Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

30. Januar 2018

Aufgaben zur Klausur **Algorithmen und Datenstrukturen** im WS 2017/18 (B_Inf, B_TInf, B_MInf, B_CGT, B_WInf, B_EComI, B_IMCA, B_ITE, B_STec)

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.

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 10 Seiten.

Aufgabe 1:

Zur Abschätzung des Wachstums von Funktionen wird die Groß-O-Notation verwendet. Es werden dabei Aussagen der Form $f \in O(g)$ gemacht.

Für welche mathematischen Objekte stehen f und g?

. . . . . .

Für welche mathematischen Objekte steht das große O?

. . . . . .

Geben Sie die mathematische Definition dafür an, was es heißt, dass $f \in O(g)$ ist.

Entwickeln Sie eine veranschaulichende Grafik für diese Definition.

Aufgabe 2:

Die folgenden Fragen beziehen sich alle auf die Beispiel-Implementierungen, die in der Vorlesung vorgestellt und diskutiert wurden. n sei dabei in allen Fällen eine natürliche Zahl größer als 0. Es werden in dieser Aufgabe Implementierungen betrachtet, die die *Map*-Schnittstelle implementieren.

1. Wieviele Java-Objekte werden für eine einfach verkettete Liste benötigt, in der n Schlüssel-Wert-Paare gespeichert sind?

   . . . . . .

2. Wieviel Speicher, gemessen in Anzahl Referenzen, wird für eine einfach verkettete Liste benötigt, in der n Schlüssel-Wert-Paare gespeichert sind?

   . . . . . .

3. Wieviele Objekte werden für einen binären Suchbaum benötigt, in dem n Schlüssel-Wert-Paare gespeichert sind?

   . . . . . .

4. Wieviel Speicher, gemessen in Anzahl Referenzen, wird für einen binären Suchbaum benötigt, in dem n Schlüssel-Wert-Paare gespeichert sind?

   . . . . . .

5. Wieviel Speicher, gemessen in Anzahl Referenzen plus Anzahl einfacher Werte, wird für einen Rot-Schwarz-Baum benötigt, in dem n Schlüssel-Wert-Paare gespeichert sind?

   . . . . . .

6. Wieviele *Leaf*-Objekte besitzt ein Patricia-Baum mit *Integer*-Schlüsseln, in dem n Schlüssel-Wert-Paare gespeichert sind?

   . . . . . .

7. Wieviele *Fork*-Objekte besitzt ein Patricia-Baum mit *Integer*-Schlüsseln, in dem n Schlüssel-Wert-Paare gespeichert sind?

   . . . . . .

8. Welches ist in Java die Speicher effizienteste Art n Schlüssel-Wert-Paare zu speichern?

   . . . . . .

   . . . . . .

9. Wieviele Objekte werden für eine Hash-Tabelle der Größe m benötigt, bei der die Schlüssel-Wert-Paare wegen der Kollisionsbehandlung in verketteten Listen gespeichert werden und sich in der Tabelle n Schlüssel-Wert-Paare befinden?

. . . . . .

10. Wieviele Speicher für Referenzen wird benötigt bei der Speicherung von n Schlüssel-Wert-Paaren in einer Hash-Tabelle der Größe m? Die Tabelle sei so organisiert, dass die Schlüssel-Wert-Paare wegen der Kollisionsbehandlung in verketteten Listen gespeichert werden.

. . . . . .

11. In welcher Komplexitätsklasse liegt die Operation *findMax* zum Zugriff auf den größten Schlüssel bei einem binären Suchbaum mit n Schlüssel-Wert-Paaren im Mittel?

. . . . . .

12. In welcher Komplexitätsklasse liegt die Operation *findMax* zum Zugriff auf den größten Schlüssel bei einem binären Suchbaum mit n Schlüssel-Wert-Paaren im schlechtesten Fall?

. . . . . .

13. In welcher Komplexitätsklasse liegt die Operation *findMax* zum Zugriff auf den größten Schlüssel bei einer aufsteigend sortierten, verketteten Liste mit n Schlüssel-Wert-Paaren im Mittel?

. . . . . .

14. In welcher Komplexitätsklasse liegt die Operation *lookup* zum Suchen eines Schlüssels in einem Patricia-Baum, der n Schlüssel-Wert-Paare enthält?

. . . . . .

15. In welcher Komplexitätsklasse liegt die Operation *lookup* zum Suchen eines Schlüssels in einem binären Patricia-Baum mit fester Schlüssellänge l, der n Schlüssel-Wert-Paare enthält?

. . . . . .

16. In welcher Komplexitätsklasse liegt die Operation *findMin* zum Zugriff auf den kleinsten Schlüssel in einem binären Patricia-Baum mit fester Schlüssellänge l, der n Schlüssel-Wert-Paare enthält?

. . . . . .

Aufgabe 3:

Vorrang-Warteschlangen sind eine effiziente Datenstruktur zur Speicherung von Elementen, denen Prioritäten zugeordnet sind, und zum Zugriff und Löschen des Elements mit der höchsten Priorität. Als Konvention gilt, je kleiner der Wert, umso höher die Priorität.

Beim Eintragen und Löschen von Elementen muss streng eine Konsistenzbedingung eingehalten werden.

Die Schnittstelle für Warteschlangen *Queue* ist hier auf die Methoden reduziert worden, die in dieser Aufgabe relevant sind.

```
public interface Queue extends Invariant {
    boolean isEmpty();
    PV findMin();
    PriorityQueue insert(P p, V v);
    PriorityQueue removeMin();
}

public interface Invariant { boolean inv(); }
```

Weiter sind die Beispiel-Klassen *P* und *PV* für die Prioritäten mit den notwendigen Eigenschaften gegeben:

```
public class P implements Comparable<P> {
    public final int prio;

    public P(int p) { prio = p; }

    public int compareTo(P p2) {
        if (prio == p2.prio) return 0;
        if (prio > p2.prio) return 1;
        return -1;
    }
}

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

    private PV(P p, V v) { fst = p; snd = v; }

    public static PV mkPair(P p, V v) {
        return new PV(p, v);
    }
}
```

Die anderen Klassen sind die aus der Vorlesung bekannten Hilfsklassen.

Entwickeln Sie die fehlenden Methodenrümpfe in der folgenden Klasse *BinaryHeap* und deren lokalen Unterklassen *Empty* und *Node*. Es sind folgende Teilaufgaben zu lösen:

1. Die Konsistenzbedingung wird durch eine Invariante implementiert. Implementieren sie die Invarianten mit Hilfe der Methoden *childGE*
2. *size* berechnet die Anzahl gespeicherter Prioritäten-Wert-Paare.
3. Die Methoden *findMin* zum Zugriff auf das Element mit der höchsten Priorität.
4. Die Methode *removeMin* löscht das Element mit der höchsten Priorität.
5. Die Methode *insert* fügt ein neues Prioritäten-Wert-Paar in die Halde ein.

```
public abstract class BinaryHeap implements Queue {

    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();

    abstract boolean childGE(P p);

    abstract BinaryHeap merge(BinaryHeap h2);
    abstract BinaryHeap merge2(Node n1);

    private static <A> A nodeExpected() {
        return undefined("Node expected");
    }
    private static final BinaryHeap EMPTY = new Empty();
```

```
private static final class Empty extends BinaryHeap {

    public boolean isEmpty() { return true; }

    public int size() {

     . . . . . .
    }
    public PV findMin() {

     . . . . . .
    }
    public BinaryHeap insert(P p, V v) {

     . . . . . .
    }
    public BinaryHeap removeMin() {

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

     . . . . . .
    }

    Empty() { }

    boolean childGE(P p) {

     . . . . . .

     . . . . . .
    }
    BinaryHeap merge(BinaryHeap h2) {

     . . . . . .
    }
    BinaryHeap merge2(Node n1) {
        return n1;
    }
}
```

```
private static final class Node extends BinaryHeap {
    final P p;
    final V v;
    final BinaryHeap l;
    final BinaryHeap r;

    Node(P p, V v, BinaryHeap l, BinaryHeap r) {
        this.p = p; this.v = v;
        this.l = l; this.r = r;
    }

    public boolean isEmpty() { return false; }

    public int size() {

     . . . . . .

     . . . . . .

     . . . . . .
    }
    public PV findMin() {

     . . . . . .

     . . . . . .
    }
    public BinaryHeap insert(P p, V v) {

     . . . . . .

     . . . . . .
    }

    public BinaryHeap removeMin() {

     . . . . . .

     . . . . . .
    }
```

```
        public boolean inv() {
            ..........
            ..........
            ..........
            ..........
            ..........
            ..........
            ..........
        }
        boolean childGE(P p) {
            ..........
            ..........
            ..........
        }
        BinaryHeap merge(BinaryHeap h2) {
            return h2.merge2(this);
        }
        BinaryHeap merge2(Node n1) {
            if ( this.p.compareTo(n1.p) <= 0)
                return this.join(n1);
            return n1.join(this);
        }
        BinaryHeap join(Node n2) {
            return setLR(this.r, this.l.merge(n2));
        }
        private Node setLR(BinaryHeap l, BinaryHeap r) {
            if ( l == this.l && r == this.r )
                return this;
            return new Node(this.p, this.v, l, r);
        }
    }
}
```

Machen sie die folgenden Zeitabschätzungen unter der Annahme, dass die Methode *merge* in einer Zeitkomplexität von $O(log(n_1 + n_2))$ läuft mit $n_1 = this.size()$ und $n_2 = h2.size()$.

Beim Löschen und Einfügen in eine Halde h sei die Anzahl der Elemente $n = h.size()$.

Mit welcher Zeitkomplexität arbeitet das Suchen ***findMin***?

. . . . . .

Mit welcher Zeitkomplexität arbeitet das Einfügen *insert*?

. . . . . .

Mit welcher Zeitkomplexität arbeitet das Löschen ***removeMin***?

. . . . . .

Mit welcher Zeitkomplexität arbeitet ein auf dieser Datenstruktur basierender Sortieralgorithmus

. . . . . .
