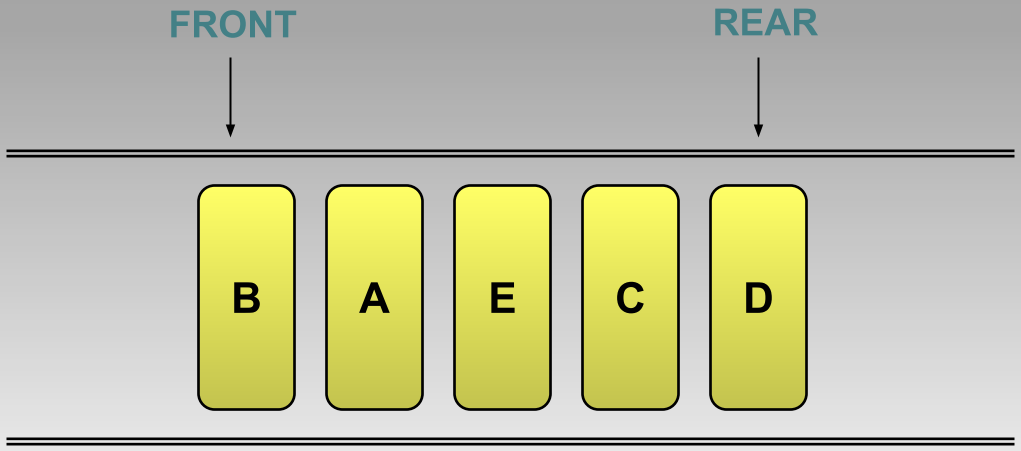
Data Structures and Algorithms

Defining Queues

- 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 much faster than the rate at which they are processed.
  - Therefore, you need to store the request somewhere until they are processed.
- 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.

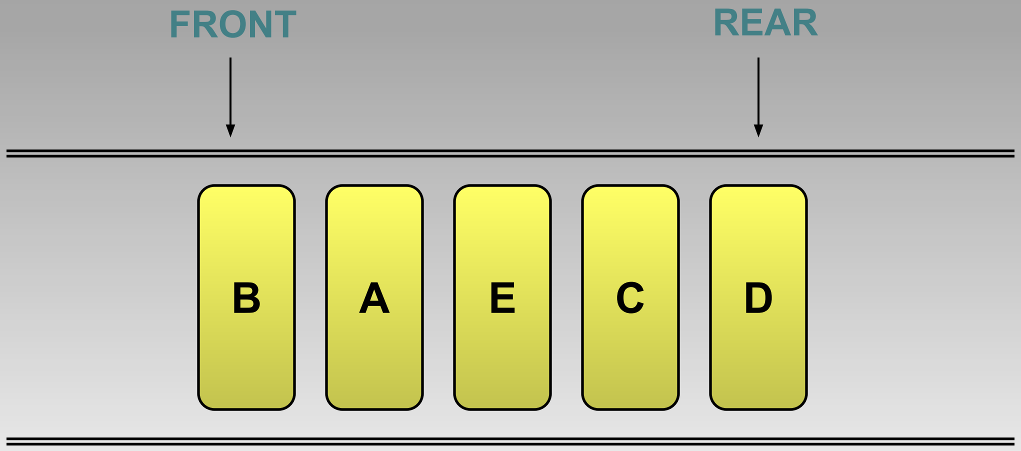
Defining Queues (Contd.)

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



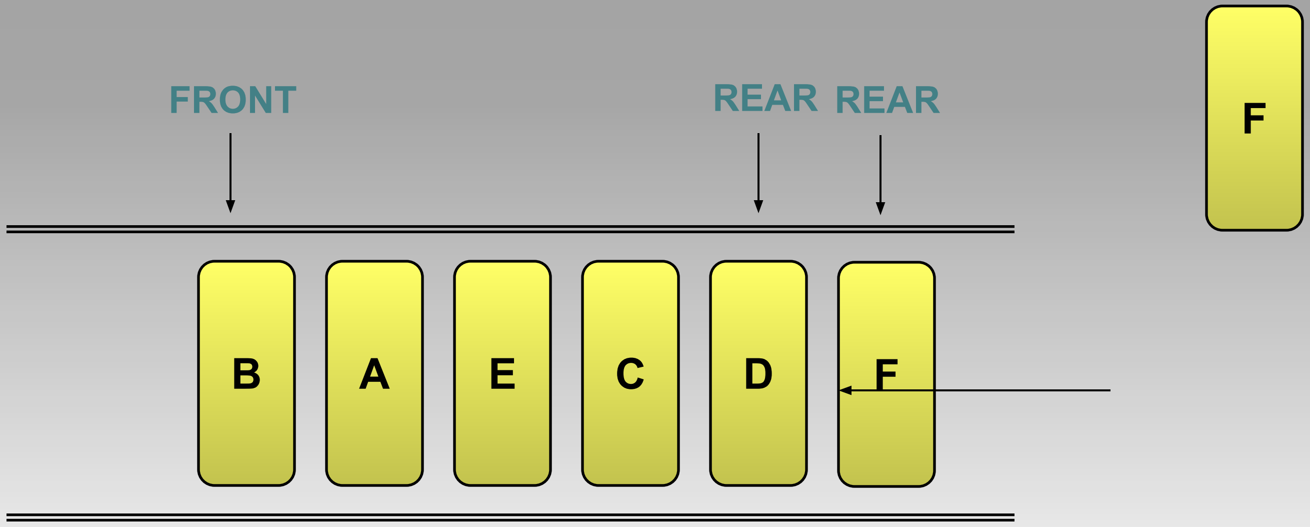
Identifying Various Operations on Queues

- Various operations implemented on a queue are:
  - Insert
  - Delete



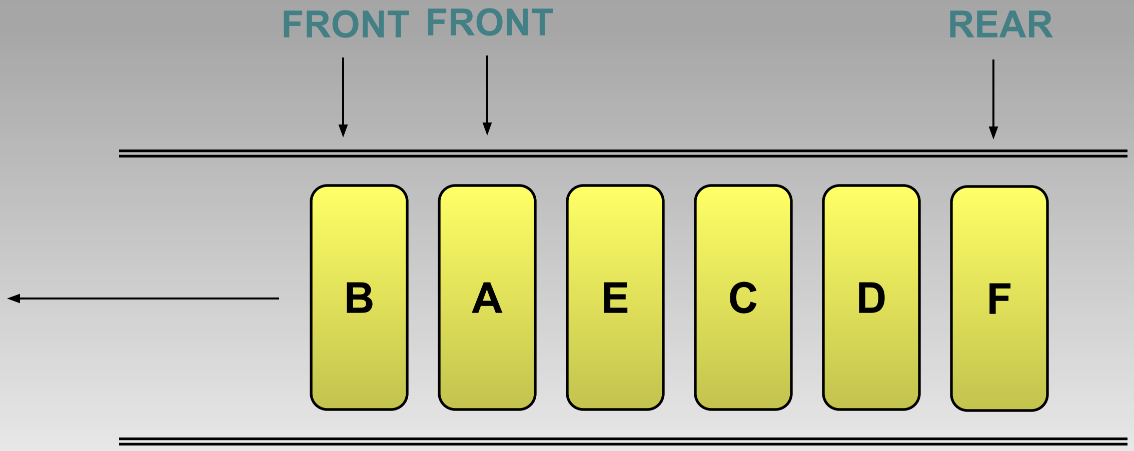
Identifying Various Operations on Queues (Contd.)

- **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.



Identifying Various Operations on Queues (Contd.)

- **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.



Just a minute

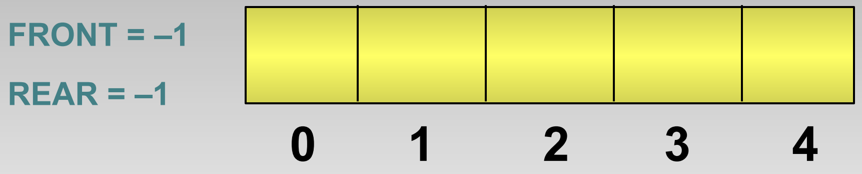
- Queues are the data structures in which data can be added at one end called _______ and deleted from the other end called __________.

Implementing a Queue Using an Array

- 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.
- 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.)

- Write an algorithm to insert values in a queue implemented through array.
- Algorithm to insert values in a queue:

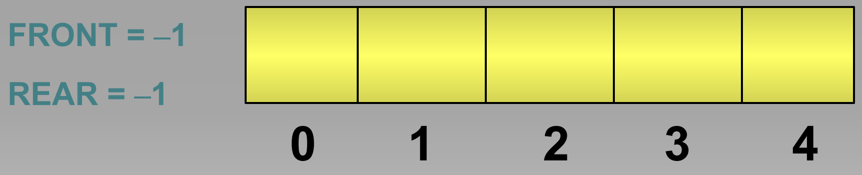
```
1. If the queue is empty:
     a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.
```

Implementing a Queue Using an Array (Contd.)

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

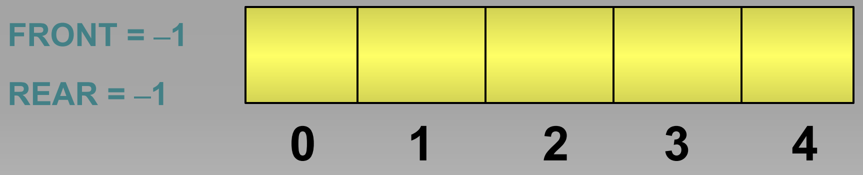
```
1.  If the queue is empty:
      a.  Set FRONT = 0.
2.  Increment REAR by 1.
3.  Store the element at index
    position REAR in the array.
```

Request number generated 3



Implementing a Queue Using an Array (Contd.)

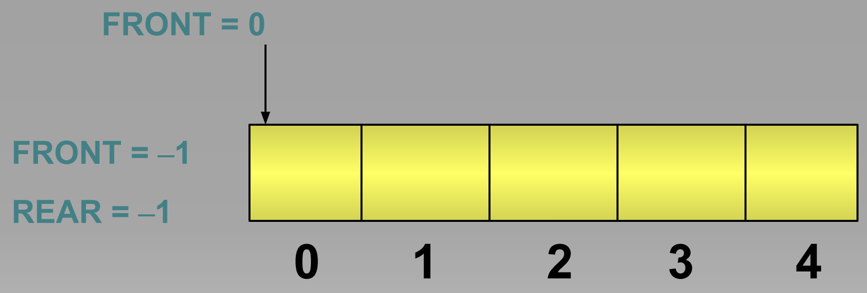
Request number generated 3



```
1.  If the queue is empty:
        a.  Set FRONT = 0.
2.  Increment REAR by 1.
3.  Store the element at index position
    REAR in the array.
```

Implementing a Queue Using an Array (Contd.)

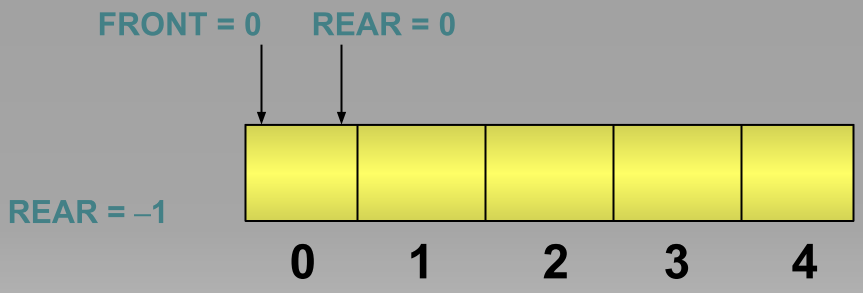
Request number generated 3



1. If the queue is empty:
   a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.

Implementing a Queue Using an Array (Contd.)

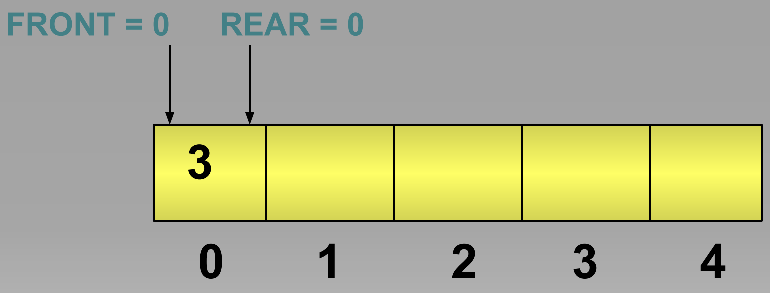
Request number generated **3**



1. If the queue is empty:
   a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.

Implementing a Queue Using an Array (Contd.)

Request number generated 3

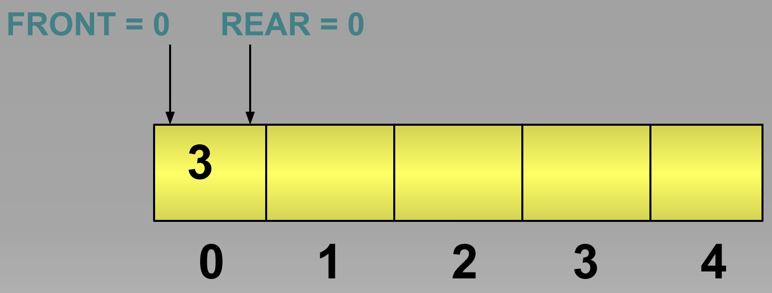


Insertion complete

1. If the queue is empty:
   a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.

Implementing a Queue Using an Array (Contd.)

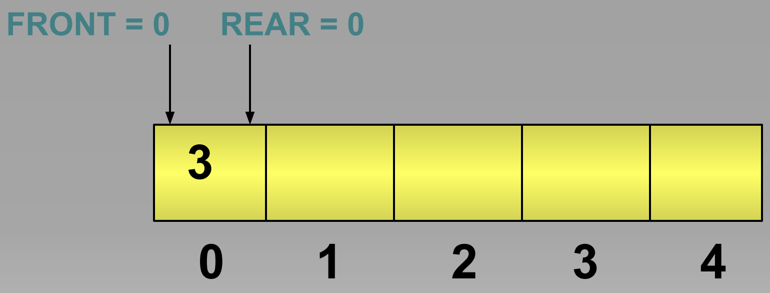
Request number generated **5**



1. If the queue is empty:
   a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.

Implementing a Queue Using an Array (Contd.)

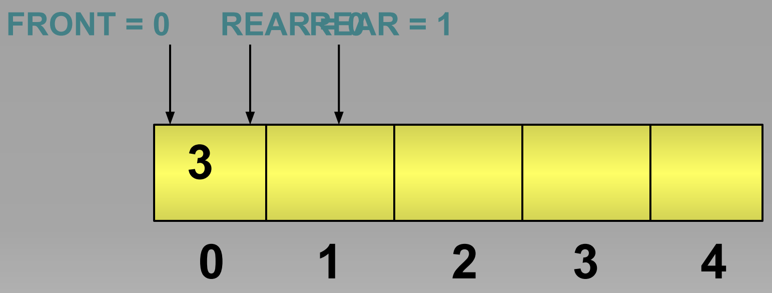
Request number generated 5



1. If the queue is empty:
   a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.

Implementing a Queue Using an Array (Contd.)

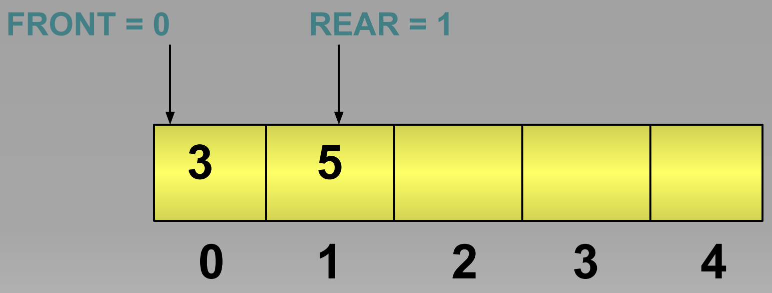
Request number generated 5



1. If the queue is empty:
   a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.

Implementing a Queue Using an Array (Contd.)

Request number generated **5**

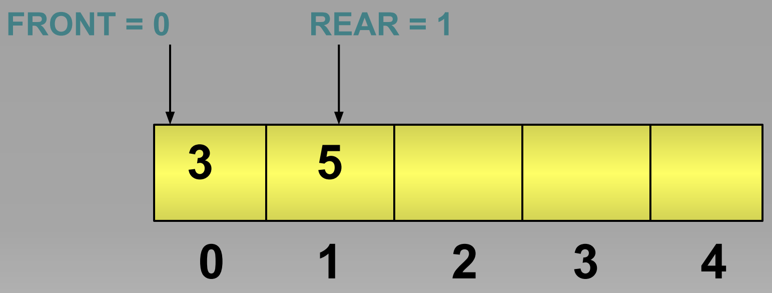


Insertion complete

1. If the queue is empty:
   a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.

Implementing a Queue Using an Array (Contd.)

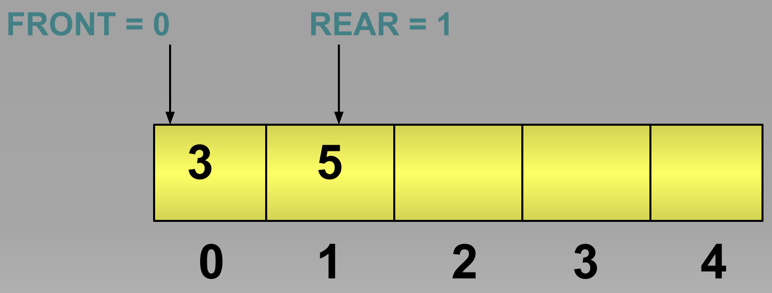
Request number generated 7



1. If the queue is empty:
   a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.

Implementing a Queue Using an Array (Contd.)

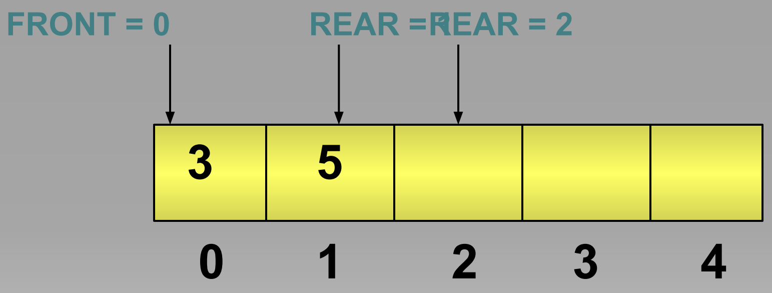
Request number generated 7



1. If the queue is empty:
   a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.

Implementing a Queue Using an Array (Contd.)

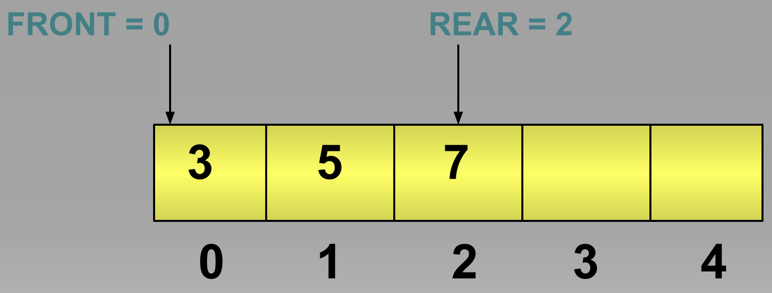
Request number generated 7



1. If the queue is empty:
   a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.

Implementing a Queue Using an Array (Contd.)

Request number generated 7

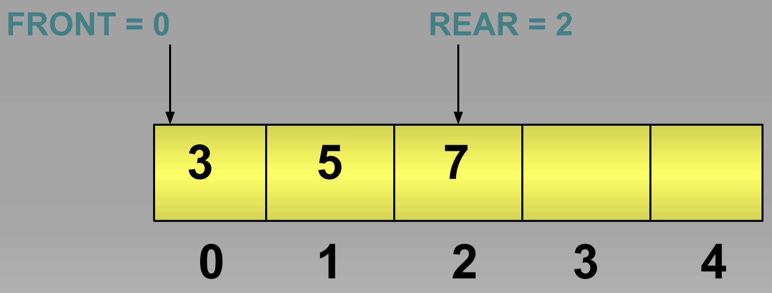


Insertion complete

1. If the queue is empty:
   a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.

Implementing a Queue Using an Array (Contd.)

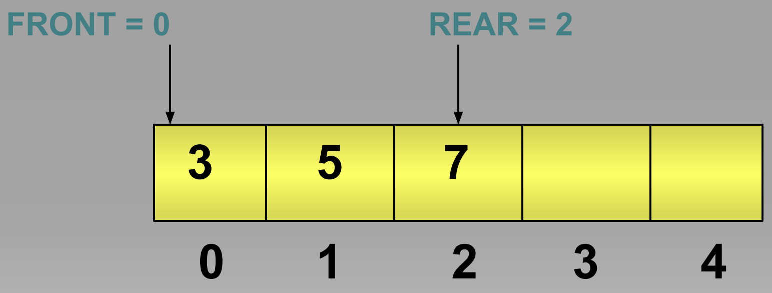
Request number generated **10**



1. If the queue is empty:
   a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.

Implementing a Queue Using an Array (Contd.)

Request number generated 10

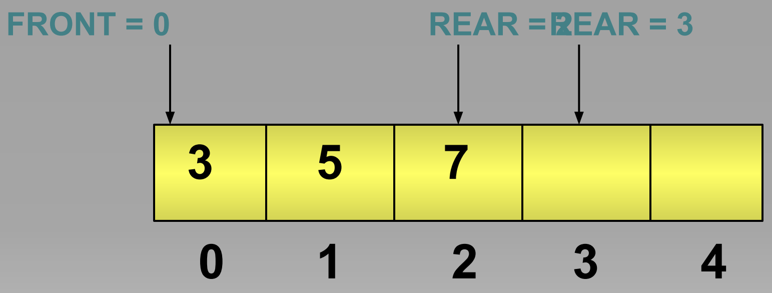


1. If the queue is empty:
   a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.

Data Structures and Algorithms

Implementing a Queue Using an Array (Contd.)

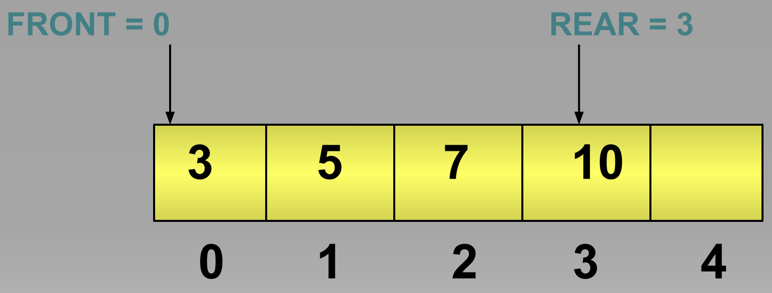
Request number generated **10**



1. If the queue is empty:
   a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.

Implementing a Queue Using an Array (Contd.)

Request number generated 10

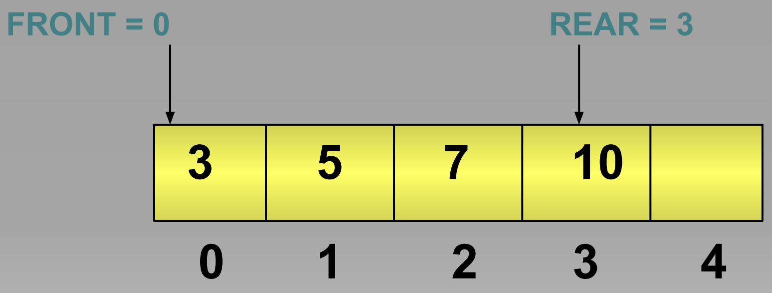


Insertion complete

1. If the queue is empty:
   a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.

Implementing a Queue Using an Array (Contd.)

Request number generated 15

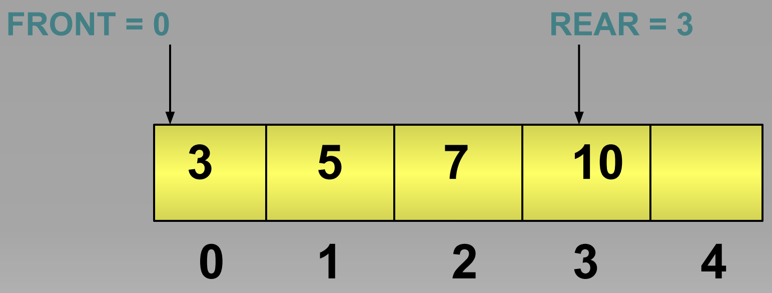


1. If the queue is empty:
   a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.

Data Structures and Algorithms

Implementing a Queue Using an Array (Contd.)

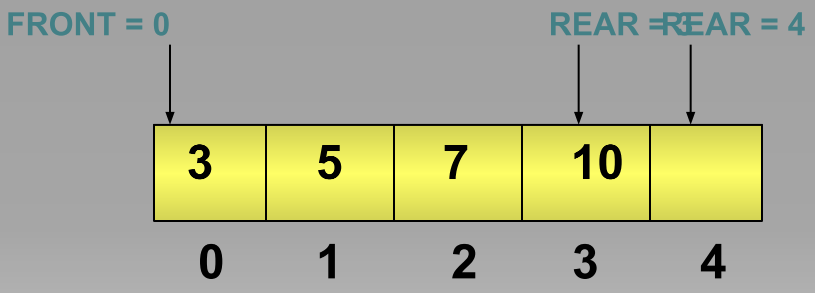
Request number generated 15



1. If the queue is empty:
   a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.

Implementing a Queue Using an Array (Contd.)

Request number generated **15**

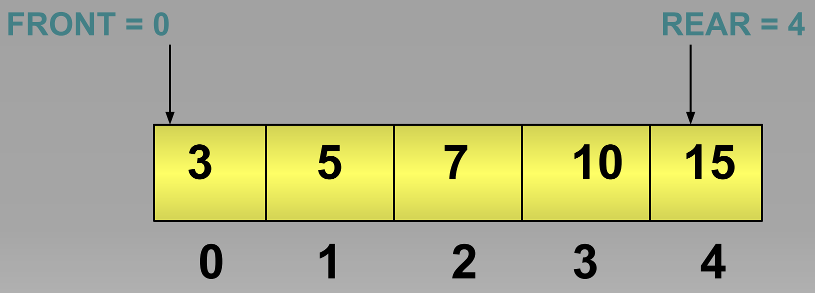


1. If the queue is empty:
   a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.

Data Structures and Algorithms

Implementing a Queue Using an Array (Contd.)

Request number generated 15



Insertion complete

1. If the queue is empty:
   a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.

Implementing a Queue Using an Array (Contd.)

- 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.)

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

```
1.  Retrieve the element at index FRONT.

2.  Increment FRONT by 1.
```

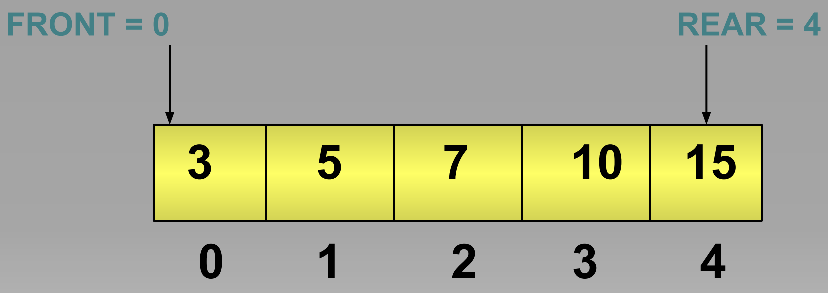
Data Structures and Algorithms

Implementing a Queue Using an Array (Contd.)

One request processed

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

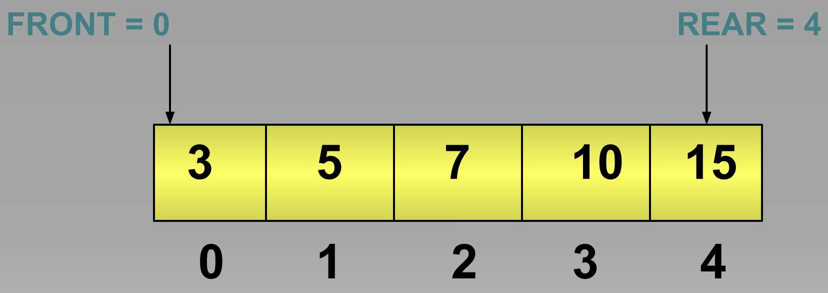
1. Retrieve the element at index FRONT.
2. Increment FRONT by 1.



Implementing a Queue Using an Array (Contd.)

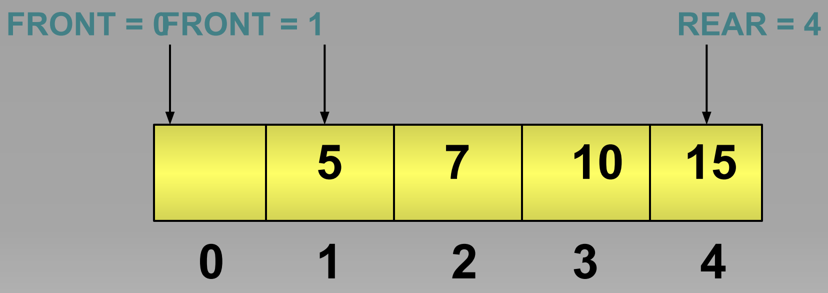
One request processed

1. Retrieve the element at index FRONT.
2. Increment FRONT by 1.



Implementing a Queue Using an Array (Contd.)

1. Retrieve the element at index FRONT.
2. Increment FRONT by 1.



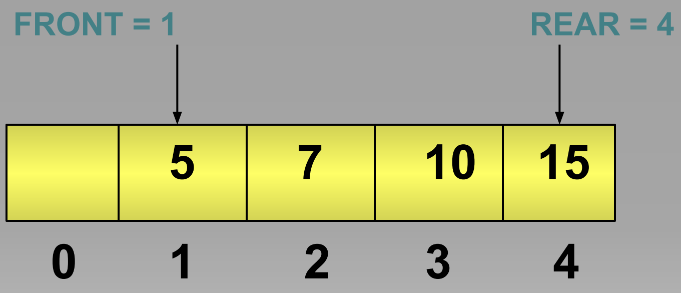
Delete operation complete

Data Structures and Algorithms

Implementing a Queue Using an Array (Contd.)

One request processed

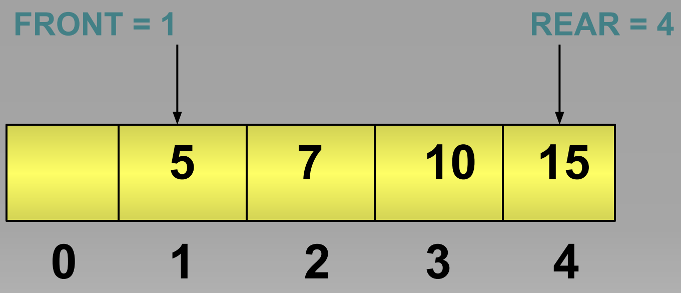
1. Retrieve the element at index FRONT.
2. Increment FRONT by 1.



Implementing a Queue Using an Array (Contd.)

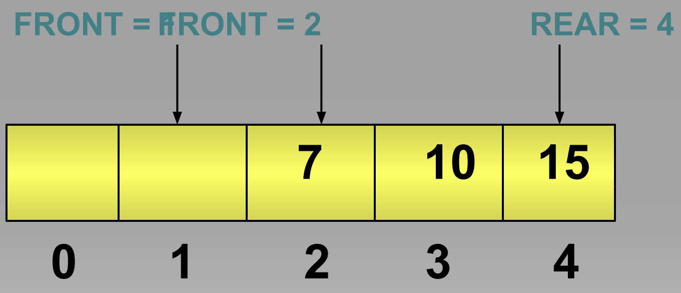
One request processed

1. Retrieve the element at index FRONT.
2. Increment FRONT by 1.



Implementing a Queue Using an Array (Contd.)

1. Retrieve the element at index FRONT.
2. Increment FRONT by 1.

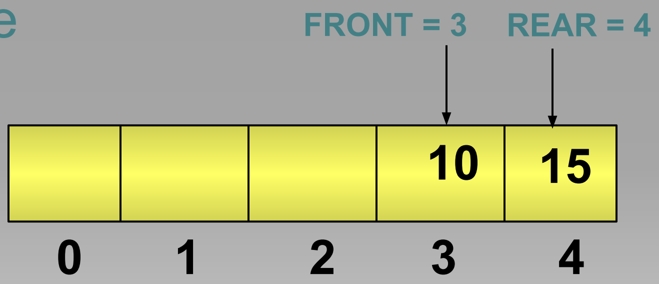


Delete operation complete

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.

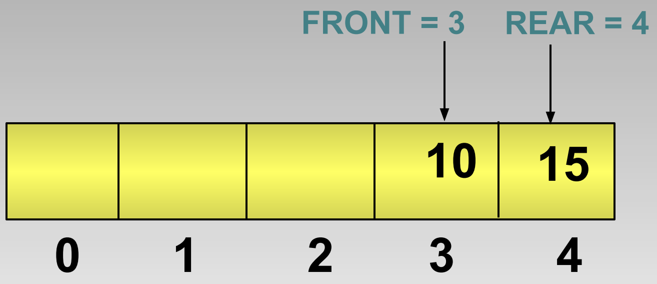
n the queue, the
y.
proach is that the
ing is discarded
e.



- 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.

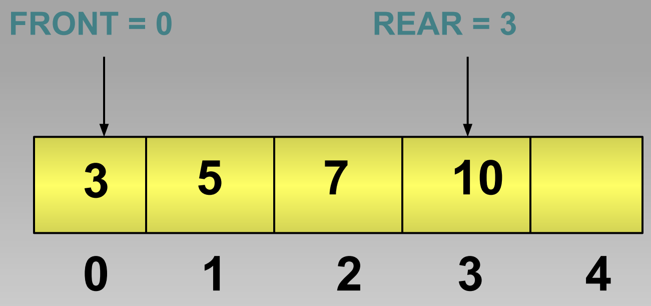
Implementing a Queue Using an Array (Contd.)

- 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.



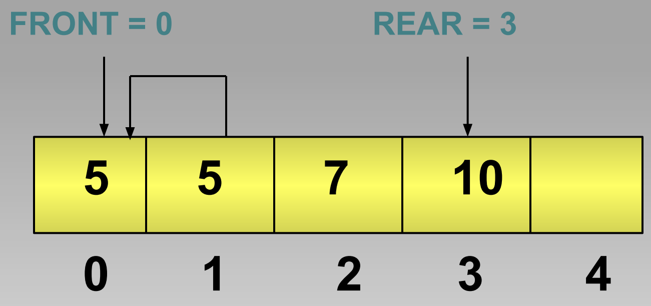
Implementing a Queue Using an Array (Contd.)

- 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.



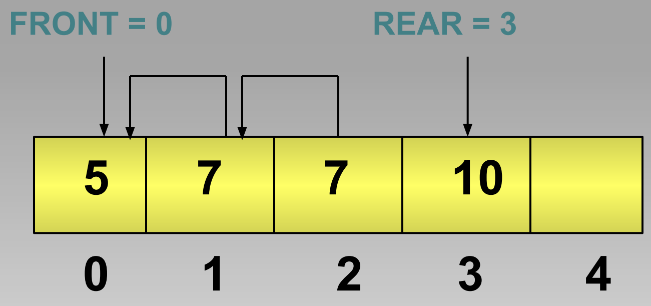
Implementing a Queue Using an Array (Contd.)

- 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.)

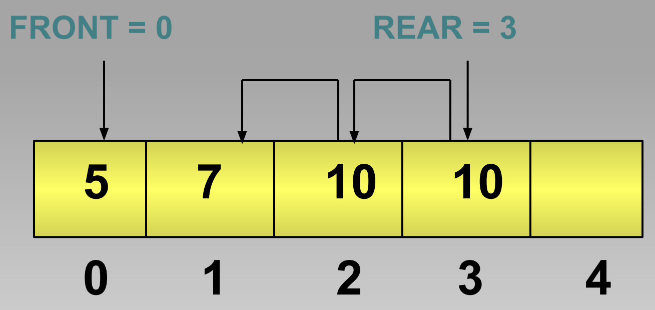
- To maintain FRONT at zero index position, every delete operation would require you to shift all the succeeding elements in the array one position left.
- Let us implement a delete operation on the following queue.



Data Structures and Algorithms

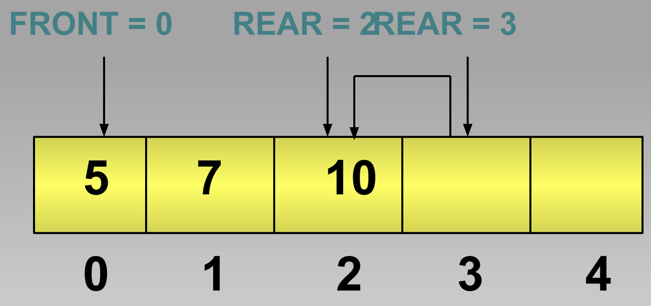
Implementing a Queue Using an Array (Contd.)

- 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.



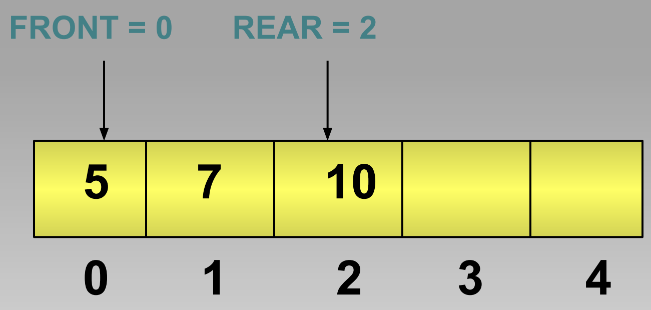
Implementing a Queue Using an Array (Contd.)

- 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.



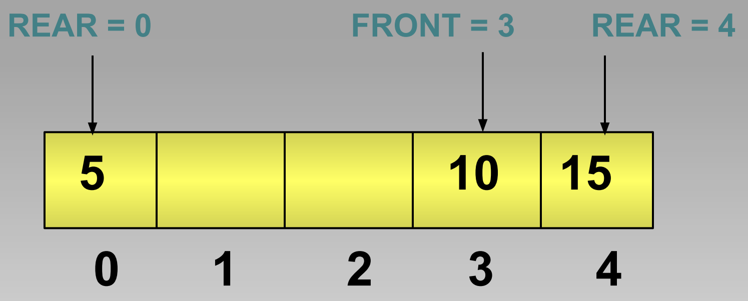
Implementing a Queue Using an Array (Contd.)

- 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.)

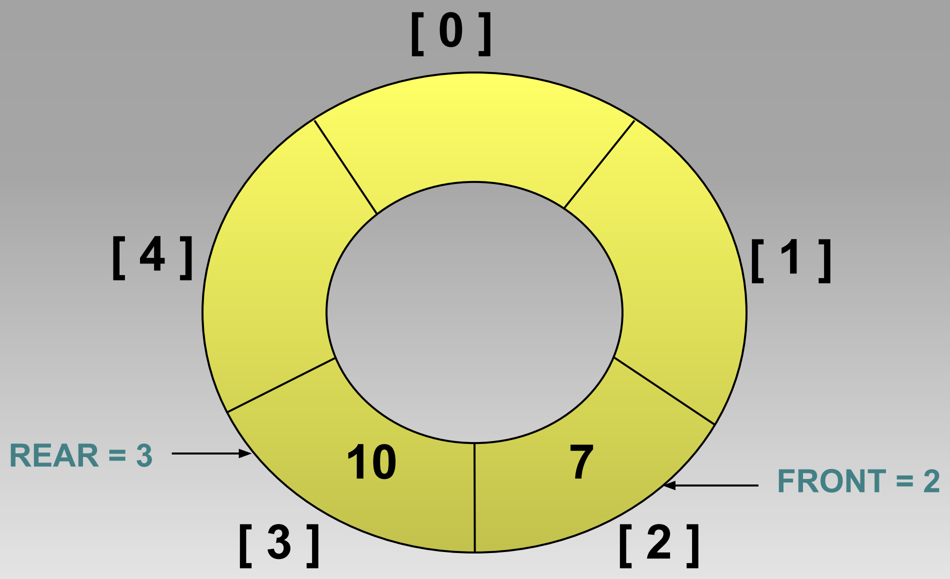
- An effective way to solve this problem is to implement the queue in the form of a circular array.
- In this approach, if REAR is at the last index position and if there is space in the beginning of an array, then you can set the value of REAR to zero and start inserting elements from the beginning.



Insert 5

Implementing a Queue Using an Array (Contd.)

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



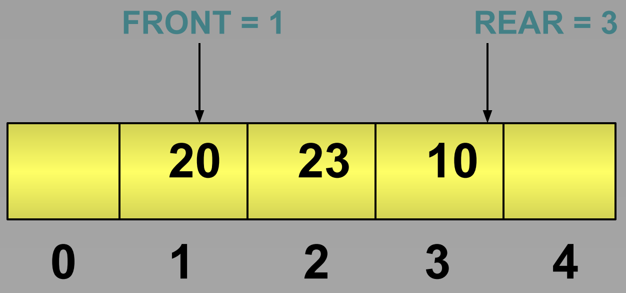
Data Structures and Algorithms

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 queue implemented in the form of a circular array.

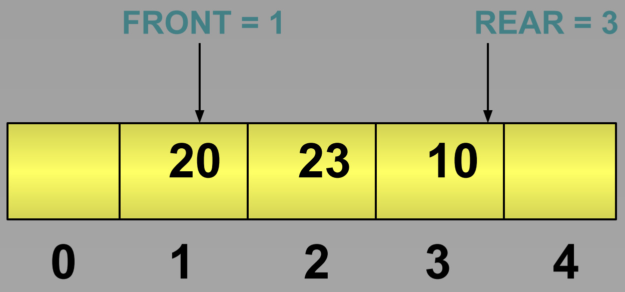
Request number generated 15



```
1.  If the queue is empty (If FRONT= –1):
    a.  Set FRONT = 0
    b.  Set REAR = 0
    c.  Go to step 4
2.  If REAR is at the last index position:
    a.  Set REAR = 0
    b.  Go to step 4
3.  Increment REAR by 1
4.  Queue[REAR] = element
```

Implementing a Queue Using an Array (Contd.)

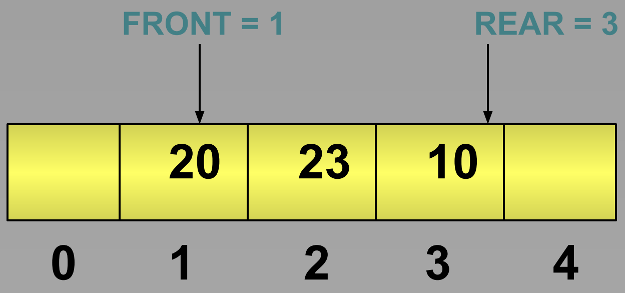
Request number generated 15



```
1. If the queue is empty (If FRONT= –1):
     a. Set FRONT = 0
     b. Set REAR = 0
     c. Go to step 4
2. If REAR is at the last index position:
     a. Set REAR = 0
     b. Go to step 4
3. Increment REAR by 1
4. Queue[REAR] = element
```

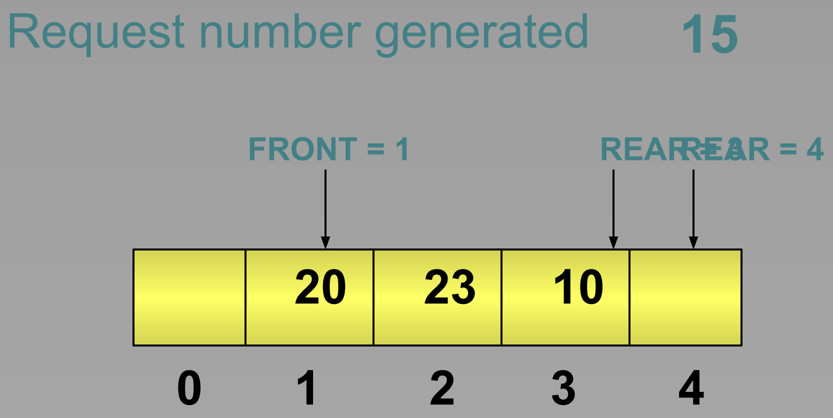
Implementing a Queue Using an Array (Contd.)

Request number generated 15



```
1. If the queue is empty (If FRONT= –1):
    a. Set FRONT = 0
    b. Set REAR = 0
    c. Go to step 4
2. If REAR is at the last index position:
    a. Set REAR = 0
    b. Go to step 4
3. Increment REAR by 1
4. Queue[REAR] = element
```

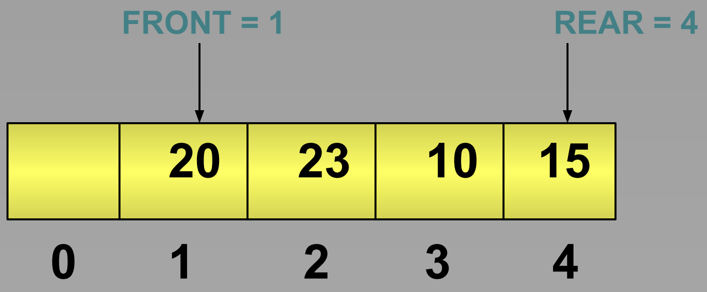
Implementing a Queue Using an Array (Contd.)



```
1.  If the queue is empty (If FRONT= –1):
        a.  Set FRONT = 0
        b.  Set REAR = 0
        c.  Go to step 4
2.  If REAR is at the last index position:
        a.  Set REAR = 0
        b.  Go to step 4
3.  Increment REAR by 1
4.  Queue[REAR] = element
```

Implementing a Queue Using an Array (Contd.)

Request number generated 15

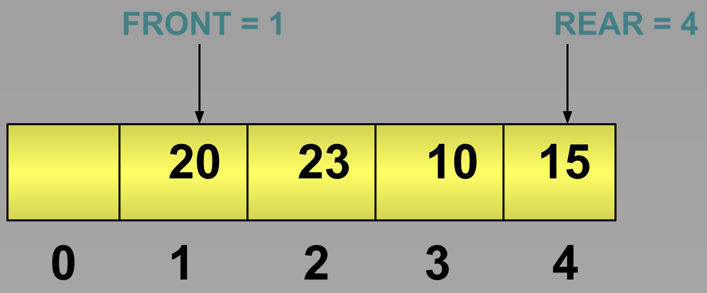


Insertion complete

```
1.  If the queue is empty (If FRONT= –1):
    a.  Set FRONT = 0
    b.  Set REAR = 0
    c.  Go to step 4
2.  If REAR is at the last index position:
    a.  Set REAR = 0
    b.  Go to step 4
3.  Increment REAR by 1
4.  Queue[REAR] = element
```

Implementing a Queue Using an Array (Contd.)

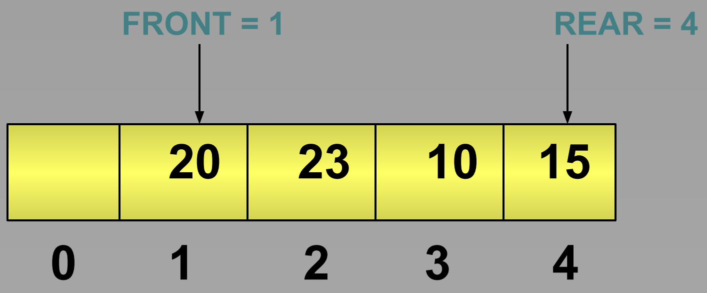
Request number generated 17



```
1. If the queue is empty (If FRONT= –1):
    a. Set FRONT = 0
    b. Set REAR = 0
    c. Go to step 4
2. If REAR is at the last index position:
    a. Set REAR = 0
    b. Go to step 4
3. Increment REAR by 1
4. Queue[REAR] = element
```

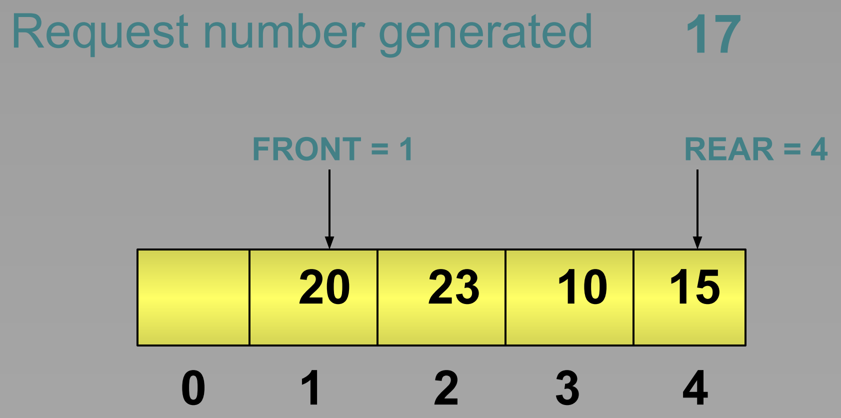
Implementing a Queue Using an Array (Contd.)

Request number generated 17



```
1. If the queue is empty (If FRONT= –1):
    a. Set FRONT = 0
    b. Set REAR = 0
    c. Go to step 4
2. If REAR is at the last index position:
    a. Set REAR = 0
    b. Go to step 4
3. Increment REAR by 1
4. Queue[REAR] = element
```

Implementing a Queue Using an Array (Contd.)

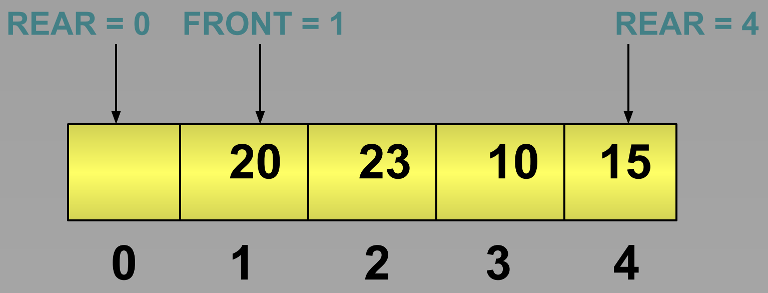


```
1. If the queue is empty (If FRONT= –1):
     a. Set FRONT = 0
     b. Set REAR = 0
     c. Go to step 4
2. If REAR is at the last index position:
     a. Set REAR = 0
     b. Go to step 4
3. Increment REAR by 1
4. Queue[REAR] = element
```

Data Structures and Algorithms

Implementing a Queue Using an Array (Contd.)

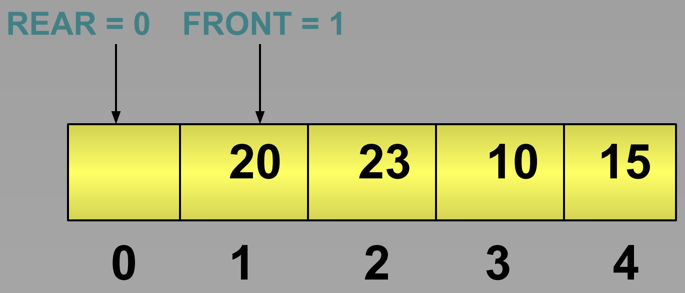
Request number generated 17



```
1.  If the queue is empty (If FRONT= –1):
    a.  Set FRONT = 0
    b.  Set REAR = 0
    c.  Go to step 4
2.  If REAR is at the last index position:
    a.  Set REAR = 0
    b.  Go to step 4
3.  Increment REAR by 1
4.  Queue[REAR] = element
```

Implementing a Queue Using an Array (Contd.)

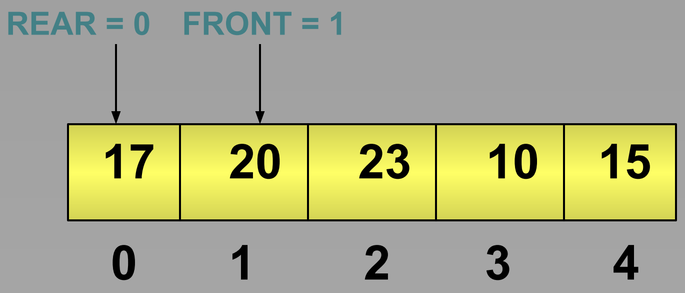
Request number generated 17



```
1. If the queue is empty (If FRONT= –1):
    a. Set FRONT = 0
    b. Set REAR = 0
    c. Go to step 4
2. If REAR is at the last index position:
    a. Set REAR = 0
    b. Go to step 4
3. Increment REAR by 1
4. Queue[REAR] = element
```

Implementing a Queue Using an Array (Contd.)

Request number generated 17

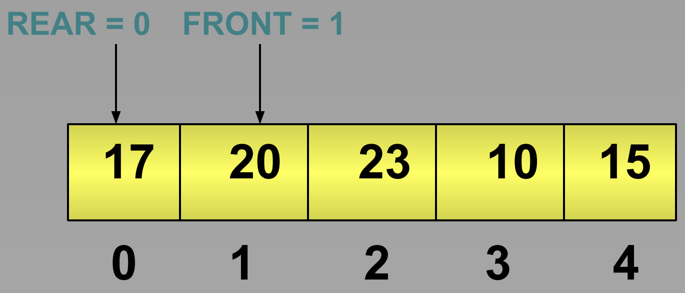


Insertion complete

```
1.  If the queue is empty (If FRONT= –1):
    a.  Set FRONT = 0
    b.  Set REAR = 0
    c.  Go to step 4
2.  If REAR is at the last index position:
    a.  Set REAR = 0
    b.  Go to step 4
3.  Increment REAR by 1
4.  Queue[REAR] = element
```

Implementing a Queue Using an Array (Contd.)

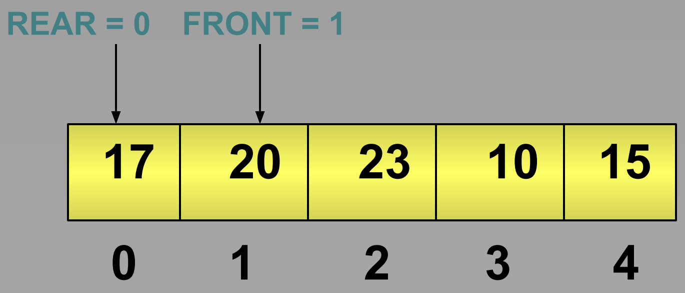
Request number generated 25



```
1.  If the queue is empty (If FRONT= –1):
        a.  Set FRONT = 0
        b.  Set REAR = 0
        c.  Go to step 4
2.  If REAR is at the last index position:
        a.  Set REAR = 0
        b.  Go to step 4
3.  Increment REAR by 1
4.  Queue[REAR] = element
```

Implementing a Queue Using an Array (Contd.)

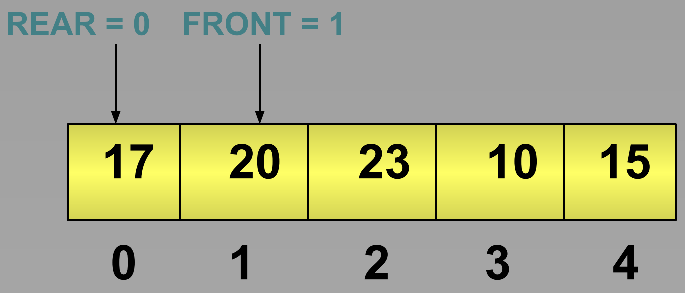
Request number generated 25



```
1.  If the queue is empty (If FRONT= –1):
        a.  Set FRONT = 0
        b.  Set REAR = 0
        c.  Go to step 4
2.  If REAR is at the last index position:
        a.  Set REAR = 0
        b.  Go to step 4
3.  Increment REAR by 1
4.  Queue[REAR] = element
```

Implementing a Queue Using an Array (Contd.)

Request number generated 25

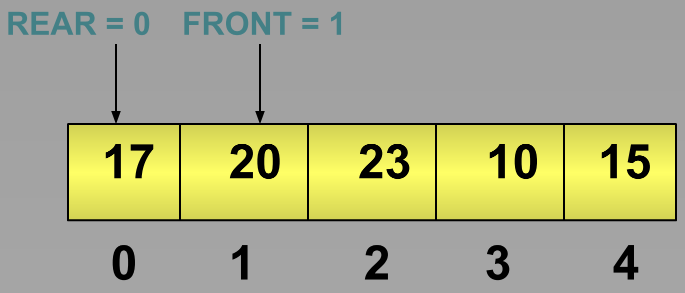


```
1. If the queue is empty (If FRONT= –1):
    a. Set FRONT = 0
    b. Set REAR = 0
    c. Go to step 4
2. If REAR is at the last index position:
    a. Set REAR = 0
    b. Go to step 4
3. Increment REAR by 1
4. Queue[REAR] = element
```

Data Structures and Algorithms

Implementing a Queue Using an Array (Contd.)

Request number generated **25**



```
1.  If the queue is empty (If FRONT= –1):
    a.  Set FRONT = 0
    b.  Set REAR = 0
    c.  Go to step 4
2.  If REAR is at the last index position:
    a.  Set REAR = 0
    b.  Go to step 4
3.  Increment REAR by 1
4.  Queue[REAR] = element
```

Before implementing an insert operation, you should always check for the queue full condition.

Data Structures and Algorithms

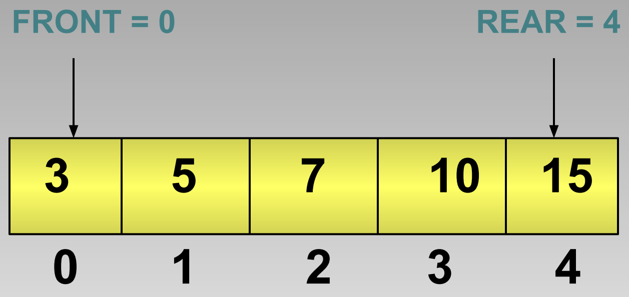
Implementing a Queue Using an Array (Contd.)

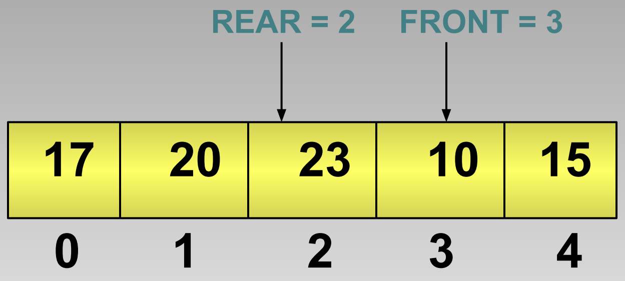
- The conditions for queue full are as follows:

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

OR

If FRONT = REAR + 1





Implementing a Queue Using an Array (Contd.)

- Modified algorithm for inserting an element in a circular queue.

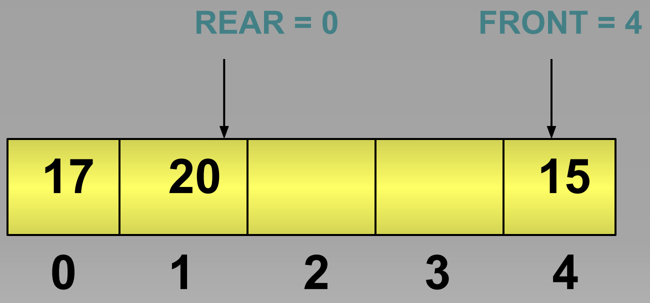
```
1.  If the queue is full:
        a.  Display “Queue overflow”
        b.  Exit
2.  If queue is empty (If FRONT = –1):
        a.  Set FRONT = 0
        b.  Set REAR = 0
        c.  Go to Step 5
3.  If REAR is at the last index position:
        a.  Set REAR = 0
        b.  Go to step 5
4.  Increment REAR by 1
5.  Queue[REAR] = element
```

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 an 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.)

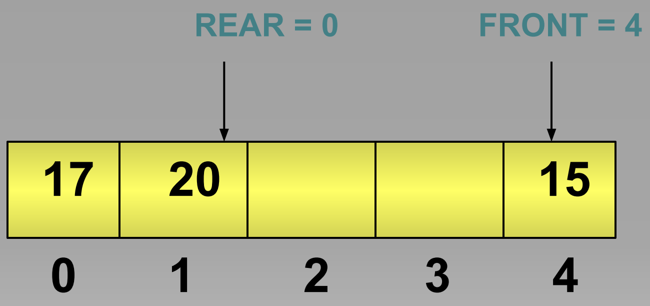
- Algorithm to delete an element from a circular queue.



```
1.  If there is only one element in the
    queue:
    a.  Set FRONT = –1
    b.  Set REAR = –1
    c.  Exit
2.  If FRONT is at the last index
    position:
    a.  Set FRONT = 0
    b.  Exit
3.  Increment FRONT by 1
```

Implementing a Queue Using an Array (Contd.)

One request processed

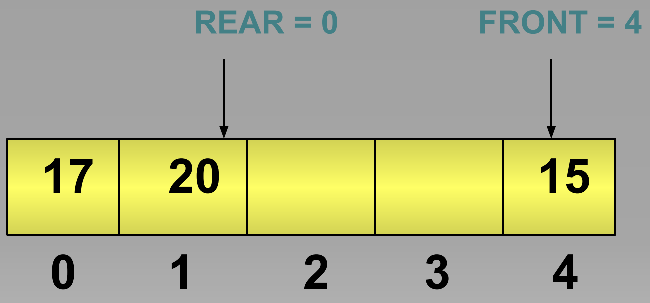


1. If there is only one element in the queue:
   a. Set FRONT = –1
   b. Set REAR = –1
   c. Exit
2. If FRONT is at the last index position:
   a. Set FRONT = 0
   b. Exit
3. Increment FRONT by 1

Data Structures and Algorithms

Implementing a Queue Using an Array (Contd.)

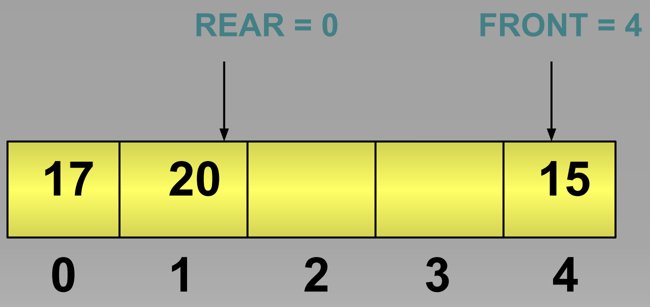
One request processed



1. If there is only one element in the queue:
   a. Set FRONT = –1
   b. Set REAR = –1
   c. Exit
2. If FRONT is at the last index position:
   a. Set FRONT = 0
   b. Exit
3. Increment FRONT by 1

Implementing a Queue Using an Array (Contd.)

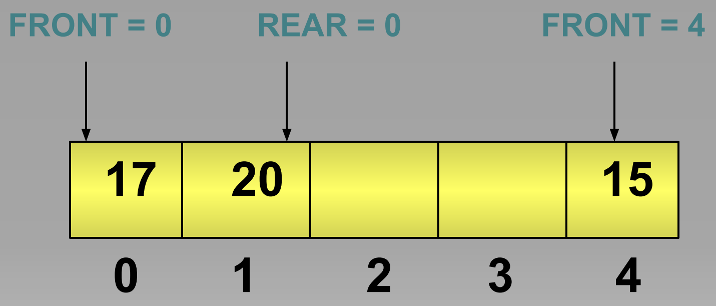
One request processed



1. If there is only one element in the queue:
   a. Set FRONT = –1
   b. Set REAR = –1
   c. Exit
2. If FRONT is at the last index position:
   a. Set FRONT = 0
   b. Exit
3. Increment FRONT by 1

Implementing a Queue Using an Array (Contd.)

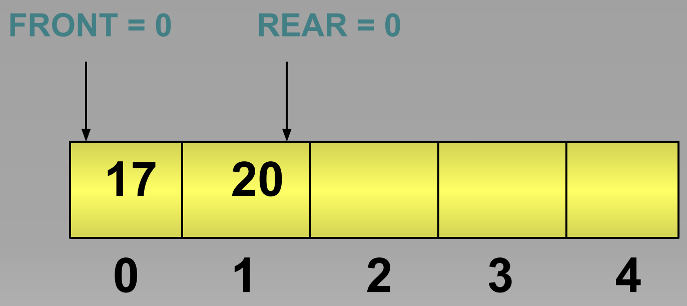
One request processed



```
1.  If there is only one element in the
    queue:
        a.  Set FRONT = –1
        b.  Set REAR = –1
        c.  Exit
2.  If FRONT is at the last index
    position:
        a.  Set FRONT = 0
        b.  Exit
3.  Increment FRONT by 1
```

Implementing a Queue Using an Array (Contd.)

One request processed

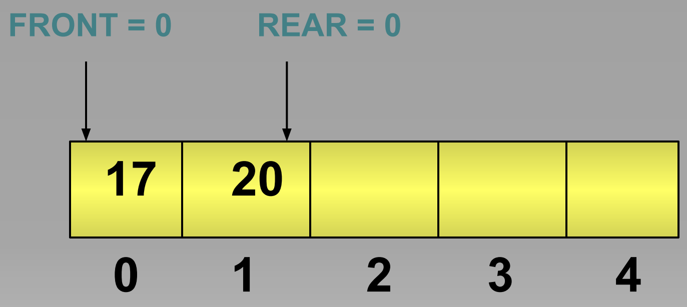


Deletion complete

```
1.  If there is only one element in the queue:
    a.  Set FRONT = –1
    b.  Set REAR = –1
    c.  Exit
2.  If FRONT is at the last index position:
    a.  Set FRONT = 0
    b.  Exit
3.  Increment FRONT by 1
```

Implementing a Queue Using an Array (Contd.)

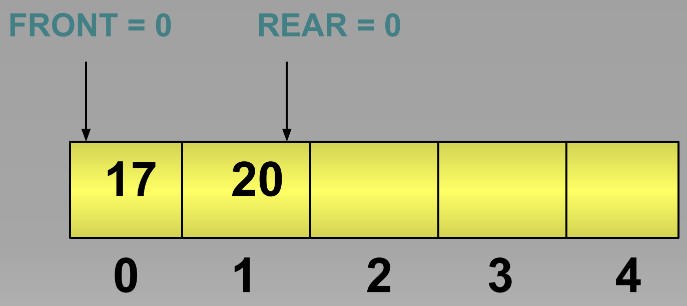
One request processed



```
1.  If there is only one element in the queue:
    a.  Set FRONT = –1
    b.  Set REAR = –1
    c.  Exit
2.  If FRONT is at the last index position:
    a.  Set FRONT = 0
    b.  Exit
3.  Increment FRONT by 1
```

Implementing a Queue Using an Array (Contd.)

One request processed

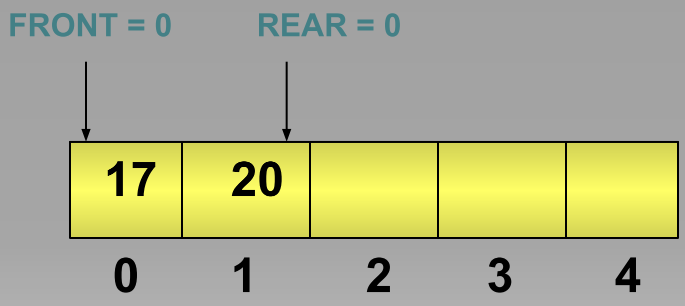


```
1.  If there is only one element in the
    queue:
      a.  Set FRONT = –1
      b.  Set REAR = –1
      c.  Exit
2.  If FRONT is at the last index
    position:
      a.  Set FRONT = 0
      b.  Exit
3.  Increment FRONT by 1
```

Data Structures and Algorithms

Implementing a Queue Using an Array (Contd.)

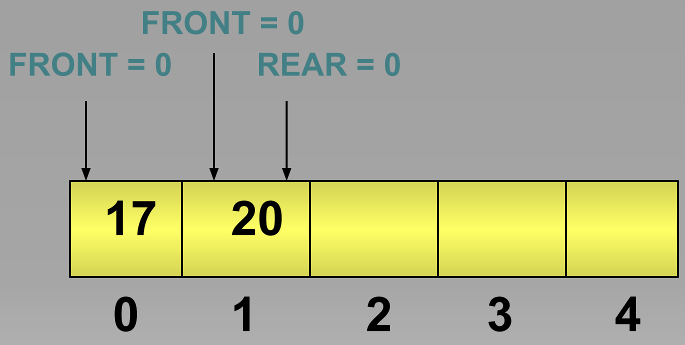
One request processed



1. If there is only one element in the queue:
   a. Set FRONT = –1
   b. Set REAR = –1
   c. Exit
2. If FRONT is at the last index position:
   a. Set FRONT = 0
   b. Exit
3. Increment FRONT by 1

Implementing a Queue Using an Array (Contd.)

One request processed

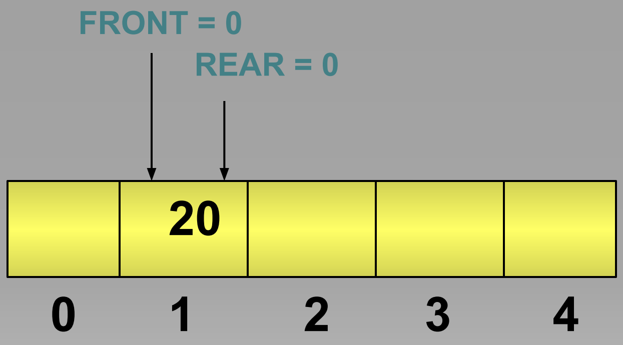


Deletion complete

1. If there is only one element in the queue:
   a. Set FRONT = –1
   b. Set REAR = –1
   c. Exit
2. If FRONT is at the last index position:
   a. Set FRONT = 0
   b. Exit
3. Increment FRONT by 1

Implementing a Queue Using an Array (Contd.)

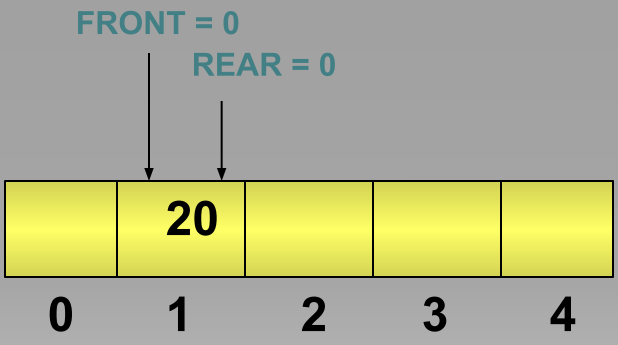
One request processed



1. If there is only one element in the queue:
   a. Set FRONT = –1
   b. Set REAR = –1
   c. Exit
2. If FRONT is at the last index position:
   a. Set FRONT = 0
   b. Exit
3. Increment FRONT by 1

Implementing a Queue Using an Array (Contd.)

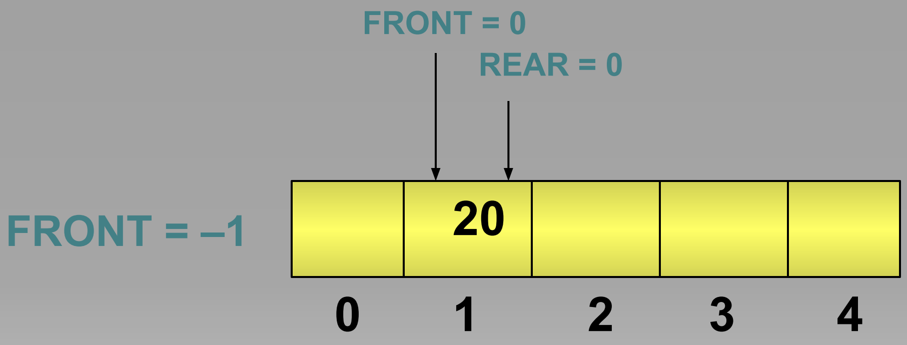
One request processed



```
1. If there is only one element in the queue:
       a. Set FRONT = –1
       b. Set REAR = –1
       c. Exit
2. If FRONT is at the last index position:
       a. Set FRONT = 0
       b. Exit
3. Increment FRONT by 1
```

Implementing a Queue Using an Array (Contd.)

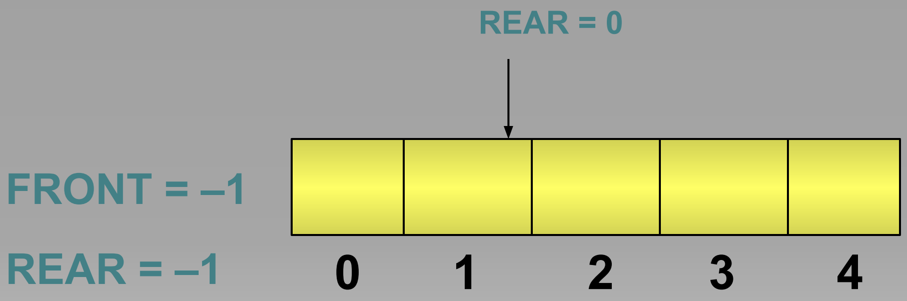
One request processed



1. If there is only one element in the queue:
   a. Set FRONT = –1
   b. Set REAR = –1
   c. Exit
2. If FRONT is at the last index position:
   a. Set FRONT = 0
   b. Exit
3. Increment FRONT by 1

Implementing a Queue Using an Array (Contd.)

One request processed



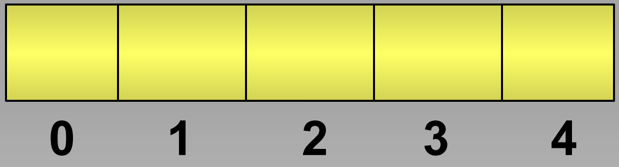
1. If there is only one element in the queue:
   a. Set FRONT = –1
   b. Set REAR = –1
   c. Exit
2. If FRONT is at the last index position:
   a. Set FRONT = 0
   b. Exit
3. Increment FRONT by 1

Implementing a Queue Using an Array (Contd.)

One request processed

FRONT = –1

REAR = –1



Deletion complete

```
1. If there is only one element in the queue:
     a. Set FRONT = –1
     b. Set REAR = –1
     c. Exit
2. If FRONT is at the last index position:
     a. Set FRONT = 0
     b. Exit
3. Increment FRONT by 1
```

Implementing a Queue Using an Array (Contd.)

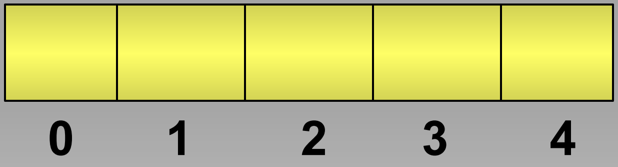
- The queue is now empty.

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

1. If there is only one element in the queue:
   a. Set FRONT = –1
   b. Set REAR = –1
   c. Exit
2. If FRONT is at the last index position:
   a. Set FRONT = 0
   b. Exit
3. Increment FRONT by 1

FRONT = –1

REAR = –1



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

Implementing a Queue Using an Array (Contd.)

- Modified algorithm for deleting an element from a circular queue:

The condition for queue empty is: FRONT = –1

```
1.  If the queue is empty: // If
                                        // FRONT = –1
     a.  Display "Queue underflow"
     b.  Exit

2.  If there is only one element in the
    queue: // If FRONT = REAR

     b.  Set FRONT = –1
     c.  Set REAR = –1
     d.  Exit

3.  If FRONT is at the last index
    position:

     b.  Set FRONT = 0
     c.  Exit

4.  Increment FRONT by 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.

Activity: Implementing a Queue Using Circular Array

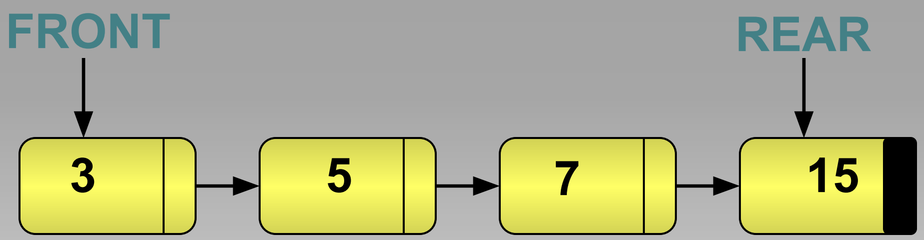
- Problem Statement:
  - Write a program to implement insert and delete operations on a queue implemented in the form of a circular array.

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.

Implementing a Queue Using a Linked List (Contd.)

- To keep track of the rear and front positions, you need to declare two variables/pointers, REAR and FRONT, that will always point to the rear and front end of the queue respectively.
- If the queue is empty, REAR and FRONT point to NULL.



Inserting an Element in a Linked Queue

- Write an algorithm to implement insert operation in a linked queue.

Inserting an Element in a Linked Queue (Contd.)

- Algorithm to insert an element in a linked queue.
- Suppose initially, the queue is empty.

Request number generated 3

REAR = NULL

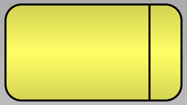
FRONT = NULL

1. Allocate memory for the new node.
2. Assign value to the data field of the new node.
3. Make the next field of the new node point to NULL.
4. If the queue is empty, execute the following steps:
   a. Make FRONT point to the new node
   b. Make REAR point to the new node
   c. Exit
5. Make the next field of REAR point to the new node.
6. Make REAR point to the new node.

Inserting an Element in a Linked Queue (Contd.)

Request number generated 3

REAR = NULL
FRONT = NULL

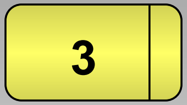


1. Allocate memory for the new node.
2. Assign value to the data field of the new node.
3. Make the next field of the new node point to NULL.
4. If the queue is empty, execute the following steps:
   a. Make FRONT point to the new node
   b. Make REAR point to the new node
   c. Exit
5. Make the next field of REAR point to the new node.
6. Make REAR point to the new node.

Inserting an Element in a Linked Queue (Contd.)

Request number generated **3**

REAR = NULL
FRONT = NULL

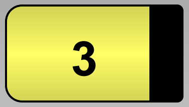


```
1. Allocate memory for the new node.
2. Assign value to the data field of the new
   node.
3. Make the next field of the new node point to
   NULL.
4. If the queue is empty, execute the following
   steps:
      a. Make FRONT point to the new node
      b. Make REAR  point to the new node
      c. Exit
5. Make the next field of REAR point to the new
   node.
6. Make REAR point to the new node.
```

Inserting an Element in a Linked Queue (Contd.)

Request number generated **3**

REAR = NULL
FRONT = NULL

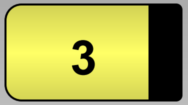


1. Allocate memory for the new node.
2. Assign value to the data field of the new node.
3. Make the next field of the new node point to NULL.
4. If the queue is empty, execute the following steps:
   a. Make FRONT point to the new node
   b. Make REAR point to the new node
   c. Exit
5. Make the next field of REAR point to the new node.
6. Make REAR point to the new node.

Inserting an Element in a Linked Queue (Contd.)

Request number generated 3

REAR = NULL
FRONT = NULL

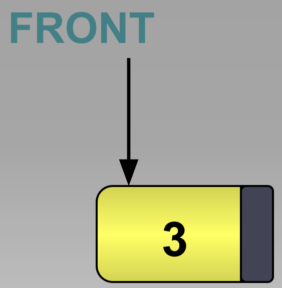


1. Allocate memory for the new node.
2. Assign value to the data field of the new node.
3. Make the next field of the new node point to NULL.
4. If the queue is empty, execute the following steps:
   a. Make FRONT point to the new node
   b. Make REAR point to the new node
   c. Exit
5. Make the next field of REAR point to the new node.
6. Make REAR point to the new node.

Inserting an Element in a Linked Queue (Contd.)

Request number generated 3

REAR = NULL
FRONT = NULL

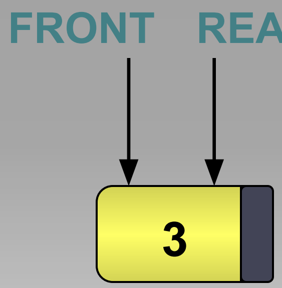


1. Allocate memory for the new node.
2. Assign value to the data field of the new node.
3. Make the next field of the new node point to NULL.
4. If the queue is empty, execute the following steps:
    a. Make FRONT point to the new node
    b. Make REAR point to the new node
    c. Exit
5. Make the next field of REAR point to the new node.
6. Make REAR point to the new node.

Inserting an Element in a Linked Queue (Contd.)

Request number generated 3

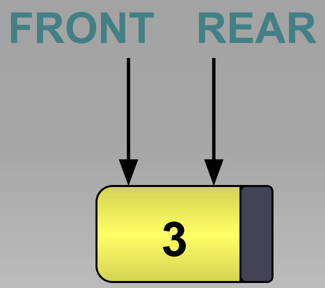
REAR = NULL



1. Allocate memory for the new node.
2. Assign value to the data field of the new node.
3. Make the next field of the new node point to NULL.
4. If the queue is empty, execute the following steps:
   a. Make FRONT point to the new node
   b. Make REAR point to the new node
   c. Exit
5. Make the next field of REAR point to the new node.
6. Make REAR point to the new node.

Inserting an Element in a Linked Queue (Contd.)

Request number generated 3

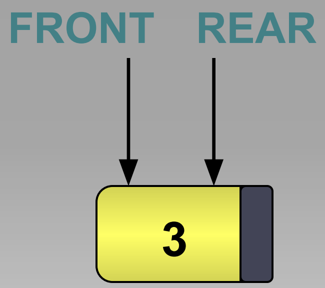


1. Allocate memory for the new node.
2. Assign value to the data field of the new node.
3. Make the next field of the new node point to NULL.
4. If the queue is empty, execute the following steps:
   a. Make FRONT point to the new node
   b. Make REAR point to the new node
   c. Exit
5. Make the next field of REAR point to the new node.
6. Make REAR point to the new node.

Insert operation complete

Inserting an Element in a Linked Queue (Contd.)

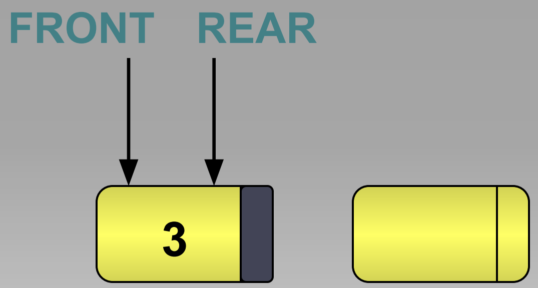
Request number generated **5**



1. Allocate memory for the new node.
2. Assign value to the data field of the new node.
3. Make the next field of the new node point to NULL.
4. If the queue is empty, execute the following steps:
   a. Make FRONT point to the new node
   b. Make REAR point to the new node
   c. Exit
5. Make the next field of REAR point to the new node.
6. Make REAR point to the new node.

Inserting an Element in a Linked Queue (Contd.)

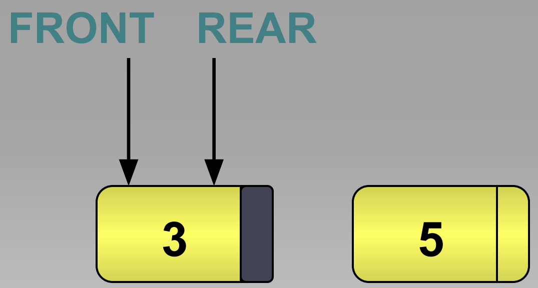
Request number generated **5**



1. Allocate memory for the new node.
2. Assign value to the data field of the new node.
3. Make the next field of the new node point to NULL.
4. If the queue is empty, execute the following steps:
   a. Make FRONT point to the new node
   b. Make REAR point to the new node
   c. Exit
5. Make the next field of REAR point to the new node.
6. Make REAR point to the new node.

Inserting an Element in a Linked Queue (Contd.)

Request number generated 5

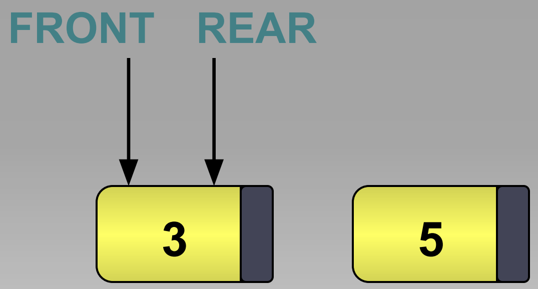


1. Allocate memory for the new node.
2. Assign value to the data field of the new node.
3. Make the next field of the new node point to NULL.
4. If the queue is empty, execute the following steps:
   a. Make FRONT point to the new node
   b. Make REAR point to the new node
   c. Exit
5. Make the next field of REAR point to the new node.
6. Make REAR point to the new node.

Data Structures and Algorithms

Inserting an Element in a Linked Queue (Contd.)

Request number generated 5

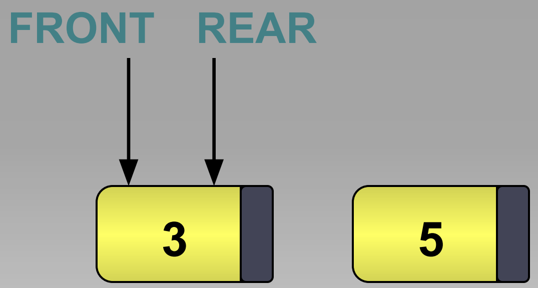


1. Allocate memory for the new node.
2. Assign value to the data field of the new node.
3. Make the next field of the new node point to NULL.
4. If the queue is empty, execute the following steps:
   a. Make FRONT point to the new node
   b. Make REAR point to the new node
   c. Exit
5. Make the next field of REAR point to the new node.
6. Make REAR point to the new node.

Data Structures and Algorithms

Inserting an Element in a Linked Queue (Contd.)

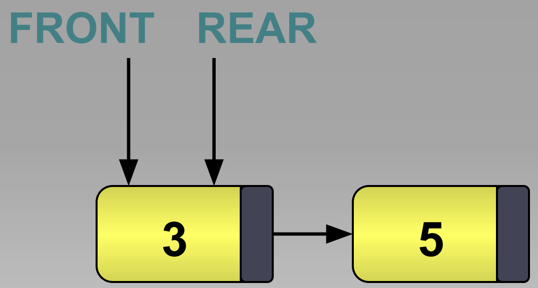
Request number generated 5



1. Allocate memory for the new node.
2. Assign value to the data field of the new node.
3. Make the next field of the new node point to NULL.
4. If the queue is empty, execute the following steps:
   a. Make FRONT point to the new node
   b. Make REAR point to the new node
   c. Exit
5. Make the next field of REAR point to the new node.
6. Make REAR point to the new node.

Inserting an Element in a Linked Queue (Contd.)

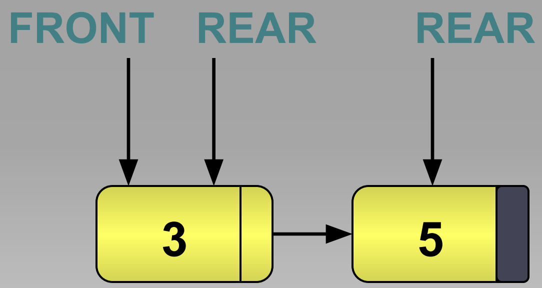
Request number generated 5



1. Allocate memory for the new node.
2. Assign value to the data field of the new node.
3. Make the next field of the new node point to NULL.
4. If the queue is empty, execute the following steps:
   a. Make FRONT point to the new node
   b. Make REAR point to the new node
   c. Exit
5. Make the next field of REAR point to the new node.
6. Make REAR point to the new node.

Inserting an Element in a Linked Queue (Contd.)

Request number generated 5



1. Allocate memory for the new node.
2. Assign value to the data field of the new node.
3. Make the next field of the new node point to NULL.
4. If the queue is empty, execute the following steps:

   a. Make FRONT point to the new node
   b. Make REAR point to the new node
   c. Exit
5. Make the next field of REAR point to the new node.
6. Make REAR point to the new node.

Insert operation complete

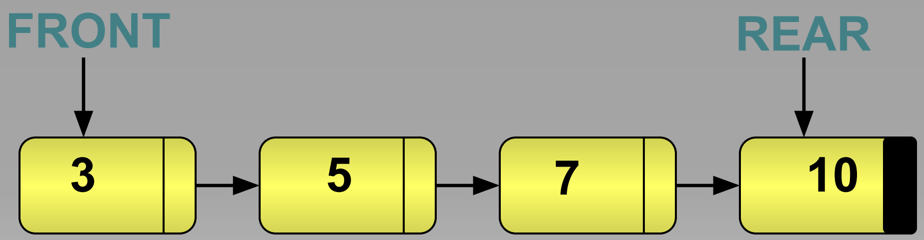
Deleting an Element from a Linked Queue

- Write an algorithm to implement the delete operation on a linked queue.

Deleting an Element from a Linked Queue (Contd.)

- One request processed

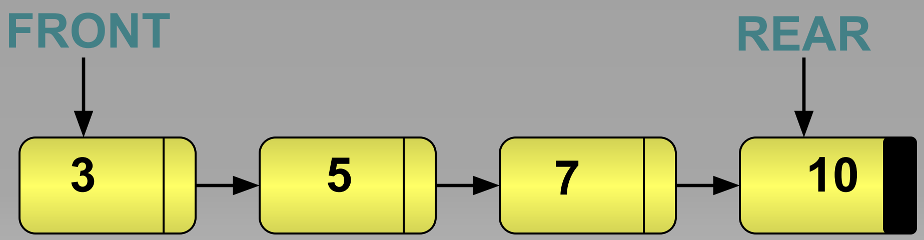
Algorithm to implement delete operation on a linked queue.



```
1.  If the queue is empty: // FRONT = NULL
        a.  Display "Queue empty"
        b.  Exit
2.  Mark the node marked FRONT as current
3.  Make FRONT point to the next node in its
    sequence
4.  Release the memory for the node marked
    as current
```

Deleting an Element from a Linked Queue (Contd.)

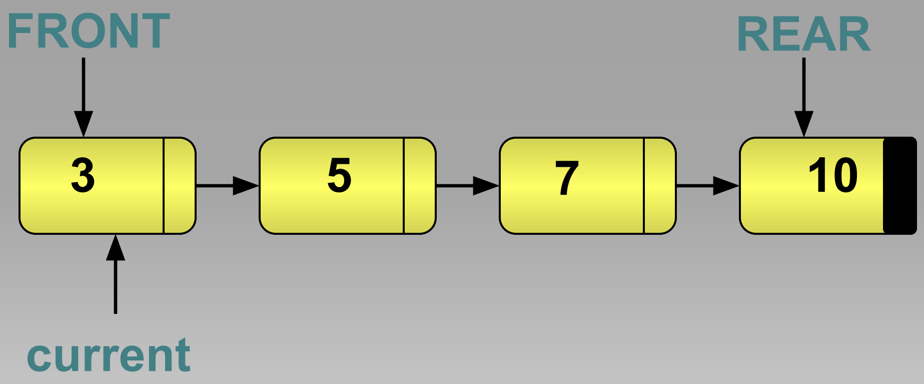
One request processed



1. If the queue is empty: // **FRONT = NULL**
   a. Display "Queue empty"
   b. Exit
2. Mark the node marked FRONT as current
3. Make FRONT point to the next node in its sequence
4. Release the memory for the node marked as current

Deleting an Element from a Linked Queue (Contd.)

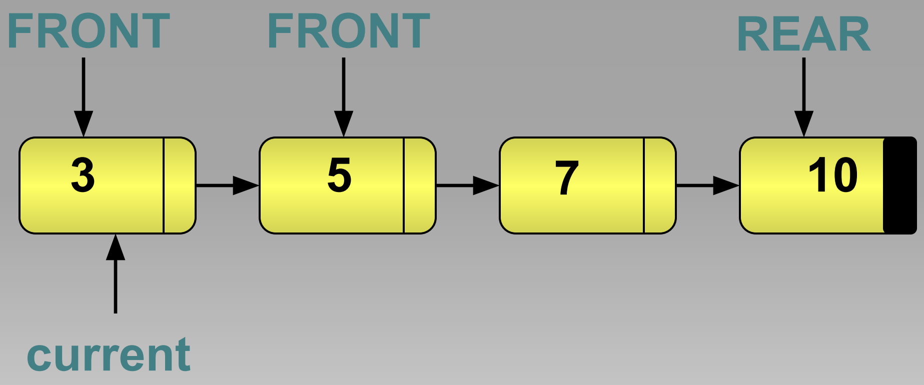
One request processed



1. If the queue is empty: // **FRONT = NULL**
   a. Display "Queue empty"
   b. Exit
2. Mark the node marked FRONT as current
3. Make FRONT point to the next node in its sequence
4. Release the memory for the node marked as current

Deleting an Element from a Linked Queue (Contd.)

One request processed



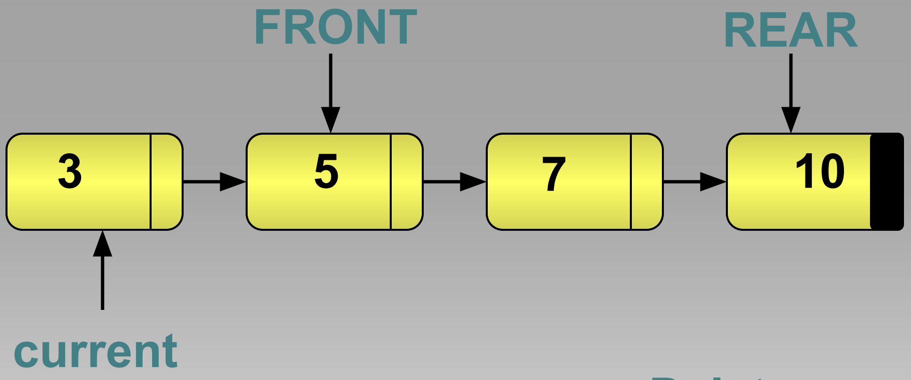
1. If the queue is empty: // **FRONT = NULL**
   a. Display "Queue empty"
   b. Exit
2. Mark the node marked FRONT as current
3. Make FRONT point to the next node in its sequence
4. Release the memory for the node marked as current

Data Structures and Algorithms

Deleting an Element from a Linked Queue (Contd.)

One request processed

1. If the queue is empty: // **FRONT = NULL**
   a. Display “Queue empty”
   b. Exit
2. Mark the node marked FRONT as current
3. Make FRONT point to the next node in its sequence
4. Release the memory for the node marked as current



Delete operation complete

Memory released

Data Structures and Algorithms

Just a minute

- How does the implementation of a linked list differ from that of a linked queue?

- Answer:
  - In a linked list, you can insert and delete elements anywhere in the list. However, in a linked queue, insertion and deletion takes place only from the ends. More specifically, insertion takes place at the rear end and deletion takes place at the front end of the queue.

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 ground 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 Hashing

- Binary search algorithm has some disadvantages:
  - It works only on sorted lists.
  - It requires a way to directly access the middle element of the list.
- An alternate searching algorithm that overcomes these limitations and provides good efficiency is hashing.

Data Structures and Algorithms

Defining Hashing

- Suppose you have to search for the record corresponding to a given key value in a given list of records.
- To retrieve the desired record, you have to search sequentially through the records until a record with the desired key value is found.
- This method is very time consuming, especially if the list is very large.
- An effective solution to search the record would be to search it with the help of an offset address.
- You can calculate the offset address of a record by using a technique called hashing.

Defining Hashing (Contdd.)

- The fundamental principle of hashing is to convert a key to an offset address to retrieve a record.
- Conversion of the key to an address is done by a relation (formula), which is known as a hashing function.
- The process of searching a record using hashing can be summarized as:
  1. Given a key, the hash function converts it into a hash value (location) within the range 1 to n, where n is the size of the storage (address) space that has been allocated for the records.
  2. The record is then retrieved at the location generated.

Limitations of Hashing

- Two limitations of hashing are:
  - It may result in collision.
  - It does not allow sequential access.

Selecting a Hash Function

- Two principal criteria in selecting a hash function are:
  - It should be quick and easy to compute.
  - It should achieve a uniform distribution of keys in the address space.
- There are various techniques that can be used to design a hash function:
  - Truncation method
  - Modular method
  - Mid Square method
  - Folding method

Resolving Collision

- A situation in which an attempt is made to store two keys at the same position is known as collision.
- Two records cannot occupy the same position. Therefore, a collision situation needs to be taken care of.
- Collision can be resolved by using a method called separate chaining.

Determining the Efficiency of Hashing

- Searching becomes faster using hashing as compared to any other searching method.
- The efficiency of hashing is ideally O(1).
- However, because of collision, the efficiency of hashing gets reduced.
- The efficiency of hashing in this case depends on the quality of the hash function.

Summary

- In this session, you learned that:
  - A queue is linear data structure in which the elements are inserted at the rear end and deleted from the front.
  - The various operation implemented on a queue are insert and remove.
  - A queue can be implemented by using arrays or linked list.
  - A queue implemented in the form of a circular array overcomes the problem of unutilized space encountered in queues implemented by using a linear array.
  - A queue implemented by using a linked list is called a linked queue.

Summary (Contd.)

- Queues offer large number of applications in various fields such as:
  - Printer spooling
  - CPU scheduling
  - Mail service
  - Keyboard buffering
  - Elevator
- The fundamental principle of hashing is to convert a given key value to an offset address to retrieve a record.
- In hashing, conversion of the key to an address is done by a relation (formula), which is known as a hashing function.
- The situation in which the hash function generates the same hash value for two or more keys is called collision.
- The occurrence of collision can be minimized by using a good hash function.

Summary (Contd.)

- Two principle criteria in selecting a hash function are:
  - It should be easy and quick to compute.
  - It should achieve a uniform distribution of keys in the address space.
- There are various techniques that can be used to design a hash function, some of which are:
  - Truncation method
  - Modular method
  - Mid Square method
  - Folding method
- The problem of collision can be resolved by using a method called separate chaining.