Writeup for first project of CMSC 420: "Data Structures" Section 0301, Spring 2018

Theme: Heaps and Priority Queues

Handout date (**nominal**): Thursday, 01-25
Handout date (**actual**): Monday, 01-15
On-time deadline: Tuesday, 02-06, 12pm (noon)
Late deadline (30% penalty): 02-08, 12pm (noon)

1 Overview

In this "warmup" programming project, you will remind yourselves of two basic, yet very powerful, data structures that you have talked about in courses such as CMSC132 and CMSC351: Binary **(Min-)Heaps** and **Priority Queues**. You will have to implement four classes:

- `LinkedMinHeap.java`
- `ArrayMinHeap.java`
- `LinearPriorityQueue.java`
- `MinHeapPriorityQueue.java`

You will first need to pull our starter code from GitHub. Then, you will find the first two classes under the (sub-)package `heaps` and the latter two under the (sub-)package `priorityqueues`. You should **not** move those classes from those packages! If you do, the submit server unit tests will **not** run against your implementation!

All the code that you write *can* be contained in those 4 source files. However, **this does not necessarily need to be the case**! You can add your own `jUnit` tests **anywhere in the code base you want**. For example, the provided data structure packages (see below) include several `jUnit` test classes under sub-packages called `test`. You yourselves do **not** have to replicate this code structure under the packages that you implement; if you prefer, you can have a separate package under the default package called, e.g, `myunittests`. It's your code, your choice. Similarly, any inner classes that you might create can be in the same file as the `public` classes that you must implement, or in different files. It does **not** matter;

the only thing that **does** matter is that the four `public` classes that you implement reside **exactly** where they are given to you in the code bases's package structure.

To complement your code base, we provide you with four additional packages:

- `exceptions`: A package which contains some global `Exceptions` used by the provided skeleton code. You should **not** rename, move or delete this package! You should also **not** change the names of the `Exception`s declared in it.
- `fifoqueues`: A package that contains interface **and implementation** of three different kids of FIFO queues. We additionally provide a unit-testing framework under a sub-package called `test`. You are **very encouraged** to read the source code of both the implementations **and** the unit test suites to get ideas for your project!
- `lists`: Similar to `fifoqueues`, but containing code on various linked lists.
- `trees`: Similar to `lists` and `fifoqueues`.

2 Prerequisites

All prerequisites for handling this project are reasonable for an advanced CS student. We expect that you are well-familiar with **Binary Search Trees, Stacks, Lists, FIFO queues** and programming in Java. Skills harnessed by a typical UMD freshman course such as CMSC 131 / 132 are more than sufficient. You will need to remind yourselves of what a **binary heap** is and how insertions and min-deletions work, as well as the ways in which they can be represented in computer memory. The structure and operation of heaps are briefly touched upon in section 3 and will also be tested on your **first homework**. They are also **fair game for your first midterm and final**.

In this class, we do not aim to test your programming, OOP or Java expertise. We want to teach you Data Structures. However, **a minimal amount of familiarity with certain OOP / Java constructs will be required.** In particular:

- You will need to do some reading about what a **fail-fast iterator** is, and how this is implemented by the built-in Java class `java.util.Iterator`. It is quite possible that you will have to implement **your own fail-fast iterators**, and to do this you will need to know when to throw a `java.util.ConcurrentModificationException`.
- You need to know what a `java.lang.RuntimeException` is, how to catch it, derive information from it and maybe even re-throw it. You also need to know what it means for a method to declare a “checked” `Exception`.
- It is almost **certain** that you will need to implement your own **inner classes** for certain `Node` types. This is a programming technique touched upon and extensively practiced in CMSC132.

3 Short reminder on Heaps and Priority Queues

Heaps and Priority Queues are examples of data structures that are considered **below** the level of CMSC420. The first **novel** data structures that we will talk about are AVL, Splay and Red-Black Trees (wait till the second week of classes for this). In this section, we will very briefly remind you of how heaps work and (re-)introduce Priority Queues.

3.1 Heaps

A **heap** is a complete binary tree (not a complete binary **search** tree!) where insertions always occur at the "rightmost" available position at the leaf level (the last position in a level-order traversal of the heap). Furthermore, when considering deletions of elements, it is only possible to delete the element at the **root** of the tree. The key property of the heap is that every node's subtree contains elements whose values are **smaller than or equal** to the node's element itself. Formally, this is known as a **minheap**; a **maxheap** is a heap where every node's subtree contains elements whose values are **greater than or equal** to the node's element itself. It should then be clear that heaps are binary trees, but not binary search trees! :) In this project, we only care about **min**heaps.

After inserting a new element in the minheap, we have to make sure the heap property is maintained. To do this, we *percolate* the new element **upwards** as far as we can: for every parent, we need to make sure that the parent's value is smaller than or equal to our own. If not, we swap ("percolate upwards") the parent with the currently examined node. This process of "upwards percolating" *globally* guarantees the heap property, a fact that we can formally prove via induction.

For example, figure 1 shows the process of "upwards percolation" for a minheap after a new insertion. Figure 2 shows another example, where the process goes all the way up to the root!

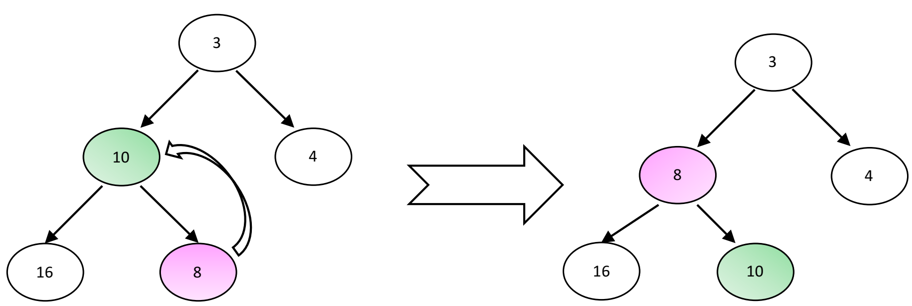

Figure 1: Inserting 8 in a minheap. Since $10 > 8$, we need to swap 10 with 8. No further percolation is necessary, since $3 < 8$.

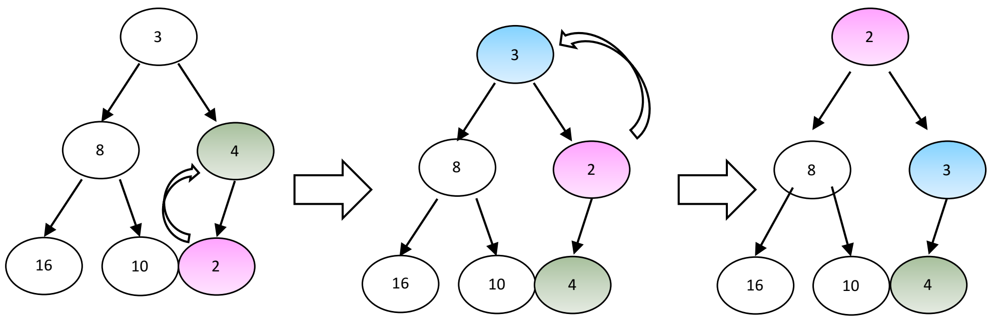

Figure 2: Inserting 2 in the heap of Figure 1 leads to consecutive swappings with 4 and 3, for a two-level upwards percolation.

As mentioned above, heaps only allow for the deletion of the root element. For min-heaps, since the root element is also the **minimum element** (might not be unique, but that's fine), the operation is often referred to as "deleting the minimum" (`deleteMin`). In the case of `deleteMin`, we retrieve the minimum element from the root node and replace it with the element in the **rightmost leaf** (the "last" element in a level-order traversal of the heap), which we subsequently throw away from the heap. Since `deleteMin` needs access to this "last element" of the heap, it is important, for efficiency reasons, to **always maintain a reference to this element**. After we copy over this element, we percolate it **down**, swapping the parent with the **minimum** of our two children at every level (figure 3). We can once again prove inductively that this "downwards percolation" process *globally* guarantees the heap property.

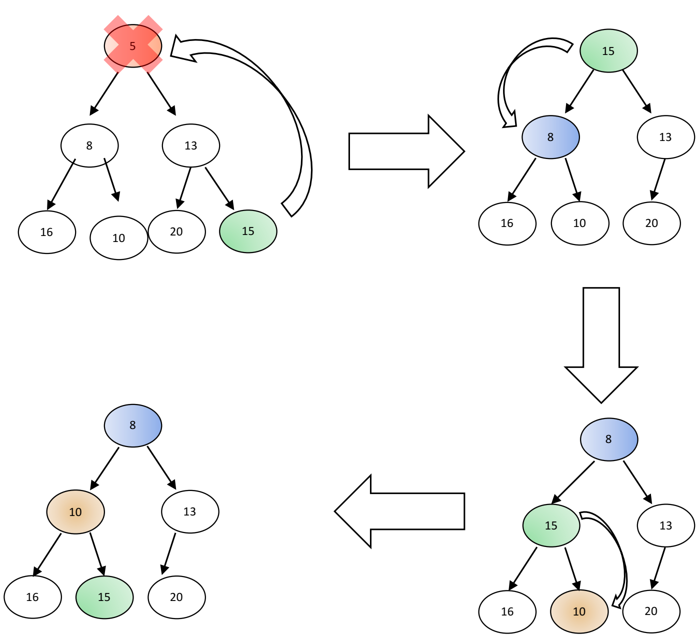

Figure 3: Deleting the minimum, replacing with "last" leaf element and percolating down.

Since heaps are complete binary trees, they can be implemented **very efficiently and compactly** using an **array** (figure 4. This is based on a breadth-first (level-order) enumeration of the nodes in the heap. This enumeration is exemplified in figure 4. Note that the node at index i has children at indices $2i + 1$ and $2i + 2$ (one or two children **might not even exist**, of course), whereas the parent of index i (**if existing**) is at index $\lfloor \frac{i-1}{2} \rfloor$.

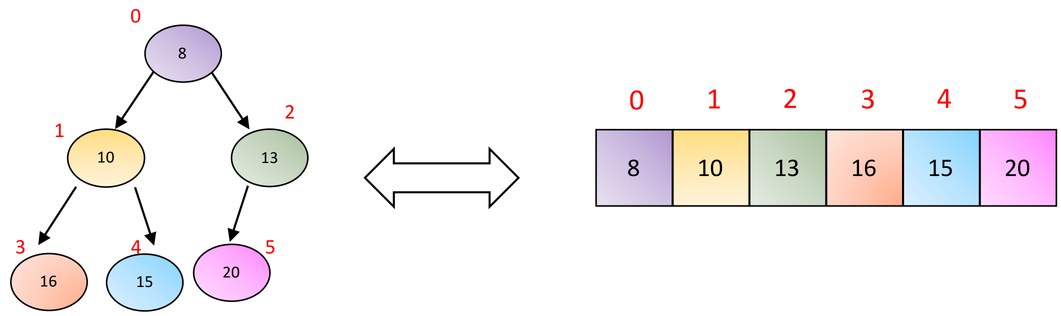

Figure 4: Implementation of the resulting minheap of Figure 3 as an array.

Implementing your minheap in this way will be your task in `ArrayMinHeap.java`. You will have to think about how your implementation will have to change from the linked implementation of `LinkedMinHeap.java` and how your percolation routines will need to be adjusted. It might help you to realize that **the exact same unit tests that you write for `LinkedMinHeap` should also be passable by `ArrayMinHeap`, and vice versa**! That is, if you were to just **copy and paste** all of the unit tests for one of the two classes in a separate file and test them against the other class, the only thing that you **should** have to change is the type of the reference in your `jUnit` tests from, say, a `LinkedMinHeap` to an `ArrayMinHeap` reference type. You could also probably think of a polymorphic approach with a `jUnit` test class that takes a `MinHeap` reference in its constructor, the reference itself instantiated by client code into either a `new LinkedMinHeap` or `new ArrayMinHeap` (both of these classes implement `MinHeap`, so through standard upcasting this would work just fine).

3.2 Priority Queues

The Priority Queue is an Abstract Data Type (ADT) with a very simple property: Every element to be enqueued is attached a certain positive integer priority, which predetermines its **order** in the queue. By convention, **smaller integers are considered “higher” in terms of priority,** such that, for example, priority 1 is considered a higher priority than 3. Dequeueings only happen from the top of the queue, after which the element “before” the first one will be available for immediate processing. We see these kinds of queues all the time in real life. For example, people with disabilities and veterans are given higher priority when boarding a plane!

As an example, let’s consider two separate cases of enqueueings:

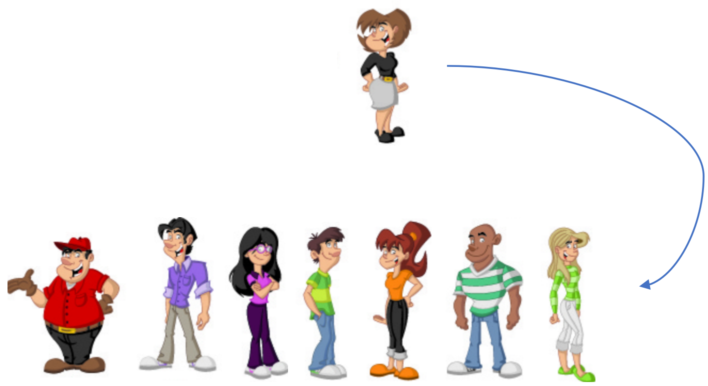

Figure 5: A simple FIFO queue enqueuing scenario. The person in the top can only be enqueued in the "back" of the queue.

In figure 5, we see an example of a traditional **FIFO** (**F**irst **I**n - **F**irst **O**ut) queue. In such a queue, we can only insert at the end, so **longer queues have longer average wait times, irrespective of the properties of the elements that are processed**. This rigidity is simply unacceptable in practice, and Priority Queues are used almost universally. A perfect example is **Operating Systems**, which maintain a Priority Queue of processes that are to be allocated CPU time. Some processes are so much more important than others that they **must** be given higher priority!

Despite their apparent simplicity, FIFO queues can be implemented in a number of different ways, each with its own different properties. For example, some implementations allow for faster enqueueings than dequeuings (or vice versa), while others use contiguous or dis-contiguous memory! Please feel free to examine the package `fifoqueue` for three different implementations of FIFO queues, **all of which adhere to the common interface**, but use **different implementations** with **wildly different properties, efficiency characteristics and memory footprints**!

In contrast with a traditional FIFO Queue, when considering a Priority Queue, every element has a certain **positive** priority attached to it:

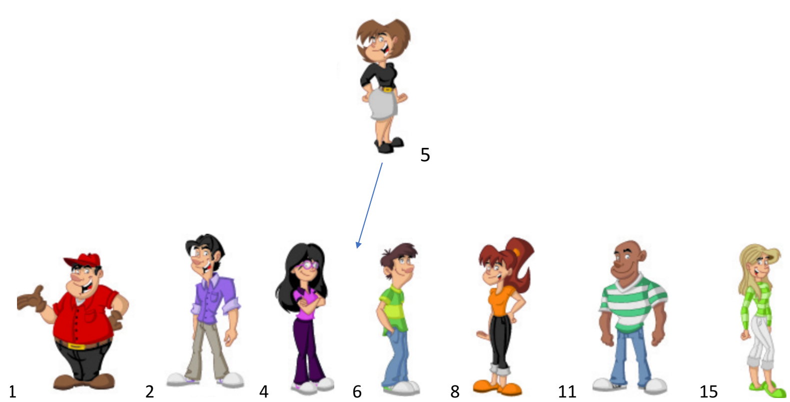

Figure 6: A simple Priority Queue enqueuing scenario.

As you are even reading these lines and looking at Figure 6 you are probably envisioning a **linear** implementation of a Priority Queue. **This is good**; keep these visualizations in mind when implementing `LinearPriorityQueue`. Naturally, such an implementation of a Priority Queue would be **very inefficient**, since it leads to $\mathcal{O}(n)$ enqueueings and, depending on the underlying implementation, maybe even $\mathcal{O}(n)$ dequeueings! A much better way to implement Priority Queues is a **binary minheap**, and this will be your task when implementing the class `MinHeapPriorityQueue`. Since minheaps are **complete** binary trees, and complete binary trees are also **balanced** (except for maybe one level of imbalance), we have enqueuings and dequeueings in $\mathcal{O}(\log_2 n)$. These efficiency **guarantees** of the minheap make it a **very desirable choice for implementing Priority Queues**.

4 Tips / Hints / Guidelines

The following are just some guidelines that **may** be helpful for you to implement your project.

- Familiarize yourselves with the most elementary `git` commands (`clone`, `pull`, `add`, `commit`, `push`, `alias`). Our ELMS page has startup resources for you under a page called "More Resources". Your elementary `git` knowledge will also be tested on your **first homework**. We **highly** suggest that you **pull** the code (this is literally a single command) **instead of downloading the zip** from GitHub. This is because, if we ever post a correction on the docs or the source code, you will only need to type **one single command** to pull **only the changes themselves**, without affecting your existing solutions! If you always download the ZIP whenever a correction is made, you will have to nitpick the files that you copy-and-paste and **possibly overwrite your work by mistake**. In fact, this is **guaranteed to happen** if you download the ZIP the first time, work on the project for a while, and when a correction is posted you decide to try out `git` and do a `git clone`. This will **immediately overwrite everything in your current directory and replace it with the file state of the latest upstream commit. IN THE NOT SO DISTANT PAST, THE INSTRUCTOR LOST TWO WEEKS' WORTH OF CODE BECAUSE OF THIS MISTAKE.**

- **Read the JavaDocs** of the interfaces that you will have to extend (`MinHeap.java, PriorityQueue.java`). Opening `index.html` from your favorite web browser and then navigating around is a fine way to start.

- **TEST FIRST, IMPLEMENT SECOND. WE CANNOT STRESS HOW EASIER THIS WILL MAKE YOUR DEVELOPMENT**. Further suggestion: find somebody from the class, **write their unit tests for them** and have **them** write **your** unit tests. Both of you should view the other one as your **adversary** in the class! Think in the following manner: *"Based on the provided interface as it is described in the JavaDocs, how can I write tests that will only be passed by a tip-top implementation? How can I try my best so the other person's code DOES NOT GET 100% credit?"*

- Read the provided data structures' source code **and** unit testing frameworks for ideas on implementation and testing.

- Think about how you would write a **stress** test for a method. You want to make sure that the method passes **any conceivable** input (of the appropriate type, of course), where **"not passing"** means **"throwing any conceivable error"**.

- Think about how you could write a unit test that **expects** a certain error to be thrown!

- When writing your `jUnit` tests, you might notice, *perhaps with a degree of disdain*, that you will have to frequently encapsulate certain calls in `try - catch` blocks, even when you **feel** you are certain that certain exceptions **should never** be thrown. For example, you will note that the method `void enqueue(T element, int priority)` in `PriorityQueue.java` throws an `InvalidPriorityException` if the argument `priority` is not a **strictly positive integer**. Even in cases where you yourself are **certain** that you are supplying a valid priority, the Java compiler will require of you to encapsulate the call within a `try - catch` block and deal with the `InvalidPriorityException` appropriately.

  This possible disdain of yours is **entirely natural**, and **you are not doing anything wrong**. In Java, when a method "checks" an `Exception`, the caller **must** encapsulate the call in a `try - catch` block that handles a `java.lang.Exception` (or even a general-case `Throwable`) of **at least that type**. Many resources, including the official Google C++ Style Guide, advise **against** handling errors with exceptions **whenever possible.**[1] In this project, however, and **in several code bases that you will have to work on in your careers,** you **will** have to deal with exceptions appropriately.

- Consider switching from Eclipse to an IDE such as NetBeans or IntelliJ.

- For this course, you will **not** need to play around with `Java 8` features. However, `IntStreams`, `Ranges`, lambdas and `Collectors` can make your unit tests **very easier** to write. For example, here are two lines of code from one of our unit tests:

  ```
  List<Integer> priorities = IntStream.range(1, MAX_PRIORITY + 1).boxed().
     collect(Collectors.toList());
  Collections.shuffle(priorities, r);
  ```

  While one could quite easily write the two lines above using for-loops, the entire point behind most functional `Java 8` features is to make developers avoid tedious, non-parallelizable and error-prone loops.

5 Submission / Grading

Credit in this project is defined by the number of submit server unit tests that you pass. However, **we will also be inspecting your source code to make sure that your work adheres to certain standards.** For example, the `ArrayMinHeap` class needs to be implemented using contiguous storage (e.g raw arrays), while your implementation of `MinHeapPriorityQueue` **must** use either one of your `MinHeap` implementations! **In order**

[1] In constructors, it is **never** possible.

to get any credit for the unit tests that correspond to a specific class, your implementation should adhere to the documentation's guidelines. In particular, using built-in Java primitives or third-party libraries that implement MinHeaps and PriorityQueues are a **very easy way to ensure zero credit!** Those requirements are also repeated in the JavaDocs.

Projects in this class are different from your typical 131/2 projects in that **we do not maintain an Eclipse - accessed CVS repository** for you or us. This means that you can no longer use the Eclipse Course Management Plugin to submit your project on the submit server. This turns out to be a good thing, since it frees you up from the need to use Eclipse if you don't want to. To submit your project, make a `.zip` or `.tar.gz` archive of your **ENTIRE PROJECT DIRECTORY** (not `src`, not a new folder containing `priorityqueue` and `heaps`, but your **ENTIRE TOP-LEVEL PROJECT DIRECTORY**). Then, upload the archive on the submit server as seen on figure 7.

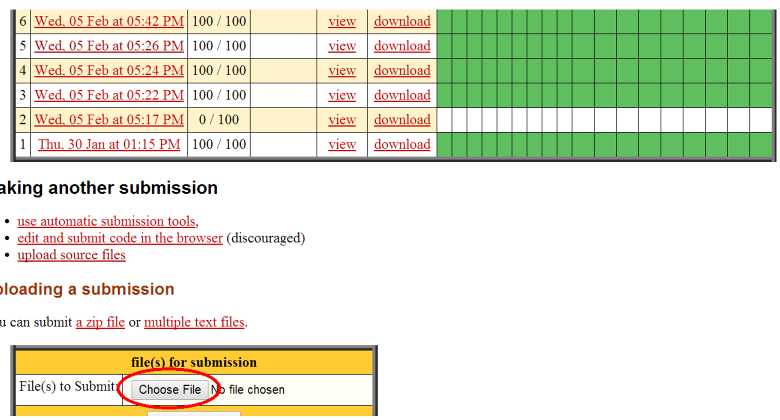

Figure 7: Uploading your project on the submit server.

All tests are release tests, and you can submit **up to 5 times** every 24 hours. **We urge you to unit-test your code thoroughly before submitting**: treat every token like a bar of gold that is not to be wasted! We will **not** share the source code of the unit tests with you, not even after the deadline for the project!

We maintain your **highest-scoring submission** for grading purposes. Finally, for the late deadline, we take 30% off your maximum possible score. This means that, if you submit late, passing all the unit tests and implementing inorder traversal **as discussed** will give you 70% of the total grade.

Finally, we should remind you that for the past few years the Software Similarity Detection System MoSS has been incorporated into the CS department's submit server. For n student submissions, it is **ridiculously easy** (literally a single click) for us to run MoSS against all $\binom{n}{2}$ pairs of submissions. MoSS is tuned towards higher than 50% Recall, which means that plagiarized submissions **will** be caught. We would much rather be spending time teaching you data structures and assisting you with your queries than going back and forth with the Honor Council; **help us help you**!