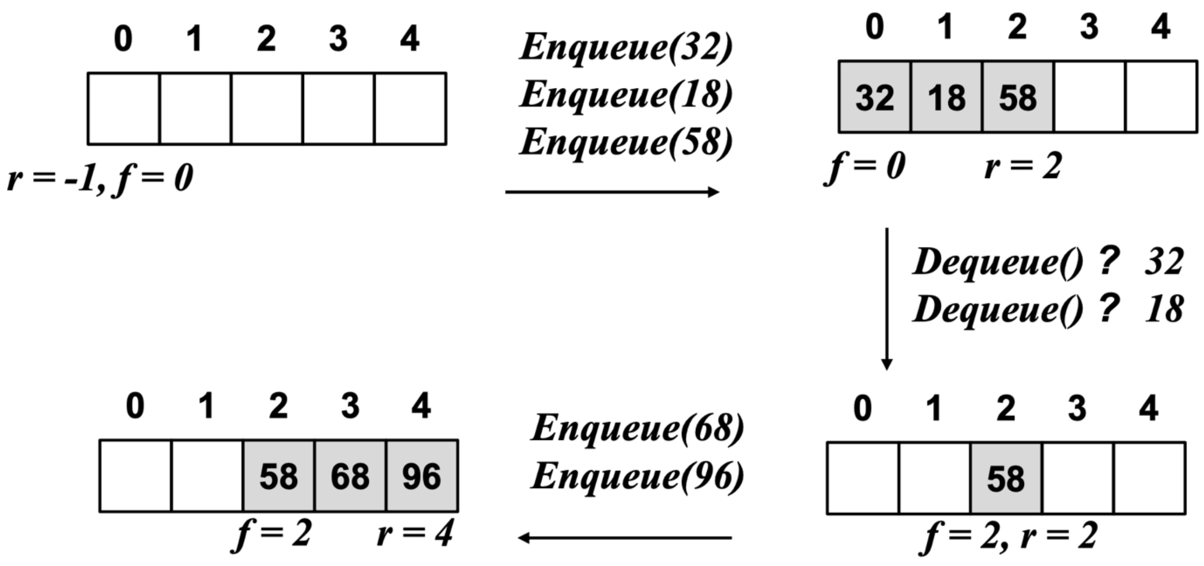
Unit 4: Queues

- The queue is a linear data structure that follows FIFO (First-In-First-Out) order to perform operations.
- In queue, enqueue (insertion) and dequeue (deletion) operations are performed at separate ends known as rear and front.
- enqueue (insertion) operation is performed at the rear end.
- dequeue (deletion) operation is performed from the front end.

The Queue ADT:

A queue of elements of type T is a finite sequence of elements of T together with the operations:

• makeEmpty(Q)	: Create an empty queue Q
• isEmpty(Q)	: Determine if the queue Q is empty or not
• enqueue (Q, x)	: Insert element x at the rear of the queue Q
• dequeue(Q)	: If the queue Q is not empty, remove the element at the front of the queue
• front(Q)	: Retrieve the element at the front of the queue Q, without deleting it



A queue with an array of size 5

Array Implementation of Linear Queue (Static Memory Allocation)

In the array implementation of linear queue, we take an array of fixed size and two variables front and rear. These two variables contain the current index at which insertion and deletion can be performed. We use one dimensional array. Hence the size is static. The queue is empty if the front == rear and full if front == 0 and rear == n.
Deletion from the empty queue causes an underflow, while insertion onto a full queue causes an overflow.

Enqueue Operation:

- The operation of adding new element to the end of the queue is known as enqueue.
- Whenever a new item is added (enqueued) to the queue, rear pointer is used.
- During enqueue operation rear is incremented by 1 and data is stored in the queue at that location indicated by rear.

Method to enqueue an element in a linear queue:

```
void enqueue (int queue [], int rear, int size, int data)
{
        if (isFull())
                System.out.println("Overflow");
        else
        {
                rear = rear + 1;
                queue[rear] = data;
                size++;
        }
}
```

Dequeue Operation:

- The operation of removing an element from the front of the queue is known as dequeue.
- Whenever an item is deleted from the queue, The front pointer is used.
- During dequeue operation, front pointer is incremented by 1 and the deleted item is returned.

Method to dequeue an element from a linear queue:

```
void dequeue (int queue [], int front, int size, int data)
{
        if (rear < front)
                System.out.println("Underflow");
        else
        {
                data = queue[front];
                front = front + 1;
                size--;
        }
}
```

Linked List Implementation of Linear Queue (Dynamic Memory Allocation)

A linear queue can be implemented using linked list. The structure of the node remains same as that of stack but we have two pointers front and rear. The front pointer points to the stating of the queue (first item inserted) and rear points to the end of the queue (last item inserted). The structure of the node is as given below:

```
class Node
{
        protected int data; protected Node next;
        public Node ()
        {
                next = null;
                data = 0;
        }
        public Node (int d)
        {
                data = d;
                next = null;
        }
}
```

Function to Enqueue an element to A Linear Queue:

```
void enqueue (int el)
{
        Node abc = new Node (el);
        abc.next= null;
        if (rear < front)
                rear = front = abc;
        else
        {
                rear.next = abc;
                rear = abc;
        }
}
```

<u>Function to Dequeue an element from a Linear Queue:</u>

```
Void dequeue ()
{
         Node temp;
        if (rear < front)
                System.out.println("Underflow occurs");
        else if (front.next == null)
        {
                rear = front = NULL;
        }
        else
        {
                temp = front;
                front = front.next;
                temp.next = null;
        }
}
```

Circular Queue:

Suppose a queue Q has maximum size 5, say 5 elements pushed and 2 elements popped.

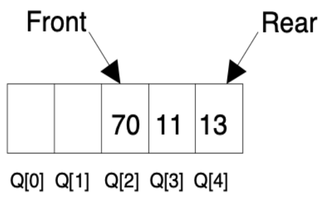
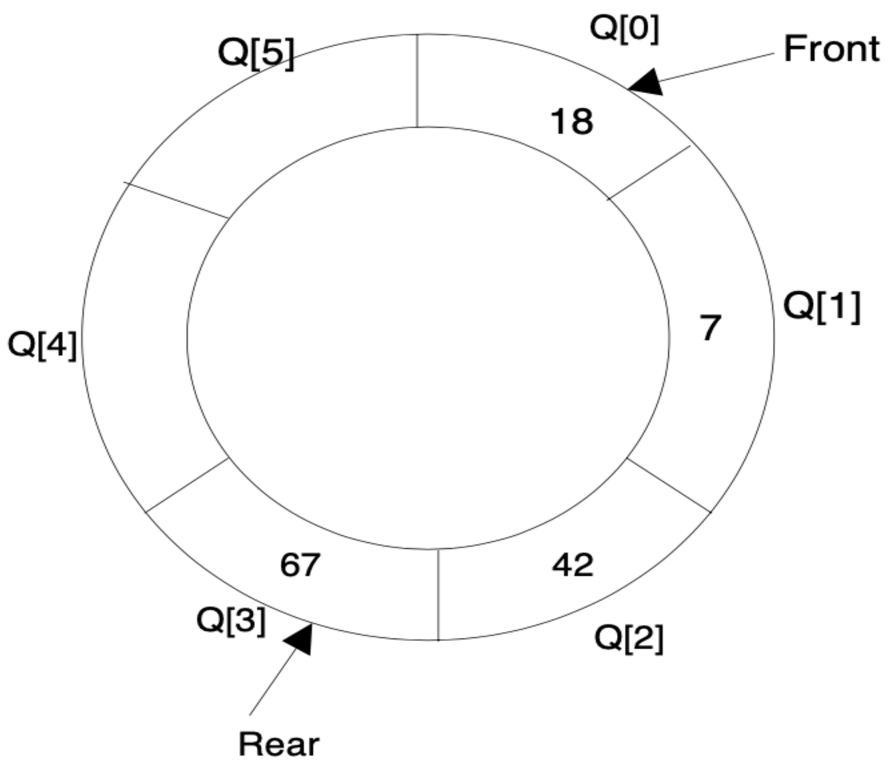


Fig. 4.10

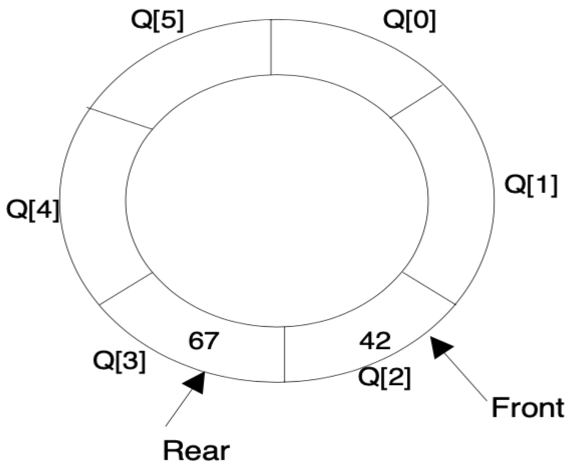
Now if we attempt to add more elements, even though 2 queue cells are free, the elements cannot be pushed. Because in a queue, elements are always inserted at the *rear* end and hence *rear* points to last location of the queue array Q[4]. That is queue is full (overflow condition) though it is empty. This limitation can be overcome if we use 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. A Circular queue After inserting 18, 7, 42, 67:

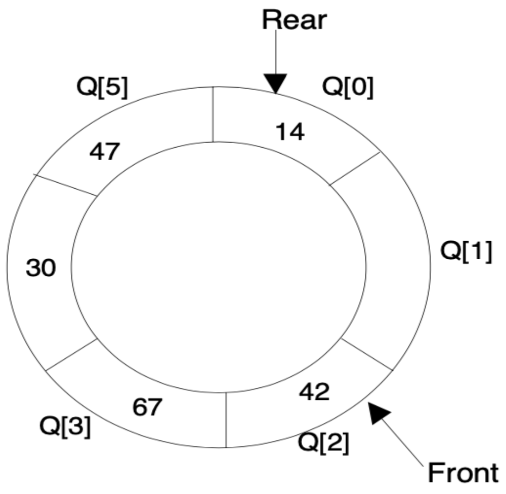


A circular queue after dequeue 18, 7:



A circular queue after enqueue 30, 47, 14:

- 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.



- 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:
  Front= (Front + 1) % SIZE

- In circular queue, we sacrifice one element of the array thus to insert n elements in a circular queue we need an array of size n+1.

<u>Function to Enqueue an element to A Circular Queue:</u>

```
public void enqueue (int item)
{
        if (isFull())
        {
                System.out.println("Overflow occurs.");
                return;
        }
        else
        {
                rear = (rear + 1) % size;
                queue[rear] = item;
                count++;
        }
}
```

<u>Function to Dequeue an element from a Circular Queue:</u>

```
int dequeue()
{
        if (isEmpty())
        {
                System.out.println("Queue is empty.");
                return -1;
        }
        else
        {
                int item = queue[front];
                front = (front + 1) % size;
                count--;
                return item;
        }
}
```

Priority Queues

In many situations, simple queues are inadequate, as when first in /first out scheduling has to be overruled using some priority criteria. In a post office example, a handicapped person may have priority over others. Therefore, when a clear is available, a handicapped person is served instead of someone from the front of the queue.
In a sequence of processes, process p2 may need to be executed before process p1 for the proper functioning of the system, even though p1 was put on the queue of waiting processes before p2. In situations like these, a modified queue or priority queue is needed. In priority queues, elements are dequeued according to their priority and current position.
The problem with a priority queue is in finding an efficient implementation that allows relatively fast enquiring. Because elements may arrive randomly to the queue, there is no guarantee that the front element will be most likely to be dequeued and that the elements put at the end will be the most likely to be dequeued. The situation is complicated because a wide spectrum of possible priority criteria can be used in different cases such as frequency of use, birthday, salary, position, status, and others.
Priority queues can be represented by two variations of linked lists. In one type of linked list, all elements are entry ordered, and in another, order is maintained by putting a new element in its proper position according to its priority. In both cases, the total operational times are O(n) because, for an unordered list, adding an element is immediate but searching is O(n), and in a sorted list, taking an element is immediate but adding an element is O(n).

Priority Queue is a queue where each element is assigned a priority. In priority queue, the elements are deleted and processed by following rules:
1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were inserted to the queue.

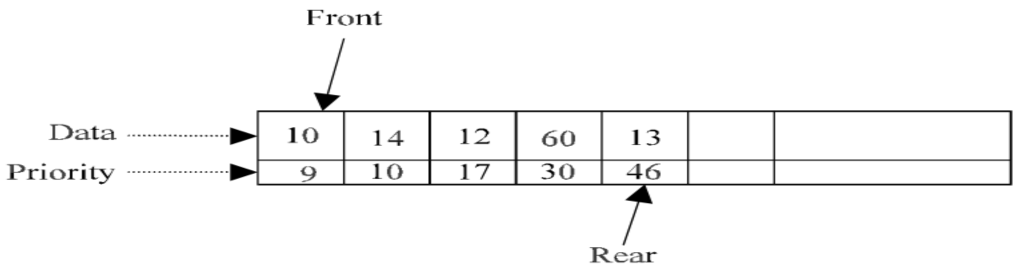
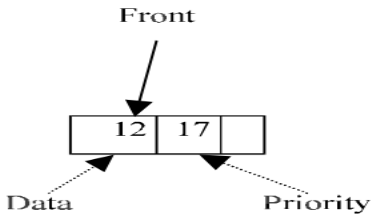


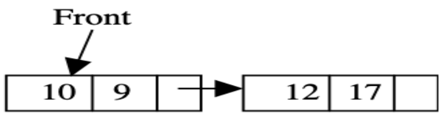
Figure: Priority queue representation using array

- A node in the priority queue will contain DATA, PRIORITY and NEXT field.
- DATA field will store the actual information; PRIORITY field will store its corresponding priority of the DATA and NEXT will store the address of the next node.
- Always we may not be pushing the data in an ascending order. From the mixed priority list, it is difficult to find the highest priority element if the priority queue is implemented using arrays.
- So, it is always better to implement the priority queue using linked list - where a node can be inserted at anywhere in the list

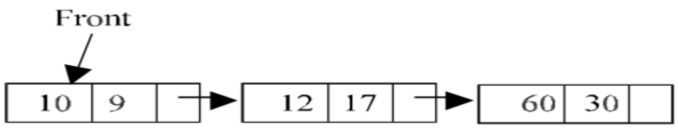
Linked List representation of Priority Queue:



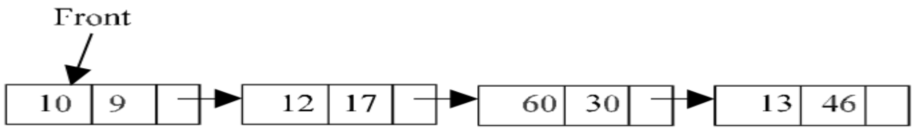
Push data = 10 Priority = 9:



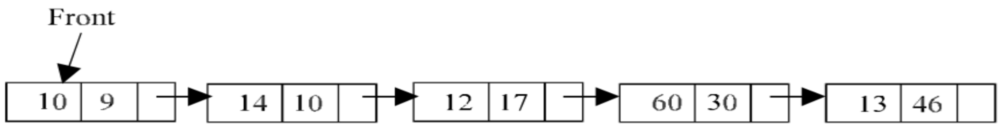
Push data = 60 Priority = 30:



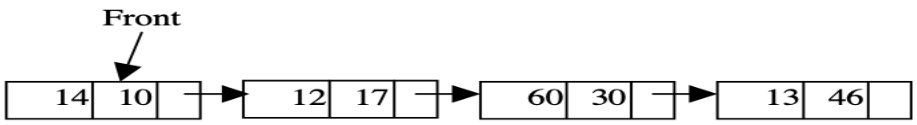
Push data = 13 Priority = 46:



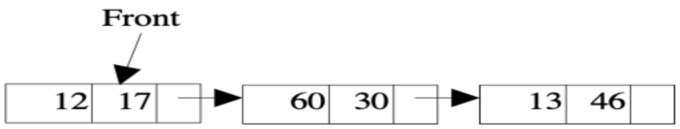
Push data = 14 Priority = 10:



Dequeue():



Dequeue():



Node class of a Priority Queue

```
class Node
{
        int data, priority;
        Node next;
        Node ()
        {
                next = null;
        }
        Node (int el, int p)
        {
                data = el;
                priority = p;
        }
        Node (int el, int p, Node n)
        {
                data = el;
                priority = p;
                next = n;
        }
}
```

Method to enqueue an element to a Priority Queue:

```
void enqueue (int el, int el_priority)
{
        Node newNode = new Node(el);
        if (front == null || el_priority < front.priority)
        {
                newNode.next = front;
                front = newNode;
        }
        else
        {
                Node tmp = front;
                while (tmp.next != NULL && tmp.next.priority  <=  el_priority)
                        tmp = tmp.next;
                newNode.next = tmp.next;
                tmp.next = newNode;
        }
}
```

Method to dequeue an element from a Priority Queue:

```
void dequeue ()
{
        Node tmp;
        if(front == NULL)
                System.out.println("\nQueue Underflow\n");
        else
        {
                tmp = front;
                front = front.next;
                temp.next = null;
        }
}
```