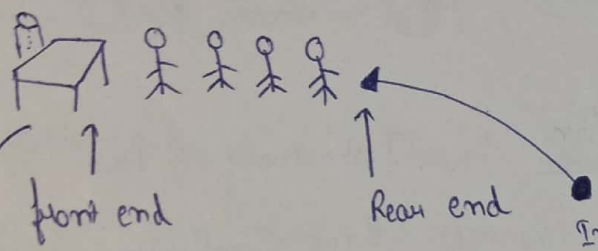
QUEUES → logical Data structure.

QueQue ADT :- It works on FIFO. (First In First Out).



In stack :-
We are inserting & deleting from same end.

Eg :-
1. Cars standing in a toll part.
2. Sometimes during our phone calls, Signal lines are busy and we get a message that you are in queue.
3. During Company calls, they said please wait, you are in queue.

Queue ADT :-

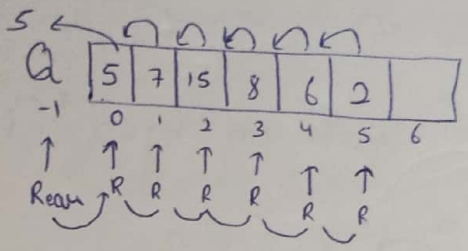
Data :-
1. Space for storing elements
2. Front (Pointer) → for deletion
3. Rear (Pointer) → for Insertion

Operations :-
1. EnQueue (x) → inserting
2. deQueue() → deleting
3. isEmpty()
4. isFull()
5. First() → first element
6. last() → last element

→ It can also be implemented using Array as well as Linked list.

★ Queue Using Single Pointer :- Using Array

Size = 7



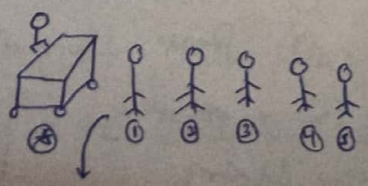
For Insertion :- Just move "Rear" Pointer and fill the value.

Time Complexity :- O(1)

For Deletion :- For Deletion, Let us suppose we take out value '5' then, this space will be blanked, so to avoid this we have to shift all the elements and move "Rear" Pointer also one step back.

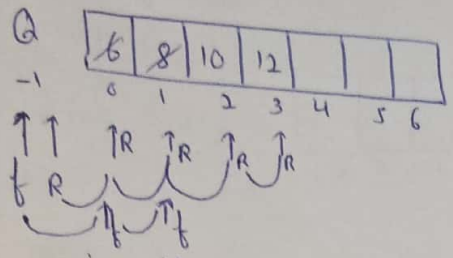
Time Complexity :- O(n) → bcz as many elements are there, we have to shift them all.

★ Queue Using Double Pointer :- Using Array



When 1st person goes out and Now to fill the vacant space we were shifting all persons one step ahead, which takes time O(n), but instead of this we will shift * only to fill vacant space & we can do this using two pointers.

Size = 7



Initially :- front = Rear = -1

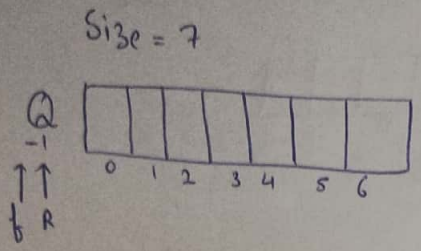
Now, Let us take two pointers "front" and "Rear", Now Simply move "Rear" pointer for insertion and insert the element, and this will take Constant time (i.e. O(1)). Now for Deletion, we will use "front" Pointer, Now move front pointer and then delete that element, it will also take Constant time (i.e. O(1)).

So, both enQueue & deQueue takes O(1) using two pointers.

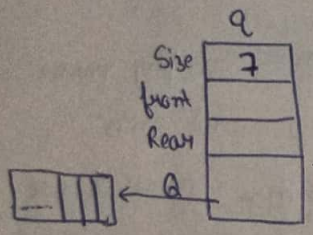
- Empty Condition :- if (front == Rear)
- Full Condition :- if (Rear == Size-1)

Note :- only for understanding :- We always take "front" Pointer before the first element, So, Now let us assume that "front" is pointing on first element and we will delete it and move "front", Now the "front" is pointing on "Rear" it means (front == Rear) but the Queue is not Empty, So the empty Condition fails, Hence we take "front" Pointer before 1st element.

★ Implementing Queue Using Array :-




```
Struct Queue
{
    int size;
    int front;  // It is an address
    int Rear;   pointer / index pointer.
    int *Q;
};
```




```
int main ()
{
    Struct Queue q;
    printf ("Enter the Size of Queue: \n");
    Scanf ("%d", &q.size);
    q.Q = (int *) malloc (q.size * sizeof (int));
    q.front = q.Rear = -1;
}
```

Now,

```
Void enQueue (Queue *q, int x)
{
    if (q->Rear == q->size - 1)
        printf ("Queue is Full");
    else
    {
        q->Rear ++;
        q->Q[q->Rear] = x;
    }
}
```

```
int deQueue (Queue *q)
{
    int x = -1;
    if (q->front == q->Rear)
        printf ("Queue is Empty");
    else
    {
        q->front ++;
        x = q->Q[q->front];
    }
    return x;
}
```

:-

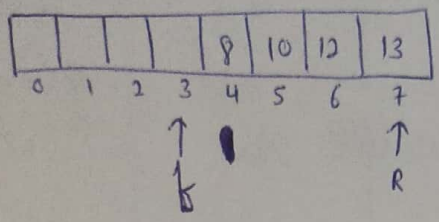
```
Struct Queue
{
    int size;
    int front;
    int Rear;
    int *Q;
};
int main ()
{
    Struct Queue q;
    printf ("Enter the size of Queue : \n");
    scanf ("%d", &q.size);
    q.Q = (int *) malloc (q.size * sizeof (int));
    q.front = q.Rear = -1;
    void enQueue (struct Queue *q, int x);
    int deQueue (struct Queue *q);
    void Display (struct Queue q);
    enQueue (&q, 15);
    enQueue (&q, 12);
    enQueue (&q, 11);
    enQueue (&q, 10);
    enQueue (&q, 150);
    Display (q);
    printf ("\n");
    printf (" First element after popping: %d \n", deQueue(&q));
    Display (q);
}
```

```
Void enQueue (struct Queue *q, int x)
{
    if (q->Rear == q->Size-1)
        printf ("Queue is full");
    else
    {
        q->Rear++;
        q->Q[q->Rear] = x;
    }
}
int deQueue (struct Queue *q)
{
    int x=-1;
    if (q->front == q->Rear)
        printf ("Queue is Empty");
    else
    {
        q->front++;
        x = q->Q[q->front];
    }
    return x;
}
Void Display (struct Queue q)
{
    int i;
    for (i=q.front+1 ; i<=q.Rear ; i++)
    {
        printf ("%d", q.Q[i]);
    }
}
```

Drawbacks of Queue Using Array :-

Drawbacks :-

(i)

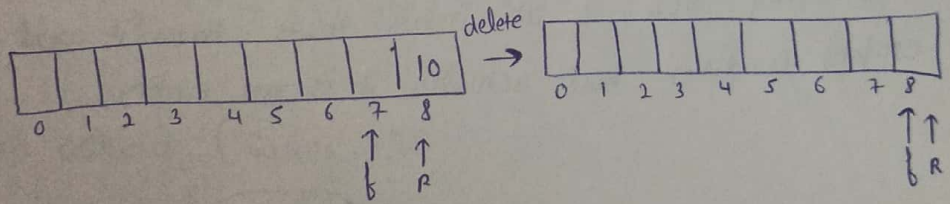


Now, if we want to insert the element in the Queue, we can't insert bcz Queue is full, but we can see that there are some vacant spaces available. but we cannot utilize those spaces bcz insertion is done from Rear end and it is at last of the Queue, but those vacant spaces are available in front end, so we can't use them

- We cannot use the space of deleted Elements.

(ii) Every location can be used only once.

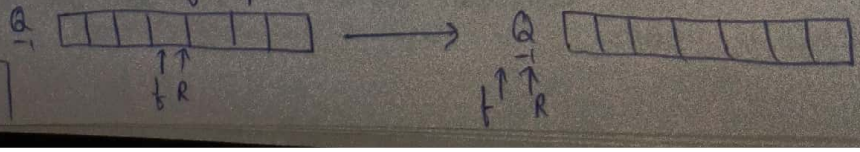
(iii)



Now, in this case, we can see that "front" is pointing on "Rear" So, Queue is empty (i.e. front == Rear), but also we can insert any element (ie. Rear == Size-1), it means Queue is full.

"A Situation where Queue is empty as well as full".

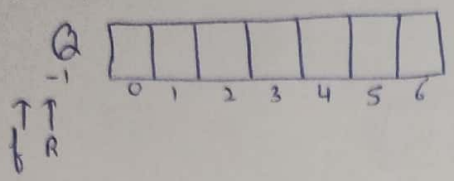
(2) Solutions :- (i) Resetting Pointers :- Whenever "f & R" pointing on same location, it means stack is empty, bring them on -1. but this method will not Assure the reusability of space.

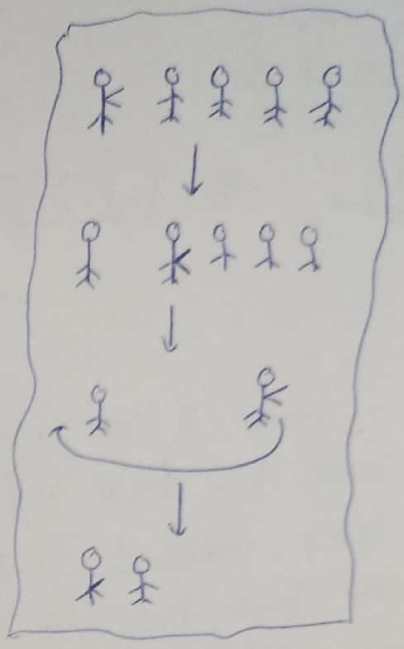


(ii) Circular Queue.

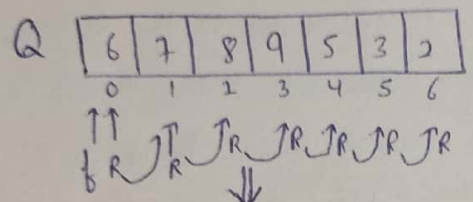
☆ Circular Queue :-

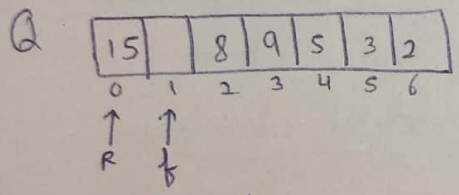
Size = 7





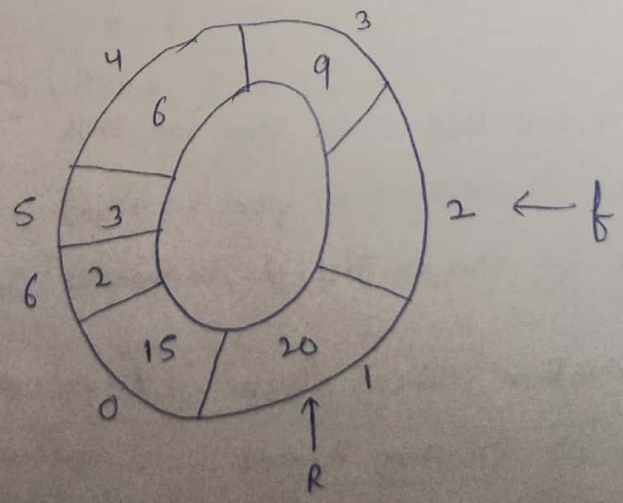
But for circular Queue, we will start from '0'.





⇓

We will not use that place for insertion where the "front" is pointing, bcz, if we insert there, then it will be a Queue empty condition, but actually it is not empty.

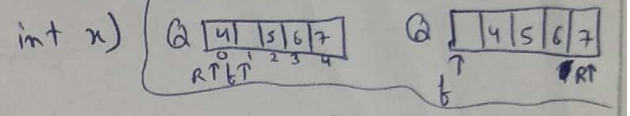


- To obtain this circular fashion, we will use mod operation.

Rear	= (Rear + 1) % Size	
0	(0+1) % 7	1
1	(1+1) % 7	2
2	(2+1) % 7	3
3	(3+1) % 7	4
4	(4+1) % 7	5
5	(5+1) % 7	6
6	(6+1) % 7	0
0		

→ it guaranteas that all the spaces will be used or we can say can be reused.

- Queue Full Condition :- if "front" is next to "Rear" than it will be stack full condition.




```
Void enQueue ( Queue *q, int x)
{
    if (q→Rear +1) % q→Size == q→front)
        printf ("Queue is full");
    else
    {
        q→Rear = (q→Rear+1) % q→Size;
        q→Q[q→Rear] = x;
    }
}
int deQueue (Queue *q)
{
    int x = -1;
    if (q→front == q→Rear)
        printf ("Queue is Empty");
    else
    {
        q→front = (q→front +1) % q→Size;
        x = q→Q[q→front];
    }
    return x;
}
```

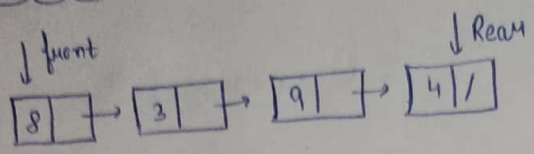
★ So, the best method for implenting a queue using Array is circular queue, so, that we can use all the spaces.

Code :-

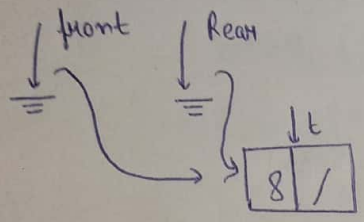
```
Struct Queue
{
    int size;
    int front;
    int Rear;
    int *Q;
};
int main()
{
    Struct Queue q;
    printf("Enter Size of Queue: \n");
    Scanf("%d", &q.size);
    q.Q = (int *) malloc(q.size * sizeof(int));
    q.front = q.Rear = 0;
    Void enqueue(struct Queue *q, int x);
    int dequeue(struct Queue *q);
    Void Display(struct Queue q);
    enqueue(&q, 15);
    enqueue(&q, 12);
    enqueue(&q, 11);
    enqueue(&q, 10);
    enqueue(&q, 150);
    Display(q);
    printf("\n");
    printf("First element after Popping : %d \n", dequeue(&q));
    Display(q);
}
```

```
void enQueue (struct Queue *q, int x)
{
  if ((q->Rear +1) % q->Size == q->front)
        printf ("Queue is Full");
  else
    {
      q->Rear = (q->Rear +1) % q->Size;
      q->Q[q->Rear] = x;
    }
}
int deQueue (struct Queue *q)
  {
    int x = -1;
    if (q->front == q->Rear)
        printf ("Queue is Empty");
    else
      {
        q->front = (q->front +1) % q->Size;
        x = q->Q [q->front];
      }
    return x;
  }
void Display (struct Queue q)
  {
    int i = q.front +1;
    do
      {
        printf("%d", q.Q[i]);
        i = (i+1) % q.Size;
      }
    while (i != (q.Rear +1) % q.Size);
  }
```

★ Queue using Linked List :-



Generally, we have only one pointer in Linked list, but here we have two pointers :- "front" and "Rear", "Rear" is used for insertion in Constant time, that's why two pointers are used.



- Empty Condition

```
if (front == NULL)
```

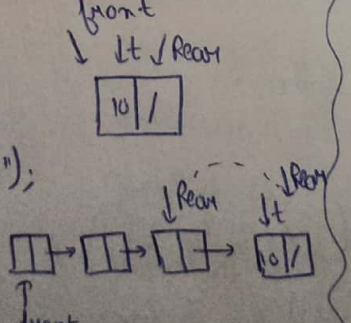
- When the first Node is created, both "front" and "Rear" will be pointing on that Node

- Full Condition

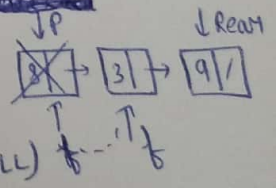
```
Node *t = new Node;
if (t == NULL)
```

it means, if new node can't be created, then Queue is full or we can say heap is full, bcz it is created in heap (i.e. Linked List)

```
void enQueue (int x)
{
  Node *t = new Node;
  if (t == NULL)
    printf ("Queue is Full");
  else
  {
    t->data = x;
    t->next = NULL;
    if (front == NULL)        // if it was very first node
      front = Rear = t;
    else
    {
      Rear->next = t;
      Rear = t;
    }
  }
}
```




```
int deQueue ()
{
  int x = -1;
  Node *p;
  if (front == NULL)
    printf ("Queue is Empty");
  else
  {
    p = front;
    front = front->next;
    x = p->data;
    free (p);
  }
  return x;
}
```



```
Struct Node
{
    int data;
    Struct Node * next;
} * front = NULL , * Rear = NULL;

int main()
{
    void enQueue (int x);
    int deQueue ();
    void Display ();
    enQueue (10);
    enQueue (20);
    enQueue (30);
    enQueue (40);
    enQueue (50);
    Display ();
    printf (" %d _", deQueue ());
}
void Display ()
{
    Struct Node *P = front;
    while(P)
    {
        printf (" %d_", P→data);
        P = P→next;
    }
    printf ("\n");
}
```

```
Void enQueue (int x)
{
    Struct Node *t;
    t= (Struct Node *) malloc (sizeof (Struct Node));
    if (t == NULL)
        printf (" Queue is full \n");
    else
    {
        t→ data = x;
        t→ next = NULL;
        if (front == NULL)
            front = Rear = t;
        else
        {
            Rear → next = t;
            Rear = t;
        }
    }
}
int deQueue ()
{
    int x = -1;
    Struct Node *t;
    if (front == NULL)
        printf (" Queue is Empty \n");
    else
    {
        x = front → data;
        t = front;
        front = front → next;
        free (t);
    }
    return x;
}
```

Double Ended Queue :- DE Queue

We can use Array as well as Linked List Here.

Now, we know we use two pointers "front" and "Rear", in which "Rear" is used for insertion and "front" is used for "deletion"

But In DE Queue, we can use both the pointers for same operation

Queue

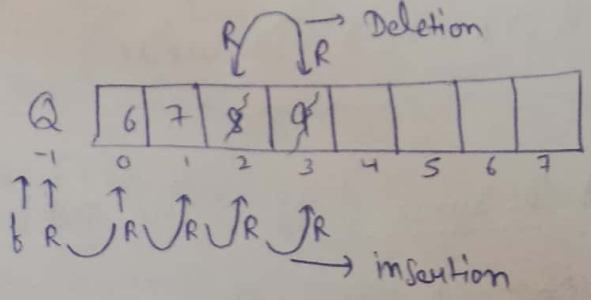
	Insert	Delete
front	✗	✓
Rear	✓	✗

DE Queue

	Insert	Delete
front	✓	✓
Rear	✓	✓

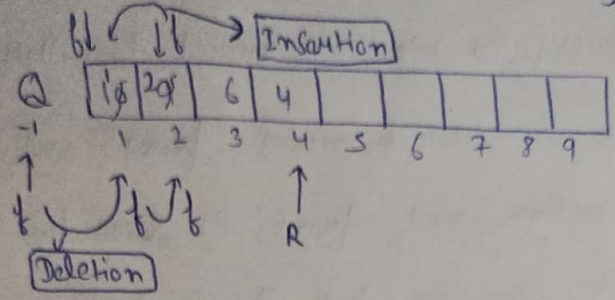
Note :- It is not working on FIFO (i.e. first in first out).

For Rear :-



For Insertion, we can simply move "Rear" pointer and increment it and put the value, and now for deletion, simply delete that value from that index and Decrement "Rear". As we already told it is not working on FIFO. So this can be possible only in DE Queue (i.e. not in Queue).

For Front :- By using "front", we can delete first, bcz there are some elements already present.



for Deletion, Simply move "front" and Delete the value and Now for insertion, Simply insert a value and then decrement "front", as it is also 'not working on FIFo.

Note :- we dont Need to write a program for this, bcz there is a (DEQueue) type of Data structure in Java, so we can implement there.

★ Two Types of Restricted DEQueue

(i) Input Restricted DEQueue (i/p)

In this Insertion can only be done by "Rear" Pointer but Deletion can be done by both.

	Insert	Delete
front	✗	✓
Rear	✓	✓

(ii) out put Restricted DEQueue(o/p)

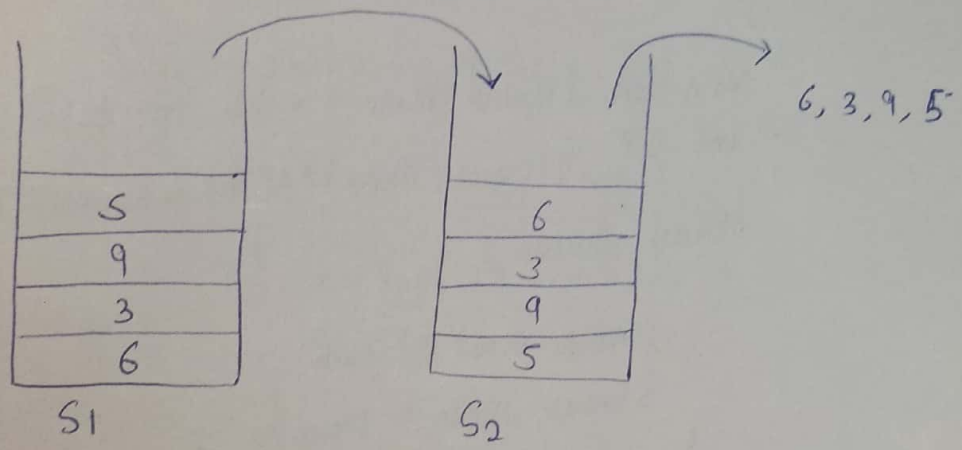
In this, Deletion can only be done by "front" Pointer but Insertion can be done by both.

	Insert	Delete
front	✓	✓
Rear	✓	✗

Queue Using 2 Stacks :-

Actually Queue is also data structure which can be implemented by using Array But we can implement it using two stacks also.

Elements → 6, 3, 9, 5, 4, 2, 8, 12, 10



Now, In this Procedure, first we will Push elements in to the stack1 and then transfer these elements to stack S2 and then PoP out the values from S2, so At the end we will see that it Works FIFO (i.e. first in first out).

Now, If Stack S2 will be empty then Simply transfer elements from S1 to S2 and then again Pop out and Continue this for all the elements.

We will use two functions enQueue and DeQueue, in which enQueue Performs Pushing into Queue and DeQueue performs transfer of elements and Popping out.

Code :- // MENU DRIVEN PROGRAM

```
Struct Node
{
  int data;
  Struct Node * next;
};
Void Push (Struct Node ** top, int data); // Forward Declaration
int PoP (Struct Node ** top); // Forward Declaration

Struct Queue
{
    Struct Node * Stack1;
    Struct Node * Stack2;
}

Void enQueue (Struct Queue *q, int x)
{
    Push (&q -> Stack1, x);
}

Void Display (Struct Node * top1, Struct Node * top2)
{
    while (top1 != NULL)
    {
      printf ("%d\n", top1 -> data);
        top1 = top1 -> next;
    }
    while (top2 != NULL)
    {
      printf ("%d\n", top2 -> data);
        top2 = top2 -> next;
    }
}
```

```
Void Dequeue (struct Stack *q)
{
    int x;
    if (q->Stack1 == NULL && q->Stack2 == NULL)
    {
        printf ("Queue is Empty");
    } return;
    if (q->Stack2 == NULL)
    {
        while (q->Stack1 != NULL)
        {
            x = POP (&q->Stack1);
            Push (&q->Stack2, x);
        }
    }
    x = POP (&q->Stack2);
    printf ("%d\n", x);
}

Void Push (struct Node **top, int data)
{
    struct Node *t = (struct Node *) malloc (sizeof (struct Node));
    if (t == NULL)
    {
        printf ("Stack overflow\n");
        return;
    }
    t->data = data;
    t->next = (*top);
    (*top) = t;
}
```

```
int Pop (struct Node **top)
{
    int x;
    struct Node *t;
    if (*top == NULL)
    {
        printf ("Stack Underflow \n");
        return;
    }
    else
    {
        t = *top;
        x = t->data;
        *top = t->next;
        free (t);
        return x;
    }
}
int main ()
{
    struct Queue *q = (struct Queue *) malloc (sizeof (struct Queue));
    int f=0, a;
    char ch = 'y';
    q->Stack1 = NULL; q->Stack2 = NULL;
    while (ch == 'y')
    {
        printf ("Enter your choice \n 1. Add to Queue \n 2. Remove from Queue
                 3. Display \n 4. Exit \n");
        scanf ("%d", &f);
        switch (f)
        {
            case 1 : printf ("Enter element to be added \n");
                     scanf ("%d", &a);
                     enque (q, a)
                     break;
            case 2 : deQueue (q);
                     break;
            case 3 : Display (q->Stack1, q->Stack2);
                     break;
            case 4 : exit (1);
                     break;
        }
    }
}
```

ADDITIONAL TOPIC

Priority Queues :-

There are two methods for implanting Priority Queues, depending upon the Situation.

1. Limited Set of Priorities
2. Element Priority.

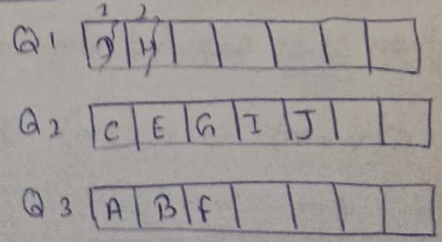
1. Limited Set of Priorities :-

This method is mainly used in operating Systems. Some operating systems allows priority based scheduling, Like in JAVA:- JVM Supports multi-Threading, so it allows priorities upon threads. so Higher priority threads will execute first. There are total 10 priorities in JAVA.

Priorities :- 3

Elements →	A	B	C	D	E	F	G	H	I	J	→ Priority 1 is maximum
Priority →	3	3	2	1	2	3	2	1	2	2	Priority 3 is minimum

Priority Queues :-



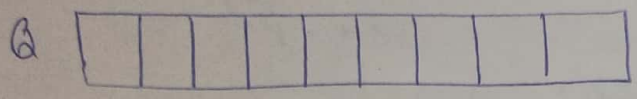
Insertion can be done on the basis of priority, that priority 1 will be added in Q1 and Soon.

Deletion can be done first in Q1, than from Q2, than from Q3 in FIFO manner (i.e. First in first out)

(2) **Element Priority** :- Element itself is a priority.

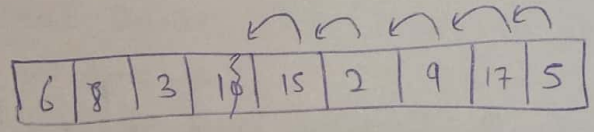
elements → 6, 8, 3, 10, 15, 2, 9, 17, 5 ... (unlimited Priorities)

Smaller the Number, Higher the Priority (Assume)



Two ways :- (i) Insert in same order
Delete max. Priority by searching it.
(ii) Insertion in Increasing order of Priority
Delete Last Element of an Array.

(i)

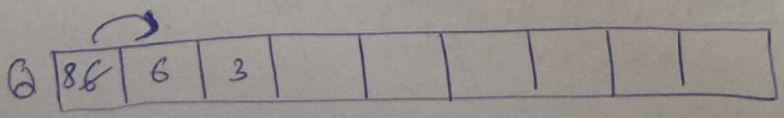


For Insertion :- Simply insert (Time Complexity :- O(1))

For Deletion :- we have to Search that element with Higher priority then shift all the elements to left side.

Time Complexity :- $O(n+n) \Rightarrow O(2n) \Rightarrow O(n)$

(ii)



For Insertion :- We have to insert an element in Sorted manner (i.e. in decreasing order of value, so that priority of last element is max.) (for Searching & Shifting)
Time Complexity :- $O(n+n) = O(2n) \Rightarrow O(n)$

For Deletion :- Simply Delete last Element (Time Complexity :- O(1))

PRACTISE SET

Ques - 1 :- Suppose a circular queue of capacity (n-1) elements is implemented with an Array of n elements. Assume that the insertion and deletion operations are carried out using REAR and FRONT as Array indeed variables, respectively. Initially, REAR = FRONT = 0; The condition to Detect full and queue Empty are

Answer :-

(i) Full : (Rear + 1) % n == front
Empty : (front + 1) % n = Rear

(ii) Full : Rear = = front
Empty : (Rear + 1) % n == front

(iii) Full : (Rear + 1) % n == front
Empty : Rear = front

(iv) Full : (front + 1) % n == Rear
Empty : Rear == front

Solution :- We know Stack Empty Condition is that when "Rear is pointing to front". and Stack full Condition is when "front" is very next to "Rear". Eg :- Array of size = 7

Rear = 0, So, front will be 1.

In (iii) (Rear + 1) % n == front
(0 + 1) % 7 == 1
1 % 7 == 1

So (iii) is correct.

Ques-2 :- Consider the following operation along with Enqueue and Dequeue operations, where k is a global parameter.

```
Multi-Dequeue (Q) {
    m = k;
    While (Q is not empty) and (m > 0) {
        Dequeue (Q);
        m = m - 1;
    }
}
```

What is the worst case time complexity of a sequence of n queue operations on an initially empty queue.

(i) $O(n+k)$ (ii) $O(n)$ (iii) $O(nk)$ (iv) $O(n^2)$

Answer-2 :-

Q-3) :- Consider the following function :

```
void f (Queue Q)
{
  int i;
  if (! isEmpty (Q))
  {
    i = Delete (Q); f(Q);
    insert (Q, i);
  }
}
```

What operation is performed by the above function f ?

Answer-3 :- (i) Leaves the Queue Q unchanged

~~(ii)~~ Reverses the order of the elements in the Queue Q.

(iii) Deletes the element at the front of the Queue Q and inserts it at the rear keeping the other elements in the same order

(iv) Empties the Queue Q.

Solution :- (a) Queue :- Q [1, 2, 3, 4]