LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Einführung in die Informatik:
Programmierung und Softwareentwicklung

Wintersemester 2018/19

Kapitel 14: Bäume

Prof. Dr. David Sabel

Lehr- und Forschungseinheit für Theoretische Informatik

Institut für Informatik, LMU München

WS 2018/19

Stand der Folien: 30. Januar 2019

Die Inhalte dieser Folien basieren – mit freundlicher Genehmigung – tlw. auf Folien von Prof. Dr. Rolf Hennicker aus dem WS 2017/18 und auf Folien von PD Dr. Ulrich Schöpp aus dem WS 2010/11

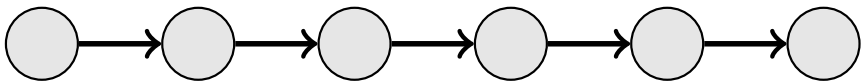
Ziele

- Den Begriff eines Baumes (und zugehörige Begriffe) in der Informatik kennenlernen
- Bäume als verkettete Datenstruktur (in Java) repräsentieren und implementieren können
- Rekursive Funktionen auf Bäumen verstehen und selbst implementieren können
- Verschiedene Möglichkeiten zum Durchlaufen von Bäumen kennenlernen (Tiefensuche und Breitensuche)

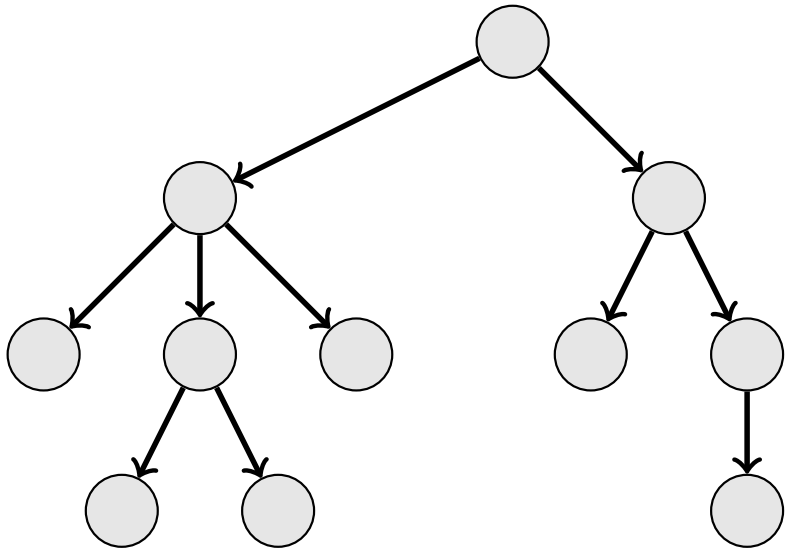
Bäume in der Informatik

- Bäume sind eine häufig verwendete Datenstruktur in der Informatik
- Bäume verallgemeinern Listen:
  - In einer Liste hat jedes Element **höchstens einen Nachfolger**
  - In einem Baum kann ein Element (Knoten) mehrere Nachfolger haben
- Bäume sind eine hierarchische Anordnung der Knoten

Liste



Baum



Bäume: Definition und Sprechweisen

- Ein (endlicher) **Baum** besteht aus einer endlichen Menge von **Knoten** N, die durch **Kanten** $N \to N'$ miteinander verbunden sind.
- In den Knoten können je nach Anwendung verschiedene Daten gespeichert sein.
- Jeder Baum (außer der leere Baum) hat einen ausgezeichneten Knoten W auf den keine Kante zeigt. (d.h. es gibt keine Kante $N \to W$ für irgendeinen Knoten N)

  Den Knoten W nennt man die **Wurzel** (oder Wurzelknoten).
- Jeder Knoten außer der Wurzel hat einen **eindeutigen Vorgänger**, d.h. für alle $N \neq W$ gibt es genau ein N' mit $N' \to N$

Bäume: Induktive Definition

Induktive Definition der Menge aller Bäume:

- Es gibt einen leeren Baum

Bäume: Induktive Definition

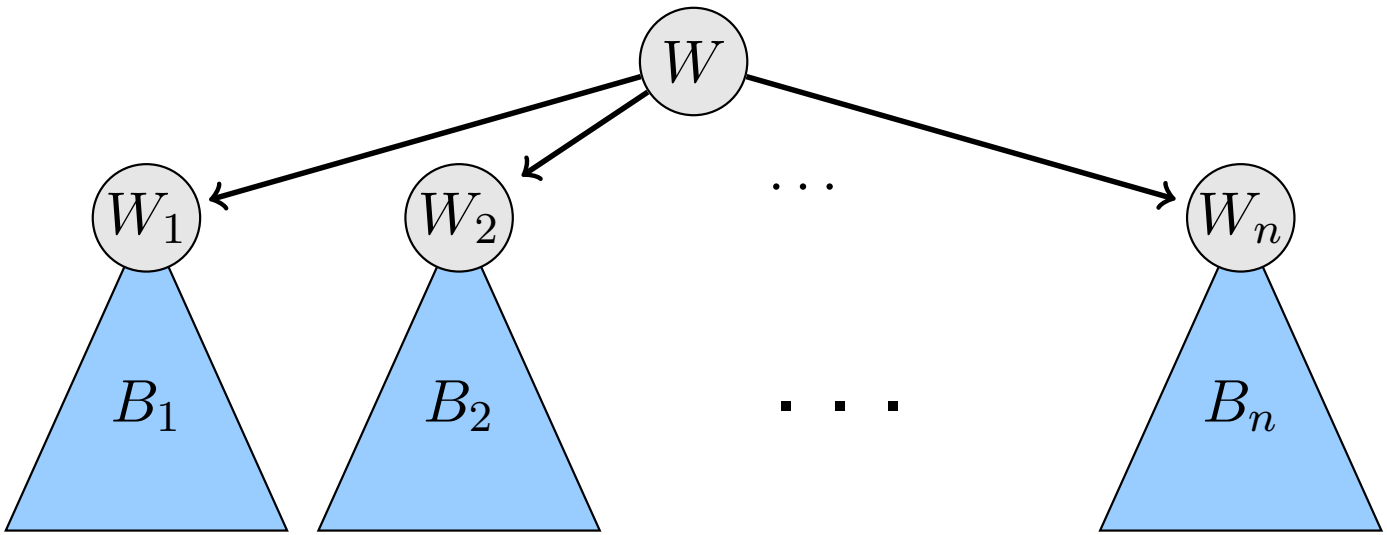
Induktive Definition der Menge aller Bäume:

- Es gibt einen leeren Baum
- Ein Blatt ist ein Baum 

Bäume: Induktive Definition

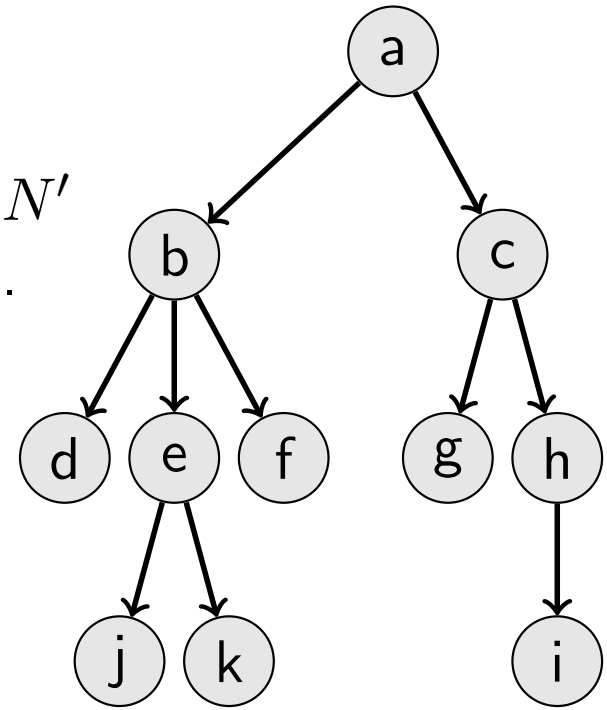
Induktive Definition der Menge aller Bäume:

- Es gibt einen leeren Baum
- Ein Blatt ist ein Baum 
- Wenn $B_1, \ldots, B_n$ (mit $n > 0$) nicht-leere Bäume mit Wurzeln $W_1, \ldots, W_n$ sind, dann kann man daraus einen größeren Baum zusammensetzen, indem man eine neue Wurzel W und Kanten $W \to W_i$ für $i = 1, \ldots, n$ hinzufügt.



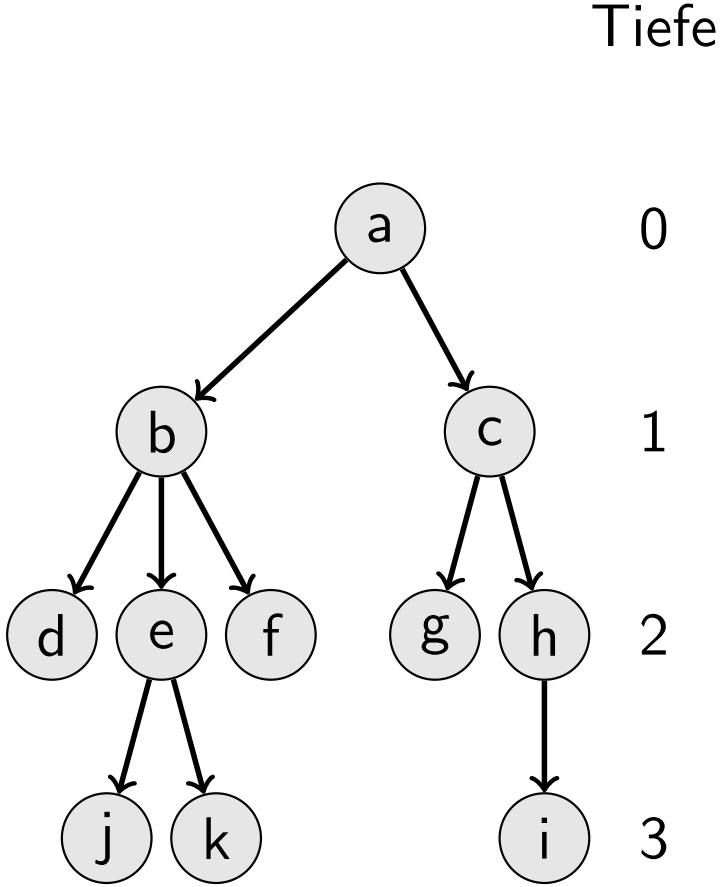
Bäume: Weitere Definitionen und Sprechweisen

- Im Beispiel: a ist die Wurzel des Baumes
- Wenn $N \to N'$, dann ist
  - N der **Elternknoten** (oder Vorgänger) von N'
    Z.B. c ist der Elternknoten von g und von h.
  - N' ein **Kind** (oder Nachfolger) von N
    Z.B. sind d, e und f Kinder von b
- Knoten, die keine Kinder haben, bezeichnet man als **Blatt**
  Z.B. sind d,j,k,f,g,i Blätter



Bäume: Weitere Definitionen und Sprechweisen (2)

- Die **Tiefe eines Knotens** im Baum ist die Anzahl der Kanten, die man auf dem Weg von der Wurzel zum Knoten durchquert
- Die **Tiefe** des Baum ist das Maximum der Tiefen aller Knoten des Baums
  Im Beispiel: 3
- Ein Baum ist ein **Binärbaum**, wenn jeder Knoten höchstens zwei Kinder hat

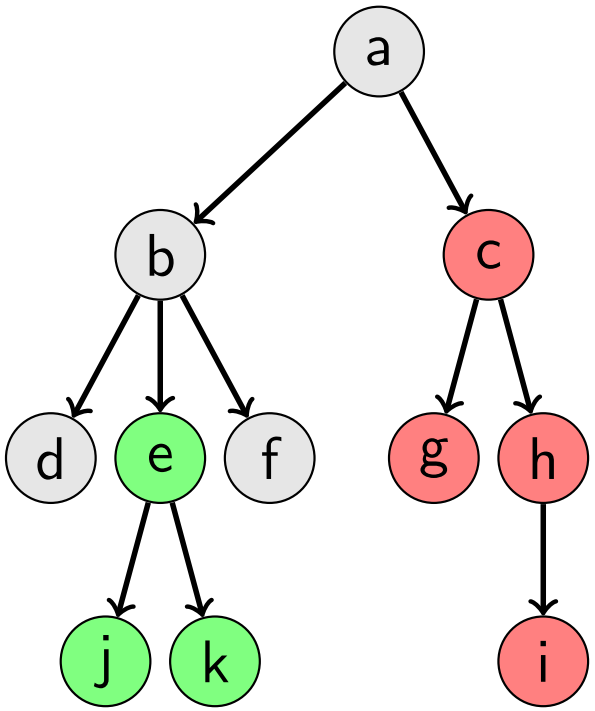


Bäume: Weitere Definitionen und Sprechweisen (3)

Jeder Knoten in einem Baum ist Wurzel eines **Teilbaums** des gegebenen Baums

Beispiele:

- Der Teilbaum mit Wurzel c ist rot markiert
- Der Teilbaum mit Wurzel e ist grün markiert
- Der Teilbaum mit Wurzel a ist der gesamte Baum
- Die Teilbäume mit Wurzel c und mit Wurzel e sind Binärbäume, die Teilbäume mit Wurzel a und mit Wurzel b sind keine Binärbaume.



Anwendungen mit Bäumen (1)

Binärbaume können zur effizienten Speicherung von Daten benutzt werden. Sie stellen einen Mittelweg zwischen Arrays und Listen dar:

Arrays

- Zugriff auf Elemente: $\mathcal{O}(1)$
- Einfügen eines Elements: $\mathcal{O}(n)$

Listen

- Zugriff auf Elemente: $\mathcal{O}(n)$
- Einfügen eines Elements: $\mathcal{O}(1)$

Geordnete Binärbäume:

- Zugriff auf Elemente: $\mathcal{O}(\log n)$
- Einfügen eines Elements: $\mathcal{O}(\log n)$

Anwendungen mit Bäumen (2)

- **Datenbanken**:
  Bäume werden zur indizierten Speicherung großer Datenmengen benutzt.
- **Betriebssysteme**:
  Dateisysteme sind in Baumstrukturen organisiert und gespeichert
- Baumartige Datenstrukturen sind in verschiedenen Anwendungen nützlich.
- Als Beispiel betrachten wir im Folgenden die Anwendung von Huffman-Bäumen zur Datenkomprimierung

Beispielanwendung: Huffman-Kodierung (1)

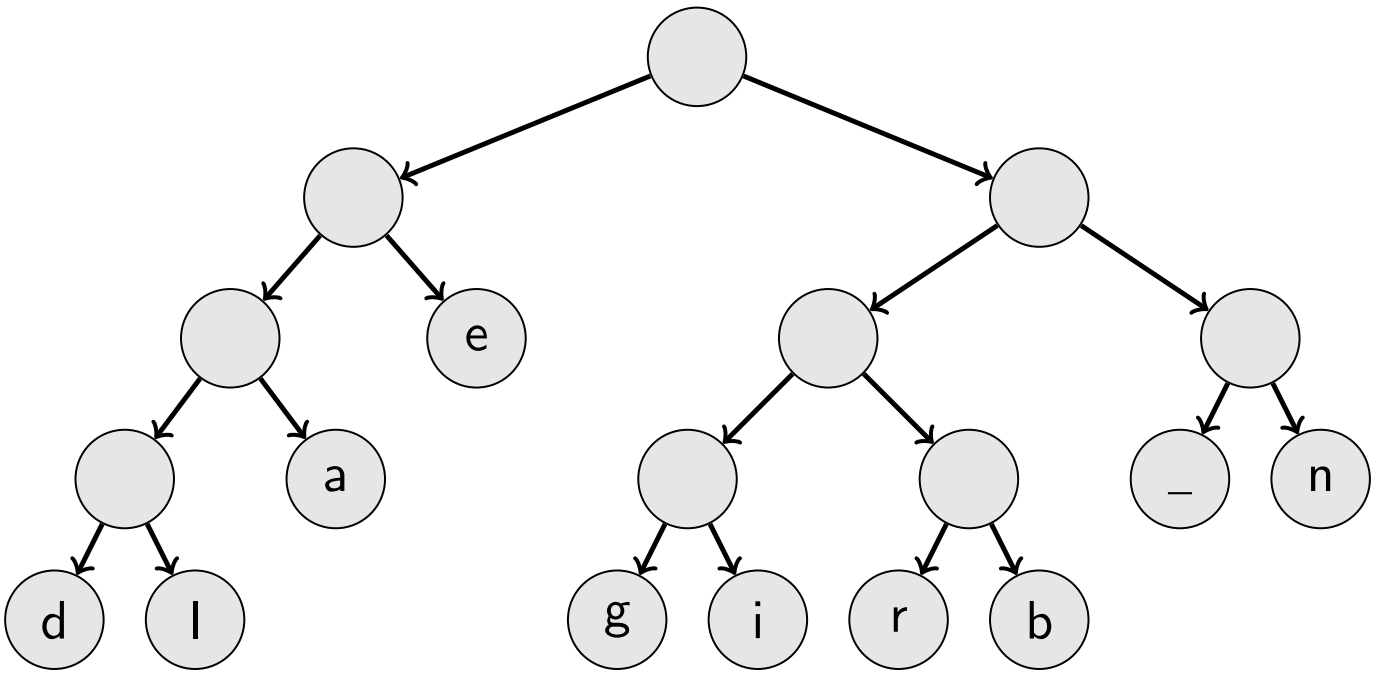
Bei der Speicherung von Text werden üblicherweise die einzelnen Zeichen kodiert und die Kodes der Zeichen dann hintereinander geschrieben.

- ASCII-Kodierung: Ein Zeichen wird mit 7 Bit kodiert
- Java benutzt Unicode: ein Zeichen benötigt 16 Bit.
  - Alle Zeichen, die einen ASCII-Kode haben, haben denselben Wert im Unicode.
  - Unicode kann viele weitere Zeichen kodieren.
- Der 34 Zeichen lange Text *„lege an eine brandnarbe nie naegel“* hat mit ASCII-Kodierung eine Länge von $34 \cdot 7$ Bit $= 238$ Bit.
- $\underbrace{1101100}_{l}\underbrace{1100101}_{e}\underbrace{1100111}_{g}\underbrace{1100101}_{e}\ldots$
- In Texten kommen nicht alle Zeichen gleich häufig vor.
  Es wäre effizienter, für oft vorkommende Zeichen einen kürzeren Kode zu verwenden.

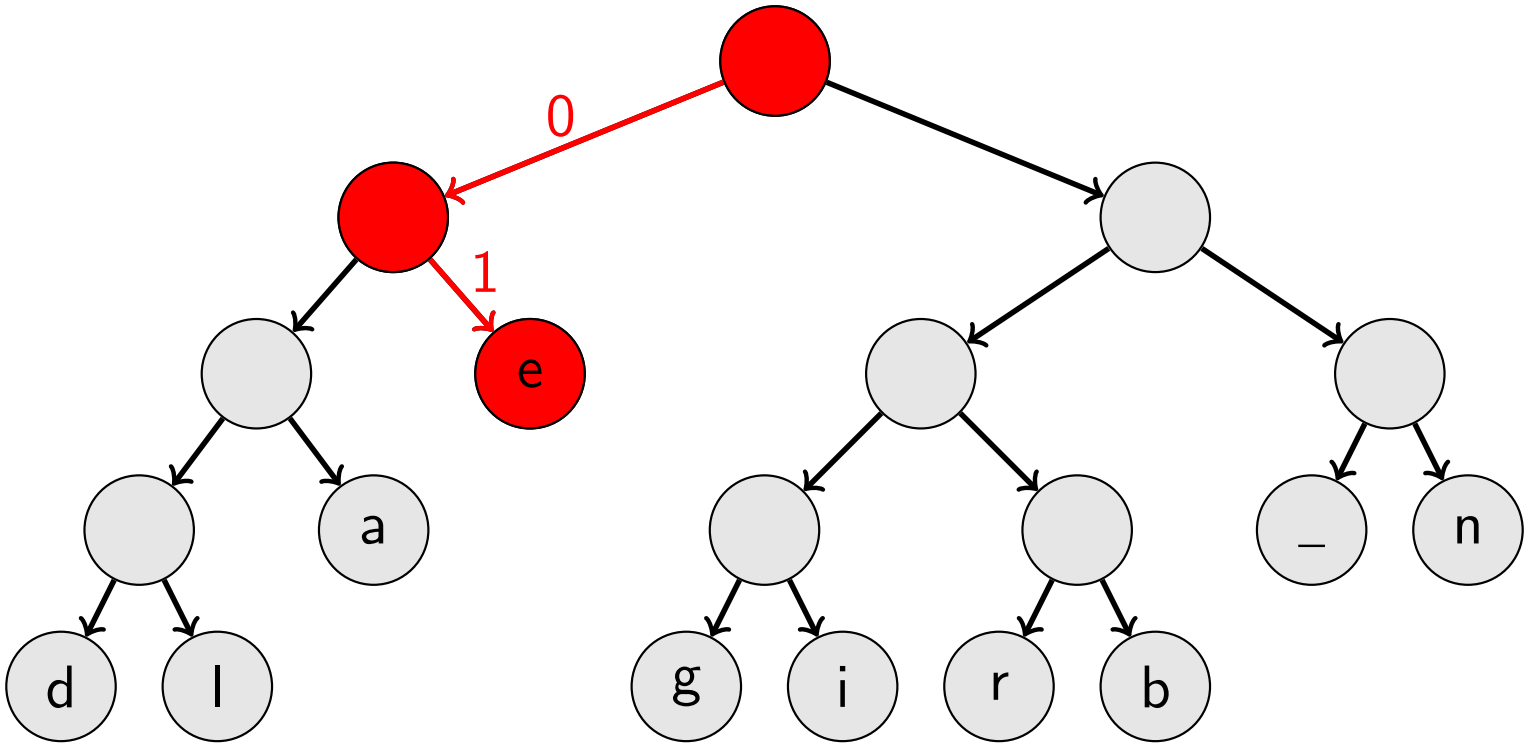
Beispielanwendung: Huffman-Kodierung (2)

- Die **Huffman-Kodierung** gibt häufig vorkommenden Zeichen einen kurzen Kode. Die Kodierung wird z.B. im ZIP-Format benutzt.
- Anhand der Häufigkeiten der einzelnen Buchstaben wird ein **Huffman-Baum** aufgestellt. (Wir erläutern die Erstellung nicht)
- Nicht-Blatt Knoten haben keine besondere Markierung, Blätter sind mit den Buchstaben markiert.

Beispiel:

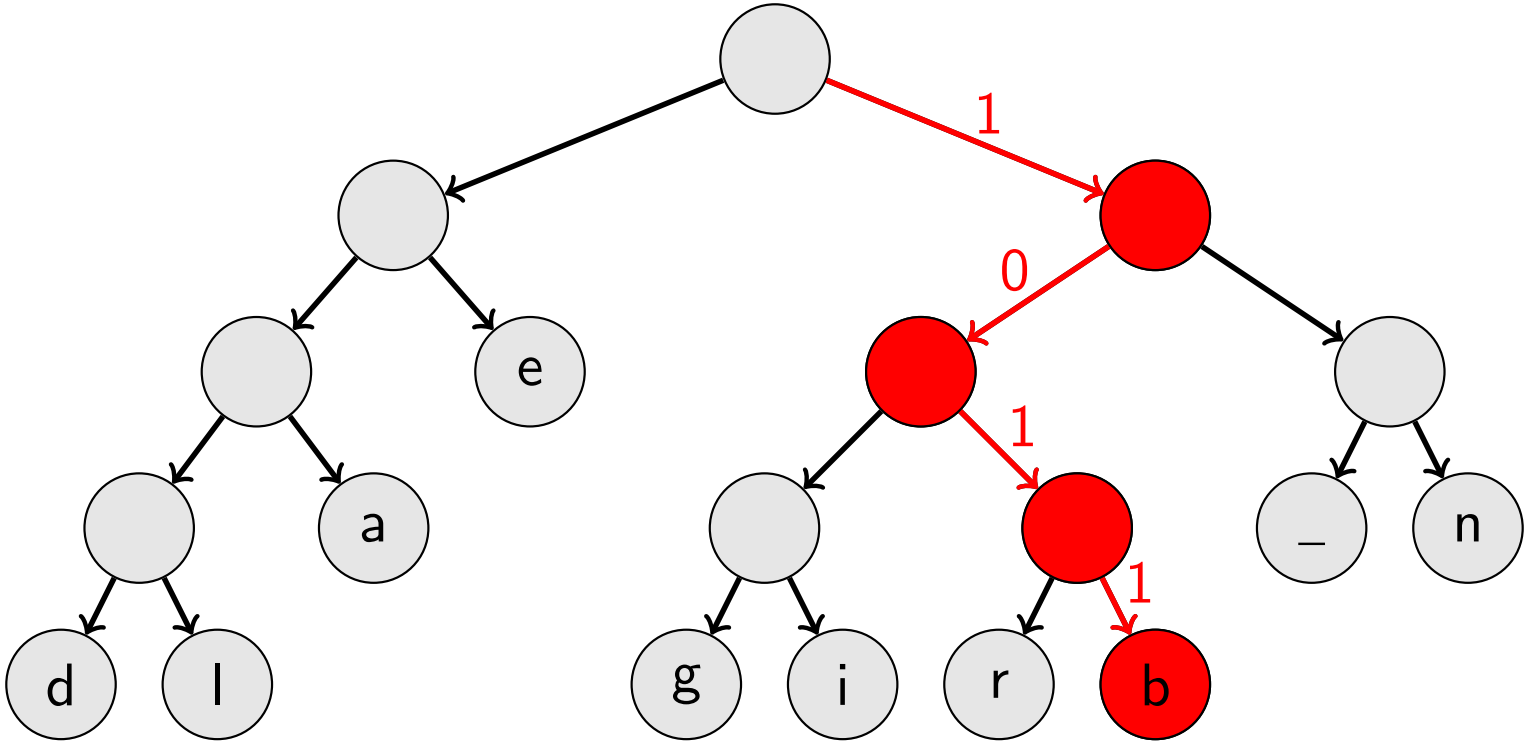


Beispielanwendung: Huffman-Kodierung (3)



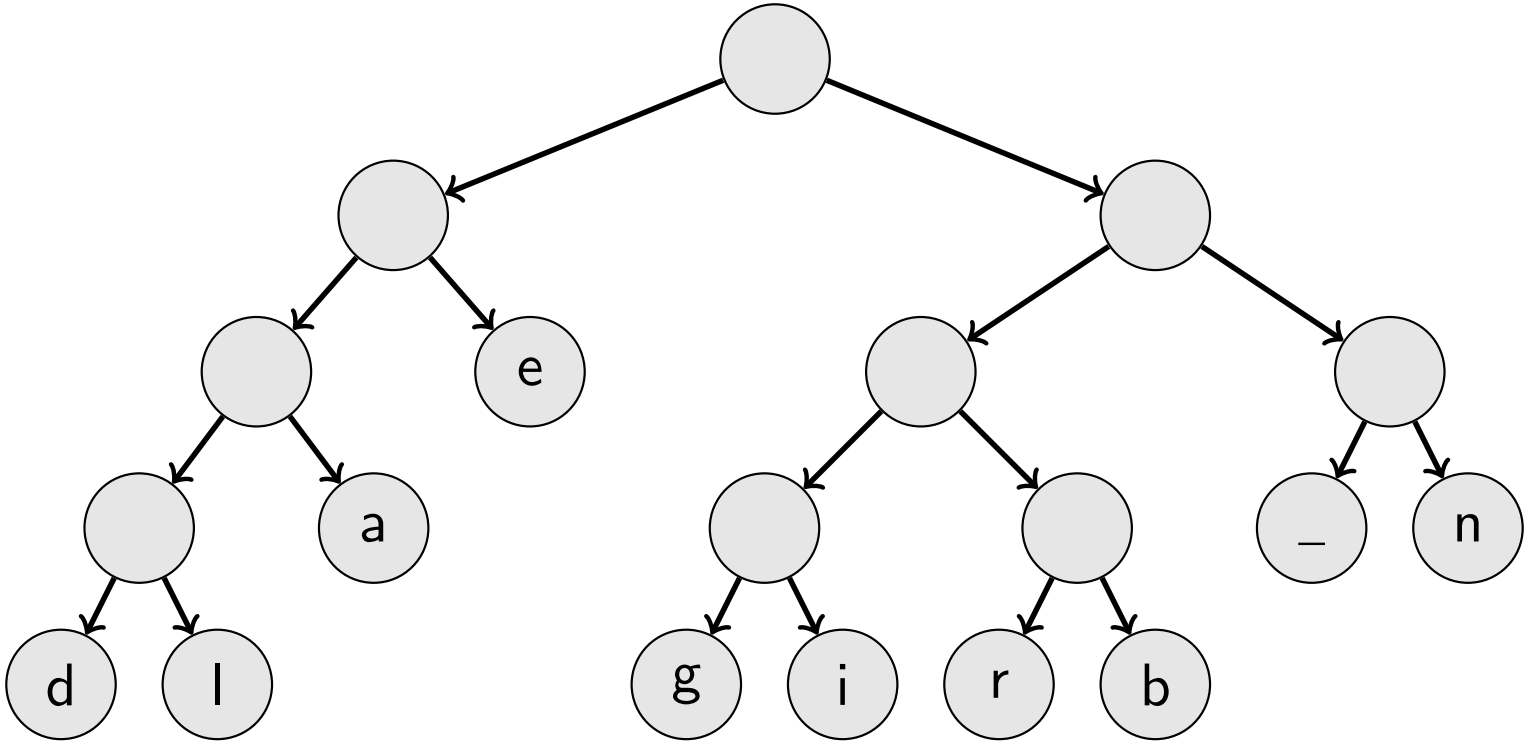
- Der Kode eines Zeichens ist die Liste der Entscheidungen, die man treffen muss, um von der Wurzel im Huffman-Baum zu dem Zeichen zu gelangen (0 = gehe links, 1 = gehe rechts).
- e = 01

Beispielanwendung: Huffman-Kodierung (3)



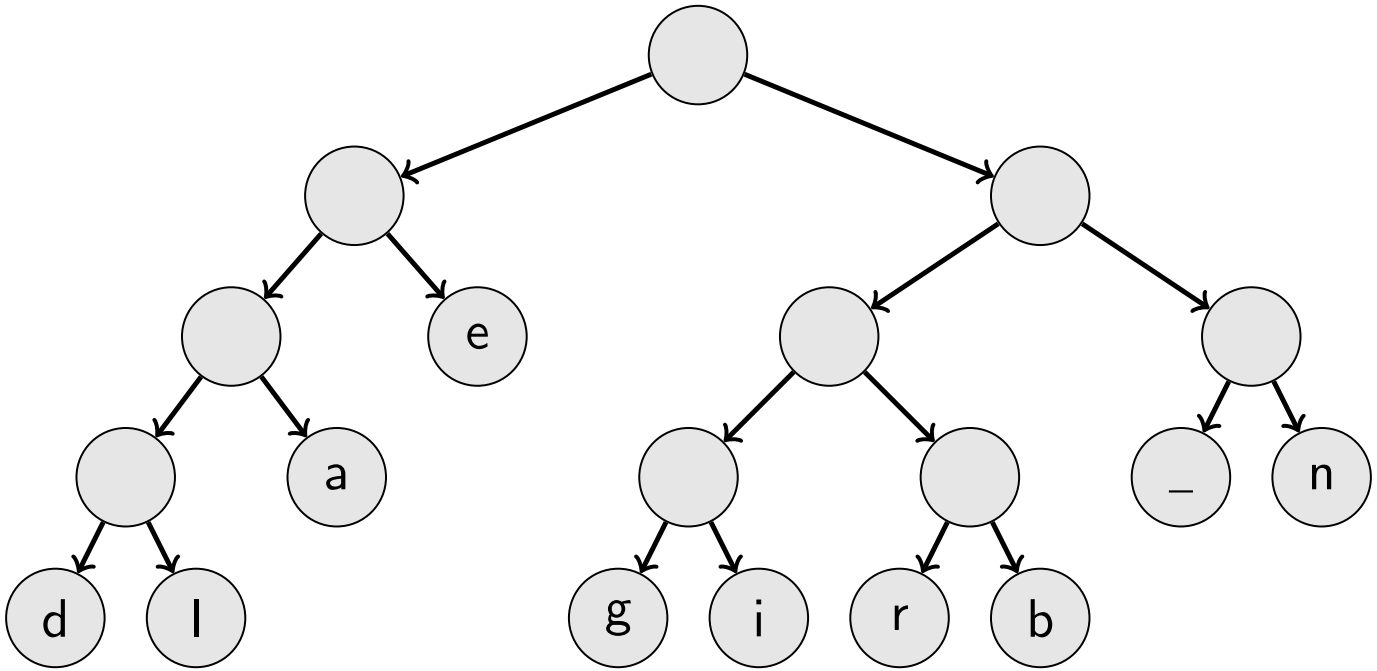
- Der Kode eines Zeichens ist die Liste der Entscheidungen, die man treffen muss, um von der Wurzel im Huffman-Baum zu dem Zeichen zu gelangen (0 = gehe links, 1 = gehe rechts).
- e = 01 und b = 1011

Beispielanwendung: Huffman-Kodierung (3)



- Der Kode eines Zeichens ist die Liste der Entscheidungen, die man treffen muss, um von der Wurzel im Huffman-Baum zu dem Zeichen zu gelangen (0 = gehe links, 1 = gehe rechts).
- e = 01 und b = 1011
- Der Baum wird so gewählt, dass häufig vorkommende Zeichen einen kurzen Kode haben.

Beispielanwendung: Huffman-Kodierung (4)



Der Kode

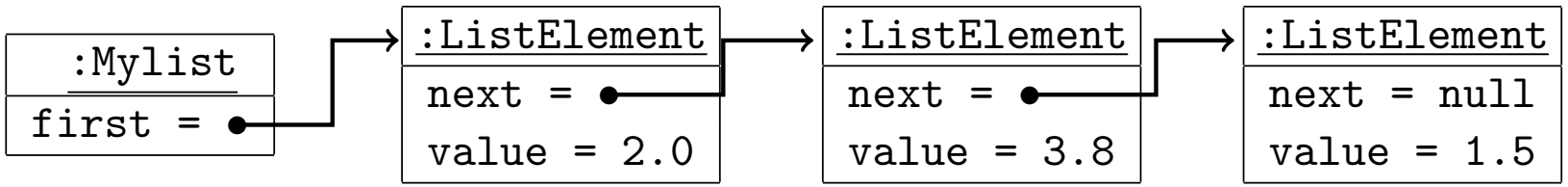
$$\underbrace{0}_{l}\underbrace{0}_{e}\underbrace{1}_{g}\underbrace{0}_{e}\underbrace{1}_{\_}\underbrace{1}_{a}\underbrace{0}_{n}\underbrace{0}_{\_}\underbrace{1}_{e}\underbrace{1}_{i}\underbrace{1}_{n}\underbrace{1}_{e}\underbrace{0}_{\_}\underbrace{0}_{b}\underbrace{1}_{r}\underbrace{1}_{a}\underbrace{1}_{n}\underbrace{0}_{d}$$

$$\underbrace{1}_{n}\underbrace{1}_{a}\underbrace{0}_{r}\underbrace{1}_{b}\underbrace{0}_{e}\underbrace{1}_{\_}\underbrace{1}_{n}\underbrace{0}_{i}\underbrace{1}_{e}\underbrace{1}_{\_}\underbrace{0}_{n}\underbrace{1}_{a}\underbrace{1}_{e}\underbrace{0}_{g}\underbrace{1}_{e}\underbrace{0}_{l}$$

benötigt 105 Bits statt 238 Bits!

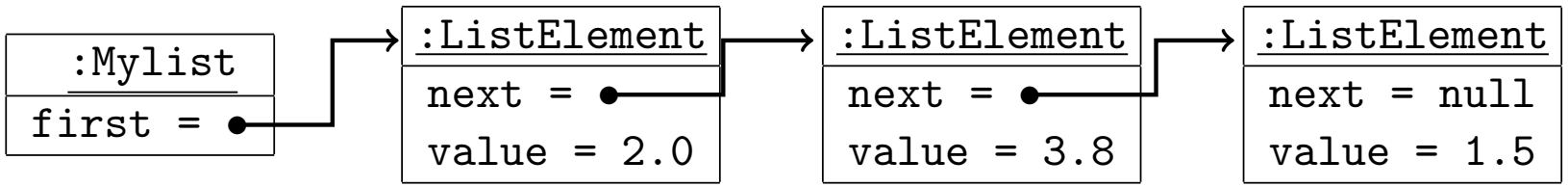
Binärbäume in Java

- Bäume können in Java als verkettete Datenstruktur ganz ähnlich wie verkettete Listen implementiert werden.
- Zur Erinnerung: Speicherung einer Liste durch eine Kette von Objekten:

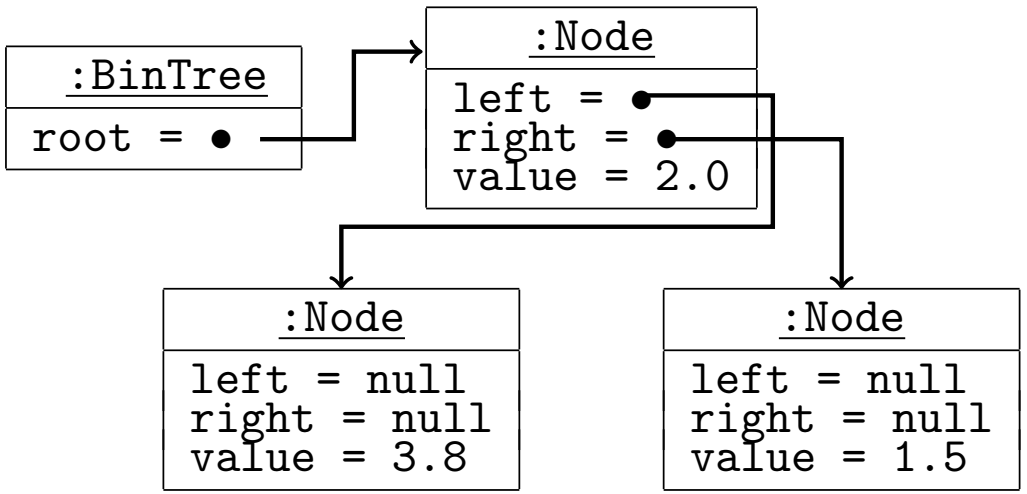


Binärbäume in Java

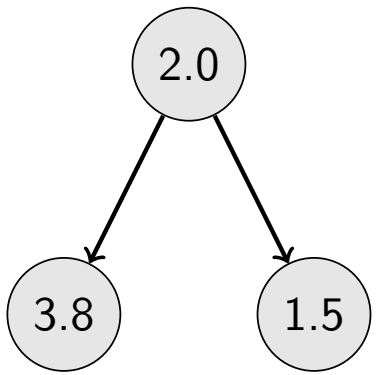
- Bäume können in Java als verkettete Datenstruktur ganz ähnlich wie verkettete Listen implementiert werden.
- Zur Erinnerung: Speicherung einer Liste durch eine Kette von Objekten:



- Anstelle **eines** Nachfolgerzeigers `next` haben die Elemente bei Binärbäumen nun **zwei** Nachfolgerzeiger `left` und `right`

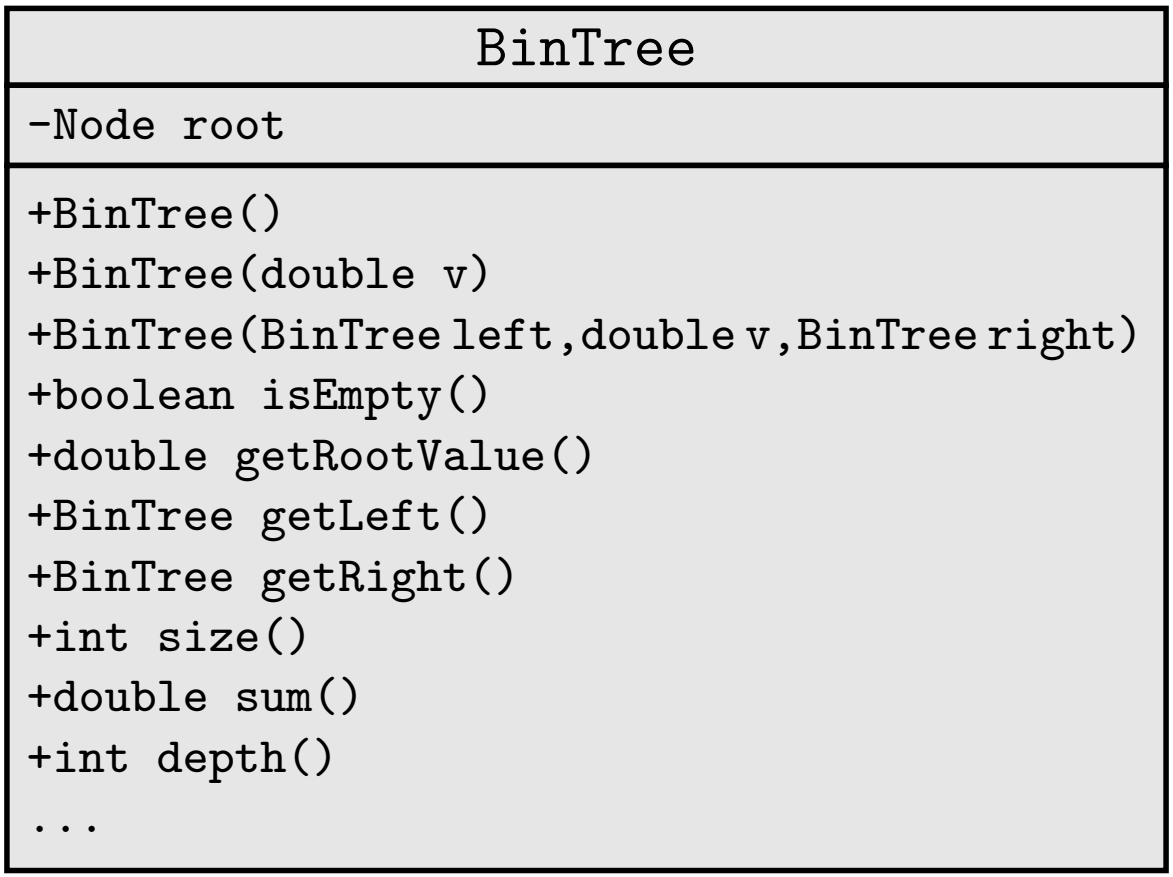


repräsentiert den Baum:



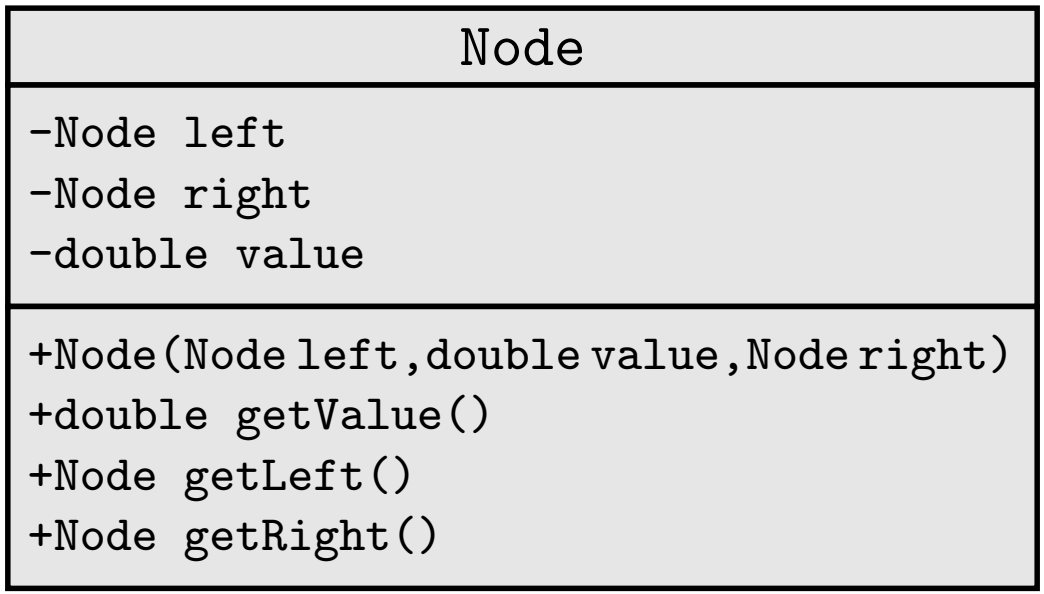
Klassen für Binärbäume

`BinTree`: Klasse für Binärbäume, die Methoden für Operationen auf Binärbäumen bereitstellt. Objekte der Klasse `BinTree` haben als Attribut eine Referenz auf den Wurzelknoten des Baumes.



Klassen für Binärbäume (2)

`Node`: Klasse, deren Objekte Baumknoten darstellen. Objekte haben einen `double`-Wert und zwei Referenzen: Eine auf die Wurzel des linken Teilbaums und eine auf die Wurzel des rechten Teilbaums. Die Referenz ist jeweils leer (null), wenn es keinen entsprechenden Teilbaum gibt.

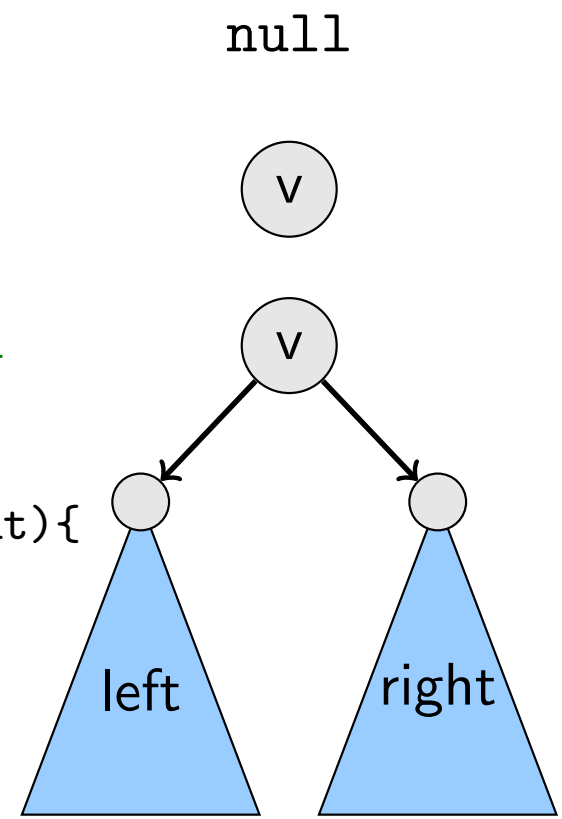


Implementierung der Klasse Node

```
public class Node {
 private Node left;
 private double value;
 private Node right;

 public Node(Node left, double value, Node right){
  this.left = left;
  this.value = value;
  this.right = right;
 }
 public double getValue() {return this.value;}
 public Node getLeft() {return this.left;}
 public Node getRight() {return this.right;}
}
```

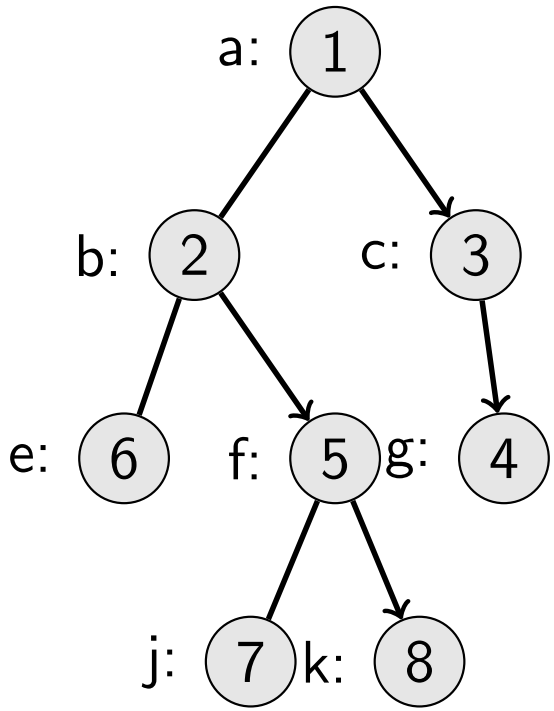
Die Klasse BinTree in Java: Konstruktoren

```
public class BinTree {
 private Node root;
 /** Erzeugt leeren Baum */
 public BinTree() {this.root = null;}
 /** Erzeugt Baum mit einem einzigen Knoten */
 public BinTree(double v){
  this.root = new Node(null,v,null);
 }
 /** Konstruiert Baum aus zwei gegebenen Teilbaeumen
  * und einer neuen Wurzel mit Wert v
  */
 public BinTree(BinTree left, double v, BinTree right){
  this.root = new Node(left.root,v,right.root);
 }
```



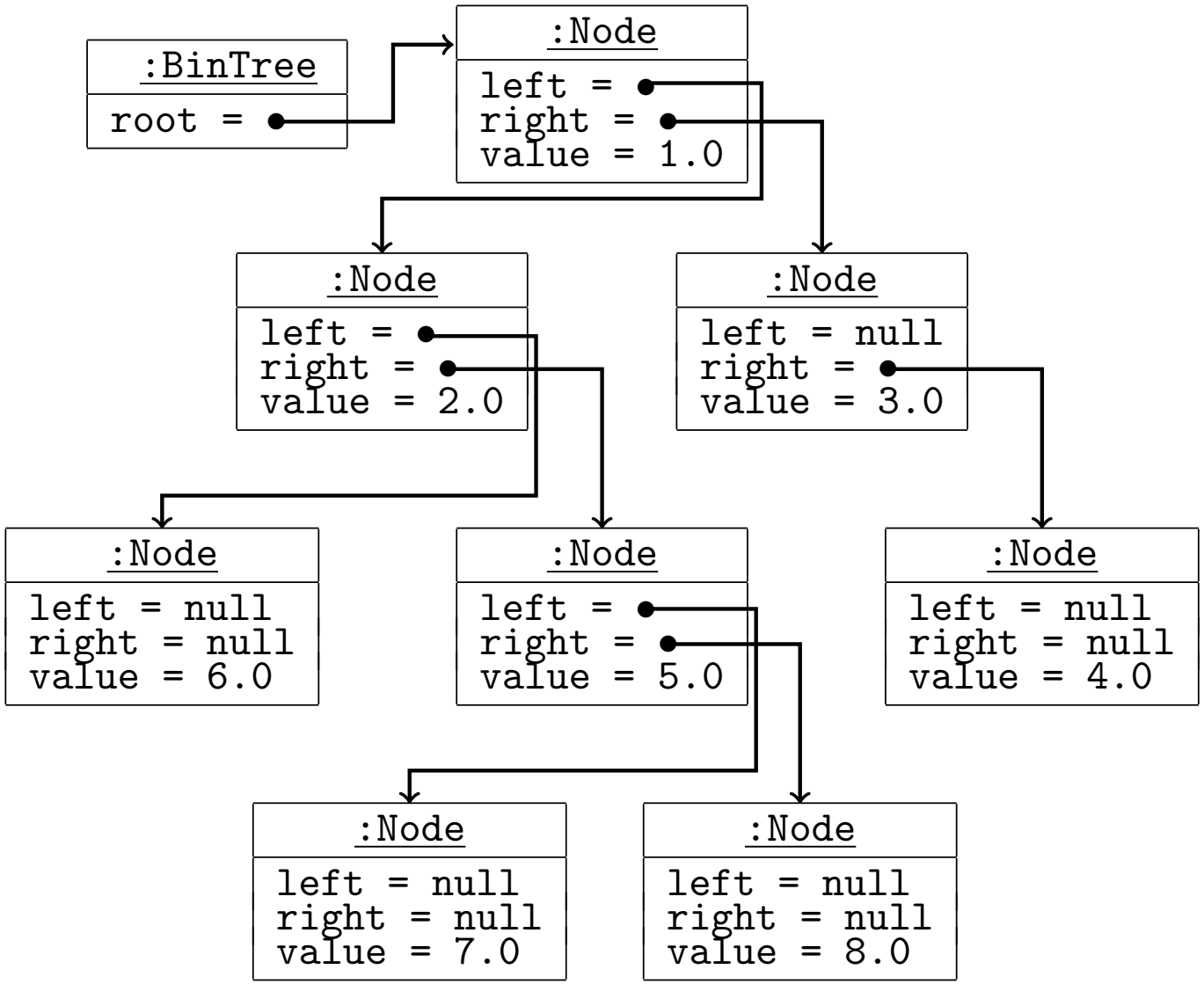
Beispiel für die Konstruktion eines Baums

```
public class BinTreeTest {
 public static void main(String[] args) {
  BinTree k = new BinTree(8);
  BinTree j = new BinTree(7);
  BinTree e = new BinTree(6);
  BinTree f = new BinTree(j,5,k);
  BinTree g = new BinTree(4);
  BinTree b = new BinTree(e,2,f);
  BinTree empty = new BinTree();
  BinTree c = new BinTree(empty,3,g);
  BinTree a = new BinTree(b,1,c);
 }
}
```



Beachte: Die Variablen repräsentieren jeweils den Teilbaum mit dem gleichnamigen Wurzelknoten

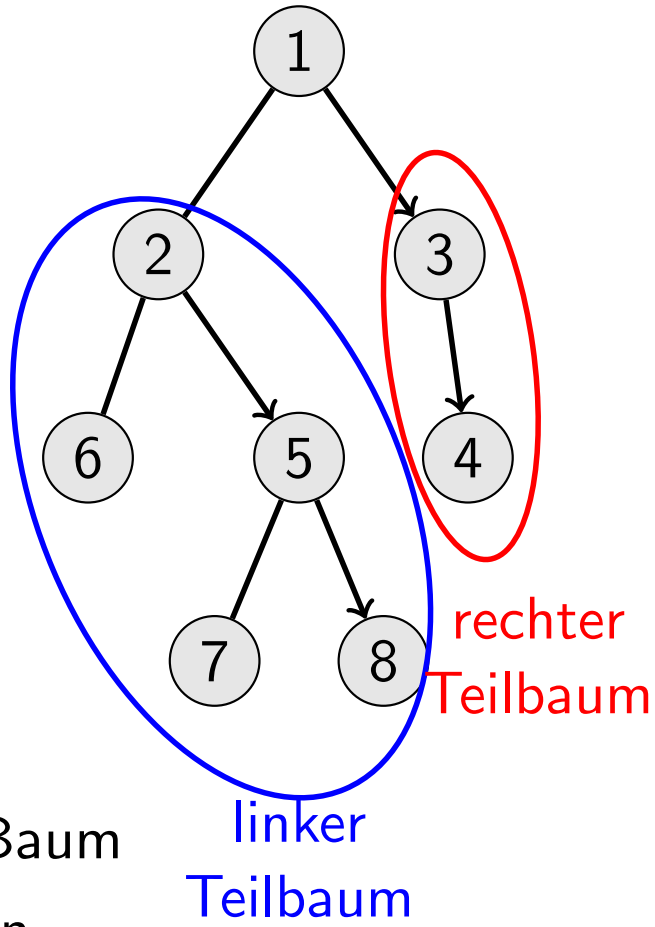
Der Baum im Heap



Operationen auf Binärbäumen

Wir implementieren folgende typische Operationen auf Binärbäumen

- `isEmpty` testet, ob ein Baum leer ist.
- Selektoren:
  - `getRootValue()` gibt den Wert an der Wurzel zurück
  - `getLeft()` gibt den linken Teilbaum unter der Wurzel zurück
  - `getRight()` gibt den rechten Teilbaum unter der Wurzel zurück
- `size()` berechnet die Anzahl der Knoten im Baum
- `sum()` berechnet die Summe der in den Knoten gespeicherten Zahlen.
- `depth()` berechnet die Tiefe des Baums



Die Klasse BinTree in Java: Methoden

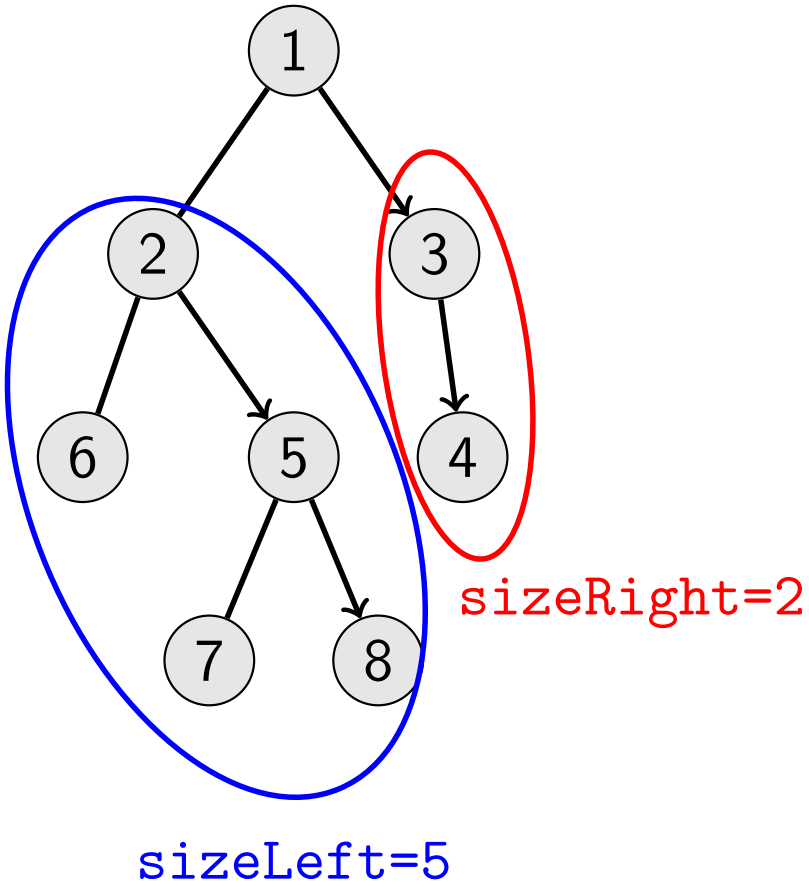
```
import java.util.NoSuchElementException;

public class BinTree {
 private Node root;
 ...

 /** Prueft, ob der Baum leer ist */
 public boolean isEmpty() {
  return this.root == null;
 }

 /** Liefert den Wert der Wurzel zurueck */
 public double getRootValue() {
  if (this.root == null) {
  throw new NoSuchElementException("Ein leerer Baum hat keine
      Wurzel!");
  }
  else {return this.root.getValue();}
 }
```

Die Klasse BinTree in Java: Methoden (2)

```
...
/** Liefert den linken Teilbaum unter der Wurzel */
public BinTree getLeft() {
 if (this.root == null) {
   throw new NoSuchElementException(
         "Ein leerer Baum hat keinen linken Teilbaum");
   }  else { BinTree result = new BinTree();
            result.root = this.root.getLeft();
            return result; }
}
/** Liefert den rechten Teilbaum unter der Wurzel */
public BinTree getRight() {
 if (this.root == null) {
   throw new NoSuchElementException(
         "Ein leerer Baum hat keinen rechten Teilbaum");
   } else { BinTree result = new BinTree();
            result.root = this.root.getRight();
            return result; }
}
```

Anzahl der Knoten

- Ein leerer Baum hat 0 Knoten
- Falls, der Baum nicht leer ist:
  - Berechne die Anzahl `sizeLeft` der Knoten im linken Teilbaum
  - Berechne die Anzahl `sizeRight` der Knoten im rechten Teilbaum
  - Gesamtanzahl der Knoten: $1 +$ `sizeLeft` $+$ `sizeRight`



```
...
public int sizeElegant() {
 if (this.isEmpty()) {return 0;} // Basisfall
 else {return 1 + this.getLeft().sizeElegant()
                + this.getRight().sizeElegant();
      }
}
...
```

Die Implementierung der Methode ist elegant, aber ziemlich ineffizient:
Grund: Die Aufrufe getLeft() und getRight() erzeugen je ein neues BinTree-Objekt, dass danach direkt wieder verworfen wird.

Effizientere Variante der Anzahl an Knoten

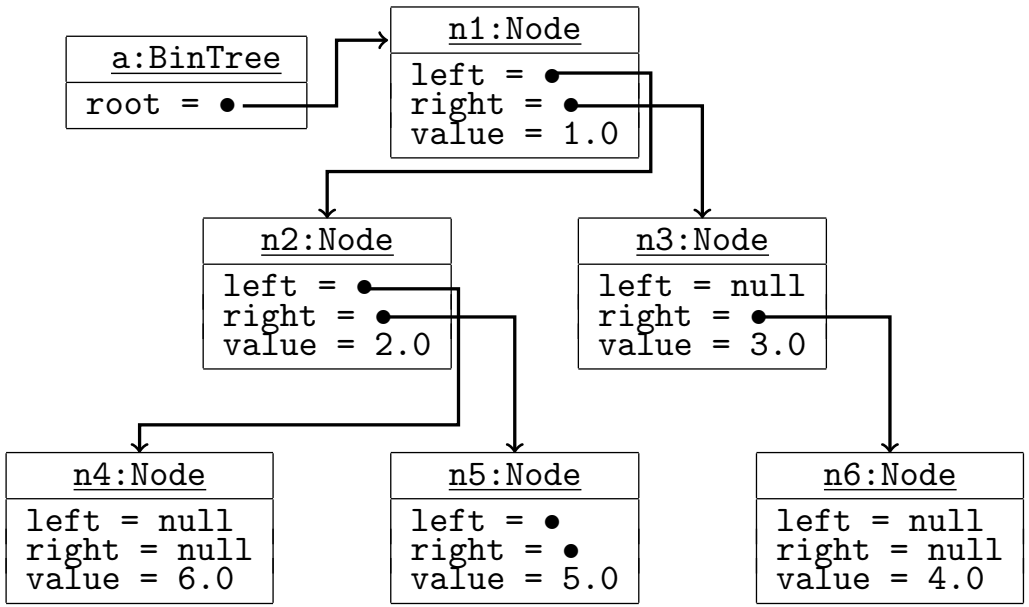
- Idee: Berechne die Anzahl direkt in der `Node`-Klasse
- D.h. füge der Node-Klasse eine `size()`-Methode hinzu
- Die `size()`-Methode der BinTree-Klasse ruft die `size()`-Methode für den Wurzelknoten auf.

Implementierung der effizienteren Größenberechung

```
public class BinTree {
 ...
 public int size() {
  if (this.isEmpty()) {return 0;}
  else {return this.root.size();}
 }
}
public class Node {
 private Node left;
 private double value;
 private Node right;
 ...
 public int size() {
   int sizeLeft = 0;
   int sizeRight = 0;
   if (this.left != null) {sizeLeft = this.left.size();}
   if (this.right != null) {sizeRight = this.right.size();}
   return 1+ sizeLeft + sizeRight;
 }
}
```

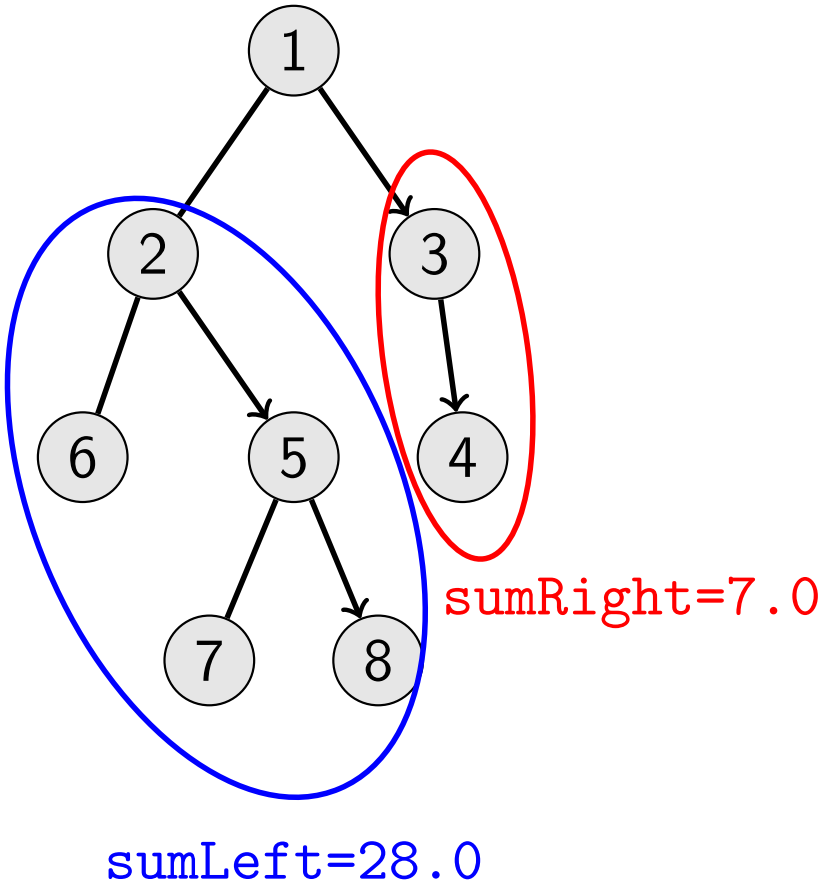
Ablauf der Berechnung

a.size()
→n1.size()
→n2.size()
→n4.size()
return 1
→n5.size()
return 1
return 1+1+1 = 3
→n3.size()
→n6.size()
return 1
return 1+1+0 = 2
return 3+2+1 = 6
return 6



Summe der Werte aller Knoten

- Ein leerer Baum hat Knotensumme 0
- Falls, der Baum nicht leer ist:
  - Berechne die Knotensumme `sumLeft` der Knoten im linken Teilbaum
  - Berechne die Knotensumme `sumRight` der Knoten im rechten Teilbaum
  - Knotensumme: value + `sumLeft` + `sumRight`



Knotensumme elegant

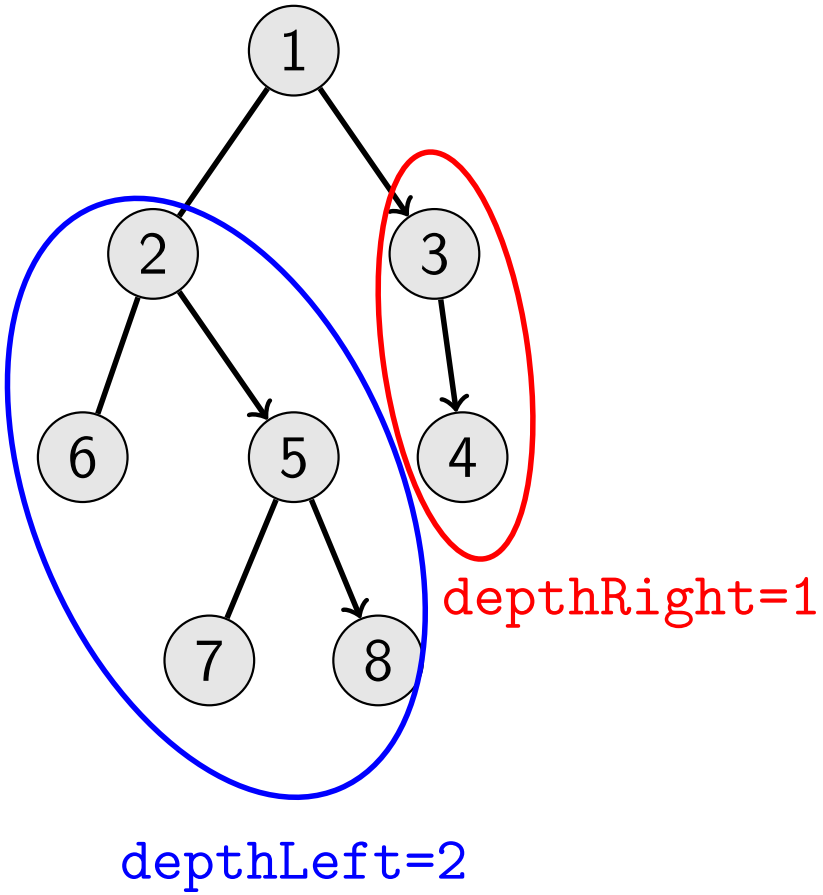
```
public class BinTree {
 ...
 /** Elegante (aber ineffiziente) Berechnung der Knotensumme */
 public double sumElegant() {
  if (this.isEmpty()) {return 0;} // Basisfall
  else {return
            this.getRootValue()+this.getLeft().sumElegant() +
                this.getRight().sumElegant();
       }
 }
```

Knotensumme effizienter

```
public class BinTree {
 ...
 /** Effizientere Berechnung der Knotensumme */
 public double sum() {
  if (this.isEmpty()) {return 0;} // Basisfall
  else {return this.root.sum();}
 }
public class Node {
 ...
 public double sum() {
   double sumLeft = 0;
   double sumRight = 0;

   if (this.left != null) {sumLeft = this.left.sum();}
   if (this.right != null) {sumRight = this.right.sum();}
   return this.value + sumLeft + sumRight;
 }
```

Tiefe eines Baums berechnen

- Ein leerer Baum hat keine Tiefe
- Die Wurzel hat Tiefe 0
- Falls, die Wurzel noch Kinder hat:
  - Berechne die Tiefe `depthLeft` des linken Teilbaums (falls dieser existiert)
  - Berechne die Tiefe `depthRight` des rechten Teilbaums (falls dieser existiert)
  - Tiefe:
    $1 + \max(\text{depthLeft}, \text{depthRight})$



Tiefe elegant

```
public class BinTree {
 ...
 /** Elegante (aber ineffiziente) Berechnung der Tiefe */
 public int depthElegant() {
  if (this.isEmpty()) {throw new IllegalArgumentException(
                              "Ein leerer Baum hat keine Tiefe.");
  }
  if (this.getLeft().isEmpty()) {
   if (this.getRight().isEmpty()) {return 0;} // Keine Kinder
   else {
    return 1+this.getRight().depthElegant(); // nur rechts ein Kind
   }
  }
  else {
   if (this.getRight().isEmpty()) {
    return 1+this.getLeft().depthElegant(); // nur links ein Kind
   }
   else {return 1+Math.max(this.getLeft().depthElegant()
                              ,this.getRight().depthElegant());
   }
  }
```

Tiefe effizienter

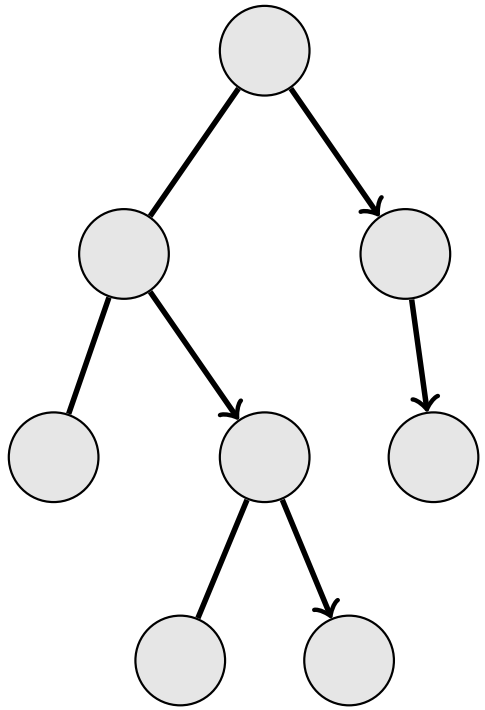
```
public class BinTree {
 ...
/** Effizientere Berechnung der Tiefe */
 public int depth() {
  if (this.isEmpty()) {throw new IllegalArgumentException(
                        "Ein leerer Baum hat keine Tiefe.");
   }
  else {return this.root.depth();}
 }
public class Node {
 ...
 public int depth() {
    int depthLeft = -1;
    int depthRight = -1;
    if (this.left != null) {depthLeft = this.left.depth();}
    if (this.right != null) {depthRight = this.right.depth();}
    return 1+Math.max(depthLeft,depthRight);
 }
}
```

Durchlaufen eines Baums

- Wie bei Listen möchte man über alle Elemente des Baums iterieren können
- Bei Listen gibt es **eine** natürliche Reihenfolge des Durchlaufs, von vorn nach hinten
- Bei Bäumen gibt es **mehrere** sinnvolle Reihenfolgen des Durchlaufs, die je nach Anwendungsfall alle benutzt werden.
- Wir betrachten die beiden bekanntesten:
  - Tiefendurchlauf (depth-first traversal)
  - Breitendurchlauf (breadth-first-traversal)

Tiefendurchlauf (depth-first traversal)

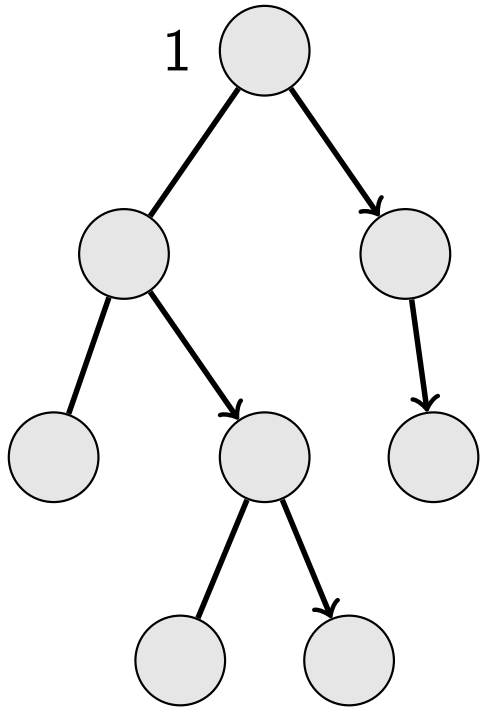
1. Besuche die Wurzel
2. Besuche linken Teilbaum mit Tiefendurchlauf
3. Besuche rechten Teilbaum mit Tiefendurchlauf



Reihenfolge der Knoten
in einem Tiefendurchlauf

Tiefendurchlauf (depth-first traversal)

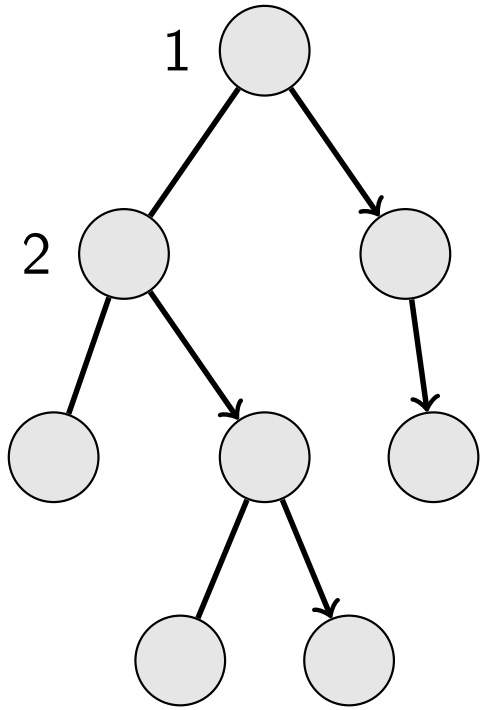
1. Besuche die Wurzel
2. Besuche linken Teilbaum mit Tiefendurchlauf
3. Besuche rechten Teilbaum mit Tiefendurchlauf



Reihenfolge der Knoten
in einem Tiefendurchlauf

Tiefendurchlauf (depth-first traversal)

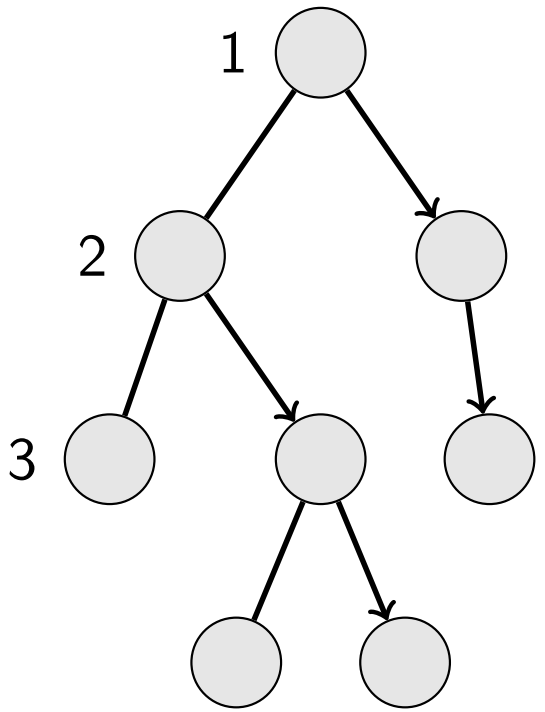
1. Besuche die Wurzel
2. Besuche linken Teilbaum mit Tiefendurchlauf
3. Besuche rechten Teilbaum mit Tiefendurchlauf



Reihenfolge der Knoten
in einem Tiefendurchlauf

Tiefendurchlauf (depth-first traversal)

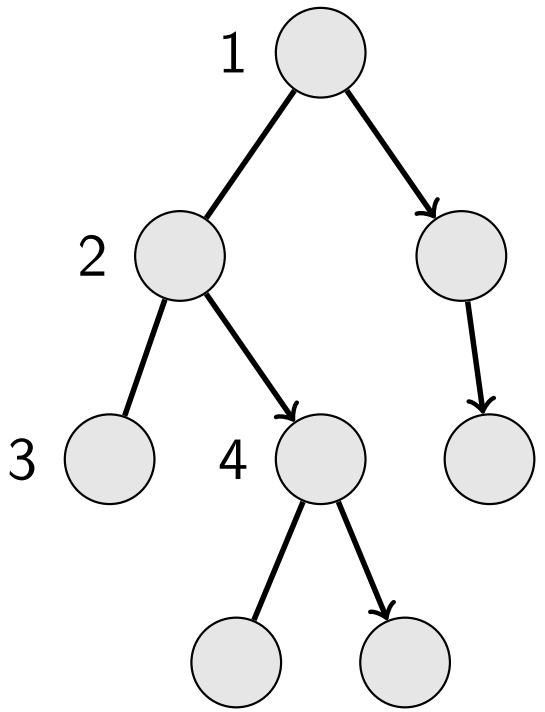
1. Besuche die Wurzel
2. Besuche linken Teilbaum mit Tiefendurchlauf
3. Besuche rechten Teilbaum mit Tiefendurchlauf



Reihenfolge der Knoten in einem Tiefendurchlauf

Tiefendurchlauf (depth-first traversal)

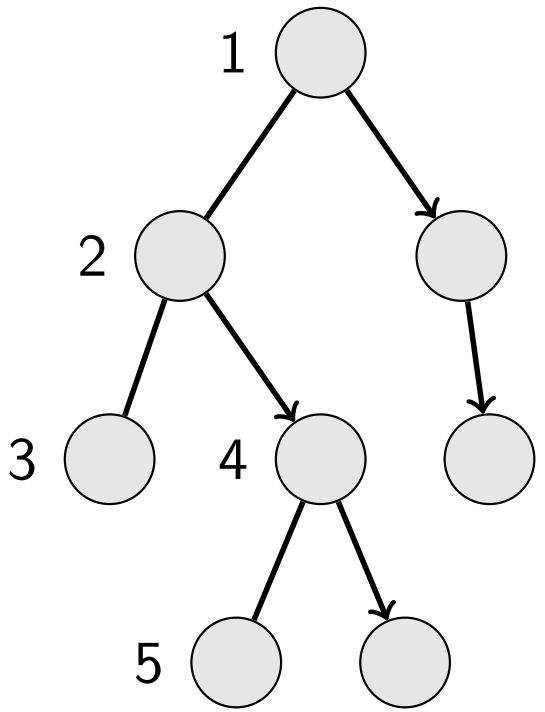
1. Besuche die Wurzel
2. Besuche linken Teilbaum mit Tiefendurchlauf
3. Besuche rechten Teilbaum mit Tiefendurchlauf



Reihenfolge der Knoten in einem Tiefendurchlauf

Tiefendurchlauf (depth-first traversal)

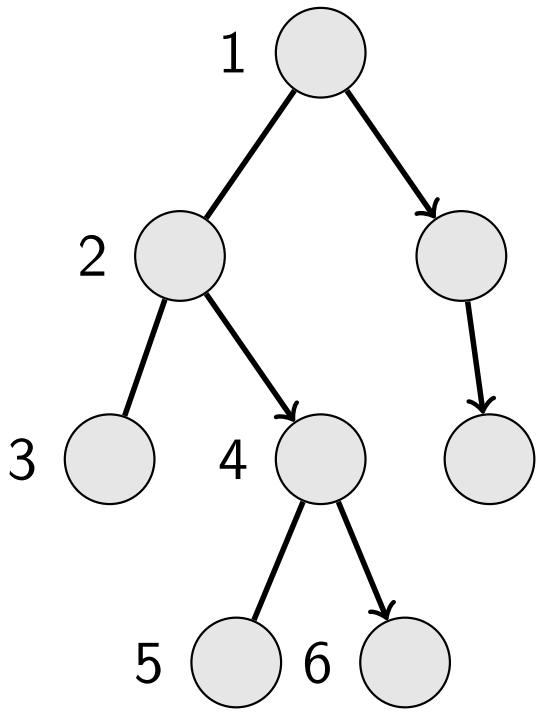
1. Besuche die Wurzel
2. Besuche linken Teilbaum mit Tiefendurchlauf
3. Besuche rechten Teilbaum mit Tiefendurchlauf



Reihenfolge der Knoten in einem Tiefendurchlauf

Tiefendurchlauf (depth-first traversal)

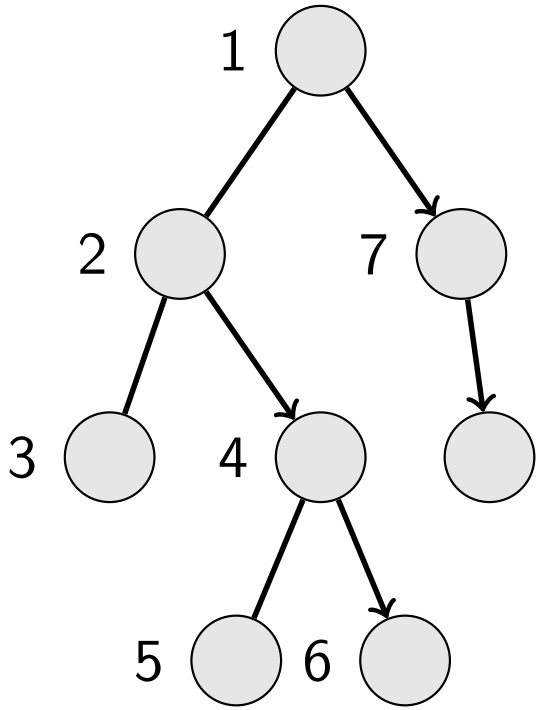
1. Besuche die Wurzel
2. Besuche linken Teilbaum mit Tiefendurchlauf
3. Besuche rechten Teilbaum mit Tiefendurchlauf



Reihenfolge der Knoten in einem Tiefendurchlauf

Tiefendurchlauf (depth-first traversal)

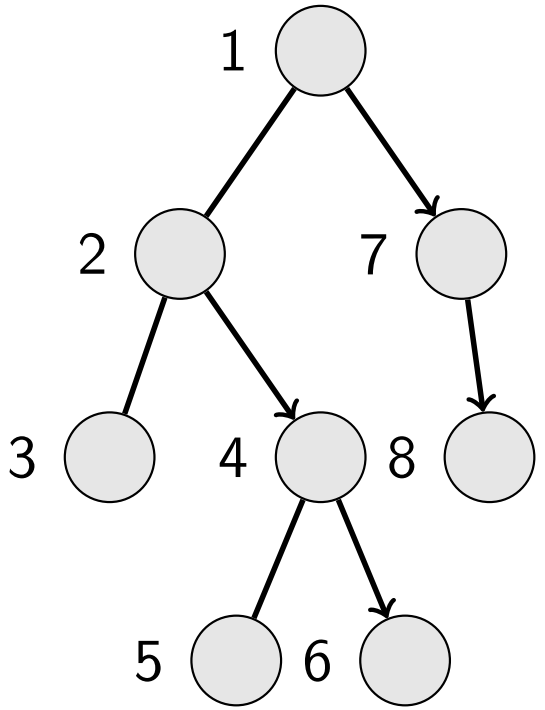
1. Besuche die Wurzel
2. Besuche linken Teilbaum mit Tiefendurchlauf
3. Besuche rechten Teilbaum mit Tiefendurchlauf



Reihenfolge der Knoten in einem Tiefendurchlauf

Tiefendurchlauf (depth-first traversal)

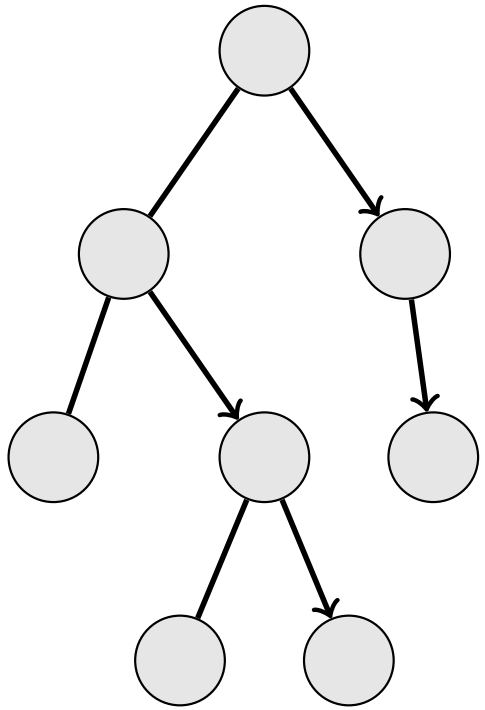
1. Besuche die Wurzel
2. Besuche linken Teilbaum mit Tiefendurchlauf
3. Besuche rechten Teilbaum mit Tiefendurchlauf



Reihenfolge der Knoten in einem Tiefendurchlauf

Breitendurchlauf (breadth-first traversal)

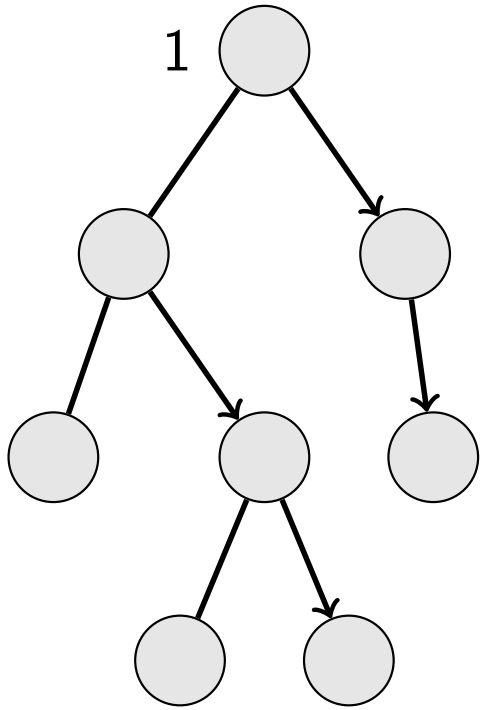
für i = 0, . . . , Tiefe des Baums:
    Besuche alle Knoten in Tiefe i
    von links nach rechts
    (d.h. in der Breite)



Reihenfolge der Knoten
in einem Breitendurchlauf

Breitendurchlauf (breadth-first traversal)

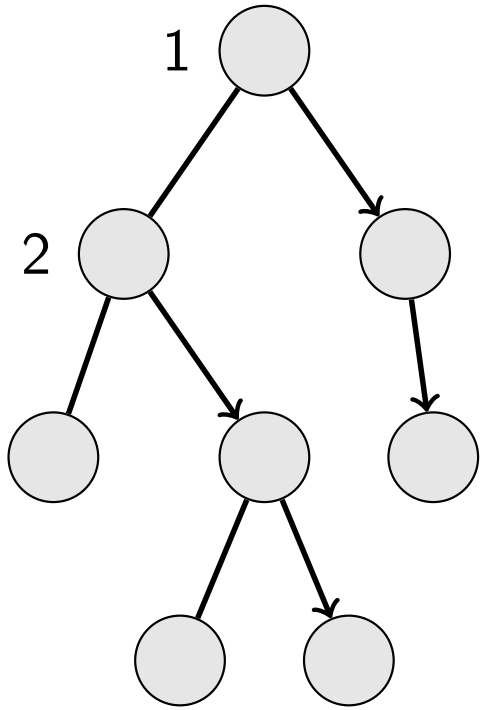
für i = 0, . . . , Tiefe des Baums:
    Besuche alle Knoten in Tiefe i
    von links nach rechts
    (d.h. in der Breite)



Reihenfolge der Knoten
in einem Breitendurchlauf

Breitendurchlauf (breadth-first traversal)

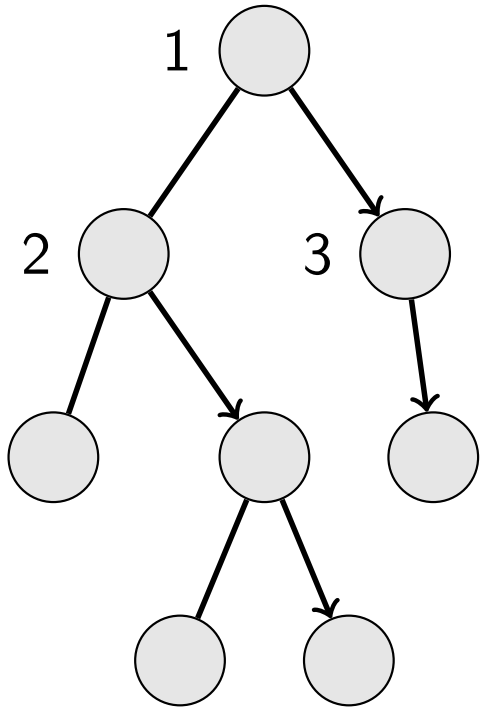
für i = 0, . . . , Tiefe des Baums:
    Besuche alle Knoten in Tiefe i
    von links nach rechts
    (d.h. in der Breite)



Reihenfolge der Knoten
in einem Breitendurchlauf

Breitendurchlauf (breadth-first traversal)

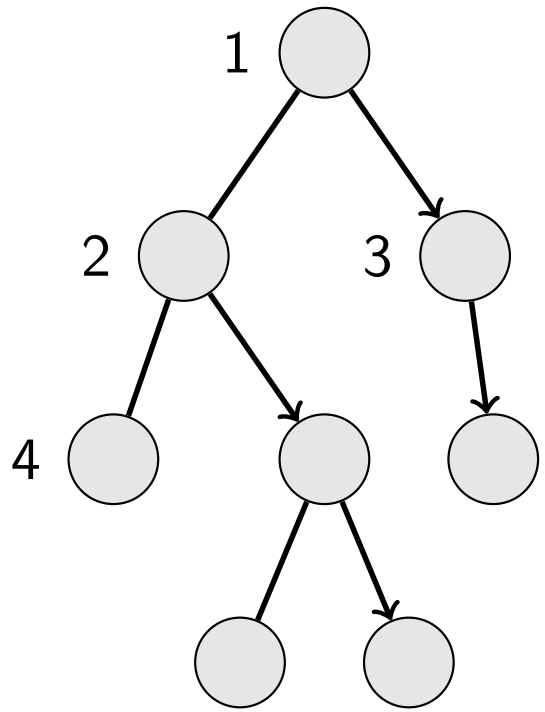
für i = 0, . . . , Tiefe des Baums:
    Besuche alle Knoten in Tiefe i
    von links nach rechts
    (d.h. in der Breite)



Reihenfolge der Knoten
in einem Breitendurchlauf

Breitendurchlauf (breadth-first traversal)

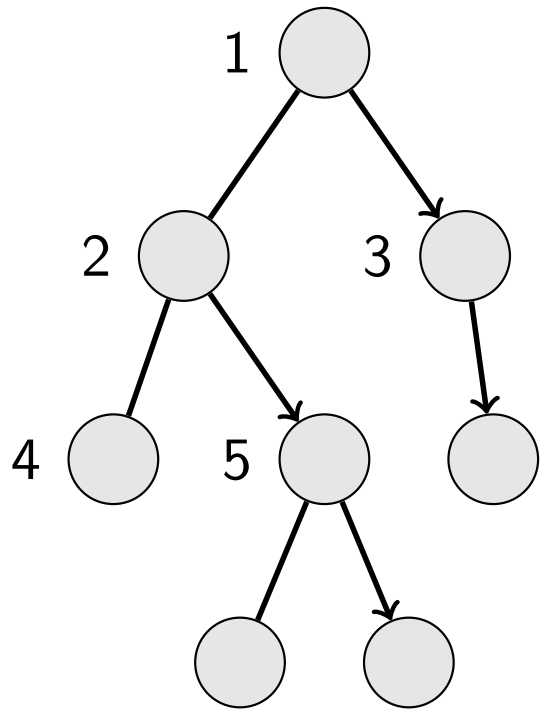
für i = 0, . . . , Tiefe des Baums:
    Besuche alle Knoten in Tiefe i
    von links nach rechts
    (d.h. in der Breite)



Reihenfolge der Knoten
in einem Breitendurchlauf

Breitendurchlauf (breadth-first traversal)

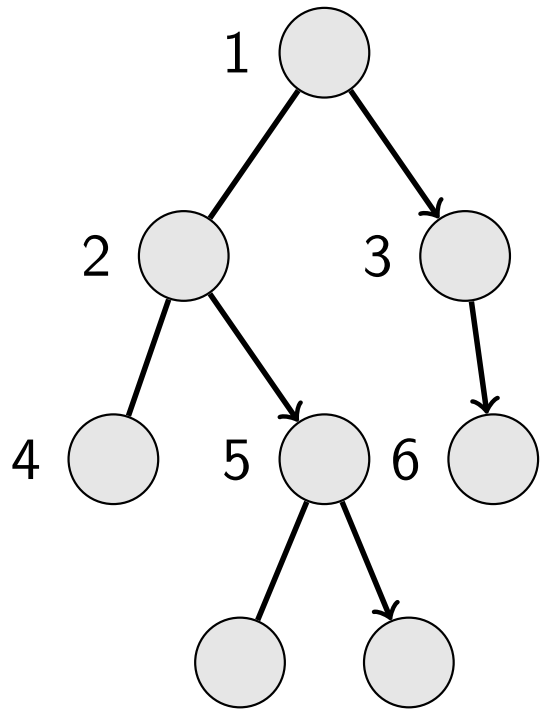
für i = 0, . . . , Tiefe des Baums:
    Besuche alle Knoten in Tiefe i
    von links nach rechts
    (d.h. in der Breite)



Reihenfolge der Knoten
in einem Breitendurchlauf

Breitendurchlauf (breadth-first traversal)

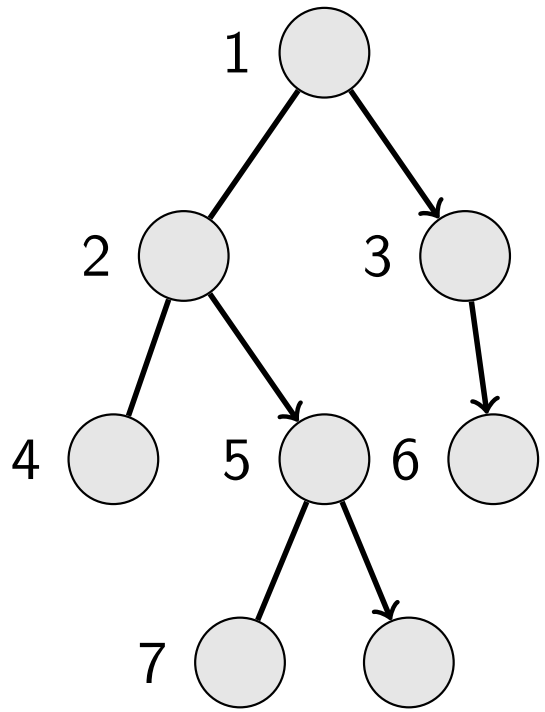
für i = 0, . . . , Tiefe des Baums:
    Besuche alle Knoten in Tiefe i
    von links nach rechts
    (d.h. in der Breite)



Reihenfolge der Knoten
in einem Breitendurchlauf

Breitendurchlauf (breadth-first traversal)

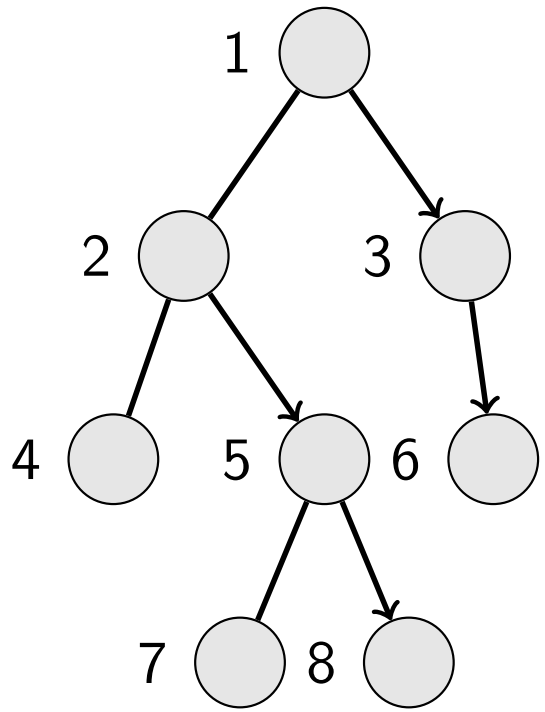
für i = 0, . . . , Tiefe des Baums:
    Besuche alle Knoten in Tiefe i
    von links nach rechts
    (d.h. in der Breite)



Reihenfolge der Knoten
in einem Breitendurchlauf

Breitendurchlauf (breadth-first traversal)

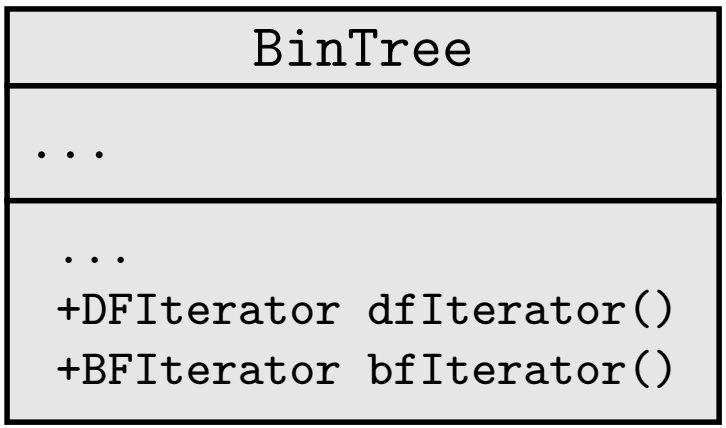
für i = 0, . . . , Tiefe des Baums:
    Besuche alle Knoten in Tiefe i
    von links nach rechts
    (d.h. in der Breite)



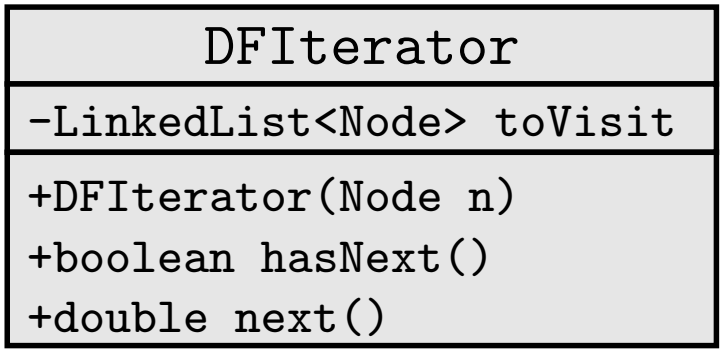
Reihenfolge der Knoten
in einem Breitendurchlauf

Baumdurchlauf mit Iteratoren

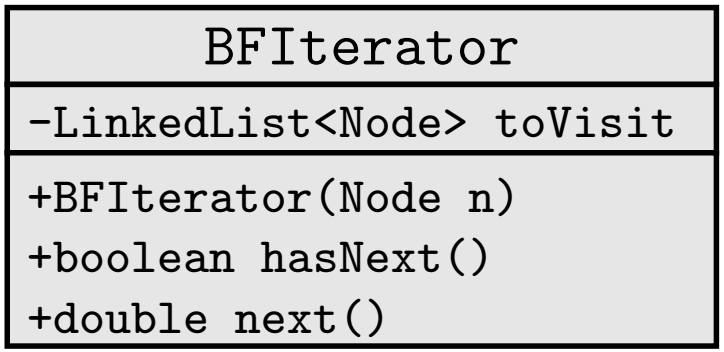
Iteratoren, die Tiefen- bzw. Breitendurchlauf implementieren:



Iterator für Tiefendurchlauf:



Iterator für Breitendurchlauf:



`next()` liefert den Wert des nächsten zu besuchenden Knotens (wenn es einen gibt, d.h. `hasNext()` liefert `true`) und geht dann zum darauffolgenden Knoten.

Erstellen der Iteratoren in BinTree

```
public DFIterator dfIterator() {
    return new DFIterator(this.root);
}
public BFIterator bfIterator() {
    return new BFIterator(this.root);
}
```

Implementierung des Tiefendurchlaufs (1)

```
import java.util.LinkedList;

public class DFIterator {
 private LinkedList<Node> toVisit;
 public DFIterator(Node n) {
  /* Linked list erstellen und Knoten n einfuegen */
   this.toVisit = new LinkedList<Node>();
      if (n!=null) {this.toVisit.addFirst(n);}
 }
 public boolean hasNext() {
  /* ist Liste nicht-leer ? */
      return !this.toVisit.isEmpty();
 }
 ...
```

Implementierung des Tiefendurchlaufs (2)

```
 ...
 public double next() {
  /* der aktuelle Knoten ist der erste i der Liste */
      Node n = this.toVisit.removeFirst();
  /* fuege Kinder in die Liste ein */
  /* Reihenfolge: left,right,alteListe */
      Node left= n.getLeft();
      Node right=n.getRight();
      if (right!=null) {this.toVisit.addFirst(right);}
      if (left!=null) {this.toVisit.addFirst(left);}
      return n.getValue();
 }
}
```

Beispiel Tiefendurchlauf

```
public static void main(String[] args) {
 BinTree k = new BinTree(8);
 BinTree j = new BinTree(7);
 BinTree e = new BinTree(6);
 BinTree f = new BinTree(j,5,k);
 BinTree g = new BinTree(4);
 BinTree b = new BinTree(e,2,f);
 BinTree empty = new BinTree();
 BinTree c = new BinTree(empty,3,g);
 BinTree a = new BinTree(b,1,c);
 DFIterator dfi = a.dfIterator();
 while (dfi.hasNext()) {
   System.out.print(dfi.next() + ",");
 }
 System.out.println();
```

Aufruf ergibt

```
1.0,2.0,6.0,5.0,7.0,8.0,3.0,4.0,
```

Implementierung des Breitendurchlaufs (1)

```
import java.util.LinkedList;

public class BFIterator {
 private LinkedList<Node> toVisit;
 public BFIterator(Node n) {
  /* Linked list erstellen und Knoten n einfuegen */
   this.toVisit = new LinkedList<Node>();
      if (n!=null) {this.toVisit.addFirst(n);}
 }
 public boolean hasNext() {
  /* ist Liste nicht-leer ? */
      return !this.toVisit.isEmpty();
 }
 ...
```

Implementierung des Breitendurchlaufs (2)

```
 ...
 public double next() {
  /* der aktuelle Knoten ist der erste i der Liste */
      Node n = this.toVisit.removeFirst();
  /* fuege Kinder in die Liste ein */
  /* Reihenfolge: alteListe,left,right */
      Node left= n.getLeft();
      Node right=n.getRight();
      if (left!=null) {this.toVisit.addLast(left);}
      if (right!=null) {this.toVisit.addLast(right);}
      return n.getValue();
 }
}
```

Beispiel Breitendurchlauf

```
public static void main(String[] args) {
 BinTree k = new BinTree(8);
 BinTree j = new BinTree(7);
 BinTree e = new BinTree(6);
 BinTree f = new BinTree(j,5,k);
 BinTree g = new BinTree(4);
 BinTree b = new BinTree(e,2,f);
 BinTree empty = new BinTree();
 BinTree c = new BinTree(empty,3,g);
 BinTree a = new BinTree(b,1,c);
 BFIterator bfi = a.bfIterator();
 while (bfi.hasNext()) {
   System.out.print(bfi.next() + ",");
 }
 System.out.println();
```

Aufruf ergibt

```
1.0,2.0,3.0,6.0,5.0,4.0,7.0,8.0,
```

Suche im Baum (1)

- Der Tiefen- und der Breitendurchlauf bieten auch die Möglichkeit nach einem Knotenwert zu suchen.
- Wir können der Klasse `BinTree` eine Methode `boolean member()` hinzufügen, die prüft, ob ein Wert im Baum enthalten ist:

```
  public boolean member(double val) {
   DFIterator dfi = this.dfIterator();
   boolean result = false;
   while (dfi.hasNext() && !result) {
    if (dfi.next() == val) {result = true;}
   }
   return result;
  }
}
```

- Analog könnten wir die Methode auch mit einem Breitendurchlauf implementieren.
- Man spricht von **Tiefensuche** und **Breitensuche**.
- Welche Variante ist besser?

Suche im Baum: Worst-Case-Komplexitäten

Annahme:

- Binärer Suchbaum hat Tiefe t und
- daher nicht mehr als 2^{t+1} Knoten

	Breitensuche		Tiefensuche	
	Platz	Laufzeit	Platz	Laufzeit
Suche erfolglos	$\mathcal{O}(2^t)$	$\mathcal{O}(2^t)$	$\mathcal{O}(t+1)$	$\mathcal{O}(2^t)$
	Platz	Laufzeit	Platz	Laufzeit
Suche erfolgreich, Knoten in Tiefe d	$\mathcal{O}(2^d)$	$\mathcal{O}(2^d)$	$\mathcal{O}(t+1)$	$\mathcal{O}(2^t)$

Breitensuche ist aufgrund des hohen Platzverbrauchs für große Bäume in der Praxis nicht empfehlenswert.

Generische Binärbaume und Methoden

- Genau wie bei Listen, können wir Bäume mit beliebigem Inhaltstyp (statt `double`-Werten) mit **generischen Klassen definieren**
- Statt der Klassen `BinTree` und `Node` definieren wir die generischen Klassen `BinTree<T>` und `Node<T>`, die über dem Typ der Knotenmarkierungen parametrisiert sind.

Generische Klasse Node<T>

```
public class Node<T> {
 private Node<T> left;
 private T value;
 private Node<T> right;

 public Node(Node<T> left, T value, Node<T> right){
  this.left = left;
  this.value = value;
  this.right = right;
 }
 public T getValue() {return this.value;}
 public Node<T> getLeft() {return this.left;}
 public Node<T> getRight() {return this.right;}
 public int size() { ... // wie vorher }
 public int depth() { ... // wie vorher }
}
```

Generische Klasse BinTree<T>

```
public class BinTree<T> {
 private Node<T> root;
 public BinTree() { this.root = null; }
 public BinTree(T v){ this.root = new Node<T>(null,v,null); }
 public BinTree(BinTree<T> left, T v, BinTree<T> right){
  this.root = new Node<T>(left.root,v,right.root); }
 public boolean isEmpty() { return this.root == null; }
 public T getRootValue() {
  if (this.root == null) {throw new NoSuchElementException(
                              "Ein leerer Baum hat keine Wurzel!");}
  else {return this.root.getValue();} }

 public Node<T> getRoot() {
  if (this.root == null) {throw new NoSuchElementException(
                               "Ein leerer Baum hat keine Wurzel!");}
  else {return this.root;} }
 public BinTree<T> getLeft() {
  if (this.root == null) {throw new NoSuchElementException(
                            "Ein leerer Baum hat keinen linken Teilbaum");}
  else { BinTree<T> result = new BinTree<T>();
         result.root = this.root.getLeft();
         return result;}   }
```

Generische Klasse BinTree<T> (2)

```
public BinTree<T> getRight() {
 if (this.root == null) {
 throw new NoSuchElementException("Ein leerer Baum hat keinen rechten
      Teilbaum");}
 else { BinTree<T> result = new BinTree<T>();
        result.root = this.root.getRight();
        return result;}    }
public int size() { ... // wie vorher }
public int depth () { ... // wie vorher }

public DFIterator<T> dfIterator() {
 return (new DFIterator<T>(this.root));
}
public BFIterator<T> bfIterator() {
 return (new BFIterator<T>(this.root));
 }
}
```

Generische Klassen für Iteratoren

```
import java.util.LinkedList;

public class BFIterator<T> {
 private LinkedList<Node<T>> toVisit;
 public BFIterator(Node<T> n) {
      this.toVisit = new LinkedList<Node<T>>();
      if (n!=null) {this.toVisit.addFirst(n);}
 }
 public boolean hasNext() {
      return !this.toVisit.isEmpty();
 }
 public T next() {
      Node<T> n = this.toVisit.removeFirst();
      Node<T> left= n.getLeft();
      Node<T> right=n.getRight();
      if (left!=null) {this.toVisit.addLast(left);}
      if (right!=null) {this.toVisit.addLast(right);}
      return n.getValue();
 }
}
```

Generische Klassen für Iteratoren (2)

```
import java.util.LinkedList;

public class DFIterator<T> {
 private LinkedList<Node<T>> toVisit;
 public DFIterator(Node<T> n) {
  /* Linked list erstellen und Knoten n einfuegen */
   this.toVisit = new LinkedList<Node<T>>();
      if (n!=null) {this.toVisit.addFirst(n);}
 }
 public boolean hasNext() {
  /* ist Liste nicht-leer ? */
      return !this.toVisit.isEmpty();
 }
 public T next() {
  /* der aktuelle Knoten ist der erste i der Liste */
      Node<T> n = this.toVisit.removeFirst();
  /* fuege Kinder in die Liste ein */
  /* Reihenfolge: left,right,alteListe */
      Node<T> left= n.getLeft();
      Node<T> right=n.getRight();
      if (right!=null) {this.toVisit.addFirst(right);}
      if (left!=null) {this.toVisit.addFirst(left);}
      return n.getValue();
 }
```

Anwendung: Huffman-Dekodierung

- Wir verwenden für den Huffman-Baum einen Baum vom Typ `BinTree<Character>`.
- Die Zeichen an inneren Knoten interessieren uns nicht (wir setzen sie auf `'\0'`.
- Wir programmieren lediglich das Dekodieren.
- Bitfolge ist repräsentiert als `LinkedList<Boolean>`
- Die Klasse `HuffmanDecode` ist Unterklasse von `BinTree<Character>`.

Hufmann-Decoder in Java

```
import java.util.*;
public class HuffmanDecoder extends BinTree<Character> {
 /* Konstruktoren */

 public HuffmanDecoder(char c) {
   /* Konstruktor von BinTree aufrufen */
   super(c);
 }

 public HuffmanDecoder(HuffmanDecoder left, char c, HuffmanDecoder right) {
    /* Konstruktor von BinTree aufrufen */
  super(left,c,right);
 }
 ...
```

Hufmann-Decoder in Java (2)

```
...
/* Dekodieren: Eingabe ist LinkedList von true und false Werten */
public String decode(LinkedList<Boolean> code) {
 Iterator<Boolean> iter = code.iterator();
 String output = "";
 while (iter.hasNext()) { // Noch Zeichen im Code
  Node<Character> currentNode = this.getRoot(); // fange mit Wurzel an
  while (!((currentNode.getLeft() == null)
          && (currentNode.getRight() == null))) {
         // noch nicht am Blatt angekommen:
         // lese false oder true aus dem iterator und setze currentNode
         // entsprechend auf linkes oder rechtes Kind
      if (iter.next()) {currentNode = currentNode.getRight(); }
      else {currentNode = currentNode.getLeft();}
  }
  // innere Schleife beendet, d.h. currentNode ist ein Blatt
  // hange Zeichen an den output an:
  output = output + currentNode.getValue();
 }
 return output;
}
}
```

Beispiel

```
import java.util.*;
public class HuffmanTest {
 public static void main(String[] args) {
  HuffmanDecoder dec =
   new HuffmanDecoder(
    new HuffmanDecoder(
     new HuffmanDecoder(
      new HuffmanDecoder(
       new HuffmanDecoder(
        new HuffmanDecoder('d'),'\0',new HuffmanDecoder('l'),'\0',
        new HuffmanDecoder('a')),'\0',
       new HuffmanDecoder('e')),'\0',
      new HuffmanDecoder(
       new HuffmanDecoder(
       new HuffmanDecoder(
         new HuffmanDecoder('g'),'\0',new HuffmanDecoder('i')),'0',
        new HuffmanDecoder(new HuffmanDecoder('r'),'\0',new
             HuffmanDecoder('b'))),'\0',
       new HuffmanDecoder(new HuffmanDecoder('_'),'\0',new
            HuffmanDecoder('n'))));
  ...
```

Beispiel

```
  ...
  String input = "000101100001110001111110011001111011101011101
                  000111100001110011010101101110111100101110111
                  001011000010001";
  LinkedList<Boolean> list = new LinkedList<Boolean>();
  for(char c : input.toCharArray()) {
    if (c == '0') {list.addLast(false);}
    else {list.addLast(true);}
  }
  System.out.println(dec.decode(list));
 }
}
```

Zusammenfassung

- Bäume sind eine wichtige Datenstruktur in der Informatik
- Binäre Bäume können in Java implementiert werden als Verallgemeinerung der einfach verketteten Listen mit bis zu zwei Nachfolgerverweisen
- Viele Operationen auf binären Bäumen werden rekursiv definiert
- In Java kann man rekursive definierte Funktion auf Bäumen implementieren
  - durch Weitergeben der Operation an die Knotenklasse oder
  - durch Fallunterscheidung bez. des leeren Baums und rekursiven Aufruf der Selektoren `getLeft()` und `getRight()` von BinTree.
- Wichtige Arten des Baumdurchlaufs sind Tiefendurchlauf und Breitendurchlauf.
- Entsprechend dazu: Tiefensuche und Breitensuche
- Auch Bäume können mit generischem Inhaltstyp programmiert werden.