

Aufgaben zur Klausur **Objektorientierte Programmierung** im WS 2010/11 (IA 252)

Zeit: 90 Minuten

erlaubte Hilfsmittel: keine

Bitte tragen Sie Ihre Antworten und fertigen Lösungen ausschließlich an den gekennzeichneten Stellen in das Aufgabenblatt ein. Ist ihre Lösung wesentlich umfangreicher, so überprüfen Sie bitte nochmals Ihren Lösungsweg.

Sollten Unklarheiten oder Mehrdeutigkeiten bei der Aufgabenstellung auftreten, so notieren Sie bitte, wie Sie die Aufgabe interpretiert haben.

Tipp: Bei der Entwicklung der Lösung kann eine kleine Skizze manchmal hilfreich sein.

Viel Erfolg!

Diese Klausur besteht einschließlich dieses Deckblattes aus 13 Seiten.

Aufgabe 1:

Vorrang-Warteschlangen (priority queues) werden zur Speicherung beliebig vieler Elemente oder Schlüssel-Wert-Paare verwendet, wobei nur der schnelle Zugriff auf das kleinste Element aus einer Menge gefordert wird. Das kleinste Element soll das mit der höchsten Priorität haben. Das effiziente Suchen eines beliebigen Elements ist nicht gefordert. Auf den Schlüsseln muss aber eine totale Ordnung definiert sein.

Eine einfache Implementierung für Vorrang-Warteschlangen ist die mit einer linearen Liste mit Sortierung und Duplikaten. Bei dieser Implementierung dauert der Zugriff auf das kleinste Element (auf den Kopf der Liste) konstant lange, aber das Einfügen läuft mit $O(n)$ mit $n =$ Anzahl Elemente in der Liste.

Mit binären Halden (binary heaps), die mit der gleichen Datenstruktur wie binäre Bäume arbeiten, ist eine Laufzeit für das Einfügen und Löschen von $O(\log n)$ möglich. Für binäre Halden gilt aber eine andere Invariante, als für binäre Bäume, und zwar dürfen beide Kinder eines Knotens keine Schlüssel haben, die kleiner sind als der Schlüssel des Knotens selbst. Diese Eigenschaft muss für alle Knoten in einer Halde gelten. Damit ist sichergestellt, dass ein Paar mit einem minimalen Schlüssel an der Wurzel eines Baumes gespeichert ist.

Das Einfügen und das Löschen von Einträgen kann auf eine gemeinsame Methode *mergeHeaps* zurückgeführt werden.

Beim Verändern einer Halde sollen nie Referenzen überschrieben werden, sondern immer neue Knoten erzeugt werden, so dass die *alte Halde* immer noch zu verwenden ist.

Das Lesen des Wurzel-Eintrags für Halden mit Schlüssel-Wertpaaren ist eine Funktion mit zwei Resultaten, dem Schlüssel und dem Wert. In C würde man dies mit einer Funktion mit einem zusätzlichen Referenzparameter für den 2. Wert realisieren, in Java ist dies nicht direkt möglich. Beachten Sie dies bei der Verwendung der entsprechenden Methode.

Die Schnittstelle für Vergleichsfunktionen findet man im JDK, die Funktion arbeitet ähnlich wie die *strcmp*-Funktion aus der C-Bibliothek. Die Wrapper-Klasse Integer implementiert dieses Interface.

```
interface Comparable {  
    public int compareTo(Object o);  
    // this < o2 : -1  
  
    // this = o2 : 0  
  
    // this > o2 : +1  
}
```

Die Hilfsklasse für die Referenzparameter.

```
public class Var {  
    public Object o;  
}
```

Die Funktion *isEqual* soll testen, ob zwei Halden nicht nur die gleichen Werte enthalten, sondern auch, dass die interne Struktur identisch ist.

Entwickeln Sie die fehlenden Methodenrumpfe so, daß in der Klasse *Heap* die Daten, wie oben beschrieben, gespeichert und verarbeitet werden.

```

public
  abstract
  class Heap {

    //—————

    // Prädikate

    public
      boolean isEmpty() {
        return
          false;
      }
    public
      boolean isEqual(Heap v2) {
        return
          false;
      }
    public abstract
      boolean inv();

    //—————

    // Selektoren

    public abstract
      Comparable minElement(Var res2);

    //—————

    public abstract
      Heap insert(Comparable k, Object a);

    public abstract
      Heap deleteMinElement();

    protected abstract
      Heap mergeHeaps(Heap h2);

    //—————

    private static final
      Heap empty = new EmptyHeap();
  }

```

```
public static
    Heap makeEmpty() {
    return
        empty;
}
```

```
public static
    Heap makeOne(Comparable k, Object a) {
    return
        new Node(empty,empty,k,a);
}
```

```
//—————
```

```
public static
    int nodesCreated = 0;
```

```
//—————
```

```
// die innere Klasse EmptyHeap
```

```
private static final
    class EmptyHeap extends Heap {
```

```
    public
        boolean isEmpty() {
            .....
            .....
        }
```

```
    public
        boolean isEqual(Heap v2) {
            .....
            .....
        }
```

```

public
    boolean inv() {
        .....
        .....
    }

public
    Heap insert(Comparable k, Object a) {
        .....
        .....
    }

public
    Comparable minElement(Var res2) {
        assert false
            : "minElement with empty heap";

        return
            null;
    }

public
    Heap deleteMinElement() {
        assert false
            : "deleteMinElement with empty heap";

        return
            null;
    }

protected
    Heap mergeHeaps(Heap h2) {
        return h2;
    }

public
    String toString() {
        return
            ".";
    }
}

// end class EmptyHeap

```

```

// die innere Klasse Node
private static final
    class Node extends Heap {
        final
            Heap l, r;
        final
            Comparable k;
        final
            Object a;

        Node(Heap l, Heap r, Comparable k, Object a) {
            this.l = l;
            this.r = r;
            this.k = k;
            this.a = a;
            ++nodesCreated;
        }
        public
            boolean isEmpty() {

                .....

                .....

            }
        public
            boolean isEqual(Heap v2) {

                .....

                .....

                .....

                .....

                .....

                .....

                .....

                .....

                .....

            }
    }

```

```
public
  boolean inv() {
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
  }
```

```
public
  Comparable minElement(Var res2) {
    .....
    .....
    .....
  }
```

```
protected
  Heap mergeHeaps(Heap h2) {
    if ( h2.isEmpty() )
      return
        this;

    return ( k.compareTo(((Node)h2).k) <= 0 )
      ? joinHeaps((Node)h2)
      : ((Node)h2).joinHeaps(this);
  }
```



```

private Heap joinHeaps(Node h2) {
    assert ( k.compareTo(h2.k) <= 0)
        : "minElement of left heap greater ";
        + "than minElement of right heap";

    return
        new Node(r, l.mergeHeaps(h2), k, a);
}

public
Heap insert(Comparable k, Object a) {
    .....
    .....
    .....
    .....
}

public
Heap deleteMinElement() {
    .....
    .....
    .....
}

public
String toString() {
    return
        " (" + l
        + " [" + k.toString() + ", " + a.toString() + " ] "
        + r + ") ";
}

}

// end Node

//—————
}

```

Gegeben sei folgendes Testprogramm:

```
public class Test {  
  
    public static void main(String [] argv) {  
  
        Var a = new Var();  
        Heap h = Heap.makeEmpty();  
        Integer k;  
  
        Heap.nodesCreated = 0;  
  
        for (int i = 1; i <= 3; ++i) {  
            h = h.insert(new Integer(i), new Integer(7 - i));  
        }  
  
        System.out.println(" 1.   :   " + h);  
        System.out.println(" 2.   :   " + Heap.nodesCreated);  
  
        Heap.nodesCreated = 0;  
        Heap h1 = h.deleteMinElement();  
  
        k = (Integer)h1.minElement(a);  
        System.out.println(" 3.   :   [ " + k + " , " + a.o + " ] ");  
  
        k = (Integer)h.minElement(a);  
        System.out.println(" 4.   :   [ " + k + " , " + a.o + " ] ");  
  
        System.out.println(" 5.   :   " + h1);  
        System.out.println(" 6.   :   " + h);  
        System.out.println(" 7.   :   " + Heap.nodesCreated);  
  
    }  
}
```

welche 7 Ausgabezeilen erzeugt dieses Programm?

.....

.....

.....

.....

.....

.....

.....

1. Diese Datenstruktur für die Halde ist so entwickelt, dass Objekte nie nach ihrer Erzeugung verändert werden. Welche Vorteile besitzt dieser Ansatz gerade in Java gegenüber einem Ansatz, bei dem Datenfelder verändert werden.

1)

2)

3)

2. Welche Nachteile besitzt dieser Ansatz gegenüber einem, bei dem Datenfelder verändert werden.

1)

2)

3)

3. Ist sichergestellt, dass diese Eigenschaft, dass keine Objekte verändert werden, auch bei Erweiterung der Klassenhierarchie (nicht dieser Quelle) um neue Klassen immer erhalten bleibt?

ja nein

Begründung:

.....

.....

4. Wäre es sinnvoll, den Konstruktor für **Node** **private** zu deklarieren?

ja nein

Begründung:

.....
.....

5. Ist es sinnvoll, die Funktion **makeEmpty** zu implementieren, anstatt die Variable *empty* als **public** zu deklarieren?

ja nein

Begründung:

.....
.....

6. Ist es sinnvoll, die Konstante **empty** als **static** zu deklarieren?

ja nein

Begründung:

.....
.....

7. Ist diese Datenstruktur *Thread-sicher*?

ja nein

Begründung:

.....
.....