COMP09044 Algorithms and Collections Coursework 2017/18 Part B

Completing and Experimenting with the BinarySearchTreeArray<E> class

Note that this coursework should be completed individually. Unlike part A, there is relatively little programming in this assignment and the main task is to test and reflect on the relative performance of three Set classes. Please read this project specification carefully before starting the work.

It is intended to contribute to the assessment of the following three learning outcomes:

L2. Design and develop implementations of collection classes, applying the principles of object-oriented programming and conforming to the relevant conventions, standards and accepted best practice in the programming language used.
L3. Identify and implement algorithms, using recursion where appropriate, for a variety of technically demanding tasks including explorations of graphs and search spaces to find and/or arrive at a destination or goal state.
L4. Recognise the implications for type safety and the guaranteeing of object invariants that arise when trying to meet requirements for persistence, and in designing classes for serialization and de-serialization take full account of these.

Completing the BinarySearchTreeArray<E> class

You will have received detailed feedback on your submission for part A. It is recommended that you go through the comments you have received and the published solution and make any corrections to your `BinarySearchTreeArray<E>` **class that are necessary to address the points raised. At the end of this process you should have a fully functional implementation of the class, based on what you submitted for part A, that you can use as the basis for part B of the coursework but, if not, you may use the published solution for the** `BinarySearchTreeArray<E>` **class as the basis for this project.**

Some Experiments with BSTs

This part of the coursework involves conducting some experiments on binary search trees using the `BinarySearchTreeArray<E>` **class that was the subject of part A of the project.**

The Structure of the Trees

The *binary tree theorem* relates the number of elements in a binary tree to the number of leaves in the tree and to the height of the tree. The theorem states these relationships to be:

$$\text{leaves}(t) <= (n(t) + 1)/2.0 <= 2^{\text{height}(t)}$$

If t is a two-tree then leaves(t) equals (n(t) + 1)/2.0, if t is full then (n(t) + 1)/2.0 equals 2$^{\text{height(t)}}$.

For sets implemented using the `BinarySearchTreeArray<E>` **class and the red-black tree of the** `java.util.TreeSet<E>` **class, you will explore how the number of elements in each set relates to the height of the tree, and for the** `BinarySearchTreeArray<E>` **class you will also explore how the number of leaves in the tree relates to the number of elements, in both cases using trees constructed with sequences of randomly generated positive integer values.**

`TreeSet<E>` **does not define a method to return the height of its tree. This can be inferred, however, from the number of comparisons required to find the element that is most distant from the root, and this same approach can be used to find the height of an instance of the** `BinarySearchTreeArray<E>` **class. Here is a method, using an** `Item` **class that counts comparisons (discussed later in this project specification), that returns the height of a binary search tree:**

```
private static long height (Set<Item> tree) {
     long maxComp = 0;
     for (Item current : tree) {
          Item.resetCompCount();
          tree.contains(current);
          if (maxComp < Item.getCompCount()) {
               maxComp = Item.getCompCount();
          }
     }
     return maxComp-1;
}
```

For the `BinarySearchTreeArray<E`**> class to determine the number of leaves you can add the following method to your** `Entry<E>` **class (this assumes you have declared a constant, equal to -1, to represent a "null reference" – here this has been called NIL):**

```
    public boolean isLeaf() {
      return left == NIL && right == NIL;
    }
```

and the following methods to the `BinarySearchTreeArray<E>` **class:**

```
    private int countLeaves(int nodeIndex) {
        if (nodeIndex == NIL) return 0;
        Entry<E> node = tree[nodeIndex];
        if (node.isLeaf()) return 1;
        int count = countLeaves(node.left);
        count += countLeaves(node.right);
        return count;
    }

    protected int leaves() {
        return countLeaves(root);
    }
```

Note that the `countLeaves()` method above is recursive. You may prefer to use an iterative implementation. In your report of the work, discuss how it, and any recursive methods in the original `BinarySearchTree<E>` class, could be implemented iteratively, and whether for any of these methods you did, in the final version of the `BinarySearchTreeArray<E>` class for this assignment, adopt an iterative or recursive implementation of the methods.

<u>The Performance of three Set classes in Searches</u>

You will also investigate the number of comparisons necessary to search for items in such trees and contrast these with each other and with the comparisons required for searching the hashtable `Set<E>` **implementation in** `java.util.HashSet<E>`**.**

Finally, in terms of these experiments, you will investigate the effect on how balanced the trees are, and on the number of comparisons required for searching for items, after large numbers of insertions and deletions have been executed for the two tree classes.

To count the number of comparisons required when performing operations on the Sets such as adding an item, removing an item and searching for an item you should write an application that uses the following class (published in the Assignments area on Moodle) as the element type for the three `Set<E>` **classes:**

```
import java.io.Serializable;
public class Item implements Comparable<Item>,
                                   Serializable {

     /**
      * Provides an immutable integer valued item that
      * counts comparisons.
      */

     private static long compCount = 0;

     private final Integer value;

     /**
      * Constructor creates an Item and sets its value
      * @param value the value for this Item
      */
     public Item(Integer value) {
          this.value = value;
     }

     /**
      * The value of this Item
      * @return the Item's value
      */
     public Integer value() {
          return value;
     }

     /**
      * Compares the value of this Item with that of
      * other according to the contract for Comparable.
      * Increments the count of comparisons.
      */
     @Override
     public int compareTo(Item other) {
          compCount++;
          return value.compareTo(other.value);
     }

     /**
      * Returns the total number of comparisons performed
      * on instances of type Item since the counter was
      * last reset (or the total if it has never been
      + reset).
      * @return the count of calls to compareTo() and
      * equals for type Item
      */
     public static long getCompCount() {
          return compCount;
     }
```

```
    /**
     * Resets the count of comparisons to zero.
     */
    public static void resetCompCount() {
        compCount = 0;
    }

    @Override
    public String toString() {
        return value.toString();
    }


    @Override
    public int hashCode() {
        return value.hashCode();
    }

    @Override
    public boolean equals(Object that) {
        compCount++;
        if (this == that) return true;
        if (!(that instanceof Item)) return false;
        Item other = (Item)that;
        return value.equals(other.value);
    }
}
```

To use the `Item` **class the basic approach is illustrated below (where** `elements` **is a** `List<Item>` **containing the list of items to search for, in this case, where each search of the tree should successfully find the item, having the same set of items as the** `BinarySearchTreeArray<Item>`**,** `bst`**):**

```
    Item.resetCompCount();
    int errorCount = 0;
    for (Item current : elements) {
        if (!bst.contains(current))
            errorCount++;
    }
    long comparisonsBST = Item.getCompCount();
    Item.resetCompCount();
```

To investigate the impact of a poor hashCode() function on HashSet, repeat at least some of the experiments using this class:

```
public class BadItem extends Item {

    public BadItem(int value) {
        super(value);
    }

    @Override
    public int hashCode() {
        return super.hashCode() % 10;
    }

}
```

You will not need to change the declarations of the three sets or any lists that you use (i.e. they can still be declared as `Set<Item>` and `List<Item>`) to add instances of `BadItem` to the collections as a `BadItem` is an `Item`.

Note that the `getEntry()` method of `HashSet<E>` compares the references to objects that have the same hash code using the == operator and only calls the `equals()` method of the objects if their hash code is the same but the references to them are different. In testing `HashSet`, therefore, you should ensure that a search does involve a call of the `equals()` method of the `Item` being searched for or the comparison will not be counted. This can be achieved by ensuring that a successful search involves using an `Item` with the same value as one in the set, but does not involve using the actual instance of `Item` that you added to the set as the parameter to methods such as `contains()` or `remove()`.

Outline of the Experiments

In your investigations you should write an application that creates an instance of your `BinarySearchTreeArray` class and also create an instance of `java.util.TreeSet` and of `java.util.HashSet`. In all three cases, the element type should be `Item`.

In your experiments, add the same sequence of `Item`s, or in the repeat, the same sequence of `BadItem`s (using randomly generated positive odd values) to each of the three sets. Once the sets have been built, determine the heights of the trees for the two tree-based sets and the number of leaves in your binary search tree class. Count the number of comparisons required to find all of the `Item`s in each set. Search the three sets for the same number of even valued `Item`s (or `BadItem`s), so that each search is unsuccessful, and again count the number of comparisons that are

required (*e.g.* for sets of size 15, say, search for 15 even values that fall within the same overall range of values as the values added to the sets).

Repeat these experiments with different numbers of elements, across a good range of sizes (up to at least tens of thousands). Plot the heights of the trees and the number of leaves in the array-based binary search tree against the size of the trees. Also plot average number of comparisons required for the successful and unsuccessful searches against the size of the sets. You can use an application such as Excel to plot the data (in the first lab, we produced comma-delimited files that Excel could directly open, you may wish to do this). In your report, discuss the results in the light of the binary tree theorem and the extent to which they confirmed or otherwise what you expected to find.

To explore how binary search trees perform over time, you should investigate and comment on the effect, if any, on the structure of the trees after large numbers of insertions and deletions have been performed on each of the tree-based sets – once you have filled the set keep the average size of the set over the experiment roughly constant. That is, keep the overall number of deletions and insertions roughly equal – for example you might delete a fifth of the elements, then add the same number of elements (not the same values that you deleted but randomly generated values) repeating this, say, 10 times and then measure the heights of the trees and the number of leaves again. Is any difference in the average number of comparisons when searching for items observed after such deletions and insertions, compared with what you found in the initial experiments? Discuss whether you expect the trees to become progressively more unbalanced over time as items are removed and new items added and whether your observations confirmed this expectation. For these experiments, you need not repeat the runs with `BadItem`**, as the focus is on the two tree-based sets.**

You will receive credit for devising any additional experiments or measures to complement those asked for above in helping to interpret them, possibly involving adding more protected methods to the `BinarySearchTreeArray<E>` **class additional to those required for the** `Set<E>` **interface. For example (but these are by no means the only possibilities), you might investigate performance when items are not added in random sequence but in ascending or descending order. Or it might be useful to provide a method to calculate the external path length of the tree and relate this to the comparison counts you obtained when searching for items and/or to**

the *external path length theorem* we discussed in the lecture on binary trees.

Making the BinarySearchTreeArray<E> class serializable

Modify the `BinarySearchTreeArray<E>` **class so that it implements the** `Serializable` **interface. Test that instances of the tree can be serialized to file and successfully recreated from file. You should refer to the notes in the Serialization subpage on Moodle for information on how to update a class to be serializable while making it tolerant to version changes and ensuring that the runtime and serialized forms of the type are only loosely coupled. You may assume that any element type for the tree implements** `Serializable` **(as does the** `Item` **class above).**

Reporting your results

Produce and submit an individual report on the work covering the experimental results and how the type was updated to be serializable. You should include a critical appraisal of your work. Note that the report must be included. For parts of the task that involve your comparing things and discussing things (such as the recursive and iterative implementation of `countLeaves()`**), the comparison and the discussion should be included in the report.**

Summary and Marking Scheme

1. **Part B (Weighted at 30% of module marks)**
   a. **Write an application to investigate the properties and performance of the** `BinarySearchTreeArray` **class and of** `TreeSet` **and** `HashSet` **as outlined above. [25 marks]**
   b. **Amend the** `BinarySearchTreeArray` **class so that instances of the class can be serialized and deserialized. Marks will be based on this successfully working and on the extent to which the type controls its serialization so that the runtime and serialized forms of the type are only loosely coupled. [25 marks]**
   c. **A report including presentation and discussion of experimental method and results, discussion of use of recursion, summary of how you implemented serialization, and a critical appraisal of the work. [50 marks]**

Note that, as with part A, *you must include an acknowledgement in your report for any sources you have used (on the web, textbooks etc) in developing or commenting on design, code and in your testing, and any source code used from sources other than the*

course textbook should be indicated in the code by a comment at the beginning and end of the code stating the source from which it was taken. If you receive any advice from anyone you should acknowledge this also and indicate what part of the work it related to. You should not share any of the application code (i.e. your program to run the experiments on the three Set classes), or the code for serializing the array-based tree class, that you produce with anyone.

The feedback due date is Wednesday 18th April 2018.

What You Should Submit

Submit all the Java source code that you have written (you do not need to upload the Netbeans, IDEA, or Eclipse project files – just the source code files will be fine) in a zip file which also contains your documentation and testing results. Include your banner ID but not your name. Submit all the files for your submission as a zip file named XXXX.zip (where XXXX are the last four digits of your Banner ID) using the link for part B in the Assignments section for this module on Moodle, by 2355 on Monday 26th March 2018.