Data structures and Algorithm

Queues

Dr. R.Preeth,

Queues

- Queue is a container of objects
- First-in first-out principle
- Line of people waiting to reserve a ticket in railway reservation counter
- Elements enter the queue at the ***rear*** and remover from the ***front***

Queue Abstract Data type

- Defines a container , where element access and deletion are restricted to the first element in the sequence
- Deletion at the front of the queue and Insertion at rear of the queue
- Enqueue(o): Insert objects o at the rear of the queue

  Input : Object ; **Output** : None.

Queue Abstract Data type

- Dequeue(): Remove and return from the queue the object at front

  Input : None; **Output** : Object.

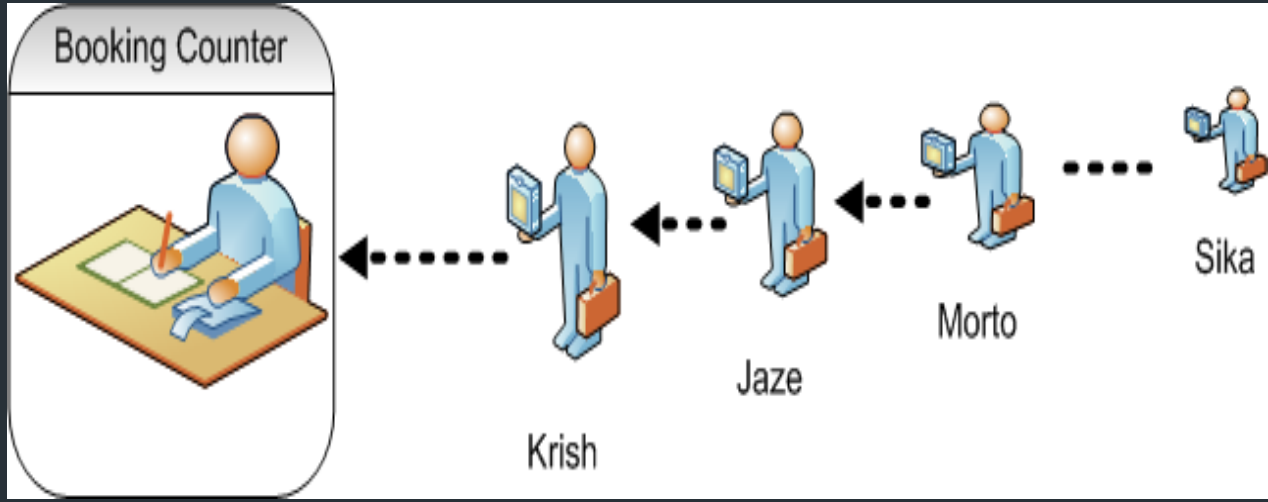
- Size(): Return the number of objects in the queue

  Input : None**; Output :** Integer**.**

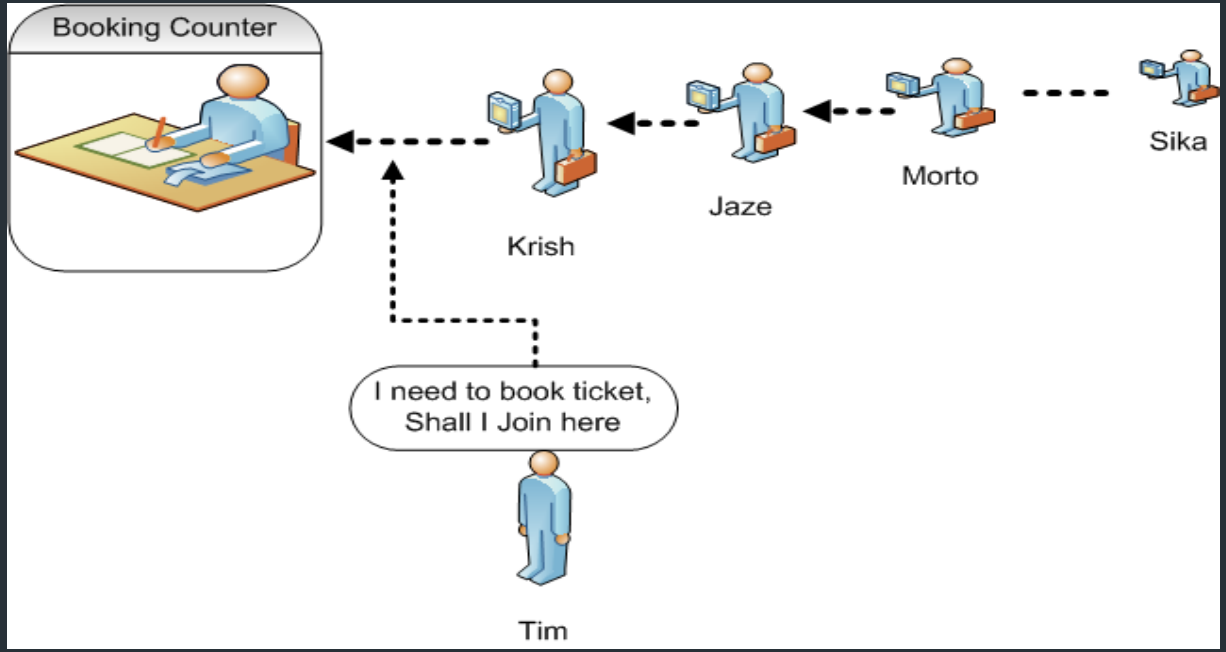
- Is Empty():Return a boolean value indicating if the queue is empty

  Input : None**; Output :** boolean**.**

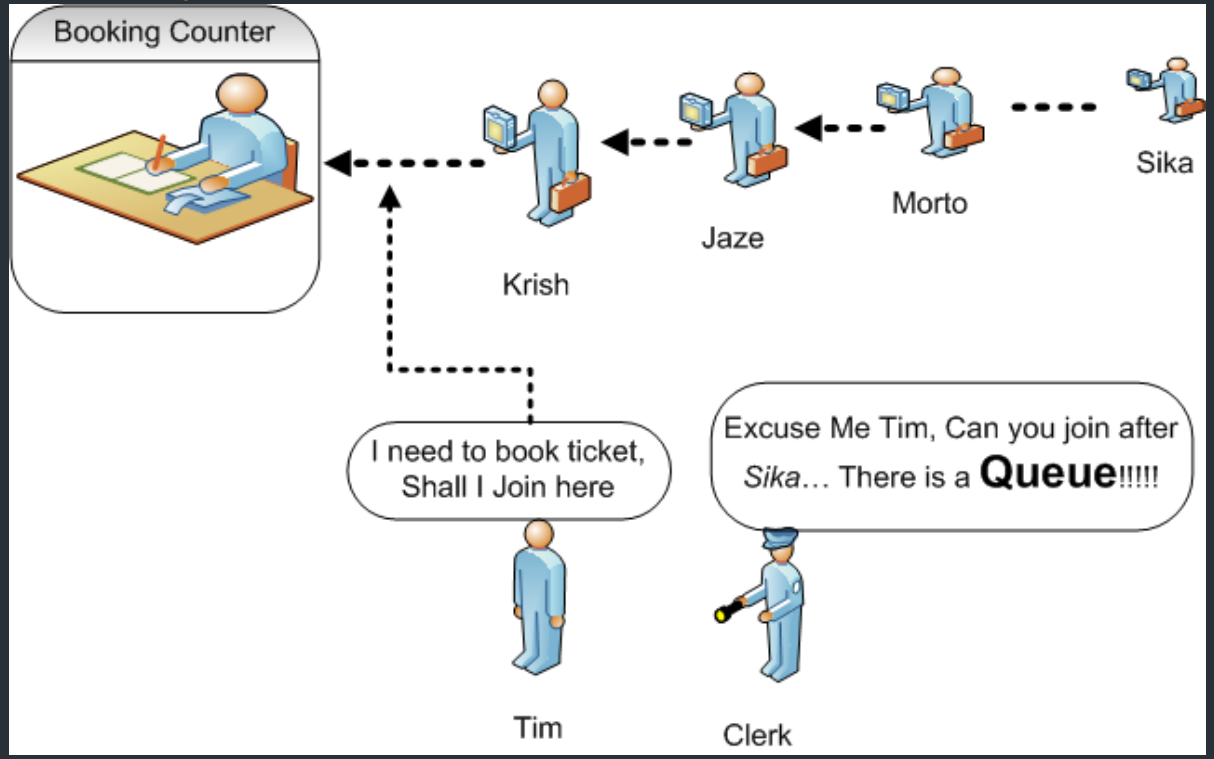
Graphical Representation of Queue



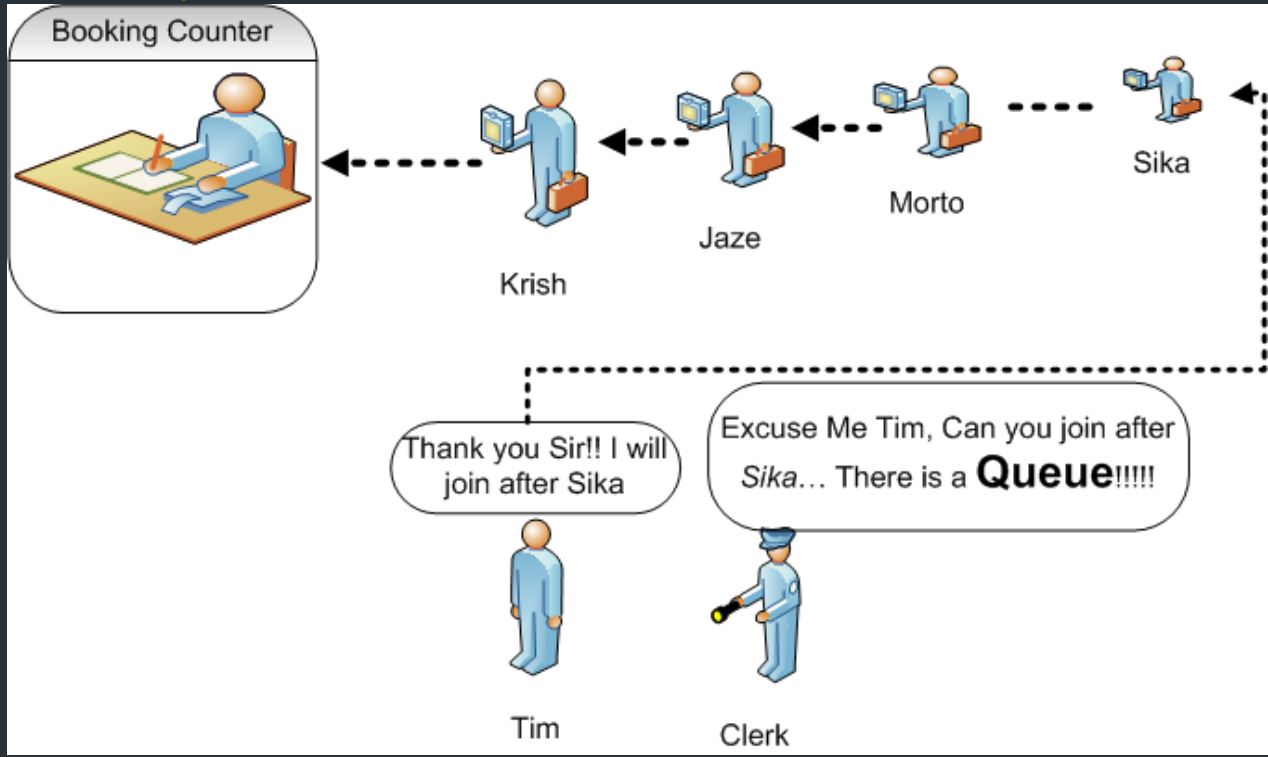
Graphical Representation of Queue



Graphical Representation of Queue



Graphical Representation of Queue



Array based implementation

- Create an Array, Q, with capacity N for storing its elements, (N=1000)
- Consider two variables *front* and *rear,* which will be initialized to -1. (i.e., queue is empty)
- Q[0] be the front of the queue and the queue grow from there

Array based implementation

- Enqueue(obj): If *rear < N-1* , it indicates that the array is not full then, increment *rear* by *1* and store the element at *Q[rear]*
- Enqueue(obj): If *rear == N-1,* then the array is fulland is said as queue overflow condition
- Dequeue(): The element at *Q[front]* will be deleted.

Array based implementation

- *Front*(): Returns the reference to the front element in the queue(i.e., *Q[front]* if queue is not empty)

- ***Time Complexity:***
  - Enqueue(Insertion): O(1)
  - Dequeue(Deletion):O(N) {Inefficient}
  - Dequeue(Deletion):O(1) { Array implementation of Queue in efficient manner(Circular)}

Isempty()

```
# define Q[N]
int front = rear=-1;
Void Isempty ()
{
If(rear==-1 && front== -1)
{
   return true
else
   return false;
}
```

Enqueue(int x)

```
Void enqueue (int x)
{
If(rear==N-1)
printf(“Overflow”);
else if(front==-1&&rear==-1)
{
Front=rear=0;
Q[rear]=x;
}
```

Enqueue(int x)

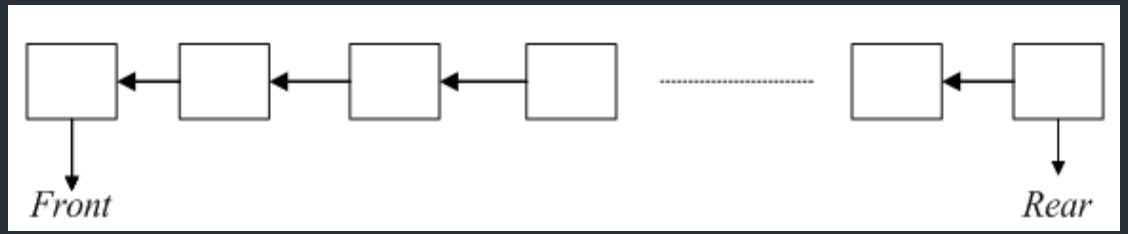
```
else
{
rear++;
Q[rear]=x;
}
}
```

Dequeue()

```
void dequeue()
{
if(front>rear)
printf(“Underflow”)
else
{
element=Q[front];
printf(Element);
front++;
}
```

Queue: Example

- Sample Illustration



- The queue grows towards the right from the left
- Two indices namely *front* and *rear* are used to traverse the elements.
- Consider a example of queue with size N=5

Queue: Example

- enqueue(5)

- enqueue(3)

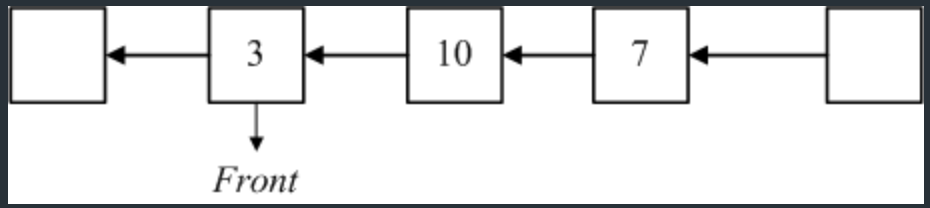
- enqueue(10)

Queue: Example

- Dequeue()

Output:5

- enqueue(7)

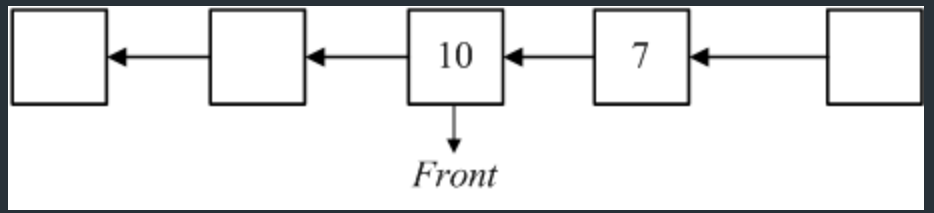


Queue: Example

- Dequeue()

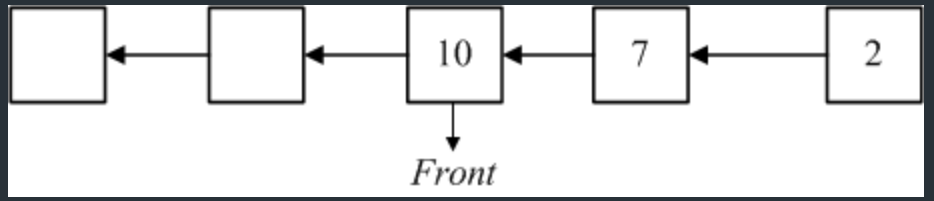
Output:3

- After performing the dequeue operation

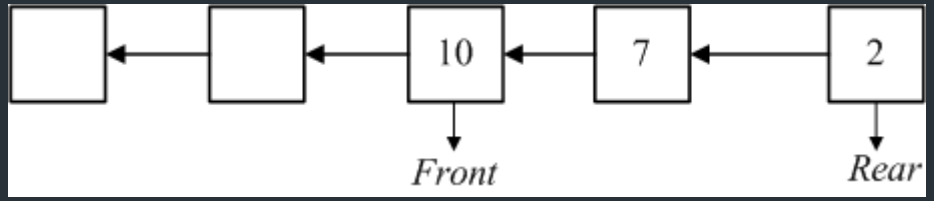


Queue: Example

- Enqueue(2):



- Now, rear == N-1, So the queue is full (i.e., Overflow), but vacant slots are available



Circular Queue

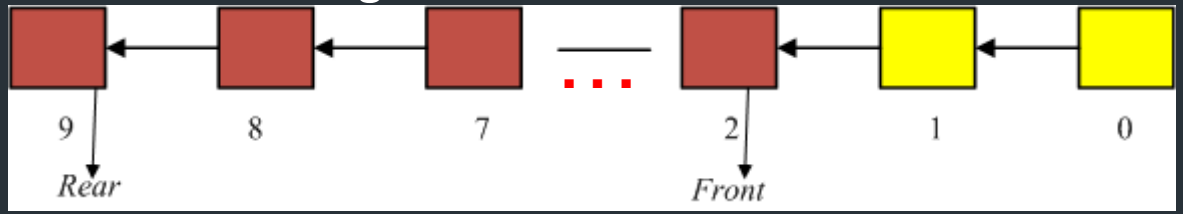
- Linear data structure in which the operations are based on FIFO(First In-First Out)
- It is also called as Ring buffer
- Instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure

Circular Queue

- The circular queue was devised to limit the memory wastage of the linear queue
- The new element is added at the very first position of the queue if the last is occupied and space is available.
- The time complexity is O(1) for all the operations

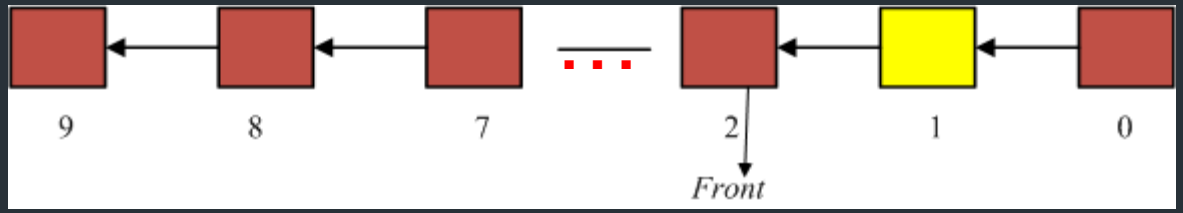
Operations: Circular Queue

- when the rear end fills up and front part of the array has empty slots, new insertions should go into the front end
- Consider the size *N* of queue Q as 10. The yellow coloured slots remain empty and brown coloured slots are having values in it.

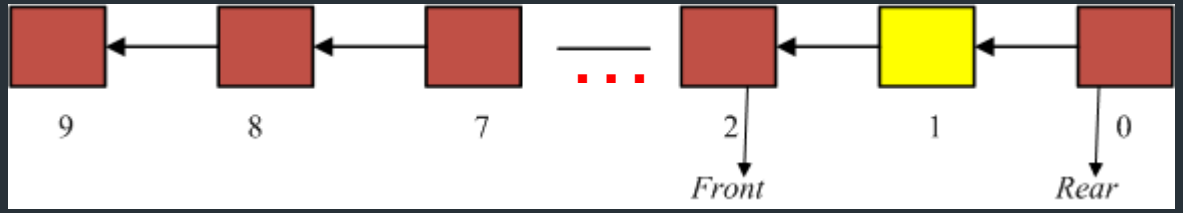


Operations: Circular Queue

- The next insertion goes into slot 0, and rear tracks it.

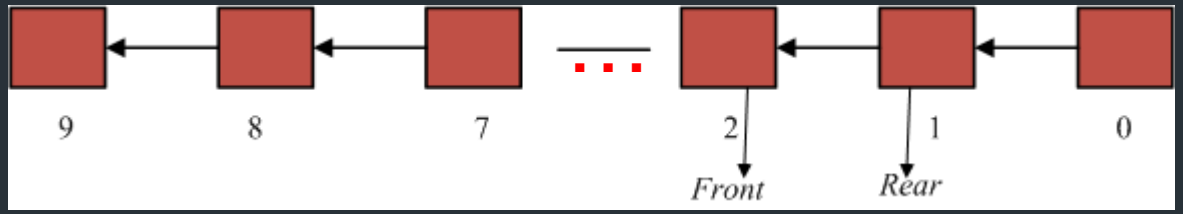


- After One Call to enqueue()



Operations: Circular Queue

- After One Call to enqueue()



- Now, the queue is full. So, memory wastage is avoided in circular queue, which leads to better performance.

Numeric for circular queue

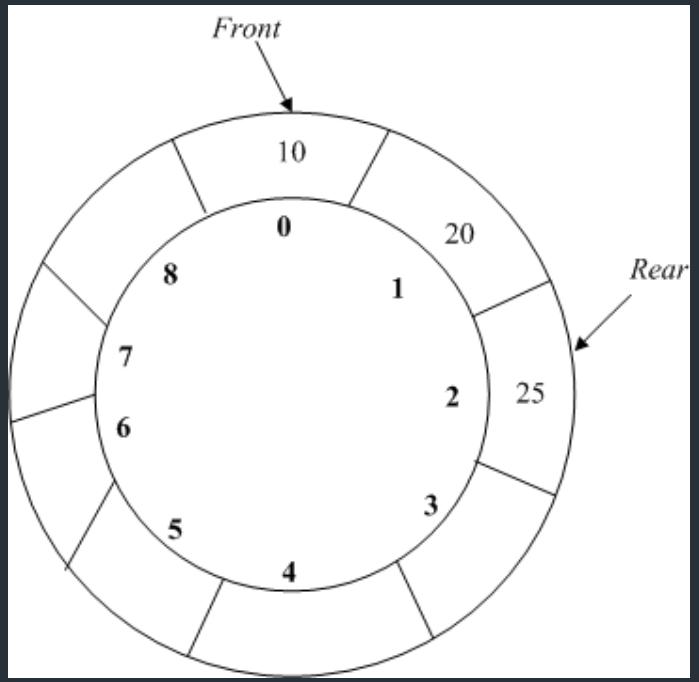
- ***front*** increases by (1 modulo size) after each dequeue( ):

  front= (front+1)%size

- ***rear*** increases by (1 modulo size) after each enqueue( ):

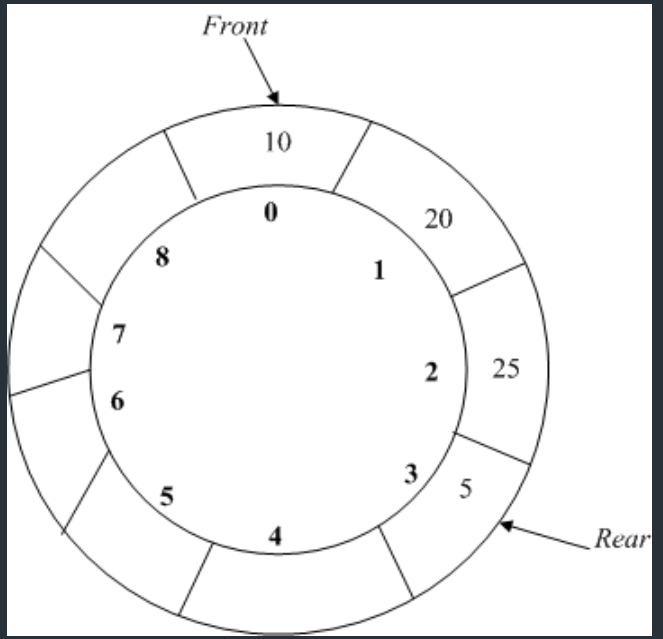
  rear= (rear+1)%size

Illustration of Circular queue



Enqueue(int x)

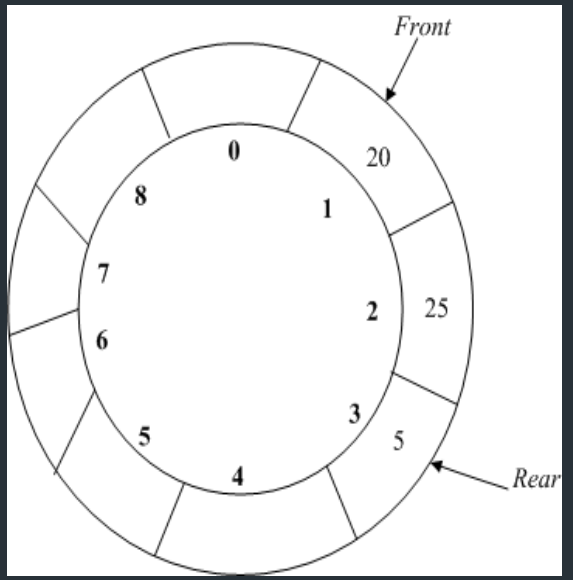
- **To add an element in circular queue, move *rear* clockwise**
- Then put it into Q[rear]



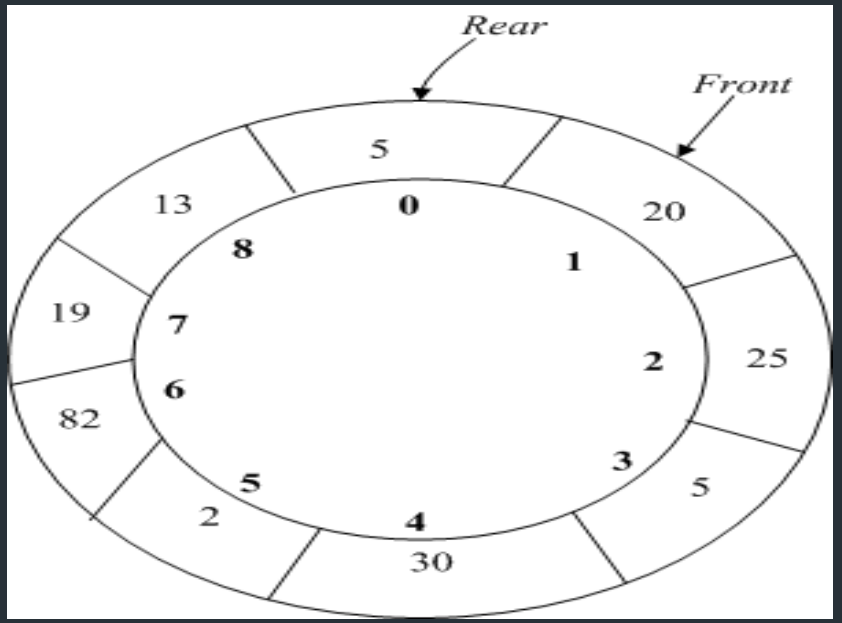
Dequeue()

- **To remove an element in circular queue, move *front* clockwise**

- Then extract it into Q[front]



Isfull()



enqueue(int x)

```
void enQueue (int value)
{
if ((rear+1 mod size)==front)
{
   printf("\nQueue is Full");
              return;
 }
  else if (front == -1 and rear==-1) /* Insert First
   Element */
   {        front = rear = 0;
              arr[rear] = value;
   }
```

enqueue(int x)

```
else   {
    rear= rear+1 mod size;
   arr[rear] = value;
 }
}
```

dequeue()

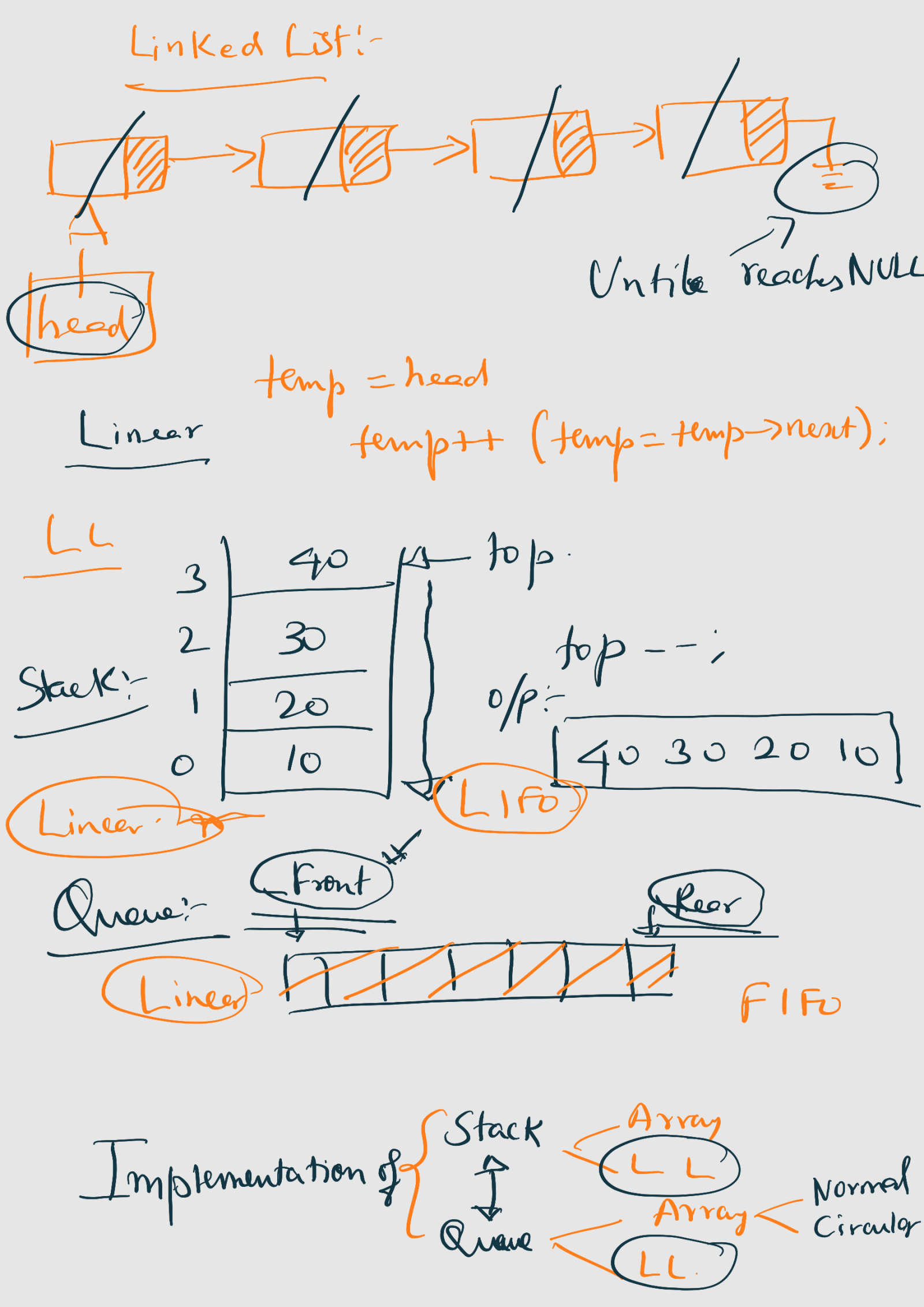
```
 int deQueue()
{
   if (front == -1 and rear==-1)
    {
   printf("\nQueue is Empty");
      }
Else if(front == rear)
{
   int data = Q[front];
front =rear= -1;
}
```

dequeue()

```
        else
            {
             data= arr[front];
              Front=Front+1 mod size;
    }
}
```

Applications

- **CPU Scheduling** is a process of determining which process will own CPU for execution while another process is on hold
  - First Come First Serve Scheduling
- In this type of algorithm, the process which requests the CPU gets the CPU allocation first. This scheduling method can be managed with a FIFO queue.


Linked List:-
A
head
Until reaches NULL
temp = head
Linear
temp++ (temp = temp->next);
LL
3
2
1
0
40
30
20
10
top
Stack:-
top--;
o/p:-
40 30 20 10
LIFO
Linear
Queue:-
Front
Rear
Linear
FIFO
Implementation of
Stack
Queue
Array
LL
Array
LL
Normal
Circular

① Queue implemented Stack

② Stack implement Queue
Enque O(1) Deque O(1)

Push O(1) Pop O(1)

Queue implemented using Stack.
FIFO LIFO

2 Stacks:-

I/p:- (10) 20 30 40 (50) 60

O/p expect Stack: Dequeue: 70, 80, 90

Enqueue- O(1)

Dequeue: 10 20

2N

Dequeue :-

$n-1$ → (a) Pop $n-1$ elements from S-1 & push it into S-2.

$O(1)$ → (b) pops out the top element of S-1

$n-1$ → (c) Pop 'n-1' elements from S-2 & push it in S-1

$O(N)$

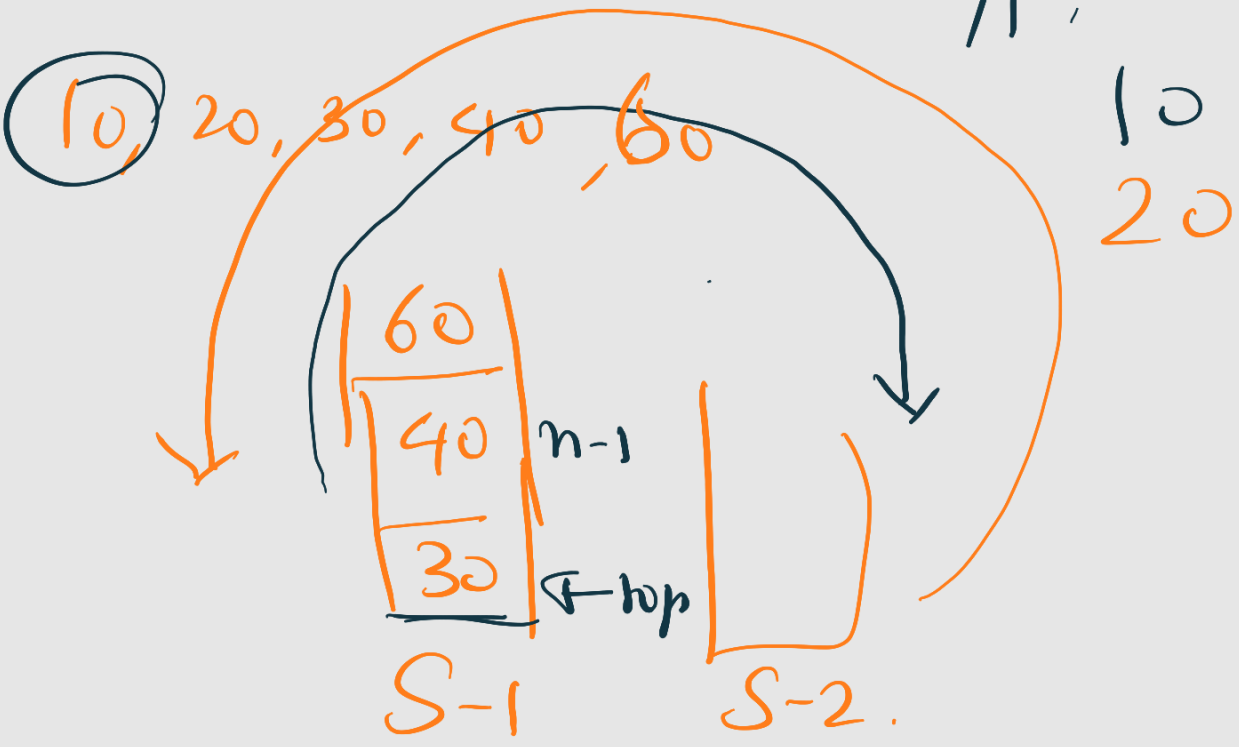
Enqueue :-

$O(1)$ ← (1) Push it in S-1

o/p :-

10

20



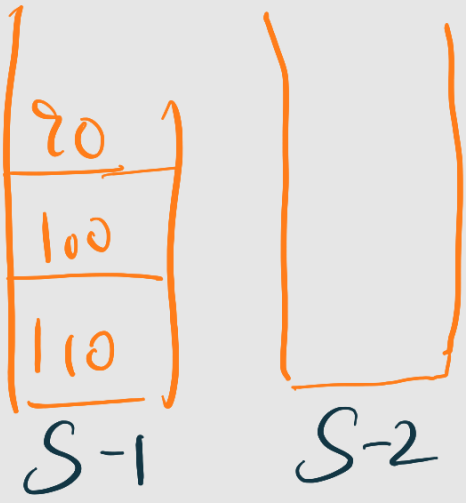
Enqueue operation as costlier - O(N)

Dequeue - O(1)

I/p:- 90, 100, 110

Ensure:-

After completion of 1 Enqueue, always S-2 is empty



Start my Enqueue:

1. Pop all elements from S-1 & push it in S-2..
2. In S-1, push the new element
3. Pop out all elements from S-2 & push in S-1

Dequeue:

① Pop out from S-1 → O(1).