Michael Henninger
Prof. Horst Veitschegger
Prof. Dr. Wolfgang Weck

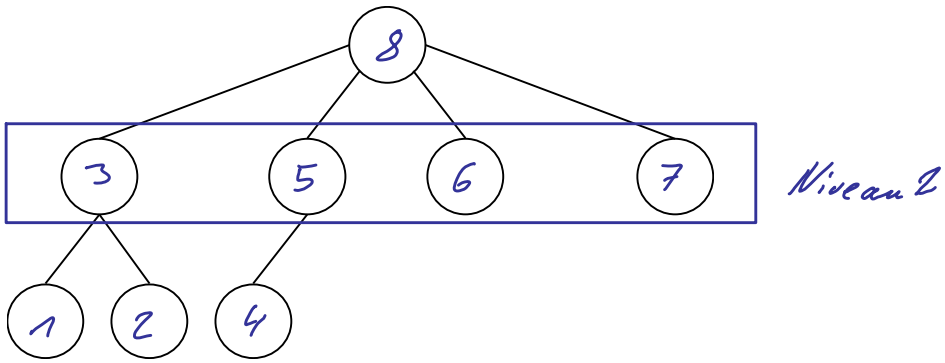
Prüfung vom 9. April 2019
Dauer: 90 Minuten / 40 Punkte

Name, Vorname: _____

Allgemeine Hinweise:
1) Erlaubte Hilfsmittel: Unterlagen in ausgedruckter oder handgeschriebener Form
2) Nicht erlaubte Hilfsmittel: Elektronische Geräte (Handy, Taschenrechner), Kommunikation mit anderen Personen
3) Schreiben Sie die Antworten direkt auf das Aufgabenblatt.
4) Bewertet werden die Korrektheit der Resultate sowie die Herleitung / Begründung.

Viel Erfolg!

Aufgabe 1: Bäume **(3 + 2 = 5 Punkte)**



a) Beantworten Sie die folgenden Fragen zum oben abgebildeten Baum:

1. Schreiben Sie in die Knoten der Zeichnung oben die Zahlen 1 bis 8, so dass diese bei einer *Post-Order-Traversierung* in aufsteigender Reihenfolge besucht / ausgegeben würden.

2. Markieren Sie alle Knoten auf dem Niveau 2.

3. Welches ist die Ordnung des abgebildeten Baums mindestens? Begründen Sie!

   4, weil Wurzel 4 (und kein Knoten mehr) Nachfolger hat

4. Geben Sie die Höhe des Baums an (keine Begründung erforderlich):

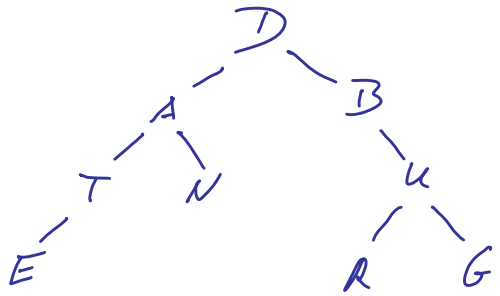
   3

5. Ist der Baum ausgefüllt? Begründen Sie Ihre Antwort.

   Nein, nicht alle inneren Knoten haben 4 Nachfolger

b) Zeichnen Sie einen Binärbaum mit den folgenden Eigenschaften:

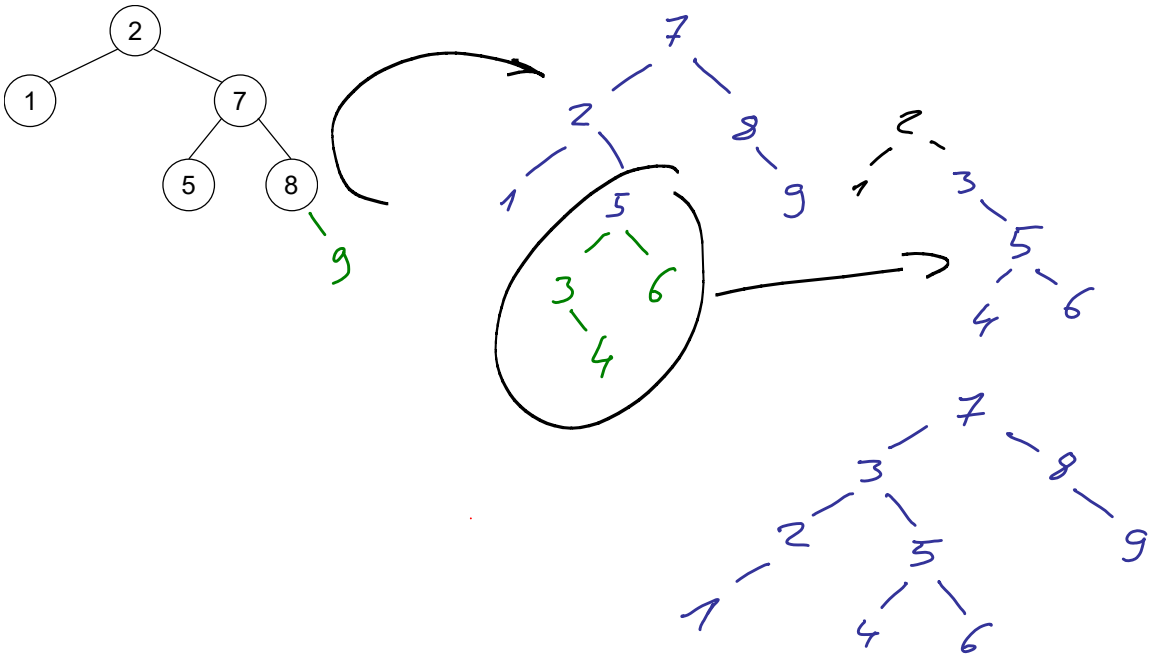
- Jeder Knoten speichert einen Buchstaben.
- Die Preorder-Reihenfolge der Knoten des Baumes ist: DATENBURG
- Die Inorder-Reihenfolge lautet: ETANDBRUG



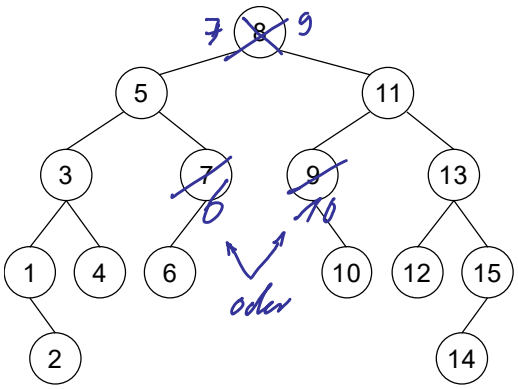
Aufgabe 2: Operationen auf Binären Suchbäumen **(3 + 1 + 1 = 5 Punkte)**

a) In den gezeichneten *AVL-Baum* werden Zahlen Schritt für Schritt eingefügt. Zeichnen Sie die neuen Knoten ein. Machen Sie eine neue Zeichnung, für den Schritt bei dem der Baum von Ausgleichsoperationen verändert wird.

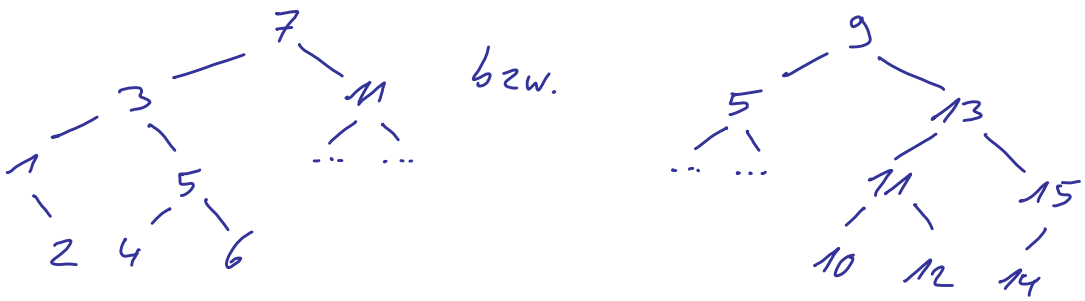
add(9), add(3), add(6), add(4)



b) Betrachten Sie diesen Baum als *einfachen binären Suchbaum* ohne Ausgleichsmechanismus. Wie sieht der Baum aus, wenn der Knoten 8 entfernt wird?



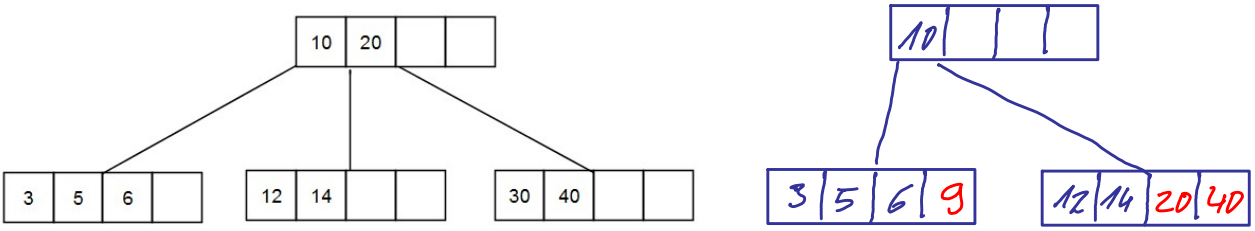
c) Sei derselbe Baum ein *AVL-Baum*. Wie sieht er dann nach dem Entfernen des Knotens 8 und dem Ausgleich aus? (Von gegenüber dem Original völlig unveränderten Teilbäumen brauchen Sie nur die Wurzel



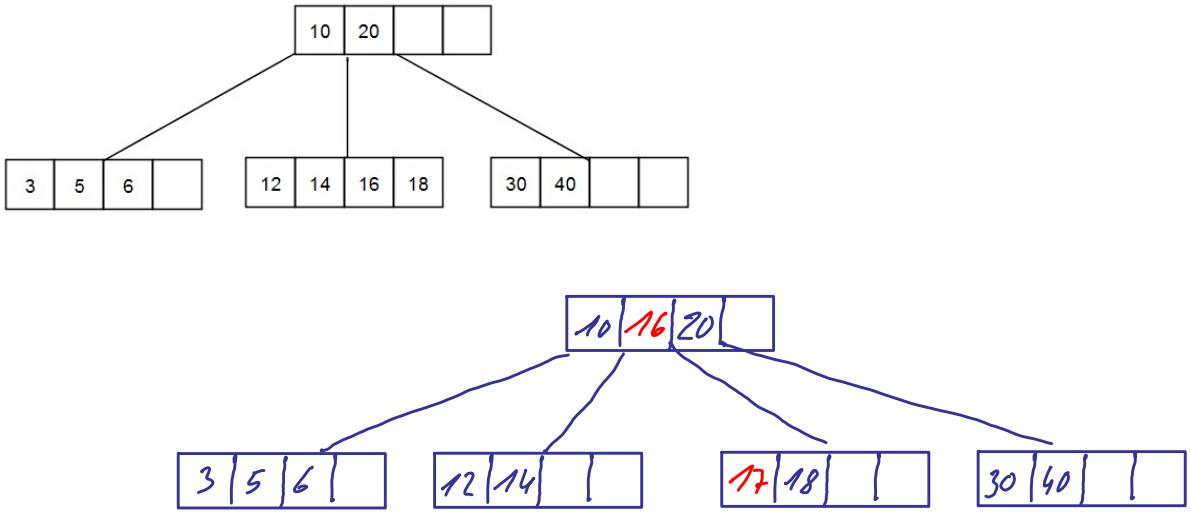
Aufgabe 3: B-Bäume **(4 Punkte)**

Führen Sie die unten aufgeführten Operationen in der angegebenen Reihenfolge auf dem abgebildeten B-Baum aus. Zeichnen Sie den kompletten Baum neu, wenn es der Übersicht dient.

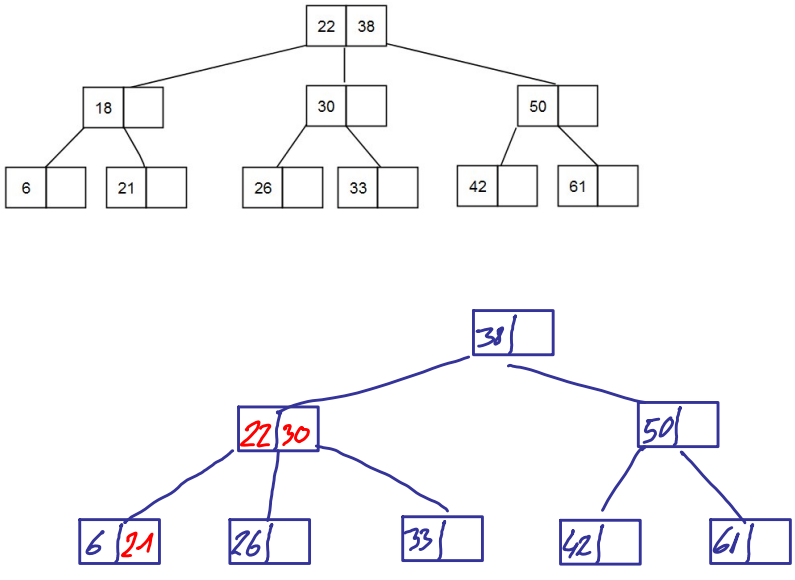
a) B-Baum der **Ordnung 2**: Add(9); Delete(30)



b) B-Baum der **Ordnung 2**: Add (17)



c) B-Baum der **Ordnung 1**: Delete(18)



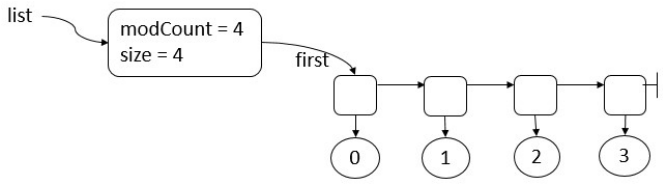
Aufgabe 4: SpecialMethod auf der einfach verketteten Liste **(1 + 2 + 1 = 4 Punkte)**

Sie haben eine einfach verkettete Liste ohne last-Variable. `null` kann nicht als Element in der Liste abgelegt werden:

```
public class SinglyLinkedList<E> implements List<E> {
   private int size = 0, modCount = 0;
   private Node<E> first;
   ...
   private static class Node<E> {
      private final E elem;
      private Node<E> next;
      ...
   }

   public boolean specialMethod(Object elem) {
      Node<E> n = first, p = null;
      while (n != null && !n.elem.equals(elem)) { p = n; n = n.next; }
      if (n != null) {
         if (p != null) {
            p.next = n.next; n.next = first; first = n;
         }
         return true;
      } else return false;
   }
   ...
}
```

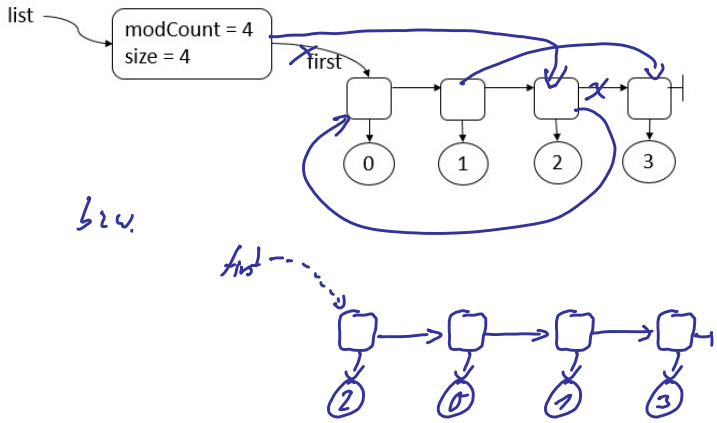
Beantworten Sie folgende Fragen:

a) Gibt der Aufruf `specialMethod(0)` auf der nachfolgenden Liste `true` oder `false` zurück?



true

b) Zeichnen Sie die Liste nachdem darauf `specialMethod(2)` aufgerufen wurde:



c) Die Methode `specialMethod` in der oben stehenden Implementation verändert den Modification Count der Liste nie. Ist das korrekt? Begründen Sie.

modCount sollte verändert werden, da die Struktur der Liste verändert wurde und Iteratoren davon abhängig sein können.

(Dass eine solche Variable eingeführt wurde, lässt vermuten, dass sie auch irgendwo verwendet wird, auch, wenn man das hier nicht direkt sieht.)

Aufgabe 5: Iterator programmieren **(7 Punkte)**

Betrachten wir eine Klasse `ArrayList` mit einem `ListIterator`. Der `ListIterator` ist als innere Klasse von `ArrayList` implmentiert und kann deswegen direkt auf alle Instanzvariablen der Liste zugreifen.

Die Methoden der `ArrayList` interessieren uns hier nicht, nur die des `ListIterators` (wobei wir zur Vereinfachung einige weggelassen haben). Vervollständigen Sie die `ListIterator`-Methoden `hasPrevious`, `previous`, `next`, `remove` und die private Methode `isValid`. Sie dürfen dem `ListIterator` auch weitere Instanzvariablen hinzufügen:

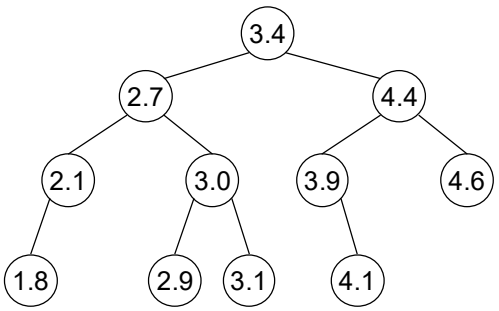
```
public class ArrayList<E> implements List<E> {
   private int size = 0
   private int modCount = 0;
   private final E[] data;
...
   public ListIterator<E> listIterator() { return new MyListIterator(); }

   private class MyListIterator implements ListIterator<E> {
      private int mcount = modCount;
      private int next = 0;
      private int toRemove = -1;


      private boolean isValid() {
         return mcount == modCount;

      }

      public boolean hasNext() {
         if (!isValid()) throw new ConcurrentModificationException();
         return next < size;
      }

      public boolean hasPrevious() {
         if (!isValid()) throw new ConcurrentModificationException();
         return next > 0;

      }

      public E next() {
         if (!isValid()) throw new ConcurrentModificationException();
         if (!hasNext()) throw new NoSuchElementException();

         toRemove = next;

         return data[next++];


      }
```

```
        public E previous() {
           if (!isValid()) throw new ConcurrentModificationException();
           if (!hasPrevious()) throw new NoSuchElementException();

           toRemove = --next;
           return data[next];



        }

        public void remove() {
           if (!isValid()) throw new ConcurrentModificationException();
           if (toRemove == -1) throw new IllegalStateException();

           for(int i = toRemove; i < size-1; i++){
                 data[i] = data[i+1];
           }
           data[size - 1] = null;

           size--;
           if (next > toRemove) next--;

           toRemove = -1;
           mcount++;
           modCount++;


        }
     }
  }
```

Aufgabe 6: Kleinstes Element über Schranke **(2 + 3 + 1 = 6 Punkte)**

In einem Binären Suchbaum seien Fliesskommazahlenwerte gespeichert. Gesucht ist der Knoten, der den kleinsten Wert *über* einer gegebenen Schranke enthält (also nie diese Zahl selbst). Wenn es keine solche Zahl gibt, sei das Ergebnis `null`.

Im nebenan dargestellten Baum ist der gesuchte Knoten für die Schranke *3.5* der mit dem Wert *3.9* und der für die Schranke *2.7* der Knoten mit dem Wert *2.9*.




```
class NumberTree {
   private class Node {
      final double x;
      Node left, right;
      ...
   }
   ...
```

a) Beschreiben Sie kurz in Worten, wo zu einer gegebenen Schranke `limit` der entsprechende Knoten in einem (Teil-)Baum mit der Wurzel `root` gesucht werden muss, wenn

i. `root.x < limit`

im rechten Teilbaum von root: root.right

ii. `root.x = limit`

im rechten Teilbaum von root: root.right

iii. `root.x > limit`

Wenn der linke Teilbaum von root ein passendes Element enthält, verwende dieses, sonst ist root.x die Lösung.

b) Programmieren Sie die Methode `Node smallestOver(Node root, double limit)` zur Lösung der oben beschriebenen Suche:

```
Node smallestOver(Node root, double limit) {
    if (root == null) return null;
    else if (root.x <= limit) return smallestOver(tree.right, limit);
    else {
       Node n = smallestOver(tree.left, limit);
       if (n != null) return n; else return root;
    }

}
```

c) Was ist die asymptotische Komplexität Ihrer Lösung bei b), wenn der Baum ein AVL-Baum und N die Anzahl der Knoten in `root` ist (keine Begründung erforderlich)?

$O(\log n)$

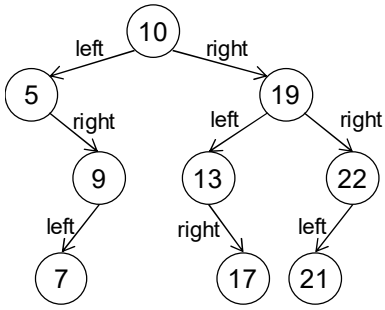
alternative Lösung zu 65)

```
Node smallestOver (Node root, double limit) {
    Node res = null;
    while (root != null) {
        if (root.x <= limit) root = root.right;
        else { res = root; root = root.left; }
    }
    return res;
}
```

Aufgabe 7: Baum zu Liste umformen **(2 + 7 = 9 Punkte)**

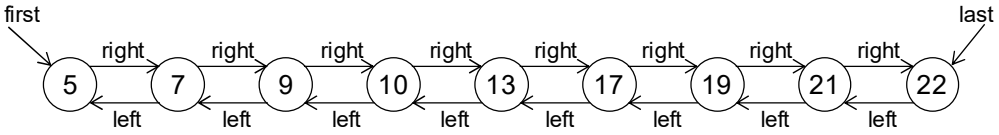
Objekte der Klasse `Node` kann man sowohl zum Speichern binärer Suchbäume als auch für doppelt verkettete Listen benutzen:

```
class Node {
   final int x;
   Node left, right;
   Node (int x) { this.x = x; }
}
```



In einem Baum verweist `left` auf den Teilbaum mit kleineren Werten und `right` auf den Teilbaum mit grösseren Werten.

In einer doppelt verketteten Liste verweist `left` auf den Vorgänger und `right` auf den Nachfolger.



Für doppelt verkettete Listen verwenden wir Objekte der Klasse `List`, die je eine Referenz auf das erste und das letzte Element enthalten.

```
class List {
   Node first, last;
   List(Node first, Node last) { this.first = first; this.last = last; }
}
```

a) Programmieren Sie eine Java-Methode, die an eine Liste `l1` eine zweite Liste `l2` hinten anhängt. Sie dürfen annehmen, dass die Methode nur aufgerufen wird, wenn beide Listen je mindestens ein Element enthalten. Die als Parameter übergebenen Listen-Objekte werden später nicht mehr verwendet.

```
List concat(List l1, List l2) {
    l1.last.right = l2.first;
    l2.first.left = l1.last;
    return new List(l1.first, l2.last);


}
```

b) Programmieren Sie in Java eine Methode `treeToList`, die einen binären Suchbaum in eine aufsteigend geordnete doppelt verkette Liste transformiert. Dabei sollen die `Node`-Objekte des Baums verwendet werden, d.h. es dürfen keine neuen `Node`-Objekte erzeugt werden.

Versuchen Sie mit möglichst wenig temporärem Zusatzspeicher auszukommen.

```
List treeToList(Node tree) {
    return tree != null ? toList(tree) : null;
}
List toList(Node tree) {    // tree ≠ null
    List l = new List(tree, tree);
    if (tree.left != null) l = concat(toList(tree.left), l);
    if (tree.right != null) l = concat(l, toList(tree.right));
    return l;
}
```

alternative Lösung (tatsächlich Speicher-ökonomisch):

```
List treeToList(Node tree) {
    List l = new List(null, null);
    inorder(tree, l);
    return l;
}
void inorder(Node tree, List l) {
    if (tree != null) {
        inorder(tree.left, l);
        if (l.last != null) l.last.right = tree; else l.first = tree;
        tree.left = l.last;
        l.last = tree;
        inorder(tree.right, l);
    }
}
```