Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

28. Januar 2016

Aufgaben zur Klausur **Algorithmen und Datenstrukturen** im WS 2015/16 (B_Inf, B_TInf, B_MInf, B_CGT, B_WInf, B_Ecom)

Zeit: 75 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.

Nutzen Sie die Rückseiten der Klausur zur Entwicklung der Lösungen und übertragen die fertigen Lösungen in das Aufgabenblatt.

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

Tipp: Bei der Entwicklung der Lösung können kleine Skizzen manchmal hilfreich sein.

Viel Erfolg!

Diese Klausur besteht einschließlich dieses Deckblattes aus 12 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 sein. 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 Knoten 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 *merge* zurückgeführt werden.

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

Die Schnittstelle für die allgemeingültigen öffentlichen Methoden ist im Interface *PriorityQueue* definiert:

```
public interface PriorityQueue {
    boolean isEmpty();
    int size();
    PV findMin();
    PriorityQueue insert(P p, V v);
    PriorityQueue removeMin();
    PriorityQueue copy();
    boolean inv();
}
```

Als Beispielklasse für die Schlüssel (Prioritäten) wird in dieser Aufgabe die Klasse *P*, für die Werte (Value) die Klasse *V* verwendet:

```
public class P implements Comparable<P> {
    public final int prio;
    public P(int p) {
        prio = p;
    }
    public static P mkP(int v) {
        return new P(v);
    }
    public int compareTo(P p2) {
        if (prio == p2.prio) return 0;
        if (prio > p2.prio) return 1;
        else return -1;
    }
    public boolean equalTo(P p2) {
        return compareTo(p2) == 0;
    }
    public String toString() {
        return " " + prio;
    }
}

public
    class V {
    public final int value;

    private V(int v) {
        value = v;
    }
    public static V mkV(int v) {
        return
            new V(v);
    }
    public String toString() {
        return " " + value;
    }
}
```

Die Methode zum Finden des kleinsten Elements liefert zwei Werte, einen Schlüssel vom Typ *P* und einen Wert vom Typ *V* zurück. Hierzu wird die Klasse *PV* verwendet:

```
public final
    class PV {
    public final P fst;
    public final V snd;

    private PV(P p, V v) {
        fst = p;
        snd = v;
    }
    public String toString() {
        return
            "(" + fst.toString() +
            ",  " + snd.toString() +
            ")";
    }
    public static PV mkPair(P p, V v) {
        return
            new PV(p, v);
    }
}
```

Die Hilfsroutinen aus *Util*:

```
public class Util {

    public static int max(int i, int j) {
        return
            i <= j ? j : i;
    }
    public static int min(int i, int j) {
        return
            j <= i ? j : i;
    }
}
```

Verwenden Sie in den Implementierungen der Methoden die in *P*, *V*, *PV* und der Klasse *Util* vorgegebenen Methoden und Funktionen (static-Methoden).

Entwickeln Sie die fehlenden Methodenrümpfe in der folgenden Klasse *BinaryHeap* und deren beiden lokalen Unterklassen *Empty* und *Node* schrittweise. Folgende Teilaufgaben sind dabei zu lösen. Die hier vorgegebene Reihenfolge ist ein sinnvoller Lösungsplan:

1. Lesen sie die gesamte Klasse *BinaryHeap* und deren Unterklassen sorgfältig durch.
2. Entwickeln Sie die Invariante, die aus den beiden Teilen in *Empty* und *Node* besteht und eine Hilfsmethode *childGE* für den Test nutzt, ob ein Teilbaum nur Schlüssel $>=$ dem momentanen Schlüssel besitzt.
3. Entwickeln Sie die beiden Implementierungen für *isEmpty*.
4. Entwickeln Sie die *copy*-Methode.
5. Entwickeln Sie die beiden Implementierungen für die Berechnung der Anzahl der Elemente *size*.
6. Zum Testen der Struktur einer Halde gibt es die beiden Methoden *depth* und *minDepth* mit denen die Länge der längsten und der kürzesten Pfade berechnet werden.
7. Implementieren Sie die in der Laufzeit konstante Methode *findMin*.
8. Implementieren Sie die Methode *insert* unter Verwendung der Hilfsmethode *merge*. *merge* mischt zwei Halden zu einer zusammen.
9. Implementieren Sie die Methode *removeMin* unter Verwendung der Hilfsmethode *merge*.
10. Veruchen Sie (z.B. mit Hilfe einer kleinen Skizze) nachzuvollziehen, wie die Hilfsmethode *join* arbeitet. Diese nimmt zwei nicht leere Halden und setzt daraus eine Halde zusammen. Dieses wird so gemacht, dass im Resultat der rechte Teilbaum von *this* nach links wandert, und der linke, nachdem er mit der Halde *n2* verschmolzen wurde, nach rechts wandert. Wenn *this* einen nicht größeren Schlüssel als *n2* besitzt, gilt für das Resultat wieder die Invariante, die beiden Halden sind also korrekt verschmolzen worden.
11. Implementieren sie die Methode *merge* für die beiden Fälle, in denen mindestens eine der Halden leer ist.
12. Implementieren sie die Methode *merge* für den Fall, das beide Halden nicht leer sind. In diesem Fall ist der Parameter von *merge* vom Typ *Node*. *merge* wird seine Hauptarbeit an *join* delegieren, wird also *join* aufrufen.
13. Geschafft, bitte einmal durchatmen.

```
public abstract
    class BinaryHeap
    implements PriorityQueue {

    public static BinaryHeap empty() {
        return
            EMPTY;
    }
    public static BinaryHeap singleton(P p, V v) {
        return
            new Node(p, v, EMPTY, EMPTY);
    }
    public abstract BinaryHeap insert(P p, V v);
    public abstract BinaryHeap removeMin();

    public BinaryHeap copy() {

        ..........................................

        ..........................................
    }
    abstract public int depth();
    abstract public int minDepth();

    abstract boolean childGE(P p);
    abstract BinaryHeap merge(BinaryHeap h2);

    private static <A> A nodeExpected() {
        throw
            new RuntimeException("Node expected");
    }

    //————————————————-


    private static final BinaryHeap EMPTY
        = new Empty();

    private static final
        class Empty
        extends BinaryHeap {

        Empty() { }
```

```
public boolean isEmpty() {
. . . . . .
}
public int size() {
. . . . . .
}
public int depth() {
. . . . . .
}
public int minDepth() {
. . . . . .
}
public PV findMin() {
. . . . . .
. . . . . .
}
public BinaryHeap insert(P p, V v) {
. . . . . .
. . . . . .
}
public BinaryHeap removeMin() {
. . . . . .
. . . . . .
}
public boolean inv() {
. . . . . .
. . . . . .
}
boolean childGE(P p) {
. . . . . .
. . . . . .
}
```

```
    BinaryHeap merge(BinaryHeap h2) {

    . . . . . .

    . . . . . .
    }
}

//————————————————-


private static final
    class Node
    extends BinaryHeap {

    /* persistent */
    final P p;
    final V v;
    final BinaryHeap l;
    final BinaryHeap r;

    Node(P p, V v, BinaryHeap l, BinaryHeap r) {
        assert p != null;

        this.p = p;
        this.v = v;
        this.l = l;
        this.r = r;
    }
    public boolean isEmpty() {

    . . . . . .
    }
    public int size() {

    . . . . . .

    . . . . . .

    . . . . . .
    }
```

```
public int depth() {
. . . . . .
. . . . . .
. . . . . .
}
public int minDepth() {
. . . . . .
. . . . . .
. . . . . .
}
public PV findMin() {
. . . . . .
. . . . . .
. . . . . .
}
public BinaryHeap insert(P p, V v) {
. . . . . .
. . . . . .
. . . . . .
}
public BinaryHeap removeMin() {
. . . . . .
. . . . . .
. . . . . .
}
```

```
public boolean inv() {
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
}
boolean childGE(P p) {
. . . . . .
. . . . . .
. . . . . .
}
BinaryHeap merge(BinaryHeap h2) {
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
}
```

```
        BinaryHeap join(Node n2) {
            assert this.p.compareTo(n2.p) <= 0;
            return
                setLR(this.r, this.l.merge(n2));
        }
        private Node setLR(BinaryHeap l, BinaryHeap r) {
            /* sinnvoll? */
            if ( l == this.l && r == this.r )
                return
                    this;
            return
                new Node(this.p, this.v, l, r);
        }
    } // end Node

}
```

Das Testen darf nicht vergessen werden. Dazu werden in der folgenden Klasse drei Halden *h1, h2, h3* aufgebaut. Skizieren Sie analog zu den Skizzen in der Vorlesung die drei Bäume. Tragen Sie dabei die Schlüssel in die Knoten ein. Ignorieren Sie in den Grafiken mögliche gemeinsame Nutzung von Java-Objekten.

```
class Main {
    BinaryHeap h1
        = BinaryHeap.empty()
        .insert(P.mkP(1), V.mkV(1))
        .insert(P.mkP(2), V.mkV(2))
        .insert(P.mkP(3), V.mkV(3));

    BinaryHeap h2
        = h1
        .insert(P.mkP(4), V.mkV(4));

    BinaryHeap h3
        = BinaryHeap.empty()
        .insert(P.mkP(3), V.mkV(3))
        .insert(P.mkP(2), V.mkV(2))
        .insert(P.mkP(1), V.mkV(1));
}
```

BinTree h1 =

BinTree h2 =

BinTree h3 =