Lab 3 Remove Duplicates from an ArrayBag

Goal

In this lab you will explore the implementation of the ADT bag using arrays. You will take an existing implementation and create a new method, `removeDuplicates`, that will guarantee that each item in the bag occurs only once by removing any extra copies.

Resources

- Appendix D: Creating Classes from Other Classes
- Chapter 1: Bags
- Chapter 2: Bag Implementations That Use Arrays

In `javadoc` directory

- *`BagInterface.html`*—Documentation for the interface `BagInterface`

In addition, there are also some online resources that you might find useful. The place to start for all things Java is www.oracle.com/technetwork/java/index.html. Here you can find links to free downloads like Java SE (The standard edition of the Java Platform). You can also find links to tutorials and documentation for Java. One useful starting point is the API documentation that documents all of the standard classes and their methods. At the time this was written, the most recent version of the standard edition was Java SE 9 and the API documentation was at docs.oracle.com/javase/9/docs/api. Please refer to the Java website for the most recent version.

Java Files

- *`ArrayBag.java`*
- *`ArrayBagExtensionsTest.java`*
- *`BagInterface.java`*

Directed Lab Work

The `ArrayBag` class is a working implementation of the *`BagInterface.java`*. The removeDuplicates method already exists but does not function yet. Take a look at that code now if you have not done so already.

Remove Duplicates

Step 1. In the `removeDuplicates` method of `ArrayBag`, implement your algorithm from the pre-lab exercises. This method will require some form of iteration. If you use the technique recommended in the pre-lab, you will use nested iteration.

Final checkpoint: Compile and run ArrayBagExtensionsTest. All tests should pass. If not, debug and retest.

Post-Lab Follow-Ups

1. Implement the `removeDuplicates` method using the private method `removeEntry`.

2. Implement the `removeDuplicates` method using just methods from the `BagInterface` along with a second bag.

3. Implement and test a new method

   ```
   boolean splitInto(BagInterface<T> first, BagInterface<T> second){
   ...
   }
   ```

   which will split and add the contents of the bag into two bags that are passed in as arguments. If there are an odd number of items, put the extra item into the first bag. The method will return a boolean value. If either bag overflows, return false. Otherwise, return true. Note that while you will directly access the array of the bag that the method is applied to, you can only use the methods from `BagInterface` on the arguments.

4. Implement and test a new method

   ```
   boolean addAll(BagInterface<T> toAdd){
   ...
   }
   ```

   which will add all of the items from the argument into the bag. The method will return a boolean value indicating an overflow. If adding the items would cause the bag to overflow, do nothing and return false. Otherwise, add the items and return true. Note that while you will directly access the array of the bag that the method is applied to, you can only use the methods from `BagInterface` on the argument.

5. Implement and test a new method

   ```
   boolean isSet(){
   ...
   }
   ```

   which will return true if the bag is also a set (has no duplicates).

6. Implement and test a new method

   ```
   T getMode(){
   ...
   }
   ```

   which will return the item with the greatest frequency. If there isn't a single item with the greatest frequency, return `null`.

Another way of implementing a bag is not to have a reference to each duplicate, but instead only keep a single reference and a count of the duplicates. For each of the remaining questions, use this new implementation.

7. Define an inner class `Node` so that it has a count of the number of times an item is in the bag and then change the implementation of each of the methods in the bag interface. For example, when you add an item, first check to see if it is already in the bag. If so, just increment the count in the node. If it is not in the bag, add a new node with a starting count of one.

8. Redo the removeDuplicates methods from the lab.

9. Implement another version of the remove method that removes a randomly selecterd entry from the gab instead of an arbitrary one. Warning: The implementation of the randomized remove method is tricky. If you have a bag that contains eight "a"s and four "b"s, remove should be twice as likely to pick an "a" as it is to pick a "b".