4.5. Queue:

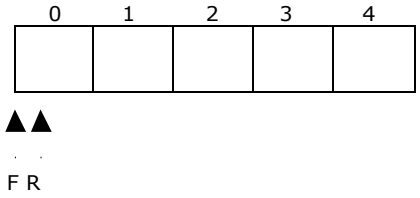
A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. Another name for a queue is a "FIFO" or "First-in-first-out" list.

The operations for a queue are analogues to those for a stack, the difference is that the insertions go at the end of the list, rather than the beginning. We shall use the following operations on queues:

- *enqueue*: which inserts an element at the end of the queue.
- *dequeue*: which deletes an element at the start of the queue.

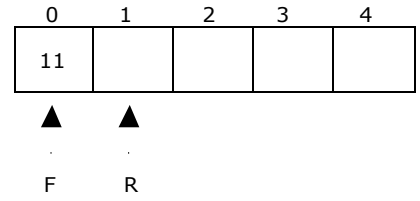
4.6.1. Representation of Queue:

Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



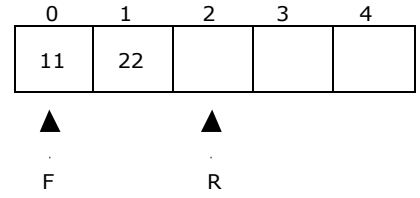
Queue Empty
FRONT = REAR = 0

Now, insert 11 to the queue. Then queue status will be:



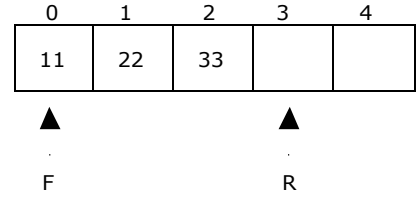
REAR = REAR + 1 = 1
FRONT = 0

Next, insert 22 to the queue. Then the queue status is:



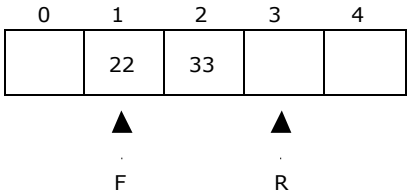
REAR = REAR + 1 = 2
FRONT = 0

Again insert another element 33 to the queue. The status of the queue is:



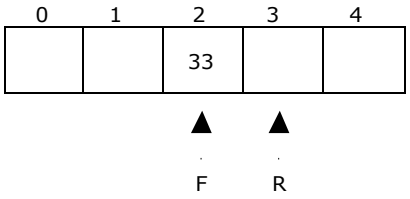
REAR = REAR + 1 = 3
FRONT = 0

Now, delete an element. The element deleted is the element at the front of the queue. So the status of the queue is:



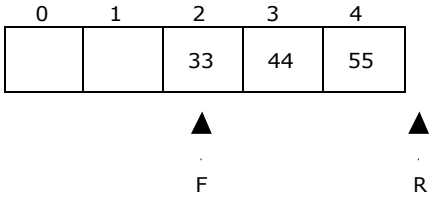
RE A R = 3
F RO NT = F R O NT + 1 = 1

Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



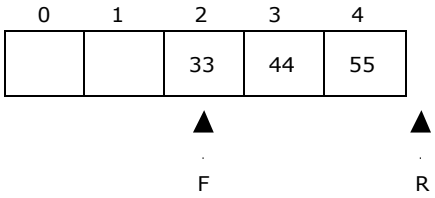
RE A R = 3
F RO NT = F R O NT + 1 = 2

Now, insert new elements 44 and 55 into the queue. The queue status is:



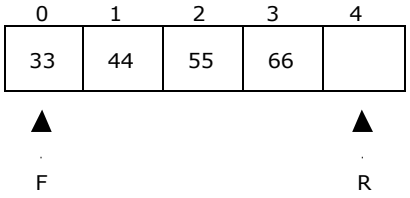
RE A R = 5
FRO NT = 2

Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



RE A R = 5
FRO NT = 2

Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To over come this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:



RE A R = 4
FRO NT = 0

This difficulty can overcome if we treat queue position with index 0 as a position that

comes after position with index 4 i.e., we treat the queue as a **circular queue.**

4.6.2. Source code for Queue operations using array:

In order to create a queue we require a one dimensional array Q(1:n) and two variables *front* and *rear*. The conventions we shall adopt for these two variables are that *front* is always 1 less than the actual front of the queue and rear always points to the last element in the queue. Thus, front = rear if and only if there are no elements in the queue. The initial condition then is front = rear = 0. The various queue operations to perform creation, deletion and display the elements in a queue are as follows:

1. insertQ(): inserts an element at the end of queue Q.
2. deleteQ(): deletes the first element of Q.
3. displayQ(): displays the elements in the queue.

```
# include <conio.h>
# define MAX 6
int Q[MAX];
int front, rear;

void insertQ()
{
        int data;
        if(rear == MAX)
        {
                printf("\n Linear Queue is full");
                return;
        }
        else
        {
                printf("\n Enter data: ");
                scanf("%d", &data);
                Q[rear] = data;
                rear++;
                printf("\n Data Inserted in the Queue ");
        }
}

void deleteQ()
{
        if(rear == front)
        {
                printf("\n\n Queue is Empty..");
                return;
        }
        else
        {
                printf("\n Deleted element from Queue is %d", Q[front]);
                front++;
        }
}

void displayQ()
{
        int i;
        if(front == rear)
        {
                printf("\n\n\t Queue is Empty");
                return;
        }
        else
        {
                printf("\n Elements in Queue are: ");
                for(i = front; i < rear; i++)
```

```
                {
                        printf("%d\t", Q[i]);
                }
        }
}
int menu()
{
        int ch;
        clrscr();
        printf("\n \tQueue operations using ARRAY..");
        printf("\n ----------**********----------\n");
        printf("\n 1. Insert ");
        printf("\n 2. Delete ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter your choice: ");
        scanf("%d", &ch);
        return ch;
}
void main()
{
        int ch;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case 1:
                                insertQ();
                                break;
                        case 2:
                                deleteQ();
                                break;
                        case 3:
                                displayQ();
                                break;
                        case 4:
                                return;
                }

                getch();
        } while(1);
}
```

4.6.3. Linked List Implementation of Queue:

We can represent a queue as a linked list. In a queue data is deleted from the front end and inserted at the rear end. We can perform similar operations on the two ends of a list. We use two pointers *front* and *rear* for our linked queue implementation.

The linked queue looks as shown in figure 4.4:

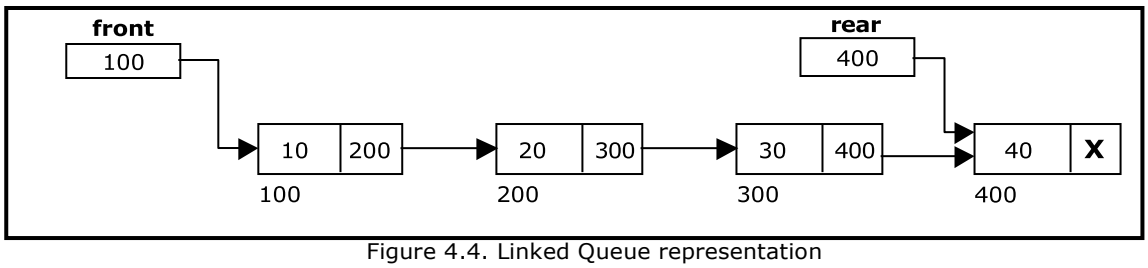


Figure 4.4. Linked Queue representation

4.6.4. Source code for queue operations using linked list:

```
# include <stdlib.h>
# include <conio.h>

struct queue
{
        int data;
        struct queue *next;
};
typedef struct queue node;
node *front = NULL;
node *rear =  NULL;

node* getnode()
{
        node *temp;
        temp = (node *) malloc(sizeof(node)) ;
        printf("\n Enter data ");
        scanf("%d", &temp -> data);
        temp -> next = NULL;
        return temp;
}
void insertQ()
{
        node *newnode;
        newnode = getnode();
        if(newnode == NULL)
        {
                printf("\n Queue Full");
                return;
        }
        if(front == NULL)
        {
                front = newnode;
                rear = newnode;
        }
        else
        {
                rear -> next = newnode;
                rear = newnode;
        }

        printf("\n\n\t Data Inserted into the Queue..");
}
void deleteQ()
{
        node *temp;
        if(front == NULL)
        {
                printf("\n\n\t Empty Queue..");
                return;
        }
        temp = front;
        front = front -> next;
        printf("\n\n\t Deleted element from queue is %d ", temp -> data);
        free(temp);
}
```

```
void displayQ()
{
	node *temp;
	if(front == NULL)
	{
		printf("\n\n\t\t Empty Queue ");
	}
	else
	{
		temp = front;
		printf("\n\n\n\t\t Elements in the Queue are: ");
		while(temp != NULL )
		{
			printf("%5d ", temp -> data);
			temp = temp -> next;
		}
	}
}

char menu()
{
	char ch;
	clrscr();
	printf("\n \t..Queue operations using pointers.. ");
	printf("\n\t ----------**********------------\n");
	printf("\n 1. Insert ");
	printf("\n 2. Delete ");
	printf("\n 3. Display");
	printf("\n 4. Quit ");
	printf("\n Enter your choice: ");
	ch = getche();
	return ch;
}

void main()
{
	char ch;
	do
	{
		ch = menu();
		switch(ch)
		{
			case '1' :
				insertQ();
				break;
			case '2' :
				deleteQ();
				break;
			case '3' :
				displayQ();
				break;
			case '4':
				return;
		}
		getch();
	} while(ch != '4');
}
```

4.7. Applications of Queue:

1. It is used to schedule the jobs to be processed by the CPU.

2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.

3. Breadth first search uses a queue data structure to find an element from a graph.

4.8. Circular Queue:

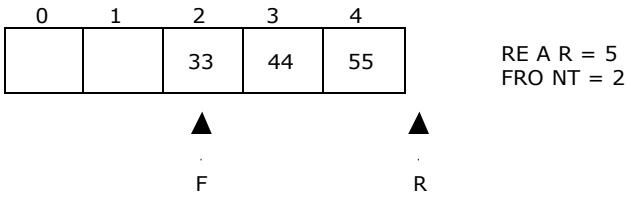
A more efficient queue representation is obtained by regarding the array Q[MAX] as circular. Any number of items could be placed on the queue. This implementation of a queue is called a circular queue because it uses its storage array as if it were a circle instead of a linear list.

There are two problems associated with linear queue. They are:

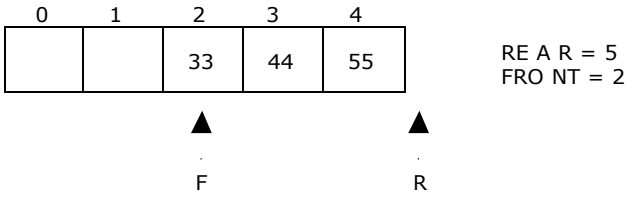
- Time consuming: linear time to be spent in shifting the elements to the beginning of the queue.

- Signaling queue full: even if the queue is having vacant position.

For example, let us consider a linear queue status as follows:



Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:

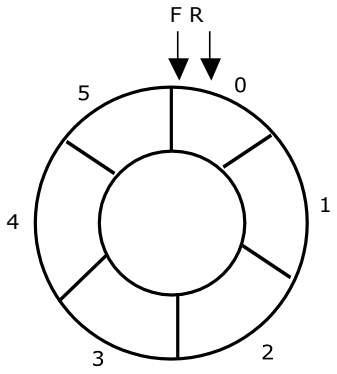


This difficulty can be overcome if we treat queue position with index zero as a position that comes after position with index four then we treat the queue as a **circular queue.**

In circular queue if we reach the end for inserting elements to it, it is possible to insert new elements if the slots at the beginning of the circular queue are empty.

4.8.1. Representation of Circular Queue:

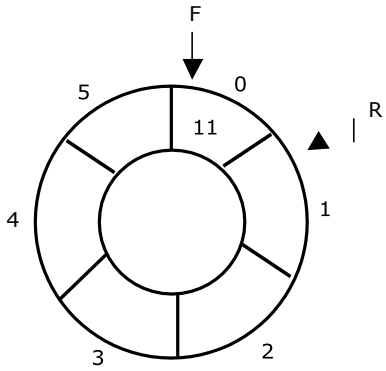
Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.



Circ u lar Q u e u e

Queue Empty
MA X = 6
F RO NT = RE A R = 0
CO U NT = 0

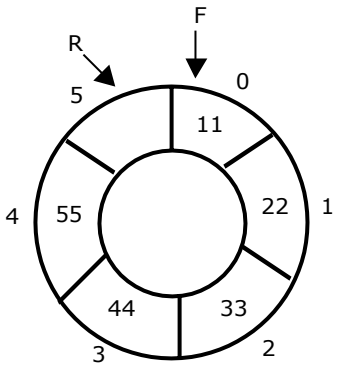
Now, insert 11 to the circular queue. Then circular queue status will be:



Circ u lar Q u e u e

FRO NT = 0
RE A R = (RE A R + 1) % 6 = 1
CO U NT = 1

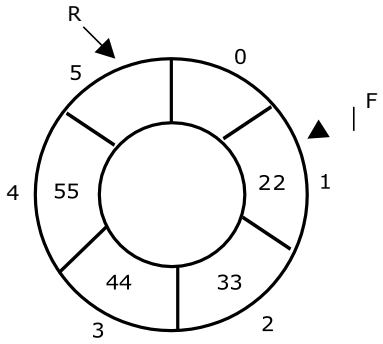
Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:



Circular Queue

FRONT = 0
REAR = (REAR + 1) % 6 = 5
COUNT = 5

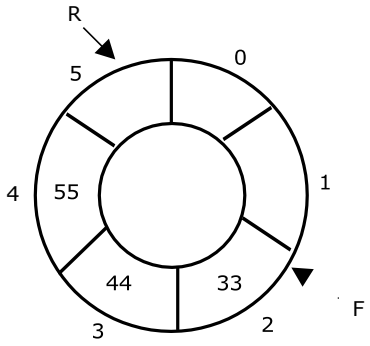
Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:



Circ u lar Q u e u e

F RO NT = (F R O NT + 1) % 6 = 1
RE A R = 5
CO U NT = CO U NT - 1 = 4

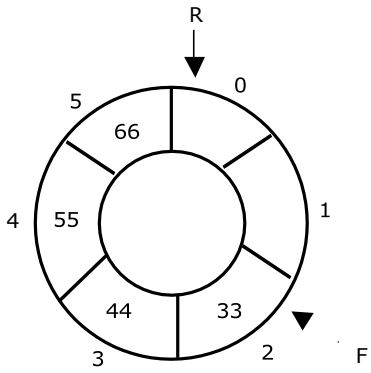
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:



Circ u lar Q u e u e

F RO NT = (F R O NT + 1) % 6 = 2
RE A R = 5
CO U NT = CO U NT - 1 = 3

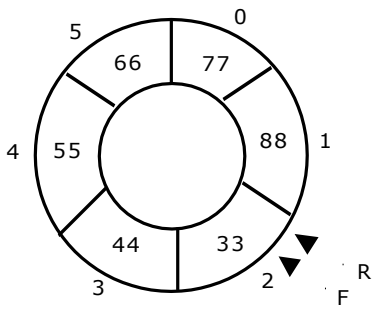
Again, insert another element 66 to the circular queue. The status of the circular queue is:



Circ u lar Q u e u e

FRO NT = 2
RE A R = (RE A R + 1) % 6 = 0
C O U NT = C O U NT + 1 = 4

Now, insert new elements 77 and 88 into the circular queue. The circular queue status is:



Circ u lar Q u e u e

F RO NT = 2, RE A R = 2
RE A R = RE A R % 6 = 2
CO U NT = 6

Now, if we insert an element to the circular queue, as COUNT = MAX we cannot add the element to circular queue. So, the circular queue is *full*.

4.8.2. Source code for Circular Queue operations, using array:

```
# include <stdio.h>
# include <conio.h>
# define MAX 6

int CQ[MAX];
int front = 0;
int rear = 0;
int count = 0;

void insertCQ()
{
        int data;
        if(count == MAX)
        {
                printf("\n Circular Queue is Full");
        }
        else
        {
                printf("\n Enter data: ");
                scanf("%d", &data);
                CQ[rear] = data;
                rear = (rear + 1) % MAX;
                count ++;
                printf("\n Data Inserted in the Circular Queue ");
        }
}


void deleteCQ()
{
        if(count == 0)
        {
                printf("\n\nCircular Queue is Empty..");
        }
        else
        {
                printf("\n Deleted element from Circular Queue is %d ", CQ[front]);
                front = (front + 1) % MAX;
                count --;
        }
}
```

```
void displayCQ()
{
	int i, j;
	if(count == 0)
	{
		printf("\n\n\t Circular Queue is Empty ");
	}
	else
	{
		printf("\n Elements in Circular Queue are: ");
		j = count;
		for(i = front; j != 0; j--)
		{
			printf("%d\t", CQ[i]);
			i = (i + 1) % MAX;
		}
	}
}

int menu()
{
	int ch;
	clrscr();
	printf("\n \t Circular Queue Operations using ARRAY..");
	printf("\n ----------**********----------\n");
	printf("\n 1. Insert ");
	printf("\n 2. Delete ");
	printf("\n 3. Display");
	printf("\n 4. Quit ");
	printf("\n Enter Your Choice: ");
	scanf("%d", &ch);
	return ch;
}

void main()
{
	int ch;
	do
	{
		ch = menu();
		switch(ch)
		{
			case 1:
				insertCQ();
				break;
			case 2:
				deleteCQ();
				break;
			case 3:
				displayCQ();
				break;
			case 4:
				return;
			default:
				printf("\n Invalid Choice ");
		}

		getch();
	} while(1);
}
```

4.9. Deque:

In the preceding section we saw that a queue in which we insert items at one end and from which we remove items at the other end. In this section we examine an extension of the queue, which provides a means to insert and remove items at both ends of the queue. This data structure is a *deque*. The word *deque* is an acronym derived from *double-ended queue*. Figure 4.5 shows the representation of a deque.

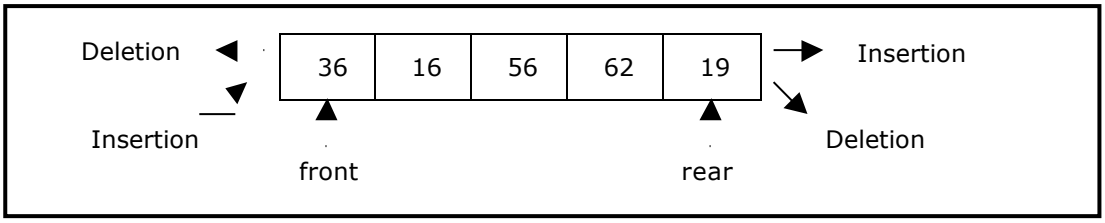


Figure 4.5. Representation of a deque.

A deque provides four operations. Figure 4.6 shows the basic operations on a deque.

- enqueue_front: insert an element at front.
- dequeue_front: delete an element at front.
- enqueue_rear: insert element at rear.
- dequeue_rear: delete element at rear.

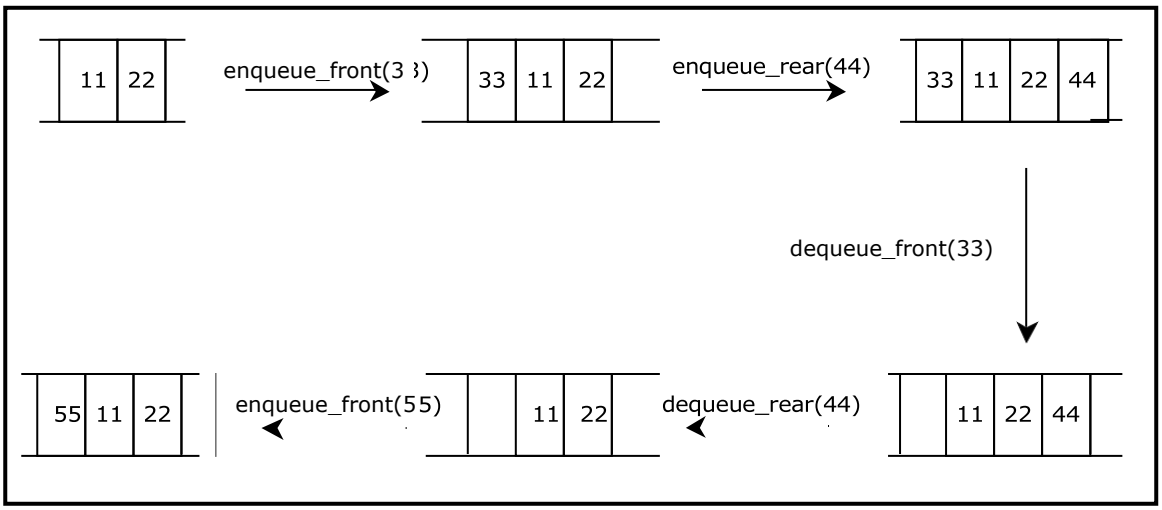


Figure 4.6. Basic operations on deque

There are two variations of deque. They are:

- Input restricted deque (IRD)
- Output restricted deque (ORD)

An Input restricted deque is a deque, which allows insertions at one end but allows deletions at both ends of the list.

An output restricted deque is a deque, which allows deletions at one end but allows insertions at both ends of the list.

4.10. Priority Queue:

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. two elements with same priority are processed according to the order in which they were added to the queue.

A prototype of a priority queue is time sharing system: programs of high priority are processed first, and programs with the same priority form a standard queue. An efficient implementation for the Priority Queue is to use heap, which in turn can be used for sorting purpose called heap sort.