Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
M.Sc. André Kless

Programmierung 2

Klausur + Lösung

Bevor Sie mit der Bearbeitung der Klausur beginnen, beachten Sie bitte folgende Hinweise:

1. Prüfen Sie die **Vollständigkeit** Ihres Exemplars. Jede Klausur umfasst
   - diese Hinweise und
   - 9 Aufgaben- und Lösungsblätter auf den Seiten 1 bis 9

   Bei Unvollständigkeit wenden Sie sich bitte **sofort** an die Aufsichtsperson.
2. Tragen Sie **auf jedem Lösungsblatt** oben an den vorgesehenen Stellen Ihren **Namen** und Ihre **Matrikelnummer** ein. Blätter ohne diese Angaben werden nicht bewertet. **Füllen Sie aus und unterschreiben** Sie das Deckblatt.
3. Hinter den Aufgaben ist jeweils hinreichend Platz für die Lösungen frei gelassen. Reicht der Platz nicht aus, benutzen Sie die **Rückseiten**, wobei die Zuordnungen von Lösungen zu den Aufgaben deutlich erkennbar sein müssen. Sollten Sie darüber hinaus noch Platz/Papier benötigen, so melden Sie sich bei der Aufsicht. Es darf kein eigenes Papier verwendet werden.
4. Hilfsmittel, die nicht im Prüfungsplan explizit angegeben wurden, sind nicht erlaubt.
5. In Programmierlösungen ist ausschließlich die Verwendung der Programmierkonstrukte erlaubt, die in der Veranstaltung eingeführt wurden.
6. Geben Sie das Deckblatt, die Seiten 1-9 mit den Aufgaben und Ihren Lösungen, ggfs. die **nummerierten** Zusatzblätter (10, 11, usw.) geordnet ab.
7. Sie haben die Klausur bestanden, wenn Sie mindestens 60 Punkte erreichen.

Ergebnis (bitte freilassen):

Aufgabe	**1**	**2**	**3**	**4**	**5**	∑
Erreichbar	20	20	30	25	25	120
Erreicht						

Note: _____

Note	5,0	4,0	3,7	3,3	3,0	2,7	2,3	2,0	1,7	1,3	1,0
Punkte	0-59	60-65	66-71	72-77	78-83	84-89	90-95	96-101	102-107	108-113	114-120

Aufgabe 1 (20 Punkte):

Kreuzen Sie in der folgenden Tabelle die korrekte Laufzeitkomplexität für die einzelnen Operationen der im Semesterverlauf behandelten Datenstrukturen an:

Für ein korrektes Kreuz gibt es 1 Punkt dazu, für ein falsches Kreuz gibt es 1 Punkt Abzug. Insgesamt können nicht weniger als 0 Punkte erreicht werden. Pro Zeile immer nur eine Antwort ankreuzen.

Datenstruktur	**Operation**	**O(1)**	**O($\log_2$ n)**	**O(n)**
DynArray	T get(int pos)	☒	☐	☐
"	void set(int pos, T e)	☒	☐	☐
"	void append(T e)	☐	☐	☒
"	void remove(int pos)	☐	☐	☒
"	boolean contains(T e)	☐	☐	☒
Ringpuffer	T get()	☒	☐	☐
"	void add(T e)	☒	☐	☐
"	void remove()	☒	☐	☐
"	boolean contains(T e)	☐	☐	☒
DVL	T getLast()	☒	☐	☐
"	void insert(int pos, T v)	☐	☐	☒
"	void removeLast()	☒	☐	☐
"	void delete(T v)	☐	☐	☒
"	boolean contains(T v)	☐	☐	☒
AVL-Baum	void insert(T v)	☐	☒	☐
"	void delete(T v)	☐	☒	☐
"	boolean contains(T v)	☐	☒	☐
Heap	T getMin()	☒	☐	☐
"	void insert(T v)	☐	☒	☐
"	void removeMin()	☐	☒	☐

Aufgabe 2 (20 Punkte):

a) Wir betrachten folgende Variante eines Integer-Ringpuffers mit capacity = 3:

- **add() überschreibt** den ältesten Eintrag, falls der Puffer voll ist
- **get()** liefert den **jüngsten** gepufferten Wert
- **remove()** löscht den **ältesten** Wert im Ringpuffer
- der **Iterator** liefert die Elemente in der Reihenfolge **"jüngster", "zweitjüngster",** …

Wie entwickelt sich im Speicher das Array des Ringpuffers unter folgenden Operationen?

add(1); add(2); remove(); remove(); add(5); add(3); add(1); add(4); add(7); remove();

Geben Sie hierzu in der Tabelle die Werte von `p`, `size` und die der internen Array-Inhalte an. Lassen Sie dabei nicht mehr gebrauchte Feldkomponenten unverändert, solange sie nicht überschrieben werden. `p` ist der Index des **ältesten** Eintrags im Ringpuffer.

Anweisung	**p**	**size**	**Array-Inhalte**
	0	0	0 0 0
add(1)	0	1	**1** 0 0
add(2)	0	2	**1 2** 0
remove()	1	1	1 **2** 0
remove()	2	0	1 2 0
add(5)	2	1	1 2 **5**
add(3)	2	2	**3** 2 **5**
add(1)	2	3	**3 1 5**
add(4)	0	3	**3 1 4**
add(7)	1	3	**7 1 4**
remove()	2	2	**7** 1 **4**

b) In welcher Reihenfolge würde nun der Iterator des Ringpuffers welche Werte liefern?

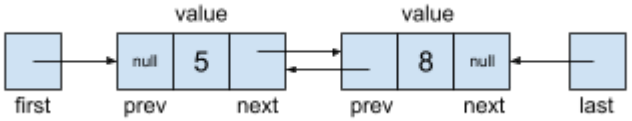
7 4

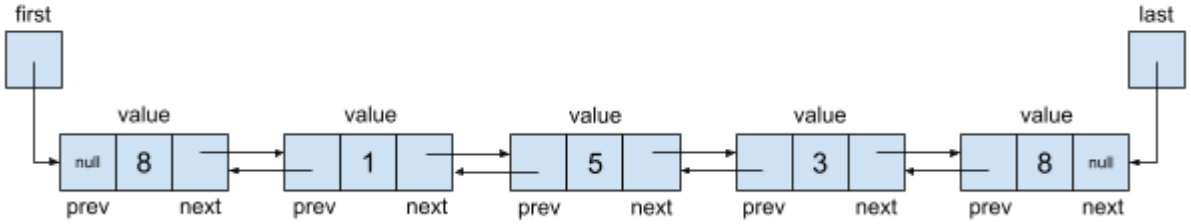
Aufgabe 3 (30 Punkte):

a) Gegeben sei folgendes Java-Programm:

```
public class Main {
    public static void main(String[] args) {
        DVL<Integer> dvl = new DVL<>();
        dvl.append(5); dvl.append(8);
        // Vorher
        dvl.addAfter(0,8); dvl.addAfter(5,3); dvl.addAfter(8,1);
        // Nachher
    }
}
```

Zeichnen Sie insgesamt zwei Speicherbilder: Ein Speicherbild, welches zunächst den Zustand der doppelt verketteten Liste (DVL) vor dem Aufruf der `addAfter`-Methode zeigt und ein Speicherbild für den Zustand danach. Die Methode `addAfter(T a, T b)` fügt den Wert `b` hinter dem ersten Vorkommen von `a` in die Liste ein. Wenn `a` in der DVL nicht enthalten ist, wird `b` am Anfang der Liste eingefügt.





b) Gegeben sei der folgende Java-Code der DVL:

```
public class DVL<T> {

    /* Instanzvariablen */

    private ListenElem first;
    private ListenElem last;
    private int size;

    /* Instanzmethoden */

    public int size() { … }
    public boolean isEmpty() { … }
    public T getFirst() { … }
    public  T getLast() { … }
    public void insert(T v) { … }
    public void append(T v) { … }
    public void removeFirst() { … }
    public void removeLast() { … }
    public void delete(T value) { … }
    public boolean contains(T value) { … }

    /* Innere Klasse */

    private class ListenElem {

        /* Instanzvariablen */
        T value;
        ListenElem prev;
        ListenElem next;

        /* Konstruktor */
        ListenElem (T v) { value = v; }
    }
}
```

Ergänzen Sie nun die Instanzmethode `addAfter(T a, T b)`. Denken Sie auch an den Sonderfall, wenn `a` am Ende der Liste ist. Sie können davon ausgehen, dass `a` und `b` nicht `null` sind. Zeigen Sie, dass Sie das “Umhängen” von Referenzen in einer DVL beherrschen.

```
public void addAfter(T a, T b) {

// a nicht enthalten? → b vorne einfügen
if (!contains(a)) { insert(b); return; }

// ermittle Listenelement von a
ListenElem la = first;
while (!la.value.equals(a) && la.next != null) {
   la = la.next;
}

// a am Ende der Liste? → b hinten einfügen
if (la.equals(last)) { append(b); return; }

// erstelle Listenelement für b
ListenElem lb = new ListenElem(b);

// setze prev und next für lb
lb.prev = la;
lb.next = la.next;

// setze next von la und prev vom Nachfolger von lb
la.next = lb;
lb.next.prev = lb;

// size erhöhen
size++;









}
```

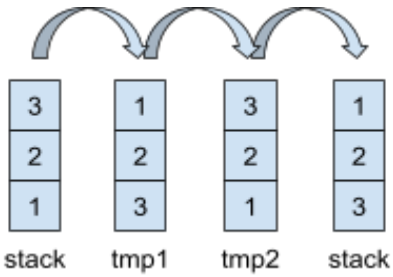
Aufgabe 4 (25 Punkte):

a) Gegeben sei folgendes Java-Programm:

```
public class StapelUtil {
    public static void main(String[] args) {
        Stapel<Integer> stack = new StapelEVL<>();
        stack.push(1);
        stack.push(2);
        stack.push(3);
        System.out.println("Vorher: " + stack);  // [1, 2, 3]
        reverseStack(stack);
        System.out.println("Nachher: " + stack);  // [3, 2, 1]
    }
}
```

```
/* Verwenden Sie folgende
Definition des Interfaces: */
interface Stapel<T> {
    int size();
    boolean isEmpty();
    T top();
    void push(T e);
    void pop();
}
```

Implementieren Sie die statische generische Methode `reverseStack`, die einen Stapel übergeben bekommt und die Reihenfolge der Elemente umkehrt. Verwenden Sie dafür zwei weitere Hilfsstacks:




```
public static <T> void reverseStack(Stapel<T> stack) {
    Stapel<T> tmp1 = new StapelEVL<>();
    Stapel<T> tmp2 = new StapelEVL<>();
    // kopiere alle Elemente von stack nach tmp1
    while (!stack.isEmpty()) {
        tmp1.push(stack.top());
        stack.pop();
    }
    // kopiere alle Elemente von tmp1 nach tmp2
    while (!tmp1.isEmpty()) {
        tmp2.push(tmp1.top());
        tmp1.pop();
    }
    // kopiere alle Elemente von tmp2 nach stack
    while (!tmp2.isEmpty()) {
        stack.push(tmp2.top());
        tmp2.pop();
    }
}
```

b) Gegeben sei folgendes Java-Programm:

```
public class SchlangeUtil {
    public static void main(String[] args) {
        Schlange<Integer> queue =
            new SchlangeEVL<>();
        queue.enqueue(1);
        queue.enqueue(2);
        queue.enqueue(3);
        Integer max = findMax(queue);
        System.out.println(max); // 3
    }
}
```

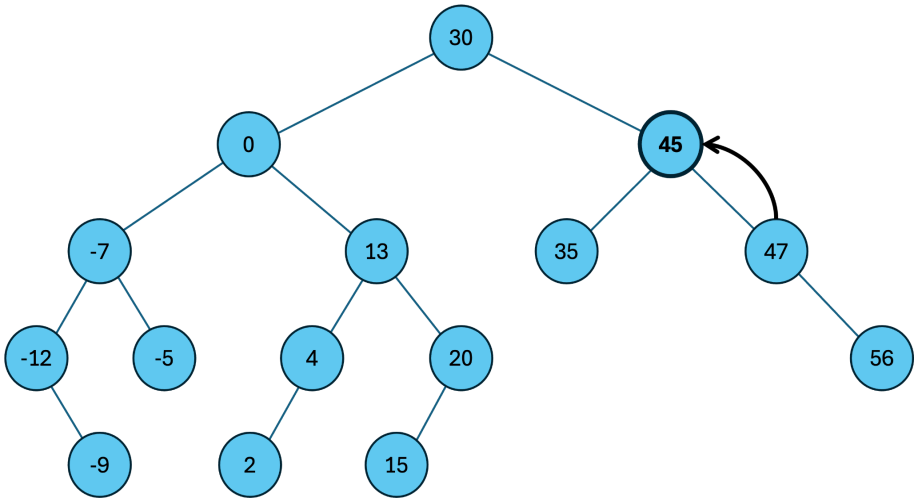
```
/* Verwenden Sie folgende
Definition des Interfaces: */
interface Schlange<T>
extends Iterable<T> {
    int size();
    boolean isEmpty();
    T front();
    void enqueue(T e);
    void dequeue();
}
```

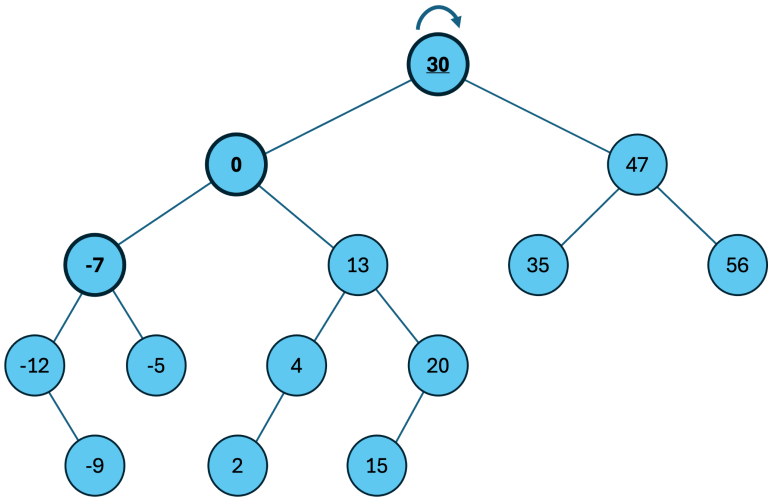
Implementieren Sie die statische generische Methode `findMax`, die eine Warteschlange übergeben bekommt und das größte darin enthaltene Element in der Warteschlange liefert, **ohne** dabei die Warteschlange zu verändern. Nehmen Sie zudem eine Typeinschränkung vor, um die Elemente in dieser Warteschlange über ihre innere Ordnung vergleichen zu können. Die `findMax`-Methode soll `null` liefern, falls die Warteschlange leer ist.

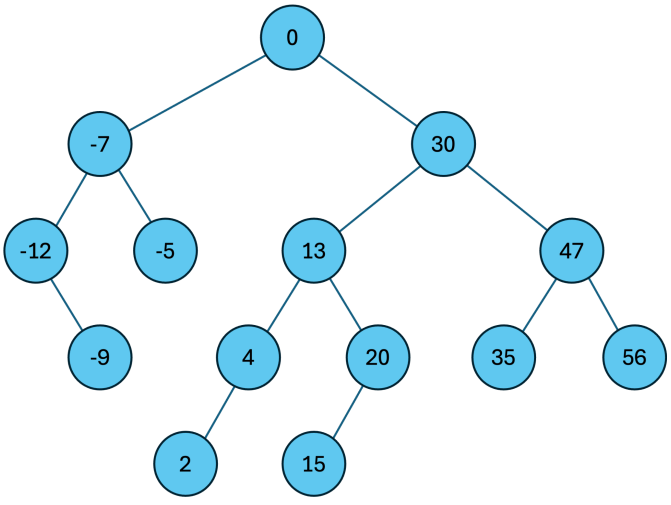
```
public static <T extends Comparable<T>> T
      findMaxElement(Schlange<T> queue) {

   // Leere Warteschlange? → Methode liefert null
   if (queue.isEmpty()) return null;

   // erstes Element als Startwert
   T max = queue.front();

   // alle Elemente der Warteschlange durchlaufen
   for (T element : queue) {
       if (element.compareTo(max) > 0) {
           max = element;
       }
   }

   return max;
}
```

Aufgabe 5 (25 Punkte):

a) Löschen Sie aus dem dargestellten AVL-Baum das Element 45. Führen Sie ggf. notwendige Rebalancierungen durch. Markieren Sie **vor** jeder Rotation den Knoten, in dem die Balance gestört ist und markieren Sie die an der Rotation beteiligten Knoten.







b) Wie Sie aus der Vorlesung wissen, liefert die *inorder*-Traversierung durch einen Suchbaum die Knoteneinträge in aufsteigend sortierter Reihenfolge. Gilt auch umgekehrt, dass die *postorder*-Traversierung die Knoteneinträge in absteigend sortierter Reihenfolge liefert?

Begründen Sie Ihre Antwort!

Nein, Gegenbeispiel: Baum mit Wurzel=5, linker Sohn=2 und rechter Sohn=7 liefert in postorder: 2-7-5

c) Stellt die Folge 2 4 7 8 5 6 9 einen Min-Heap dar?
Wenn ja: Zeichnen Sie den entsprechenden linksvollständigen Binärbaum.
Wenn nein: Begründen Sie, warum es sich nicht um einen Min-Heap handelt.

Es handelt sich **nicht** um einen Min-Heap, denn im Knoten mit Eintrag 7 ist die Heap-Eigenschaft verletzt:
Die beiden Söhne des Knotens haben die Einträge 6 und 9, im Vaterknoten steht also nicht das Minimum.

d) Entfernen Sie das minimale Element aus dem unten im Array dargestellten (Min-)Heap und stellen Sie die Heap-Eigenschaft wieder her. Fügen Sie anschließend in den resultierenden Heap das Element 48 ein und stellen Sie die Heap-Eigenschft wieder her.

Geben Sie nach dem Einfügen bzw. Entfernen eines Elementes an, welche Operation Sie zur Wiederherstellung der Heap-Eigenschaft verwenden (upheap/downheap).

Geben Sie nach jeder Vertauschung die vertauschten Elemente an. Gleichbleibende Einträge müssen Sie nicht abschreiben.

Array-Index i:	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
Operation	**0**	**4**	**3**	**8**	**26**	**5**	**7**	**63**	**70**	**42**	**32**	**31**	**35**
removeMin	**35**	4	3	8	26	5	7	63	70	42	32	31	_
downheap	**3**	4	**35**	8	26	5	7	63	70	42	32	31	
	3	4	**5**	8	26	**35**	7	63	70	42	32	31	
	3	4	**5**	8	26	**31**	7	63	70	42	32	**35**	
insert 48	3	4	5	8	26	31	7	63	70	42	32	35	**48**