**Phase Three**

**Due: Friday, February 14th *immediately after class***

This assignment is the third of 4 phases to code graph theory! Now that we can represent graphs in our computers and store portions of that graph in paths, we need to set up the data structures that the big algorithm uses. In this phase, we’ll code stacks, queues, priority queues, and a couple comparators.

You will turn in 6 classes for phase three.

1. MyStack

2. MyQueue

3. MyPQ

4. DijkstraComparator

5. PrimComparator

6. A tester

You have been provided with an interface, DataStructure, and it can be found on the google drive.

**CLASS: MyStack, MyQueue, MyPQ**

So. Before we can proceed to the actual algorithm, we need to deal with a minor Java annoyance. Unfortunately, the most recent ancestor of Stack and PriorityQueue is Collection, which lacks a remove() method with no parameters. Hence, Stack has pop() and PriorityQueue has poll(), both coming from different parent classes. Furthermore, there is no Queue class (there is an interface), but there is an ArrayDeque class.

We want our algorithm to be completely independent to the data structure it is using to store Paths. Since the API isn't playing nice, we need to write wrappers to Stack, PriorityQueue, and whatever you will use for Queue.

Provided is a MyDataStructure<E> interface. You need to create the MyStack<E>, MyQ<E>, and MyPQ<E> classes. All of these classes should implement the DataStructure<E> interface.

For **MyStack<E>**, use the built-in Stack class in java.util.

For **MyQ<E>,** create a queue using ArrayList, ArrayDeque, LinkedList, or something else.

For **MyPQ**, use the built-in PriorityQueue class in java.util. Furthermore, for the constructor, use the constructor that takes in a Comparator<E> (more in the next section). Use 10 for the size parameter. Refer to the API for more details.

**CLASS: DijkstraComparator, PrimComparator**

As you noticed from the last section, we are going to use Comparator<E>. This interface is nice in that we can have the comparison outside of the classes using it instead of inside like with Comparable. Thus, we can have multiple comparisons.

You will create two classes. For each class, they need to implement the Comparator<Path> interface. By default, the compare method from Comparator takes in two Objects. However, to avoid casting to Path, have the generic after Comparator be <Path>; this way, your compare methods take in two Paths.

Your **DijkstraComparator** should implement the compare method from Comparator. It returns a one if the first Path has a larger path distance than the second one, 0 if equal, -1 if smaller.

Your **PrimComparator** should do the same thing but using the last edge weight instead.

**CLASS: Tester/Runner/Whatever**

Test test test!

You should test each class, and make sure to test your PQ with both comparators!