Integrated MSc(IT)

SEMESTER – III

DATA STRUCTURES

UNIT– III

Dr. Purna Tanna

QUEUE

- Introduction To Queues
- Queue as an ADT
- Representation of Queues as an Array
- Representation of Queues as a Linked List
- Circular Queues
- DEQUES
- Introduction of Priority Queue
- Application of Queues

Introduction to Queue

- Queue is also an **abstract data type** or a linear data structure, in which the first element is **inserted from one end called REAR(also called tail),** and the **deletion** of existing element takes place **from the other end called as FRONT(also called head).**
- For example, people waiting in line for a rail ticket form a queue.
- This makes queue as **FIFO** data structure, which means that element inserted first will also be removed first

Introduction to Queue

- The process to **add** an element into queue is called **Enqueue** and the process of **removal** of an element from queue is called **Dequeue**.

Introduction to Queue

Basic features of Queue

- Like Stack, Queue is also an ordered list of elements of similar data types.
- Queue is a FIFO(First in First Out) structure.
- Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
- peek( ) function is often used to return the value of first element without dequeuing (deleting) it.

Applications of Queue

- Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
- In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.
- A real world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world example can be seen as queues at ticket windows & bus-stops.

Implementation of Queue

- Queue can be implemented using an Array, Stack or Linked List.
- Initially the head(FRONT) and the tail(REAR) of the queue points at the first index of the array (starting the index of array from 0).
- As we add elements to the queue, the tail keeps on moving ahead, always pointing to the position where the next element will be inserted, while the head remains at the first index.
- When we remove element from Queue, **We remove the element from head position and then move head to the next position.**

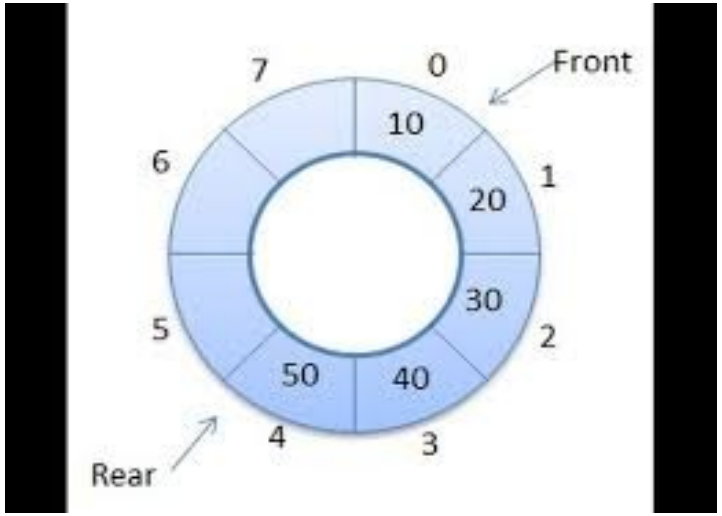
Types of Queue

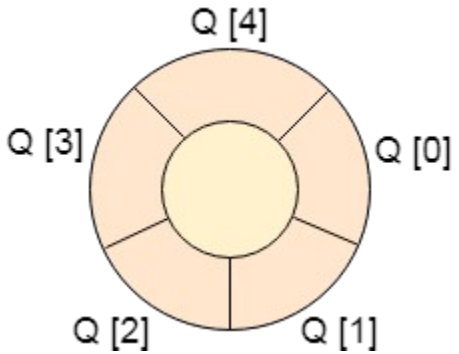
- **Simple(Linear) Queue:** In Simple queue Insertion occurs at the rear of the list, and deletion occurs at the front of the list.

Types of Queue

- **Circular Queue:** A circular queue is a queue in which all nodes are treated as circular such that the first node follows the last node.

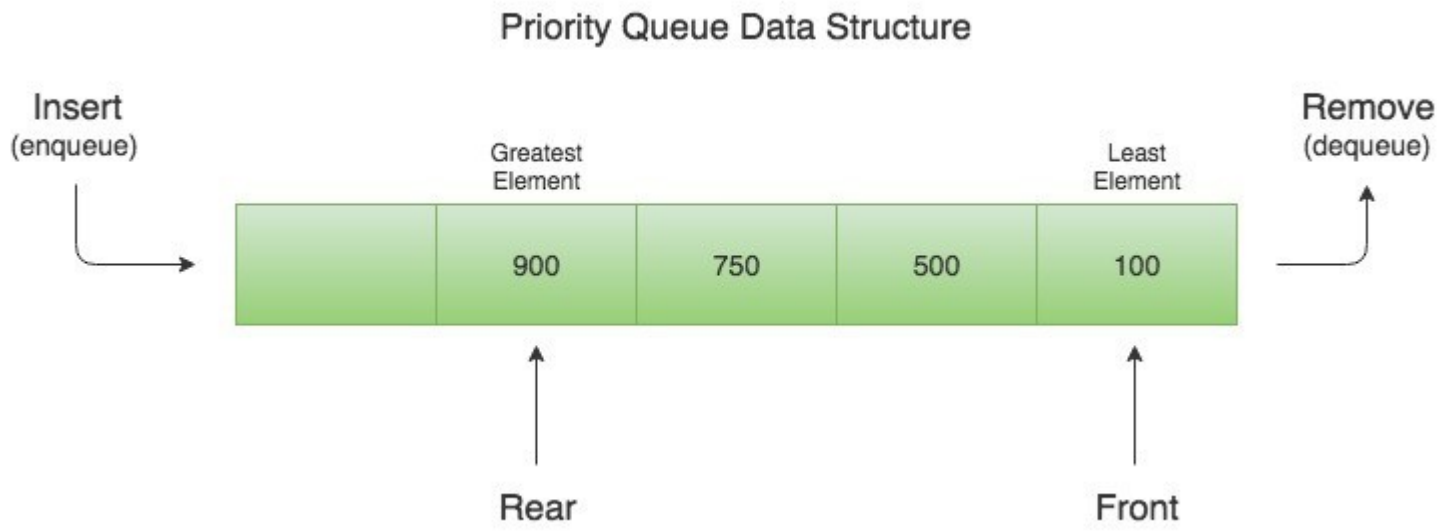
- Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.





Types of Queue

- **Priority Queue**: A priority queue is a queue that contains items that have some preset priority. When an element has to be removed from a priority queue, the item with the highest priority is removed first.



Introduction of Priority Queue

- Priority Queue is an abstract data type in which **each element is assigned a priority**.
- The priority of the element will be used to determine the order in which these elements will be processed.
- **Priority Queue is an extension of queue with following properties.**

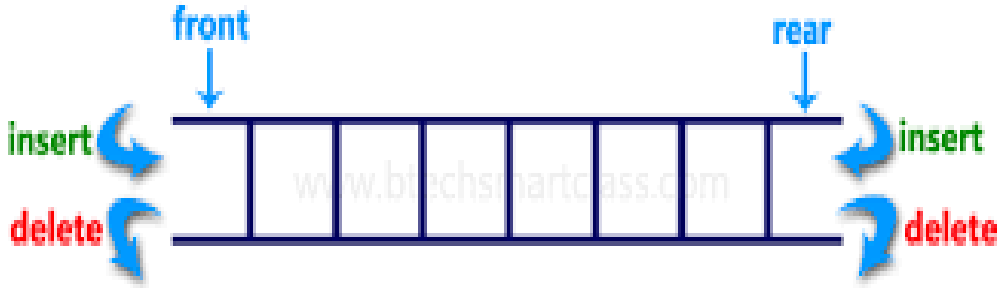
i. Every item has a priority associated with it.

ii. An element with high priority is dequeued before an element with low priority.

iii. If two elements have the same priority, they are served according to their order in the queue.

Types of Queue

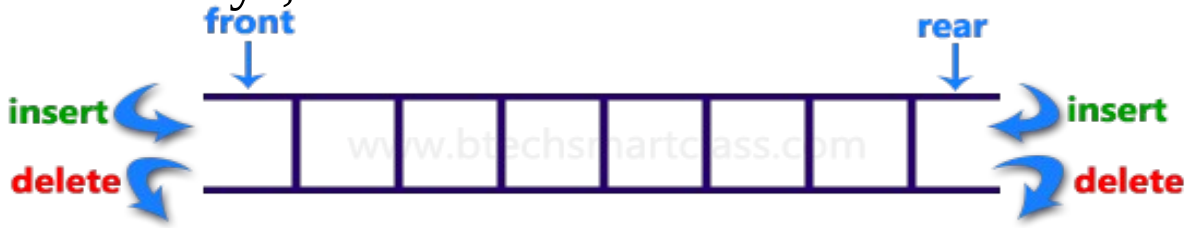
- **Dequeue (Double Ended queue):** In dequeue(double ended queue) Insertion and Deletion occur at both the ends i.e. front and rear of the queue.



DEQUES

- A **DEQUE**, also known as a **double-ended queue.**
- Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at **both the ends (front and rear)**. That means, we can insert at both front and rear positions and can delete from both front and rear positions.
- No elements can be added and deleted from the middle.
- Double Ended Queue can be represented in TWO ways, those are as follows...

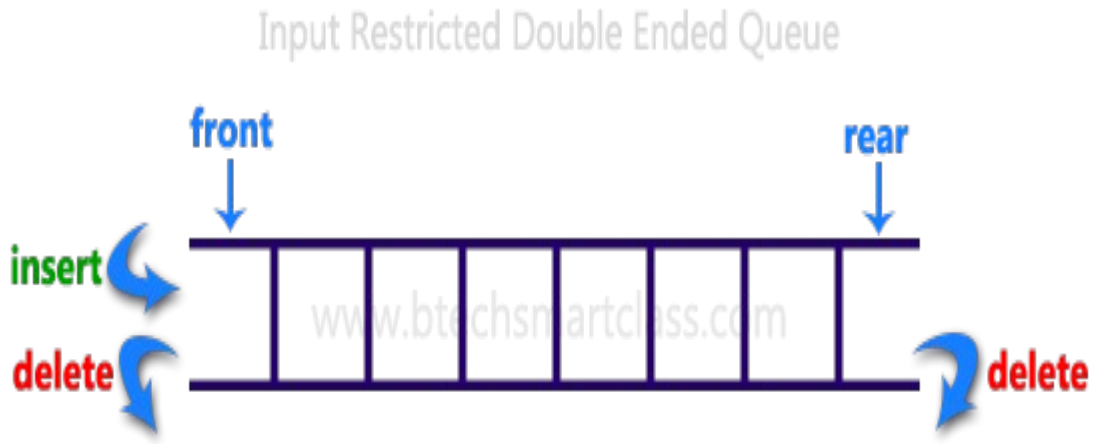
1. **Input Restricted Double Ended Queue**



2. **Output Restricted Double Ended Queue**

Input Restricted Double Ended Queue

- **In input restricted double-ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.**



Output Restricted Double Ended Queue

- In output restricted double ended queue, the **deletion operation is performed at only one end and insertion operation is performed at both the ends.**

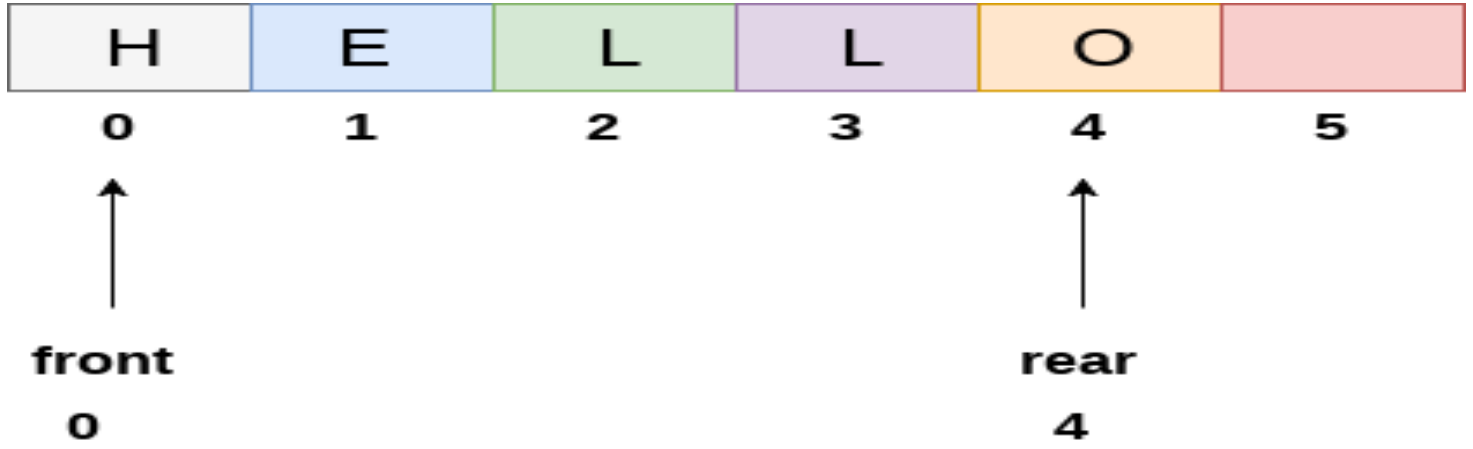
Implementation of Queue

- **Example:**

- Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.

Implementation of Queue

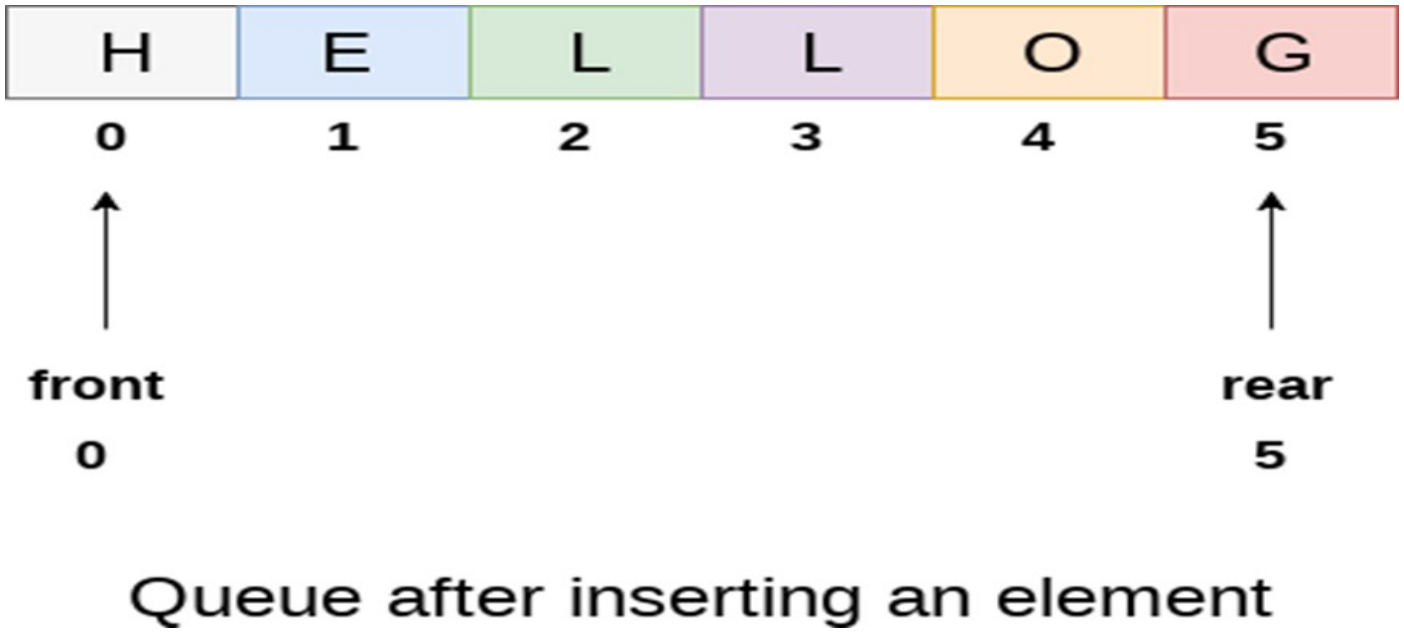
- The below figure shows the queue of characters forming the English word "HELLO". Since, No deletion is performed in the queue till now, therefore the value of front remains 0 . However, the value of rear increases by one every time an insertion is performed in the queue.



Queue

Implementation of Queue

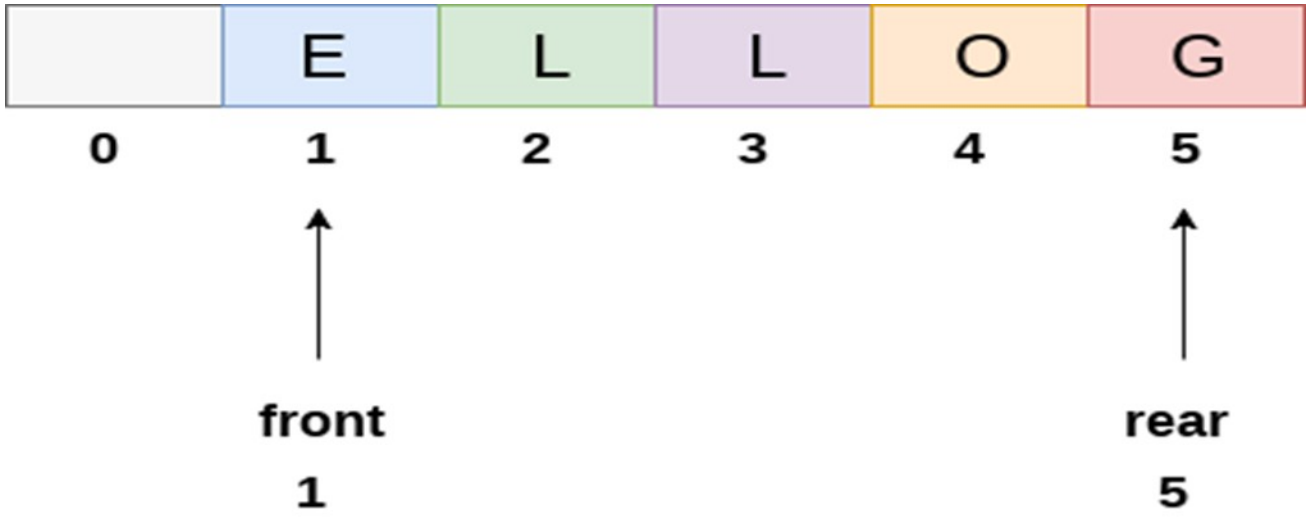
- After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same



Queue after inserting an element

Implementation of Queue

- After deleting an element, the value of front will increase from 0 to 1. however, the queue will look something like following.



Queue after deleting an element

Basic operations of Queue

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.
- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.
- In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

Representation of Queues as an Array

- **Algorithm to insert any element in a queue:**
- Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.
- If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.
- Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Enqueue Operation (Simple Queue)

Step1: IF REAR=MAX-1 then,

Write OVERFLOW

[END OF IF]

Step2: IF FRONT== -1 AND REAR == -1 then,

SET FRONT = REAR = 0

ELSE

SET REAR = REAR + 1

Step3: SET QUEUE[REAR] = NUM

Step4: EXIT

Representation of Queues as an Array

- **Algorithm to delete any element in a queue:**
- If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.
- Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

Dequeue Operation (Simple Queue)

Step1: IF FRONT = -1 OR FRONT > REAR then,

Write UNDERFLOW

ELSE

SET FRONT = FRONT + 1

SET VAL= QUEUE[FRONT]

[END OF IF]

Step2: EXIT

Peek Operation (Simple Queue)

Step1: IF FRONT = -1 OR FRONT > REAR then,

Write UNDERFLOW

[END OF IF]

Step2: return QUEUE[FRONT]

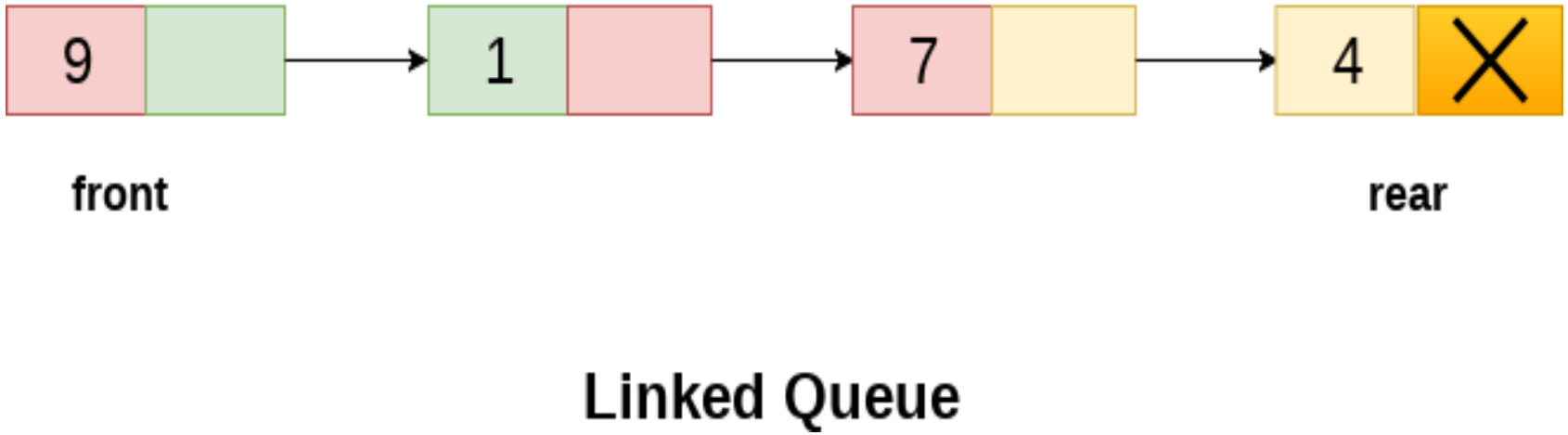
Step3: EXIT

Representation of Queues as a Linked List

- The array implementation can not be used for the large scale applications where the queues are implemented.
- One of the alternative of array implementation is linked list implementation of queue.
- The storage requirement of linked representation of a queue with n elements is o(n) while the time requirement for operations is o(1).
- In a linked queue, each node of the queue consists of two parts i.e. **data part and the link part. Each element of the queue points to its immediate next element in the memory.**

Representation of Queues as a Linked List

- In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The **front pointer** contains the a**ddress of the starting element of the queue** while the **rear** pointer contains **the address of the last element of the queue.**
- **Insertion and deletions are performed at rear and front end respectively.** If front and rear both are NULL, it indicates that the queue is empty.



Linked Queue

Enqueue Operation (Simple Queue)

Step1: Allocate memory for the new node and name it as PTR

Step2: SET PTR → DATA = VAL

Step3: IF FRONT = NULL, then

SET FRONT = REAR = PTR;

SET FRONT → NEXT = REAR → NEXT = NULL

ELSE

SET REAR → NEXT = PTR

SET REAR = PTR

SET REAR → NEXT = NULL

[END OF IF]

Step4: END

Dequeue Operation (Simple Queue)

Step1: IF FRONT = NULL, then

Write “Underflow”

[END OF IF]

Step2: SET PTR = FRONT

Step3: FRONT = FRONT → NEXT

Step4: FREE PTR

Step5: END

Peek Operation (Simple Queue)

Step1: IF FRONT = NULL, then

Write “Underflow”

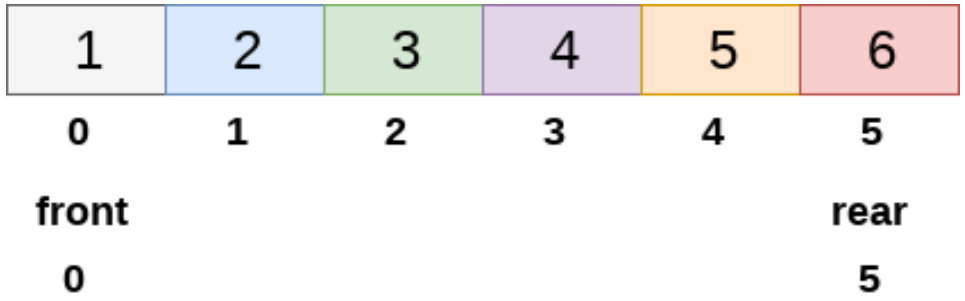
[END OF IF]

Step2: return FRONT → DATA

Step3: EXIT

Why we use Circular Queue??

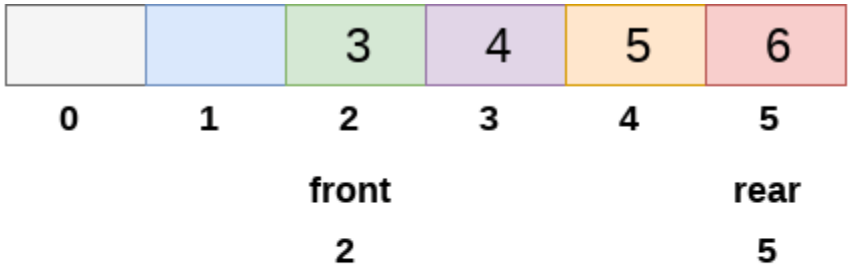
- In linear list insertions can be done only at one end called the rear and deletions is always done from the other end called the front.
- Here front= 0 and rear =5.
- Now if you want to insert another value, it will not be possible because the queue is completely full. There is no space where the value can be inserted.



Queue

Why we use Circular Queue??

- Consider a scenario in which two successive deletions are made. The queue will be then as shown in fig.



Queue after deletion of first 2 elements

- Here front= 2 and rear =5.
- Now suppose we want to insert new element in the queue shown in fig. Even though there is space available, the overflow condition exists because the condition rear = MAX-1 still holds true. This is the major drawback of a linear queue.

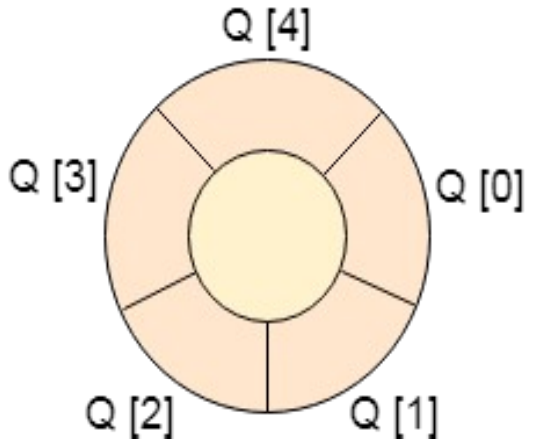
Circular Queue

- **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'**.
- It follows the principle of FIFO(First In First Out), but 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 will be full when front = -1 and rear = max-1.

  Implementation of circular queue is similar to that of a linear queue

  Only the logic part that is implemented in the case of insertion

  and deletion is different from that in a linear queue.



Circular Queue

- **For insertions we have to check following three conditions:**
- If (front==rear+1) OR (front= 0 and rear = MAX-1), then print that the circular queue is full.
- If rear != MAX-1, then the value will be inserted and rear will be incremented.
- If front!=0 and rear= MAX-1, then it means that the queue is not full. So set rear=0 and insert the new element there.

Circular Queue

- **For deletion we have to check following three conditions:**
- If front= -1 then there are no elements in the queue. So underflow condition will be reported
- If the queue is not empty and after returning the value on the front, front=rear, (means the element is last element) then the queue has now become empty and so, front and rear are set to -1.
- If the queue is not empty and after returning the value on the front, front= MAX-1 then front is set to 0.

Enqueue Operation (Circular Queue)

Step1: IF FRONT = 0 and REAR=MAX-1 then,

Write OVERFLOW

ELSE IF FRONT = -1 and REAR = -1, then

SET FRONT = REAR = 0

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

SET REAR = 0

ELSE

SET REAR = REAR + 1

[END OF IF]

Step2: SET QUEUE[REAR] = VAL

Step3: EXIT

Dequeue Operation (Circular Queue)

Step1: IF FRONT = -1 then,

Write UNDERFLOW

SET VAL =-1

[END OF IF]

Step2: SET VAL = QUEUE[FRONT]

Step3: 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]

Step4: EXIT