P10 Food Delivery

Programming II (CS300) Spring 2020
Department of Computer Sciences
University of Wisconsin-Madison

Due: 9:59PM on April 22, 2020
Pair Programming **is NOT Allowed.**

Overview and Learning Objectives:

In order to reduce contacts during the coronavirus outbreak, UW has decided to use robots for food delivery. Through this assignment you will develop a program to assign a robot to each student who orders food online based on the distance between students and available robots. You will implement a PriorityQueue (using the Heap data structure) to help make delivery assignments.

This assignment will give you practice working with the heap data structure to implement a priority queue. You will also get more experience with object oriented design, and dynamic binding by overriding a number of methods that are common throughout the Java API.

Grading Rubric:

5 points	**Pre-Assignment Quiz:** You should not have access to this write-up until after you have completed the corresponding pre-assignment quiz through Canvas.
16 points	**Immediate Automated Tests:** Upon submission of your assignment to Gradescope, you will receive feedback from automated grading tests about whether specific parts of your submission conform to this write-up specification. If these tests detect problems in your code, they will attempt to give you some feedback about the kind of defect that they noticed. Note that passing all of these tests does NOT mean your program is correct. To become more confident in this, you must write and run additional tests of your own.
29 points	**Manual Grading and Supplemental Automated Tests:** When your final grade feedback appears on Gradescope, it will include the feedback from these additional automated grading tests, as well as feedback from human graders who review the code in your submission by hand.
50 points	**TOTAL**

0. Project Files and Setup

Create a new Java Project in Eclipse called: P10 Food Delivery.

1. Student and FoodRobot Classes

Create a new class called Student. Each student should have only 3 private instance fields:

- private int x;
- private int y;
- private int id;

Among these fields, x and y represent the location of a student, and id is a unique number for a student. The Student class has a constructor that should take 3 int variables as input: in the order that they are listed above. This class must also have 3 public methods to access each of the 3 fields separately, and an overridden toString() method that returns a string containing all three fields. The format of your toString() will not be checked by any grading tests, but a useful format to match the sample log in the final step is: "**id**(**x**, **y**)". Since the student data cannot be changed, you don't need setters for them (and are not allowed to include any mutator methods in this class).

Similarly, create another new class called FoodRobot. Each robot should have 3 private instance fields as a student, but each robot has its own unique **name** whose type is String instead of **id**. The FoodRobot class has a constructor that should take 2 int variables and a String variable as input (x, y, and then name in that order). It should have 3 public methods to access each of the 3 fields separately, and an overridden toString() method that returns a string containing all three fields. The format of your toString() will not be checked by any grading tests, but a useful format to match the sample log in the final step is: "**name**(**x**, **y**)". Like the Student data, the FoodRobot's data will not be changed from outside this class, so do not include any mutator methods in this class.

2. Delivery Class

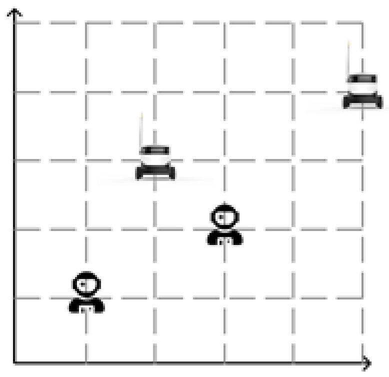
Create a new class called Delivery to keep track of a FoodRobot + Student pair that could be matched to fulfill a delivery. This class also stores the Manhattan distance (https://en.wikipedia.org/wiki/Taxicab_geometry) between the positions of its Student and FoodRobot:

ManhattanDistance(p1, p2) = Math.abs(p1.x - p2.x) + Math.abs(p1.y - p2.y)

Each delivery object contains only the following three fields:

- private int studentId;
- private String robotName;
- private int distance;

The priority that is used to fulfill deliveries favors the Delivery (FoodRobot + Student pair) with the shortest Manhattan distance. If there are multiple deliveries with the same shortest Manhattan distance, the smaller student id has higher priority; and if there are multiple robots with the same distance to that student, then the lexicographically earlier robot name has higher priority.



An example (Image Credit to Rayne Xu)

This figure shows one possible scenario, student 1 is located at (1, 1) and student 2 is located (3, 2), robot A is located at (2, 3) and robot B is located at (5, 4). According to our priority rules, the delivery from robot A to student 2 should be fulfilled first. Then after that is done, the delivery from robot B to student 1 should be fulfilled second.

The Delivery class has one constructor that takes 2 input variables: the first has type Student and the second has type FoodRobot. The Manhattan distance between the input student and robot should be calculated to initialize the distance field in the constructor. In addition, this class should override the methods: equals, toString, and compareTo as described below.

The Delivery class should implement the Comparable interface so that it is comparable to other Delivery objects. This is where you will define the prioritization described above. The return value of your compareTo method should be any negative number whenever the object that it is called on has a higher priority than the object passed as an argument. When the argument passed has a higher priority than the object that this method is called on, the return value should be any positive number. Notice there is no tie in our implementation.

When you override the equals method for the Delivery class, note that this method must take an argument of type Object (otherwise you will be overloading instead of overriding the method). Your implementation of this method will be designed to work differently depending on whether the runtime type of the object passed as an argument to this method has type Delivery, Student, or FoodRobot:

- Delivery: when comparing this Delivery object to an argument of type Delivery, the equals method should return true under two different circumstances: either the studentIds within the two delivery objects are equal, or the robotNames within the two delivery objects are equal. When neither of these circumstances are found, equals should return false.
- Student: when comparing this Delivery object to an argument of type Student, the equals method should return true only when this delivery object's studentId is equal to the student argument's id. Otherwise it should return false.
- FoodRobot: when comparing this Delivery object to an argument of type FoodRobot, the equals method should return true only when this delivery object's robotName is equal to the food robot's name. Otherwise it should return false.

We are providing the following toString implementation for to include within your Delivery class:

```
@Override
public String toString() {
  return "The distance between " + studentId + " and " + robotName +
      " is " + distance;
}
```

3. DeliveryQueue Class

Create a new class called DeliveryQueue, where you should implement your own heap-based Priority Queue. This heap stores elements of type Delivery in a zero-index based array that represents a min heap (where the shortest distance, which has the highest priority is always stored at the root). This class MUST define the following instance and static fields ONLY, and you are not allowed to add any additional instance or static field to this class.

- private static final int INITIAL_CAPACITY = 20;
- private Delivery[] heap;
- private int size;

The next step is to implement a no-argument constructor where you should properly initialize the instance fields. You will also need to implement several public methods within this class:

- public void offerDelivery(Delivery) – this method adds a new delivery to this priority queue. If the heap is already at capacity when this method is called, it should first create a new larger capacity array (twice the previous heap capacity), copy the old arrays contents into this larger array, and then use this array as the heap going forward (until it is filled and replaced with an even larger one).
- public Delivery pollBestDelivery() – removes and returns the highest priority delivery object from this priority queue, and also removes all other delivery objects that "equals" (with matching studentIds or robotNames) that highest priority one. After removing these deliveries, be sure to heapify your array to maintain its heap structure. If the heap is empty, then this method should throw a NoSuchElementException with the message: "Warning: Empty Heap!"
- public Delivery peek() – returns (without removing) the highest priority delivery. If the heap is empty, then this method should throw a NoSuchElementException with the message "Warning: Empty Heap!"
- public int getSize() – returns the number of Delivery objects currently in this priority queue
- public boolean isEmpty() – returns true when this priority queue is empty, and false otherwise

To implement these important methods for a priority queue, be sure to factorize them with help of private methods. In fact, you are required to write the following private help methods:

- private void percolateUp(int index) – recursively propagates heap order violations up
- private void percolateDown(int index) – recursively propagates heap order violations down
- private void heapify() – eliminates all heap order violations from the heap array

There are many additional helper methods that you may find it helpful to write and make use of for this implementation, but which are not required for this assignment. For example: int getParentIndex(int), int getLeftChildIndex(int), int getRightChildIndex(int), void swap(int,int), etc.

In addition, we provide you with the following implementation of a toString method. This method should help you to check and test the individual operations on your heap.

```java
@Override
public String toString() {
  String string = "This DeliveryQueue contains " + size + " elements";
  if (size == 0) {
    return string;
  }
  string += ": [ ";
  for(int i=0; i<size; i++)
    string += "\n" + heap[i].toString();
  string += " ]\n";
  return string;
}
```

4. DeliveryQueueTester Class

Create a new class called DeliveryQueueTester. You will not submit this file to gradescope, but it will be helpful in finding and fixing any bugs that may be in your DeliveryQueue. You will certainly want to create tests here to check the basic functionality of your DeliveryQueue. And if you get feedback from gradescope about anything not working, this will be a good place to start creating a test method to reproduce the kind of bug that is reported to you.

Below is a simple example of a test method that checks offerDelivery, pollBestDelivery, peek, and getSize().

```java
public static boolean testOfferDelivery() {
    // create a new DeliveryQueue
    DeliveryQueue minHeap = new DeliveryQueue();
    // create some Student and FoodRobot objects
    Student one = new Student(1,1,1);
    Student two = new Student(3,2,2);
    FoodRobot a = new FoodRobot(2,3,"A");
    FoodRobot b = new FoodRobot(5,4,"B");
    // create some Delivery objects and add them to the DeliveryQueue
    minHeap.offerDelivery( new Delivery(one,a) );
    minHeap.offerDelivery( new Delivery(one,b) );
    minHeap.offerDelivery( new Delivery(two,a) );
    minHeap.offerDelivery( new Delivery(two,b) );

    // check if the size is correct (2 students * 2 foodRobots = 4 deliveries)
    if(minHeap.getSize() != 4) return false;
    // check first (highest priority delivery to be returned)
    String bestDelivery = minHeap.pollBestDelivery().toString();
    if(!bestDelivery.equals("The distance between 2 and A is 2")) return false;
    // check if the size is correct (only delivery w/student 1 + robot B left)
    if(minHeap.getSize() != 1) return false;
    // check last (lowest priority delivery to be returned)
    String worstDelivery = minHeap.peek().toString();
    if(!worstDelivery.equals("The distance between 1 and B is 7")) return false;

    // only return true after all previous tests pass
    return true;
}
```

5. DeliverySchedulingApp Class

After you make sure that your DeliveryQueue is functioning as expected, you will approach this last step of the assignment. This driver is provided for you to demonstrate how to use your DeliveryQueue to develop a small scheduling application. Double check if you have downloaded this DeliverySchedulingApp.java and example.txt and put them in the P10 Food Delivery workspace. Then you can run this as a Java application. This app is called Food Delivery Scheduler, and you are allowed to type simple commands to enter new student orders, make new robots available for delivery, and to schedule the fulfillment of deliveries within the system. The following is a partial log showing the expected behavior that result from running the commands in the provided example.txt file. Creating more files like this, is another way to further test your DeliveryQueue implementation.

```
Please enter command: f example.txt
Adding Student: 22
Adding Robot: C3P0
Adding Student: 50
Adding Robot: G1ad0s
Fulfilling Order: The distance between 50 and C3P0 is 2
Students waiting: [22(2,2)]
Robots available: [G1ad0s(1,0)]
Adding Robot: Jonny5
Adding Robot: SevenOfNine
Adding Student: 88
Fulfilling Order: The distance between 88 and SevenOfNine is 2
Fulfilling Order: The distance between 22 and G1ad0s is 3
Adding Robot: R2D2
Adding Robot: HAL-9000
Adding Student: 14
Fulfilling Order: The distance between 14 and Jonny5 is 2
Adding Robot: T-1000
Adding Student: 2
Students waiting: [2(0,2)]
Robots available: [R2D2(2,2), HAL-9000(9,0), T-1000(10,0)]
Adding Robot: B-4
Adding Student: 101
Adding Student: 110
Fulfilling Order: The distance between 101 and T-1000 is 1
Fulfilling Order: The distance between 2 and B-4 is 2
Fulfilling Order: The distance between 110 and HAL-9000 is 2
```

6. Assignment Submission

Congratulations on completing this CS300 assignment! After reviewing your code and ensuring that it conforms to the requirements of the course style guide, submit these four files to gradescope: Student.java, FoodRobot.java, Delivery.java, DeliveryQueue.java. Your score for this assignment is based on your submission marked “active” and made prior to the hard deadline.