3 : Stacks and Queues

IT3206 – Data Structures and Algorithms

Level II - Semester 3

© e-Learning Centre, UCSC

Overview

- This section provides important features of stacks & queues to understand how and why they organize the data so uniquely.
- The section will also illustrate the implementation of stacks & queues by using both arrays as well as linked lists.
- Finally, the section will discuss in detail some of the very useful areas where stacks & queues are primarily used.

Intended Learning Outcomes

- At the end of this lesson, you will be able to;
  - Explain the usage stacks and queues
  - Demonstrate the basic operations of stacks and queues
  - Apply stacks and queues for problem-solving

© e-Learning Centre, UCSC

List of subtopics

© e-Learning Centre, UCSC

3.1.1 Introduction to Stacks

- A stack is a simple data structure used for storing data (similar to Linked Lists).
- In a stack, the order in which the data arrives is important.
- A pile of plates in a cafeteria is a good example of a stack. The plates are added to the stack as they are cleaned, and they are placed on the top. When a plate, is required it is taken from the top of the stack. The first plate placed on the stack is the last one to be used.

© e-Learning Centre, UCSC

3.1.1 Introduction to Stacks cont.

- A stack is an **ordered list** in which insertion and deletion are done at **one end**, called **top**.
- The last element inserted is the first one to be deleted.
- Hence, it is called the **Last in First out** (LIFO) or First in Last out (FILO) list.
- The INSERT operation on a stack is often called **PUSH**.
- DELETE operation which does not take an element argument is often called **POP.**
- Trying to pop out an empty stack is called **underflow** and trying to push an element in a full stack is called **overflow.**

© e-Learning Centre, UCSC

3.1.1 Introduction to Stacks cont.

Main stack operations

- Push (int data): Inserts data onto stack.
- int Pop(): Removes and returns the last inserted element from the stack.

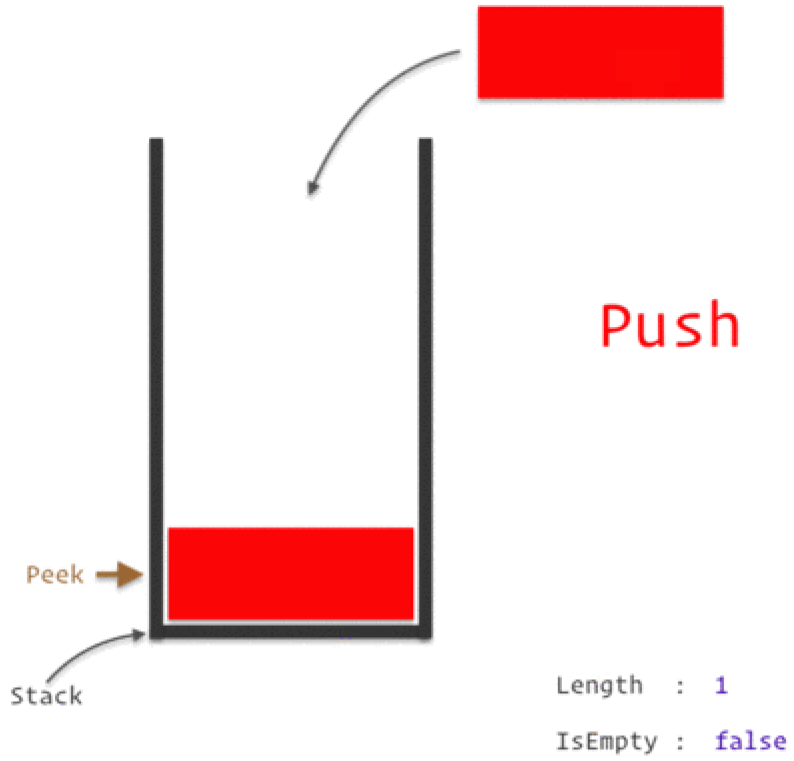
Auxiliary stack operations

- int IsEmptyStack(): Indicates whether any elements are stored in the stack or not.
- int IsFullStack(): Indicates whether the stack is full or not.
- int Top(): Returns the last inserted element without removing it.
- int Size(): Returns the number of elements stored in the stack.

© e-Learning Centre, UCSC

3.1.1 Introduction to Stacks cont.

How stack works



https://miro.medium.com/max/800/0*SESFJYWU5a-3XM9m.gif

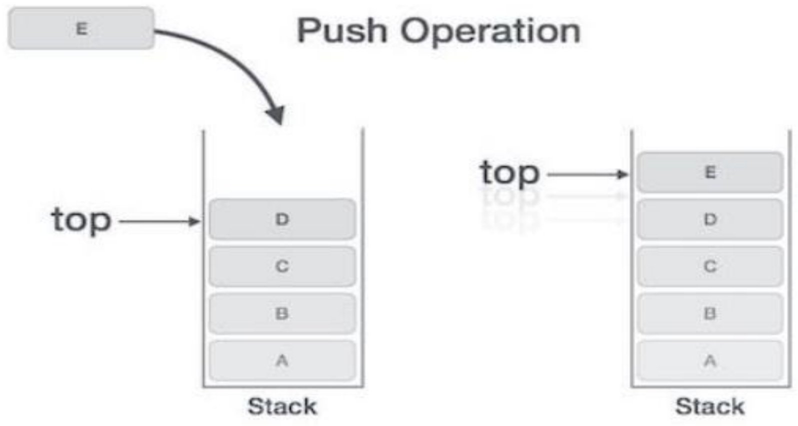
© e-Learning Centre, UCSC

3.1.1 Introduction to Stacks cont.

- **Push Operation Algorithm**
  - Step 1 − Checks if the stack is full.
  - Step 2 − If the stack is full, produces an error and exit.
  - Step 3 − If the stack is not full, increments top to point next empty space.
  - Step 4 − Adds data element to the stack location, where top is pointing.

- **Push Operation Pseudocode**

```
begin procedure push: stack, data
  if stack is full
    return null
  endif
  top ← top + 1
  stack[top] ← data
end procedure
```



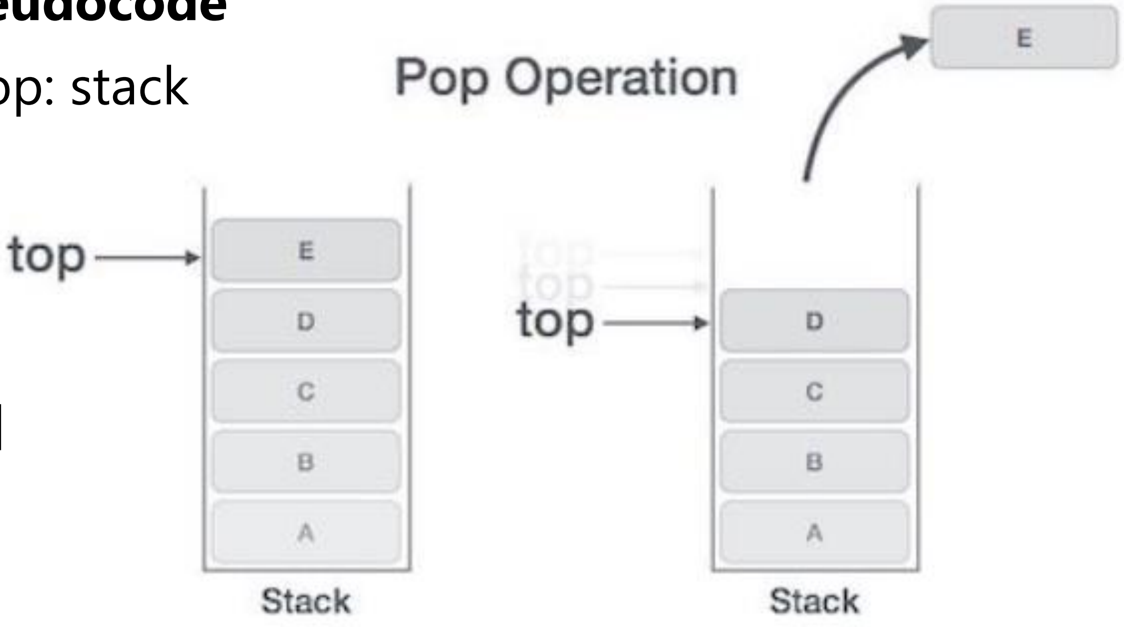
© e-Learning Centre, UCSC

3.1.1 Introduction to Stacks cont.

- **Pop Operation Algorithm**
  - Step 1 − Checks if the stack is empty.
  - Step 2 − If the stack is empty, produces an error and exit.
  - Step 3 − If the stack is not empty, accesses the data element at which top is pointing.
  - Step 4 − Decreases the value of top by 1.

- **Pop Operation Pseudocode**

```
begin procedure pop: stack
   if stack is empty
      return null
   endif
   data ← stack[top]
   top ← top - 1
   return data
end procedure
```



3.1.1 Introduction to Stacks cont.

- **IsEmptyStack Operation Pseudocode**

```
begin procedure IsEmptyStack

    if top less than 1
        return true
    else
        return false
    endif

end procedure
```

© e-Learning Centre, UCSC

3.1.1 Introduction to Stacks cont.

- **IsFullStack Operation Pseudocode**

```
begin procedure IsFullStack

   if top equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

3.1.1 Introduction to Stacks cont.

- **Top Operation Pseudocode**

```
begin procedure Top
   return stack[top]
end procedure
```

© e-Learning Centre, UCSC

3.1.1.1 Array based Stack implementation

- To push an item into an empty stack, we insert it at array location 0. (since all java arrays start at 0)

- To push the next item into the location 0 over the location to make room for new item.

© e-Learning Centre, UCSC

3.1.1.1 Array based Stack implementation Cont.

- This is easily done by defining an auxiliary integer variable known as stack pointer, which stores the array index currently being used as the top of the stack.
- A stack can be implemented with an array and an integer.
- The integer TOS (top of stack) provides the array index of the top element of the stack, when TOS is –1 , the stack is empty.

© e-Learning Centre, UCSC

Stacks specifies two data fields such as

- The array (which is expanded as needed stores the items in the stack)
- TopOfStack (TOS) (gives the index of the current top of the stack, if stack is empty, this index is –1.)

© e-Learning Centre, UCSC

Algorithms For Pushing & Popping

PUSH

- If stack is not full then
- Add 1 to the stack pointer.
- Store item at stack pointer location.

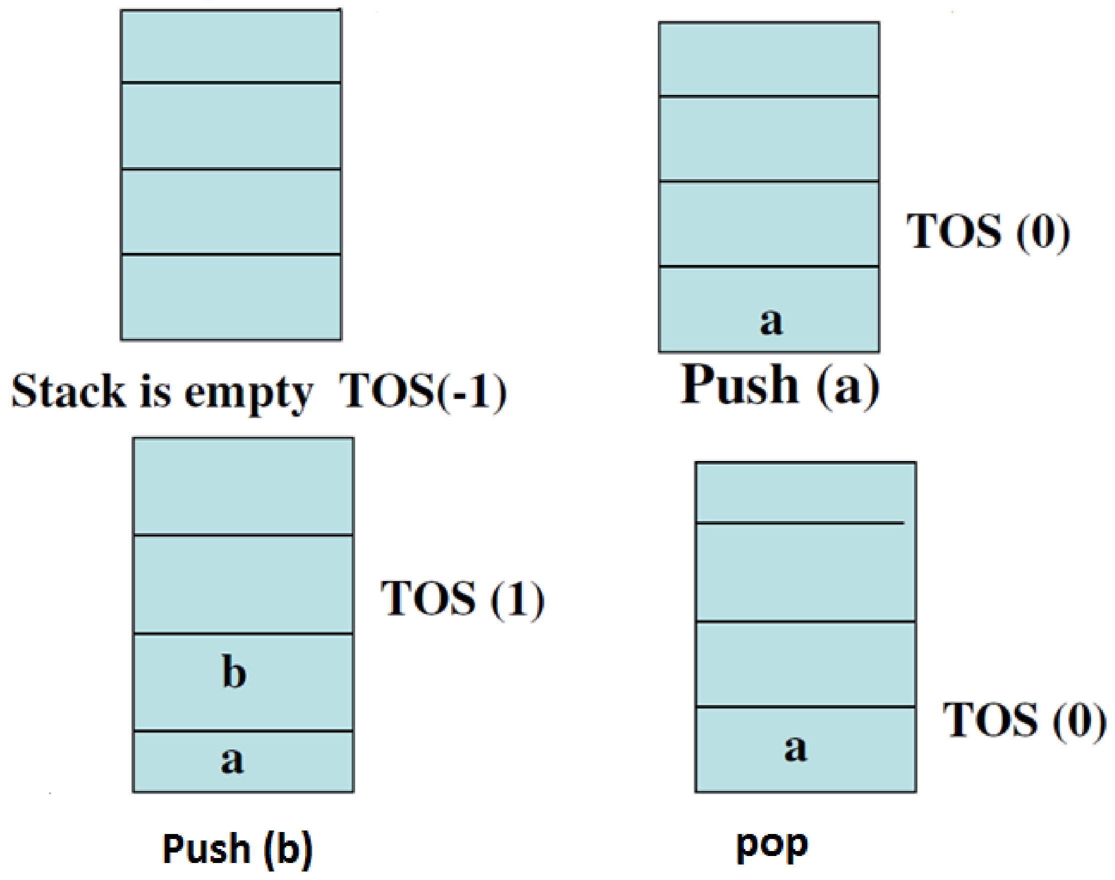
Algorithms For Pushing & Popping cont.

<u>POP</u>

- If stack is not empty then
- Read item at stack pointer location.
- Subtract 1 from the stack pointer.

How to stack routines work:

empty stack; push(a),push(b); pop



© e-Learning Centre, UCSC

Java implementation

Zero parameter constructor for array based stack

```
public stackar( )
/* construct the stack
{
 thearray= new object[default-capacity];
Tos = -1;
}
```

Isempty : returns true if stack is empty, fals otherwise

Isempty( )

```
public Boolean Isempty( )
{
return tos= = -1
}
```

Isfull( ) : returns true if stack is full , false otherwise

Isfull( )

```
public Boolean isfull( )
{
return tos= = stacksize-1; (default capacity)
}
```

push method for array based stack

Insert a new item into the stack

```
public void push (object x)
{
If isfull( )
throw new stackexception (“ stack is full”)
theArray[++tos]=x;
}
```

Pop method for array based stack

Remove the most recently inserted item from the stack

Exception underflow if the stack is empty

```
public void pop( ) throws underflow
{
If (isempty(  ))
throw new underflow (“stackpop”);
tos - - ;
}
```

Top method for array based stack

Return the most recently inserted item from the stack

Exception underflow if the stack is empty

```
public object top( ) throws underflow
{
if (isEmpty( ))
throw new underflow( “stacktop”)
return theArray[tos];
}
```

© e-Learning Centre, UCSC

Top and pop method for array based stack

Return and remove the most recently inserted item from the stack

Exception underflow if the stack is empty

```
public object topandpop( ) throws underflow
{
if isEmpty( ) )
throw new underflow (“stack topandpop”);
return theArray[tos - - ];
}
```

Java Code for a Stack

```
import java.io.*; // for I/O
class StackX
{
private int maxSize; // size of stack array
private double[ ] stackArray;
private int top; // top of stack
//--------------------------------------------------------------

public StackX(int s) // constructor
{
maxSize = s; // set array size
stackArray = new double[maxSize]; // create array
top = -1; // no items yet
}
//--------------------------------------------------------------
```

© e-Learning Centre, UCSC

Java Code for a Stack

```
public void push(double j) // put item on top of stack {
stackArray[++top] = j; // increment top, insert item
}
//--------------------------------------------------

public double pop() // take item from top of stack {
return stackArray[top--]; // access item, decrement top
}
//--------------------------------------------------

public double peek() // peek at top of stack {
return stackArray[top];
}
//--------------------------------------------------
-
```

Java Code for a Stack

```
//--------------------------------------------------------------
-
public boolean isFull() // true if stack is full
{
return (top == maxSize-1);
}
//--------------------------------------------------------------
-
} // end class StackX
```

Java Code for a Stack

```
class StackApp {
public static void main(String[] args)
{
StackX theStack = new StackX(10); // make new stack
theStack.push(20); // push items onto stack
theStack.push(40);
theStack.push(60);
theStack.push(80);
while( !theStack.isEmpty() ) // until it's empty,
{ // delete item from
stack double value = theStack.pop();
System.out.print(value); // display it
System.out.print(" ");
} // end while
System.out.println("");
} // end main()
} // end class StackApp
```

© e-Learning Centre, UCSC

Java Code for a Stack

The main() method in the StackApp class creates a stack that can hold 10 items, pushes 4 items onto the stack, and then displays all the items by popping them off the stack until it's empty.

Here's the output:

```
80 60 40 20
```

StackX Class Methods

The constructor creates a new stack of a size specified in its argument.

The fields of the stack comprise a variable to hold its maximum size (the size of the array), the array itself, and a variable top, which stores the index of the item on the top of the stack.
(Note that we need to specify a stack size only because the stack is implemented using an array. If it had been implemented using a linked list, for example, the size specification would be unnecessary.)

© e-Learning Centre, UCSC

StackX Class Methods

The **push()** method increments top so it points to the space just above the previous top, and stores a data item there. Notice that top is incremented before the item is inserted.

The **pop()** method returns the value at top and then decrements top. This effectively removes the item from the stack; it's inaccessible, although the value remains in the array (until another item is pushed into the cell).

The **peek()** method simply returns the value at top, without changing the stack.

The **isEmpty()** and isFull() methods return true if the stack is empty or full, respectively. The top variable is at –1 if the stack is empty and **maxSize-**1 if the stack is full.

© e-Learning Centre, UCSC

Error Handling

There are different philosophies about how to handle stack errors. What happens if you try to push an item onto a stack that's already full, or pop an item from a stack that's empty?

We've left the responsibility for handling such errors up to the class user. The user should always check to be sure the stack is not full before inserting an item:

Error Handling

```
if( !theStack.isFull() )
    insert(item);
else
    System.out.print("Can't insert, stack is full");
```

• In the interest of simplicity, we've left this code out of the main() routine (and anyway, in this simple program, we know the stack isn't full because it has just been initialized).

• We do include the check for an empty stack when main() calls pop().

© e-Learning Centre, UCSC

3.1.1.2 Linked List based Stack implementation

- The other way of implementing stacks is by using Linked lists.
- To implement a push, we create a new node in the list and attach it as the new first element.
- To implement a pop, we merely advance the top of the stack to the second item in the list (if there is one).
- An empty stack is represented by an empty linked list.
- Clearly, each operation is performed in constant time because, by restricting operations to the first node, we have made all calculations independent of the size of the list

© e-Learning Centre, UCSC

3.1.1.2 Linked List based Stack implementation cont.

- Skeleton for linked list-based stack class

```
public class ListStack<AnyType> implements Stack<AnyType>
{
public boolean isEmpty( ){
        return topOfStack == null;
         }
public void makeEmpty( ){
        topOfStack = null;
                }
public void push( AnyType x ) {}
public void pop( ) {}
public AnyType top( ) {}
public AnyType topAndPop( ) {}
private ListNode<AnyType> topOfStack = null;
}
```

3.1.1.2 Linked List based Stack implementation cont.

- Skeleton for linked list-based stack class

```
class ListNode<AnyType>
{
public ListNode( AnyType theElement ){
     this( theElement, null );
     }

public ListNode( AnyType theElement, ListNode<AnyType> n
){
     element = theElement; next = n;
     }

public AnyType element;
```

3.1.1.2 Linked List based Stack implementation cont.

- The push and pop routines for the ListStack class

```
public void push( AnyType x ){
    topOfStack = new ListNode<AnyType>( x, topOfStack );
    }

public void pop( ){
    if( isEmpty( ) )
        throw new UnderflowException( "ListStack pop" );
    topOfStack = topOfStack.next;
}
```

3.1.1.2 Linked List based Stack implementation cont.

- The top and topAndPop routines for the ListStack class

```
public AnyType top( ){
    if( isEmpty( ) )
        throw new UnderflowException( "ListStack top" );
    return topOfStack.element;
}
public AnyType topAndPop( ){
    if( isEmpty( ) )
        throw new UnderflowException( "ListStack topAndPop" );
    AnyType topItem = topOfStack.element;
    topOfStack = topOfStack.next;
    return topItem;
}
```

© e-Learning Centre, UCSC

3.1.2 Applications of Stacks

Following are some of the applications of stacks

Direct Applications

- Balancing of symbols
- Infix-to-postfix conversion
- Evaluation of postfix expression
- Implementing function calls (including recursion)
- Finding of spans (finding spans in stock markets, refer to Problems section)
- Page-visited history in a Web browser [Back Buttons]
- Undo sequence in a text editor
- Matching Tags in HTML and XML
- more direct applications using stacks?

© e-Learning Centre, UCSC

3.1.2 Applications of Stacks cont.

Indirect Applications

- Auxiliary data structure for other algorithms
  (Example: Tree traversal algorithms)
- Component of other data structures
  (Example: Simulating queues)

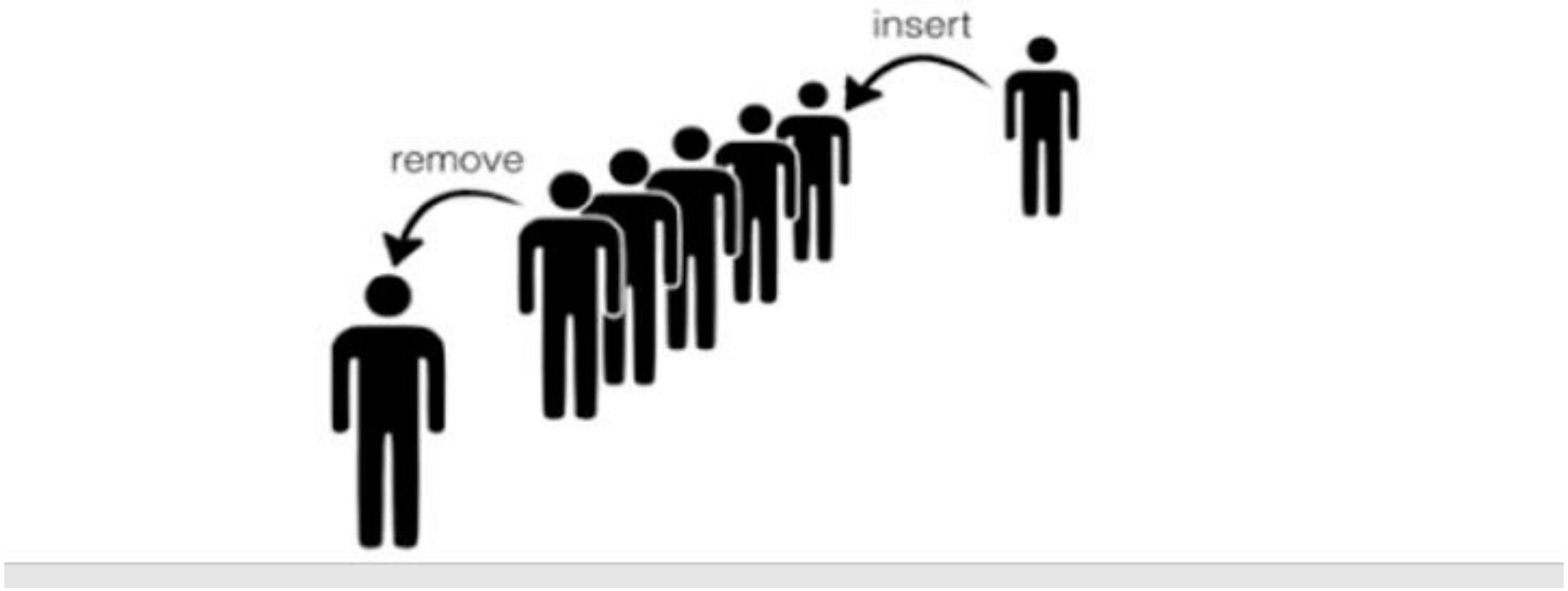
© e-Learning Centre, UCSC

3.2.1 Introduction to queues

- A queue is a data structure used for storing data (similar to Linked Lists and Stacks).
- In general, a queue is a line of people or things waiting to be served in sequential order starting at the beginning of the line or sequence.

Queues

LAST IN
LAST OUT
ONE WAY
FIRST IN
FIRST OUT



3.2.1 Introduction to queues cont.

- A queue is an ordered list in which **insertions** are done at one end (**rear**) and **deletions** are done at other end (**front**).

- The first element to be inserted is the first one to be deleted. Hence, it is called **First in First out** (FIFO) or Last in Last out (LILO) list.

- When an element is **inserted** in a queue, the concept is called **EnQueue**.

- When an element is **removed** from the queue, the concept is called **DeQueue**.

- DeQueueing an empty queue is called **underflow** and EnQueuing an element in a full queue is called **overflow**.

© e-Learning Centre, UCSC

3.2.1 Introduction to queues cont.

Main Queue Operations

- EnQueue(int data): Inserts an element at the end of the queue
- int DeQueue(): Removes and returns the element at the front of the queue

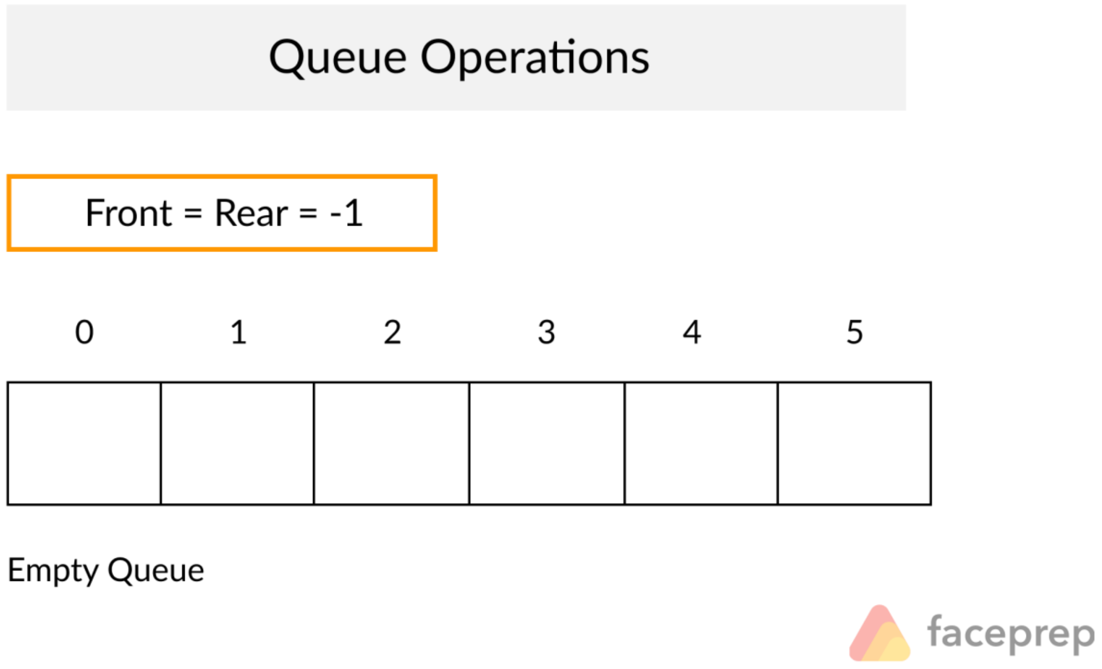
Auxiliary Queue Operations

- int Front(): Returns the element at the front without removing it
- int IsEmptyQueue: Indicates whether no elements are stored in the queue or not
- int QueueSize(): Returns the number of elements stored in the queue

© e-Learning Centre, UCSC

3.2.1 Introduction to queues cont.

How queue works



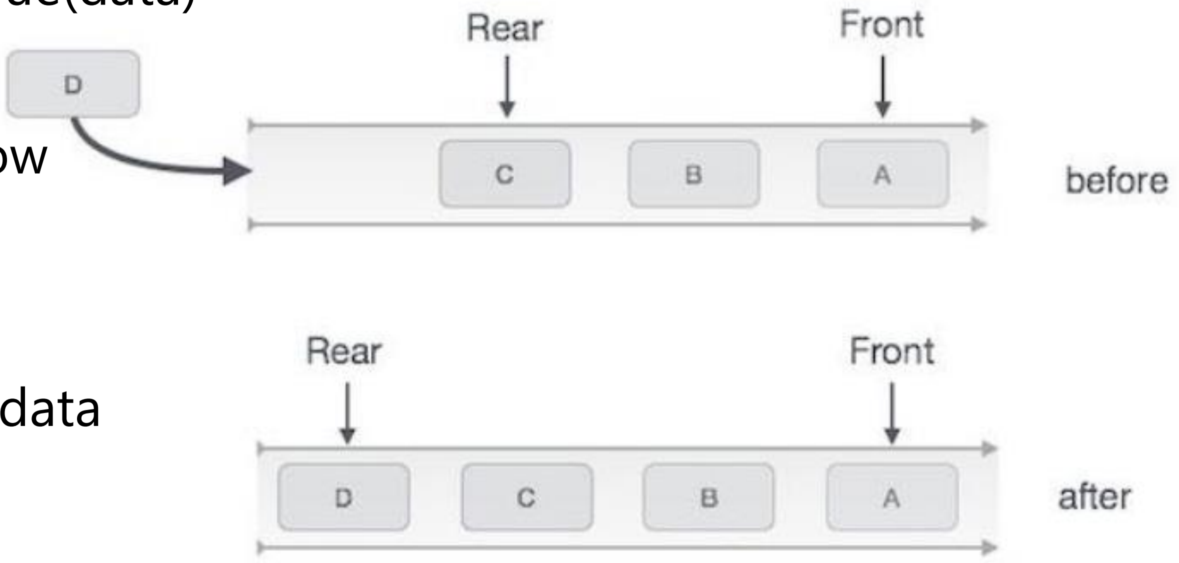
https://i1.faceprep.in/Companies-1/queue-operations.gif

© e-Learning Centre, UCSC

3.2.1 Introduction to queues cont.

- **EnQueue Operation Algorithm**
  - Step 1 − Check if the queue is full.
  - Step 2 − If the queue is full, produce overflow error and exit.
  - Step 3 − If the queue is not full, increment rear pointer to point the next empty space.
  - Step 4 − Add data element to the queue location, where the rear is pointing.
- **EnQueue Operation Pseudocode**

```
procedure enqueue(data)
  if queue is full
    return overflow
  endif
  rear ← rear + 1
  queue[rear] ← data
  return true
end procedure
```

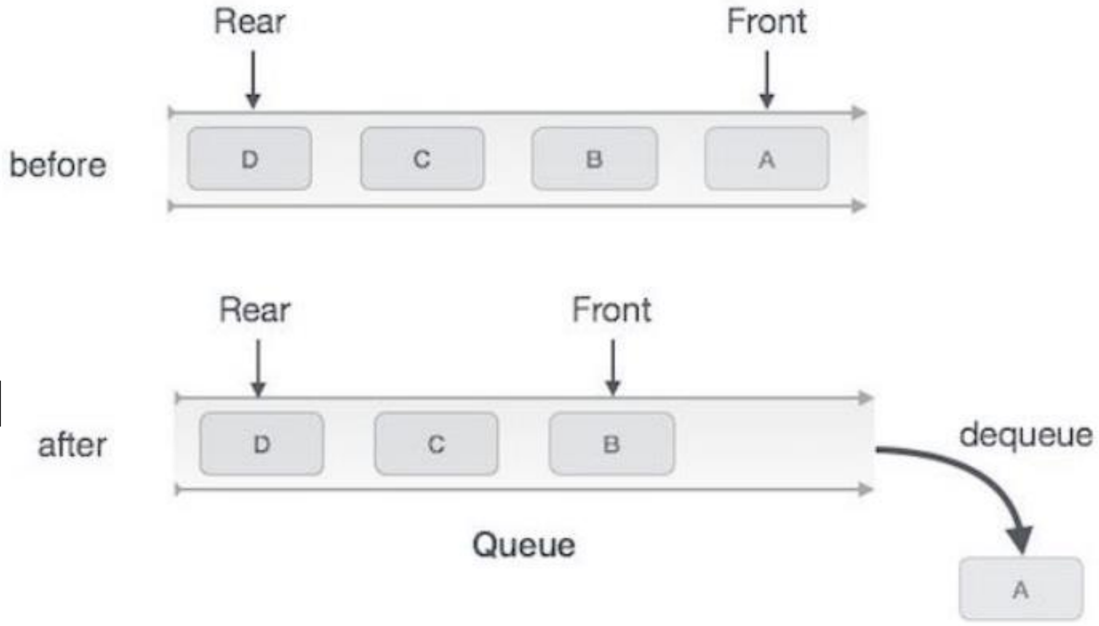


© e-Learning Centre, UCSC

3.2.1 Introduction to queues cont.

- **DeQueue Operation Algorithm**
    - Step 1 – Check if the queue is empty.
    - Step 2 – If the queue is empty, produce underflow error and exit.
    - Step 3 – If the queue is not empty, access the data where front is pointing.
    - Step 4 – Increment front pointer to point to the next available data element.
- **DeQueue Operation Pseudocode**

```
procedure dequeue
  if queue is empty
    return underflow
  end if
  data = queue[front]
  front ← front + 1
  return true
end procedure
```



3.2.1 Introduction to queues cont.

- **IsEmptyQueue Operation Algorithm**

```
begin procedure IsEmptyQueue

   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif

end procedure
```

3.2.1 Introduction to queues cont.

- **Front Operation Algorithm**

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

3.2.1.1 Array based Queue implementation

- The queue.java program features a Queue class with insert(), remove(), peek(), isFull(), isEmpty(), and size() methods.
- The main() program creates a queue of five cells, inserts four items, removes three items, and inserts four more. The sixth insertion invokes the wraparound feature. All the items are then removed and displayed.
- The output looks like this:
- **40 50 60 70 80**

The Queue.java Program

```
import java.io.*; // for I/O
class Queue {
private int maxSize;
private int[] queArray;
private int front;
private int rear;
private int nItems;
//-------------------------------------------------------------
```

The Queue.java Program Cont.

```
public Queue(int s) // constructor
{
maxSize = s;
queArray = new int[maxSize];
front = 0;
rear = -1;
nItems = 0;
}
//------------------------------------------------------
```

© e-Learning Centre, UCSC

The Queue.java Program

```
public void insert(int j) // put item at rear of queue
{
if(rear == maxSize-1) // deal with wraparound
rear = -1;
queArray[++rear] = j; // increment rear and
insert
nItems++; // one more item
}
//--------------------------------------------------
```

The Queue.java Program Cont.

```
public int remove() // take item from front of queue
{
int temp = queArray[front++]; // get value and incr front
if(front == maxSize) // deal with wraparound
front = 0;
nItems--; // one less item
return temp;
}
//--------------------------------------------------------------
```

The Queue.java Program

```
public int peekFront() { // peek at front of queue
return queArray[front];
}
//--------------------------------------------------------------
public boolean isEmpty() { // true if queue is empty
return (nItems==0);
}
//--------------------------------------------------------------
public boolean isFull() { // true if queue is full
return (nItems==maxSize);
}
```

The Queue.java Program Cont.

```
//-------------------------------------------------
 public int size() { // number of items in queue
 return nItems;
 }
 //-------------------------------------------------
 } // end class Queue
```

The Queue.java Program

```
class QueueApp {
public static void main(String[] args) {
Queue theQueue = new Queue(5); // queue holds 5 items
theQueue.insert(10); // insert 4 items
theQueue.insert(20);
theQueue.insert(30);
theQueue.insert(40);
theQueue.remove(); // remove 3 items
theQueue.remove(); // (10, 20, 30)
theQueue.remove();
theQueue.insert(50); // insert 4 more items
theQueue.insert(60); // (wraps around)
theQueue.insert(70);
theQueue.insert(80);
while( !theQueue.isEmpty() ) { // remove and display all items
int n = theQueue.remove(); // (40, 50, 60, 70, 80)
System.out.print(n);
```

© e-Learning Centre, UCSC

The Queue.java Program

```
System.out.print(" ");
}
System.out.println("");
} // end main()
} // end class QueueApp
```

3.2.1.2 Linked List based Queue implementation

- The ListQueue class is similar to the ListStack class.
- The only new thing here is that we maintain two references instead of one.

3.2.1.2 Linked List based Queue implementation cont.

- **Skeleton for the linked list-based queue class.**

```
public class ListQueue<AnyType>
{
public ListQueue( ){}
public boolean isEmpty( ){}
public void enqueue( AnyType x ){}
public AnyType dequeue( ){}
public AnyType getFront( ){}
public void makeEmpty( ){}

private ListNode<AnyType> front;
private ListNode<AnyType> back;
}
```

© e-Learning Centre, UCSC

3.2.1.2 Linked List based Queue implementation cont.

- **Constructor for the linked list-based ListQueue class**

```
public ListQueue( ){
     front = back = null;
}
```

3.2.1.2 Linked List based Queue implementation cont.

- **The enqueue and dequeue routines for the ListQueue class**

```
public void enqueue( AnyType x ) {
    if( isEmpty( ) )
        back = front = new ListNode<AnyType>( x );
    else
        back = back.next = new ListNode<AnyType>( x );
}
public AnyType dequeue( ){
    if( isEmpty( ) )
        throw new UnderflowException( "ListQueue dequeue" );
    AnyType returnValue = front.element;
    front = front.next;
    return returnValue;
}
```

© e-Learning Centre, UCSC

3.2.1.2 Linked List based Queue implementation cont.

- **Supporting routines for the ListQueue class**

```
public AnyType getFront( ){
    if( isEmpty( ) )
        throw new UnderflowException( "ListQueue getFront" );
    return front.element;
}
public void makeEmpty( ){
    front = null;
    back = null;
}
public boolean isEmpty( ){
    return front == null;
}
```

3.2.2 Applications of queues

Following are some of the applications of queues

Direct Applications

- Operating systems schedule jobs (with equal priority) in the order of arrival (e.g., a print queue).
- Simulation of real-world queues such as lines at a ticket counter or any other first-come first-served scenario requires a queue.
- Multiprogramming.
- Asynchronous data transfer (file IO, pipes, sockets).
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

Indirect Applications

- Auxiliary data structure for algorithms
- Component of other data structures

3.2.3 Circular queue

- When we insert a new item in the queue in the Workshop applet, the Front arrow moves upward, toward higher numbers in the array.
- When we remove an item, Rear also moves upward.
- we may find the arrangement counter-intuitive, because the people in a line at the movies all move forward, toward the front, when a person leaves the line.
- We could move all the items in a queue whenever we deleted one, but that wouldn't be very efficient.
- Instead we keep all the items in the same place and move the front and rear of the queue.

© e-Learning Centre, UCSC

A Circular Queue cont.

- To avoid the problem of not being able to insert more items into the queue even when it's not full, the Front and Rear arrows wrap around to the beginning of the array. The result is a circular queue.

© e-Learning Centre, UCSC

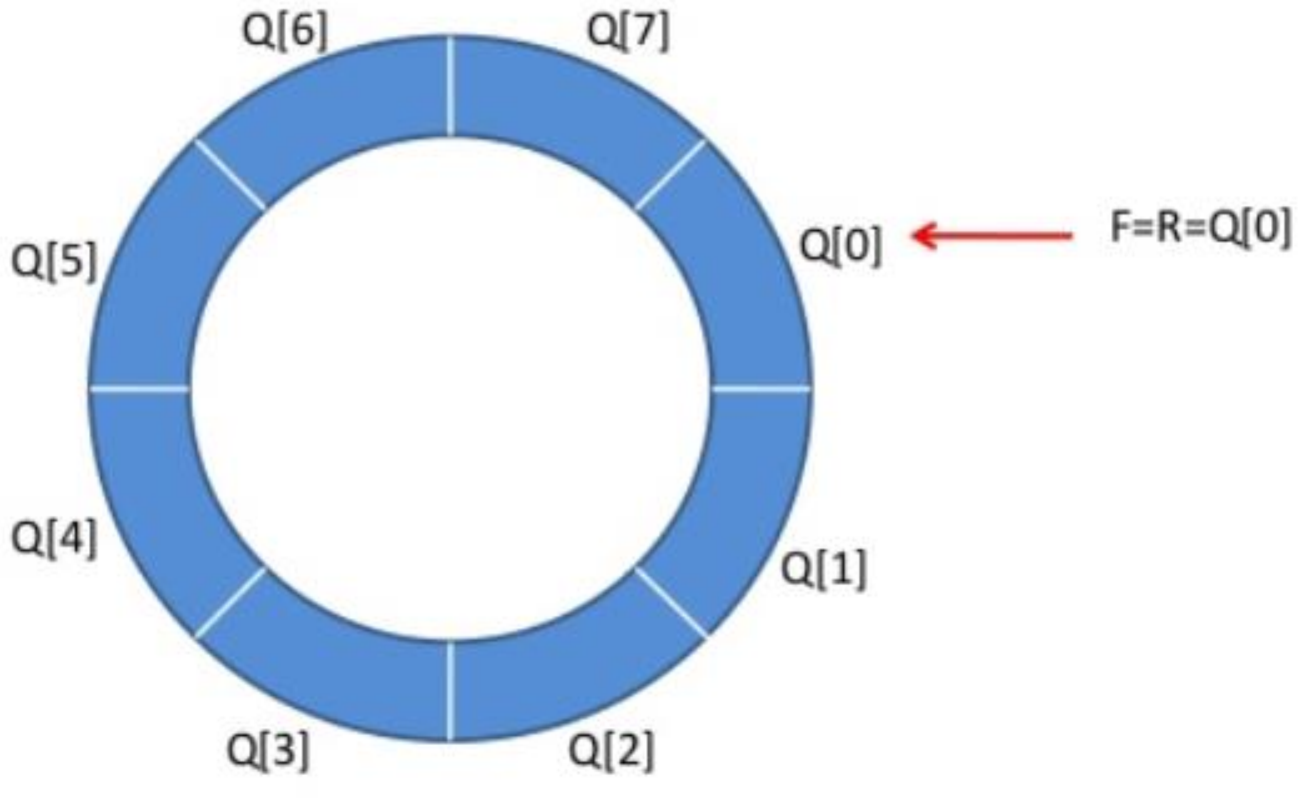
Circular Queue Cont.

- There are two problems associated with linear queue.
- They are:
- Time consuming: linear time to be spent in shifting the elements to the beginning of the queue.
- Signaling (indicating) queue full: even if the queue is having vacant position.

© e-Learning Centre, UCSC

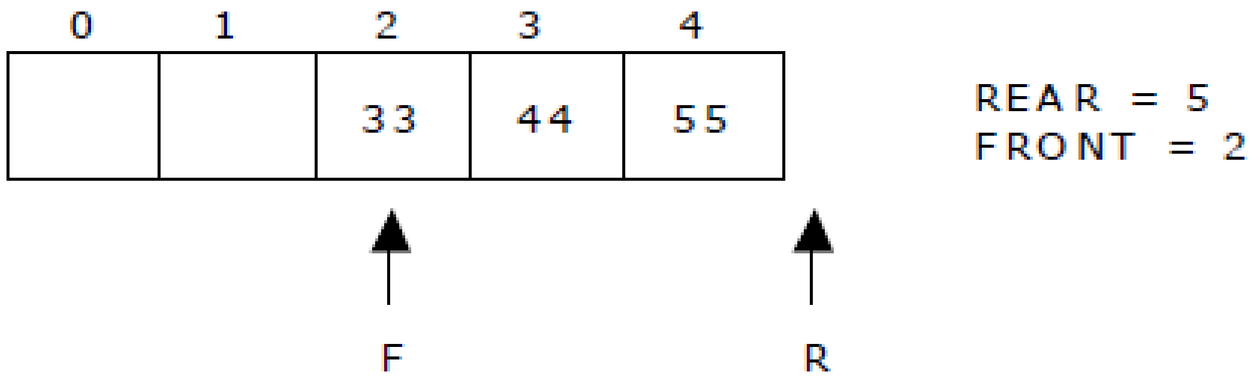
Circular Queue Cont.

- e.g.:



© e-Learning Centre, UCSC

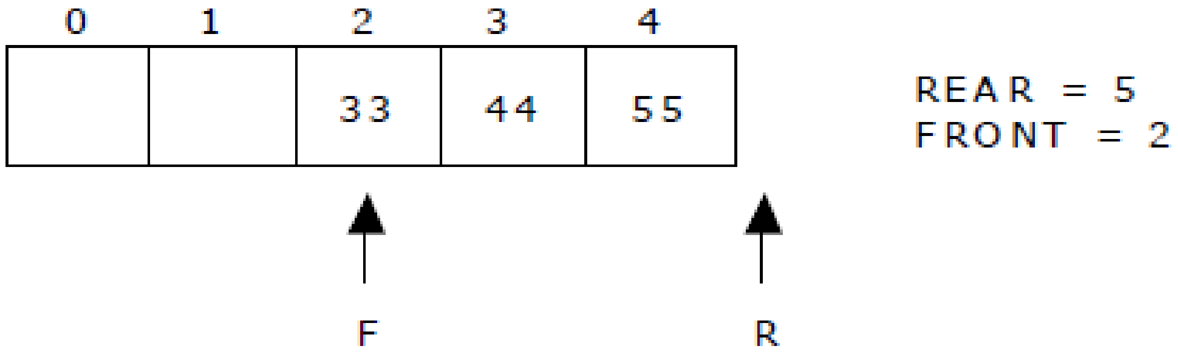
For example, let us consider a linear



- If one wants to insert 66 to the queue.
- The rear crossed the maximum size of the queue (i.e., 5).
- There will be queue full signal.
- The queue status is as follows:

For example, let us consider a linear

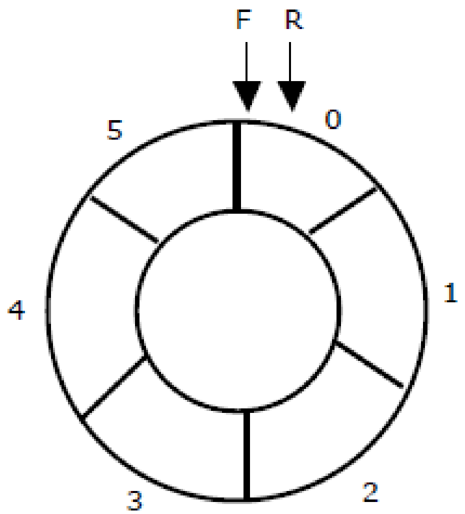
- The queue status is as follows:



- This difficulty can be overcome if we treat queue position with index zero as a position
- that comes after position with index four then we treat the queue as a **circular queue.**

Representation of Circular Queue cont..

- Let us consider a circular queue, which can hold maximum (MAX) of six elements.
- Initially the queue is empty.



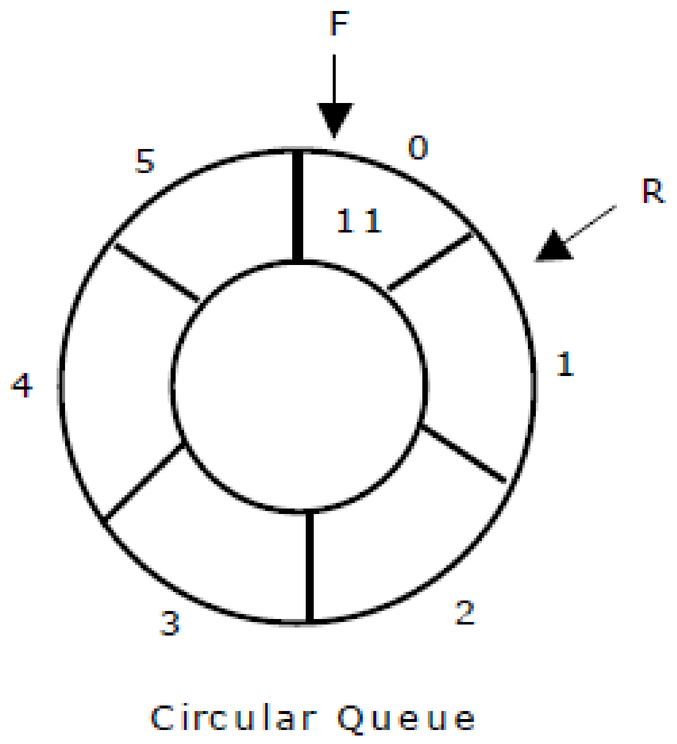
Circular Queue

Queue Empty
MAX = 6
FRONT = REAR = 0
COUNT = 0

© e-Learning Centre, UCSC

Representation of Circular Queue cont.

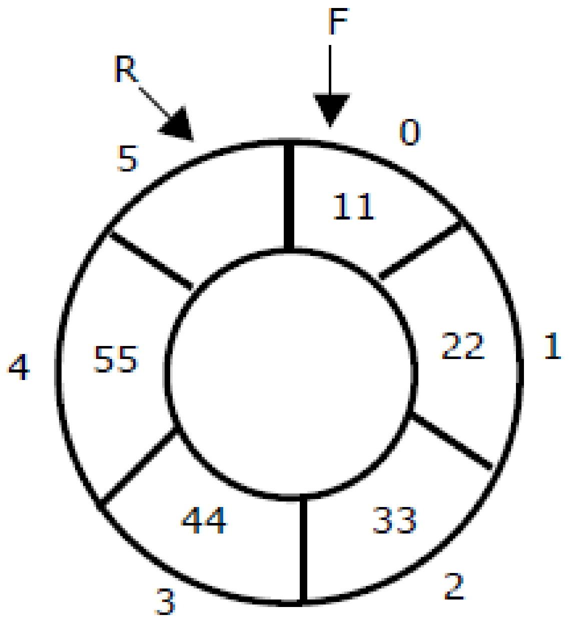
- Now, insert 11 to the circular queue. Then circular queue status will be:



```
FRONT = 0
REAR = (REAR + 1) % 6 = 1
COUNT = 1
```

Representation of Circular Queue cont..

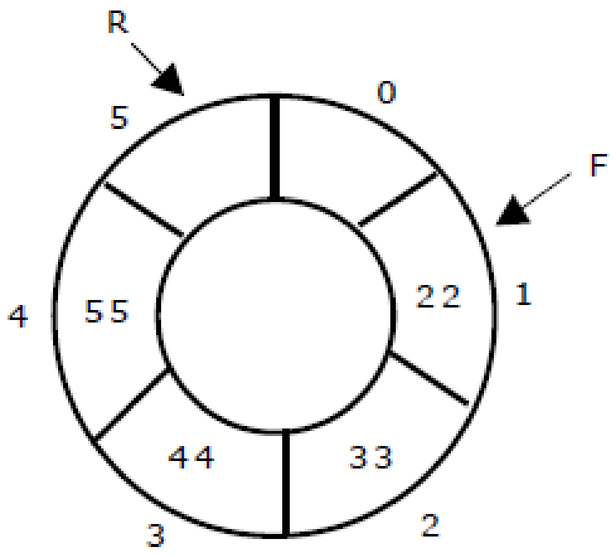
- Insert new elements 22, 33, 44 and 55 into the circular queue



FRONT = 0
REAR = (REAR + 1) % 6 = 5
COUNT = 5

Representation of Circular Queue cont.

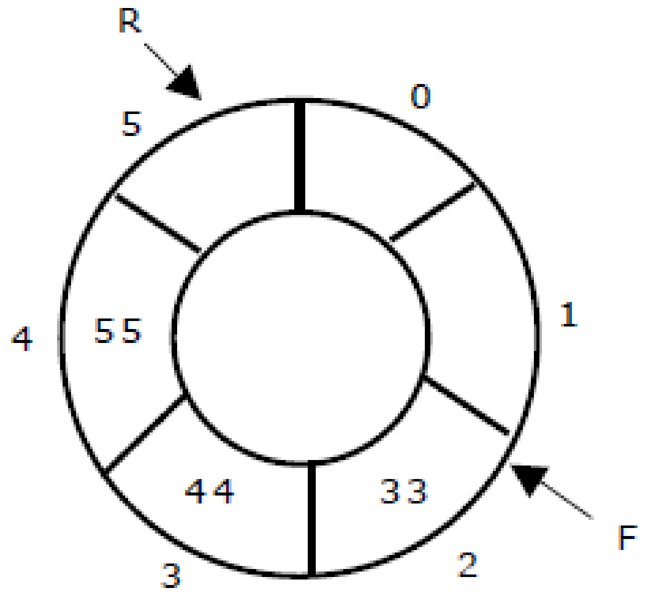
- Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted



```
FRONT = (FRONT + 1) % 6 = 1
REAR = 5
COUNT = COUNT - 1 = 4
```

Representation of Circular Queue cont.

- Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted.

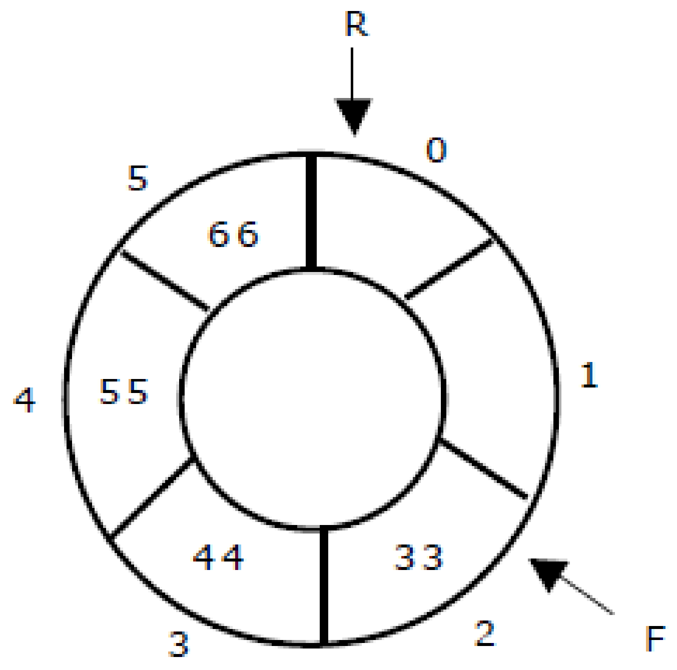


```
FRONT = (FRONT + 1) % 6 = 2
REAR = 5
COUNT = COUNT - 1 = 3
```

© e-Learning Centre, UCSC

Representation of Circular Queue cont.

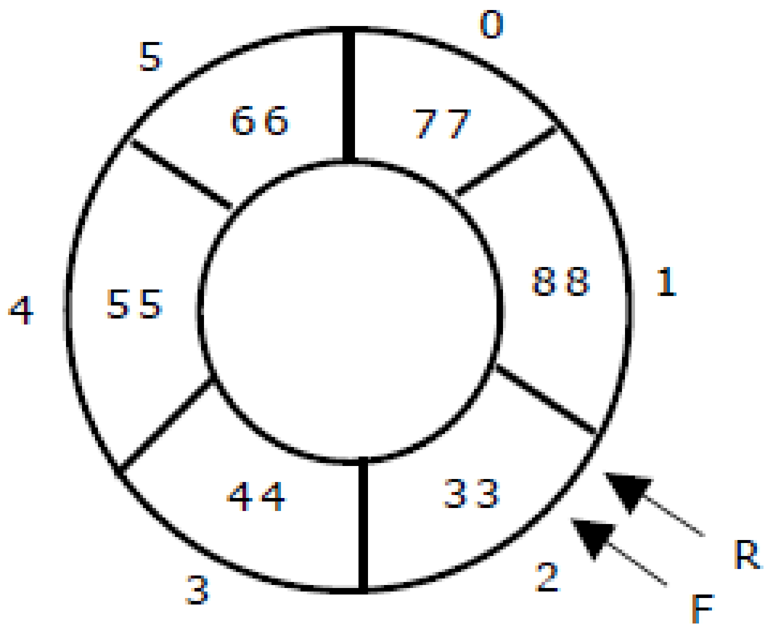
- Again, insert another element 66 to the circular queue



```
FRONT = 2
REAR = (REAR + 1) % 6 = 0
COUNT = COUNT + 1 = 4
```

Representation of Circular Queue cont..

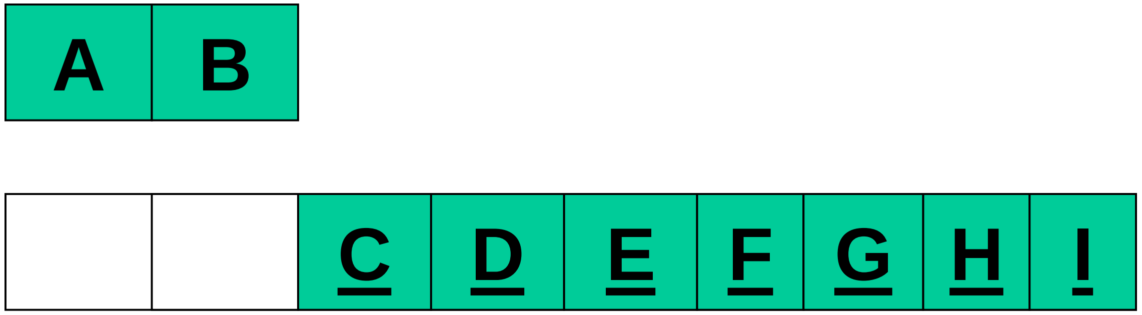
- Now, insert new elements 77 and 88 into the circular queue



FRONT = 2, REAR = 2
REAR = REAR % 6 = 2
COUNT = 6

Wraparound routine(increment)

```
//internal method to increment with wraparound
// param x any index in theArray's range.
// return x+1, or 0, if x is at the end of the array.
Private int increment( int x)
{
If (++x==theArray.length)     //maxarray-1
x=0
return x;
}
```



© e-Learning Centre, UCSC

Wraparound routine(increment)

The increment routine adds 1 to its parameter and returns the new value. Since it implements wraparound, if the result would equal the array size it is wraparound to zero.

© e-Learning Centre, UCSC

Representation of Circular Queue cont..

- Now, if we insert an element to the circular queue, as COUNT = maxarray-1, we cannot add the element to circular queue.
- So, the circular queue is *full*.

© e-Learning Centre, UCSC

3.2.4. Priority Queues

- A priority queue is a more specialized data structure than a stack or a queue.
- However, it's a useful tool in a surprising number of situations.
- Like an ordinary queue, a priority queue has a front and a rear, and items are removed from the front.
- However, in a priority queue, items are ordered by key value, so that the item with the lowest key (or in some implementations the highest key) is always at the front.
- Items are inserted in the proper position to maintain the order.

© e-Learning Centre, UCSC

Efficiency of Priority Queues

- In the priority-queue implementation we show here, insertion runs in O(N) time, while deletion takes O(1) time.

Summary

3.1	A stack is a linear data structure in which elements are added and removed only from one end. **LIFO**
3.2	A queue is a linear data structure in which the element that is inserted first is the first one to be taken out. **FIFO**
3.2.4	A priority queue is a data structure in which each element is assigned a priority.

© e-Learning Centre, UCSC