

# DS

## MODULE-: DATA STRUCTURES



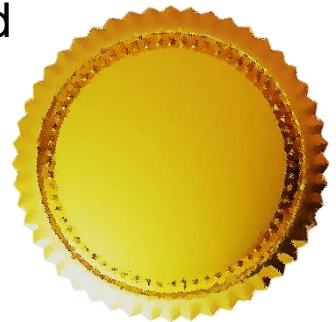
**Rohini Desai**

# Certificate


This is to certify that the e-book titled "Data Structures" comprises all elementary learning tools for a better understating of the relevant concepts. This e-book is comprehensively compiled as per the predefined eight parameters and guidelines.



Signature  
Prof. Rohini Desai  
Assistant Professor  
Department of IT



Date: 27-07-2017

 **DISCLAIMER:** *The information contained in this e-book is compiled and distributed for educational purposes only. This e-book has been designed to help learners understand relevant concepts with a more dynamic interface. The compiler of this e-book and Vidyalkar Institute of Technology give full and due credit to the authors of the contents, developers and all websites from wherever information has been sourced. We acknowledge our gratitude towards the websites YouTube, Wikipedia, and Google search engine. No commercial benefits are being drawn from this project.*

# Unit III: Stack and Queue

## Contents

- Introduction, Operations on the Stack, Memory Representation of Stack. Array Representation of Stack.
- **Applications of Stack:** Evaluation of Arithmetic Expression, Matching Parenthesis, Infix and postfix operations, Recursion.
- Queue: Introduction, Queue, Operations on the Queue,
- Memory Representation of Queue: Array representation of queue, Linked List Representation of Queue, Circular Queue.
- Some special kinds of queues: Deque, Priority Queue, Application of Priority Queue, Applications of Queues.

## Recommended Books

### References:

- A Simplified Approach to Data Structures
- An Introduction to Data Structure with Applications
- Data Structure and Algorithm
- Schaum's Outlines Data structure

## Prerequisites and Linking

Unit III	Pre-Requisites	Sem. II	Sem. III	Sem. IV	Sem. V	Sem. VI
Stack and Queue	-	OOPS	-	-	-	-

## UNIT III- Stack and Queue

### Stack

- Stack is one of the most commonly used linear data structures of variable size
- In arrays the insertion and deletion of an element can take place at any position of the array but in the case of stack the insertion and deletion of an element can occur only at one end which is known as **TOP**
- In stack insertion operation is known as **PUSH** and deletion operation is known as **POP**
- Stack is also called **Last In First Out (LIFO)** list which means that last item added to the stack will be the first item to be removed from the stack
- **Operations On The Stack**
  - Two basic operations performed on the stack are **Push** and **Pop**
  - **Push** operation refers to the insertion of the new element into the stack which will be inserted on the top of the stack
  - We can perform the push operation only when the stack is not full i.e. stack has sufficient space to accommodate the new element
  - Thus when the stack is full and an attempt is made to insert a new element into the stack then the condition known as stack overflow occurs
  - **Pop** operation refers to the removal of an element from the top of the stack.
  - We can perform the stack operation only when the stack is not empty
  - Therefore we must ensure the stack is not empty before applying the pop operation
  - When stack is empty and we are attempting to remove an element from the stack then the condition known as stack underflow occurs
- **Memory Representation Of Stack**
  - There are two common ways to implement a stack by arrays or by linked list
  - Array representation is acceptable if an accurate estimate of the stack's maximum size can be determined before the program runs
  - The linked list representation of the stack is more suitable if an accurate estimate of the stack's size cannot be made in advance
- **Array Representation Of Stack**
  - Representation of stack using an array is the simplest form of implementation but it has certain restrictions
    - Stack must contain homogenous data elements
    - One must specify the upper bound of the array i.e. maximum size of the stack must be defined before implementing it. Generally stack grows and shrinks overtime but it is confined to the space allocated to the array implementing that stack
  - While implementing the stack using an array a variable Top is used to hold the index of the stack's topmost element
  - Initially stack is empty and Top has the value zero and when the first element is inserted into the stack the value of the Top is incremented by 1 and so on
  - Each time a new element is inserted into the stack the value of the Top will be incremented by one before placing the new element into the stack
  - Each time an element is deleted from the stack variable Top is decremented by 1
  - Another variable Max is also used that represent the maximum number of elements that can be pushed into the stack

➤ **Algorithm for Push and Pop Operation for array representation of stack**

Algorithm: Push Operation – Inserting a new element Data at the top of the stack represented by an array S of size MAX with stack index variable Top pointing to the topmost element of the stack

- Step 1: If Top = MAX Then  
    Print "Stack is full Overflow condition"  
    Exit  
    [End If]

- Step 2: Set Top = Top + 1
- Step 3: Set S[Top] = Data
- Step 4: Exit

Algorithm: Pop Operation – Deleting an element from the stack represented by an array S and returns the element Data which is at the top of the stack

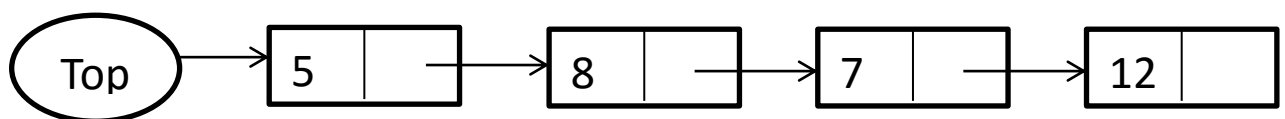
- Step 1: If Top = Null Then  
    Print "Stack is empty Underflow condition"  
    Exit  
    [End If]

- Step 2: Set Data = S[Top]
- Step 3: Set Top = Top - 1
- Step 4: Exit

- When stack is implemented as an array there is a need to allocate sufficient fixed space for the stack.
- If initially we reserve large amount of space for the stack then it will decrease the number of overflows but there will be wastage of memory as memory remains unutilized for most amount of time.
- On the other hand if we allocate small amount of memory space for the stack then it can increase the number of overflows.
- For resolving such overflows more memory space can be added to the stack as and when required, but this addition of memory will require the time which may be more expensive than the space saved.

➤ **Linked List Representation Of Stack**

- Stack can also be implemented using linked list
- By implementing the stack using the linked list we can eliminate the drawbacks which come across while implementing the stack using an array i.e. while implementing the stack using linked list there is no need to know in advance about the maximum size of the stack
- Linked representation of the stack allows it to grow without any prior fixed limit
- The Push and Pop operation can be performed on the linked list implementation of the stack by inserting and deleting an element at the beginning of the linked list



- **Algorithm for Push and Pop Operation for array representation of stack**  
Algorithm: Push Operation – Insert a new element 'Data' at the top of the stack represented by the linked list with a stack pointer variable 'Top' pointing to the topmost node of the stack.

- Step 1: If Free = Null Then  
    Print "Free space not available, Overflow condition"  
    Exit  
    [End If]
- Step 2: Set New = Free
- Step 3: Set New--> Info = Data And New-->Next = Top
- Step 4: Set Top = New
- Step 5 : Exit

Algorithm: Pop Operation – Delete an element from the stack represented by the linked list and returns the new element 'Data' which is at the top of the stack.

- Step 1: If Top = Null Then  
    Print "Stack is empty Underflow condition"  
    Exit  
    [End If]
- Step 2: Set Data = Top-->Info And Temp = Top
- Step 3: Set Top = Top-->Next
- Step 4: Set Temp-->Next = Free And Free = Temp
- Step 5 : Exit

- **Applications Of Stack – Evaluation Of Arithmetic Expression**
- An important application of stack is the compilation of arithmetic expressions in the programming languages
  - The compiler must be able to translate the expression which is written in the usual notation known as **infix notation** to a form which is known as **reverse polish notation**.
  - Compilers accomplish this task of notation conversion i.e. infix to postfix with the help of stack

### 1. Infix Notation

- In infix notation operator is placed between its operands. For eg to multiply m and n we write  $m \times n$
- It is the notation that is used by most of the people to solve any mathematical or arithmetic expression
- While solving the infix notation the main consideration is the precedence of operators and their associativity.
- With this notation we have to distinguish between  $(q \times r) + s$  and  $q \times (r + s)$  by using either operator precedence or by applying some parenthesis
- Such type of expressions cannot be solved accurately if we do not follow the rules of operator precedence and their associativity

- The main problem with this notation is that the order of operator and operands in the expression does not uniquely decide the order in which operations are to be carried out
- The precedence of various operators and its associativity is given below
  - Brackets (Inner To Out And Left To Right)
  - Exponent ^ (Left to Right)
  - \* / (Left to Right)
  - + - (Left to Right)
  - = (Right to Left)
- Consider the expression  
 $E = 4 - 2^4 + 8 \times 3 + 18 / 3 + 6$  will be solved as follows  
 $E = 4 - 16 + 8 \times 3 + 18 / 3 + 6$   
 $E = 4 - 16 + 24 + 18 / 3 + 6$   
 $E = 4 - 16 + 24 + 6 + 6$   
 $E = -12 + 24 + 6 + 6$   
 $E = 12 + 6 + 6 = 24$
- Consider another expression  
 $E = 5 \times 3 + (60 - 36) / 6 + (3 \times 10) + 7$   
 $E = 5 \times 3 + 24 / 6 + 30 + 7$   
 $E = 15 + 4 + 30 + 7 = 56$

## 2. Prefix Notation (Polish Notation)

- In this notation the operator is placed before **its operands**. For eg to multiply m and n we write **xmn**
- The main characteristic of this notation is that the order in which the operations are to be carried out is completely determined by the position of the operators and operands in the expression.
- While solving the arithmetic expression which is written in prefix/polish notation there is no need to take care of any precedence rule and there is no need to put the parenthesis in the expression



**Example 1:**

$$\begin{aligned}
I_{in} &= (a - b) / c \\
&= [- a b] / c \\
I_{pre} &= / - a b c
\end{aligned}$$

**Example 2:**

$$\begin{aligned}
I_{in} &= (x - y) \times ((z + v) / f) \\
&= [- x y] \times ([+ z v] / f) \\
&= [- x y] \times [/ + z v f] \\
I_{pre} &= \times - x y / + z v f
\end{aligned}$$

**Example 3:**

$$\begin{aligned}
I_{in} &= ((a + b) / d ^ ((e - f) + g)) \\
&= ([+ a b] / d ^ ([- e f] + g)) \\
&= ([+ a b] / d ^ [+ - e f g]) \\
&= [+ a b] / [^ d + - e f g] \\
I_{pre} &= / + a b ^ d + - e f g
\end{aligned}$$

**Example 4 :**

$$\begin{aligned}
I_{in} &= (x * y) + (z + ((a + b - c) * d)) - i * (j / k) \\
&= (x * y) + (z + (([+ a b] - c) * d)) - i * (j / k) \\
&= (x * y) + (z + ([- + a b c] * d)) - i * (j / k) \\
&= (x * y) + (z + [* - + a b c d]) - i * (j / k) \\
&= [* x y] + (z + [* - + a b c d]) - i * (j / k) \\
&= [* x y] + [+ z * - + a b c d] - i * (j / k) \\
&= [* x y] + [+ z * - + a b c d] - i * [/ j k] \\
&= [* x y] + [+ z * - + a b c d] - [* i / j k] \\
&= [+ * x y + z * - + a b c d] - [* i / j k] \\
I_{pre} &= - + * x y + z * - + a b c d * i / j k
\end{aligned}$$

**3. Postfix Notation (Reverse Polish Notation)**

- In this notation the operator is placed after **its operands**. For eg to multiply m and n we write **m n x**
- The main characteristic of this notation is that there is no need of parenthesis to designate the hierarchy of operators. In this notation order of the operators is completely determined by the order of operands and its operators.
- In computer an arithmetic expression which is written in infix notation is evaluated in 2 steps
  - Expression is converted in reverse polish notation
  - Converted expression is evaluated

- The reason behind this notation conversion is that postfix notation is very efficient for the point of view of evaluation done by computer.
- As postfix notations are scanned from left to right operands are simply placed into stack and operators may be immediately applied to operands which are at the top of the stack.

**Example 1:**

$$\begin{aligned}
 I_{in} &= (a - b) / c \\
 &= [a b -] / c \\
 I_{post} &= a b - c /
 \end{aligned}$$

**Example 2:**

$$\begin{aligned}
 I_{in} &= (x - y) \times ((z + v) / f) \\
 &= [x y -] \times ([z v +] / f) \\
 &= [x y -] \times [z v + f /] \\
 I_{post} &= x y - z v + f / \times
 \end{aligned}$$

**Example 3:**

$$\begin{aligned}
 I_{in} &= ((a + b) / d ^ ((e - f) + g)) \\
 &= ([a b +] / d ^ ([e f -] + g)) \\
 &= ([a b +] / d ^ [e f - g +]) \\
 &= ([a b +] / [d e f - g + ^]) \\
 &= [a b + d e f - g + ^ /] \\
 I_{post} &= [a b + d e f - g + ^ /]
 \end{aligned}$$

**Example 4:**

$$\begin{aligned}
 I_{in} &= (x * y) + (z + ((a + b - c) * d)) - i * (j / k) \\
 &= (x * y) + (z + (([a b +] - c) * d)) - i * (j / k) \\
 &= (x * y) + (z + ([a b + c -] * d)) - i * (j / k) \\
 &= (x * y) + (z + [a b + c - d *]) - i * (j / k) \\
 &= [x y *] + (z + [a b + c - d *]) - i * (j / k) \\
 &= [x y *] + [z a b + c - d * +] - i * (j / k) \\
 &= [x y *] + [z a b + c - d * +] - i * [j k /] \\
 &= [x y *] + [z a b + c - d * +] - [i j k / *] \\
 &= [x y * z a b + c - d * + +] - [i j k / *] \\
 &= [x y * z a b + c - d * + + i j k / * -] \\
 I_{post} &= x y * z a b + c - d * + + i j k / * -
 \end{aligned}$$

➤ **Algorithm: Convert an arithmetic expression I written in an infix notation into its equivalent postfix expression P**

- Step 1: Push a left parenthesis ( onto the stack
  - Step 2: Append a right parenthesis ) at the end of the given expression I
  - Step 3: Repeat steps from 4 to 8 by scanning I character by character from left to right until the stack is empty
  - Step 4: If the current character in I is a white space, ignore it
  - Step 5: If the current character in I is an operand, write it as the next element of the postfix expression P
  - Step 6: If the current character in I is a left parenthesis ( push it onto the stack
  - Step 7: If the current character in I is an operator Then
    - Pop operators (if there are any) at the top of the stack while they have **equal or higher precedence** than the current operator and put the popped operators in the postfix expression P
    - Push the currently scanned operator on the stack
  - Step 8: If the current character in I is a right parenthesis Then
    - Pop operators from the top of the stack and insert them in the postfix expression P until a left parenthesis is encountered at the top of the stack
    - Pop and discard the left parenthesis ( from the stack
- [End Loop]

- Step 9: Exit

Example: Consider an expression I = ( 6 + 2 ) \* 5 – 8 / 4

Now I = ((6 + 2) \* 5 – 8 / 4 )

Character Scanned	Status Of Stack	Postfix Expression P
(	(	
(	((	
6	((	6
+	((+	6
2	((+	2
)	(	6 2 +
*	(*	6 2 +
5	(*	6 2 + 5
-	(-	6 2 + 5 *
8	(-	6 2 + 5 * 8
/	(-/	6 2 + 5 * 8
4	(-/	6 2 + 5 * 8 4
)	Null	6 2 + 5 * 8 4 / -

<https://www.youtube.com/watch?v=vXPL6UavUeA>

➤ Evaluation Of Postfix Notation

Algorithm: Evaluate an arithmetic expression P written in postfix notation and calculate the result of the expression in the variable R

- Step 1: Scan **P** from left to right and Repeat steps 2 and 3 for each scanned character until end of expression
- Step 2: If the scanned character is an **operand** push it onto the stack
- Step 3: If the scanned character is an **operator** Then
  - Pop the two elements **a** and **b** from the stack where a is the top element and b is the next top element
  - Apply the operator on the operands **a** and **b** and push the result onto the stack
- [End Loop]
- Step 4: Set **R = Stack[Top]**;
- Step 5: Print **R**
- Step 6: Exit

Example: Consider converted Postfix expression  $P = 6\ 2\ +\ 5\ * \ 8\ 4\ /\ -$

Character Scanned	Status Of Stack
6	6
2	6 2
+	8
5	8 5
*	40

8	40 8
4	40 8 4
/	40 2
-	38

### ➤ Matching Parenthesis

- A stack can be used for syntax verification of the arithmetic expression for ensuring that for each left parenthesis in the expression there is a corresponding right parenthesis
- To accomplish this task of parenthesis matching the expression is scanned from left to right character by character
- Whenever a left parenthesis is encountered we push it onto the stack
- The parenthesis encountered can be of any type square [, round ( or curly{
- When we encounter a right parenthesis ], or ), or } the status of the stack is checked
- If the stack is empty then we have a right parenthesis in an expression that does not have the corresponding left parenthesis in the expression showing the mistake
- If the stack is not empty we will pop the topmost element from the stack and compare it with the scanned right parenthesis
- If both the parameters are not of the same type then it shows a mistake in the expression and if both the parenthesis are of the same type then the same procedure is repeated until the whole expression is scanned until stack is empty.

#### **Algorithm: Syntax verification by scanning an arithmetic expression I from left to right character by character using a stack**

- Step 1: Scan the expression I from left to right and repeat steps 2 and 4 for each scanned character until the end of the expression is reached
- Step 2: If the scanned character is a **left parenthesis** then push it onto the stack
- Step 3: If the scanned character is an **operator** or **operand** then ignore it
- Step 4: If the scanned character is a **right parenthesis** Then
  - If **Stack[Top] = Null** Then
    - Print "There is no left parenthesis corresponding to right parenthesis"
    - Exit
  - [End If]
  - Pop the top element from the stack and compare it with currently scanned right parenthesis
  - If both are not corresponding Then
    - Print "The braces are not in proper order"
    - Exit
  - [End If]
  - [End Loop]
- Step 5: If **Stack[Top] != Null** Then
  - Print "There is no right parenthesis corresponding to the left parenthesis"

- Exit  
[End If]
  - Step 6: Exit
- Example: Consider arithmetic expression I

$$P = [(5+6)*7 - \{7/4\} + (3*2)-8]$$

Character Scanned	Status Of Stack
[	[
(	[ (
)	[
{	[ {
}	[
(	[ (
)	[
]	Null

### ➤ Recursion

- Recursion is a very important and powerful tool for developing algorithms for various problems
- Recursion is the ability of the procedure to call itself or calling to some other procedure which may result in call to original procedure
- Two important conditions that must be satisfied by a recursive procedure are
  - There must be a decision criterion that stops the further call to the procedure called base criteria
  - Each time a procedure calls itself either directly or indirectly it must be nearer to the solution i.e. nearer to the base criteria
- A procedure having these two properties is called a well defined procedure and can be defined recursively
- Recursive procedure can be implemented in various programming languages like C, C++ etc

### ➤ Factorial Function Using Recursion

- The factorial of a positive number n is a product of positive integers from 1 to n
- The factorial of a number is represented symbolically by placing a ! next to it
- The factorial of a positive number n will be defined as  $N! = 1 * 2 * 3 * 4 * \dots * (n-1) * n$
- The formal definition of factorial function can be given as

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n - 1) & \text{if } n \geq 1 \end{cases}$$

<https://www.youtube.com/watch?v= OmRGjbyzno&list=PL2 aWCzGMA wLz3g66WrxFGSXvSsvyFzCO>

**Algorithm: Calculate the value of n! Recursively**

```
Factorial(n)
If n = 0 Then
Set Fact = 1
Return
Else
Set Fact = n * Factorial(n - 1)
Return
[End If]
```

➤ **Fibonacci Series Using Recursion**

- A Fibonacci series is a sequence of numbers which is usually denoted by  $F_0, F_1, F_2, \dots, F_n$  (0,1,1,2,3,5,8,13.....)
- Here  $F_0=0, F_1=1$  and  $F_2=F_0 + F_1, F_3=F_1 + F_2$  and so on
- The formal definition of Fibonacci series function can be given as

$$Fibo(n) = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ Fibo(n - 1) + Fibo(n - 2) & \text{if } n > 1 \end{cases}$$

**Algorithm: Find the nth term of a Fibonacci series recursively**

```
Fibonacci(n)
If n = 0 Then
Set Fibo = 0
Return
Else If n = 1 Then
Set Fibo = 1
Return
Else
```

```
Set Fibo = Fibonacci(n - 1) + Fibonacci(n - 2)
Return
[End If]
```

### **Recursion**

- Recursion is generally used for repetitive computation in which each action is defined in terms of previous result
- The factors that affect the choice of procedure for solving a given problem
  - Computer memory required
  - Processing time required
  - Time required for developing the algorithm
  - Time required for debugging
- Recursion is a top down approach to problem solving
- It divides the program into pieces or selects one key step postponing the rest

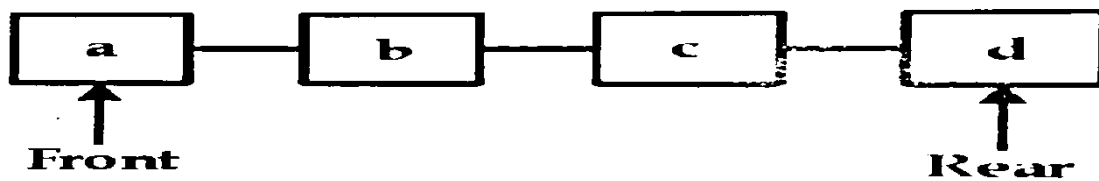
### ➤ **Recursion And Iterative Methods**

- Demerits of Recursion
  - Many programming languages do not support recursion hence recursive mathematical function is implemented using iterative method
  - Even though mathematical functions can be easily implemented using recursion it is always at the cost of execution time and memory space
  - A recursive procedure can be called from within or outside itself and to ensure its proper functioning it has to save the return addresses in the same order so that a return to the proper location will result when the return to a calling statement is made
  - A special care is required to put a stopping condition at which the recursive function will stop
- Demerits of Iterative methods
  - Mathematical function such as factorial and Fibonacci series generation can be easily implemented using recursion rather than iteration
  - In iterative techniques looping of statement is very much necessary



# Queue

- Queue is a linear collection of elements in which insertion takes place at one end known as rear and deletion takes place at another end known as front of the queue.
- Queue is also a restricted data structure like stack because the new element can be added at its one end known as rear of queue and element can be removed from the other end known as front of the queue
- Queue is known as **First In First Out(FIFO)** List.
- The element of a general queue are processed on First Come First Serve basis(FCFS)

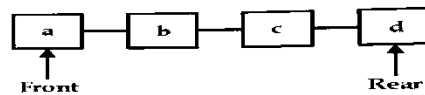


**A Queue with Four Elements**

## ➤ Operations on the Queue

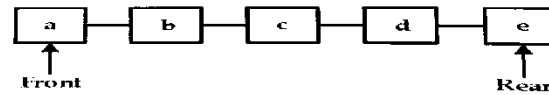
- Two Basic operations which are performed on the queue are: Insertion , Deletion
- The insertion operation refers to the addition of a new element at the rear end of the queue.
- Insertion operation can be performed only when queue has space to accommodate the new element.
- The condition of attempting to insert an element in a queue having no space results in a state called overflow condition.
- Deletion refers to the removal of an element from the front of the queue.
- Deletion operation can be performed only when the queue is not empty.
- The condition of attempting to delete an element from an empty queue is known as underflow condition.

attempting to delete an element from the queue. Consider a list of four elements (**a**, **b**, **c**, **d**) where **a** is the front element and **d** is the rear element. The queue of these elements can be represented diagrammatically as shown in figure below:



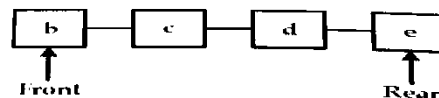
**A Queue with Four Elements**

New element **e** will be inserted at the rear end in the given queue after the element **d** as shown in figure below:



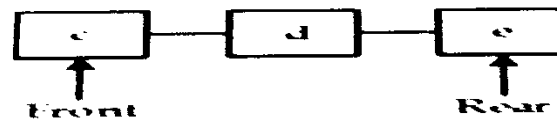
**Queue after the Insertion of a New Element**

Suppose one element is to be deleted from the queue. The element **a** will be deleted from the front end of the queue as shown in figure below:



**Queue after the Removal of an Element**

Let us delete another element from the queue. The element at the front end of the queue i.e. **b** will be deleted from the queue. The queue after the deletion of front element **b** will look like as shown in figure below:



**Queue after the Removal of another Element**

### ➤ Memory Representation Of Queue

- Queue can be represented in memory by means of an array or linked list
- Representing the queue using an array imposes constraints that are associated with the array
- Array representation of queue is acceptable if the accurate estimate of the queue length can be made prior to the execution of the program
- The linked list representation of the queue is more suitable if the accurate estimate of the queue size cannot be made in advance.

### ➤ Array Representation Of Queue

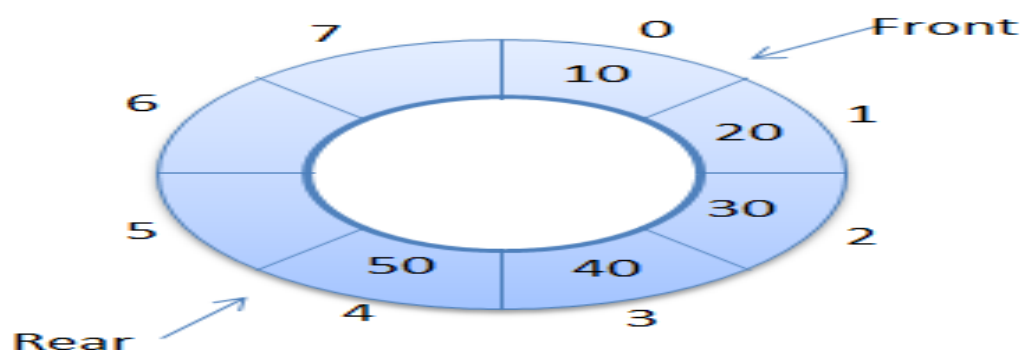
- Representing queue using array is very simple but puts some restrictions while representing the queue in memory
  - The element that to be inserted into the queue must be of same type(Homogenous)
  - One must specify the upper bound of the array i.e maximum size of the queue must be define before implementing it. That is queue represented using array is confined to the space allocated to that array. Queue grows and shrinks overtime but an array has constant size
  - While representing queue using array , it is programmer's responsibility to enforce the FIRST IN FIRST (FIFO) order.
- While implementing queue using array, we need to use two variables Front and Rear to keep the track of the beginning and end of the queue
- However implementation of queue using array is not straight forward as in the case of the stack.
- In array implementation of stack, insertion and deletion is restricted at one end.
- Therefore the elements of the stack occupy only the front part of the array whereas in queue the elements are inserted at the rear end of the queue and deleted from the front end of the queue

- So if the last position of the array is occupied then it will not possible to add more elements even though some front positions will remain vacant in the queue

<https://www.youtube.com/watch?v=wI5x1OFEccs>

### ➤ Circular Queue

- Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called as “Ring Buffer”
- Graphical representation of a circular queue is as follows



**Algorithm: Insert a new element 'Data' into a queue.**

- Step 1: If Front = 1 And Rear = n Then  
     Print : “Queue is Full, Overflow Condition”  
     Exit  
   [End If]

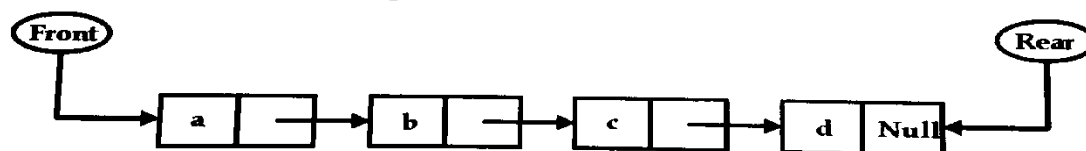
- Step 2: If  $\text{Front} = \text{Rear} + 1$  Then  
     Print : "Queue is Full, Overflow Condition"  
     Exit  
   [End If]
- Step 3: If  $\text{Rear} = \text{Null}$  Then  
     Set  $\text{Front} = 1$  And  $\text{Rear} = 1$   
   Else If  $\text{Rear} = n$  Then  
     Set  $\text{Rear} = 1$   
   Else  
     Set  $\text{Rear} = \text{Rear} + 1$   
   [End IF]
- Step 4: Set  $Q[\text{Rear}] = \text{Data}$
- Step 5: Exit

**Algorithm: Delete an element from the queue**

- Step 1: If  $\text{Front} = \text{Null}$  Then  
     Print : "Queue is Empty, Underflow Condition"  
     Exit  
   [End If]
- Step 2: Set  $\text{Data} = Q[\text{Front}]$
- Step 3: If  $\text{Front} = \text{Rear}$  Then  
     Set  $\text{Front} = \text{Null}$  and  $\text{Rear} = \text{Null}$   
   Else  
     If  $\text{Front} = n$  Then  
       Set  $\text{Front} = 1$   
     Else  
       Set  $\text{Front} = \text{Front} + 1$   
     [End IF]  
   [End IF]
- Step 4: Exit

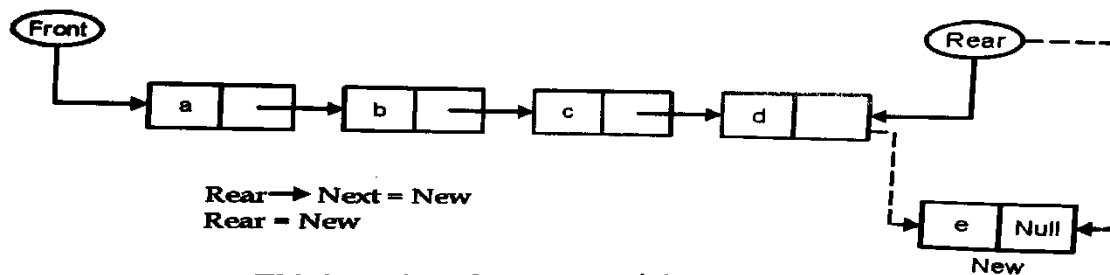
➤ **Linked List Representation of Queue**

- An alternative and efficient way of representing queue is by using linked list. The advantages of such a representation over an array representation are similar to those of linked list over the array
- Here two pointer variables **Front** and **Rear** contain the addresses of the element at the front and rear end of the queue.
- Initially when there is no element in the queue, both pointers **Front** and **Rear** will have **Null** indicating an empty queue.

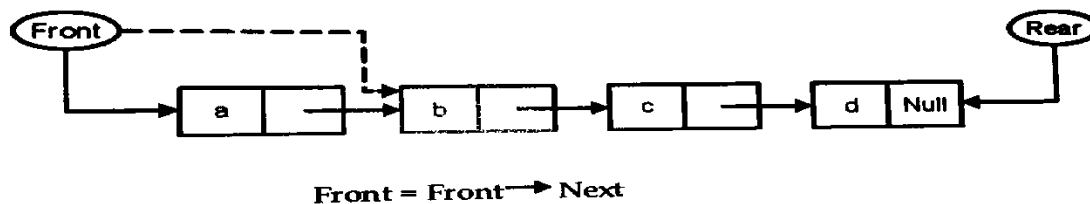


**A Queue Maintained using a Linked List**

- Insertion of new element in queue



- Deletion of an element from a queue



#### Deletion of an element from the Queue

**Algorithm: Insert a given element 'Data' in a queue which is implemented using a linked list 'Q' having variable 'Front' which contains the address of 1<sup>st</sup> element of the queue and variable 'Rear' which contains the address of the last element of the queue.**

- Step 1: If Free = Null  
Print : "No Free Space Available For Insertion"  
Exit  
[End If]
- Step 2: Allocate memory to node New
- Set New = Free And Free = Free --> Next
- Step 3: Set New --> Info = Data And New --> Next = Null
- Step 4: If Rear = Null Then  
Set Front = New And Rear = New  
Else  
Set Rear --> Next = New And Rear = New  
[End If]
- Step 5: Exit

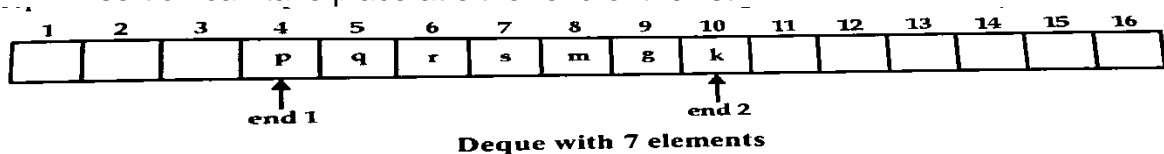
**Algorithm: Remove an element from the queue which is maintained using a linked list 'Q' having variable 'Front' which contains the address of 1<sup>st</sup> element of the queue and variable 'Rear' which contains the address of the last element of the queue.**

- Step 1: If Free = Null  
Print : "Queue is Empty"  
Exit  
[End If]
- Step 2: Set Data = Front --> Info , Temp = Front
- Step 3: If Front = Rear Then
- Set Front = Null And Rear = Null
- Else  
Set Front = Front --> Next  
[End If]
- Step 4: Deallocate memory taken by node Temp

- Set Temp --> Next = Free , Free = Temp
  - Step 5: Exit
- Special Kinds of Queue -
- Deque (Double Ended Queue)
  - Priority Queue

### ➤ Double Ended Queue(Deque)

- Double ended queue is the linear queue data structure in which insertion and deletion operations are not restricted to one end but rather insertion and deletion operations can be performed on either of the two ends.
- These operations cannot be performed at any other position except the two ends of the list
- Deque data structure is also known as Deck.
- Instead of using the notation of Front and Rear, we use two variables end1 and end2 to represent the index of the two ends of the queue.
- An element can be inserted either at end1 or end2. Similarly the element can be deleted either at end1 or end2.
- A Deque can be categorized into two categories:
- Input Restricted Deque – Insertion operation is restricted to one end but the deletion can take place at either end of the list.
- Output Restricted Deque –Deletion operation is restricted to one end but the insertion can take place at either end of the list.

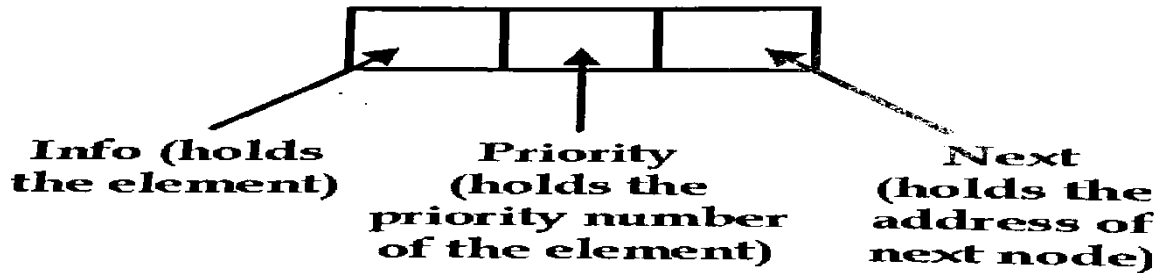


### ➤ Priority Queue

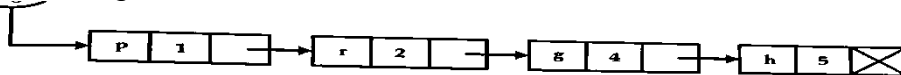
- Priority Queue is a special kind of queue data structure in which insertion and deletion operations are performed according to some special rule rather than just First In First Out(FIFO) rule.
- In case of Priority queue, a priority is associated with each element.
- The elements are inserted or deleted according to this priority number.
- The following two rules are applied to process the elements in the priority queue:
  - The elements with higher priority are processed before the elements with lower priority.
  - In case of elements with same priority, elements are processed according to First In First Out(FIFO) rule. That is, the element with same priority will be processed in the same order in which these are inserted into the queue.
- Priority queue can be represented into the memory in various ways. The three main ways to represent a priority queue are:
- Priority queue using linked list.
- Priority queue using multiple queues
- Priority queue using heap structure

#### 1. Priority Queue Using Linked List

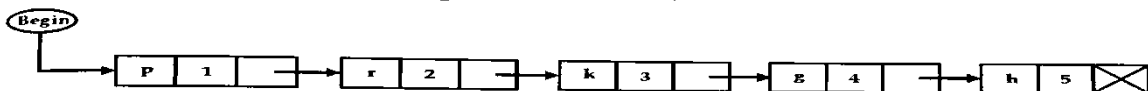
- In linked list representation of priority queue, each node of the linked list is divided into three parts:
  - Info part holds the element of the queue
  - Priority part holds the priority number of the element.
  - Next holds the address of next node of the linked list.
- In linked list representation of priority queue, insertion and deletion of an element takes place according to the priority number of the element whereas deletion of the element takes place from front end of the linked list.



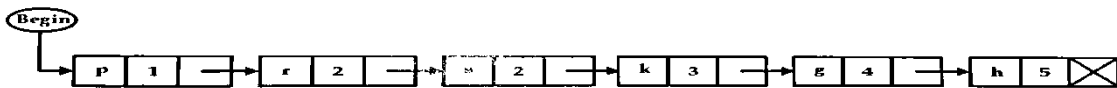
### Priority Queue Using Linked List



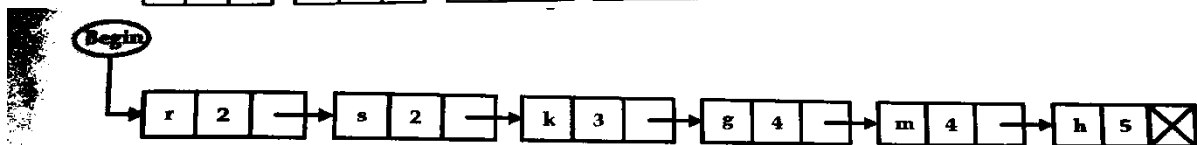
Now, let us insert an element **k** with priority 3. This element will be inserted as the 3<sup>rd</sup> node i.e. between the node **r** and **g** as shown below:



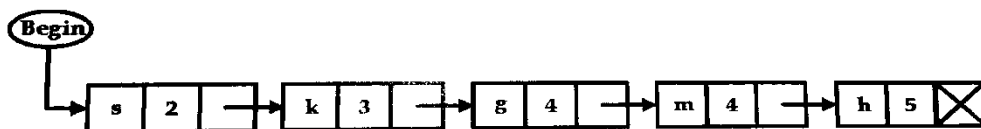
Now, let us insert an element **s** with priority 2. This element will be inserted as 3<sup>rd</sup> node (i.e. between the node **r** and **k**) as shown below:



Let us insert one more element **m** with priority 4. This element will be inserted as 6<sup>th</sup> node (i.e. between the node **g** and **h**) as shown below:



After deleting another element, the priority queue will be



## 2. Priority Queue Using Multiple Queues

- Priority queue can be implemented using multiple arrays in the form of multiple queues.
- In this representation of priority queue, a separate queue is maintained for all the elements with same priority level which follows the general FIFO order.
- These separate queues may be circular queues for its efficient use and will also have separate variables Front and Rear.

- While using multiple queues representation of priority queues, following information must be known in advance,
- The maximum number of priority levels.
- The maximum number of elements with same priority.

Let us suppose that we have maximum of 5 priority levels (i.e. priority number 1, 2, 3, 4, and 5) and maximum of 4 elements with same priority. So, the priority queue can accommodate maximum of  $5 \times 4 = 20$  elements. Elements with priority  $p$  are inserted in the respective queue of priority  $p$  according to FIFO rule whereas the elements can be deleted according to the priority of the element (starting from the 1<sup>st</sup> queue onwards).

Following is the representation of priority queue using multiple queues:

Priority	Front	Rear	1 2 3 4				
1	0	0	<table border="1"><tr><td></td><td></td><td></td><td></td></tr></table>				
2	0	0	<table border="1"><tr><td></td><td></td><td></td><td></td></tr></table>				
3	0	0	<table border="1"><tr><td></td><td></td><td></td><td></td></tr></table>				
4	0	0	<table border="1"><tr><td></td><td></td><td></td><td></td></tr></table>				
5	0	0	<table border="1"><tr><td></td><td></td><td></td><td></td></tr></table>				

#### Priority queue representation using multiple queues

After insertion and deletion of the following elements in sequence, the resulting representation will be:

Insert  $p$  with priority 3  
 Insert  $n$  with priority 5  
 Insert  $g$  with priority 3  
 Insert  $s$  with priority 4  
 Insert  $m$  with priority 3  
 Delete an element  
 Insert  $k$  with priority 4

Insert  $d$  with priority 3  
 Delete an element  
 Insert  $b$  with priority 2  
 Insert  $l$  with priority 2  
 Delete an element  
 Insert  $r$  with priority 1  
 Insert  $v$  with priority 2

Priority	Front	Rear	1 2 3 4				
1	1	1	<table border="1"><tr><td>r</td><td></td><td></td><td></td></tr></table>	r			
r							
2	2	3	<table border="1"><tr><td>b</td><td>l</td><td>v</td><td></td></tr></table>	b	l	v	
b	l	v					
3	3	4	<table border="1"><tr><td>p</td><td>g</td><td>m</td><td>d</td></tr></table>	p	g	m	d
p	g	m	d				
4	1	2	<table border="1"><tr><td>s</td><td>k</td><td></td><td></td></tr></table>	s	k		
s	k						
5	1	1	<table border="1"><tr><td>n</td><td></td><td></td><td></td></tr></table>	n			
n							

### 3. Priority Queue Using Heap Structure

- Heap data structure can be used to implement priority queue.
- Heap(Max Heap) is almost complete binary tree data structure in which every parent element is larger than or equal to its child elements.



- Min Heap is almost complete binary tree data structure in which every parent element is smaller than or equal to its child elements.
- Both types (Max and Min heap) can be used for implementing priority queue.
- This is because, the root element in case of max heap is always the largest element. This means that if we have largest element as the highest priority element then it is easy to process because we know that it is present at the root node of the tree.
- Similarly, if we have smallest element as the highest priority element then we can process it using Min Heap

### ➤ Applications of Priority Queue

- In case of a time sharing systems, where different jobs are to be processed by the same processor, priority queues are used by the operating systems to manage the processes. If the operating system of the computer implements the shortest job first policy then in that case, a shortest job will always be processed before the longer job i.e. shortest job will be given priority over the longer job by the operating system. The different jobs waiting for the processor time will form a priority queue.
- Priority queue can be used to manage bandwidth on a transmission line from a network router. In case limited bandwidth, all other queues can be halted to send traffic from the highest priority queue.
- Priority queue can be used in Huffman Coding that requires one to repeatedly obtain the two lowest-frequency trees. A priority queue makes this efficient.
- Priority queue has its application in event driven simulation, numerical computation, data compression, graph searching, computational number theory, artificial intelligence, discrete optimization etc.

### ➤ Applications of Queue

- Queue is used to access the shared resources e.g. printer queues.
- Queue is used as buffer between the faster processor and slow input/output devices.
- Queue is used to implement multiprogramming concepts.
- Queue is used by the operating systems for processing management.
- Queue can be used as components for other data structures.
- Queue is used as a buffer in Leaky Bucket algorithm to control the flow of data at the router.

Questions:

1. What is Stack? How it is different from linear data structures?
2. Convert infix expression to their equivalent prefix and postfix expression  
 $a*b+(c+d)-(e+f)+g*h/k^2$
3. What is recursion? Explain factorial of a number using recursion.
4. Write an algorithm to convert an arithmetic expression written in infix notation into its equivalent postfix expression.
5. Write an algorithm for basic operation of Linked list representation of queue.
6. Write a short note on Deque
7. Compare stack and queue data structure.
8. Write an algorithm for two basic operation of Linked list representation of stack.
9. Convert infix expression to their equivalent prefix and postfix expression

10.  $b+c*d-e+(e^2*f)$
11. What is recursion? Explain Fibonacci series using recursion.
12. Write an algorithm to evaluate an arithmetic expression in postfix notation and calculates the result of the expression.
13. Write an algorithm for basic operation of array representation of circular queue.
14. Explain memory representation of priority queue using multiple queues.
15. What is queue? Explain the similarities and difference between stack and queue.
16. Write an algorithm for two basic operation of array representation of stack.
17. Convert infix expression to their equivalent prefix and postfix expression
18.  $(a*b*c^2+d)+(c/d+c)$
19. Explain demerits of recursion and demerits of iterative methods.
20. Write an algorithm for syntax verification by scanning an arithmetic expression from left to right character by character using stack.
21. What is priority queue? Write the applications of priority queue.
22. What is circular queue? Explain its applications.
23. Explain memory representation of priority queue using Linked List.

Multiple Choice questions:

1. An algorithm that calls itself directly or indirectly is known as
  - a. Sub algorithm
  - b. Recursion**
  - c. Polish Notation
  - d. Traversal Algorithm
2. Which of the following name does not relate to stack?
  - a. FIFO List**
  - b. LIFO List
  - c. Piles
  - d. Push-down Lists
3. The term push and pop is related to the
  - a. Array
  - b. Lists
  - c. Stacks**
  - d. All the above
4. Which one of the following is an application of Stack Data Structure?
  - a. Managing function calls
  - b. The stock span problem
  - c. Arithmetic expression evaluation
  - d. All the above**
5. When new data are to be inserted into a data structure, but there is no available space this situation is usually called
  - a. Underflow
  - b. Overflow**
  - c. Housefull
  - d. Saturated
6. Which data structure allows deleting data elements from front and inserting at rear?
  - a. Stacks

- b. Queues**
  - c. Deques
  - d. Binary Search tree
- 7. Identify the data structure which allows deletions at both ends of the list but insertion at only one end.
  - a. Input- restricted Deque**
  - b. Output-restricted Deque
  - c. Priority queues
  - d. None of the above
- 8. A data structure where elements can be added or removed at either end but not in the middle
  - a. Linked Lists
  - b. Stacks
  - c. Queues
  - d. Deque**
- 9. Which one of the following is an application of Queues Data Structure?
  - a. When a resource is shared among multiple consumers.
  - b. When data is transferred asynchronously between two processes
  - c. Loading Balancing
  - d. All of the above**
- 10. \_\_\_\_\_ is not operation that can be performed on queue
  - a. Insertion
  - b. Deletion
  - c. Retrieval
  - d. Traversal**