CS 2S03: Principles of Programming

Due on November 23rd

Dr. Jacques Carette

1 Goals

The goals of this assignment are:

1. work on dynamic data-structures
2. learn about interfaces and part of Java's collections

2 The Task

You will give actual implementations of the data structures (as given by interfaces) *Stack*, *Queue* and *PriQueue*.

2.1 The Interfaces

On the assignment page, you will find the Java files `MyStack.java`, `MyQueue.java` and `MyPriorityQueue.java` which are *interfaces* that you must implement. The names have had "My" prepended to them so as to not clash with similar items in Java's own standard library.

2.2 The implementations

You will implement:

1. A class `ListChar` that implements a (linked) *list of characters*. You will make sure `ListChar` implements the `MyStack` interface. You have to implement the linked list yourself (but feel free to use any code from class, the slides, or the textbook).
2. A class `StackChar` which internally uses Java's `ArrayList` to implement the `MyStack` interface.
3. A class `SnocList` of reversed lists of characters (see next section for details).
4. A class `SnocQueue` which internally uses your `SnocList` and implements the `MyQueue` interface.
5. A class `AList` of *attributed lists* (see below for details).
6. A class `AListPQueue` which uses `AList` internally and also implements the `MyPriorityQueue` interface.

For each of the above, you should implement (override) the following methods:

- `equals`, for testing equality of two data structures. The autograder will use this method for testing, make sure it is right! The signature **must** be

```
public boolean equals(Object l) {
    // your code here
    // compares l to 'this' (current Object)
}
```

- `toString`, for a visual representation of a data structure. You will use this method in your own tests. This is supposed to return a String which is an accurate representation of *the full internal state* of your data structure. Different structures should have different String representations, and `equal` structures should have the *same* String representation.

2.3 SnocList

A `SnocList` is a linked-list in reverse order: when you create a new node, it goes at the *end* of the list. In other words, you start with

```
class SnocList {
  private char c;
  private SnocList l;
  SnocList(char c, SnocList l) { this.c = c; this.l = l }
}
```

but `new SnocList('a', new SnocList('p', new SnocList('p',null)))` represents the list p, p, a. You should add additional methods to this, as suits your purposes.

2.4 AList

An `AList` is a linked-list with an extra integer, which is regarded as an "attribute" of the list. Even though it is called `priority`, this is *just a list.* All of the extra structure has to be implemented *above* it. Roughly:

```
// class for a list where each node is decorated with an integer.
class AList {
  private int hd;
  private int priority;
  private AList tl;
  AList(final int a, final int b, final AList ll) {
    this.hd = a;
    this.priority = b;
    this.tl = ll; }

  // you may implement some additional helper routines here, but
  // they should not implement functionality related to the
  // ``meaning'' of 'priority'
}
```

2.5 Testing

As with all other assignments, testing is important. Thus for each class

- For each of `ListChar, StackChar, SnocQueue, AListPQueue,` create 'scenarios' of uses (i.e. for Stack, sequences of push/pop/top/isEmpty calls).
- You should create 10 scenarios for each of `ListChar, StackChar, SnocQueue, AListPQueue`. 3 of your test cases should throw exceptions (which your JUnit tests should test for). Another 3 should involve sequences of operations of length at least 15.
- For the non-exception tests, you should be testing against your `toString` routine and an expected output.
- Your test class should be called `TestingA4`.
- Unlike previous assignments, a test should contain *exactly one* assertion. In other words, you should actually have 40 *separate tests*, where each test is a **single assertion**. So as to minimize code duplication, you should look into Test Fixtures.

- It is a good idea to test all your classes and helper routines; additional tests are encouraged.

2.6 Notes

- Make that your implementations do not 'leak' the details of their internal data representation. In other words, the internal data should be declared `private`.
- If an action to be taken is not legal (like looking at the top of an empty stack), you should throw an exception. See the provided class `EmptyContainerException`.
- For `void` methods, such as popping an empty stack or deleting the highest priority item of an empty Priority Queue, just do nothing.
- **Important**: For your priority queue, your elements should actually be stored (internally) in priority order, with higher numbers indicating higher priority. Equal priorities should be sorted alphabetically, so that `a` is higher priority than `b` (use Java's `compareTo` method on strings).
- We will use your `equals` method in the autograder. Make sure you get this one right! [It is in your best advantage to write extra tests which ensure this method works properly].

3 Submission Requirements

- Make sure all your work is under package `cs2s03`.
- A *single* zip file called `a4_<student_number>.zip` containing all your java files, including your JUnit test files.
- Make sure your classes are named as above.
- Extra files are OK.

4 Marking Scheme

- The code will be worth 60%, the tests 40%. Part of the scheme for code and tests will include style marks.

5 Bonus

Each one of these will be worth extra marks:

- (easy) Implement all of the above using Java's generics instead of using 'char' everywhere. Keep priorities as `int`, and assume that the underlying type is `Comparable` for sorting.
- (medium) Implement a `PriQueue` using a doubly-linked circular list.

Remember that, even for the bonus, proper testing is worth 40% !