Assignment 7

Priority Queues

Introduction

In Part 1 of this assignment you will be implementing the `PriorityQueue` interface using a generic array-based Heap data structure. A reference-based implementation is provided for you (`LinkedPriorityQueue.java` and `A7Node.java`), so you can run the tester and compare the run times of the two implementations.

In Part 2, you will implement a small application modeling a To-Do List for a work day (called `ToDoList.java`). The `ToDoList` class uses the `HeapPriorityQueue` you implement from Part 1. Although there is a tester provided for this assignment, it does not include a comprehensive set of sets for each method. You should add your own tests to test cases not considered.

Note: The automated grading of your assignment will include different and additional tests to those found in the `A7Tester.java` file. You are expected to write additional tests until you are convinced each method has full test coverage. The `displayResults` and test coverage videos provide more information about code testing.

Objectives

Upon finishing this assignment, you should be able to:

- Write an implementation of an array-based Heap;
- Implement a `compareTo` method to specify the order of two Comparable objects;
- Write code that uses an implementation of the Priority Queue ADT.

Submission and Grading

Attach `HeapPriorityQueue.java`, `Task.java`, and `ToDoList.java` to the BrightSpace assignment page. Remember to click **submit** afterward. You should receive a notification that your assignment was successfully submitted.

If you chose not to complete some of the methods required, you **must** provide a stub for the incomplete method(s) in order for our tester to compile. If you submit files that do not compile with our tester, you will receive a zero grade for the assignment. It is your responsibility to ensure you follow the specification and submit the correct files. Additionally, your code must not be written to specifically pass the test cases in the tester, instead, it must work on all valid inputs. We may change the input values during grading and we will inspect your code for hard-coded solutions. This video explains stubs.

Be sure you submit your assignment, not just save a draft. ALL late and incorrect submissions will be given a ZERO grade. A reminder that it is OK to talk about your assignment with your classmates, but not to share code electronically or visually (on a display screen or paper). We will be using plagiarism detection software.

Overview

The `LinkedPriorityQueue` implementation provided does not make use of the O(logn) heap insertion and removal algorithms we discussed in lecture. Instead, when an item is inserted, the queue is searched from beginning to end until the correct position to insert the item is found.

The `HeapPriorityQueue` insertion implementation you write should improve on this implementation so that the `insert` runs in O(logn) using the `bubbleUp` algorithm covered in lecture. Similarly, your `removeMin` should use the `bubbleDown` algorithm.

Part 2 asks you to complete the implementation of the `Task` and `ToDoList` classes. Using these classes with a priority queue will give you experience building a small application that models a system that helps prioritize daily tasks from a to-do list. In the system, all tasks are given priority depending on how urgent they are. All tasks added to the system are stored in the priority queue such that high priority tasks are handled (i.e., removed from the priority queue) first.

Note that the priority queue in this assignment extends the `Comparable` class. All classes that implement Java's `Comparable` interface must provide an implementation of the `compareTo` method, which is a method that allows us to compare two objects and return a number that represents which of the two should come first when sorted. The `compareTo` method is helpful for a priority queue as it can be used to compare priority values to determine how items should be ordered within the priority queue.

When implemented correctly, an object's `compareTo` method has the following behavior:

– returns 0 if the two objects being compared are ordered equally;

– returns a negative value if this object should be ordered before the other object;

– returns a positive value if this object should be ordered after the other object.

For example:

```
Integer a = 7;
Integer b = 9;
a.compareTo(b); // returns a negative value.
a.compareTo(a); // returns 0

String x = "computer";
String y = "science";
x.compareTo(y); // returns a negative value.
y.compareTo(x); // returns a positive value
```

Part 1 Instructions

1. Download and unzip all of the files found in `a7-files.zip`.
2. The documentation for the methods you will implement are all found in `PriorityQueue.java`. We have also provided some comments in `HeapPriorityQueue.java` to help you with your implementation.
3. Compile and run the test program `A7Tester.java` with the `LinkedPriorityQueue` to understand the behavior of the tester:

   ```
   javac A7Tester.java
   java A7Tester linked
   ```
   Note: the `A7Tester` is executed with the word "linked" as an argument. This argument allows us to run all of the tests against the `LinkedPriorityQueue` implementation provided for you. You will notice it is extremely slow with larger file input sizes (for the largest input size we will only test it with the heap implementation).

4. Compile and run `A7Tester.java` using the `HeapPriorityQueue` implementation:

   ```
   javac A7Tester.java
   java A7Tester
   ```
   (a) Uncomment one of the tests in the tester;
   (b) Implement one of the methods in HeapPriorityQueue.java;
   (c) Once all of the tests have passed compare the runtime against the LinkedList implementation (outlined in step 3).

HINT: The first big thing you need to think about is whether you will store the root element in the heap an index 0 or index 1. We have provided a `rootIndex` field for you that should be set to 0 or 1 depending on what method you choose. This choice will affect how you access parent and child indexes, as well as how you print out the contents of the heap when debugging (you will notice an if-statement in the `toString` provided).

Another thing to think about is creating helper methods to help with the `bubbleUp` and `bubbleDown` methods. We made a number of helper methods in my implementation. The names of our helper methods are `isLeaf`, `minChildIndex`, and `swap`. It might be a good idea to implement your own helper methods (possibly even more than we implemented for our solution).

Part 2 Instructions

For this part of the assignment, you will be creating an application to support the modelling of a to-do list for a worker's daily tasks. You are asked to write the software that will manage handling tasks waiting to be done based on the priority level of their urgency and their insertion time at the to-do list.

Imagine you are a worker adding tasks to a to-do list at a work day. Each time a task is added, it gets a title and priority, and waits in the to-do list until its time to be done comes. You must make sure high priority tasks are done before lower priority tasks.

For example:

- A task (A) is added to the to-do list with a priority level 3 urgency. You add it to the queue.
- A task (B) is added to the to-do list with a priority level 1 urgency. You add it to the queue ahead of the previous task (A), because priority level 1 urgencies must be handled first.
- Another task (C) is added to the to-do list, also with a priority level 1 urgency. You add it to the queue ahead of task A (priority level 3) but behind task B (also priority level 1) because both B and C have the same priority level, but C arrived at a later time.
- The worker is now ready to do a task, so task B gets done and is removed from the to-do list (so is removed from the priority queue).
- A task (D) is added to the to-do list with priority level 2. You add it to the queue ahead of task A (priority level 3) but behind task C (priority level 1).

The given tester (`A7Tester.java`) will test the functionality of your `ToDoList` implementation and mimics a scenario similar to the one above. First, you will need to implement the `compareTo` and equals methods in the `Task.java` file. Next, you will complete the implementation of the `ToDoList` class making use of the `HeapPriorityQueue` you implemented in Part 1. Read the tests provided and documentation carefully to help you understand how the classes you will be writing will be used.

CRITICAL: Any compile or runtime errors will result in a **zero grade** (if the tester crashes, it will not be able to award you any points for any previous tests that may have passed). Make sure to compile and run your program **before** submitting it!