DATA STRUCTURE

DS Group

ABES Engineering College
College Code-032

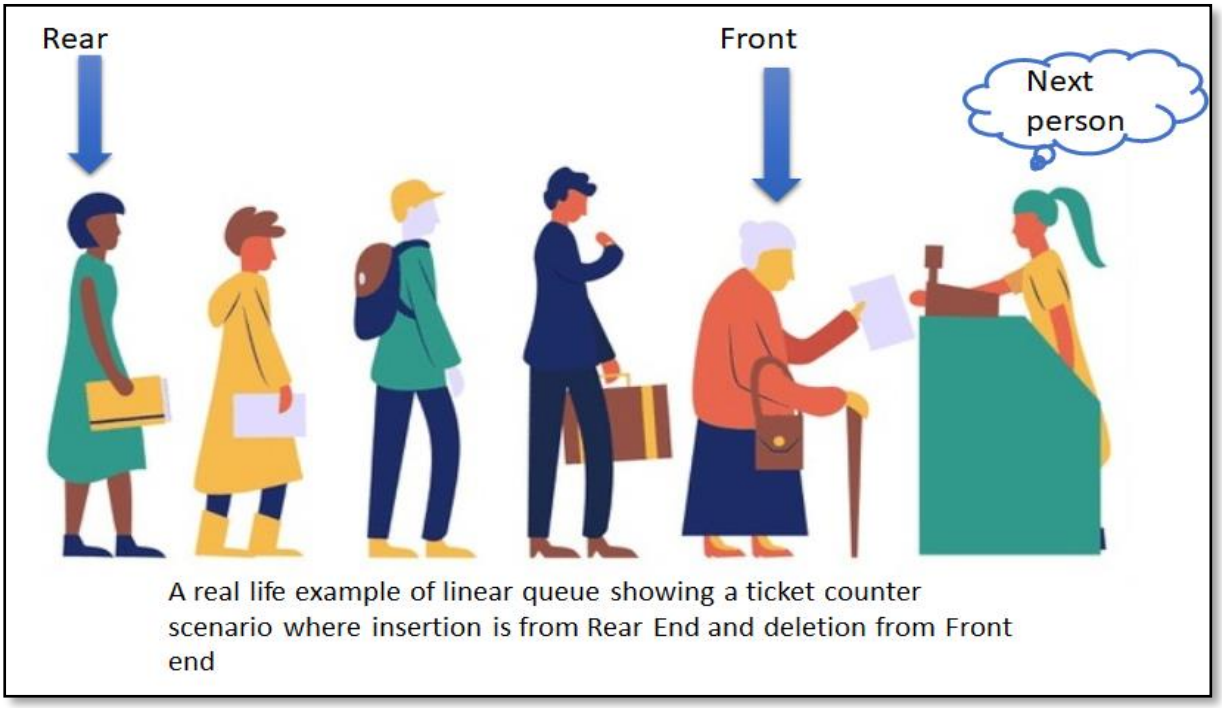
Authors	Akhilesh Kumar Srivastava, Amit Pandey, Amrita Jyoti, Asmita Dixit, Manish Srivastava, Puneet Kumar Goyal
Authorized By	
Creation/Revision Date	16th August 2021
Version	1.0

Chapter 8
Queue

Table of Contents

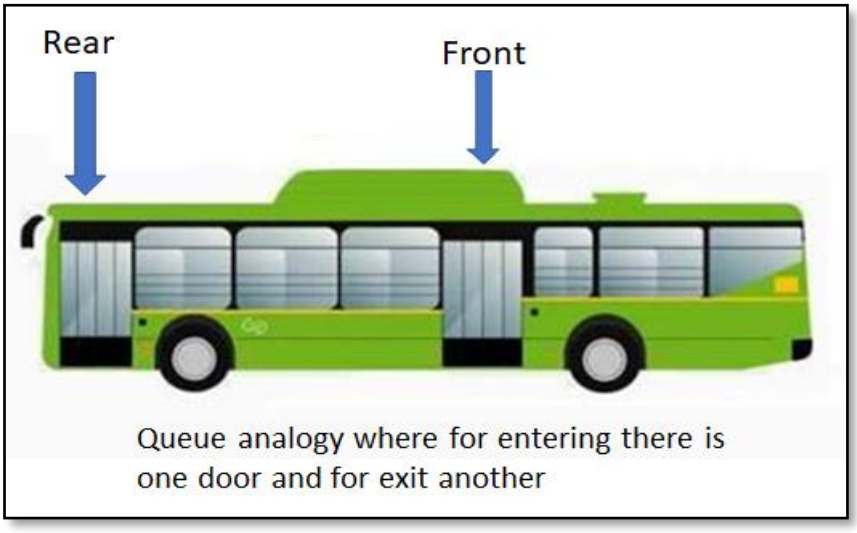
8.1 Introduction

Consider a Scenario where tickets are distributed from a ticket counter. The tickets are distributed in the order of arrival of persons in the Queue. This is a real-life queue where the concept of FIFO (First In First Out) is followed. The person who has bought the ticket will leave from the Front end and a new person who wants to buy the ticket will stand in the Queue at the tail (Rear) end.



A real life example of linear queue showing a ticket counter scenario where insertion is from Rear End and deletion from Front end

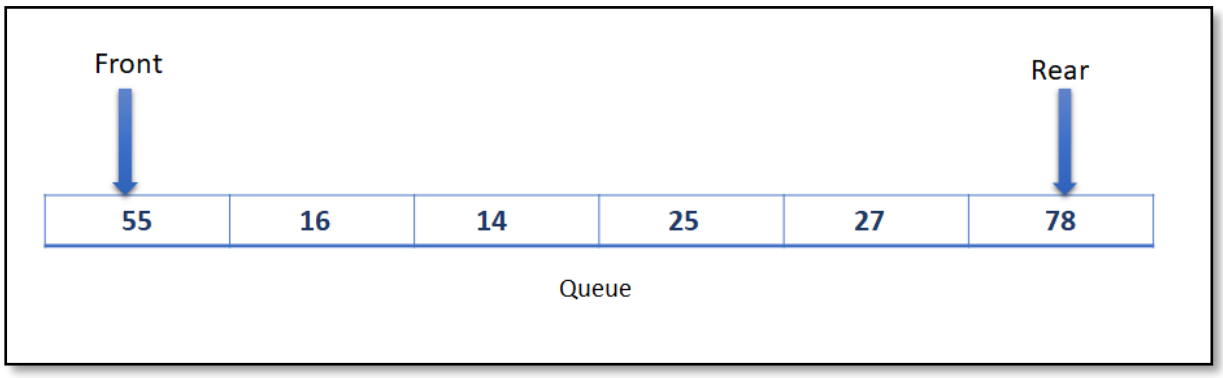
Consider another situation of DTDC bus facility where for the entry of passengers there is one door in the bus and for the exit of passengers another door so that the insertion and deletion can be done in a smooth pattern for same source same destination.



Queue analogy where for entering there is one door and for exit another

8.2 Definition

A Queue is an ordered collection of items into which items may be inserted at one end called **Rear** and removed from another end called **Front**. Ordered means First in First out (FIFO) or First come first serve (FCFS).



Queues are required at every place where there are two entities and a mismatch between the speed of two entities. E.g. you may be very fast in typing but an editor like MS Word may not be able to take your typed characters in the same speed. Hence a buffer is maintained in between that keeps all the typed characters in the sequence.

Another situation where many students are sending the print commands to a printer in a lab. The printer maintains a queue of all the print requests and prints them in the order the commands are being received.
Similarly, any message delivery system on a network divides the messages in the form of packets. The packets are accumulated at the receiver end to ensure delivery. The sender and receiver both maintain a buffer to ensure no loss of packets.

8.3 Types of Queue

There are four types

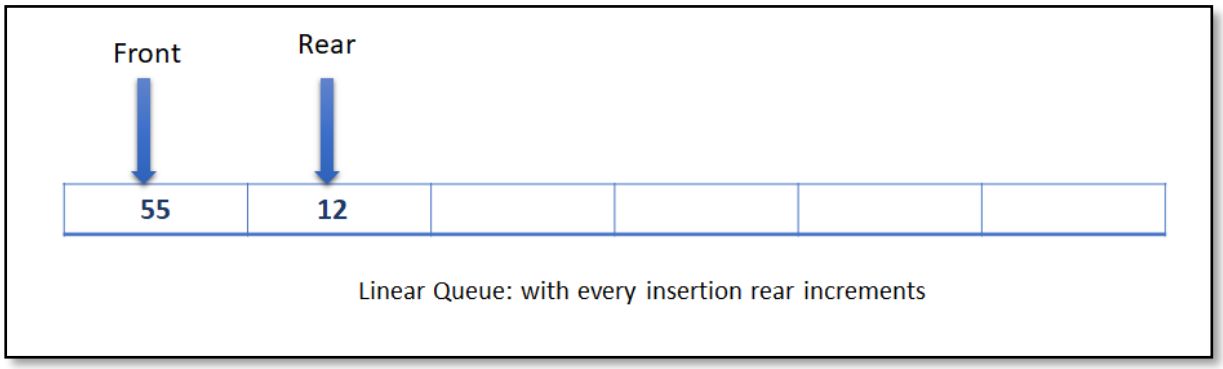
- Linear Queue
- Circular Queue
- Double Ended Queue
- Priority Queue

1. Linear Queue

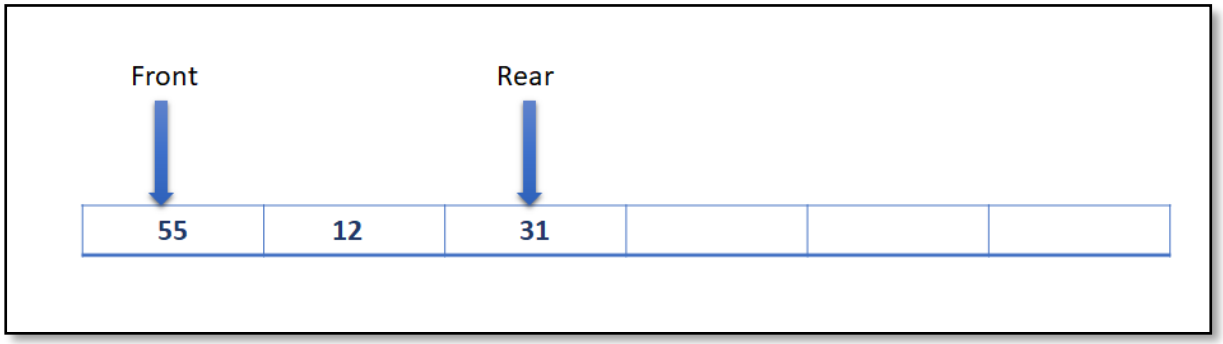
Consider a situation of Bhandara/ Langar in which people wanting the food service are maintaining a queue based on their arrival. A person serving the food starts from head of the queue and visits till the tail of the queue while serving the food to every person in the

queue. Upon reaching to the end of the Queue, the service stops. We need one shot queue only in this case.

Similar is the Linear Queue specially designed for the one-time operation. If the length of the queue has been set as N, only N items can be inserted and N items can be deleted (served).



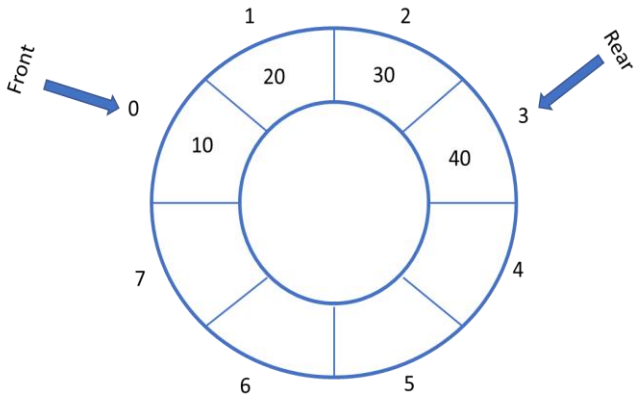
Linear Queue: with every insertion rear increments



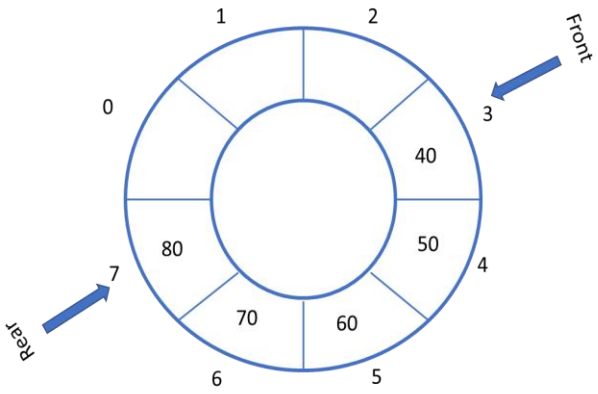
2. Circular Queue

In Circular Queue the Queue Buffer can be re-used in a circular manner. Circular Queue can also be termed as Ring Buffer.

The Circular Queue can be represented as follows:



For every insertion REAR increments to next index



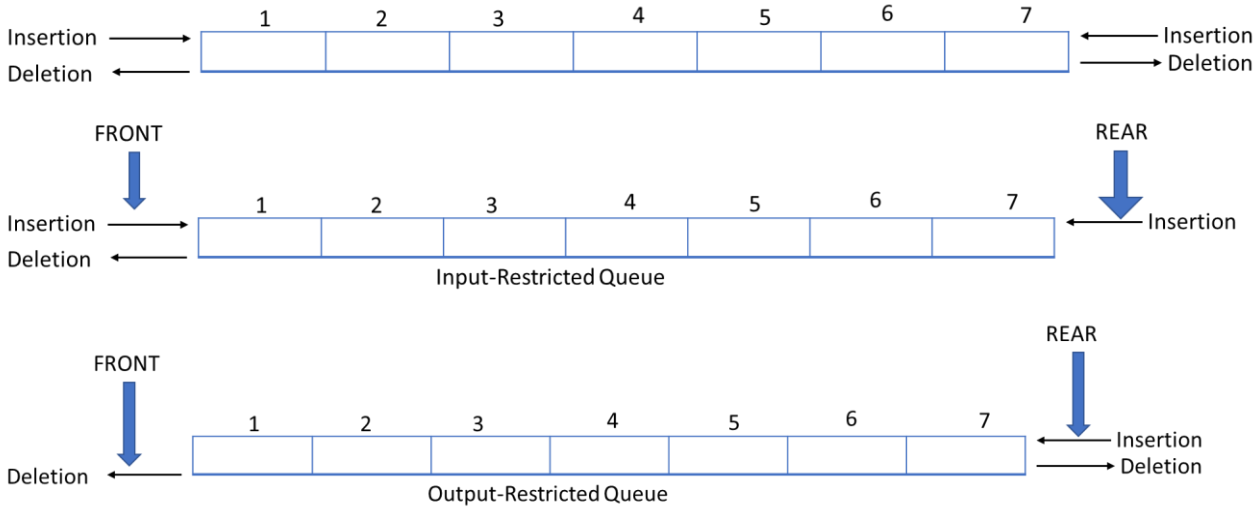
For every deletion FRONT increments to next index

3. Double Ended Queue (Dequeue)

In this type of Queue insertions and deletions can be done at both the ends. There are two variations of Dequeue Possible.

- Input-Restricted Dequeue (Where Deletions are performed from both the ends but Insertion is restricted to either of the ends)
- Output-Restricted Dequeue (Where Insertions are performed at both the ends but Deletion is restricted to either of the ends)

The Dequeue is not bound to the FIFO concept.

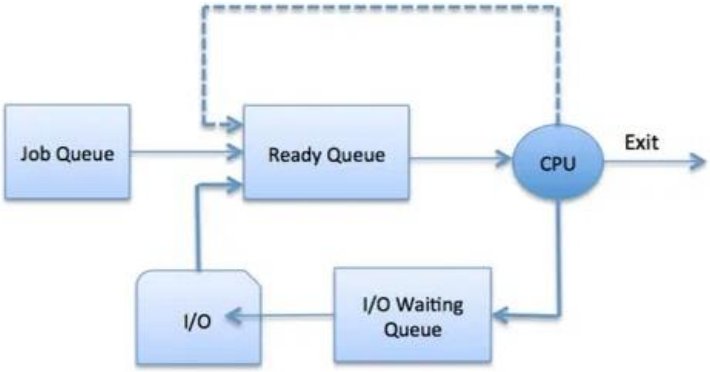


4. Priority Queue

In Priority Queue each item to be inserted has a pre-defined priority associated with it. In priority queue, insertion of items takes place at the Rear in the order of item arrival, deletion of items takes place at the Front based on the priority of items.

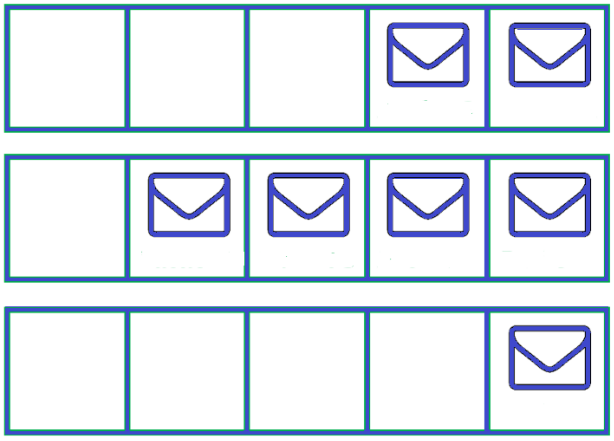
8.4 Applications of Queue

1. **Queue used in CPU Scheduling**: The different states of the processes are defined by the queues in which they are placed. The queue data structure manages the entire conduction of CPU scheduling as the process that comes first comes out first.



2. **Resource allocation management in Operating System**: The different hardware and software resources are managed by queue, as in printer the request made first is printed first. That is, spooling process in printer.

3. **Queue in-network Routing:** While delivery network packets over the network queue data structure is used, as the packet or mail delivered first must be received first at receiver end.



4. **Call -Center Waiting process in a telephonic call:** The management of different calls in queue order is one real-life scenario of queue data structure.

5. **Queue data structures are used for Interrupt Handling in Operating System:** For interrupt handling in operating system different queues are used.

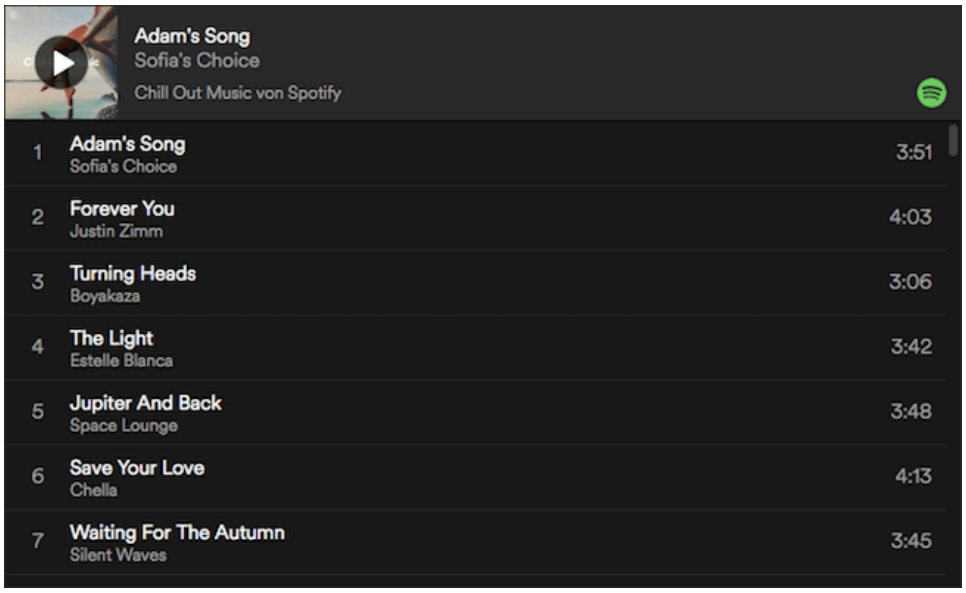
8.5 Primitive Operations of Queue

Following operations are performed on Queue in order to achieve FIFO behavior.

a) Initialization
b) Emptiness check
c) Insertion (Enqueue)
d) Deletion (Dequeue)

8.6 Linear Queue

When I am stressed or want to be relaxed, I simply choose my favorite songs and pile them in a line such that the song I added first in the playlist will be played and deleted from the list and once it is deleted second song is played and so on. In the same way, Linear Queue also operates. Elements are added at one end known as **Rear** end and is deleted from the other end known as **Front** End.



Queue Example: - Simple Analogy of Playlist

8.6.1 Primitive Operations using Array

Let us explain the primitive operations one by one assuming Array implementation. There are two elements in each Queue

- Array to store data (we are naming it as Data [ ])
- Rear (to store the index of the Rear element)
- Front (to store the index of the Front element)

If Queue were declared with the name Q, its elements would be represented as:
Q.Rear: representing the Rear pointing to the Last element inserted in Queue.

Q.Front: representing the Front pointing to the element which will be served next in the Queue.
Q.Data[ ]: is the buffer storing all the elements.

1. Initialization
Every Queue, before being used, should be initialized. Initialization must depict that the Queue contains zero elements at the beginning. For example, if we assume the array indices to vary from 0 to N-1 (where N is the array size), If Rear is initialized to 0 index, it means an element is there at the 0^{th} index. Hence, we should initialize the Rear to invalid index, say -1. Front is initialized at 0.

ALGORITHM Initialize (Queue Q)
Input: Linear Queue Q
Output: None
BEGIN:
Q.Rear= -1
Q.Front = 0
} Rear initialized at -1, Front at 0 index
END;

This can be shown with the given diagram.

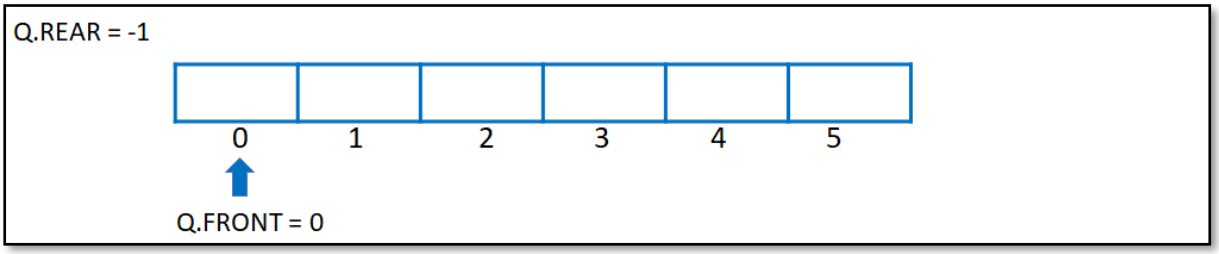


Figure: - Queue Initialization

Time Complexity: Time Complexity of this Operation is Θ(1) as there are only two statements to execute.
Space Complexity: Since there is no auxiliary space used in the Algorithm, the space function is 0 (constant) i.e., the Space Complexity of this Operation is Θ (1).

2. Emptiness check
If there are no elements in the Queue then it is said to be empty. Number of elements in a linear queue can be found by the formula: Rear – Front + 1.
E.g.
1. If Rear is at 5 index and Front at 2, number of elements will be 5 – 2 + 1 = 4.
2. If Rear is at 4 index and Front at 5, number of elements will be 4 – 5 + 1 = 0
3. If Rear is at -1 index and Front at 0, number of elements will be -1 – 0 + 1 = 0

If this is equal to zero, the queue will be empty. The function given below is a Boolean valued function that returns TRUE or FALSE based on emptiness.

ALGORITHM Empty(Queue Q)
Input: Circular Queue CQ
Output: True or False based on Emptiness
BEGIN:

```
IF Q.Rear – Q.Front + 1 = = 0  THEN
        RETURN TRUE
ELSE
        RETURN FALSE
```

If Rear – Front +1 equals zero meaning that zero elements are there in the Queue

END;

The example given below show the empty queue.

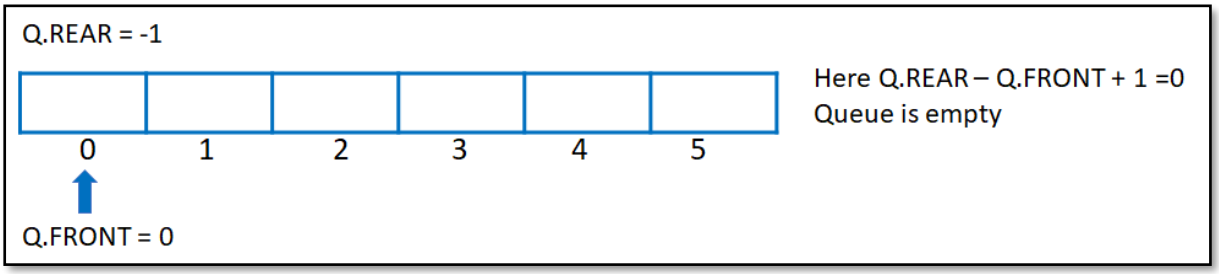


Figure:- Empty Queue

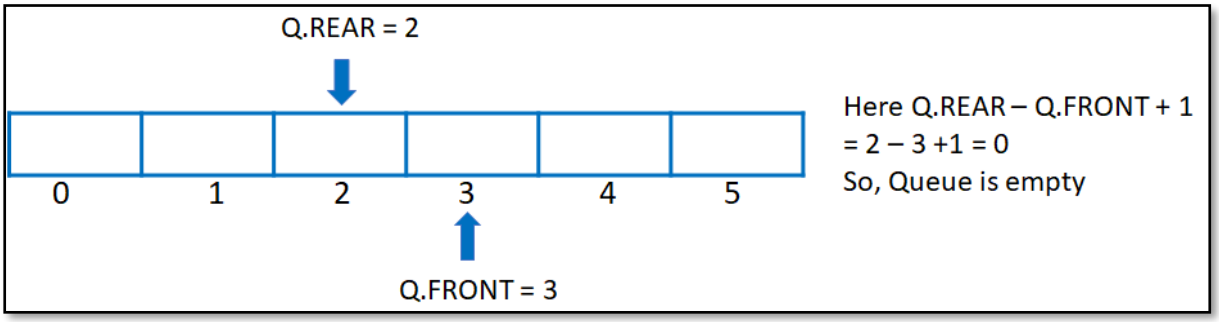


Figure:- Empty Queue

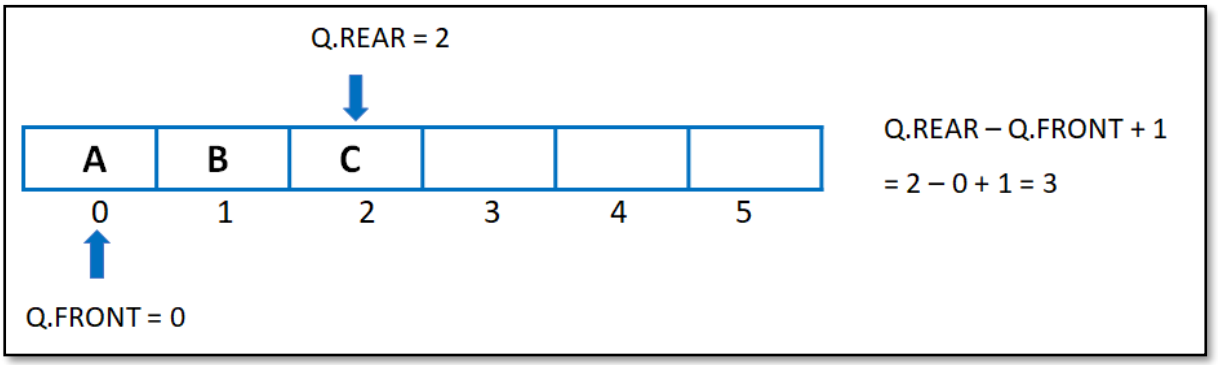


Figure:- Non-Empty Queue

Time Complexity: Time Complexity of this Operation is Θ(1) as there are only two statements to execute even when the condition is TRUE or FALSE. Here condition check is considered as the first statement and return as the second one.
Space Complexity: Since there is no auxiliary space used in the Algorithm, the space function is 0 (constant) i.e. Space Complexity of this Operation is Θ (1).

3. Insertion

Insertion operation in Queue is given a standard name Enqueue. Rear end is used to insert an item. Insertion requires the overflow check. If the Rear has reached to the Maximum index in the Queue, further insertions will not be possible. If the Rear is less than the maximum index, insertion is performed by incrementing the Rear to the next higher index and then setting the item to be inserted at an updated Rear position.

Overflow: *Consider a bucket where water is filled completely. If we pour more water into it, the water will overflow. Similarly, an attempt to insert an element in the full Queue leads to the condition of overflow.*

ALGORITHM EnQueue (Queue Q, x)
Input: Circular Queue CQ and x a new element to be inserted
Output: None
BEGIN:

```
IF Q.Rear = = MAXQUEUE -1 THEN
        WRITE ("Queue Overflows")
        EXIT(1)
ELSE
        Q.Rear =Q.Rear + 1
        Q.Item[Q.Rear] = x
```

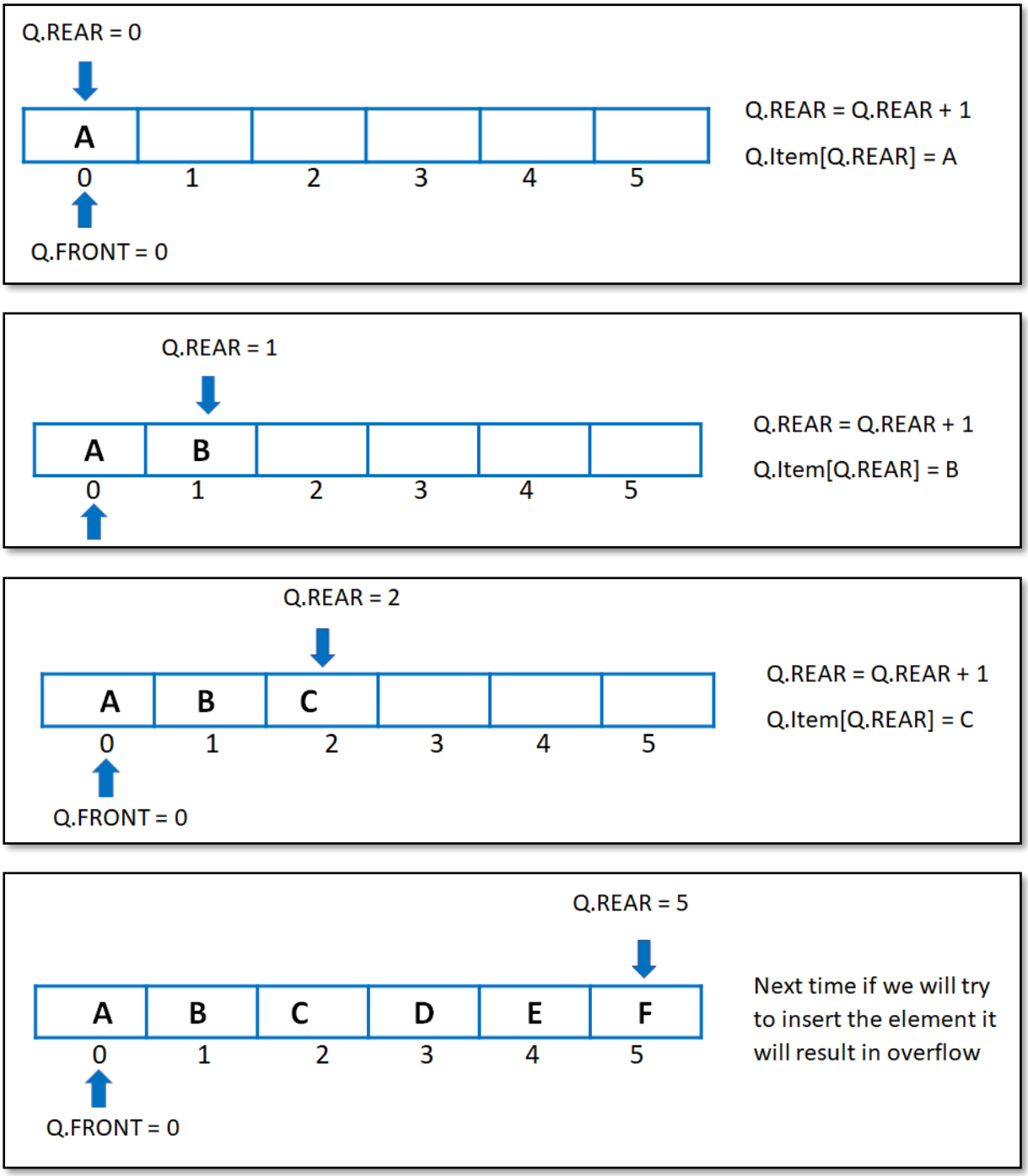
If no more insertion possible

Increment the rear to next index

Add new element in circular queue buffer

END;

The enqueue operation can be explained using the following diagrams. Suppose I am standing in a queue at movie theater, I am the first one to be in queue followed by two more persons. I am representing myself as P and rest two as Q and R. So, the order of insertion in queue will be like this: -



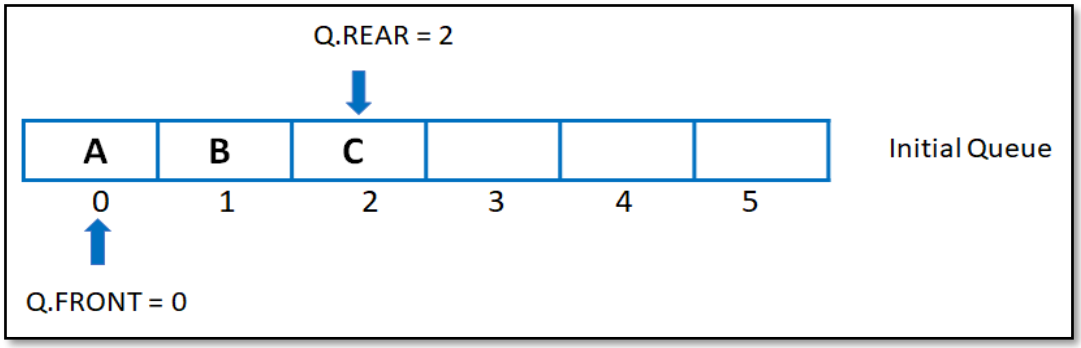
Time Complexity: Time Complexity of this Operation is Θ(1) as there are only three statements to execute even if the condition is TRUE or FALSE. Here condition check is considered as first statement.

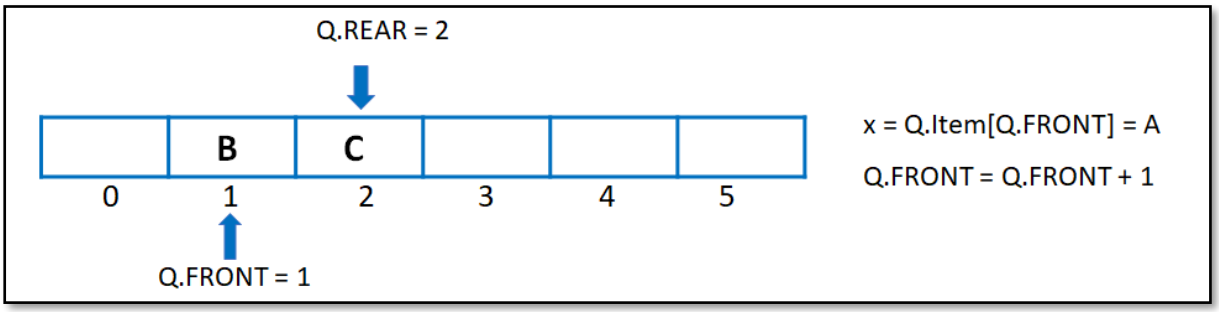
Space Complexity: Since there is no auxiliary space used in the Algorithm, the space function is 0 (constant) i.e., Space Complexity of this Operation is Θ (1).

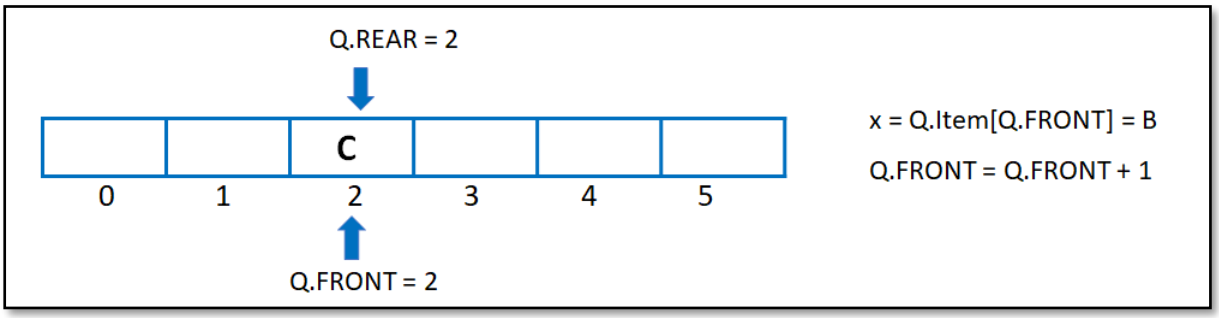
4. Deletion

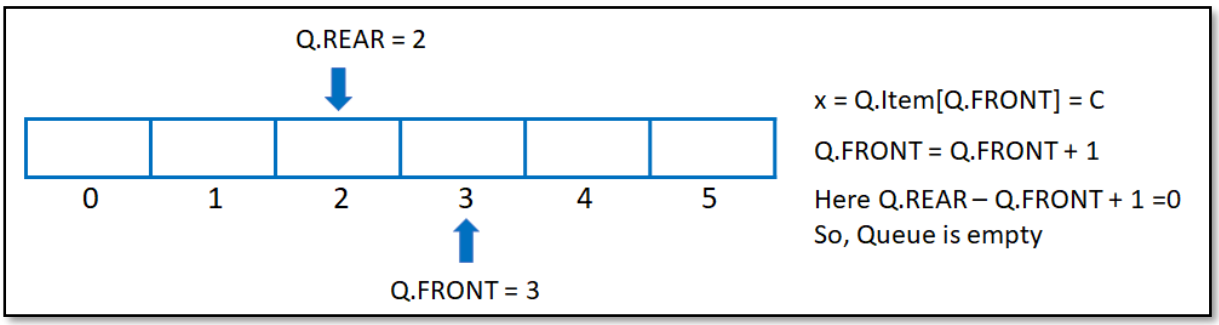
Deletion operation in Queue is given a standard name Dequeue. In the above example, the first one to get a ticket will be P and will be the first one to be deleted, followed by Q and R.

Deletion requires the underflow check (An attempt to delete an item in the empty data structure results in the underflow). If the Queue is Empty, this will result in the underflow. If the Queue is not empty, the Front element is saved in some temporary and Front is updated to the next higher index.









ALGORITHM DeQueue(Queue Q)

Input: Circular Queue CQ

Output: Deleted item from front index of Circular Queue

BEGIN:

IF EMPTY (Q) THEN

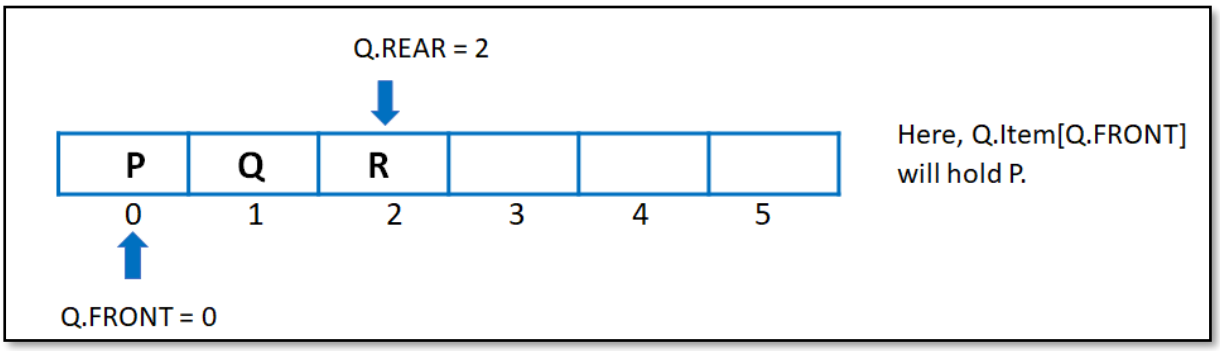
```
            WRITE ("Queue Underflows")
            EXIT (1)                                  } If no more deletion possible
      ELSE
            x = Q.Item[Q. Front]                      } Saving the front index element in x

            Q.Front = Q.Front + 1                     } Increment the front index to next index
            RETURN x
END;
```

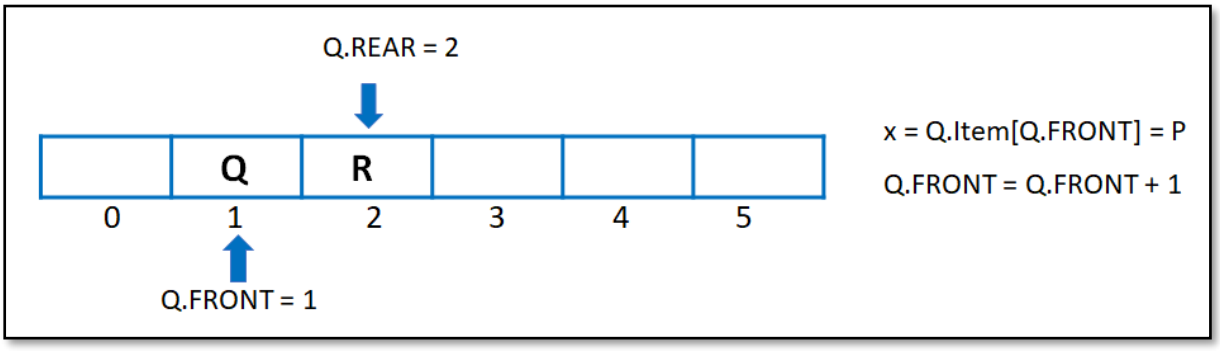
Time Complexity: Time Complexity of this Operation is Θ(1) as there are only three statements to execute even if the condition is TRUE or FALSE. Here condition check is considered as first statement.

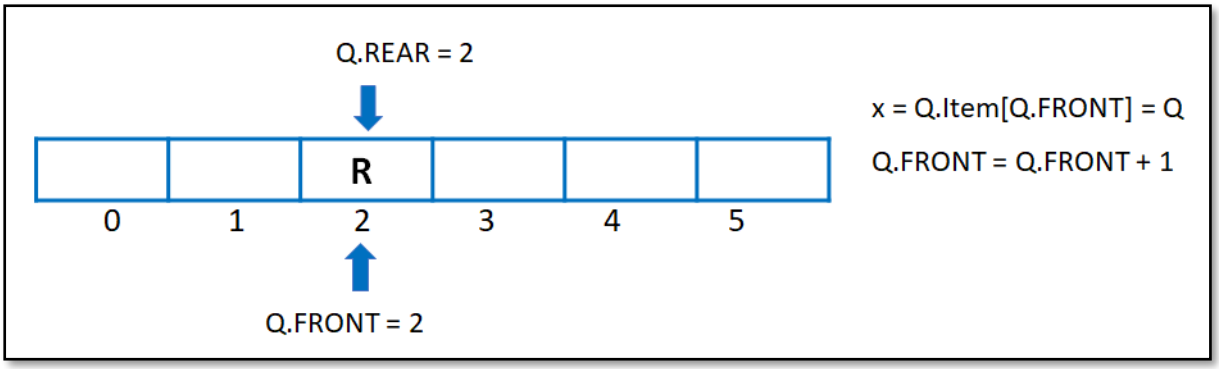
Space Complexity: Since there is no auxiliary space used in the Algorithm, the space function is 0 (constant) i.e., Space Complexity of this Operation is Θ (1).

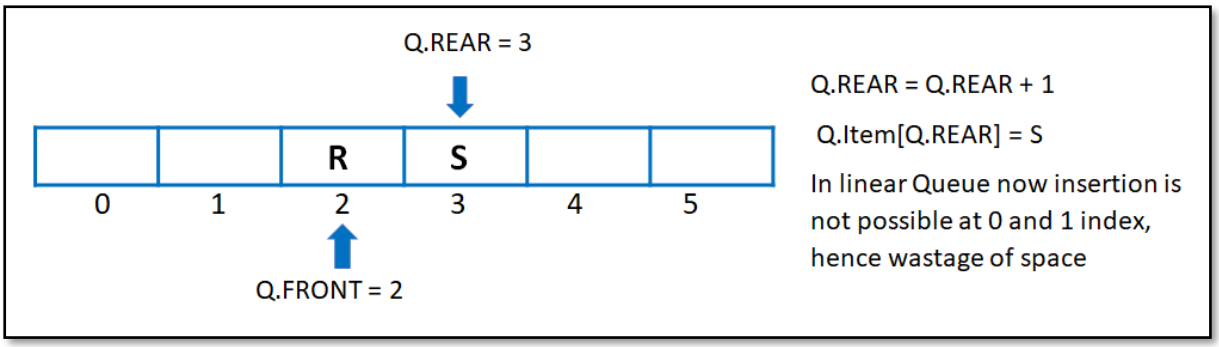
8.6.2 Limitation of Linear Queue

There is a disadvantage in array implementation of Linear Queue. The memory space gets wasted in case Rear has moved to higher indexes. E.g.





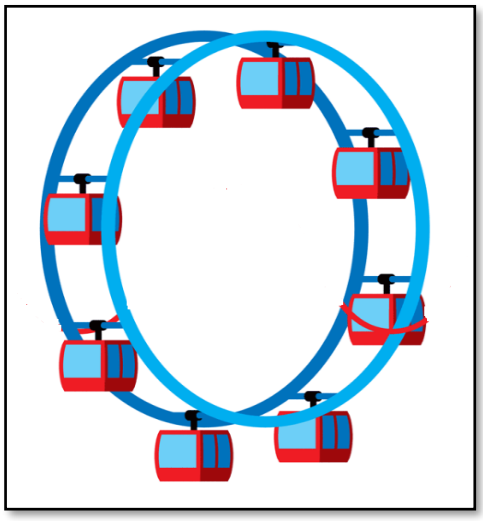




8.7 Circular Queue

8.7.1 Introduction

Just like the adventure swing has limited seats, every time the last seat is filled and the first seat is empty, it can be filled again from start in a circular manner. The same concept is applied to circular queue where the buffer to store the data is limited but just with the use of modulo operation the buffer can be re-filled circularly. Just like Linear queue the Circular Queue also follows the concept of First In First Out (FIFO) but here the additional quality is to re-use the space from the beginning if empty.



8.7.2 Why Circular Queue?

Circular Queue overcomes the limitation faced in Linear queue. That means in a Linear queue if the Rear has reached the last index of queue buffer but there are few spaces free in the beginning yet Rear cannot be replaced to first index, although this can now be done in Circular Queue for both insertion and deletion.

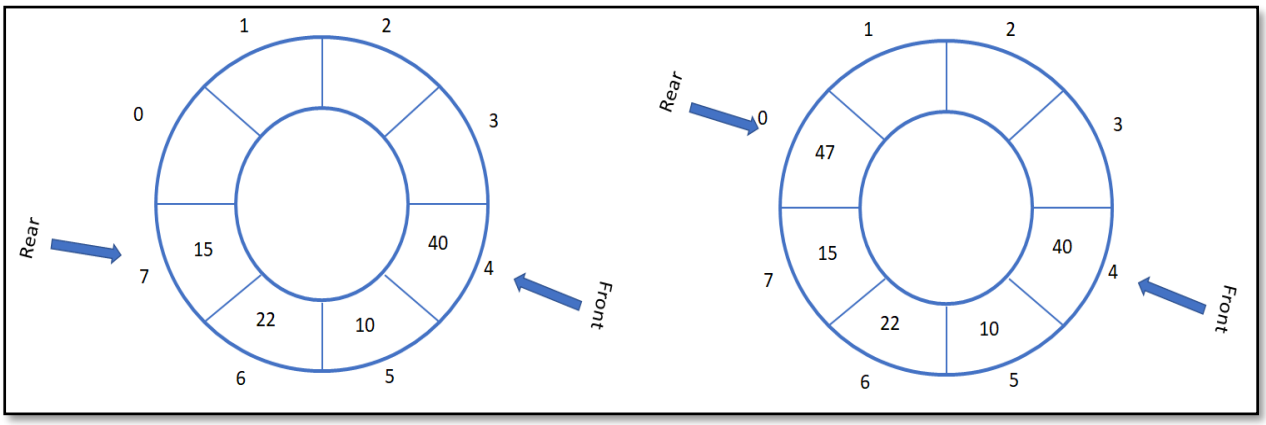


Figure: In Circular Queue if the Rear has reached the last index and there is space from the first index, then Rear can be started again from first index in circular fashion.

The Limitation of Linear queue can simply be overcome in Circular queue by a mathematical rule for both insertion and deletion:

	In Linear Queue	**In Circular Queue**
Insertion:	Rear =Rear+1	Rear = (Rear+1)%MAXSIZE
Deletion :	Front = Front +1	Front = (Front+1)%MAXSIZE

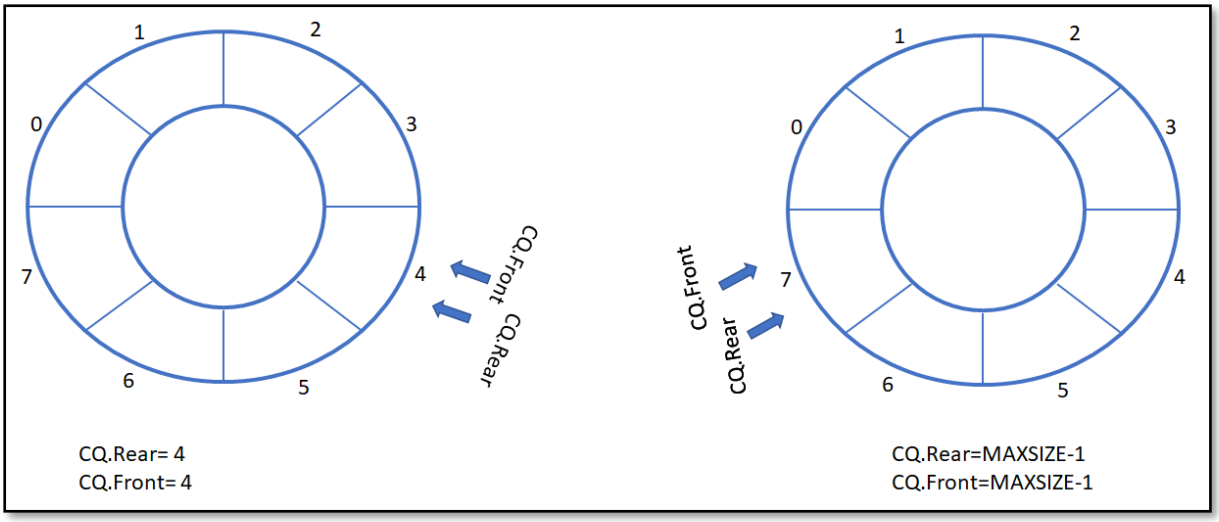
8.7.3 Primitive Operations of Circular Queue

1. Initialization

Here we initialize the Front and Rear end of Queue. Usually, Rear and Front are initialized at largest index of the array. However, it can be initialized at any index but both should be initialized at the same index.

ALGORITHM Initialize(CQueue CQ)
Input: Circular Queue CQ
Output: None
BEGIN:
    CQ.Rear=MAXSIZE-1
    CQ.Front=MAXSIZE-1
END;

} Initializing Rear and Front at MAXSIZE-1



Time Complexity: Time Complexity of this Operation is Θ(1) as there are only two statements to execute.

Space Complexity: Since there is no auxiliary space used in the algorithm, the space function is 0 (constant) i.e., the Space Complexity of this Operation is Θ (1).

2 Emptiness check

If there are no elements in the Queue then it is said that it is empty. If Front and Rear both are at the same index, the queue will be empty. The function given below is a Boolean valued function that returns TRUE or FALSE based on emptiness.

ALGORITHM Empty(CQueue CQ)

Input: Circular Queue CQ

Output: True or False based on Emptiness

BEGIN:

```
IF CQ.Front == CQ.Rear
    RETURN TRUE
ELSE
    RETURN FALSE
```

CQ.Front = = CQ.Rear means that Circular Queue is empty.

END;

3. Insertion

Insertion operation in Queue is given a standard name Enqueue. Rear end is used to insert an item. Insertion requires the overflow check. We first update the Rear and check if it is equal to Front. This means there are N-1 elements already on the queue. If we allow this updation then Rear will become equal to Front. This is the situation of emptiness; hence we should avoid this situation from taking place. Next, revert the changes in the Rear and declare that Queue overflows. This simply means that in a N-sized Queue we can

accommodate only N-1 elements. If underflow does not take place, insert the item to be inserted at the updated Rear position.

ALGORITHM EnQueue(CQueue CQ, x)
Input: Circular Queue CQ and x a new element to be inserted
Output: None
BEGIN:

```
IF (CQ.Rear+1)%MAXSIZE== CQ.Front THEN
        WRITE ("QUEUE OVERFLOW")
        EXIT(1)
CQ.Rear=(CQ.Rear+1)%MAXSIZE

CQ.Buffer[CQ.Rear]=x
```

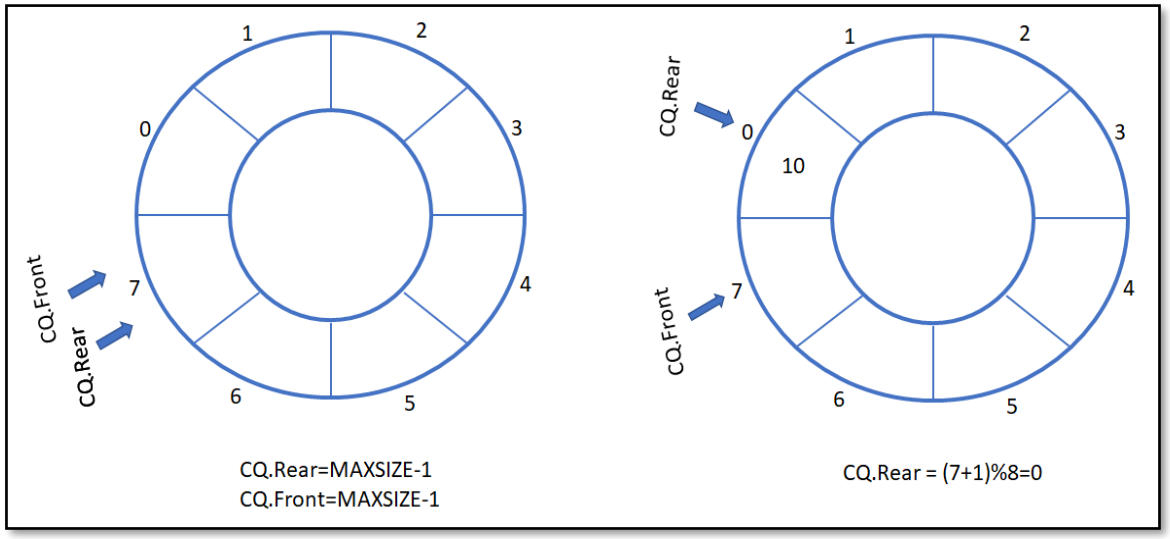
END;

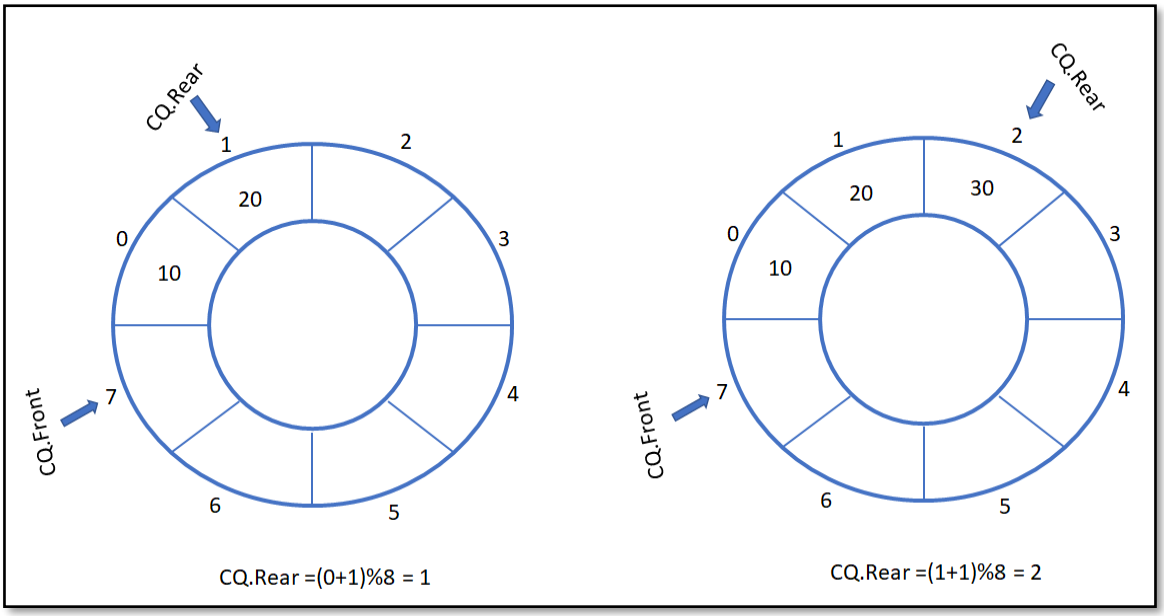
If no more insertion possible

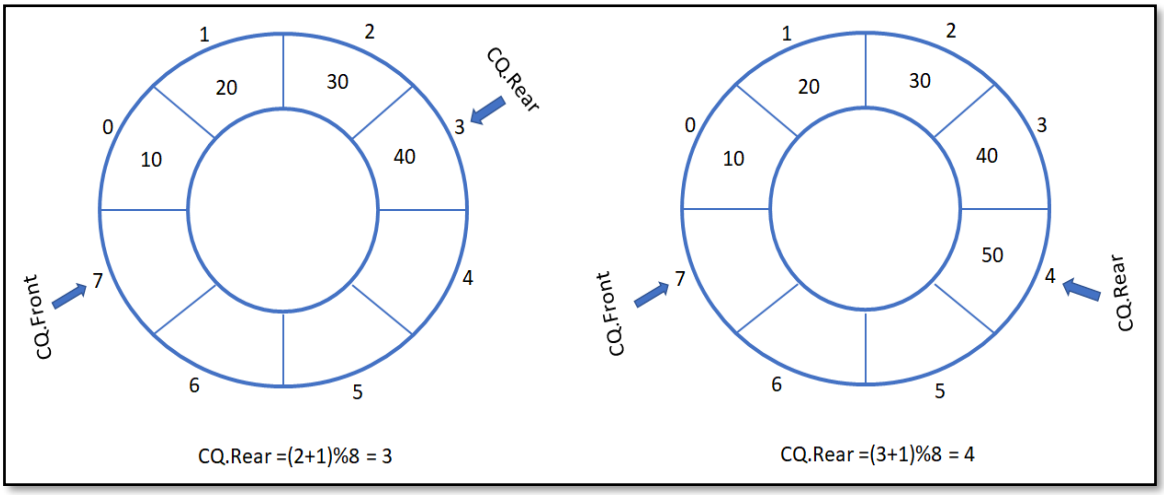
Increment the rear to next index or from last index to first circularly

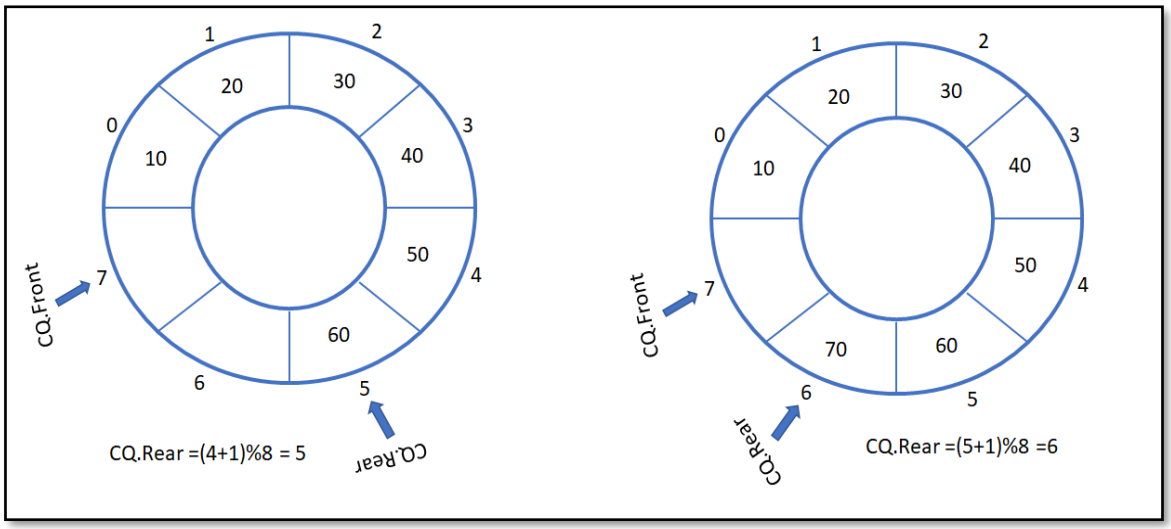
Add new element in circular queue buffer

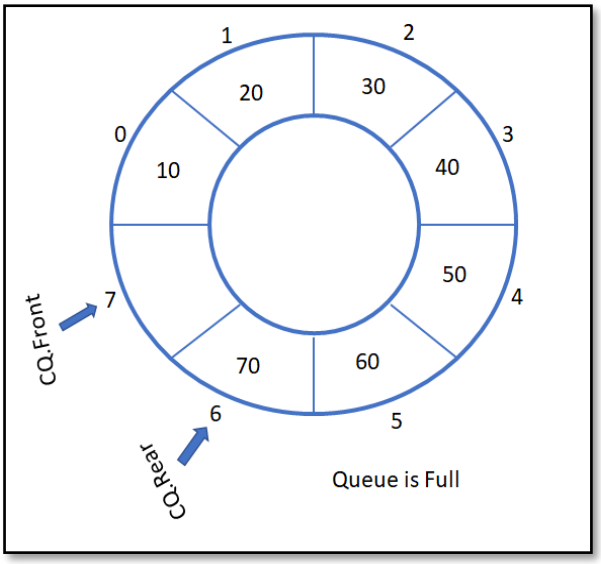
Time Complexity: Time Complexity of this Operation is Θ(1) as there are only three statements to execute.
Space Complexity: Since there is no auxiliary space used in the Algorithm, the space function is 0 (constant) i.e., Space Complexity of this Operation is Θ (1).




CQ.Rear
1
2
20
3
0
10
CQ.Front
7
4
6
5
CQ.Rear =(0+1)%8 = 1
CQ.Rear
1
2
20
30
3
0
10
CQ.Front
7
4
6
5
CQ.Rear =(1+1)%8 = 2


1
2
20
30
CQ.Rear
3
0
10
40
CQ.Front
7
4
6
5
CQ.Rear =(2+1)%8 = 3
1
2
20
30
3
0
10
40
50
CQ.Front
7
4
CQ.Rear
6
5
CQ.Rear =(3+1)%8 = 4


1
2
20
30
3
0
10
40
50
CQ.Front
7
4
60
6
5
CQ.Rear =(4+1)%8 = 5
CQ.Rear
1
2
20
30
0
10
3
40
50
CQ.Front
7
4
70
60
6
CQ.Rear
5
CQ.Rear =(5+1)%8 =6



4. Deletion

Deletion operation in Queue is given a standard name Dequeue. Deletion requires the underflow check (An attempt to delete an item in the empty data structure results in the underflow). If the Queue is Empty, Deletion from the Queue will not be possible. If the Queue is not empty, Front is updated to the next higher index and Front element is saved in some temporary.

ALGORITHM DeQueue(CQueue CQ)

Input: Circular Queue CQ

Output: Deleted item from front index of Circular Queue

BEGIN:

```
IF Empty(CQ) THEN
        WRITE("Queue Underflow")
        EXIT(1)
CQ.Front=(CQ.Front+1)%MAXSIZE

x=CQ.Buffer[CQ.Front]
RETURN x
```

If no more deletion possible

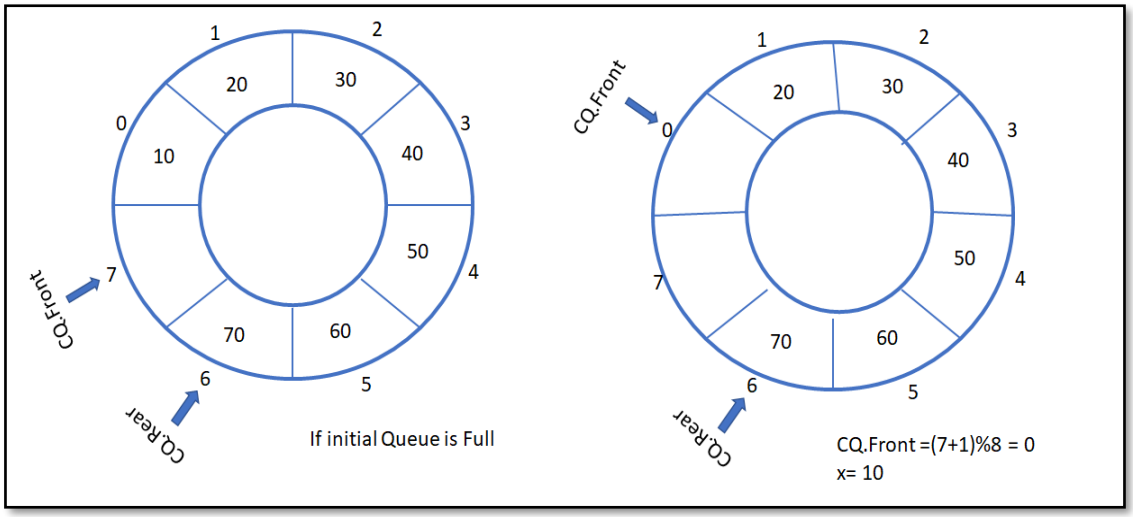
Increment the front index to next index or from the last index to first circularly

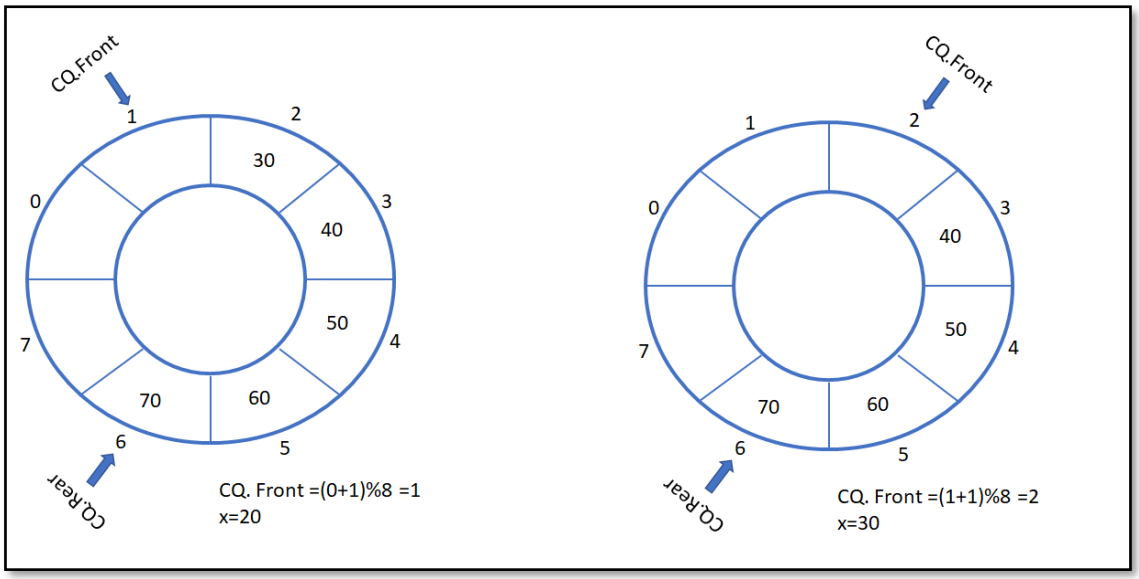
Saving the front index element in x
Returning the deleted element

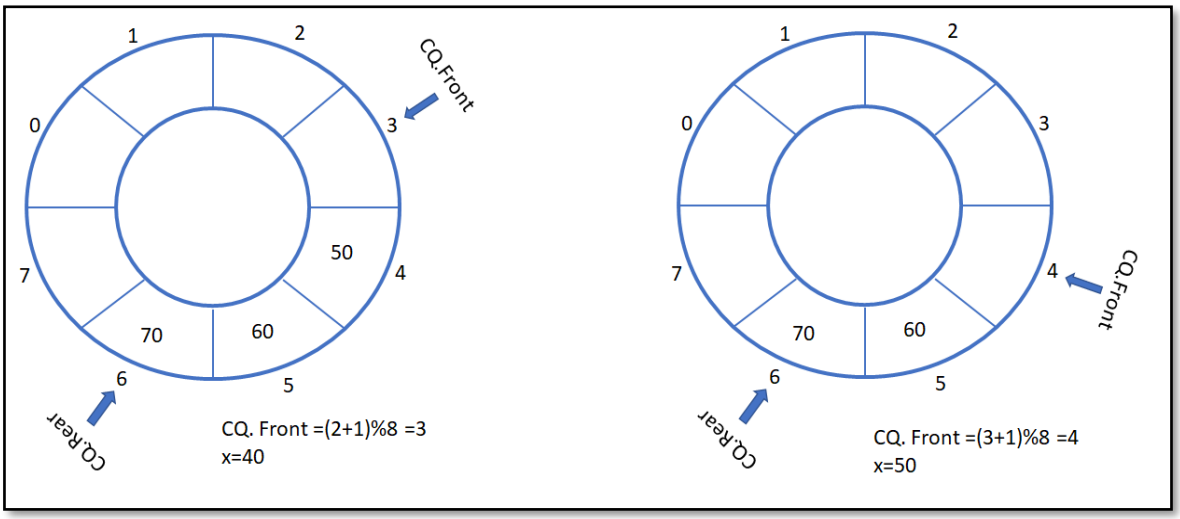
END;

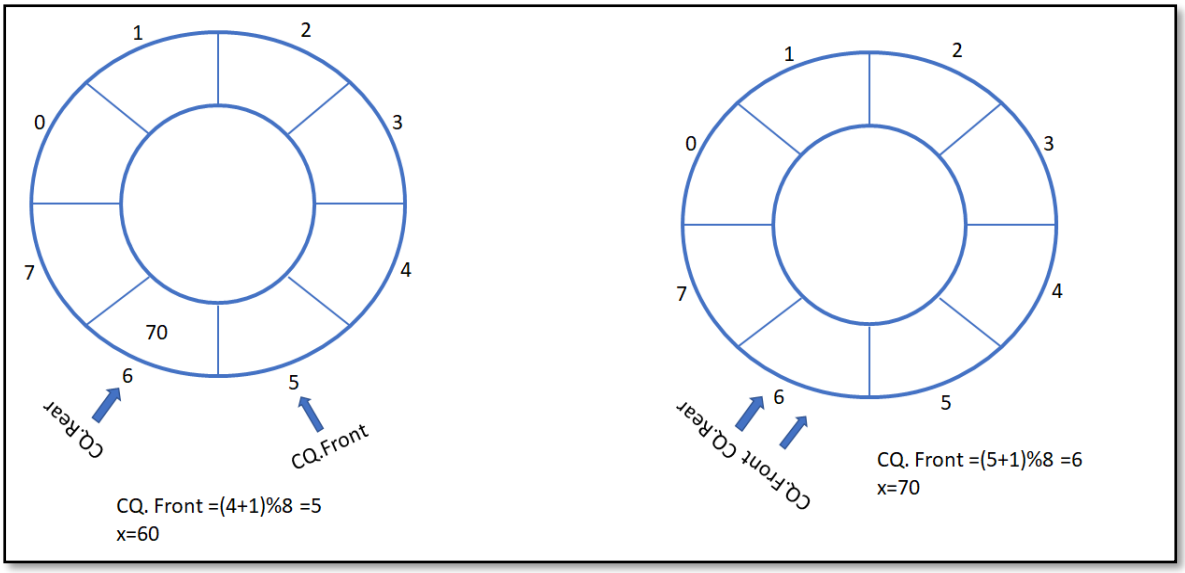
Time Complexity: Time Complexity of this Operation is Θ(1) as there are only four statements to execute.

Space Complexity: Since there is no auxiliary space used in the Algorithm, the space function is 0 (constant) i.e., Space Complexity of this Operation is Θ (1).


1
2
20
30
0
3
10
40
50
7
4
70
60
6
5
CQ.Front
CQ.Rear
If initial Queue is Full
CQ.Front
CQ.Rear
CQ.Front =(7+1)%8 = 0
x= 10


CQ.Front
1
2
30
0
3
40
50
7
4
70
60
6
5
CQ.Rear
CQ. Front =(0+1)%8 =1
x=20
CQ.Front
CQ. Front =(1+1)%8 =2
x=30


1
2
CQ.Front
0
3
50
7
4
70
60
6
5
CQ.Rear
CQ. Front =(2+1)%8 =3
x=40
CQ.Front
CQ. Front =(3+1)%8 =4
x=50



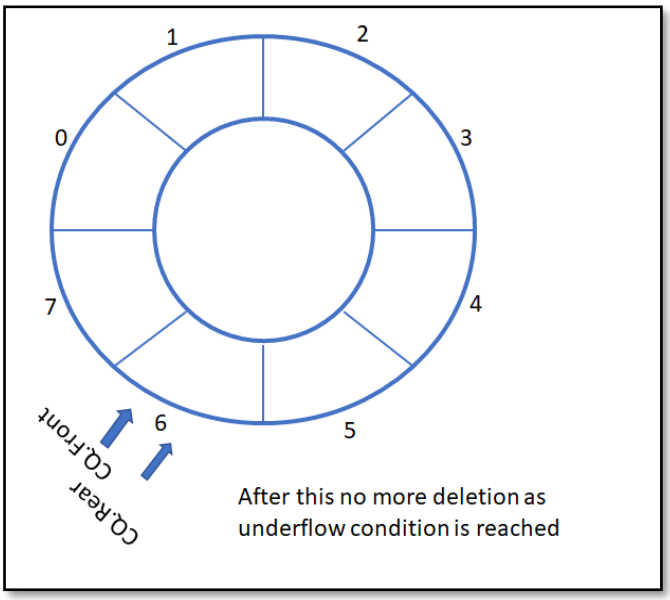


Figure : Deletion in Queue

8.8 Double Ended queue (DEQUE)

8.8.1 Introduction

Consider a situation where a limited size buffer is used to store the recent activities of software. Take an example of MS word where all typing activities are saved in the buffer which can be undone (refer to the LIFO activity in Stack chapter). If we keep on typing, then the buffer will get full at some point in time. We then need to remove some of the characters stored in the buffer to store the new ones. The removed characters will certainly be the old ones. That means we can perform the deletions at one end for the undo and at another end to remove the old character. Here we need two ends and deletion is required at both ends.

Similar to the application is the storage of internet browsing history. Some old browsing data will get removed to accommodate the new ones.

8.8.2 Definition

Double Ended queue is a special type of queue where insertion and deletion are performed through both ends from the Rear and Front end. In other words, insertion is possible from both Front and rear end and deletion is also possible from both Front and Rear end.

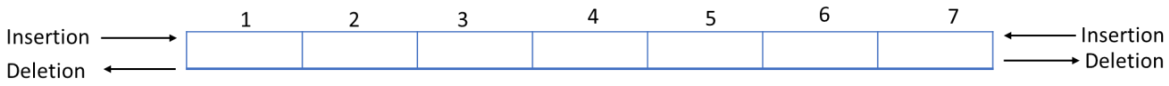


Figure: DEQUE Representation

8.8.3 Primitive Operations of Double-Ended Queue

1. Initialization

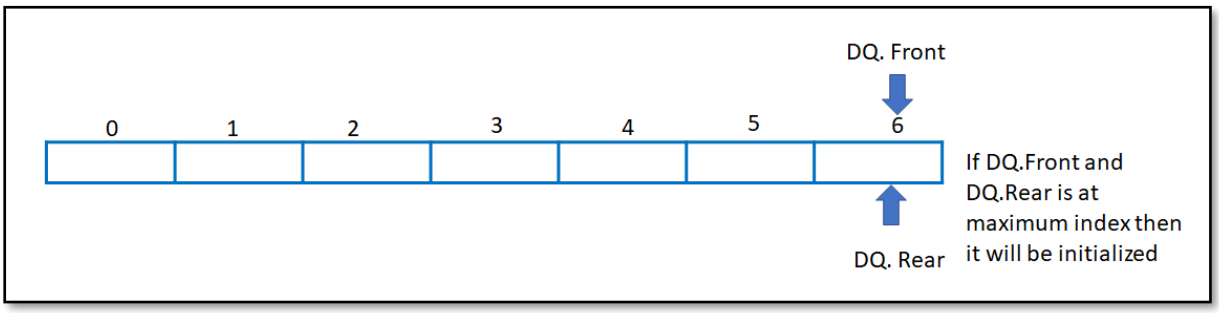
Here we initialize the Front and Rear end of Queue. Usually, Rear and Front are initialized at the largest index of the array. However, it can be initialized at any index, but both should be initialized at the same index.

ALGORITHM Initialize(DeQue DQ)
Input: Double Ended Queue DQ
Output: None
BEGIN:

```
DQ.Rear=MAXSIZE-1
DQ.Front=MAXSIZE-1
```

} Initializing Rear and Front at MAXSIZE-1

END;



Time Complexity: Time Complexity of this Operation is Θ(1) as there are only two statements to execute.
Space Complexity: Since there is no auxiliary space used in the algorithm, the space function is 0 (constant) i.e., the Space Complexity of this Operation is Θ (1).

2. Emptiness check

If there are no elements in the Queue then it is said that it is empty. If Front and Rear both are at the same index, the queue will be empty. The function given below is a Boolean valued function that returns TRUE or FALSE based on emptiness.

ALGORITHM Empty(DeQue DQ)

Input: Double Ended Queue DQ

Output: True or False based on Emptiness

BEGIN:

```
IF DQ.Front == DQ.Rear THEN
    RETURN TRUE
ELSE
    RETURN FALSE
```

DQ.Front = = DQ.Rear means that DeQue is empty.

END;

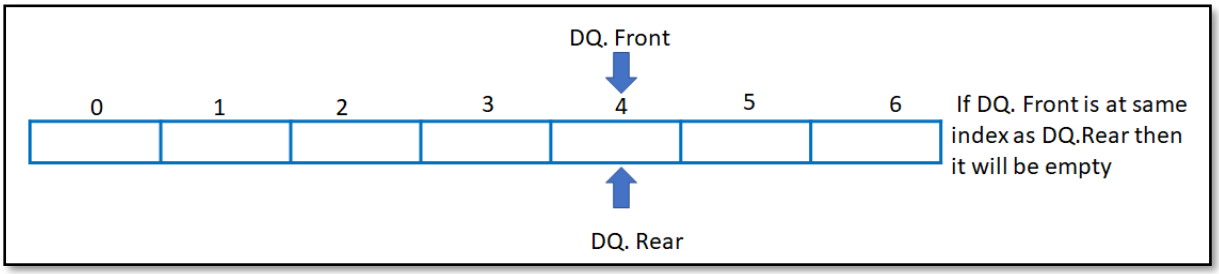


Fig: Empty DEQueue

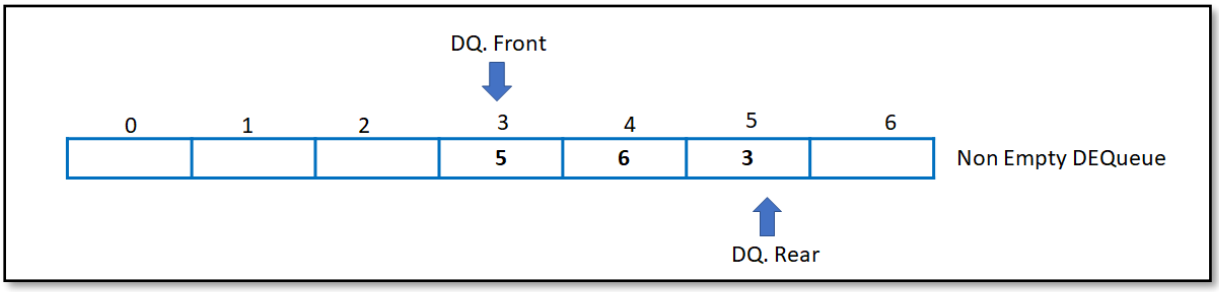


Fig: Non Empty DEQueue

3. Insertion

There are two insertions possible for the Deque. One at front and another one at the Rear end. For insertion at Rear end, Rear is incremented. If it has reached the maximum index, it is brought back to 0.

ALGORITHM EnQueueRear(DeQue DQ, x)

Input: Double Ended Queue DQ and x a new element to be inserted

Output: None

BEGIN:

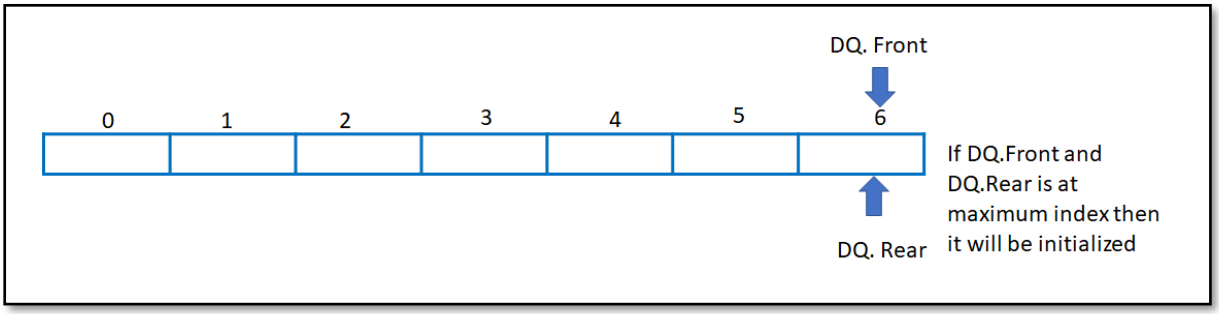
```
IF (DQ.Rear+1)%MAXSIZE == DQ.Front THEN
```

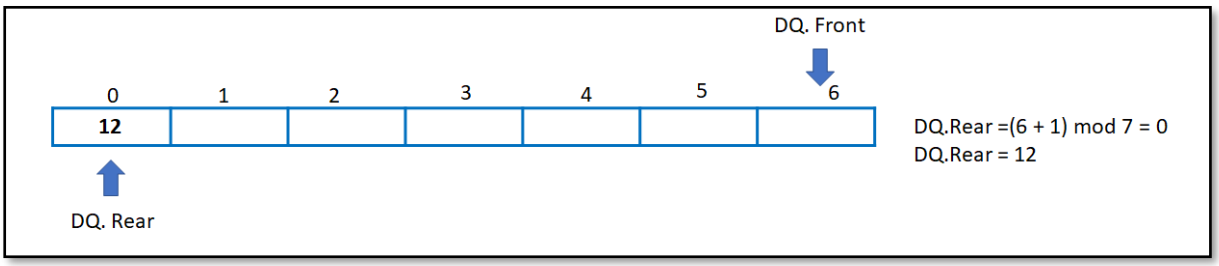
If no more insertion possible

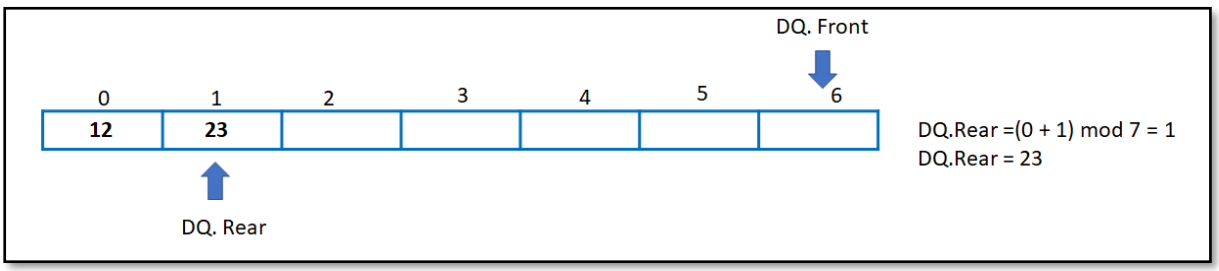
```
            WRITE ("QUEUE OVERFLOW")
            EXIT(1)
        DQ.Rear=(DQ.Rear+1)%MAXSIZE
        DQ.Buffer[DQ.Rear] = x
END;
```

Increment the rear to next index or from last index to first circularly

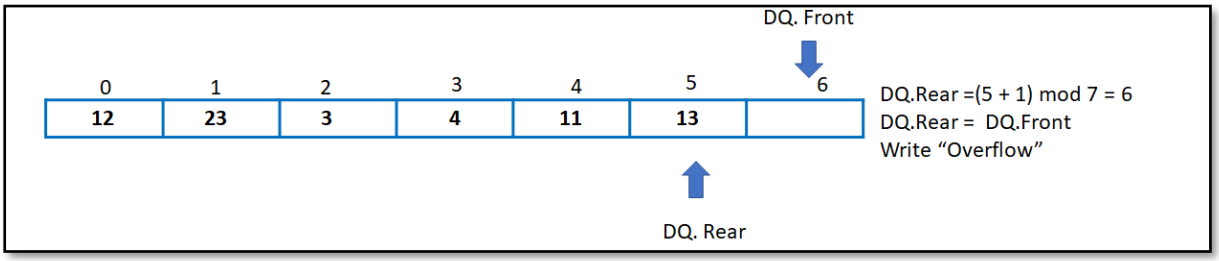
Add new element in circular queue buffer







.
.
.



Time Complexity: Time Complexity of this Operation is Θ(1) as there are only two statements

to execute even when condition is TRUE or FALSE. Here condition check is considered as first

statement and return as the second one.

Space Complexity: Since there is no auxiliary space used in the Algorithm, the space function is 0 (constant) i.e. Space Complexity of this Operation is Θ (1).

For insertion at Front end, Front is decremented. If it has reached the minimum index, it is brought to the Maximum index.

ALGORITHM EnQueueFront(DeQue DQ, x)
Input: Double Ended Queue DQ and x a new element to be inserted
Output: None
BEGIN:

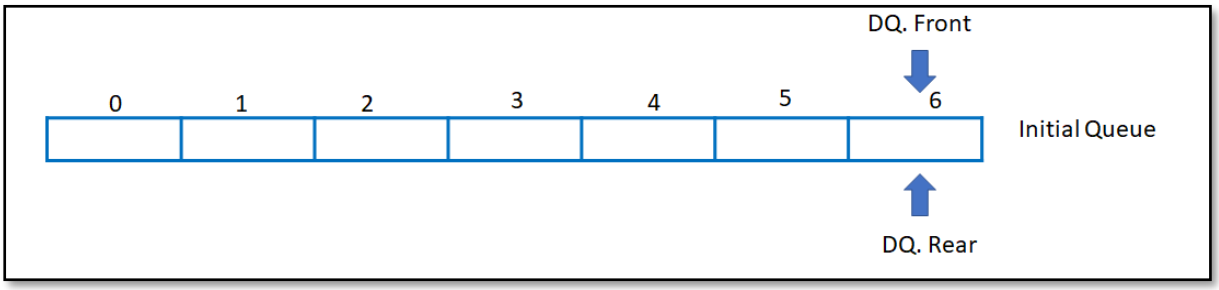
```
IF (DQ.Front – 1 + MAXSIZE)%MAXSIZE== DQ.Rear THEN
        WRITE (“QUEUE OVERFLOW”)
        EXIT(1)
DQ.Front=(DQ.Front – 1)%MAXSIZE

DQ.Buffer[DQ.Front] = x
```

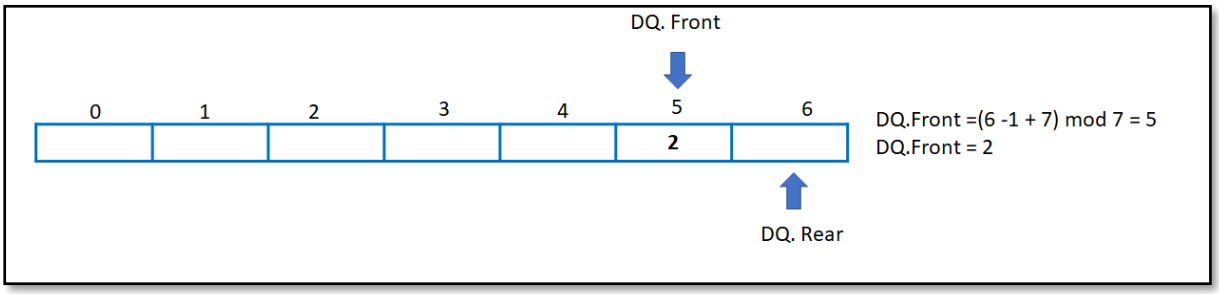
END;

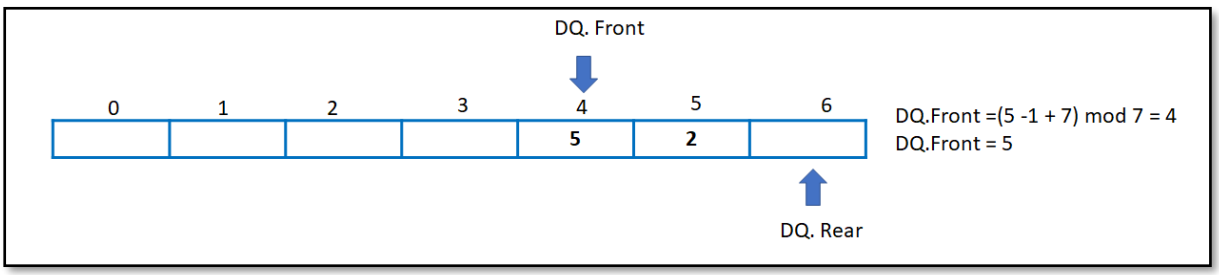
If no more insertion possible

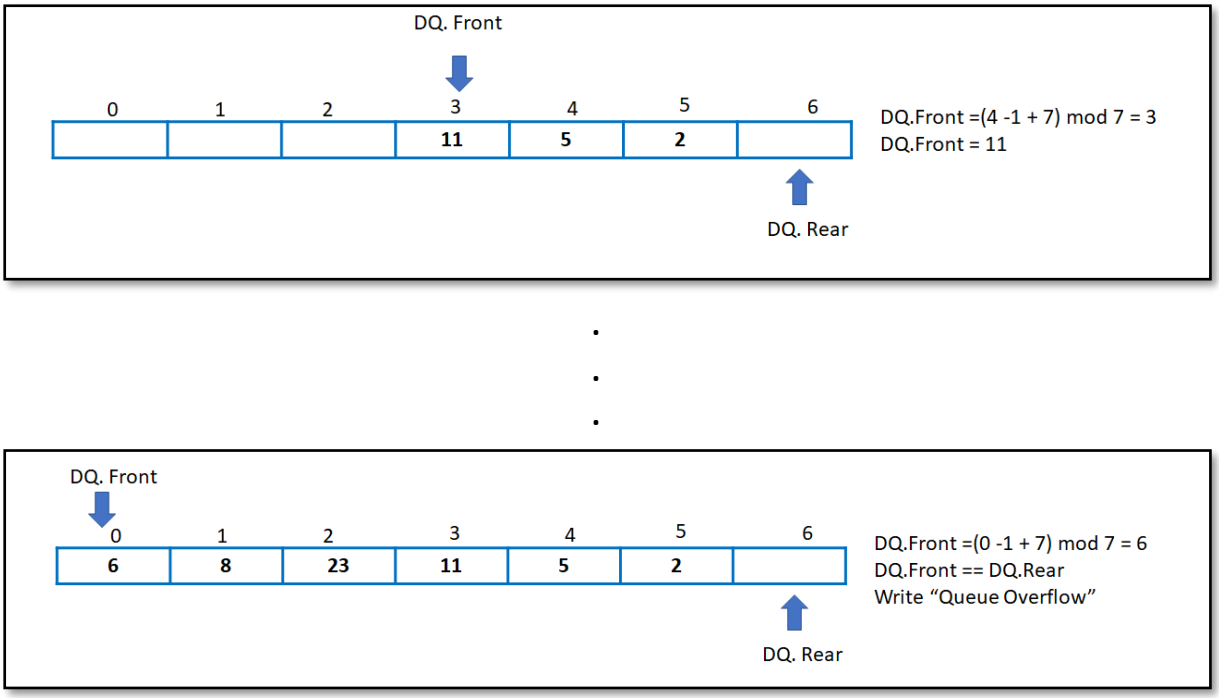
Increment the rear to next index or from last index to first circularly

Add new element in circular queue buffer









Time Complexity: Time Complexity of this Operation is Θ(1) as there are only two statements to execute even when the condition is TRUE or FALSE. Here condition check is considered as the first statement and return as the second one.

Space Complexity: Since there is no auxiliary space used in the Algorithm, the space function is 0 (constant) i.e. Space Complexity of this Operation is Θ (1).

4. Deletion

There are two Deletions possible for the Deque. One at front and another one at the Rear end.

For Deletion at Front end, Front is incremented. If it has reached to the maximum index, it is brought back to 0.

ALGORITHM DeQueueFront(DeQue DQ)

Input: Double Ended Queue DQ

Output: Deleted item from front index of DeQueue

BEGIN:

```
IF Empty(DQ) THEN
        WRITE("QUEUE UNDERFLOW")
        EXIT(1)

DQ.Front=(CQ.Front+1)%MAXSIZE

x=DQ.Buffer[CQ.Front]
RETURN x
```

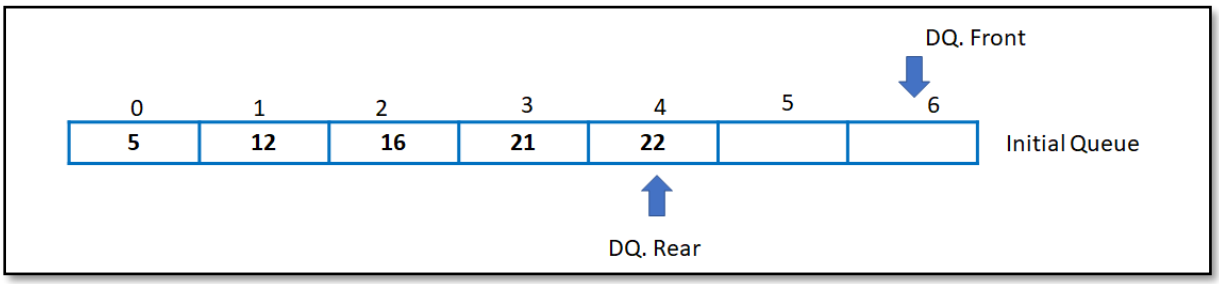
If no more deletion possible

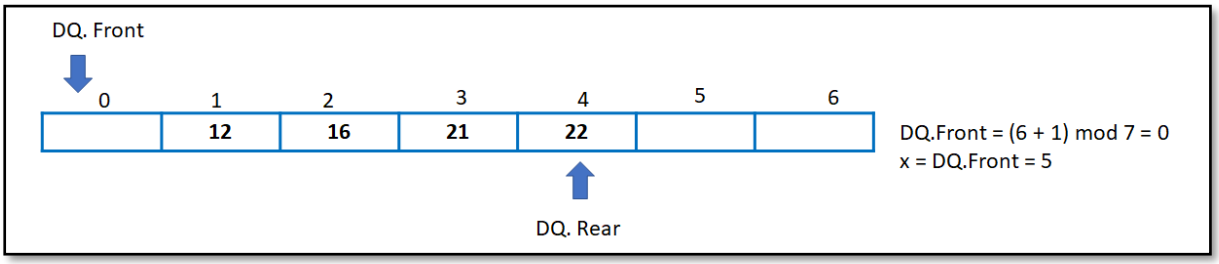
Increment the front index to next index or from the last index to first

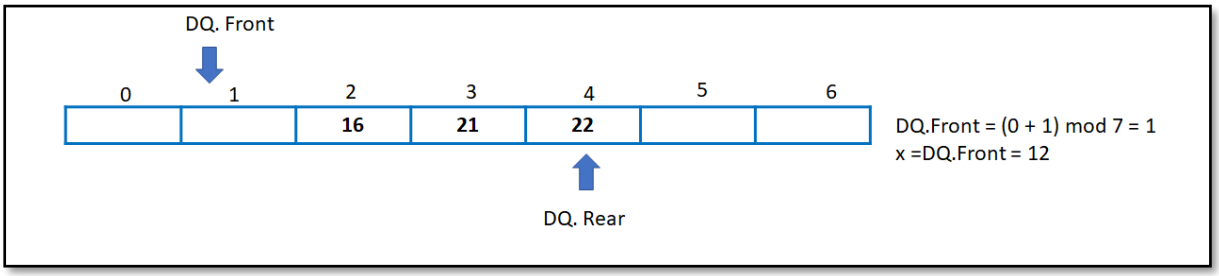
Saving the front index element in x
Returning the deleted element

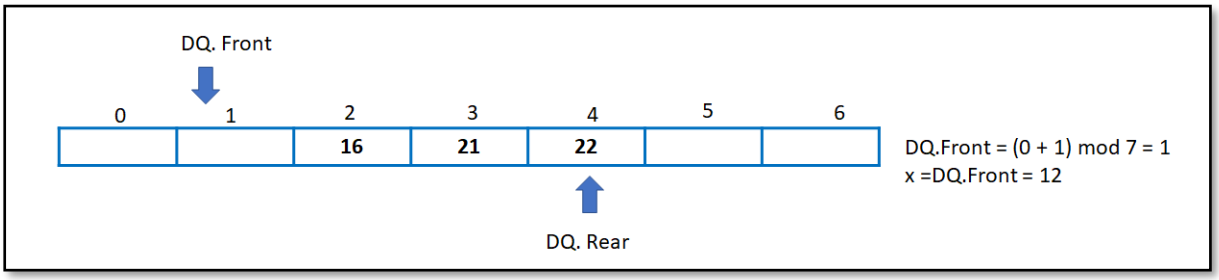
END;

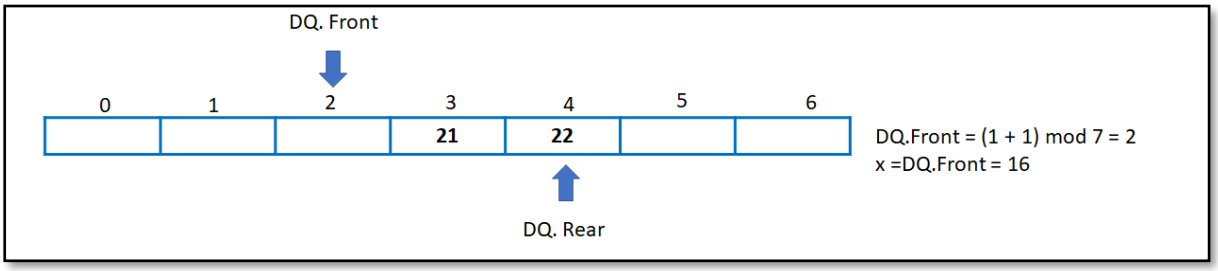
For Deletion at Rear end, Rear is decremented. If it has reached the minimum index, it is brought back to the Maximum index.

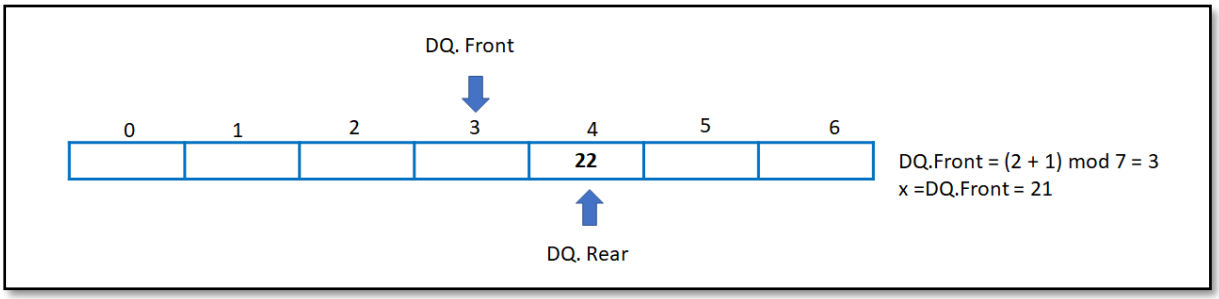


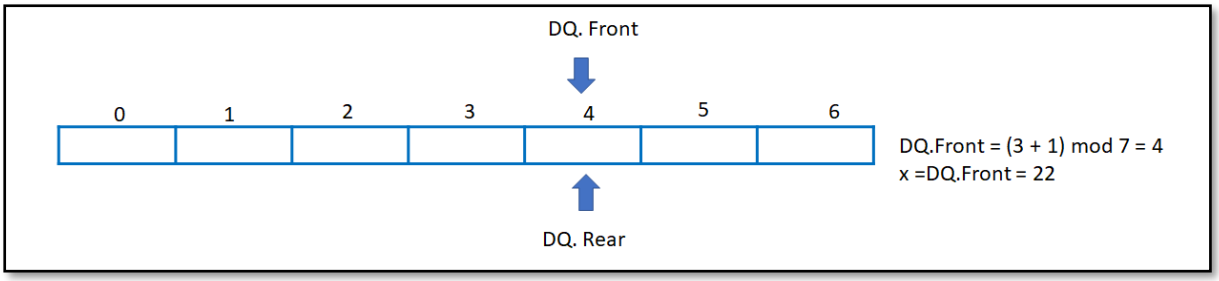


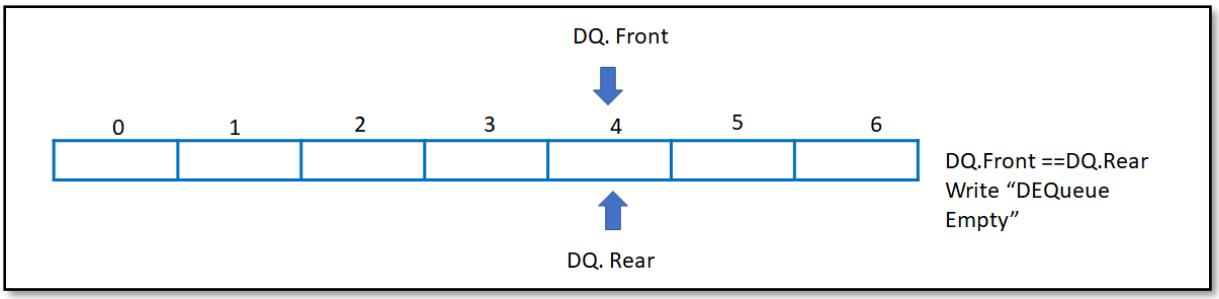












Time Complexity: Time Complexity of this Operation is Θ(1) as there are only two statements to execute even when the condition is TRUE or FALSE. Here condition check is considered as the first statement and return as the second one.

Space Complexity: Since there is no auxiliary space used in the Algorithm, the space function is 0 (constant) i.e. Space Complexity of this Operation is Θ (1).

ALGORITHM DeQueueRear(DeQueue CQ)

Input: Double Ended Queue DQ

Output: Deleted item from Rear index of DeQueue

BEGIN:

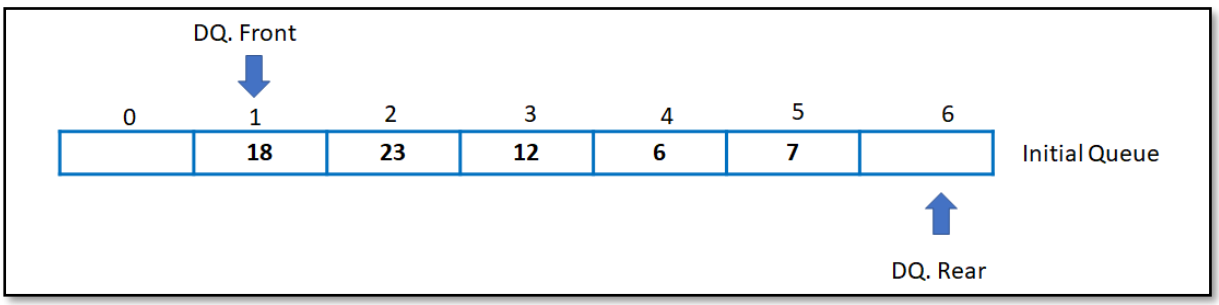
```
        IF Empty(DQ) THEN
                WRITE("QUEUE UNDERFLOW")
                EXIT(1)
        DQ.Rear=(DQ.Rear -1 + MAXSIZE)%MAXSIZE

        x=DQ.Buffer[DQ.Rear]
        RETURN x
```

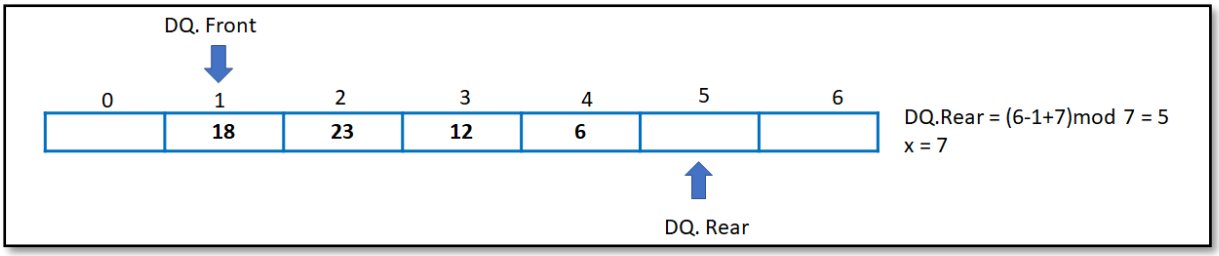
If no more deletion possible

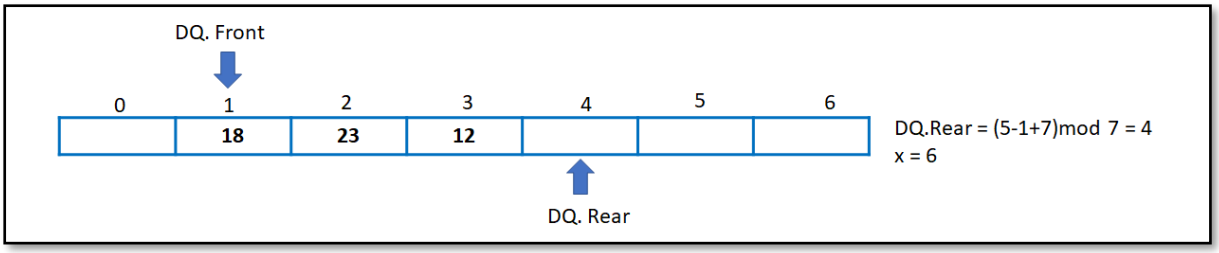
Increment the front index to next index or from the last index to first circularly

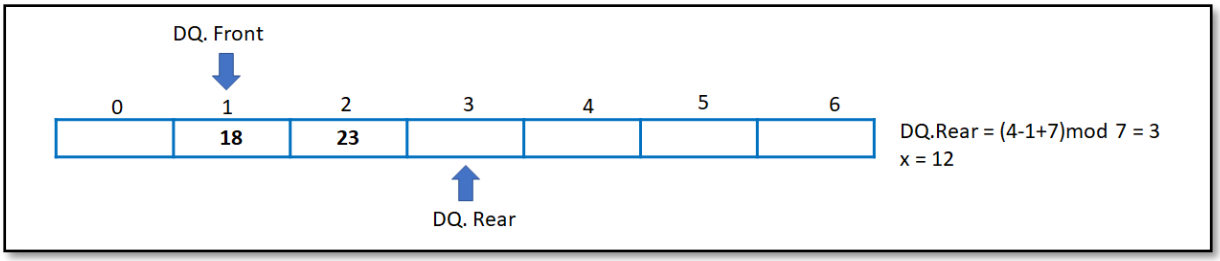
Saving the front index element in x
Returning the deleted element

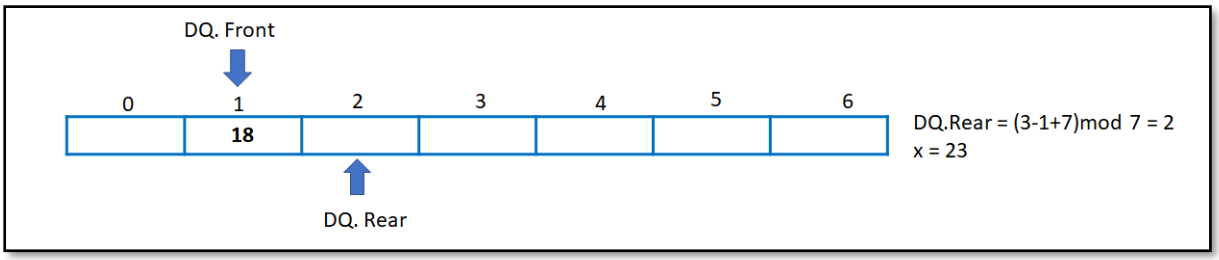
END;

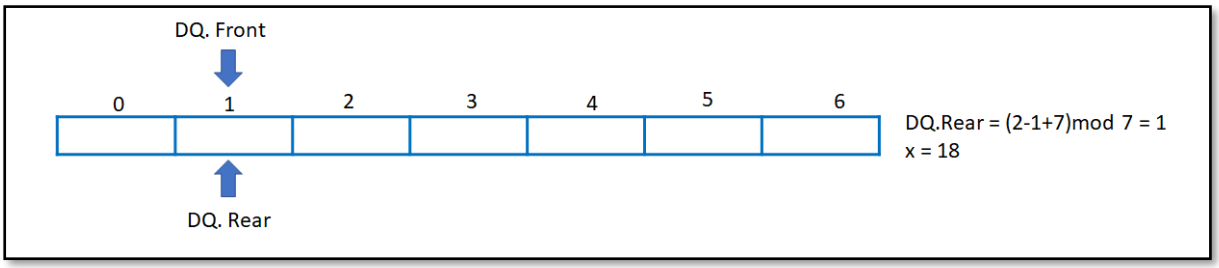


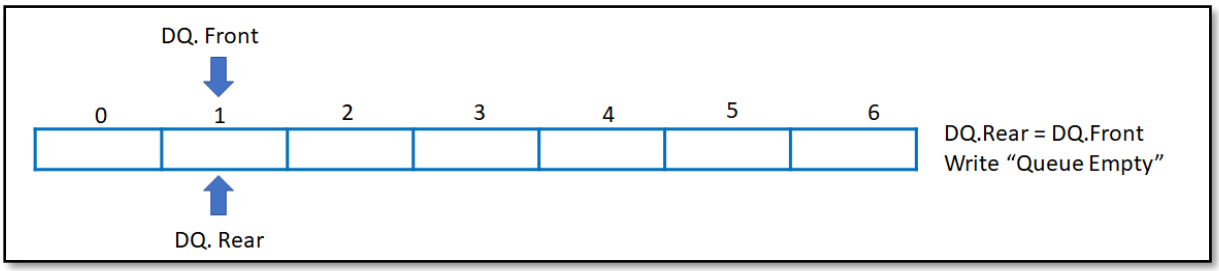












Time Complexity: Time Complexity of this Operation is Θ(1) as there are only two statements to execute even when the condition is TRUE or FALSE. Here condition check is considered as the first statement and return as the second one.

Space Complexity: Since there is no auxiliary space used in the Algorithm, the space function is 0 (constant) i.e. Space Complexity of this Operation is Θ (1).

8.9 Priority Queue

8.9.1 Introduction

Consider a situation where students are assembled in a queue to deposit their fees. Meanwhile, a faculty member arrives to take some service from the same window. Students may probably give respect to the faculty and will allow him to take the service on priority.

Consider another situation in which an Operating system is required to execute the tasks on the basis of priority. The processes of different priorities will keep coming into the system that needs to be executed. Higher priority process should be available readily for execution and incoming processes should be arranged according to the priority.

Priority Queues are the data structure in which the intrinsic ordering of the elements does determine the result of the basic operations.

There are two types of Priority Queue in general: Ascending, Descending.
Ascending PQ: Lower numbers are considered higher priority e.g. 1 means highest priority. Upon subsequent deletions, the ascending sequence of priority numbers will appear.

Descending PQ: Higher numbers are considered a higher priority. Upon subsequent deletions, the descending sequence of priority numbers will appear.

Priority Queue is an extension of queue with the following inherent properties:

- Every element has a priority associated with it.
- The higher priority element will be served (deleted) first followed by the lower priority element.
- Equal Priority Elements are treated in first come first serve manner.

Primitive Operations expected on Priority Queue:

- Insertion
- Deletion of Min or Max Element (Depending on the priority)

8.9.2 Applications of Priority Queue

Some of the applications of the Priority Queue is listed as under

- Printing of papers in order of length
- Routing of packets in the order of urgency

- Scheduling of jobs at CPU based on priority
- Sorting of numbers, picking Minimum or Maximum element first
- Text Compression using Huffman Coding, Selection of frequent symbols for compression
- Prim's Algorithm for Minimal spanning Tree
- Dijkstra's Algorithm for Single Source Shortest Path, etc.

8.9.3 Implementation of Priority Queue

Priority Queue can be implemented in using the following data structures

Using Unsorted list (Array)
Using Unsorted list (Linked-List)
Using Sorted list (Array)
Sorted list (Linked-List)
Binary Heap
Binominal Heap
Fibonacci Heap

Each of the implementation method has certain merits and demerits. These are discussed in detail in the subsequent sections.

8.9.4 Implementation through Sorted Array (Ascending)

1. Insertion (EnQueue)

Insertion operation performed using this method requires searching the desired location for insertion followed by shifting elements towards the right. If there are N elements in the Array initially, and i is the location of the insertion, (N-i) shifting will take place. The total effort required in this case will be i+(N-i) = N.

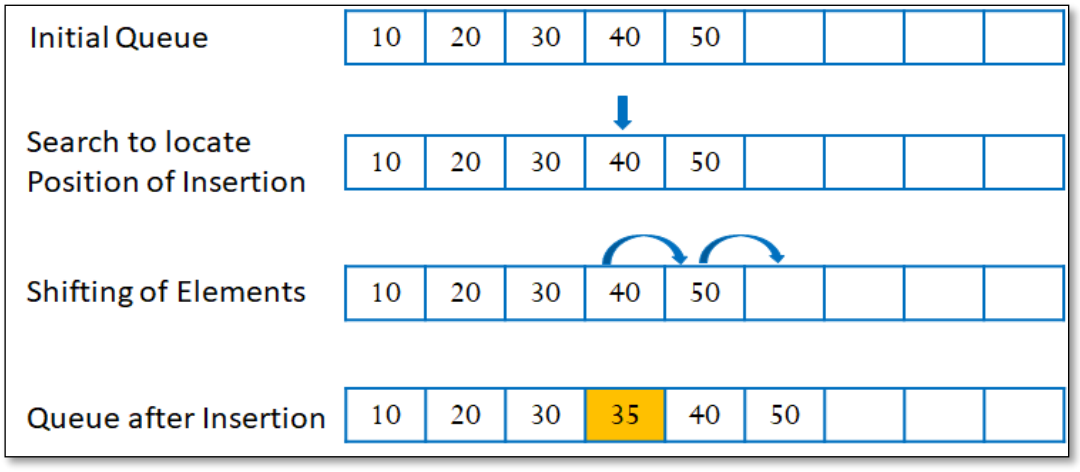


Figure: Showing the process of EnQueue(35)

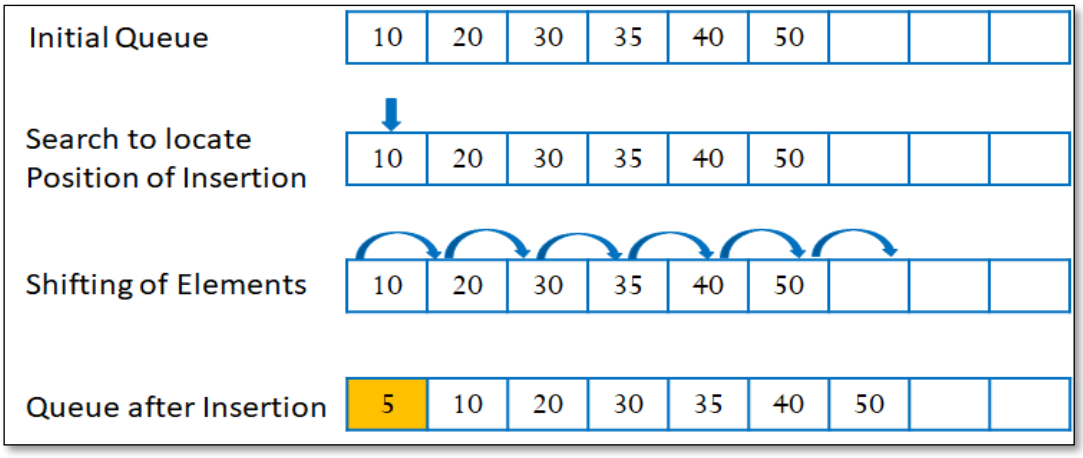


Figure: Showing the process of EnQueue(5)

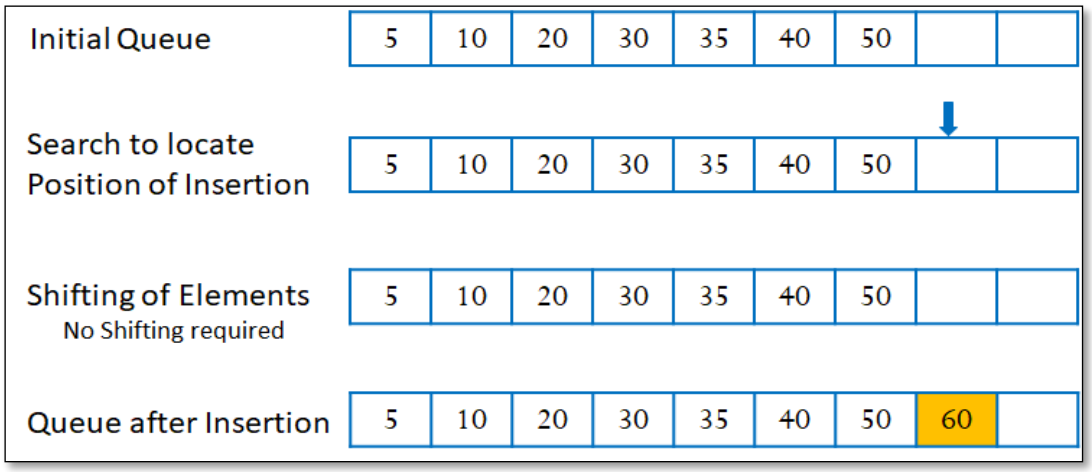


Figure: Showing the process of EnQueue(60)

ALGORITHM EnQueue(A[ ], N, key)

BEGIN:

```
i =0
WHILE A[i]<key DO
        i = i+1
```

The search operation to find the insertion location of the key to be inserted. Here Key is compared with the elements present in the sorted array and location is returned as the

```
FOR j =N-1 TO i STEP–1 DO
        A[j+1] = A[j]
```

Elements are shifting to make the location 'i' vacant to insert the key

```
A[i] = key
```

Insertion of key at i location

```
N=N+1
```

Incrementing the array size by 1

END;

END;

Time Complexity: Θ(N)

If the desired place for the insertion is 'i' (found with the search) then 'n–i' shifting will be required. Search operations require i statement executions and shifting requires n-i statement executions. Apart from searching and shifting, three other statements will get executed. Total statements execution is N+3 i.e. **Θ(N).**

Space Complexity: Θ (1)

Only two variables (i and j) are taken here; hence the space complexity is constant, i.e. Θ(1).

2. Deletion (DeQueue)

Deletion requires the deletion of minimum element (that represents the highest priority). A total of N- 1 shifting will be required in such a case. Total effort required will be 1 (for deletion) + (N-1) for shifting.

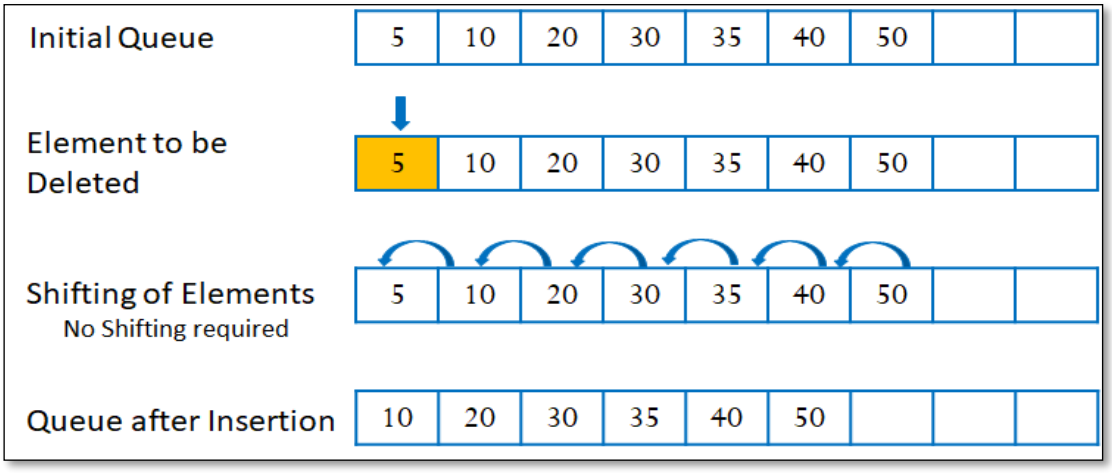


Figure: Showing the process of DeQueue()

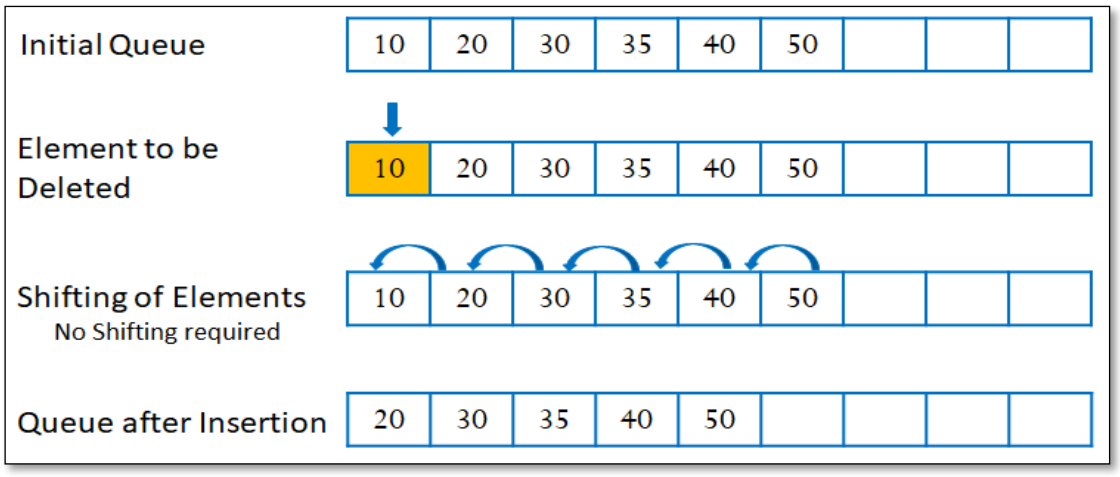


Figure: Showing the process of DeQueue()

ALGORITHM DeQueue(A[ ],N)

BEGIN:

x = A[i] } Saving the element to be deleted

```
        FOR j = i+1 TO N-1 DO
                A[j–1] =A[j]
```
Shifting of elements from $(i+1)^{th}$ index to N^{th} index. An element is shifted at one lower index to the current index.

```
        N = N–1
```
decrementing the array size by 1.

```
        RETURN x
```
Returning the deleted element

```
END;
```

Time Complexity: In the above algorithm, there are 3 statements outside the loop that are compulsory to be executed in any case. When the element is to be deleted from the beginning, N-1 shifting will be required. Total statements for execution are N+2; hence the complexity is **O(N)** in the worst case.
When the element is to be deleted from the end, no shifting is required. There will be 3 statements for execution in total, which is constant. This is the best case. Hence the Time complexity in this case is Ω(1).

Space Complexity: Θ(1)
The only variables taken here are x and j; hence the space complexity is constant, i.e. Θ(1).

8.9.5 Implementation through Sorted Array (Descending)

1. Insertion (EnQueue)
Insertion operation performed using this method requires searching the desired location for insertion followed by shifting elements towards the right. If there are n elements in the Array initially, and i is the location of the insertion, (n-i) shifting will take place. The total effort required in this case will be i+(n-i) = n.

```
ALGORITHM EnQueue(A[ ], N, key)
BEGIN:
        i=0
        WHILE i<N AND key<A[i] DO
                i++
```
Searching for appropriate position of insertion

```
        FOR j = N-1 TO i STEP–1 DO
                A[j+1] = A[j]
```
Shifting of elements from $(N-1)^{th}$ index to i^{th} index. An element is shifted at one higher index to the current index.

```
        A[i] =Key
```
Placing Key at i^{th} index.

```
        N = N+1
```
Incrementing the array size by 1.

```
END;
```

Time Complexity: Θ(N)

If the desired place for the insertion is 'i' (found with the search) then 'n–i' shifting will be required. Search operations require i statement executions and shifting requires n-i statement executions. Apart from searching and shifting, three other statements will get executed. Total statements execution is N+3 i.e. **Θ(N).**

Space Complexity: Θ (1)

Only two variables (i and j) are taken here; hence the space complexity is constant, i.e. Θ(1).

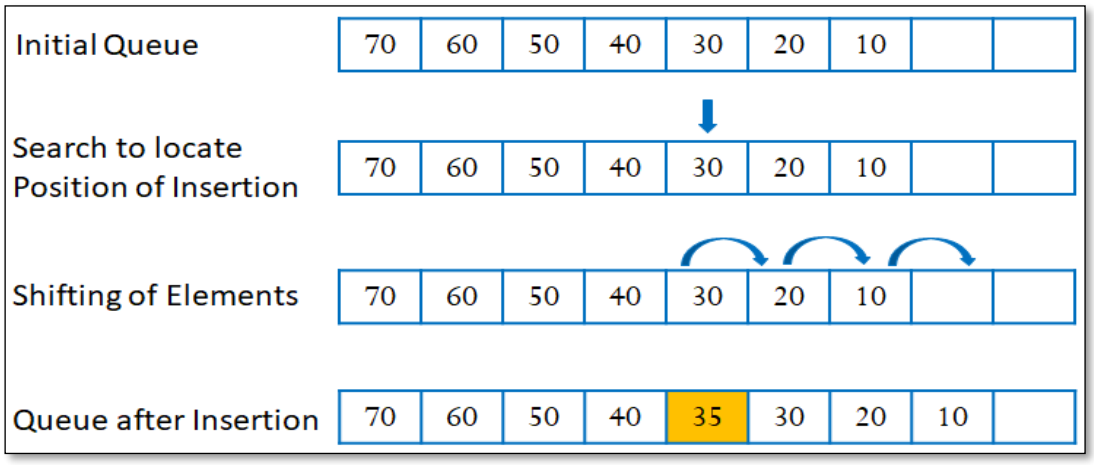


Figure: Showing the process of EnQueue(35)

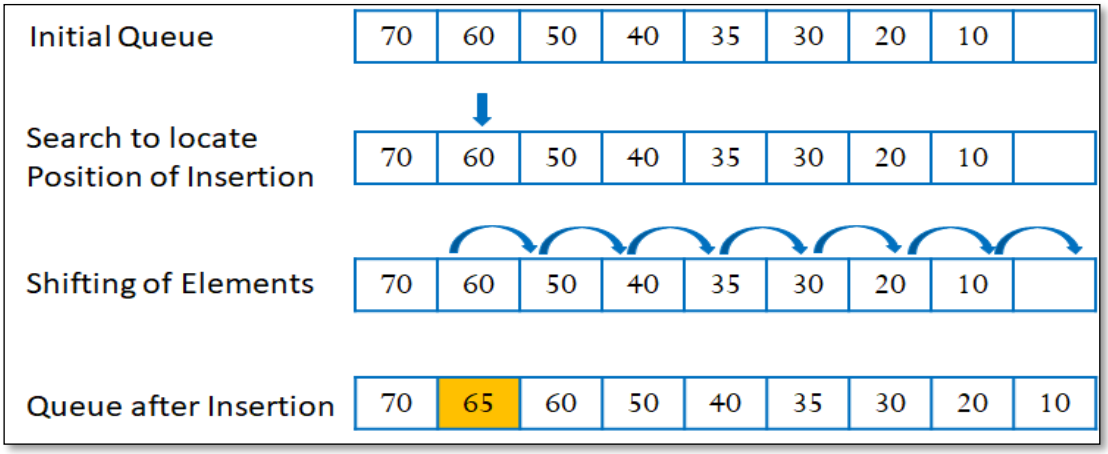


Figure: Showing the process of EnQueue(65)

2. Deletion (DeQueue)

Deletion will remove the last element (highest) priority. This will keep the deletion effort constant.

ALGORITHM DeQueue(A[ ],N)

BEGIN:

X= A[N-1]

N=N-1

RETURN X

END;

Time Complexity: Θ(1) as the last element will be deleted and returned. Total 3 statements to execute

Space Complexity: Θ(1) as only one variable X is used here

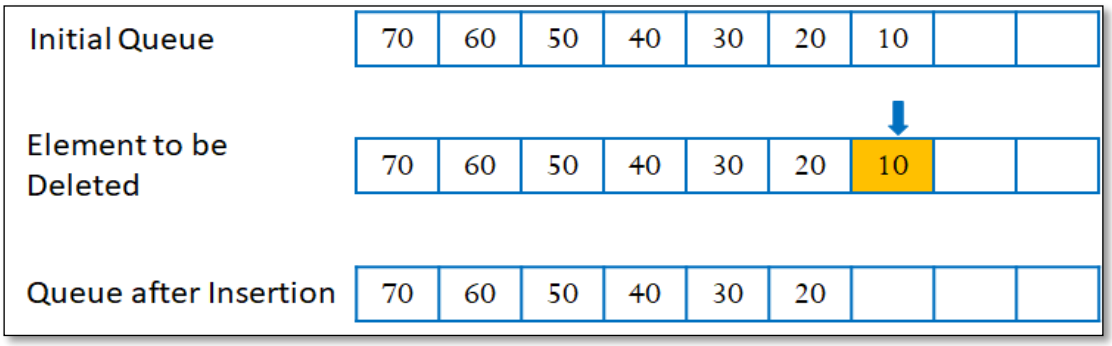


Figure: Showing the process of DeQueue()

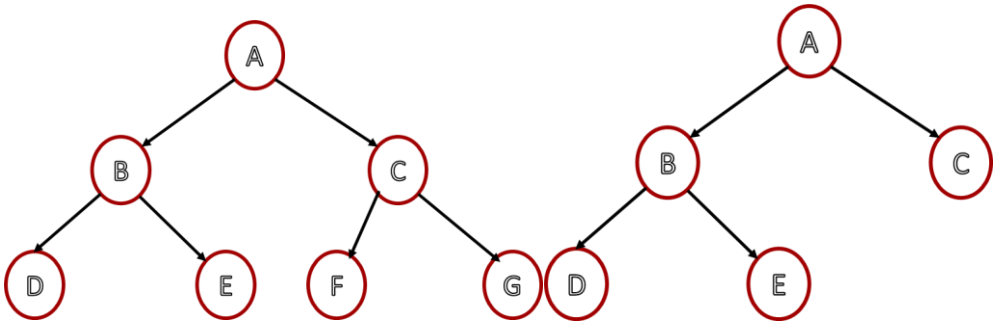
8.9.7 Using Binary Heap

Method of ordered Array insertion requires O(N) for either of the Enqueue and DeQueue operations. If implemented through Binary heap, the complexity will reduce to $O(Log_2N)$.

Consider the thrasher for the thrashing of barley in the farm field. The thrasher throws the wheat that automatically takes the shape of a pyramid. Let us say we are packing the wheat in the sacks by taking some wheat from the heap; the heap regains the structure automatically.

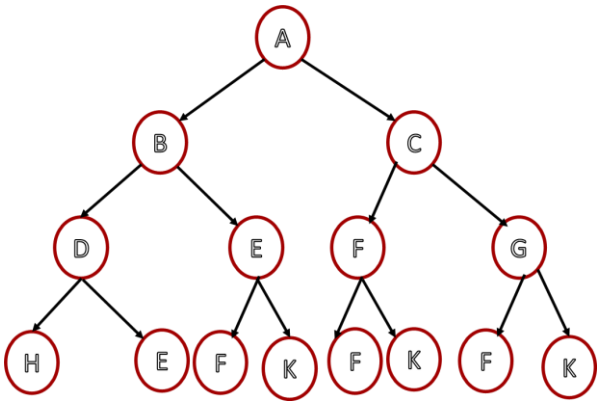
Complete Binary Tree

It is a type of binary tree in which all levels are completely filled except possibly the last level.
If the last level is not full, then all the nodes should be filled from the left.



Note:- In the above diagram, nodes fill from left to right and level via level.

Derive an expression to define the relationship between height and the number of nodes in a binary tree.



Height at level	Number of nodes at Level h
h = 0	$2^0 = 1$
h = 1	$2^1 = 2$
h = 2	$2^2 = 4$

Therefore, total number of nodes(N) in a binary tree of height h

$= 2^0 + 2^1 + 2^2 + \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots + 2^h$

As this expression forms a G.P.

$= (2^{(h+1)} - 1)/ (2-1)$

$N = 2^{(h+1)} - 1$

$N + 1 = 2^{(h+1)}$

Taking log on both side, we get

$\log_2(N + 1) = h+1$

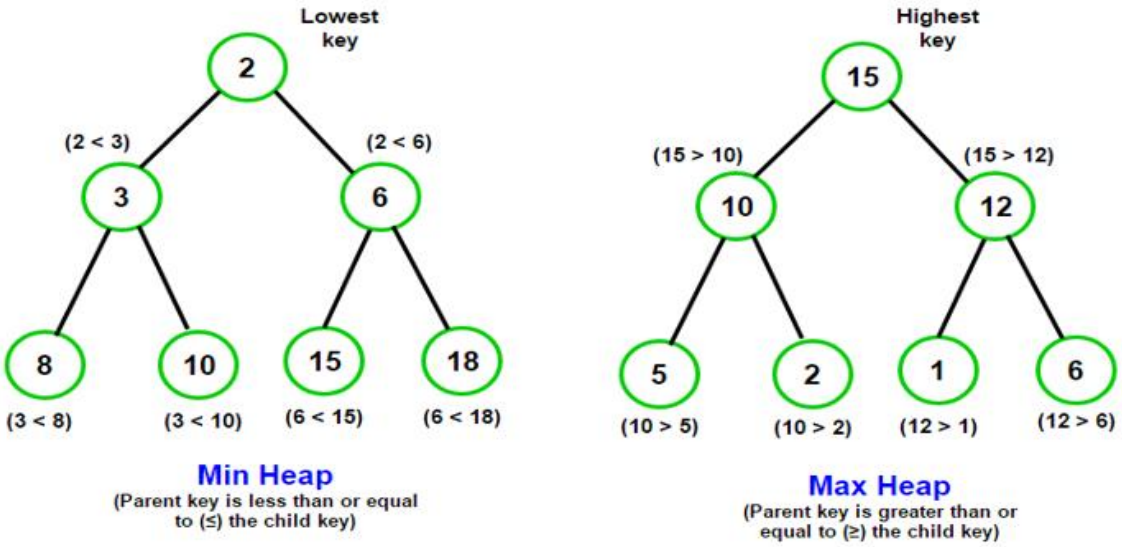
$$\mathbf{h = \log_2(N + 1)-1}$$

This can be represented as $O(\log_2 N)$

Heap

A Binary Heap is a Complete Binary Tree which makes it suitable to be implemented using array.
A Binary Heap is categorized into either Min-Heap or Max-Heap.
In a Min Binary Heap, the value at the root node must be the smallest among all the values present in Binary Heap. Furthermore, this property of Min-Heap must be true repeatedly for all subtrees.
In a Max Binary Heap, the value at the root node must be the largest among all the Binary Heap values. This property of Max-Heap must be true repeatedly for all subtrees.

As heap is a complete binary tree, the height of tree having N nodes will always O(log N).



Implicit Binary Tree Representation

For the Binary heap implemented using Array, we need to take advantage of implicit Binary Tree implementation.

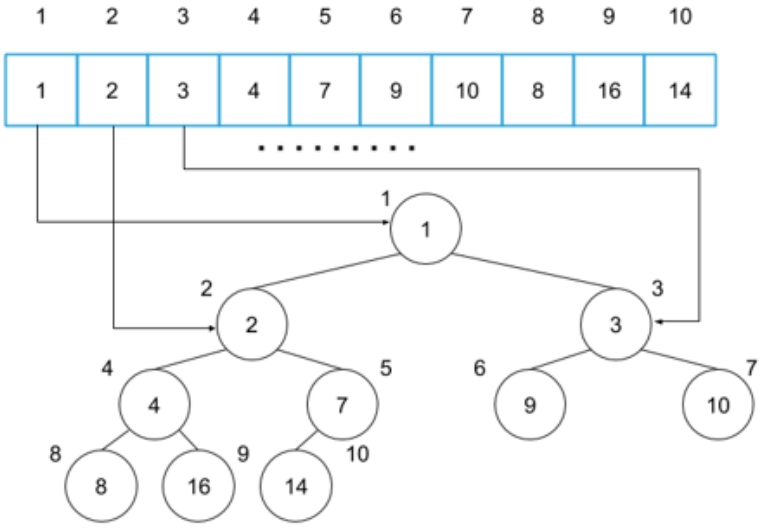


Figure e: Implicit Binary Tree Representation

In this case, parent-child relationships can be shown below: (It is assumed that array indexes are starting at 1).

A root node | i = 1, the first item of the array
A left child node | left(i) = 2*i
A right child node | right(i)=2*i+1
A parent node | parent(i) = i / 2

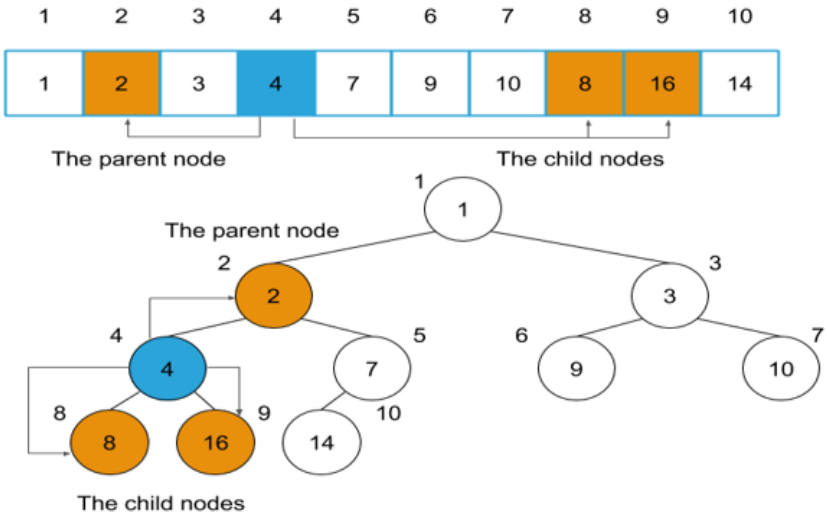


Figure f: Showing DeQueue() Operation

1. Insertion

Insertion operation inserts the key (the data to be inserted) at the N+1 index. Since heap property will be disturbed by this, we will have to move up in the tree to adjust to regain the heap property.

Following are the steps to insert an element in Max Heap.

- First, update the size of the tree by adding 1 in the given size.
- Then insert the new element at the end of the tree.
- Now perform the Heapify operation to place the new element at its correct position in the tree and make the tree either as max heap or min heap.

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

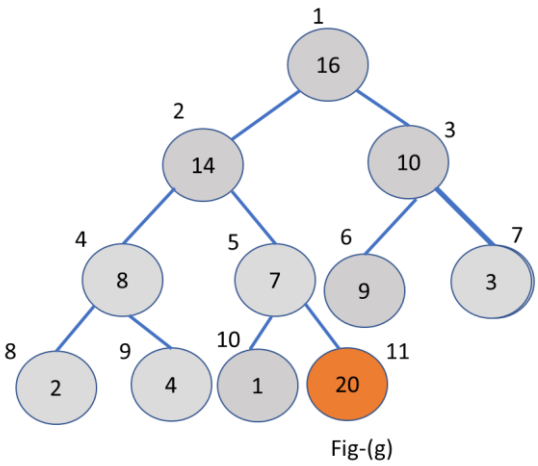


Fig-(g)

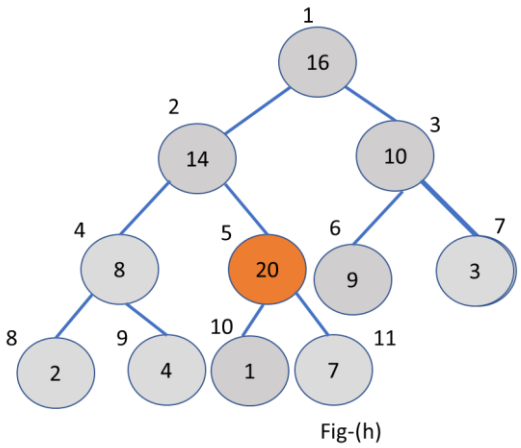


Fig-(h)

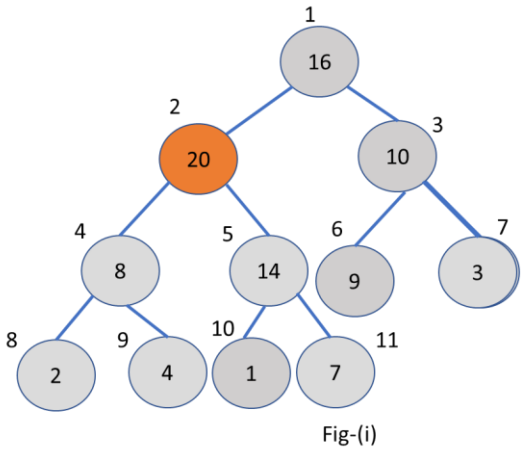


Fig-(i)

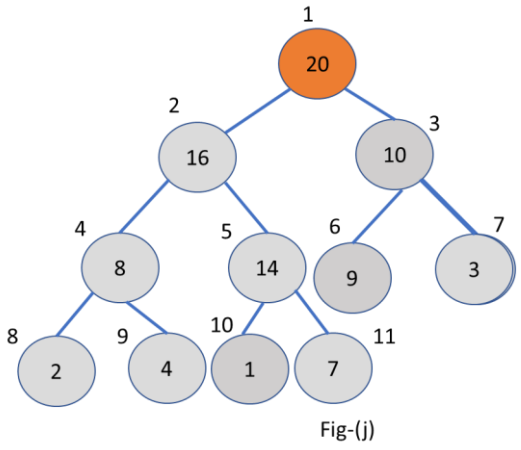


Fig-(j)

1	2	3	4	5	6	7	8	9	10	11
20	16	10	8	14	9	3	2	4	1	7

Figure g, h, i, j showing the insertion of item 20 and corresponding action of Reheapify up

This is the array representation of the given max-heap.

ALGORITHM InsertNode(A[ ], N, item)

Input: An array containing N elements that follow the Heap Property and an item to be inserted

Output: None

BEGIN:

```
N = N + 1;
A[ N ] = item;
ReHeapifyUp( A[ ], N)
```

Updating the heap size to N+1
Insertion of the item at last index
Call of function to fix the inserted element to follow the Heap property

END;

ALGORITHM ReHeapifyUp(A[ ], N)

Input: An array containing N elements. First N-1 elements follow the Heap Property.

Output: An array containing N elements that follow the Heap Property

BEGIN:

```
i = N
WHILE i>0 AND A[i] > A[i/2] DO
        Exchange(A[i], A[i/2])
        i = i/2
```

Exchanging the parent and child information to regain Heap order property

END;

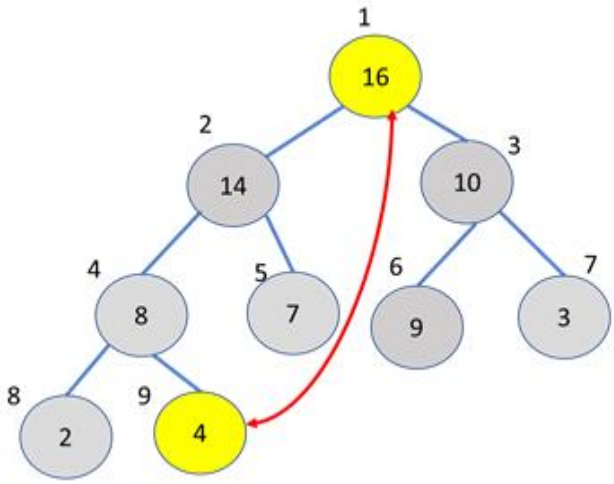
Time Complexity: As the process requires the movement from leaf to root while doing the comparision and exchenges, the complexity will be equal to height of the tree i.e. $O(\log_2 n)$
Space Complexity: A variable i is the only required item here, the space complexity is O(1)

Deletion

To delete root element from Max Heap.

Following are the steps to delete an element from Max Heap.

a) First, exchange the root element with the last element in a heap.
b) Then remove the last element by decreasing the size of the tree by 1.
c) Now perform the Heapify operation to make the tree either as max heap or min-heap.



1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4

Figure k: Deletion of the element from Binary Heap. Root Node selected for deletion

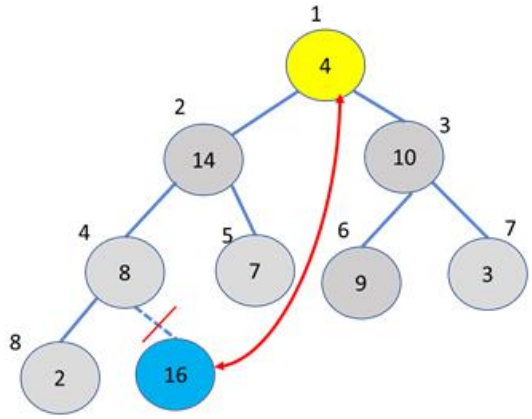


Figure l

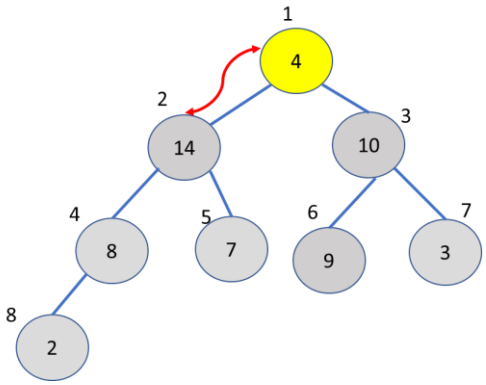


Figure m

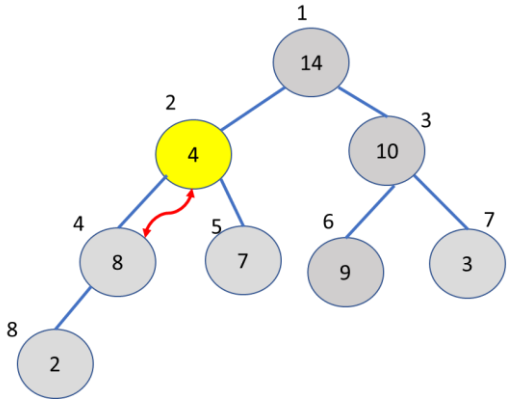


Figure n

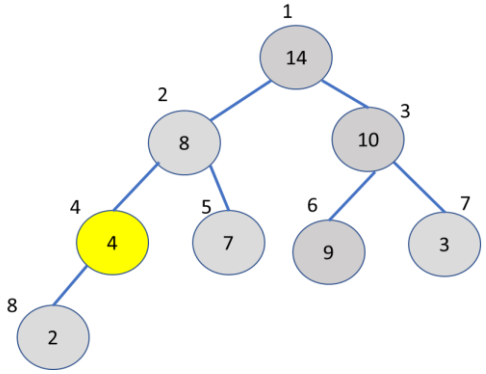


Figure o

Figure l,m,n,o: showing the Deletion followed by MaxHeapify to regain Heap Shape

ALGORITHM DeQueue(A[ ], N)
Input: An array containing N elements that follows the Heap Property
Output: Deleted element
BEGIN:

```
x = A[1]
lastitem = A[N]
A[1] = lastitem
```

Saving the first element
Putting last element at the first index

```
N = N - 1
MaxHeapify(A,1,N)
RETURN x
```

Reducing the Heap size by 1
Readjusting the Heap to regain the Heap property
Returning the saved root element

END;

Time Complexity: As the process requires the movement from root to leaf while doing the comparison and exchanges, the complexity will be equal to the height of the tree i.e. $O(\log_2 n)$
Space Complexity: A variable i is the only required item here; the space complexity is O(1)

Symbol	Insertion	Deletion
Unsorted Array	O(N)	O(N)
Sorted Array (Ascending)	O(N)	O(N)
Sorted Array (Descending)	O(N)	O(1)
Sorted Linked List	O(N)	O(1)
Binary Heap	$O(Log_2N)$	$O(Log_2N)$

8.10 Implementation of Queue using Stack

Queue using Stack is a method of implementing the process of First in First Out by the use of 2 Stacks.

In the first Stack initially, the elements will be stored in the Last in Last Out pattern. Now, the elements of Stack 1 will be copied to Stack 2, which will again reverse the element order.

Ultimately, when the element popped out from Stack 2, they will be displayed in the same order as inserted.

ALGORITHM Initialize(S1,S2)
Input: Two Stacks named S1 and S2
Output: None
BEGIN:

S1.Top1=-1 } Initialize Stack S1

S2.Top2=-1 } Initialize Stack S2

END;

ALGORITHM IsEmpty(S1, S2)
Input: Two Stacks named S1 and S2
Output: TRUE or FALSE
BEGIN:

```
IF S2.Top2==-1 AND S1.Top1==-1 THEN
    RETURN TRUE
ELSE
    RETURN FALSE
```

END;

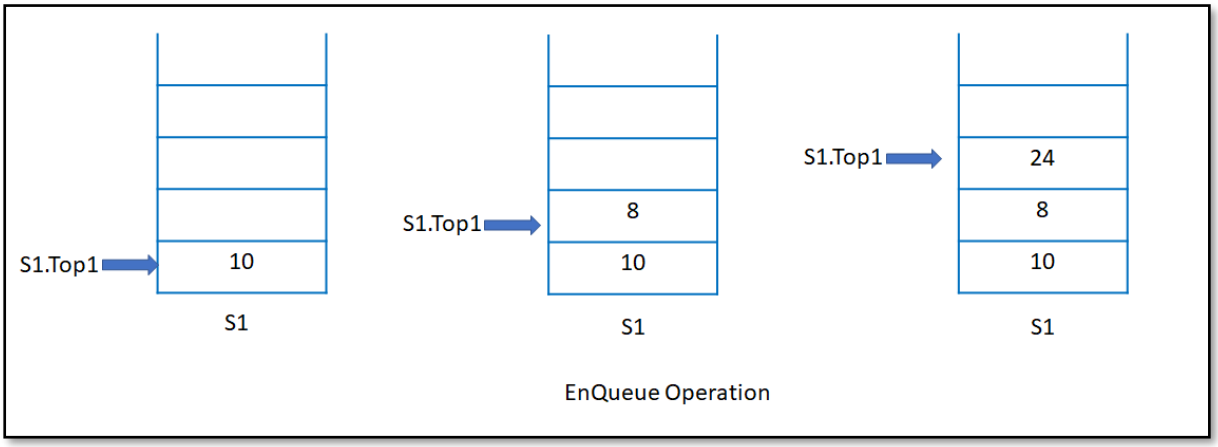
ALGORITHM EnQueue(x, S1)
Input: Stack S1 , data element for insertion x
Output: Updated Stack S1 after insertion

BEGIN:

PUSH(S1,x)

END;

Initially insert the elements in first Stack



Time Complexity: Since push operation requires Θ(1) time. So, the time complexity of EnQueue operation will be Θ(1).

Space Complexity: Since there is no auxiliary space used in the Algorithm, the space function is 0 (constant) i.e., Space Complexity of this Operation is Θ (1).

ALGORITHM DeQueue(S1,S2)

Input: Stack S1 and S2

Output: Updated Stack S2

BEGIN:

```
IF IsEmpty( ) THEN
        WRITE("Queue Empty")
        EXIT( )
ELSE
        WHILE !IsEmpty(S1) THEN
                PUSH(S2,POP(S1))
y = POP(S2)

WHILE !IsEmpty(S2)
        PUSH(S1,POP(S2))

RETURN y
```

END;

In case both stacks are empty, exit

Pop all the elements of stack 1 and Push into stack 2

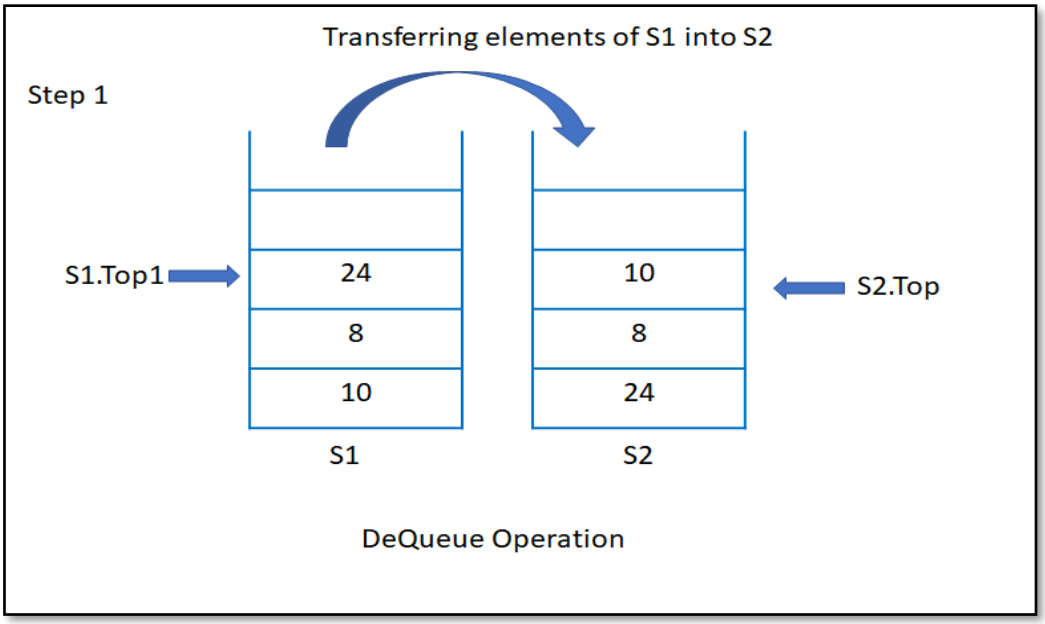
Store the removed element of S2 into y

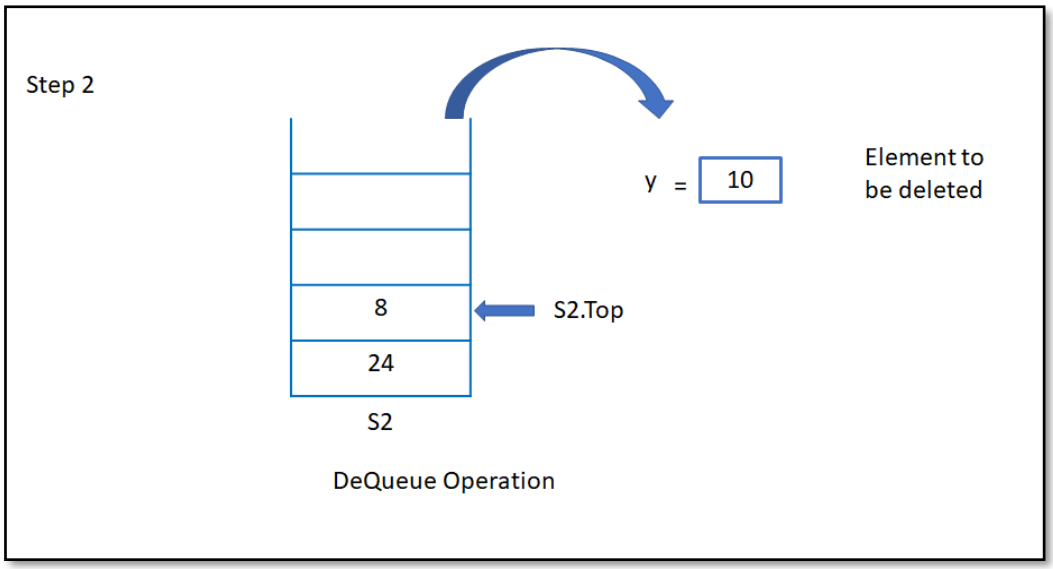
Pop all the elements of stack 2 and Push back into stack 1

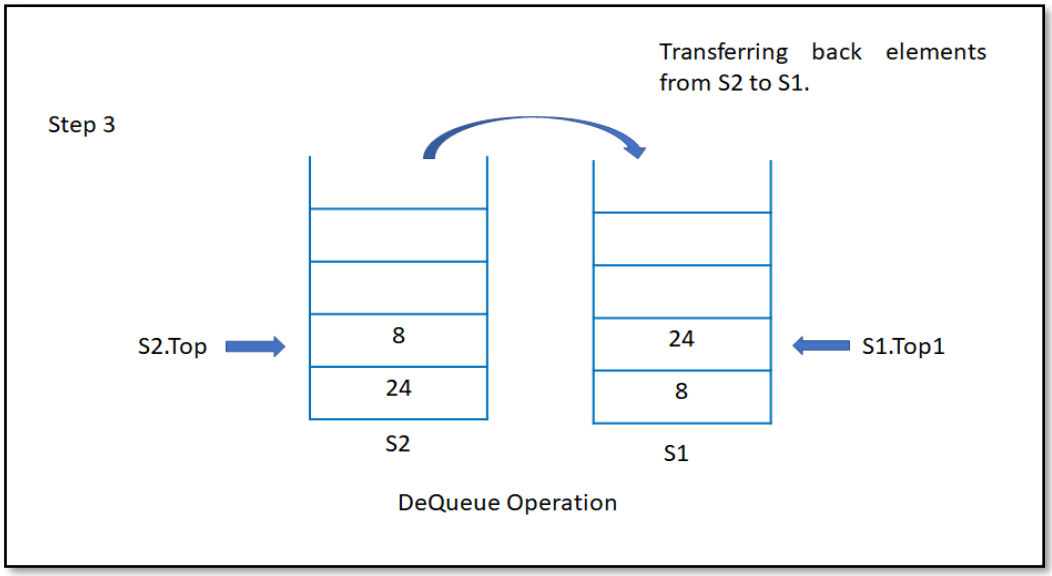
Return the element inserted first just like queue

Time Complexity: Push operation requires Θ(n) time due to a while loop in both S1 and S2. So, the time complexity of DeQueue operation will be Θ(n).

Space Complexity: Since elements of S1 will be copied to S2. There is auxiliary space used in the Algorithm, i.e., Space Complexity of this Operation is Θ (n).


Transferring elements of S1 into S2
Step 1
S1.Top1
24
8
10
S1
10
8
24
S2
S2.Top
DeQueue Operation


Step 2
y = 10
Element to be deleted
8
S2.Top
24
S2
DeQueue Operation

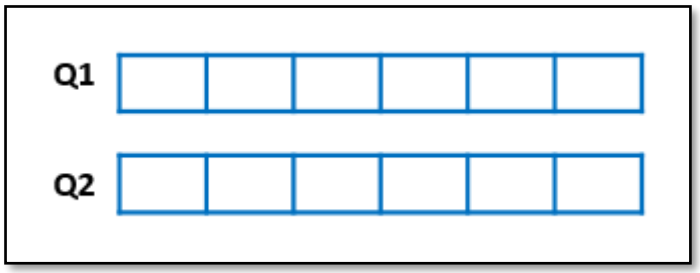


8.11 Implementation of Stack Using Queue

We can consider this problem as an example of Abstraction. User is able to access the stack (push and pop) but have no concern about the data structure to be used in order to implement the stack (which data structure to be used to store actual data of stack).
While implementing stack by using array or linked list, push() and pop() operation takes constant time, but when we implement stack by using queue, both of these operations cannot take constant time, at least one of them will be costly.

Array	Push() and pop() takes Θ(1) time
Linked List	Push() and pop() takes Θ(1) time
Queue	Push()=Θ(1) pop= Θ(n) or Push()=Θ(n) pop= Θ(1)

Stack can be implemented using the Queue by taking two Queues. We name these queues as Q1 and Q2. Let us now see all the primitive operations.



ALGORITHM InitializeStack(Queue Q1, Queue Q2)
Input: Two Queues Q1 and Q2
Output: None
BEGIN:

```
        InitializeQueue(Q1)
        InitializeQueue(Q2)
END;
```

Initialize Queues Q1 and Q2

Time Complexity: Time Complexity of this Operation is Θ(1) as there are only two statements to execute.
Space Complexity: Since there is no auxiliary space used in the Algorithm, the space function is 0 (constant) i.e., Space Complexity of this Operation is Θ (1).

ALGORITHM IsEmptyStack(Queue Q1, Queue Q2)
Input: Two Queues Q1 and Q2
Output: True or False based on emptiness
BEGIN:

```
        IF IsEmptyQueue(Q1) THEN
                RETURN TRUE
        ELSE
                RETURN FALSE
END;
```

If Q1 is empty, The Stack will be empty

Time Complexity: Time Complexity of this Operation is Θ(1) as there are only two statements to execute even when condition is TRUE or FALSE. Here condition check is considered as first statement and return as the second one.
Space Complexity: Since there is no auxiliary space used in the Algorithm, the space function is 0 (constant) i.e. Space Complexity of this Operation is Θ (1).

Method 1:

ALGORITHM Push(Queue Q1, Queue Q2, Item)
Input: Two Queues Q1 and Q2 and item to be inserted
Output: None
BEGIN:

```
        EnQueue(Q2, item)
```

Insert item into Q2

```
        WHILE !IsEmpty(Q1) DO
                EnQueue(Q2, DeQueue(Q1))
```

Remove all elements from Q1 and Insert into Q2

```
        TEMP=Q1
        Q1=Q2
        Q2=temp
END;
```

Exchange the names Q1 and Q2

Time Complexity: Time Complexity of push Operation is Θ (N) as N-1 element copied from Q1 to Q2.

Space Complexity: Space Complexity of this push Operation is Θ(N) as we are using two queues. If we consider queues are already given then its complexity is Θ(1).

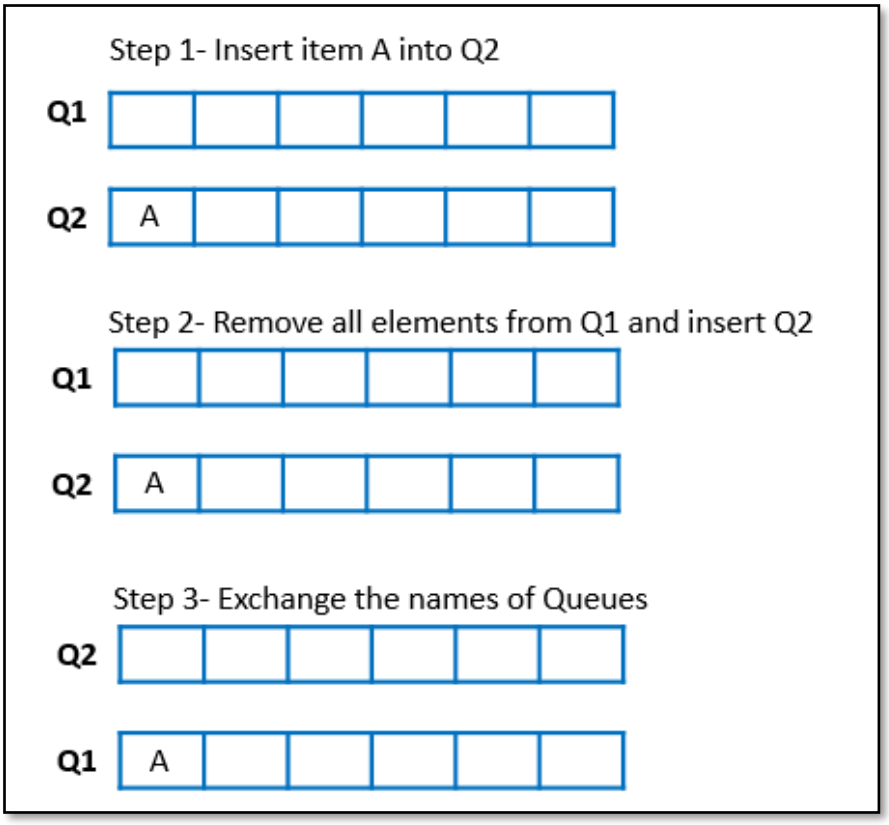


Figure: steps for PUSH A

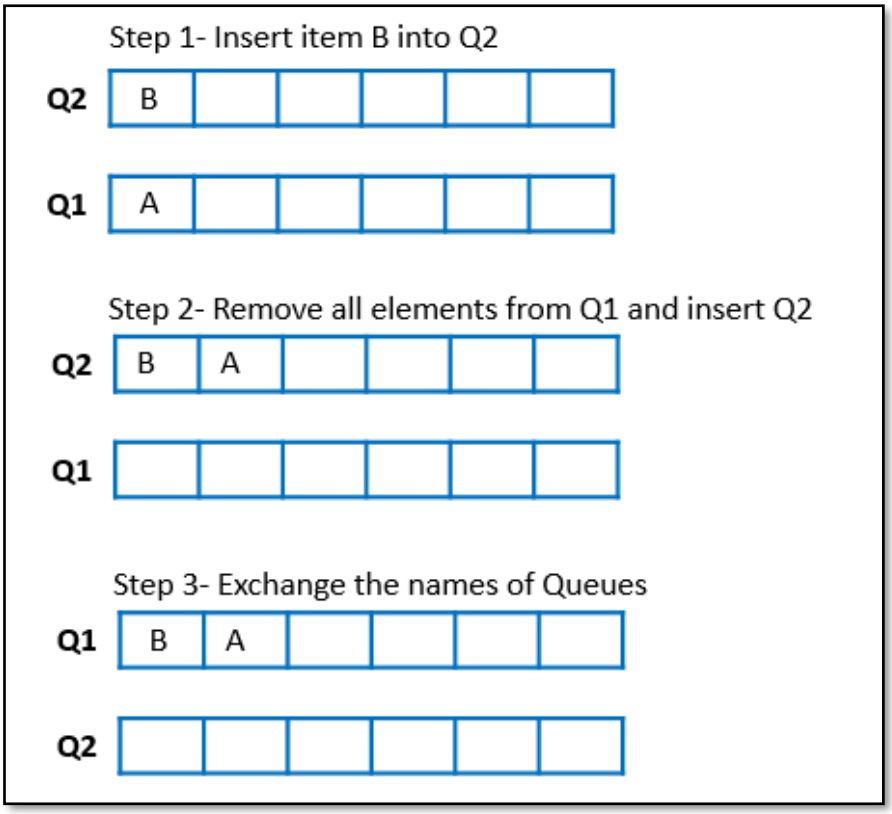


Figure: Steps for PUSH B

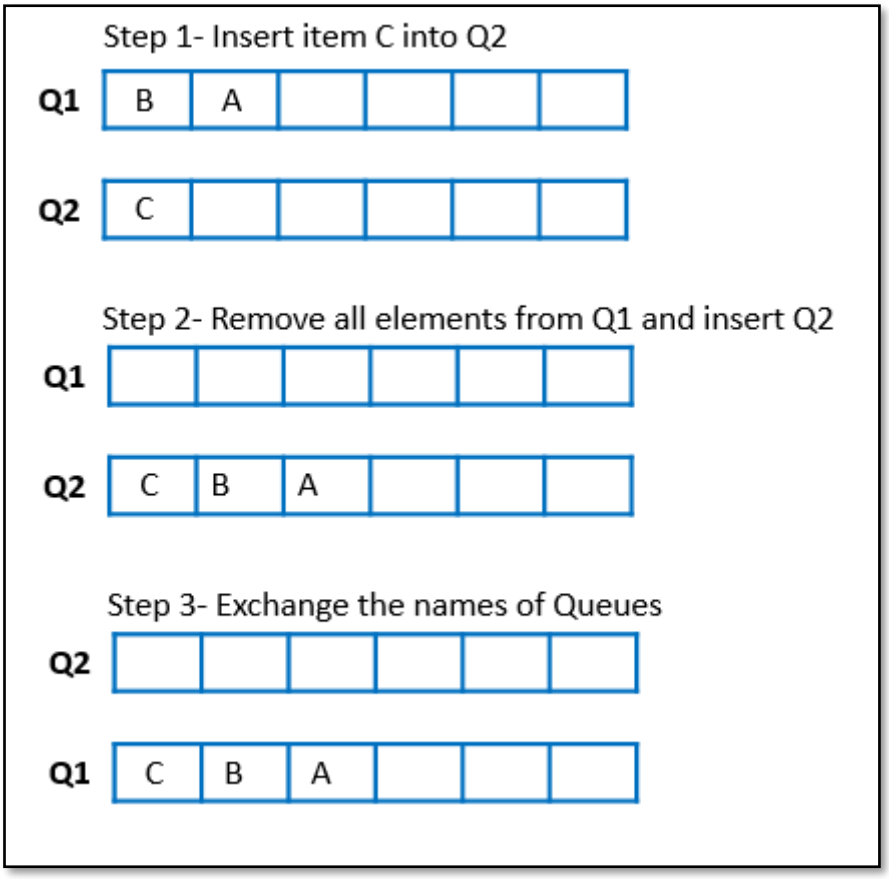


Figure: Steps for PUSH C

ALGORITHM Pop(Queue Q1, Queue Q2)
Input: Two Queues Q1 and Q2
Output: Deleted Element
BEGIN:

```
IF IsEmptyStack(Q1) THEN
        WRITE("Stack Underflows")
        EXIT(1)
```

If Q1 is empty then stack underflows

```
x = DeQueue(Q1)
RETURN x
```

Remove element from Queue Q1 and Return

END;

Time Complexity: Time Complexity of pop Operation is O(1) as last inserted item deleted first.

Space Complexity: Space Complexity of this pop Operation is O(n) as we are using two queues. If we consider queues are already given then its complexity is O(1).

Let us suppose that some items already pushed into stack. Let item=P to be pushed into stack then we push x into Q2 and pop every item from Q1 and pushed into queue Q2.

Method 2:

ALGORITHM Push(Queue Q1,Item)
Input: Queue Q1 and item to be inserted
Output: None
BEGIN:

```
EnQueue(Q1, item)
```

Insert item into Q1

END;

Time Complexity: Time Complexity of push Operation is O(1) as simply insert the item at rear.
Space Complexity: Space Complexity of this push Operation is O(n) as we are using two queues. If we consider queues are already given then its complexity is O(1).

ALGORITHM Pop(Queue Q1, Queue Q2)
Input: Two Queues Q1 and Q2
Output: Deleted element
BEGIN:

```
IF IsEmptyStack(Q1) THEN
    WRITE("Stack Underflows")
    EXIT(1)
```

If Q1 is empty then stack underflows

```
WHILE !IsEmpty(Q1) DO
    x = DeQueue(Q1)
    EnQueue(Q2,x)
```

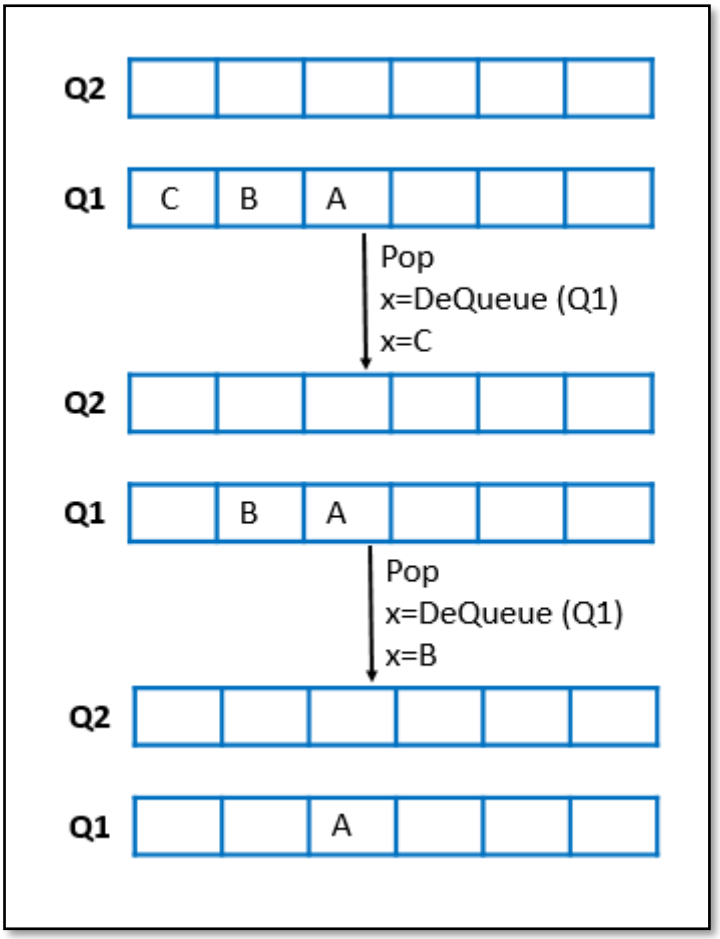
Remove all elements from Q1 and Insert into Q2

```
TEMP=Q1
Q1=Q2
Q2=temp
```

Exchange the names Q1 and Q2

END;

Time Complexity: Time Complexity of pop Operation is O(n) as n-1 element copied from Q1 to Q2.
Space Complexity: Space Complexity of this pop Operation is O(n) as we are using two queues. If we consider queues are already given then its complexity is O(1).



8.12 Implementation of Queue using Linked List

Queue works on the principle of FIFO – First in first out scheme. To ensure the FIFO behavior, we can think of insertion at the end and deletion of items from the beginning of the Linked List.

1. Initialization

ALGORITHM Initialize(Front, Rear)
Input: A Queue with Rear and Front end
Output: None
BEGIN:
Front = NULL
Rear = NULL
END;

} Rear and Front both initialized to NULL

2. Emptiness Check

The function will check if the given Deque is empty and TRUE or FALSE accordingly.

ALGORITHM IsEmpty(Rear, Front)
Input: A Queue with Rear and Front end
Output: True or False based on emptiness
BEGIN:
IF Rear = NULL AND Front = NULL THEN
RETURN TRUE
ELSE
RETURN FALSE
END;

3. Insertion

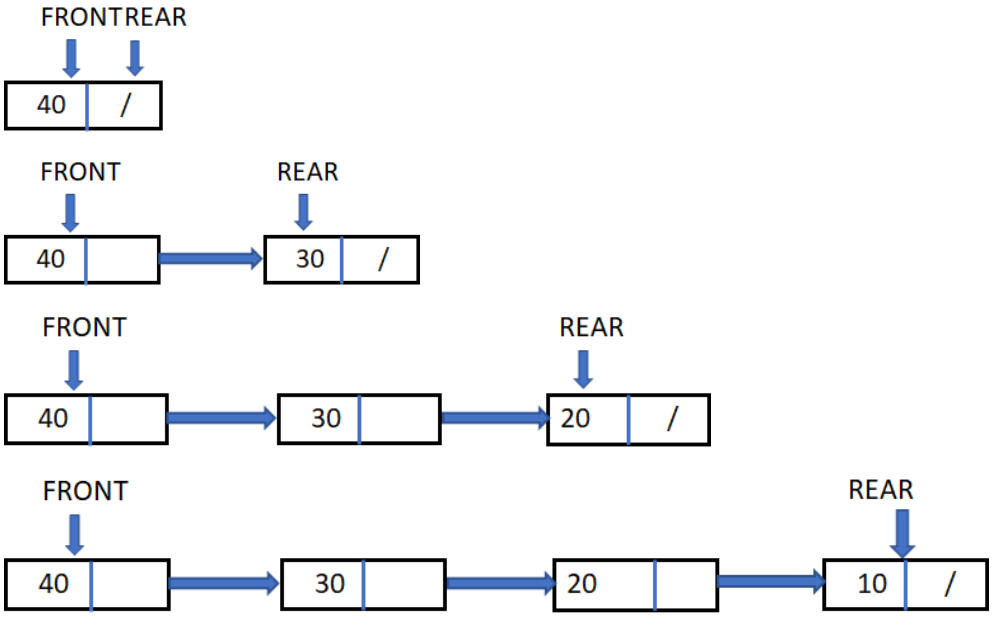
ALGORITHM Enqueue(Front, Rear, item)
Input: A Queue with Rear and Front end
Output: None
BEGIN:
IF Front = = NULL AND Rear = = NULL THEN
InsBeg(Rear, item)
Front = Rear
ELSE
InsAfter(Rear, item)
Rear = Rear →NEXT
END;

Insertion as the first node

Insertion after the node at Rear End

ALGORITHM InsAfter(Rear, item)
BEGIN:
P =GetNode()
P → Info = item
P → Next = NULL
Rear → Next = P
Rear = Rear → Next
END;

ALGORITHM InsBeg(Rear, item)
BEGIN:
P =GetNode()
P → Info = item
P → Next = NULL
Rear = P
END;



4. Deletion

ALGORITHM Dequeue(Front, Rear)

BEGIN:

IF Front = = NULL AND Rear = = NULL THEN
WRITE (“Queue Underflow”)
RETURN

Checking the underflow Condition

ELSE
x = Delbeg(Front)

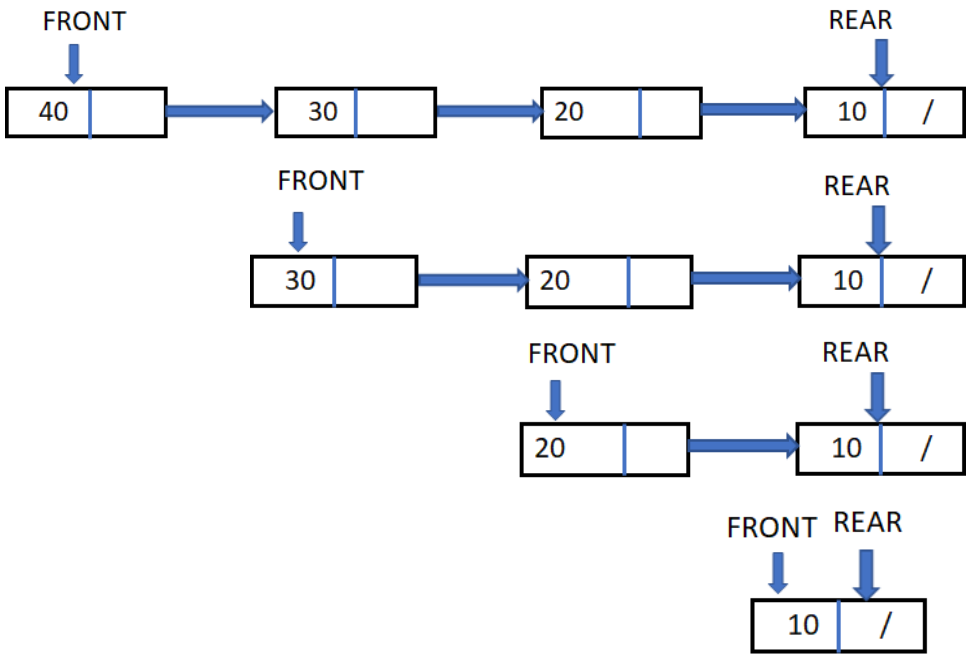
Performing the deletion of node at Front

IF Front = = NULL THEN
Rear = NULL

In case a single node was there in the Queue.
After the deletion Queue should depict empty

END;

ALGORITHM DelBeg(Rear, Front)
Input: A Queue with Rear and Front end
Output: Deleted element
BEGIN:
Q = Front → DATA
Front = Front → Next
RETURN Q
END;



Explanation: The above algorithm performs QUEUE implementation using Linked List. For each QUEUE Initialization, Insertion and deletion operation Initialize( ), Enqueue( ), Dequeue ( ) functions are created respectively.

Complexity:

Time Complexity: Enqueue operation: O(1)

Dequeue operation: O(1)

Space Complexity: Enqueue operation: O(1)

Dequeue operation: O(1)

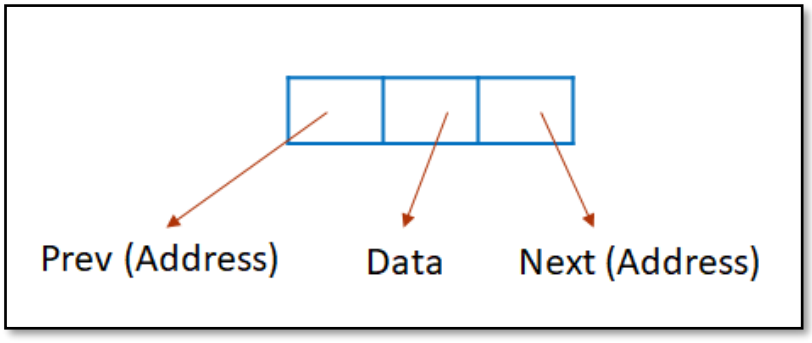
8.13 Implementation of Double-Ended Queue using Linked List

For the Linked list implementation of Double-ended queue, we are considering the linked list nodes to contain 3 fields

Prev – Contains the address of the previous node

Data – Contains the data element of Queue

Next – Contains the address of the next node



Primitive operations on Deque are listed as under

- Initialization
- Emptiness check
- Insert at left
- Delete from left
- Insert at right
- Delete from Right

1. Initialize

Both Front and Rear need to contain Null to depict that initially, there is no data item in the Deque.

ALGORITHM InitializeDeque(Rear, Front)
Input: A Deque with Rear and Front end
Output: None
BEGIN:
    Rear = NULL
    Front = NULL
END;

2. Emptiness Check

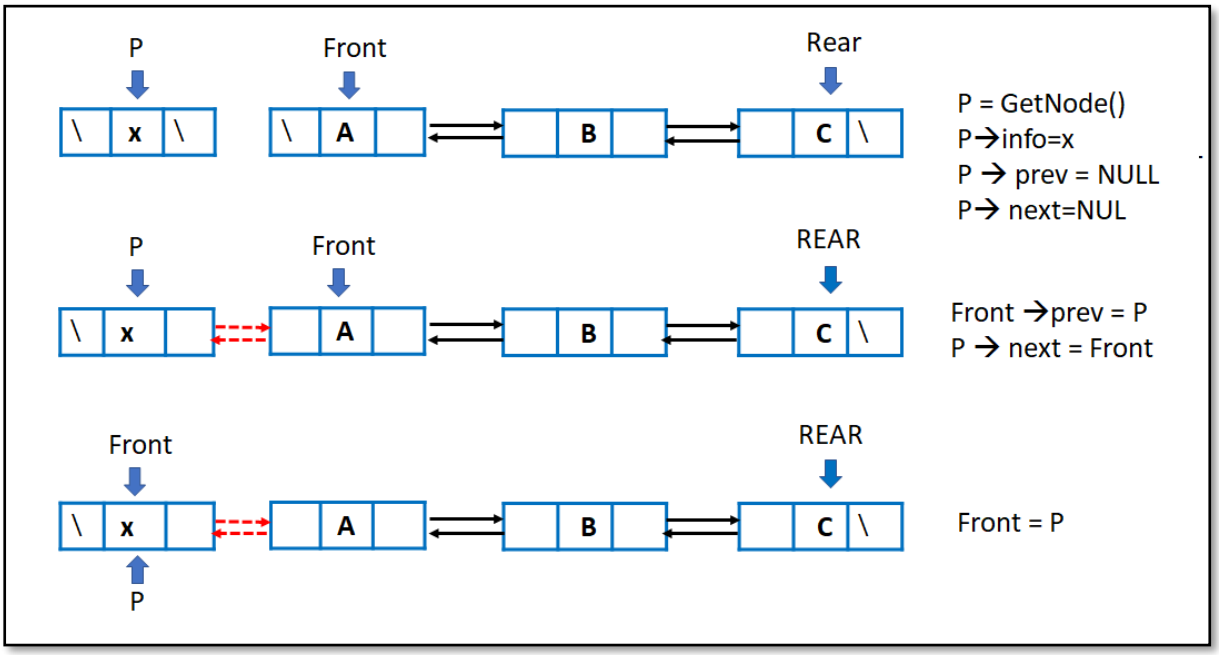
The function will check if the given Deque is empty and TRUE or FALSE accordingly.

ALGORITHM IsEmpty(Rear, Front)
Input: A Deque with Rear and Front end
Output: True or False based on emptiness
BEGIN:
    IF Rear = NULL AND Front = NULL THEN
        RETURN TRUE
    ELSE
        RETURN FALSE
END;

3. Insertion at the Front end

ALGORITHM: InsFront (Front, Rear, item)
Input: A Deque with Rear and Front and an item to be inserted
Output: None
BEGIN:
    R = GetNode() // R will be the new node allocated
    R → info = x // R data part will contain value x
    R → prev = NULL // R node previous address part will point to null

R → next = NULL // R node next address part will point to null
IF Front == NULL THEN
Front=R
Rear = R
ELSE
Front →prev = R // R will be inserted as a immediate left of P
R →next = Front // R immediate right will be P
Front = R
END;



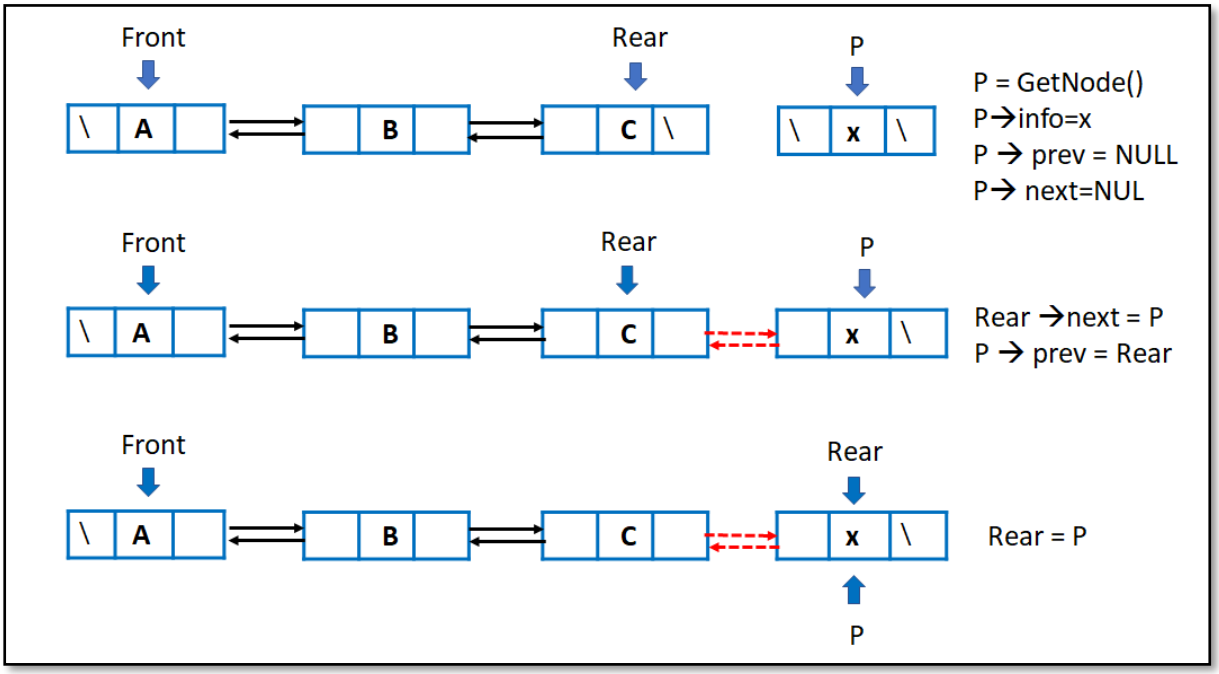
4. Insertion at the Rear end

ALGORITHM: InsRear (Front, Rear, x)
Input: A Deque with Rear and Front and an item to be inserted
Output: None
BEGIN:
P = GetNode() // R will be the new node allocated
P → info = x // R data part will contain value x
P → prev = NULL // R node previous address part will point to null
P → next = NULL // R node next address part will point to null

IF Rear == NULL THEN
Front = P
Rear = P
ELSE

Rear →next = P // R will be inserted as a immediate left of P
P →Prev = Rear // R immediate right will be P
Rear = P

END;



Time Complexity: Time Complexity of this Operation is Θ(1) as there is only one statement to execute.
Space Complexity: Since there is no auxiliary space used in the Algorithm, the space function is 0 (constant) i.e., Space Complexity of this Operation is Θ (1).

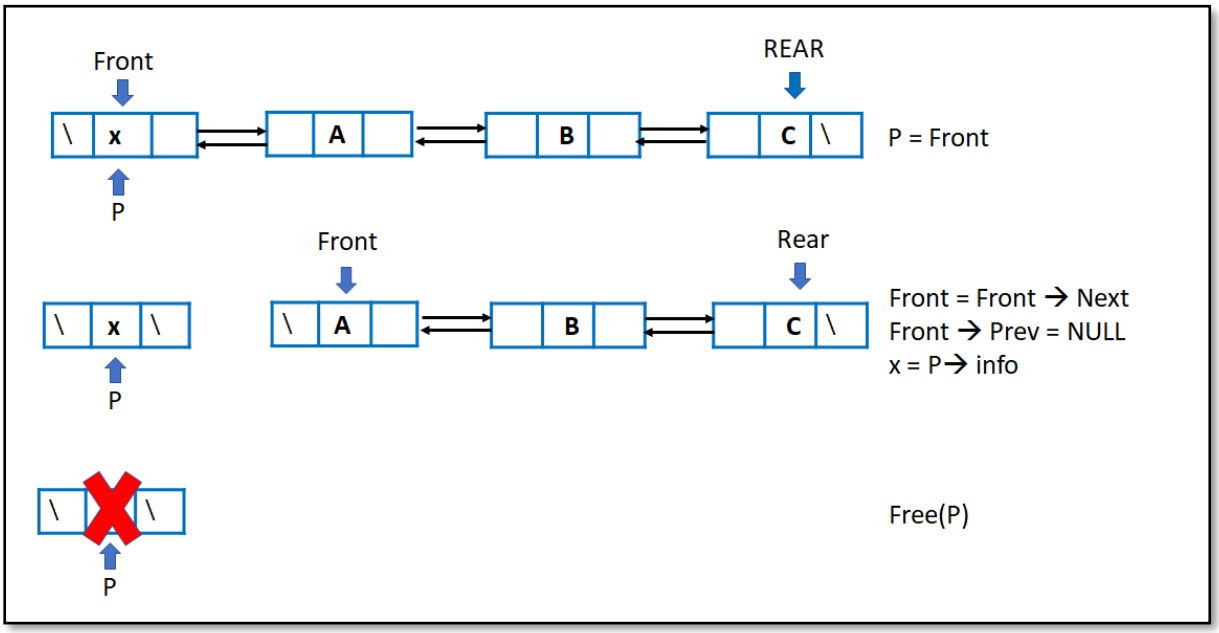
5. Deletion from Front End

If we want to delete a node from the Front end, there would be two cases assuming it is non-empty. In the first case there will be only one node, while in second there would be more than that. Now, if there would be only one node, both Front and Rear will point to address null after deletion.

The following figure shows the deletion from the Front end.

ALGORITHM DelFront (Front, Rear)
Input: A Deque with Rear and Front
Output: Deleted Element
BEGIN:
P = Front
Front = Front → Next

```
        IF (Front == NULL)
                Rear = NULL
        ELSE
                Front → Prev = NULL
        x = P→ info
        Free(P)
        RETURN x
END;
```



Time Complexity: Time Complexity of this Operation is Θ(1) as there is only one statement to execute.

Space Complexity: Since there is no auxiliary space used in the Algorithm, the space function is 0 (constant) i.e., Space Complexity of this Operation is Θ (1).

6. Deletion from Rear End

If we want to delete a node from rear end than there would be two cases assuming it to be non-empty. In first case there will be only one node while in second there would be more than that. Now if there would be only one node than after deletion both Front and rear will point to address null. On other hand if there exists more than one node than we need to shift rear to its previous address.

Let's see the deletion through rear end.

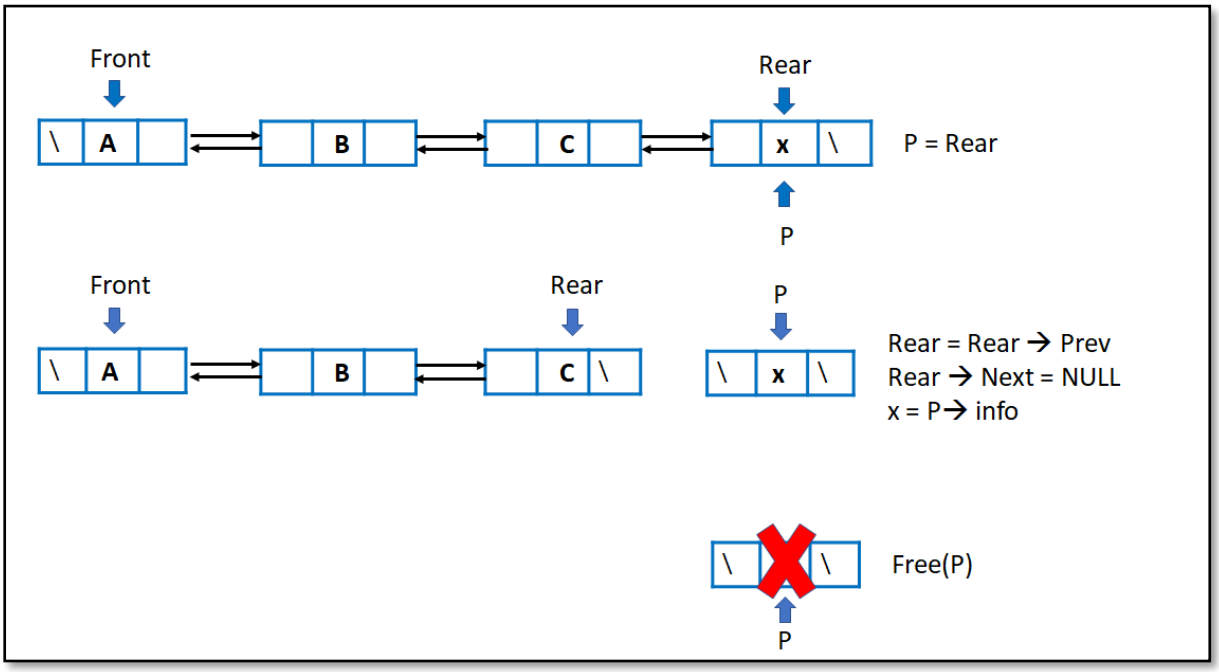
ALGORITHM DelRear (Front, Rear)

Input: A Deque with Rear and Front

Output: Deleted Element

BEGIN:

```
        P = Rear
        Rear = Rear → Prev
        IF (Rear == NULL)
                Front = NULL
        ELSE
                Rear → Next = NULL
        x = P→ info
        Free(P)
        RETURN x
END;
```

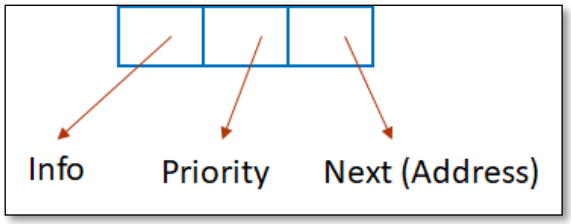


Time Complexity: Time Complexity of this Operation is Θ(1) as there is only one statement to execute.

Space Complexity: Since there is no auxiliary space used in the Algorithm, the space function is 0 (constant) i.e., Space Complexity of this Operation is Θ (1).

8.14 Priority Queue Implementation using Linked List

For priority queue implementation using link list, add extra field priority is used in the node (along with 2 usual fields as Info and Next).



1. Initialization

ALGORITHM InitializePQueue(START)
Input: A Priority Queue with START as the address of the First Element
Output: None
BEGIN:
START = NULL
END;

2. Emptiness Check

ALGORITHM IsEmptyPQueue(START)
Input: A Priority Queue with START as the address of the First Element
Output: TRUE or FALSE based on Emptiness
BEGIN:
IF START == NULL THEN
RETURN NULL
ELSE
RETURN FALSE
END;

3. Insertion

For insertion of an item in the Priority Queue, the following steps can be performed.

a) If the start contains NULL, then it simply inserts the info as a first node.
b) If start does not contain NULL, then first compare the priority field and insert according to the priority.
c) In the case of equal priority, insertion takes place after the element with the same priority.

ALGORITHM EnQueue(START, Item, Py)
Input: A Priority Queue with START as the address of the First Element, item and priority
Output: None
BEGIN:
P=GetNode()
P→info = Item
P→Priority = Py
Q = NULL
R = START
WHILE R!=NULL AND Py >= R→priority DO
Q=R
R=R→Next

```
        IF Q==NULL THEN
                InsBeg(START, Item, Py)
        ELSE
                InsAft(START, Item, Py)
END;
```

Time Complexity: Algorithm takes O(N) time because traversal is required in the list.
Space Complexity: Algorithm requires a single variable for traversal and another one for the new node. The total space is constant; hence the space complexity is O(1).

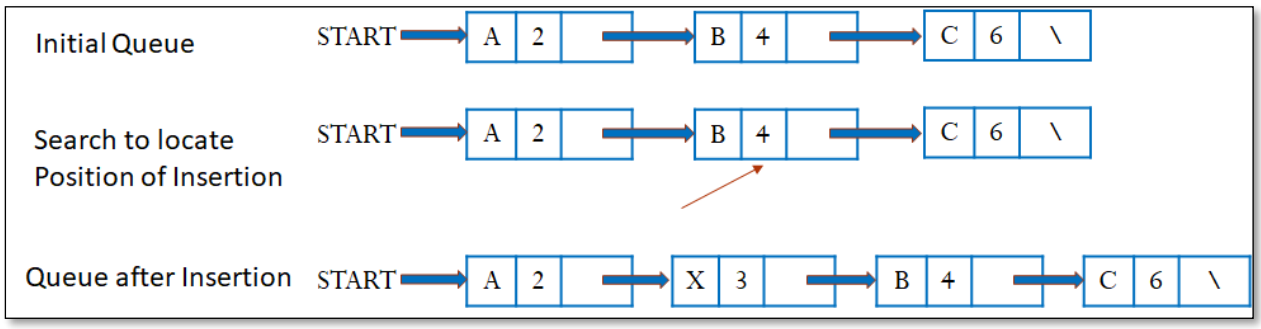


Figure a: Showing Insertion of Key X with Priority 3

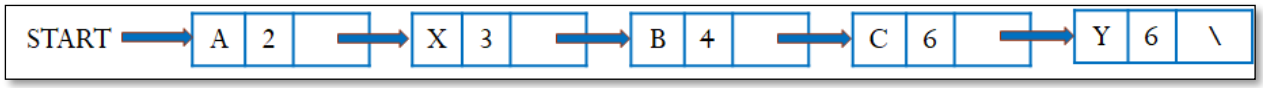


Figure b: Showing Insertion of Key Y with Priority 6

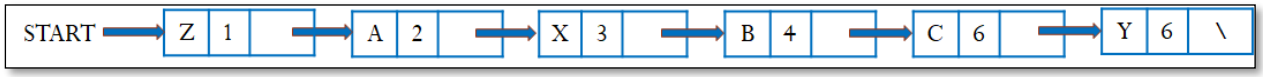


Figure c: Showing Insertion of Key Y with Priority 6

4. Deletion

The first node represents the highest priority node. Therefore, in case of deletion, delete this node.

ALGORITHM DeQueue(START)
Input: A Priority Queue with START as the address of the First Element
Output: Deleted Element with the highest priority
BEGIN:

```
        IF START==NULL THEN
                WRITE("Void Deletion")
                EXIT(1)
        ELSE
                Item=DelBeg(START)
```

RETURN Item

END;

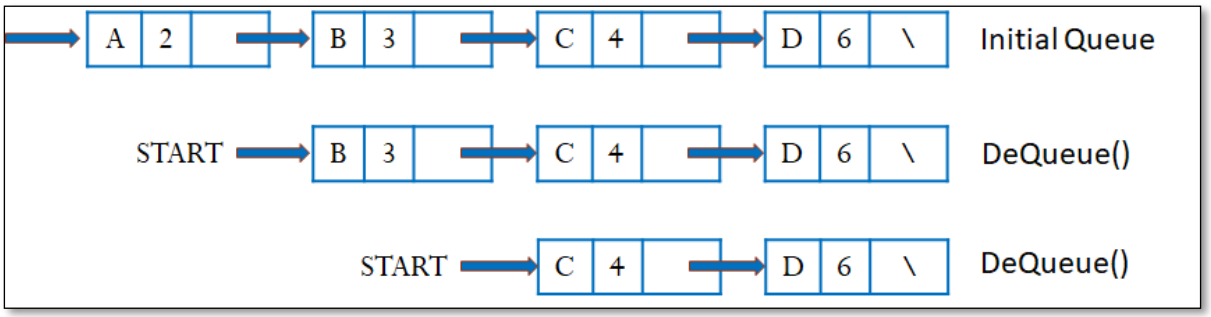


Figure d: Showing DeQueue() Operation

Time Complexity: Algorithm takes O(1) time because of single pointer adjustment.
Space Complexity: Algorithm requires a single variable only; hence space complexity is O(1).

8.15 Competitive Coding Problems

1. Queue Reversal

Given a Queue Q containing N elements. The task is to reverse the Queue.

Example 1:

Input:
6
4 3 1 10 2 6

Output:
6 2 10 1 3 4

Example 2:

Input:
4
4 3 2 1

Output:
1 2 3 4

2. Task Scheduler

Given a character array task that represents the tasks a CPU needs, each letter represents a different task. Tasks could be done in any order. Each task is done in one unit of time. For each unit of time, the CPU could complete either one task or just be idle.

However, there is a non-negative integer n that represents the cooldown period between two **same tasks** (the same letter in the array), that is that there must be at least n units of time between any two same tasks.

Return *the least number of units of time the CPU will take to finish all the given tasks*.

Example 1:

Input: tasks = ["A","A","A","B","B","B"], n = 2

Output: 8

Explanation:

A -> B -> idle -> A -> B -> idle -> A -> B

There are at least 2 units of time between any two same tasks.

Example 2:

Input: tasks = ["A","A","A","B","B","B"], n = 0

Output: 6

Explanation: In this case, any permutation of size 6 would work since n = 0.

["A","A","A","B","B","B"]

["A","B","A","B","A","B"]

["B","B","B","A","A","A"]

...

And so on.

Example 3:

Input: tasks = ["A","A","A","A","A","A","B","C","D","E","F","G"], n = 2

Output: 16

Explanation:

One possible solution is

A -> B -> C -> A -> D -> E -> A -> F -> G -> A -> idle -> idle -> A -> idle -> idle -> A

8.16 Multiple Choice Questions

1	Suppose a stack implementation supports an instruction REVERSE, which reverses the order of elements on the stack, in addition to the PUSH and POP instructions. Which one of the following statements is TRUE (with respect to this modified stack)?
A	A queue cannot be implemented using this stack.
B	A queue can be implemented where ENQUEUE takes a single instruction and DEQUEUE

	takes a sequence of two instructions
C	A queue can be implemented where ENQUEUE takes a sequence of three instructions and DEQUEUE takes a single instruction.
D	A queue can be implemented where both ENQUEUE and DEQUEUE take a single instruction each
AN	**C**

2	A queue is implemented using a non-circular singly linked list. The queue has a head pointer and a tail pointer. Let n denote the number of nodes in the queue. Let enqueue be implemented by inserting a new node at the head, and dequeue be implemented by deleting a node from the tail. Which one of the following is the time complexity of the most time-efficient implementation of enqueue and dequeue, respectively, for this data structure?
A	O(n), O(n)
B	O(1), O(1)
C	O(1), O(n)
D	O(n), O(1)
AN	**C**

3	How many Queues are required to implement stack if queue size is infinite?
A	2
B	1
C	3
D	Not possible
AN	**B**

4	The following sequence of operation is performed on a Queue Q : Enqueue(20) , Enqueue(30) , Enqueue(40), Dequeue , Dequeue , Enqueue(100), Enqueue(70) , Enqueue(34), Dequeue , Dequeue , Dequeue .The sequence of values removed from the queue will be ?
A	40 , 30 , 20 , 100 , 70
B	20 , 30 , 100 , 70 , 34
C	40 , 30 , 20 , 34 , 70
D	20 , 30 , 40 , 100 , 70
AN	**D**
DL	**M**
TP	

5	The following sequence of operation is performed on a Queue Q : Enqueue(20) , Enqueue(30) , Enqueue(40), Dequeue , Dequeue , Enqueue(100),

	Enqueue(70) , Enqueue(34), Dequeue , Dequeue , Dequeue .The number of elements present in the queue will be ?
A	0
B	1
C	2
D	3
AN	**B**
DL	**E**
TP	

6	The following sequence of operation is performed on a Queue Q : Enqueue(20) , Enqueue(30) , Enqueue(40), Dequeue , Dequeue , Enqueue(100), Enqueue(70) , Enqueue(34), Dequeue , Dequeue , Dequeue .The Maximum size of the queue will be ?
A	4
B	3
C	5
D	6
AN	**A**
DL	**M**

7	The data structure required for Breadth First Traversal in graph is ?
A	Linked List
B	Stack
C	Queue
D	Array
AN	**C**
DL	**E**
TP	

8	A circular queue is implemented using an array of size 10. The array index starts with 0, front is 6, and rear is 9 . The insertion of the next element takes place at the array index?
A	10
B	1
C	0
D	2
AN	**C**
DL	**E**
TP	

9	A circular queue is implemented using an array of size 10. The array index starts with 0, front is 6, and rear is 5 . The insertion of the next element takes place at the array index?
A	5
B	6
C	0
D	None of the above
AN	**D**

10	Suppose we have a circular array implementation of the queue class, with 10 items stored in the queue in q[2] through q[11]. The capacity is 42 . Where does the push function place the new entry in the array?
A	q[12]
B	q[1]
C	q[11]
D	q[2]
AN	**A**
11	**Consider a standard Circular Queue 'q' implementation (which has the same condition for Queue Full and Queue Empty) whose size is 11 and the elements of the queue are q[0], q[1], q[2].....,q[10]. The front and rear pointers are initialized to point at q[2] . In which position will the ninth element be added?**
A	**q[1]**
B	**q[0]**
C	**q[9]**
D	**q[10]**
AN	**B**

12	Consider the following pseudo-code. Assume that IntQueue is an integer queue. What does the function fun do?
	`void fun(int n)`
	`{`
	`  IntQueue q = new IntQueue();`
	`  q.enqueue(0);`
	`  q.enqueue(1);`
	`  for (int i = 0; i < n; i++)`
	`  {`
	`  int a = q.dequeue();`
	`  int b = q.dequeue();`
	`  q.enqueue(b);`

	q.enqueue(a + b);
	ptint(a);
	}
	}

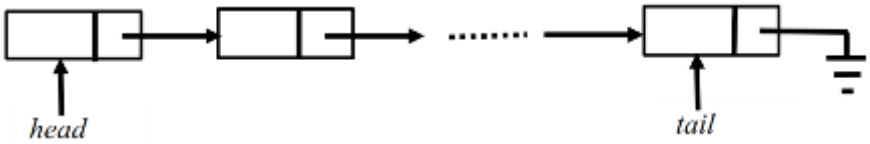
A	Prints numbers from 0 to n-1
B	Prints numbers from n-1 to 0
C	Prints first n Fibonacci numbers in reverse order.
D	Prints first n Fibonacci numbers
AN	**D**

13	Following is C like pseudo-code of a function that takes a Queue as an argument and uses a stack S to do the processing.
	void fun(Queue *Q)
	{
	Stack S;
	while (!isEmpty(Q))
	{
	push(&S, deQueue(Q));
	}
	while (!isEmpty(&S))
	{
	enQueue(Q, pop(&S));
	}
	}
	What does the above function do in general?

A	Removes the last from Q
B	Keeps the Q same as it was before the cal
C	Makes Q empty
D	Reverses the Q
AN	**D**

14	What are the minimum enqueue and dequeue operations needed to perform pop operation for a stack which is implemented with two queues if there are already 10 elements in the first queue
A	enqueue- 10, dequeue- 9
B	enqueue- 10, dequeue- 10
C	enqueue- 9, dequeue- 10
D	enqueue- 8, dequeue- 10

AN	**C**
DL	**Easy**

15.	How many queues are needed to implement a stack?
A	1
B	2
C	3
D	None of the above
B	ANS
16.	The queue data structure is to be realized by using stack . The number of stack needed would be ISRO CS 2015
A	2
B	4
C	1
D	It can not be implemented.
A	ANS
17.	What is the most appropriate data structure to implement a priority queue? UGC Net CS 2010
A	Heap
B	Linked List
C	Circular array
D	Binary tree
A	ANS
18.	A queue is implemented using a non-circular singly linked list. The queue has a head pointer and a tail pointer, as shown in the figure. Let n denote the number of nodes in the queue. Let enqueue be implemented by inserting a new node at the head, and dequeue be implemented by deleting a node from the tail.  Which one of the following is the time complexity of the most time-efficient implementation of enqueue and dequeue, respectively, for this data structure? GATE 2018
A	θ(1),θ(1)
B	θ(1),θ(n)
C	θ(n),θ(1)
D	θ(n),θ(n)

B	ANS
19.	A circular queue has been implemented using a singly linked list where each node consist of a value and a single pointer pointing to the next node. We maintain exactly two external pointers FRONT and REAR pointing to the front node and rear node of the queue, respectively. Which of the following statement is/ are CORRECT for such a circular queue, so that insertion and deletion operations can be performed in 0 (1) time? i. Next pointer of front node point to the rear node. II. Next pointer of rear node points to the front node. GATE 2017
A	Both I and II
B	Neither I nor II
C	I only
D	II only
D	ANS
20.	A queue is implemented using an array such that ENQUEUE and DEQUEUE operations are performed efficiently. Which one of the following statements is CORRECT (n refers to the number of items in the queue)? GATE 2016
A	Both operations can be performed in O (1) time.
B	At most one operation can be performed in *O* (1) time but the worst-case time for the other operation will be Ω(n)
C	The worst-case time complexity for both operations will be Ω(n)
D	Worst case time complexity for both operations will be Ω(logn)
A	ANS

21.	Consider the following operation along with Enqueue and Dequeue operations on queues, where k is a global parameter.
	MultiDequeue(Q){
	m = k
	while (Q is not empty) and (m > 0) {
	Dequeue(Q)
	m = m – 1
	}
	}
	What is the worst-case time complexity of a sequence of n queue operations on an initially empty queue? GATE 2013

A	Θ(n)

B	Θ(n + k)
C	Θ(nk)
D	Θ($n^{2)}$
A	ANS
22.	Suppose a circular queue of capacity (n −1) elements is implemented with an array of n elements. Assume that the insertion and deletion operations are carried out using REAR and FRONT as array index variables, respectively. Initially, REAR = FRONT = 0. The conditions to detect queue full and queue empty are GATE 2012
A	full: (REAR+1) mod n == FRONT empty: REAR == FRONT
B	full: (REAR+1) mod n == FRONT empty: (FRONT+1) mod n == REAR
C	full: REAR == FRONT empty: (REAR+1) mod n == FRONT
D	full: (FRONT+1) mod n == REAR empty: REAR == FRONT
A	ANS
23.	Queue is an appropriate data structures for implementation of
	a) Implementation of Breadth First Search.
	b) Implementation of depth First Search.
	c) Process Scheduling.
	Which option is correct?
A	A and B
B	A and C
C	B and C
D	A, B and C
B	Ans
24.	Consider the following two statement S1: A queue can be implemented using two stack. S2: A stack can be implemented using two queues. Which option is correct? UGC NET 2016
A	S1 is correct but S2 is not correct.
B	S2 is correct but S2 is not correct.
C	Both S1 and S2 are correct.
D	Both S1 and S2 are incorrect.

C	ANS