Lösung zur Prüfung vom 30. Oktober 2020
Dauer: 90 Minuten / 40 Punkte

Aufgabe 1: MoveToFront-Liste **(6 + 4 = 10 Punkte)**

Werden Objekte in einer einfach verketteten Liste unsortiert und mit Set-Semantik gespeichert und deutlich unterschiedlich oft gesucht, lohnt es sich, gesuchte Objekte jeweils an den Anfang der Liste zu verschieben. Eine neue Suche nach dem gleichen Objekt führt dann wesentlich schneller zum Erfolg. Oft gesuchte Objekte versammeln sich so mit der Zeit am Anfang der Liste, selten oder nie gesuchte Objekte am Ende.

Beispiel: Auf einer Liste mit den Werten 1 -> 5 -> 29 -> 14 -> 8 wird *contains(14)* aufgerufen. Dann enthält die Liste anschliessend die Werte in folgender Reihenfolge 14 -> 1 -> 5 -> 29 -> 8.

In dieser Aufgabe sollen Sie dieses Move-to-front-Verfahren auf zwei Arten implementieren.

a) Implementieren Sie für folgende Klasse `MyList` die Methode `contains(E elem)` entsprechend dem oben beschriebenen Verfahren. Dafür sollen Sie die Node-Objekte direkt manipulieren und alle Instanzvariablen der Klasse `MyList` wenn nötig aktualisieren. Sie dürfen also keine Ihnen bekannten Methoden des List-Interfaces wiederverwenden. Die Liste soll auch den Wert *null* enthalten können.

```
public class MyList<E> {
   private int size = 0;
   private Node<E> first = null;
   private int modCount = 0;

   private static class Node<E> {
      E elem;
      Node<E> next;
   }

   // returns true, if elem can be found in this list
   // if elem can be found, elem will move to first position afterwards
   public boolean contains(E elem) {
      Node<E> n = first, p = null;
      while (n != null && !Objects.equals(n.elem,elem)) {
         p = n;
         n = n.next;
      }
      if (n != null) {
         if (p != null) {
            p.next = n.next;
            n.next = first;
            first = n;
            modificationCount++;
         }
         return true;
      } else return false;
   }
}
```

b) Programmieren Sie nun eine möglichst effiziente Methode `boolean contains(List<E> l, E elem)`, die das obige Verfahren mit beliebigen Listen auf der Basis der folgenden Interfaces mit den Ihnen bekannten Methoden umsetzt:

```
public interface List<E> {
   public boolean add(E elem);
   public void add(int index, E elem);
   public boolean contains(E elem);
   public boolean remove(E elem);
   public E remove(int index);
   public Iterator<E> iterator();
   ...
}
```

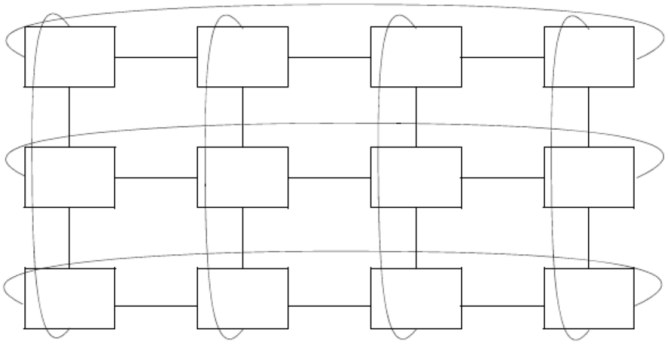
```
public interface Iterator<E> {
   public boolean hasNext();
   public E next();
   public void remove();
}
```

Die Liste soll auch den Wert *null* enthalten können.

```
  public boolean contains(List<E> l, E elem) {
     Iterator<E> it = l.iterator();
     E e = null;
     boolean found = false;
     while (it.hasNext() && !found) {
        e = it.next();
        found = (e == elem || (e != null && e.equals(elem));
     }
     if (found) {
        it.remove();
        l.add(0, e);
     }
     return found;
  }

Oder

  public boolean contains2(List<E> l, E elem) {
     Iterator<E> it = l.iterator();
     while (it.hasNext()) {
        E e = it.next();
        if (e == elem || (e != null && e.equals(elem))) {
           it.remove();
           l.add(0, e);
           return true;
        }
     }
     return false;
  }

Oder

  public boolean contains3(List<E> l, E elem) {
     if (l.remove(elem)) {
        l.add(0, elem);
        return true;
     } else return false;
  }
```

Aufgabe 2: Grid Iterator **(7 Punkte)**

Doppelt verkettete Ringlisten kann man auch zweidimensional einsetzen und damit ein *Grid* aufspannen:



Diese Datenstruktur ist aus Elementen des folgenden Typs aufgebaut:

```
private class Element<T> {
    T data;
    Element<T>  right;
    Element<T>  left;
    Element<T>  up;
    Element<T>  down;
}
```

Die einzelnen Zeilen und Spalten besitzen kein Ende; die jeweils letzten Elemente sind immer wieder mit dem ersten Element verbunden. In der Klasse `CircularGrid` zeigt eine Referenz `anchor` auf ein beliebiges Element im Grid. Ihre Aufgabe ist es nun (auf der nächsten Seite), für die Klasse `CircularGrid` einen internen Iterator zu implementieren, der

- zeilenweise durch die Elemente im `CircularGrid` iteriert,
- eine ConcurrentModificationException wirft, falls der Iterator ungültig wird,
- eine IllegalStateException wirft, falls next() kein Element mehr zurückgeben kann.

```
public class CircularGrid<T> {
    Element<T> anchor;
    long modCount = 0;

    private class Element<T> {...}
    ...

    public Iterator iterator() { return new CircularGridIterator(); }
    private class CircularGridIterator implements Iterator {

        long expectedModCount=modCount;
        Element<T> next = anchor;
        Element<T> rowstart = next;

        @Override
        public boolean hasNext() {

            return next!= null;

        }

        @Override
        public T next() {

            if (expectedModCount != modCount) {
                throw new ConcurrentModificationException();
            }
            if (next == null) {
                throw new IllegalStateException(); // or NoSuchElementException()
            }
            T data = next.data;
            if(next.right != rowstart) next = next.right;
            else if (rowstart.down != anchor) {
                rowstart = rowstart.down;
                next = rowstart;
            }
            else next = null;
            return data;

        }
    }
}
```

Aufgabe 3: Wahl der geeignetsten Datenstruktur **(3 Punkte)**

Entscheiden und begründen Sie für die untenstehenden Situationen, welche der verfügbaren Datenstrukturen verwendet werden sollen. Zur Verfügung stehen:

A: Liste, einfach verkettet, sortiert, Bag-Semantik
B: Liste, doppelt verkettet, unsortiert, Bag-Semantik
C: Binärer Suchbaum, Bag-Semantik
D: AVL-Baum, Set-Semantik

Situationen:

a) Sie wollen eine Applikation implementieren, die in einer Sammlung von Dokumenten nach Personennamen sucht. Diese Personenamen sollen in einer Datenstruktur gespeichert werden. Am Ende werden alle erkannten Namen auf der Konsole ausgegeben. Jeder Name darf maximal 1x erscheinen. In welcher der Datenstrukturen A bis D speichern Sie die Namen und weshalb?

D: Set-Semantik

b) Die Minigolfanlage Dägerli in Windisch speichert von jedem Spieler das Total der benötigten Schläge für alle Bahnen. Am Ende des Tages werden diese Daten auf verschiedene Arten ausgewertet. Beispielsweise wird berechnet, wie viele Schläge im Durchschnitt benötigt wurden. Welche der Datenstrukturen A bis D verwenden Sie, um die Anzahl Schläge aller Personen abzulegen und weshalb?

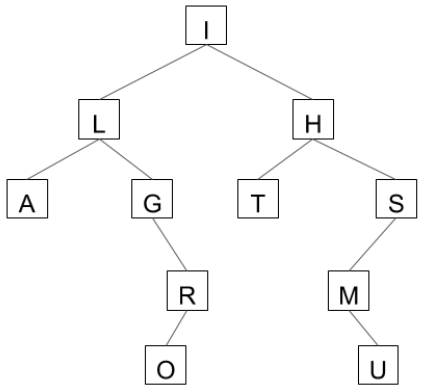
B: Schnelles speichern, am Ende müssen sowieso alle durchlaufen werden (1 Punkt)
C: Im Schnitt schnelles speichern, aber nicht garantiert O(n) iterieren (0.5 Punkte)

c) Sie implementieren eine eigene Queue-Klasse. Welche der Datenstrukturen A bis D verwenden Sie und weshalb?

B: Behält Einfüge-Reihenfolge

Aufgabe 4: Bäume **(2 + 2 + 3 + 1 = 8 Punkte)**

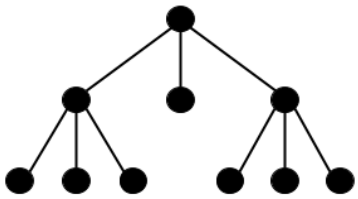
a) Geben Sie für den folgenden Binärbaum die Preorder- und Postorder-Reihenfolge an.



Preorder: I L A G R O H T S M U

Postorder: A O R G L T U M S H I

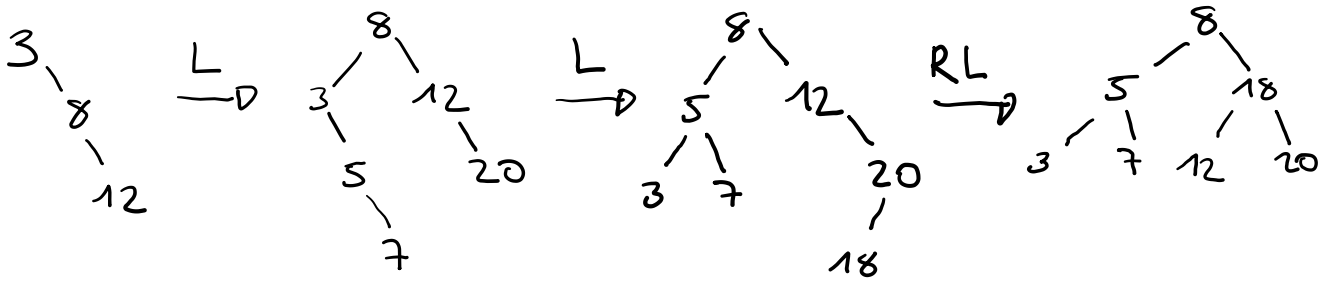
b) Beantworten Sie die untenstehenden Fragen für den folgenden Baum der Ordnung 3.



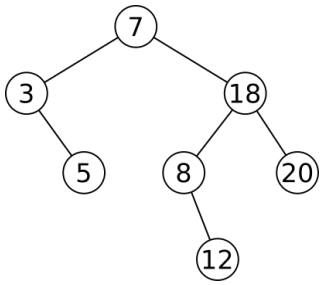
Was ist die Höhe dieses Baumes?	3
Ist dieser Baum vollständig?	Nein
Ist dieser Baum ausgefüllt?	Ja
Wie viele innere Knoten hat dieser Baum?	3

c) Fügen Sie die folgenden Schlüssel nacheinander in einen leeren AVL-Baum ein. Zeichnen Sie den Baum mindestens nach jeder Rotation neu und geben Sie bei jeder Rotation an, um welchen Rotationstyp (z.B. R, L, LR, RL) es sich handelt.

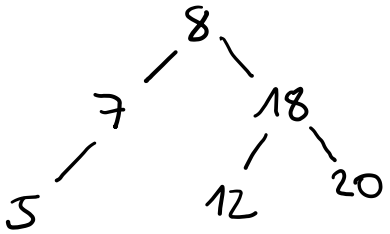
3, 8, 12, 5, 20, 7, 18



d) Löschen Sie aus dem folgenden AVL-Baum den Knoten mit dem Schlüssel 3.



Löschen 3:



Aufgabe 5: AVL-Baum-Check **(5 + 2 = 7 Punkte)**

Damit ein binärer Suchbaum ein AVL-Baum sein kann, müssen seine Knoten balanciert sein. Sie werden eine rekursive Methode int isAVL implementieren, mit der diese Bedingung überprüft werden kann. Dafür verwenden Sie die folgende Klassendefinition von Node:

```
class Node< K extends Comparable<? Super K>> {
      Node<K> left, right;
      K key;
}
```

Die Methode `int isAVL(Node n)` soll die Höhe des (Teil-)baums mit Wurzel `n` zurückgeben, wenn es sich um einen balancierten Baum handelt. Ist der Baum nicht balanciert, soll -1 zurückgeliefert werden.

a) Programmieren Sie die Methode `int isAVL(Node n).` Denken Sie daran, dass ein Baum leer sein kann!

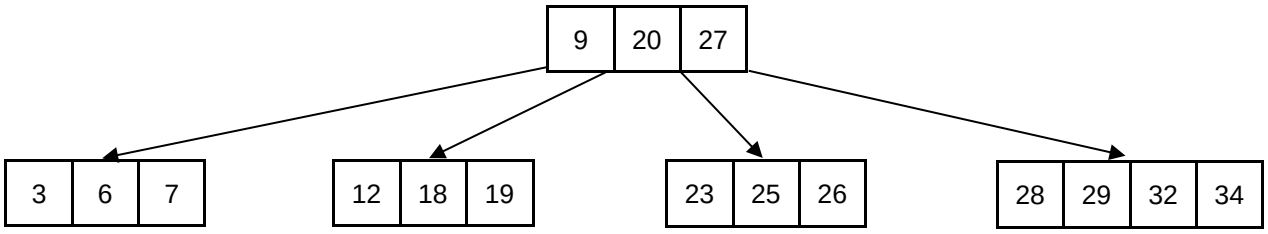
```
int isAVL(Node n){
     if (n==null) return 0;
     int l = isAVL(n.left);
     int r = isAVL(n.right);
     if (l==-1 || r==-1) return -1;
     int diff = r-l;
     if(diff<-1 || diff>1) return -1;
     return Math.max(r,l) + 1;
}
```

b) Welche Laufzeit hat Ihre Methode `isAVL`? Begründen Sie Ihre Antwort!

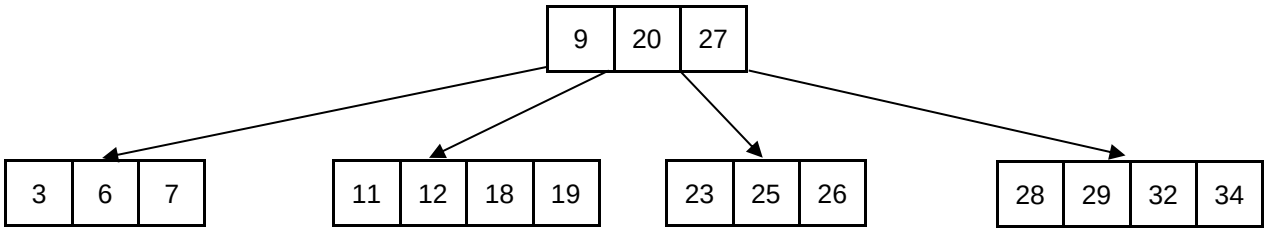
O(n), weil jeder Knoten genau 1x besucht wird und der Aufwand pro Knoten konstant ist.

Aufgabe 6: B-Bäume **(3 + 2 = 5 Punkte)**

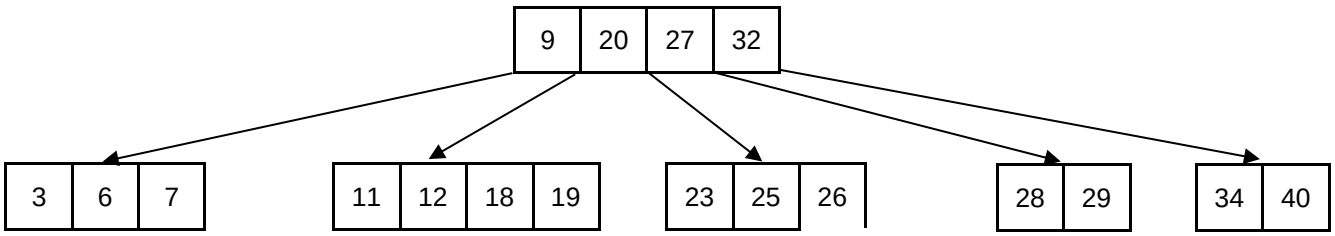
a) Fügen Sie in den nachfolgenden B-Baum der Ordnung n = 2 die Schlüssel 11, 40 und 15 nacheinander ein. Zeichnen Sie nach jedem Einfügen den daraus resultierenden B-Baum.



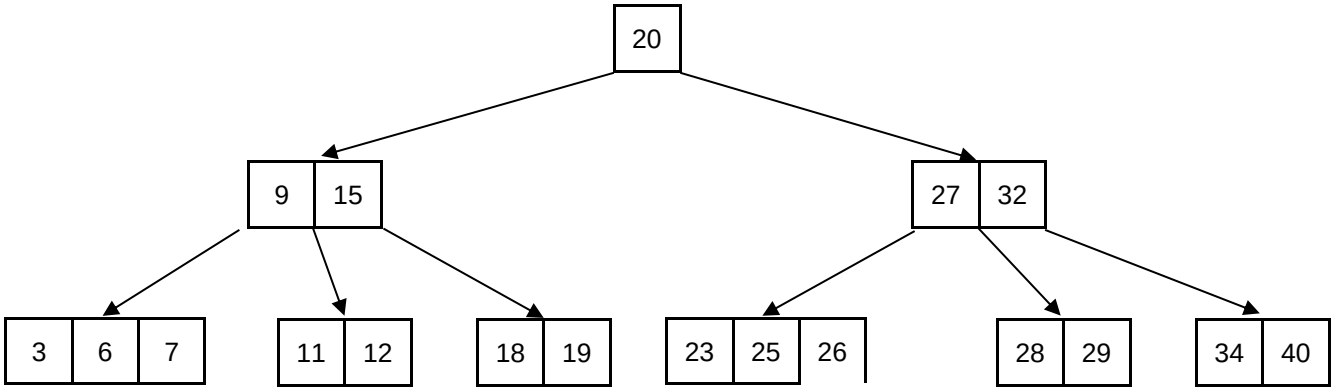
Einfügen 11:



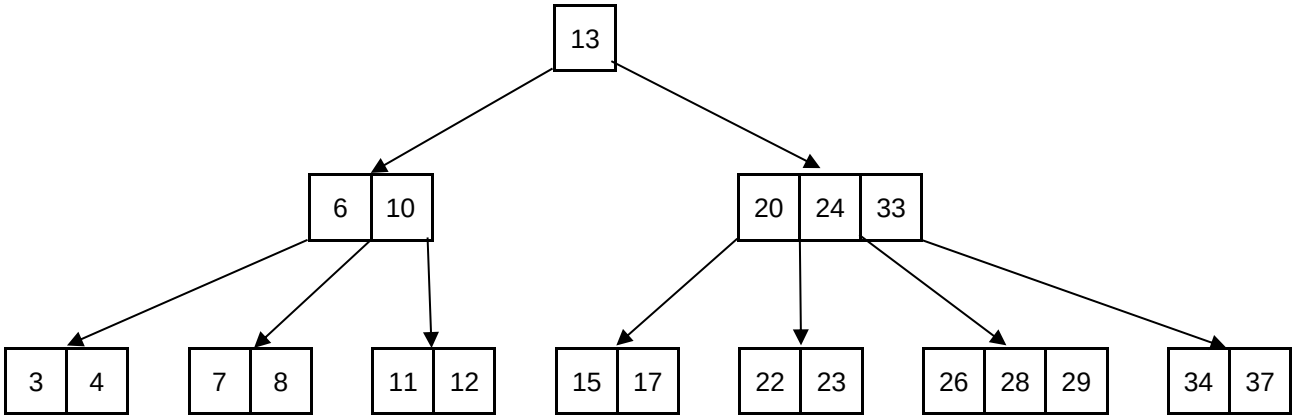
Einfügen 40:



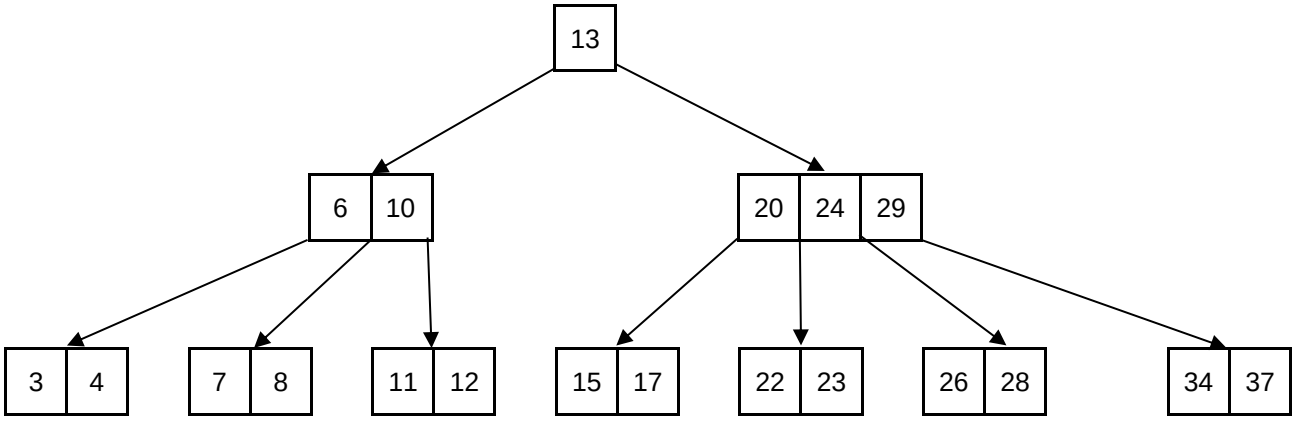
Einfügen 15:



b) Löschen Sie aus dem folgenden B-Baum der Ordnung n = 2 nacheinander die Elemente 33 und 10 und zeichnen Sie nach jedem Löschen den daraus resultierenden B-Baum.



Löschen 33:



Löschen 10: