

## Modul Datenstrukturen, Algorithmen und Programmierung 1

### Zusatzaufgaben Binärer Suchbaum

- Die Aufgaben stammen aus den Klausuren der vergangenen Jahre.
- Die Aufgaben sind unterschiedlich schwer und unterschiedlich umfangreich. Dieses wurde in den Klausuren durch die Zahl der zugeordneten Punkte berücksichtigt.
- Alle Aufgaben lassen sich mit den Kenntnissen der Vorlesungsinhalte bis einschließlich Kapitel 12 bearbeiten.
- Die Aufgaben dienen in verschiedener Art dem Training für die Klausur:
  - In Programmierung bereits geübte Studierende können sich an die Form der Klausuraufgaben gewöhnen.
  - Bei allen Studierenden wird das algorithmische Denken trainiert, das zum Lösen der Aufgaben zu fortgeschrittenen Themen benötigt wird.
- Zu diesen Aufgaben werden keine Beispiellösungen veröffentlicht.
- Die Zusammenstellung ist ausgehend von den vorliegenden Klausuren erfolgt. Möglicherweise sind einzelne Aufgaben – oder Varianten davon – bereits in die Übungs- oder Praktikumsaufgaben übernommen worden und tauchen jetzt doppelt auf.
- Die Aufgaben der hier präsentierten Form haben in den Klausuren etwa 15 bis 25 Prozent des Gesamtumfangs ausgemacht.
- In allen Aufgaben soll die aus der Vorlesung bekannte Klasse `BinarySearchTree<T extends Comparable<T>>` ergänzt werden, die im Anhang noch einmal aufgeführt ist.
- Bei der Implementierung der geforderten Methoden dürfen aber **nur** die **im Anhang** aufgeführten Methoden genutzt werden.
- Bei allen Aufgaben müssen Vergleiche von Inhalten mit der Methode `compareTo` vorgenommen werden – auch dann, wenn dieses nicht ausdrücklich in der Aufgabenstellung erwähnt wird. Das ergibt sich zwangsläufig aus der Implementierung der Klasse `BinarySearchTree`.

## Aufgabe 1

Vervollständigen Sie die Methode `int countNodes( int top, int bottom )`.

Die Methode `countNodes` soll die Anzahl der Knoten zurückgeben, die im Baum auf den Ebenen von einschließlich `top` bis einschließlich `bottom` liegen. Ist `top` größer als `bottom`, soll `0` zurückgegeben werden. **Nicht existierende Ebenen** sollen bei der Ausführung von `countNodes` einfach **übergangen werden**. Die Wurzel des Baums liegt auf der Ebene `0`.

```
public int countNodes( int top, int bottom )
{
    if ( top <= bottom && !isEmpty() )
    {
        
    }
    else
    {
        return 0;
    }
}
```

## Aufgabe 2

Vervollständigen Sie die Methode `int sortedUpTo( int n )`.

Die Methode `sortedUpTo` soll die kleinsten `n` Inhalte des Baums in aufsteigender Reihenfolge auf dem Bildschirm ausgeben. Ist `n` negativ oder 0, soll keine Ausgabe erfolgen. Hat der Baum keine `n` Inhalte, sollen alle Inhalte ausgegeben werden.

Hinweis: Nutzen Sie den Rückgabewert, um die Anzahl der noch auszugebenden Inhalte zurückzugeben.

```
public int sortedUpTo( int n )
```

 $\{$ 

```
if ( n > 0 && !isEmpty() )
```

{

}

**else**

{


}

}

### Aufgabe 3

Vervollständigen Sie die Methode `T largest0n( int level )`.

Die Methode `largest0n` soll den Inhalt des Knotens zurückgeben, der im Baum auf der Ebene `level` den größten Wert hat. Die Wurzel des Baums liegt auf der Ebene `0`. Der Vergleich soll mit der Methode `equals` vorgenommen werden. Existiert kein Knoten auf der Ebene `level`, soll `null` zurückgegeben werden.

```
public T largest0n( int level )
{
    if ( !isEmpty() && level >= 0 )
    {
        
    }
    else
    {
        return null;
    }
}
```

## Aufgabe 4

Vervollständigen Sie die Methode `int isNiceTree()`.

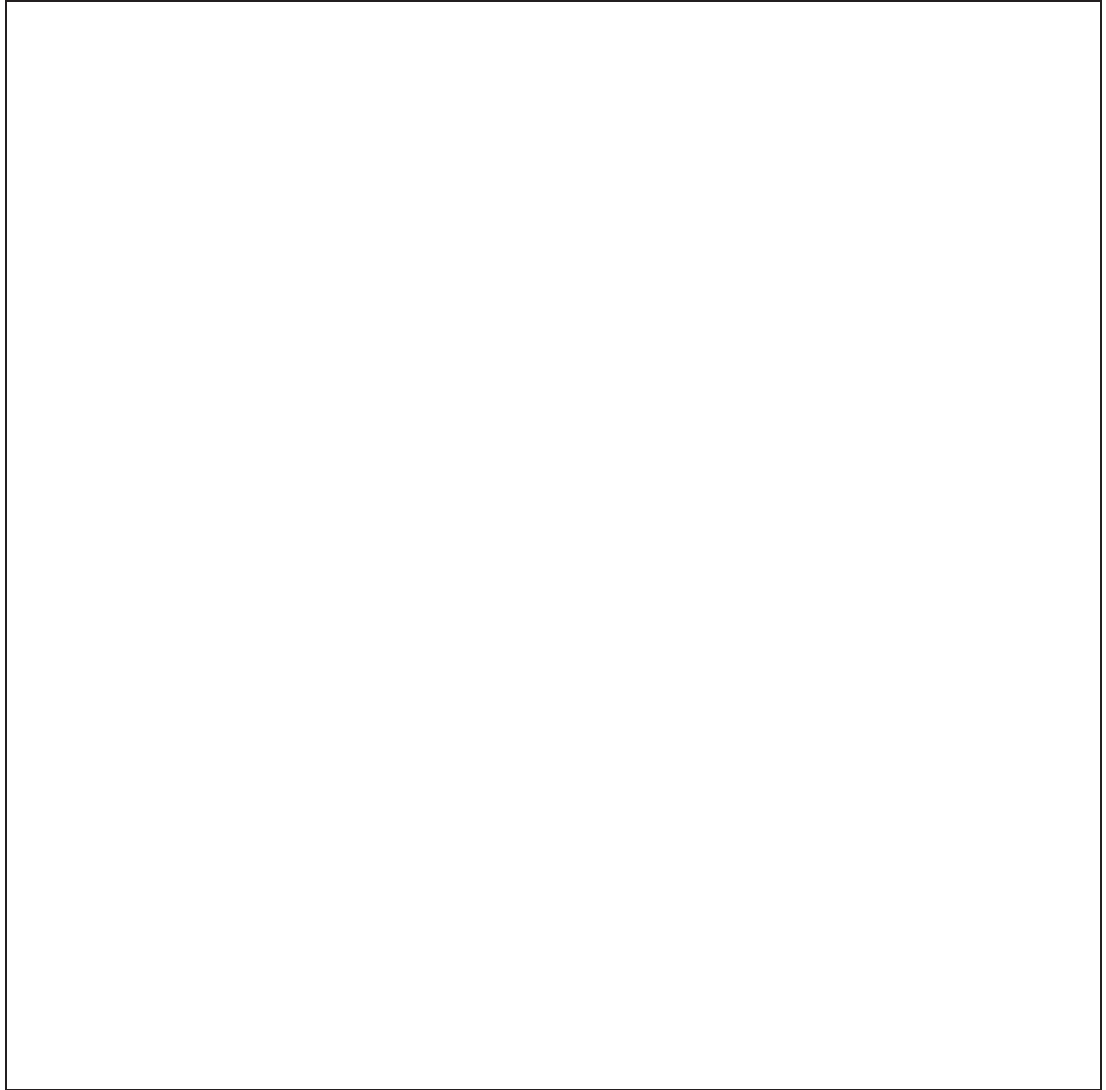
Die Methode `isNiceTree` soll die Anzahl der Blätter des Baums zurückgeben, falls sich für jeden Knoten die Anzahlen der Blätter im linken und im rechten Teilbaum um höchstens 1 unterscheiden. Sonst soll ein negativer Wert zurückgegeben werden.

```
public int isNiceTree()
```

```
{
```

```
    if ( isEmpty() )
```

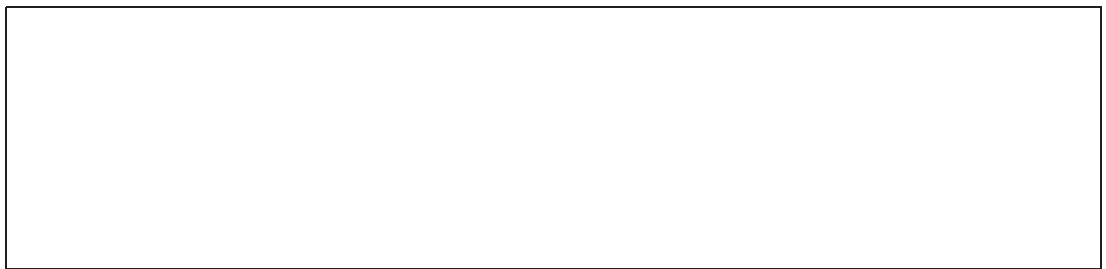
```
    {
```



```
    }
```

```
    else
```

```
    {
```




```
    }
```

```
}
```

## Aufgabe 5

Vervollständigen Sie die Methode `int countNodes( int level )`.

Die Methode `countNodes` soll die Anzahl der Knoten zurückgeben, die im Baum auf den Ebenen mit **ungerader** Position liegen. Die Wurzel des Baums liegt auf der Ebene `0`, die zu den geraden Ebenen zählt. Der Aufruf soll initial mit dem Argument `0` erfolgen als `countNodes( 0 )`.

```
public int countNodes( int level )
{
    if ( !isEmpty() )
    {
        
    }
    else
    {
        return 0;
    }
}
```

## Aufgabe 6

Vervollständigen Sie die Methode `boolean smallerExists( T obj )`.

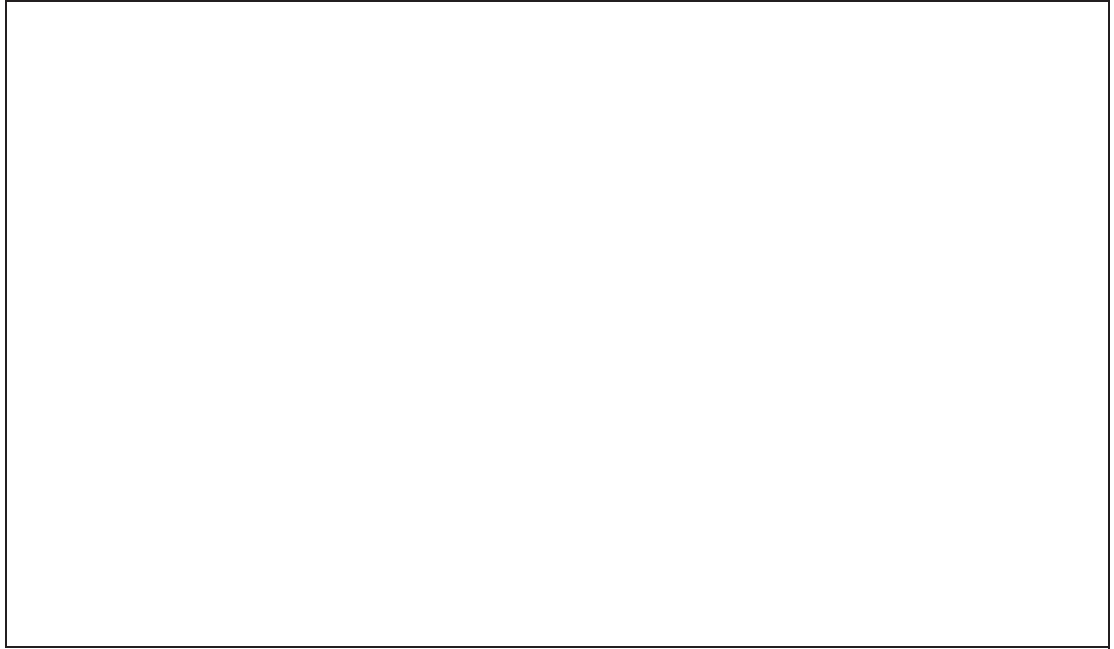
Die Methode `smallerExists` soll `true` zurückgeben, falls im Baum ein Inhalt existiert, der kleiner als `obj` ist. Sonst soll `false` zurückgegeben werden. Gehen Sie davon aus, dass `obj` immer ungleich `null` ist.

```
public boolean smallerExists( T obj )
```

```
{
```

```
    if ( !isEmpty() )
```

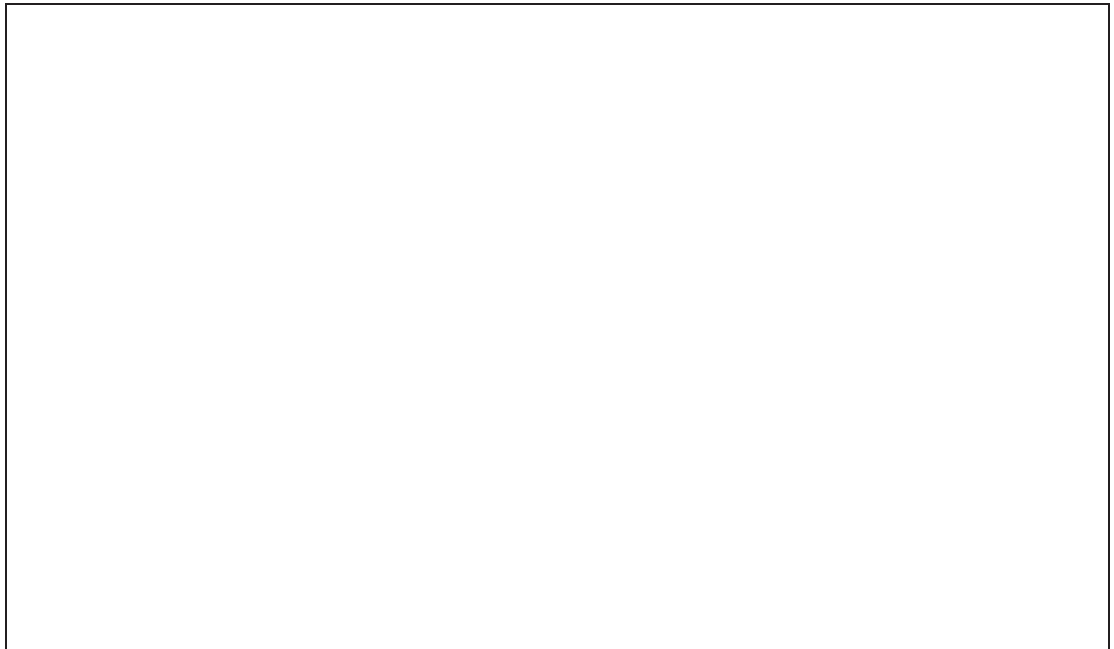
```
    {
```



```
    }
```

```
    else
```

```
    {
```



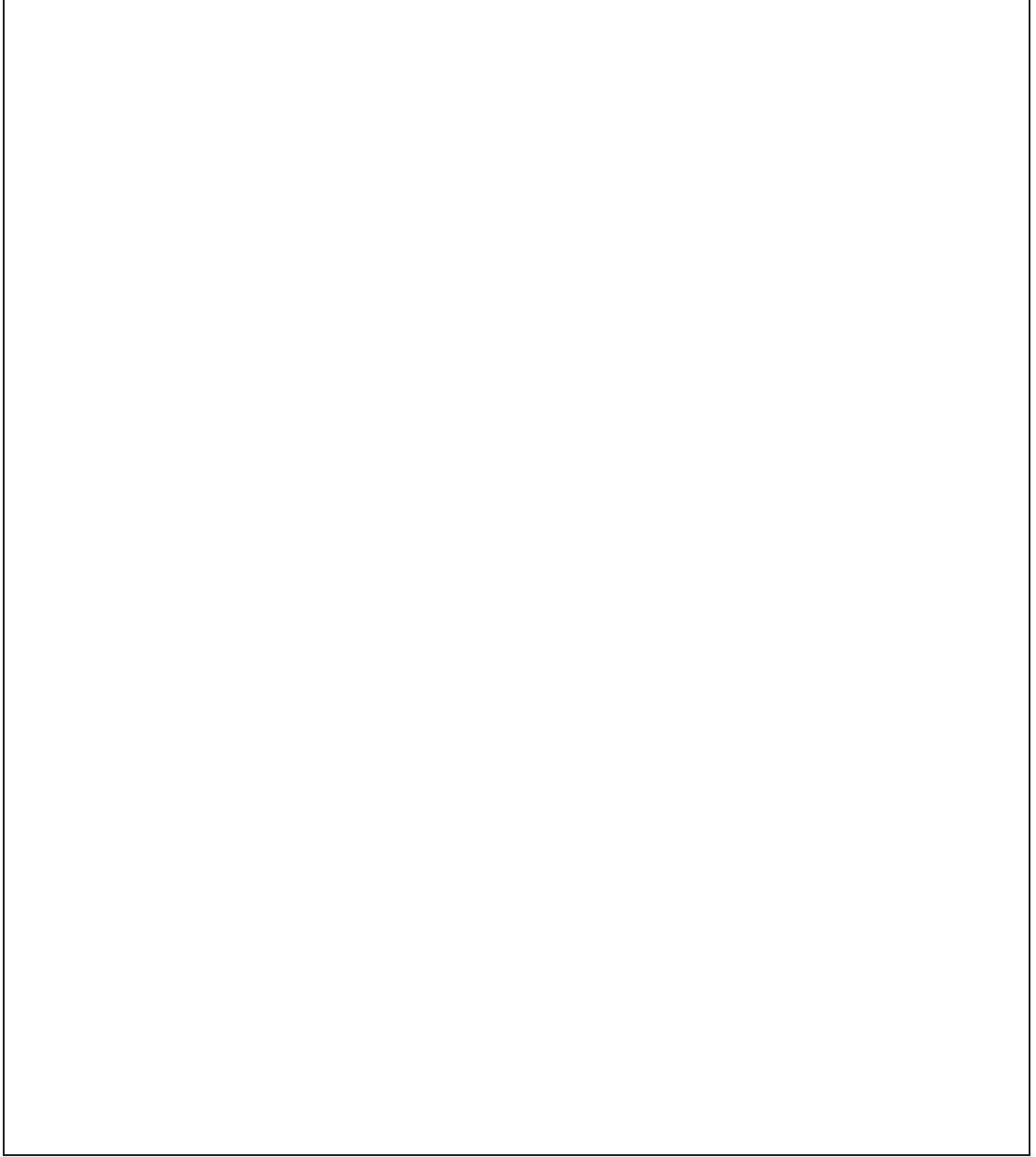
```
    }
```

```
}
```

## Aufgabe 7

Vervollständigen Sie die Methode `BinarySearchTree<T> subTree( T c )`.

Falls der Inhalt `c` im Baum existiert, soll die Methode `subTree` den Teilbaum entfernen, dessen Wurzel `c` als Inhalt enthält. Diese Wurzel soll mit ihrem Teilbaum als Ergebnis zurückgegeben werden. Kommt `c` nicht als Inhalt im Baum vor, soll `null` zurückgegeben werden.

```
public BinarySearchTree<T> subTree( T o )
{
    if ( !isEmpty() )
    {
        
    }
    else
    {
        return null;
    }
}
```



## Aufgabe 8

Vervollständigen Sie die Methode `boolean isDirectParent( T parent, T child )`.

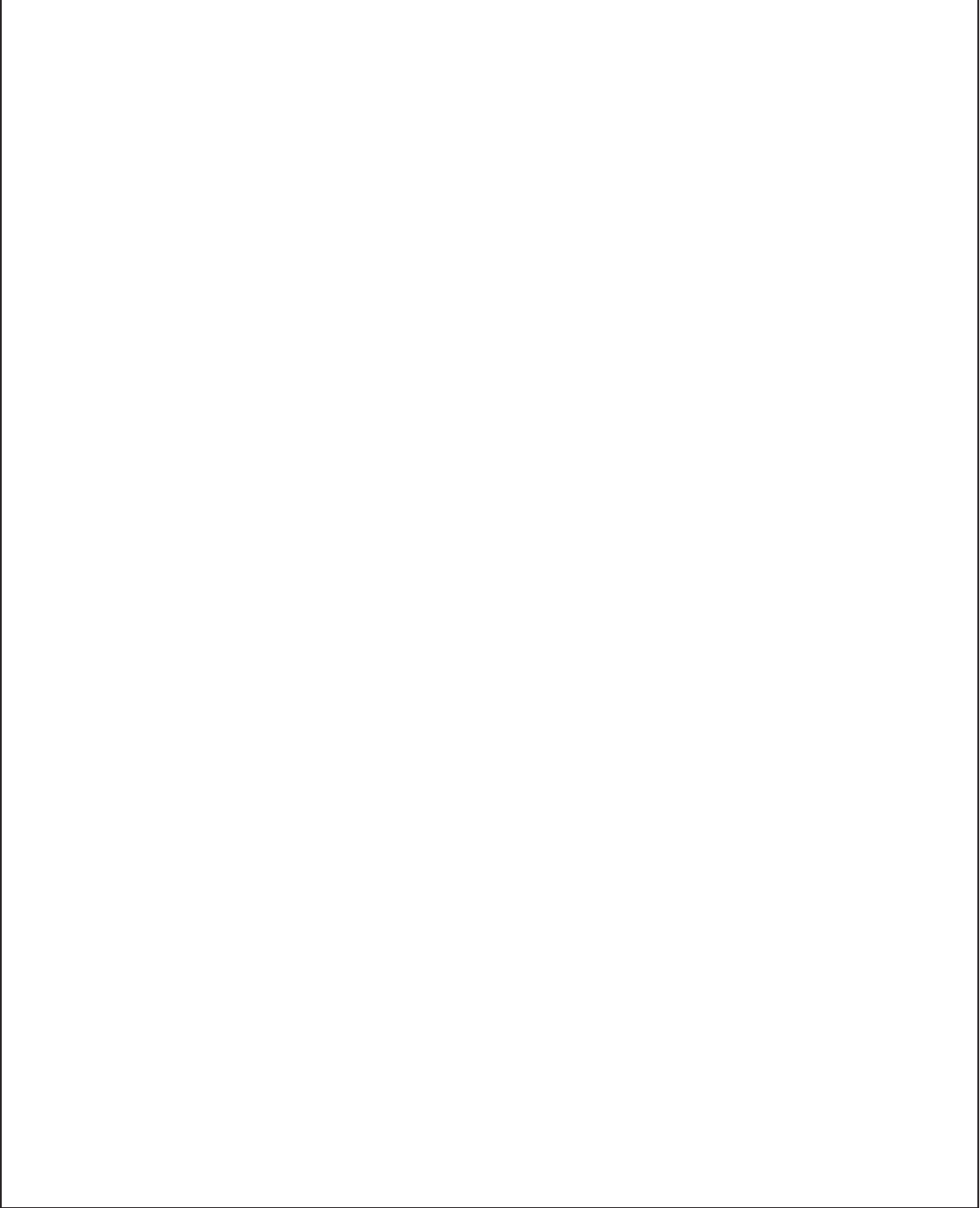
Die Methode `isDirectParent` soll `true` zurückgeben, falls `parent` und `child` im Baum als Inhalte existieren und `child` der Inhalt eines Knotens ist, der direktes Kind des Knotens mit dem Inhalt `parent` ist. Sonst soll `false` zurückgegeben werden. Gehen Sie davon aus, dass `parent` und `child` immer ungleich `null` sind.

```
public boolean isDirectParent( T c1, T c2 )
```

```
{
```

```
    if ( !isEmpty() )
```

```
    {
```



```
    }
```

```
    else
```

```
    {
```

```
        return false;
```

```
    }
```

```
}
```

## Aufgabe 9

Vervollständigen Sie die Methode `int shortest()`.

Die Methode `shortest()` gibt die Anzahl der Knoten auf dem kürzesten Pfad von der Wurzel bis zu einem Knoten mit **genau einem direkten** Nachfolger zurück. Gibt es keinen solchen Knoten, soll `0` zurückgegeben werden. Besteht der Baum nur aus einem Knoten, ist also die Wurzel zugleich das einzige Blatt, soll `1` zurückgegeben werden.

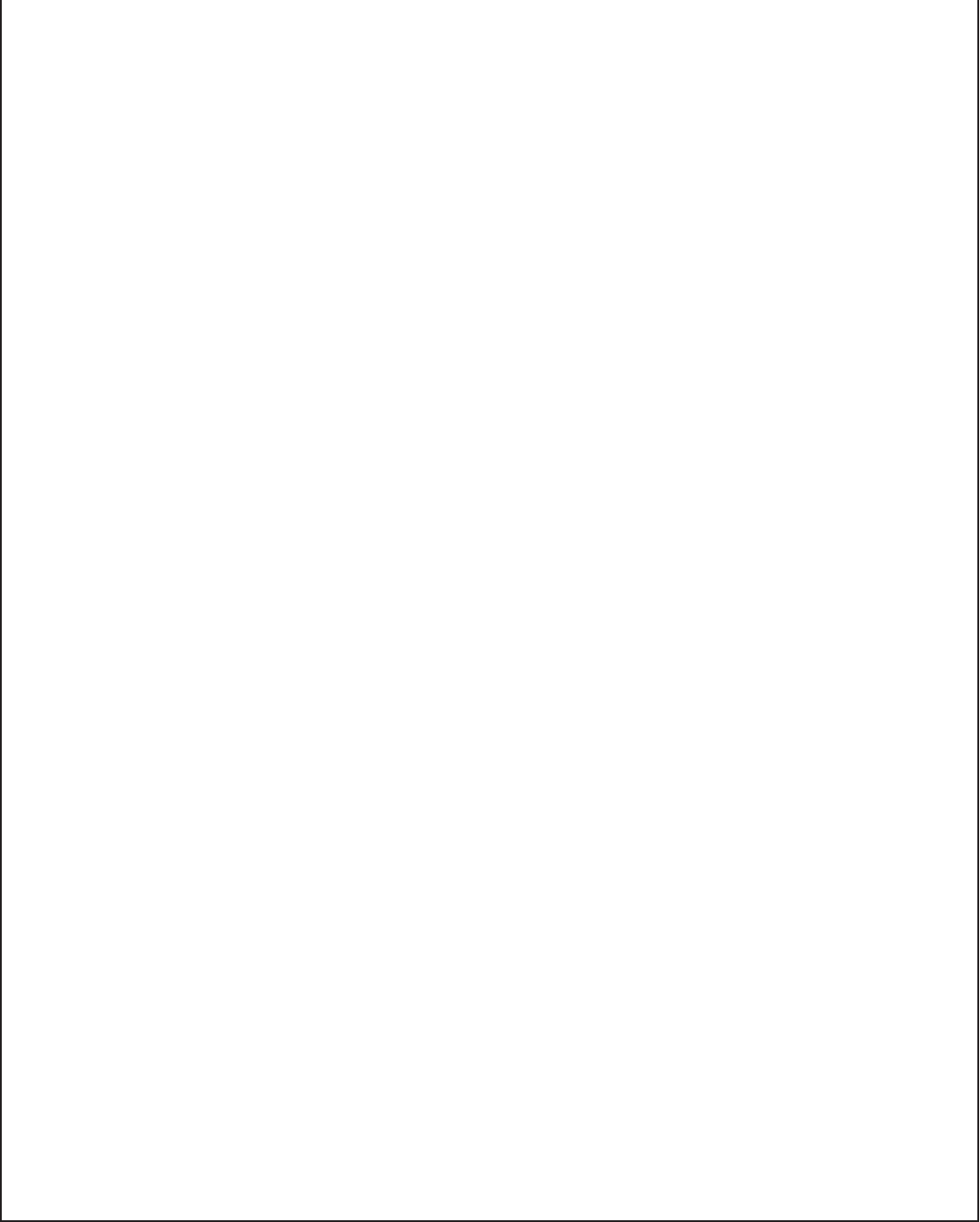
```
public int shortest()
{
    if ( !isEmpty() )
    {

    }
    else
    {
        return 0;
    }
}
```

## Aufgabe 10

Vervollständigen Sie die Methode `boolean succeeds( T pred, T succ )`.

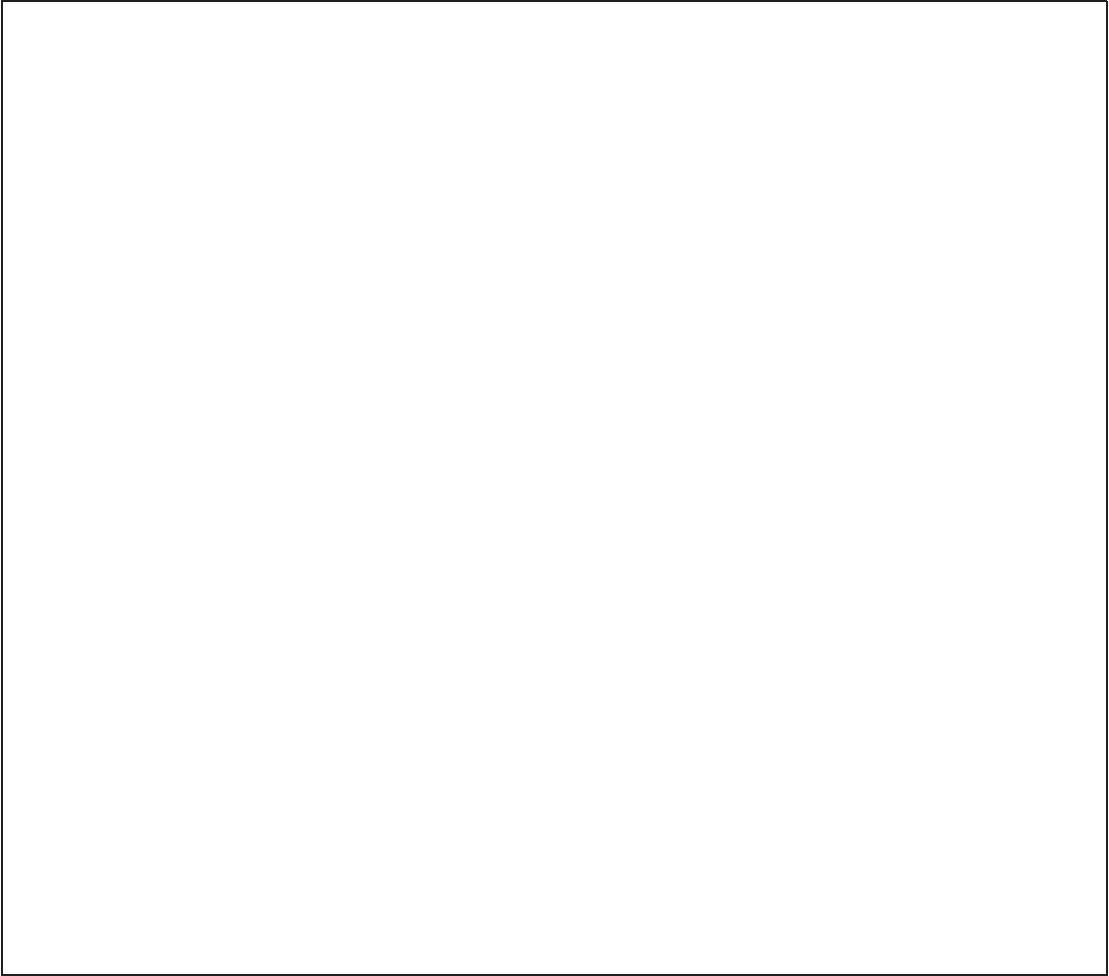
Die Methode `succeeds` gibt `true` zurück, falls `pred` und `succ` im Baum als Inhalte existieren und `succ` der Inhalt eines Knotens ist, der ein – möglicherweise entfernt liegender – Nachfolger des Knotens mit dem Inhalt `pred` ist. Sonst wird `false` zurückgegeben. Gehen Sie davon aus, dass `pred` und `succ` immer ungleich `null` sind.

```
public boolean succeeds( T pred, T succ )
{
    if ( !isEmpty() )
    {
        
    }
    else
    {
        return false;
    }
}
```

## Aufgabe 11

Vervollständigen Sie die Methode `boolean completePath()`.

Die Methode `completePath` soll `true` zurückgeben, falls es im Baum mindestens einen Pfad von der Wurzel zu einem Blatt gibt, auf dem alle inneren Knoten einen linken und einen rechten Nachfolgeknoten besitzen, die beide keine leeren Bäume sind. Existiert kein solcher Pfad, soll `false` zurückgegeben werden. Hat der Baum keine inneren Knoten oder ist er leer, so soll `true` zurückgegeben werden.

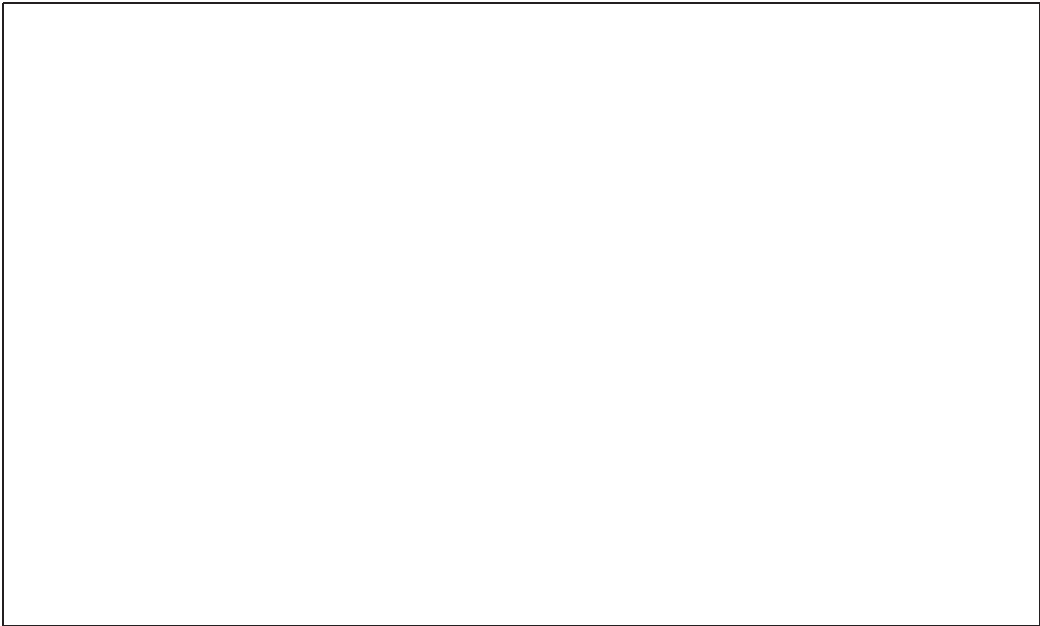
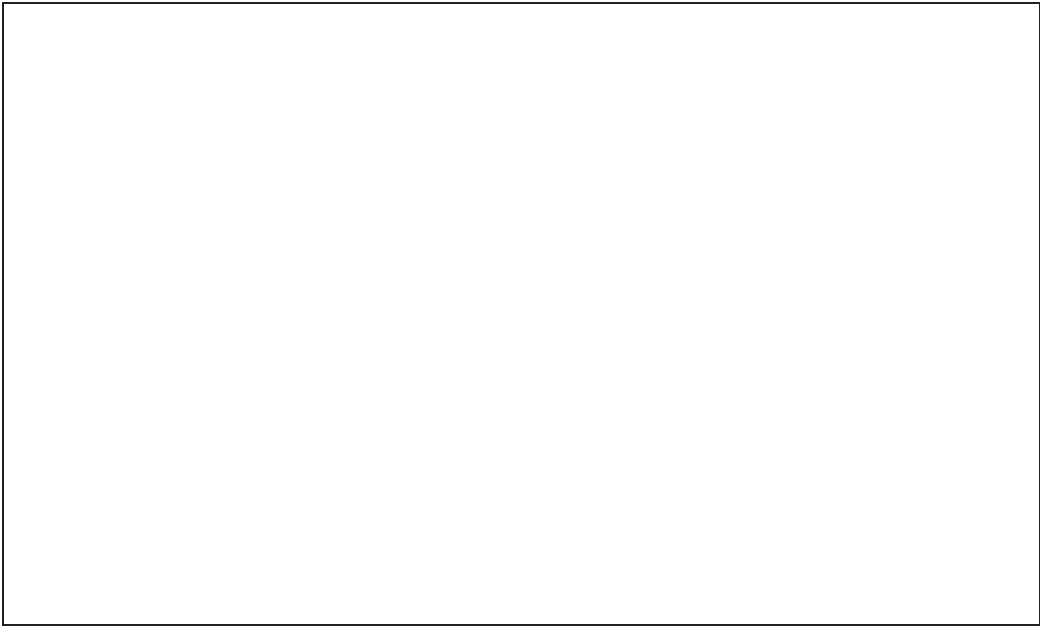
```
public boolean completePath()
{
    if ( isEmpty() || isLeaf() )
    {
        return true;
    }
    else
    {
        
    }
}
```

## Aufgabe 12

Vervollständigen Sie die Methode `T maxOfLess( T obj )`.

Die Methode `maxOfLess` soll den größten im Baum abgelegten Inhalt zurückgeben, der kleiner als `obj` ist. Gibt es keinen solchen Inhalt, so soll `null` zurückgegeben werden.

Der Vergleich soll durch einen Aufruf von `compareTo` erfolgen.

```
public T maxOfLess( T obj )
{
    if ( isEmpty() || obj == null )
    {
        return null;
    }
    else
    {
        if ( getContent().compareTo( obj ) >= 0 )
        {
            
        }
        else
        {
            
        }
    }
}
```

## Aufgabe 13

Vervollständigen Sie einen weiteren Konstruktor für die Klasse `BinarySearchTree<T>`.  
Der Konstruktor soll einen binären Suchbaum erstellen, der die beiden als Parameter übergebenen Inhalte enthält.

```
public BinarySearchTree( T p1, T p2 )  
{
```

```
}
```

## Aufgabe 14

Vervollständigen Sie die Methode `int levelOfMax()`.

Die Methode `levelOfMax` soll die Ebene zurückgeben, auf der der Knoten mit dem größten Inhalt liegt.

Die Wurzel liegt auf der Ebene `0`.

- Ist der Baum leer, so soll `-1` zurückgegeben werden.
- Der Vergleich soll durch einen Aufruf von `compareTo` erfolgen.

```
public int levelOfMax()  
{
```

```
    if ( isEmpty() )
```

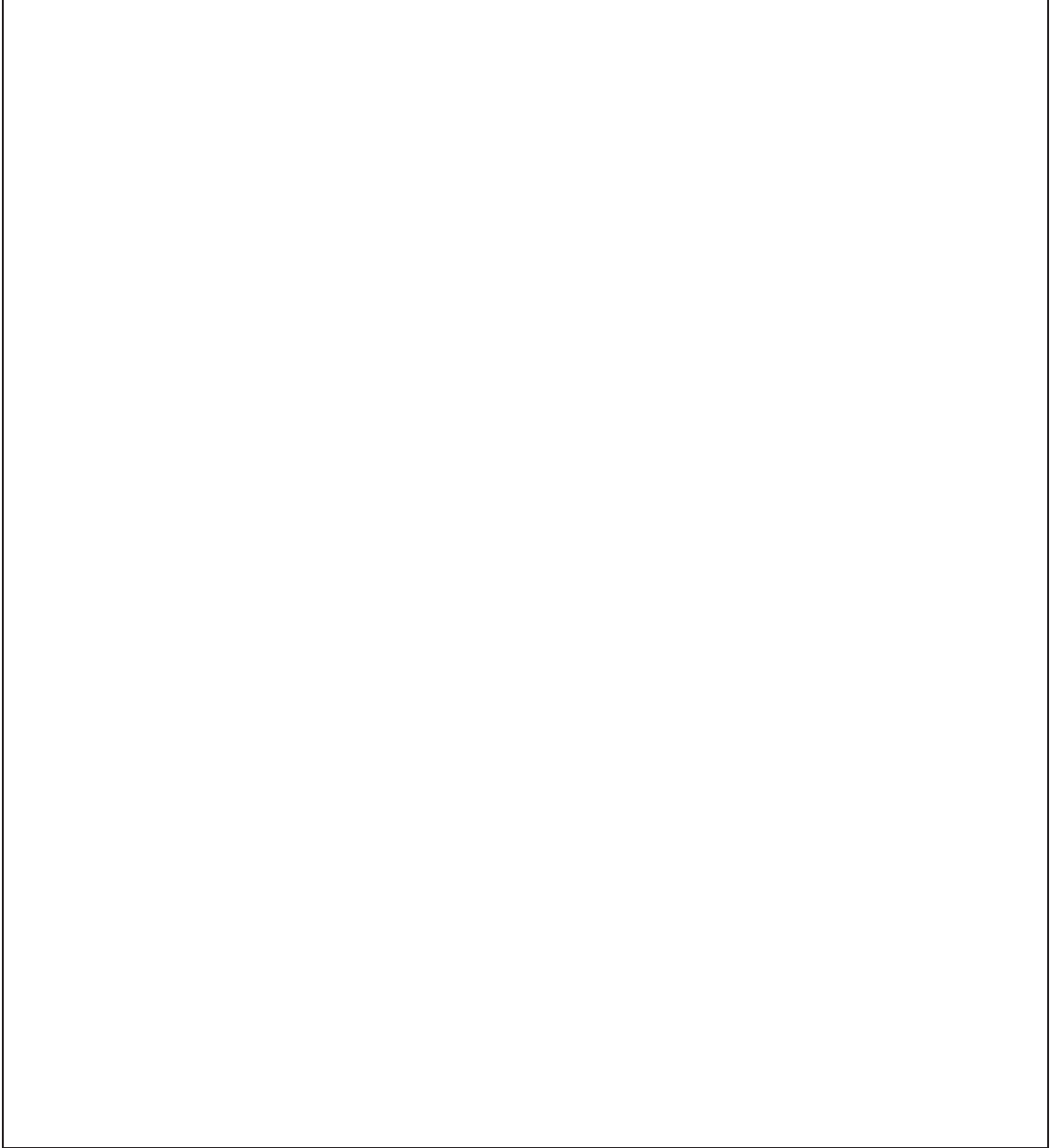
```
    {
```

```
        return -1;
```

```
    }
```

```
    else
```

```
    {
```



```
    }
```

```
}
```

## Aufgabe 15

Vervollständigen Sie die Methode `int balancedLeaves()`.

Die Methode `balancedLeaves` soll einen Wert größer oder gleich 0 zurückgeben, wenn sich für jeden Knoten die Anzahlen der Blätter im linken und im rechten Teilbaum um höchstens 1 unterscheiden. Sonst soll ein negativer Wert zurückgegeben werden.

```
public int balancedLeaves()
```

```
{
```

```
    if ( isEmpty() )
```

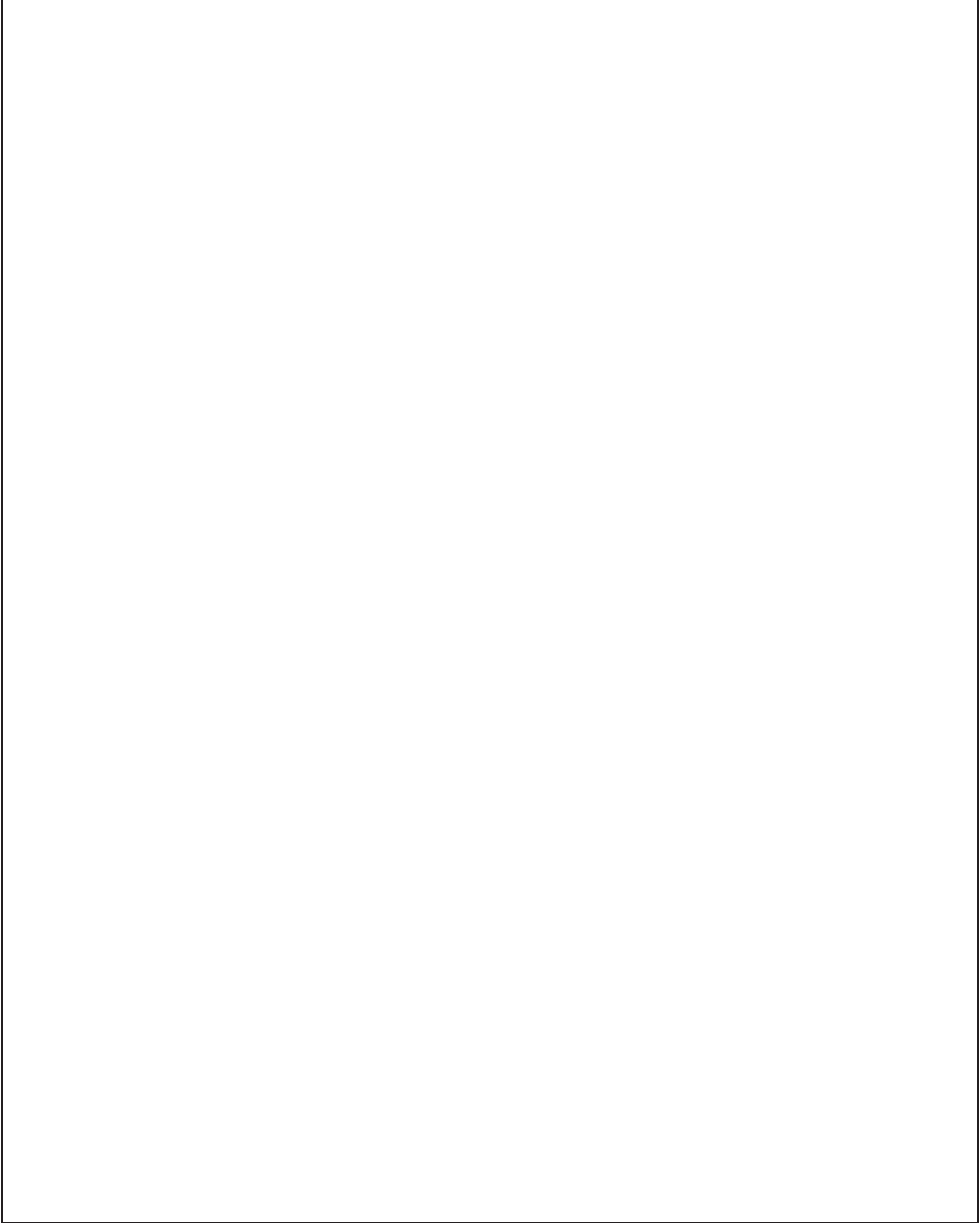
```
    {
```

```
        return 0;
```

```
    }
```

```
    else
```

```
    {
```



```
    }
```

```
}
```



## Aufgabe 16

Vervollständigen Sie die Methode `rightComplete()`.

Die Methode `rightComplete` soll **true** zurückgeben, falls alle inneren Knoten des Baums einen rechten Nachfolgeknoten besitzen, der kein leerer Baum ist. Hat der Baum keine inneren Knoten, so wird ebenfalls **true** zurückgegeben. Sonst soll **false** zurückgegeben werden.

```
public boolean rightComplete()
{
    if ( isEmpty() || isLeaf() ) )
    {
        return true;
    }
    else
    {
        }
    }
}
```



## Aufgabe 17

Vervollständigen Sie die Methode `shortestPath()`.

Die Methode `shortestPath` soll die Anzahl der Knoten auf dem kürzesten Weg zwischen der Wurzel des Baums und einem Blatt zurückgeben. Besteht der Baum nur aus der Wurzel, so soll `1` zurückgegeben werden. Ist der Baum leer, wird `0` zurückgegeben werden.

```
public int shortestPath()
```

```
{
```

```
    if ( isEmpty() )
```

```
    {
```

```
        return 0;
```

```
    }
```

```
    else
```

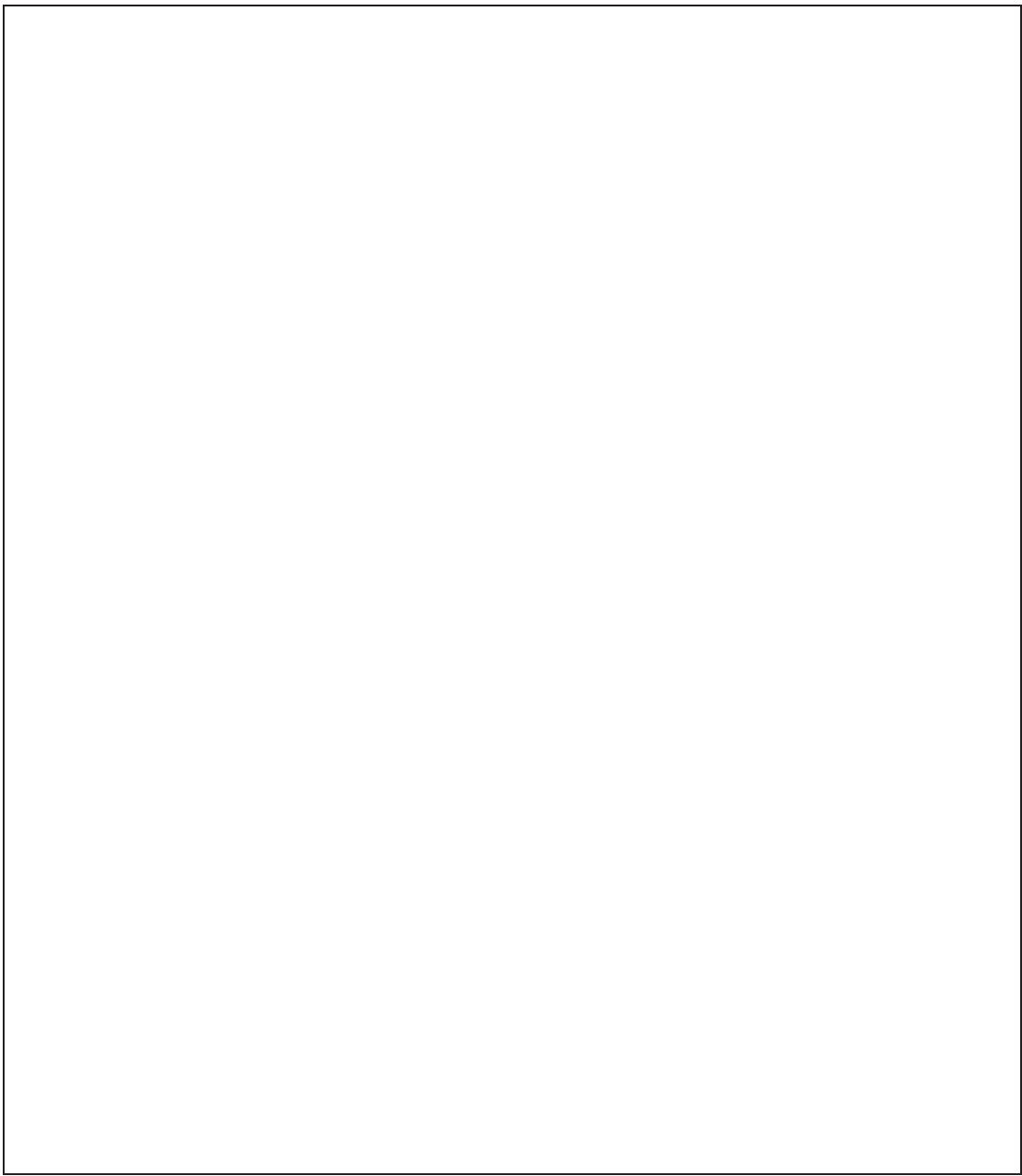
```
    {
```

```
    }
```

## Aufgabe 18

Vervollständigen Sie die Methode `oneLevel()`.

Die Methode `oneLevel` soll einen positiven Wert zurückgeben, wenn zwischen jedem Blatt des Baums und der Wurzel die gleiche Anzahl von Kanten liegt. Besitzen die Blätter unterschiedliche Abstände zur Wurzel, soll ein negativer Wert zurückgegeben werden. Ist der Baum leer, wird `0` zurückgegeben.

```
public int oneLevel()
{
    if ( isEmpty() )
    {
        return 0;
    }
    else
    {
        
    }
}
```

## Aufgabe 19

Vervollständigen Sie den Konstruktor `BinarySearchTree( T t1, T t2, T t3 )`.  
Der Konstruktor soll einen Baum mit den Inhalten `t1`, `t2` und `t3` erzeugen. Die den Parameter übergebenen Inhalte sollen in aufsteigender Reihenfolge vorliegen.

Vergleiche sollen durch einen Aufruf der Methode `compareTo` erfolgen.

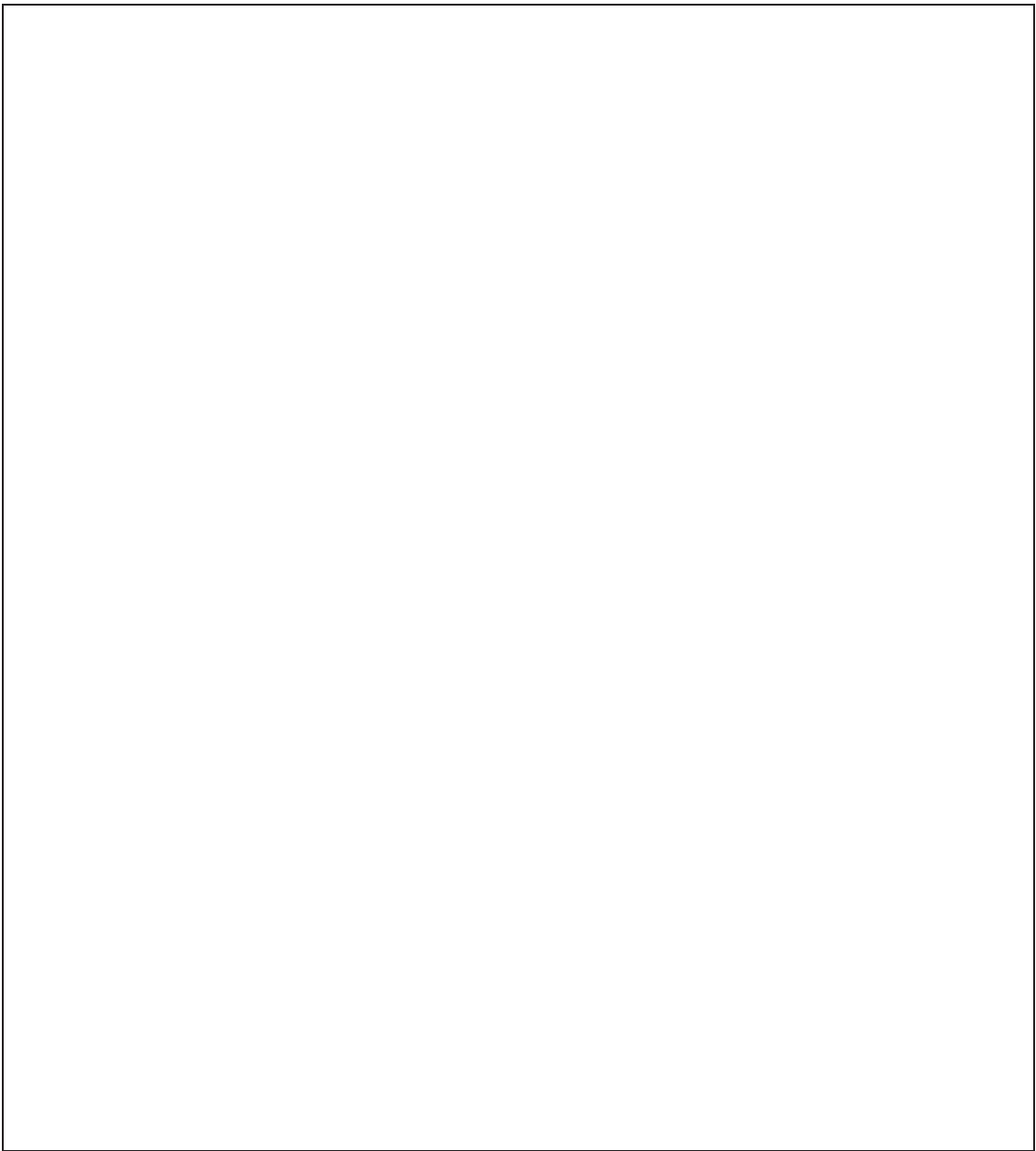
```
public BinarySearchTree( T t1, T t2, T t3 )
{
    if ( t1.compareTo( t2 ) < 0 && t2.compareTo( t3 ) < 0 )
    {
        
    }
    else
    {
        throw new RuntimeException();
    }
}
```

## Aufgabe 20

Vervollständigen Sie die Methode `int inDiffOut()`.

Die Methode `inDiffOut` soll die Differenz aus der Anzahl der inneren Knoten und der Anzahl der Blätter zurückgeben. Besitzt der Baum mehr Blätter als innere Knoten, soll also ein negativer Wert erzeugt werden.

Ist der Baum leer, so soll `0` zurückgegeben werden.

```
public int inDiffOut()
{
    if ( isEmpty() )
    {
        return 0;
    }
    else
    {
        
    }
}
```

## Aufgabe 21

Vervollständigen Sie die Methode `int levelOf( T obj )`.

Die Methode `levelOf` soll die Ebene zurückgeben, auf der sich der Knoten mit dem Inhalt `obj` befindet. Die Wurzel liegt auf der Ebene `0`.

Ist `obj` nicht im Baum enthalten, so soll `-1` zurückgegeben werden.

Vergleiche sollen durch einen Aufruf der Methode `compareTo` erfolgen.

```
public int levelOf( T obj )
{
```

```
if ( isEmpty() )
```

 $\{$ 

```
return -1;
```

}

**else**

 $\{$ 

}

## Aufgabe 22

Vervollständigen Sie die Methode `int completeLevels()`.

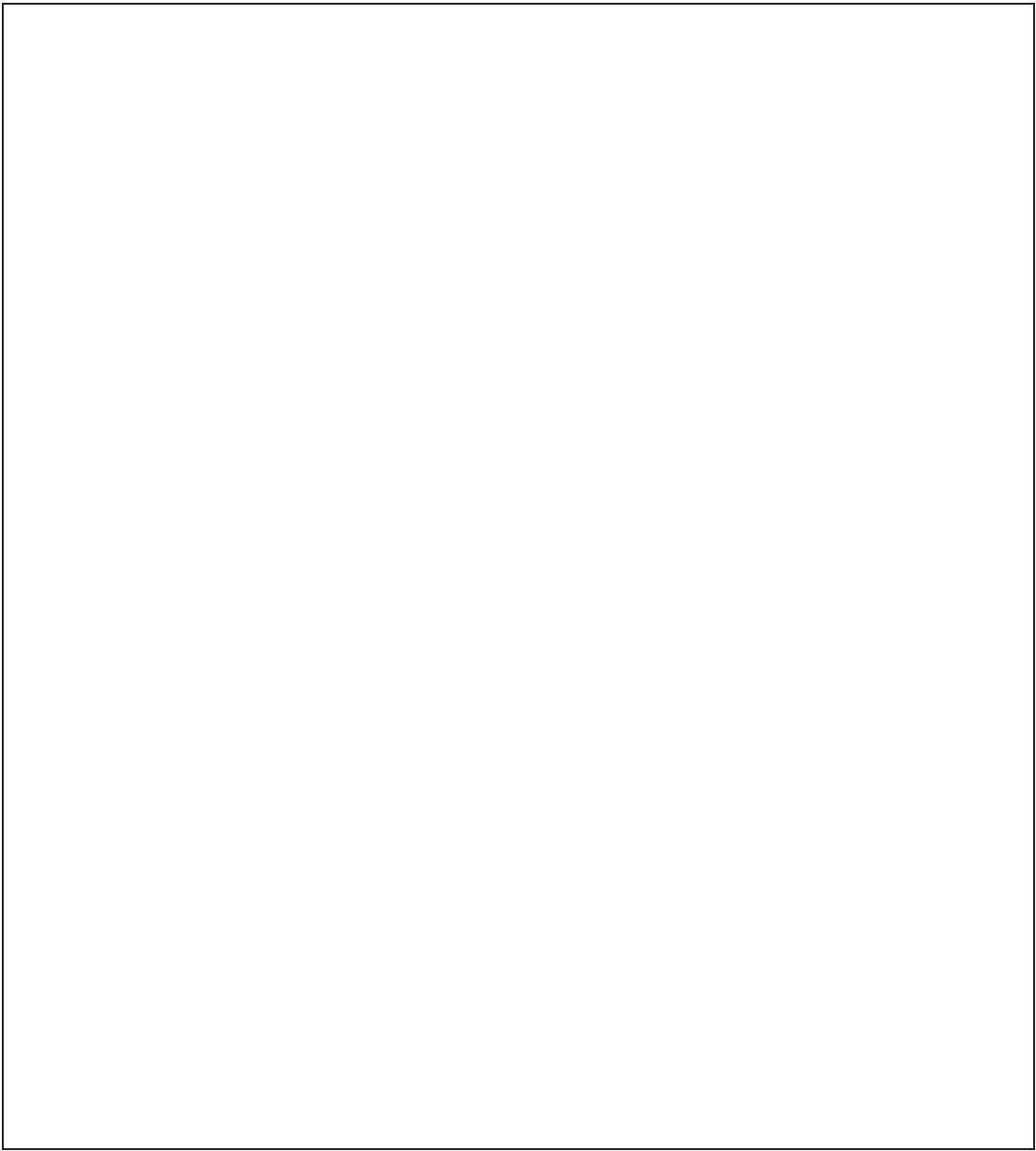
Die Methode `completeLevels` soll von der Wurzel aus die Anzahl der *vollständigen Ebenen* bestimmen.

Die *Ebene  $n$*  wird durch die Knoten gebildet, deren Pfad bis zur Wurzel  $n$  Verweise besitzt.

Die *Ebene  $n$*  ist *vollständig*, wenn sie  $2^n$  Knoten enthält.

Wenn eine Wurzel existiert, so liegt diese auf der Ebene  $0$ , die damit wegen  $2^0 = 1$  vollständig ist.

Ist der Baum leer, so soll  $0$  zurückgegeben werden.

```
public int completeLevels()
{
    if ( isEmpty() )
    {
        return 0;
    }
    else
    {
        
    }
}
```

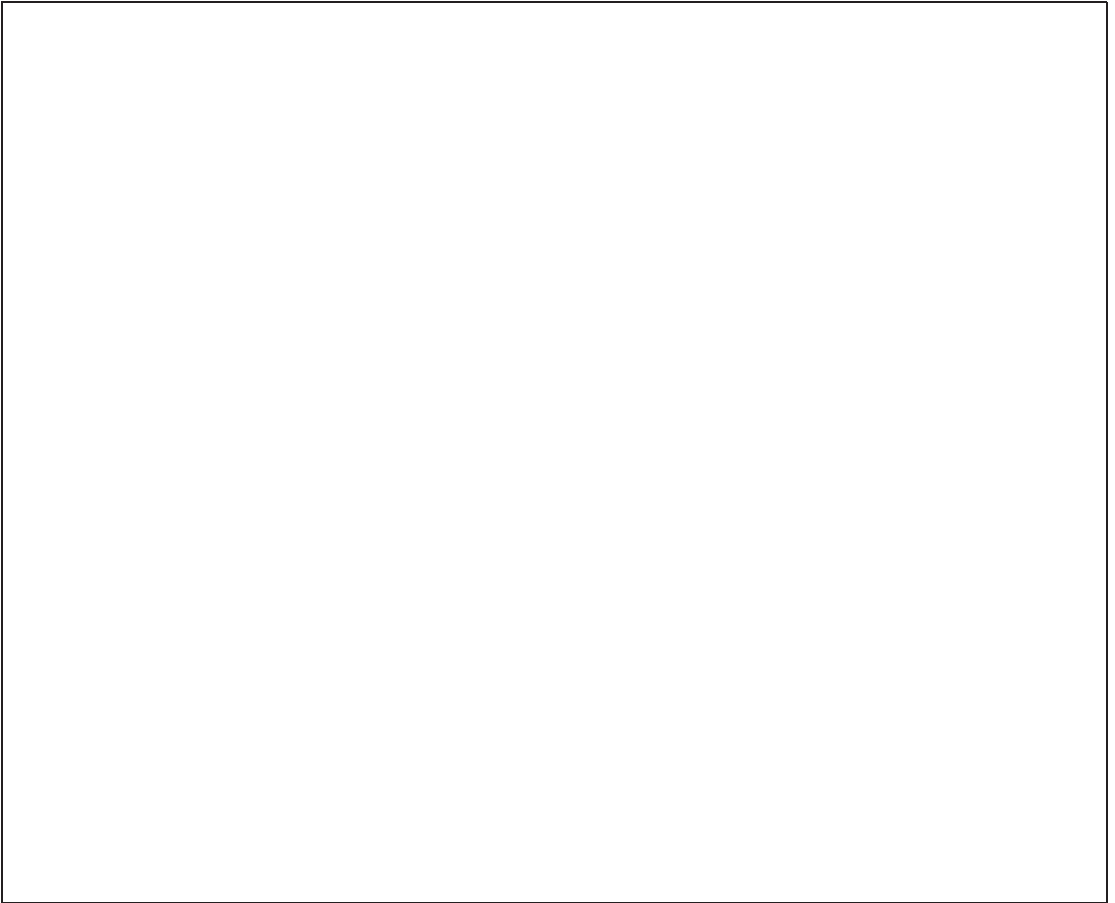
## Aufgabe 23

Vervollständigen Sie die Methode `innerNode()`.

Die Methode `innerNode` soll **true** zurückgeben, falls es mindestens **einen** inneren Knoten im Baum gibt, der einen linken und einen rechten Nachfolgeknoten besitzt, die beide keine leeren Bäume sind. Hat der Baum keine inneren Knoten oder ist er leer, so soll **false** zurückgegeben werden.

```
public boolean innerNode()
{
    if ( isEmpty() || isLeaf() ) )
    {
        return false;
    }
    else
    {

```



```
    }
}
```




## Aufgabe 24

Vervollständigen Sie die Methode `leavesOnLevel( int level )`.

Die Methode `leavesOnLevel` soll die Anzahl der Blätter zurückgeben, die auf der Ebene `level` des Baums liegen. Die Wurzel soll der Ebene `0` zugeordnet werden.

Ist der Baum leer, wird `0` zurückgegeben werden.

```
public int leavesOnLevel( int level )
{
    if ( level >= 0 )
    {
        if ( isEmpty() )
        {
            return 0;
        }
        else
        {
            
        }
    }
    else
    {
        throw new RuntimeException();
    }
}
```

## Aufgabe 25

Ergänzen Sie die aus der Vorlesung bekannte Klasse `BinarySearchTree<T>`. Die zum Lösen dieser Aufgabe notwendige Beschreibung der Klasse finden Sie im Anhang. Bei der Implementierung der geforderten Methoden dürfen nur die im Anhang aufgeführten Methoden genutzt werden.

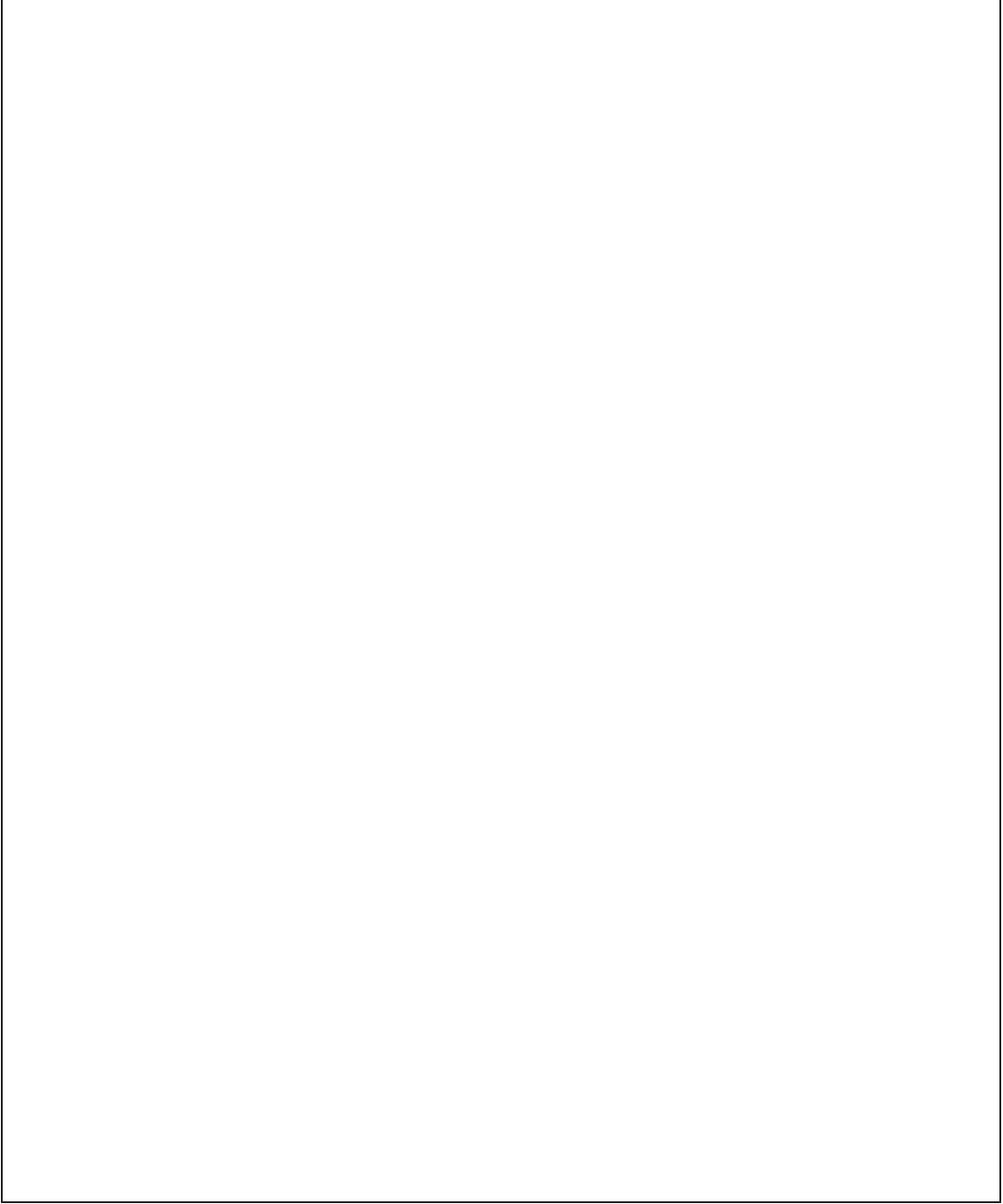
Vervollständigen Sie die Methode `void toList( DoublyLinkedList<T> list )`. Die Methode `toList` soll alle Inhalte des Baums in absteigender Sortierung an die als Argument übergebene Liste anhängen. Dabei kann die Methode `add` der Klasse `DoublyLinkedList` genutzt werden.

```
public void toList( DoublyLinkedList<T> list )
```

```
{
```

```
    if ( !isEmpty() )
```

```
    {
```



```
    }
```

```
}
```

## Aufgabe 26

Vervollständigen Sie die Methode `boolean samePath( T t1, T t2 )`.

Die Methode `samePath` soll `true` liefern, wenn die Inhalte `t1` und `t2` auf dem gleichen Pfad zwischen der Wurzel und einem Blatt liegen. Sonst soll `false` zurückgegeben werden.

```
public boolean samePath()
{
    boolean foundT1 = false;
    boolean foundT2 = false;

    BinarySearchTree<T> current = ;

    while ( !current.isEmpty() )
    {

    }

    return false;
}
```

## Aufgabe 27

Vervollständigen Sie die Methode `void deleteAllLeaves()`.

Die Methode `deleteAllLeaves` soll alle Blätter des Baums entfernen, die zum Zeitpunkt des Aufrufs der Methode in dem Baum enthalten sind.

```
public void deleteAllLeaves()
{
    if ( !isEmpty() )
    {
        if (  )
        {

        }
        else
        {

        }
    }
}
```

#### Anhang – Programmcode der Klasse `BinarySearchTree<T extends Comparable<T>>`

```
public class BinarySearchTree<T extends Comparable<T>> {
    private T content;
    private BinarySearchTree<T> leftChild, rightChild;
    public BinarySearchTree() { ... }
    public T getContent() { ... }
    public boolean isEmpty() { ... }
    public boolean isLeaf() { ... }
}
```

#### Anhang – Programmcode der Klasse `DoublyLinkedList<T>`

```
public class DoublyLinkedList<T> {
    private Element first, last;
    private int size;
    public DoublyLinkedList() { ... }
    public int size() { ... }
    public boolean isEmpty() { ... }
    // Element
    private static class Element {
        private T content;
        private Element pred, succ;
        public Element( T c ) { ... }
        public T getContent() { ... }
        public void setContent( T c ) { ... }
        public boolean hasSucc() { ... }
        public Element getSucc() { ... }
        public void connectAsSucc( Element e ) { ... }
        public void disconnectSucc() { ... }
        public boolean hasPred() { ... }
        public Element getPred() { ... }
        public void connectAsPred( Element e ) { ... }
        public void disconnectPred() { ... }
    }
}
```

#### Anhang – Programmcodes der Interfaces

```
public interface Iterator<T> {
    public abstract boolean hasNext();
    public abstract T next();
}
```

```
public interface Iterable<T> {
    public abstract Iterator<T> iterator();
}
```

```
public interface Comparable<T> {
    public abstract int compareTo( T t );
}
```

// Der Rückgabewert ist positiv, falls das  
// ausführende Objekt größer als t ist.