CMPSCI 187 / Fall 2015
Heaps and Priority Queues

Due on November 23, 2015 4PM EST

Mark Corner and Gerome Miklau

CMPSCI 187

Contents

Overview **3**

Learning Goals 3

General Information **3**

Policies 3

Test Files 4

Problem 1 **4**

Import Project into Eclipse 4

Extend and Implement `AbstractArrayHeap` 5

Write `MinQueue` and `MaxQueue` 6

Write Tests 6

Export and Submit 7

Overview

For this assignment you will be implementing a heap using an array; and implementing a min-priority queue and a max-priority queue. You will also be required to write JUnit tests, and these tests will be a major focus of this assignment. **Your tests will be tested and graded**.

Learning Goals

- Develop your understanding of the priority queue abstraction and its implementation as a heap.
- Meet the `Comparator` interface, and demonstrate your ability to use it.
- Demonstrate your ability to read and understand a specification by implementing it.
- Demonstrate your ability to write new JUnit test cases from a specification to test an implementation of that specification.

General Information

Read this entire document. If, after a careful reading, something seems ambiguous or unclear to you, then communicate to the course staff immediately.

Start this assignment as soon as possible. Do not wait until 5pm the night before the assignment is due to tell us you don't understand something, as our ability to help you will be minimal.

Reminder: Copying partial or whole solutions, obtained from other students or elsewhere, is academic dishonesty. Do not share your code with your classmates, and do not use your classmates' code. If you are confused about what constitutes academic dishonesty you should re-read the course syllabus and policies. We assume you have read the course information in detail and by submitting this assignment you have provided your virtual signature in agreement with these policies.

You are responsible for submitting project assignments that compile and are configured correctly. If your project submission does not follow these policies exactly you may receive a grade of zero for this assignment.

Policies

- For some assignments, it will be useful for you to write additional java files. Any java file you write that is used by your solution MUST be in the provided `src` directory you export.
- The course staff are here to help you figure out errors (not solve them for you), but we won't do so for you after you submit your solution. When you submit your solution, **be sure to remove all compilation errors from your project**. Any compilation errors in your project will cause the autograder to fail, and you will receive a zero for your submission. **No Exceptions!**

Test Files

In the `test` directory, we provide several JUnit test cases that will help you keep on track while completing the assignment. We recommend you run the tests often and use them to help create a checklist of things to do next. *But you should be aware that we deliberately do not provide you the full test suite we use when grading*.

We recommend that you think about possible cases and add new `@Test` cases to these files as part of your programming discipline. Simple tests to add will consider questions such as:

- Do your methods handle edge cases such as integer arguments that may be positive, negative, or zero. Many methods only accept arguments that are in a particular range.
- Does your code handle unusual cases, such as empty or maximally-sized data structures?

More complex tests will be assignment-specific. To build good test cases, think about ways to exercise methods. Work out the correct result for a call of a method with a given set of parameters by hand, then add it as a test case. Note that we will not be looking at your test cases (unless otherwise specified by the assignment documentation), they are just for your use.

Before submitting, make sure that your program compiles with and passes all of the original tests. If you have errors in these files, it means the structure of the files found in the `src` directory have been altered in a way that will cause your submission to lose some (or all) points.

Problem 1

Import Project into Eclipse

Begin by downloading the starter project and importing it into your workspace. It is very important that you **do not rename** this project as its name is used during the autograding process. If the project is renamed, your assignment will not be graded, and you will receive a zero.

The imported project may have some errors, but these should not prevent you from getting started. Specifically, we may provide JUnit tests for classes that do not yet exist in your code. You can still run the other JUnit tests.

The project should normally contain the following root items:

`src` This is the source folder where all code you are submitting must go. You can change anything you want in this folder (unless otherwise specified in the problem description and in the code we provide), you can add new files, etc.

`support` This folder contains support code that we encourage you to use (and must be used to pass certain tests). You must not change or add anything in this folder. To help ensure that, we suggest that you set the support folder to be read-only. You can do this by right-clicking on it in the package explorer, choosing Properties from the menu, choosing Resource from the list on the left of the pop-up Properties window, unchecking the Permissions check-box for Owner-Write, and clicking the OK button. A dialog box will show with the title "Confirm recursive changes", and you should click on the "Yes" button.

Problem 1 continued on next page...

`test` The test folder where all of the public unit tests are available.

`JUnit 4` A library that is used to run the test programs.

`JRE System Library` This is what allows Java to run; it is the location of the Java System Libraries.

If you are missing any of the above or if errors are present in the project (other than as specifically described below), seek help immediately so you can get started on the project right away.

Extend and Implement `AbstractArrayHeap`

For this part of the assignment, you will implement a **heap**. A heap is a binary tree of ordered objects that respects the heap property. The heap property states that, for every node in the tree, the children of that node will be "less than (or equal to)" the node itself. In addition to this the tree must be complete. Your implementation will keep the tree in an array (actually an `ArrayList`) and will use implicit pointers, as discussed in the book and in class.

Start this part of the assignment by opening up the `AbstractArrayHeap` class (in the `support` directory). Review what is already provided for you and what you have to implement. Your implementation must be a class named `StudentArrayHeap` that extends `AbstractArrayHeap`. There is a file with that name already in your `src` directory. It will not compile as distributed; you will need to complete it.

Your heap is going to store pairs consisting of a priority and a value. For comparison purposes all that will be compared is the priority. The definition of the class is generic and uses two type variables `P` and `V`. `P` is the type of the priority values and `V` the type of the paired value. Note that the declaration for `P` doesn't state that it must extend `Comparable`. So how do we compare them? The constructor for the `AbstractArrayHeap` class takes an argument, a comparator of type `Comparator<P>`. This allows us to provide arbitrary ordering for `Priorities`. If we want a max-heap, we simply pass in a `Comparator` that says larger values come first. For a min-heap we use a `Comparator` that says smaller values come first.

Since we are storing priority-value pairs you will notice that our `add` method is a little bit different in that it takes in two arguments: a **priority** and a **value**. This allows us to insert two elements with the same value but different priorities. After receiving these inputs, we transform them into an `Entry<P,V>` and use this to store our element. This is commonly referred to as a **priority-value pair**.

How does `bubbleUp` work? The `add` method simply creates an entry and adds it to the end of the array. Then it calls to `bubbleUp` on the index of the newly added element. The `bubbleUp` method will cause the new entry to float up the heap until it is in a location such that the properties of a heap are maintained.

How does `bubbleDown` work? The `remove` method has the opposite problem; it removes the first element in the array, and replaces it with the very last. It then calls `bubbleDown` to bubble the element down until the heap property is restored.

Write `MinQueue` and `MaxQueue`

Using the `StudentArrayHeap` class implemented in the previous part, finish two classes: **`MinQueue<V>`** and **`MaxQueue<V>`**. Each of these should be a `PriorityQueue` that use `Integer` priorities. The `MinQueue` is implemented such that lower integer values have the highest priority. The `MaxQueue` is implemented such that higher integer values have the highest priority. (Stubs for these classes are present in your `src/structures` directory.

To do this part, you will also need to finish two classes, one called `IntegerComparator` and the other called `ReverseIntegerComparator`, both of which implement the **`Comparator<Integer>`** interface. There are stubs for these classes already in your `src/comparators` folder. An example `Comparator` has been provided in the support package: `comparators.StringLengthComparator`. This `Comparator` orders `Strings` such that shorter `Strings` have lower priority than longer `Strings`.

Write Tests

An important difference between this assignment and previous assignments is that we will be grading the tests that you write. We will do this by creating a number of flawed implementations of the five classes you are writing, and check that your tests discover the flaws. We will also run your tests on a flawless (well, as close as we can make it) version to ensure that your tests are not rejecting valid code.

There are five existing test classes: one that ought to be called `StudentArrayHeapTest` but is actually called `AbstractArrayHeapTest`; two for your priority queues: `MaxQueueTest` and `MinQueueTest`; and two for your comparators: `IntegerComparatorTest` and `ReverseIntegerComparatorTest`. These are all in your `test` directory. **Add your tests to these files, and leave these files in the `test` directory.**

The existing tests will give you some idea how to write your new tests, and there are a couple of pointers we would like to give you:

1. Keep each test small; it should test only one thing. Sometimes you will have to do several other things first in order to do that one thing, but only test one thing. For instance, if you wanted to check that adding five elements and then removing four left a heap with a size of one, then you would have to add five elements, remove four elements, and finally test that the size is one.
2. If you look at the tests we use for grading you will see that they often test several things in a single test. That's because our tests have a different purpose. Your tests should each test only one thing.
3. Write a test for each thing that is given in the specification; if the specification says "if this happens, then the method returns that," then write a test that makes "this" happen, and ensures that the method returns "that".
4. Also write tests to verify the basic operation. For instance, if you're testing a stack you should write a test to ensure that if you do a `push` and then a `pop`, the popped element should be the same element that was pushed on.
5. Each test must have an `@Test` annotation, which may have arguments:
   - To test whether a method throws the exception that it is supposed to under certain circumstances (e.g., a `NullPointerException`), annotate your test as follows:

```
1 @Test (timeout = 100, expected = NullPointerException.class)
```

Note that you need to put the `.class` at the end, and that you should replace `NullPointerException` with the name of whatever exception you're actually checking for.

- The `timeout = 100` annotation is a way of ensuring that the test can recover from infinite loops; the test will be aborted after (in this case) 100 milli-seconds.

Export and Submit

When you have completed the changes to your code, you should export an archive file containing the entire Java project. To do this, click on the `heap-queues-student` project in the package explorer. Then choose "File → Export" from the menu. In the window that appears, under "General" choose "Archive File". Then choose "Next" and enter a destination for the output file. Be sure that the project is named **`heap-queues-student`**. Save the exported file with the `zip` extension (any name is fine).

Once you have the zip file ready, you can submit it to the online autograder. Go to autograder.markdcorner.com create an account with the correct email address and student id. Select the right project from the drop down and submit. It shouldn't take more than a minute to grade your project, but if there is a backlog of projects then it may take longer. If you encounter any errors that look like the autograder failed and not your project, please let the instructors know.