Data Structures

Lecture 6 :

The Queues

Assist. Prof. Dr Abdul Hadi Mohammed

The Queues

A **queue** is logically a **first in first out** (**FIFO** or first come first serve) linear data structure.

The concept of **queue** can be understood by our real life problems. For example a customer come and join in a queue to take the train ticket at the end (rear) and the ticket is issued from the front to end of queue. That is, the customer who arrived first will receive the ticket first. It means the customers are serviced in the order in which they arrive at the service centre.

The Queues

It is a homogeneous collection of elements in which new elements are added at one end called REAR, and the existing elements are deleted from other end called FRONT. The basic operations are:

1. Insert (or add) an element to the queue (push)

2. Delete (or remove) an element from a queue (pop).

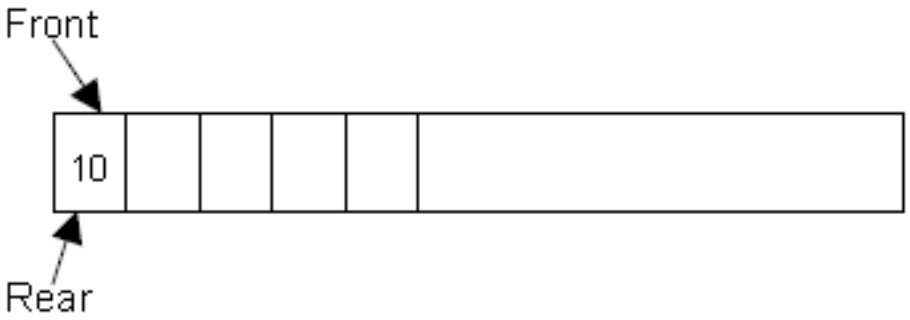
Push operation will insert an element to queue, at the rear end, by incrementing the array index. **Pop** operation will delete from the front end by decrementing the array index and will assign the deleted value to a variable.

The Queues

Following figure will illustrate the basic operations on queue.

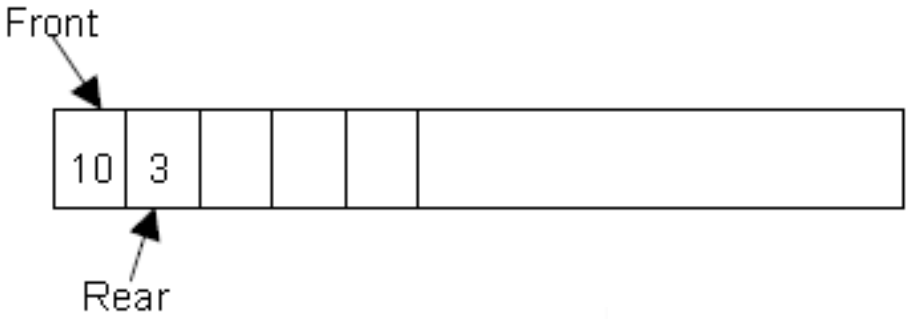
Rear = –1
Front = –1

Queue is empty.



Rear = 0
Front = 0

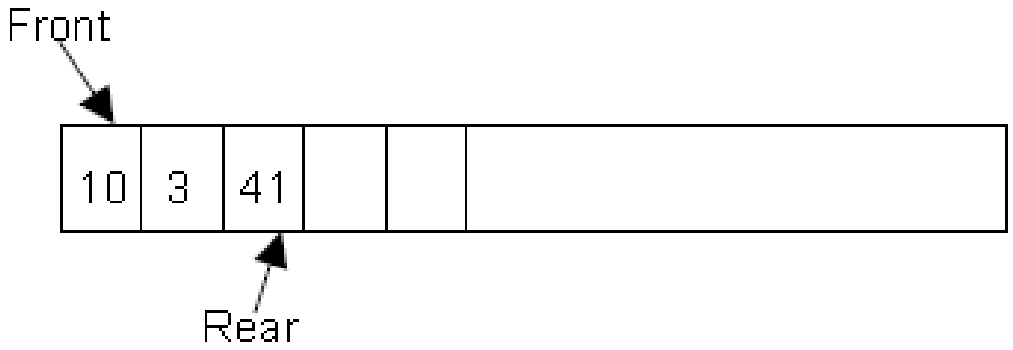
push(10)



Rear = 1
Front = 0

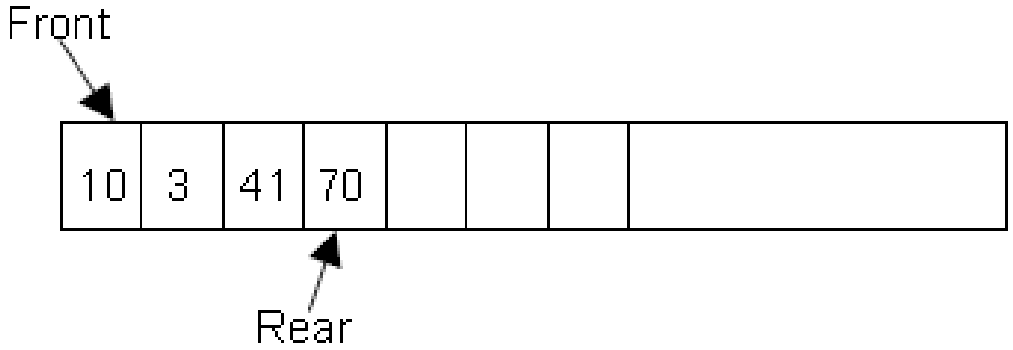
push(3)

The Queues



Rear = 2
Front = 0

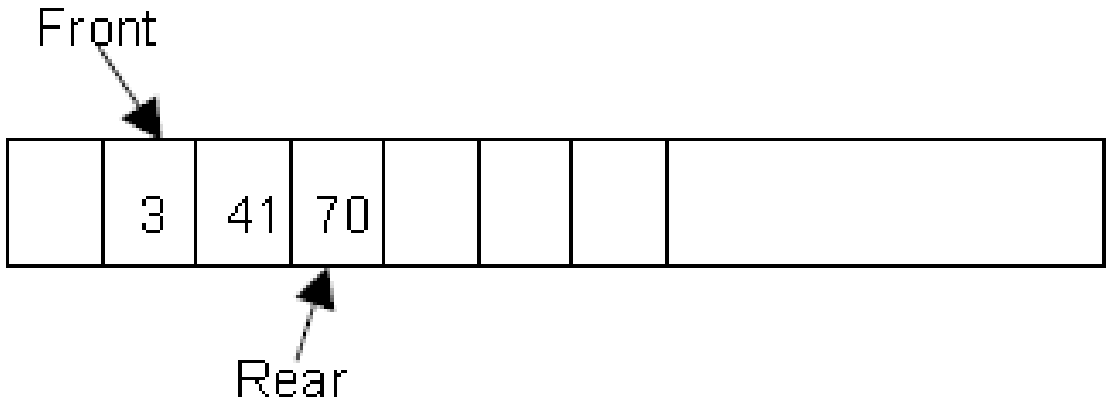
push(41)



Rear = 3
Front = 0

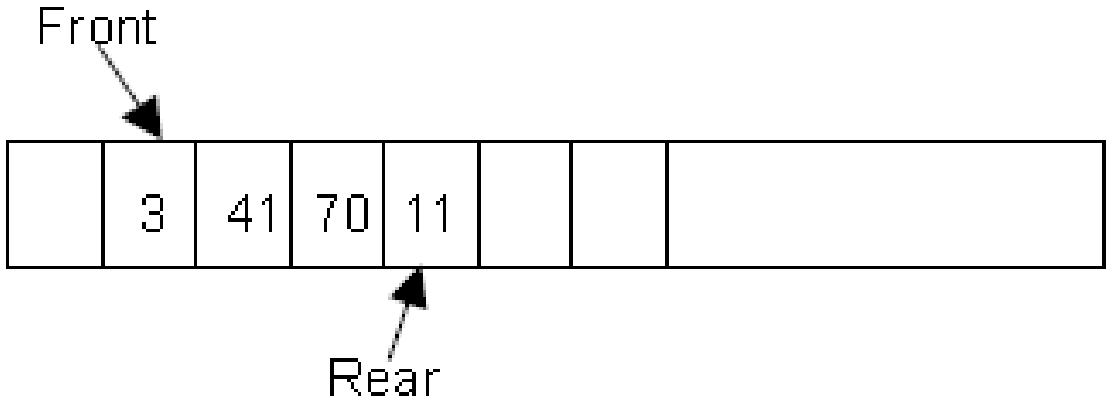
push(70)

The Queues



Rear = 3
Front = 1

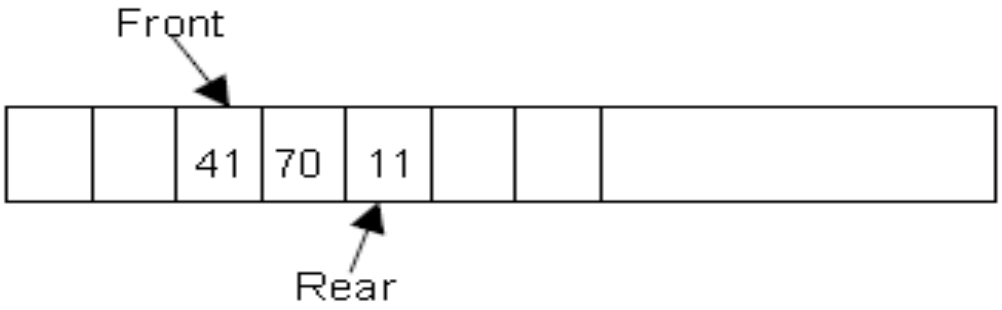
x = pop() (i.e., x = 10)



Rear = 4
Front = 1

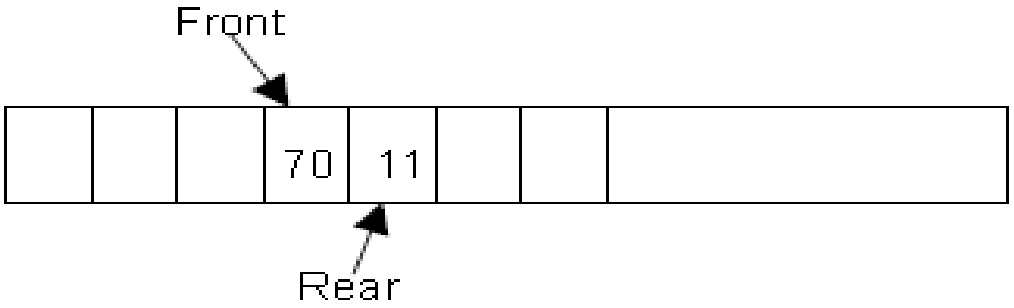
push(11)

The Queues



Rear = 4
Front = 2

x = pop() (i.e.; x = 3)



Rear = 4
Front = 3

x = pop() (i.e., x = 4 1)

Total number of elements present in the queue

is **front-rear+1**, when implemented using arrays, **right?**

The Queues

Queue can be implemented in two ways:

1. Using arrays (static)

2. Using pointers (dynamic)

Let us discuss underflow and overflow conditions when a queue is implemented using arrays**. If we try to pop an element from queue when it is empty, underflow occurs.** It is not possible to delete any element when there is no element in the queue.

Suppose maximum size of the queue is 50**. If we try to push an element to queue, overflow occurs.** When queue is full it is naturally not possible to insert any more elements

6.1. ALGORITHM FOR QUEUE OPERATIONS

Let Q be the array of some specified size say SIZE

The Queues

6.1.1. INSERTING AN ELEMENT INTO THE QUEUE

1. Initialize front = –1 rear = –1 // front = 0 rear = –1
2. Input the value to be inserted and assign to variable “data”
3. If (rear >= SIZE – 1)
    (a) Display “Queue Overflow”
    (b) Exit
4. Else
    (a) rear = rear +1
    (b) Q[rear] = data
5. Exit

The Queues

6.1.2. DELETING AN ELEMENT FROM QUEUE

1. If (front >= rear) // front = -1

    (a) front = 0, rear = –1

    (b) Display “The queue is empty”

    (c) Exit

2. Else

    (a) data = Q[front]

    (b) front = front +1

3. Exit

1. If (front = -1)

(a) Display “The queue is empty”

(b) Exit

2. Else

(a) data = Q[front]

(b) front = front +1

3. Exit

The Queues

//PROGRAM TO IMPLEMENT QUEUE USING:

1- ARRAYS

2- Pointers

Assignment within Lab

The Queues

6.2. OTHER QUEUES

There are three major variations in a simple queue. They are

1. **Circular Queue**
2. **Double Ended Queue (DE-Queue)**
3. **Priority Queue**

Priority queue is generally implemented using linked list, which is discussed in Ch 5 sec 13. The other two queue variations are discussed in the following sections.

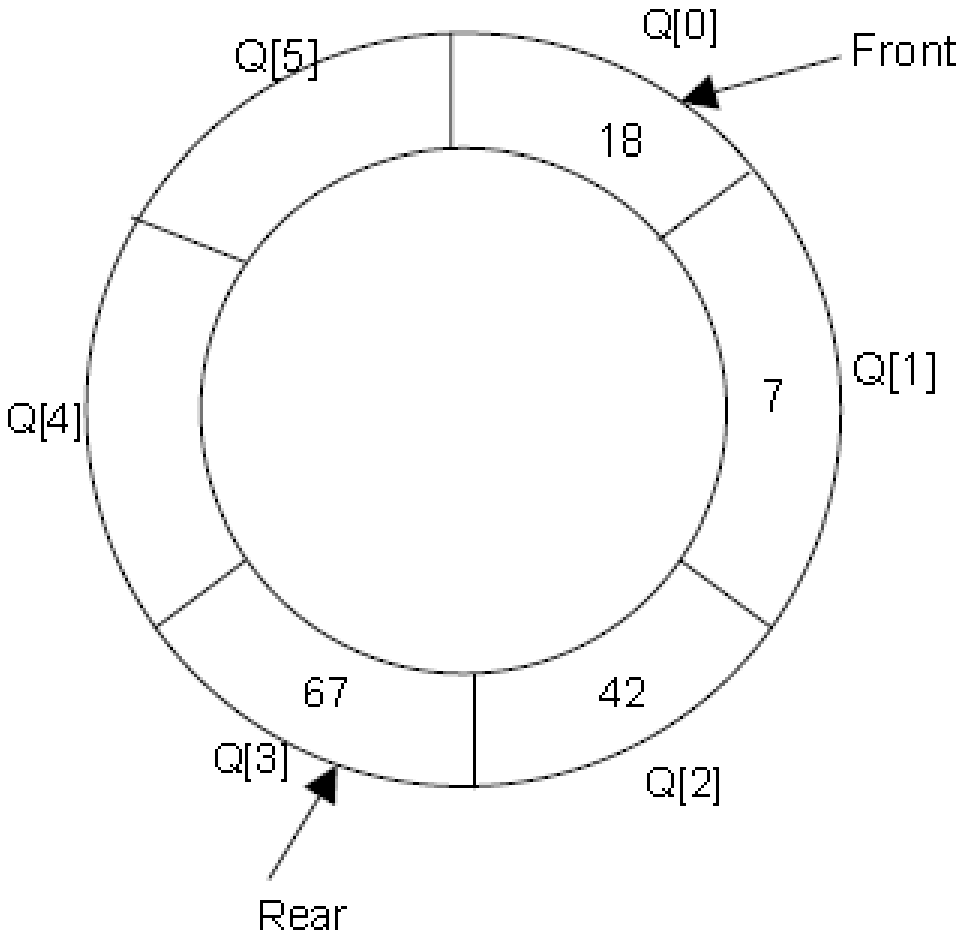
The Queues

6.3. CIRCULAR QUEUE

In circular queues the elements Q[0],Q[1],Q[2] .... Q[n – 1] is represented in a circular fashion with Q[1] following Q[n]. **A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location at the queue is full.**

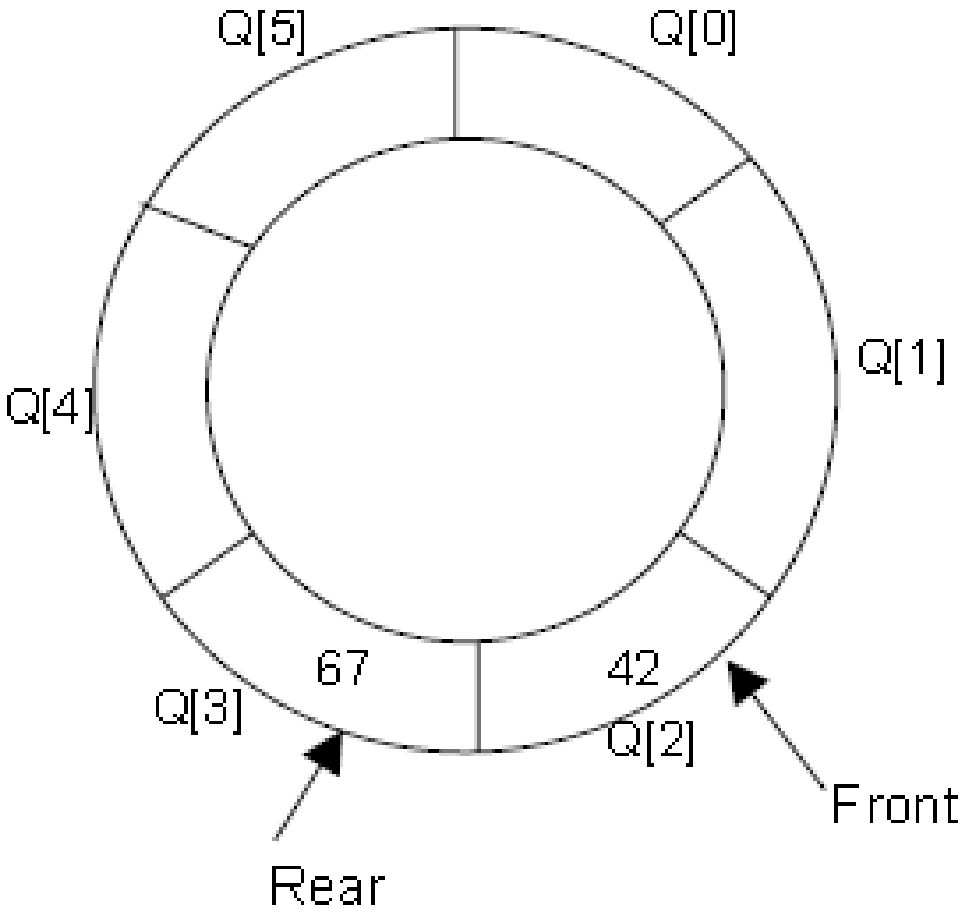
Suppose Q is a queue array of 6 elements. Push and Pop operation can be performed on circular. The following figures will illustrate.

The Queues



A circular queue after inserting 18,7,42,67,

The Queues

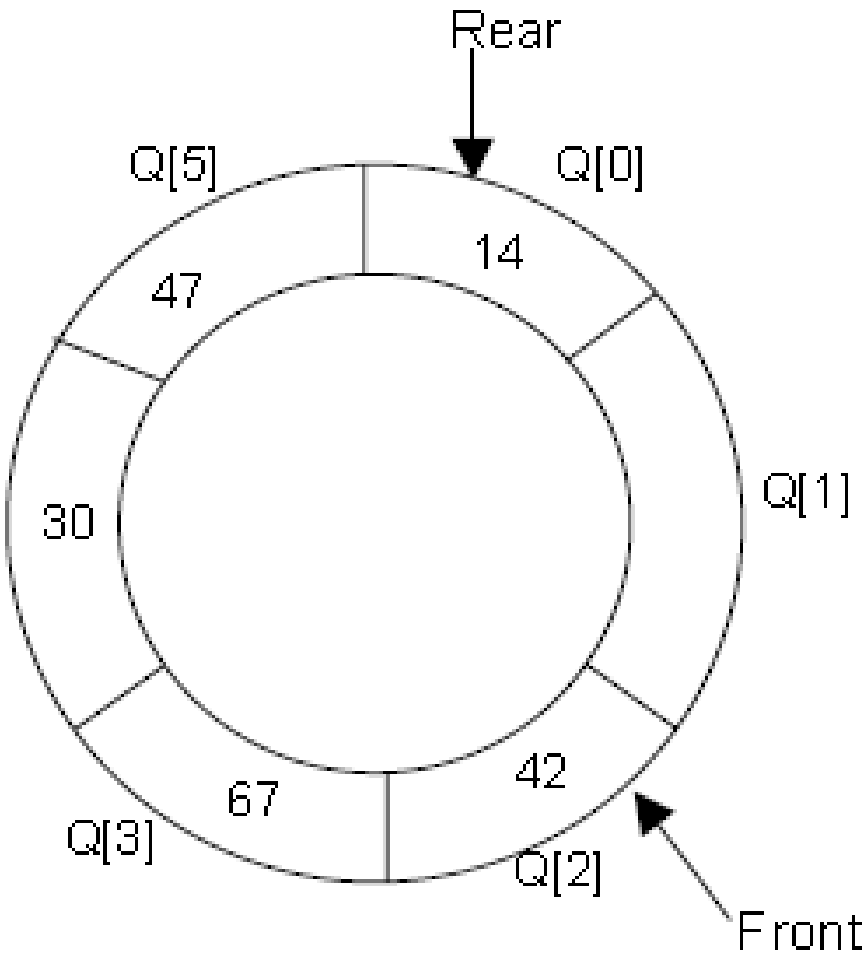


A circular queue after popping 18, 7

The Queues

After inserting an element at last location Q[5], the next element will be inserted at the very first location (i.e., Q[0]) **that is circular queue is one in which the first element comes just after the last element.**

The Queues



A circular queue after pushing 30,47,14

The Queues

At any time the **position** of the element to be **inserted** will be calculated by the relation **Rear = (Rear + 1) % SIZE** After **deleting** an element from circular queue the position of the front end is calculated by the relation **Front= (Front + 1) % SIZE** After locating the position of the new element to be inserted, rear, compare it with front.

The Queues

Algorithm for Inserting an element to circular Queue

1. **Initialize front = – 1; rear = – 1**

2. rear = (rear + 1) % SIZE

3. If (rear is equal to front) **// or (front = rear + 1)**

(a) Display “Queue is full” (b) Exit

4. Else

(a) Input the value to be inserted and assign to variable “data”

5. If (front is equal to – 1)

(i) front = 0 **(ii) rear = 0**

6. Q[rear] = data

7. Repeat steps **2 to 6** if we want to insert more elements

8. Exit

The Queues

Algorithm for Deleting an element from a circular queue

1. If (front is equal to – 1)

(a) Display “Queue is empty” (b) Exit

2. Else

(a) data = Q[front]

3. If (front is equal to rear)

(a) front = –1 **(b) rear = –1**

4. Else

(a) front = (front +1) % SIZE

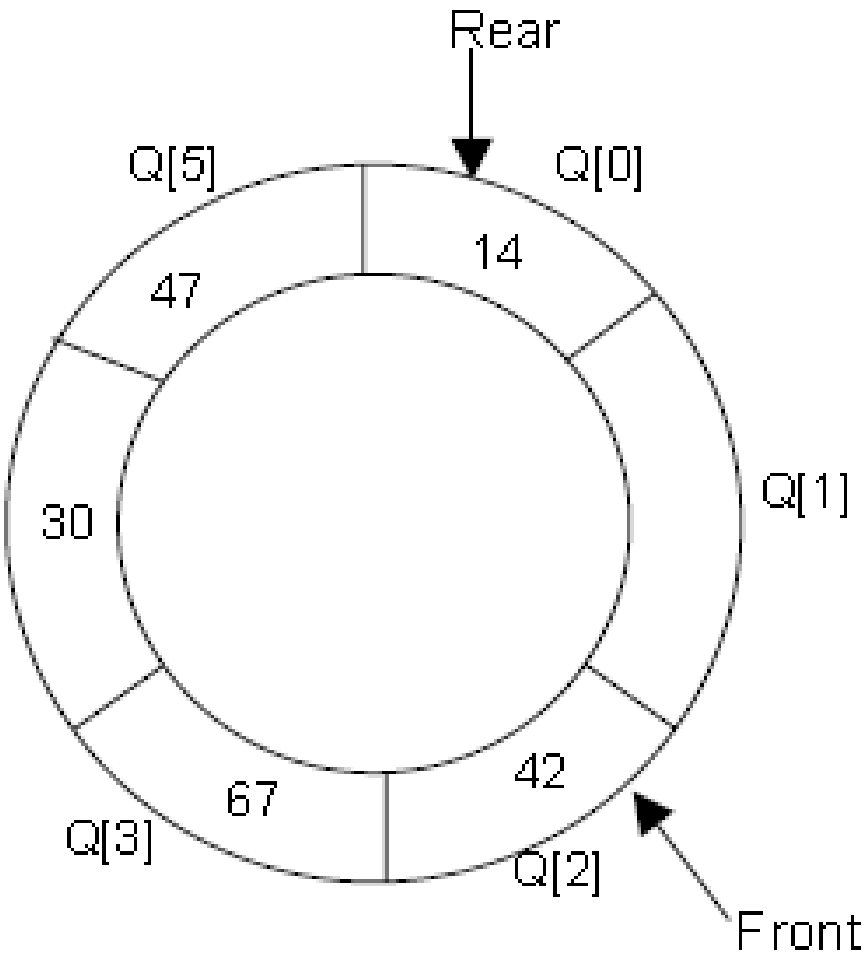
5. Repeat the steps **1 to 4** if we want to delete more elements

6. Exit

The Queues

```
void circular_queue::insert()
{
        int data;
        //Checking for overflow condition
        if ((front == 0 && rear == MAX-1) || (front == rear + 1))
        {
                cout<<"\nQueue Overflow \n";
                return;
        }
        if (front == –1) /*If queue is empty */
        {
                front = 0;
                rear = 0;
        }
        else
                if (rear == MAX-1) /*rear is at last position of queue */
                        rear = 0;
                else
                        rear = rear + 1;
        cout<<"\nInput the element for insertion in queue:";
        cin>> data;
        cqueue_arr[rear] = data;
}/*End of insert()*/
```

The Queues



A circular queue after pushing 30, 47, 14

The Queues

```
void circular_queue::del()
{
        //Checking for queue underflow
        if (front == –1)
        {
                cout<<“\nQueue Underflow\n”;
                return;
        }
        cout<<“\n Element deleted from queue is:”<<cqueue_arr[front]<<“\n”;
        if (front == rear)
        {
                front = –1;
                rear = –1;
        }
        else
                if(front == MAX-1)
                        front = 0;
                else
                        front = front + 1;
}/*End of del()*/
```

The Queues

// PROGRAM TO IMPLEMENT CIRCULAR

QUEUE USING ARRAY

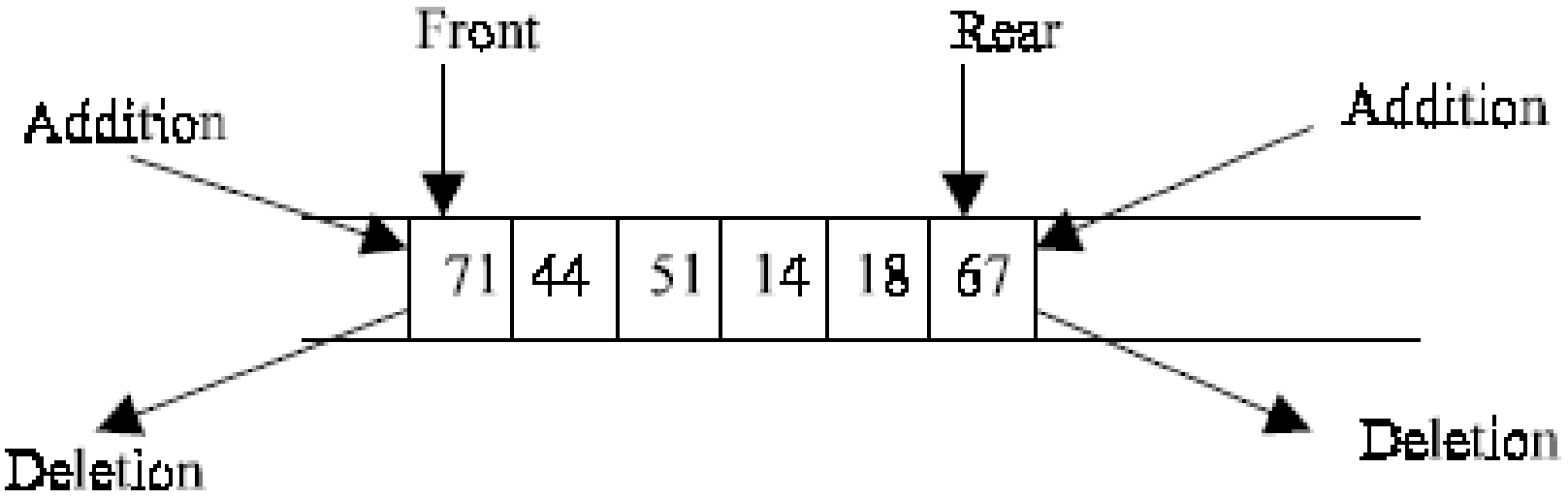
within the lab

The Queues

6.4. DE-QUEUES

A **dequeue** is a homogeneous list in which elements can be added or inserted (called push operation) and deleted or removed from both the ends (which is called pop operation), i. e; **we can add a new element at the rear or front end** and also **we can remove an element from both front and rear end**. Hence it is called **Double Ended Queue**.

The Queues



A dequeue

The Queues

There are two types of deque depending upon the restriction to perform **insertion or deletion** operations at the two ends.

They are:

1. Input restricted de-queue

2. Output restricted de-queue

The Queues

An **input restricted**, which allows insertion at only 1 end, rear end, but allows deletion at both ends, rear and front end of the lists.

An **output-restricted**, which allows deletion at only one end, front end, but allows insertion at both ends, rear and front ends of the lists.

The possible operation performed on deque is

1. Add an element at the rear end
2. Add an element at the front end
3. Delete an element from the front end
4. Delete an element from the rear end

Only 1st, 3rd and 4th operations are performed by **input-restricted deque** and 1st, 2nd and 3rd operations are performed by **output-restricted deque.**

The Queues

6.4.1. ALGORITHMS FOR INSERTING AN ELEMENT

INSERT AN ELEMENT AT THE RIGHT SIDE OF THE DE-QUEUE

1. Input the DATA to be inserted
2. **If ((left == 0 && right == MAX–1) || (left == right + 1))**
    (a) Display “Queue Overflow” (b) Exit
3. If (left == –1)
    (a) left = 0 (b) right = 0
4. Else
    (a) if (right == MAX –1)
        (i) right= 0
    (b) else
        (i) right = right+1
5. Q[right] = DATA
6. Exit

The Queues

INSERT AN ELEMENT AT THE LEFT SIDE OF THE DE-QUEUE

1. Input the DATA to be inserted
2. **If ((left == 0 && right == MAX–1) || (left == right+1))**
    (a) Display “Queue Overflow” (b) Exit
3. If (left == – 1)
    (a) left = 0 (b) right = 0
4. Else
    (a) if (left == 0)
        (i) left = MAX – 1
    (b) else
        (i) left = left – 1
5. Q[left] = DATA
6. Exit

The Queues

6.4.2. ALGORITHMS FOR DELETING AN ELEMENT

DELETE AN ELEMENT FROM THE RIGHT SIDE OF THE DE-QUEUE

1. If (left == – 1)

(a) Display “Queue Underflow” (b) Exit

2. DATA = Q [right]

3. If (left == right)

(a) left = – 1 (b) right = – 1

4. Else

(a) if(right == 0)

(i) right = MAX-1

(b) else

(i) right = right-1

5. Exit

The Queues

DELETE AN ELEMENT FROM THE LEFT SIDE OF THE DE-QUEUE

1. If (left == – 1)

(a) Display “Queue Underflow” (b) Exit

2. DATA = Q [left]

3. If(left == right)

(a) left = – 1 (b) right = – 1

4. Else

(a) if (left == MAX-1)

(i) left = 0

(b) Else

(i) left = left +1

5. Exit

6.5. APPLICATIONS OF QUEUE

1. Round robin techniques for processor scheduling.

2. Printer server routines (in drivers) are designed using queues.

3. All types of customer service software (like Railway/Air ticket reservation).