Data Structures

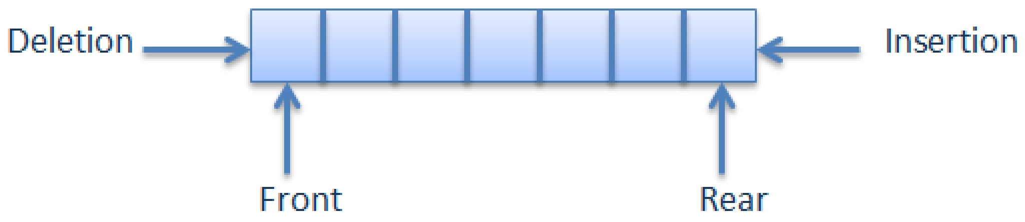
IT 205

Dr. Manish Khare

Lecture – 11&12
25-Jan-2018

Queue

- Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.



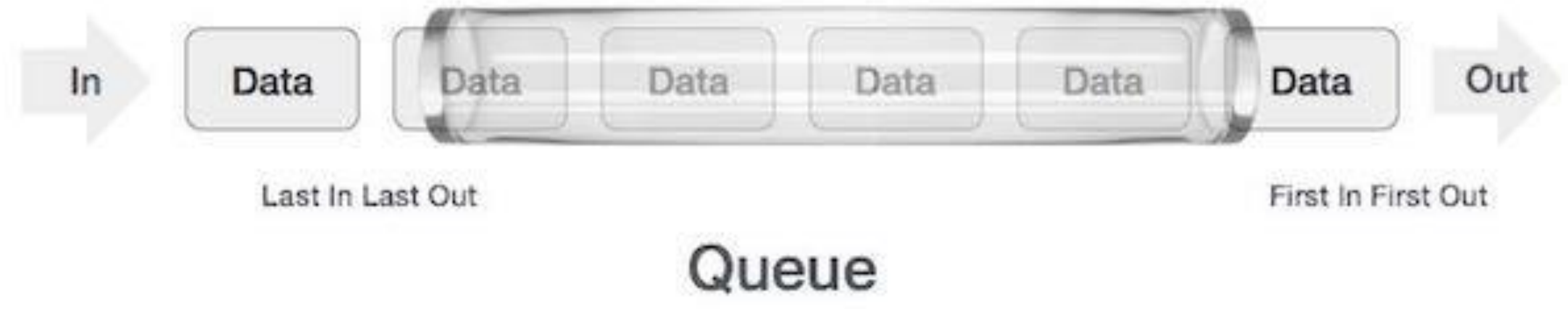
- **Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on FIFO principle.**

- A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

LAST IN
LAST OUT
ONE WAY
FIRST IN
FIRST OUT

Queue Representation

- As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure –



- As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

- Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues –
  - **enqueue()** – add (store) an item to the queue.
  - **dequeue()** – remove (access) an item from the queue.
- Few more functions are required to make the above-mentioned queue operation efficient. These are –
  - **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.

Peek()

- This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows.
  - **Algorithm**

```
begin procedure peek
   return queue[front]
end procedure
```

- Implementation of peek() function in C/C++ programming language

```
int peek()
{
   return queue[front];
}
```

Isfull()

- As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function

```
begin procedure isfull
  if rear equals to MAXSIZE
           return true
  else
          return false
 endif
end procedure
```

- Implementation of isfull() function in C/C++ programming language

```
bool isfull()
{
    if(rear == MAXSIZE - 1)
 return true;
    else
return false;
}
```

Isempty()

- Algorithm of isempty() function −

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

```
begin procedure isempty
   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif
end procedure
```

```
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
      return false;
}
```

Implementation of Queue

- Queue data structure can be implemented in two ways. They are as follows...
  - **Using Array**
  - **Using Linked List**

- When a queue is implemented using array, that queue can organize only limited number of elements. When a queue is implemented using linked list, that queue can organize unlimited number of elements.

Queue using Array

- The implementation of queue data structure using array is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using **FIFO (First In First Out) principle** with the help of variables **'front'** and '**rear**’.
- Initially both '**front**' and '**rear**' are set to -1. Whenever, we want to insert a new value into the queue, increment '**rear**' value by one and then insert at that position.
- Whenever we want to delete a value from the queue, then increment 'front' value by one and then display the value at '**front**' position as deleted element.

Queue Operations using Array

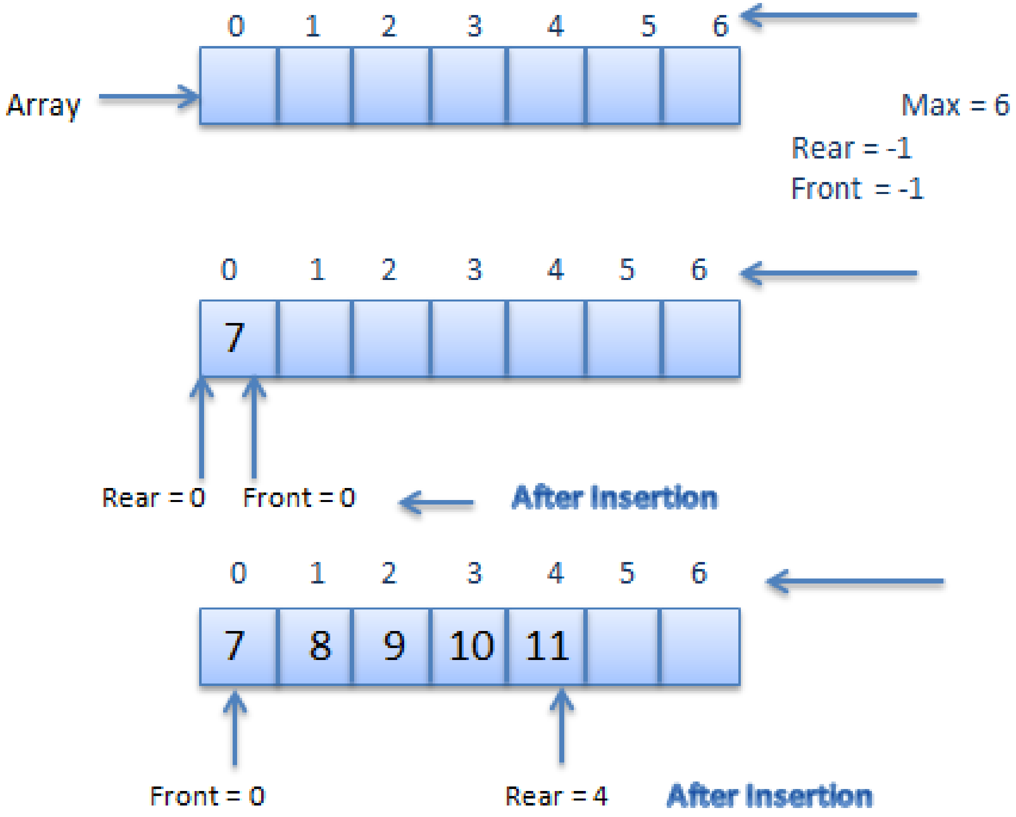
- Queue data structure using array can be implemented as follows...

  Before we implement actual operations, first follow the below steps to create an empty queue.

- **Step 1:** Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
- **Step 2:** Declare all the **user defined functions** which are used in queue implementation.
- **Step 3:** Create a one dimensional array with above defined SIZE (**int queue[SIZE]**)
- **Step 4:** Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**. (**int front = -1, rear = -1**)
- **Step 5:** Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

enQueue(value) - Inserting value into the queue

- In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at **rear**position. The enQueue() function takes one integer value as parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...
  - **Step 1:** Check whether **queue** is **FULL**. (**rear == SIZE-1**)
  - **Step 2:** If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
  - **Step 3:** If it is **NOT FULL**, then increment **rear** value by one (**rear++**) and set **queue[rear]** = **value**.



0 1 2 3 4 5 6

Array

Max = 6

Rear = -1

Front = -1

0 1 2 3 4 5 6

7

Rear = 0 Front = 0

After Insertion

0 1 2 3 4 5 6

7 8 9 10 11

Front = 0

Rear = 4

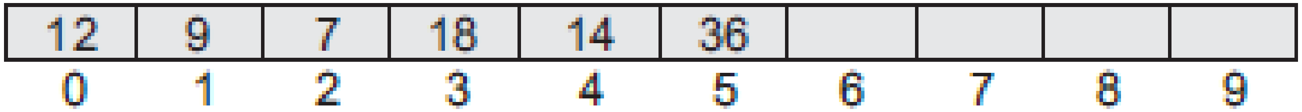
After Insertion

```
Step 1: IF REAR = MAX-1
            Write OVERFLOW
            Goto step 4
        [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

deQueue() - Deleting a value from the Queue

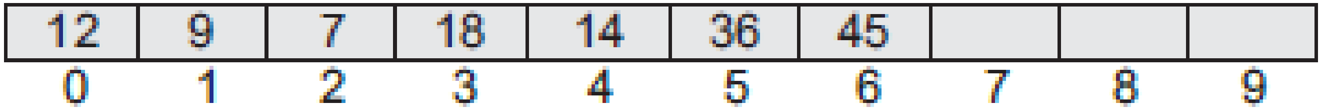
- In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from **front** position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

  - **Step 1:** Check whether **queue** is **EMPTY**. (**front == rear**)
  - **Step 2:** If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
  - **Step 3:** If it is **NOT EMPTY**, then increment the **front** value by one (**front ++**). Then display **queue[front]** as deleted element. Then check whether both **front** and **rear** are equal (**front == rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front** = **rear** = **-1**).



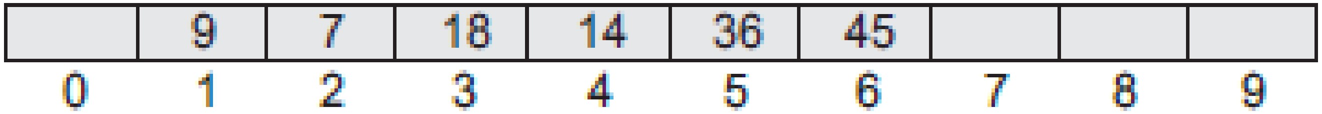
12	9	7	18	14	36				
0	1	2	3	4	5	6	7	8	9

Queue at any instant



12	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

Queue after insertion of a new element



	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

Queue after deletion of an element

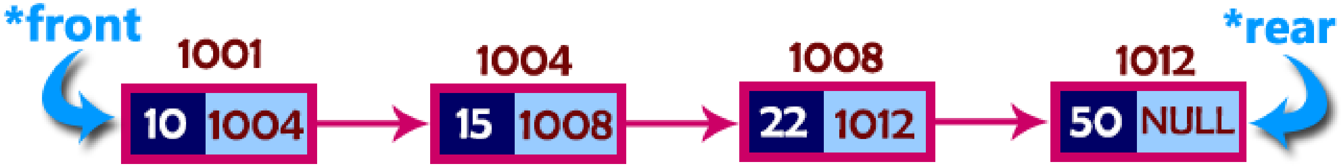
```
Step 1: IF FRONT = -1 OR FRONT > REAR
            Write UNDERFLOW
        ELSE
            SET FRONT = FRONT + 1
        [END OF IF]
Step 2: EXIT
```

display() - Displays the elements of a Queue

- We can use the following steps to display the elements of a queue...
  - **Step 1:** Check whether **queue** is **EMPTY**. (**front == rear**)
  - **Step 2:** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
  - **Step 3:** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front+1**'.
  - **Step 4:** Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value is equal to **rear** (**i** <= **rear**)

Queue using linked list

- The major problem with the queue implemented using array is, It will work for only fixed number of data. That means, the amount of data must be specified in the beginning itself.
- Queue using array is not suitable when we don't know the size of data which we are going to use.
- A queue data structure can be implemented using linked list data structure. The queue which is implemented using linked list can work for unlimited number of values. That means, queue using linked list can work for variable size of data (No need to fix the size at beginning of the implementation).
- The Queue implemented using linked list can organize as many data values as we want.
- In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.



- In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

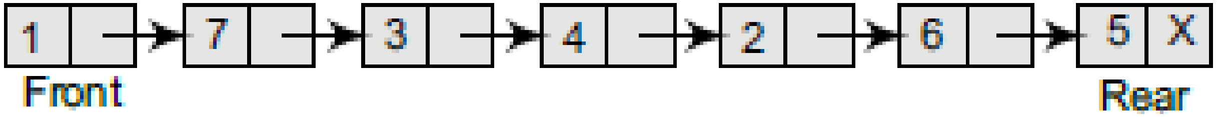
Queue Operations using Linked List

- To implement queue using linked list, we need to set the following things before implementing actual operations.

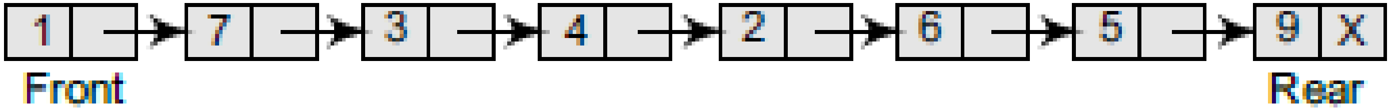
  - **Step 1:** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
  - **Step 2:** Define a '**Node**' structure with two members **data** and **next**.
  - **Step 3:** Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.
  - **Step 4:** Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

- We can use the following steps to insert a new node into the queue...
  - **Step 1:** Create a **newNode** with given value and set '**newNode → next**' to **NULL**.
  - **Step 2:** Check whether queue is **Empty** (**rear == NULL**)
  - **Step 3:** If it is **Empty** then,
    set **front** = **newNode** and **rear** = **newNode**.
  - **Step 4:** If it is **Not Empty** then, set **rear → next** = **newNode** and **rear** = **newNode**.



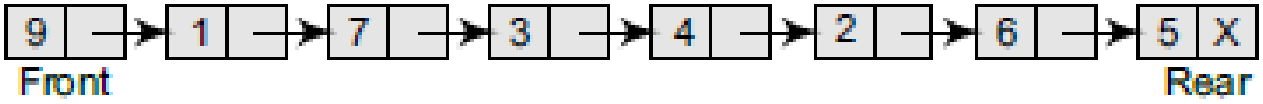
Linked Queue



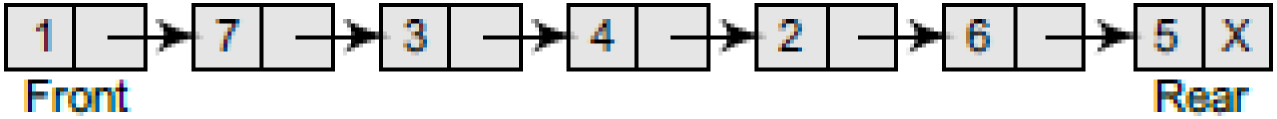
Linked Queue after inserting a new node

deQueue() - Deleting an Element from Queue

- We can use the following steps to delete a node from the queue...
  - **Step 1:** Check whether **queue** is **Empty** (**front == NULL**).
  - **Step 2:** If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function
  - **Step 3:** If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.
  - **Step 4:** Then set '**front = front → next**' and delete '**temp**' (**free(temp)**).



Linked Queue



Linked Queue after deletion of an element

display() - Displaying the elements of Queue

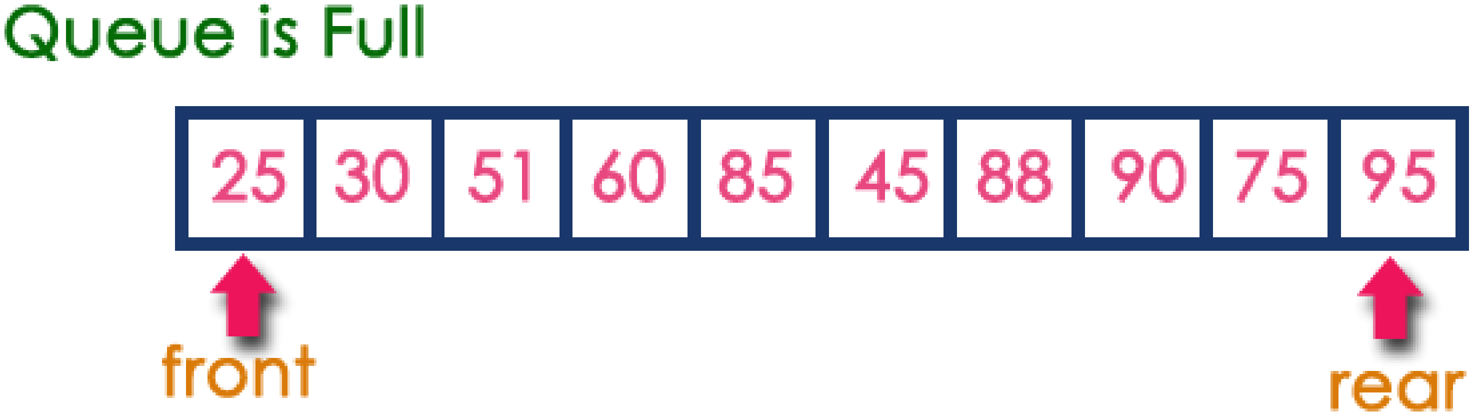
- We can use the following steps to display the elements (nodes) of a queue...
- **Step 1:** Check whether queue is **Empty** (**front** == **NULL**).
- **Step 2:** If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.
- **Step 3:** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.
- **Step 4:** Display '**temp** → **data** --->' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp** → **next** != **NULL**).
- **Step 4:** Finally! Display '**temp** → **data** ---> **NULL**'.

Type of Queue

- A queue data structure can be classified into the following types:
  - 1. Circular Queue
  - 2. Deque
  - 3. Priority Queue
  - 4. Multiple Queue

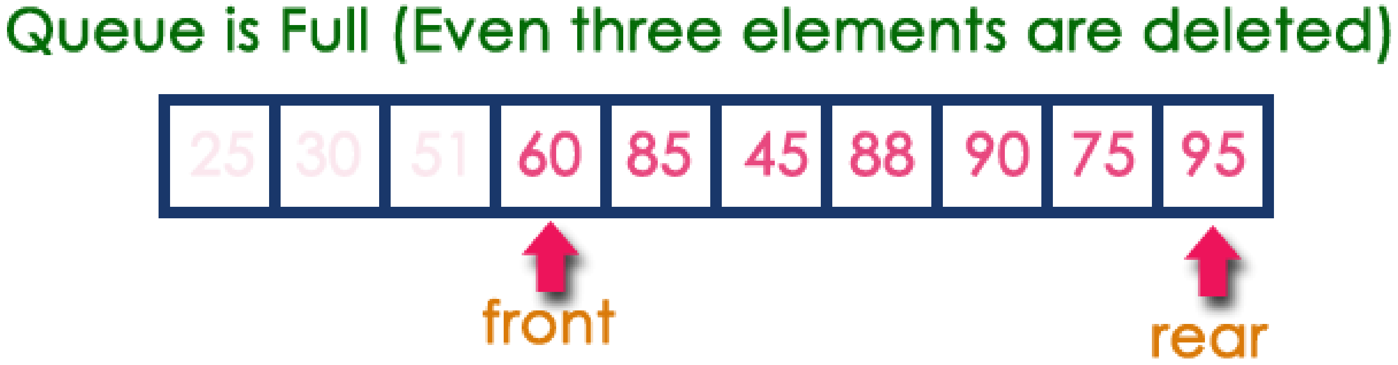
Problems in Queue

- In a normal Queue Data Structure, we can insert elements until queue becomes full. But once if queue becomes full, we can not insert the next element until all the elements are deleted from the queue.
- For example consider the queue below.
- After inserting all the elements into the queue.



Problems in Queue

- Now consider the following situation after deleting three elements from the queue...



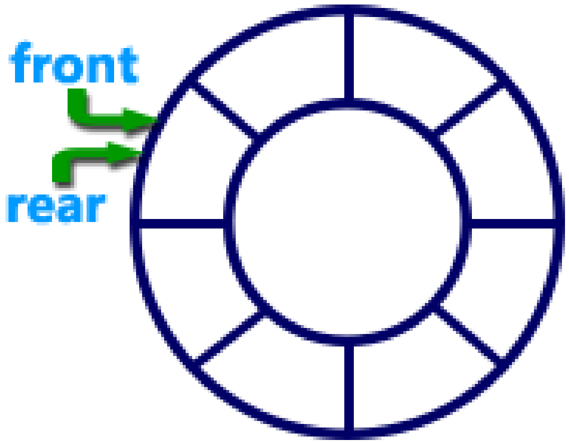
- This situation also says that Queue is Full and we can not insert the new element because, '**rear**' is still at last position. In above situation, even though we have empty positions in the queue we can not make use of them to insert new element. This is the major problem in normal queue data structure.

- To resolve this problem, we have two solutions. First, shift the elements to the left so that the vacant space can be occupied and utilized efficiently. But this can be very time-consuming, especially when the queue is quite large.

- The second option is to use a circular queue. In the circular queue, the first index comes right after the last index.

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.



- The circular queue will be full only when front = 0 and rear = Max – 1.

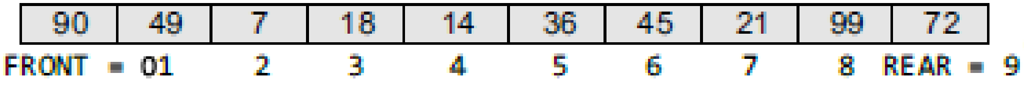
- A circular queue is implemented in the same manner as a linear queue is implemented.

- The only difference will be in the code that performs insertion and deletion operations.

Implementation of Circular Queue

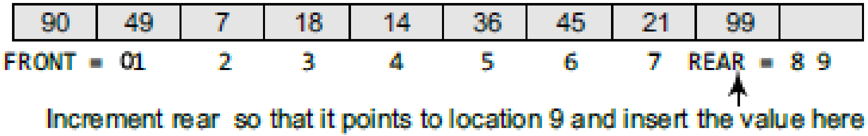
- To implement a circular queue data structure using array, we first perform the following steps before we implement actual operations.
  - **Step 1:** Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
  - **Step 2:** Declare all **user defined functions** used in circular queue implementation.
  - **Step 3:** Create a one dimensional array with above defined SIZE (**int cQueue[SIZE]**)
  - **Step 4:** Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**. (**int front = -1, rear = -1**)
  - **Step 5:** Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

- For insertion, we now have to check for the following three conditions:
  - If front = 0 and rear = MAX – 1, then the circular queue is full. Look at the queue given in below Fig. which illustrates this point.



90	49	7	18	14	36	45	21	99	72
FRONT = 01		2	3	4	5	6	7	8	REAR = 9

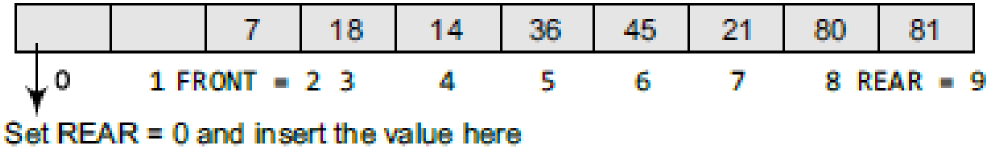
  - If rear != MAX – 1, then rear will be incremented and the value will be inserted as illustrated in below Fig.



90	49	7	18	14	36	45	21	99	
FRONT = 01		2	3	4	5	6	7	REAR = 8	9

Increment rear so that it points to location 9 and insert the value here

  - 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, as shown in following Fig.



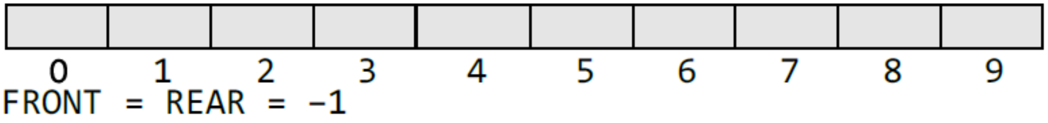
		7	18	14	36	45	21	80	81
0	1	FRONT = 2	3	4	5	6	7	8	REAR = 9

Set REAR = 0 and insert the value here

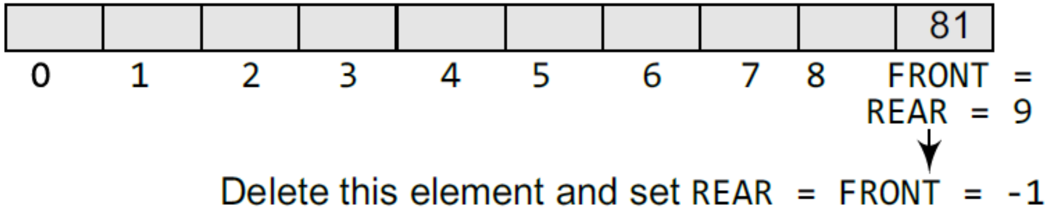
enQueue(value) - Inserting value into the Circular Queue

- In a circular queue, enQueue() is a function which is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as parameter and inserts that value into the circular queue. We can use the following steps to insert an element into the circular queue...
  - **Step 1:** Check whether **queue** is **FULL**. (**(rear == SIZE-1 && front == 0)** || **(front == rear+1)**)
  - **Step 2:** If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
  - **Step 3:** If it is **NOT FULL**, then check **rear == SIZE - 1 && front != 0** if it is **TRUE**, then set **rear = -1**.
  - **Step 4:** Increment **rear** value by one (**rear++**), set **queue[rear]** = **value** and check '**front == -1**' if it is **TRUE**, then set **front = 0**.

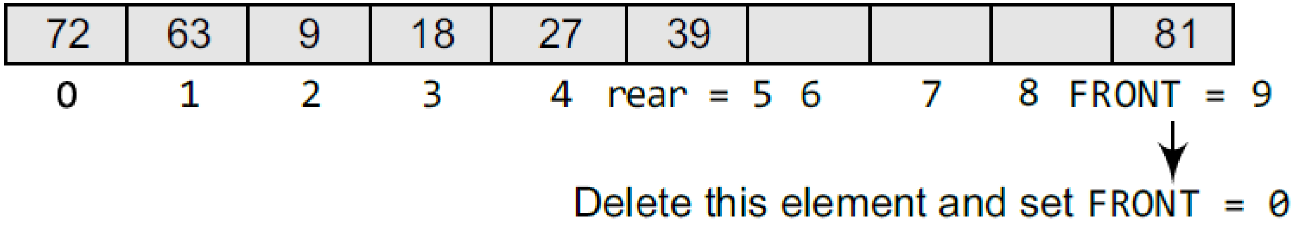
- To delete an element, again we check for three conditions.
  - Look at following Fig. If front = –1, then there are no elements in the queue. So, an underflow condition will be reported.



  - If the queue is not empty and front = rear, then after deleting the element at the front the queue becomes empty and so front and rear are set to –1. This is illustrated in following fig.



- If the queue is not empty and front = MAX–1, then after deleting the element at the front, front is set to 0. This is shown in Fig.



72	63	9	18	27	39				81
0	1	2	3	4	rear = 5	6	7	8	FRONT = 9

Delete this element and set FRONT = 0

deQueue() - Deleting a value from the Circular Queue

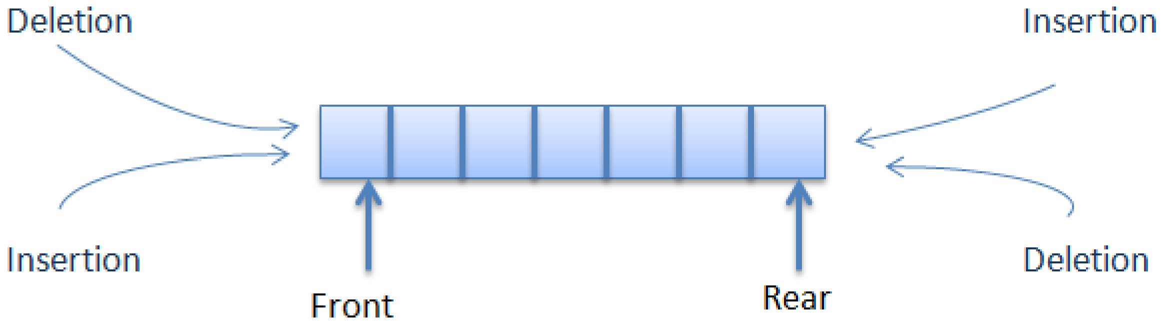
- In a circular queue, deQueue() is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from **front** position. The deQueue() function doesn't take any value as parameter. We can use the following steps to delete an element from the circular queue...
  - **Step 1:** Check whether **queue** is **EMPTY**. (**front == -1 && rear == -1**)
  - **Step 2:** If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
  - **Step 3:** If it is **NOT EMPTY**, then display **queue[front]** as deleted element and increment the **front** value by one (**front ++**). Then check whether **front == SIZE**, if it is **TRUE**, then set **front = 0**. Then check whether both **front - 1** and **rear** are equal (**front -1** == **rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front** = **rear** = **-1**).

display() - Displays the elements of a Circular Queue

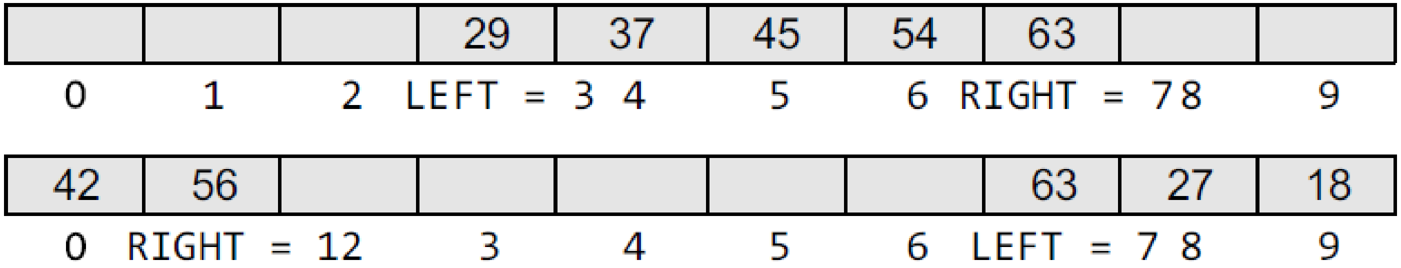
- We can use the following steps to display the elements of a circular queue...
- **Step 1:** Check whether **queue** is **EMPTY**. (**front == -1**)
- **Step 2:** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front**'.
- **Step 4:** Check whether '**front <= rear**', if it is **TRUE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.
- **Step 5:** If '**front <= rear**' is **FALSE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until'**i <= SIZE - 1**' becomes **FALSE**.
- **Step 6:** Set **i** to **0**.
- **Step 7:** Again display '**cQueue[i]**' value and increment **i** value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.

Dequeue

- 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.



- However, no element can be added and deleted from the middle. In the computer’s memory, a deque is implemented using a circular doubly linked list.
- In a deque, two pointers are maintained, LEFT and RIGHT, which point to either end of the deque.
- The elements in a deque extend from the LEFT end to the RIGHT end and since it is circular, Dequeue[N–1] is followed by Dequeue[0].



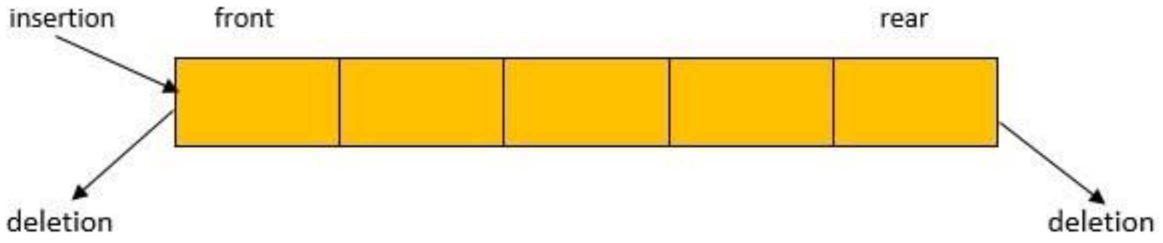
			29	37	45	54	63		
0	1	2	LEFT = 3	4	5	6	RIGHT = 7	8	9

42	56						63	27	18
0	RIGHT = 1	2	3	4	5	6	LEFT = 7	8	9

- Double Ended Queue can be represented in TWO ways, those are as follows...
  - Input Restricted Double Ended Queue
  - Output Restricted Double Ended Queue

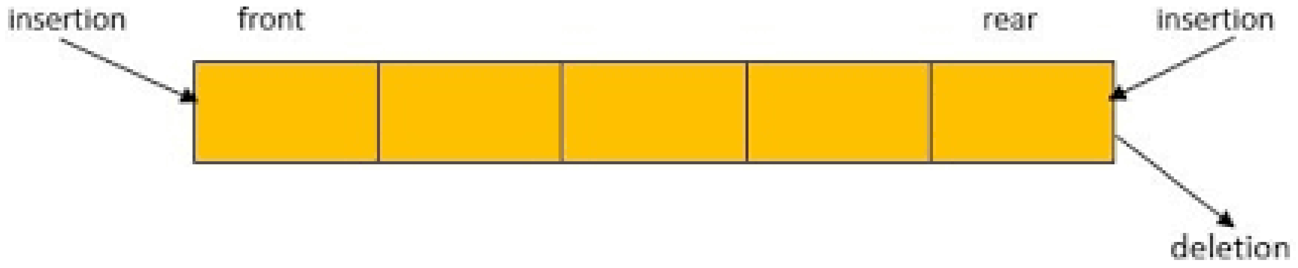
Input Restricted Double Ended Queue

- In this dequeue, insertions can be done only at one of the ends, while deletions can be done from both ends



Output Restricted Double Ended Queue

- In this dequeue, deletions can be done only at one of the ends, while insertions can be done on both ends



Priority Queue

- A priority queue is a data structure in which each element is assigned a priority. The priority of the element will be used to determine the order in which the elements will be processed.

- The general rules of processing the elements of a priority queue are -
  - An element with higher priority is processed before an element with a lower priority.
  - Two elements with the same priority are processed on a first-come-first-served (FCFS) basis.

- A priority queue can be thought of as a modified queue in which when an element has to be removed from the queue, the one with the highest-priority is retrieved first. The priority of the element can be set based on various factors.
- Priority queues are widely used in operating systems to execute the highest priority process first. The priority of the process may be set based on the CPU time it requires to get executed completely.
- For example, if there are three processes, where the first process needs 5 ns to complete, the second process needs 4 ns, and the third process needs 7 ns, then the second process will have the highest priority and will thus be the first to be executed.
- However, CPU time is not the only factor that determines the priority, rather it is just one among several factors. Another factor is the importance of one process over another.

- In case we have to run two processes at the same time, where one process is concerned with online order booking and the second with printing of stock details, then obviously the online booking is more important and must be executed first.

- Priority Queue is an extension of queue with following properties.
  - 1) Every item has a priority associated with it.
  - 2) An element with high priority is dequeued before an element with low priority.
  - 3) If two elements have the same priority, they are served according to their order in the queue.
- A typical priority queue supports following operations.
  - **insert(item, priority):** Inserts an item with given priority.
  - **getHighestPriority():** Returns the highest priority item.
  - **deleteHighestPriority():** Removes the highest priority item.