Programming Overview:

Implement a priority queue according to the `PriorityQueue.java` interface provided. Your implementation should be written in a file named `ThreeHeap.java`, representing a three-heap. A three-heap is a min heap like the binary heap we've been discussing in class, except that each element in the tree structure can have up to three children instead of up to 2 children. Everything else about heaps that we've discussed translates from the binary heap to the three-heap: the structure property and the heap order property should hold for your heap to be correct.

Implement your `ThreeHeap.java` as an array, as we discussed in lecture for binary heaps. Your ThreeHeap should still follow the contiguous structure (complete tree) form of the binary min heap discussed in class. For your `ThreeHeap.java` implementation, be sure it implements the `PriorityQueue` interface provided, and do not use parts of the Java collections framework. Your `ThreeHeap.java` implementation of a priority queue only needs to work with doubles. The buildQueue operation for your ThreeHeap should run in at most O(NLogN) time asymptotically. Described below in the extra credit section, you can earn extra points for implementing buildQueue in O(N) time. **Your buildQueue operation should reset your ThreeHeap to be empty before adding the given elements to the priority queue. A better name for this method would have been replaceQueue.**

Your priority queue should allow duplicates. That is, two or more copies of the same value should be allowed to exist in the heap at the same time. For example, when you call `deleteMin` and you have say {3, 3, 6, 7} in the heap, it would just return one of the 3's, then on the next deletemin it would return the other 3. It does not matter which 3 is returned first. This corresponds to a "tie" in priority - like two print jobs of the same size. According to our definition of priority queue, the only thing that must be guaranteed is that all the 3's will be returned before a 6 or 7 would be returned, and that the 6 would be returned before the 7.

Your implementation should automatically grow (if interested you may also make it shrink - this is optional) as necessary. Note: Growing an array implementation one element at a time is not likely to be the most time efficient solution. For your array-based implementation, you should start with a small array (say, 10 elements) and resize to use an array twice as large whenever the array becomes full, copying over the elements in the smaller array. It is allowable (but not required) to do this by using the Arrays.copyOf method found in java.util.Arrays. You may use the length field of an array.

We will be testing the code that you submit - so we will expect that you have done the same! Any testing code should be submitted in a file called `MyClient.java`. This can be a normal client program that uses your ThreeHeap (or you can use JUnit if you want). Your testing code will not be graded on style, as long as you have made a reasonable attempt at testing the functionality of your heap, you will earn the points.

For this assignment we will be grading more strictly for things like style and efficiency than we did on HW #1. Make sure you are commenting your code, looking for redundancy,

structural mistakes, formatting of your code, efficiency of your algorithms, and preserving the abstraction of your Heap.

Provided code:

1. PriorityQueue.java - interface that corresponds to a PriorityQueueADT for doubles.
2. A helper class: EmptyHeapException.java. (You should not need to import anything to use EmptyHeapException. Just place `EmptyHeapException.java` in the same folder as your other java files.)

Write-Up Questions

Remember, we have written homework guidelines available as a resource.

1. **Testing:** Please briefly discuss how you went about testing your heap implementation.
2. **Big-O Analysis of your Heap:** What is the worst case big-O running time of buildHeap, isEmpty, size, insert, findMin, and deletemin operations on your heap. [For this analysis you should ignore the cost of growing the array. That is, assume that you have enough space when you are inserting a value.]
3. **Asymptotic runtime:** For each of the following program fragments, determine the asymptotic runtime in terms of n.

```
a. public void mysteryOne(int n) {
b.   int sum = 0;
c.   for (int i = n; i >= 0; i--) {
d.     if ((i % 5) == 0) {
e.       break;
f.     } else {
g.       for (int j = 1; j < n; j*=2) {
h.         sum++;
i.       }
j.     }
k.   }
l. }
```

m.

```
n. public void mysteryTwo(int n) {
o.   int x = 0;
p.   for (int i = 0; i < n; i++) {
q.     for (int j = 0; j < (n * (n + 1) / 3); j++) {
r.       x += j;
s.     }
t.   }
u. }
```

v.

```
w. public void mysteryThree(int n) {
x.   for (int i = 0; i < n; i++) {
y.     printCats(n);
z.   }
aa.   }
```

```
bb.
cc.    public void printCats(int n) {
dd.      for (int i = 0; i < n; i++) {
ee.        System.out.println("catsmoop");
ff.      }
gg.    }
```
hh.

4. **Psuedocode and Runtime:**
    a. Write pseudocode for a function that calculates the largest difference between any two numbers in an array of positive integers with a runtime in $\Theta(n^2)$.
    For example, the largest difference between any two numbers in the array storing the values [4, 6, 3, 9, 2, 1, 20] would be 19.
    b. Can this funtion be written with a runtime Θ(n)? If yes, write the pseudocode. If no, why? What would have to be different about the input to do so?
    c. Can this funtion be written with a runtime Θ(1)? If yes, write the pseudocode. If no, why? What would have to be different about the input to do so?
5. **Amortized Runtime Analysis:** For this analysis you did in question 2, you ignored the cost of growing the array. You assumed you had enough space in the array when inserting the value. Now, for each of the following resize algorithms, re-evaluate the runtime of the insert operation, without assuming that the array has enough space. Explain why each answer is the same or different than your answer for question 2. You do not have to write pseudocode for these algorthims, just explain their asymptotic amortized runtime and why it is or isn't different than your answer for question 2.
    a. You start with an array of size 5. When the array is full, you make an array that has 5 extra slots and copy over your elements to the new array.
    b. You start with an array of size 10. When the array is full, you make an array that is 1.5 times larger and copy over your elements.
    c. The algorithm you used for insert. You start with an array of some size. When the array is full you double the array and copy over your elements.
6. **Above and Beyond:** Did you implement anything for extra credit?

What to turn in:

Due: Friday, Janaury 27th, 11pm

You should submit a zip file to the Canvas dropbox with the name `cse373_17wi_hw2.zip`. This zip should include the following files. Code:

1. `ThreeHeap.java`
2. `MyClient.java`, Your testing code for ThreeHeap.java. (Any reasonable attempt at being the client of your Heap will get credit.
3. Electronic version of your project report in pdf, MS word, or text format (please check with us first if you need to submit in another format).

Extra Credit

(3 points) Implement Floyd's Method in buildQueue so it runs in O(N) time.