Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

3. März 2016

Aufgaben zur 2. Klausur **Algorithmen und Datenstrukturen** im WS 2015/16 (B_Inf, B_TInf, B_MInf, B_WInf, B_CGT)

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:

Seien f_1, f_2, g_1, g_2 Funktionen vom Typ $\mathbb{N} \to \mathbb{R}$.
Weiter gelte $f_i(n) \in O(g_i(n))$ und $c_i \in \mathbb{R}$ für $i \in \{1, 2\}$.

1. Aus welcher Komplexitätsklasse ist $f(n) = c_1 * (f_1(n) + c_2 * n)$

   .....

2. Aus welcher Komplexitätsklasse ist $f(n) = c_1 * f_1(n) * n^2 + c_2 * f_1(n) * n^2$

   .....

3. Aus welcher Komplexitätsklasse ist $f(n) = f_1(n) * (f_2(n) + c_2 * n)$

   .....

4. Aus welcher Komplexitätsklasse ist $f(n) = (f_1(n))^2 + (f_1(n))^3$

   .....

Aufgabe 2:

In der Vorlesung sind mehrere Datenstrukturen zur Laufzeit- und Speicherplatz-effizienten Implementierung von Mengen und Maps behandelt worden. In dieser Aufgabe soll eine weitere Möglichkeit teilweise entwickelt werden zur Implementierung von Mengen von ganzen Zahlen.

Eine Menge von ganzen Zahlen kann in zusammenhängende Bereiche (Intervalle, eng. *Range*s) aufgeteilt werden. Diese Intervalle können in einer sortierten, einfach verketteten Liste gespeichert werden. So kann die Menge $m_1 = \{1, 2, 3, 5, 7, 8, 9\}$ durch eine dreielementige Liste von Bereichen $l_1 = [1..3, 5..5, 7..9]$ repräsentiert werden.

Für eine effiziente Implementierung ist zu beachten, dass immer mit der minimalen Anzahl von Intervallen gearbeitet wird. So sollte nach dem Einfügen des Wertes 4 in die Menge m_1 eine neue Liste l_2 mit nur noch 2 Intervallen entstehen. Wird dann anschließend noch die 6 eingefügt, genügt ein Intervall zur Darstellung des Resultats.

Einige nützliche Hilfsmethoden sind in einer Klasse *Util* vordefiniert.

```
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;
    }
}
```

Als erstes soll eine Klasse *Range* entwickelt werden zur Darstellung von Bereichen. Die Klasse *Range* enthält 2 Variablen für die untere und obere Genze eines Intervalls. Intervalle können leer sein, also keine Elemente enthalten. Alle leeren Intervalle werden einheitlich durch genau ein leeres Intervall repräsentiert (siehe Konstruktor *Range*).

Intervalle können geordnent werden. Diese Ordnung wird später noch benötigt werden. Hierfür wird die vordefinierte Java-Schnittstelle *Comparable* mit der Methode *compareTo* verwendet. Die Methode soll als Resultate genau -1, 0 und +1 liefern, 0 bei Gleichheit, -1 wenn *this* kleiner als *r2* ist, +1 bei größer. Das leere Intervall ist das kleinstmögliche Intervall. Für die übrigen Intervalle gilt: Ein Intervall ist kleiner, wenn seine untere Grenze kleiner ist oder wenn die untere Grenze gleich ist und die obere Grenze kleiner ist. Entwickeln Sie als erstes in der Klasse *Range* die Methode *compareTo*.

overlap ist eine Methode, mit der getestet werden kann, ob zwei Intervalle sich überlappen, also gemeinsame Elemente besitzen oder ob die Intervalle zusammenstoßen, also dass es keine Zahlen zwischen den Intervallen gibt. Leere Intervalle überlappen per Definition nie.

add ist eine Methode um zwei überlappende (siehe *overlap*) Intervalle zu einem Intervall zu verschmelzen. Überlappen die Intervalle nicht, so soll *add* **this** als Resultat liefern.

```
public
    class Range implements Comparable<Range>{
    public final int lb;
    public final int ub;

    public Range(int i) {
        this(i,i);
    }

    public Range(int lb1, int ub1) {
        if (lb1 > ub1) {
            lb = 0;
            ub = −1;
        } else {
            lb = lb1;
            ub = ub1;
        }
    }

    public boolean isEmpty() {
        return
            lb > ub;
    }
```

public int compareTo(Range r2) {

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

}

public boolean overlap(Range r2) {

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

}

```
    public Range add(Range r2) {
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    }
}
```

Ende der Klasse *Range*

Die zu implementierende Klasse *RangeSet* soll die Schnittstelle aus *IntSet* implementieren.

```
public interface IntSet {
    boolean inv();
    boolean isEmpty();
    boolean member(int i);
    int findMin();
    int size();
    IntSet insert(Range r);
    IntSet insert(int i);
    IntSet union(IntSet s2);
}
```

Bei der Entwicklung von *RangeSet* ist es empfehlenswert, die Methoden in der hier aufgelisteten Reihenfolge zu implementieren.

Hinweise zu den einzelnen Methoden der Klasse *RangeSet* und ihren lokalen Unterklassen:

RangeSet, Node, Empty: Die Datenstruktur ist rekursiv definiert. Daraus folgt, das viele der Methoden sinnvollerweise auch eine rekursive Struktur besitzen. Das Arbeiten mit Schleifen ist wenig zielführend.

inv: Die Invariante soll überprüfen, ob die in der Einleitung beschriebenen Anforderungen an die Liste der Intervalle (Sortierung, minimale Anzahl von Intervallen, ...) gelten. *compareTo* aus *Range* muss hier genutzt werden.

insert: Beachten Sie, dass beim Einfügen eines Intervall unter Umständen mehrere Intervalle in der Liste verschmolzen werden.

insert, union: *RangeSet* soll als persistente Datenstruktur realisiert werden, d.h. Objekte verändern ihren Zustand nicht. Die lokale Hilfsmethode *setRS* ist hier hilfreich.

union: Durch die Sortierung der Intervalle kann ein effizentes Verschmelzen der beiden Listen erreicht werden. Die Laufzeit sollte linear in der Anzahl der Intervalle in beiden Listen sein. Ein mehrfaches Durchlaufen von (Teil-)Listen muss also vermieden werden. *insert* kann hier als Baustein sinnvoll genutzt werden.

```
public
    abstract class RangeSet implements IntSet {

    public static RangeSet empty() {

     . . . . . .
    }

    public static RangeSet singleton(Range r1) {

     . . . . . .

     . . . . . .

     . . . . . .
    }

    public static RangeSet singleton(int i) {

     . . . . . .
    }

    abstract public RangeSet insert(Range r);
    abstract public RangeSet  union(IntSet s2);

    public RangeSet insert(int i) {

     . . . . . .
    }

    private static final RangeSet EMPTY = new Empty();

    static class Empty extends RangeSet {
        Empty() {}

        public boolean inv() {

     . . . . . .
        }

        public boolean isEmpty() {
            return true;
        }
```

```
    public boolean member(int i) {
. . . . . .
    }

    public int findMin() {
. . . . . .
. . . . . .
    }

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

    public RangeSet insert(Range r) {
. . . . . .
. . . . . .
. . . . . .
    }

    public RangeSet  union(IntSet s2) {
. . . . . .
. . . . . .
    }
}

static class Node extends RangeSet {
    final Range r;
    final RangeSet rs;

    Node(Range r, RangeSet rs) {
        this.r = r;
        this.rs = rs;
    }
```

```
    public boolean inv() {
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
    }
    public boolean isEmpty() {
. . . . . .
    }
    public boolean member(int i) {
. . . . . .
. . . . . .
. . . . . .
    }
    public int findMin() {
. . . . . .
    }
    public int size() {
. . . . . .
. . . . . .
    }
```

```
    public RangeSet insert(Range r2) {
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
    }
```

```
        public RangeSet  union(IntSet s2) {
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
        }

        private RangeSet setRS(RangeSet rs) {
            return new Node(r, rs);
        }
    }
}
```
