Dual Implementation Bag ADT Assignment

This assignment was jointly created by Eduardo and Raunak. Kindly direct any queries about the assignment to exb406@case.edu and rxb641@case.edu (make sure to include both in the email so that either one of us can get back to you).

For this assignment, there will be 2 *additional* office hours on top of the usual TA office hours:

- Eduardo – Monday February 5th from 6 PM – 7PM on Zoom (meeting ID: 366 362 9703, passcode: 529155)
- Raunak – Wednesday February 7th from 6 PM – 7PM at Millis Schmitt Lecture Hall

This programming assignment involves implementing the `**DualImplementationBag**` ADT in Java, which allows switching between two internal data structures, `ArrayList` and `LinkedList`. You may use the in-built lists found in java and need not implement your own. The main task is to create a class that can efficiently perform standard Bag operations like add, remove, and check for an element's presence, using either of the two structures. Additionally, the assignment includes creating the `**DemoBag**` class to add functionalities to the `DualImplementationBag` class.

Your ADTs should use the **exact names** specified below, including the case to ensure your code compiles against the grading test set.

Classes Overview:

1. `**DualImplementationBag<T>**`: The main class representing the Bag ADT with two internal data structures (ArrayList and LinkedList).

2. `**DemoBag**`: A class to demonstrate and test the functionalities of `DualImplementationBag<T>`.

`DualImplementationBag<T>` Class:

Fields:
- `useArrayList`: Boolean flag to choose between ArrayList and LinkedList.
- `internalList`: List<T> to store the bag's elements.

Constructor:
- `DualImplementationBag(boolean useArrayList)`: Initializes the bag with either an ArrayList or LinkedList.

Core Methods:
- `setUseArrayList(boolean useArrayList)`: Switches the internal data structure.
- `add(T element)`: Adds an element. Parameter: `element` to add.
- `remove(T element)`: Removes an occurrence of an element. Parameter: `element` to remove.

- `contains(T element)`: Checks if an element is in the bag. Parameter: `element` to check.
- `isEmpty()`: Checks if the bag is empty.
- `size()`: Returns the number of elements.
- `getFrequencyOf(T element)`: Counts occurrences of an element. Parameter: `element` to count.
- `get(int index)`: Gets the element at the given index. Parameter: `index` to find

`DemoBag` Class:

-**Methods**:
- `removeAll(DualImplementationBag<T> bag, T element)`: Removes all occurrences of an element. Parameter: `element` to remove completely.
- `retainAll(DualImplementationBag<T> bag, T element)`: Retains only elements equal to the specified element. Parameter: `element` to retain.
- `union(DualImplementationBag<T> otherBag1, DualImplementationBag<T> otherBag2)`: Combines contents with another bag. Parameter: `otherBag` to unite with.
- `intersection(DualImplementationBag<T> otherBag1, DualImplementationBag<T> otherBag2)`: Creates a bag with elements common to both bags. Parameter: `otherBag` for intersection.
- `difference(DualImplementationBag<T> otherBag1, DualImplementationBag<T> otherBag2)`: Forms a bag with elements in this bag but not in the other. Parameter: `otherBag` to differentiate from.

Classes Interaction:
- `DemoBag` uses instances of `DualImplementationBag<T>` to demonstrate each method, showing the bag's behavior with different internal structures.

Return Types:
Here are the return types for each method:

- `setUseArrayList(boolean useArrayList)`: **void** - No return value, just sets the internal data structure.
- `add(T element)`: **boolean** - Returns `true` if addition is successful.
- `remove(T element)`: **boolean** - Returns `true` if the element was found and removed.
- `contains(T element)`: **boolean** - Returns `true` if the element is found in the bag.
- `isEmpty()`: **boolean** - Returns `true` if the bag is empty.
- `size()`: **int** - Returns the number of elements in the bag.
- `getFrequencyOf(T element)`: **int** - Returns the count of how many times an element appears in the bag.
- `get(int index)`: **T** – returns the element at the index
- `removeAll(DualImplementationBag<T> bag, T element)`: **void** - Removes all occurrences of an element, no return value.
- `retainAll(DualImplementationBag<T> bag, T element)`: **void** - Retains only elements equal to a specified element, no return value.

- `union(DualImplementationBag<T> otherBag1, DualImplementationBag<T> otherBag2)`: **DualImplementationBag<T>** - Returns a new `DualImplementationBag<T>` that is the union of two bags.
- `intersection(DualImplementationBag<T> otherBag1, DualImplementationBag<T> otherBag2)`: **DualImplementationBag<T>** - Returns a new `DualImplementationBag<T>` that contains only elements common to both bags.
- `difference(DualImplementationBag<T> otherBag1, DualImplementationBag<T> otherBag2)`: **DualImplementationBag<T>** - Returns a new `DualImplementationBag<T>` that contains elements in the first bag but not in the second.

Comments:

Please provide sufficient comments in your source code to help the TAs read it.
Comments should aid the reader to understand the code. Comments that restate what is already clear from the code are redundant and not helpful. Nor are comments that are not consistent with the code.

Testing:

You need to use JUnit tests to verify correctness.

The tests should be well thought-out and have clear reasoning behind them, which you should explain in your comments. Having a large number of tests that are essentially equivalent will receive deductions. The goal of testing is to make the process of implementation and debugging faster and more efficient, so don't waste your time writing a bunch of unnecessary tests.
Here are some guidelines from the *Practice of Programming*:

• Test as you write the code. This goes along with writing incrementally. Once you implement a part of the assignment, write a test to check its correctness.

• Test code at its boundaries. This is also referred to as testing "corner cases." Do conditions branch in the right way? Do loops execute the correct number of times? Does it work for an empty input? A single input? An exactly full array? Etc.

• Test systematically. Does your test code check every condition? Does every line get executed? Especially test parts you're going to reuse. Otherwise, debugging can be much more difficult.

• Test automatically. Have test suite that automatically runs all the tests. JUnit helps provides this functionality. This helps identify errors as soon as they are introduced, which would otherwise can be very difficult to debug.

• Stress test. Does your code work on large inputs? Random inputs? It's easy to make implicit assumptions that aren't always valid. Stress testing can help uncover these.

Submission:

Please generate a single zip file containing all your *.java files and Junit tests needed for this assignment. Name your file **'P1_YourCaseID_YourLastName.zip'**. Submit your zip file electronically to Canvas.

Rubric for Dual Implementation Bag ADT Assignment:

Total Points: 100

1. Correctness and Functionality (40 points):
   - Core Methods Implemented Correctly: 20 points
   - Extended Methods Function Correctly: 20 points

2. Efficiency (20 points):
   - Efficient Implementation of Core Methods: 10 points
   - Efficient Implementation of Extended Methods: 10 points

3. Code Quality (20 points):
   - Readability and Structure: 10 points
   - Proper Documentation and Comments: 10 points

4. Demonstration and Testing in `DemoBag` (20 points):
   - Comprehensive Testing of All Methods: 10 points
   - Effective Demonstration of Dynamic Structure Switching: 10 points. When `setUseArrayList(boolean useArrayList)` method is called, it should ensure that the current state of the bag (i.e., all its elements) is preserved while transitioning between the data structures.

This rubric is based on the aspects of correctness, efficiency, code quality, and demonstration. The aim is to encourage not just functional code, but also efficient, well-documented, and clearly structured programming practices.