**Queue**

According to Webster, a **queue** is a "waiting line," and as such, they abound in everyday life.

In our daily life, we have experienced standing in queues for various reasons such as purchasing tickets or getting admission to educational institutes. In all such places, we have to wait in a queue for our turn to get the service.

Queues are one of the most common data processing structures. They are frequently used in most system software such as operating systems, network and database implementations, and other areas. Queues are very useful in time-sharing and distributed computer systems where many widely distributed users share the system simultaneously. Whenever a user places a request, the operating system adds the request at the end of the queue of jobs waiting to be executed. The CPU executes the job at the front of the queue.

Similarly a queue is a common example of a linear list or an ordered list where data can be inserted at and deleted from different ends. The end at which data is inserted is called the *rear* and that from which it is deleted is called the *front*. These limits guarantee that the data is processed in the sequence in which they are entered. In short, a queue is a *first in first out* (FIFO) or *last in last out* (LILO) structure.

In this topic we consider queues, which are similar to stacks and have at least as many applications as stacks. Queues can be implemented using arrays as the basic storage structures, but as we will discover, a bit more effort is required to construct an efficient array-based implementation of a queue than for a stack. We will also look at linked-list implementations of queues.

**Introduction to Queues**

As an abstract data type, a queue is a special kind of list in which the basic insert and delete operations are restricted to the ends of the list. Unlike stacks in which elements are popped and pushed only at one end of the list, items are removed from a queue at one end, called the **front** (or **head)** of the queue, and elements are added only at the other end called the **back** (or **rear or tail).**

**Queue ADT**

**Collection of Data Elements**

A sequence of data items with the property that items can be removed only at one end, called the *front* of the queue, and items can be added only at the other end, called the *back* of the queue.

**Basic Operations**

• Construct a queue (usually empty)

• Check if queue is empty

• enqueue: Add an element at the back of the queue

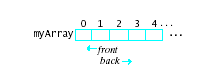
• front: Retrieve the element at the front of the queue

• dequeue: Remove the element at the front of the queue

Since a queue is a linear data structure, it can be implemented using either arrays or linked lists. For the former, we use static memory allocation and for the latter, we use dynamic memory allocation. Let us see how a queue can be implemented using arrays.

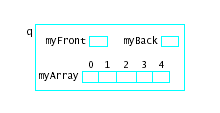
Let us see the implementation of the various operations on the queue using arrays. ***Create*** This operation should create an empty queue. Here max is the maximum initial size that is defined.

myArray to store the elements of the queue:



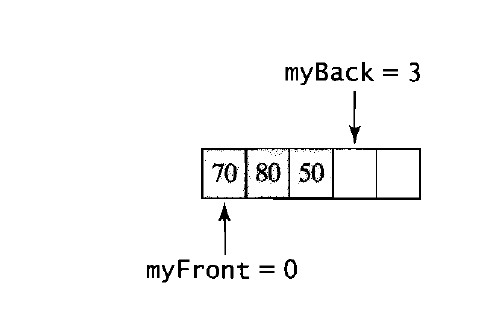
two integer variables:

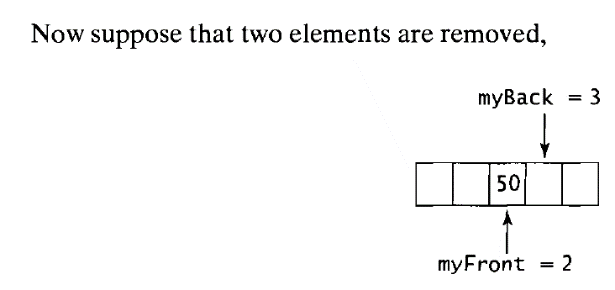
Front The position in the array of the element that can be removed the position of the front queue element. Back The position in the array at which an element can be added the position *following* the last queue element Thus, we might picture a Queue object q as follows:

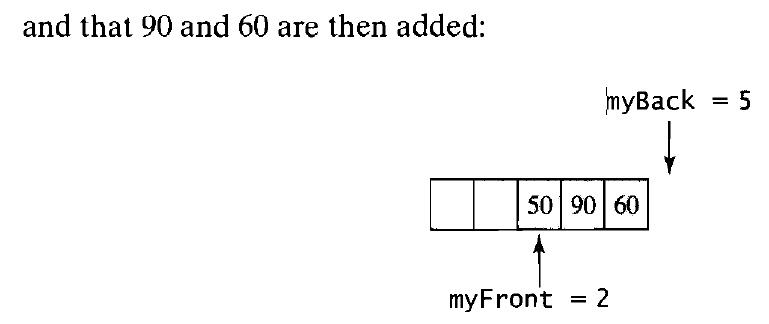


An item is added to the queue by storing it at position myBack of the array, provided that myBack does not exceed some maximum size specified for the array, and then incrementing myBack by 1.

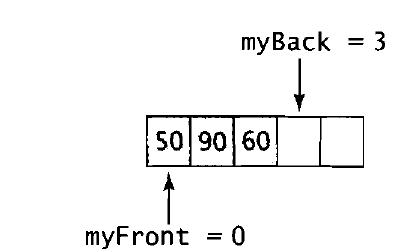
The difficulty with this implementation is that elements "walk off the end" of the array, so that eventually all the array elements may have to be shifted back to the beginning positions. For example, consider a queue in which the capacity of the array data member is 5 and whose elements are integers. The sequence of operations add 70, add 80, add 50 produces the following configuration:







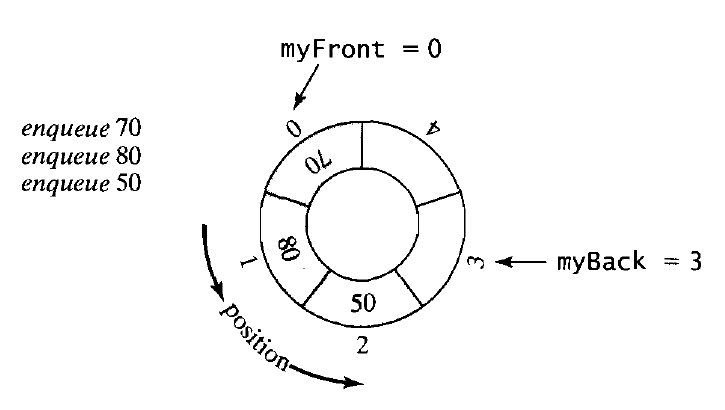
Before another item can be inserted into the queue, the elements in the array must be shifted back to the beginning of the array:

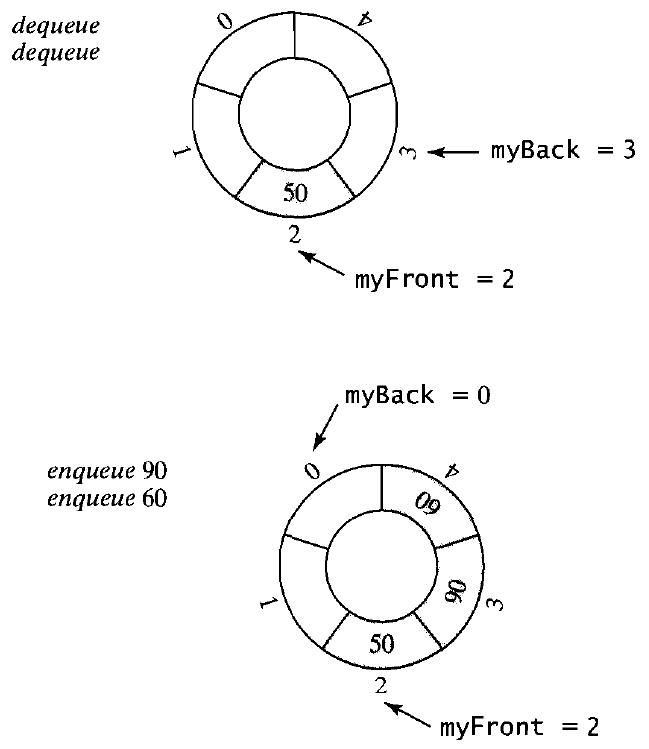


This shifting of array elements is very inefficient, especially when the elements are large objects.

A better alternative is to simply let myBack start over at the beginning of the array when it goes off the end. What we are doing, therefore, is thinking of the array as *circular,* with the first element following the last. This can be done by incrementing myFront and myBack using addition modulo the array's capacity. For the sequence of operations just considered, this implementation yields the following configurations:

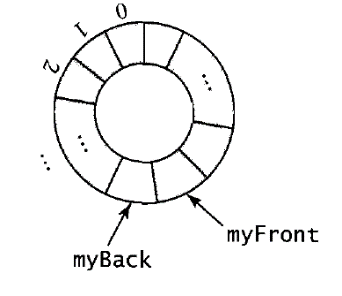
myFront





queue-full condition.

To see how this condition can be detected, suppose that the array is almost full, with only one empty location remaining:



If an item were stored in this location, myBack would be incremented by 1, giving it the same value as myFront. However, the condition myFront == myBack indicates that the queue is empty. Thus, we *cannot distinguish between an empty queue and a*  *full queue* if this location is used to store an element. But we can avoid this difficulty if we maintain one empty position in the array so that myFront will never equal myBack except when the queue is empty.

There are several alternatives to keeping an empty slot in the array. One commonly used is to add an integer data member count to the class, which stores the number of elements currently in the queue. Another is to use a boolean data member full instead of the integer member count, which is set to true when the queue

becomes full and is false otherwise. Queue classes developed using these approaches

are left as exercises.

**Using a Static Array to Store the Queue Elements**

Figure 8.2 shows a Queue class that uses a circular static array myArray with capacity QUEUE\_CAPACITY to store the queue elements. An empty slot is maintained between the front and back elements so that queue-full and queue-empty conditions can be distinguished. The data members myFront and myBack record the position of the front element and the position following the last element, respectively.

The constructor simply initializes both of the members myFront and myBack to 0 (or to any other value in 0, 1, ... ,QUEUE\_CAPACITY - 1); and a queue will be empty when the boolean expression myFront == myBack is true.

An algorithm for the *enqueue* operation is as follows:

#ifndef QUEUE

#define QUEUE

#include <iostream>

#include<ostream>

using namespace std;

const int QUEUE\_CAPACITY = 128;

class Queue {

public:

Queue();

bool isEmpty() const;

void enqueue(const int & value);

void display(ostream & out) const;

int front() const;

int dequeue();

private:

/\*\*\*\*\* Data Members \*\*\*\*\*/

int myArray[QUEUE\_CAPACITY];

int Front,

Back;

}; // end of class declaration

#endif

#include <iostream>

#include<ostream>

#include "Queue.h"

using namespace std;

Queue::Queue() : Front(0), Back(0) {}

bool Queue::isEmpty() const {

return (Front == Back);

}

void Queue::enqueue(const int & value) {

int newBack = (Back + 1) % QUEUE\_CAPACITY;

if (newBack != Front){ // queue isn't full

myArray[newBack] = value;

Back = newBack;

}

else {

cout << "\*\*\* Queue full -- can't add new value \*\*\*\n"

"increase value of QUEUE\_CAPACITY in Queue.h\n";

exit(0);

}

}

void Queue::display(ostream & out) const {

for (int i = Front; i != Back; i = (i + 1) % QUEUE\_CAPACITY)

out << myArray[i] << " ";

cout << endl;

}

//--- Definition of front()

int Queue::front() const {

if (!isEmpty())

return (myArray[Front]);

else {

cout << "\*\*\* Queue is empty \*\*\*\n";

exit(0);

}

}

//--- Definition of dequeue()

int Queue::dequeue() {

if (!isEmpty())

Front = (Front + 1) % QUEUE\_CAPACITY;

else {

cout << "\*\*\* Queue is empty -- "

"can't remove a value \*\*\*\n";

exit(0);

}

}

**Linked Queues**

As we noted for stacks in Chapter 7, array-based implementations of a container place an upper limit on the size of the container, because an array has a fixed size.

This means, in particular, that to use one of the array-based queue classes of the preceding section, one must know in advance what capacity to specify for the queues that will be needed and then either set the value of QUEUE\_CAPACITY in

Queue. H

**A Natural Linked-List Implementation**

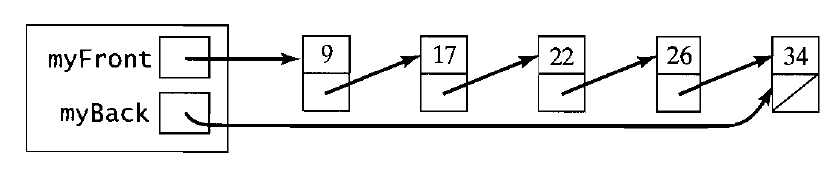
Another alternative to an array-based implementation of queues is to proceed as we did for stacks in Section 7.3 and use a linked list to store the queue elements. Because a linked list can grow by adding new nodes, a linked queue can grow as large as necessary

(or until available memory is exhausted); there is no need to set some upper bound on the capacity in advance. Also, a linked queue can shrink as elements are removed by returning deleted nodes to the memory heap.

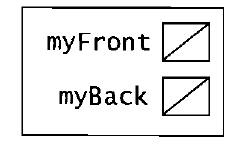
The linked lists considered up to now have provided access to only the first node in the list by maintaining a pointer to this node. If we make this first node the front of the queue, as seems natural, then the operations to retrieve the front element and to remove the front element are implemented in the same way as the top and pop operations for a linked stack. However, adding values at the back of the queue would require traversing the entire list each time to find the back of the queue. This list traversal can be avoided if we adopt the approach of the array-based implementations in the preceding section and maintain two pointers, myFront, which points to the node at the front of the queue, and myBack, which points to the node at

the back. For example, a linked queue containing the integers 9, 17,22,26, and 34 in

this order, might be pictured as



In this implementation, the constructor can simply initialize myFront and myBack to be null pointers, which represents an empty queue:



The empty operation need only check whether myFront (or myBack) is null. The copy constructor and assignment operator must traverse the linked list storing the queue elements to make a copy, as described for linked stacks.

Retrieving the element at the front of the queue is trivial,

return myFront->data;

provided that the queue is not empty. Removing the front element of a nonempty queue is simple deletion of the first node in a linked list, but care must be taken if the queue becomes empty to reset myBack to null:

ptr = myFront;

myFront = myFront->next;

delete ptr;

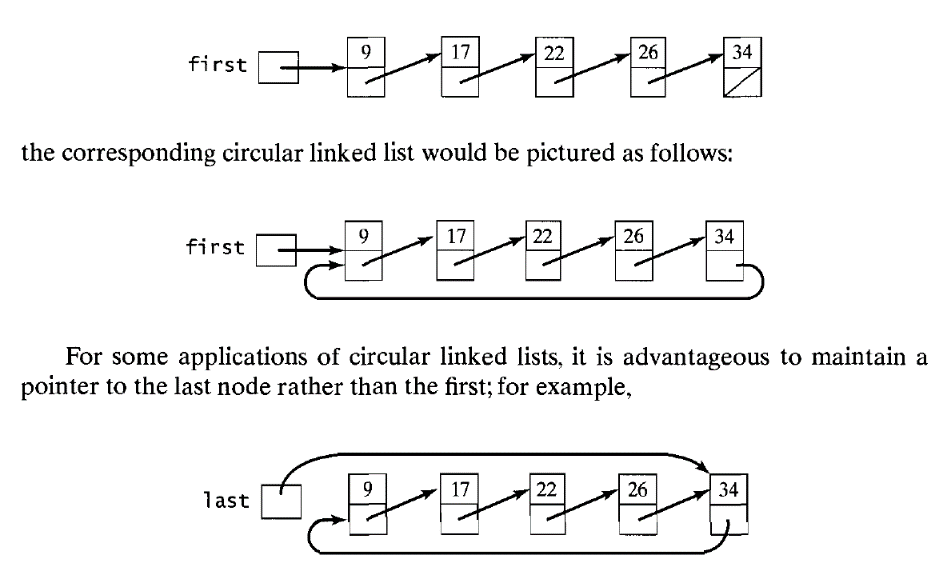
if (myFront == NULL) *II* queue is now empty

myBack = 0;

**Using a Circular linked list**

In the preceding section we described an implementation of queues in which the array that stored the elements was thought of as a circular array with the first element following the last. This suggests that it might be useful to consider an analogous **circular linked list** obtained by setting the link of the last node in a linked list of the kind we've been considering, sometimes called a **linear linked** list, to point to the first

node. For example, for the linear linked list



In this case, we have direct access to the last node and almost direct access to the first node, since 1ast->next points to the first node in the list. This variation is thus especially useful when it is necessary to access repeatedly the elements at the ends of the list. In particular, it is well suited for linked queues. Developing a queue class that uses a circular linked list in this manner is left as an exercise.