CSCI 2720, Data Structures

Assignment 2

The goals of this assignment are:

1. Introducing you to a new structure of the queue, namely, the priority queue.
2. Continue practicing good software engineering approach, as well as, using object-oriented concepts in your design and implementation.

Priority Queue

A priority queue is a queue in which items are stored in an order determined by a numeric priority value; where a task with priority 1 comes before a task with priority 2. This interpretation means lower numerical value has a higher priority. Thus, a priority queue is not a FIFO structure.

Requirements:

1- Implement a **generic** ADT **PriorityQueue** using C++ templates and a **single linear linked list** that stores the elements in priority order.
2- The priority queue files should include the functional specification (comments) of all operations.
3- Implement the driver program **"PriorityQueueDr.cpp"** to test the priority queue implementation.

This queue type requires one external pointer. However, to enhance the implementation, you can use a list with two pointers: front and rear.

If you defined two pointers front and rear, I should see code enhancement in your enqueue function.

The complexity of the operations enqueue, makeEmpty and printQueue should be O(n), all other queue operations should be O(1).

PriorityQueue Functional Specification:

The ***PriorityQueue*** class is exported through the PriorityQueue.h header file. The operations you need to implement are always the same – you can use your header file from assignment1" the array queue"- what changes is the internal implementation and data members; which is of little concern to the client. The methods exported by the ***PriorityQueue*** class are specified in table 1 on the next page.

As you can see from the table, the methods in the ***PriorityQueue*** class are like those in the Queue ADT you have implemented in assignment 1.

There are two differences:

(1) The **Enqueue** method takes an extra argument indicating the priority of the new item; this priority determines the appropriate position of the new item.

(2) There is a new **PeekPriority** method, which returns the priority associated with the queue item with highest priority (the item at the front of the queue).

Table 1: Priority Queue Functions

enqueue	Adds value at the appropriate position in the queue as determined by the numeric value *priority*. As in conventional English usage, lower numeric values correspond to a higher urgency, so that a task of *priority* 1 comes before a task with priority 2. If the priority parameter is missing, it defaults to *priority* 1
dequeue	Removes the value at the front of the queue and returns it to the caller. Calling *dequeue* on an empty queue throws an exception
peek	Returns the value of the most urgent item in the queue without removing it from the queue. Calling **peek** on an empty queue throws an exception
peekPriority	Returns the priority of the most urgent item in the queue without removing it from the queue. Calling **peek** on an empty queue throws an exception.
makeEmpty	Removes all elements from the queue.
length	Returns the number of elements currently in the queue
printQueue	Prints Queue items ordered from front to rear
isEmpty	Returns true if the queue is empty and false otherwise
isFull	Returns true if the queue is full and false otherwise

Of course, you need a constructor and a destructor!

Implement a test program **"PriorityQueueDr.cpp"** to test the priority queue (you may use assignment 1 driver, changing the enqueue call and any change in the commands).

This program should initialize a priority queue object and then manipulate it by reading commands from an input text file called **"inFile.txt**". Each of these commands begins with a command keyword that usually appears alone on the input line and each command keyword corresponds to one of the methods in the ***PriorityQueue*** class. Your test program should implement those commands by calling the corresponding method and then outputs the result, if any, to a text file called **outFile.txt**. The list command calls the printQueue function, which is extremely useful during testing. (See testing section on page 4)

The enqueue command is followed by two additional values: the value to be added to the priority queue (a string) and its associated priority value (an integer) this command should call the **enqueue** function which requires linear time to find the correct insertion point, but can then add the new value in constant time; the **dequeue** method runs in $O(1)$ time. You may enhance the implementation by avoiding search for some cases!

Note: Queue Node is defined as a struct template:

```
template class<ItemType>
struct QNode{
ItemType  info;
int priority;
QNode<ItemType> * next;};
```

In case of two items with same priority we treat them as in FIFO structure. The following will help with understanding how should your ***PriorityQueue*** work with character data type for ItemType:

Function call	**Output of PrintQueue after the function call**	
Q.enqueue('C', 2)	**C**	
Q.enqueue('D', 4)	**C D**	
Q.enqueue('A', 1)	**A C D**	
Q.enqueue('B', 2)	**A C B D**	
Q.dequeue ( )	**C B D**	
Q.peek()	**C B D**	**note: output of cout<< Q.peek() is C**
Q.peekPriority()	**C B D**	**note: output of cout<< Q.peekPriority() is 2**
Q.dequeue ( )	**B D**	

What to submit?

Submit your ***Assignment2*** directory to **odin** , the directory must contain the files:

1. PriorityQueue.h
2. PriorityQueue.cpp
3. PriorityQueueDr.cpp
4. inFile.txt
5. The makefile and a README.txt file telling us how to compile your program and how to execute it. Include your name at the top of this file.

Testing:

We will test your implementation using our **"inFile.txt"** file with String data type; results are to be saved in outFile.txt.

When you test your program, before submission, make sure that your program **PriorityQueueDr.cpp** handles all commands and exceptions. Commands corresponds to our **"inFile.txt"** file (i.e use enqueue, not Enqueue) Always have the quit command in your file to avoid an infinite loop. See the content of "inFile.txt" in the left box below.

For the test cases in the "inFile .txt" on the left box below, the expected output of your program is shown in the right box (content of outFile.txt).

Note: Our test cases are not limited to the sequence in this "inFile.txt".

inFile.txt

```
enqueue C 2
enqueue D 4
enqueue A 1
enqueue B 3
list
dequeue
list
peekPriority
peek
length
dequeue
dequeue
list
isEmpty
dequeue
dequeue
enqueue K 4
enqueue G 1
list
length
makeEmpty
list
length
isFull
dequeue
quit
```

outFile.txt

```
C is enqueued
D is enqueued
A is enqueued
B is enqueued
Queue: A C B D
A was dequeued
Queue: C B D
Priority of the front item is 2
Front item is C
Number of items in the Queue is: 3
C was dequeued
B was dequeued
Queue: D
Queue is not empty
D was dequeued
EmptyQueue exception thrown
K is enqueued
G is enqueued
Queue: G K
Number of items in the Queue is: 2
Queue:
Number of items in the Queue is: 0
Queue is not full
EmptyQueue exception thrown
Testing completed
```

Evaluation of this assignment will include:

1) Evaluating test cases using a pass or fail metric.
2) Examining code to check if big O constraints are satisfied.
3) Programming style. In addition to ensuring your program compiles and runs, you are also responsible for proper documentation. Proper documentation includes proper function commenting (i.e. purpose, pre-, and post- conditions) and explicit directions on how to compile and run your programs.
4) Submitting to a wrong directory or submitting different file names will NOT be graded.

See course syllabus for late submission evaluation

Clarification

After studying analysis of algorithms, you should choose the implementation that gives you better O(n), enhance O(n) for some cases, or choose simplicity and time to code! The enqueue would be enhanced if you define two-pointers (front and rear), but you need more statements/conditions in all Modifier functions (functions that change the queue state) including the constructor. Thus, **if you defined two pointers front and rear, I should see code enhancement in your enqueue function.**

For this assignment, no files are attached to restrict your work. In your PriorityQueue You may add private/helper functions. For example, a function to search for the position of the added item.
You may reuse your FIFO queue driver (assignment 1 driver).

If you defined one pointer front: The **enqueue()** is your SortedLinkedList addItem function with a slight modification; we will search for the correct position and insert; The difference that we will **allow duplicate keys** (priorities). You need to modify findItem(QNode*& Predecessor) to stop at the correct position. Do not throw the Duplicate exception if the new item is already in the list!
Example:
if we have 3 customers in the queue with same priority you need to insert after the last one. In the queue below:
Zara 2--> Adam 3 --> Saleh 3 --> David 5 --> NULL (front = address of the node containing Zara.)
if you insert Ahmad 3 the queue will be:

Zara 2--> Adam 3 --> Saleh 3 --> Ahmad 3 -->David 5 -->NULL

Now let us insert Andrew 2, queue will be:

Zara 2--> Andrew 2 -->Adam 3 --> Saleh 3 --> Ahmad 3 -->David 5 -->NULL
Great! we compare according to priority and then we insert based on arrival time!