Grundlagen der Programmkonstruktion
Übungsblatt 8

1 Comparable-Interface

Klassen, die das generische `Comparable<A>`-Interface implementieren, enthalten eine Methode `int compareTo(A elem)`, mit der die aktuelle Instanz (`this`) mit dem Parameter `elem` verglichen wird.

1.1 Klasse `Num` (0.5)

Die folgende Klasse `Num` repräsentiert ganze Zahlen:

```
public class Num {
    private int n;

    public Num(int n) {
        this.n = n;
    }
}
```

Implementieren Sie das `Comparable`-Interface in dieser Klasse `Num`, sodass Sie Instanzen von `Num` mit anderen Instanzen von `Num` vergleichen können.

Lesen Sie dazu zunächst sorgfältig die Dokumentation zu `Comparable`. Diese finden Sie hier: `http://docs.oracle.com/javase/6/docs/api/java/lang/Comparable.html`). Benutzen Sie Generizität so, dass Sie keine Typabfrage benötigen (d.h. kein `instanceof` oder `getClass()`).

1.2 Klasse `IntList` (1)

Implementieren Sie das Comparable-Interface in der Klasse `IntList`, die Sie aus der letzten Übungseinheit und der Vorlesung bereits kennen. Eine Instanz von `IntList` ist dabei größer/kleiner als eine andere Instanz von `IntList`, wenn das erste Element größer/kleiner ist. Sollte das erste Element beider `IntList`s gleich sein, so wird das zweite Element verglichen, danach das dritte etc. . . Sollte eine Liste kein weiteres Element mehr enthalten, die andere jedoch schon, so ist die kürzere Liste kleiner, ansonsten sind beide Listen gleich.

Hinweise

- Beachten Sie bei dieser Aufgabe, dass die `compareTo`-Methode die Listen nicht verändern darf.
- Sie können allen Klassen beliebige Methoden hinzufügen.
- Das Kopieren der Listen in eine andere vordefinierte Datenstruktur ist nicht erlaubt.

2 Klasse `GenQueue<A>` (1.5)

Eine Queue ist eine Datenstruktur mit zwei Methoden, vergleichbar mit einem Stack. Diese Methoden sind `void enqueue(A e)` und `A dequeue()`. `void enqueue(A e)` fügt ein Element der Queue hinzu (wie die Methode `void push(A e)` bei einem Stack). `A dequeue()` löscht ein Element von der Queue und gibt dieses Element zurück (ähnlich wie die Methode `A pop()` bei einem Stack). Im Gegensatz zu einem Stack gibt die Methode `A dequeue()` jedoch *nicht* das zuletzt auf die Queue gelegte Element zurück, sondern das erste Element, das der Queue hinzugefügt wurde (first-in-first-out).

Erstellen Sie eine generische Klasse `GenQueue` mit einem Typparameter, die eine solche Queue repräsentieren soll (für beliebig viele Elemente). Erstellen Sie einen Konstruktor, der eine leere Queue zurückliefert. Den Fehlerfall, dass `dequeue` auf eine leere Queue ausgeführt wird, müssen Sie nicht behandeln.

Verwenden Sie für diese Aufgabe *keine* anderen bereits defininierte Klassen!

Beispiel:

```
GenQueue<String> queue = new GenQueue<String>();
queue.enqueue("A");
queue.enqueue("B");
System.out.println(queue.dequeue()); // "A"
System.out.println(queue.dequeue()); // "B"
```

3 Klasse `GenTree<A>`

Die folgenden Aufgaben beziehen sich auf das Beispielprogramm `GenTree.java` aus der Vorlesung vom 03. Dezember (`http://www.complang.tuwien.ac.at/franz/programmkonstruktion/pk12w14p/`).

Sie dürfen allen Klasse beliebige Methoden hinzufügen.

3.1 A getMimimum() (0.5)

Schreiben Sie eine Methode `A getMimimum()`, die das kleinste Element in einem Objekt von `GenTree` zurückliefert. Bei Ihrer Lösung soll jeder Knoten im Baum maximal einmal besucht werden.

3.2 boolean add(A e) (0.7)

Ändern Sie die Methode `void add(A e)` so ab, dass sie ein Element nur dann dem Baum hinzufügt, wenn er das Element `e` noch nicht enthält. In diesem Fall soll die Methode `true` zurückliefern. Sollte das Element bereits im Baum vorhanden sein, so soll der Baum nicht geändert werden und `false` zurückgeliefert werden.

Beachten Sie bei dieser Aufgabe, dass jeder Knoten im Baum nur einmal besucht werden darf, d.h. ein Aufruf von `boolean contains(e)` mit einem folgenden Aufruf von `void add(e)` ist *nicht* erlaubt.