Computer Science II – Data Structures and Abstraction (CS23001)

Lab 6: Queues

Objectives:

The goal of this lab is to learn the following:

- Learn about queues
- Practice use of dynamic linked lists and Arrays

Lab Submissions:

Please submit all Source Code organized in separate folders (numbered as exercises) online through Blackboard by the end of the day on next Friday. No printouts or emailed submissions will be accepted.

Please ensure that your source code is properly organized (indented, and commented). Create separate folders for each exercise and ensure that all needed files (headers, source, makefile) are in each folder. Zip all folders together before submitting online..

In a comment on top of your driver file, write the Test Cases that you have used to test your program. Be sure you choose the Test Cases according to the "Testing" rules covered both in class/slides and in your book.

Be sure that you always separate the class declaration from its implementation and you create a Makefile for its compilation. This requirement MUST be satisfied in **every lab** of this course.

Lab Grading:

*The grade for each lab is based on lab attendance and code (100%). Lab attendance and submission of Source Code are mandatory for each lab. Grade distribution for **lab work** includes coding (90%) as well as proper indentation and commenting of the code and test cases (10%).*

Good Luck!

Lab Deliverables:

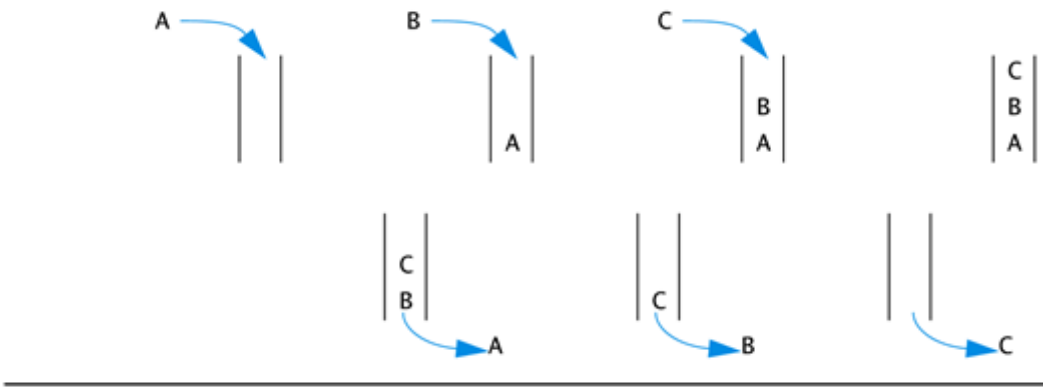
- Complete working source code of each exercise (where it applies).
- Screen shot for each exercise to show that the code works.

6.1 Introduction to Queues

In Chapter 8 you learned about queues. There are two ways to implement queues, one using arrays and other using linked lists. We have previously used pointers to define *linked lists* which will now be used to implement a queue (In 6.1 and 6.2).

A **queue** is a linked list with *front* and *rear* pointers. Items are added to the queue using the *rear* pointer and removed from *front* pointer. Queue uses the principle of First-In First-Out (FIFO) which is shown below:

DISPLAY 13.20 A Queue



The operation to add an item in the queue is called **push** or **enqueue** operation and will always use *rear* pointer. The operation to remove an item from the queue is called **pop** or **dequeue** operation and will always use *front* pointer. The queue implementation will look like this:

DISPLAY 13.21 Interface File for a Queue Class *(part 1 of 2)*

```
1  //This is the header file queue.h. This is the interface for the class Queue,
2  //which is a class for a queue of symbols.
3  #ifndef QUEUE_H
4  #define QUEUE_H
5  namespace queuesavitch
6  {
7       struct QueueNode
8       {
9            char data;
10           QueueNode *link;
11      };
12      typedef QueueNode* QueueNodePtr;
13
14      class Queue
15      {
16      public:
17           Queue();
18           //Initializes the object to an empty queue.
19           Queue(const Queue& aQueue);
20           ~Queue();
```

DISPLAY 13.21 Interface File for a Queue Class *(part 2 of 2)*

```
21          void add(char item);
22          //Postcondition: item has been added to the back of the queue.
23          char remove();
24          //Precondition: The queue is not empty.
25          //Returns the item at the front of the queue and
26          //removes that item from the queue.
27          bool empty() const;
28          //Returns true if the queue is empty. Returns false otherwise.
29      private:
30          QueueNodePtr front;//Points to the head of a linked list.
31                             //Items are removed at the head
32          QueueNodePtr back;//Points to the node at the other end of the
33                            //linked list. Items are added at this end.
34      };
35  }//queuesavitch
36  #endif //QUEUE_H
```

Exercise 6.1

Implement the queue class as shown above. Use your queue to store the names of 5 students waiting outside Student Services to meet with advisors. You may want to add more data to each node including student ID, Major, etc.

Add a member function displayQueue which displays the contents of the queue. Write the main function to test the queue. Call your program *ex6_1.cpp*. Be sure to show the contents of queue after adding or removing each student.

It is strongly encouraged that you make your program more user friendly by providing a menu of options for user (for example: user can choose to add an element to the queue or remove it or display entire queue).

Note: Queue can be implemented using arrays as well as linked list. This exercise requires you to implement the linked list version of queues but you should also look at the array implementation to enhance your understanding.

6.2 Queue using STL

C++ Standard Template Library (STL) has a queue template class that can be also be used. You can find more details about the STL version in Display 18.9 of Chapter 18 of Savitch textbook (for CS1).

The STL version looks like this:

```
template <class Item>
class queue<Item>
{
public:
        queue();
        void push(const Item& entry);
        void pop();
        bool empty() const;
        Item front() const;
        ...
};
```

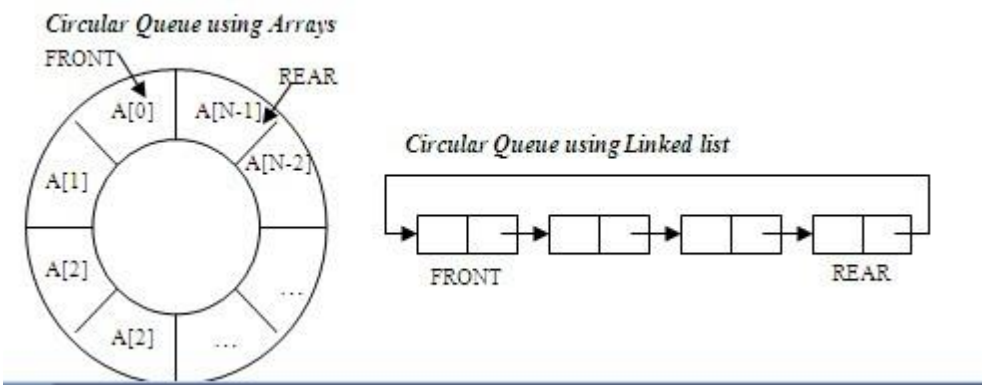
Exercise 6.2

Repeat the last exercise using STL version of queue. Call your program *ex6_2.cpp*

6.3 Circular Queue

Another useful data structure is **circular queue.** If you recall a queue can be implemented using an array or a linked list. The array implementation of the queue uses the first (0) and last (N-1) indices of the array as the *front* or *rear* of the queue while in the linked list implementation the *front* and *rear* pointers "point" to the first and last nodes of the queue respectively. In the linked list implementation of the circular queue, the pointer to the 'link' field in the "last node" points back to the "first node" of the queue whereas in the array implementation, the next index after the N-1 index will be 0. (Of course, after this change it is no longer clear which node, if any, is intrinsically "first").

If we use a circular queue, then we need only one pointer to implement the queue, since the front node and the rear node are adjacent nodes, as show in picture below:



We observe that in the picture above we have called the two pointers of the linked list as FRONT and REAR. In practice, we can only use a single pointer called `rear_ptr` because it turns out that it gives a more efficient implementation than having it point to the first node in the queue, since the access is not random as in the array.

Exercise 6.3

Write the circular queue class (you may use the linked list version or dynamic array/vector based implementation). Call it *cqueue*. Use your circular queue class to simulate a real world 3-color (Green, Yellow, and Red) traffic light system. The Red and the Green lights stay on for a duration selected by the user, while the Yellow stays on for a very small amount of time that is already predefined in the system. After the duration has passed, the light switches to the next color. Use your imagination to implement any additional feature the traffic light may have.

Call your program *ex6_3.cpp*.