



Bäume 1

Thomas Röfer

Preisverleihung

Begriffsdefinitionen

Eigenschaften

Implementierung von Bäumen

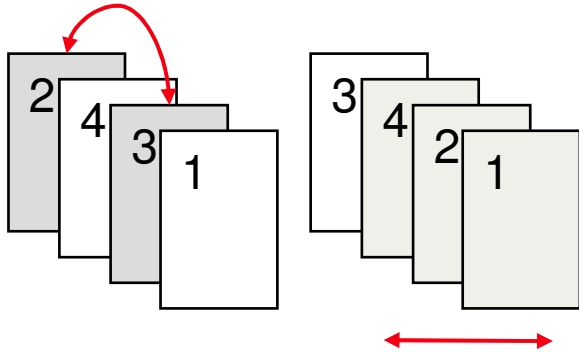
Durchlaufen von Bäumen

Breitensuche/Tiefensuche

Huffman-Kodierung

Rückblick „Sortieren“

Permutieren nach Dijkstra



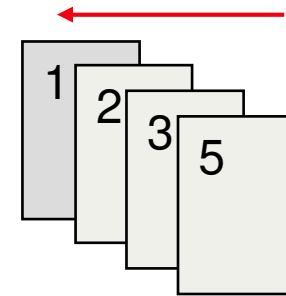
Sortieren durch ...

$$S = \text{perm}(T), s_0 \leq s_1 \leq \dots \leq s_{n-1}$$

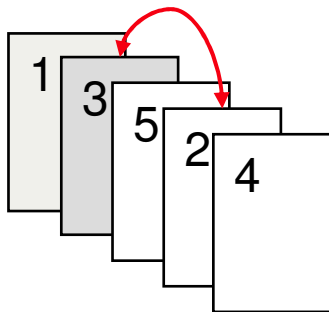
oder

$$S = \text{perm}(T), s_0 \geq s_1 \geq \dots \geq s_{n-1}$$

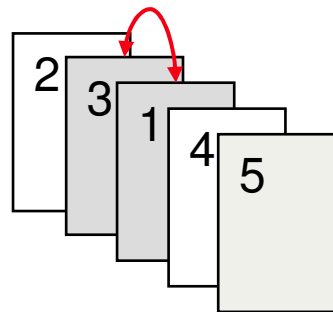
... Einfügen



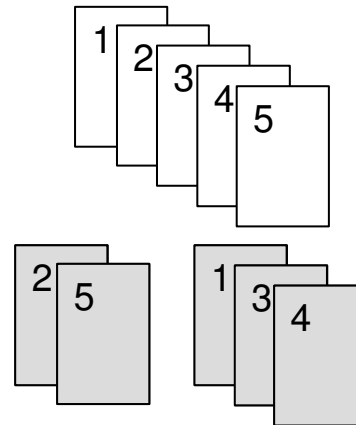
... Auswählen



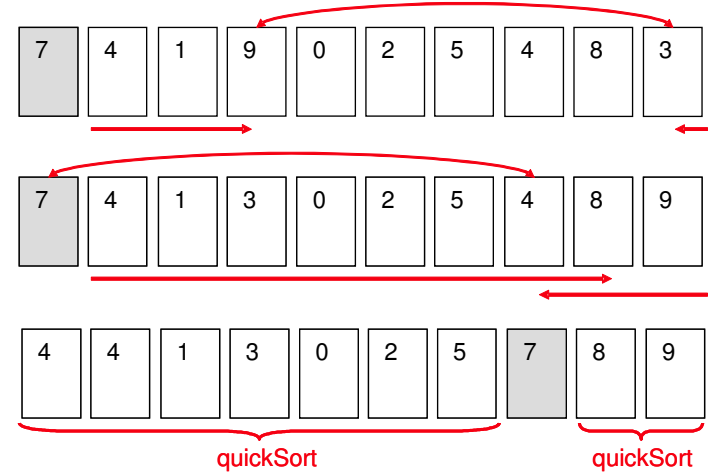
... Vertauschen
(BubbleSort)



... Mischen



Quicksort





Vier gewinnt – Preisverleihung

| | JM | K | F | H | A | | |
|----|----|---|---|---|---|----|---|
| JM | / | 3 | 6 | 6 | 3 | 18 | 2 |
| K | 3 | / | 6 | 6 | 6 | 21 | 1 |
| F | 0 | 0 | / | 1 | 3 | 4 | 5 |
| H | 0 | 0 | 4 | / | 6 | 10 | 3 |
| A | 3 | 0 | 3 | 0 | / | 6 | 4 |

JM K = 3

| | T | F | N | N ^e | H | | |
|----------------|---|---|---|----------------|---|----|---|
| T | / | 6 | 6 | 6 | 6 | 24 | 1 |
| F | 0 | / | 3 | 0 | 0 | 3 | 5 |
| N | 0 | 3 | / | 2 | 3 | 8 | 4 |
| N ^e | 0 | 6 | 2 | / | 6 | 14 | 2 |
| H | 0 | 6 | 3 | 0 | / | 9 | 3 |

| | JM | K | T | N ^e | | |
|----------------|----|---|---|----------------|----|---|
| JM | / | 3 | 6 | 4 | 13 | 1 |
| K | 3 | / | 6 | 1 | 10 | 2 |
| T | 0 | 0 | / | 6 | 6 | 3 |
| N ^e | 1 | 4 | 0 | / | 5 | 4 |



Schnittstelle für „Vier gewinnt“

```
package kimatch.connect4;

public abstract class
    Connect4Player
{
    public void begin() {}
    public void end() {}
    public abstract String
        getPlayerName();
    public abstract void
        onMove(MoveEventArgs e);
    public abstract void
        onOpponentMove(
            OpponentMoveEventArgs e);
}
```

```
public final class MoveEventArgs
{
    public MoveEventArgs(
        boolean[] availableColumns) ...
    public boolean isAvailable(
        int column) ...
    public void select(int column) ...
    public int getSelectedColumn() ...
}

public final class
    OpponentMoveEventArgs
{
    public OpponentMoveEventArgs(
        int column) ...
    public int getSelectedColumn() ...
}
```

Einleitung

▶ Motivation

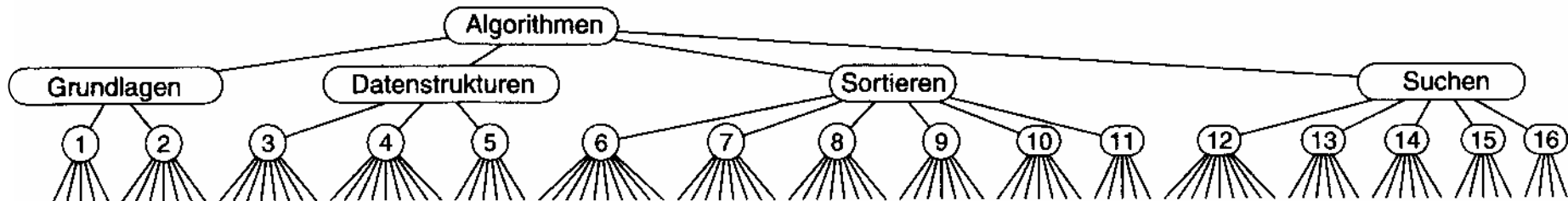
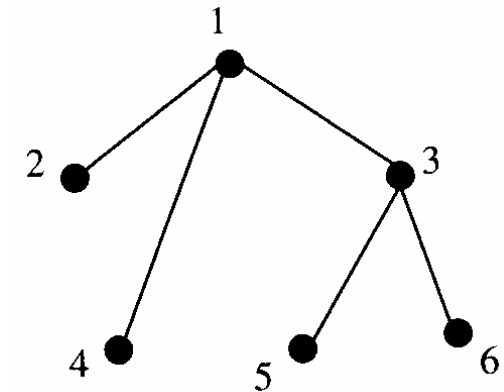
- ▶ Reihungen sind statisch bzw. nur teuer zu vergrößern und zu verkleinern
- ▶ Listen sind dynamisch, aber man kann in ihnen nur linear suchen

▶ Ansatz

- ▶ Man kann Bäume als verallgemeinerte Listenstruktur ansehen
- ▶ Jeder Knoten hat nicht einen, sondern viele Nachfolger

▶ Beispiel

- ▶ Z.B. Teile und Kapitel von „Algorithmen in Java“



Begriffe

▶ Knoten

- ▶ Kann einen Namen haben (*Etikett, Label*) und/oder andere Informationen tragen
- ▶ Kann mehrere *untergeordnete Knoten* haben (*Kindknoten, Nachfolger*)
- ▶ Kann maximal einen *übergeordneten Knoten* haben (*Elternknoten, Vorgänger*)
- ▶ Hat ein Knoten keinen übergeordneten Knoten, ist er die *Wurzel* des Baums
- ▶ Der *Verzweigungsgrad* eines Knotens ist die Anzahl seiner Kindknoten
- ▶ Hat ein Knoten keine Nachfolger, ist er ein *Blatt* (*Terminalknoten*)

▶ Kante

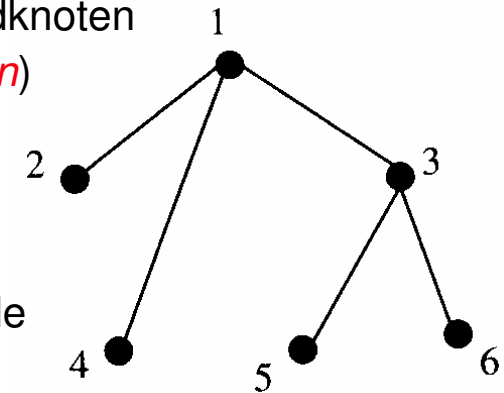
- ▶ Verbindet jeweils zwei Knoten (Eltern- mit Kindknoten)

▶ Pfad

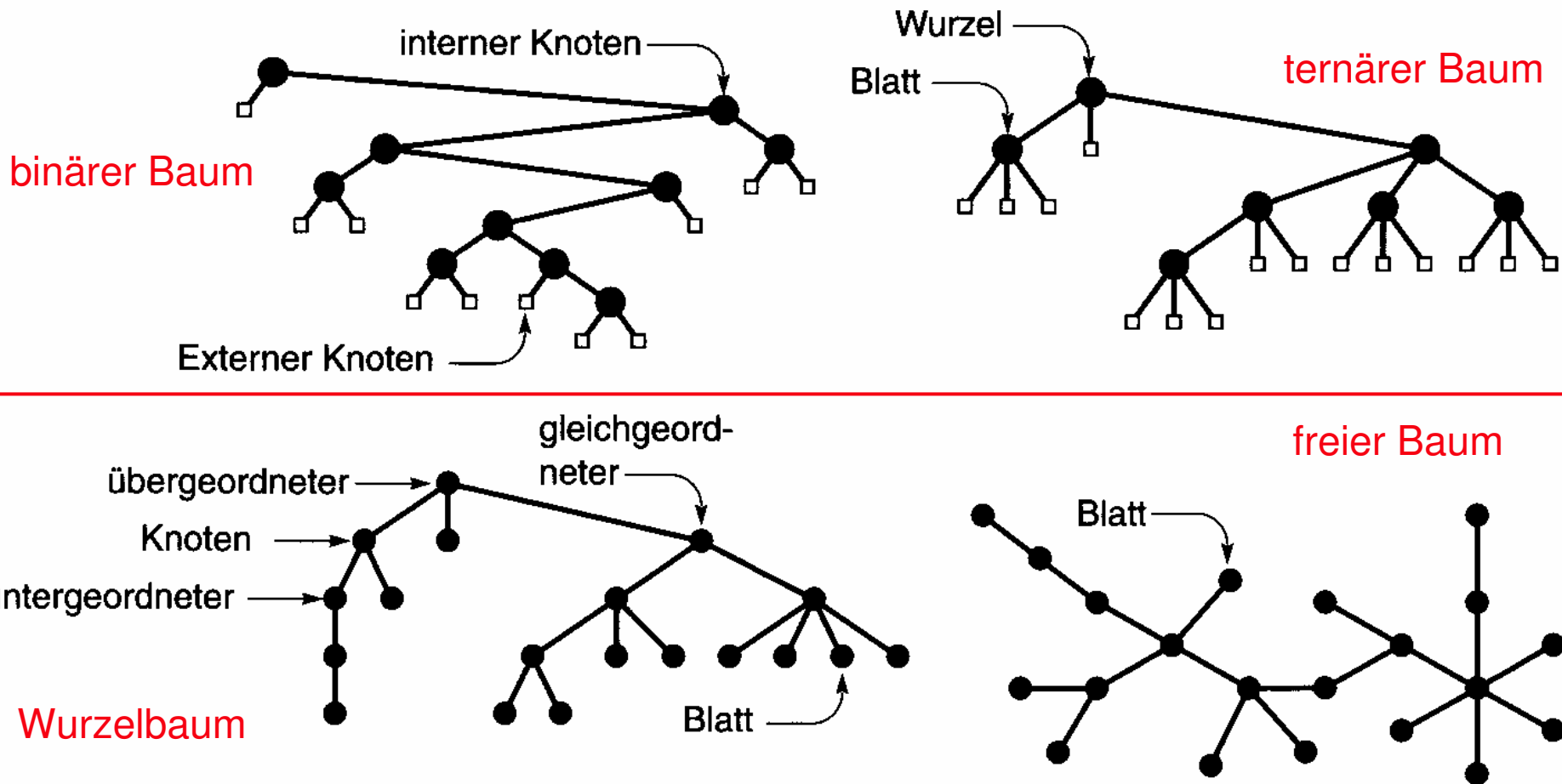
- ▶ Folge von unterschiedlichen Knoten, wobei aufeinander folgende Knoten durch Kanten verbunden sind

▶ Baum

- ▶ Nichtleere Sammlung von Knoten und Kanten
- ▶ Jeweils zwei Knoten sind genau durch einen Pfad verbunden
- ▶ Mehrere nicht miteinander verbundene Bäume bezeichnet man als *Wald*

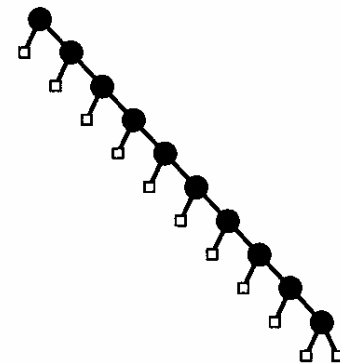
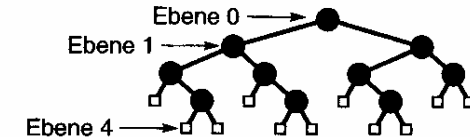
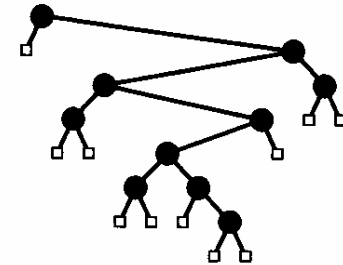


Begriffe



Begriffe

- ▶ **Ordnung eines Baums**
 - ▶ Der maximale Verzweigungsgrad seiner Knoten
- ▶ **Binärbaum**
 - ▶ Die Ordnung des Baums ist 2
- ▶ **Geordneter Baum**
 - ▶ Zwischen den Kindern der Knoten ist eine *Ordnung* definiert
 - ▶ z.B. *erster, zweiter, dritter ... Kindknoten*
- ▶ **Tiefe eines Knotens**
 - ▶ Der Abstand zwischen der Wurzel und dem Knoten (Anzahl der Kanten)
 - ▶ Die Wurzel hat die Tiefe 0
 - ▶ Alle Knoten mit derselben Tiefe gehören zur selben *Ebene* (zum selben *Niveau*)
- ▶ **Höhe eines Baums**
 - ▶ Die maximale Tiefe eines Knotens im Baum



Definitionen und Eigenschaften

▶ Voller Baum

- ▶ Ein Baum heißt voll, wenn der *Verzweigungsgrad* aller inneren Knoten der *Ordnung des Baums* entspricht

▶ Vollständiger Baum

- ▶ Ein Baum heißt vollständig, wenn die *Tiefe* aller *Blätter* der *Höhe des Baums* entspricht

▶ Ein Baum mit n Knoten hat $n-1$ Kanten

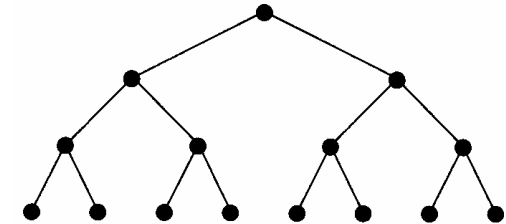
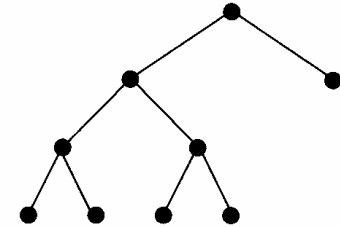
- ▶ Zu jedem *Knoten* führt eine *Kante*, nur zur *Wurzel* nicht

▶ Ein vollständiger Binärbaum der Höhe n hat 2^n Blätter

- ▶ Ein Binärbaum, der nur aus der Wurzel besteht, hat ein Blatt
- ▶ Ein vollständiger Binärbaum der Höhe $n+1$ hat doppelt so viele Blätter wie einer der Höhe n

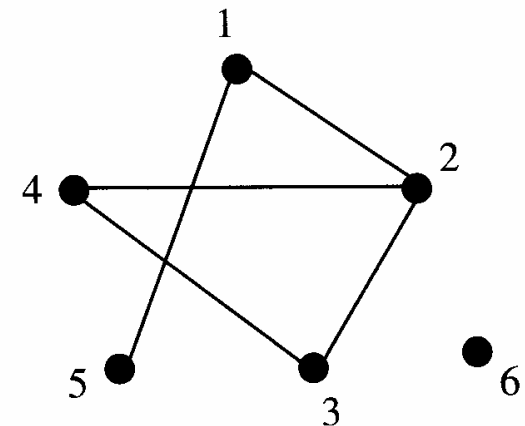
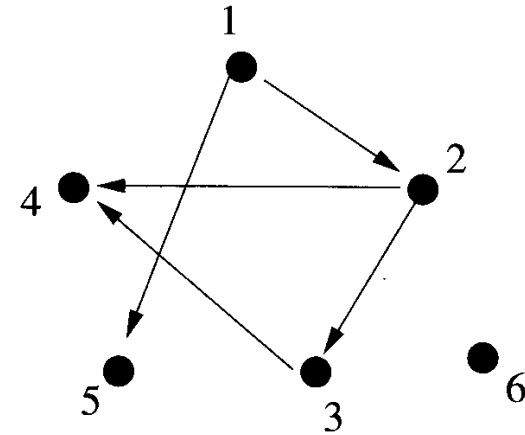
▶ Ein vollständiger Binärbaum der Höhe n hat $2^{n+1} - 1$ Knoten

- ▶ Ein Binärbaum, der nur aus der Wurzel besteht, hat einen Knoten
- ▶ Mit jeder Ebene kommt ein Knoten mehr hinzu, als schon im Baum vorhanden sind: $knoten(n+1) = 2 \times knoten(n) + 1$

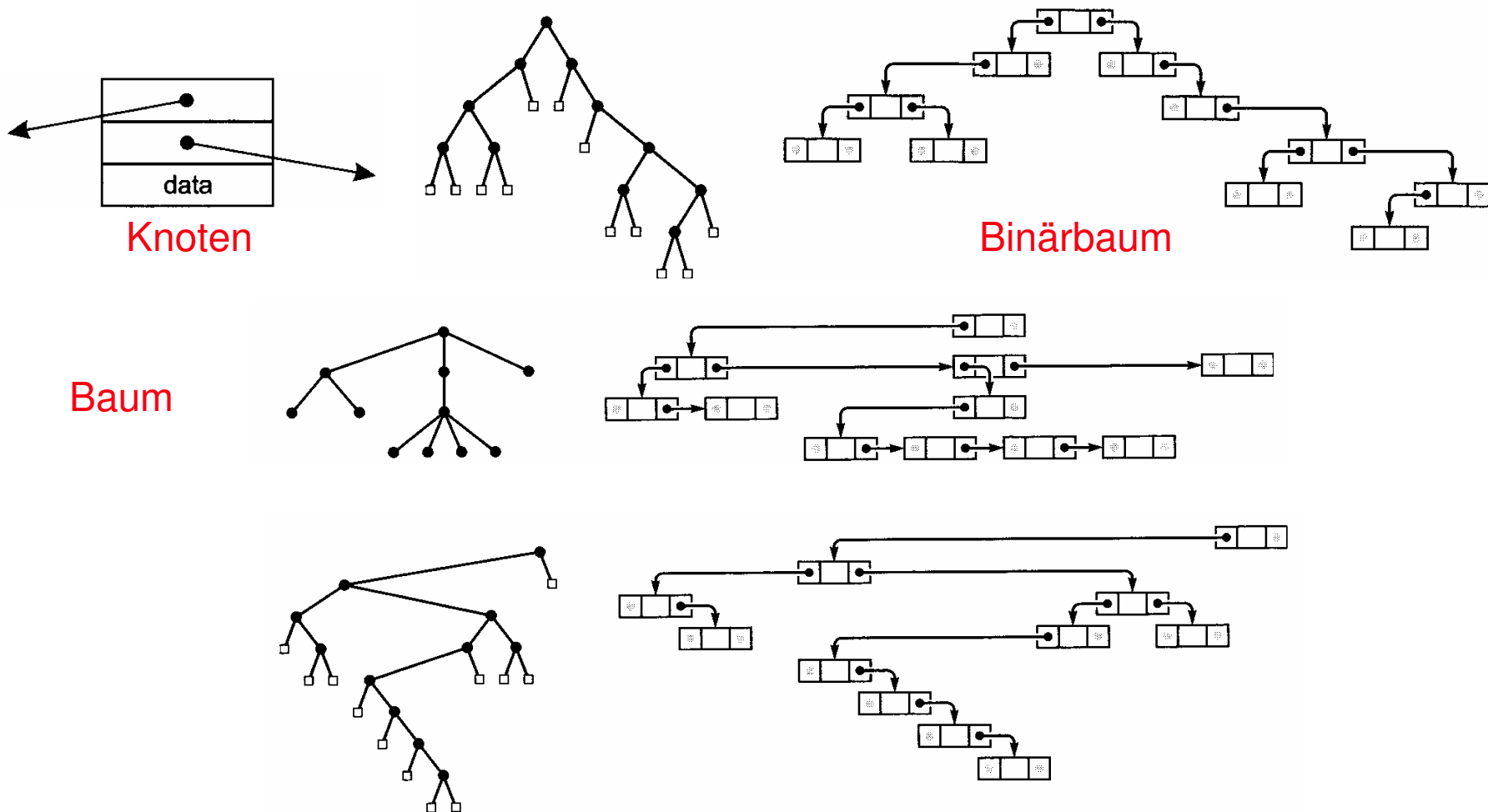


Graphen und Bäume

- ▶ **Andere Sichtweise**
 - ▶ Bäume können nicht nur als *Verallgemeinerung von Listen* gesehen werden, sondern auch als *Spezialfall von Graphen*
- ▶ **Baum mit gerichteten Kanten**
 - ▶ Ein Baum mit gerichteten Kanten verhält sich in seiner Implementierung analog zu einer einfach verketteten Liste
 - ▶ Alle Kanten zeigen von Elternknoten zu Kindknoten
- ▶ **Baum mit ungerichteten Kanten**
 - ▶ Ein Baum mit ungerichteten Kanten verhält sich in seiner Implementierung analog zu einer doppelt verketteten Liste
 - ▶ Kindknoten haben zusätzlich einen Verweis auf ihren Elternknoten



Implementierung von Bäumen





Implementierung eines Binärbaums

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

    Node(Object o, Node l, Node r)
    {
        data = o;
        left = l;
        right = r;
    }

    boolean isLeaf()
    {
        return left == null &&
            right == null;
    }
}
```

```
boolean isInnerNode()
{
    return !isLeaf();
}

class Tree
{
    private Node root;

    Tree(Node n)
    {
        root = n;
    }

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



Durchlaufen eines Baums

- ▶ **Ansatz**
 - ▶ Die Knoten eines Baums können in verschiedenen Reihenfolgen durchlaufen werden
 - ▶ Beim Durchlaufen können Operationen auf den in den Knoten enthaltenen Daten durchgeführt werden
- ▶ **Mögliche Reihenfolgen**
 - ▶ Präorder-Sequenz
 - ▶ Postorder-Sequenz
 - ▶ Inorder-Sequenz
 - ▶ Level-Order-Sequenz

```
interface NodeAction
{
    void action(Node n);
}

class NodePrintAction
    implements NodeAction
{
    public void action(Node n)
    {
        System.out.print (
            n.data.toString() + " ");
    }
}
```

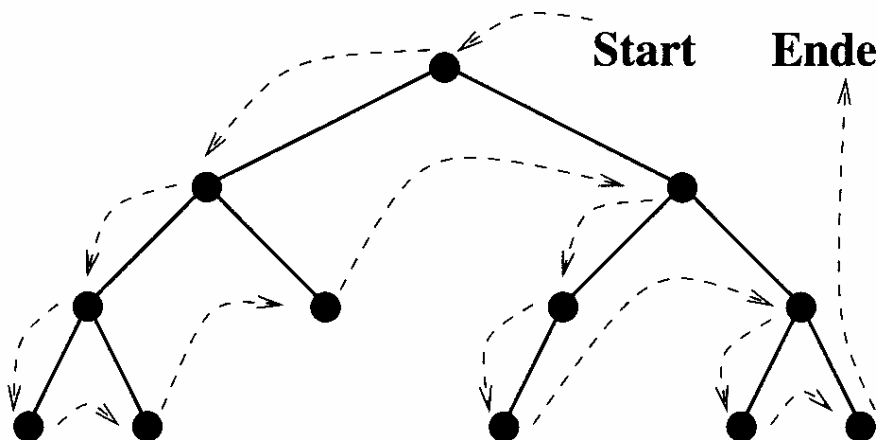
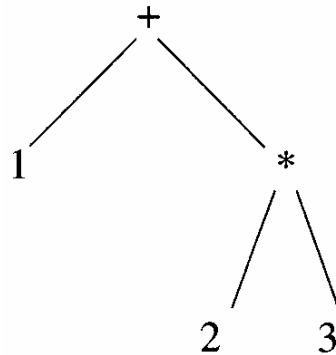
Präorder-Sequenz

▶ Ansatz

- ▶ In jedem Knoten wird zuerst der Knoten verarbeitet, dann die Kindknoten

▶ Beispiel

- ▶ + 1 * 2 3



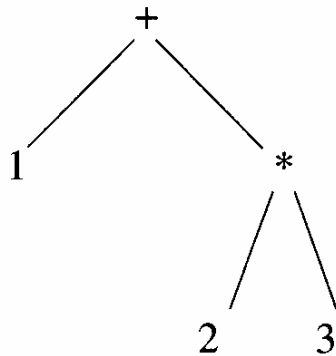
```

void preorder(NodeAction a)
{
    if(!isEmpty())
    {
        a.action(root);
        new Tree(root.left)
            .preorder(a);
        new Tree(root.right)
            .preorder(a);
    }
}
  
```

Postorder-Sequenz

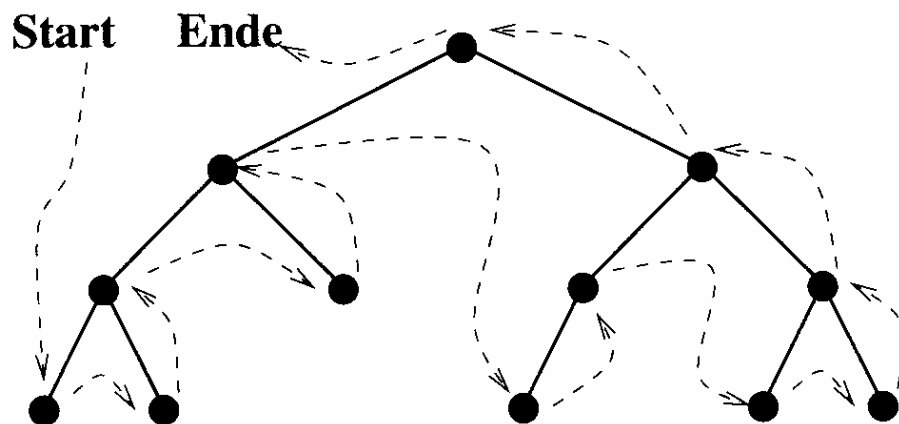
▶ Ansatz

- ▶ In jedem Knoten werden zuerst die Kindknoten verarbeitet, dann der Knoten selbst



▶ Beispiel

- ▶ 1 2 3 * +



```
void postorder(NodeAction a)
{
    if(!isEmpty())
    {
        new Tree(root.left)
            .postorder(a);
        new Tree(root.right)
            .postorder(a);
        a.action(root);
    }
}
```

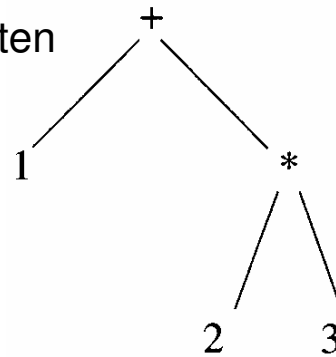
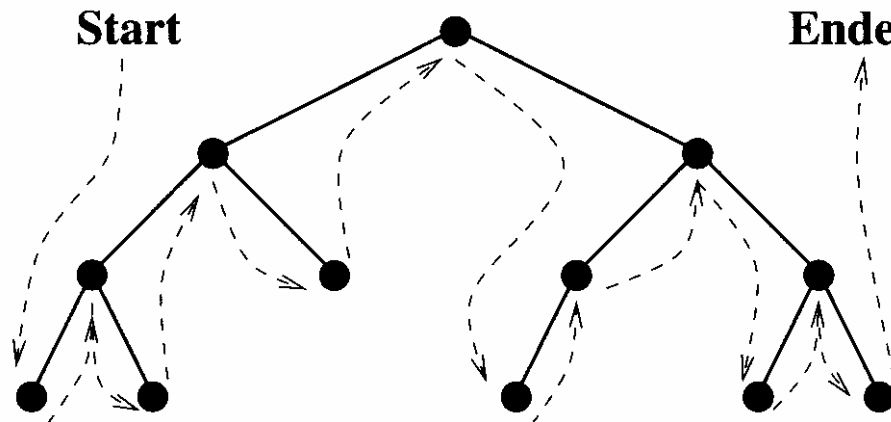
Inorder-Sequenz

Ansatz

- ▶ In jedem Knoten wird zuerst der linke Kindknoten verarbeitet, dann der Knoten selbst, dann der rechte Kindknoten
- ▶ Funktioniert nur bei Binärbäumen

Beispiel

- ▶ $1 + 2 * 3$



```
void inorder(NodeAction a)
{
    if(!isEmpty())
    {
        new Tree(root.left)
            .inorder(a);
        a.action(root);
        new Tree(root.right)
            .postorder(a);
    }
}
```

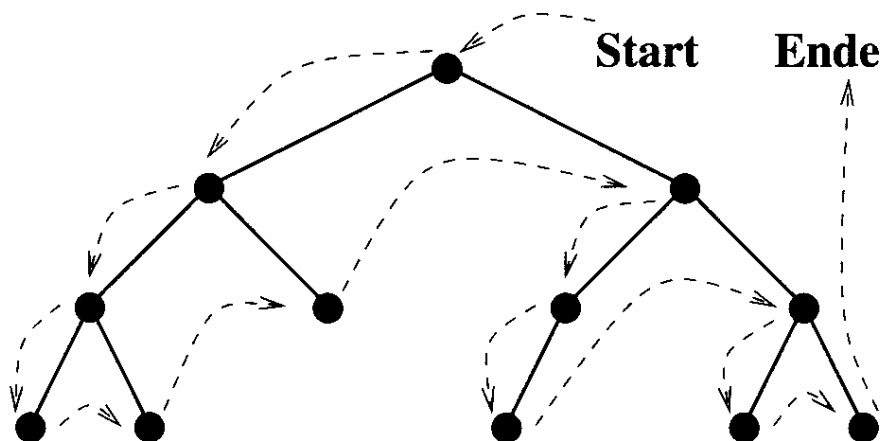
Präorder-Sequenz ohne Rekursion

▶ Ansatz

- ▶ Statt Rekursion wird ein *Stapel* verwendet, um die offenen Knoten zu speichern
- ▶ Der Stapel ist vom Typ *last-in-first-out*

▶ Hinweis

- ▶ Kann ebenfalls für Postorder- und Inorder-Sequenz genutzt werden



```
void preorder(NodeAction a)
{
    Stack stack = new Stack();
    stack.push(root);
    while(!stack.isEmpty())
    {
        Node node = (Node) stack.pop();
        if(node != null)
        {
            a.action(node);
            stack.push(node.right);
            stack.push(node.left);
        }
    }
}
```

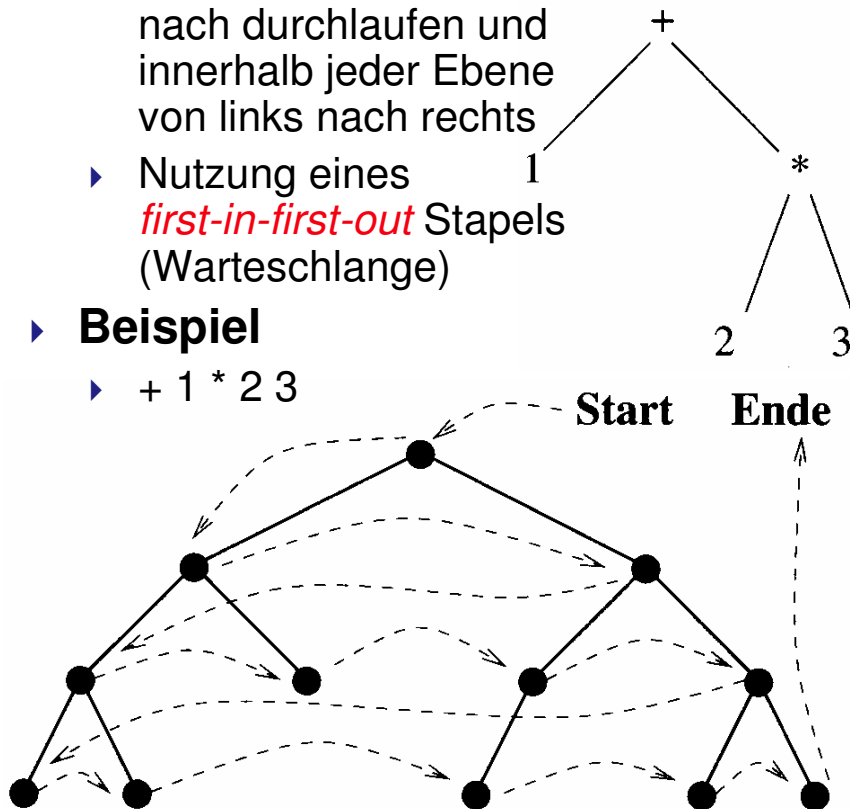
Level-Order-Sequenz

Ansatz

- Die Knoten werden ihrer Ebene nach durchlaufen und innerhalb jeder Ebene von links nach rechts
- Nutzung eines *first-in-first-out* Stapels (Warteschlange)

Beispiel

- + 1 * 2 3



```

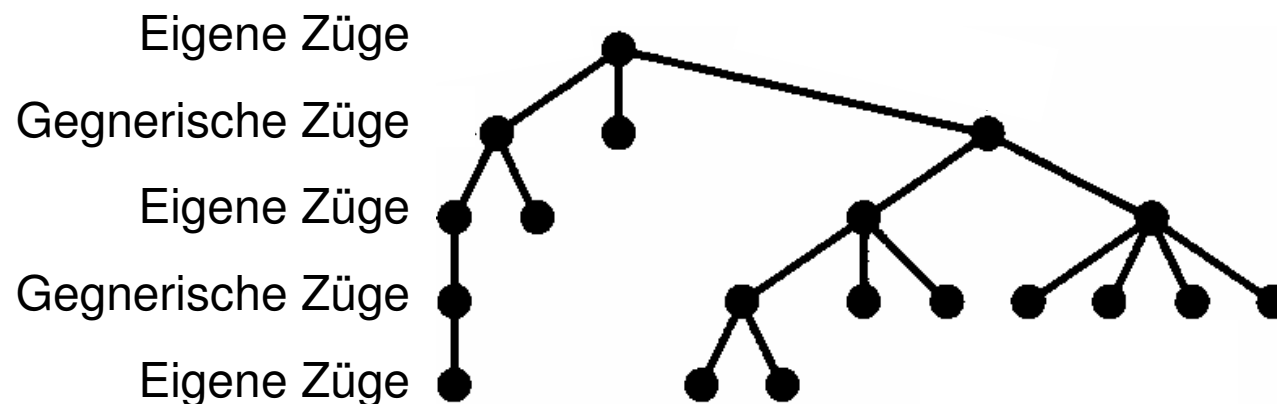
void levelOrder(NodeAction a)
{
    Queue stack = new Queue();
    stack.push(root);
    while(!stack.isEmpty())
    {
        Node node = (Node) stack.pop();
        if(node != null)
        {
            a.action(node);
            stack.push(node.left);
            stack.push(node.right);
        }
    }
}

```

Spielprobleme

▶ Spielprobleme

- ▶ Bei Spielproblemen (z.B. Vier gewinnt) werden mögliche Zugfolgen durchprobiert und das Ergebnis bewertet
- ▶ Dabei wird unter allen eigenen Zügen der mit der besten Bewertung gesucht (Bewertung *maximieren*)
- ▶ Allerdings muss man davon ausgehen, dass der Gegner den für einen selbst ungünstigsten Zug auswählen wird (Bewertung *minimieren*)
- ▶ Daher spricht man dabei von einem Mini-Max-Problem



Spielprobleme – Tiefen-/Breitensuche

▶ **Tiefensuche**

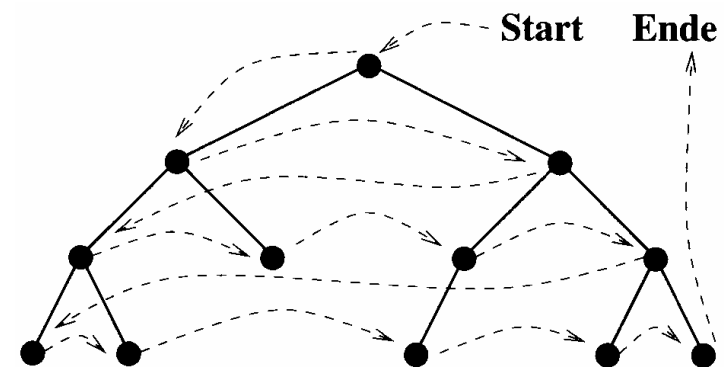
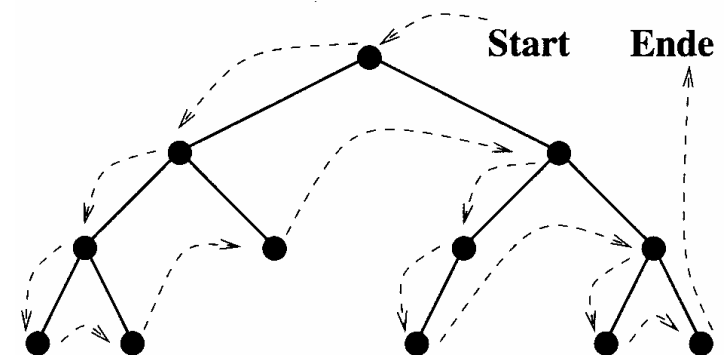
- ▶ Durchsuche den Baum der möglichen Zugfolgen bis zu einer bestimmten Tiefe
- ▶ Verwende den besten gefundenen Zug

▶ **Breitensuche**

- ▶ Durchsuche den Baum der möglichen Zugfolgen in Level-Order-Reihenfolge so lange, bis die Zeit um ist
- ▶ Verwende den besten gefundenen Zug

▶ **Inkrementelle Tiefensuche**

- ▶ Durchsuche den Baum der möglichen Zugfolgen bis zu einer bestimmten Tiefe
- ▶ Ist noch Zeit übrig, erhöhe die Tiefe um 1 und beginne von vorne
- ▶ Verwende den besten gefundenen Zug





Huffman Komprimierung

▶ Kodierung mit fester Bitanzahl

▶ A: 000 C: 010 E: 100 G: 110
 B: 001 D: 011 F: 101 H: 111

▶ BACADAEAFABBAAAGAH wird mit $18 \times 3 = 54$ Bits kodiert:
001 000 010 000 011 000 100 000 101 000 001 001 000 000 000 110 000 111

▶ Ansatz

▶ Wenn Zeichen in einem Text unterschiedlich häufig vorkommen, wäre es günstig, die häufigeren mit weniger Bits zu kodieren und die seltenen mit mehr

▶ Kodierung mit variabler Bitanzahl

▶ A: 0 C: 1010 E: 1100 G: 1110
 B: 100 D: 1011 F: 1101 H: 1111

▶ BACADAEAFABBAAAGAH wird mit 42 Bits kodiert:
100 0 1010 0 1011 0 1100 0 1101 0 100 100 0 0 0 1110 0 1111

▶ Probleme

▶ Wenn Zeichen durch unterschiedlich viele Bits repräsentiert werden, muss man erkennen können, wo ein Zeichen endet und wo das nächste beginnt
▶ Zudem braucht man ein Lexikon, in dem steht, welche Bitfolge welchem Zeichen entspricht

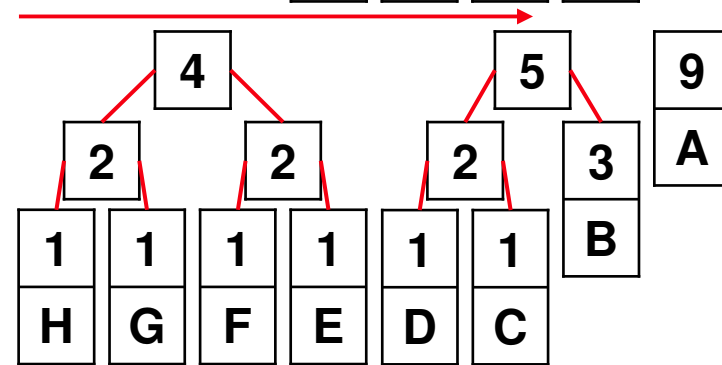
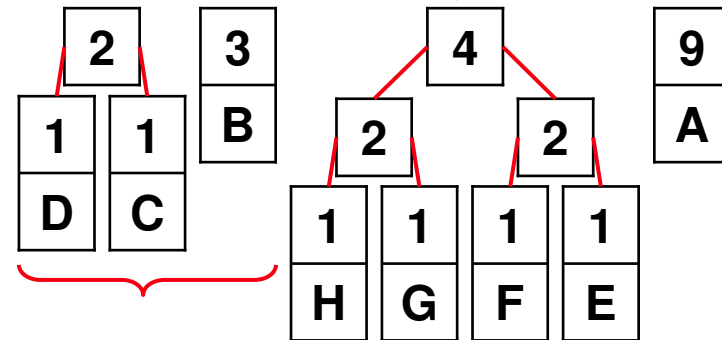
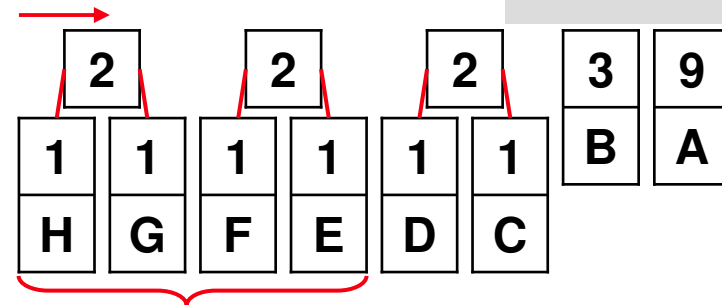
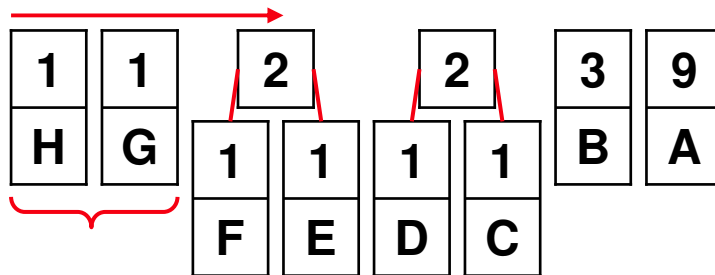
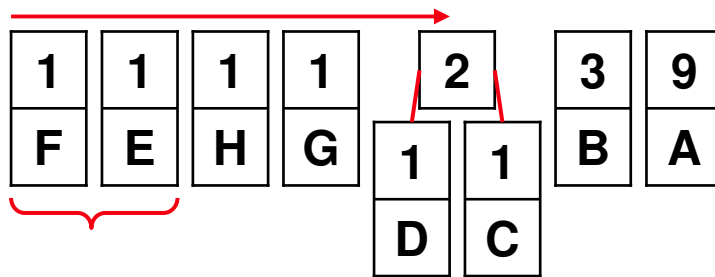
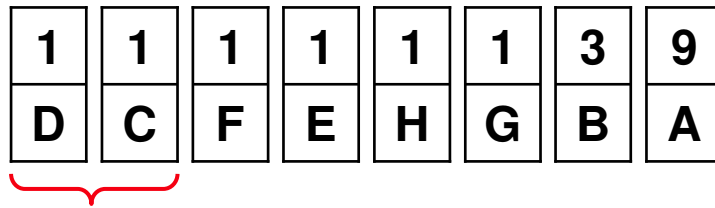


Aufbau eines Huffman-Baumes

- ▶ **Gegeben sei eine leere Liste von Binärbäumen**
- ▶ **Für jedes Zeichen**
 - ▶ Erzeuge einen Binärbaum, bei dem die Wurzel ein Blatt ist und das Zeichen und seine Häufigkeit enthält.
 - ▶ Füge den Baum aufsteigend nach der in der Wurzel vermerkten Häufigkeit sortiert in die Liste ein
- ▶ **Solange die Liste mehr als einen Baum enthält**
 - ▶ Entnimm die beiden ersten Elemente aus der Liste
 - ▶ Erzeuge einen neuen Binärbaum, bei dem der linke Kindknoten der ursprünglich erste Baum in der Liste ist und der rechte Kindknoten der ursprünglich zweite Baum
 - ▶ Die in der Wurzel vermerkte Häufigkeit ist die Summe der Häufigkeiten der beiden Teilbäume
 - ▶ Füge den Baum aufsteigend nach der in der Wurzel vermerkten Häufigkeit sortiert in die Liste ein
- ▶ **Der letzte in der Liste verbliebene Binärbaum ist der Huffman-Baum**



Beispiel: BACADAEAFABBAAGAH



Beispiel: BACADAEAFABBAAGAH

► **Eigentliche Kodierung**

- Durch Pfad von der Wurzel zum Zeichen
- Wenn nach rechts verzweigt wird, schreibe eine 0
- Wenn nach links verzweigt wird, schreibe eine 1

► **Beispiel**

- C: links, rechts, links rechts
→ 1010

