COSC-211: DATA STRUCTURES
HW4: PRIORITY QUEUES

Due Friday, March 2, 9:00am

Reminder regarding intellectual responsibility: This is an individual assignment, and the work you submit should be your own. Do not look at anyone else's code, and do not show anyone your code (except for me and the course TAs).

1 Introduction

We have talked a lot in class about how to analyze the asymptotic performance of our data structures. Typically we've been interested in a big-O analysis of the worst case performance: we want an upper bound (that's the big-O part) on how long it can take our operations to run on *any* input (that's the worst case part). Your goal in this assignment is to explore the relationship between our asymptotic analysis and the actual, empirical runtimes of our data structure operations.

This assignment comes in a very long document with lots of information about the Java constructs we are using and the code that I've provided for the experiments. Here is a summary of what your actual tasks are in this assignment:

1. Implement a heap-based priority queue (Section 2).
2. Run some timing experiments to empirically measure how long different operations take (Section 3.2).
3. Write up a short report on your findings (Section 3.3).

Your experiments will take a long time to run, so I strongly suggest you **start this assignment early** so that you get your heap implementation working with sufficient time remaining to run the experiments.

1.1 Setup

Begin by downloading some files:

```
$ wget -nv -i https://goo.gl/6ywtir
```

You should now have five Java files:

- `PriorityQueue.java` is an interface that specifies the methods that must be contained in any class that wants to call itself a priority queue (see Section 1.2).
- `PQSorted.java`, `PQUnsorted.java`, and `PQHeap.java` give three different implementations of a priority queue. `PQSorted.java` and `PQUnsorted.java` are complete (you should not change any of the code in these classes), and you will fill in `PQHeap.java` (see Section 2).
- `Tester.java` provides some code that you will use to run your timing experiments (see Section 3).

1.2 Tutorial on Java Interfaces

Suppose we are writing a program to display web pages. Each web page consists of a large number of different components, including images, text, links to other pages, ads, etc., each of which might be specified in its own class. These different components all need to be formatted to show up properly on the page, and the way in which we do the formatting depends on what the component is (e.g., displaying text might involve looking up a font, whereas displaying an image might involve scaling the image to match the size screen on which it's being displayed). One way of handling this would be for each class to provide its own set of methods that manage display functions. Maybe the `Text` class has a `printToScreen()` method that simply prints out whatever text it is storing. The `Image` class, on the other hand, might require different methods `renderForChrome()`, `renderForMobile()`, etc., because what needs to be done to display an image might be platform-dependent. In order to write our program that displays all of the components of a web page, we need to keep track of what particular methods exist to do the displaying for each type of component. Life would be much easier if we could just call something like `theComponent.display()` and have the component's class figure out what needs to be done to handle its particular version of displaying!

The Java language provides a construct that captures this idea: the *interface*. The purpose of an interface is to specify a desired set of functions that a class might provide. For example, we might have an interface `WebDisplayable` that tells us that any object that claims it can be displayed on a web site must provide methods to display the object in Chrome, in Safari, and on a mobile device. The interface itself does not provide any details about how to do the displaying; instead it tells us what methods the object must implement.

This is very similar to what we've been doing when we have defined a new abstract data type (ADT). The definition of the ADT has involved a list of operations we want our ADT to support. For example, we said that a Stack is a list that must allow us to push a new item onto the top of the list, pop an item off the top of the list, check how many elements are stored in the list, and check if the list is empty. The ADT is defined by its desired behavior; as we've seen, there are generally many ways to implement the data structure to produce the desired behavior. Similarly, an interface specifies a set of desired behaviors for a class; there may be many classes that implement the same interface, each of which provides its own own way of achieving the desired behaviors.

Creating interfaces

An interface is contained in a `.java` file, just like a class. It looks something like this:

```
public interface WebDisplayable {

   public void displayChrome();

   public void displaySafari();

   public void displayMobile(String s, int t);

}
```

The first line in the file looks like a class declaration, except that it has the word `interface` instead of the word `class`. The rest of the file consists of a list of method headers. Each line ends with a semicolon, and none of the methods are actually implemented here. At this point, all we are doing is listing the desired functions of any class that implements `WebDisplayable`. In this case, our `WebDisplayable` interface says that any class that is `WebDisplayable` must include three specific methods. These methods can have input parameters (as `displayMobile()` does in our example) and they can have return types.

In our interface, we do not specify how to make these functions happen: there are no method bodies. This is left to the classes that *implement* our interface.

Implementing interfaces

When we say that a class *implements* an interface, we are making a promise that the class will provide an implementation of all of the methods specified in the interface. For example, we might have a class that begins with the following line:

```
public class Jpeg implements WebDisplayable {
```

As usual, we have the class declaration `public class Jpeg`. This is followed by the words `implements WebDisplayable`. By including these words, we are asserting that `Jpeg` will contain methods with headers that *exactly* match the headers listed in the `WebDisplayable` interface.

`Jpeg` can also contain methods that are *not* listed in `WebDisplayable`. We might, for example, want `Jpeg` to include a method `convertToGif()` that converts an image file with the extension `.jpeg` to one with the extension `.gif`. This method exists independently from the `WebDisplayable` interface.

A class can implement multiple interfaces at the same time. For example, we might want our `Jpeg` class to also implement an interface `Compressable` that includes a `compress()` method. We would then declare our `Jpeg` class as follows:

```
public class Jpeg implements WebDisplayable, Compressable {
```

This class declaration is now a promise that `Jpeg` will include methods with headers that match all the methods listed in the `WebDisplayable` interface, *and* all the methods listed in the `Compressable` interface.

Using interfaces

The advantage of using interfaces is that they allow us to work more easily with multiple different classes that are capable of the same behaviors. In our web site display example, we might imagine that we start off with a set of different components that we want to display. The class corresponding to each of these components (`Image`, `Text`, `Ad`, etc.) should implement the `WebDisplayable` interface, and it would be nice if we could somehow deal with a list of object that are `WebDisplayable`, without having to worry about what specific type each object is. For example, we might like to write the following method:

```
public void showWebsiteChrome(WebDisplayable[] components) {
   for(int i = 0; i < components.length; i++) {
      components[i].displayChrome();
   }
}
```

The interesting thing about this method is that the input parameter's type is an *interface*, not a class. What this says is that all of the objects in the input array will be instances of some class that implements `WebDisplayable`. This is nice because it allows us a little flexibility (we can store multiple different types of objects in the array) while also providing some restrictions (the only types of objects we can store in the array are those that are `WebDisplayable`).

We can do this. A variable can be *declared* as an instance of an interface, rather than an instance of a specific class. For example, we can say:

```
WebDisplayable myComponent;
```

This says that `myComponent` is going to be some sort of object that is `WebDisplayable`. However, because the `WebDisplayable` interface doesn't contain any method bodies, when we *initialize* the variable we must use a specific class that implements `WebDisplayable`. For example, we could say:

```
myComponent = new Image();
```

since the `Image` class implements `WebDisplayable`. If we then issue the method call `myComponent.displayChrome()`, we will call the particular version of the `displayChrome()` method that lives in the `Image` class, because `Image` is the instantiated type of `myComponent`.

Interfaces in this assignment

The `PriorityQueue.java` file that you downloaded is an interface. It lists four methods that any class claiming to be a priority queue must provide: `add(Integer toAdd)`, `remove()`, `size()`, and `isEmpty()`. You will see that the `PQSorted` and `PQUnsorted` classes that I have provided both implement the `PriorityQueue` interface, and both do indeed provide implementations of the methods specified in the interface. The `PQHeap` class, which you will write, should also implement `PriorityQueue`.

In the `measureTimes()` method in `Tester.java`, you will see the line:

```
PriorityQueue queue = new PQSorted();
```

The variable `queue` is declared as a `PriorityQueue`, and it is initialized as a `PQSorted`. The declaration as type `PriorityQueue` means that when `queue` is initialized, it must be initialized to an instance of `PQSorted` or `PQUnsorted` (or, once you write the class, `PQHeap`). When you want to test each different type of priority queue, you can simply change the initialization on the right-hand side.

2 Task 1: Heap Implementation

The PQHeap.java file that you downloaded is currently empty. Your job is to fill it in, using the array-based Heap implementation that we discussed in class. Specifically, your PQHeap should implement the PriorityQueue interface, and you will need to fill in the contents of each method specified in the PriorityQueue interface. You also likely will want to write extra supporting methods, like siftUp and siftDown.

Make sure you *test your code* thoroughly before you move on to the next part of the assignment. You might want to add some code to Tester.java to convince yourself that your add and remove methods are working properly. If you have errors in your heap code that may affect the timing results that you obtain in the second part of this assignment. Furthermore, any submitted code that does not compile will not receive credit.

3 Experiments

Your goal in this part of the assignment is to empirically measure the performance of each of the three priority queue implementations. I have provided code in the measureTimes() method of the Tester class that runs a particular timing experiment. The goal of this experiment is to time how long the add and remove methods take to run on particular values of n (as usual, n is the number of items in our priority queue). Section 3.1 describes the code that I have provided, Section 3.2 describes the experiments you will run, and Section 3.3 describes what you should include in your written report of your experiments.

3.1 Understanding the provided code

Read through the Tester code, which does the following:

- Takes as input an int reps and an int[] record. The array record is a list of all values of n you're interested in (sorted in increasing order), and reps indicates how many times you want to repeat the experiment.
- For each replication of the experiment, we do the following:
  - Create a new (initially empty) priority queue
  - Add randomly generated elements to the priority queue one at a time. If we're currently adding the nth element for one of the values of n stored in our record array, record the time (using the command System.nanoTime()) before and after adding the element; the difference between these two times gives us the duration of the add operation.
  - Remove all of the elements from the priority queue one at a time. If we're removing from a priority queue that currently has n elements in it for one of the values of n stored in our record array, measure the duration of the remove operation.

- After completing all replications of the experiment, compute the average duration (in milliseconds) of `add` and `remove` for each `n` of interest, and return an array of these averages (`times[i][0]` contains the average `add` time for the `i`th element of interest, and `times[i][1]` contains the average `remove` time for the `i`th element of interest).

Do not change any of the code in the `measureTimes()` method, except for the line `PriorityQueue queue = new PQSorted();`, which you can change to test each of the three implementations. I have set up this method to run a very particular type of timing experiment, and if you change what it does, you will not get the same results. Again: **DO NOT** change this method!

Q: But I have a faster/better/cooler way of timing the methods. Can I use that way instead?
A: Nope! Again, there are many ways you *could* measure the performance of your data structures, but I want you to use this method because it will reveal some interesting behaviors that aren't visible using other timing strategies.

Q: OK, so what's so special about the way your experiments are set up?
A: The short answer is that my setup ensures that the methods do more or less the same thing every time they are called on the same value of `n`. Consider what would happen if instead we interleaved the calls to `add` and `remove`, like so:

```
for(int i = 1; i <= record[record.length] + 1; i++) {
   for(int j = 0; j < reps; j++) {
      int toAdd = (int)(Math.random() * 10000);
      // include timing code
      add(toAdd);

      // include timing code
      remove();

      int toAdd = (int)(Math.random() * 10000);
      add(toAdd);
   }
}
```

That is, all I've done is change the order of the `n` and `reps` loops. Now suppose we are on a value of `n` on which we have to resize our array. Using this new version of the timing experiment, we resize during the first call to `add` that happens on the first iteration of the `reps` loop, but not on any subsequent iteration. So now our timing data is not capturing the effect of having to resize on certain values of `n`. The code that I've provided in `Tester` does not have this problem: we create and use an entirely new priority queue on every replication of the experiment.

Q: What's the point of the replications?
A: The timing data is very noisy: the same operation can take a different amount of time depending on lots of factors that have little or nothing to do with what we're trying to measure. For example, in the sorted array implementation, our `add` method will have a different runtime depending on if the element we're adding happens to be larger than everything that's already in our priority queue,

smaller than everything that's already in our priority queue, or somewhere in between. If Java's garbage collector happens to run while we're in the middle of timing a method, the time for that method will be artificially inflated. If lots of users are running code on `remus/romulus` at the same time, that can slow things down because you're all competing for resources. And so on. Running many replications of the same experiment reduces this noise because the chances that an unusual bad event happens every time are reasonably low. However, *it is nearly impossible to completely eliminate noise*. Don't worry if you see outliers in your results: this is to be expected!

3.2 Task 2: Run your experiments

Fill in the `run()` method of `Tester` to run some experiments. Your `run()` method should call `measureTimes()` to actually do the measurements. Your job in `run()` is to decide what input parameters to use—that is, for what values of `n` you are interested in recording the timing results, and how many replications you want to run. Some things you might want to consider include:

- Your goal is to get a sense of how well the asymptotic analysis matches up to the empirical results. Asymptotic analysis deals with very large values of `n`, so you will need your `n`s to get high enough to identify a clear trend.
- On the other hand, as `n` gets very large, it will take a long time to run your experiments. You should be patient (it's reasonable to let your experiment run for, say, an hour per implementation) but not too patient (your experiments shouldn't be running for days).
- Do you want to use a different maximum value of `n` for the three different priority queue implementations? How does this question relate to the first two points above?
- Are there *particular* values of `n` that you want to test because they might have different behavior than other choices of `n`? (Hint: when do you call `resize()`? Does this happen for all `n`?)
- How many times do you want to replicate your experiments? You'll want to run enough replications of each experiment to try to reduce the noise. However, as with large values of `n`, running many replications will make your experiments take a long time.

To give you a rough sense of what kinds of `n`s are reasonable, I ran my `PQSorted` experiments with values of `n` up to around 500000.

You are welcome to write your code on your personal computer if you want, but **please run your experiments on `remus/romulus`**.

3.3 Task 3: Write up your results

There is a short written component to this assignment. The purpose of this component is to get you to reflect on the results of your experiments and think about how your empirical results relate to what we learn from the asymptotic analysis of our data structures. Specifically, your tasks are as follows:

1. Explain what choices you made when running your experiments in Section 3.2. What values of `n` did you record? Why did you make these choices? How many replications did you run for each experiment?

2. Graph the results you obtained in your experiments. You should have a separate graph for each implementation (sorted array, unsorted array, and heap). The x-axis should be n (the number of elements in the priority queue) and the y-axis should be the average time per operation in milliseconds. Each graph should include two data sets, one for the `add` operations and one for the `remove` operations.

3. Write a short (∼1-2 paragraph) discussion of your observations. Your response should address at least the following questions:

   - In general, is the asymptotic analysis a good predictor of the shape of the empirical runtime results?
   - Does the asymptotic analysis give you the full picture of the empirical runtimes? What do you learn from the graphs that you don't learn from the asymptotic analysis? What do you learn from the asymptotic analysis without having to run any experiments?
   - In what cases (which implementations and operations) does the worst-case asymptotic analysis give a tight upper bound for all n? When does it not? Why not?

   Feel free to note any other interesting observations that you made about the results!

4 Submit your work

Submit your `PQHeap.java` using either the submission web site or (from `remus/romulus`) the `cssubmit` command:

```
cssubmit PQHeap.java
```

and use the numeric menu to select the correct course and assignment. You do not need to submit the `PQSorted.java`, `PQUnsorted.java`, or `Tester.java` files.

Type up your written responses and bring a hard copy to class.

This assignment is due on Friday, March 2, 9:00am.