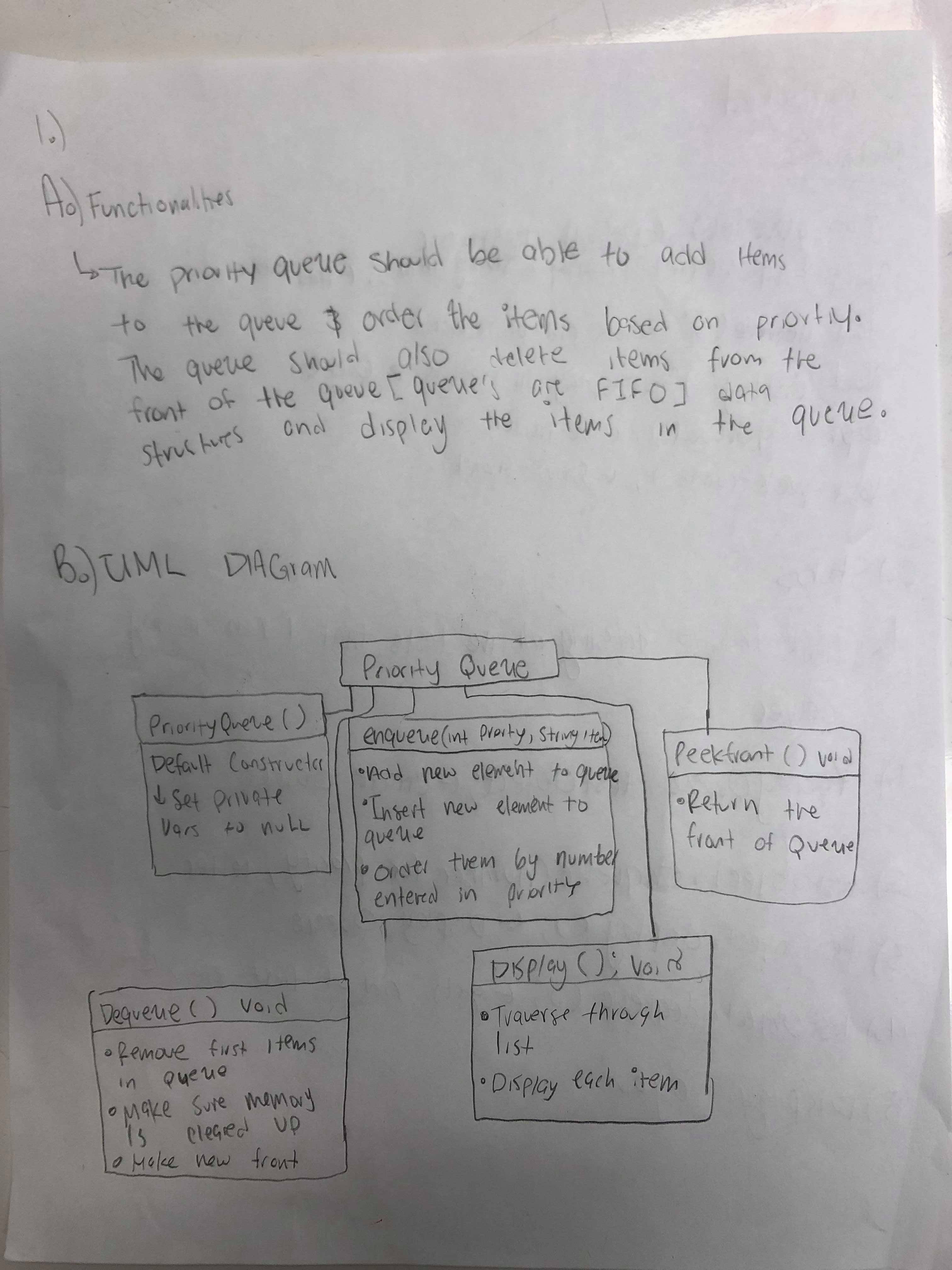
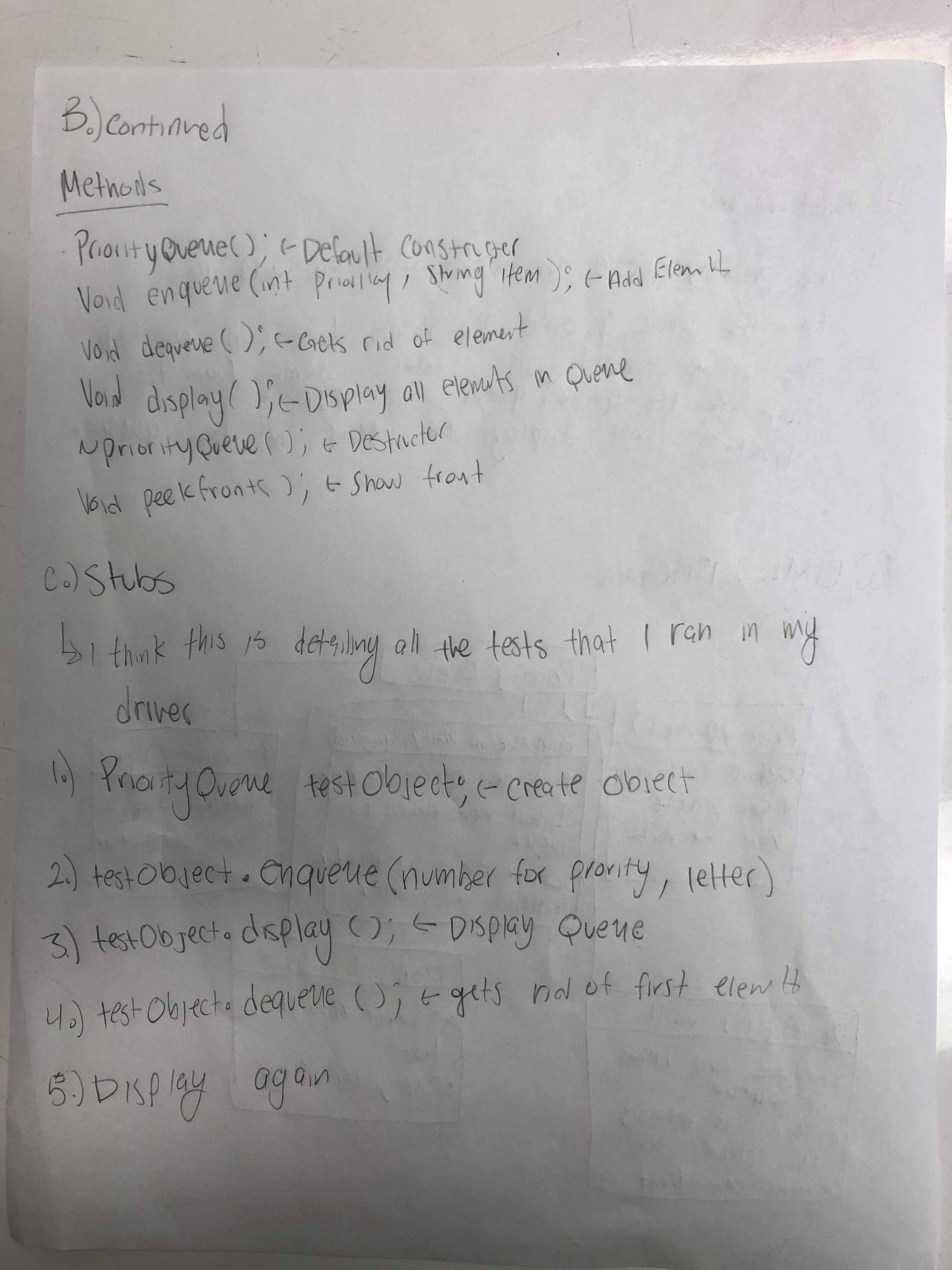
Mir Hossain

CS302

Homework Assignment 3

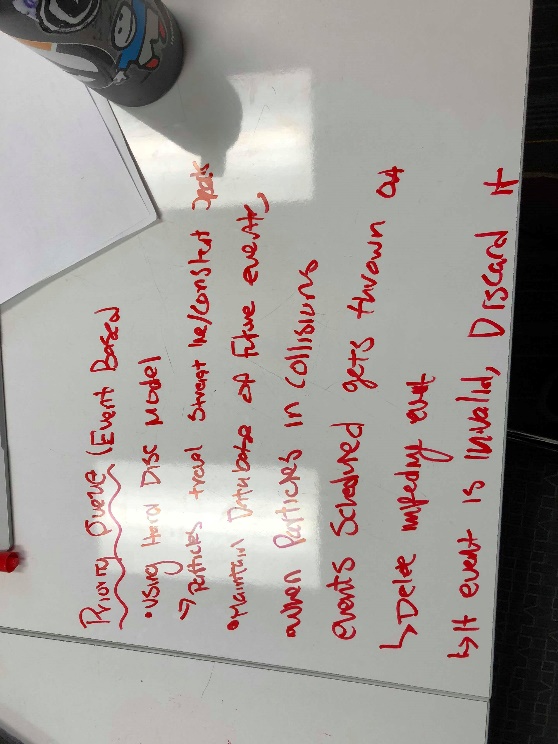
**Exercise 1**

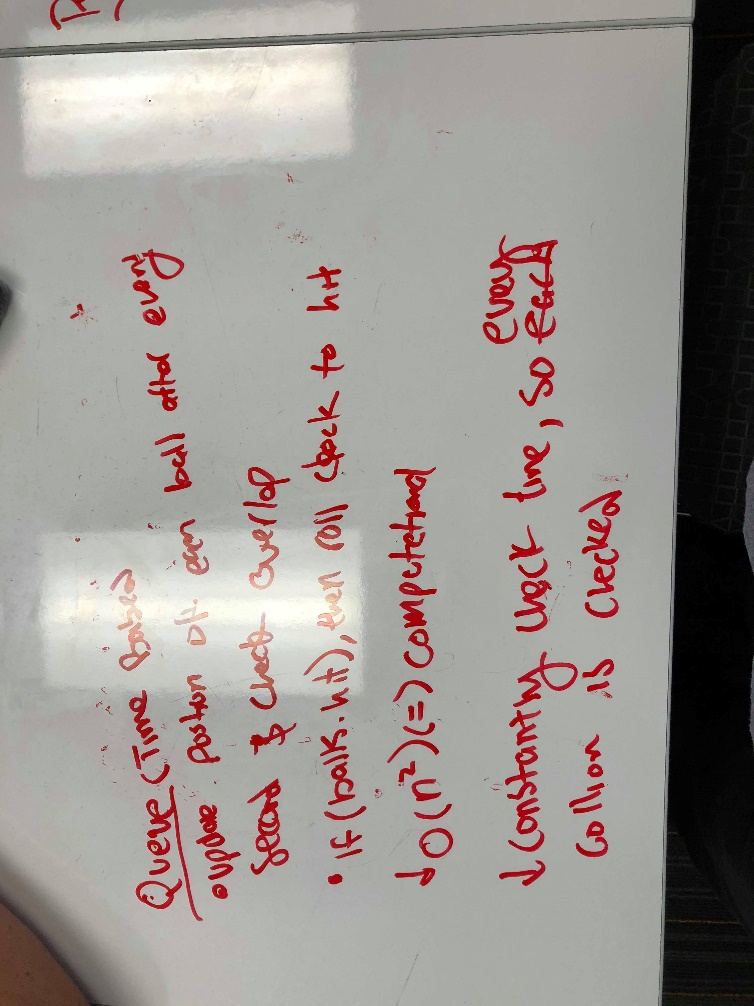


**Exercise 2**

Programming Assignment: Compile proj3 file

**Exercise 3**

****



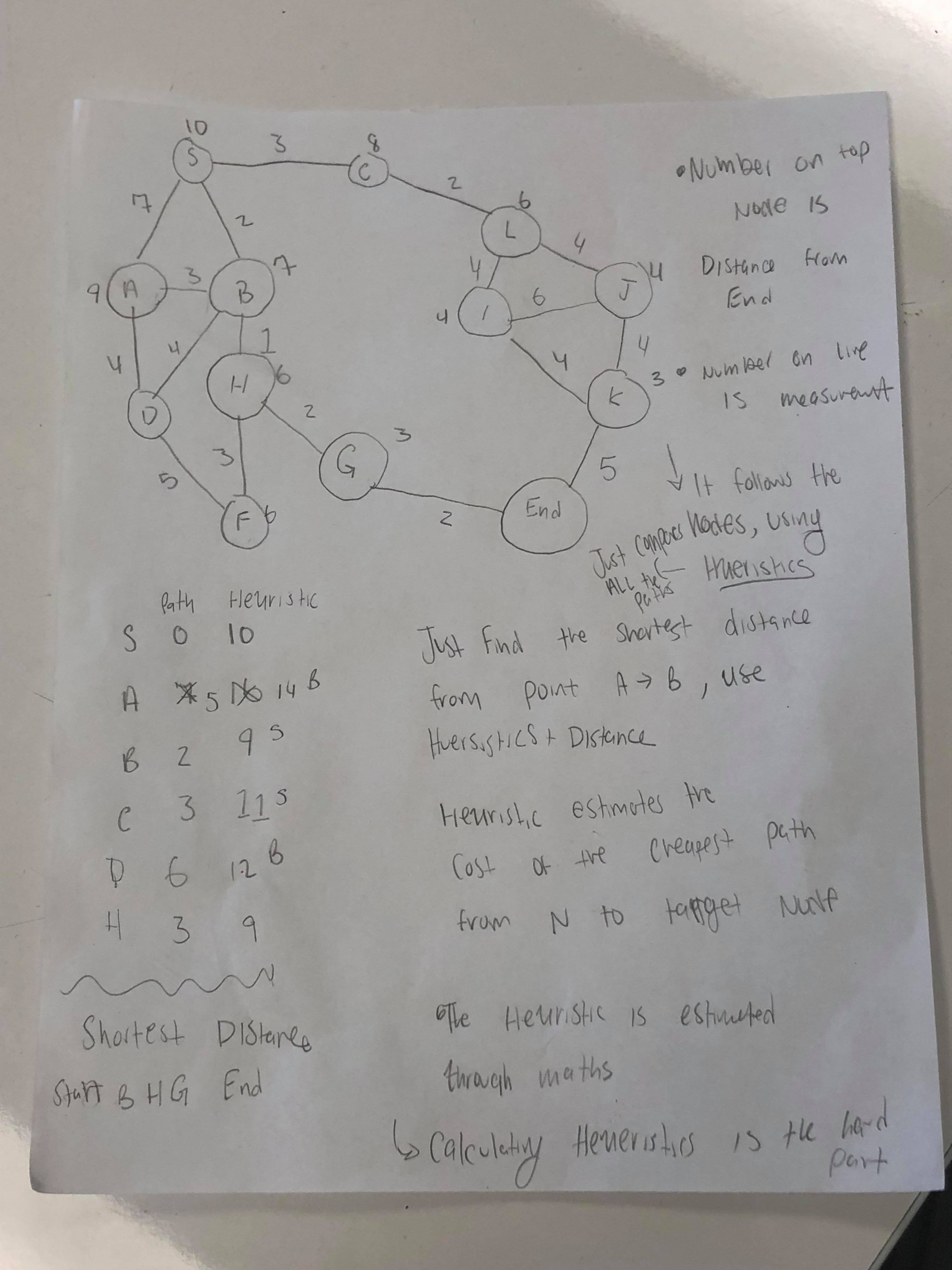
**Exercise 3 Continued**

There are two different solutions to figuring out order of collisions in a game of pool, time driven(using a queue) and event driven(using priority queue).In a time driven solution, the program running would have to track the position of every single ball using some unit of time while simultaneously checking for overlap (collisions). In the instance of a collision the program would have to roll back its clock used for tracking and record the exact time of collision. There are two primary problems with this implementation. First, this time driven implementation is O(n2) complexity. The second problem is that if the interval and unit of time is too large then the program would miss instances of collisions. So, the unit and interval of time would have to be very small to ensure accurate results however, that would cause the program to use a large amount of computation power and would require a significant amount of time.

When implementing an event driven model, we can just track when a ball collides with another ball. The collisions of the balls can be modeled using the Hard Disc Model that says all particles travel in straight line trajectories. This would allow the priority queue to keep track of all possible and future collisions so then the only obstacle would be to order when a collision did occur. As the particles would collide, we could keep them in the priority queue and throw out any invalid event. Then we could order the collisions based on time.

**Answer:**The priority only checks of an instance of collisions while a queue would be checking the position of every single ball using a very small unit of time.

**Exercise 4**

****

The A\* is considered an intelligent searching algorithm because of the method it used to find the end goal. Most search algorithms use a method where it does a check and then finds the end goal eventually. The issue with this is that other search algorithms normally go through every single instance (node) and then would find the end goal. These types of methods would be expensive (computational and time).

The way A\* works is that every single node is given some heuristic value. This value is computed through a method by person making the program. This is more of an estimate than an exact value. Then the path is the actual value from each point to the end goal. An example would be if someone was trying to find the shortest distance from the university to a restaurant. There would be several points on a map each representing a node. The method used to find the shortest distance would use a separate algorithm to determine the heuristic values beforehand and assign each note the computed heuristic value. Afterwards, the program would calculate the paths for each node from the previous node. The measurement for the path has be as accurate as possible. Now that A\* has all the values it needs; the algorithm can effectively traverse through the node. At the starting point, the algorithm would compare each node connected and then add the heuristic to the path. After comparisons would be finished the algorithms would choose the node with the smallest (heuristic + path) and put them in a priority queue. The node with the smallest value would have the highest priority and the algorithm would continue this operation until the end-point would be found. In the case of obstacles (ex: closed roads), then the implementation can just flag nodes and make it so A\* wouldn’t bother checking them. Unlike other search algorithms, A\* doesn’t necessarily check every single node, it only checks the ones that make sense. This makes the searching algorithm much faster and require less power.