Datenstrukturen, Algorithmen und Programmierung 1
Probeklausur

Name: _____ **Matr.-Nr.:** _____

Unterschrift: _____

Aufgabe	mögliche Punkte	erreichte Punkte
1	10	
2	6	
3	12	
4	12	
5	14	
6	6	
7	18	
8	13	
9	9	
Summe	100	

Würde diese Klausur bewertet, so würde die Note anhand der folgenden Tabelle bestimmt werden.

Benotungsskala	
Mindest-punktzahl	*Note*
94	*1,0*
88	*1,3*
82	*1,7*
76	*2,0*
70	*2,3*
64	*2,7*
58	*3,0*
52	*3,3*
46	*3,7*
40	*4,0*

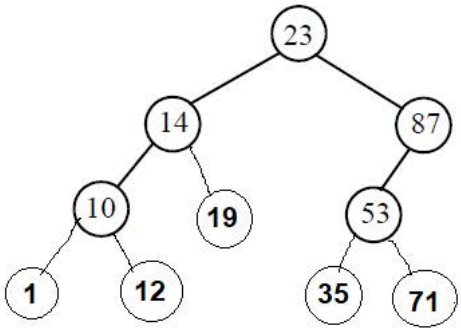
Zum Bestehen der Klausur müssten mindestens 40 Punkte erreicht werden.

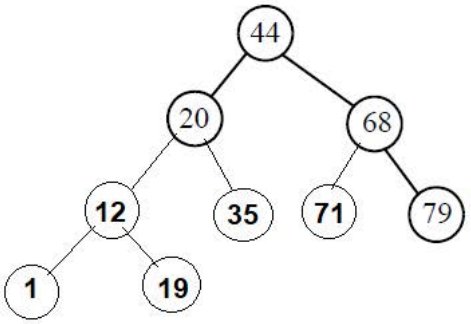
***wichtige Hinweise*:**

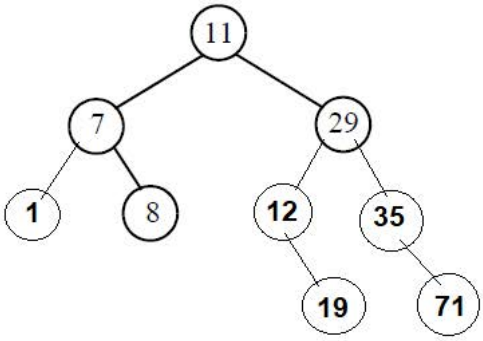
- Im Anhang dieser Klausur finden Sie die eventuell benötigten Programmtexte zu den vorgegebenen, bereits aus der Vorlesung bekannten Klassen und Interfaces.
- In dieser Klausur dürfen Sie immer nur vorgegebene Java-Programmtexte *mit Bezug* zu den durch Rahmen ☐ gekennzeichneten Stellen ergänzen, idealerweise durch Schreiben innerhalb dieser Rahmen, aber auch durch Angabe einer eindeutigen Beziehung durch Pfeile oder Markierungen.
- An den vorgegebenen Programmtexten dürfen Sie *keine* Änderungen vornehmen.
- An *keiner* Stelle dieser Klausur dürfen Sie konzeptionell außerhalb von Rahmen ☐ liegende Programmtexte ergänzen.
- Gehen Sie bei allen Aufgaben davon aus, dass an einen Feld-Parameter immer ein Feld-Objekt (und nicht `null`) als Argument übergeben wird.

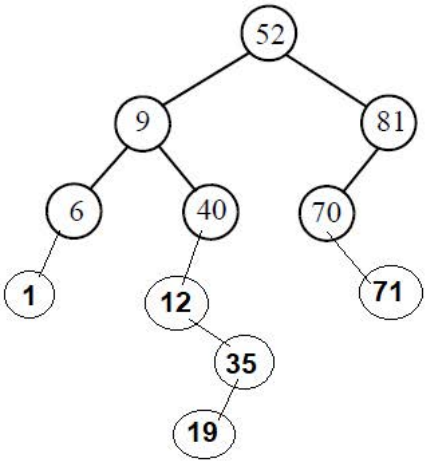
Aufgabe 1 (Datenstrukturen, Algorithmen, Syntaxdiagramme)

a) [2 Punkte] **(Binäre Suchbäume)**
Fügen Sie die Werte *12*, *35*, *19*, *71* und *1* in *genau* dieser Reihenfolge in jeden der unten angegebenen binären Suchbäume ein, indem Sie die Zeichnungen entsprechend ergänzen.









b) [2 Punkte] (**Quicksort**)

Sortieren Sie die nachfolgenden Zahlen mit dem *Quicksort*-Algorithmus. Wählen Sie als Pivotelement das letzte Element des zu sortierenden Bereichs. Geben Sie die Werte der gewählten Pivotelemente in der Reihenfolge an, in der sie beim Sortieren bestimmt werden.

zu sortierenden Zahlenfolge: 13 26 8 28 34 20 10 12 6

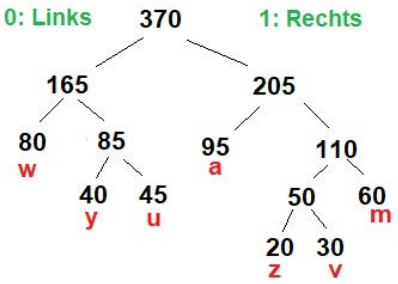
Folge der Werte der Pivotelemente: 6 13 20 12 34 10 28 8 26

Aufgabe 1 (Datenstrukturen,Algorithmen,Syntaxdiagramme) – Fortsetzung

c) [4 Punkte] **(Huffman-Codierung)**
Gegeben seien die in der folgenden Tabelle aufgeführten Zeichen, die in einem Text mit den angegebenen Häufigkeiten auftreten. Bestimmen Sie mit dem in der Vorlesung vorgestellten Algorithmus die *Huffman-Codierung* für diese Zeichen und vervollständigen Sie die Tabelle.

Zeichen	Häufigkeit	*Huffman-Codierung*
z	20	1100
v	30	1101
w	80	00
m	60	111
y	40	010
a	95	10
u	45	011

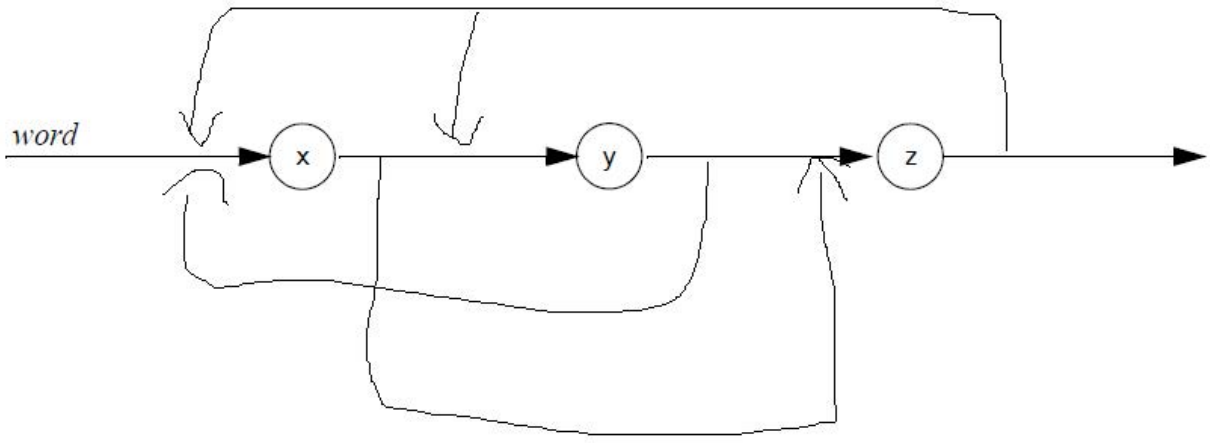
Nach Häufigkeit sortieren:
20 30 40 45 60 80 95



d) [2 Punkte] (**Syntaxdiagramm**)

Vervollständigen Sie das Syntaxdiagramm *word* **nur mit Kanten** derart, dass es alle Wörter zulässt, die die folgenden Regeln einhalten:

- Ein Wort darf nur die folgenden Zeichen enthalten: `x` `y` `z`
- Ein Wort muss mit dem Zeichen `x` beginnen.
- Ein Wort muss mit dem Zeichen `z` enden.
- Zwei gleiche Zeichen dürfen nicht aufeinander folgen.
- Ein Wort muss mindestens zwei Zeichen enthalten.
- Beispielwörter: `xz` `xyz` `xyxz` `xzyxz` `xyzxzyxz`



___ / 10

Aufgabe 2 (Polymorphie)

a) [2 Punkte] Geben Sie die Zeichenfolge an, die durch die Methode `run` ausgegeben wird.

```
public static void m( double p ) { System.out.print( "A " ); }
public static void m( int p ) { System.out.print( "L " ); }
public static void m( char p ) { System.out.print( "X " ); }

public static void run()
{
    m( 'a' );
    m( 7 / 3.0 );
    m( 'a' / 'b' );
    m( new Integer( 5 ) );
}
```

Ausgabe: X A L L

b) [4 Punkte] Gegeben sind die beiden folgenden Klassenhierarchien. Geben Sie die Zeichenfolge an, die durch die Methode `run` ausgegeben wird.

```
class All { /* ... */ }
class Most extends All { /* ... */ }
class Special extends Most { /* ... */ }

class Top {
     public void m( All p ) { System.out.print("A "); }
}

class Middle extends Top {
     public void m( All p ) { System.out.print("M "); }
     public void m( Special p ) { System.out.print("L "); }
}

class Bottom extends Middle {
     public void m( Most p ) { System.out.print("V "); }
     public void m( Special p ) { System.out.print("X "); }
}

class Test {
     public static void run()
     {
          All all = new All();
          Most most = new Most();
          Special special = new Special();

          Top x = new Middle();
          Top y = new Bottom();
          Middle z = new Bottom();

          x.m( most );
          x.m( special );
          y.m( all );
          y.m( special );
          z.m( all );
          z.m( most );
     }
}
```

Info: Zur Compilezeit wird m(All p) ausgewählt!

Ausgabe: M M M M M M

___ / 6

Aufgabe 3 (Klassen)

a) [6 Punkte] Die Klasse `Ints` besitzt Konstruktoren und weitere Methoden, die hier aber nicht verwendet werden sollen. Vervollständigen Sie die Klasse `Ints` um die folgenden zusätzlichen Methoden:

- `void set( int index, int val )` setzt den Wert im Feld `numbers` am Index `index` auf den Wert `val`, falls der Index gültig ist. Sonst geschieht nichts.
- `int count( int val )` gibt die Häufigkeit zurück, mit der der Wert `val` im Feld `numbers` auftritt.
- `void substitute( int oldVal, int newVal )` ersetzt im Feld `numbers` jedes Vorkommen des Wertes `oldVal` durch den Wert `newVal`.

```
public class Ints
{
    private int[] numbers;

    // ... (Konstruktoren und weitere Methoden sind nicht von Interesse)

    public void set( int index, int val )
    {
        if (index >= 0 && index < numbers.length)
        {
                numbers[index] = val;
        }

    }

    public int count( int val )
    {
        int quantity = 0;
        for ( int cand : numbers )
        {
            if (cand == val)
            {
                    quantity++;
            }

        }
        return quantity;
    }

    public void substitute( int oldVal, int newVal )
    {
        for ( int i=0; i < numbers.length ; i++ )
        {
            if (numbers[i] == oldVal)
            {
                    numbers[i] = newVal;
            }

        }
    }
}
```

Aufgabe 3 (Klassen) – Fortsetzung

b) [6 Punkte] Die Klasse `Storage` besitzt als Attribut ein Feld der aus Aufgabenteil a) bekannten Klasse `Ints`. Die Klasse `Storage` besitzt auch Konstruktoren und weitere Methoden, die hier aber nicht verwendet werden sollen. Vervollständigen Sie die Klasse `Storage` um zwei Methoden.
Beachten Sie bei den Lösungen, dass eine Referenz auch auf `null` verweisen kann.

- `boolean contains( int p )` gibt `true` zurück, falls der Wert von `p` mindestens einmal in den in `values` abgelegten Werten vorkommt. Sonst wird `false` zurückgegeben.
- `boolean allAround( int p )` gibt `true` zurück, falls jedes der Elemente des Feldes `values` auf ein `Ints`-Objekt (und nicht auf `null`) verweist und in jedem dieser `Ints`-Objekte der Wert von `p` mindestens einmal vorkommt. Sonst wird `false` zurückgegeben.

```
public class Storage
{
    private Ints[] values;

    // ... (Konstruktoren und weitere Methoden sind nicht von Interesse)

    public boolean contains( int p )
    {
        for( Ints cand : values )
        {
            if (cand != null && cand.count(p) > 0)
            {
                    return true;
            }
        }
        return false;
    }

    public boolean allAround( int p )
    {
        for( Ints cand : values )
        {
            if (cand == null || cand.count(p) == 0)
            {
                    return false;
            }
        }
        return true;
    }
}
```

___ / 12

Aufgabe 4 (Methoden)

a) [8 Punkte] Vervollständigen Sie die Methode `void compress( Object[] arr )`.
Die Methode `compress` soll die in einem Feld erreichbaren Objekte so verschieben, dass die Reihenfolge der Objekte unverändert bleibt, aber alle auf `null` verweisenden Referenzen an das Ende des Feldes verschoben werden. Nach der Ausführung von `compress` soll es also einen Index `c` geben, für den gilt:

für alle `i<c` gilt `arr[i]!=null`
für alle `k>=c` gilt `arr[k]==null`

```
public static void compress( Object[] arr )
{
    int positionNextNull = 0;
    int positionNextObject = 0;
    while ( positionNextNull < arr.length && positionNextObject < arr.length)
    {
        if ( arr[positionNextNull] != null )
        {
            positionNextNull++;
        }
        else
        {
            if (positionNextObject <= positionNextNull)
            {
                positionNextObject = positionNextNull + 1;
            }

            while (positionNextObject < arr.length && arr[positionNextObject] == null)
            {
                positionNextObject++;
            }

            if (positionNextObject < arr.length)
            {
                arr[positionNextNull] = arr[positionNextObject];
                arr[positionNextObject] = null;
            }
        }
    }
}
```

Aufgabe 4 (Methoden) – Fortsetzung

b) [4 Punkte] Vervollständigen Sie die Methode `boolean contains( int[] arr, int n, int limit)`, die rekursiv arbeiten soll. Die Methode `contains` soll `true` zurückgeben, wenn das Feld `arr` im Bereich der Indizes `0` ... `limit` der Wert `n` mindestens einmal vorkommt. Sonst soll `false` zurückgegeben werden. Wird ein unzulässiges Argument für `limit` übergeben, soll eine `IllegalArgumentException` geworfen werden.
Die Implementierung darf **keine** Schleifen enthalten.

```
public static boolean contains( int[] arr, int n, int limit)
{
    if ( limit < 0 || limit >= arr.length )
    {
        throw new IllegalArgumentException();
    }
    else
    {
        if ( arr[limit] == n )
        {
                return true;
        }
        else if ( limit == 0 )
        {
                return false;
        }
        else
        {
                return contains( arr, n, limit - 1);
        }
    }
}
```

___ / 12

Aufgabe 5 (Entwurfsmuster Iterator)

a) [8 Punkte] Die Klasse `MultiSequence` besitzt Konstruktoren und weitere Methoden, die hier aber nicht verwendet werden sollen. Ergänzen Sie die Klasse `MultiSequence` um einen Iterator, der einen Durchlauf über alle Inhalte des Feldes `numbers` ermöglicht.
Gehen Sie davon aus, dass jede Zeile des zweidimensionalen Feldes `numbers` immer jeweils mindestens einen `int`-Wert enthält.

```
public class MultiSequence
{
    private int[][] numbers;

    // ... (Konstruktor und weitere Methoden sind nicht von Interesse)

    public Iterator<Integer> iterator()
    {
        return new MultiIterator();
    }

    private class MultiIterator implements Iterator<Integer>
    {
        private int dim1, dim2;

        public MultiIterator()
        {
            dim1 = 0;
            dim2 = 0;
        }

        public boolean hasNext()
        {
            return dim1 < numbers.length && dim2 < numbers[dim1].length;
        }

        public Integer next()
        {
            if ( hasNext() )
            {
                int value = numbers[dim1][dim2];
                if (++dim2 >= numbers[dim1].length)
                {
                        dim1++;
                        dim2 = 0;
                }
                return value;
            } else {
                throw new IllegalStateException();
            }
        }
    }
}
```

Aufgabe 5 (Entwurfsmuster Iterator) – Fortsetzung

b) [6 Punkte] Die Methode `boolean oneInBoth( MultiSequence p1, MultiSequence p2 )` soll `true` zurückgeben, falls es mindestens einen `int`-Wert gibt, der in `p1` und in `p2` vorkommt. Sonst soll `false` zurückgegeben werden. Die Klasse `MultiSequence` soll aus Aufgabenteil a) übernommen werden.

```
public static boolean oneInBoth( MultiSequence p1, MultiSequence p2 )
{
    Iterator<Integer> iter1 = [ p1.iterator() ];

    while ( [ iter1.hasNext() ] )

    {
        int iValue = [ iter1.next(); ];

        Iterator<Integer> iter2 = [ p2.iterator() ];

        while ( [ iter2.hasNext() ] )
        {
            [ if (iValue == iter2.next())
              {
                  return true;
              } ]
        }
    }
    return false;
}
```

___ / 14

Aufgabe 6 (Entwurfsmuster Strategie)

Hinweis: Die Klasse `DoublyLinkedList<T>` finden Sie im Anhang.

a) [2 Punkte] Vervollständigen Sie zu der vorgegebenen Klasse `DoublyLinkedList<T>` eine Strategie-Klasse so, dass ein Objekt dieser Strategie als Argument der Methode `deleteSelected` bewirkt, dass alle Elemente aus der aufrufenden Liste, deren Inhalt auf **`null`** verweist, gelöscht werden.

```
public class RemoveNullStrategy<S>
implements DoublyLinkedList.DeletionStrategy<S>
{
    public boolean select( S ref )
    {
        return ref == null;
    }
}
```

b) [4 Punkte] Vervollständigen Sie zu der vorgegebenen Klasse `DoublyLinkedList<T>` eine Strategie-Klasse so, dass ein Objekt dieser Strategie als Argument der Methode `deleteSelected` bewirkt, dass jedes dritte Element aus der ausführenden Liste erhalten bleibt. Dabei soll das erste Element und dann das 4., 7., 10. usw Element erhalten werden – sofern diese Elemente überhaupt existieren.

```
public class ThirdStrategy<S>
implements DoublyLinkedList.DeletionStrategy<S>
{
    private int remain;

    public ThirdStrategy()
    {
        remain = 3;
    }

    public boolean select( S ref )
    {
        if (remain == 3)
        {
                remain = 1;
                return false;
        }
        else
        {
                remain++;
                return true;
        }
    }
}
```

___ / 6

Aufgabe 7 (Methoden zur Klasse `DoublyLinkedList<T>`)

Ergänzen Sie die aus der Vorlesung bekannte Klasse `DoublyLinkedList<T>`, die Sie im Anhang finden. Bei der Implementierung der geforderten Methoden dürfen **nur** die **im Anhang** aufgeführten Methoden vorausgesetzt und genutzt werden.

a) [4 Punkte] Vervollständigen Sie die Methode `isPalindrome()`.
Die Methode `isPalindrome` soll den Wert `true` zurückgeben, falls die aufrufende Liste ein Palindrom ist, also die Ausgabe der Inhalte vom ersten zum letzten Elemente die gleiche Folge ergibt wie die Ausgabe vom letzten zum ersten Element. Sonst soll `false` zurückgegeben werden. Der Vergleich der Inhalte soll mit der Methode `equals` erfolgen. Gehen Sie davon aus, dass jedes Element der Liste eine Referenz auf ein Objekt – und nicht auf `null` – enthält.
Anmerkung: Eine Liste mit weniger als zwei Elementen ist immer ein Palindrom.

```
public boolean isPalindrome()
{
    Element<T> up = first;
    Element<T> down = last;
    int m = size()/2;
    for ( int i = 0; i < m; i++ )
    {
        if (!up.getContent().equals(down.getContent()))
        {
                return false;
        }

        up = up.getSucc();
        down = down.getPred();

    }
    return true;
}
```

Aufgabe 7 (Methoden zur Klasse `DoublyLinkedList<T>`) – Fortsetzung

b) [6 Punkte] Vervollständigen Sie die Methode `middle( int n )`.
Die Methode `middle` soll eine Liste zurückgeben, die die mittleren `n` Elemente der ausführenden Liste in unveränderter Reihenfolge enthält. Diese Elemente sollen zugleich aus der ausführenden Liste entfernt werden.

- Die Anzahl der in der Liste verbleibende Elemente im Anfangsstück darf um ein Element größer als die Anzahl der Elemente im verbleibenden Endstück sein.
- Besitzt diese Liste beim Aufruf der Methode `middle` nicht **mindestens** `n+2` Elemente, soll eine `IllegalStateException` geworfen werden.

```
public DoublyLinkedList<T> middle( int n )
{
    if ( size() >= n + 2 )
    {
        DoublyLinkedList<T> newList = new DoublyLinkedList<>() ;
        Element<T> current = first;

        int start = (size() - n) / 2;        // Startindex für das mittlere Segment

        // Zum Start des mittleren Segments navigieren
        for (int i = 0; i < start; i++)
        {
                current = current.next;
        }

        // Mittlere Elemente kopieren und aus der Original Liste entfernen
        for (int i = 0; i < n; i++)
        {
                newList.add(current.content);
                Element<T> toDelete = current;
                current = current.next;
                this.remove(toDelete);
        }

        return newList;

    } else {
        throw new IllegalStateException();
    }
}
```

Aufgabe 7 (Methoden zur Klasse `DoublyLinkedList<T>`) – Fortsetzung

c) [8 Punkte] Vervollständigen Sie die Methode `longestSequence()`.
Die Methode `longestSequence` soll die Länge der längsten Teilfolge von unmittelbar aufeinander folgenden gleichen Inhalten zurückgeben. Der Vergleich der Inhalte soll mit der Methode `equals` erfolgen. Für eine leere Liste soll der Wert `0` zurückgegeben werden.

```
public int longestSequence()
{
    int maximum = 0;
    if ( size() > 0 )
    {
        Element<T> current;
        int count = 1;
        maximum = 1;
        current = first.getSucc();
        while ( current != null )
        {
            if (current.getPred().getContent().equals(current.getContent()))
            {
                    count++;
                    if (count > maximum)
                    {
                            maximum = count;
                    }
            }
            else
            {
                    count = 1;
            }

            current = current.getSucc();









        }
    }
    return maximum;
}
```

___ / 18

Aufgabe 8 (Methoden zur Klasse `BinarySearchTree<T implements Comparable<T>>`)

Ergänzen Sie die aus der Vorlesung bekannte Klasse `BinarySearchTree<T extends Comparable<T>>`, die Sie im Anhang finden. Bei der Implementierung der geforderten Methoden dürfen **nur** die **im Anhang** aufgeführten Methoden vorausgesetzt und genutzt genutzt werden.

a) [7 Punkte] Vervollständigen Sie die Methode `checkTree()`.
Die Methode `checkTree` soll die Höhe des Baums zurückgeben, falls sich für jeden einzelnen Knoten die Höhen seines linken und seines rechten Teilbaums höchstens um den Wert `2` unterscheiden. Gibt es mindestens einen Knoten, dessen beide Teilbäume einen größeren Unterschied aufweisen, soll der Wert `-1` zurückgegeben werden.
Ein leerer Baum hat die Höhe `0`, ein Blatt die Höhe `1`.

```
public int checkTree()
{
    if ( isEmpty() )
    {
        return 0;
    }
    else
    {
        int leftHeight = leftChild.checkTree();
        int rightHeight = rightChild.checkTree();

        if (leftHeight == -1 || rightHeight == -1 || Math.abs(leftHeight - rightHeight) > 2)
        {
                return -1;
        }
        else
        {
                return Math.max(leftHeight, rightHeight) + 1;
        }

    }
}
```

Aufgabe 8 (Methoden zur Klasse `BinarySearchTree`) – Fortsetzung

b) [6 Punkte] Vervollständigen Sie die Methode `leavesAtLevel( int n )`.
Die Methode `leavesAtLevel` soll die Anzahl der Blätter auf der Ebene `n` des Baums zurückgeben. Die Wurzel eines Baums liegt immer auf der Ebene `0`. Besitzt der Baum keine Ebene `n`, so existiert auf der gesuchten Ebene kein Knoten und somit auch kein Blatt: Dementsprechend wird `0` zurückgegeben.

```
public int leavesAtLevel( int n )
{
    if ( n < 0 || isEmpty() )
    {
        return 0;
    }
    else
    {
        if (n == 0)
        {
                if (root.isLeaf())
                {
                        return 1;
                }
                else
                {
                        return 0;
                }
        }
        else
        {
                return  root.getLeft().leavesAtLevel(n - 1) +
                        root.getRight().leavesAtLevel(n - 1);
        }
    }
}
```

___ / 13

Aufgabe 9 (Lambda-Ausdrücke)

[9 Punkte] Gegeben sind die Interfaces `BoolBiFunction`, `IntFunction` und die Klasse `IntData`.

```
public interface BoolBiFunction
{
    boolean apply( int p1, int p2 );
}
```

```
public interface IntFunction
{
    int apply( int p );
}
```

```
public class IntData
{
    private int[] intValues;

    public IntData( int[] p ) { intValues = p; }

    public int compute( BoolBiFunction f, IntFunction g, int x )
    {
        int result = 0;
        for ( int value : intValues )
        {
            if ( f.apply( value, x ) )
            {
                result = result + g.apply( value );
            }
        }
        return result;
    }
}
```

Für die folgenden Teilaufgaben gilt: Die Referenz `id` verweist auf ein `IntData`-Objekt.

a) Ergänzen Sie Argumente derart, dass der Aufruf von `compute` die Häufigkeit liefert, mit der ein Wert `w` im Feld `intValues` vorkommt.

```
id.compute( (val, w) -> val == w, val -> 1, w );
```

b) Ergänzen Sie Argumente derart, dass der Aufruf von `compute` die Summe der Quadrate derjenigen Werte des Feldes `intValues` liefert, die größer als ein Wert `w` sind.

```
id.compute( (val, w) -> val > w, val -> val * val, w );
```

c) Ergänzen Sie Argumente derart, dass der Aufruf von `compute` die Summe derjenigen Werte des Feldes `intValues` liefert, die keine Vielfachen eines Wertes `w` sind.

```
id.compute( (val, w) -> val % w != 0, val -> val, w );
```

___ / 9

Anhang – Programmcode der Klasse `BinarySearchTree`

```
public class BinarySearchTree<T implements 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`

```
public class DoublyLinkedList<T> {
    private Element first, last;
    private int size;
    public DoublyLinkedList() { ... }
    public int size() { ... }
    public boolean isEmpty() { ... }
    // strategy
    public static interface DeletionStrategy<S> {
        public abstract boolean select( S ref );
    }
    public void deleteSelected( DeletionStrategy<T> s ) {
        Element current = first;
        while ( current != null ) {
            Element candidate = current;
            current = current.getSucc();
            if ( s.select( candidate.getContent() ) ) { remove( candidate ); }
        }
    }
    private void remove( Element e ) { ... }
    // 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 – Programmcode des Interface `Iterator`

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

Anhang – Programmcode des Interface `Comparable`

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