6.1 INTRODUCTION

All of us are familiar with queues. In simple terms, a queue is a line of persons waiting for their turn at some service counter/window. A service counter can be a ticketing window of a cinema hall, a ticketing window of a bus stand or railway station, a service counter of a bank, a service counter of a gas agency, a fees deposit counter in a college, etc.. Depending on the type of service provided by the service counter and the number of persons interested in this service, there can be queues of varying lengths. The service at the service counter is provided on the First-Come First-Serve (FCFS) basis i.e. in order of their arrival in the queue.

Suppose that at a service counter, t_1 units of time is needed to provide a service to a single person, and on average a new person arrives at the service counter in t_2 units of time. The following possibilities may arise:

1. If $t_1 < t_2$, then the service counter will be free for some time before a new person arrives at the service counter. Hence, no queue in this case.
2. If $t_1 > t_2$, then service counter will be busy for some time even after the arrival of a new person. Therefore, this person have to wait for some time. Similarly, other persons arriving at the service counter have to wait. Hence, a queue will be formed.
3. If $t_1 = t_2$, then as a person leaves the service counter, a new persons arrives at the service counter. Hence, no queue in this case also.

This chapter describes a queue, its representation in computer memory, and the implementation of different operations that can be performed on them. It also describes a deque and a priority queue in brief.

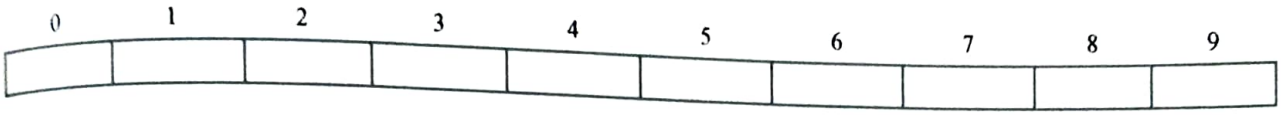
6.2 QUEUE DEFINED

In the context of data structures, a *queue* is a linear list in which insertions can take place at one end of the list, called the *rear* of the list, and deletions can take place only at other end, called the *front* of the list. The behaviour of a queue is like a Fast-In-First-Out (FIFO) system. Figure 6.1 shows a queue as black-box in which elements enters from one side and leaves from other side.

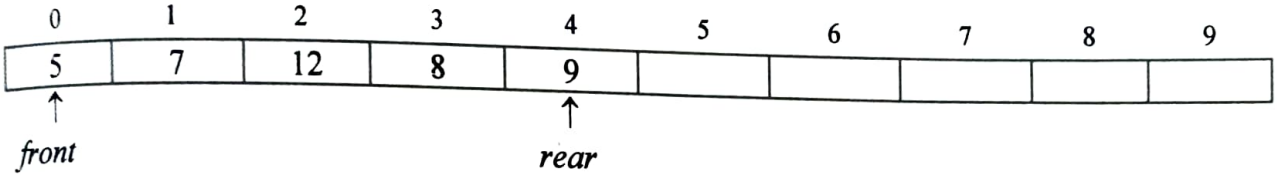
In queue terminology, the insertion and deletion operations are known as *enqueue* and *dequeue* operations.

Example 6.1

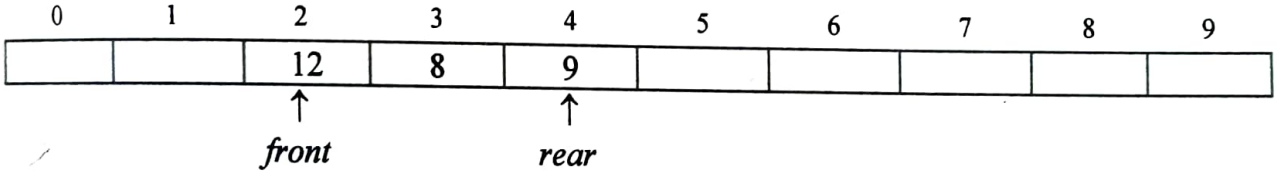
The following figure illustrates a queue which can accommodate maximum of 10 elements.



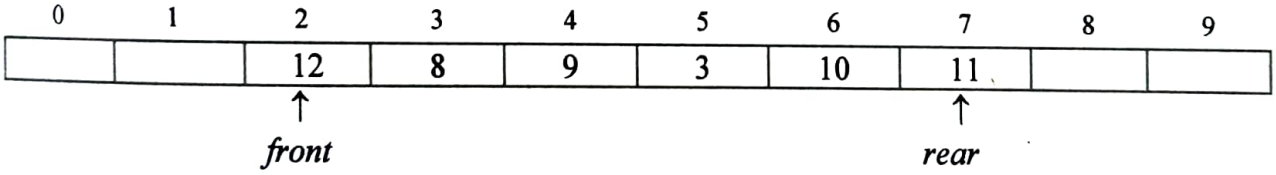
(*a*) An empty queue



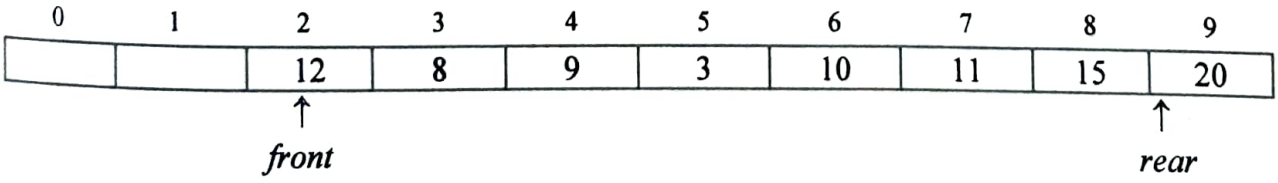
(*b*) Queue after enqueuing elements 5, 7, 12, 8, and 9 in order



(*c*) Queue after dequeuing elements 5, 7 from queue



(*d*) Queue after enqueuing elements 3, 10, 11 in a queue



(*e*) Queue after enqueuing elements 15 and 20 in a queue

Figure 6.1 Illustration of a queue whose elements are integers values

6.3 OPERATIONS ON QUEUES

The following operations are performed on queues.

1. **CreateQueue(q)** – to create q as an empty queue.
2. **Enque(q, i)** – to insert (add) element i in a queue q.
3. **Dequeue(q)** – to access and remove an element of queue q.
4. **Peek(q)** – to access the first element of the queue q without removing it from the queue q.
5. **IsFull(q)** – to check whether the queue q is full.
6. **IsEmpty(q)** – to check whether the queue q is empty.

All of these operation runs in O(1) time.

Note that if the queue q is empty then it is not possible to dequeue the queue q. Similarly, as there is no element in the queue q, the *Peek(q)* operations is also not valid. Therefore, we must ensure that the queue q is not empty before attempting to perform these operations.

Likewise, if the queue q is full then it is not possible to enqueue a new element in to the queue q. Therefore, we must ensure that the queue q is not full before attempting to perform enqueue operation.

6.4 REPRESENTATION OF QUEUES IN MEMORY

Queues, like stacks, can also be represented in memory using a linear array or a linear linked list. Even using a linear, we have two implementation of queues viz. *linear* and *circular*. The representation of these two is same, the only difference is in their behaviour. To start with, let us consider array representation of queues.

6.4.1 Representing a Queue using an Array

To implement a queue we need two variables, called *front* and *rear*, that holds the index of the first and last element of the queue and an array to hold the elements of the queue. Let us suppose that the elements of the queue are of integer type, and the queue can store maximum of 10 such elements i.e. queue size is 10.

The following are the necessary declarations

```
#define  MAX  10

typedef struct
{
   int front, rear;
   int elements[MAX];
} queue;

queue q;
```

We have defined our own data type named *queue*, which is a structure whose first element *front* will be used to hold the index of the first element of the queue. The second element *rear* will be used to hold the index of the last element of the queue. An array *elements* of size MAX of integer type to hold the elements of the queue. The last line declares variable *q* of type *queue*.

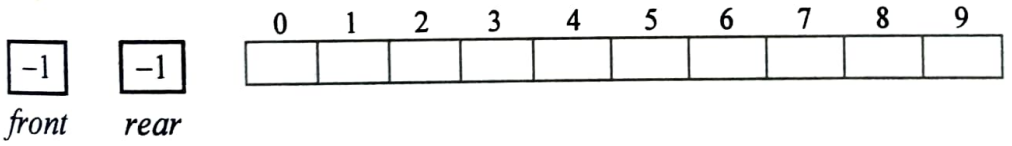
In addition to above declarations, we will use the declaration

```
typedef enum { false, true } boolean;
```

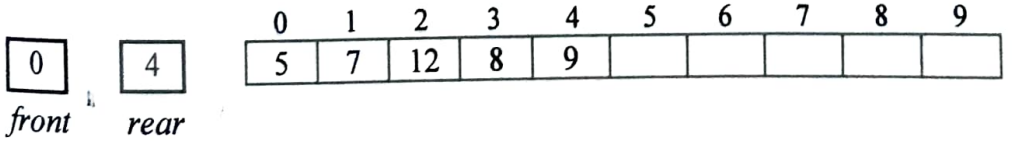
to signify the use of *boolean* kind of data and/or operations. This statement defined new data type named *boolean* which can take value *false* or *true*.

With these declarations, we will write functions for various operation to be performed on queue.

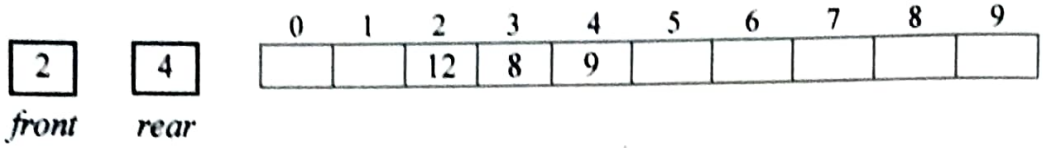
With these declarations, queue of Figure 6.1 will be represented in computer memory as shown in Figure 6.2.



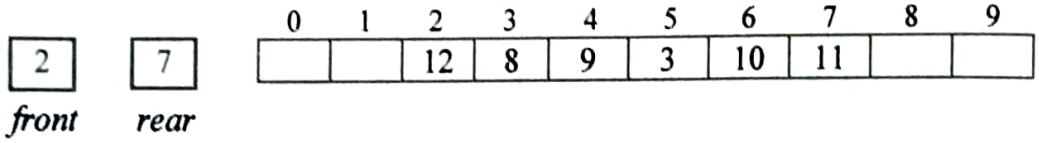
(*a*) Representation of queue of Figure 6.1(*a*) in memory



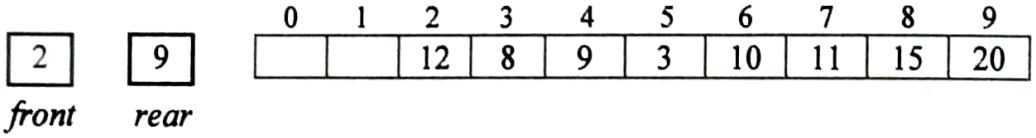
(*b*) Representation of queue of Figure 6.1(*b*) in memory



(*c*) Representation of queue of Figure 6.1(*c*) in memory



(*d*) Representation of queue of Figure 6.1(*d*) in memory



(*e*) Representation of queue of Figure 6.1(*d*) in memory

Figure 6.2 Array representation of linear queue of Figure 6.1

You must have observed, as shown in Figure 6.2(*e*), that it is not possible to enqueue more elements as such, though two positions in the linear queue are vacant. To overcome this problem, the elements of the queue are moved forward, so that the vacant positions are shifted towards the rear end of the linear queue. After shifting, *front* and *rear* are adjusted properly, and then the element is enqueued in the linear queue as usual. For example. We want to enqueue one more element, say 60, after making adjustments the queue will look as shown in Figure 6.3 .

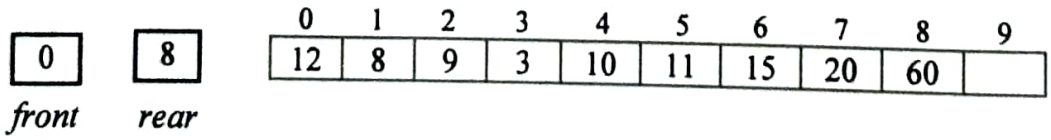
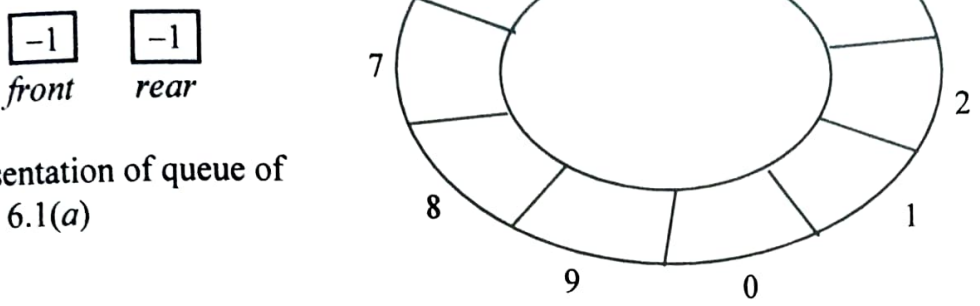
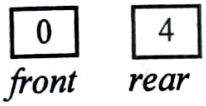


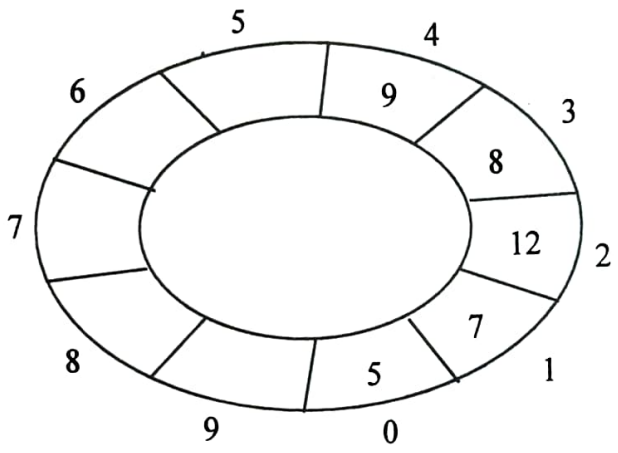
Figure 6.3 State of the queue after making adjustments and then enqueueing element 60

However, this difficulty can be overcome if we treat the queue position with index 0 as a position that comes after position with index 9 i.e. we treat the queue as circular as shown in Figure 6.4.

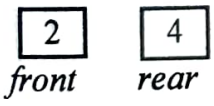


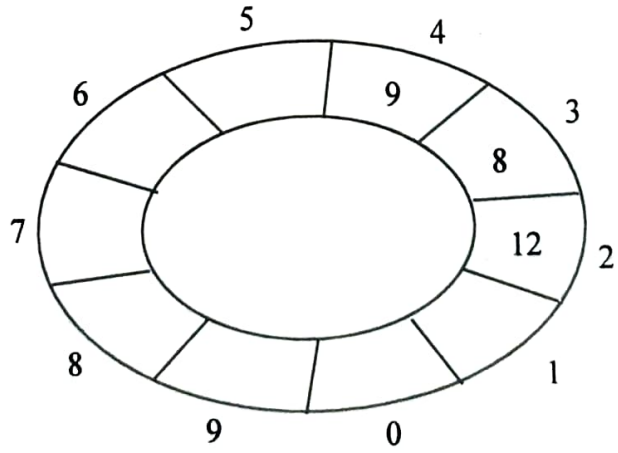
(*a*) Representation of queue of Figure 6.1(*a*)



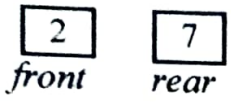


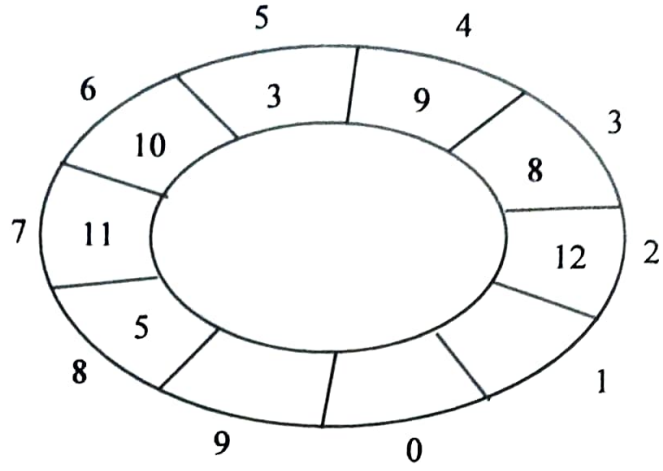
(*b*) Representation of queue of Figure 6.1(*b*)



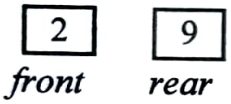


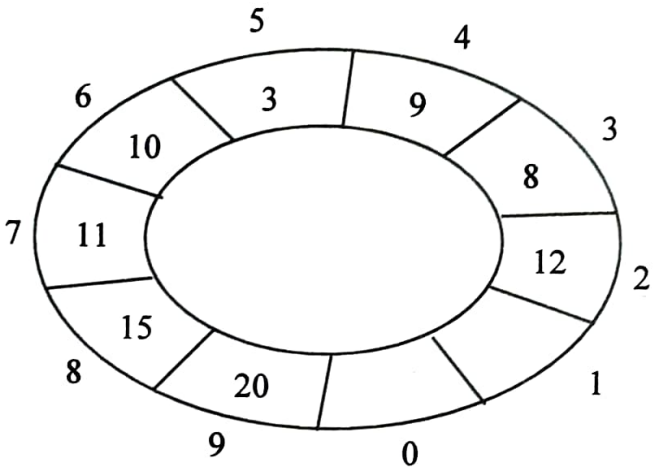
(*c*) Representation of queue of Figure 6.1(*c*)





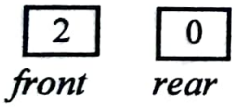
(*d*) Representation of queue of Figure 6.1(*d*)

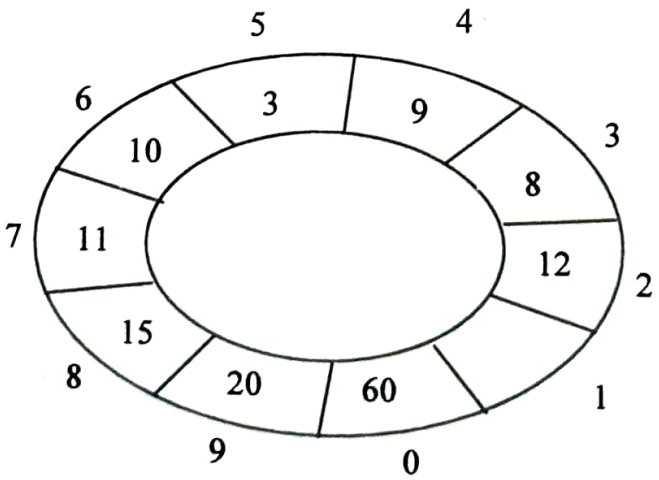




(*e*) Representation of queue of Figure 6.1(*e*)

Now to enqueue one more element, say 60, the *rear* is reset to 0, and the elements is inserted at that index as shown below.





(*f*) Representation of queue of Figure 6.3

Figure 6.4 (*a*) – (*f*) Illustration of Operations on Circular Queues

6.4.1.1 Implementation of Operations on a Linear Queue

The implementation of different operations on a linear queue are described below.

6.4.1.1.1 Creating an Empty Linear Queue

Before we can use a queue, it must be created/initialized. As the index of array elements can take any value in the range 0 to MAX–1, the purpose of initializing the queue is served by assigning value –1 (as sentinel value) to *front* and *rear* variables. This simple task can be accomplished by the following function

Listing 6.1

```
void CreateQueue( queue *v )
{
   v->front = v->rear = -1;
}
```

6.4.1.1.2 Testing a Linear Queue for Underflow

Before we remove an element from a queue, it is necessary to test whether a queue still have some elements i.e. to test that whether the queue is empty or not. If it is not empty then the dequeue operation can be performed to remove the front element. This test is performed by comparing the value of *front* with sentinel value –1 as shown in the following function

Listing 6.2

```
boolean IsEmpty( queue *v )
{
   if ( v->front == -1  )
      return true;
   else
      return false;
}
```

The above function returns value *true* if the test condition is satisfied; otherwise it returns value *false*.

6.4.1.1.3 Testing a Linear Queue for Overflow

Before we insert new element in a queue, it is necessary to test whether queue still have some space to accommodate the incoming element i.e. to test that whether the queue is full

or not. If it is not full then the ***enqueue*** operation can be performed to insert the element at rear end of the queue. This test is performed by test condition

```
( front = 0 ) and ( rear = MAX - 1 )
```

i.e. front element is at position with index 0 and the last element is at position with index MAX–1, which indicates that no position in linear queue is vacant.

This task is accomplished by the following function

Listing 6.3

```
boolean IsFull( queue *v )
{
    if (  ( v->front == 0 ) && ( v->rear == (MAX - 1) ) )
        return true;
    else
        return false;
}
```

The above function returns value *true* if the test condition is satisfied; otherwise it returns value *false*.

6.4.1.1.4 Enqueue Operation on Linear Queue

Even if a linear queue is not full, there is another problem that may arise during enqueue operation. That problem is that, the value of the *rear* variable is equal to MAX–1, though the value of the *front* variable is not equal to 0. This possibility implies that the incoming element cannot be enqueued unless the elements are moved forward, and variables *front* and *rear* are adjusted accordingly. Which is, of course, a time consuming process.

There are two conditions, which can occur, even if the queue is not full. These are

- If a linear queue is empty, then the value of the *front* and *rear* variables will be NIL (sentinel value), then both *front* and *rear* are set to 0.
- If a linear queue is not empty, then there are further two possibilities:
  - ◊ If the value of the *rear* variable is less than MAX–1, then the *rear* variable is incremented.
  - ◊ If the value of the *rear* variable is equal to MAX–1, then the elements of the linear queue are moved forward, and the *front* and *rear* variables are adjusted accordingly.

This task is accomplished as shown in the following function

Listing 6.4

```
void Enqueue( queue *v, int value )
{
   if ( IsEmpty(v) )
      v->front = v->rear = 0;
   else if ( v->rear == ( MAX - 1) )
   {
      for ( i = v->front, i <= v->rear; i++ )
           v->elements[i - v->front] = v->elements[i];
      v->rear = v->rear - v->front + 1;
      v->front = 0;
   }
   else
      v->rear++;
   v->elements[v->rear] = value;
}
```

6.4.1.1.5 Dequeue Operation on a Linear Queue

The front element of a linear queue is assigned to a local variable, which later on will be returned via the *return* statement. After assigning the front element of a linear queue to a local variable, the value of the *front* variable is modified so that it points to the new front.

There are two possibilities:

- If there was only one element in a linear queue, then after dequeue operation queue will become empty. This state of a linear queue is reflected by setting *front* and *rear* variables to sentinel value NIL.
- Otherwise the value of the *front* variable is incremented.

This task is accomplished as shown in the following function

Listing 6.5

```
int Dequeue( queue *v )
{
   int temp;
   temp = v->elements[v->front];
   if ( v->front == v->rear )
      v->front = v->rear = -1;
   else
```

```
        v->front++;
    return temp;
}
```

6.4.1.1.6 Accessing Front Element

The element in the front of queue is accessed as shown in the following function

Listing 6.6

```
int Peek( queue *v )
{
    return (v->elements[v->front]);
}
```

6.4.1.2 Limitations of a Linear Queue

The only limitation of linear queue is that —

If the last position of the queue is occupied, it is not possible to enqueue any more elements even though some positions are vacant towards the front positions of the queue.

However, this limitation can be overcome by moving the elements forward, such that the first element of the queue goes to position with index 0, and the rest of the elements move accordingly. And finally the *front* and *rear* variables are adjusted appropriately.

But this operation may be very time consuming if the length of the linear queue is very long. As mentioned earlier, this limitation can be overcome if we treat that the queue position with index 0 comes immediately after the last queue position with index MAX–1. The resulting queue is known as *circular queue*.

6.4.1.3 Implementation of Operations on a Circular Queue

The operations of initialization, testing for underflow for circular queue is similar to that of a linear queue. However, other operations are slightly different. These are described below.

6.4.1.3.1 Testing a Circular Queue for Overflow

Before we insert new element in a circular queue, it is necessary to test whether a circular queue still have some space to accommodate the incoming element i.e. to test that whether a

circular queue is full or not. If it is not full then the enqueue operation can be performed to insert the element at rear of a circular queue. This test is performed by test conditions

- (front = 0) and (rear = MAX-1)
- front = rear + 1

If any one of these two conditions is satisfied, it means circular queue is full.

This task is accomplished by the following function

Listing 6.7

```
boolean Isfull( queue *v )
{
   if  (((v->front==0)&&(v->rear==(MAX-1)))||(v->front==v->rear+1))
      return true;
  else
      return false;
}
```

The above function returns value *true* if the test condition is satisfied; otherwise it returns value *false*.

6.4.1.3.2 Enqueue Operation on a Circular Queue

Before the enqueue operation, there are three conditions which can occur, even if a circular queue is not full. These are

- If the queue is empty, then the value of the *front* and *rear* will be –1 (sentinel value), then both *front* and *rear* are set to 0.
- If the queue is not empty then the value of the *rear* will be the index of the last element of the queue, then the *rear* variable is incremented.
- If the queue is not full and the value of *rear* variable is equal to MAX–1, then *rear* variable is set to 0.

This task is accomplished as shown in the following function

Listing 6.8

```
void Enqueue( queue *v, int value )
{
    if ( v->front == NIL )
        v->front = v->rear = 0;
```

```
    else if ( v->rear == (MAX-1) )
        v->rear = 0;
    else
        v->rear++;
    v->elements[v->rear] = value;
}
```

6.4.1.2.3 Dequeue Operation on a Circular Queue

The front element of a circular queue is assigned to a local variable, which later on will be returned via the *return* statement. After assigning the front element of a circular queue, the value of the *front* variable is modified so that it points to the new front.

There are two possibilities:

- If there was only one element in a circular queue, then after dequeue operation the circular queue will become empty. This state of a circular queue is reflected by setting *front* and *rear* variables to sentinel value NIL.
- If value of the front variable is equal to MAX–1, then set *front* variable to 0.

If none of the above conditions hold, then the *front* variable is incremented.

This task is accomplished as shown in the following function.

Listing 6.9

```
int Dequeue( queue *v )
{
    int temp;
    temp = v->elements[v->front];
    if ( v->front == v->rear )
        v->front = v->rear = -1;
    else if ( v->front == (MAX-1))
        v->front = 0;
    else
        v->front++;
    return temp;
}
```