Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
M.Sc. André Kless

Programmierung 2

Klausur + Lösung • Wintersemester 2024/25

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
   - 11 Aufgaben- und Lösungsblätter auf den Seiten 1 bis 11

   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 1-11 mit den Aufgaben und Ihren Lösungen, ggfs. die **nummerierten** Zusatzblätter (12, 13, 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 an, welche Datenstruktur für den jeweils beschriebenen Anwendungsfall am besten geeignet ist. Pro Anwendungsfall bitte nur eine Datenstruktur ankreuzen. Geben Sie in der letzten Tabellenspalte die Worst-Case-Laufzeit der häufigsten im Anwendungsfall beschriebenen Operation an. Mögliche Laufzeiten: O(1), $O(\log_2 n)$ oder O(n).

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 bitte nur eine Datenstruktur ankreuzen.

Anwendungsfall	**DynArray**	**Ringpuffer**	**DVL**	**AVL-Baum**	**Heap**	**Laufzeit**
Verwaltung von Aufträgen, die möglichst effizient den Auftrag mit der höchsten Priorität liefern soll.	☐	☐	☐	☐	☒	O(1)
Playlist bei der am häufigsten zum nächsten/vorherigen Song gewechselt wird.	(☒)	☐	☒	☐	☐	O(1)
Verwaltung von Befehlen, wobei die letzten x Befehle rückgängig machbar sein sollen. Häufigste Operation ist das Hinzufügen eines weiteren Befehls.	☐	☒	(☒)	☐	☐	O(1)
Wörterbuch bei dem möglichst effizient ein bestimmter Eintrag lexikographisch ermittelbar sein soll.	☐	☐	☐	☒	☐	$O(\log_2 n)$
Verwaltung einer Produktliste. Häufigste Operation ist der Zugriff auf ein bestimmtes Produkt über dessen fortlaufenden Index.	☒	☐	☐	☐	☐	O(1)

Aufgabe 2 (20 Punkte):

Wir betrachten folgende Variante eines Ringpuffers mit capacity = 3:

- **add() überschreibt** den ältesten Eintrag, falls der Puffer voll ist.
- **get()** liefert den **jüngsten** gepufferten Wert.
- **remove()** löscht den **jüngsten** Wert im Ringpuffer.

Gegeben sei der folgende Java-Code der abstrakten Oberklasse `Ringpuffer`:

```
public abstract class Ringpuffer<T> {

    /* Instanzvariablen */

    protected T[] elements;
    protected int size;
    protected int p;

    /* Konstruktor */

    public Ringpuffer(int capacity) {
        elements = (T[]) new Object[capacity];
        this.size = 0;
        this.p = 0;
    }

    /* Instanzmethoden */

    public int size() { return this.size; }

    public boolean isEmpty() { return (size == 0); }

    public boolean contains(T e) {
        for (int i = 0; i < size; i++) {
            if (elements[i].equals(e))
            return true;
        }
        return false;
    }

    public abstract void add(T e);

    public abstract T get();

    public abstract void remove();
}
```

a) Ergänzen Sie in der folgenden Unterklasse den Inhalt der Instanzmethoden:

```
public class MeinRingpuffer<T> extends Ringpuffer<T> {

    public MeinRingpuffer(int capacity) { super(capacity); }

    @Override
    public void add(T e) {

        elements[(p + size) % elements.length] = e;
        if (size < elements.length) {
            size++;
        } else {
            p++;
        }

    }

    @Override
    public T get() {   // soll null liefern, falls Ringpuffer leer ist

        if(isEmpty()) return null;

        return elements[(p + size - 1) % elements.length];




    }

    @Override
    public void remove() {

        if(isEmpty()) return;

        size--;



    }
}
```

b) Wir ergänzen nun in der Oberklasse `Ringpuffer` einen Iterator als innere Klasse, der die Werte im Ringpuffer in der folgenden Reihenfolge liefert: **“jüngster”, “zweitjüngster”,** … . Ergänzen Sie den fehlenden Code der inneren Klasse:

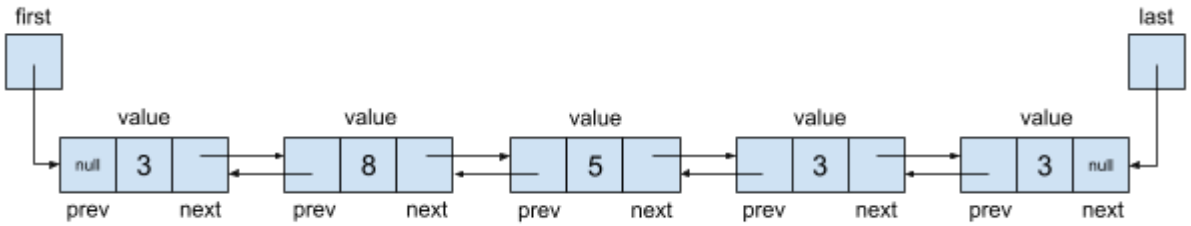
```
class MeinIterator implements Iterator<T> {

    private int i = 0;









    @Override
    public boolean hasNext() {

        return (i < size);









    }

    @Override
    public T next() {

        return elements[(p+size-1-i++)%elements.length];









    }
}
```

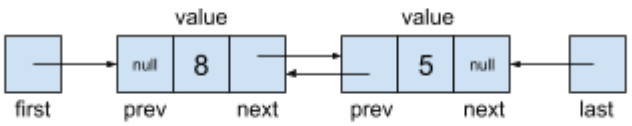
Aufgabe 3 (30 Punkte):

a) Gegeben sei folgendes Java-Programm:

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

Zeichnen Sie insgesamt zwei Speicherbilder: Ein Speicherbild, welches zunächst den Zustand der doppelt verketteten Liste (DVL) vor dem Aufruf der `deleteAll`-Methode zeigt und ein Speicherbild für den Zustand danach. Die Methode `deleteAll(T value)` löscht alle Vorkommen von `value` aus der Liste.





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 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 (auf der nächsten Seite) die Instanzmethode `deleteAll(T value)`.

```
public void deleteAll(T value) {

// Alle Einträge in der Liste durchgehen
ListenElem elem = first;
while (elem != null) {

   // Abfrage, ob Eintrag gelöscht werden muss
   if (elem.value.equals(value)) {

       // Nachfolger des Vorgängers neu setzen
       if (elem.prev != null)
           elem.prev.next = elem.next;
       else
           first = elem.next;  // "first" aktualisieren

       // Vorgänger des Nachfolgers neu setzen
       if (elem.next != null)
          elem.next.prev = elem.prev;
       else
           last = elem.prev;  // "last" aktualisieren

       size--;
   }

   elem = elem.next;
}
```

```
}
```

Aufgabe 4 (25 Punkte):

a) Gegeben sei folgendes Java-Programm:

```
public class SchlangeUtil {

  public static void main(String[] args) {

    Schlange<String> queue = new SchlangeDynArray<>();

    queue.enqueue("A");
    queue.enqueue("B");
    queue.enqueue("C");
    queue.enqueue("D");
    queue.enqueue("E");

    int rollbackCount = 2;

    System.out.println(queue); // A B C D E
    processRequests(queue, rollbackCount);
    System.out.println(queue); // D E
  }
}
```

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

/* 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 `processRequests`, die eine Warteschlange verarbeitet, die Kundenanfragen enthält (hier zur Vereinfachung nur Strings). Die verarbeiteten Anfragen sollen dabei zusätzlich in einem **Hilfsstack** (`StapelEVL` oder `StapelDynArray`) gespeichert werden. Am Ende sollen die letzen `rollbackCount` Anfragen wieder zurück in die Warteschlange geschoben werden. Der Stapel stellt sicher, dass die zurückgenommenen Anfragen am Ende wieder in ihrer ursprünglichen Reihenfolge in der Warteschlange stehen (siehe Testprogramm oben).

```
public static <T> void processRequests(Schlange<T> queue,
          int rollbackCount) {

   Stapel<T> stack = new StapelEVL<>();  // Hilfsstack anlegen

   while (!queue.isEmpty()) {  // solange Warteschlange nicht leer
       stack.push(queue.front());  // in Hilfsstack schieben
       queue.dequeue();            // aus Warteschlange entfernen
   }

   for (int i = 0; i < rollbackCount; i++) {  // Rollbacks machen
       queue.enqueue(stack.top());  // in Warteschlange schieben
       stack.pop();                 // aus Hilfsstack entfernen
   }
}
```

b) Gegeben sei folgendes Java-Programm:

```
public class SchlangeUtil {
    public static void main(String[] args) {
        Stapel<Integer> stack = new StapelEVL<>();
        stack.push(2);
        stack.push(4);
        System.out.println(stack);              // [2, 4]
        System.out.println(isSorted(stack));    // true
        stack.push(3);
        System.out.println(stack);              // [2, 4, 3]
        System.out.println(isSorted(stack));    // false
    }
}
```

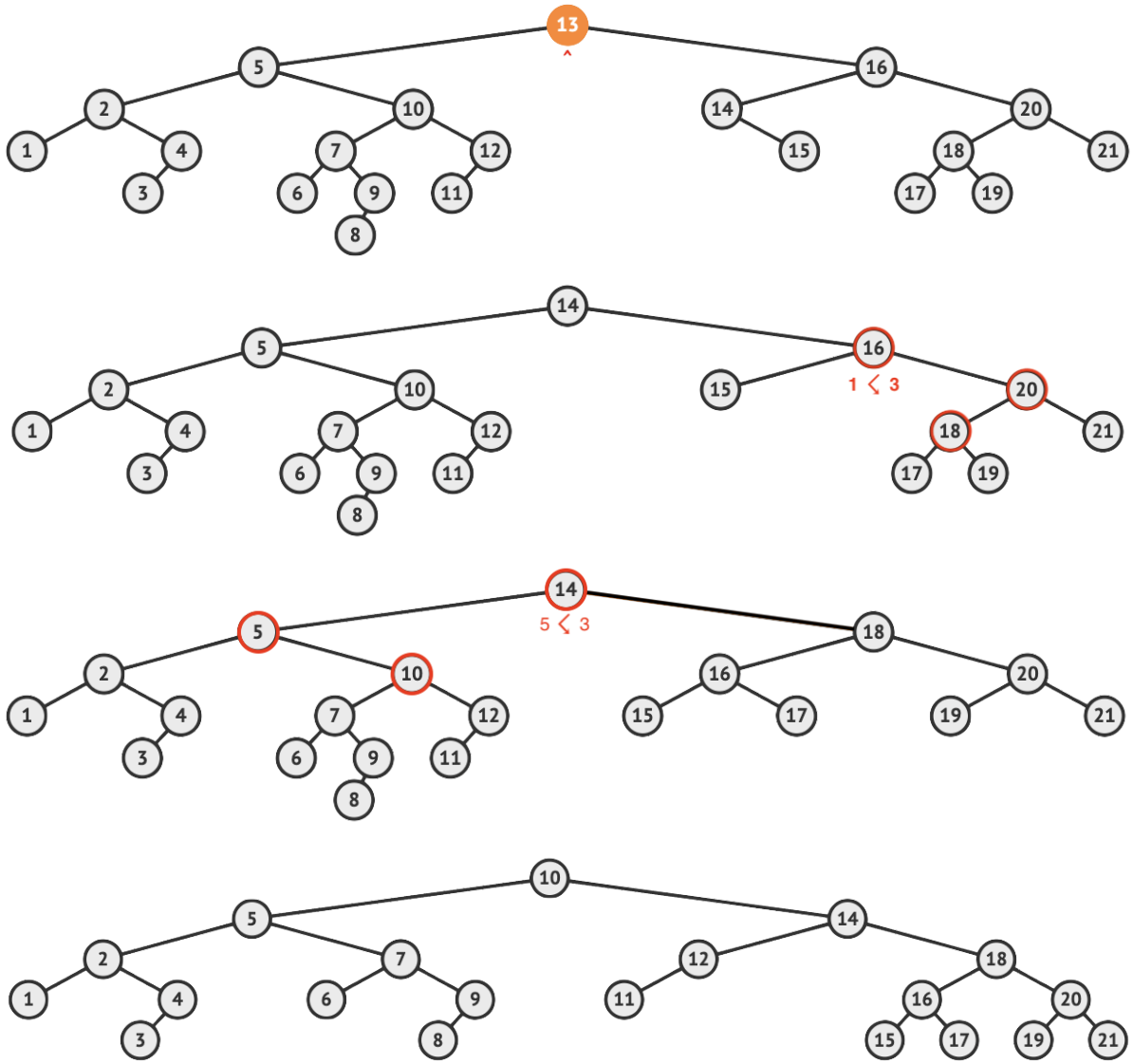
```
/* Verwenden Sie folgende
Definition des Interfaces: */
interface Stapel<T> extends
                        Iterable<T> {
    int size();
    boolean isEmpty();
    T top();
    void push(T e);
    void pop();
}
```

Implementieren Sie die statische **generische** Methode `isSorted`, die einen Stapel (Stack) übergeben bekommt und prüft, ob die darin enthaltenen Elemente aufsteigend sortiert sind. Der Stapel soll dabei **nicht** verändert werden. Nehmen Sie zudem eine Typeinschränkung vor, um Elemente im Stapel über ihre innere Ordnung vergleichen zu können. Ein leerer Stapel und ein Stapel der nur ein Element enthält, ist immer sortiert.

```
public static <T extends Comparable<T>> boolean isSorted(
                                                Stapel<T> stack) {

    // Leerer Stapel oder nur ein Element? → sortiert
    if (stack.isEmpty() || stack.size() == 1) return true;

    // Variable, um sich das vorherige Element zu merken.
    T previous = null;

    // Wenn ein vorheriges Element größer ist, dann nicht sortiert.
    for (T current : stack) {
        if (previous != null && previous.compareTo(current) > 0) {
            return false;

        }
        // Das aktuelle Element wird zum vorherigen Element.
        previous = current;
    }

    return true; // Alle Elemente waren in aufsteigender Reihenfolge.
}
```

Aufgabe 5 (25 Punkte):

a) Löschen Sie aus dem dargestellten AVL-Baum den Wert 13. Der gelöschte Knoten soll durch den **Inorder-Nachfolger** ersetzt werden. Führen Sie 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) Geben Sie allgemeine Formeln an, mit denen in einem array-basierten Heap für ein Element am Index i

- der Index des Elternknotens,
- der Index des linken Kindes,
- der Index des rechten Kindes berechnet werden kann.

Hinweis: Indizes beginnen bei 0.

Elternknoten: `(i - 1) / 2`

Linkes Kind: `2 * i + 1`

Rechtes Kind: `2 * i + 2`

c) Entfernen Sie das maximale Element aus dem unten im Array dargestellten (Max-)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	**50**	**45**	**40**	**30**	**35**	**25**	**20**	**10**	**5**	**15**	**12**	**18**	**8**
removeMax	8												-
downheap	45	8											
downheap		35			8								
downheap					15					8			
insert 48													48
upheap						48							25
upheap			48			40							
upheap	48		45										