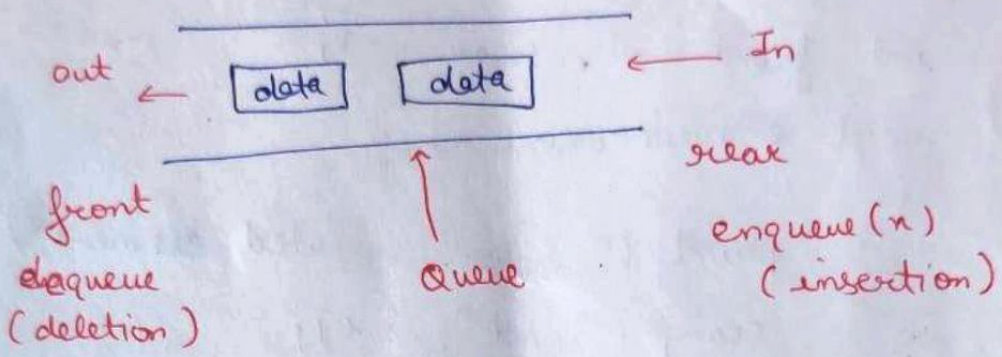
Queue

- It is defined as a linear data structure i.e. open at both ends.
- Operations are performed in FIFO (First In First Out) or LIFO (Last In First Out) order.



Operations :→

1. Enqueue(x) :→ insertion from rear/tail (from end).
2. Dequeue() :→ deletion from front/head.
3. peek()/front() :→ get the first element.
4. isfull() :→ Queue is full or not (Overflow).
5. isempty() :→ Queue is empty or not (Underflow).
6. display() :→ display all the elements.

Applications :→

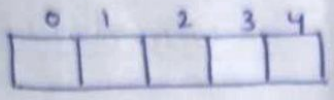
- In operating Systems.
- In buffers like CD player or maintain the play list in MP3.
- In waiting list for shared resources.

Implementation :→ Can be done by using

- Arrays
- Linked list
- Stack

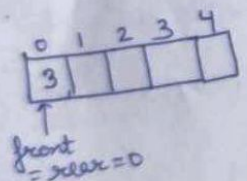
i) Arrays :→

front = rear = -1

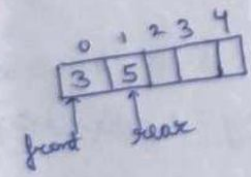


a) Insertion or enqueue (x) :→

```
#define N 5
int queue[N];                    // queue of size 5
int front = -1, rear = -1;       // Queue is empty
void enqueue (int x)
{
    printf (" enter inserted element");
    scanf (" %d", &x);
  if ( rear == N-1)
  {
      printf (" Overflow"),
  }
  elseif ( front == -1 && rear == -1)
    {
        front = rear = 0;
        int queue [rear] = x;
    }
  else
    {
        rear ++ ;
        queue [rear] = x;
    }
}
```



→



b) Deletion or dequeue () :→

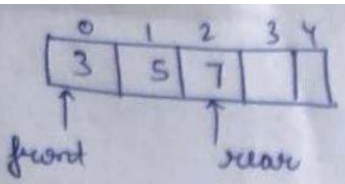
```
void dequeue ()
{
    if ((front == -1) && (rear == -1))
    {
        printf (" underflow ");
    }
    else if ( front == rear )
    {
        front = rear = -1 ;
    }
    else
    {
        front ++ ;
    }
}
```

only one element in queue

c) Display () :→

```
void display()
{
    if ( front == -1 && rear == -1)
    {
        printf (" Queue is empty ");
    }
    else
    {
        for ( i = front ; i <= rear ; i++)
        {
            printf ("%d", queue [i]);
        }
    }
}
```

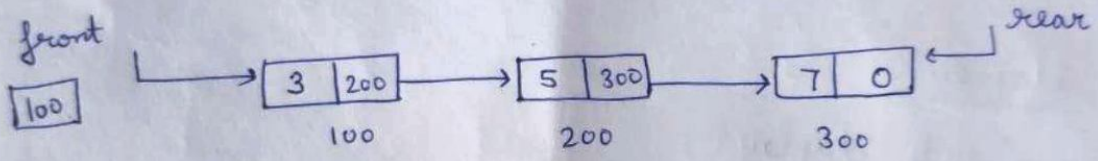
Output :- 3 5 7



d) Peek() :→

```
void peek()
{
    if (rear == -1 && front == -1)
    {
        printf("Queue is empty");
    }
    else
    {
        printf("%d", queue[front]);
    }
}
```

(ii) Linked List :→



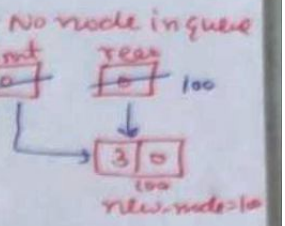
a) Enqueue(x) :→

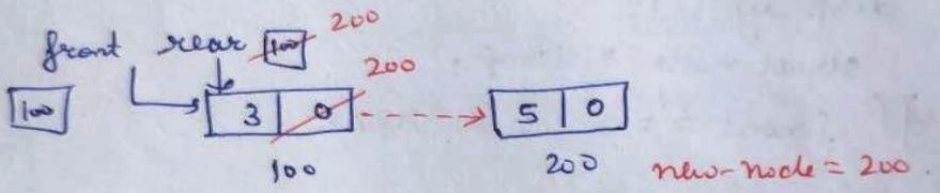
```
struct node
{
    int data;
    struct node * next;
};
struct node * front = * rear = 0;
void enqueue (int x)
{
    struct node * new_node;
    new_node = (struct node *) malloc (size of (struct node));
    printf ("enter data");
```

```
scanf ("%d", &x);
    new-node -> data = x;
    new-node -> next = 0;

if ( front == 0 && rear == 0 )
{
        front = rear = new-node;
}

else
{
    rear -> next = new-node;
    rear = new-node;
}
```



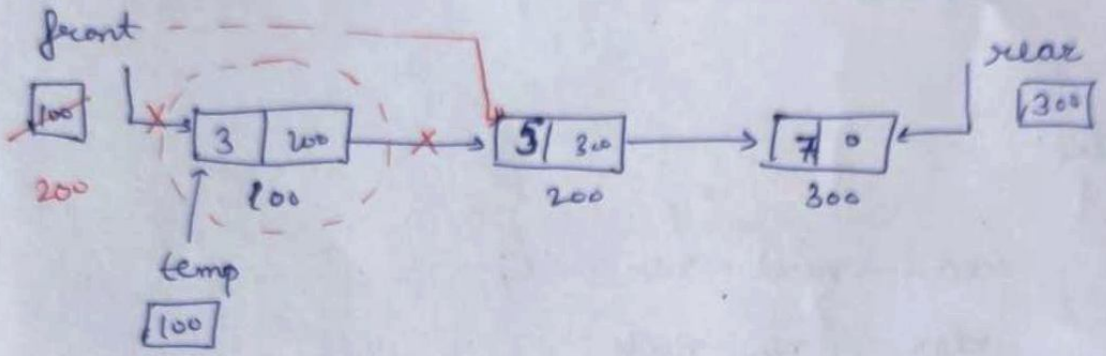


b) Dequeue :->

```
Struct node
{
    int data;
    Struct node *next;
};
Struct node * front = * rear = 0;
void dequeue ()
{
    Struct node * temp;
    temp = front;
if ( front == 0 && rear == 0 )
{
    Printf (" Queue is empty ");
};
```

Ans:-

```
        else
        {
            front = front -> next;
            free (temp);
        }
    }
```



c) display () :-

```
void display ()
{   struct node * temp;
    if ( front == 0  && rear == 0)
    {
        printf (" Queue is empty");
    }
    else
    {
        temp = front;
        while (temp != 0)
        {
            printf ("%d", temp -> data);
            temp = temp -> next;
        }
    }
}
```

d) Peek () :-

```
void peek ()
{
```

```
int front = rear = -1;
void enqueue (int x)
{
    if (front == -1 && rear == -1)
    {
        front = rear = 0
        queue [rear] = x;
    }
    else if ((rear+1) % N == front)
    {
        printf (" Overflow");
    }
    else
    {
        rear = (rear+1) % N;
        queue [rear] = x;
    }
}
```

// no element in queue

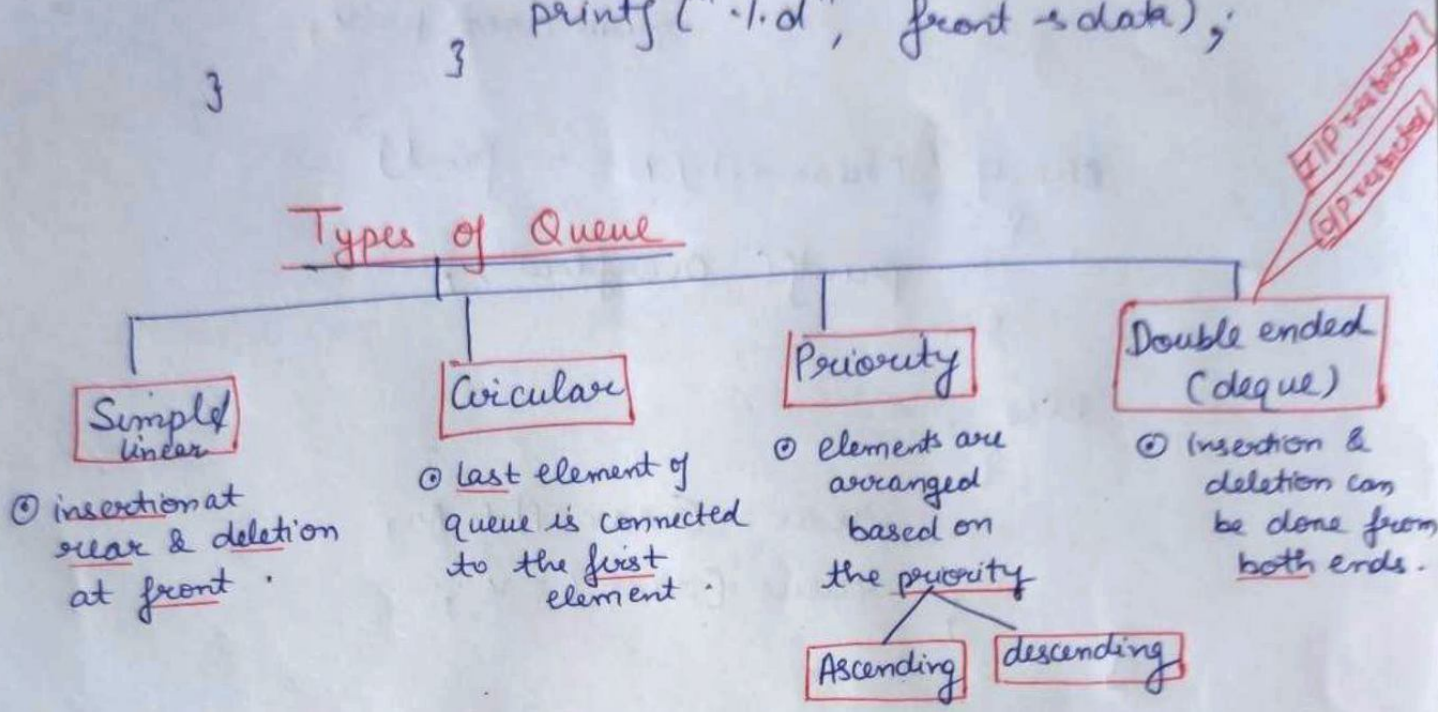
(b) Deletion OR Dequeue :-

```
void dequeue ()
{
    if (front == -1 && rear == -1)
    {
        printf (" Underflow");
    }
    else if (front == rear)
    {
        front = rear = -1;
    }
    else
    {
        front = (front+1) % N;
    }
}
```

Ans

```
if (front == 0 && rear == 0)
{
    printf(" Queue is empty ");
}
else
{
    printf("%d", front->data);
}
```

Types of Queue

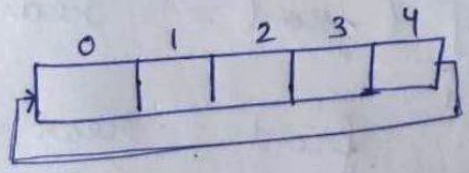


- Simple/linear: ⊙ insertion at rear & deletion at front.
- Circular: ⊙ last element of queue is connected to the first element.
- Priority: ⊙ elements are arranged based on the priority
- Double ended (deque): ⊙ insertion & deletion can be done from both ends.

Circular :→ ⊙ Also known as Ring buffer.
⊙ It can do better memory utilization.

Implementation can be done by
- Arrays
- Linked list

(i) **Arrays :→**



Enqueue / Insertion :→

```
#define N 5
int queue[N];
```

b ⓒ Display :→

```
void display()
{
        int i = front;
    if (front == -1 && rear == -1)
    {
        printf("overflow"/"Queue is empty");
    }
    else
    {
        while (i != rear)
        {
            printf("%d", queue[i]);
            i = (i+1) % N;
        }
        printf("%d", queue[rear]);
    }
}
```

ⓓ Peek :→

```
void peek()
{
    if (front == -1 && rear == -1)
    {
        printf("Queue is empty");
    }
    else
    {
        printf("%d", queue[front]);
    }
}
```

Ans:

Linked list :->

a) **Insertion :->**

```
Struct node
{
    int data;
    Struct node * next;
};
Struct node * front=0, *rear=0;
void enqueue (int x)
{
    Struct node * new_node;
new_node = (struct node*)malloc (size of (Struct node));
printf ("enter inserted data");
scanf ("%d", &x);
    new_node -> data = x;
    new_node -> next = 0;
if (rear == 0 && front == 0)
{
        printf ("Queue is empty");
        front = rear = new_node;
        rear -> next = front;
}
else
{
        rear -> next = new node;
        rear = new_node;
        rear -> next = front;
}
```

b) deletion :->

```
Struct node
 {
     int data;
     Struct node * next;
 };
 Struct node * front = 0 , * rear = 0;
 void dequeue()
 {
     Struct node * temp;
       temp = front;
   if ( front == 0 && rear == 0)
     {
         printf(" Underflow");
     }
  else if ( front == rear )
    {
       front = rear = 0;
         free ( temp);
    }
  else
     {
       front = front -> next;
       rear -> next = front;
         free (temp);
     }
```

c) display() :->

```
Struct node
{
    int data;
    struct node *next;
};
Struct node * front = 0, * rear = 0;
void display()
{
    Struct node *temp;
    temp = front;
    if ( front == 0 && rear == 0)
    {
        printf(" Queue is empty");
    }
    else
    {
        while ( temp -> next ! = front )
        {
            printf ("%d", temp -> data);
            temp = temp -> next;
        }
        printf("%d", temp -> data);
    }
}
```

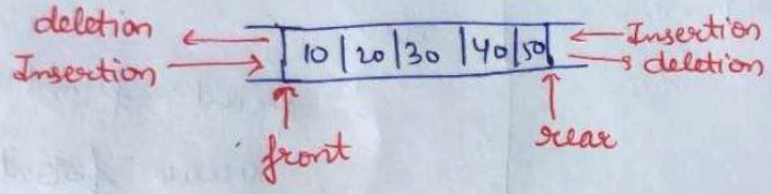
d) peek() :->

```
Struct node
{
    int data
    Struct node * next;
};
```

```
Struct node * front = 0 , * rear = 0;
void peek()
{
    if (front == 0 && rear == 0)
    {
        printf("Queue is empty");
    }
    else
    {
        printf("%d", front->data);
    }
}
```

Double Ended Queue (deque) :→

- Elements can be added or removed at either ends but not in the middle.



There are two types of deque :→

1) Input restricted deque :→ insertion at one end, but deletion can be done at both the ends.

2) Output restricted deque :→ deletion at one end, but insertion can be done at both the ends.

Operations on deque :→

1) Insertion at front end
2) Insertion at rear end
3) deletion at front end
4) deletion at rear end.

a) Insertion :→ (At rear)

```
#define N 5;
int queue [N];
int front = rear = -1;
void insertatrear(int x)
{
    printf("enter x"); scanf("%d", &x);
    if (rear == (N-1))
    {
        printf("Overflow");
    }
    else if (front = rear = -1)
    {
        front = rear = 0;
        queue [rear] = x;
    }
    else
    {
        rear = rear + 1;
        queue[rear] = x;
    }
}
```

(At front)

```
#define N 5;
int queue [N];
int front = rear = -1;
void insertatfront (int x)
{
    printf("enter x");
    scanf("%d", &x);
    if (front == 0)
    {
        printf("Overflow");
    }
    else if (front = rear = -1)
    {
        front = rear = 0;
        queue [front] = x;
    }
    else
    {
        front = front - 1;
        queue [front] = x;
    }
}
```

b) Deletion (At front)

```
void deleteat_front()
{
    if (front == -1 && rear == -1)
    {
        printf("Underflow");
    }
    else if (front == rear)
    {
        front = rear = -1;
    }
    else
    {
        front = front + 1;
    }
}
```

(At rear)

```
void deleteat rear()
{
    if (front == -1 && rear == -1)
    {
        printf("Underflow");
    }
    else if (front == rear)
    {
        front = rear = -1;
    }
    else
    {
        rear = rear - 1;
    }
}
```

⊙ display() & peek() is same as in linear / Simple Queue.

⊙ Implementation using linked list is same as insertion at beginning, insertion at ending, deletion at begin, deletion at end from linked list (Unit - I refer that)

⊙ It can also implemented by using circular Queue.

Priority Queue :-

⊙ Highest priority come first in priority queue.

Characteristics :-

1) Each element has some priority.
2) Higher priority will be deleted first.
3) Same priority will use FIFO principle.

Operations :-

1) poll() → deletion
2) add() → insertion
3) display()
4) peek()
5) isempty()
6) is full()

Types :-

1) Ascending order :- Lowest number have higher priority.

Eg:- 1, 2, 3, 4
↓
Highest priority.

2) descending order :- Highest number have ~~higher~~ priority.
higher

Eg:- 4 3 2 1
↓ Highest

Implementation

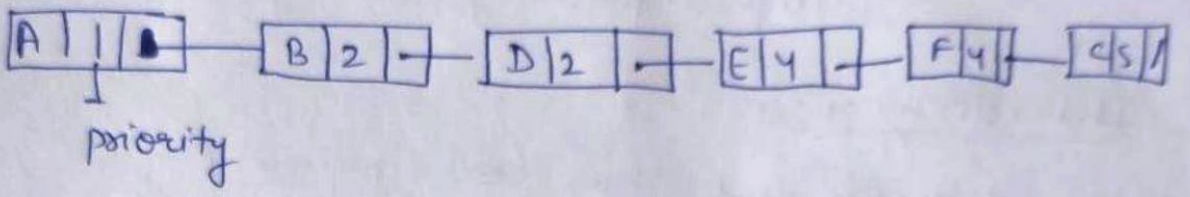
Array Linked list

Eg:-

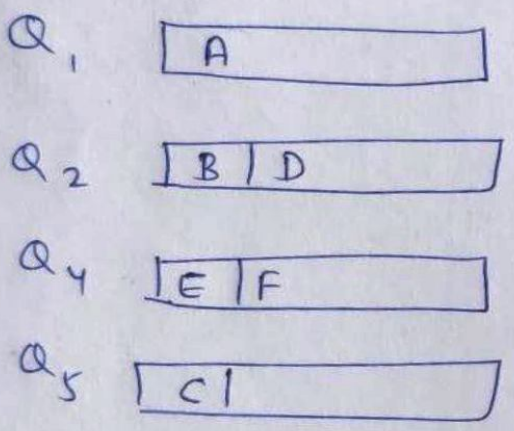
	A	B	C	D	E	F
Priority	1	2	5	2	4	4

Sol.

Linked list :→ First priority element comes first.



In array, multiple queues will be used, (for each priority we will make a new queue).



* Insertion & deletion will be same as linear queue on each queue.