Data Structures and Algorithms

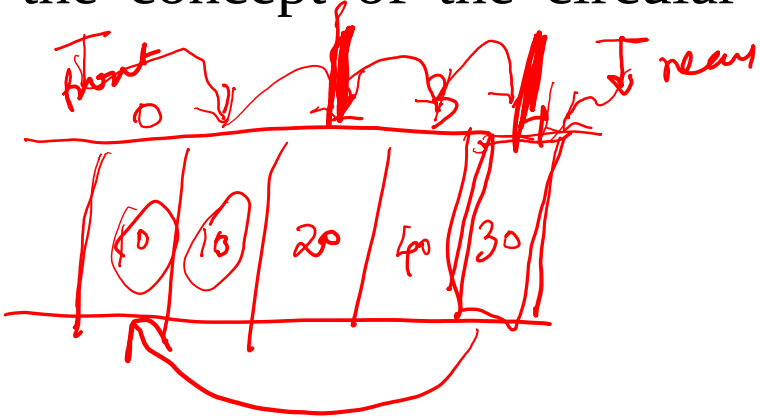
Module 3: Circular Queue

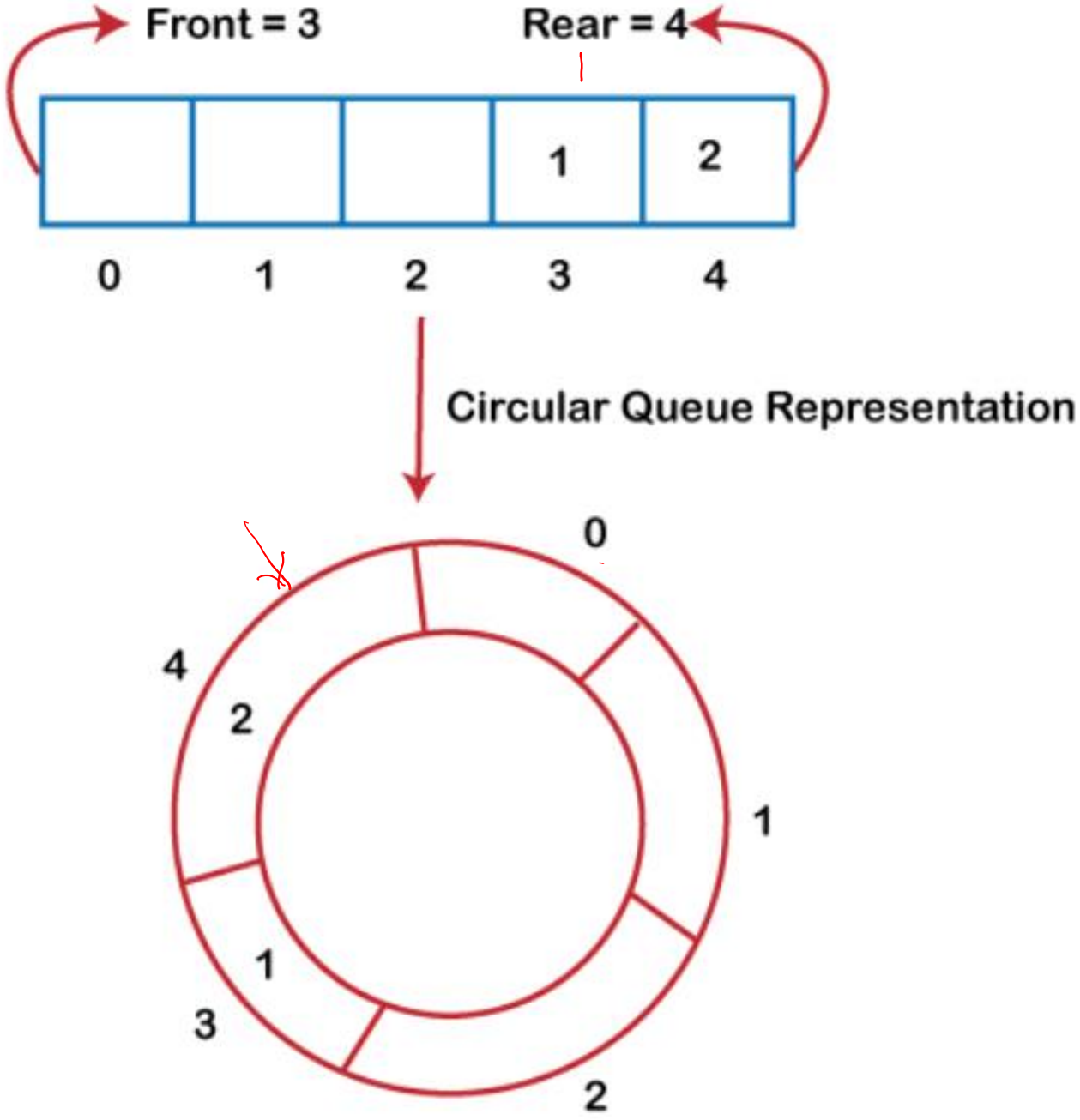
Indian Institute of Information Technology Sri City, Chittoor

Circular Queue

Why was the concept of the Circular Queue introduced?

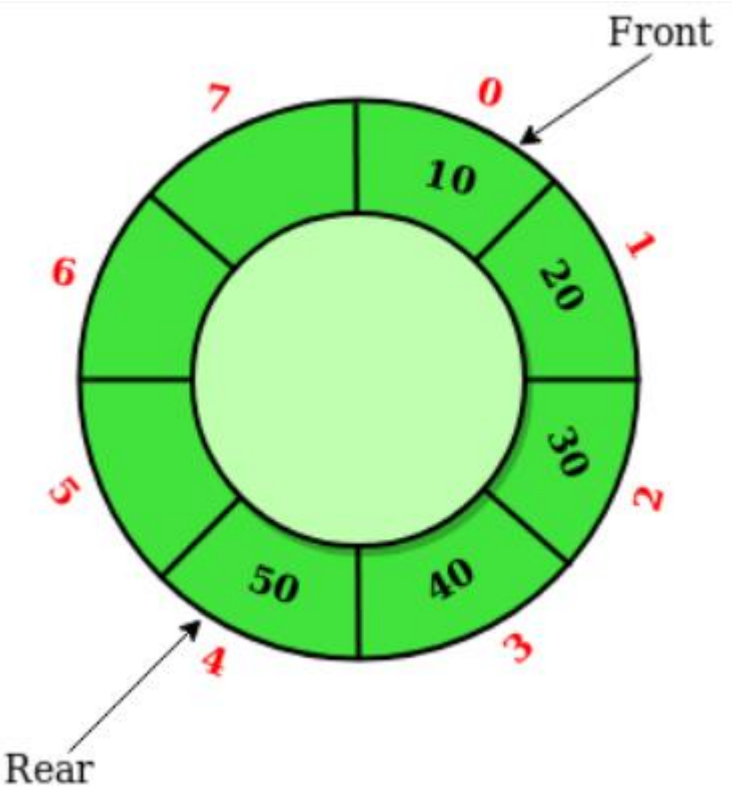
- There was one limitation in the array implementation of *Queue.*
- ➢ If the rear reaches to the end position of the Queue then there might be possibility that *some vacant spaces are left in the beginning* which cannot be utilized.
- So, to overcome such limitations, the concept of the circular queue was introduced.



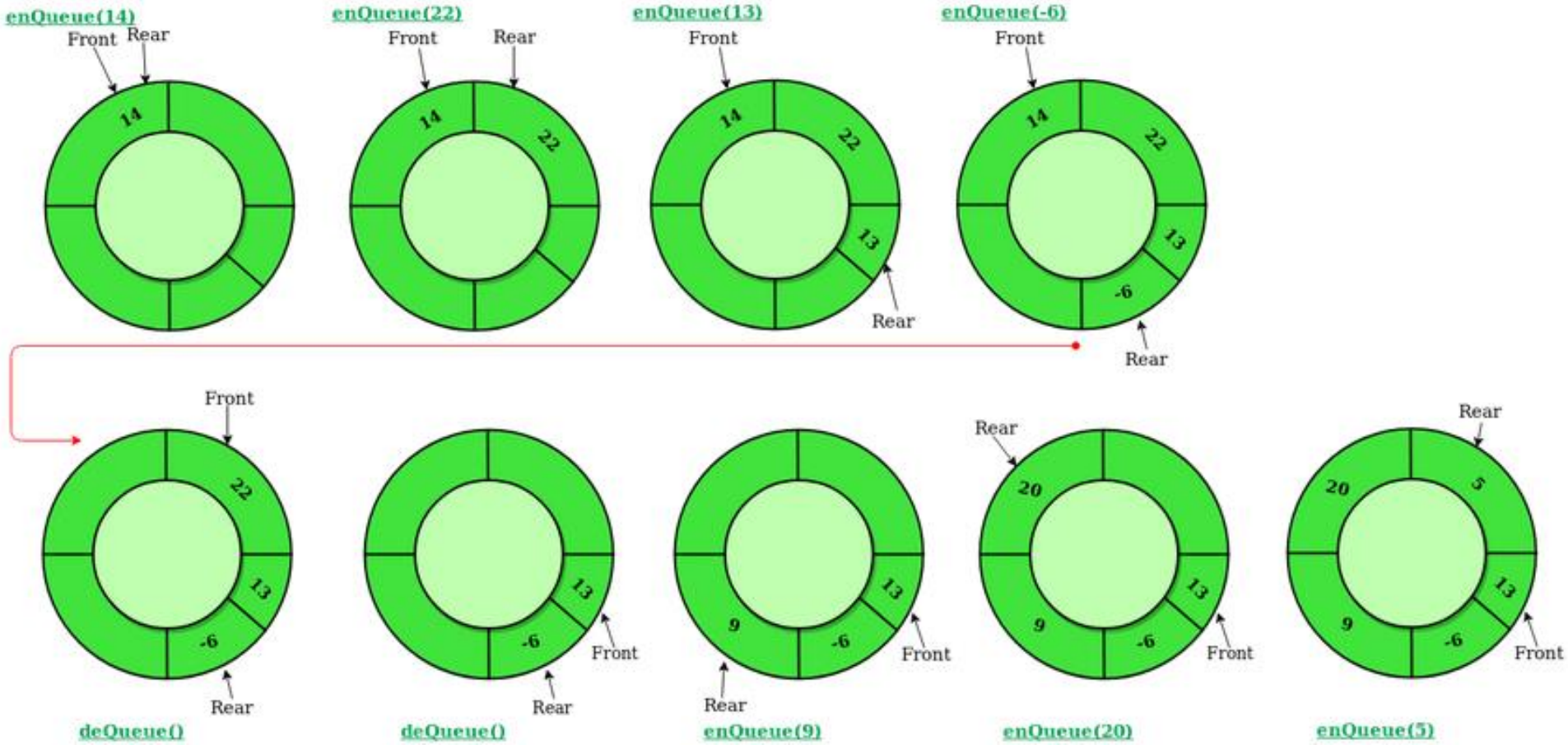

Front = 3
Rear = 4
1
2
0
1
2
3
4
Circular Queue Representation
0
1
2
3
4
1
2

Circular Queue

- Circular Queue is a 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. It is also called **'Ring Buffer'**.



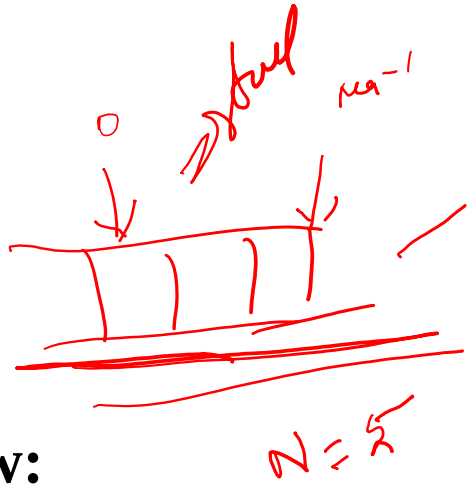
- In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.



Operations on Circular Queue

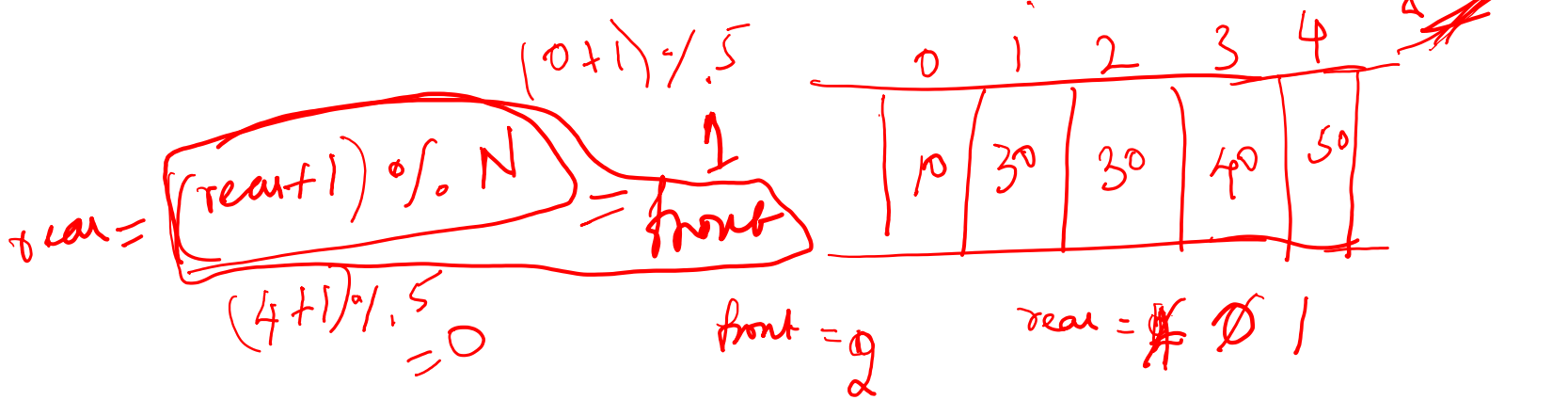
- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.
- **enQueue(value):** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at *Rear position*.
- **deQueue():** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from *Front position.*

Enqueue Operation



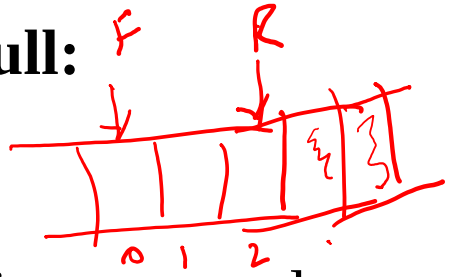
The steps of enqueue operation are given below:

- First, check whether the **Queue is full or not.**
- Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
- When we insert another new element, the rear gets incremented, i.e., ***rear=rear+1***.

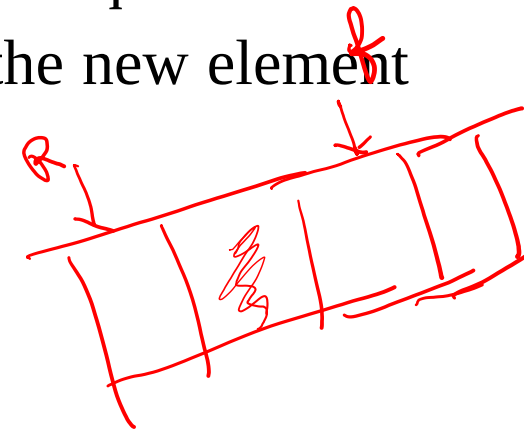


Scenarios for inserting an element

There are two scenarios in which queue is not full:



- **If rear != max - 1**, then rear will be incremented to **mod(maxsize)** and the new value will be inserted at the rear end of the queue.
- **If front != 0 and rear = max - 1**, it means that queue is not full, then set the value of rear to 0 and insert the new element there.



Contd...

There are two cases in which the element cannot be inserted:

- When **front == 0 && rear = max-1**, which means that front is at the first position of the Queue and rear is at the last position of the Queue.
- **front == rear + 1;**

Algorithm: Enqueue Operation

Step 1: IF (REAR+1)%MAX = FRONT

Write " OVERFLOW "

Goto step 4

[End OF IF]

Step 2: IF FRONT = -1 and REAR = -1

SET FRONT = REAR = 0

ELSE IF REAR = MAX - 1 and FRONT ! = 0

SET REAR = 0

ELSE

SET REAR = (REAR + 1) % MAX

[END OF IF]

Step 3: SET QUEUE[REAR] = VAL

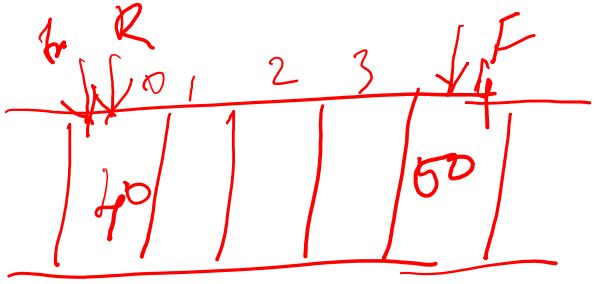
Step 4: EXIT

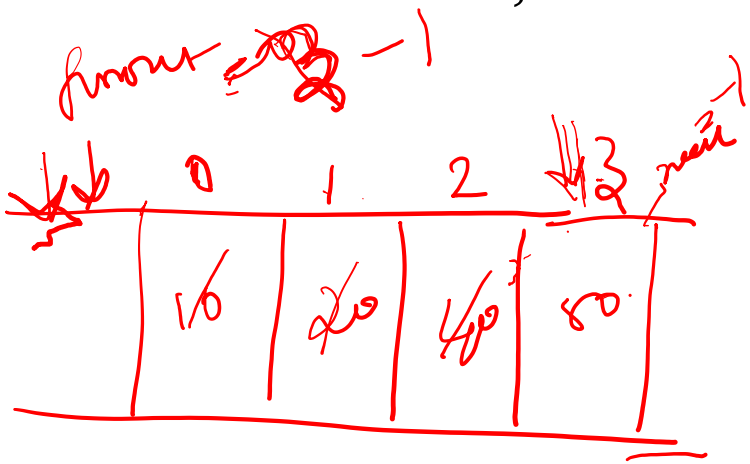
Time Complexity : O(1)

Dequeue Operation

The steps of dequeue operation are given below:

- First, we check whether the **Queue is empty or not.** If the queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the ***value of front gets incremented by 1.***
- If there is only one element left which is to be deleted, then the ***front and rear are reset to -1.***





Algorithm

Step 1: IF FRONT = -1

Write " UNDERFLOW "

Goto Step 4

[END of IF]

Step 2: SET VAL = QUEUE[FRONT]

Step 3: IF FRONT = REAR

SET FRONT = REAR = -1

ELSE

IF FRONT = MAX -1

SET FRONT = 0

ELSE

SET FRONT = FRONT + 1

[END of IF]

[END OF IF]

Step 4: EXIT

Time Complexity : O(1)

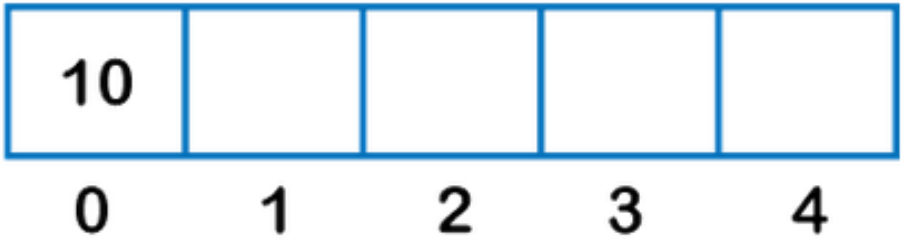
The enqueue and dequeue operation through the diagrammatic representation

Initially

Front = -1

Rear = -1

Insert 10

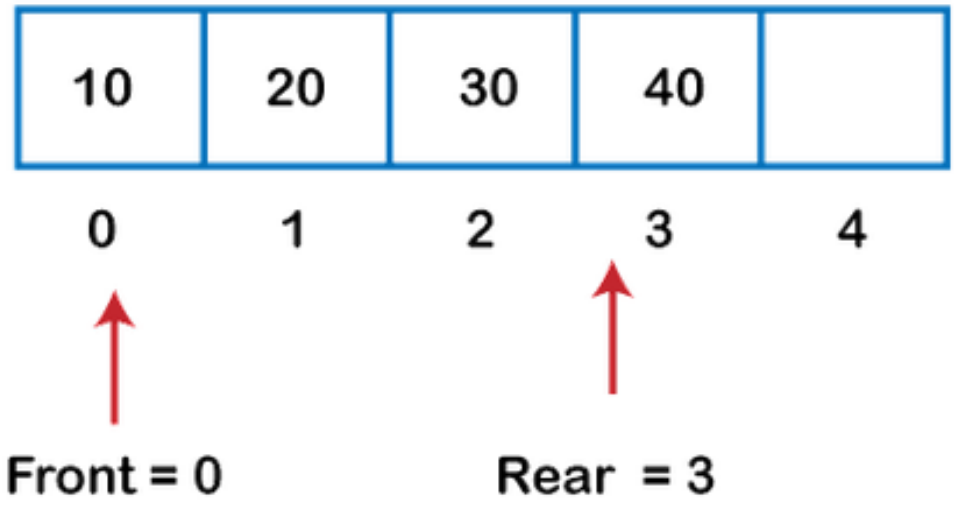


Front = 0

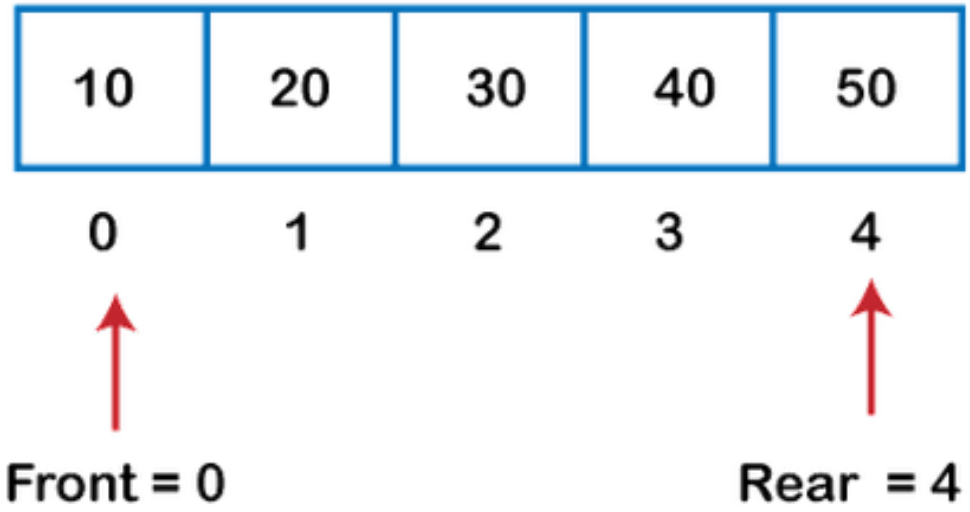
Rear = 0

Contd...

Insert 20, 30, and 40

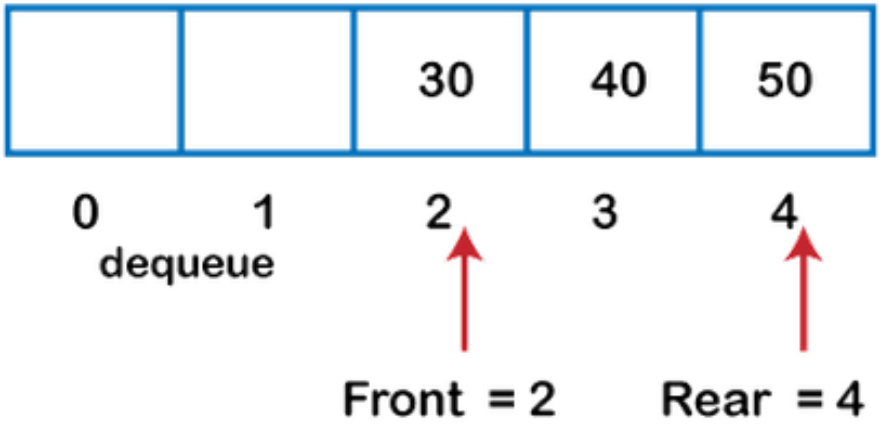


Insert 50

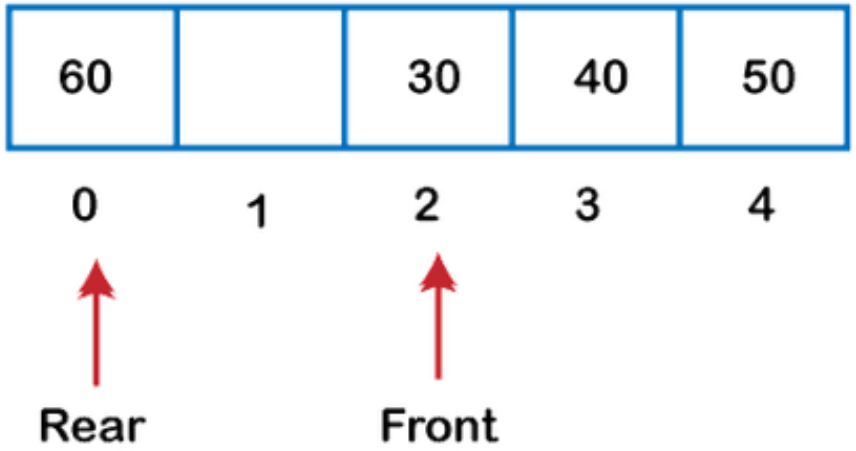


Contd...

Delete 10 and 20

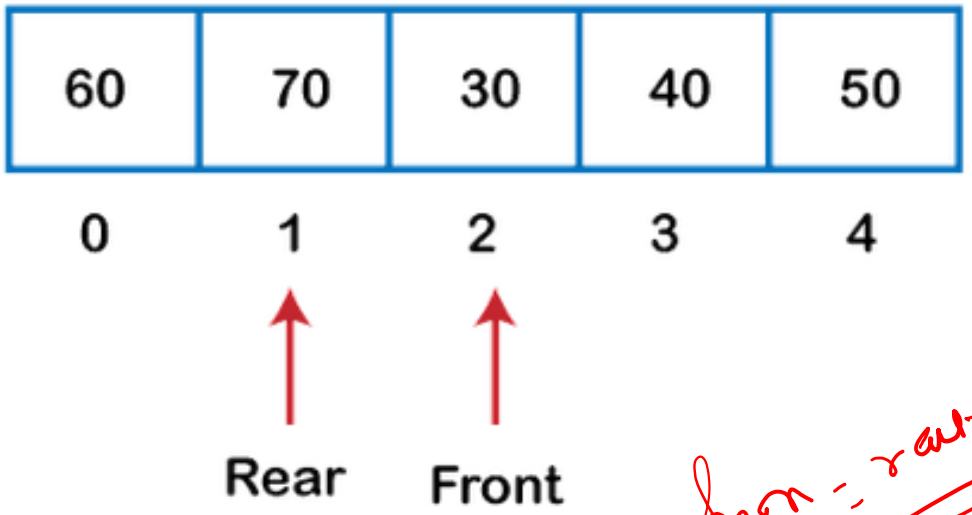


Insert 60



Contd...

Insert 70



Implementation of Insertion in circular queue using Array

```c
#include <stdio.h>

# define max 6
int queue[max];  // array declaration
int front=-1;
int rear=-1;
// function to insert an element in a circular queue
void enqueue(int element)
{
   if(front==-1 && rear==-1)   // condition to check queue is empty
   {
      front=0;
      rear=0;
      queue[rear]=element;
   }
   else if((rear+1)%max==front)  // condition to check queue is full
   {
      printf("Queue is overflow..");
   }
   else
   {
      rear=(rear+1)%max;       // rear is incremented
      queue[rear]=element;     // assigning a value to the queue at the rear position.
   }
}
```

Implementation of Deletion in circular queue using Array

```
// function to delete the element from the queue
int dequeue()
{
   if((front==-1) && (rear==-1))  // condition to check queue is empty
   {
      printf("\nQueue is underflow..");
   }
 else if(front==rear)
{
  printf("\nThe dequeued element is %d", queue[front]);
  front=-1;
  rear=-1;
}
else
{
   printf("\nThe dequeued element is %d", queue[front]);
  front=(front+1)%max;
}
}
```

Display the elements using Array

```
void display()
{
   int i=front;
   if(front==-1 && rear==-1)
   {
      printf("\n Queue is empty..");
   }
   else
   {
      printf("\nElements in a Queue are :");
      while(i<=rear)
      {
         printf("%d,", queue[i]);
         i=(i+1)%max;
      }
   }
}
```

Implementation of circular queue using linked list

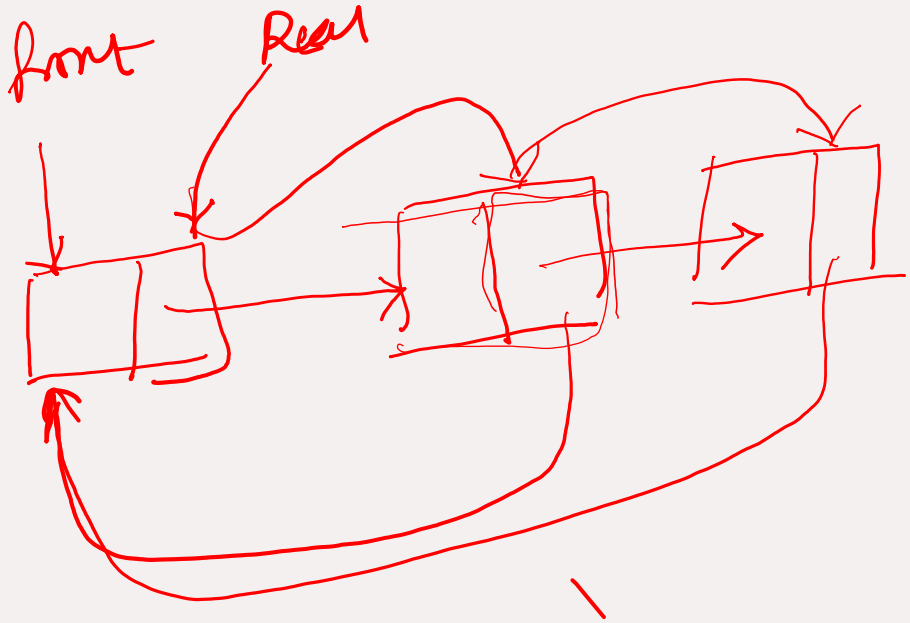
- As we know that linked list is a linear data structure that stores two parts, i.e., **data part and the address part** where address part contains the address of the next node.
- Here, linked list is used to implement the circular queue; therefore, the linked list follows the properties of the Queue.
- When we are implementing the circular queue using linked list then both the ***enqueue and dequeue*** operations take ***O(1)*** time.

Declaration of struct type node

```
struct node
{
    int data;
    struct node *next;
};
struct node *front=-1;
struct node *rear=-1;
```

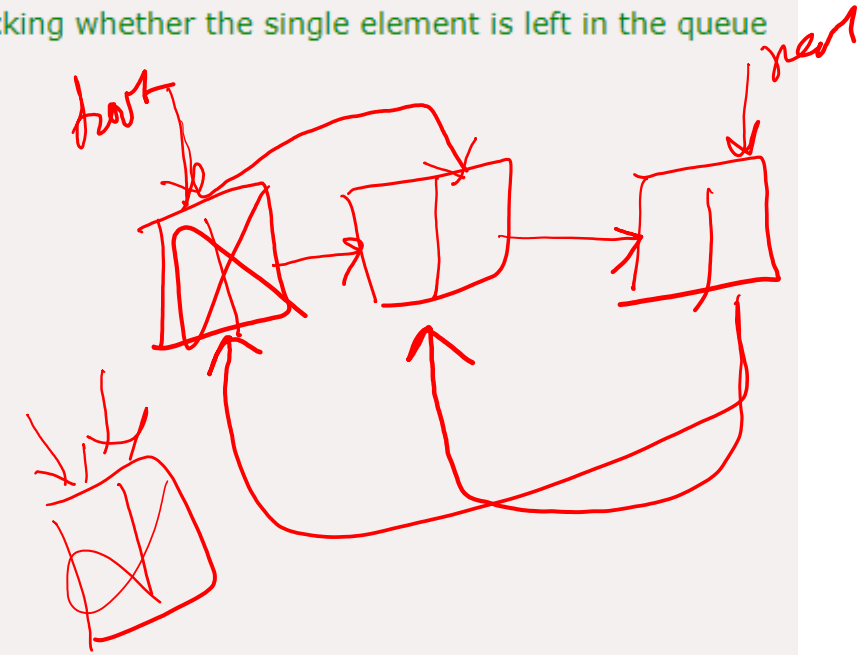
Insert an element in the Queue

```
void enqueue(int x)
{
   struct node *newnode;  // declaration of pointer of struct node type.
   newnode=(struct node *)malloc(sizeof(struct node));  // allocating the memory to the newnode
   newnode->data=x;
   newnode->next=0;
   if(rear==-1)  // checking whether the Queue is empty or not.
   {
      front=rear=newnode;
      rear->next=front;
   }
   else
   {
      rear->next=newnode;
      rear=newnode;
      rear->next=front;
   }
}
```



Delete an element from the Queue

```
void dequeue()
{
   struct node *temp;   // declaration of pointer of node type
   temp=front;
   if((front==-1)&&(rear==-1))  // checking whether the queue is empty or not
   {
      printf("\nQueue is empty");
   }
   else if(front==rear)  // checking whether the single element is left in the queue
   {
      front=rear=-1;
      free(temp);
   }
   else
   {
      front=front->next;
      rear->next=front;
      free(temp);
   }
}
```



Get the front element of the Queue

```
int peek()
{
   if((front==-1) &&(rear==-1))
   {
      printf("\nQueue is empty");
   }
   else
   {
      printf("\nThe front element is %d", front->data);
   }
}
```

Display all the elements of the queue

```
void display()
{
   struct node *temp;
   temp=front;
   printf("\n The elements in a Queue are : ");
   if((front==-1) && (rear==-1))
   {
      printf("Queue is empty");
   }

   else
   {
      while(temp->next!=front)
      {
         printf("%d,", temp->data);
         temp=temp->next;
      }
      printf("%d", temp->data);
   }
}
```

Thank You !!!