Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

30. Januar 2017

---

Aufgaben zur Klausur **Algorithmen und Datenstrukturen** im WS 2016/17 ( B_Inf, B_TInf, B_MInf, B_CGT, B_WInf, B_Ecom, B_ITE)

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 13 Seiten.

---

**Aufgabe 1:**

Seien $f_1, f_2, f_3, g_1, g_2, g_3$ 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 $i \in \{1, 2, 3\}$.

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

   .....

2. Aus welcher Komplexitätsklasse ist $f(n) = f_1(n) - f_2(n) * f_3(n)$

   .....

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

   .....

4. Aus welcher Komplexitätsklasse ist $f(n) = f_1(n) * f_1(n)$

   .....

5. Aus welcher Komplexitätsklasse ist $f(n) = c_1 * n^3 + c_2 * n * \log n + c_3 * n^2$

   .....

---

**Aufgabe 2:**

Verkettete Listen sind für viele Operationen auf Mengen eine ineffiziente Datenstruktur. Aber wenn die Elemente in den Listen geordnent sind, können Mengenvereinigung und Mengendurchschnitt effizient implementiert werden.

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

```
public interface Set {
    boolean isEmpty();
    boolean member(E e);
    Set union(Set m2);
    Set intersect(Set m2);
}
```

Weiter ist die Beispiel-Elementklasse *E* mit den notwendigen Eigenschaften der Elemente gegeben:

```
public class E implements Comparable<E> {
    private int value;

    public E(int v) { value = v; }

    public int compareTo(E e2) {
        if (value == e2.value) return 0;
        if (value > e2.value) return 1;
        return -1;
    }
}
```

Entwickeln Sie die fehlenden Methodenrümpfe in der folgenden Klasse *OrderedList* und deren beiden lokalen Unterklassen *Empty* und *Node*. Es sind zwei Teilaufgaben zu lösen, die Methode *union* (Mengenvereinigung) für die beiden Unterklassen und die Methode *intersect* für den Mengendurchschnitt.

Wie aus der Methode *member* zu erkennen ist, sind die Elemente der Mengen aufsteigend sortiert. Die Mengenimplementierung ist persistent, es werden also nie Objekte verändert, sondern immer, wenn notwendig, neue Objekte konstruiert (**final**-Attribute, *setter*-Funktionen).

```
import static java.lang.Integer.signum;

public abstract
    class OrderedList
    implements Set {

    public static OrderedList empty() { return EMPTY; }
    public abstract OrderedList union(Set m2);
    public abstract OrderedList intersect(Set m2);

    protected Node cons(E e) { return new Node(e, this); }
    private Node node() { return nodeExpected(); }
    private E elem() { return nodeExpected(); }
    private OrderedList next() { return nodeExpected(); }

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

    private static final OrderedList EMPTY = new Empty();

    //————————————————————-


    private static final class Empty extends OrderedList {
        public boolean isEmpty() { return true; }
        public boolean member(E e) { return false; }

        public OrderedList union(Set m2) {

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

         ..........................................................
        }

        public OrderedList intersect(Set m2) {

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

         ..........................................................
        }
    }

    //————————————————————-
```

```
private static final class Node extends OrderedList {
    final E e;
    final OrderedList next;

    Node(E e, OrderedList next) {
        this.e = e;
        this.next = next;
    }
    public boolean isEmpty() { return false; }
    public boolean member(E e) {
        switch (signum(e.compareTo(this.e))) {
        case −1: return false;
        case 0: return true;
        case +1: return next.member(e);
        }
        return false;
    }
    Node node() { return this; }
    E elem() { return e; }
    OrderedList next() { return next; }

    private Node setNext(OrderedList next) {
        if ( next == this.next )
            return this;
        return next.cons(this.e);
    }
```

**public** OrderedList **union**(Set m2) {

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

}

```
        public OrderedList intersect(Set m2) {
            .....
            .....
            .....
            .....
            .....
            .....
            .....
            .....
            .....
            .....
            .....
            .....
            .....
            .....
        }
}
```

Mit welcher Zeitkomplexität arbeitet *union*?

.....

Mit welcher Zeitkomplexität arbeitet *intersect*?

.....

Mit welcher Zeitkomplexität würde *union* mit unsortierten Listen arbeiten?

.....

**Aufgabe 3:**

Binäre Suchbäume sind eine Laufzeit effizientere Implementierung für Mengen und Maps als verkettete Listen. Aber beim Eintragen und Löschen von Elementen muss streng eine Konsistenzbedingung eingehalten werden. In dieser Aufgabe wird mit Maps gearbeitet.

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

```
public interface Map extends Invariant {
    boolean isEmpty();
    Object lookup(K k);
    int size();
}

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

Weiter ist die Beispiel-Klasse *K* für die Schlüssel mit den notwendigen Eigenschaften gegeben:

```
public class K implements Comparable<K> {
    public final int key;

    public K(int k) { key = k; }

    public int compareTo(K k2) {
        if (key == k2.key) return 0;
        if (key > k2.key) return 1;
        return -1;
    }
}
```

Entwickeln Sie die fehlenden Methodenrümpfe in der folgenden Klasse *BinaryTree* und deren beiden 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 beiden Methoden *allLS* (alle kleiner) und *allGT* (alle größer).
2. *size* berechnet die Anzahl gespeicherter Schlüssel-Wert-Paare.
3. Die Methoden *depth* und *minDepth* werden benötigt, um die Ausbalanciertheit eines Baumes zu testen.
4. *balanced* ist ein Prädikat, das berechnet, ob ein Baum perfekt ausbalaciert ist.

```
import static java.lang.Integer.signum;

public abstract class BinaryTree implements Map {

    public static BinaryTree empty() {
        return EMPTY;
    }
    public static BinaryTree singleton(K k, Object v) {
        return new Node(k, v, EMPTY, EMPTY);
    }

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

    public boolean balanced() {

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

        ..........................................................
    }

    Node node() { return nodeExpected(); }
    K key() { return nodeExpected(); }
    Object value() { return nodeExpected(); }
    BinaryTree left() { return nodeExpected(); }
    BinaryTree right() { return nodeExpected(); }

    abstract boolean allLS(K k);
    abstract boolean allGT(K k);

    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; }

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

    private static final BinaryTree EMPTY = new Empty();
```

```
private static final class Empty extends BinaryTree {

    public boolean isEmpty() { return true; }

    public int size() {

    ..........................................................
    }

    public int depth() {

    ..........................................................
    }

    public int minDepth() {

    ..........................................................
    }

    public Object lookup(K k) { return null; }

    public boolean inv() {

    ..........................................................
    }

    boolean allLS(K k) {

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

    ..........................................................
    }
    boolean allGT(K k) {

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

    ..........................................................
    }

}
```

```
private static final class Node extends BinaryTree {
    final K k;
    final Object v;
    final BinaryTree l;
    final BinaryTree r;

    Node node() { return this; }
    K key() { return k; }
    Object value() { return v; }
    BinaryTree left() { return l; }
    BinaryTree right() { return r; }

    Node(K k, Object v, BinaryTree l, BinaryTree r) {
        this.k = k; this.v = v;
        this.l = l; this.r = r;
    }
    public boolean isEmpty() { return false; }
    public Object lookup(K k) {
        switch (signum(k.compareTo(this.k))) {
        case −1: return l.lookup(k);
        case 0: return v;
        case +1: return r.lookup(k);
        }
        return null;
    }
    public int size() {

    . . . . . .

    . . . . . .
    }

    public int depth() {

    . . . . . .

    . . . . . .
    }

    public int minDepth() {

    . . . . . .

    . . . . . .
    }
```

```
        public boolean inv() {
            .....
            .....
            .....
            .....
            .....
        }
        boolean allLS(K k) {
            .....
            .....
            .....
            .....
            .....
        }
        boolean allGT(K k) {
            .....
            .....
            .....
            .....
            .....
        }
    }
}
```

Mit welcher Zeitkomplexität arbeitet das Suchen *lookup*, wenn das Prädikat *balanced* gilt?

. . . . . .

Mit welcher Zeitkomplexität arbeitet das Einfügen in einen Baum t, wenn *t.size() == t.depth()* gilt?

. . . . . .

In welcher Komplexitätsklasse liegt die Zeit für die Vereinigung zweier als binärer Suchbäume implementierten Maps im Mittel?

. . . . . .

In welcher Komplexitätsklasse liegt die Zeit für die Vereinigung zweier binärer Suchbäume implementierten Maps im schlimmsten Fall?

. . . . . .

---