

## CHAPTER 8

# Queues

**LEARNING OBJECTIVE**

A queue is an important data structure which is extensively used in computer applications. In this chapter we will study the operations that can be performed on a queue. The chapter will also discuss the implementation of a queue by using both arrays as well as linked lists. The chapter will illustrate different types of queues like multiple queues, double ended queues, circular queues, and priority queues. The chapter also lists some real-world applications of queues.

**8.1 INTRODUCTION**

Let us explain the concept of queues using the analogies given below.

- People moving on an escalator. The people who got on the escalator first will be the first one to step out of it.
- People waiting for a bus. The first person standing in the line will be the first one to get into the bus.
- People standing outside the ticketing window of a cinema hall. The first person in the line will get the ticket first and thus will be the first one to move out of it.
- Luggage kept on conveyor belts. The bag which was placed first will be the first to come out at the other end.
- Cars lined at a toll bridge. The first car to reach the bridge will be the first to leave.

In all these examples, we see that the element at the first position is served first. Same is the case with queue data structure. A queue is a **FIFO** (First-In, First-Out) data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the **REAR** and removed from the other end called the **FRONT**.

Queues can be implemented by using either arrays or linked lists. In this section, we will see how queues are implemented using each of these data structures.

## 8.2 ARRAY REPRESENTATION OF QUEUES

Queues can be easily represented using linear arrays. As stated earlier, every queue has **FRONT** and **REAR** variables that point to the position from where deletions and insertions can be done, respectively. The array representation of a queue is shown in Fig. 8.1.

12	9	7	18	14	36				
0	1	2	3	4	5	6	7	8	9

**Figure 8.1** Queue

12	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

**Figure 8.2** Queue after insertion of a new element

	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

**Figure 8.3** Queue after deletion of an element

of the queue. The queue after deletion will be as shown in Fig. 8.3.

Here, **FRONT** = 1 and **REAR** = 6.

```

Step 1: IF REAR = MAX-1
        Write OVERFLOW
        Goto step 4
    [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
        SET FRONT = REAR = 0
    ELSE
        SET REAR = REAR + 1
    [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT

```

**Figure 8.4** Algorithm to insert an element in a queue

```

Step 1: IF FRONT = -1 OR FRONT > REAR
        Write UNDERFLOW
    ELSE
        SET VAL = QUEUE[FRONT]
        SET FRONT = FRONT + 1
    [END OF IF]
Step 2: EXIT

```

**Figure 8.5** Algorithm to delete an element from a queue

### Operations on Queues

In Fig. 8.1, **FRONT** = 0 and **REAR** = 5. Suppose we want to add another element with value 45, then **REAR** would be incremented by 1 and the value would be stored at the position pointed by **REAR**. The queue after addition would be as shown in Fig. 8.2. Here, **FRONT** = 0 and **REAR** = 6. Every time a new element has to be added, we repeat the same procedure.

If we want to delete an element from the queue, then the value of **FRONT** will be incremented. Deletions are done from only this end

However, before inserting an element in a queue, we must check for overflow conditions. An overflow will occur when we try to insert an element into a queue that is already full. When **REAR** = **MAX** - 1, where **MAX** is the size of the queue, we have an overflow condition. Note that we have written **MAX** - 1 because the index starts from 0.

Similarly, before deleting an element from a queue, we must check for underflow conditions. An underflow condition occurs when we try to delete an element from a queue that is already empty. If **FRONT** = -1 and **REAR** = -1, it means there is no element in the queue. Let us now look at Figs 8.4 and 8.5 which show the algorithms to insert and delete an element from a queue.

Figure 8.4 shows the algorithm to insert an element in a queue. In Step 1, we first check for the overflow condition. In Step 2, we check if the queue is empty. In case the queue is empty, then both **FRONT** and **REAR** are set to zero, so that the new value can be stored at the 0<sup>th</sup> location. Otherwise, if the queue already has some values, then **REAR** is incremented so that it points to the next location in the array. In Step 3, the value is stored in the queue at the location pointed by **REAR**.

Figure 8.5 shows the algorithm to delete an element from a queue. In Step 1, we check for underflow condition. An underflow occurs if **FRONT** = -1 or **FRONT** > **REAR**. However, if queue has some values, then **FRONT** is incremented so that it now points to the next value in the queue.

### PROGRAMMING EXAMPLE

1. Write a program to implement a linear queue.

```

#include <stdio.h>
#include <conio.h>

```

```

#define MAX 10 // Changing this value will change length of array
int queue[MAX];
int front = -1, rear = -1;
void insert(void);
int delete_element(void);
int peek(void);
void display(void);
int main()
{
    int option, val;
    do
    {
        printf("\n\n ***** MAIN MENU *****");
        printf("\n 1. Insert an element");
        printf("\n 2. Delete an element");
        printf("\n 3. Peek");
        printf("\n 4. Display the queue");
        printf("\n 5. EXIT");
        printf("\n Enter your option : ");
        scanf("%d", &option);
        switch(option)
        {
            case 1:
                insert();
                break;
            case 2:
                val = delete_element();
                if (val != -1)
                    printf("\n The number deleted is : %d", val);
                break;
            case 3:
                val = peek();
                if (val != -1)
                    printf("\n The first value in queue is : %d", val);
                break;
            case 4:
                display();
                break;
        }
    }while(option != 5);
    getch();
    return 0;
}

void insert()
{
    int num;
    printf("\n Enter the number to be inserted in the queue : ");
    scanf("%d", &num);
    if(rear == MAX-1)
        printf("\n OVERFLOW");
    else if(front == -1 && rear == -1)
        front = rear = 0;
    else
        rear++;
    queue[rear] = num;
}

int delete_element()
{
    int val;

```

```

        if(front == -1 || front>rear)
        {
            printf("\n UNDERFLOW");
            return -1;
        }
        else
        {
            val = queue[front];
            front++;
            if(front > rear)
            front = rear = -1;
            return val;
        }
    }
    int peek()
    {
        if(front==-1 || front>rear)
        {
            printf("\n QUEUE IS EMPTY");
            return -1;
        }
        else
        {
            return queue[front];
        }
    }
    void display()
    {
        int i;
        printf("\n");
        if(front == -1 || front > rear)
        printf("\n QUEUE IS EMPTY");
        else
        {
            for(i = front;i <= rear;i++)
            printf("\t %d", queue[i]);
        }
    }
}

```

### Output

```

***** MAIN MENU *****
1. Insert an element
2. Delete an element
3. Peek
4. Display the queue
5. EXIT
Enter your option : 1
Enter the number to be inserted in the queue : 50

```

**Note** The process of inserting an element in the queue is called enqueue, and the process of deleting an element from the queue is called dequeue.

## 8.3 LINKED REPRESENTATION OF QUEUES

We have seen how a queue is created using an array. Although this technique of creating a queue is easy, its drawback is that the array must be declared to have some fixed size. If we allocate space for 50 elements in the queue and it hardly uses 20–25 locations, then half of the space will

be wasted. And in case we allocate less memory locations for a queue that might end up growing large and large, then a lot of re-allocations will have to be done, thereby creating a lot of overhead and consuming a lot of time.

In case the queue is a very small one or its maximum size is known in advance, then the array implementation of the queue gives an efficient implementation. But if the array size cannot be determined in advance, the other alternative, i.e., the linked representation is used.

The storage requirement of linked representation of a queue with  $n$  elements is  $O(n)$  and the typical time requirement for operations is  $O(1)$ .

In a linked queue, every element has two parts, one that stores the data and another that stores the address of the next element. The `START` pointer of the linked list is used as `FRONT`. Here, we will also use another pointer called `REAR`, which will store the address of the last element in the queue. All insertions will be done at the rear end and all the deletions will be done at the front end. If `FRONT = REAR = NULL`, then it indicates that the queue is empty.

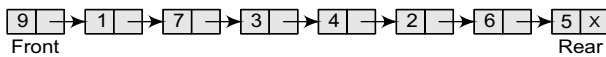
The linked representation of a queue is shown in Fig. 8.6.

### Operations on Linked Queues

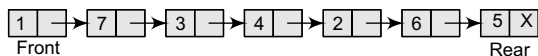
A queue has two basic operations: `insert` and `delete`. The `insert` operation adds an element to the end of the queue, and the `delete` operation removes an element from the front or the start of the queue. Apart from this, there is another operation `peek` which returns the value of the first element of the queue.

#### Insert Operation

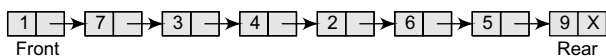
The `insert` operation is used to insert an element into a queue. The new element is added as the last element of the queue. Consider the linked queue shown in Fig. 8.7.



**Figure 8.6** Linked queue



**Figure 8.7** Linked queue



**Figure 8.8** Linked queue after inserting a new node

```

Step 1: Allocate memory for the new node and name it as PTR
Step 2: SET PTR->DATA = VAL
Step 3: IF FRONT = NULL
        SET FRONT = REAR = PTR
        SET FRONT->NEXT = REAR->NEXT = NULL
    ELSE
        SET REAR->NEXT = PTR
        SET REAR = PTR
        SET REAR->NEXT = NULL
    [END OF IF]
Step 4: END

```

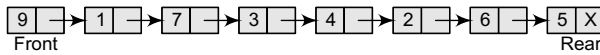
**Figure 8.9** Algorithm to insert an element in a linked queue

To insert an element with value 9, we first check if `FRONT=NULL`. If the condition holds, then the queue is empty. So, we allocate memory for a new node, store the value in its `DATA` part and `NULL` in its `NEXT` part. The new node will then be called both `FRONT` and `REAR`. However, if `FRONT != NULL`, then we will insert the new node at the rear end of the linked queue and name this new node as `REAR`. Thus, the updated queue becomes as shown in Fig. 8.8.

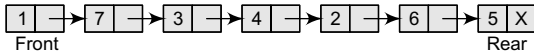
Figure 8.9 shows the algorithm to insert an element in a linked queue. In Step 1, the memory is allocated for the new node. In Step 2, the `DATA` part of the new node is initialized with the value to be stored in the node. In Step 3, we check if the new node is the first node of the linked queue. This is done by checking if `FRONT = NULL`. If this is the case, then the new node is tagged as `FRONT` as well as `REAR`. Also `NULL` is stored in the `NEXT` part of the node (which is also the `FRONT` and the `REAR` node). However, if the new node is not the first node in the list, then it is added at the `REAR` end of the linked queue (or the last node of the queue).

### Delete Operation

The delete operation is used to delete the element that is first inserted in a queue, i.e., the element whose address is stored in `FRONT`. However, before deleting the value, we must first check if `FRONT=NULL` because if this is the case, then the queue is empty and no more deletions can be done.



**Figure 8.10** Linked queue



**Figure 8.11** Linked queue after deletion of an element

```

Step 1: IF FRONT = NULL
        Write "Underflow"
        Go to Step 5
      [END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT->NEXT
Step 4: FREE PTR
Step 5: END
  
```

**Figure 8.12** Algorithm to delete an element from a linked queue

If an attempt is made to delete a value from a queue that is already empty, an underflow message is printed. Consider the queue shown in Fig. 8.10.

To delete an element, we first check if `FRONT=NULL`. If the condition is false, then we delete the first node pointed by `FRONT`. The `FRONT` will now point to the second element of the linked queue. Thus, the updated queue becomes as shown in Fig. 8.11.

Figure 8.12 shows the algorithm to delete an element from a linked queue. In Step 1, we first check for the underflow condition. If the condition is true, then an appropriate message is displayed, otherwise in Step 2, we use a pointer `PTR` that points to `FRONT`. In Step 3, `FRONT` is made to point to the next node in sequence. In Step 4, the memory occupied by `PTR` is given back to the free pool.

### PROGRAMMING EXAMPLE

#### 2. Write a program to implement a linked queue.

```

#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct node
{
    int data;
    struct node *next;
};
struct queue
{
    struct node *front;
    struct node *rear;
};
struct queue *q;
void create_queue(struct queue *);
struct queue *insert(struct queue *,int);
struct queue *delete_element(struct queue *);
struct queue *display(struct queue *);
int peek(struct queue *);
int main()
{
    int val, option;
    create_queue(q);
    clrscr();
    do
    {
        printf("\n *****MAIN MENU*****");
        printf("\n 1. INSERT");
        printf("\n 2. DELETE");
    }
  
```

```

        printf("\n 3. PEEK");
        printf("\n 4. DISPLAY");
        printf("\n 5. EXIT");
        printf("\n Enter your option : ");
        scanf("%d", &option);
        switch(option)
        {
            case 1:
                printf("\n Enter the number to insert in the queue:");
                scanf("%d", &val);
                q = insert(q, val);
                break;
            case 2:
                q = delete_element(q);
                break;
            case 3:
                val = peek(q);
                if(val != -1)
                    printf("\n The value at front of queue is : %d", val);
                break;
            case 4:
                q = display(q);
                break;
        }
    }while(option != 5);
    getch();
    return 0;
}

void create_queue(struct queue *q)
{
    q->rear = NULL;
    q->front = NULL;
}

struct queue *insert(struct queue *q, int val)
{
    struct node *ptr;
    ptr = (struct node*)malloc(sizeof(struct node));
    ptr->data = val;
    if(q->front == NULL)
    {
        q->front = ptr;
        q->rear = ptr;
        q->front->next = q->rear->next = NULL;
    }
    else
    {
        q->rear->next = ptr;
        q->rear = ptr;
        q->rear->next = NULL;
    }
    return q;
}

struct queue *display(struct queue *q)
{
    struct node *ptr;
    ptr = q->front;
    if(ptr == NULL)
        printf("\n QUEUE IS EMPTY");
    else
    {
        printf("\n");
    }
}

```

```

        while(ptr!=q->rear)
        {
            printf("%d\t", ptr->data);
            ptr = ptr->next;
        }
        printf("%d\t", ptr->data);
    }
    return q;
}
struct queue *delete_element(struct queue *q)
{
    struct node *ptr;
    ptr = q->front;
    if(q->front == NULL)
        printf("\n UNDERFLOW");
    else
    {
        q->front = q->front->next;
        printf("\n The value being deleted is : %d", ptr->data);
        free(ptr);
    }
    return q;
}
int peek(struct queue *q)
{
    if(q->front==NULL)
    {
        printf("\n QUEUE IS EMPTY");
        return -1;
    }
    else
        return q->front->data;
}

```

**Output**

```

*****MAIN MENU*****
1. INSERT
2. DELETE
3. PEEK
4. DISPLAY
5. EXIT
Enter your option : 3
QUEUE IS EMPTY
Enter your option : 5

```

**8.4 TYPES OF QUEUES**

A queue data structure can be classified into the following types:

1. Circular Queue
2. Deque
3. Priority Queue
4. Multiple Queue

We will discuss each of these queues in detail in the following sections.

**8.4.1 Circular Queues**

In linear queues, we have discussed so far that insertions can be done only at one end called the REAR and deletions are always done from the other end called the FRONT. Look at the queue shown in Fig. 8.13.

54	9	7	18	14	36	45	21	99	72
0	1	2	3	4	5	6	7	8	9

**Figure 8.13** Linear queue

Here, FRONT = 0 and REAR = 9.



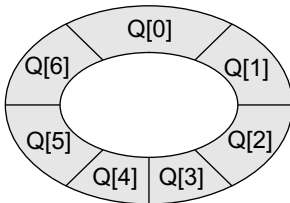
Now, if you want to insert another value, it will not be possible because the queue is completely full. There is no empty space where the value can be inserted. Consider a scenario in which two successive deletions are made. The queue will then be given as shown in Fig. 8.14.

			7	18	14	36	45	21	99	72
0	1	2	3	4	5	6	7	8	9	

**Figure 8.14** Queue after two successive deletions

Here,  $\text{FRONT} = 2$  and  $\text{REAR} = 9$ .

Suppose we want to insert a new element in the queue shown in Fig. 8.14. Even though there is space available, the overflow condition still exists because the condition  $\text{REAR} = \text{MAX} - 1$  still holds true. This is a major drawback of a linear queue.



**Figure 8.15** Circular queue

90	49	7	18	14	36	45	21	99	72
FRONT = 0	1	2	3	4	5	6	7	8	REAR = 9

**Figure 8.16** Full queue

90	49	7	18	14	36	45	21	99	
FRONT = 0	1	2	3	4	5	6	7	REAR = 8	9

Increment rear so that it points to location 9 and insert the value here

**Figure 8.17** Queue with vacant locations

		7	18	14	36	45	21	80	81
0	1	FRONT = 2	3	4	5	6	7	8	REAR = 9

Set  $\text{REAR} = 0$  and insert the value here

**Figure 8.18** Inserting an element in a circular queue

```

Step 1: IF FRONT = 0 and REAR = MAX - 1
        Write "OVERFLOW"
        Goto step 4
    [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
        SET FRONT = REAR = 0
    ELSE IF REAR = MAX - 1 and FRONT != 0
        SET REAR = 0
    ELSE
        SET REAR = REAR + 1
    [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT

```

**Figure 8.19** Algorithm to insert an element in a circular queue

To resolve this problem, we have two solutions. First, shift the elements to the left so that the vacant space can be occupied and utilized efficiently. But this can be very time-consuming, especially when the queue is quite large.

The second option is to use a circular queue. In the circular queue, the first index comes right after the last index. Conceptually, you can think of a circular queue as shown in Fig. 8.15.

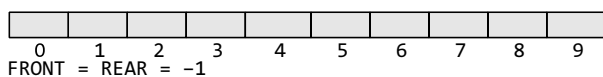
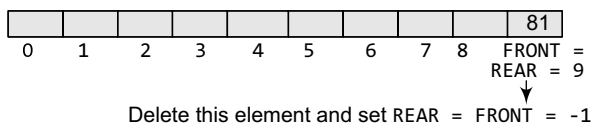
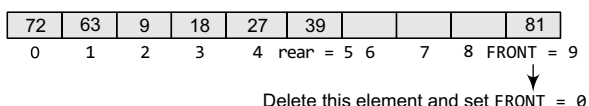
The circular queue will be full only when  $\text{FRONT} = 0$  and  $\text{REAR} = \text{MAX} - 1$ . A circular queue is implemented in the same manner as a linear queue is

implemented. The only difference will be in the code that performs insertion and deletion operations. For insertion, we now have to check for the following three conditions:

- If  $\text{FRONT} = 0$  and  $\text{REAR} = \text{MAX} - 1$ , then the circular queue is full. Look at the queue given in Fig. 8.16 which illustrates this point.
- If  $\text{REAR} \neq \text{MAX} - 1$ , then  $\text{REAR}$  will be incremented and the value will be inserted as illustrated in Fig. 8.17.
- If  $\text{FRONT} \neq 0$  and  $\text{REAR} = \text{MAX} - 1$ , then it means that the queue is not full. So, set  $\text{REAR} = 0$  and insert the new element there, as shown in Fig. 8.18.

Let us look at Fig. 8.19 which shows the algorithm to insert an element in a circular queue. In Step 1, we check for the overflow condition. In Step 2, we make two checks. First to see if the queue is empty, and second to see if the  $\text{REAR}$  end has already reached the maximum capacity while there are certain free locations before the  $\text{FRONT}$  end. In Step 3, the value is stored in the queue at the location pointed by  $\text{REAR}$ .

After seeing how a new element is added in a circular queue, let us now discuss how deletions are performed in this case. To delete an element, again we check for three conditions.

**Figure 8.20** Empty queue**Figure 8.21** Queue with a single element**Figure 8.22** Queue where FRONT = MAX-1 before deletion

```

Step 1: IF FRONT = -1
        Write "UNDERFLOW"
        Goto Step 4
    [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
        SET FRONT = REAR = -1
    ELSE
        IF FRONT = MAX - 1
            SET FRONT = 0
        ELSE
            SET FRONT = FRONT + 1
        [END of IF]
    [END OF IF]
Step 4: EXIT

```

**Figure 8.23** Algorithm to delete an element from a circular queue

- Look at Fig. 8.20. If FRONT = -1, then there are no elements in the queue. So, an underflow condition will be reported.
- If the queue is not empty and FRONT = REAR, then after deleting the element at the front the queue becomes empty and so FRONT and REAR are set to -1. This is illustrated in Fig. 8.21.
- If the queue is not empty and FRONT = MAX-1, then after deleting the element at the front, FRONT is set to 0. This is shown in Fig. 8.22.

Let us look at Fig. 8.23 which shows the algorithm to delete an element from a circular queue. In Step 1, we check for the underflow condition. In Step 2, the value of the queue at the location pointed by FRONT is stored in VAL. In Step 3, we make two checks. First to see if the queue has become empty after deletion and second to see if FRONT has reached the maximum capacity of the queue. The value of FRONT is then updated based on the outcome of these checks.

### PROGRAMMING EXAMPLE

#### 3. Write a program to implement a circular queue.

```

#include <stdio.h>
#include <conio.h>
#define MAX 10
int queue[MAX];
int front=-1, rear=-1;
void insert(void);
int delete_element(void);
int peek(void);
void display(void);
int main()

```

```

{
    int option, val;
    clrscr();
    do
    {
        printf("\n ***** MAIN MENU *****");
        printf("\n 1. Insert an element");
        printf("\n 2. Delete an element");
        printf("\n 3. Peek");
        printf("\n 4. Display the queue");
        printf("\n 5. EXIT");
        printf("\n Enter your option : ");
        scanf("%d", &option);
        switch(option)
        {
            case 1:
                insert();
                break;
            case 2:
                val = delete_element();
                if(val!=-1)

```

```

        printf("\n The number deleted is : %d", val);
        break;
    case 3:
        val = peek();
        if(val!=-1)
            printf("\n The first value in queue is : %d", val);
        break;
    case 4:
        display();
        break;
    }
    }while(option!=5);
    getch();
    return 0;
}

void insert()
{
    int num;
    printf("\n Enter the number to be inserted in the queue : ");
    scanf("%d", &num);
    if(front==0 && rear==MAX-1)
        printf("\n OVERFLOW");
    else if(front==--1 && rear==--1)
    {
        front=rear=0;
        queue[rear]=num;
    }
    else if(rear==MAX-1 && front!=0)
    {
        rear=0;
        queue[rear]=num;
    }
    else
    {
        rear++;
        queue[rear]=num;
    }
}

int delete_element()
{
    int val;
    if(front==--1 && rear==--1)
    {
        printf("\n UNDERFLOW");
        return -1;
    }
    val = queue[front];
    if(front==rear)
        front=rear=-1;
    else
    {
        if(front==MAX-1)
            front=0;
        else
            front++;
    }
    return val;
}

int peek()
{
    if(front==--1 && rear==--1)
    {
        printf("\n QUEUE IS EMPTY");
        return -1;
    }

```

```

    }
    else
    {
        return queue[front];
    }
}
void display()
{
    int i;
    printf("\n");
    if (front == -1 && rear == -1)
        printf ("\n QUEUE IS EMPTY");
    else
    {
        if(front<rear)
        {
            for(i=front;i<=rear;i++)
                printf("\t %d", queue[i]);
        }
        else
        {
            for(i=front;i<MAX;i++)
                printf("\t %d", queue[i]);
            for(i=0;i<=rear;i++)
                printf("\t %d", queue[i]);
        }
    }
}
}

```

### Output

```

***** MAIN MENU *****
1. Insert an element
2. Delete an element
3. Peek
4. Display the queue
5. EXIT
Enter your option : 1
Enter the number to be inserted in the queue : 25
Enter your option : 2
The number deleted is : 25
Enter your option : 3
QUEUE IS EMPTY
Enter your option : 5

```

## 8.4.2 Deques

A deque (pronounced as ‘deck’ or ‘dequeue’) is a list in which the elements can be inserted or deleted at either end. It is also known as a *head-tail linked list* because elements can be added to or removed from either the front (head) or the back (tail) end.

However, no element can be added and deleted from the middle. In the computer’s memory, a deque is implemented using either a circular array or a circular doubly linked list. In a deque, two pointers are maintained, LEFT and RIGHT, which point to either end of the deque. The elements in a deque extend from the LEFT end to the RIGHT end and since it is circular, Dequeue[N-1] is followed by Dequeue[0]. Consider the deques shown in Fig. 8.24.

There are two variants of a double-ended queue. They include

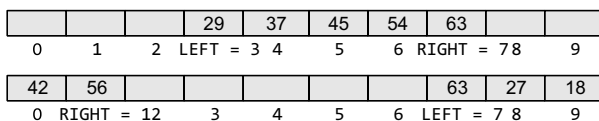


Figure 8.24 Double-ended queues

- *Input restricted deque* In this dequeue, insertions can be done only at one of the ends, while deletions can be done from both ends.
- *Output restricted deque* In this dequeue, deletions can be done only at one of the ends, while insertions can be done on both ends.

## PROGRAMMING EXAMPLE

4. Write a program to implement input and output restricted deque.

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
int deque[MAX];
int left = -1, right = -1;
void input_deque(void);
void output_deque(void);
void insert_left(void);
void insert_right(void);
void delete_left(void);
void delete_right(void);
void display(void);
int main()
{
    int option;
    clrscr();
    printf("\n *****MAIN MENU*****");
    printf("\n 1.Input restricted deque");
    printf("\n 2.Output restricted deque");
    printf("Enter your option : ");
    scanf("%d",&option);
    switch(option)
    {
        case 1:
            input_deque();
            break;
        case 2:
            output_deque();
            break;
    }
    return 0;
}

void input_deque()
{
    int option;
    do
    {
        printf("\n INPUT RESTRICTED DEQUE");
        printf("\n 1.Insert at right");
        printf("\n 2.Delete from left");
        printf("\n 3.Delete from right");
        printf("\n 4.Display");
        printf("\n 5.Quit");
        printf("\n Enter your option : ");
        scanf("%d",&option);
        switch(option)
        {
            case 1:
                insert_right();
                break;
            case 2:
                delete_left();
                break;
            case 3:
                delete_right();
                break;
            case 4:
                display();
                break;
        }
    }
    while(option!=5);
}
```

```
}
void output_deque()
{
    int option;
    do
    {
        printf("OUTPUT RESTRICTED DEQUE");
        printf("\n 1.Insert at right");
        printf("\n 2.Insert at left");
        printf("\n 3.Delete from left");
        printf("\n 4.Display");
        printf("\n 5.Quit");
        printf("\n Enter your option : ");
        scanf("%d",&option);
        switch(option)
        {
            case 1:
                insert_right();
                break;
            case 2:
                insert_left();
                break;
            case 3:
                delete_left();
                break;
            case 4:
                display();
                break;
        }
    }while(option!=5);
}
void insert_right()
{
    int val;
    printf("\n Enter the value to be added:");
    scanf("%d", &val);
    if((left == 0 && right == MAX-1) || (left == right+1))
    {
        printf("\n OVERFLOW");
        return;
    }
    if (left == -1) /* if queue is initially empty */
    {
        left = 0;
        right = 0;
    }
    else
    {
        if(right == MAX-1) /*right is at last position of queue */
            right = 0;
        else
            right = right+1;
    }
    deque[right] = val ;
}
void insert_left()
{
    int val;
    printf("\n Enter the value to be added:");
    scanf("%d", &val);
    if((left == 0 && right == MAX-1) || (left == right+1))
    {
        printf("\n OVERFLOW");
        return;
    }
}
```

```

        if (left == -1)/*If queue is initially empty*/
        {
            left = 0;
            right = 0;
        }
        else
        {
            if(left == 0)
                left=MAX-1;
            else
                left=left-1;
        }
        deque[left] = val;
    }
    void delete_left()
    {
        if (left == -1)
        {
            printf("\n UNDERFLOW");
            return ;
        }
        printf("\n The deleted element is : %d", deque[left]);
        if(left == right) /*Queue has only one element */
        {
            left = -1;
            right = -1;
        }
        else
        {
            if(left == MAX-1)
                left = 0;
            else
                left = left+1;
        }
    }
    void delete_right()
    {
        if (left == -1)
        {
            printf("\n UNDERFLOW");
            return ;
        }
        printf("\n The element deleted is : %d", deque[right]);
        if(left == right) /*queue has only one element*/
        {
            left = -1;
            right = -1;
        }
        else
        {
            if(right == 0)
                right=MAX-1;
            else
                right=right-1;
        }
    }
    void display()
    {
        int front = left, rear = right;
        if(front == -1)
        {
            printf("\n QUEUE IS EMPTY");
            return;
        }
        printf("\n The elements of the queue are : ");
    }

```

```

        if(front <= rear )
        {
            while(front <= rear)
            {
                printf("%d",deque[front]);
                front++;
            }
        }
        else
        {
            while(front <= MAX-1)
            {
                printf("%d", deque[front]);
                front++;
            }
            front = 0;
            while(front <= rear)
            {
                printf("%d",deque[front]);
                front++;
            }
        }
        printf("\n");
    }
}

```

**Output**

```

***** MAIN MENU *****
1.Input restricted deque
2.Output restricted deque
Enter your option : 1
INPUT RESTRICTED DEQUEUE
1.Insert at right
2.Delete from left
3.Delete from right
4.Display
5.Quit
Enter your option : 1
Enter the value to be added : 5
Enter the value to be added : 10
Enter your option : 2
The deleted element is : 5
Enter your option : 5

```

**8.4.3 Priority Queues**

A priority queue is a data structure in which each element is assigned a priority. The priority of the element will be used to determine the order in which the elements will be processed. The general rules of processing the elements of a priority queue are

- An element with higher priority is processed before an element with a lower priority.
- Two elements with the same priority are processed on a first-come-first-served (FCFS) basis.

A priority queue can be thought of as a modified queue in which when an element has to be removed from the queue, the one with the highest-priority is retrieved first. The priority of the element can be set based on various factors. Priority queues are widely used in operating systems to execute the highest priority process first. The priority of the process may be set based on the CPU time it requires to get executed completely. For example, if there are three processes, where the first process needs 5 ns to complete, the second process needs 4 ns, and the third process needs 7 ns, then the second process will have the highest priority and will thus be the first to be executed. However, CPU time is not the only factor that determines the priority, rather it is just one among several factors. Another factor is the importance of one process over another. In case we have to run two processes at the same time, where one process is concerned with online order booking



and the second with printing of stock details, then obviously the online booking is more important and must be executed first.

### Implementation of a Priority Queue

There are two ways to implement a priority queue. We can either use a sorted list to store the elements so that when an element has to be taken out, the queue will not have to be searched for the element with the highest priority or we can use an unsorted list so that insertions are always done at the end of the list. Every time when an element has to be removed from the list, the element with the highest priority will be searched and removed. While a sorted list takes  $O(n)$  time to insert an element in the list, it takes only  $O(1)$  time to delete an element. On the contrary, an unsorted list will take  $O(1)$  time to insert an element and  $O(n)$  time to delete an element from the list.

Practically, both these techniques are inefficient and usually a blend of these two approaches is adopted that takes roughly  $O(\log n)$  time or less.

### Linked Representation of a Priority Queue

In the computer memory, a priority queue can be represented using arrays or linked lists. When a priority queue is implemented using a linked list, then every node of the list will have three parts: (a) the information or data part, (b) the priority number of the element, and (c) the address of the next element. If we are using a sorted linked list, then the element with the higher priority will precede the element with the lower priority.

Consider the priority queue shown in Fig. 8.25.

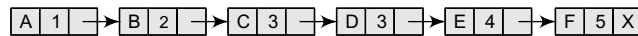


Figure 8.25 Priority queue

Lower priority number means higher priority. For example, if there are two elements A and B, where A has a priority number 1 and B has a priority number 5, then A will be processed before B as it has higher priority than B.

The priority queue in Fig. 8.25 is a sorted priority queue having six elements. From the queue, we cannot make out whether A was inserted before E or whether E joined the queue before A because the list is not sorted based on FCFS. Here, the element with a higher priority comes before the element with a lower priority. However, we can definitely say that C was inserted in the queue before D because when two elements have the same priority the elements are arranged and processed on FCFS principle.

**Insertion** When a new element has to be inserted in a priority queue, we have to traverse the entire list until we find a node that has a priority lower than that of the new element. The new node is inserted before the node with the lower priority. However, if there exists an element that has the same priority as the new element, the new element is inserted after that element. For example, consider the priority queue shown in Fig. 8.26.

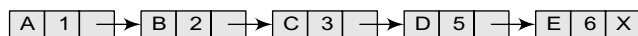


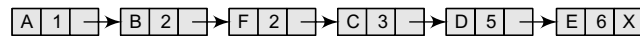
Figure 8.26 Priority queue

If we have to insert a new element with data = F and priority number = 4, then the element will be inserted before D that has priority number 5, which is lower priority than that of the new element. So, the priority queue now becomes as shown in Fig. 8.27.



Figure 8.27 Priority queue after insertion of a new node

However, if we have a new element with data = F and priority number = 2, then the element will be inserted after B, as both these elements have the same priority but the insertions are done on FCFS basis as shown in Fig. 8.28.



**Figure 8.28** Priority queue after insertion of a new node

**Deletion** Deletion is a very simple process in this case. The first node of the list will be deleted and the data of that node will be processed first.

### Array Representation of a Priority Queue

When arrays are used to implement a priority queue, then a separate queue for each priority number is maintained. Each of these queues will be implemented using circular arrays or circular queues. Every individual queue will have its own FRONT and REAR pointers.

We use a two-dimensional array for this purpose where each queue will be allocated the same amount of space. Look at the two-dimensional representation of a priority queue given below. Given the FRONT and REAR values of each queue, the two-dimensional matrix can be formed as shown in Fig. 8.29.

FRONT[K] and REAR[K] contain the front and rear values of row  $\kappa$ , where  $\kappa$  is the priority number. Note that here we are assuming that the row and column indices start from 1, not 0. Obviously, while programming, we will not take such assumptions.

FRONT	REAR	
3	3	1 2 3 4 5
1	3	1 [ A ]
4	5	2 [ B C D ]
4	1	3 [ E F ]
		4 [ I G H ]

**Figure 8.29** Priority queue matrix

FRONT	REAR	
3	3	1 2 3 4 5
1	3	1 [ A ]
4	1	2 [ B C D ]
4	1	3 [ R E F ]
		4 [ I G H ]

**Figure 8.30** Priority queue matrix after insertion of a new element

**Insertion** To insert a new element with priority  $\kappa$  in the priority queue, add the element at the rear end of row  $\kappa$ , where  $\kappa$  is the row number as well as the priority number of that element. For example, if we have to insert an element R with priority number 3, then the priority queue will be given as shown in Fig. 8.30.

**Deletion** To delete an element, we find the first non-empty queue and then process the front element of the first non-empty queue. In our priority queue, the first non-empty queue is the one with priority number 1 and the front element is A, so A will be deleted and processed first. In technical terms, find the element with the smallest  $\kappa$ , such that FRONT[K]  $\neq$  NULL.

### PROGRAMMING EXAMPLE

5. Write a program to implement a priority queue.

```

#include <stdio.h>
#include <malloc.h>
#include <conio.h>
struct node
{
    int data;
    int priority;
    struct node *next;
}
  
```

```

struct node *start=NULL;
struct node *insert(struct node *);
struct node *delete(struct node *);
void display(struct node *);
int main()
{
    int option;
    clrscr();
    do
    {
        printf("\n *****MAIN MENU*****");
        printf("\n 1. INSERT");
        printf("\n 2. DELETE");
        printf("\n 3. DISPLAY");
        printf("\n 4. EXIT");
        printf("\n Enter your option : ");
        scanf( "%d", &option);
        switch(option)
        {
            case 1:
                start=insert(start);
                break;
            case 2:
                start = delete(start);
                break;
            case 3:
                display(start);
                break;
        }
    }while(option!=4);
}
struct node *insert(struct node *start)
{
    int val, pri;
    struct node *ptr, *p;
    ptr = (struct node *)malloc(sizeof(struct node));
    printf("\n Enter the value and its priority : " );
    scanf( "%d %d", &val, &pri);
    ptr->data = val;
    ptr->priority = pri;
    if(start==NULL || pri < start->priority )
    {
        ptr->next = start;
        start = ptr;
    }
    else
    {
        p = start;
        while(p->next != NULL && p->next->priority <= pri)
            p = p->next;
        ptr->next = p->next;
        p->next = ptr;
    }
    return start;
}
struct node *delete(struct node *start)
{
    struct node *ptr;
    if(start == NULL)
    {
        printf("\n UNDERFLOW" );
        return;
    }
    else

```

```

    {
        ptr = start;
        printf("\n Deleted item is: %d", ptr->data);
        start = start->next;
        free(ptr);
    }
    return start;
}
void display(struct node *start)
{
    struct node *ptr;
    ptr = start;
    if(start == NULL)
        printf("\nQUEUE IS EMPTY" );
    else
    {
        printf("\n PRIORITY QUEUE IS : " );
        while(ptr != NULL)
        {
            printf( "\t%d[priority=%d]", ptr->data, ptr->priority );
            ptr=ptr->next;
        }
    }
}
}

```

**Output**

```

*****MAIN MENU*****
1. INSERT
2. DELETE
3. DISPLAY
4. EXIT
Enter your option : 1
Enter the value and its priority : 5 2
Enter the value and its priority : 10 1
Enter your option : 3
PRIORITY QUEUE IS :
10[priority = 1] 5[priority = 2]
Enter your option : 4

```

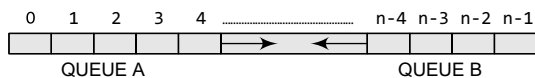
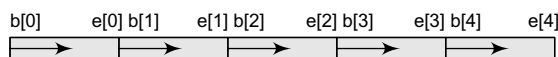
**8.4.4 Multiple Queues**

When we implement a queue using an array, the size of the array must be known in advance. If the queue is allocated less space, then frequent overflow conditions will be encountered. To deal with this problem, the code will have to be modified to reallocate more space for the array.

In case we allocate a large amount of space for the queue, it will result in sheer wastage of the memory. Thus, there lies a tradeoff between the frequency of overflows and the space allocated.

So a better solution to deal with this problem is to have multiple queues or to have more than one queue in the same array of sufficient size. Figure 8.31 illustrates this concept.

In the figure, an array `QUEUE[n]` is used to represent two queues, `QUEUE A` and `QUEUE B`. The value of `n` is such that the combined size of both the queues will never exceed `n`. While operating on these queues, it is important to note one thing—`QUEUE A` will grow from left to right, whereas `QUEUE B` will grow from right to left at the same time.

**Figure 8.31** Multiple queues**Figure 8.32** Multiple queues

Extending the concept to multiple queues, a queue can also be used to represent `n` number of queues in the same array. That is, if we have a `QUEUE[n]`, then each `QUEUE i` will be allocated an equal amount of space bounded by indices `b[i]` and `e[i]`. This is shown in Fig. 8.32.