Lab 5

In this lab, you will continue implementing the RecursiveBag class we’ve been working on the last couple of periods.

Recursive Bag

Begin by downloading the code we’ve been working on from the course webpage (http://courses.knox.edu/cs142, under “Assignments”: it’s also displayed as the code from yesterday’s lecture). The given code is packaged as a .zip file, which you should download and import into Eclipse. (Remember to do so as an Existing Project rather than an archive.) I called the project RecursiveBag.

Begin with the toString method that we were working on in class yesterday. Create a JUnit Test Case for it. Specifically, create a bag, put some items in it, and then give an assertion that toString returns something of the form [hi, hello, =), hi]; i.e. a comma-separated list of values surrounded by square braces. The test case will fail until you do the following two things to complete the toString methods:

1. The first problem is that our methods put a trailing comma after the last value. This occurs in the DataNode.toString method, which adds a comma after each value. Instead, it should only add a comma after its value if next refers to another DataNode and not to an EmptyNode. You can do this with the following test:

   ```java
   if(next instanceof RecursiveBag.DataNode)
   //add the comma
   ```

2. The second problem is the lack of square braces. Add these in the RecursiveBag.toString method.

After you have toString working, write a method removeAll that removes all occurrences of its argument rather than just one of them. This will be similar to the remove method that we wrote in class, but needs to always recurse in order to remove multiple occurrences. Set this method up similarly to how we did remove; it should return a Node reference to the beginning of the new bag. In order to use it, remember that you need to write removeAll methods for RecursiveBag, DataNode, and EmptyNode, plus adding the signature to the Node interface.

For the next couple of methods, let’s write variations of add that worry about the order of elements. (As I’ve said before, this is violating the idea of the Bag ADT, but the order is of interest for many applications and it is maintained in some related ADTs.) Begin by writing a method addAfter that takes two arguments, adding the first argument to the bag after the first occurrence of the second argument. If the second argument doesn’t appear in the bag, then addAfter does nothing. You’ll need to create test cases for this; use assertions involving toString so you can observe the order in which values are stored.

Once you have addAfter, a slight variation of it should give you addBefore, which adds a new value immediately before the first occurrence of a given value.

Then write a version of add that takes a number and the value to add. The number is the index at which the value should be appear in the bag after the method is complete. For example, calling add(0, "Hi") would add the value "Hi" to be the new first value in the bag, add(1, "Hi") would place it immediately after the current first element, etc. If the index is invalid (i.e. negative or greater than the number of values in the bag), the method should throw an IndexOutOfBoundsException. Again, the Node versions of this method should be structured to return a reference to the new front of its list. To handle the indices, the method will want to decrement its numeric argument each time it makes a recursive call (index k relative to a particular node is index k – 1 relative to the next node since that node is the front of a list that has one fewer element).
If you complete those, try to write some other methods that use indices:

- **get(int)**, a method that takes an integer index and returns the value at that position. (Again, throwing an `IndexOutOfBoundsException` if the index is invalid.)

- **indexOf(T)**, which takes a value and returns the index at which it occurs in the bag. The method returns -1 if the value does not appear.

If you have additional time, try to write other useful list methods using recursion (toArray, replace, ...).