Queue in Data Structure

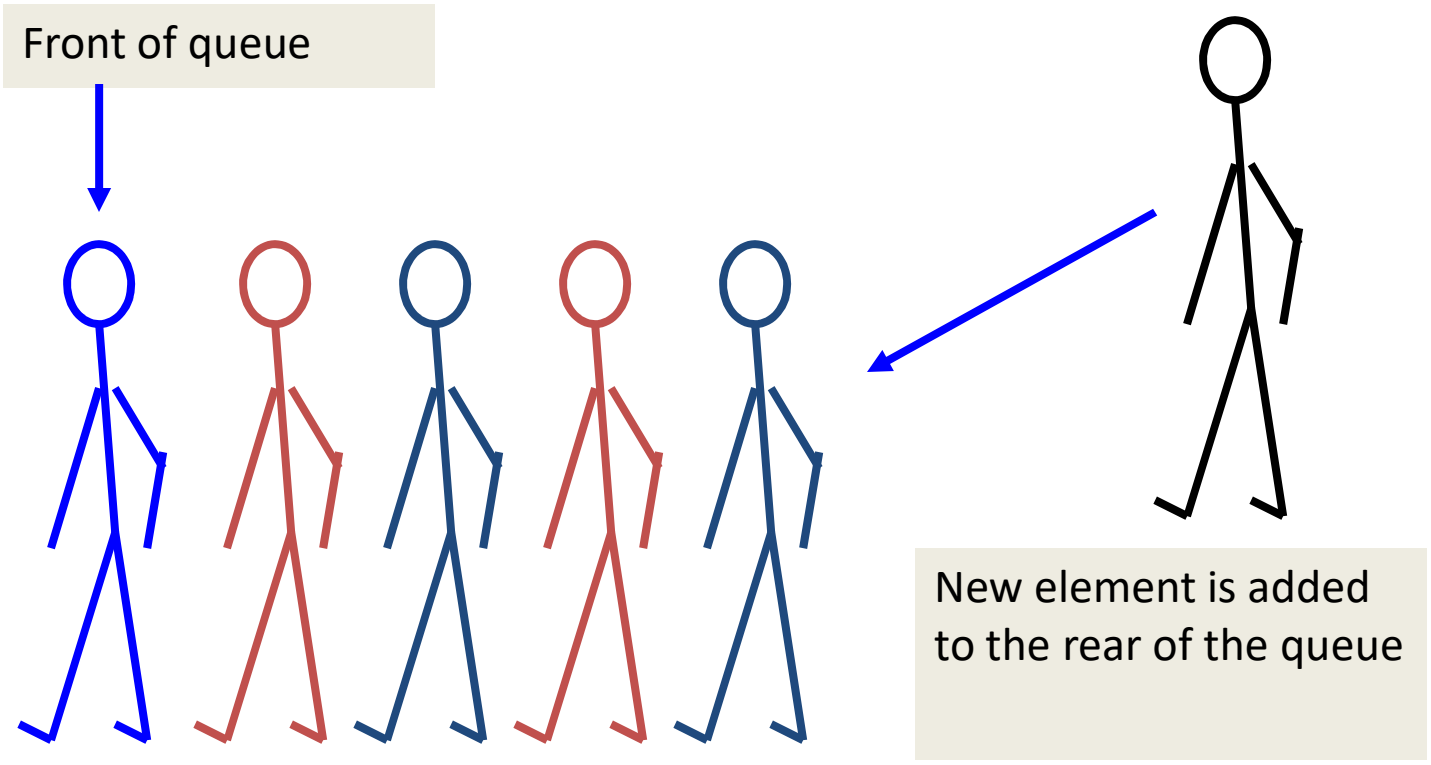
Dr. Anish Soni

Queue

- **Queue** is a data structure which stores its elements in an linearly ordered manner. Inserting element at one end (rear) and deleting element from another end (front).
- A queue is a FIFO (First-In, First-Out) data structure in which the element that is inserted first is the first one to be taken out.
- The term queue comes from the analogy that people are in queue waiting for services. First come, First served (FCFS)

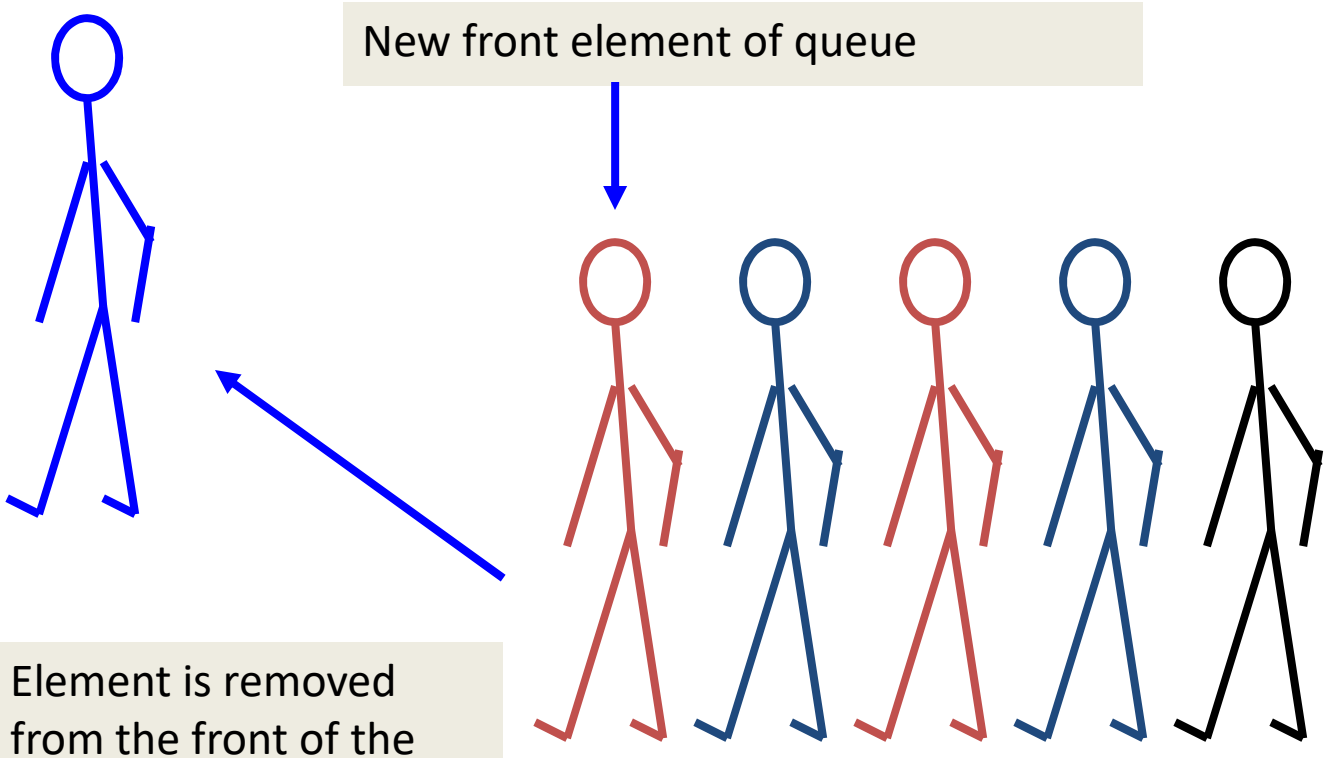
Conceptual View of a Queue

Adding an element



Conceptual View of a Queue

Removing an element



Operations on a Queue

Operation	Description
dequeue	Removes an element from the front of the queue
enqueue	Adds an element to the rear of the queue
first	Examines the element at the front of the queue without removing it
isEmpty	Determines whether the queue is empty
size	Determines the number of elements in the queue

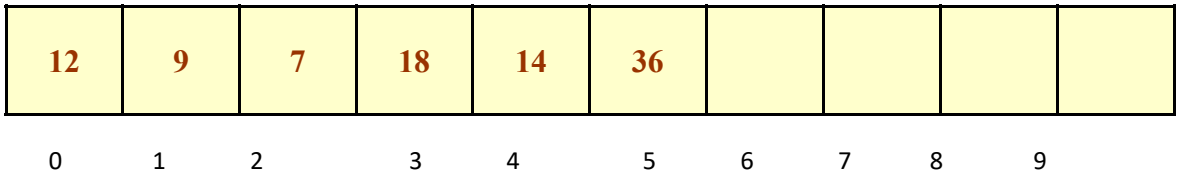
Operations on a Queue

- When queue is empty
  FRONT = -1 and REAR = -1
- Adding an element in queue will increased value of REAR by 1
  REAR = REAR + 1
- Removing an element from queue will increased value of FRONT by 1
  FRONT = FRONT + 1

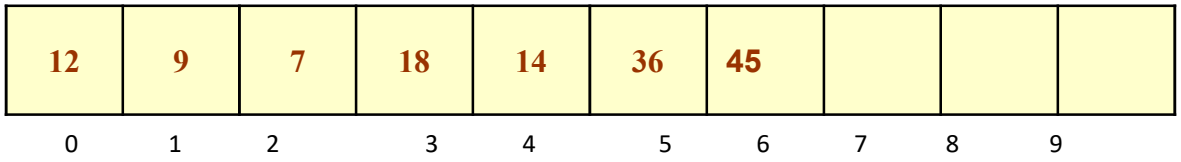
- Queue implementations
  1. using an array
  2. using a linked list

Array Representation of Queues

- Queues can be easily represented using arrays.
- Every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.
- Consider the queue shown in figure

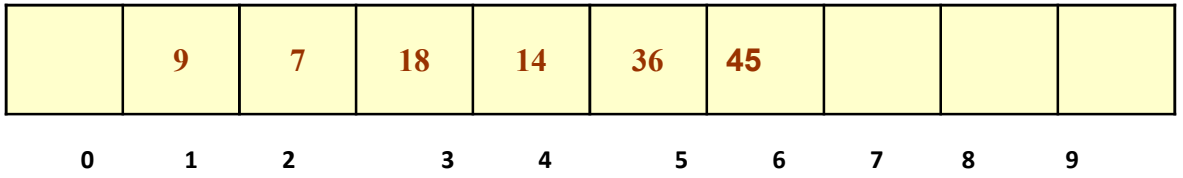


- Here, front = 0 and rear = 5.
- If we want to add one more value in the list say with value 45, then rear would be incremented by 1 and the value would be stored at the position pointed by rear.



Array Representation of Queues

- Now, front = 0 and rear = 6. Every time a new element has to be added, we will repeat the same procedure.
- Now, if we want to delete an element from the queue, then the value of front will be incremented. Deletions are done from only this end of the queue.



- Now, front = 1 and rear = 6.

Array Representation of Queues

- Before inserting an element in the queue we must check for **overflow** conditions.
- An overflow occurs when we try to insert an element into a queue that is already full, i.e. when rear = MAX – 1, where MAX specifies the maximum number of elements that the queue can hold.
- Similarly, before deleting an element from the queue, we must check for underflow condition.
- An underflow occurs when we try to delete an element from a queue that is already empty. If front = -1 and rear = -1, this means there is no element in the queue.

Algorithm for Insertion Operation

```
Algorithm to insert an element in a queue
Step 1: IF REAR=MAX-1, then;
            Write OVERFLOW
           Goto Step 4
        [END OF IF]
Step 2: IF FRONT == -1 and REAR = -1, then
            SET FRONT = REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: Exit
```

Time complexity: O(1)

Algorithm for Deletion Operation

```
Algorithm to delete an element from a queue
Step 1: IF FRONT = -1 OR FRONT > REAR, then
            Write UNDERFLOW
            Goto Step 2
         ELSE
            SET VAL = QUEUE[FRONT]
            SET FRONT = FRONT + 1
        [END OF IF]
Step 2: Exit
```

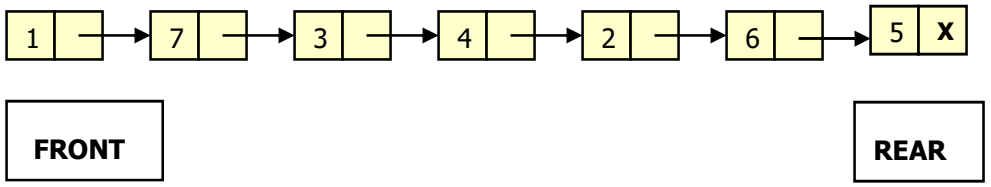
Time complexity: O(1)

Queue by Linked List

- Using a singly linked list to hold queue elements,
  Using FRONT pointer pointing the start element,
  Using REAR pointer pointing to the last element.

- Insertions is done at the rear end using REAR pointer
  Deletions is done at the front end using FRONT pointer

- If FRONT = REAR = NULL, then the queue is empty.



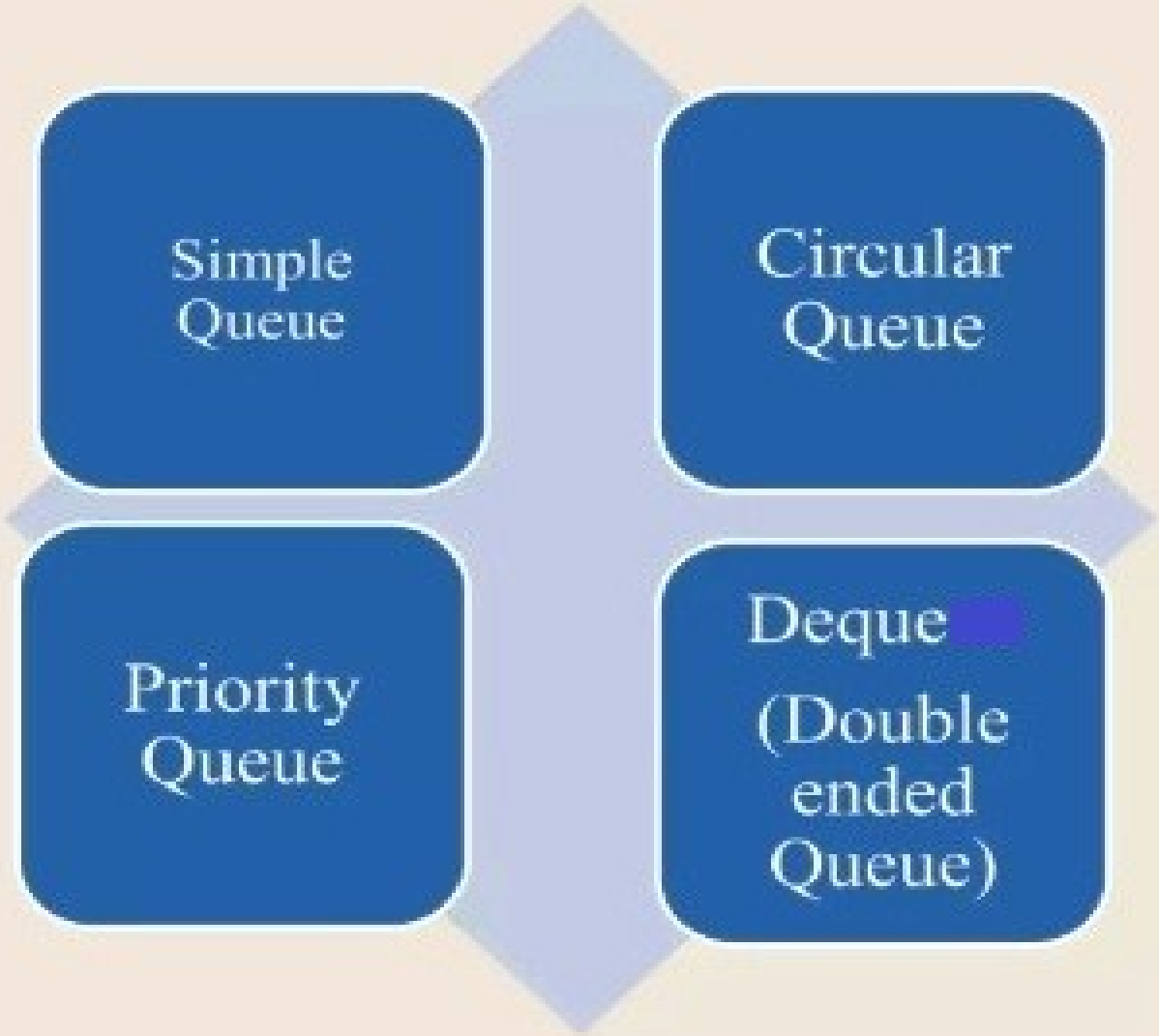
Inserting an Element in a Linked Queue

```
Algorithm to insert an element in a linked queue

Step 1: Allocate memory for the new node and
        name the pointer as PTR
Step 2: SET PTR->DATA = VAL
Step 3: 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]
Step 4: END

Time complexity: O(1)
```

Deleting an Element from a Linked Queue

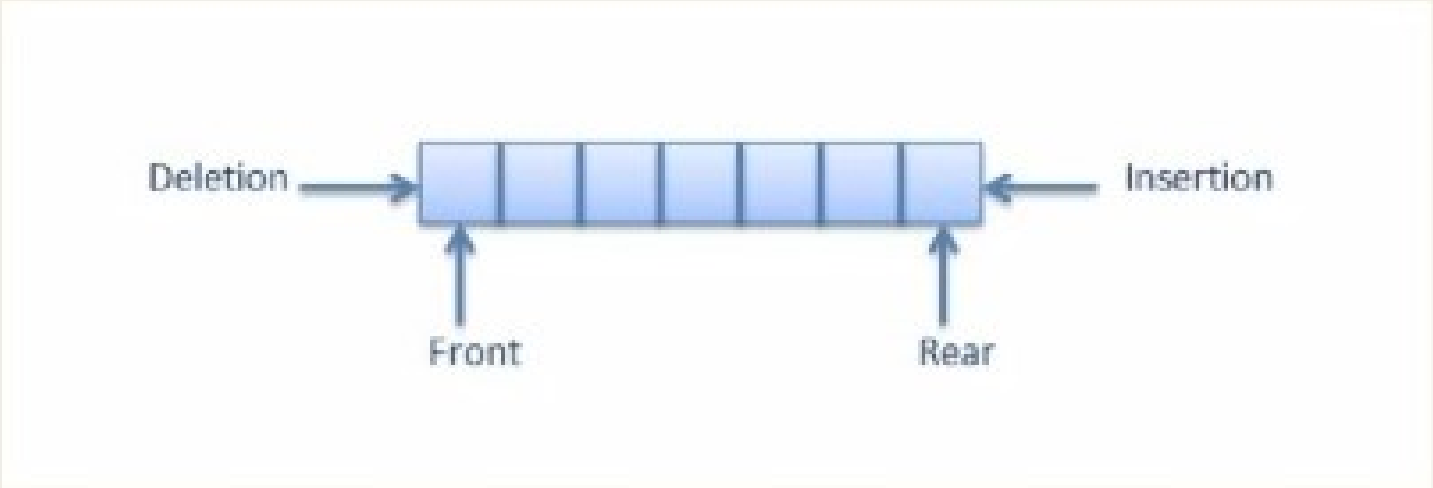
```
Algorithm to delete an element from a
linked queue

Step 1: IF FRONT = NULL, then
          Write "Underflow"
          Go to Step 5
        [END OF IF]
Step 2: SET PTR = FRONT
Step 3: FRONT = FRONT->NEXT
Step 4: FREE PTR
Step 5: END

Time complexity: O(1)
```

Type of queue



Simple Queue

- Simple queue defines the simple operation of queue in which insertion occurs at the rear of the list and deletion occurs at the front of the list.



- **Insert Operation**

➢ Function: **LQINSERT (Q,F,R,N,Y)**. Given F & R, pointers to the front and rear elements of a queue, a vector Q consisting of N elements and an element Y, this procedure inserts Y at the rear of the queue.

➢ Set **F=R=-1**, when queue is empty.

Algorithm

Step-1: [Check overflow?]
if R>=N-1
then Write ('Overflow')
Return

Step-2: [Increment rear pointer R]
R ← R + 1

Step-3: [Insert element]
Q[R] ← Y

Step-4: [Set front pointer F]
if F=-1
then F ← 0
Return

- **Delete Operation**

➢ Function: **LQDELETE (Q,F,R)**. Given F & R, pointers to the front a rear elements of a queue, a queue Q consisting to which they correspond, this function deletes and returns the last element of t queue. Y is a temporary variable.

Algorithm

Step-1: [Check underflow?]
if F=-1
then Write ('Underflow')
Return

Step-2: [Delete element]
Y ← Q[F]

Step-3: [Check for queue empty or has an element?]
if F=R
then F=R=-1
else
F ← F + 1

Step-4: [Return element]
Return(Y)

Example

REAR =-1 and FRONT=0

After 1 insertion REAR=0 and insert item 10 into queue.

REAR =0 and FRONT=0

10				

Now insert 20 into queue

REAR =1 and FRONT=0

10	20			

Suppose now we delete one item from queue, as we know that deletion can be done from FRONT end in queue.
Now, REAR =1 and FRONT=1

	20			

Now we insert 30, 40 and 50 into queue respectively.
REAR =2 and FRONT=1

	20	30		

REAR =3 and FRONT=1

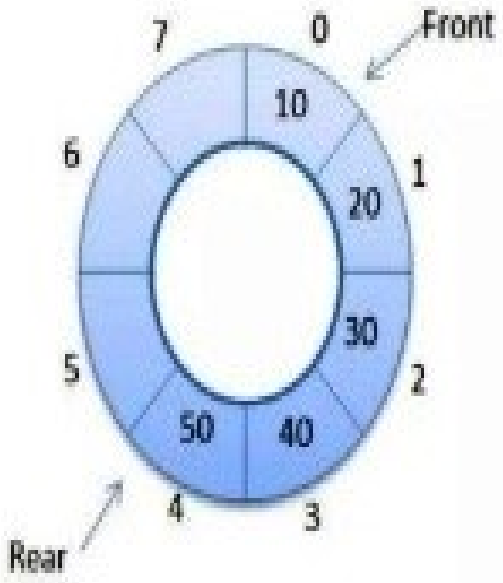
	20	30	40	

REAR =4 and FRONT=1

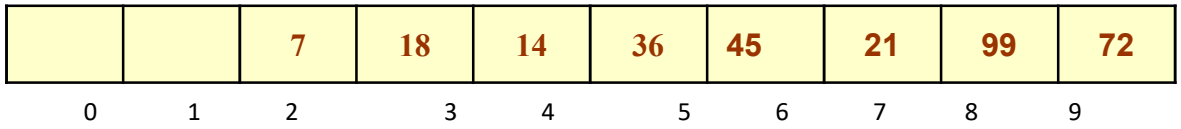
	20	30	40	50

2. Circular Queue

- In circular queue, all the elements are arranged in a circular form.
- Here, the left spaces can be reutilized to insert new variables which was not the case with linear queue.



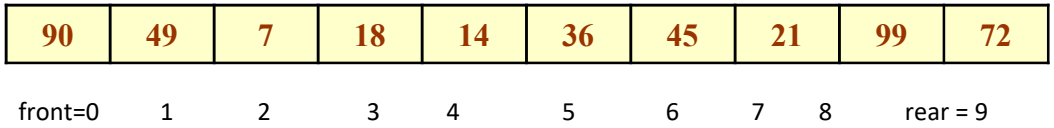
Circular Queues



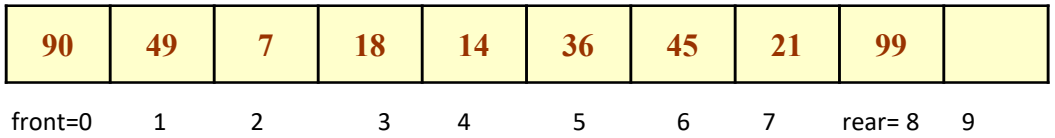
- We will explain the concept of circular queues using an example.
- In this queue, front = 2 and rear = 9.
- Now, if you want to insert a new element, it cannot be done because the space is available only at the left of the queue.
- If rear = MAX – 1, then OVERFLOW condition exists.
- This is the major drawback of an array queue. Even if space is available, no insertions can be done once rear is equal to MAX – 1.
- This leads to wastage of space. In order to overcome this problem, we use circular queues.
- In a circular queue, the first index comes right after the last index.
- A circular queue is full, only when front=0 and rear = Max – 1.

Inserting an Element in a Circular Queue

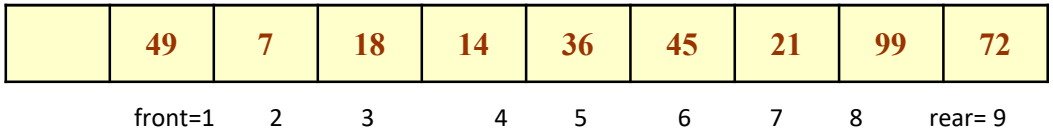
- For insertion we check for three conditions which are as follows:
- If front=0 and rear= MAX – 1, then the circular queue is full.



- If rear != MAX – 1, then the rear will be incremented and value will be inserted

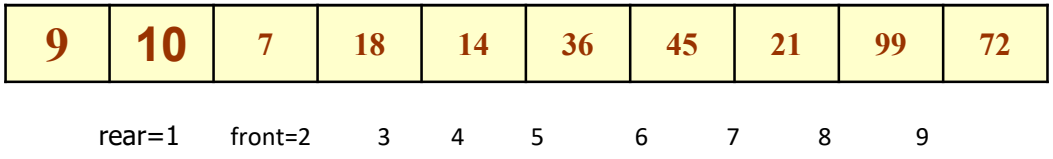


- If front!=0 and rear=MAX -1, then it means that the queue is not full. So, set rear = 0 and insert the new element.



Inserting an Element in a Circular Queue

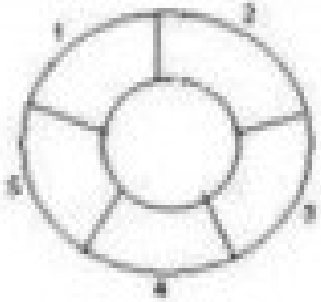
- rear = front -1 overflow



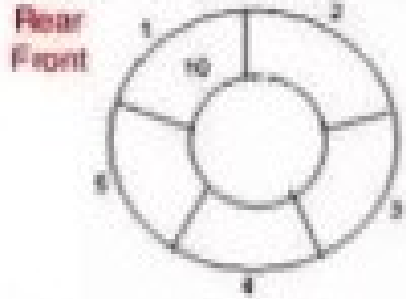
Example

Example: Consider the following circular queue with N = 5.

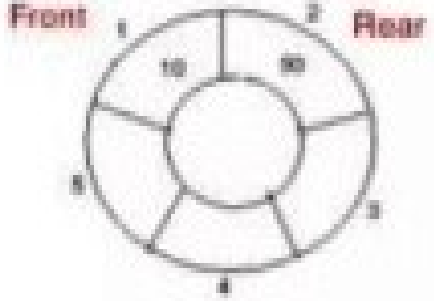
1. Initially, Rear = 0, Front = 0.



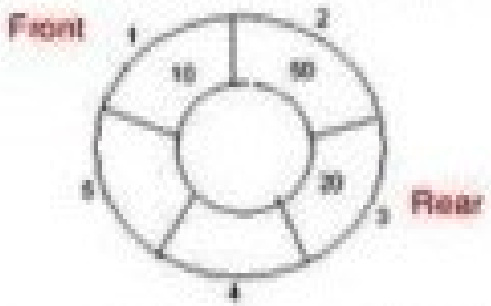
2. Insert 10, Rear = 1, Front = 1.



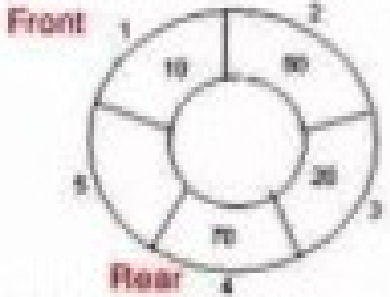
3. Insert 50, Rear = 2, Front = 1.



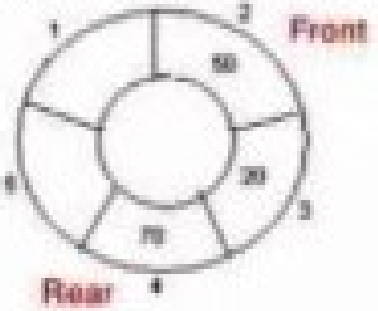
4. Insert 20, Rear = 3, Front = 0.



5. Insert 70, Rear = 4, Front = 1.



6. Delete front, Rear = 4, Front = 2.

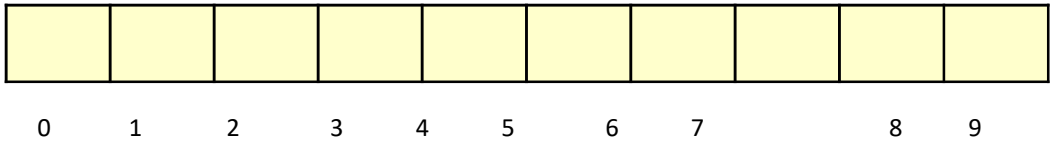


Algorithm to Insert an Element in a Circular Queue

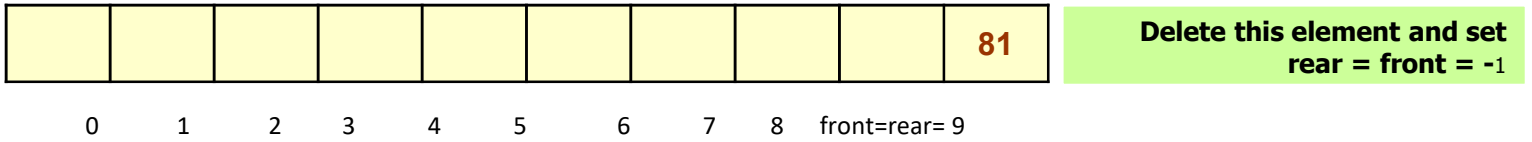
```
Step 1: IF FRONT=0 and REAR=MAX-1, or REAR = FRONT - 1 then
            Write "OVERFLOW"
            Goto Step 4
        [END OF IF]

Step 2: 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]
Step 3: SET QUEUE[REAR] = VAL
Step 4: Exit
```

Deleting an Element from a Circular Queue

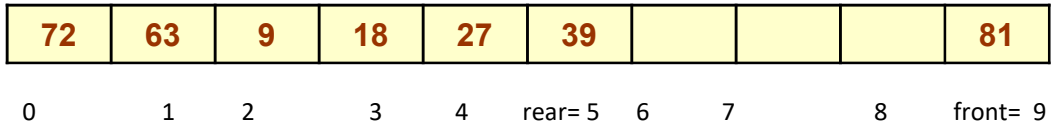
- To delete an element again we will check for three conditions:
  - If front = -1, then it means there are no elements in the queue. So an underflow condition will be reported.



  - If the queue is not empty and after returning the value on front, if front = rear, then it means now the queue has become empty and so front and rear are set to -1.



  - If the queue is not empty and after returning the value on front, if front = MAX -1, then front is set to 0.



Algorithm to Delete an Element from a Circular Queue

```
Step 1: IF FRONT = -1, then
           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
```

Deques

- A deque (double-ended queue) is a list in which elements can be inserted or deleted at either end.
- Also known as a head-tail linked list because elements can be added to or removed from the front (head) or back (tail).
- A deque can be implemented either using a circular array or a circular doubly linked list.
- In a deque, two pointers are maintained, LEFT and RIGHT which point to either end of the deque.

Dequeue
(Double ended Queue)

- In Double Ended Queue, insert and delete operation can be occur at both ends that is front and rear of the queue

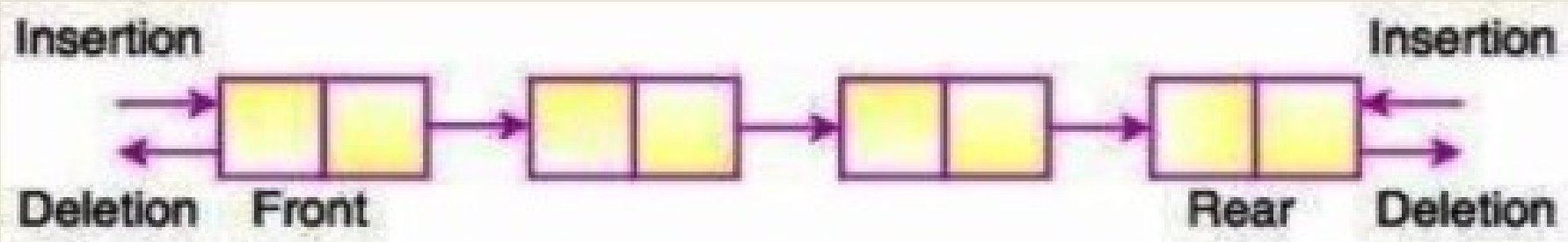


Fig. Double Ended Queue (Dequeue)

Key Characteristics-Deque

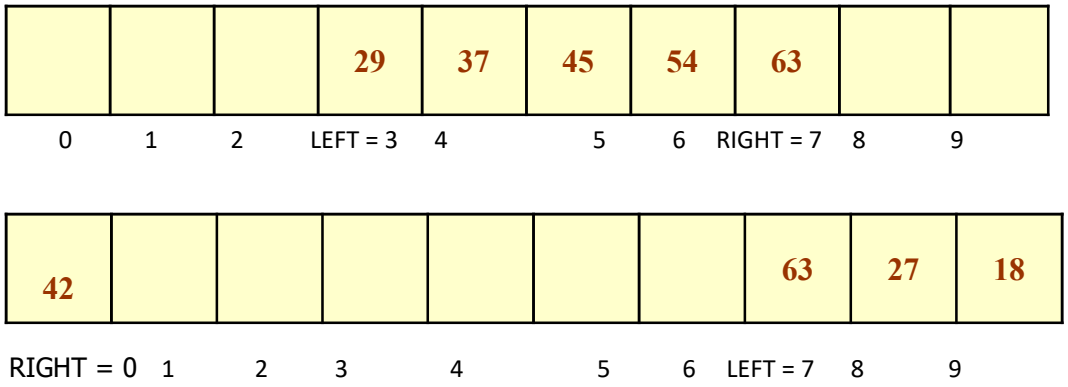
- **Two-Way Entry/Exit:** Unlike a standard queue (FIFO) or stack (LIFO), you have full control over both ends.
- **Dynamic Size:** Usually implemented using a dynamic array or a doubly linked list.
- **No Middle Access:** While you can manipulate the ends efficiently, accessing or removing elements from the middle is generally slow .

Core Operations-Deque

- **insertFront()**: Adds an item to the front.
- **insertLast()**: Adds an item to the rear.
- **deleteFront()**: Removes an item from the front.
- **deleteLast()**: Removes an item from the rear.

Deque variants

- There are two variants of deques:
  - ***Input restricted deque****:* In this dequeue insertions can be done only at one of the ends while deletions can be done from both the ends.
  - ***Output restricted deque****:* In this dequeue deletions can be done only at one of the ends while insertions can be done on both the ends.



Priority Queues

- A priority queue is a queue in which each element is assigned a priority.
- The priority of elements is used to determine the order in which these elements will be processed.
- The general rule of processing elements of a priority queue can be given as:
  - An element with higher priority is processed before an element with lower priority
  - Two elements with same priority are processed on a first come first served (FCFS) basis
- Priority queues are widely used in operating systems to execute the highest priority process first.
- In computer's memory priority queues can be represented using arrays or linked lists.

Key Characteristics

- **Priority Assignment**: Every element in a priority queue is associated with a priority value, which can be based on factors like numerical value, urgency, or importance.
- **Priority-Based Deletion**: When an element is removed (dequeued), it is always the one with the highest priority. This differs from a normal queue which follows the First-In, First-Out (FIFO) principle.
- **FIFO for Same Priority**: If two elements have the same priority, they are processed in the order they were inserted, following the standard FIFO rule.

Common Types

- **Max-Priority Queue:** The element with the largest value has the highest priority.
- **Min-Priority Queue:** The element with the smallest value has the highest priority

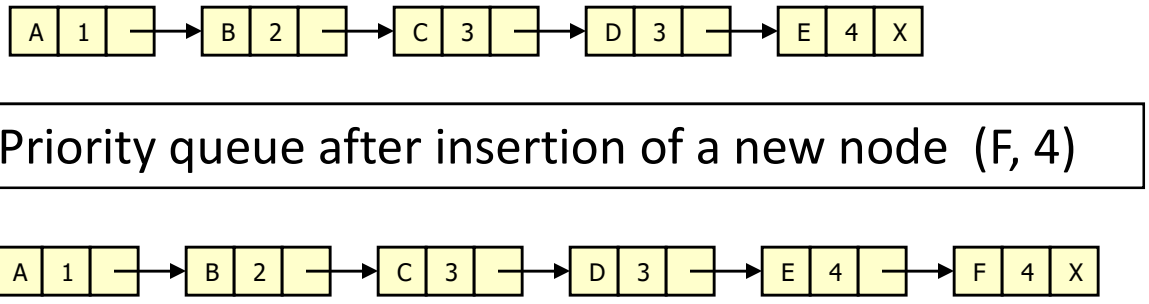
Core Operations

The primary operations supported by a priority queue include:

- **insert(item, priority) (Enqueue)**: Adds a new element with its priority to the queue. The position depends on its priority.
- **extractMax() or extractMin() (Dequeue)**: Removes and returns the element with the highest priority. This is the central operation.
- **peek() (or top())**: Returns the element with the highest priority without removing it

Linked List Representation of Priority Queues

- When a priority queue is implemented using a linked list, then every node of the list contains three parts: (1) the information or data part, (ii) the priority number of the element, (iii) and address of the next element.
- If we are using a sorted linked list, then element having higher priority will precede the element with lower priority.



Array Representation of Priority Queues

- When arrays are used to implement a priority queue, then a separate queue for each priority number is maintained.
- Each of these queues will be implemented using circular arrays or circular queues. Every individual queue will have its own FRONT and REAR pointers.
- We can use a two-dimensional array for this purpose where each queue will be allocated same amount of space.
- Given the front and rear values of each queue, a two dimensional matrix can be formed.

Applications of Queues

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.

2. Queues are used to transfer data asynchronously e.g., pipes, file IO, sockets.

3. Queues are used as buffers on MP3 players and portable CD players, iPod playlist.

Applications of Queues

4. Queues are used in Playlist for jukebox to add songs to the end, play from the front of the list.

5. Queues are used in OS for handling interrupts. When programming a real-time system that can be interrupted, for example , by a mouse click, it is necessary to process the interrupts immediately before proceeding with the current job. If the interrupts have to be handled in the order of arrival, then a FIFO queue is the appropriate data structure