Solving Approches

Interative

- loops are used

```
int factortial(int num){
        int fact = 1;
        for(int i = 1 ; i <= num ; i++)
                fact *= i;
        return fact;
}
```

Time complexity = no of iterations
= n

Time complexity = O(n)

Recursive

- recursion
  formula : n! = n * (n-1)!
  stop : n == 1 (1! or 0! = 1)

```
int recFactorial(int num){
        if(num == 1)
                return 1;
        return num * recFactorial(num-1);
}
```

Time complexity = no of recursive function calls
= n

Time complexity = O(n)

Binary Search

0 1 2 3 4 5 6 7

arr = {11, 22, 33, 44, 55, 66, 77, 88};

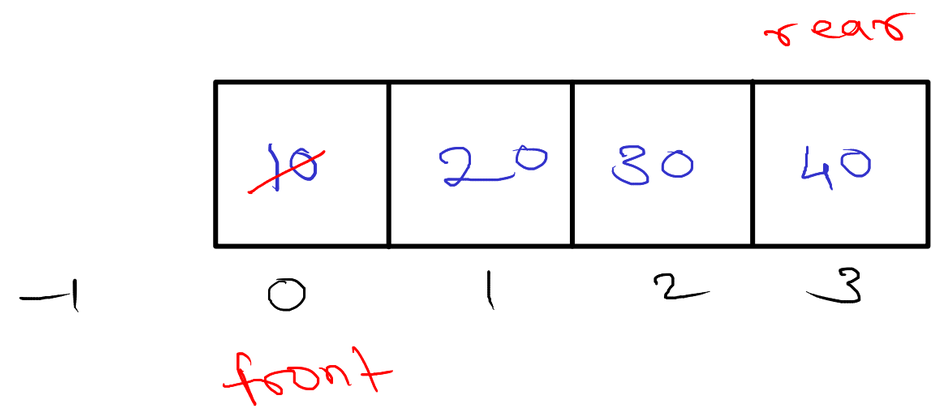
```
int main(){
        i = BS(arr, 0, 7, 66);
}
```

```
int BS(arr, 0, 7, 66)
{
        if(0 > 7)
        m = (0 + 7)/2 = 3;
        if(66 == 44)
        else if(66 < 44)
        else
        return BS(arr, 4, 7, 66);
}
```

```
int BS(arr, 4, 7, 66)
{
        if(4 > 7)
        m = (4 + 7)/2 = 5;
        if(66 == 66)
                return 5;
}
```

Linear Queue

- linear data structure of similar data elements
- data is inserted from one end (rear).
- data is removed from another end (front).
- works on principle of First In First Out (FIFO).



Cinditions:

1. is Empty

rear == front

2. is Full

rear == SIZE - 1

Operations:

1. Add/Insert/Push/Enqueue:

a. reposition the rear
b. add data at rear index

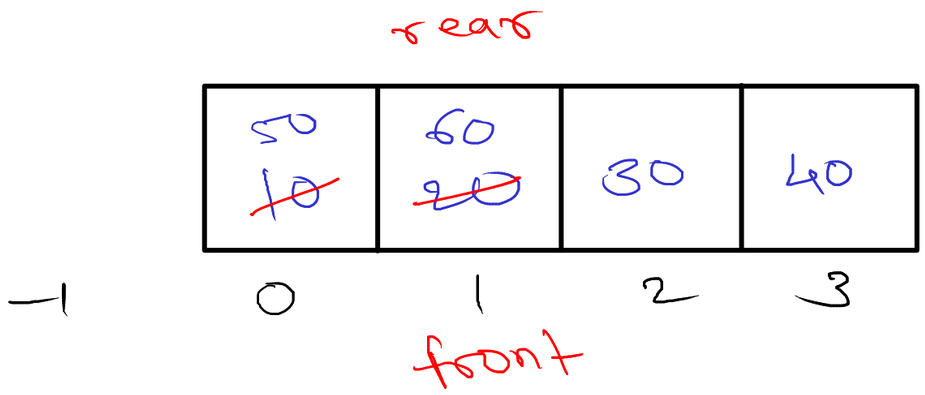
2. Delete/Remove/Pop/Dequeue:

a. reposition the front

3. Peek/collect:

a. read data of front + 1 index

Circular Queue



SIZE = 4

rear = (rear + 1) % SIZE

front = (front + 1) % SIZE

front = rear = -1

= (-1 + 1) % 4 = 0

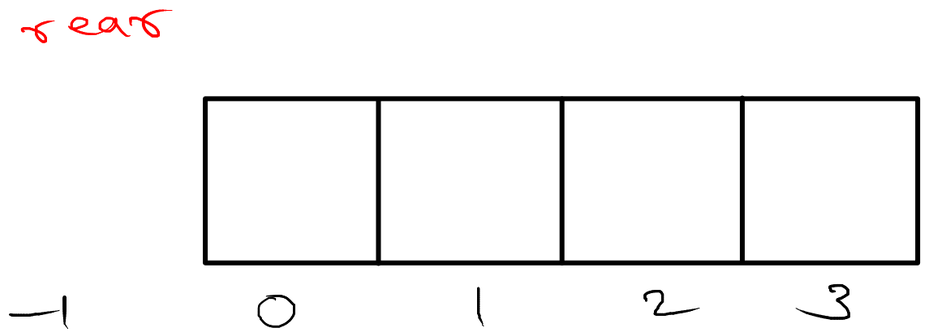
= (0 + 1) % 4 = 1

= (1 + 1) % 4 = 2

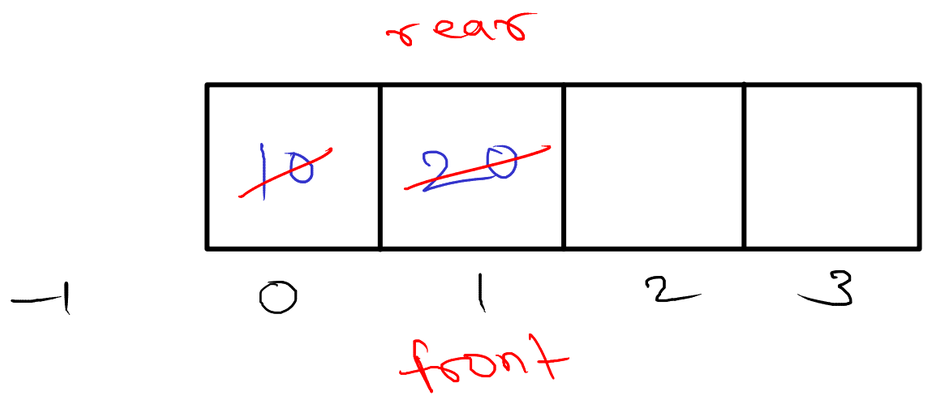
= (2 + 1) % 4 = 3

= (3 + 1) % 4 = 0

Circular Queue - Empty

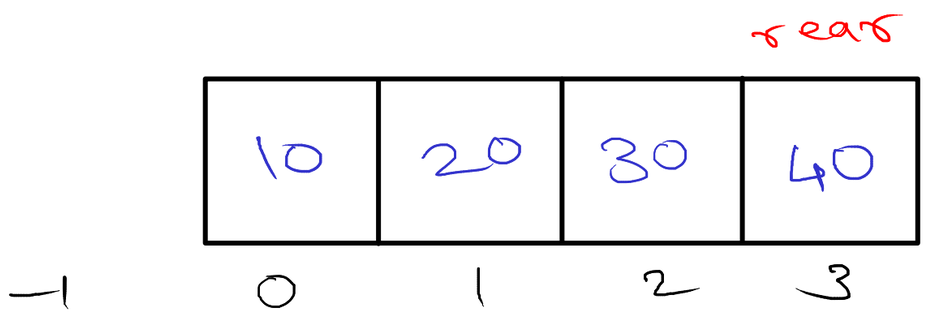


front == rear && rear == -1



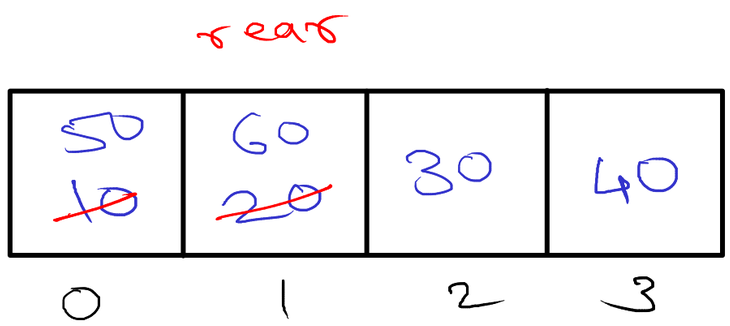
pop()
a. reposition front
b. if(front == rear)
front = rear = -1

Circular Queue - Full



front == -1 && rear == SIZE-1

||



front == rear && rear != -1