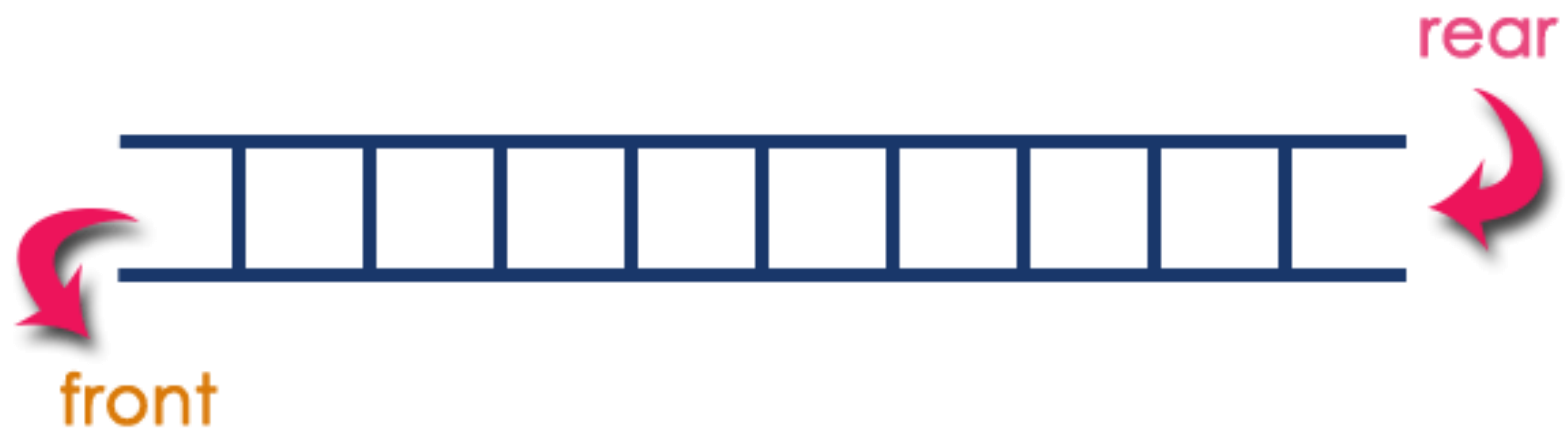
Queue

- Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends.

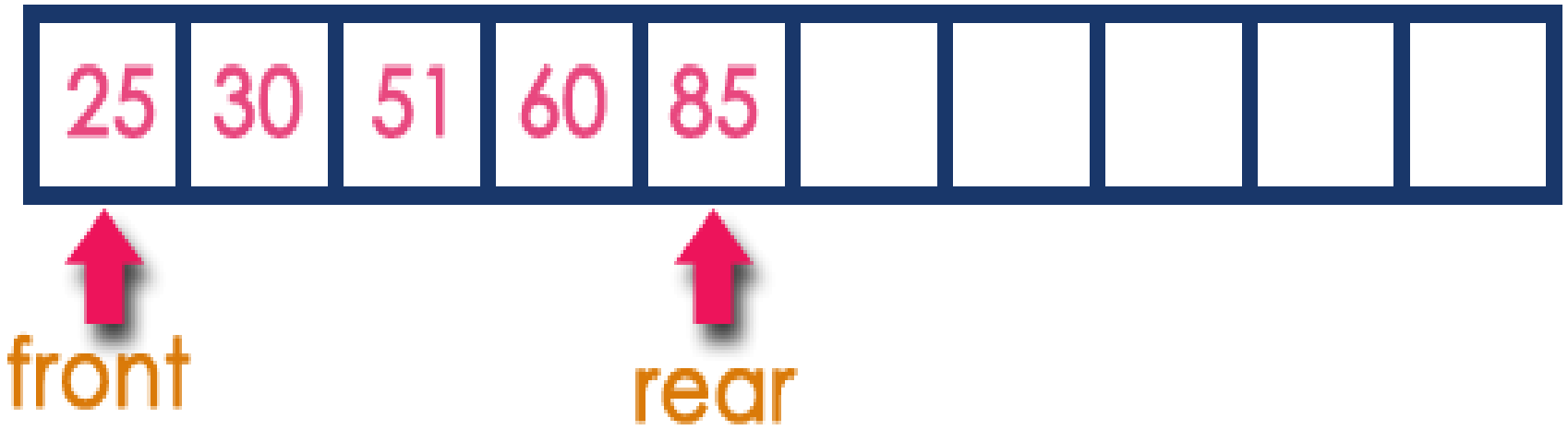
- The insertion is performed at one end and deletion is performed at another end.
- In a queue data structure, the insertion operation is performed at a position which is known as '**rear**'

- The deletion operation is performed at a position which is known as '**front**'.

- In queue data structure, the insertion and deletion operations are performed based on **FIFO (First In First Out)** principle.



After Inserting five elements...



Operations on a Queue

The following operations are performed on a queue data structure...

- **enQueue(value) - (To insert an element into the queue)**
- **deQueue() - (To delete an element from the queue)**
- **display() - (To display the elements of the queue)**

Queue data structure can be implemented in two ways. They are as follows...

- **Using Array**
- **Using Linked List**

Enqueue

Step 1 - Check whether **queue** is **FULL.** (**rear == SIZE-1**)

Step 2 - If it is **FULL,** then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.

Step 3 - If it is **NOT FULL,** then increment **rear** value by one (**rear++**) and set **queue[rear] = value.**

Dequeue

Step 1 - Check whether **queue** is **EMPTY**. (**front == rear**)

Step 2 - If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.

Step 3 - If it is **NOT EMPTY**, then increment the **front** value by one (**front ++**). Then display **queue[front]** as deleted element. Then check whether both **front** and **rear** are equal (**front == rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front = rear = -1**).

Display

Step 1 - Check whether **queue** is **EMPTY**. (**front == rear**)

Step 2 - If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.

Step 3 - If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front+1**'.

Step 4 - Display '**queue**[**i**]' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value reaches to **rear** (**i** <= **rear**)

```
void enQueue(int value){
   if(rear == MAX_SIZE-1)
      printf("\nQueue is Full!not possible!!!");
   else{
      if(front == -1)
            front = 0;
      rear++;
      queue[rear] = value;
      printf("\nInsertion success!!!");
   }
}
```

```
void insert(int value) {
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode -> next = NULL;
   if(front == NULL)
      front = rear = newNode;
   else{
      rear -> next = newNode;
      rear = newNode;
   }
   printf("\nInsertion is Success!!!\n"); }
```

```
void deQueue(){
   if(front >rear)
printf("\nQueue is Empty!!! ");
   else{
      printf("\nDeleted : %d", queue[front]);
      front++;
   }
}
```

```
void delete()
{
   if(front == NULL)
      printf("\nQueue is Empty!!!\n");
   else{
      struct Node *temp = front;
      front = front -> next;
      printf("\nDeleted element: %d\n", temp->data);
      free(temp);
   }
}
```

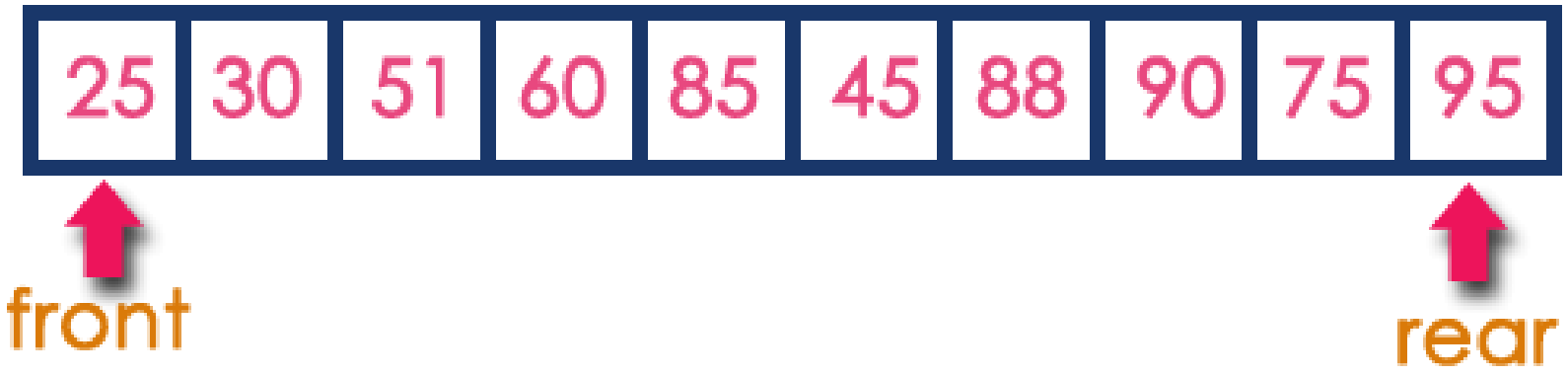
```
void display(){
   if(rear == -1)
      printf("\nQueue is Empty!!!");
   else{
      int i;
      printf("\nQueue elements are:\n");
      for(i=front; i<=rear; i++)
            printf("%d\t",queue[i]);
   }
}
```

```
void display()
{
   if(front == NULL)
      printf("\nQueue is Empty!!!\n");
   else{
      struct Node *temp = front;
      while(temp->next != NULL){
            printf("%d--->",temp->data);
            temp = temp -> next;
      }     printf("%d--->NULL\n",temp->data);
   }
}
```

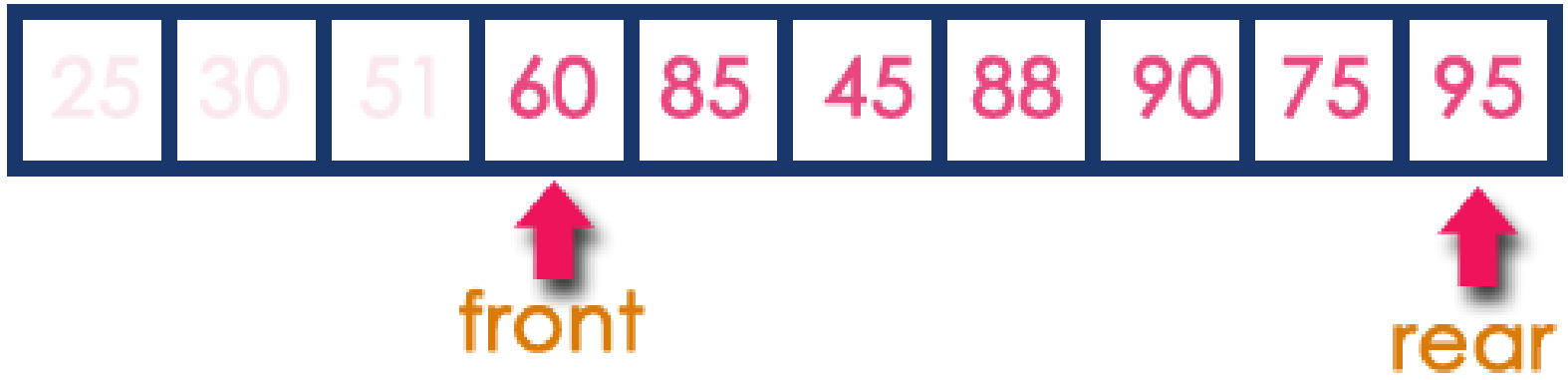
Types of Queues in Data Structure

- Simple **Queue.**
- Circular **Queue.**
- Priority **Queue.**
- Dequeue (Double Ended **Queue**)

Queue is Full



Queue is Full (Even three elements are deleted)

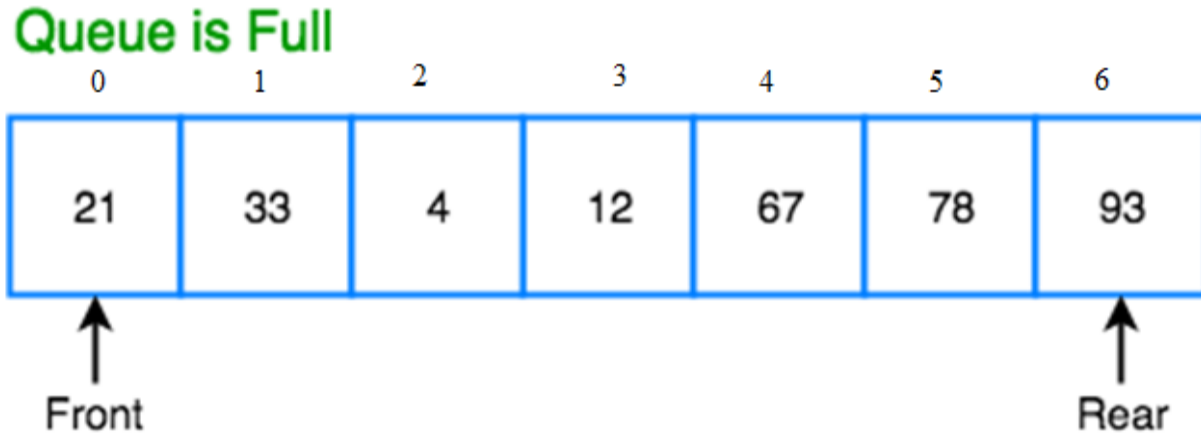


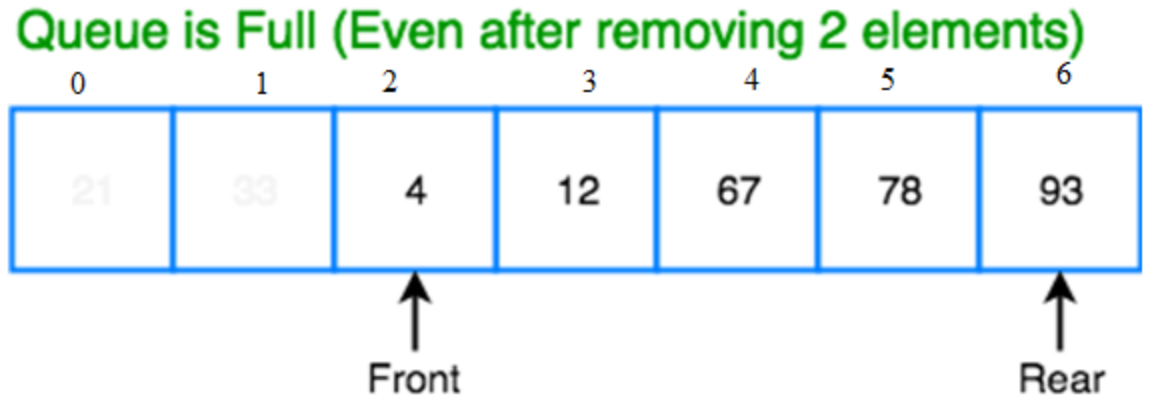
Circular Queue

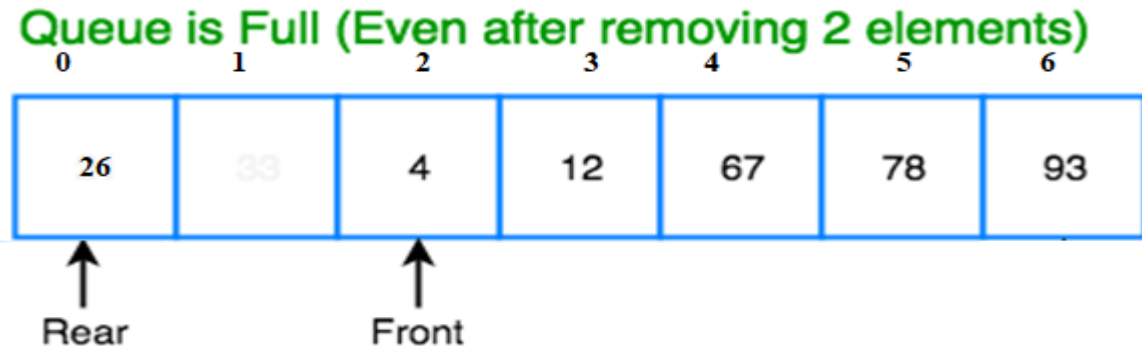
linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

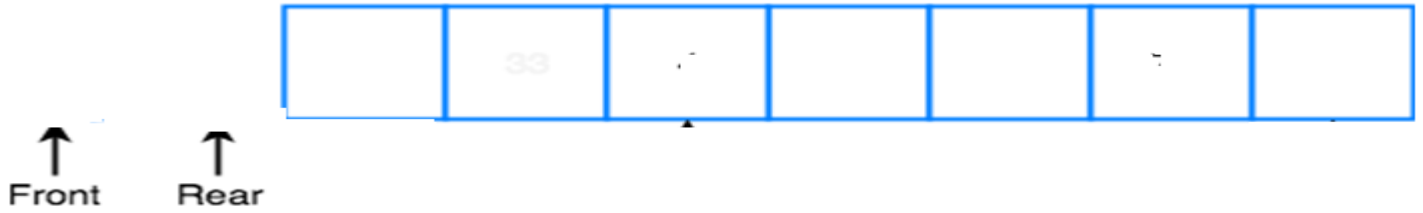
CPU Scheduling
Memory Management
Traffic Management

```
void insert(){ int no;
if((front ==0 && rear == max-1) || front == rear+1)
{ printf("\nCircular Queue Is Full !\n");
return; }
printf("\nEnter a number to Insert :");
scanf("%d",&no);
if(front==-1)
front=front+1;
if(rear==max-1)  rear=0;
else rear=rear+1; CQueue[rear]=no;
}
```

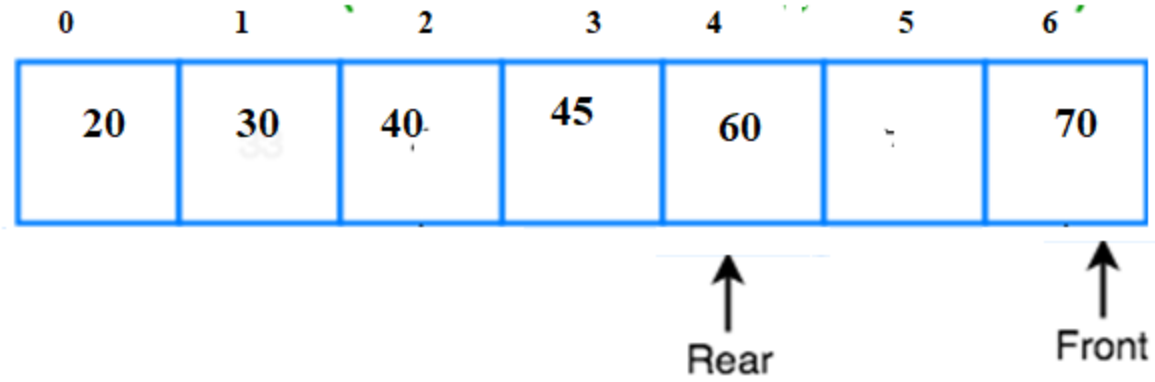




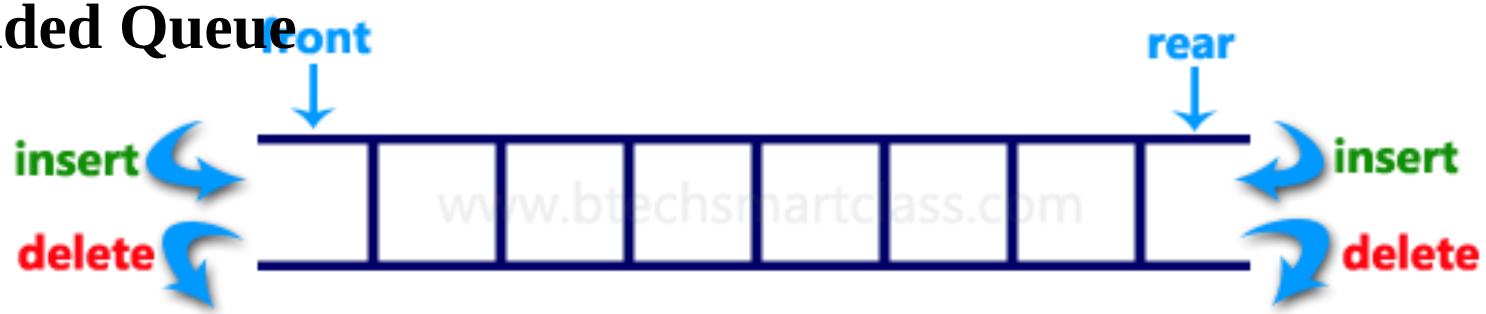





```
int delete() {
int e;
if(front==-1) { printf("\nThe Circular Queue is Empty !!\n");  }
e=CQueue[front];
if(front==max-1)
front=0;
else if(front==rear)
{ front=-1; rear=-1; }
else front=front+1;
printf("\n%d was deleted !\n",e); return e; }
```

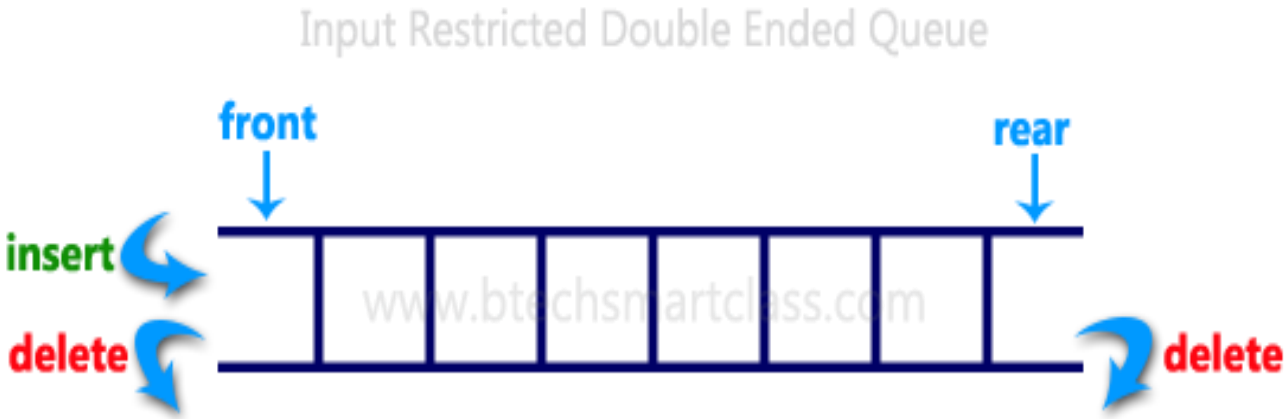


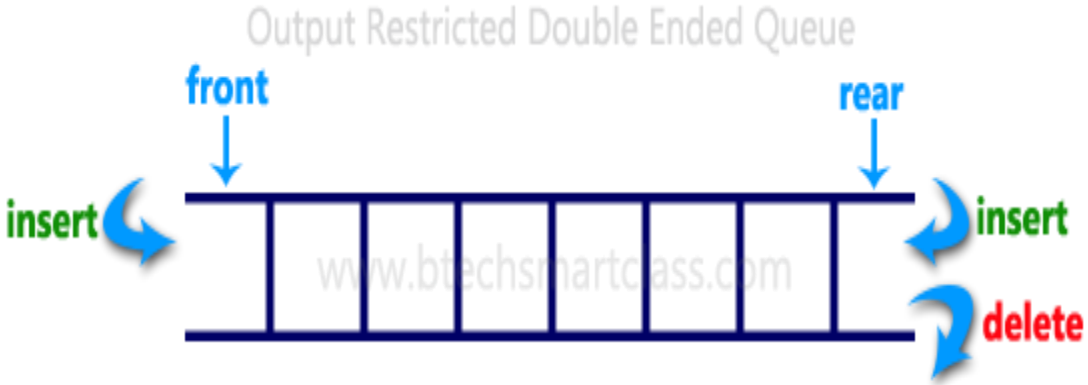
Double Ended Queue



Double Ended Queue can be represented in TWO ways, those are as follows...

- Input Restricted Double Ended Queue
- Output Restricted Double Ended Queue





Priority Queue

- It is collection of elements where elements are stored according their priority levels.
- Inserting and removing of elements from queue is decided by the priority of the elements.
- An elements of higher priority is processed first.
- Two elements having same priority will be processed first come first served basis