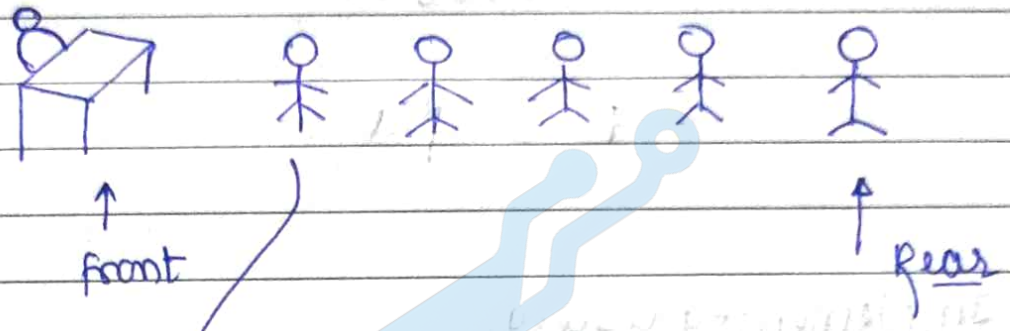
Section-14

Queues

Queue ADT :-

↳ works in FIFO (first in first out)



Insertion is done at Rear end

deletion is done at front end

Queue ADT

Data:

1. Space for ~~sorting~~ storing elements
2. Front - for deletion
3. Rear - for insertion

Operations :-

1. enqueue(x)
2. dequeue()

3. isEmpty()

4. isfull()

5. first()

6. Last()

7. Display()

Implemented using

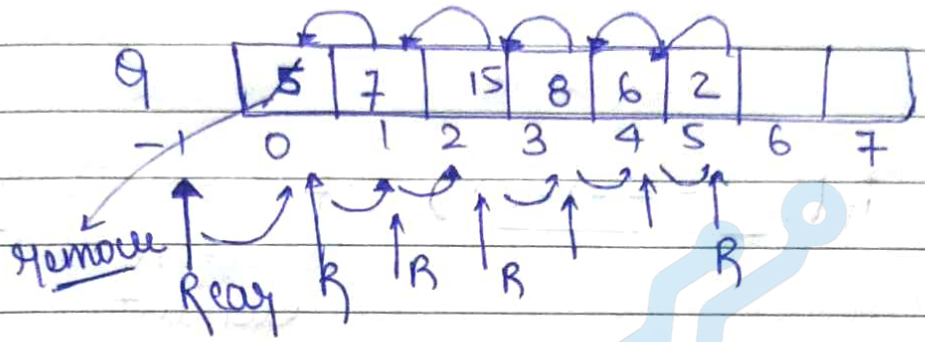
- Array
- Linked list

Queue using Array

1. Queue using single Pointer
2. Queue using front and rear
3. Drawbacks of Queue using Array

- Queue using Single Pointer

Size=8

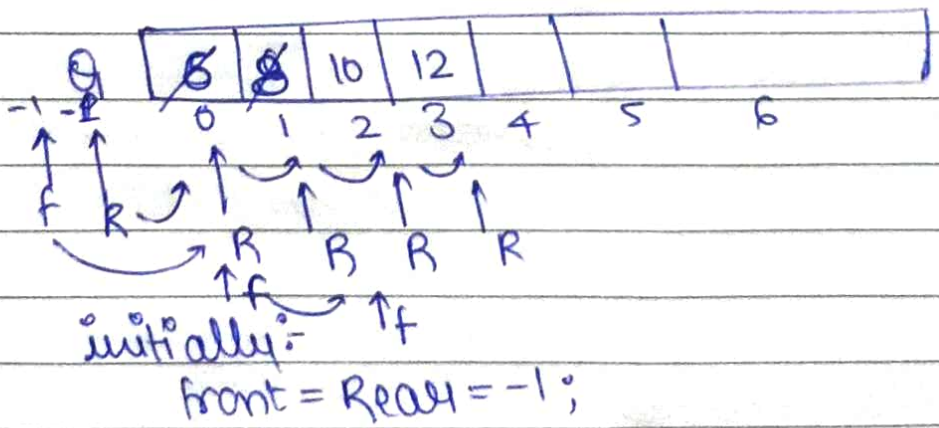


Insert – O(1)

delete – O(n)

- Queue using two Pointers

Size=7



initially:-
front = Rear = -1;

Insert – O(1)
Delete – O(1)

enqueue - O(1)

Dequeue - O(1)

Queue is Empty

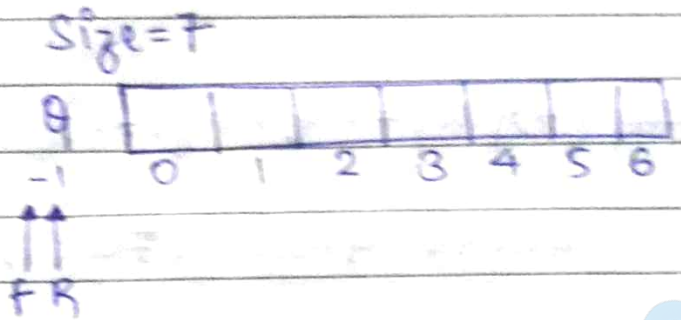
↳ if(front == rear) if front is pointing is before that element

if (front > Rear) if front is pointing at that element

Queue is ~~Empty~~ Full

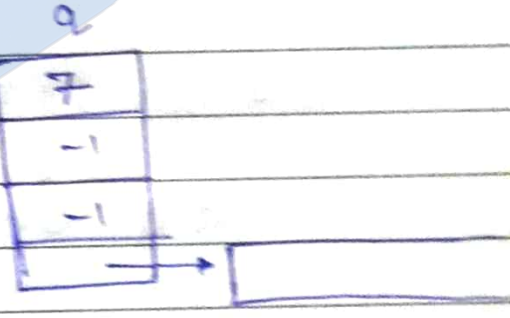
if (Rear = size - 1)

Implementation of Queue using Array




```
struct Queue
{
    int size;
    int front;
    int rear;
    int *Queue;
};
```

```
int main()
{
    struct Queue q;
```

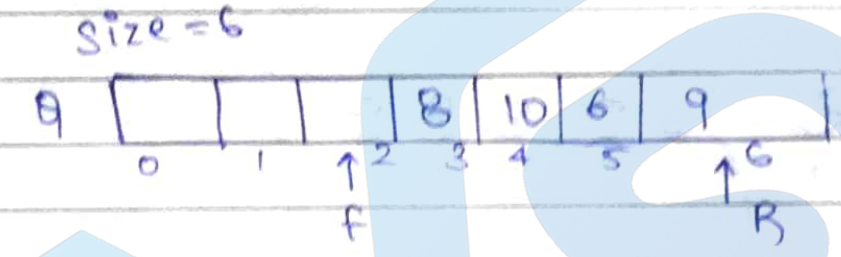



```
    cout<<"Enter Size: ";
    cin >>q.size;

    q.Q = (int*)malloc(q.size*sizeof(int));

    q.front = q.rear = -1;

}
```

```
void enqueue(Queue *q, int x)
{
    if (q->Rear == q->size-1)
    {
        cout<<"Queue is Full";
    }
    else
    {
        q->Rear++;
        q->Q[q->Rear] = x;
    }
}

void dequeue(Queue *q)
{
    int x=-1;
    if (q->front == q->Rear)
    {
        cout<<"Queue is Empty";
    }
    else
    {
        q->front++;
```

```
    x = q->a[q->front];
  }
  return x;
}
```

Drawback of Queue using Array



- We cannot reuse the space of deleted element.

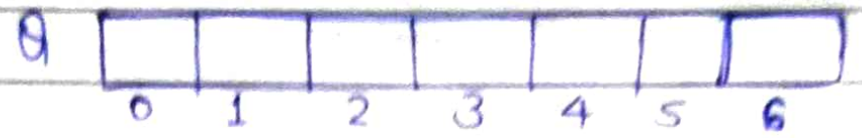
- Every location used only once.
- A situation where queue is empty, and full also.

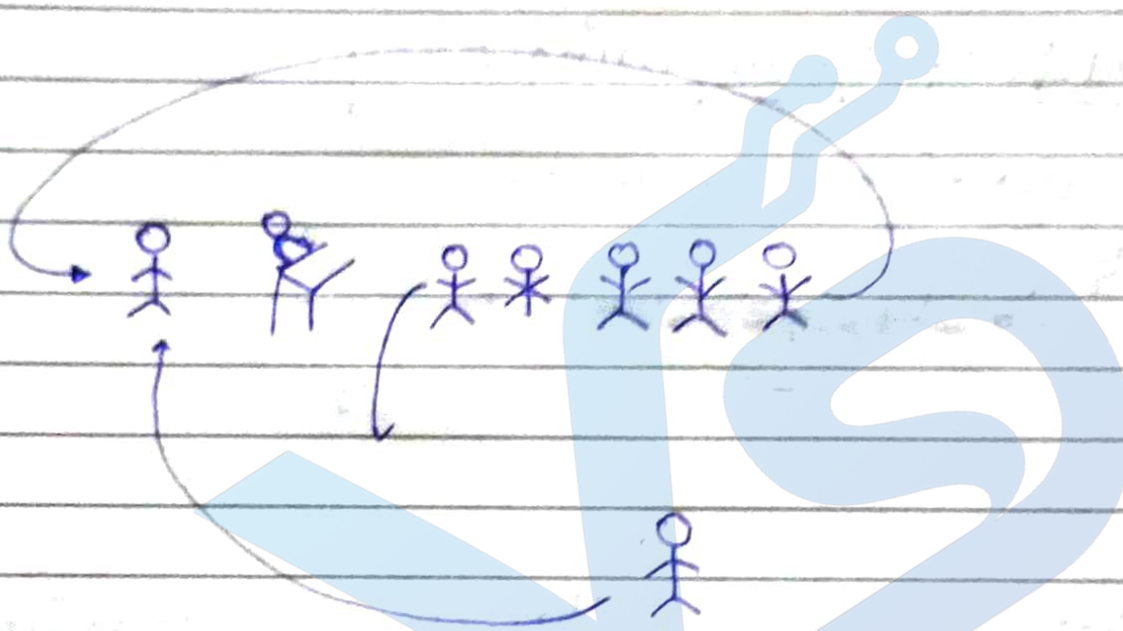
Sol^n:-

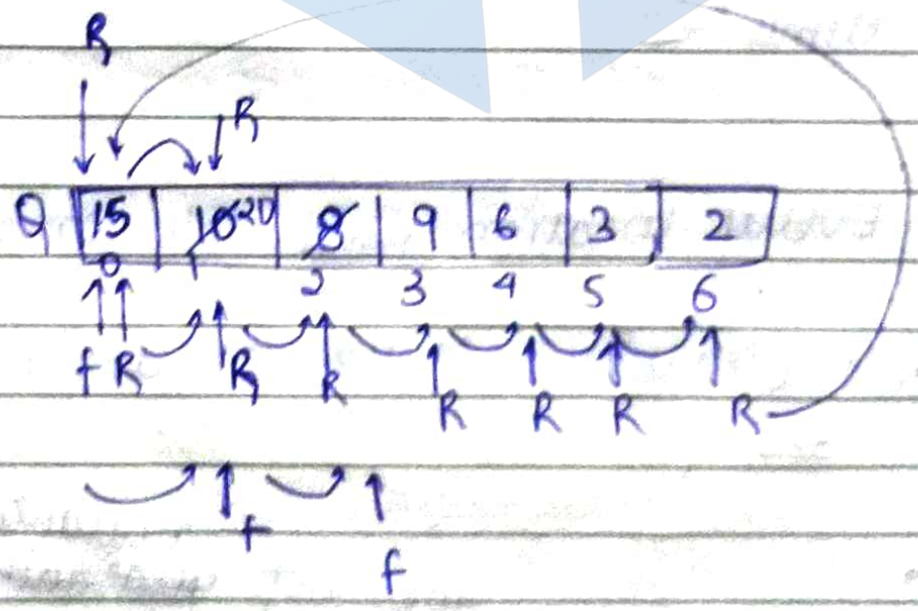
- Resetting Pointers → does not guarantee, to reuse the space.
  when queue is empty then front = rear = -1;
- Circular Queue

Circular Queue

Size = 7



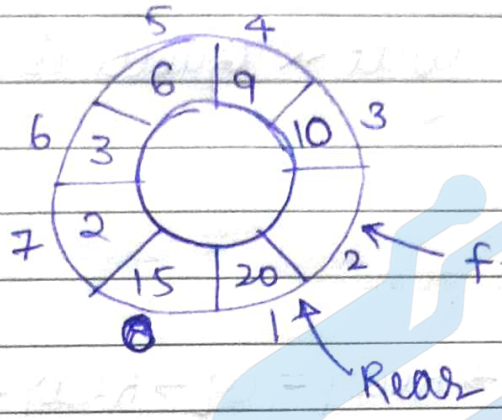




Best method to implement Queue using Array

↳ Circular Queue

Representation



Rear = (Rear+1) % size

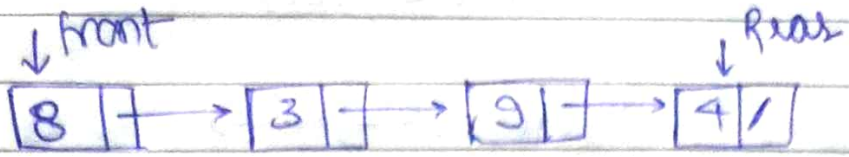
0	(0+1)%8	1
1	(1+1)%8	2
2	(2+1)%8	3
3	(3+1)%8	4
4	(4+1)%8	5
5	(5+1)%8	6
6	(6+1)%8	7
7	(7+1)%8	0

```
void enqueue (struct Queue *q, int x)
{
    if ((q->Rear+1)%q->size == q->front)
        cout<<"Queue is Full";
    else
    {
        q->Rear = (q->Rear+1)%q->size;
        q->Q[q->Rear] = x;
    }
}
```

OR)

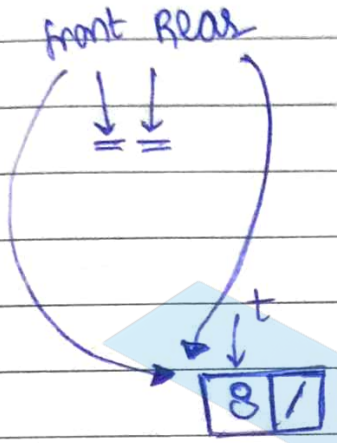
```
int dequeue (struct Queue *q)
{
    int x=-1;
    if (q->front == q->Rear)
        cout<<"Queue is Empty";
    else
    {
        q->front = (q->front+1)%q->size;
        x = q->Q[q->front];
    }
    return x;
}
```

Queue using linked list :-



Rear → insertion

front → Delete



Empty

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

Full

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

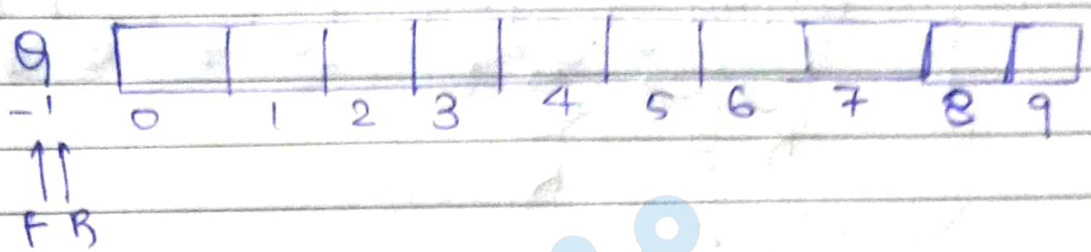
```
void enqueue (int x)
{
    NOde *t = new NOde;
    if(t==NULL)
        cout<<"Queue is full";
    else{
```

```
        t->data = x;
        t->next = NULL;

        if (front == NULL)
            front = rear = t;

        else
        {
            rear->next = t;
            rear = t;
        }
    }
}
```

OR

```
int dequeue()
{
    int x = -1;
    Node *p;
    if (front == NULL)
        cout << "Queue is Empty";

    else
    {
        p = front;
        front = front->next;
        x = p->data;
        delete (p)
    return x;}
```

Double Ended Queue (DEQUEUE)



front and Rear are used for both insertion and deletion

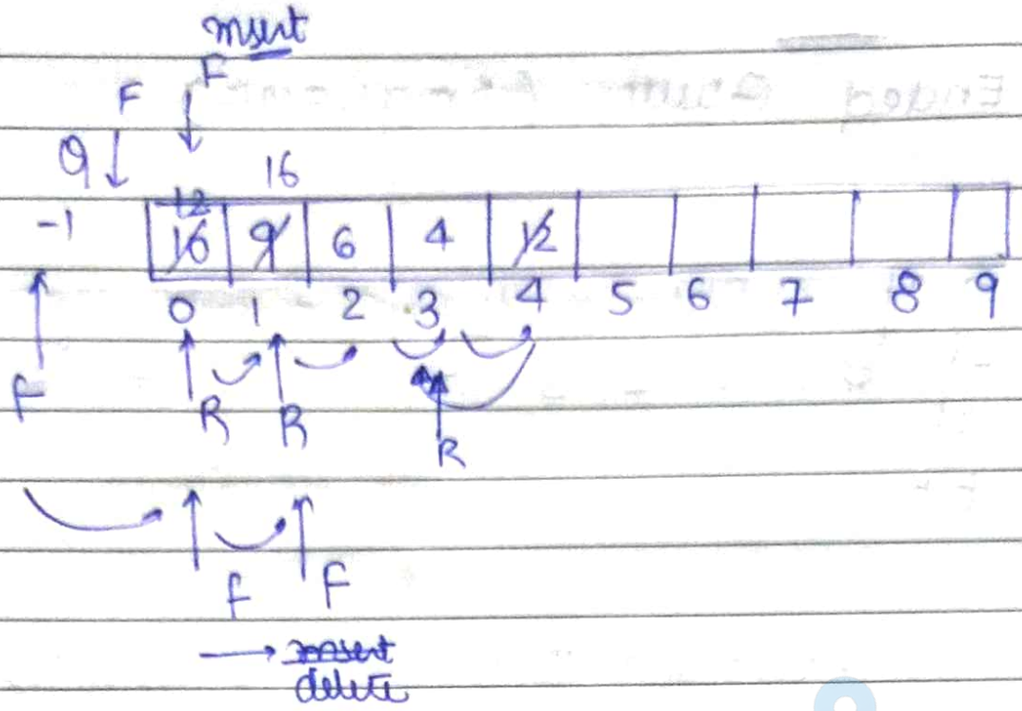
Queue

	Insert	Delete
front	×	✓
Rear	✓	×

DE Queue

	Insert	Delete
front	✓	✓
Rear	✓	✓

DE Queue not following FIFO



i/p Restricted

	insert	Delete
front	×	✓
Rear	✓	✓

→ Insert from front

→ Delete from front

→ Insert from rear

→ Delete from rear

o/p Restricted

	insert	Delete
front	✓	✓
Rear	✓	×

Priority Queue

1. Limited set of Priority
2. Element Priority

- Limited set of Priority ↓

  Mostly useful in operating System

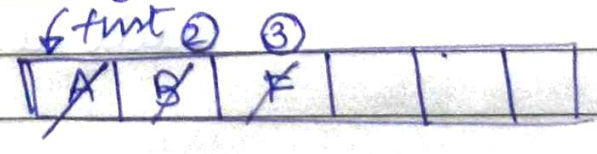
Priority = 3

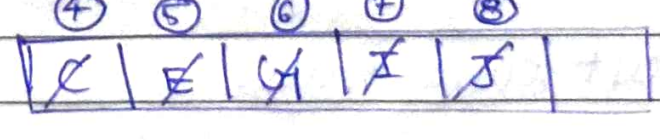
Element →	A	B	C	D	E	F	G	H	I	J
Priority →	1	1	2	3	2	1	2	3	2	2

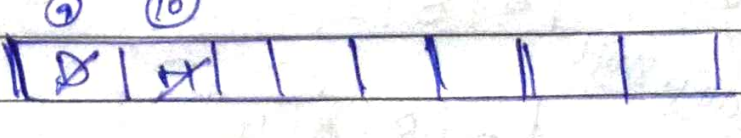
↑ high priority

↑ low priority

Priority Queues







sequence of delete

↓

first delete element is which has high priority.

• Element Priority

Element → 6, 8, 3, 10, 15, 2, 9, 17, 5, 8

Smaller Number → High Priority

Q | 2

1. Insert same order
   ↳ Delete max priority by searching it.
2. Insert in increasing order of Priority
   Delete last element of Array

1. Insert same order

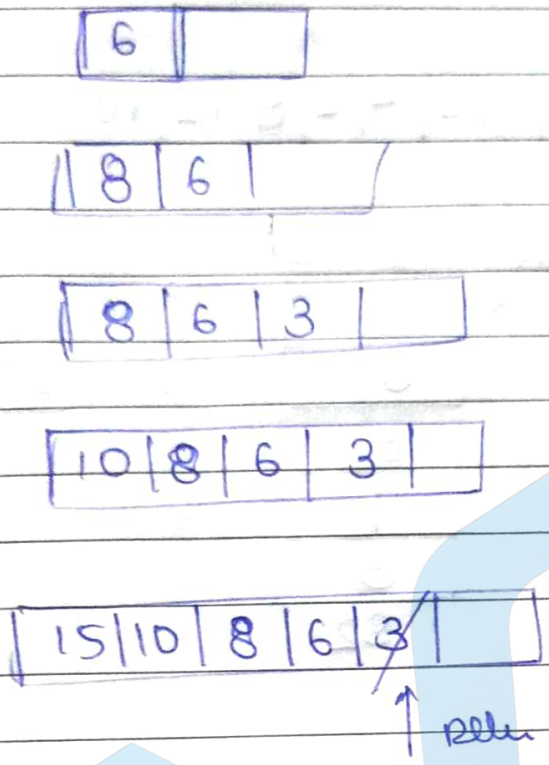
insert - O(1)
delete - O(n) → search + shift

| 6 | 8 | 3 | 10 | 15 | 2 | 9 | 17 | 5 | 8 |

| 6 | 8 | 3 | 10 | 15 | 9 | 17 | 5 | 8 |

| 6 | 8 | 10 | 15 | 9 | 17 | 5 | 8 |

2. Insert in Insearching order of Priority

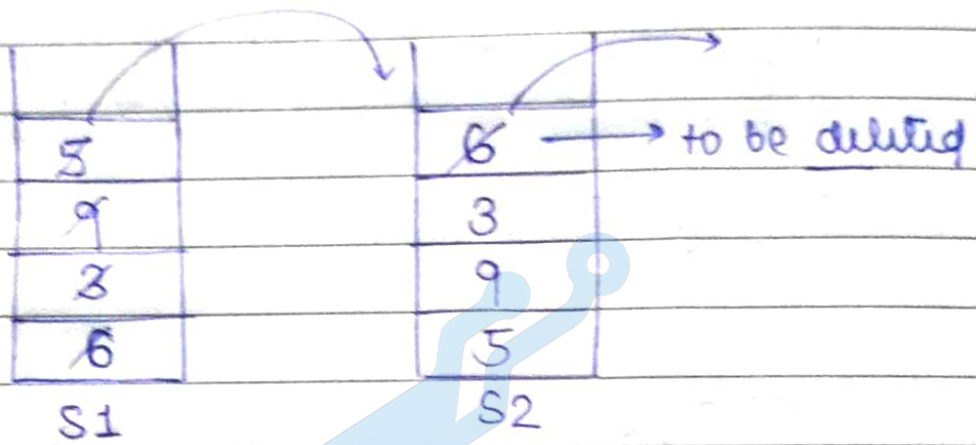


Insert → O(n)

Delete → O(1)

Queue using two stacks :-

element → 6, 3, 9, 5, 4, 2, 8, 12, 10 ....




```
enqueue(int x)
{
    push(&S1, x);
}

int dequeue()
{
    int x = -1;
    if(isEmpty(S2))
    {
        if(isEmpty(S1))
        {
            cout << "Queue Empty";
            return
        }
    }
    return x;
}
```

```
else
{
        while(!isEmpty(S1))
        {
                push(&S2, pop(&S1))
        }
}

return pop(&S2);

}
```