SE/CS 2S03: Principles of Programming

Due on November 29th

Dr. Jacques Carette

Idea

The goals of this assignment are:

1. work on dynamic data-structures
2. learn about interfaces and part of Java's collections
3. write code+tests

The Task

You will implement `Stack`, `Queue` and `PriQueue` in multiple ways. `SnocList` and `PList` will be defined below.

1. Using your own implementation of a class `List` (of characters), implement a class `Stack` (of characters).
2. Using Java's `ArrayList` class, implement `Stack` (of characters).
3. Using your own implementation of a class `SnocList` (of characters), implement a `Queue`.
4. Using your own implementation of `PList`, implement a `PriorityQueue`.

The details

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.

A `Plist` is a linked-list with an extra integer. 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 also has a priority.
// enforces *no* other invariants.
class PList {
  private int hd;
  private int priority;
  private PList tl;
  PList(final int a, final int b, final PList ll) {
    this.hd = a;
    this.priority = b;
    this.tl = ll; }

  // you may implement some additional helper routines here, but
  // they should not implement 'PriQueue' functionality,
  // just list-with-extra-data functionality
}
```

For each part 1 − 4 above, make sure your internal data-representation (which should be `private`) does not 'leak'. In other words, for part 3, your code would start

```
class Queue {
  private SnocList l;
```

For concreteness, implement the following interfaces:

```
public interface Stack {
  public char top();
  public void pop();
  public void push(char);
  public boolean isEmpty();
  public void show(PrintStream p);
}
public interface Queue {
  public char peek(); // front
  public void dequeue(); // front
  public void enqueue(char); // back
  public boolean isEmpty();
  public void show(PrintStream p);
}
public interface PriQueue {
  public char next(); // highest priority
  public void deleteItem(); // highest priority
  public void insertItem(int, char); // int priority, then alphabetical
  public boolean isEmpty();
  public void show(PrintStream p);
}
```

When the action to be taken is not legal (like looking at the top of an empty stack), throw a (new) exception. 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 equal priorities sorted alphabetically by contents).

The `show` method is for debugging: it should print a human-readable version of what is in your data-structure. Such methods are usually removed from production code.
You will also need to:

- For each item 1 − 4, create 'scenarios' of uses (i.e. for Stack, sequences of push/pop/top/isEmpty calls).
- You should create 10 scenarios for each. 3 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 you `show` routine and an expected output.

Submission Requirements

- A *single* zip file containing all your java files, including your JUnit test files.

Marking Scheme

- Programs which do not compile will be given a mark of 0, no matter how *close* your code might be to the correct answer.
- The code will be worth 60%, the tests 40%.

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 *skip lists* (see Wikipedia for details).
- (medium-hard) Implement a `PriQueue` using a doubly-linked circular list.

Yes, you may do multiple bonus parts. Remember that, even for the bonus, proper testing is worth 40% !