**Customer Priority Queue Design Document**

Ian Cowan

**Purpose**

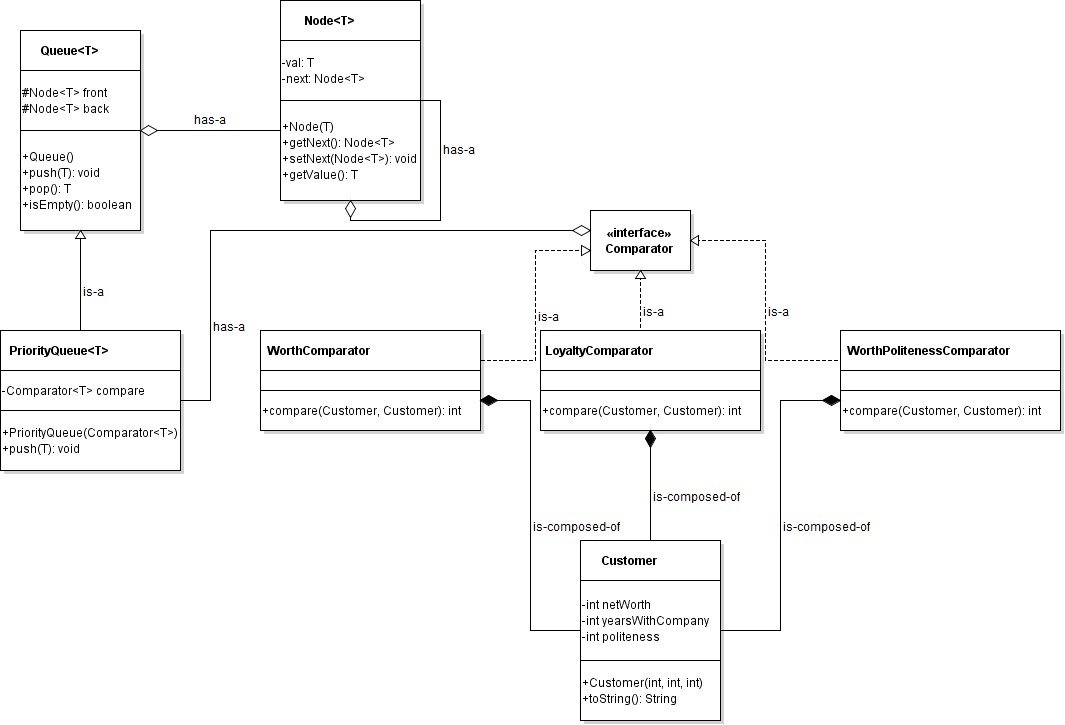
We want to have a data structure that can store information in a sorted order, from greatest to least, and we want this data structure to replicate a queue, but return the information in the sorted order. This should be implemented in a way such that it can be used with any objects, but we will specifically be storing customer objects to track customers. The customer objects should store relevant information, like loyalty, net worth, and politeness of the customer. We should be able to compare the customers in a variety of different ways, and we will use this to sort them in our data structure we create.

**Specifications**

Our data structure, we’ll call a priority queue, should be able to sort from greatest to least depending on the comparator we pass upon creation of the priority queue. The priority queue should be able to store any type of objects we want to store within it: the priority queue should be implemented using generics. We want to use a link based queue that we already created, so the priority queue will be link based and we will only override one method from the queue and this will be the method that takes care of the sorting, the push() method.

The Customer class should contain three different comparison classes that extend the built in Comparator interface. There should be a comparator to compare two customers’ worth, loyalty, and then the last comparator should compare two customers’ worth but if they are worth the same amount, it should then compare the customers’ politeness. Then if their politeness is equal, it should return that they are equal.

**Design Overview**



**Figure 1. Customer and Priority Queue UML Diagram**

Our PriorityQueue class is a generic where the user of the class must declare the type being stored within the queue prior to use. A PriorityQueue object has a Comparator object of the type being stored within the queue, and this is used for the sorting of the queue. We must override the constructor, to take a Comparator object upon creation, and the push() method so we can actually sort when adding to the queue. The queue implementation that we are using stores the information using links, so this will affect how the PriorityQueue push() method will work.

The Customer class is going to be used to store customers, and we are storing a customer’s netWorth, yearsWithCompany, and politeness. The Customer class is composed of three other classes that are all used to create comparators, WorthComparator, LoyaltyComparator, and WorthPolitenessComparator. All of these are in the same format where they have only one method, compare(), that takes two Customer objects and then returns either -1, 0, or 1 depending on whether the first Customer object is less than, equal to, or greater than the second Customer object respectively. Each of these comparator classes implement Java’s built in Comparator interface so that we can store them as a Comparator and they can be used interchangeably for different priority queues.

**Detailed Design Overview**

When creating a PriorityQueue object, a Comparator object of the same type must be passed for the comparisons to be made when adding to the queue. When an object is then added to the priority queue, it passes through a series of tests to check and see where it should actually be inserted to maintain the proper order. First, it checks to see if the queue is empty, using the isEmpty() method. If the priority queue is empty, it simply sets both the front and the back of the priority queue to the newly created Node. If it is not empty, the object then goes through a series of three comparisons to check and see where it should be inserted into the queue. The first one is if the new object is greater than or equal to the front. If it is, it will simply be inserted into the front of the priority queue and the operation will be complete. Next, we check to see if the object is less than the back. If it is, it will be inserted into the back and the operation will be complete. If neither of these cases are true, we must loop through the queue and find the proper location for the object.



**Figure 2. Pseudocode for within the push() method**

This loop works by checking to see when the object in question is greater than an object in the queue, and then the object is inserted and the loop is broken out of carefully. We do not want the object to be inserted beyond the first location of insertion because greater than is transitive, so if the loop continued the object would be inserted in between every remaining object.



**Figure 3. Pseudocode within the loop within the push() method**

Finally, once this is complete the new object should have been inserted somewhere within the priority queue because these are the only possible three cases.

When pushing an object, we must be sure to update the links correctly, because this is not as simple as adding an object to the back of the queue. When inserting, we must be sure that the newly created Node gets linked to the next node in line, and the previous node gets linked to the new node that we are inserting. Failing to make sure these are linking correctly would cause the queue to not be linked correctly, possibly branching off in a completely different direction (this is to be discussed more in the analysis). Popping simply takes the top node off the queue, returns the object within the node and does not readd the node to the queue. When checking to see if the queue is empty, we simply check to see if the front is null, that is the front is not linked to any nodes, and if so we will return true.

**Analysis**

The time complexity of each method within the priority queue are listed in table 1 below. Most of the operations are fairly simple when considering the time complexity, except for push() could take a considerable amount of time depending on the size of the queue.

|  |  |
| --- | --- |
| Method | Time Complexity |
| PriorityQueue() | , Constant Time |
| push() (if the queue is empty) | , Constant Time |
| push() (if the new object is greater than the front) | , Constant Time |
| push() (if the new object is less than the back) | , Constant Time |
| push() (in any other case) | , Linear Time |
| push() | , Linear Time |
| pop() | , Constant Time |
| isEmpty() | , Constant Time |

**Table 1. Time complexity of PriorityQueue methods**

We made a decision to sort the objects when they were pushed to the priority queue, within the push() method. This slows down the pushing process, but also leaves the popping process time complexity alone. Our other option would be to sort each time the user calls pop(), but this would slow down the popping process. We chose to sort when pushing because this allows the queue to be stored in an ordered manner and there is a greater chance that when pushing an object is going to be placed at the front or the back whereas when popping the queue would most likely have to be looped through many more times.

The choice to check the front and back cases set up the loop to work properly. The loop skips the front because in order for inserting to work, there must be a previous node to update. The loop also would never check the back because of how it always checks to see if the new object is greater than the object that already exists. This would end up never checking to see if the new object should be placed in the back of the queue.

When looping through the queue using the while loop, there must be a place that the loop is broken out of using “break” because if not, the new object would be inserted through before each of the next objects until the queue ends. This would break the queue because there would then be objects of greater value behind objects of less value, breaking the sorting of the queue, and it would also end up having up to the length of the queue at insertion minus one of the new object inserted into the queue. This is not the appropriate behaviour.

If the linking does not get updated correctly, as referenced in the detailed design overview, the newly created node may either not be accessible or may point to the previous node, causing the nodes after the inserted node to not be accessible. The following diagrams show how inserting a node should work properly, and then three broken cases caused by improper linking.



**Figure 4. Proper node linking when pushing a new node**



**Figure 5. Improper node linking when pushing a new node, causing the new node to not be accessible**



**Figure 6. Improper node linking when pushing a new node causing the new node to not be accessible and the previous nodes after the previous node to no longer be accessible**



**Figure 7. Improper node linking when pushing a new node causing the new node to be accessible, but causing the previous node and the new node to be read as infinitely repeating nodes**

**Conclusion**

Overall, our implementation of the PriorityQueue and Customer classes work fairly well. All of the linking is working correctly, and the queue sorts correctly when pushing new objects. The design we chose mitigates most of the risk that could go along with the priority queue, especially the risks associated with the links, and is relatively not very complex in regards to the time complexity. Most of the methods run in constant time, and the only one that does not is linear time. This project is very scalable because it uses generics, so the priority queue can be used with any class that uses comparators. Overall, this project can be considered as a success and all of the requirements have been met.