Queues

Week 9

Lectures 1-2

In this session, you will learn to:

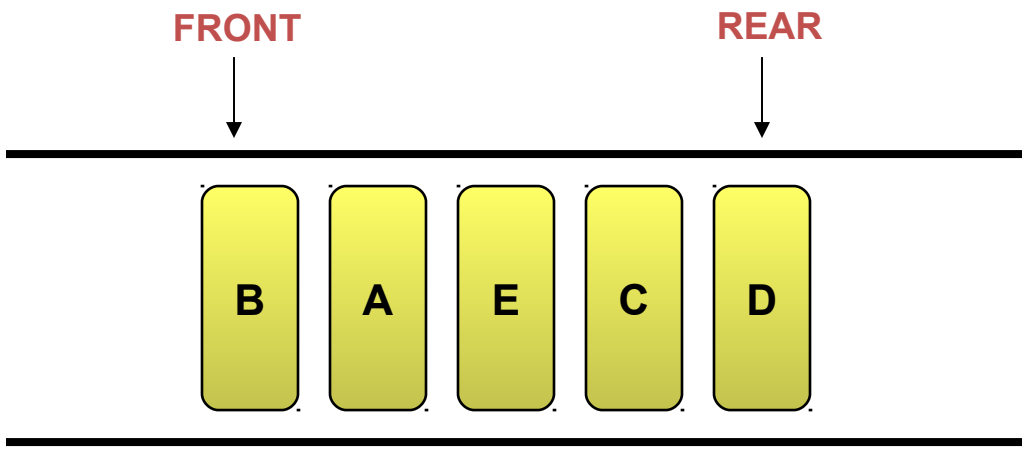
- Overview of Queue
- Applications of Queues
- Insertion and deletion in a Linear queue
- Implementation of Circular Queue
- Dequeue
- Priority Queue

- Consider a situation where you have to create an application with the following set of requirements:
  - Application should serve the requests of multiple users.
  - At a time, only one request can be processed.
  - The request, which came first should be given priority.

- However, the rate at which the requests are received is sometimes faster than the rate at which they are processed.
  - Therefore, you need to store the request somewhere until they are processed.
  - The times during which requests arrive faster than they can be processed is called a *burst*.
  - A process that generates requests are widely varying rates is called *bursty*.

- How can you solve this problem?
  - You can solve this problem by storing the requests in such a manner so that they are retrieved in the order of their arrival.
  - A data structure called queue stores and retrieves data in the order of its arrival.
  - A queue is also called a FIFO list.

- Elements are inserted at the rear end and deleted from the front end.
- Queue is a list of elements in which an element is inserted at one end and deleted from the other end of the queue.

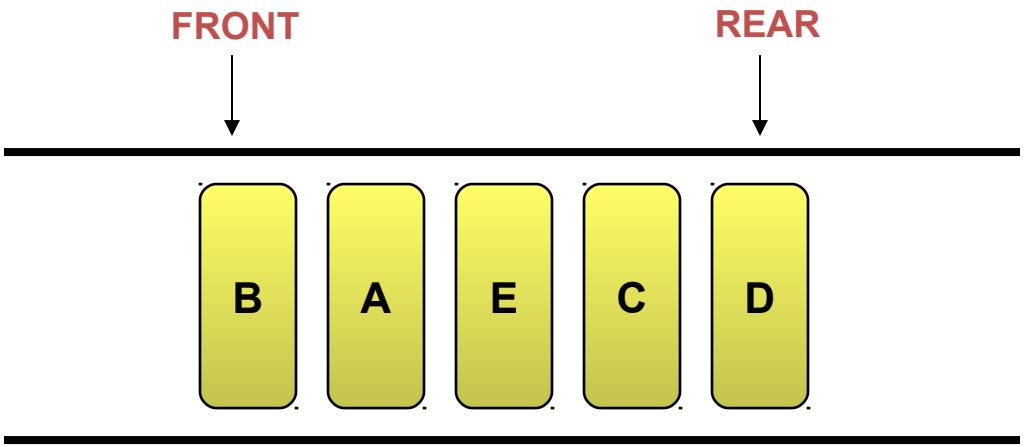


- A queue is also known as a _____ list.

- Answer:
  - FIFO

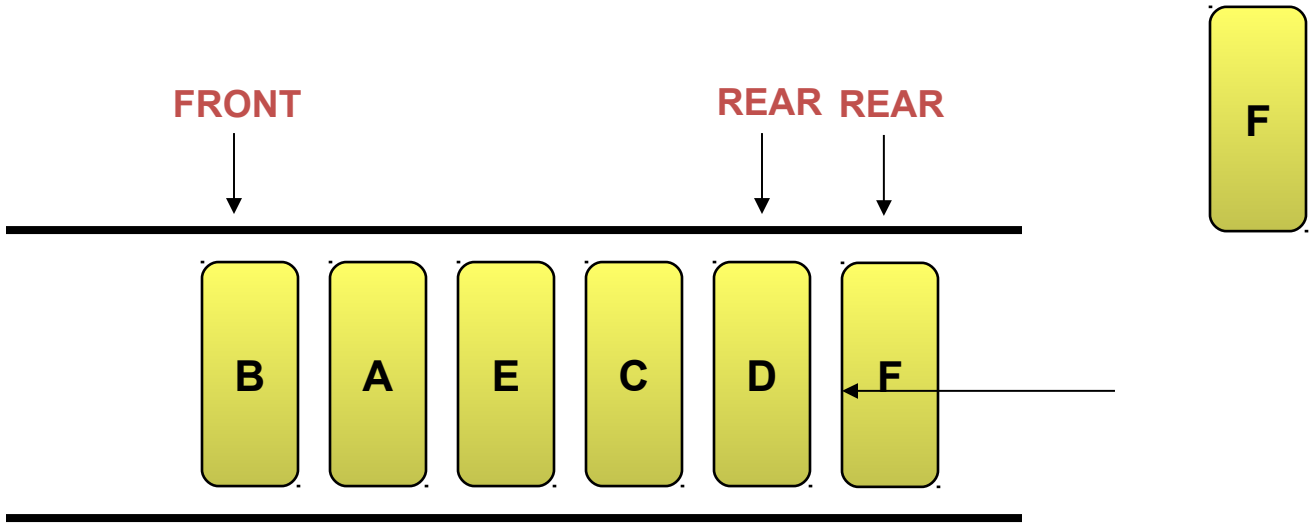
Various operations implemented on a queue are:

- Insert
- Delete



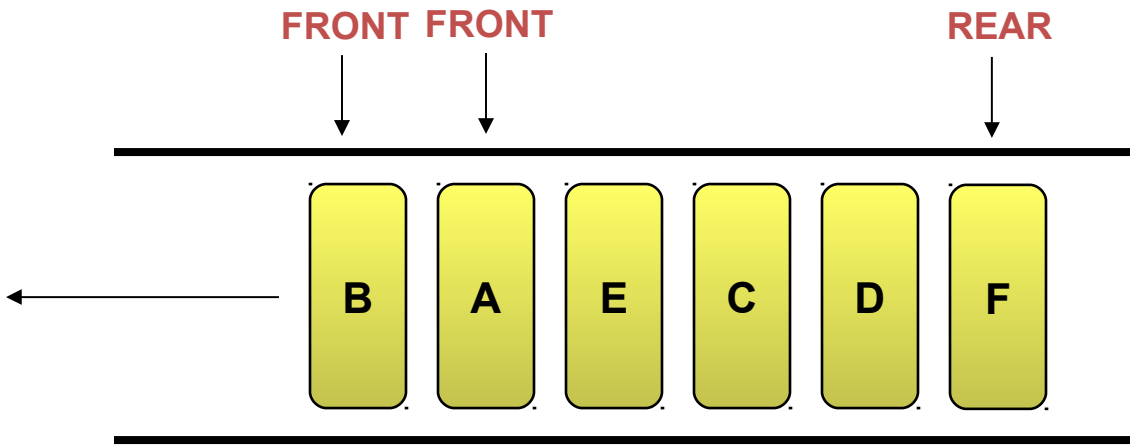
Insert: It refers to the addition of an item in the queue.

- Suppose you want to add an item F in the following queue.
- Since the items are inserted at the rear end, therefore, F is inserted after D.
- Now F becomes the rear end.



Delete: It refers to the deletion of an item from the queue.

- Since the items are deleted from the front end, therefore, item B is removed from the queue.
- Now A becomes the front end of the queue.



- Queues are the data structures in which data can be added at one end called _____ and deleted from the other end called _____.

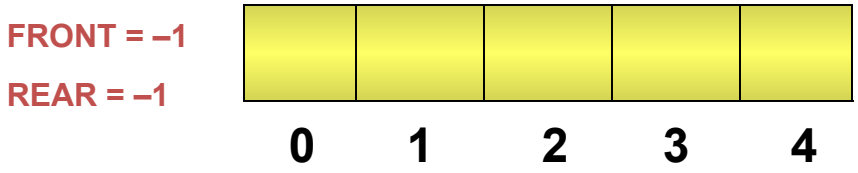
- Answer:
  - rear, front

- Problem Statement:

  Consider a scenario of a bank. When the customer visits the counter, a request entry is made and the customer is given a request number. After receiving request numbers, a customer has to wait for some time. The customer requests needs to be queued into the system and processed on the basis of their arrival. You need to implement an appropriate data storage mechanism to store these requests in the system.

Implementing a Queue Using an Array (Contd.)

- To keep track of the rear and front positions, you need to declare two integer variables, REAR and FRONT.
- If the queue is empty, REAR and FRONT are set to –1.
- How can you solve this problem?
  - You can solve this problem by implementing a queue.
- Let us implement a queue using an array that stores these request numbers in the order of their arrival.



- To insert a request number, you need to perform the following steps:
  - Increment the value of REAR by 1.
  - Insert the element at index position REAR in the array.

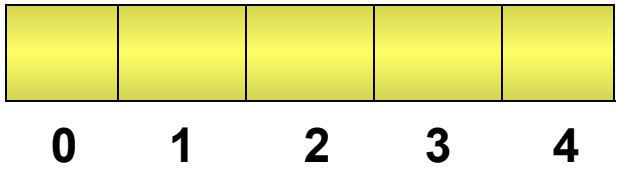
Implementing a Queue Using an Array (Contd.)

- Let us now insert request numbers in the following queue.

```
1. IF(FRONT==-1 )
     1.1 FRONT = 0
2. REAR = REAR+1
3. QUEUE[REAR] = ITEM
```

FRONT = –1

REAR = –1



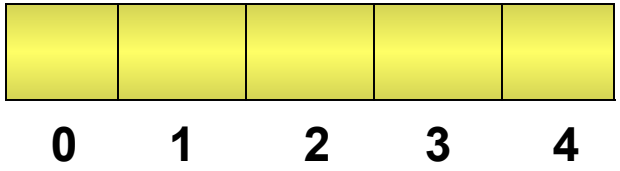
Implementing a Queue Using an Array (Contd.)

Request number generated **3**

1. IF(FRONT==-1)
   1.1 FRONT = 0
2. REAR = REAR+1
3. QUEUE[REAR] = ITEM

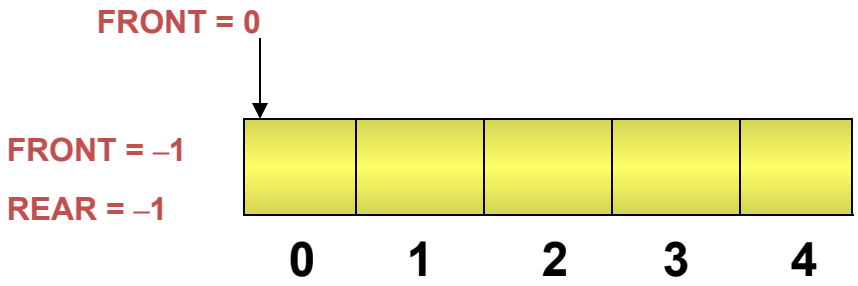
FRONT = –1

REAR = –1



Implementing a Queue Using an Array (Contd.)

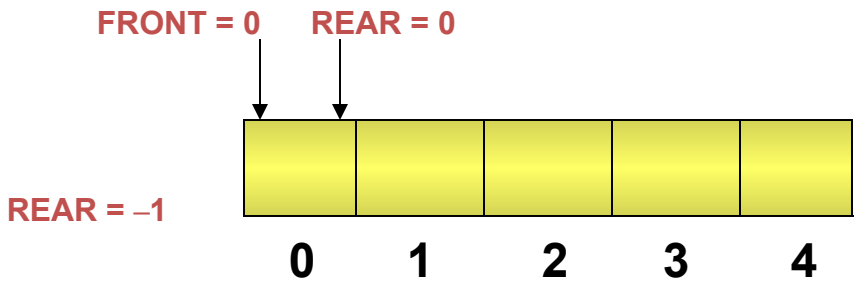
Request number generated **3**



1. IF(FRONT==-1)
   1.1 FRONT = 0
2. REAR = REAR+1
3. QUEUE[REAR] = ITEM

Implementing a Queue Using an Array (Contd.)

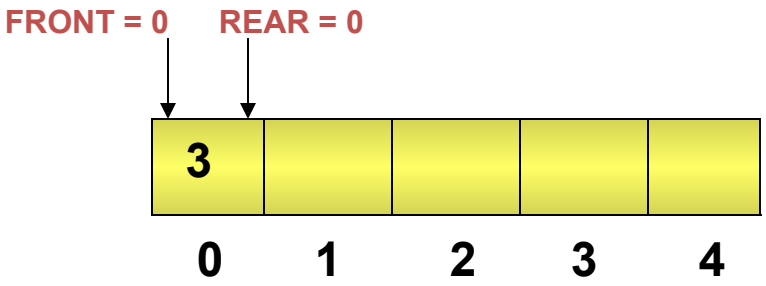
Request number generated **3**



1. IF(FRONT==-1)
   1.1 FRONT = 0
2. REAR = REAR+1
3. QUEUE[REAR] = ITEM

Implementing a Queue Using an Array (Contd.)

Request number generated **3**

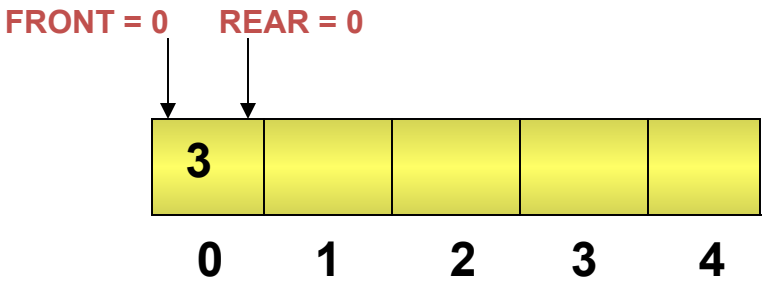


1. IF(FRONT==-1)
   1.1 FRONT = 0
2. REAR = REAR+1
3. QUEUE[REAR] = ITEM

Insertion complete

Implementing a Queue Using an Array (Contd.)

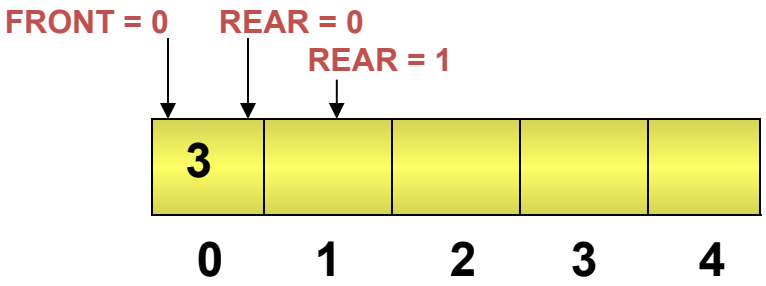
Request number generated **5**



1. IF(FRONT==-1)
   1.1 FRONT = 0
2. REAR = REAR+1
3. QUEUE[REAR] = ITEM

Implementing a Queue Using an Array (Contd.)

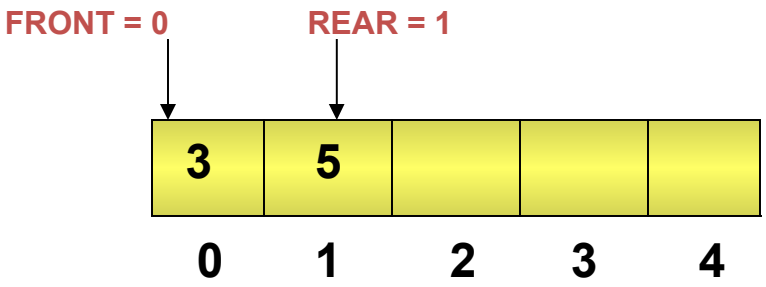
Request number generated **5**



1. IF(FRONT==-1)
   1.1 FRONT = 0
2. REAR = REAR+1
3. QUEUE[REAR] = ITEM

Implementing a Queue Using an Array (Contd.)

Request number generated **5**

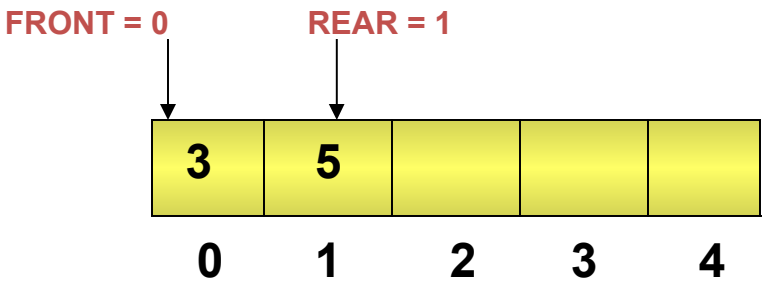


Insertion complete

1. IF(FRONT==-1)
   1.1 FRONT = 0
2. REAR = REAR+1
3. QUEUE[REAR] = ITEM

Implementing a Queue Using an Array (Contd.)

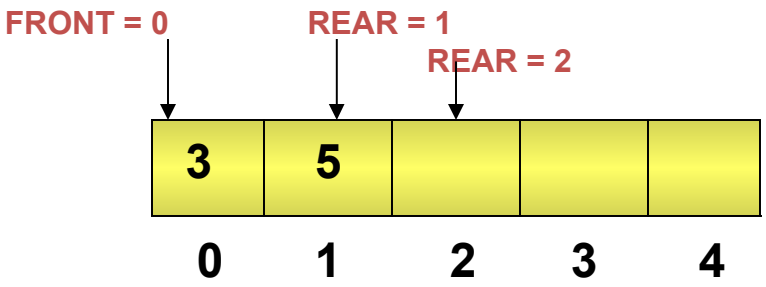
Request number generated **7**



1. IF(FRONT==-1)
   1.1 FRONT = 0
2. REAR = REAR+1
3. QUEUE[REAR] = ITEM

Implementing a Queue Using an Array (Contd.)

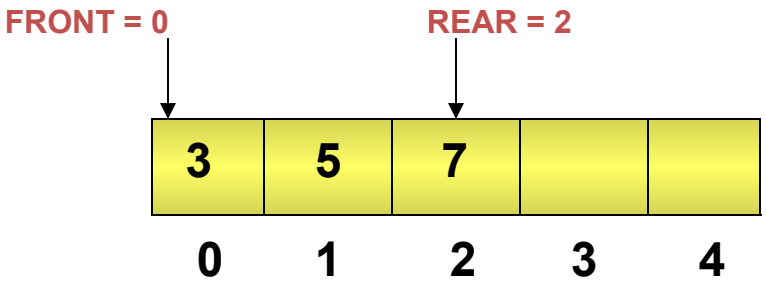
Request number generated 7



1. IF(FRONT==-1)
   1.1 FRONT = 0
2. REAR = REAR+1
3. QUEUE[REAR] = ITEM

Implementing a Queue Using an Array (Contd.)

Request number generated **7**

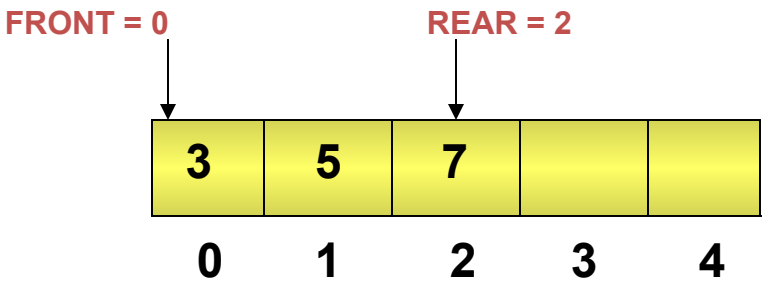


Insertion complete

1. IF(FRONT==-1)
   1.1 FRONT = 0
2. REAR = REAR+1
3. QUEUE[REAR] = ITEM

Implementing a Queue Using an Array (Contd.)

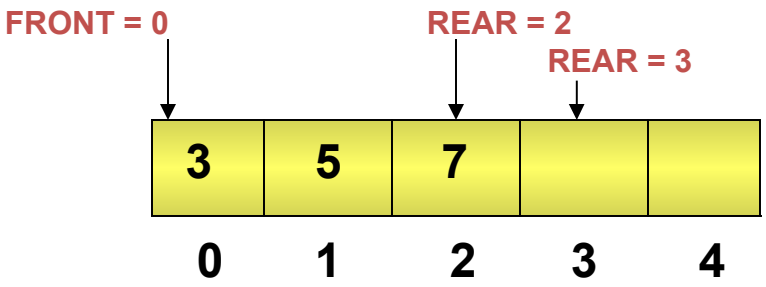
Request number generated **10**



1. IF(FRONT==-1)
   1.1 FRONT = 0
2. REAR = REAR+1
3. QUEUE[REAR] = ITEM

Implementing a Queue Using an Array (Contd.)

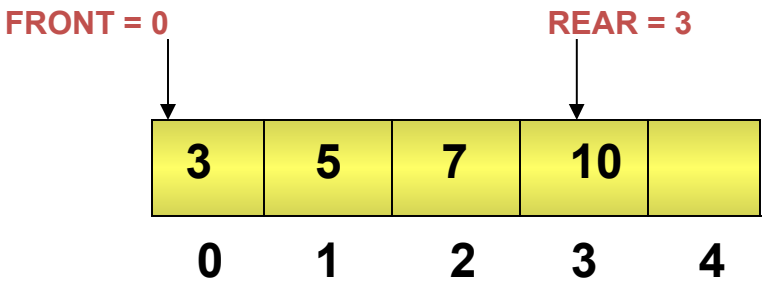
Request number generated **10**



1. IF(FRONT==-1)
   1.1 FRONT = 0
2. REAR = REAR+1
3. QUEUE[REAR] = ITEM

Implementing a Queue Using an Array (Contd.)

Request number generated **10**

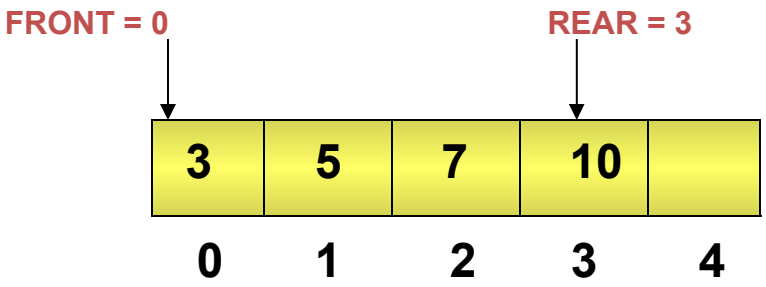


Insertion complete

1. IF(FRONT==-1)
   1.1 FRONT = 0
2. REAR = REAR+1
3. QUEUE[REAR] = ITEM

Implementing a Queue Using an Array (Contd.)

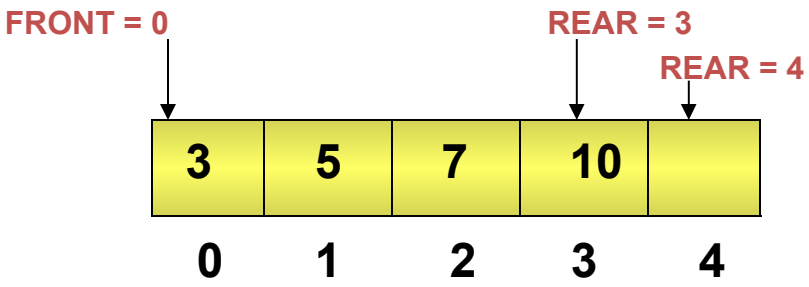
Request number generated **15**



1. IF(FRONT==-1)
   1.1 FRONT = 0
2. REAR = REAR+1
3. QUEUE[REAR] = ITEM

Implementing a Queue Using an Array (Contd.)

Request number generated **15**

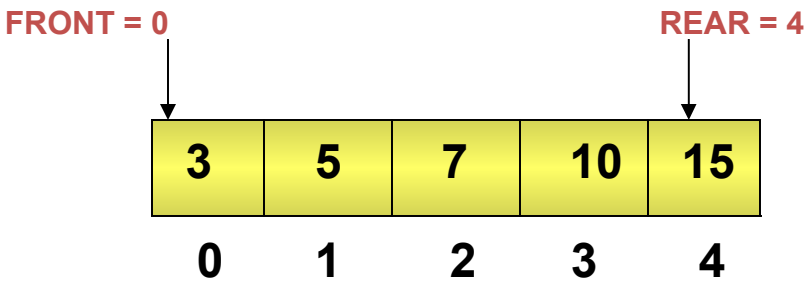


1. IF(FRONT==-1)
   1.1 FRONT = 0
2. REAR = REAR+1
3. QUEUE[REAR] = ITEM

Implementing a Queue Using an Array (Contd.)

Request number generated **15**

1. IF(FRONT==-1)
   1.1 FRONT = 0
2. REAR = REAR+1
3. QUEUE[REAR] = ITEM



Insertion complete

Algorithm to insert data in a Queue

FRONT = -1, REAR = -1, TO REPRESENT QUEUE IS EMPTY

Algorithm INSERT(QUEUE[N],FRONT,REAR,ITEM)

{

//QUEUE is an array of size N ,ITEM is element to be inserted.

1. if (REAR == N-1)

1.1 Print "OVERFLOW"

else

1.1 if (FRONT == -1)

1.1.1 FRONT = 0

1.2 REAR = REAR+1

1.3 QUEUE[REAR] = ITEM

}

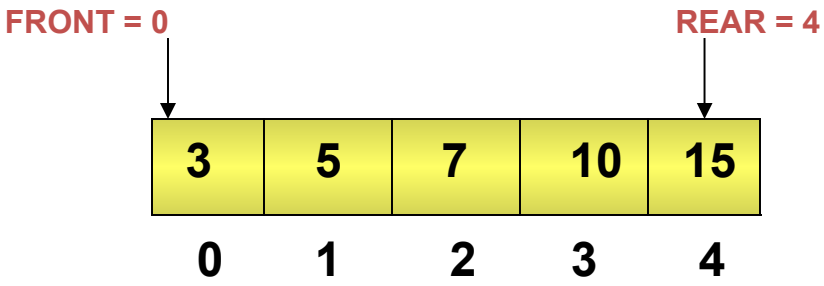
- The requests stored in the queue are served on a first-come-first-served basis.

- As the requests are being served, the corresponding request numbers needs to be deleted from the queue.

Implementing a Queue Using an Array (Contd.)

- Let us see how requests are deleted from the queue once they get processed.

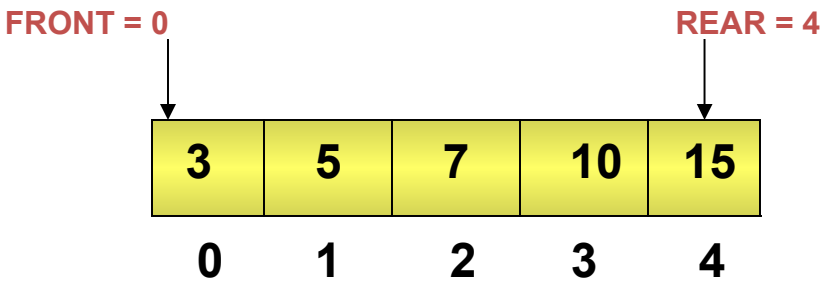
```
1.If( FRONT == – 1 )
    1.1 print “QUEUE EMPTY”
2. else
    2.1  ITEM = QUEUE[FRONT]
    2.2  FRONT = FRONT +1
```



Implementing a Queue Using an Array (Contd.)

One request processed

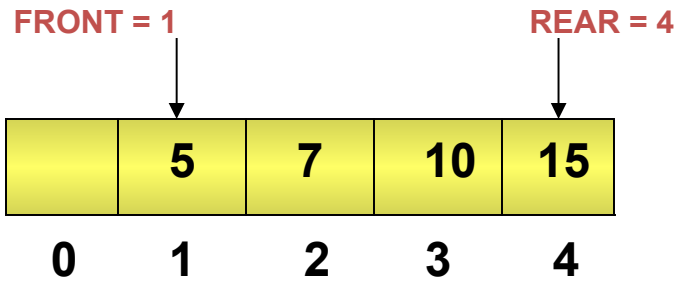
```
1.If( FRONT == – 1 )
    1.1 print “QUEUE EMPTY”
2. else
    2.1  ITEM = QUEUE[FRONT]
    2.2  FRONT = FRONT +1
```



Implementing a Queue Using an Array (Contd.)

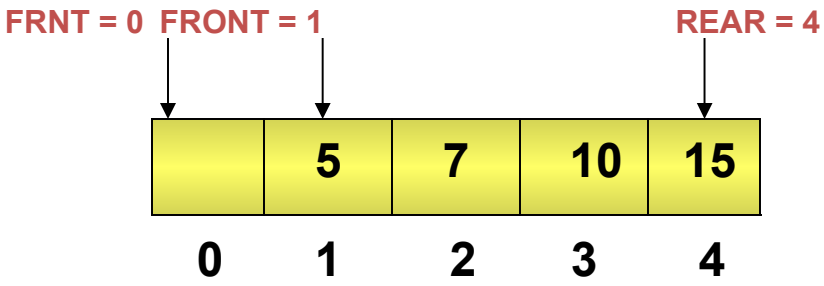
One request processed

```
1.If( FRONT == – 1 )
      1.1 print “QUEUE EMPTY”
2. else
      2.1  ITEM = QUEUE[FRONT]
      2.2  FRONT = FRONT +1
```



Implementing a Queue Using an Array (Contd.)

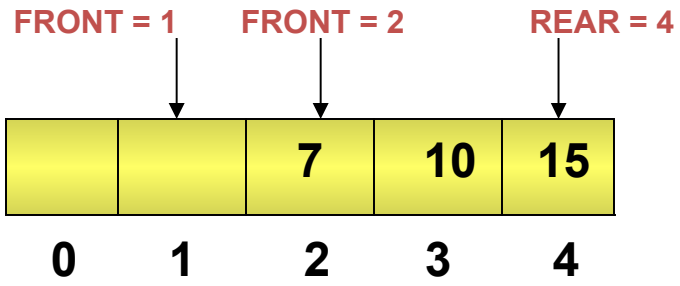
```
1.If( FRONT == – 1 )
      1.1 print “QUEUE EMPTY”
2. else
      2.1  ITEM = QUEUE[FRONT]
      2.2  FRONT = FRONT +1
```



Delete operation complete

Implementing a Queue Using an Array (Contd.)

```
1.If( FRONT == – 1 )
      1.1 print “QUEUE EMPTY”
2. else
      2.1  ITEM = QUEUE[FRONT]
      2.2  FRONT = FRONT +1
```



Delete operation complete

- Write an algorithm to delete an element from a queue implemented through array.

Algorithm DELETE(QUEUE[N],ITEM,FRONT,REAR)

{

1. if ((FRONT == – 1) || (FRONT == REAR+1))

1.1 Print “QUEUE EMPTY”

2. else

2.1 ITEM = QUEUE[FRONT]

2.2 FRONT = FRONT +1

}

Applications of Queues

Queues offer a lot of practical applications, such as:

- Printer Spooling
- CPU Scheduling
- Mail Service
- Keyboard Buffering
- Elevator

PRINTER SPOOLING

- A printer may receive multiple print requests in a short span of time.
- The rate at which these requests are received is much faster than the rate at which they are processed.
- Therefore, a temporary storage mechanism is required to store these requests in the order of their arrival.
- A queue is the best choice in this case, which stores the print requests in such a manner so that they are processed on a first-come-first-served basis.

CPU Scheduling

- A CPU can process one request at a time.
- The rate at which the CPU receives requests is usually much greater than the rate at which the CPU processes the requests.
- Therefore, the requests are temporarily stored in a queue in the order of their arrival.
- Whenever CPU becomes free, it obtains the requests from the queue.
- Once a request is processed, its reference is deleted from the queue.
- The CPU then obtains the next request in sequence and the process continues.
- In a time sharing system, CPU is allocated to each request for a fixed time period.

CPU Scheduling (Contd.)

- All these requests are temporarily stored in a queue.
- CPU processes each request one by one for a fixed time period.
- If the request is processed within that time period, its reference is deleted from the queue.
- If the request is not processed within that specified time period, the request is shifted to the end of the queue.
- CPU then processes the next request in the queue and the process continues.

Mail Service

- In various organizations, many transactions are conducted through mails.
- If the mail server goes down, and someone sends you a mail, the mail is bounced back to the sender.
- To avoid any such situation, many organizations implement a mail backup service.
- Whenever there is some problem with the mail server because of which the messages are not delivered, the mail is routed to the mail's backup server.
- The backup server stores the mails temporarily in a queue.
- Whenever the mail server is up, all the mails are transferred to the recipient in the order in which they arrived.

Keyboard Buffering

- Queues are used for storing the keystrokes as you type through the keyboard.
- Sometimes the data, which you type through the keyboard is not immediately displayed on the screen.
- This is because during that time, the processor might be busy doing some other task.
- In this situation, the data is temporarily stored in a queue, till the processor reads it.
- Once the processor is free, all the keystrokes are read in the sequence of their arrival and displayed on the screen.

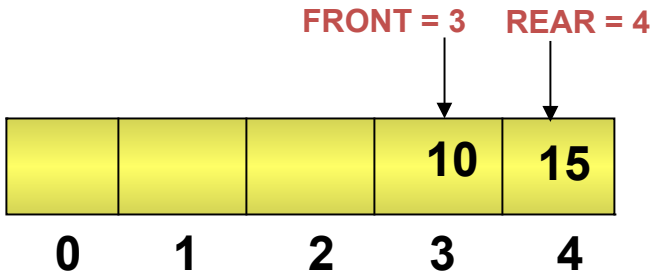
Elevator

- An elevator makes use of a queue to store the requests placed by users.
- Suppose the elevator is currently on the first floor. A user on the ground floor presses the elevator button to request for the elevator. At almost the same time a user on the second floor also presses the elevator button.
- In that case, the elevator would go to the floor on which the button was pressed earlier, that is, the requests will be processed on a FCFS basis.
- However, if one of the users had been on the first floor and the other had been on the ninth floor, then irrespective of who pressed the button first, the elevator would go to the ground floor first because the distance to the ground floor is much less than the distance to the ninth floor. In this case, some sort of a priority queue would be required.

Implementing a Queue Using an Array (Contd.)

- To implement an insert or delete operation, you need to increment the values of REAR or FRONT by one, respectively.
- However, these values are never decremented.

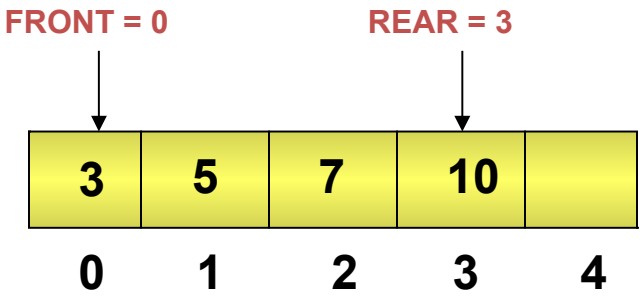
- As you delete elements from the queue, the queue moves down the array.
- The disadvantage of this approach is that the storage space in the beginning is discarded and never used again.
- Consider the following queue.



- REAR is at the last index position.
- Therefore, you cannot insert elements in this queue, even though there is space for them.
- This means that all the initial vacant positions go waste.

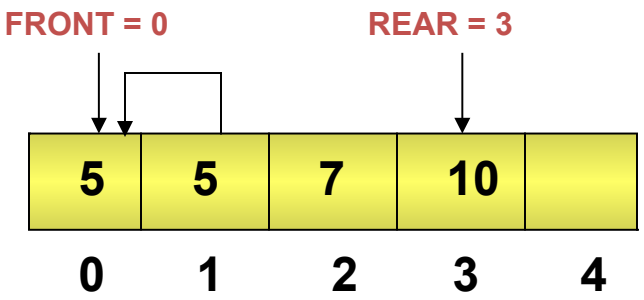
Implementing a Queue Using an Array (Contd.)

- To maintain FRONT at zero index position, every delete operation would require you to shift all the succeeding elements in the array one position left.
- Let us implement a delete operation on the following queue.
- How can you solve this problem?
  - One way to solve this problem is to keep FRONT always at the zero index position.
  - Refer to the following queue.



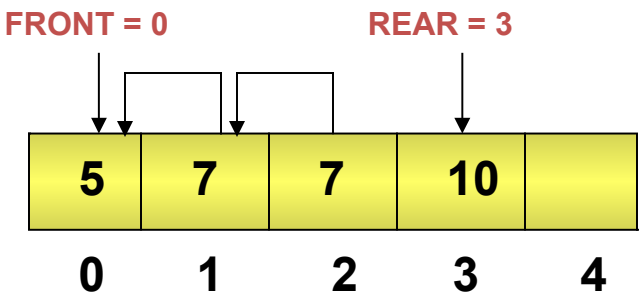
Implementing a Queue Using an Array (Contd.)

- To maintain FRONT at zero index position, every delete operation would require you to shift all the succeeding elements in the array one position left.
- Let us implement a delete operation on the following queue.



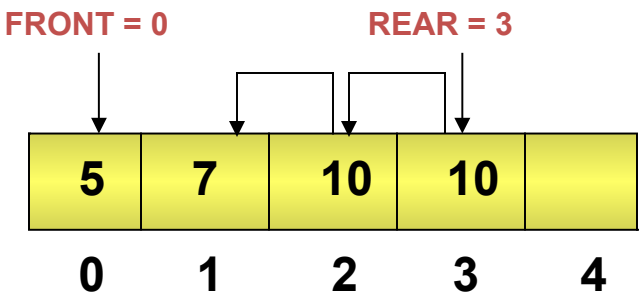
Implementing a Queue Using an Array (Contd.)

- To maintain FRONT at zero index position, every delete operation would require you to shift all the succeeding elements in the array one position left.
- Let us implement a delete operation on the following queue.



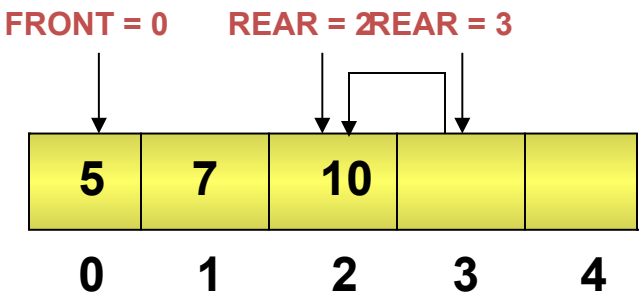
Implementing a Queue Using an Array (Contd.)

- To maintain FRONT at zero index position, every delete operation would require you to shift all the succeeding elements in the array one position left.
- Let us implement a delete operation on the following queue.



Implementing a Queue Using an Array (Contd.)

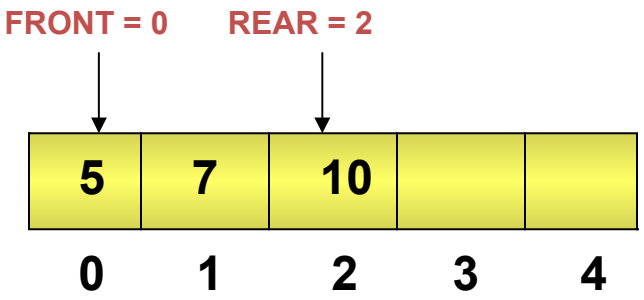
- To maintain FRONT at zero index position, every delete operation would require you to shift all the succeeding elements in the array one position left.
- Let us implement a delete operation on the following queue.



Implementing a Queue Using an Array (Contd.)

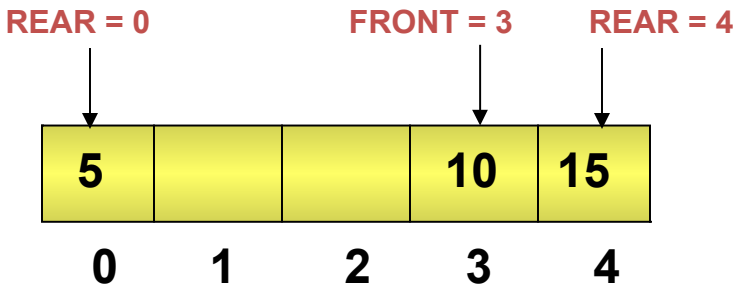
- Advantage of this approach:
  - It enables you to utilize all the empty positions in an array. Therefore, unlike the previous case, there is no wastage of space.

- Disadvantage of this approach:
  - Every delete operation requires you to shift all the succeeding elements in the queue one position left.
  - If the list is lengthy, this can be very time consuming.



Implementing a Queue Using an Array (Contd.)

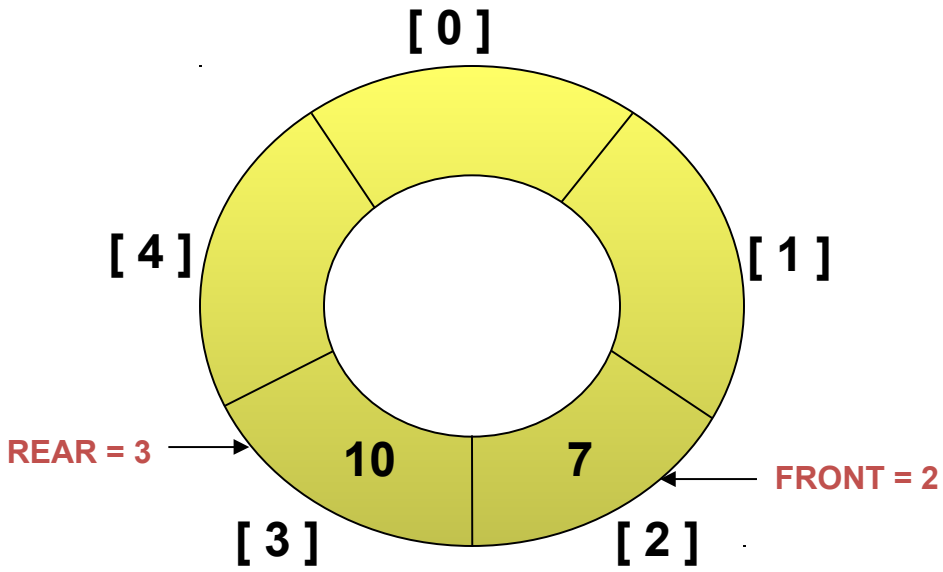
- An effective way to solve this problem is to implement the queue in the form of a circular array.
- In this approach, if REAR is at the last index position and if there is space in the beginning of an array, then you can set the value of REAR to zero and start inserting elements from the beginning.



Insert 5

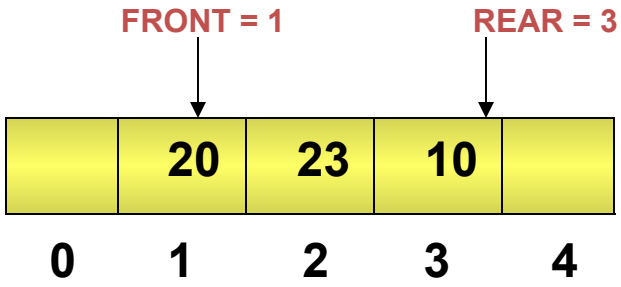
Implementing a Queue Using an Array (Contd.)

- The cells in the array are treated as if they are arranged in a ring.



Implementing a Queue Using an Array (Contd.)

Request number generated **15**

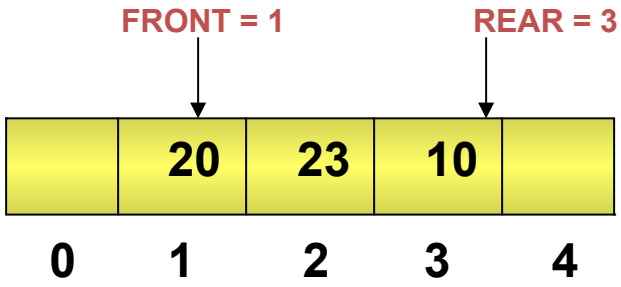


1. IF(FRONT==-1)
   - 1.1 FRONT = 0
   - 1.2 REAR == 0
   - 1.3 GO TO STEP 4
2. IF (REAR == N-1)
   - 2.1 REAR = 0
   - 2.2 GO TO STEP 4
3. REAR = REAR + 1
4. QUEUE[REAR] = ITEM

Implementing a Queue Using an Array (Contd.)

- Write an algorithm to insert values in a queue implemented as a circular array.

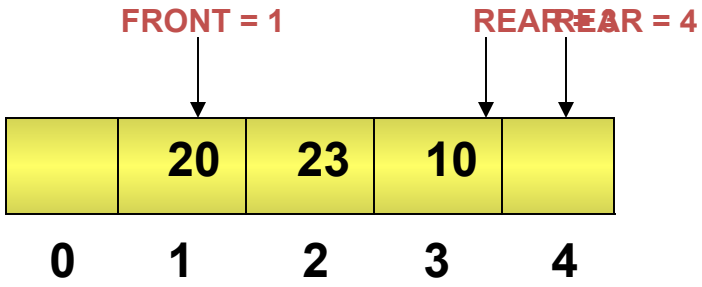
- Let us now implement a few insert operations on the following circular queue:




```
1. IF(FRONT==-1 )
        1.1 FRONT = 0
        1.2 REAR = 0
        1.3 GO TO STEP 4
2. IF ( REAR == N-1 )
        2.1 REAR = 0
        2.2 GO TO STEP 4
3. REAR = REAR + 1
4. QUEUE[REAR] = ITEM
```

Implementing a Queue Using an Array (Contd.)

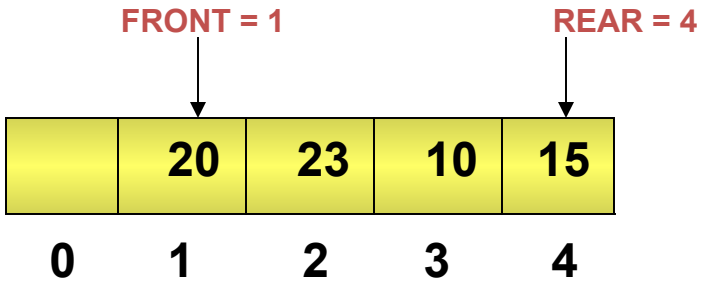
Request number generated **15**



1. IF(FRONT==-1)
   - 1.1 FRONT = 0
   - 1.2 REAR = 0
   - 1.3 GO TO STEP 4
2. IF (REAR == N-1)
   - 2.1 REAR = 0
   - 2.2 GO TO STEP 4
3. REAR = REAR + 1
4. QUEUE[REAR] = ITEM

Implementing a Queue Using an Array (Contd.)

Request number generated **15**

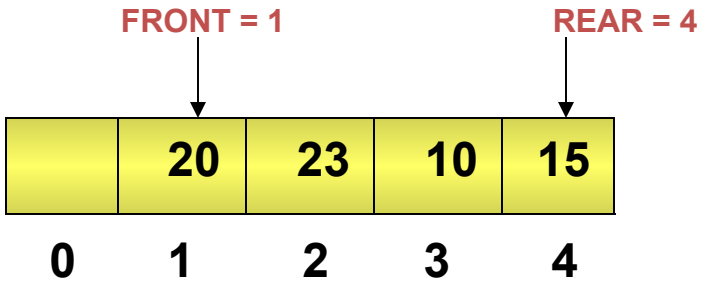


Insertion complete

```
1. IF(FRONT==-1 )
        1.1 FRONT = 0
        1.2 REAR = 0
        1.3 GO TO STEP 4
2. IF ( REAR == N-1 )
        2.1 REAR = 0
        2.2 GO TO STEP 4
3. REAR = REAR + 1
4. QUEUE[REAR] = ITEM
```

Implementing a Queue Using an Array (Contd.)

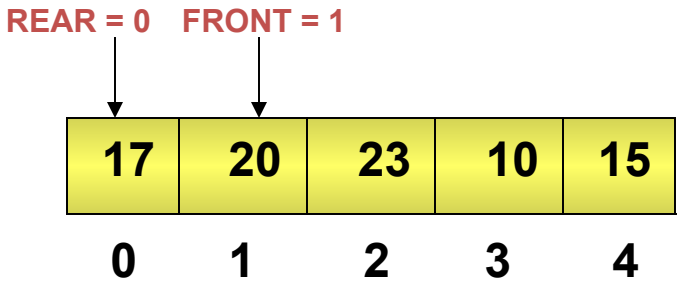
Request number generated **17**



1. IF(FRONT==-1)
   - 1.1 FRONT = 0
   - 1.2 REAR = 0
   - 1.3 GO TO STEP 4
2. IF (REAR == N-1)
   - 2.1 REAR = 0
   - 2.2 GO TO STEP 4
3. REAR = REAR + 1
4. QUEUE[REAR] = ITEM

Implementing a Queue Using an Array (Contd.)

Request number generated **17**



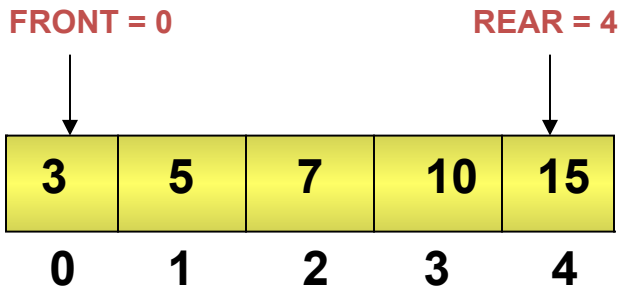
Insertion complete

1. IF(FRONT==-1)
   - 1.1 FRONT = 0
   - 1.2 REAR = 0
   - 1.3 GO TO STEP 4
2. IF (REAR == N-1)
   - 2.1 REAR = 0
   - 2.2 GO TO STEP 4
3. REAR = REAR + 1
4. QUEUE[REAR] = ITEM

Implementing a Queue Using an Array (Contd.)

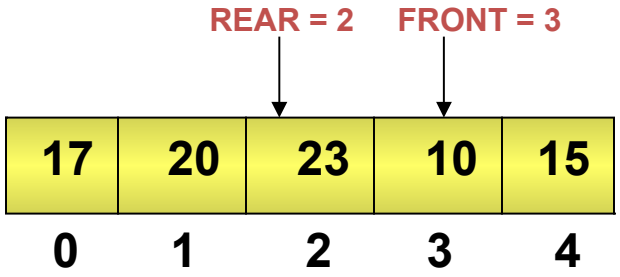
- The conditions for queue full are as follows:

If FRONT == 0 and REAR is at the last index position



OR

If FRONT == REAR + 1



Algorithm to insert data in a Circular Queue

FRONT = -1, REAR = -1, TO REPRESENT QUEUE IS EMPTY

```
Algorithm INSERT(QUEUE[N],FRONT,REAR,ITEM)
{
//QUEUE  is an array of  size N ,ITEM is element to be inserted.
1. if ((FRONT == REAR+1) || ((FRONT == 0) && (REAR == N-1)))
        1.1 DISPLAY "QUEUE OVERFLOW"
        1.2 exit
2. else
        2.1 if(FRONT == -1)
                2.1.1 FRONT = 0
                2.1.2 REAR = 0
        2.1 else if ( REAR == N-1 )
                2.1.1 REAR = 0
        2.1 else
                2.1.1 REAR = REAR + 1
        2.2 QUEUE[REAR] = ITEM
}
```

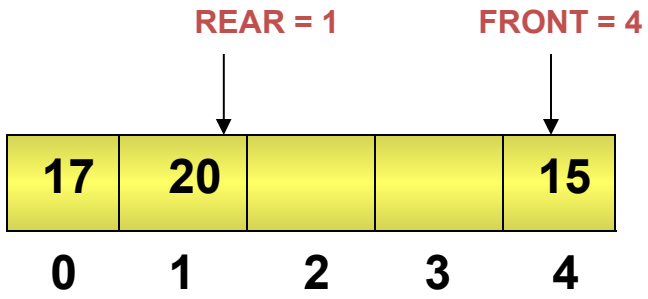
Implementing a Queue Using an Array (Contd.)

- Write an algorithm to implement delete operation in a queue implemented as a circular array.

- Now let us implement a delete operation on a queue implemented in the form of a circular array.
- To delete an element, you need to increment the value of FRONT by one. This is same as that of a linear queue.
- However, if the element to be deleted is present at the last index position, then the value FRONT is reset to zero.
- If there is only one element present in the queue, then the value of FRONT and REAR is set to –1.

Implementing a Queue Using an Array (Contd.)

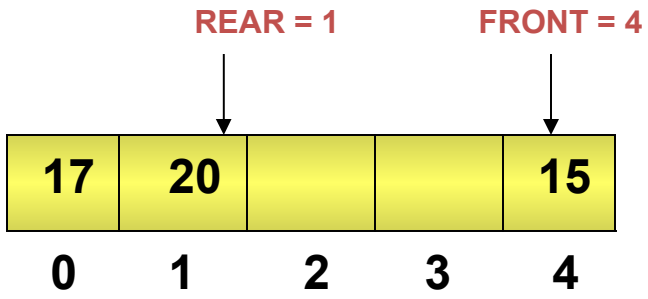
- Algorithm to delete an element from a circular queue.



1. IF(FRONT== REAR)
   - 1.1 FRONT = -1
   - 1.2 REAR = -1
   - 1.3 EXIT
2. IF (FRONT == N-1)
   - 2.1 FRONT = 0
   - 2.2 EXIT
3. FRONT = FRONT + 1

Implementing a Queue Using an Array (Contd.)

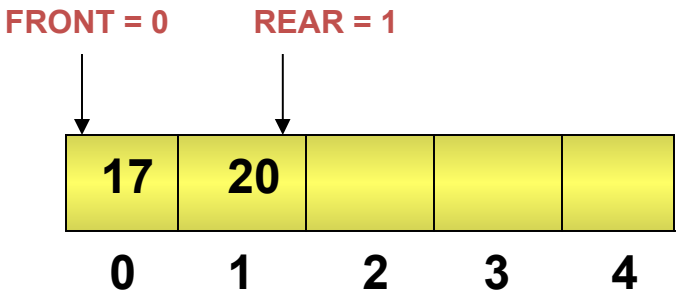
One request processed



1. IF(FRONT== REAR)
   - 1.1 FRONT = -1
   - 1.2 REAR = -1
   - 1.3 EXIT
2. IF (FRONT == N-1)
   - 2.1 FRONT = 0
   - 2.2 EXIT
3. FRONT = FRONT + 1

Implementing a Queue Using an Array (Contd.)

One request processed

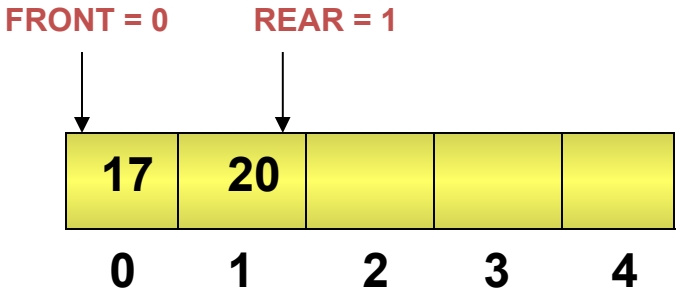


Deletion complete

```
1. IF(FRONT== REAR)
        1.1 FRONT = -1
        1.2 REAR = -1
        1.3 EXIT
2. IF ( FRONT == N-1 )
        2.1 FRONT = 0
        2.2 EXIT
3. FRONT = FRONT + 1
```

Implementing a Queue Using an Array (Contd.)

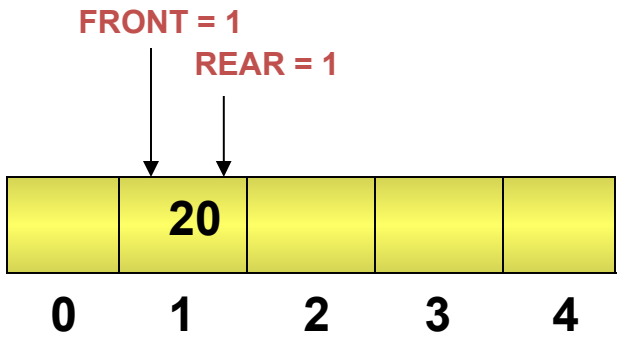
One request processed



1. IF(FRONT== REAR)
   - 1.1 FRONT = -1
   - 1.2 REAR = -1
   - 1.3 EXIT
2. IF (FRONT == N-1)
   - 2.1 FRONT = 0
   - 2.2 EXIT
3. FRONT = FRONT + 1

Implementing a Queue Using an Array (Contd.)

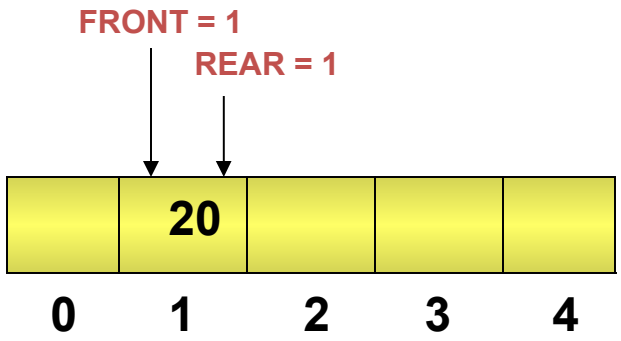
One request processed



1. IF(FRONT== REAR)
   - 1.1 FRONT = -1
   - 1.2 REAR = -1
   - 1.3 EXIT
2. IF (FRONT == N-1)
   - 2.1 FRONT = 0
   - 2.2 EXIT
3. FRONT = FRONT + 1

Implementing a Queue Using an Array (Contd.)

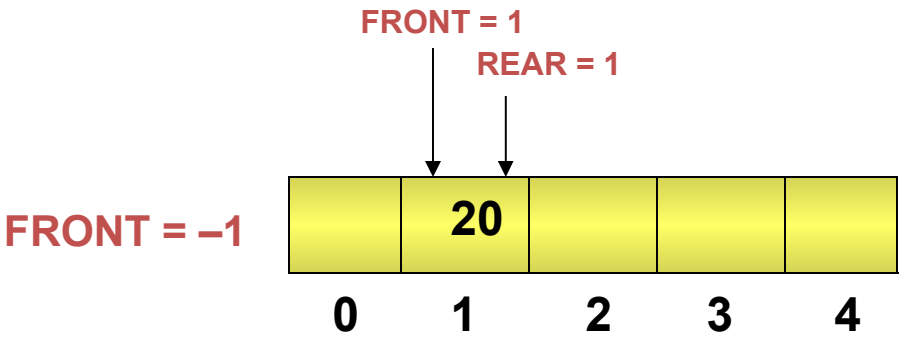
One request processed



1. IF(FRONT== REAR)
   - 1.1 FRONT = -1
   - 1.2 REAR = -1
   - 1.3 EXIT
2. IF (FRONT == N-1)
   - 2.1 FRONT = 0
   - 2.2 EXIT
3. FRONT = FRONT + 1

Implementing a Queue Using an Array (Contd.)

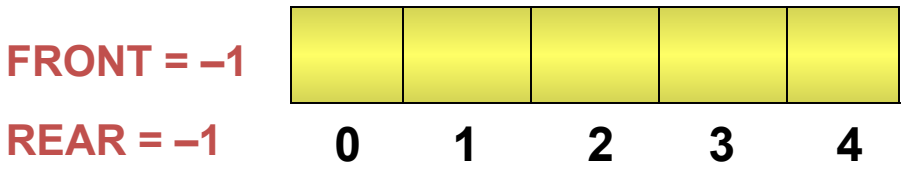
One request processed



1. IF(FRONT== REAR)
   - 1.1 FRONT = -1
   - 1.2 REAR =-1
   - 1.3 EXIT
2. IF (FRONT == N-1)
   - 2.1 FRONT = 0
   - 2.2 EXIT
3. FRONT = FRONT + 1

Implementing a Queue Using an Array (Contd.)

One request processed

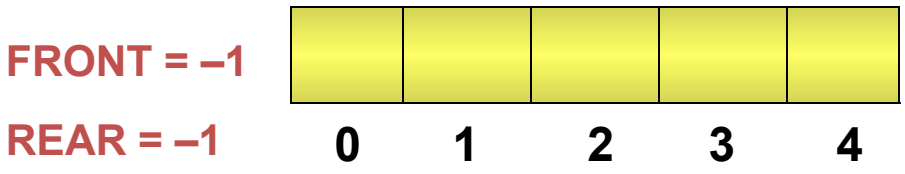


1. IF(FRONT== REAR)
   - 1.1 FRONT = -1
   - 1.2 REAR = -1
   - 1.3 EXIT
2. IF (FRONT == N-1)
   - 2.1 FRONT = 0
   - 2.2 EXIT
3. FRONT = FRONT + 1

Implementing a Queue Using an Array (Contd.)

One request processed

```
1. IF(FRONT== REAR)
        1.1 FRONT = -1
        1.2 REAR = -1
        1.3 EXIT
2. IF ( FRONT == N-1 )
        2.1 FRONT = 0
        2.2 EXIT
3. FRONT = FRONT + 1
```

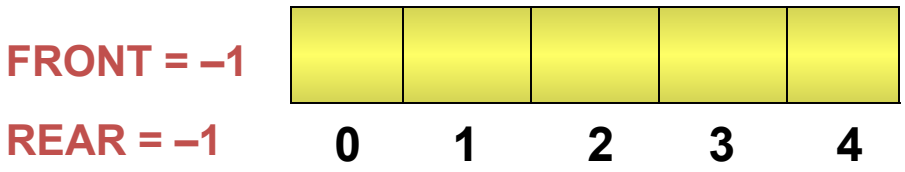


Deletion complete

Implementing a Queue Using an Array (Contd.)

The queue is now empty.

- At this stage, if you try to implement a delete operation, it will result in queue underflow.




```
1. IF(FRONT== REAR)
        1.1 FRONT = -1
        1.2 REAR = -1
        1.3 EXIT
2. IF ( FRONT == N-1 )
        2.1 FRONT = 0
        2.2 EXIT
3. FRONT = FRONT + 1
```

- Therefore, before implementing a delete operation, you first need to check whether the queue is empty or not.

Algorithm to delete data in a Circular Queue

FRONT = -1, REAR = -1, TO REPRESENT QUEUE IS EMPTY

```
Algorithm DELETE(QUEUE[N],FRONT,REAR,ITEM)
{
//QUEUE  is an array of  size N ,ITEM is element to be inserted.
1. if (FRONT == -1)
      1.1 DISPLAY "QUEUE UNDERFLOW"
2. else
      2.1 ITEM = QUEUE[FRONT]
      2.2 if(FRONT== REAR)
                  2.2.1 FRONT = -1
                  2.2.2 REAR =-1
                  2.2.3 EXIT
      2.2 else if ( FRONT == N-1 )
                  2.2.1 FRONT = 0
                  2.2.2 EXIT
      2.2 else
                  2.4.1 FRONT = FRONT + 1
}
```

Just a minute

- What is the advantage of implementing a queue in the form of a circular array, instead of a linear array structure?

- Answer:
  - If you implement a queue in the form of a linear array, you can add elements only in the successive index positions. However, when you reach the end of the queue, you cannot start inserting elements from the beginning, even if there is space for them at the beginning. You can overcome this disadvantage by implementing a queue in the form of a circular array. In this case, you can keep inserting elements till all the index positions are filled. Hence, it solves the problem of unutilized space.

Implementing a Queue Using a Linked List

- What is the disadvantage of implementing a queue as an array?
  - To implement a queue using an array, you must know the maximum number of elements in the queue in advance.
- To solve this problem, you should implement the queue in the form of a linked list.

APPLICATION OF CIRCULAR QUEUE

- Process Scheduling of using round-robin technique

DEQUEUE

- Double-ended Queue (dequeue), often abbreviated to Deque.

- A data structure that implements a queue for which elements can be added to or removed from the front (head) or back (tail).

- It is also often called a head-tail Linked List.

- This differs from the queue or First-In-First-Out List (FIFO), where elements can only be added to one end and removed from the other.

DEQUEUE

This general data class has some possible sub-types:

- **An input-restricted deque** is one where deletion can be made from both ends, but insertion can only be made at one end.

- **An output-restricted deque** is one where insertion can be made at both ends, but deletion can be made from one end only.

DEQUEUE

Various operations of Dequeu are:-

1. Insertion / enque from rear
2. Insertion / enque from front
3. Deletion / deque from front
4 Deletion / deque from rear
5. display

Algorithm to Insert / enqueue from rear

FRONT = -1, REAR = -1, TO REPRESENT DEQUEUE IS EMPTY

```
Algorithm INSERTREAR(QUEUE[N],FRONT,REAR,ITEM)
{
//QUEUE  is an array of  size N ,ITEM is element to be inserted.

1. if (REAR == N-1)
                    1.1 PRINT "OVERFLOW"
2. else
                    2.1 IF(FRONT==-1)
                                2.1.1 FRONT = 0
                    2.2. REAR = REAR+1
                    2.3 QUEUE[REAR] = ITEM
}
```

Algorithm to Insert / enqueue from front

FRONT = -1, REAR = -1, TO REPRESENT DEQUEUE IS EMPTY

```
Algorithm INSERTFRONT(QUEUE[N],FRONT,REAR,ITEM)
{
//QUEUE  is an array of  size N ,ITEM is element to be inserted.
1. if( FRONT == 0)
        1.1 Print "CAN NOT INSERT"
2. else
        2.1 if (FRONT == -1)
                2.1.1 FRONT = 0
                2.1.2 REAR = 0
        2.1 else
                2.1.1 FRONT = FRONT-1
        2.3 QUEUE[FRONT] = ITEM
}
```

Algorithm to Delete / dequeue from front

FRONT = -1, REAR = -1, TO REPRESENT DEQUEUE IS EMPTY

```
Algorithm DELETE(QUEUE[N],ITEM,FRONT,REAR)
{
 1. if (( FRONT == – 1 ) || (FRONT == REAR+1))
        1.1 Print  “QUEUE EMPTY”
2. else
        2.1  ITEM = QUEUE[FRONT]
        2.2  FRONT = FRONT +1
}
```

Algorithm to Delete / dequeue from rear

FRONT = -1, REAR = -1, TO REPRESENT DEQUEUE IS EMPTY

```
Algorithm DELETE(QUEUE[N],ITEM,FRONT,REAR)
{
 1. if (( FRONT == – 1 ) || (FRONT == REAR+1))
        1.1 Print  “QUEUE EMPTY”
2. else
        2.1  ITEM = QUEUE[REAR]
        2.2  REAR = REAR - 1
}
```

APPLICATION OF DEQUEUE

- Job scheduling algorithm.

- Deque is used in storing a web browser's history. Recently visited URLs are added to the front of the dequeue, and the URL at the back of the dequeue is removed after some specified number of insertions at the front.

PRIORITY QUEUE

- Is a collection of elements where the elements are stored according to their priority levels.
- Following rules are applied to maintain a priority queue:-
  - The elements with a higher priority is processed before any element of lower priority.
  - If there are elements with a same priority ,then the element added first in the queue would get processed.

APPLICATION OF PRIORITY QUEUE

- Job Scheduling Algorithm.

Algorithm to insert data in a priority Queue

f = -1, r = -1, TO REPRESENT QUEUE IS EMPTY

Algorithm INSERT(q[N], f, r, item(process no , priority))

{

//q is an array of size N ,item is element to be inserted.

1. if(r==N-1)

   Print “overflow"

2. if(f==-1)

   2.1 f=0 , r=0

   2.2 Enter process no and priority ,item // object of structure

   2.3 q[r]=item

   else

   2.1 r=r+1

   2.2 j=r-1

   2.3 Enter process no and priority, item // object of structure

   2.4 while(q[j] . p < item . p && j >= 0)

   2.4.1q[j+1] = q[j]

   2.4.2 j=j-1

   2.5 q[j+1] = item;

}

```
Algorithm DELETE(q[N],item,f,r)
   {
    1. if (( f == – 1 ) || (f == r+1))
                    1.1 Print  “QUEUE EMPTY”

    else
                    1.1  f = f +1
    }
```