
RGPVNOTES.IN

Program : **B.Tech**

Subject Name: **Data Structure**

Subject Code: **CS-303**

Semester: **3rd**

# Data Structure Lecture Notes

## Unit -2

**STACKS:**

**STACKS AS ADT:**

A stack is a linear data structure in which an element may be inserted or deleted only at one end called the top end of the stack i.e. the elements are removed from a stack in the reverse order of that in which they were inserted into the stack.

A stack follows the principle of last-in-first-out (LIFO) system. According to the stack terminology, PUSH and POP are two terms used for insert and delete operations.

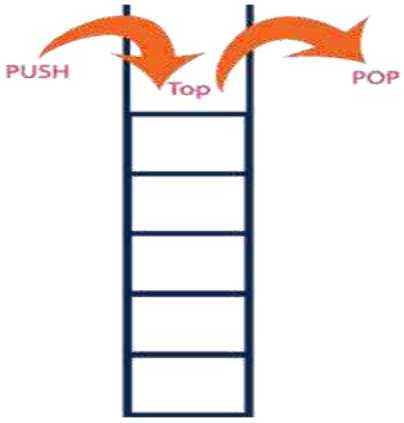




**Figure 2.1: Stack**

In a stack, the insertion operation is performed using a function called "push" and deletion operation is performed using a function POP.

In the figure 2.1, PUSH and POP operations are performed at top position in the stack. That means, both the insertion and deletion operations are performed at one end (i.e., at Top)

## Representation of Stacks

A stack may be represented by means of a one way list or a linear array. Unless, otherwise stated, each of the stacks will be maintained by a linear array STACK; a variable TOP contains the location of the top element of the stack. A variable N gives the maximum number elements that can be held by the stack. The condition where TOP is NULL, indicate that the stack is empty. The condition where TOP is N, will indicate that the stack is full.

**DIFFERENT IMPLEMENTATION OF STACK:**

Stack data structure can be implementing in two ways. They are as follows...

1. Using Array
2. Using Linked List

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

**Stack Using Array:**

A stack data structure can be implemented using one dimensional array. But stack implemented using array, can store only fixed number of data values. This implementation is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using LIFO principle with the help of a variable 'top'. Initially top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

**Stack Operations using Array:**

1. **push(value) - Inserting value into the stack**
   In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at top position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack...

- Step 1: Check whether stack is FULL. (top == SIZE-1)
- Step 2: If it is FULL, then display "Stack is FULL!!! Insertion is not possible!!!" and terminate the function.
- Step 3: If it is NOT FULL, then increment top value by one (top++) and set stack[top] to value (stack[top] = value).

2. **pop() - Delete a value from the Stack**
   In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from top position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

- Step 1: Check whether stack is EMPTY. (top == -1)
- Step 2: If it is EMPTY, then display "Stack is EMPTY!!! Deletion is not possible!!!" and terminate the function.
- Step 3: If it is NOT EMPTY, then delete stack[top] and decrement top value by one (top--).

3. **display() - Displays the elements of a Stack**
   We can use the following steps to display the elements of a stack...

- Step 1: Check whether stack is EMPTY. (top == -1)
- Step 2: If it is EMPTY, then display "Stack is EMPTY!!!" and terminate the function.
- Step 3: If it is NOT EMPTY, then define a variable 'i' and initialize with top. Display stack[i] value and decrement i value by one (i--).
- Step 4: Repeat above step until i value becomes '0'.

**MULTIPLE STACKS:**

When a stack is created using single array, we cannot able to store large amount of data, thus this problem is rectified using more than one stack in the same array of sufficient array. This technique is called as Multiple Stack.

In order to maintain 2 stacks, there should be 2 top variables to indicate the top of both the stacks. Both the stacks can grow up to any extent from 1$^{st}$ position to maximum, hence there should be one

more variable to keep track of the total number of values stored. The overflow condition will appear.If count becomes equal to or greater than array size and the underflow condition from empty stack will appear if count=0.

The structure for such implementation can be given as:

```
struct multistack
{
        Int top0,top1;
        Int count;
        Int num;
};
```

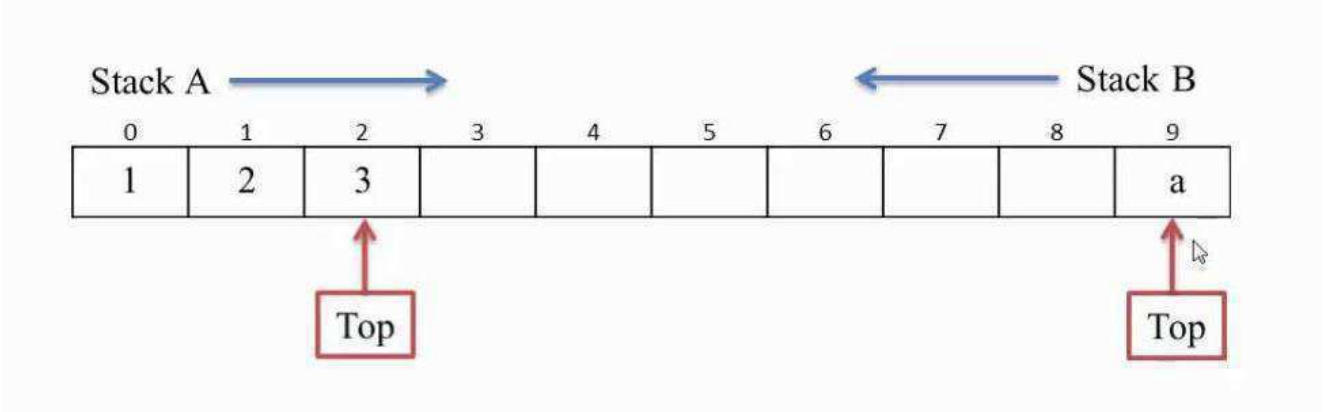


**Figure 2.2: Multiple Stacks**

**APPLICATION OF STACK:**



There are two important applications of stacks.

- Recursion
- Reversing a String
- Calculation of Arithmetic Expression

**Recursion:**

Recursion means function calling itself.

A function is called recursive if the function definition refers to itself or does refer to another function which in turn refers back to the same function. In-order for the definition not to be circular, it must have the following properties:

(i) There must be certain arguments called base values, for which the function does not refer to itself.

(ii) Each time the function does refer to itself, the argument of the function must be closer to a base value.

A recursive function with those two properties is said to be well defined.

Let us consider the factorial of a number and its algorithm described recursively:

We know that N! = N * (N-1)!

(N-1)! = N-1 * (N-2)! and so on up to 1.

FACT(N)

```
1.    if N=1
          return 1
2.    else
      return N * FACT(N-1)
3.    end
```

Let N be 5.

Then according to the definition FACT(5) will call FACT(4), FACT(4) will call FACT(3), FACT(3) will call FACT(2), FACT(2) will call FACT(1). Then the execution will return back by finishing the execution of FACT(1), then FACT(2) and so on up to FACT(5) as described below.

```
1)    5! = 5 * 4!
2)          4! = 4 * 3!
3)                3! = 3 * 2!
4)                      2! = 2 * 1!
5)                            1! = 1
6)                      2! = 2 * 1 = 2
7)                3! = 3 * 2 = 6
8)          4! = 4 * 6 = 24
9)    5! = 5 * 24 = 120
```

**Calculation of Arithmetic Expression:**

An expression is a collection of operators and operands that represents a specific value. This section deals with the mechanical evaluation or compilation of infix expression. The stack is find to be more efficient to evaluate an infix arithmetical expression by first converting to a suffix or postfix expression and then evaluating the suffix expression. This approach will eliminate the repeated scanning of infix expressions in order to obtain its value.

A normal arithmetic expression is normally called as infix expression. E.g A+B
A Polish mathematician found a way to represent the same expression called polish notation or prefix expression by keeping operators as prefix. E.g +AB

We use the reverse way of the above expression for our evaluation. The representation is called Reverse Polish Notation (RPN) or postfix expression. E.g. AB+

**CONVERSION OF INFIX TO POSTFIX NOTATION USING STACK:**

Any expression can be represented using three types of expressions (Infix, Postfix and Prefix). We can also convert one type of expression to another type of expression like Infix to Postfix, Infix to Prefix, Postfix to Prefix and vice versa.

To convert any Infix expression into Postfix or Prefix expression we can use the following procedure...

- Find all the operators in the given Infix Expression.
- Find the order of operators evaluated according to their Operator precedence.
- Convert each operator into required type of expression (Postfix or Prefix) in the same order.

Example:

Consider the following Infix Expression to be converted into Postfix Expression...

**D = A + B * C**

- Step 1: The Operators in the given Infix Expression : = , + , *
- Step 2: The Order of Operators according to their preference : * , + , =
- Step 3: Now, convert the first operator * ----- D = A + B C *
- Step 4: Convert the next operator + ----- D = A BC* +
- Step 5: Convert the next operator = ----- D ABC*+ =

Finally, given Infix Expression is converted into Postfix Expression as follows...

**D A B C * + =**

**EVALUATION OF POSTFIX EXPRESSION:**

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

- Read all the symbols one by one from left to right in the given Postfix Expression
- If the reading symbol is operand, then push it on to the Stack.
- If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped oparands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
- Finally! perform a pop operation and display the popped value as final result.

**RECURSION:**

Recursion is a programming technique that allows the programmer to express operations in terms of themselves. In C, this takes the form of a function that calls itself. A useful way to think of recursive functions is to imagine them as a process being performed where one of the instructions is to "repeat the process".

A simple example of recursion would be:

```
void recurse()
{
   recurse(); /* Function calls itself */
}

int main()
{
   recurse(); /* Sets off the recursion */
   return 0;
}
```

**QUEUE:**

A queue is a sequential storage structure that permits access only at the two ends of the sequence. We refer to the ends of the sequence as the front and rear. A queue inserts new elements at the rear and removes elements from the front of the sequence. You will note that a queue removes elements in the same order in which they were stored, and hence a queue provides FIFO (first-in / first-out), or FCFS (first-come / first-served), ordering.

**Operations on Queue:**

Mainly the following four basic operations are performed on queue:

**enQueue(value) - Inserting value into the queue**

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at rear position. The enQueue() function takes one integer value as parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

- Step 1: Check whether queue is FULL. (rear == SIZE-1)
- Step 2: If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.
- Step 3: If it is NOT FULL, then increment rear value by one (rear++) and set queue[rear] = value.

**deQueue() - Deleting a value from the Queue**

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

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

**display() - Displays the elements of a Queue**

We can use the following steps to display the elements of a queue...

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

**Front:** Get the front item from queue.
**Rear:** Get the last item from queue.

**Graphical Presentation:**

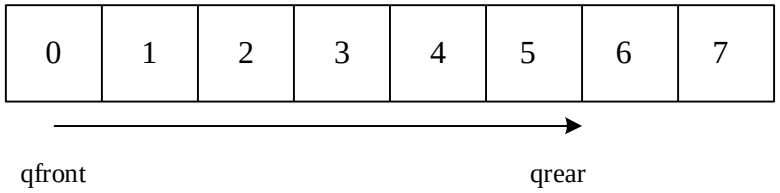


**Figure 2.3: Queue**

**Time Complexity:**

Time complexity of all operations like enqueue(), dequeue(), isFull(), isEmpty(), front() and rear() is O(1). There is no loop in any of the operations.

**QUEUES AS ADT:**

In case of Queue we know that what to implement but how to implement is not known, hence queue is called ADT.

**DIFFERENT IMPLEMENTATION OF QUEUE:**

- Circular Queue.
- Priority Queue.
- Dqueue

**CIRCULAR QUEUE:**

Circular Queue is also a linear data structure, which follows the principle of **FIFO**(First In First Out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.

**Basic features of Circular Queue**

In case of a circular queue, head pointer will always point to the front of the queue, and tail pointer will always point to the end of the queue.

Initially, the head and the tail pointers will be pointing to the same location, this would mean that the queue is empty.

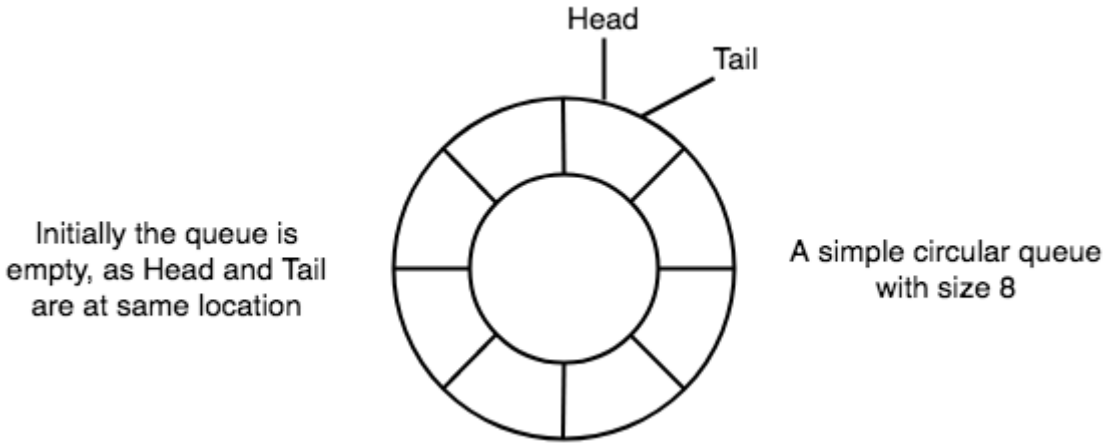


**Figure 2.4: Circular Queue**

New data is always added to the location pointed by the tail pointer, and once the data is added, tail pointer is incremented to point to the next available location.

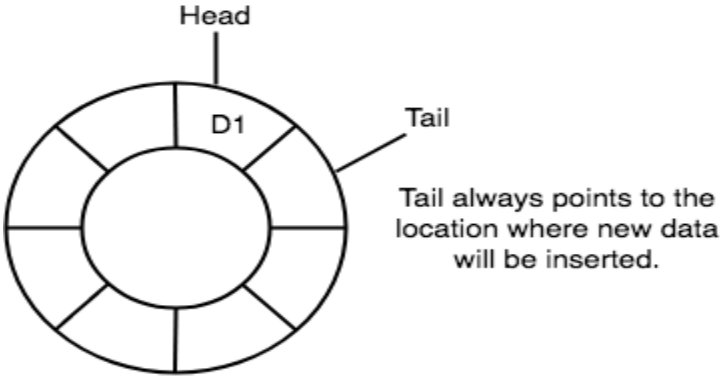


**Figure 2.5: Circular Queue Insertion**

In a circular queue, data is not actually removed from the queue. Only the head pointer is incremented by one position when **dequeue** is executed. As the queue data is only the data between head and tail, hence the data left outside is not a part of the queue anymore, hence removed.

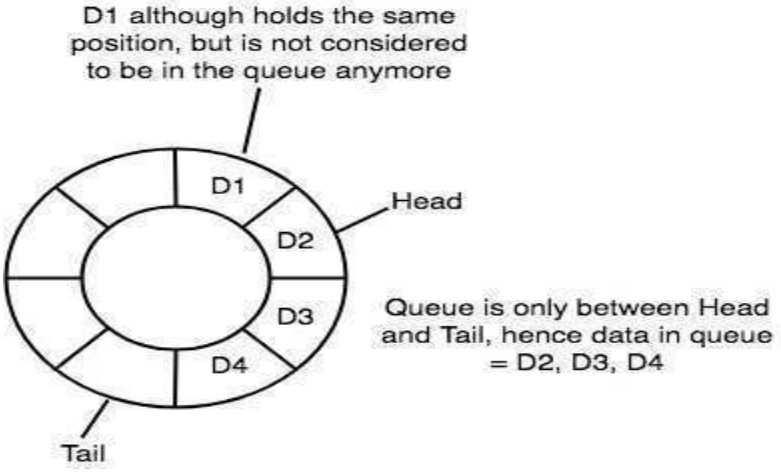


**Figure 2.6: Circular Queue Insertion**

The head and the tail pointer will get reinitialized to **0** every time they reach the end of the queue.

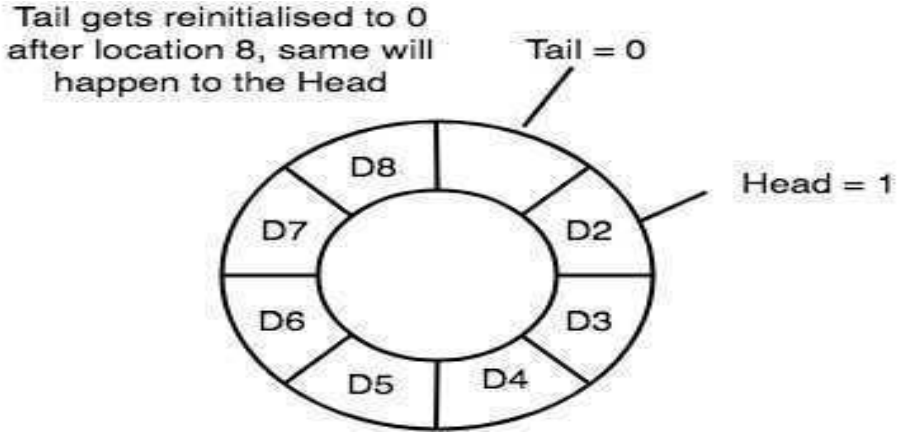


**Figure 2.7: Circular Queue Insertion**

Also, the head and the tail pointers can cross each other. In other words, head pointer can be greater than the tail. Sounds odd? This will happen when we dequeue the queue a couple of times and the tail pointer gets reinitialized upon reaching the end of the queue.

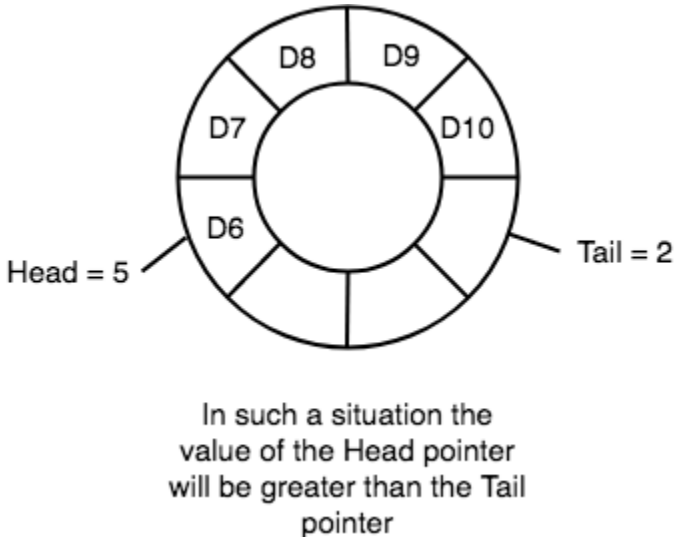


**Figure 2.8: Circular Queue deletion**

**Application of Circular Queue**

- Computer controlled Traffic Signal System uses circular queue.
- CPU scheduling and Memory management.

**CONCEPT OF DQUEUE:**

A **dqueue**, also known as a double-ended queue, is an ordered collection of items similar to the queue. It has two ends, a front and a rear, and the items remain positioned in the collection. New items can be added at either the front or the rear. Likewise, existing items can be removed from either end. In a sense, this hybrid linear structure provides all the capabilities of stacks and queues in a single data structure.

It is important to note that even though the deque can assume many of the characteristics of stacks and queues, it does not require the LIFO and FIFO orderings that are enforced by those data structures. It is up to you to make consistent use of the addition and removal operations.

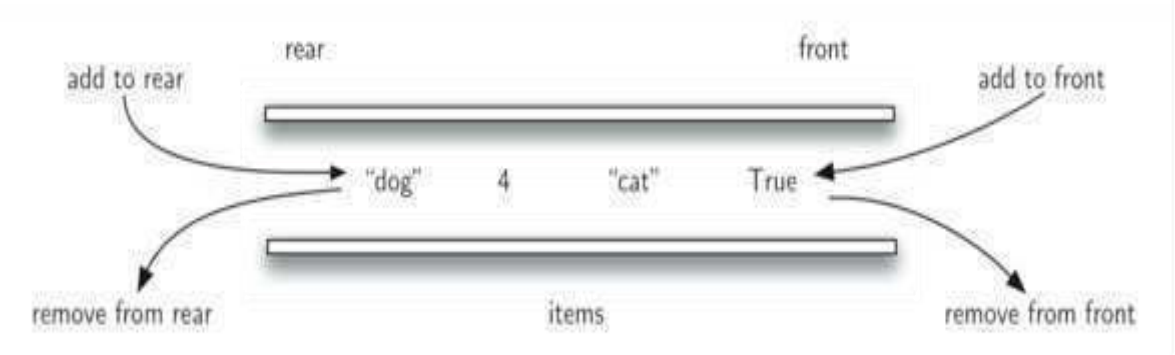


**Figure 2.9: Dqueue**

**Operation on Deque:**

- **insertFront():** Adds an item at the front of Deque.
- **insertLast():** Adds an item at the rear of Deque.
- **deleteFront():** Deletes an item from front of Deque.
- **deleteLast():** Deletes an item from rear of Deque.

In addition to above operations, following operations are also supported

- **getFront():** Gets the front item from queue.
- **getRear():** Gets the last item from queue.
- **isEmpty():** Checks whether Deque is empty or not.
- **isFull():** Checks whether Deque is full or not.

**PRIORITY QUEUE:**

Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

**Basic Operations**

- insert / enqueue − add an item to the rear of the queue.
- remove / dequeue − remove an item from the front of the queue.



**Priority Queue Representation**

We're going to implement Queue using array in this article. There is few more operations supported by queue which are following.

- **Peek** − get the element at front of the queue.
- **isFull** − check if queue is full.
- **isEmpty** − check if queue is empty.

**Insert / Enqueue Operation**

Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.

**Remove / Dequeue Operation**

Whenever an element is to be removed from queue, queue get the element using item count. Once element is removed. Item count is reduced by one.

**Applications of Deque:**

Since Deque supports both stack and queue operations, it can be used as both. The Deque data structure supports clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications.

**QUEUE SIMULATION:**

Queuing simulation as a method is used to analyze how systems with limited resources distribute those resources to elements waiting to be served, while waiting elements may exhibit discrete variability in demand, i.e. arrival times and require discrete processing time.

Queuing theory based analysis is regularly used for e.g. telecommunications, computer networks, predicting computer performance, traffic, call centres, etc.

**APPLICATIONS OF QUEUES:**

Queue is used when things don't have to be processed immediately, but have to be processed in **F**irst **I**n First **O**ut order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.

- When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
- When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.


RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in