

**SCHOOL OF COMPUTING**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**UNIT – IV –DATA STRUCTURES– SCSA1203**

**Basic Queue Operations - Representation of a Queue using array - Applications of Queues - Round robin Algorithm - Enqueue - Dequeue - Circular Queues - Priority Queues**

**4.0 QUEUE**

A Queue is a **linear structure** which follows a particular order in which the operations are performed. It is an **abstract data structure** similar to stack.

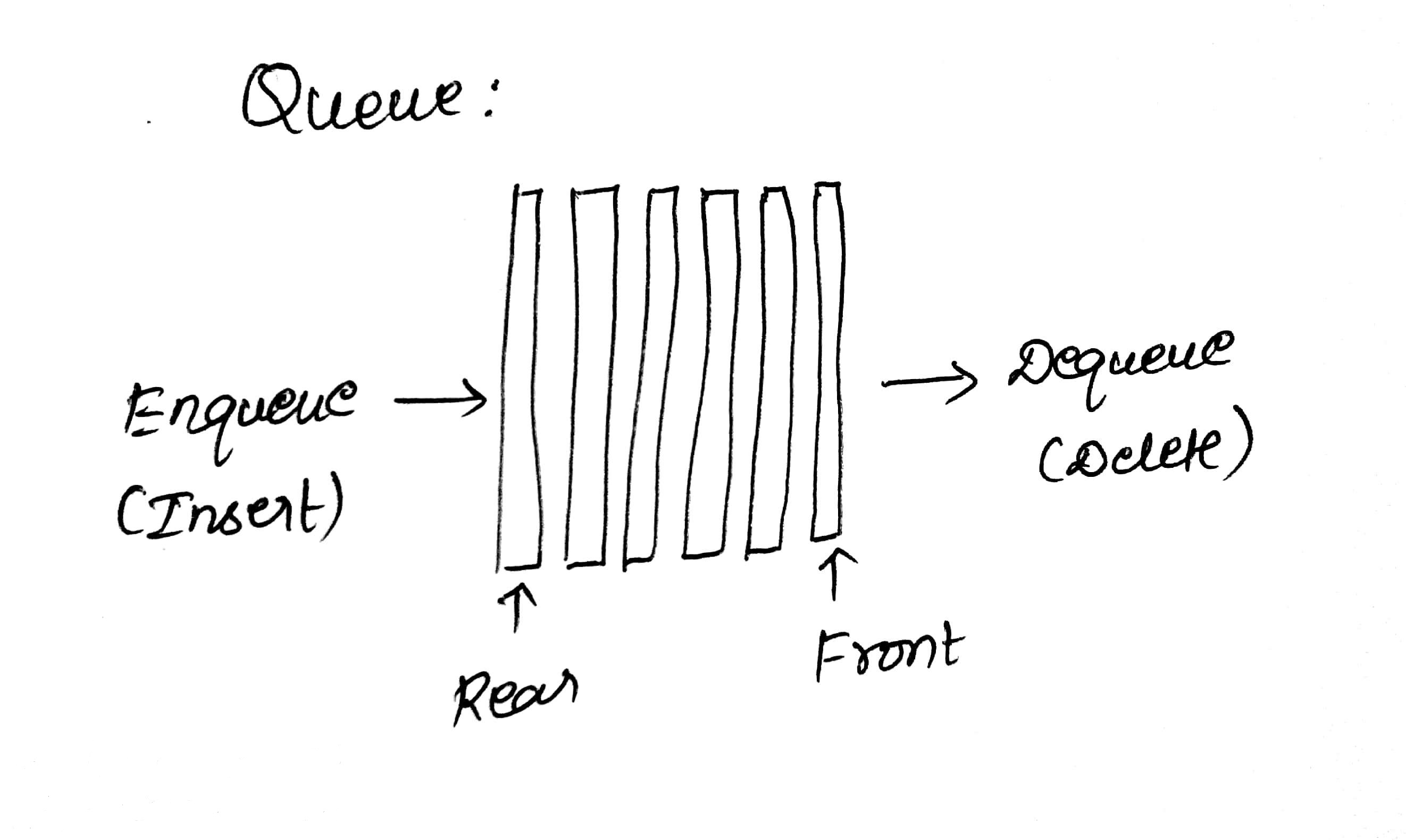
Queue is consisting of two ends such as,

i)**Enqueue** – Using this we can insert the data

ii)**Dequeue** – Using this we can delete the data

It follows two methodology consists of **FIFO** – First In First Out and **LILO** – Last In Last Out

It maintains two pointers consists of **FRONT** and **REAR**



**Fig 4.1 Structure of a queue**

**4.1 EXAMPLES**

* A queue of people at ticket-window: The person who comes first gets the ticket first. The person who is coming last is getting the tickets in last. Therefore, it follows first-in-first-out (FIFO) strategy of queue.
* Luggage checking machine: Luggage checking machine checks the luggage first that comes first. Therefore, it follows FIFO principle of queue.
* Patients waiting outside the doctor's clinic: The patient who comes first visits the doctor first, and the patient who comes last visits the doctor last. Therefore, it follows the first-in-first-out (FIFO) strategy of queue.



**Fig 4.2 Schematic representation of a queue**

**4.2 IMPLEMENTATION OF QUEUE**

The queue will be implemented in two ways,

1. Array
2. Linked list
3. Stack

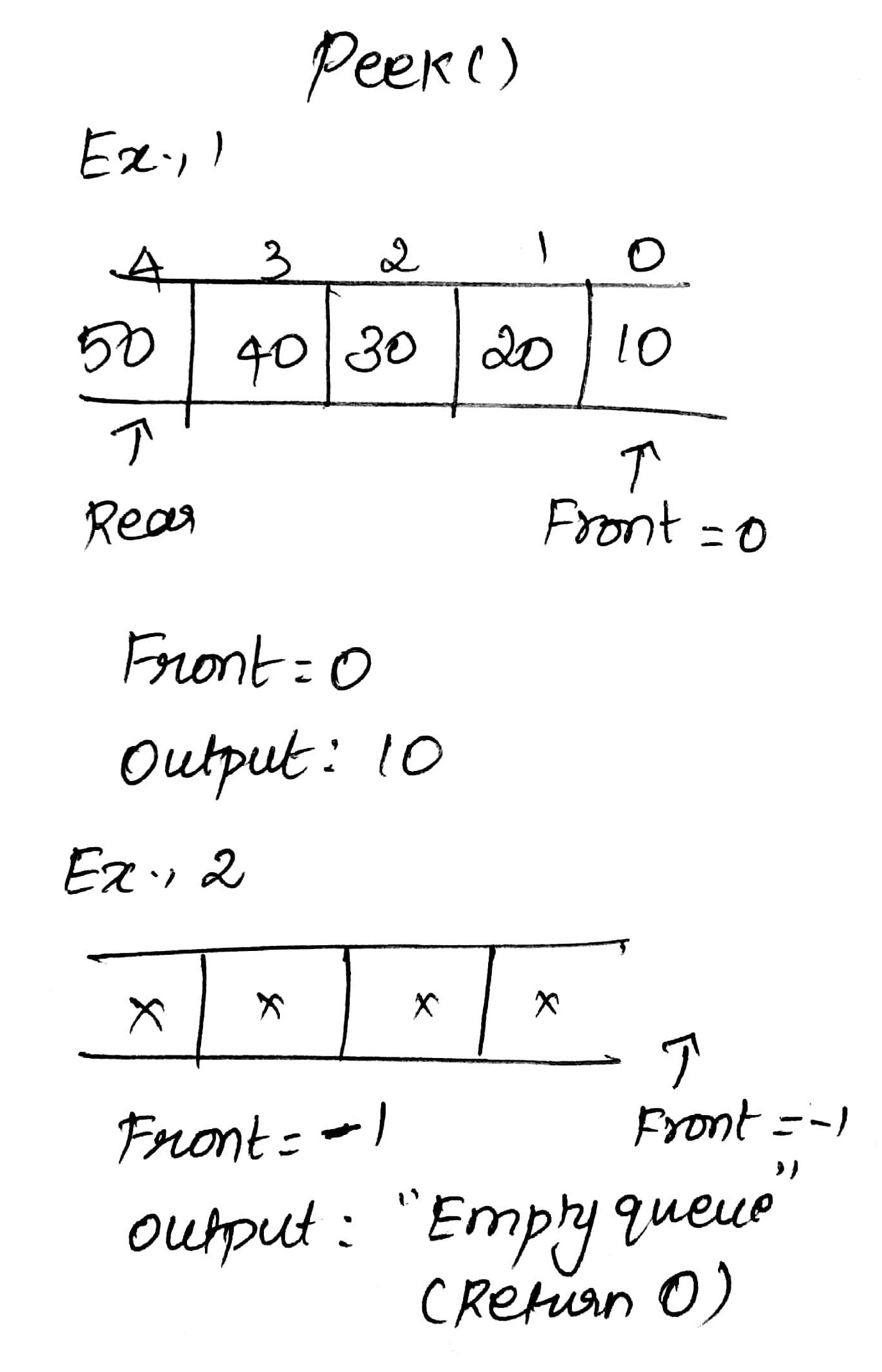
**BASIC OPERATIONS IN QUEUE**

1. peek() – Front
2. isfull() – Check queue is full or not
3. isempty() – Check queue is empty or not
4. Enqueue – Add at rear position
5. Dequeue – Add at front position
6. **Peek()**

It helps to access or view front element of the queue.

Algorithm peek (front, queue[])

1. If front = -1
2. Return 0
3. Else
4. Return queue[front]



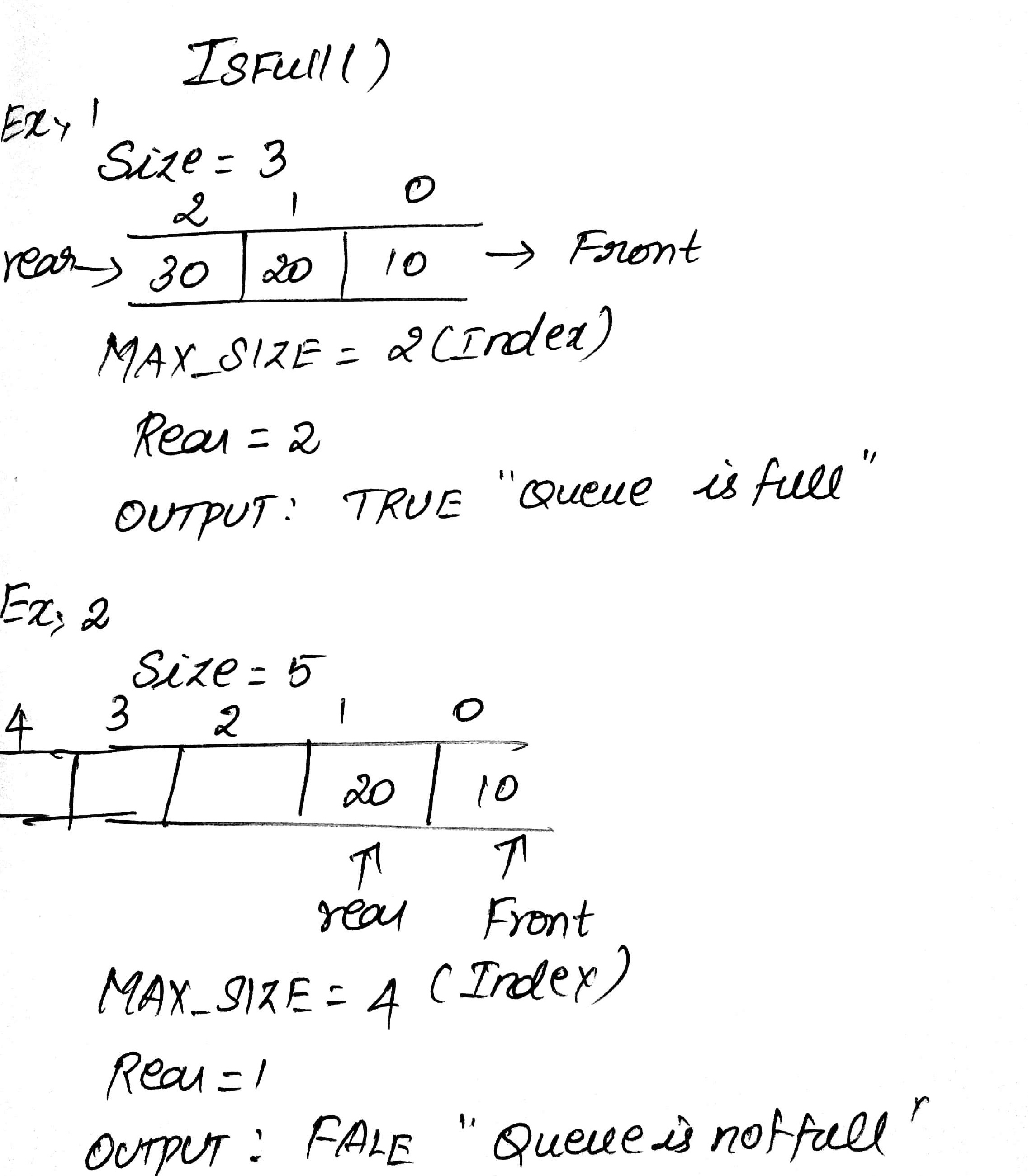
**Fig. 4.3 Example for Peek() operation**

1. **Isfull()**

It is used to check whether the queue is full or not.

Algorithm isfull (rear, queue[])

1. If rear = MAX\_SIZE
2. Return TRUE
3. Else
4. Return FALSE



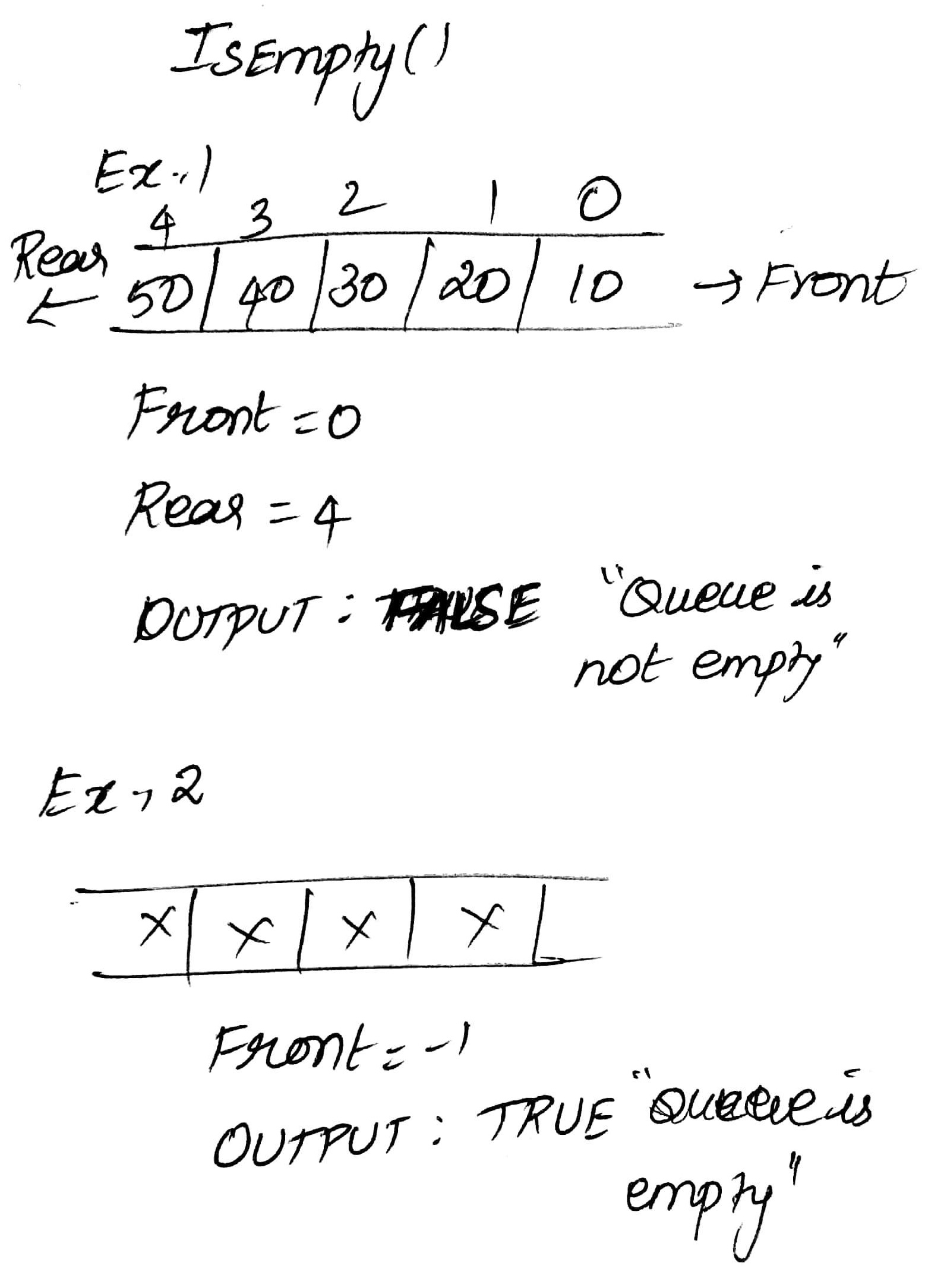
**Fig. 4.4 Example for IsFull() operation**

1. **Isempty()**

It helps to check whether the queue is empty or not.

Algorithm isempty (queue[], front)

1. If front <=-1
2. Return TRUE
3. Else
4. Return FALSE



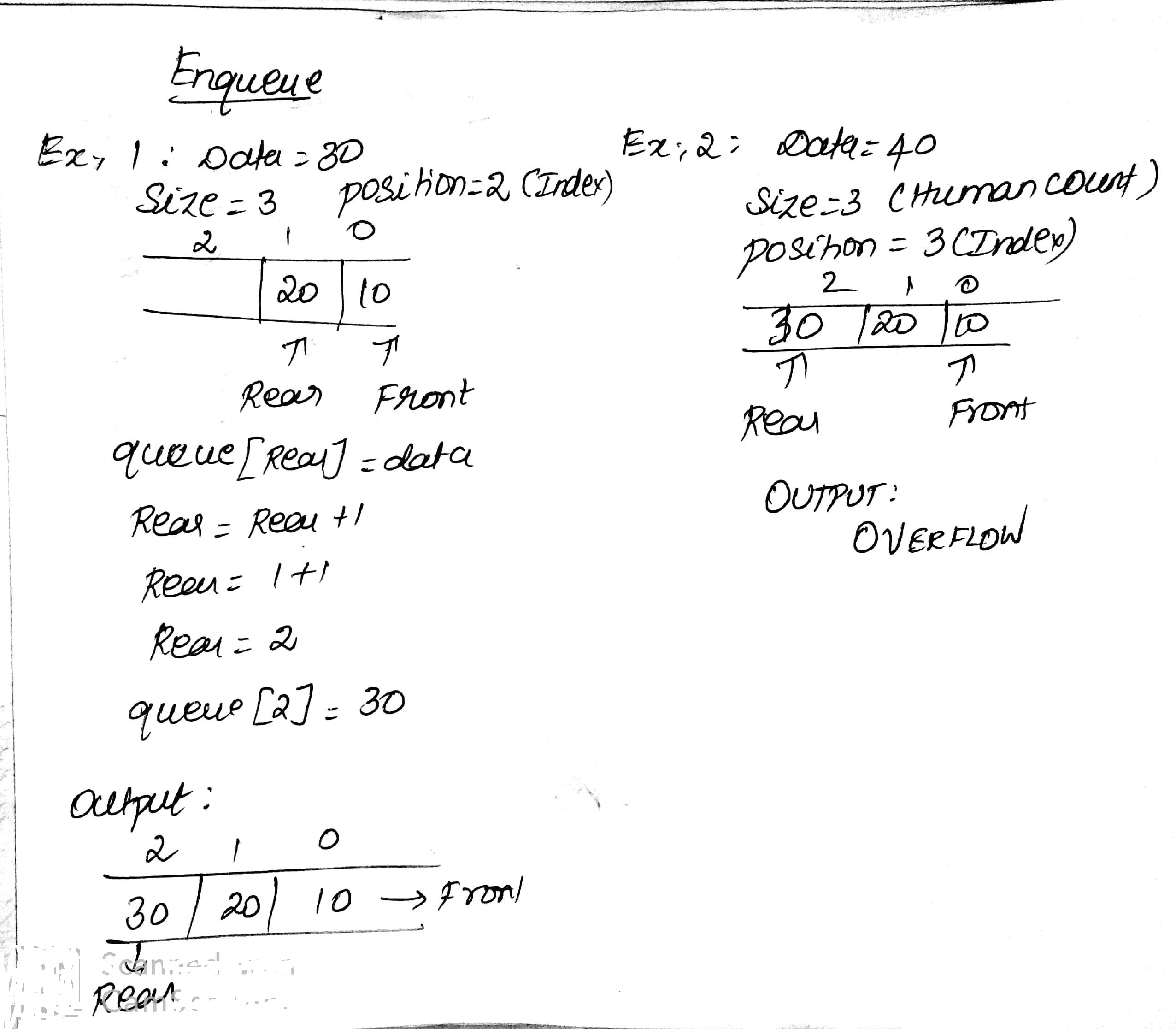
**Fig. 4.5 Example for IsEmpty() operation**

1. **Enqueue:**

The operation of inserting an element inside the queue at rear position called enqueue.

Algorithm enqueue (data, front, rear, queue[])

1. If isfull (queue[])
2. Return OVERFLOW
3. Else
4. Rear = Rear + 1
5. queue[Rear] = data



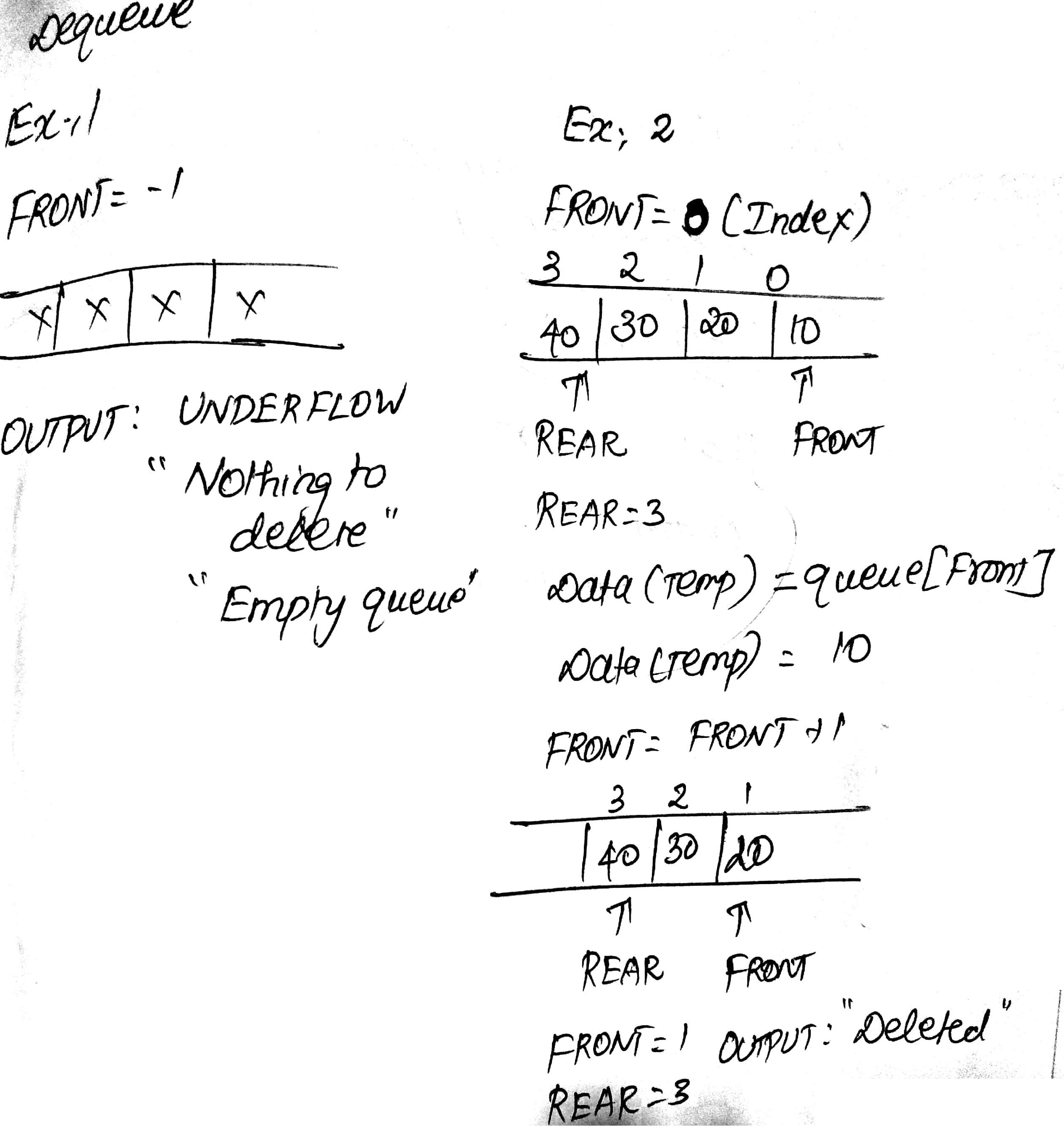
**Fig. 4.6 Example for Enqueue() operation**

1. **Dequeue:**

The operation of deleting an element from the queue at front position called dequeue.

Algorithm dequeue (data, front, rear, queue[])

1. If isempty (queue[])
2. Return UNDERFLOW
3. Else
4. Data = queue[front]
5. Front = Front + 1



**Fig. 4.7 Example for Dequeue() operation**

**4.3 REPRESENTATION OF QUEUE USING ARRAY**

Queue can be implemented using array by restricting the array to follow First In First Out while inserting, deleting and displaying an element.

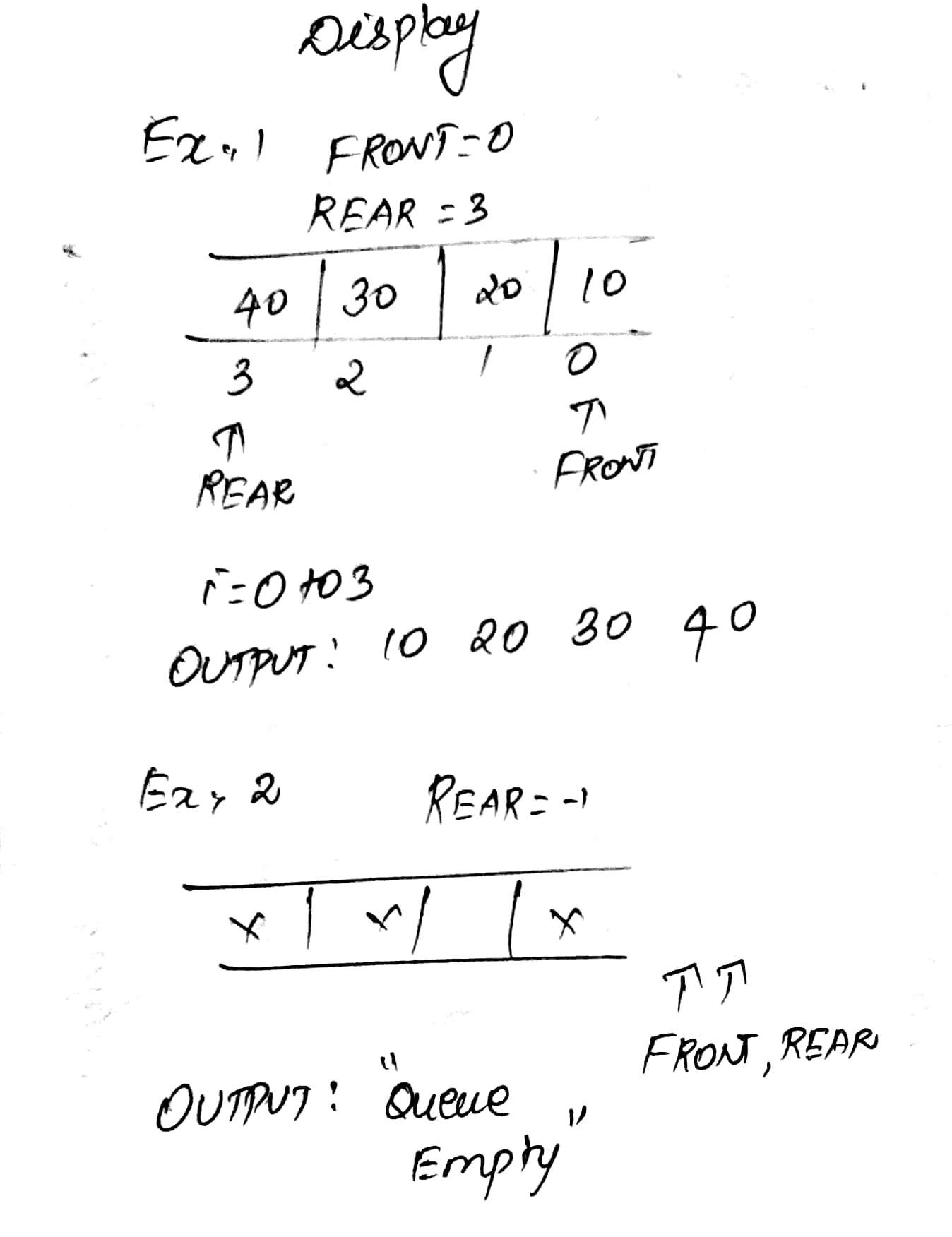
(For enqueue and dequeue refer previous topic)

**4.3.1 Display**

The operation is used to display all the element present in the queue.

Algorithm display (queue[], front, rear)

1. If rear == -1
2. Return 0
3. Else
4. For i = front to rear
5. Print queue[i]



**Fig. 4.8 Example for Display() operation**

**4.4 APPLICATION OF QUEUES**

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.
4. In recognizing palindrome.
5. In shared resources management.
6. Keyboard buffer.

**4.5 ROUND ROBIN ALGORITHM: (**For algorithm circular queue)

Round robin algorithm is a CPU scheduling algorithm that is effectively used to schedule or order the processes inside the queue. The process residing in CPU may be in one of the five states given below,

1. **Stack** – Process creation
2. **Ready** – Process in ready queue
3. **Running** – Process running currently
4. **Waiting** – Process in waiting queue
5. **Terminate** – Process completed

In Round robin algorithm each process can reside in the CPU for only particular amount of time called **time quantum**.

Each process after completing its **burst time** fully will come out of the queue, until that it will be lined inside the queue again and again in a circular manner.

**Round Robin algorithm is an application for circular queue**.

Example: **Given are the processes with arrival time and burst time. Find the waiting time and turnaround time.**

**Time quantum = 2 seconds**

|  |  |  |
| --- | --- | --- |
| **PROCESS** | **ARRIVAL TIME** | **BURST TIME** |
| P1 | 0 | 4 |
| P2 | 1 | 5 |
| P3 | 2 | 2 |
| P4 | 3 | 3 |
| P5 | 4 | 1 |

**4.5.1 SOLUTION:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **PROCESS NO.** | **ARRIVAL TIME(AT)** | **BURST TIME(BT)** | **REMAINING BURST TIME(RBT)** | | | **COMPLETION TIME(CT)** | **WAITING TIME(WT)**  **(CT-BT)** | **TURN ARROUND TIME (TAT)**  **(CT-AT)** |
| P1 | 0 | 4 | 2 | 2 | 0 | 11 | 7 | 11 |
| P2 | 1 | 5 | 2 | 2 | 1 | 15 | 10 | 14 |
| P3 | 2 | 2 | 2 | 0 | 0 | 6 | 4 | 4 |
| P4 | 3 | 3 | 2 | 1 | 0 | 14 | 11 | 11 |
| P5 | 4 | 1 | 1 | 0 | 0 | 9 | 8 | 5 |

**4.5.2 Grant Chart:**

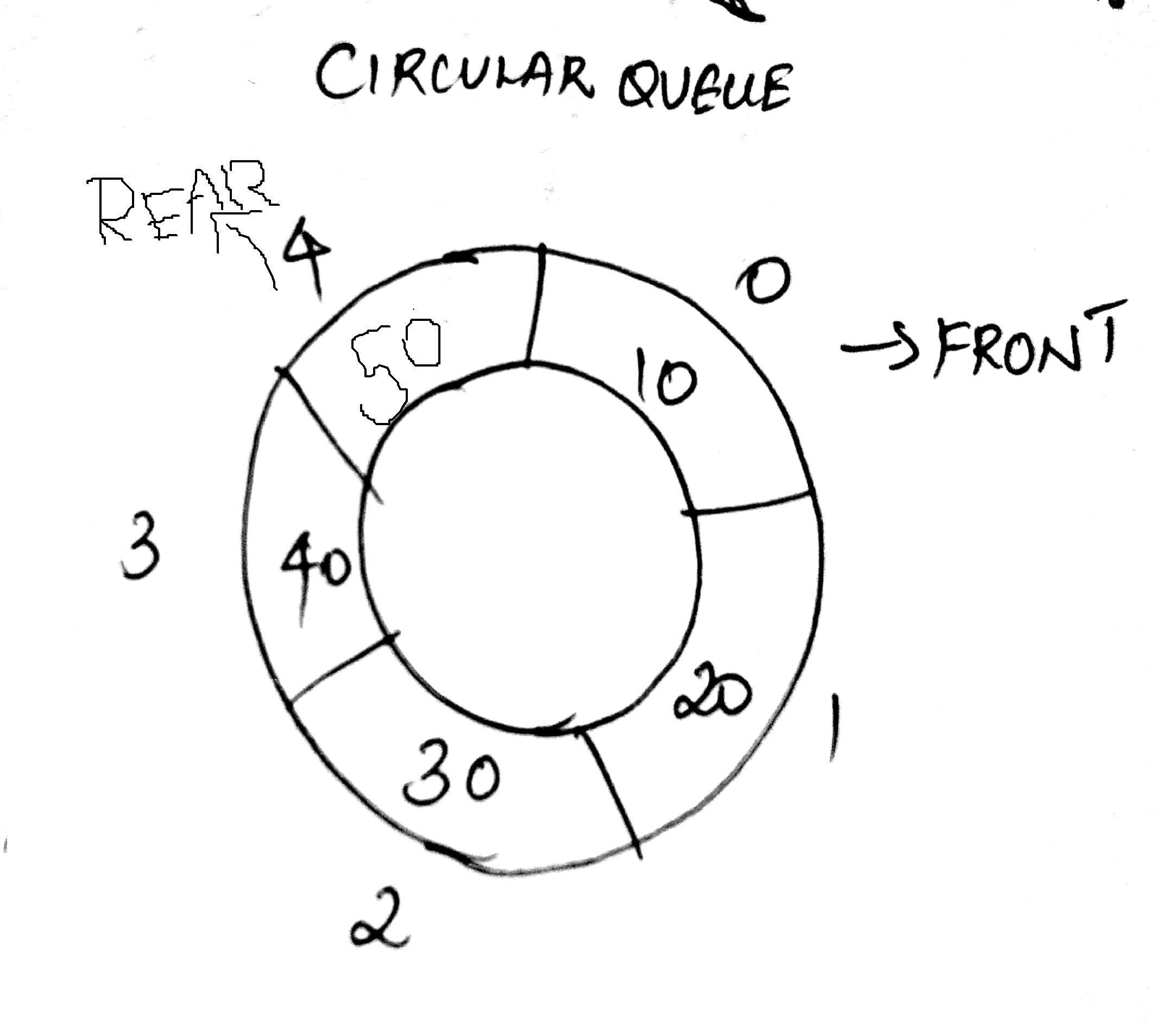
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **P1** | **P2** | **P3** | **P4** | **P5** | **P1** | **P2** | **P4** | **P2** |

**Where 0,2,4,6,8,9,11,13,14,15 represents the time taking for each process under the condition of time quantum = 2 sec.**

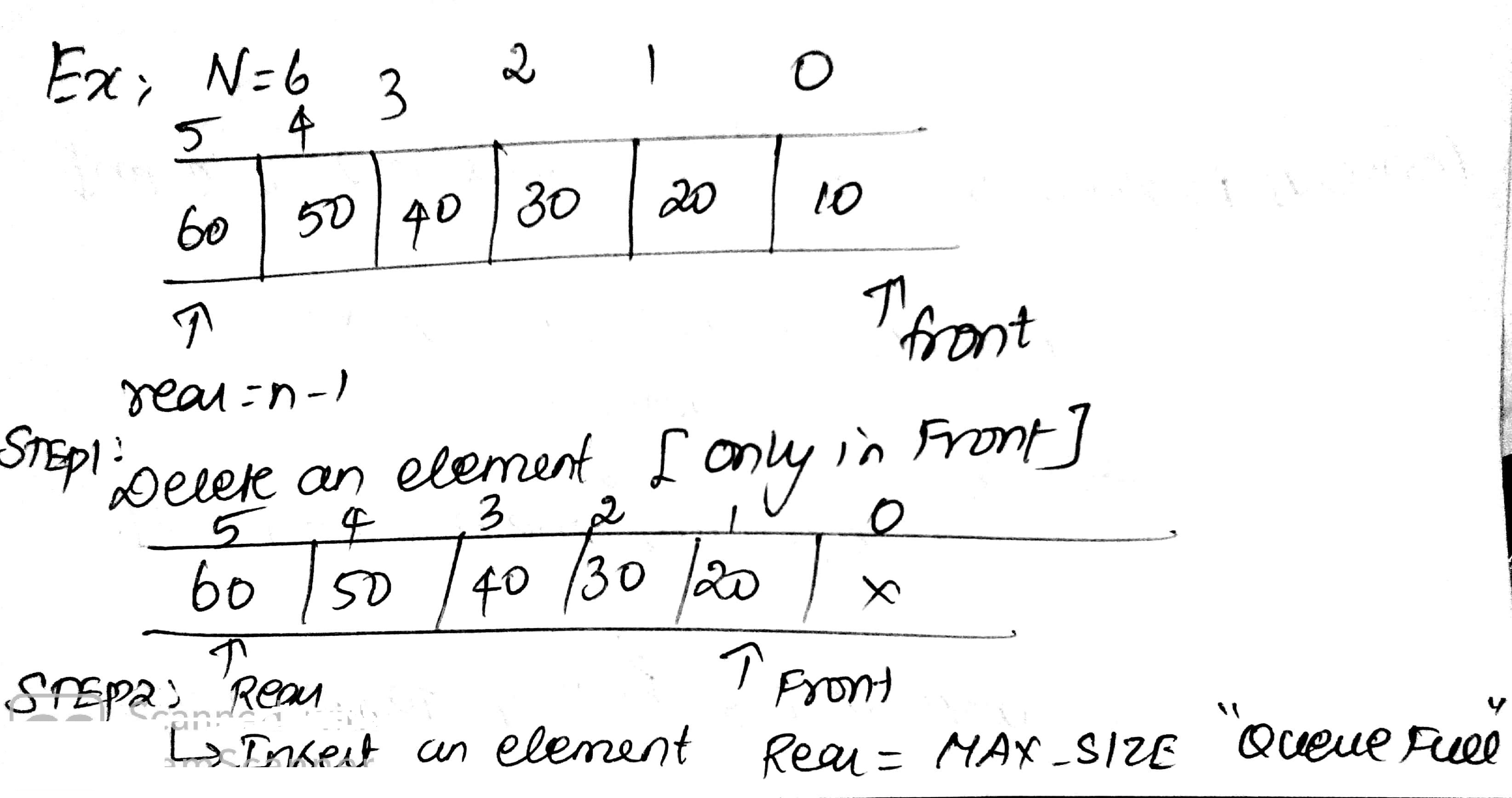
0 2 4 6 8 9 11 13 14 15

**4.6 CIRCULAR QUEUE**

It is a linear data structure in which the operation are performed based on **FIFO** principle. In circular queue the **last position is connected back to the first position** to make circle. It is also called as **ring buffer**.



**Fig. 4.9 Representation of a circular queue**



**Fig. 4.10 Inserting an element into a circular queue**

**In normal queue we can insert element at rear position until queue becomes full. But cannot able to insert in front if the queue is full but front position having space. To overcome this shown in above diagram circular queue is implemented**.

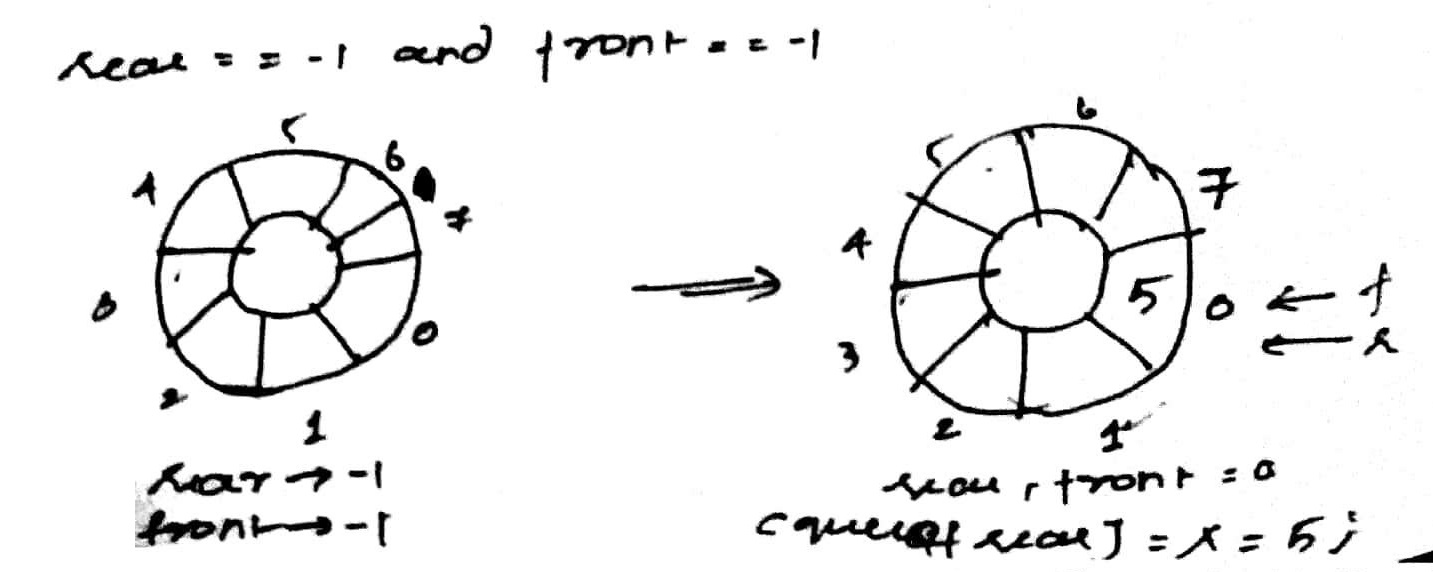
**4.6.1 Operation on circular queue**

1. Front: Get the Front item from queue.
2. Rear: Get the Rear item form queue.
3. Enqueue: (value)

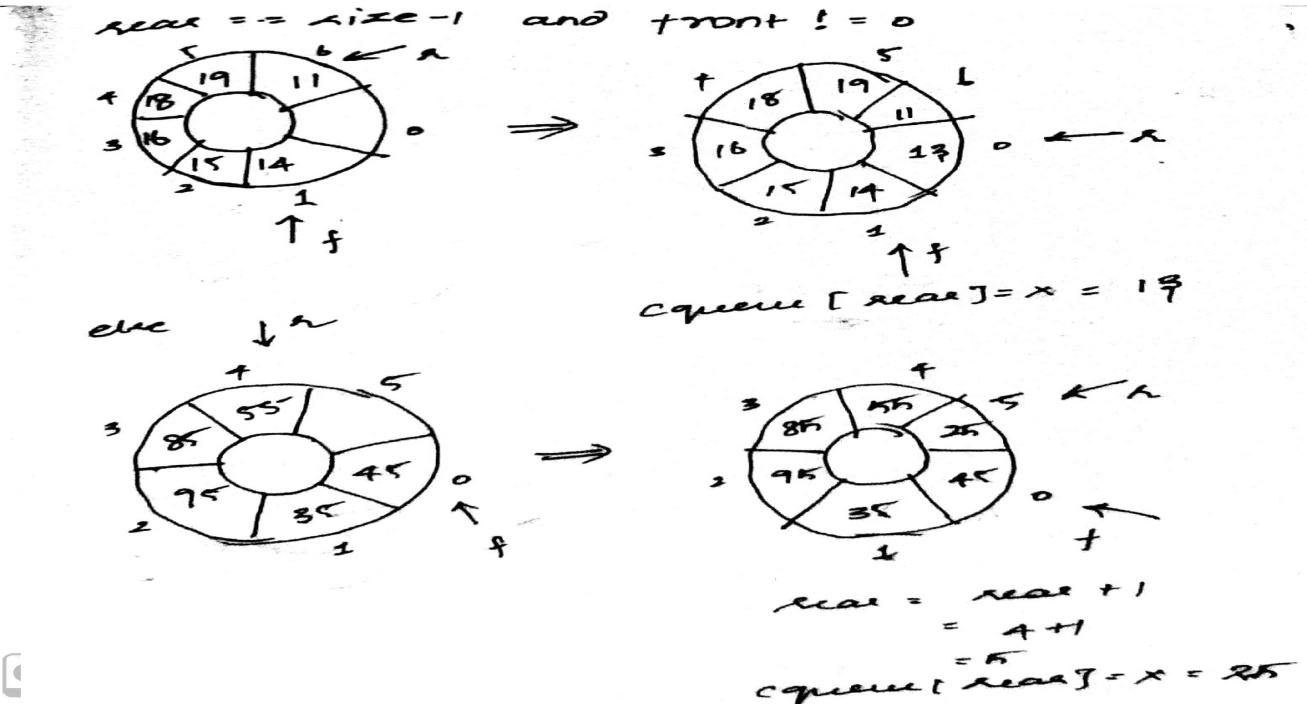
It is used to insert an element into the circular queue at rear position.

Algorithm Enqueue\_circularqueue (enqueue [], rear, front)

1. If rear >= size – 1 and fron = 0
2. Return OVERFLOW
3. Else if rear == -1 and front == -1
4. Front = 0
5. Rear = 0
6. Else if rear == sixe -1 and front !=0
7. Rear = 0
8. Else
9. Rear = Rear + 1
10. Enqueue [rear] = x



**Fig. 4.11 Operations into a circular queue**



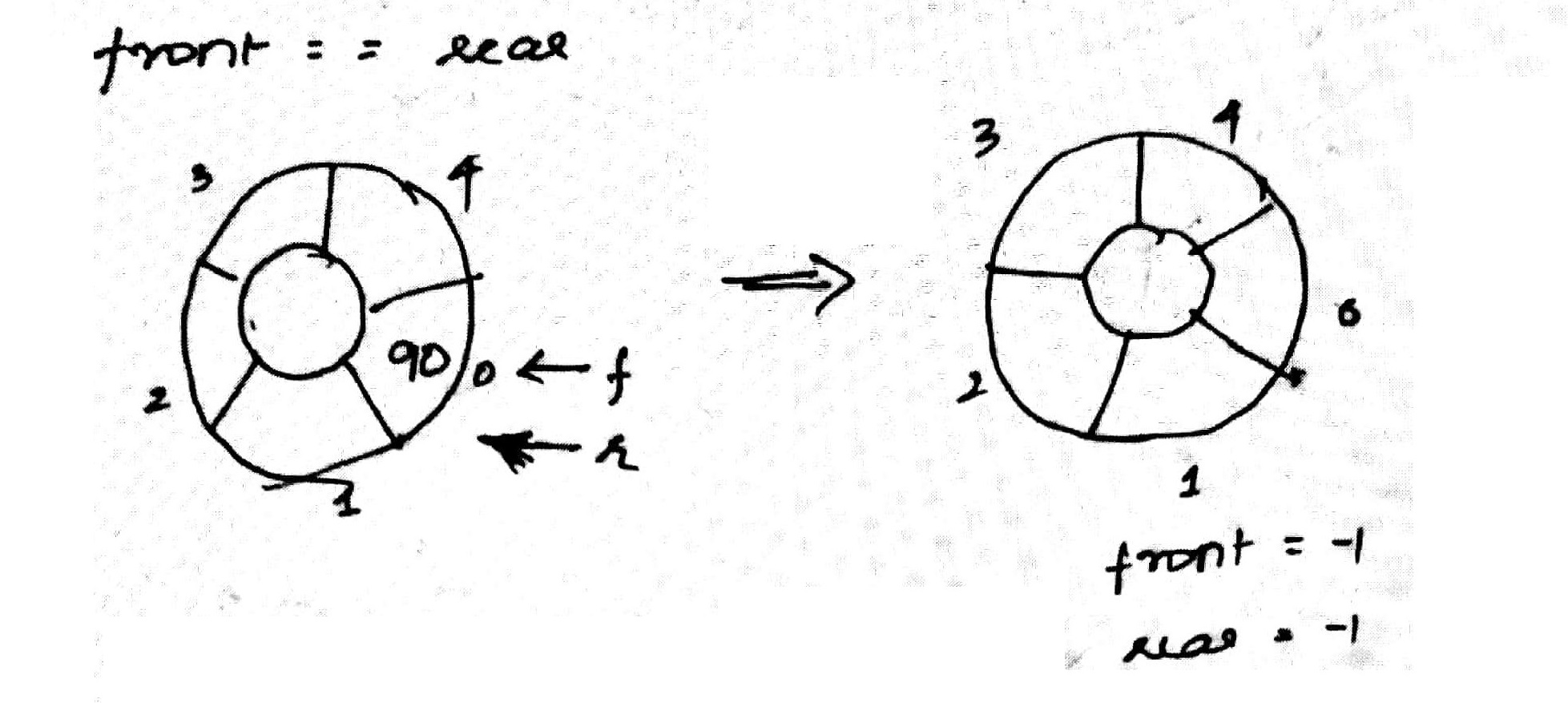
**Fig. 4.12 Insertion in a circular queue**

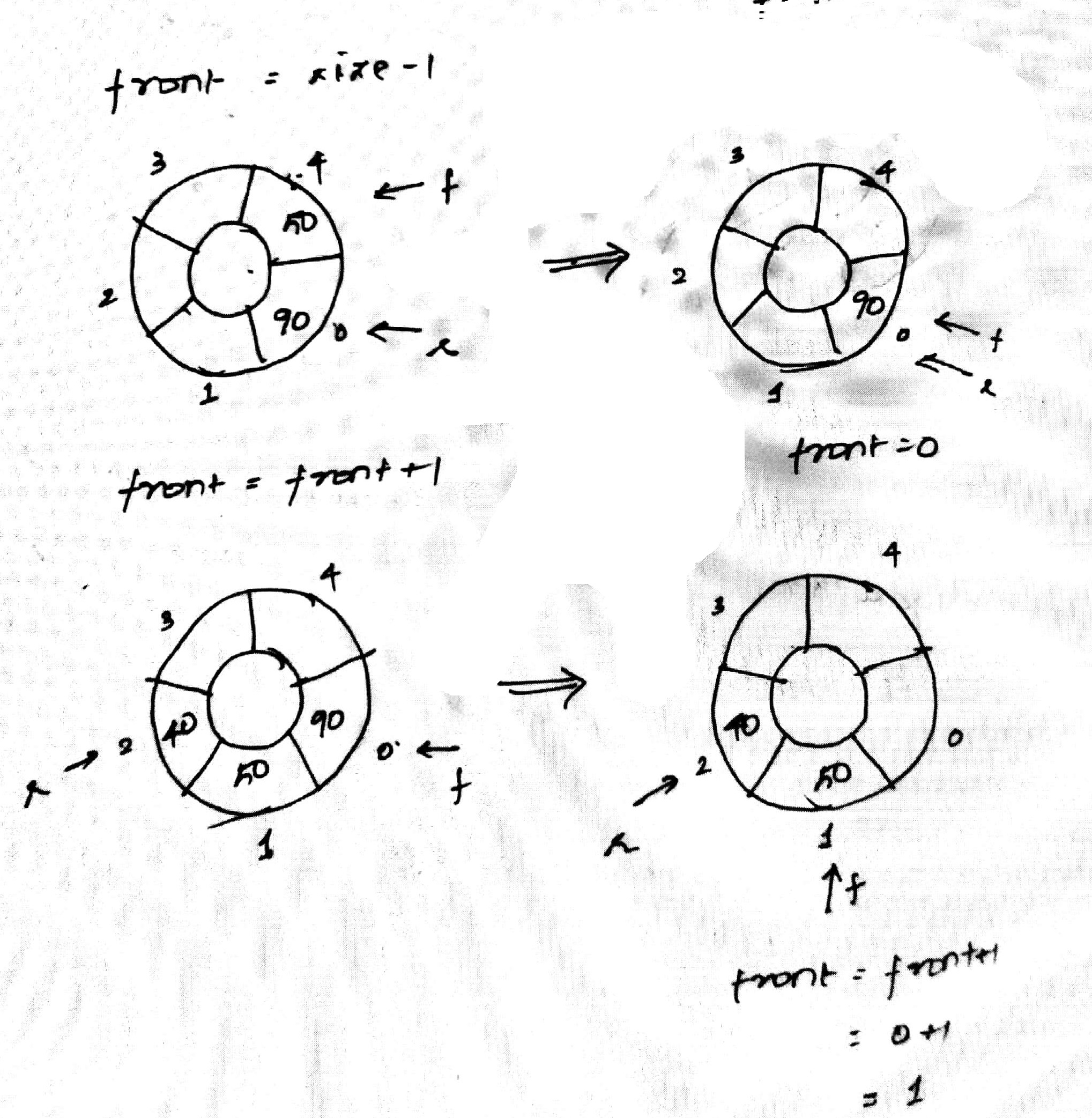
1. Dequeue:

This function is used to delete an element from the circular queue at front position.

Algorithm Dequeue\_circularqueue (dequeue [], rear, front)

1. If front == -1
2. Return UNDERFLOW
3. Else
4. Return dequeue[front]
5. If front == rear
6. Front = -1
7. Rear = -1
8. Else
9. If front = size -1
10. Front = 0
11. Else
12. Front = Front + 1





**Fig. 4.13 Deletion in a circular queue**

**4.7 PRIORITY QUEUE**

Priority queue is an extension of queue data structure where each element has a priority associated with it. An element with high priority is dequeued before an element with low priority. If two elements have same priority, they are dequeued according to their order in the queue.

Algorithm enqueue (pqueue[], front, rear, X)

1. If rear >= SIZE -1
2. Return OVERFLOW
3. If front == -1 and rear ==-1
4. Front ++
5. Rear ++
6. Pqueue [rear]=x
7. Else
8. Call check (x)
9. Rear ++

Algorithm check (x, pqueue [], front, rear)

1. For I =0 to rear
2. If data >= pqueue [i]
3. For j=rear + 1 to 0
4. Pqueue [j] = pqueue [j-1]
5. Pqueue[i]=x
6. Break
7. Pqueue[i] = x

Algorithm pdqueue (pqueue[], front, rear, X)

1. If front == -1 and rear == -1
2. Return UNDERFLOW
3. For I = 0 to rear
4. If data == pqueue [i]
5. For j=I to rear
6. Pqueue[j]=pqueue[j+1]
7. Rear – –

**ALL THE BEST**