hochschule mannheim

PR2 – Programmierung 2 FAKULTÄT FÜR INFORMATIK
Cyber Security Bachelor (CSB), Medizinische Informatik Bachelor (IMB)
Sommersemester 2020 [02.06.2020]

PR2 Aufgabenblatt für Testate Nr. 4

Allgemeine Hinweise

Ausgabe der Übung:	Di, 02.06.2020, 16:50 Uhr
Abgabe:	Mo, 15.06.2020, 16:00 Uhr
Testat am:	Di, 16.06.2020, 13:40–16:50 (online)

- Die Abnahme der Übungen gilt als **Prüfungsleistung**. Bei einer Verhinderung durch Krankheit ist eine ärztliche Bescheinigung der Arbeitsunfähigkeit vorzulegen.
- Laden Sie Ihre Lösungen bis zur Deadline in Moodle bei der entsprechenden Unteraufgabe hoch. Quellcodes müssen einheitlich und sinnvoll formatiert sein (vorzugsweise mithilfe Ihrer IDE wie Eclipse oder IntelliJ). **Ausarbeitungen in einem anderen Format werden nicht berücksichtigt.**
- Während der Abnahme sind die Ergebnisse am Rechner live zu demonstrieren.
- Bei der Abnahme der Übung ist der Studentenausweis vorzulegen.

Lernziele

- Gebundene generische Typen spezifizieren und anwenden können.
- Datenstruktur binärer Suchbaum verstehen, selbst entwickeln und anwenden können.
- Mit AVL-Bäumen arbeiten können.
- Datenstruktur Heap verstehen, selbst entwickeln und anwenden können.
- Hashing-Verfahren und Anwendung für `Set`-Implementierung und kryptographische Hashes kennen.
- Mit Schnittstellen von Sortierverfahren programmieren können.

Aufgabe 1

`Comparator` für generischen Record

Entwickeln Sie eine generische Klasse namens `Tripel`. Sie soll im Package `de.hsmannheim.inf.pr2.ads` sein. Diese Klasse soll drei Instanzvariablen namens `u`, `v` und `w` haben. Alle drei Instanzvariablen sollen unterschiedliche Typen haben können.

Diese drei Typen sollen als Typ-Parameter der Klasse angegeben werden können. Für alle drei Typen soll gelten, dass sie das `Comparable`-Interface implementieren. `Tripel` soll einen öffentlichen Konstruktor haben, der alle drei Instanzvariablen als Parameter erwartet. Andere Konstruktoren oder Methoden brauchen für `Tripel` nicht programmiert werden.

Zu der Klasse `Tripel` soll eine zweite Klasse namens `TripelComparator` im selben Package hinzukommen. Diese Klasse soll das `Comparator`-Interface implementieren und ermöglichen, Objekte vom Typ `Tripel` nach folgenden Kriterien zu vergleichen:

1. zuerst nach `u` aufsteigend, dann nach `v` aufsteigend und dann nach `w` aufsteigend
2. zuerst nach `u` absteigend, dann nach `v` absteigend und dann nach `w` absteigend
3. zuerst nach `v` aufsteigend, dann nach `u` aufsteigend und dann nach `w` aufsteigend
4. zuerst nach `v` absteigend, dann nach `u` absteigend und dann nach `w` absteigend
5. zuerst nach `w` aufsteigend, dann nach `v` aufsteigend und dann nach `u` aufsteigend
6. zuerst nach `w` absteigend, dann nach `v` absteigend und dann nach `u` absteigend

Objekte vom Typ `TripelComparator` sollen „immutable“ sein. Das bedeutet, dass alle Instanzvariablen von `TripelComparator` `final` markiert sein müssen und nur im Konstruktor gesetzt werden sollen. Als Konstruktor ist genau einer mit einem Parameter vorzusehen. Der Parameter muss eine Zahl zwischen 1 und 6 sein und bezeichnet die Sortierreihenfolge nach obiger Liste.

Demonstrieren Sie die Funktion von `TripelComparator` mit jeweils einem Testfall für jedes der sechs Sortierkriterien (jeweils eine Test Methode). Sie können dafür die statische Methode `sort` in der Klasse `java.util.Collections` verwenden.

Aufgabe 2

Eigenschaft eines binären Suchbaumn prüfen

Entwickeln Sie die neue Klasse `AVLTreeNode`. Sie soll von der Klasse `TreeNode<E>` (aus Package `de.hsmannheim.inf.pr2.ads`) erben (ist in Moodle vorhanden).

```
public class TreeNode<E> {
    protected E value; // Wert des Knotens.
    protected TreeNode<E> left; // Linker Teilbaum.
    protected TreeNode<E> right; // Rechter Teilbaum.
    // ...
}
```

`AVLTreeNode` muss eine generische Klasse sein. Da es sich um einen Suchbaum handelt, muss (im Gegensatz zu `TreeNode`) der Typ-Parameter *„bounded“* sein.

Programmieren Sie eine Methode namens `isSearchTree()` für `AVLTreeNode<E>`, die erkennt, ob ein Baum die Eigenschaft eines binären Suchbaums erfüllt. Wenn die Methode an einem Objekt `n` vom Typ `AVLTreeNode<E>` aufgerufen wird, muss geprüft werden, ob der Baum, dessen Wurzel `n` ist, ein geordneter binärer Suchbaum ist. Falls das für den ganzen Baum mit der Wurzel `n` gilt, soll das Ergebnis `true` sein, sonst `false`.

Aufgabe 3
AVL-Kriterium bei binärem Baum prüfen

Programmieren Sie eine Methode namens `isAVLTree()` für `AVLTreeNode<E>`, die erkennt, ob ein Baum binärer Suchbaum ist und das AVL-Kriterium erfüllt. Wenn die Methode an einem Objekt `n` vom Typ `AVLTreeNode<E>` aufgerufen wird, muss geprüft werden, ob der Baum, dessen Wurzel `n` ist, an allen Knoten das AVL-Kriterium erfüllt. Falls das für den ganzen Baum mit der Wurzel `n` gilt, soll das Ergebnis `true` sein, sonst `false`.

Anstatt eine Instanzvariable (z. B. `height`) an jedem `AVLTreeNode` zu verwalten, sollten Sie der Einfachheit halber eine Hilfsmethode `height()` programmieren.

Andere Methoden – insb. zum Einfügen und Löschen von Elementen – brauchen für diese Aufgabe nicht umgesetzt werden.

Aufgabe 4
Heap-Eigenschaft bei binärem Baum prüfen

Entwickeln Sie die neue Klasse `HeapTreeNode`. Sie soll von der Klasse `TreeNode<E>` (aus Package `de.hsmannheim.inf.pr2.ads`) erben (ist in Moodle vorhanden, s. Aufgabe 2). `HeapTreeNode` muss eine generische Klasse sein. Da es sich um einen partiell geordneten Baum handelt, muss (im Gegensatz zu `TreeNode`) der Typ-Parameter *„bounded“* sein.

Programmieren Sie eine Methode namens `isMinHeapTree()` für `HeapTreeNode<E>`, die erkennt, ob ein Baum die Ordnungseigenschaft eines Min-Heap-Baums erfüllt (Richtung: Wurzel hat kleinsten Schlüssel im Baum). Die Balanciertheit braucht hier nicht geprüft werden. Wenn die Methode an einem Objekt `n` vom Typ `HeapTreeNode<E>` aufgerufen wird, muss geprüft werden, ob der Baum, dessen Wurzel `n` ist, ein geordneter Heap-Baum ist. Falls das für den ganzen Baum mit der Wurzel `n` gilt, soll das Ergebnis `true` sein, sonst `false`.

Analog dazu soll auch die Methode `isMaxHeapTree()` programmiert werden, die erkennt, ob ein Baum die Eigenschaft eines Max-Heap-Baums erfüllt (Richtung: Wurzel hat größten Schlüssel im Baum).

Aufgabe 5
Array für binären Heap-Baum liefern

Programmieren Sie eine Methode für `HeapTreeNode<E>`, die eine `ArrayList` des Baums liefert, an dessen Wurzel diese Methode aufgerufen wird. Alle Elemente aus dem Baum sollen in der `ArrayList` enthalten sein und die `ArrayList` soll genau so groß sein, wie Elemente in dem Baum sind. Die Reihenfolge der Elemente in der `ArrayList` soll dem des Heap-Arrays aus der Vorlesung entsprechen.

Es muss nicht geprüft werden, ob der Baum die Heap-Eigenschaften erfüllt. Die Methode muss nur korrekt arbeiten, wenn der Baum die Heap-Eigenschaften erfüllt.

Aufgabe 6
Iterator eines Heap-Baums liefern

Machen Sie `HeapTreeNode<E>` iterierbar, indem Sie das Interface `Iterable<E>` implementieren. Verwenden Sie die Methode `array` aus Aufgabe 5.

Aufgabe 7

Methode zur Erzeugnung zufälliger Strings fester Länge

Programmieren Sie eine statische Methode zur Erzeugung zufälliger Strings einer festen Länge, die als Parameter für die Methode angegeben werden soll. Wenn Sie wollen, können Sie eigene Regeln definieren und umsetzen, um aussprechbare Wörter zu erzeugen. Alternativ können Sie auch ein (hinreichend großes) Wörterbuch verwenden. Erlaubt sind aber auch unaussprechbare Kauderwelsch-Wörter wie „bxprnlyes“.

Aufgabe 8

Programm für Laufzeitvergleich unterschiedlicher `Set`-Implementierungen entwickeln

Entwickeln Sie ein Programm, mit dem Sie die Laufzeit der `Set`-Implementierungen `java.util.TreeSet` und `java.util.HashSet` vergleichen können:

- Fügen Sie dem jeweiligen `Set` eine parametrisierbare Anzahl N von Wörtern gleicher Länge (z. B. 8 Zeichen, s. Aufgabe 7) hinzu. Messen Sie die Zeit, die dies dauert, mit der Funktion `System.currentTimeMillis()`.
- Fügen Sie dem jeweiligen `Set` eine parametrisierbare Anzahl $N \times 2$ von Wörtern gleicher Länge (z. B. 8 Zeichen, s. Aufgabe 7) hinzu. Messen Sie die Zeit, die es dauert, für N Wörter gleicher Länge zu prüfen, ob es jeweils in dem `Set` enthalten ist.

Aufgabe 9

Laufzeitvergleich `Set`-Implementierungen durchführen und diskutieren

Führen Sie den Laufzeitvergleich für `add` und `contains` bei den `Set`-Implementierungen `java.util.HashSet` und `java.util.TreeSet` für die $N = 100, 1000, 10000, 100000, 100000$ durch. Stellen Sie die Ergebnisse grafisch dar und interpretieren Sie das Ergebnis in wenigen Sätzen.

Geben Sie ein Textdokument ab, aus dem die von Ihnen ermittelten Laufzeiten für `add` und `contains` für `TreeSet` und `HashSet` hervorgehen. Das Dokument soll auch eine grafische Darstellung der Messwerte und einen kurzen Text mit der Interpretation der Ergebnisse beinhalten.

Aufgabe 10

Live-Testat

Wird während des Testats bekannt gegeben.