levelorder

- Bei diesem Verfahren geht man nicht zuerst in die Tiefe, sondern die Strategie heißt **Breite zuerst** (breadth first)
  - Dies kommt besonders bei **Suchbäumen** (search tree) zum Einsatz, deren Knoten Spielpositionen und deren Kanten Spielzüge darstellen (z. B. für Schach, Dame etc.)
    - Wir suchen dann im Baum eine Gewinnstellung, die in möglichst wenigen Zügen erreichbar ist
    - Solche Suchbäume sind sehr tief und werden daher nur begleitend zur Suche Schicht für Schicht generiert, bis der gesuchte Knoten gefunden wurde
    - Der Einfachheit halber werden wir aber in der Folge von einem bereits generierten Baum ausgehen

Baumdurchläufe: Levelorder

- Um der Breite nach durch einen Baum zu wandern, müssen wir uns alle Knoten einer Ebene merken
- Diese Knoten speichern wir für späteren Zugriff in einer Warteschlange (queue) ab
- Die Warteschlange kann sehr lang werden
  - Im schlimmsten Fall erhält sie eine Länge von $n/2$ bei n Knoten (die Anzahl der Blätter eines vollständigen Binärbaums)
  - Bei den rekursiven Methoden („depth first“) preorder, inorder oder postorder wird der (implizit oder explizit) verwendete Stack maximal so groß wie die Tiefe des bei der Rekursion betrachteten Baumes
  - Dieser ist - wie wir unten genauer zeigen werden - um eins tiefer als der zu durchlaufende Baum selbst. Damit ist die maximale Tiefe $1+\log_2 n$.

Baumdurchläufe: Levelorder

```
package tree;
public class Tree {
  protected Node root;
  // ...

 /**
   * Applies the function p.action
   * to any node of the tree in levelorder
   * sequence.
   */
  public void levelorder(NodeActionInterface p){
    Queue queue = new Queue();
    queue.append(root);
    while( !queue.isEmpty()) {
      Object tmp = queue.get();
      if( tmp != null && tmp instanceof Node)
        // empty tree?
        // instanceof error must not happen
        {
          Node tmpn = (Node) tmp;

          p.action(tmpn.data);      // apply function
          queue.append(tmpn.left);     // left subtree
          queue.append(tmpn.right);    // right subtree
        }
    }
  }

}
```

Speicherbedarf in rekursiven Baumdurchläufen

- Betrachte Rekursion in Klassenmethoden von `Tree`
- Aufwand dazu:
  - Vor Rekursion: Erzeugung von zwei neuen Bäumen erforderlich:

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

- Für jeden (nicht leeren) Knoten werden zwei Bäume generiert und wieder zerstört
- Auf Laufzeitstapel Speicher für $2n$ Referenzvariablen und $2n$ Knotenvariablen (muss nicht gleichzeitig existieren) wobei n = Anzahl der Knoten

Speicheroptimierung durch Mantelprozedur (jacket)

```
package tree;
public class Tree {
    protected Node root;
    // ...
    public void preorder(NodeActionInterface f){
        traversePreorder(root, f);
    }

    private void traversePreorder(Node n, NodeActionInterface f) {
       // trivial case
       if (n == null) return;

       // Action
       f.action(n);

       // Recursion
       traversePreorder(n.left, f);
       traversePreorder(n.right, f);
    }
}
```

Zeitbedarf in rekursivem Baumdurchläufe

- Wir haben je Knoten und für jeden nicht existenten Nachfolger eines Knoten einen Prozeduraufruf.
- In vollständigem Binärbaum der Tiefe k mit $n=2^k-1$ Knoten ist die Anzahl dieser Aufrufe mit leeren Teilbaum $2 \cdot 2^{k-1} = n+1$, also sogar größer als die Anzahl der nichttrivialen Aufrufe!
- Lösung: Test vor jedem rekursiven Aufruf

  ```
  if (n.left!=null) traversePreorder(n.left, f);
  ```

- Dann ist aber Test

  ```
  if (n == null) return;
  ```

  unnötig (außer zu Beginn)
- Lösung: zweite Mantelprozedur

  ```
  traversePreorderNonEmpty(root, f);
  ```

  mit Aufruf nur wenn Baum nicht leer.

Effizienter Präorder Baumdurchlauf

```
package tree;
public class Tree {
  protected Node root;
  // ...
  public void preorder(NodeActionInterface f){
    if (root == null) return; // empty tree
    else traversePreorderNonEmpty(root, f);
  }
  private void traversePreorderNonEmpty (Node n, NodeActionInterface f) {
    // Non-trivial case!
    // Action
    f.action(n);
    // Recursion
    if (n.left != null) traversePreorderNonEmpty(n.left, f);
    if (n.right != null) traversePreorderNonEmpty(n.right, f);
  }
}
```