Kapitel 14

Bäume

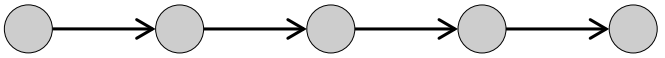
Ziele

- Den Begriff des Baums in der Informatik kennenlernen
- Bäume als verkettete Datenstruktur repräsentieren können
- Rekursive Funktionen auf Bäumen verstehen und schreiben können
- Verschiedene Möglichkeiten zum Durchlaufen von Bäumen kennenlernen („depth-first“, „breadth-first“)

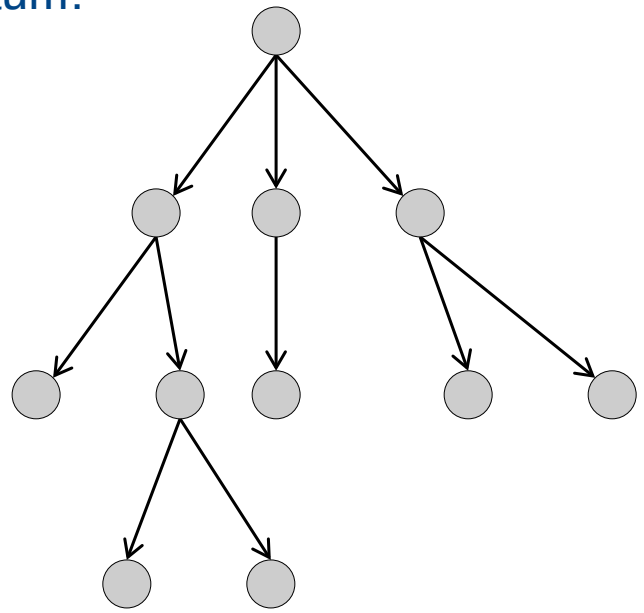
Bäume

- Bäume sind eine der am meisten verwendeten Datenstrukturen in der Informatik.
- Bäume verallgemeinern Listen:
  - In einer **Liste** hat jeder Knoten höchstens **einen Nachfolger**.
  - In einem **Baum** kann ein Knoten **mehrere Nachfolger** haben.

Liste:

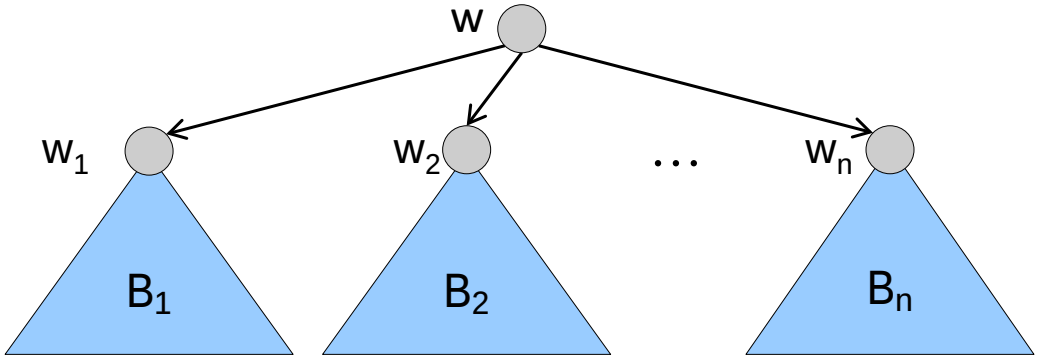


Baum:



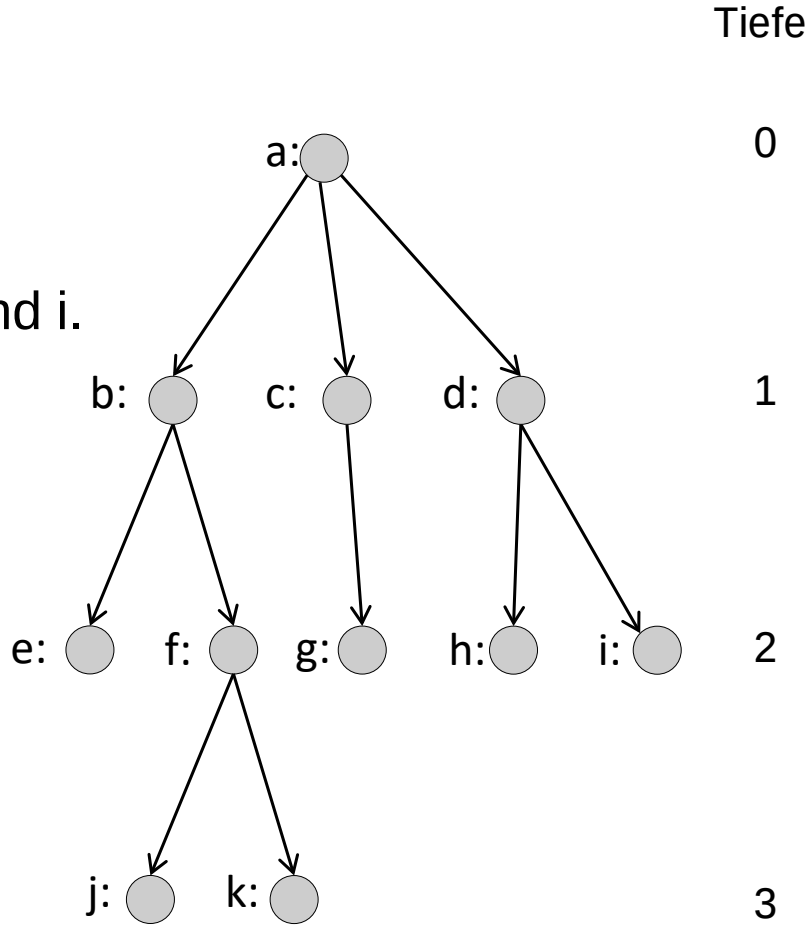
Bäume: Definition

- Ein Baum besteht aus **Knoten**, die durch **Kanten** miteinander verbunden sind. Jeder Baum hat einen **Wurzelknoten**, auf den keine Kante zeigt.
- In Knoten können je nach Anwendung verschiedene Daten gespeichert sein.
- Die Menge aller Bäume wird durch folgende Regeln konstruiert:
  - Es gibt einen leeren Baum.
  - Ein einzelner Knoten ohne irgendwelche Kanten ist ein Baum.
  - Ist $n > 0$ und sind $B_1, B_2, \ldots, B_n$ Bäume mit Wurzelknoten $w_1, w_2, \ldots, w_n$, so kann man diese zu einem größeren Baum zusammensetzen, indem man einen neuen Knoten w hinzufügt und diesen mit $w_1, w_2, \ldots, w_n$ verbindet. Der neue Knoten ist dann Wurzelknoten des so aufgebauten Baums.



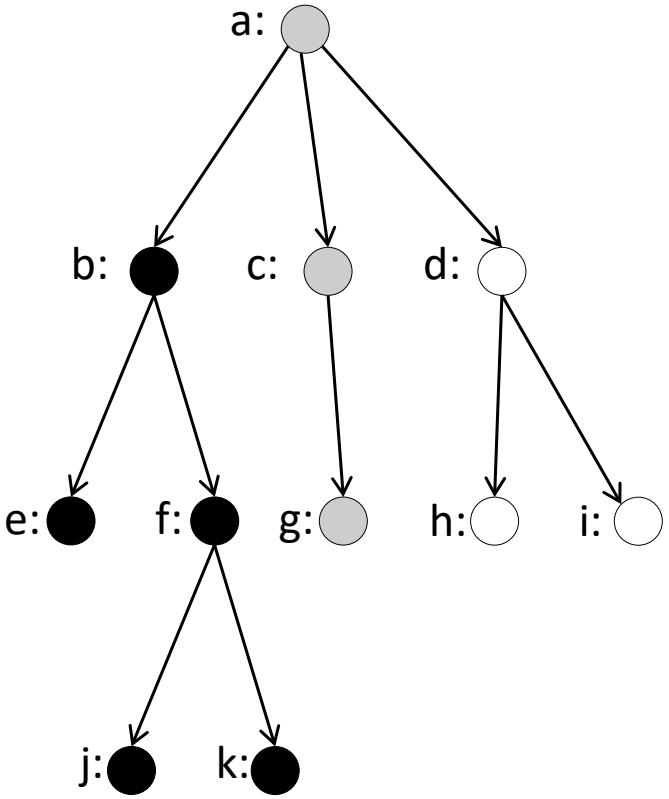
Bäume: Terminologie (1)

- a ist der **Wurzelknoten** des Baums.
- h und i sind die **Nachfolger-** oder auch **Kindknoten** des Knotens d.
- d ist **Vorgänger**- oder **Elternknoten** von h und i.
- Knoten ohne Nachfolger (hier: e, j, k, g, h, i) heißen **Blattknoten** (oder kurz **Blätter**).
- Die **Tiefe eines Knotens** im Baum ist die Anzahl der Schritte, die benötigt werden, um den Knoten von der Wurzel zu erreichen.
- Die **Tiefe des Baums** ist das Maximum der Tiefen aller Knoten des Baums. (hier: 3)
- Ein Baum ist ein **Binärbaum** wenn jeder Knoten darin höchstens zwei Nachfolger hat.



Bäume: Terminologie (2)

- Jeder Knoten in einem Baum ist Wurzel eines **Teilbaums** des gegebenen Baums.
- **Beispiele**: Der Teilbaum mit Wurzel b ist schwarz markiert und der mit Wurzel d ist weiß markiert. Der Teilbaum mit Wurzel a ist der ganze Baum selbst.
- Die Teilbäume mit Wurzeln b, c und d sind **Binärbäume**.
- Der gesamte Baum ist kein Binärbaum, da der Knoten a drei Nachfolger hat.



Bäume: Anwendungen (1)

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

- Arrays:
  - Zugriff auf Elemente: O(1)
  - Einfügen eines Elements: O(n)
- Listen:
  - Zugriff auf Elemente: O(n)
  - Einfügen eines Elements: O(1)
- Geordnete Binärbäume:
  - Zugriff auf Elemente: O(log n)
  - Einfügen eines Elements: O(log n)

Bäume: Anwendungen (2)

- **Datenbanken:**
  - Bäume werden zur indizierten Speicherung großer Datenmengen benutzt.
- **Betriebssysteme:**
  - Dateisysteme sind in Baumstrukturen organisiert und gespeichert.

- Baumartige Datenstrukturen sind in verschiedensten 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.

Zeichen	ASCII-Code
a	1100001
e	1100101
g	1100111
l	1101100

- Der 34 Zeichen lange Text „lege an eine brandnarbe nie naegel“ hat mit ASCII-Kodierung eine Länge von 34*7 Bit = 238 Bit:
- 1101100 1100101 1100111 1100101…

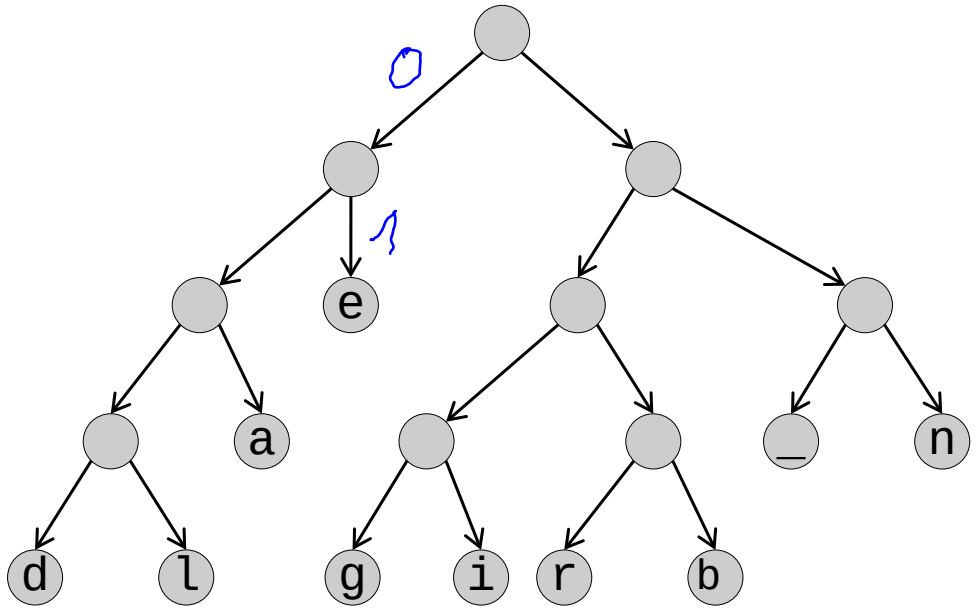
  l e g e

- 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. Diese Kodierung wird z.B. im ZIP-Format benutzt.

- Anhand der Häufigkeiten der einzelnen Buchstaben wird ein **Huffman-Baum** aufgestellt. (Wir erklären hier nicht, wie der Baum errechnet wird.)
- 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 zu linkem Kindknoten,
  1 = gehe zu rechtem Kindknoten)
- Beispiele:
  - e = 01 (erst links, dann rechts)
  - b = 1011
- Der Baum wird so gewählt, dass häufig vorkommende Zeichen einen kurzen Kode haben.



Beispielanwendung: Huffman-Kodierung (3)

Der Text

lege an eine brandnarbe nie naegel

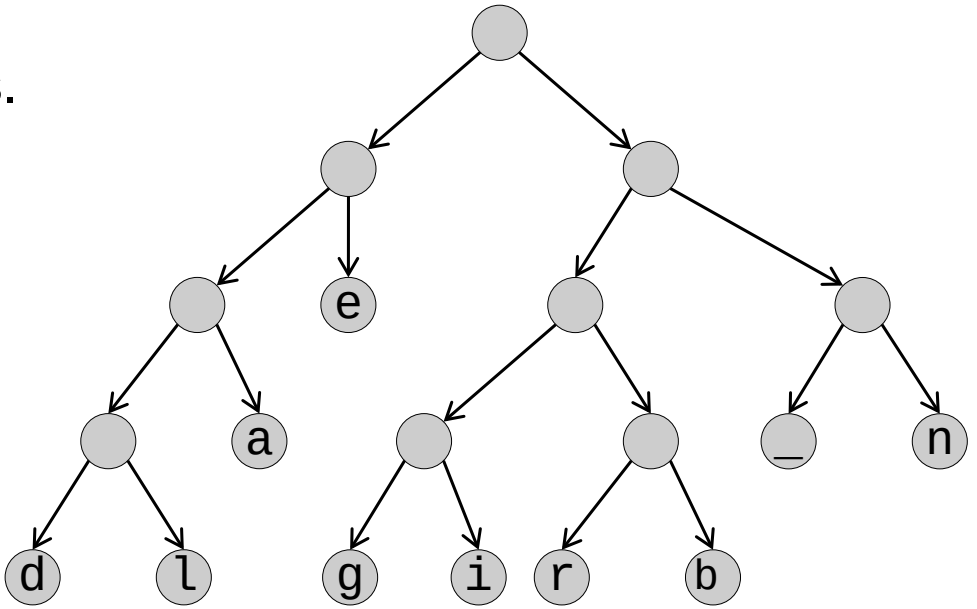
wird kodiert als:

0001011000011100011111100110011110111010111010000111 0

l e g e

000111001101010110111011110010111011100101100001000 1

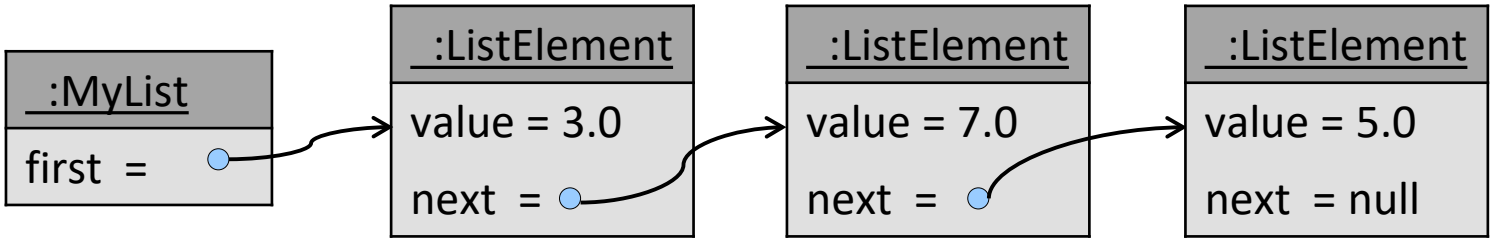
Der Text benötigt 105 Bits statt 238 Bits.



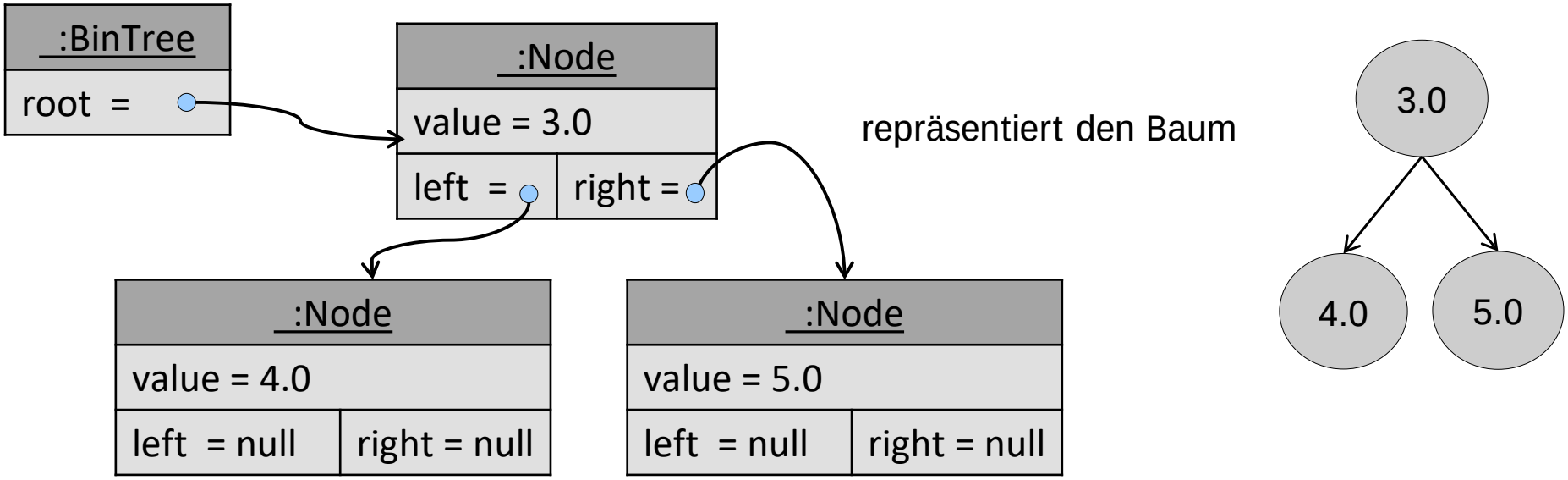
Binärbäume in Java

Bäume können in Java als verkettete Datenstruktur ganz ähnlich wie verkettete Listen implementiert werden.

- Zur Erinnerung: Speicherung einer Liste durch eine Kette von Objekten:



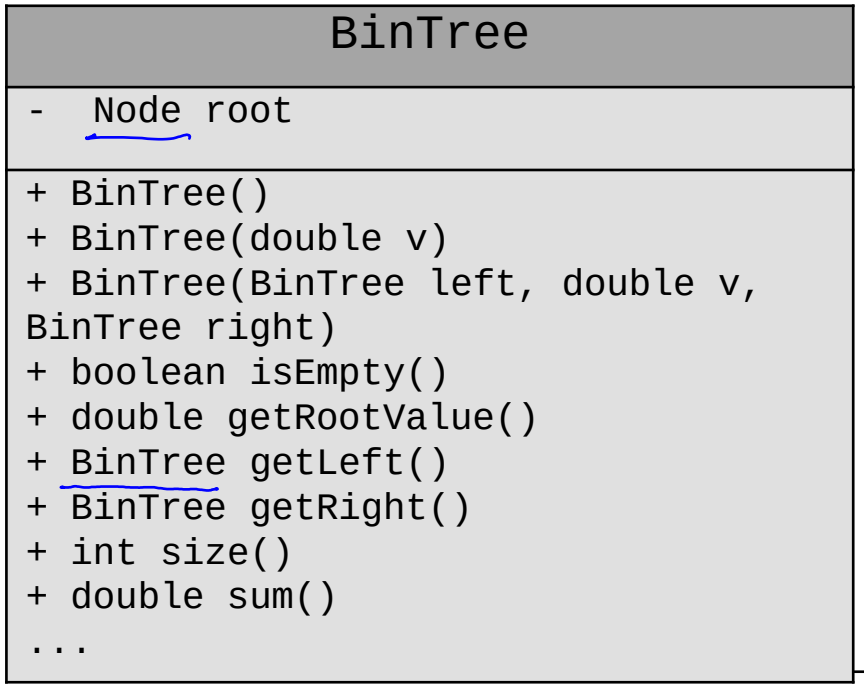
- Anstelle *eines* Nachfolgezeigers `next` haben die Elemente bei Binärbäumen nun *zwei* Nachfolgerzeiger `left` und `right`.

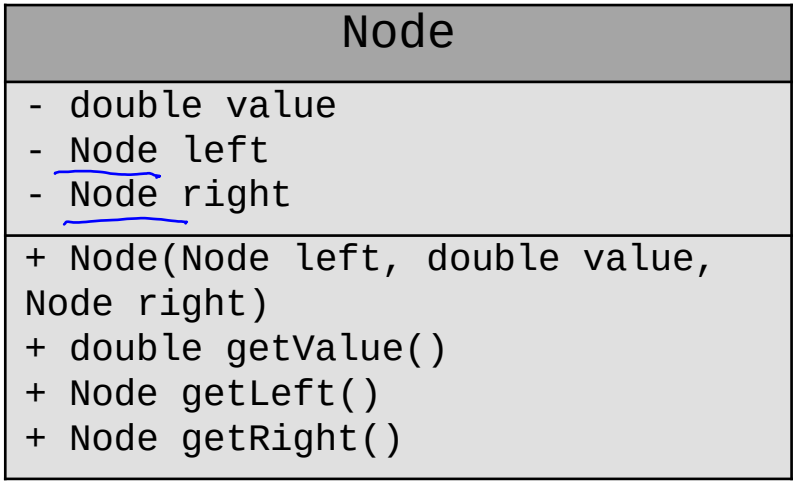


Klassen für Binärbäume

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

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



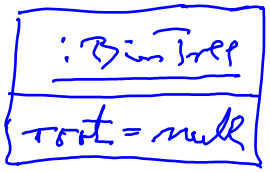


Implementierung der Klasse Node

```
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;
    }
}
```

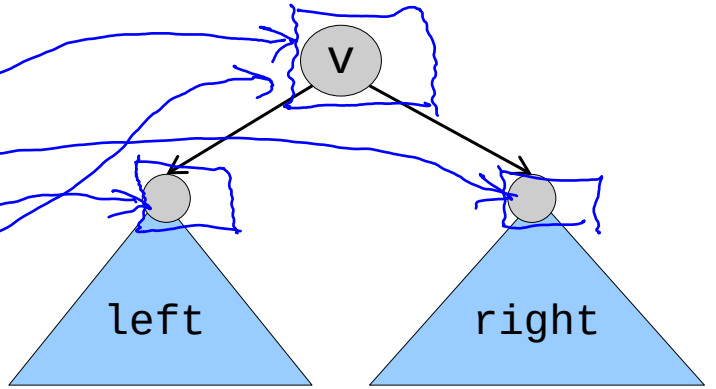
Die Klasse `BinTree` in Java: Konstruktoren

```
public class BinTree {
  private Node root;

  /** Erzeugt den leeren Baum */
  public BinTree() {
    this.root = null;
  }

  /** Erzeugt einen Baum mit einem einzigen Knoten */
  public BinTree(double v) {
    this.root = new Node(null, v, null);
  }

  /** Konstruiert einen Baum, aus zwei gegebenen
   * Teilbäumen und einer neuen Wurzel mit Wert v. */
  public BinTree(BinTree left, double v, BinTree right) {
    this.root = new Node(left.root, v, right.root);
  }
}
```



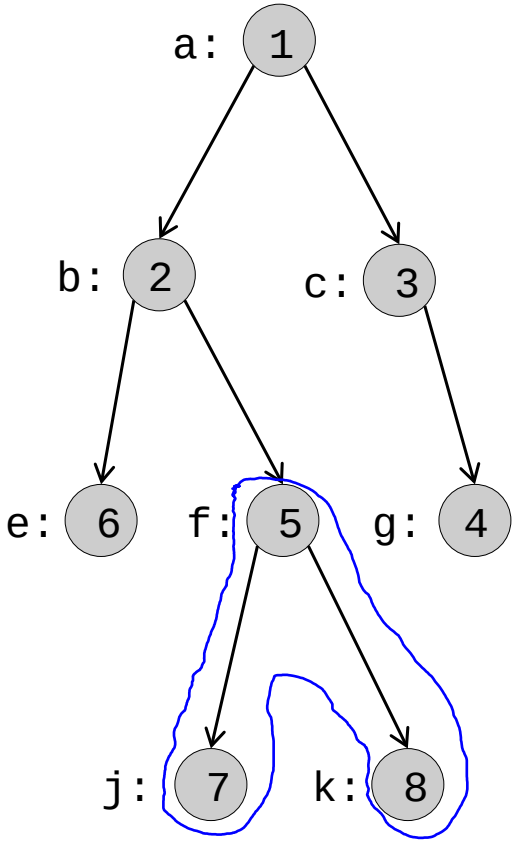
repräsentierter Baum:

leer

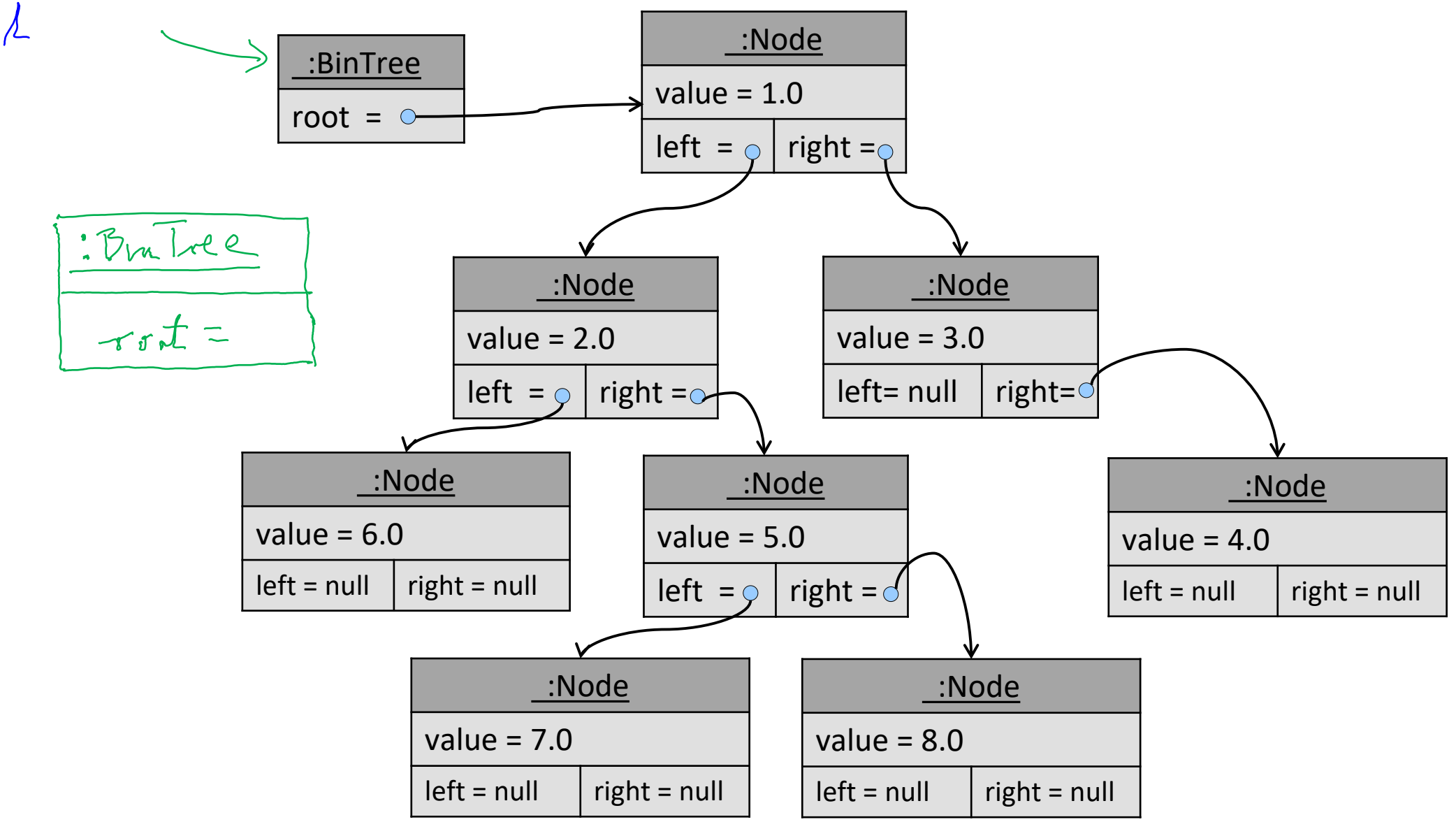


Beispiel für die Konstruktion eines Baums

```
public class BinTreeTest {

  public static void main(String[] args) {

    BinTree k = new BinTree(8);
    BinTree j = new BinTree(7);
    BinTree f = new BinTree(j, 5, k);
    BinTree e = new BinTree(6);
    BinTree b = new BinTree(e, 2, f);
    BinTree g = new BinTree(4);
    BinTree empty = new BinTree();
    BinTree c = new BinTree(empty, 3, g);
    BinTree a = new BinTree(b, 1, c);

    // Die Variablen repräsentieren jeweils
    // den Teilbaum mit dem gleichnamigen
    // Wurzelknoten.
  }
}
```



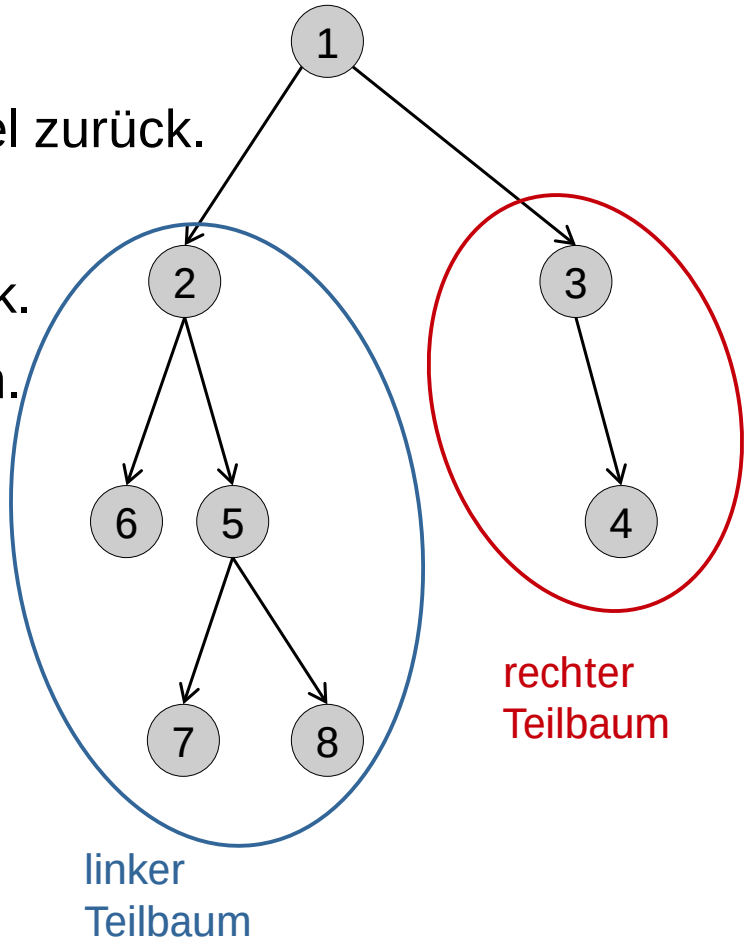
Der Baum im Heap



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 zurück.
  - `getRight()` gibt den rechten Teilbaum zurück.
- `size()` berechnet die Anzahl der Knoten im Baum.
- `sum()` berechnet die Summe der in den Knoten gespeicherten Zahlen.



Die Klasse `BinTree` in Java: Methoden

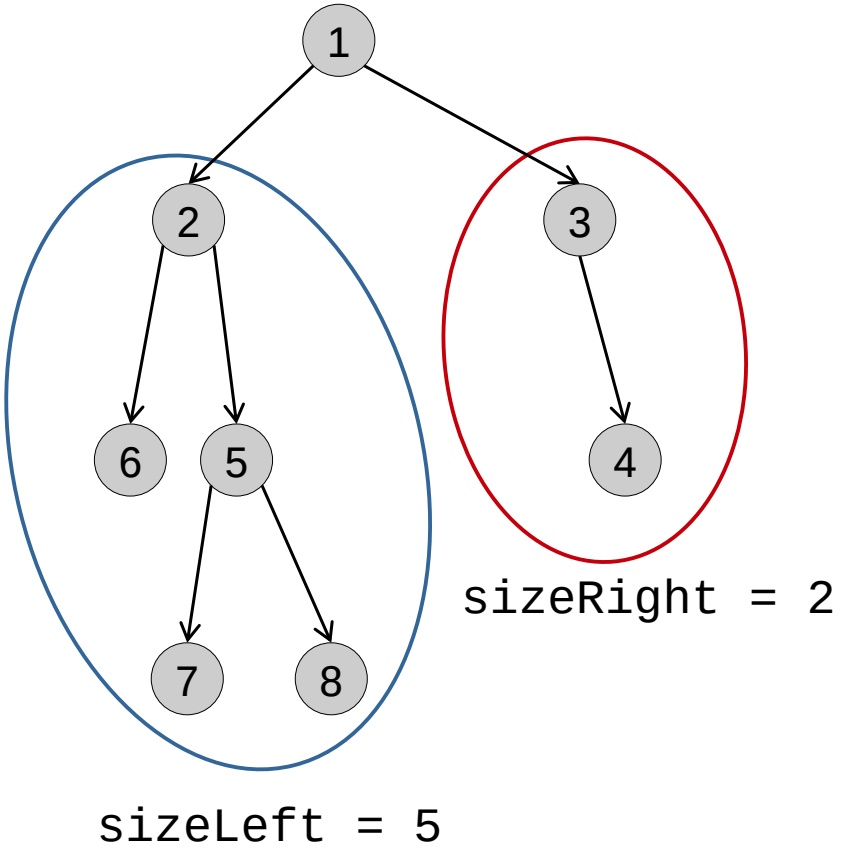
```
import java.util.NoSuchElementException;

public class BinTree {
    private Node root;
    ...

    public boolean isEmpty() {
      return this.root == null;
    }
    public double getRootValue() {
      if (this.root == null) {
         throw new NoSuchElementException(
                 "Ein leerer Baum hat keine Wurzel.");
      }
      return this.root.getValue();
    }
    public BinTree getLeft() {
      if (this.root == null) {
          throw new NoSuchElementException(
                 "Ein leerer Baum hat keinen linken Teilbaum.");
      }
      BinTree l = new BinTree();
      l.root = this.root.getLeft();
      return l;
    } //getRight analog }
```

this.isEmpty()

Anzahl der Knoten

Die Anzahl der Knoten in einem binären Baum kann **rekursiv** wie folgt berechnet werden:

- Berechne die Anzahl `sizeLeft` der Knoten im linken Teilbaum.
- Berechne die Anzahl `sizeRight` der Knoten im rechten Teilbaum.
- Gesamtanzahl der Knoten:
  `1 + sizeLeft + sizeRight`



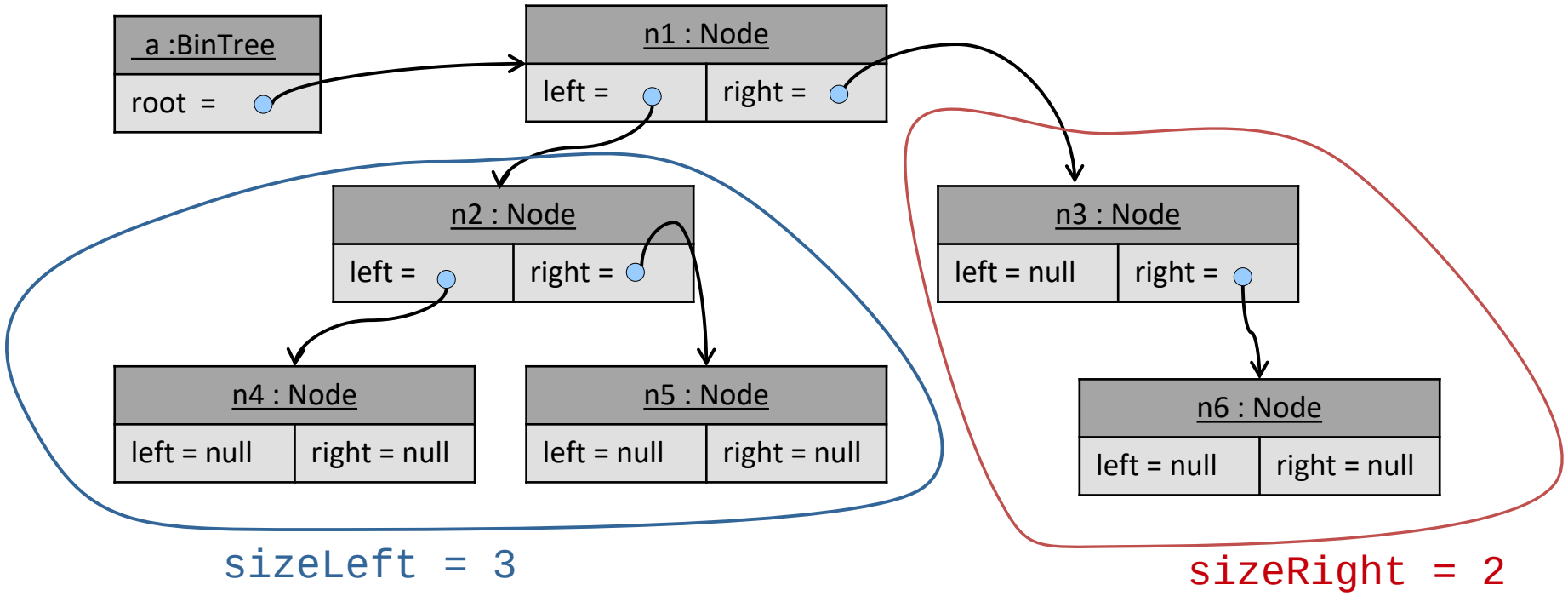
Anzahl der Knoten: Implementierung

```
public class BinTree {
   ...

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

Diese Implementierung ist elegant aber nicht effizient, da die Methodenaufrufe getLeft() und getRight() immer ein zusätzliches BinTree-Objekt erzeugen.

Effizientere Implementierung (1)

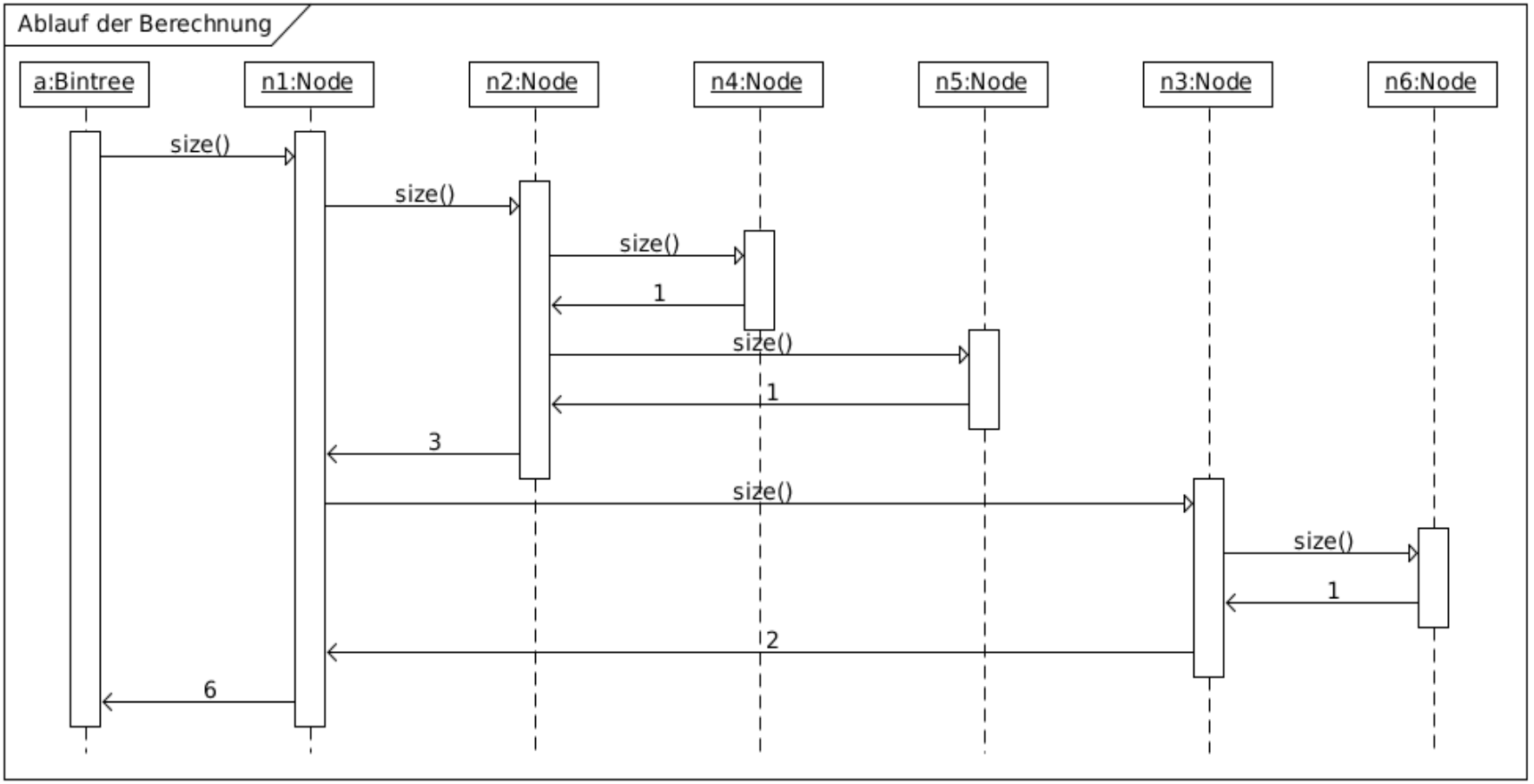
- Deligiere die Berechnung der Größe des Baums an die `Node`-Objekte, d.h. füge in der Klasse `Node` noch eine Methode `size()` hinzu.
- Ein `Node`-Objekt berechnet die Größe des von ihm repräsentierten Teilbaums, indem es rekursiv die `size()`-Methode auf seinem linken und rechten `Node`-Objekt aufruft und so die Werte `sizeLeft` und `sizeRight` berechnen lässt. Die Größe des Teilbaums ist dann `1+sizeLeft+sizeRight`.

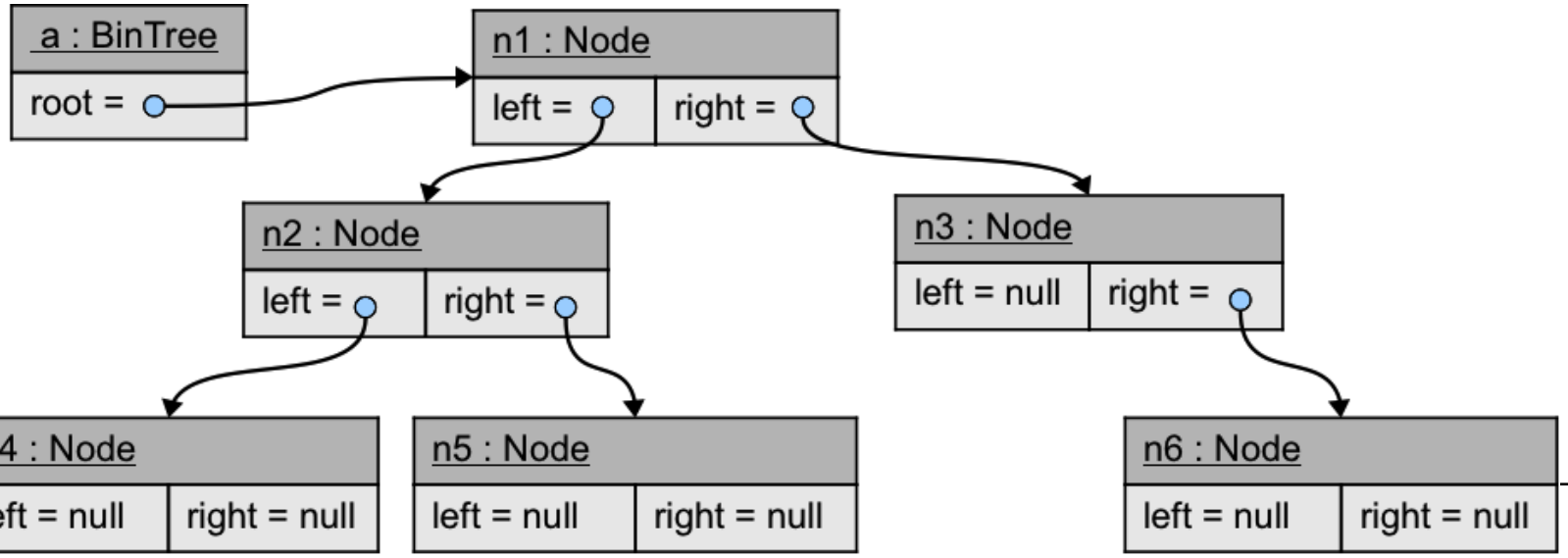


Effizientere Implementierung (2)

```
public class BinTree {

   ...

   public int size() {
     if (this.root == null)
        return 0;
     else
        return this.root.size();
   }
}
```

```
public class Node {

   ...

   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;
   }
}
```

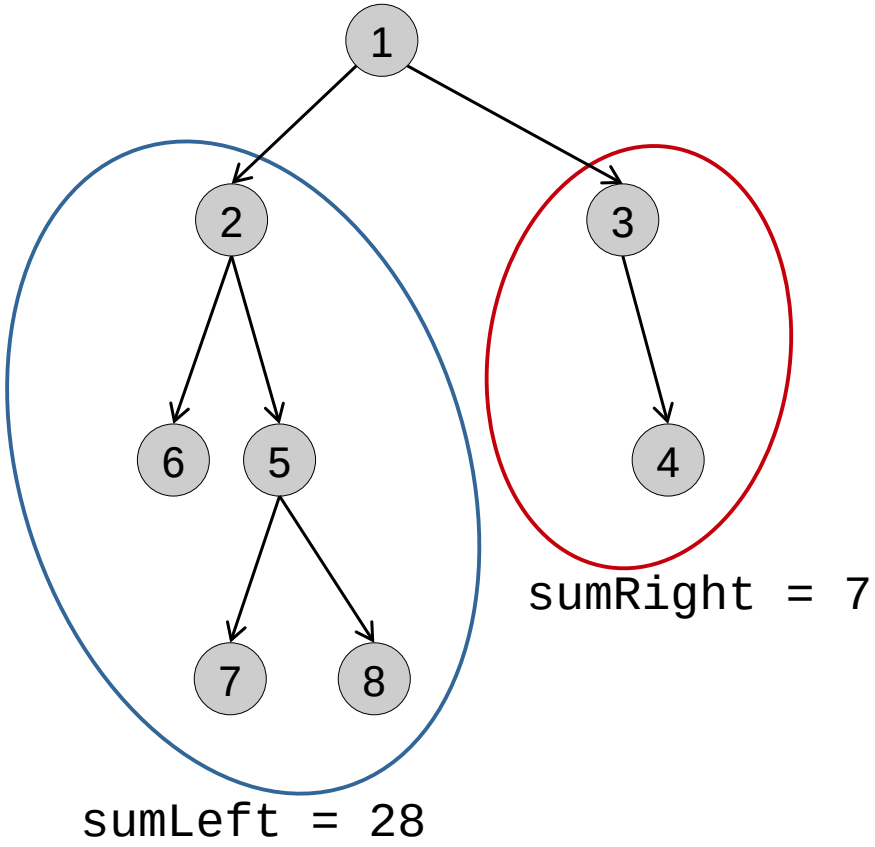

Ablauf der Berechnung
a:Bintree
n1:Node
n2:Node
n4:Node
n5:Node
n3:Node
n6:Node
size()
size()
size()
1
size()
1
3
size()
size()
1
2
6


a : BinTree
root =
n1 : Node
left =
right =
n2 : Node
left =
right =
n3 : Node
left = null
right =
n4 : Node
left = null
right = null
n5 : Node
left = null
right = null
n6 : Node
left = null
right = null

Summe der Werte aller Knoten

Die Summe der in einem Binärbaum gespeicherten Zahlen kann ebenfalls rekursiv berechnet werden:

- Berechne die Summe `sumLeft` im linken Teilbaum.
- Berechne die Summe `sumRight` im rechten Teilbaum.
- Gesamtsumme der Knoten: `value + sumLeft + sumRight`



Summe der Werte aller Knoten: Implementierung

```
public class BinTree {
   ...

   public double sum() {
      if (isEmpty()) return 0;
      else
            return this.getRootValue() + this.getLeft().sum() + this.getRight().sum();
   }
}
```

Effizientere Implementierung

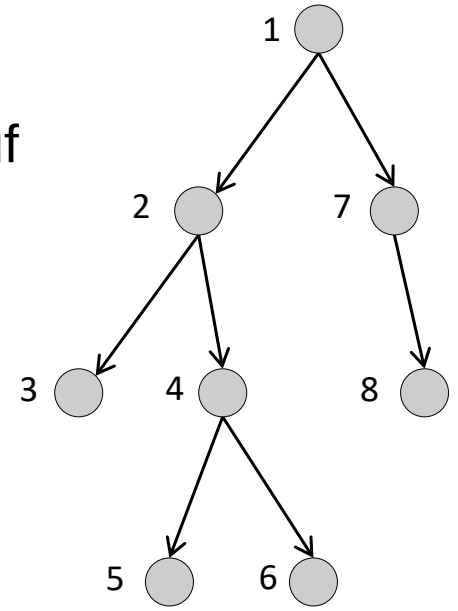
```
public class BinTree {

   ...

   public double sum() {
     if (this.root == null)
        return 0;
      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;
   }
}
```

Durchlaufen eines Baums (1)

- Wie bei Listen möchte man über alle Elemente des Baums iterieren können.
- Bei Listen gibt es *eine* natürliche Ordnung des Durchlaufs, von vorn nach hinten.
- Bei Bäumen gibt es mehrere sinnvolle Reihenfolgen des Durchlaufs, die je nach Anwendungsfall alle benutzt werden. Wichtige Beispiele:
  - **Tiefendurchlauf** („depth-first traversal“):
    - Besuche Wurzel
    - Besuche linken Teilbaum mit Tiefendurchlauf
    - Besuche rechten Teilbaum mit Tiefendurchlauf

Reihenfolge der Knoten
in einem Tiefendurchlauf:

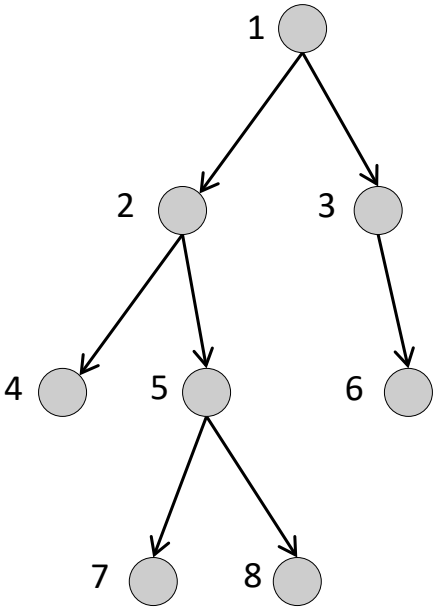


Durchlaufen eines Baums (2)

- **Breitendurchlauf** („breadth-first traversal“)

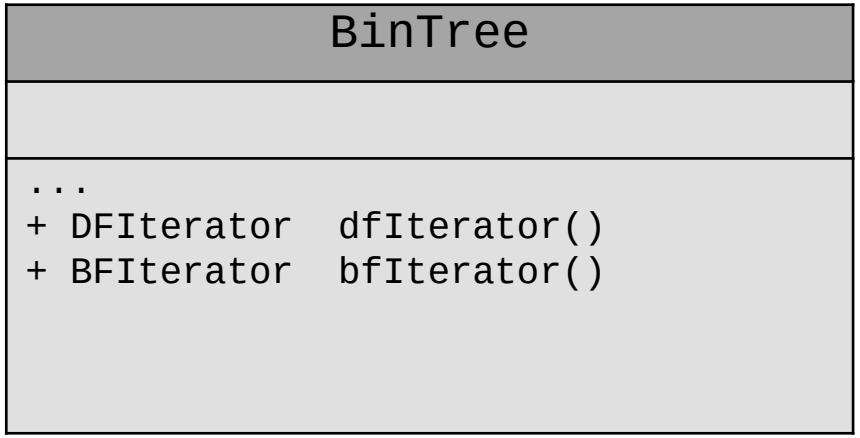
  Besuche die Knoten nach der Breite des Baumes geordnet.

Reihenfolge der Knoten
in einem Breitendurchlauf:



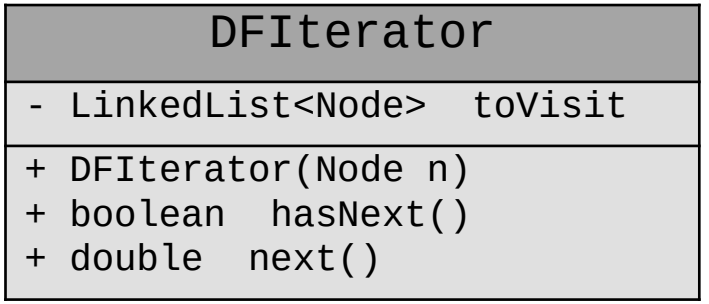
Baumdurchlauf mit Iteratoren

Implementierung zweier Iteratoren, welche die beiden Arten des Baumdurchlaufs realisieren.

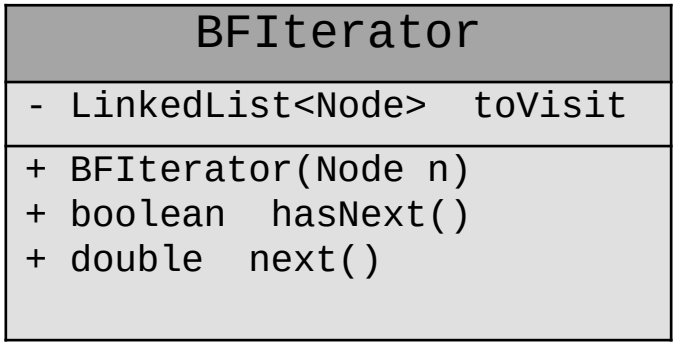



```
public DFIterator dfIterator() {
    return new DFIterator(this.root);
}
public BFIterator bfIterator() {
    return new BFIterator(this.root);
}
```

Iterator für Tiefendurchlauf:



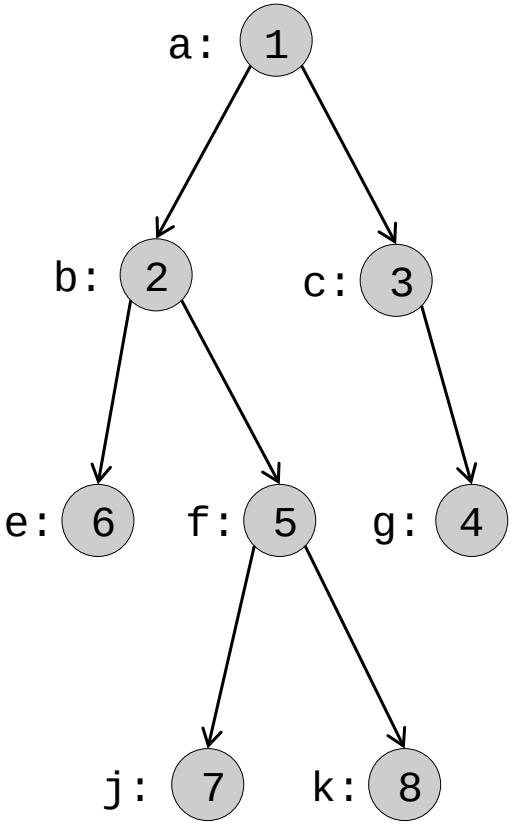
Iterator für Breitendurchlauf:



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

Beispiel für einen Tiefendurchlauf

```
public class BinTreeTest {

  public static void main(String[] args) {

    BinTree k = new BinTree(8);
    BinTree j = new BinTree(7);
    BinTree f = new BinTree(j, 5, k);
    BinTree e = new BinTree(6);
    BinTree b = new BinTree(e, 2, f);
    BinTree g = new BinTree(4);
    BinTree empty = new BinTree();
    BinTree c = new BinTree(empty, 3, g);
    BinTree a = new BinTree(b, 1, c);

    DFIterator it = a.dfIterator();
    while (it.hasNext()) {
      System.out.print(it.next() + "  ");
    }
  }
}
```



Ausgabe: `1.0  2.0  6.0  5.0  7.0  8.0  3.0  4.0`

Klasse `DFIterator` für Tiefendurchlauf (1)

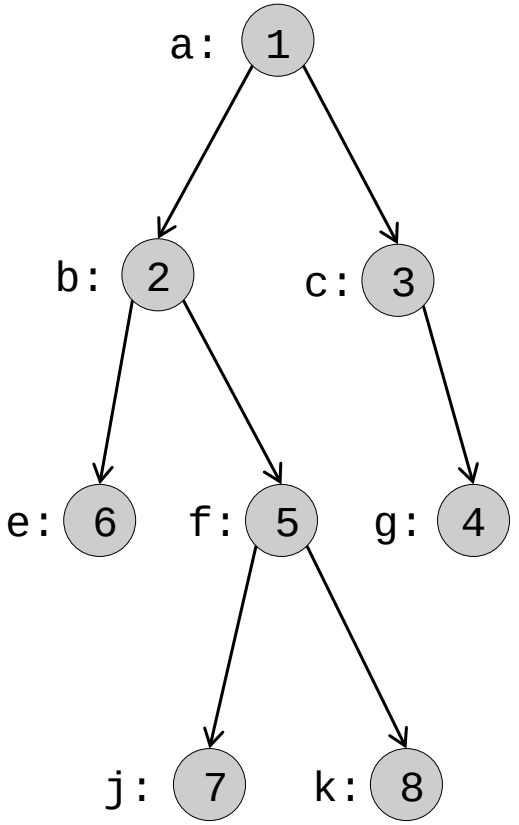
```
import java.util.LinkedList<E>;

public class DFIterator {
    // Liste der Wurzeln aller noch zu besuchender Teilbäume,
    // in der Reihenfolge, in der sie besucht werden sollen.
    private LinkedList<Node> toVisit;

    // Der Iterator durchläuft den Baum mit Wurzel
    // im Tiefendurchlauf.
    public DFIterator(Node n) {
       this.toVisit = new LinkedList<Node>();
       if (n != null)
           this.toVisit.addFirst(n);
    }

    public boolean hasNext() {
        return !this.toVisit.isEmpty();
    }
}
```

Klasse `DFIterator` für Tiefendurchlauf (2)

```
//Die Methode soll nur aufgerufen werden, wenn vorher mit hasNext() getestet wurde,
//dass es ein nächstes Element gibt.
public double next() {
   // Das erste Element der Liste toVisit ist die Wurzel des
   // nächsten zu besuchenden Baums. Beim Tiefendurchlauf
   // soll diese Wurzel als nächster Knoten besucht werden.
   Node n = this.toVisit.removeFirst();

   Node left = n.getLeft();
   Node right = n.getRight();

   // Als nächstes muss der Baum mit Wurzel left besucht werden
   // und danach der Baum mit Wurzel right.
   if (right != null) {
      this.toVisit.addFirst(right);
   }
   if (left != null) {
      this.toVisit.addFirst(left);
   }
   // Nun ist left ganz vorne in toVisit, gefolgt von right.
   return n.getValue(); // Rückgabe des Werts der aktuellen Wurzel
}
}
```

Beispiel für einen Breitendurchlauf

```
public class BinTreeTest {

  public static void main(String[] args) {

    BinTree k = new BinTree(8);
    BinTree j = new BinTree(7);
    BinTree f = new BinTree(j, 5, k);
    BinTree e = new BinTree(6);
    BinTree b = new BinTree(e, 2, f);
    BinTree g = new BinTree(4);
    BinTree empty = new BinTree();
    BinTree c = new BinTree(empty, 3, g);
    BinTree a = new BinTree(b, 1, c);

    BFIterator it = a.bfIterator();
    while (it.hasNext()) {
      System.out.print(it.next() + " ");
    }
  }
}
```



Ausgabe: `1.0  2.0  3.0  6.0  5.0  4.0  7.0  8.0`

Klasse `BFIterator` für Breitendurchlauf (1)

```
import java.util.LinkedList<E>;

public class BFIterator {
    // Liste der Wurzeln aller noch zu besuchender Teilbäume,
    // in der Reihenfolge, in der sie besucht werden sollen.
    private LinkedList<Node> toVisit;

    // Der Iterator besucht den Baum mit Wurzel n als
    // Breitendurchlauf.
   public BFIterator(Node n) {
      this.toVisit = new LinkedList<Node>();
       if (n != null) {
          this.toVisit.addFirst(n);
       }
    }

    public boolean hasNext() {
        return ! this.toVisit.isEmpty();
    }
}
```

Bemerkung: Dieser Teil ist identisch zum Iterator für Tiefensuche.

Klasse `BFIterator` für Breitendurchlauf (2)

```
public double next() {
    Node n = toVisit.removeFirst();
    Node left = n.getLeft();
    Node right = n.getRight();

    // Im Unterschied zum Tiefendurchlauf werden die beiden
    // Teilbäume von n hier hinten angefügt.
    if (left != null) {
        this.toVisit.addLast(left);
    }
    if (right != null) {
        this.toVisit.addLast(right);
    }
    return n.getValue();
}
```

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 rekursiv definierte Funktionen auf Bäumen implementieren
  - durch Weitergeben der Operation an die Knotenklasse oder
  - durch Fallunterscheidung bzgl. des leeren Baums und rekursiven Aufruf der Selektoren `getLeft()` und `getRight()` von `BinTree`.
- Wichtige Arten des Baumdurchlaufs sind Tiefendurchlauf und Breitendurchlauf.