

Lab 6.1 Queues

Project 6.1 An Improved Queue Class

Project 6.2 Queues—A Circular Linked Implementation

NOTES TO THE INSTRUCTOR

The purpose of this lab exercise is to lead students through an examination of a container class, namely, a Queue class, and then in the accompanying projects, to have them develop improved versions on their own. These tie in directly with Chapter 8 of the text *ADTs, Data Structures, and Problem Solving with C++, 2E*, which parallels closely the development of the Stack class in Chapter 7. One way that it has been used successfully is for students to work through Chapter 8 and this lab exercise on their own as the Stack class is developed in detail in class presentations.

The Stack class is implemented in various ways in Chapters 7 and 9, improving its efficiency and/or flexibility:

- Use a static array to store the elements
- Use a dynamic array to store the elements
- Use a linked list to store the elements
- Make it a class template
- Use a vector to store the elements (leading to STL's stack adapter)

The implementation of the Queue class can be developed by students in a similar way:

- Lab 6.1: Use a static “circular” array to store the elements
- Project 6.1: Use a dynamic “circular” array to store the elements
- Project 6.2: Use a circular linked list to store the elements
- Project 7.2: Make it a class template

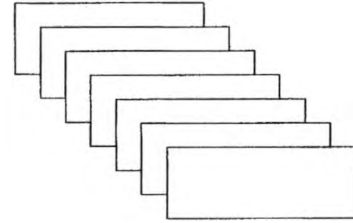
Notes:

1. In the lab exercise, the students will be using the static-circular-array-based Queue class from the text. The files `Queue.h` and `Queue.cpp` for this class and a driver program `qtester.cpp` can be downloaded from the website whose URL is given in the preface.
2. For the projects, the students can save themselves considerable typing if they use these `Queue.h` and `Queue.cpp` files and modify the implementation details.

Course Info: _____ Name: _____

Lab 6.1 Queues

Background: Standing in line is one of those universal experiences that we've all had, whether it is at an airport, waiting to check in; in the line outside the ice cream place, waiting to get a cone; or winding our way through a maze at an amusement park, waiting to get on a ride. Each of these different situations puts us in a *queue*, which is another word for a "waiting line." This rather natural organization is sometimes abbreviated to *FIFO* (*First In First Out*) to distinguish it from a stack structure which is *LIFO* (*Last In First Out*).



Objective: In this lab you will first work through the development of the *static-array-based* Queue class in Section 8.2 of the text *ADTs, Data Structures, and Problem Solving with C++, 2E*. You should work through Section 8.2 before (or while) you work through the lab exercise. This will be a big help, as you will be doing the same thing here as is done in the text. You will need your textbook for the exercises on the last page of this lab exercise.

After you understand the implementation of a queue using a static array to store the queue elements, your instructor may have you do Project 6.1, which modifies the implementation to use a *dynamic array* as described at the end of Section 8.2, or Project 6.2, which uses a *circular linked list* to store the queue elements, or both.

Queue as an ADT

If you think of a queue as an ADT (Abstract Data Structure)—as described in Section 8.1—then you have to think in terms of both its data items and its operations. A queue is an ordered collection of data items with the property that items are only removed at one end called the *front* of the queue and can only be added at the other end called the *back* of the queue; and the basic operations are:

construction: create an empty queue
empty: determine if the queue is empty
enqueue: add a data value to the back of the queue
front: retrieve the data value at the front of the queue
dequeue: remove the data value at the front of the queue

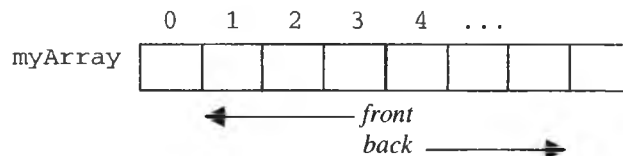
Designing the Queue class

Many considerations go into the design of any class. Two of the major ones are:

- Selection and/or design of appropriate *data structures to store the data items* of the ADT
- Development of *algorithms for the basic operations* of the ADT

We consider first the question of what data structures to use to store the queue elements. In practice, a queue may grow to have a large number of elements, and you don't know in advance how much storage will be needed. In our first implementation, however, we will proceed in the same way as the first implementation of the *Stack* class in Chapter 7, in which an upper bound on the size of the container is set and a static array is used to store the elements.

It is helpful to use graphic representations of our data structures in order to visualize what is happening during the design process. Since we are using arrays to implement a queue, an appropriate diagram is a set of boxes representing the elements of the array with the indices of the array elements shown above the diagram:




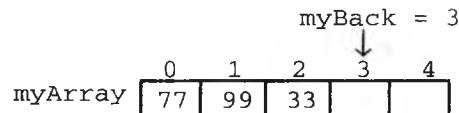
We can study the behavior of the queue by making successive diagrams to show the state of the storage structures as we add and remove queue elements.

A First Attempt: One way to use this array to store queue elements is to imitate our first attempt at implementing a stack:

- Keep the front of the queue fixed at position 0.
- Use a `myBack` data member to keep track of the position in the array at which an element can be added, that is, the position following the last queue element.

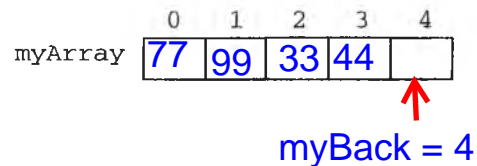
However, this would mean shifting the array elements every time a value is removed from the queue.

 **1** To demonstrate this, consider the following picture of a queue (with `QUEUE_CAPACITY = 5`) containing 77, 99, and 33:

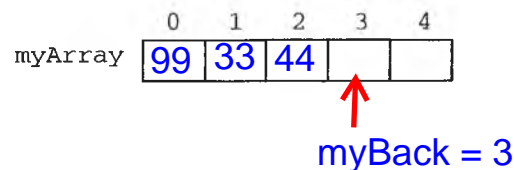


Draw diagrams like the preceding to picture the queue after each of the following operations:

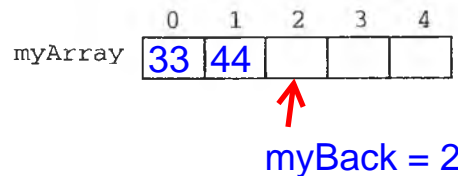
Add 44 to the queue
(fill in the array elements
and show `myBack`):



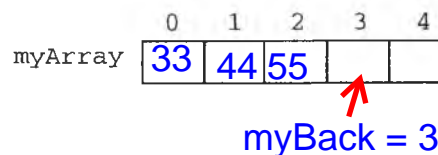
Remove an element
(fill in the array elements
and show `myBack`):



Remove another element
(fill in the array elements
and show `myBack`):



Add 55 to the queue
(fill in the array elements
and show `myBack`):




In this approach, adding elements to the queue is very efficient, since it involves only moving `myBack` to the right (incrementing the array index). Removing an element at the front of the queue, however, involves not only moving `myBack` to the left (decrementing the array index) but also moving all the elements in the array to the left. As the size of the queue grows, this can become a very expensive process.

A Second Attempt: Avoid Array Shifting: We need to avoid this shifting of array elements every time a value is removed from the queue. We are already keeping track of the back of the queue by using a data member `myBack`. So we might imitate the second (improved) array-based implementation of the `Stack` class described in Section 7.2 and use the same idea to keep track of the front of the queue; that is, use two data members to keep track of array indices:

- `myFront`: The position in the array of the element that can be removed, i.e., the *front* queue element
- `myBack`: The array position at which the next element can be added, i.e., the position following the last queue element

These are the storage structures we will use in our `Queue` class.

 **2** To see this, download the files `Queue.h`, `Queue.cpp`, and `qtester.cpp` using the procedure described by your instructor. (They can be downloaded from the website whose URL is given in the preface.) The `Queue` class library is also given in Figure 8.2 of the text.

Examine the `Queue.h` file and note the following:

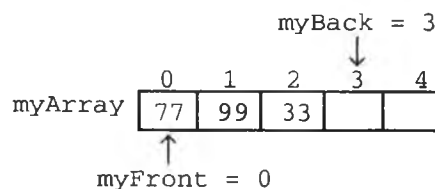
- The `const` declaration of `QUEUE_CAPACITY`. (This is eliminated in Project 6.1.)
- The `typedef` statement used to make `QueueElement` a synonym for the type of the queue's elements. (This will be replaced by a template declaration in a later project (Project 7.2).)
- The data members in the `Queue` class:
 - The array `myArray` with capacity `QUEUE_CAPACITY` and elements of type `QueueElement`. (This changes to a dynamic array in Project 6.1.)
 - The two integer variables:
 - `myFront`
 - `myBack`

Check here after you have done this X

Implementing Enqueue and Dequeue Operations: Now that we have decided on the data members for our `Queue` class, we must implement the basic operations. We look first at the enqueue and dequeue operations. A natural way to implement these is to continue with our imitation of the `Stack` class:

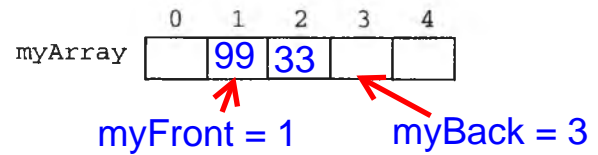
<u>Operation</u>	<u>Algorithm</u>
Enqueue:	Store the new value in <code>myArray[myBack]</code> , provided there is room in the array. Increment <code>myBack</code> by 1.
Dequeue:	Increment <code>myFront</code> by 1, provided the queue is not empty.

However, this implementation has problems. To demonstrate what they are, we will begin with the following queue, obtained by adding three elements 77, 99, and 33 to an empty queue:

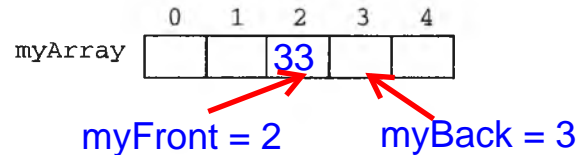


3 Draw diagrams like the preceding to picture the queue after each of the following operations:

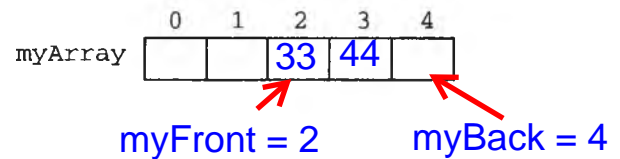
Remove an element
(fill in the array elements and
show myFront and myBack):



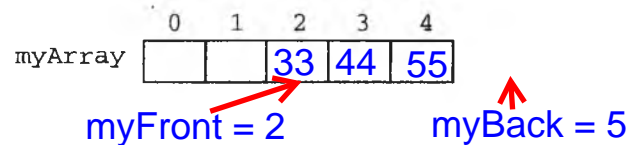
Remove another element
(fill in the array elements and
show myFront and myBack):



Add 44 to the queue
(fill in the array elements and
show myFront and myBack):



Add 55 to the queue
(fill in the array elements and
show myFront and myBack):

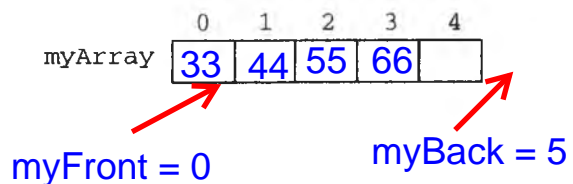


Notice that there are still open slots in the array, so we should be able to add some new elements to the queue. However, we can't use the enqueue algorithm. Why not?

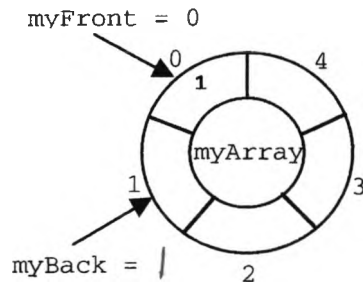
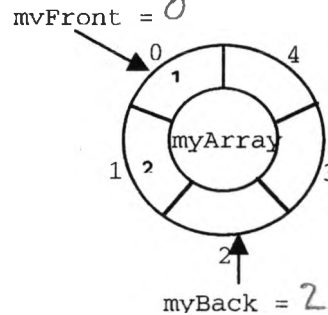
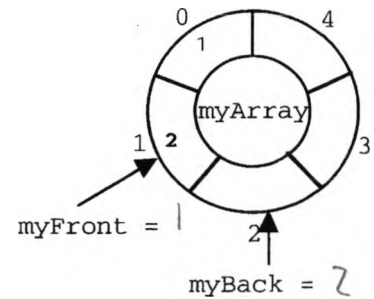
myBack has been incremented to be out of bounds

If we want to add another value to the queue, the elements in the array must be shifted back to the beginning of the array (thus setting myFront back to 0). Show the queue after this is done and the value 66 is added to the queue.

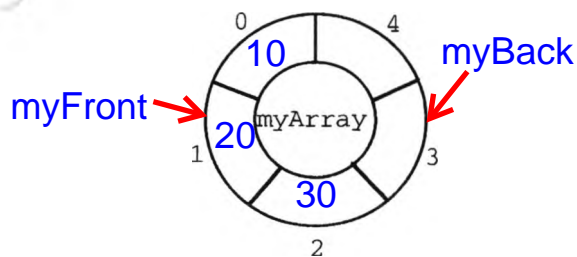
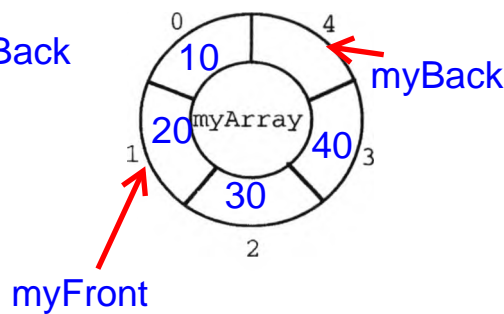
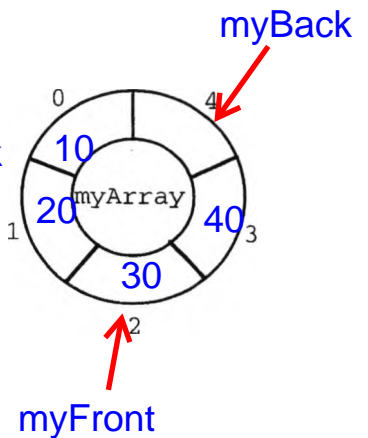
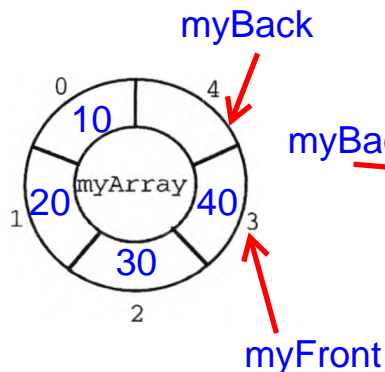
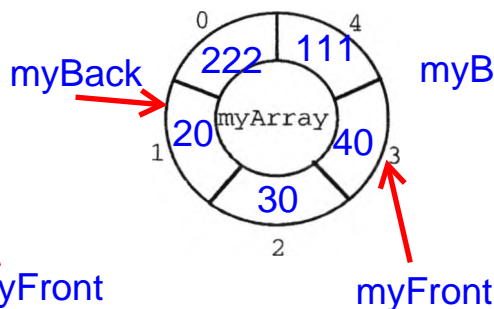
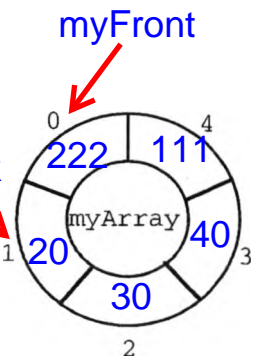
Add 66 to the queue
(fill in the array elements and
show myFront and myBack):



Using a circular array: This shifting of array elements can be avoided if we think of the array as *circular*, with the first element following the last. This can be done by incrementing `myFront` and `myBack` using addition module of the capacity of the array—that is, $(x + 1) \% \text{QUEUE_CAPACITY}$. For example, suppose we start with the following three operations, where the array capacity is 5:

Enqueue 10Enqueue 20Dequeue

4 To show that you understand this approach, continue these operations with the following, starting with the last diagram above. Fill in the array elements, add arrows for `myBack` and `myFront`, and give their values (as in the preceding diagrams).

Enqueue 30Enqueue 40DequeueDequeueEnqueue 111 and 222Dequeue two elements

Now that you've seen how the idea of a circular representation of an array works, study the following algorithms to make sure you understand how they work and that they produce the queues in the preceding sequence of operations and diagrams:

OperationAlgorithm


Enqueue: Store the new value in `myArray[myBack]` provided there is room in the array

Replace `myBack` by $(\text{myBack} + 1) \% \text{QUEUE_CAPACITY}$
 (for example, for array capacity 5 and `myBack == 4` we would get
 $(4+1) \% 5 = 0$)

Dequeue: Replace `myFront` by $(\text{myFront} + 1) \% \text{QUEUE_CAPACITY}$ provided the queue is not empty.

The preceding algorithm for enqueue contains the condition "provided there is room in the array" and the algorithm for dequeue has the condition "provided the queue is not empty." So we need to examine how to check these conditions. Checking whether a queue is empty is one of the basic queue operations, so we will look at it first.

Determining whether a queue is empty:

 **5.1** Look at the queue in the last diagram you filled in on the preceding page. It should have one element remaining. Record below the values of `myFront` and `myBack`:

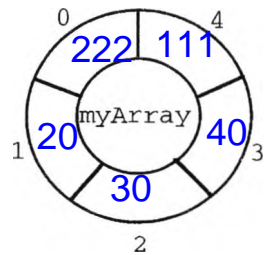
`myFront` = 0 `myBack` = 1
 containing: 222


Using the diagram at the right, give a picture of the queue after this element is removed.

Record below the new values of `myFront` and `myBack`.

`myFront` = 1 `myBack` = 1

From this example, you should see that *to determine whether a queue is empty*, we can just check to see if `myFront == myBack`.

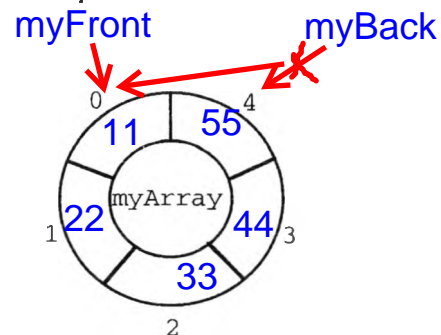
**Determining whether a queue is full:**

 **5.2** For this, we will construct a full queue step by step. Starting with an empty queue, add four values, 11, 22, 33, 44, to the empty queue. Using the diagram at the right, give a picture of the queue and record the values of `myFront` and `myBack` below:

`myFront` = 0 `myBack` = 4

Now add 55 to the queue and modify your picture at the right and record the new values of `myFront` and `myBack` below:

`myFront` = 0 `myBack` = 0



Oops! The condition *queue is full* produces the same result as *queue is empty*: `myFront == myBack`.


Consequently, the condition that `myFront` has the same value as `myBack` is ambiguous. It could mean either that the queue is full or that the queue is empty!

A Solution of the Empty/Full Conundrum

There are a variety of ways to solve this problem. One solution is to keep a count of the elements in the queue. We could add a data member `mySize` to the class and use it to keep track of the number of elements in the queue. Then the queue is empty if `mySize == 0`, and the queue is full if `mySize == QUEUE_CAPACITY`. This solution is quite simple and direct, but it does require incrementing the count with every enqueue operation and decrementing it with every dequeue operation.

Another solution, and one that we'll use here, is to keep a gap between the front and the back of the queue so that the condition `myFront == myBack` can never happen for nonempty queues. Then the algorithms for our basic operations become:

<u>Operation</u>	<u>Algorithm</u>
Constructor	Set <code>myBack</code> and <code>myFront</code> to 0 (or any other value in the range 0 through <code>QUEUE_CAPACITY - 1</code>). Thus the array begins with the empty condition true, i.e., <code>myBack == myFront</code> .
Empty	Check if <code>myBack == myFront</code> .
Enqueue	Compute <code>newBack = (myBack + 1) % QUEUE_CAPACITY</code> . If <code>newBack != myFront</code> // queue is not full Set <code>myArray[myBack] = value to be added</code> and then set <code>myBack = newBack</code> . Else Signal a queue-full condition and take appropriate action.
Front	If the queue is not empty Return <code>myArray[myFront]</code> . Else Signal that the queue is empty and return a garbage value.
Dequeue	If the queue is not empty Set <code>myFront = (myFront + 1) % QUEUE_CAPACITY</code> . Else Signal that the queue is empty.

-  [6] Now examine the `Queue.cpp` file and look at each of the function definitions for the preceding operations and see how they implement the corresponding algorithms.

Then compile `Queue.cpp` and `qtester.cpp`, link them, and execute the resulting binary executable.

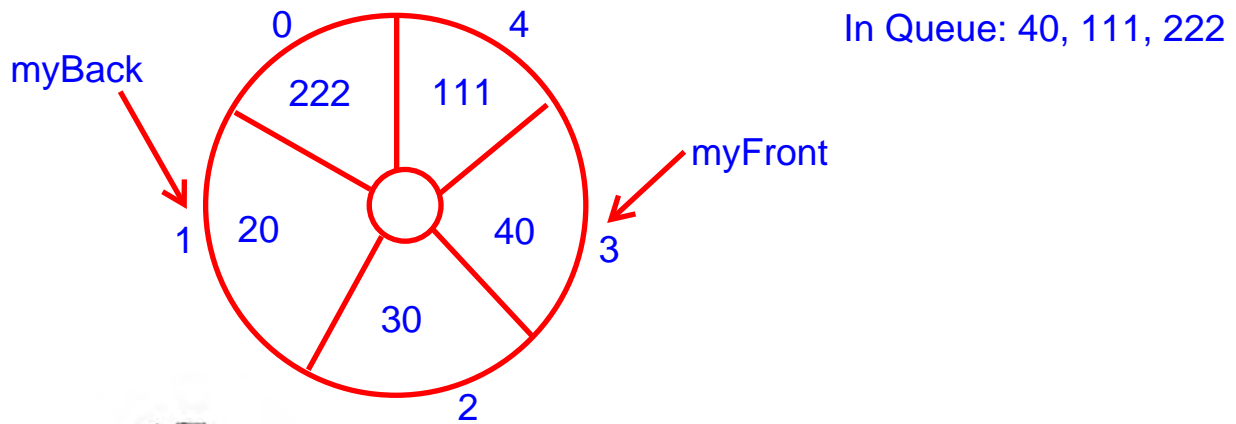
- (a) What is the front element in the full queue with which the program begins? 1
- (b) Remove an element, and then add 999. What is now the front element of the queue? 2
- (c) What happens if you attempt to add another element to the queue?

Error: Queue is full so new value is not added

- (d) Select the “dump the queue” option. What value do you get now for the front of the queue? Garbage
- (e) What happens if you now try to remove an element from the queue?

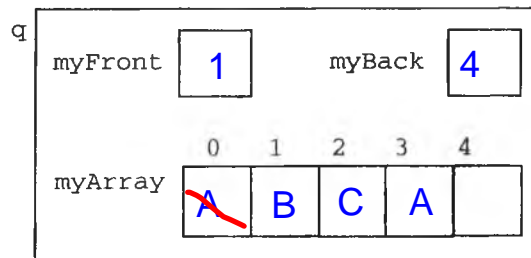
You cannot remove a value that doesn't exist

- 7 In the definition of `display()` in `Queue.cpp`, look at the for loop and note how it “increments” the control variable `i` using modular addition. Also, give an example below (a diagram is okay) of a queue for which `display()` would fail if the termination condition in the for loop was `i < myBack` instead of `i != myBack`. (Hint: One of the queues in this lab exercise is an example.)

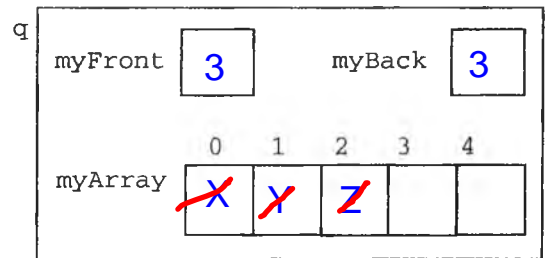


- 8 To show that you understand the implementation used in the `Queue` class, you are to do the following exercises from the text book in Exercises 8.2. Give your answers below.

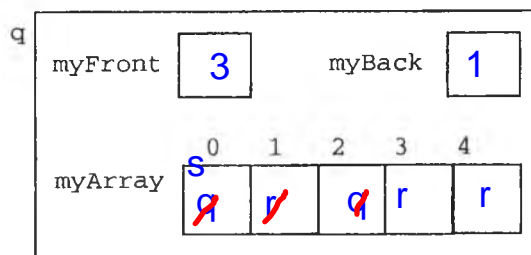
#1)



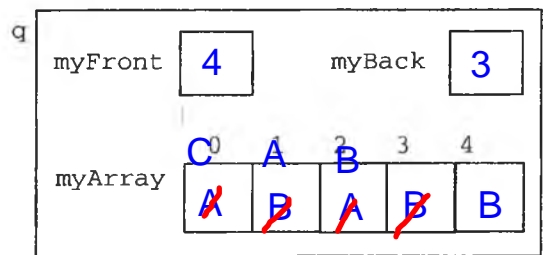
#2)



#3)



#4)



You have finished! Hand in this lab exercise with the answers filled in.

Project 6.1 An Improved Queue Class

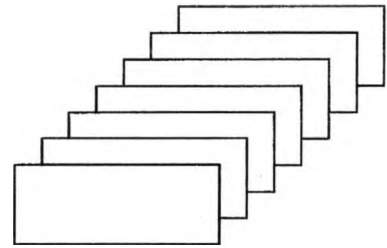
You are to modify the Queue class used in Lab 6.1 so that:

- It uses a dynamic array to store the queue elements as described in Section 8.2 (pp. 408–9) of the text *ADTs, Data Structures, and Problem Solving with C++*, 2E and in Exercise 5 of Section 8.2. This will involve some minor changes in some of the function members and also adding:
 - a destructor
 - a copy constructor, and
 - an assignment operator

to the class.

Studying the corresponding modification of the Stack class in Section 7.2 should show you what you need to do for the Queue class. You should be able to use many of the ideas and techniques used for stacks.

- It contains a `size()` function member that returns the number of elements in the queue (see Exercise 7 of Section 8.2 of the text).
- You are also expected to develop a driver program to test the new and modified function members thoroughly. You might use the menu-driven program `qtester.cpp` from Lab Exercise 6.1 and add more menu options. See the driver program for the dynamic-array-based version of the Stack class in Figure 7.10 of the text for ideas of what to include.



Hand in:

Printouts showing:

1. Listings of your Queue class header and implementation files and the driver program.
2. A demonstration that the class library and driver program compile and link okay.
3. One or more executions of the driver program that demonstrate that all the queue operations are correct.

Attach the grade sheet on the next page to your printouts.

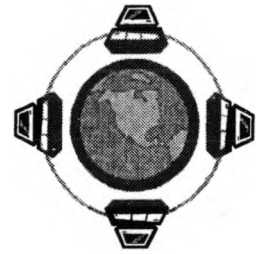
Course Info: _____ **Name:** _____

Project 6.1 Grade Sheet

<u>Category</u>	<u>Points Possible</u>	<u>Points Received</u>
Correctness of:		
Constructor	15	_____
Empty, enqueue, dequeue, front, and display operations	25	_____
Destructor	15	_____
Copy constructor	15	_____
Assignment	15	_____
Size function member	15	_____
Documentation	15	_____
Style (white space, alignment, etc.)	5	_____
Driver adequately tests the class	30	_____
Total.....	150	_____

Project 6.2 Queues—A Circular Linked Implementation

You are to modify the `Queue` class used in Figure 8.3 of the text *ADTs, Data Structures, and Problem Solving with C++, 2E*, so that it uses a circular linked list to store the queue elements as described in Section 8.3 and in Exercise 8 of Section 8.3. You should be able to use the class declaration in Figure 8.3, but change the private section so there is only one data member:



- A pointer `myBack` to the last node in the linked list.

You will need to modify the definitions of the function members in the implementation file shown in Figure 8.3 so they work correctly for a circular linked list with access via a pointer to the last node.

You are also expected to develop a driver program to test the new and modified function members thoroughly. You might use the menu-driven program `qtester.cpp` from Lab Exercise 6.1 and add more menu options. See the driver program for the linked version of the `Stack` class in Figure 7.10 of the text for ideas of what to include.

Hand in:

Printouts showing:

1. Listings of your `Queue` class header and implementation files and the driver program.
2. A demonstration that the class library and driver program compile and link okay.
3. One or more executions of the driver program that demonstrate that all the queue operations are correct.

Attach the grade sheet on the next page to your printouts.

Course Info: _____ **Name:** _____

Project 6.2 Grade Sheet

<u>Category</u>	<u>Points Possible</u>	<u>Points Received</u>
Correctness of:		
Constructor	5	_____
Empty operation	5	_____
Enqueue operation	10	_____
Dequeue operation	10	_____
Front operation	10	_____
Display operation	15	_____
Destructor	15	_____
Copy constructor	15	_____
Assignment	15	_____
Documentation	15	_____
Style (white space, alignment, etc.)	5	_____
Driver adequately tests the class	30	_____
Total	150	_____