

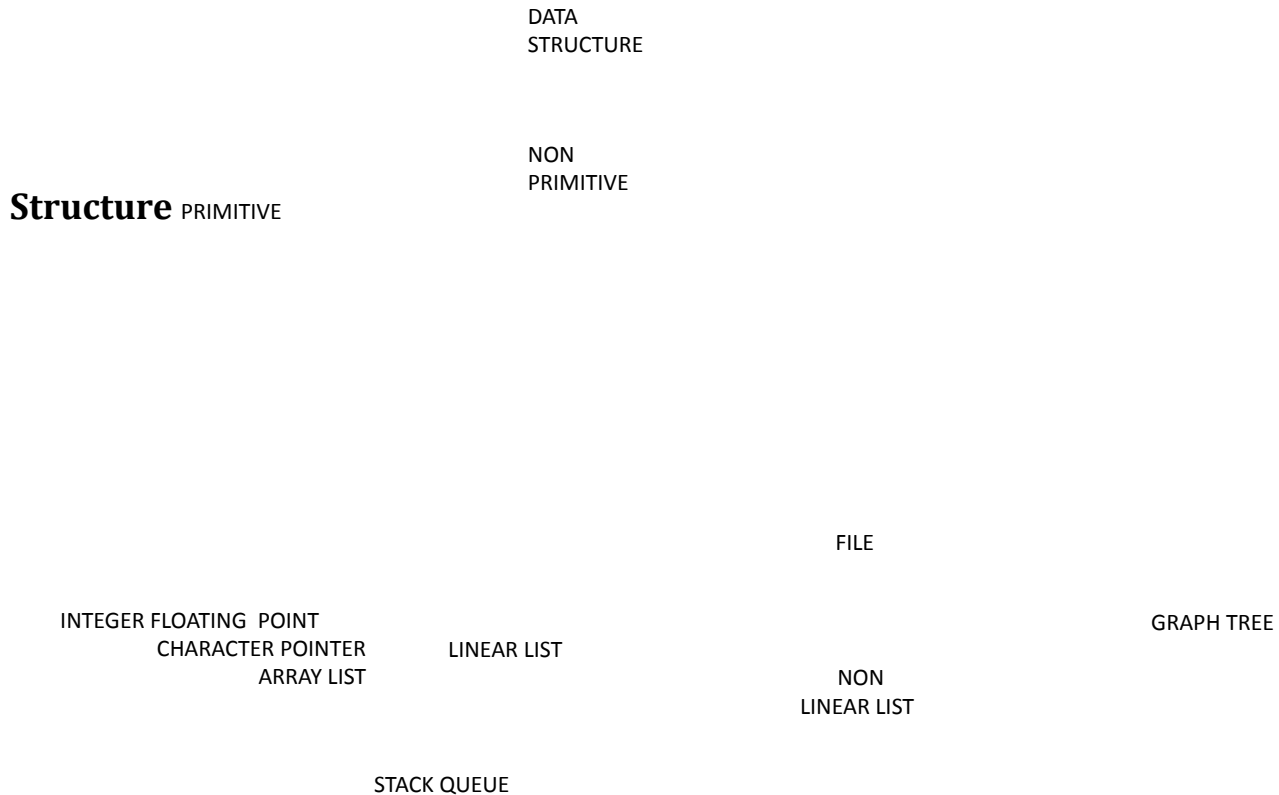
## Introduction to Data Structure

- Computer is an electronic machine which is used for data processing and manipulation. • When programmer collects such type of data for processing, he would require to store all of them in computer's main memory.
- In order to make computer work we need to know
  - Representation of data in computer.
  - Accessing of data.
  - How to solve problem step by step.
- For doing this task we use data structure.

## What is Data Structure?

- **Data structure** is a representation of the logical relationship existing between individual elements of data.
- Data Structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.
- We can also define data structure as a mathematical or logical model of a particular organization of data items.
- The representation of particular data structure in the main memory of a computer is called as **storage structure**.
- The storage structure representation in auxiliary memory is called as **file structure**. • It is defined as the way of storing and manipulating data in organized form so that it can be used efficiently.
- Data Structure mainly specifies the following four things
  - Organization of Data
  - Accessing methods
  - Degree of associativity
  - Processing alternatives for information
- Algorithm + Data Structure = Program
- Data structure study covers the following points
  - Amount of memory require to store.
  - Amount of time require to process.
  - Representation of data in memory.
  - Operations performed on that data.

## Classification of Data



Data Structures are normally classified into two broad categories

1. Primitive Data Structure
2. Non-primitive data Structure

## Data types

A particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it.

## Primitive Data Structure

- Primitive data structures are basic structures and are directly operated upon by machine instructions.
- Primitive data structures have different representations on different computers.
- Integers, floats, character and pointers are examples of primitive data structures.
- These data types are available in most programming languages as built in type.

- Integer: It is a data type which allows all values without fraction part. We can use it for whole numbers.
- Float: It is a data type which use for storing fractional numbers.
- Character: It is a data type which is used for character values.

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure 2



## Introduction to Data Structure

Pointer: A variable that holds memory address of another variable are called pointer.

### Non primitive Data Type

- These are more sophisticated data structures.
- These are derived from primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items.
- Examples of Non-primitive data type are Array, List, and File etc.
- A Non-primitive data type is further divided into Linear and Non-Linear data structure ○
  - Array:** An array is a fixed-size sequenced collection of elements of the same data type. ○
  - List:** An ordered set containing variable number of elements is called as Lists.
  - **File:** A file is a collection of logically related information. It can be viewed as a large list of records consisting of various fields.

### Linear data structures

- A data structure is said to be Linear, if its elements are connected in linear fashion by means of logically or in sequence memory locations.
- There are two ways to represent a linear data structure in memory,
  - Static memory allocation
  - Dynamic memory allocation
- The possible operations on the linear data structure are: Traversal, Insertion, Deletion, Searching, Sorting and Merging.
- Examples of Linear Data Structure are Stack and Queue.
- Stack: Stack is a data structure in which insertion and deletion operations are performed at one end only. ○
  - The insertion operation is referred to as 'PUSH' and deletion operation is referred to as 'POP' operation. ○
  - Stack is also called as Last in First out (LIFO) data structure.
- Queue: The data structure which permits the insertion at one end and Deletion at another end, known as Queue.
  - End at which deletion is occurs is known as FRONT end and another end at which insertion occurs is known as REAR end.
  - Queue is also called as First in First out (FIFO) data structure.

### Nonlinear data structures

- Nonlinear data structures are those data structure in which data items are not arranged in a sequence. •
- Examples of Non-linear Data Structure are Tree and Graph.

- **Tree:** A tree can be defined as finite set of data items (nodes) in which data items are arranged in branches and sub branches according to requirement.
  - Trees represent the hierarchical relationship between various elements.
  - Tree consist of nodes connected by edge, the node represented by circle and edge lives connecting to circle.

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **3**



## Introduction to Data Structure

- **Graph:** Graph is a collection of nodes (Information) and connecting edges (Logical relation) between nodes.
  - A tree can be viewed as restricted graph.
  - Graphs have many types:
    - Un-directed Graph
    - Directed Graph
    - Mixed Graph
    - Multi Graph
    - Simple Graph
    - Null Graph
    - Weighted Graph

### Difference between Linear and Non Linear Data Structure

Linear Data Structure	Non-Linear Data Structure
Every item is related to its previous and next time.	Every item is attached with many other items.
Data is arranged in linear sequence.	Data is not arranged in sequence.
Data items can be traversed in a single run.	Data cannot be traversed in a single run.
Eg. Array, Stacks, linked list, queue.	Eg. tree, graph.
Implementation is easy.	Implementation is difficult.

### Operation on Data Structures

Design of efficient data structure must take operations to be performed on the data structures into account. The most commonly used operations on data structure are broadly categorized into following types

#### 1. Create

The create operation results in reserving memory for program elements. This can be done by declaration statement. Creation of data structure may take place either during compile-time or run-time. malloc() function of C language is used for creation.

## 2. Destroy

Destroy operation destroys memory space allocated for specified data structure. free() function of C language is used to destroy data structure.

## 3. Selection

Selection operation deals with accessing a particular data within a data structure.

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **4**



# Introduction to Data Structure

## 4. Updation

It updates or modifies the data in the data structure.

## 5. Searching

It finds the presence of desired data item in the list of data items, it may also find the locations of all elements that satisfy certain conditions.

## 6. Sorting

Sorting is a process of arranging all data items in a data structure in a particular order, say for example, either in ascending order or in descending order.

## 7. Merging

Merging is a process of combining the data items of two different sorted list into a single sorted list.

## 8. Splitting

Splitting is a process of partitioning single list to multiple list.

## 9. Traversal

Traversal is a process of visiting each and every node of a list in systematic manner.

## Time and space analysis of algorithms

### Algorithm

- An essential aspect to data structures is algorithms.
- Data structures are implemented using algorithms.
- An algorithm is a procedure that you can write as a C function or program, or any other language. •

An algorithm states explicitly how the data will be manipulated.

### Algorithm Efficiency

- Some algorithms are more efficient than others. We would prefer to choose an efficient algorithm, so it would be nice to have metrics for comparing algorithm efficiency.

- The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process.
- Usually there are natural units for the domain and range of this function. There are two main complexity measures of the efficiency of an algorithm
- **Time complexity**
  - **Time Complexity** is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure 5



## Introduction to Data Structure

- "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.
- **Space complexity**
  - **Space complexity** is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.
  - We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this.
  - We can use bytes, but it's easier to use, say, number of integers used, number of fixed-sized structures, etc. In the end, the function we come up with will be independent of the actual number of bytes needed to represent the unit.
  - Space complexity is sometimes ignored because the space used is minimal and/or obvious, but sometimes it becomes as important an issue as time.

### Worst Case Analysis

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched is not present in the array. When  $x$  is not present, the search () functions compares it with all the elements of array [] one by one. Therefore, the worst case time complexity of linear search would be.

### Average Case Analysis

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed. So we sum all the cases and divide the sum by  $(n+1)$ .

## Best Case Analysis

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when  $x$  is present at the first location. The number of operations in worst case is constant (not dependent on  $n$ ). So time complexity in the best case would be.

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure 6



## Linear Data Structure

### Explain Array in detail

#### One Dimensional Array

- Simplest data structure that makes use of computed address to locate its elements is the one dimensional array or vector; number of memory locations is sequentially allocated to the vector.
- A vector size is fixed and therefore requires a fixed number of memory locations.
- Vector  $A$  with subscript lower bound of “one” is represented as below....

$L_0$

element or node

- The address of  $A_i$  is given equation

**$\text{Loc}(A_i) = L_0 + C(i-1)$**  • Let's consider

the more general case of representing a vector  $A$  whose lower bound for its subscript is given by some variable  $b$ . The location of  $A_i$  is then given by  **$\text{Loc}(A_i) = L_0 + C(i-b)$**  •

$L_0 + (i-1)C$

#### Two Dimensional Array

**$A[i]$**

- $L_0$  is the address of the first word allocated to the first element of vector  $A$ .

- $C$  words are allocated for each

- Two dimensional arrays are also called table or matrix, two dimensional arrays have two subscripts • Two dimensional array in which elements are stored column by column is called as column major matrix • Two dimensional array in which elements are stored row by row is called as row major matrix • First subscript denotes number of rows and second subscript denotes the number of columns • Two dimensional array consisting of two rows and four columns as above Fig is stored sequentially by

columns :  $A[1, 1], A[2, 1], A[1, 2], A[2, 2], A[1, 3], A[2, 3], A[1, 4], A[2, 4]$  • The address of element  $A[i, j]$  can be obtained by expression  **$\text{Loc}(A[i, j]) = L_0 + (j-1)*2 + i-1$**  • In general for two dimensional array consisting of  $n$  rows and  $m$  columns the address element  $A[i, j]$  is given by  **$\text{Loc}(A[i, j]) = L_0 + (j-1)*n + (i-1)$**

- In row major matrix, array can be generalized to arbitrary lower and upper bound in its subscripts, assume

that  $b_1 \leq i \leq u_1$  and  $b_2 \leq j \leq u_2$

**b1, u2**

**b1, b2**  
 $[1, 1][1, 2][1, 3][1, m][1, 2, 1][2, 2][2, 3][2, m]$   
**row 1 row 2**  
**col 1 col 2 col 3 col 4**  $[1, 1][1, 2][1, 3][1, 4][2, 1][2, 2][2, 3][2, 4]$   
 $[n, 1][n, 2][n, 3][n, m]$   
**u1, b2**  
**Row major matrix** **Column major matrix**  
**No of Columns = m = u2 - b2 + 1**

- For row major matrix :  $Loc(A[i, j]) = L_0 + (i - b_1) * (u_2 - b_2 + 1) + (j - b_2)$

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **1**



## Linear Data Structure

### Applications of Array

1. Symbol Manipulation (matrix representation of polynomial equation)
2. Sparse Matrix

### Symbol Manipulation using Array

- We can use array for different kind of operations in polynomial equation such as addition, subtraction, division, differentiation etc...
- We are interested in finding suitable representation for polynomial so that different operations like addition, subtraction etc... can be performed in efficient manner
- Array can be used to represent Polynomial equation

#### • Matrix Representation of Polynomial equation

	$Y$	$Y^2$	$Y^3$	$Y^4$
$X$	$XY$	$XY^2$	$XY^3$	$XY^4$
$X^2$	$X^2Y$	$X^2Y^2$	$X^2Y^3$	$X^2Y^4$
$X^3$	$X^3Y$	$X^3Y^2$	$X^3Y^3$	$X^3Y^4$
$X^4$	$X^4Y$	$X^4Y^2$	$X^4Y^3$	$X^4Y^4$

e.g.  $2x^2 + 5xy + y^2$

is represented in matrix form as below  $Y$



$Y^2Y^3Y^4$   
e.g.  $x^2+3xy+Y^2+Y-X$

X  
 $X^2X^3X^4$

0	0	1
0	5	0

2	0	0
0	0	0
0	0	0

X  
 $X^2X^3X^4$

0	0
-1	3
1	0

0	0	0	0	0
0	0	0	0	0

is represented in matrix form as below Y

$Y^2Y^3Y^4$

- Once we have algorithm for converting the polynomial equation to an array representation and another algorithm for converting array to polynomial equation, then different operations in array (matrix) will be corresponding operations of polynomial equation

## What is sparse matrix? Explain

- An  $m \times n$  matrix is said to be sparse if “many” of its elements are zero.
- A matrix that is not sparse is called a dense matrix.
- We can devise a simple representation scheme whose space requirement equals the size of the non zero elements.

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure 2



## Linear Data Structure

### • Example:-

- The non-zero entries of a sparse matrix may be mapped into a linear list in row-major order. For example the non-zero entries of  $4 \times 8$  matrix of below fig.(a) in row major order are 2, 1, 6, 7, 3, 9, 8, 4, 5

```

0 0 0 2 0 0 1 0
0 6 0 0 7 0 0 3
0 0 0 9 0 8 0 0
0 4 5 0 0 0 0 0

```

Fig (a)  $4 \times 8$  matrix

```

Terms 0 1 2 3 4 5 6 7 8
Row 1 1 2 2 2 3 3 4 4
Column 4 7 2 5 8 4 6 2 3
Value 2 1 6 7 3 9 8 4 5

```

Fig (b) Linear Representation of above matrix

- To construct matrix structure we need to record
  - (a) Original row and columns of each non zero entries
  - (b) No of rows and columns in the matrix

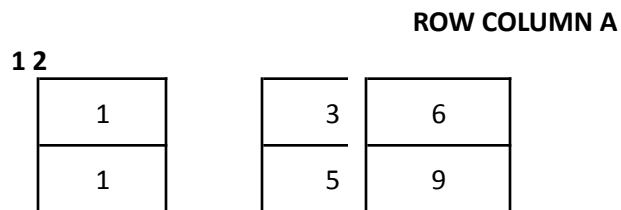
- So each element of the array into which the sparse matrix is mapped need to have three fields: row, column and value
- A corresponding amount of time is saved creating the linear list representation over initialization of two dimension array.

```

          0 0 6 0 9 0 0
A =       12 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0
2 0 0 7 8 0 4 10 0 0 0 0 0 0 0 0 5

```

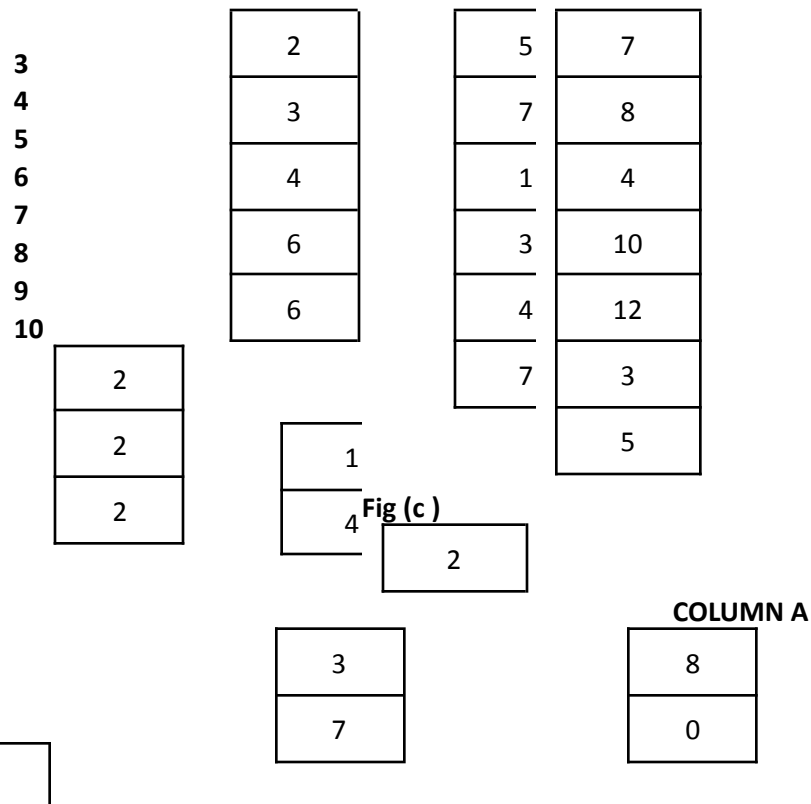
- Here from  $6 \times 7 = 42$  elements, only 10 are non zero.  $A[1,3]=6$ ,  $A[1,5]=9$ ,  $A[2,1]=2$ ,  $A[2,4]=7$ ,  $A[2,5]=8$ ,  $A[2,7]=4$ ,  $A[3,1]=10$ ,  $A[4,3]=12$ ,  $A[6,4]=3$ ,  $A[6,7]=5$ .
- One basic method for storing such a sparse matrix is to store non-zero elements in one dimensional array and to identify each array elements with row and column indices fig (c).

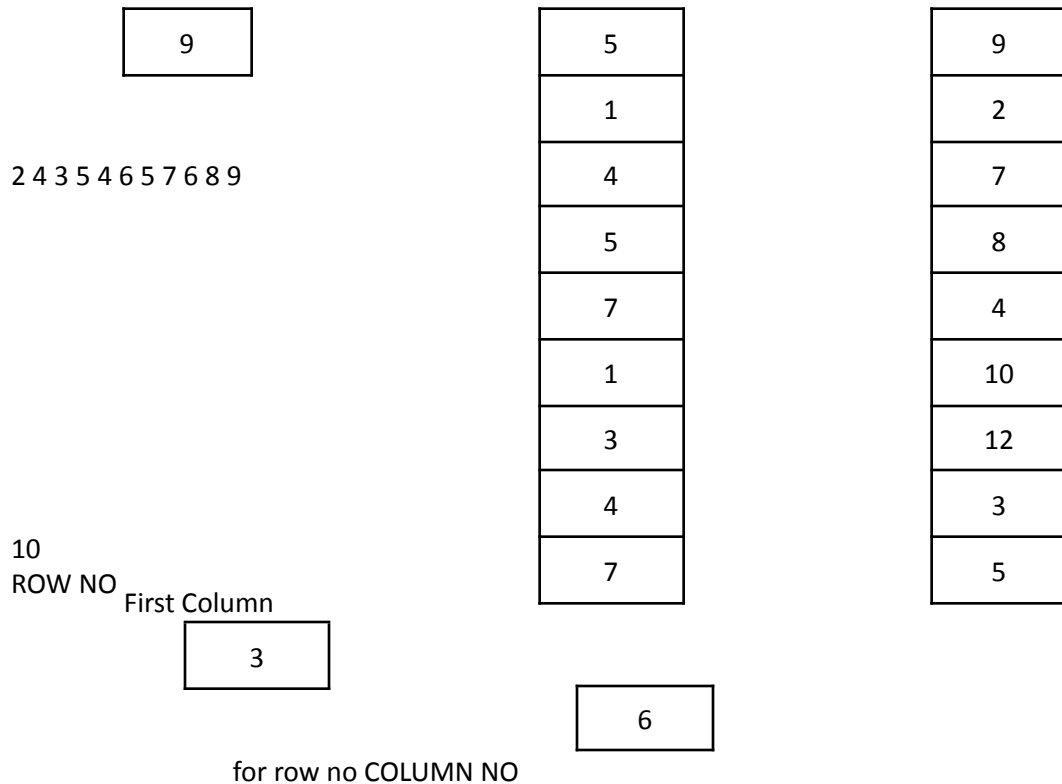


Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **3**



## Linear Data Structure





**Fig(d)**

- A more efficient representation in terms of storage requirement and access time to the row of the matrix is shown in Fig (d). The row vector changed so that its  $i^{\text{th}}$  element is the index to the first of the column indices for the element in row  $i$  of the matrix.

## Linked Representation of Sparse matrix

Typical node to represent non-zero element is

Row Number	Column Number	Value	Pointer To Next Node
------------	---------------	-------	----------------------

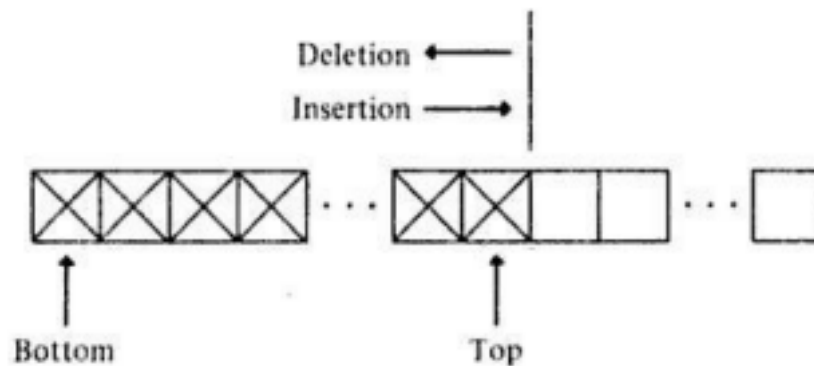
1 3 6 1 5 9 2 1 2 2 4 7

2 5 8 2 7 4 3 1 10 4 3 12

6 4 3 6 7 5 NULL

## Write algorithms for Stack Operations – PUSH, POP, PEEP

- A linear list which allows insertion and deletion of an element at one end only is called stack.
- The insertion operation is called as **PUSH** and deletion operation as **POP**.
- The most and least accessible elements in stack are known as top and bottom of the stack respectively.
- Since insertion and deletion operations are performed at one end of a stack, the elements can only be removed in the opposite orders from that in which they were added to the stack; such a linear list is referred to as a LIFO (last in first out) list.



Alternative representation of a stack

- A pointer TOP keeps track of the top element in the stack. Initially, when the stack is empty, TOP has a value of “one” and so on.
- Each time a new element is inserted in the stack, the pointer is incremented by “one” before, the element is placed on the stack. The pointer is decremented by “one” each time a deletion is made from the stack.

## Applications of Stack

- Recursion
- Keeping track of function calls
- Evaluation of expressions
- Reversing characters
- Servicing hardware interrupts
- Solving combinatorial problems using backtracking.

### Procedure : PUSH (S, TOP, X)

- This procedure inserts an element x to the top of a stack which is represented by a vector S containing N elements with a pointer TOP denoting the top element in the stack.

1. [Check for stack overflow]
  - If  $TOP \geq N$
  - Then write ('STACK OVERFLOW')
  - Return
2. [Increment TOP]
  - $TOP \leftarrow TOP + 1$
3. [Insert Element]
  - $S[TOP] \leftarrow X$
4. [Finished]
  - Return

### Function : POP (S, TOP)

- This function removes the top element from a stack which is represented by a vector S and returns this element. TOP is a pointer to the top element of the stack.

1. [Check for underflow of stack]
  - If  $TOP = 0$
  - Then Write ('STACK UNDERFLOW ON POP')
  - Take action in response to underflow
  - Return
2. [Decrement Pointer]
  - $TOP \leftarrow TOP - 1$
3. [Return former top element of stack]
  - Return ( $S[TOP + 1]$ )

### Function : PEEP (S, TOP, I)

- This function returns the value of the  $i^{\text{th}}$  element from the TOP of the stack which is represented by a vector S containing N elements. The element is not deleted by this function.

1. [Check for stack Underflow]
  - If  $TOP - I + 1 \leq 0$
  - Then Write ('STACK UNDERFLOW ON PEEP')
  - Take action in response to Underflow
  - Exit
2. [Return  $I^{\text{th}}$  element from top of the stack]
  - Return ( $S[TOP - I + 1]$ )

**Write an algorithm to change the  $i^{\text{th}}$  value of stack to value X**

**PROCEDURE : CHANGE (S, TOP, X, I)**

- This procedure changes the value of the  $I^{\text{th}}$  element from the top of the stack to the value containing in X. Stack is represented by a vector S containing N elements.

1. [Check for stack Underflow]  
If  $\text{TOP} - I + 1 \leq 0$   
Then Write ('STACK UNDERFLOW ON CHANGE')  
Return
2. [Change  $I^{\text{th}}$  element from top of the stack]  
 $S[\text{TOP} - I + 1] \leftarrow X$
3. [Finished]  
Return



## Linear Data Structure

**Write an algorithm which will check that the given string belongs to following grammar or not.  $L = \{wcw^R \mid w \in \{a,b\}^*\}$  (Where  $w^R$  is the reverse of  $w$ )**

**Algorithm : RECOGNIZE**

- Given an input string named STRING on the alphabet {a, b, c} which contains a blank in its rightmost character position and function NEXTCHAR which returns the next symbol in STRING, this algorithm determines whether the contents of STRING belong to the above language. The vector S represents the stack, and TOP is a pointer to the top element of the stack.

**1.** [Initialize stack by placing a letter 'c' on the top]

TOP  $\leftarrow$  1

S [TOP]  $\leftarrow$  'c'

2. [Get and stack symbols either 'c' or blank is encountered]
  - NEXT  $\leftarrow$  NEXTCHAR (STRING)
  - Repeat while NEXT  $\neq$  'c'
    - If NEXT = ' '
      - Then Write ('Invalid String')
      - Exit
    - Else Call PUSH (S, TOP, NEXT)
    - NEXT  $\leftarrow$  NEXTCHAR (STRING)
3. [Scan characters following 'c'; Compare them to the characters on stack]
  - Repeat While S [TOP]  $\neq$  'c'
    - NEXT  $\leftarrow$  NEXTCHAR (STRING)
    - X  $\leftarrow$  POP (S, TOP)
    - If NEXT  $\neq$  X
      - Then Write ('INVALID STRING')
      - Exit
4. [Next symbol must be blank]
  - If NEXT  $\neq$  ' '
    - Then Write ('VALID STRING')
    - Else Write ('INVALID STRING')
5. [Finished]
  - Exit



## Linear Data Structure

**Write an algorithm for push, pop and empty operations on stack. Using above functions write an algorithm to determine if an input character string is of the form  $a^i b^j$  where  $i \geq 1$  i.e. no of a should be equal to no of b**

### Algorithm RECOGNIZE

- Given an input string named STRING on alphabet 'a' and 'b' which contain blank (' ') on right most character function NEXTCHAR which returns the next symbol from STRING. This algorithm determines if an input string is of form  $a^i b^j$  where  $i > 1$  i.e no of 'a' should be equal to no of 'b'. the vector S represent the stack and TOP is the pointer to the top element of stack. Counter is a counter B for 'b' occurrence.

1. [Initialize stack and counter]



```

TOP  0
COUNTER_B  0
2. [Get and stack character 'a' from whole string and count the occurrence of 'b' ]
NEXT  NEXTCHAR(STRING)
Repeat while NEXT != ''
    IF NEXT = 'a'
    Then PUSH (S,TOP,NEXT)
    Else COUNTER_B  COUNTER_B+1
    NEXT  NEXTCHAR(STRING)
3. [Pop the stack until empty and decrement the COUNTER_B]
Repeat while TOP != 0
    POP (S,TOP)
    COUNTER_B  COUNTER_B-1
4. [Check for grammar]
If COUNTER_B != 0
Then write ('INVALID STRING')
Else write ('VALID STRING')

```

## What is recursion? Write a C program for GCD using recursion.

- A procedure that contains a procedure call to itself or a procedure call to second procedure which eventually causes the first procedure to be called is known as recursive procedure.
- There are two important conditions that must be satisfied by any recursive procedure a.
  - Each time a procedure calls itself it must be nearer in some sense to a solution
  - b. There must be a decision criterion for stopping the process or computation
- There are two types of recursion
  - Primitive Recursion: this is recursive defined function. E.g. Factorial function
  - Non-Primitive Recursion: this is recursive use of procedure. E.g. Find GCD of given two numbers

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure 9



## Linear Data Structure

### *C program for GCD using recursion*

```

#include<stdio.h>
int Find_GCD(int, int);

void main()
{
    int n1, n2, gcd;
    scanf("%d %d",&n1, &n2);
    gcd = Find_GCD(n1, &n2);
    printf("GCD of %d and %d is %d", n1, n2, gcd);
}

```

```

    }

    int Find_GCD(int m, int n)
    {
        int gcdVal;
        if (n>m)
        {
            gcdVal = Find_GCD(n,m);
        }
        else if (n==0)
        {
            gcdVal = m;
        }
        else
        {
            gcdVal = Find_GCD(n, m%n);
        }
        return(gcdVal);
    }

```

## Write an algorithm to find factorial of given no using recursion

### Algorithm: FACTORIAL

Given integer N, this algorithm computes factorial of N. Stack A is used to store an activation record associated with each recursive call. Each activation record contains the current value of N and the current return address RET\_ADDE. TEMP\_REC is also a record which contains two variables PARAM & ADDRESS.TOP is a pointer to the top element of stack A. Initially return address is set to the main calling address. PARAM is set to initial value N.

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **10**



## Linear Data Structure

1. [Save N and return Address]  
CALL PUSH (A, TOP, TEMP\_REC)
2. [Is the base criterion found?]  
If N=0  
then FACTORIAL = 1  
GO TO Step 4  
Else PARAM = N-1

ADDRESS Step 3

GO TO Step 1

**3. [Calculate N!]**

FACTORIAL N \* FACTORIAL

**4. [Restore previous N and return address]**

TEMP\_REC POP(A,TOP)

(i.e. PARAM N, ADDRESS RET\_ADDR)

GO TO ADDRESS

## Give difference between recursion and iteration

Iteration	Recursion
In iteration, a problem is converted into a train of steps that are finished one at a time, one after another	Recursion is like piling all of those steps on top of each other and then quashing them all into the solution.
With iteration, each step clearly leads onto the next, like stepping stones across a river	In recursion, each step replicates itself at a smaller scale, so that all of them combined together eventually solve the problem.
Any iterative problem is solved recursively	Not all recursive problem can solved by iteration
It does not use Stack	It uses Stack

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **11**



## Linear Data Structure

### Write an algorithm to convert infix expression to postfix expression.

Symbol	Input precedence function F	Stack precedence function G	Rank function R
+, -	1	2	-1
*, /	3	4	-1
^	6	5	-1
Variables	7	8	1
(	9	0	-

)	0	-	-
---	---	---	---

### Algorithm : REVPOL

- Given an input string INFIX containing an infix expression which has been padded on the right with ‘)’ and whose symbol have precedence value given by above table, a vector S used as a stack and a NEXTCHAR which when invoked returns the next character of its argument. This algorithm converts INFIX into reverse polish and places the result in the string POLISH. The integer variable TOP denotes the top of the stack. Algorithm PUSH and POP are used for stack manipulation. The integer variable RANK accumulates the rank of expression. Finally the string variable TEMP is used for temporary storage purpose.



## Linear Data Structure

- [Initialize stack]  
TOP ← 1  
S[TOP] ← ‘(‘
- [Initialize output string and rank count ]  
POLISH ← ‘ ‘  
RANK ← 0
- [Get first input symbol]  
NEXT ← NEXTCHAR (INFIX)
- [Translate the infix expression ]  
Repeat thru step 7 while NEXT != ‘ ‘

5. [Remove symbols with greater precedence from stack]

IF TOP < 1

Then write ('INVALID')

EXIT

Repeat while G (S[TOP]) > F(NEXT)

TEMP POP (S, TOP)

POLISH POLISH O TEMP

RANK RANK + R(TEMP)

IF RANK <1

Then write 'INVALID')

EXIT

6. [Are there matching parentheses]

IF G(S[TOP]) != F(NEXT)

Then call PUSH (S,TOP, NEXT)

Else POP (S,TOP)

7. [Get next symbol]

NEXT NEXTCHAR(INFIX)

8. [Is the expression valid]

IF TOP != 0 OR RANK != 1

Then write ('INVALID ')

Else write ('VALID ')



## Linear Data Structure

**Trace the conversion of infix to postfix form in tabular**

**form. (i) ( A + B \* C / D - E + F / G / ( H + I ) )**

Input Symbol	Content of stack	Reverse polish	Rank
	(		0
(	((		0

A	((		0
+	(( +	A	1
B	(( + B	A	1
*	(( + *	A B	2
C	(( + * C	A B	2
/	(( + /	A B C *	2
D	(( + / D	A B C *	2
-	(( -	A B C * D / +	1
E	(( - E	A B C * D / +	1
+	(( +	A B C * D / + E -	1
F	(( + F	A B C * D / + E -	1
/	(( + /	A B C * D / + E - F	2
G	(( + / G	A B C * D / + E - F	2
/	(( + /	A B C * D / + E - F G /	2
(	(( + / (	A B C * D / + E - F G /	2
H	(( + / ( H	A B C * D / + E - F G /	2
+	(( + / ( +	A B C * D / + E - F G / H	3
I	(( + / ( + I	A B C * D / + E - F G / H	3
)	(( + /	A B C * D / + E - F G / H I +	3
)	(	A B C * D / + E - F G / H I + / +	1
)		A B C * D / + E - F G / H I + / +	1

Postfix expression is: **A B C \* D / + E - F G / H I + / +**



## Linear Data Structure

(ii)  $(A + B) * C + D / (B + A * C) + D$

Input Symbol	Content of stack	Reverse polish	Rank
	(		0
(	((		0
A	(( A		0
+	(( +	A	1
B	(( + B	A	1
)	(	A B +	1
*	( *	A B +	1
C	( * C	A B +	1
+	( +	A B + C *	1
D	( + D	A B + C *	1
/	( + /	A B + C * D	2
(	( + / (	A B + C * D	2
B	( + / ( B	A B + C * D	2
+	( + / ( +	A B + C * D B	3
A	( + / ( + A	A B + C * D B	3

*	( + / ( + *	A B + C * D B A	4
C	( + / ( + * C	A B + C * D B A	4
)	( + /	A B + C * D B A C * +	3
+	( +	A B + C * D B A C * + / +	1
D	( + D	A B + C * D B A C * + / +	1
)		A B + C * D B A C * + / + D +	1

Postfix expression is: **A B + C \* D B A C \* + / + D +**



## Linear Data Structure

**Convert the following string into prefix: A-B/(C\*D^E)**

**Step-1 : reverse infix expression**

) E ^ ) D \* C ( ( / B - A

**Step-2 : convert '(' to ')' and ')' to '(' and append extra ')' at last ( E ^ ( D \***

C ) ) / B - A

**Step-3 : Now convert this string to postfix**

Input Symbol	Content of stack	Reverse polish	Rank
--------------	------------------	----------------	------



	(		0
(	((		0
E	((E		0
^	((^	E	1
(	((^(	E	1
D	((^(D	E	1
*	((^(*	ED	2
C	((^(*C	ED	2
)	((^	EDC*	2
)	(	EDC*^	1
/	(/	EDC*^	1
B	(/B	EDC*^	1
-	(-	EDC*^B/	1
A	(-A	EDC*^B/	1
)		EDC*^B/A-	1

**Step 4 : Reverse this postfix expression**

**- A / B ^ \* C D E**



## Linear Data Structure

**Translate the following string into Polish notation and trace the content of stack:  $(a + b^c d) * (e + f / d)$**

**Step-1 : reverse infix expression**

$) d / f + e ( * ) d ^ c ^ b + a ($

**Step-2 : convert '(' to ')' and ')' to '(' and append extra ')' at last**

$( d / f + e ) * ( d ^ c ^ b + a ) )$

**Step-3 : Now convert this string to postfix**

Input symbol	Content of stack	Reverse polish	Rank
	(		0
(	((		0
d	((d		0
/	((/	d	1
f	((/f	d	1
+	((+	df/	1
e	((+e	df/	1
)	(	df/e+	1
*	(+	df/e+	1
(	(*	df/e+	1
d	(*d	df/e+	1
^	(*^	df/e+d	2
c	(*^c	df/e+d	2
^	(*^^	df/e+dc	3
b	(*^^b	df/e+dc	3
+	(*+	df/e+dc b ^^	2
a	(*+a	df/e+dc b ^^	2
)	(*	df/e+dc b ^^ a +	2
)		df/e+dc b ^^ a + *	1

**Step 4 : Reverse this postfix expression**

**\* + a ^ ^ b c d + e / f d**

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **17**



## Linear Data Structure

**Write an algorithm for evaluation of postfix expression and evaluation the following expression showing every status of stack in tabular form.**

**(i) 5 4 6 + \* 4 9 3 / + \* (ii) 7 5 2 + \* 4 1 1 + / -**

**Algorithm: EVALUAE\_POSTFIX**

- Given an input string POSTFIX representing postfix expression. This algorithm is going to evaluate postfix expression and put the result into variable VALUE. A vector S is used as a stack PUSH and POP are the function used for manipulation of stack. Operand2 and operand1 are temporary variable TEMP is used for temporary variable NEXTCHAR is a function which when invoked returns the next character. PERFORM\_OPERATION is a function which performs required operation on OPERAND1 AND OPERAND2.

**1. [Initialize stack and value]**

TOP 1

VALUE 0

**2. [Evaluate the prefix expression]**

Repeat until last character

TEMP NEXTCHAR (POSTFIX)

If TEMP is DIGIT

Then PUSH (S, TOP, TEMP)

Else OPERAND2 POP (S, TOP)

OPERAND1 POP (S, TOP)

VALUE    PERFORM\_OPERATION(OPERAND1, OPERAND2, TEMP)  
 PUSH (S, POP, VALUE)

3. [Return answer from stack]

Return (POP (S, TOP))

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **18**



## Linear Data Structure

**Evaluate (i): 5 4** pop two values from stack opn2 =

**6 + \* 4 9 3 / + \***

Empty Stack

Read and push  
 operands 5, 4, 6

**6**

**4**

**5**

**7**

7, opn1 = 50, and  
 push the answer

Read Operator \*,

Read Operator +, pop

two values from stack  
 opn2 = 6, opn1 = 4,  
 and push the answer  
 10

Read Operator +, pop  
 two values from stack  
 opn2 = 3, opn1 = 4,  
 and push the answer  
 7

**10 5**

**3**

**4**

Read Operator \*, pop  
 two values  
 from stack opn2 = 10,  
 opn1 = 5, and push  
 the answer 50

**3**  
**9**  
**4**  
 Read Operator /, pop  
 two values  
 from stack opn2 = 3,  
 opn1 = 9, and  
 Read and push  
 operands 4, 9, 3

**50**

push the answer 3 **350**

350  
50

Poped value **350** is the answer **50**

**Evaluate (ii) : \* 7 5 2 + \* 4**

**1 1 + /** - Empty Stack

operands 7, 5, 2 answer 47

**2 5 7**

**7 5**

**49**

Read and push  
operands 4, 1, 1

Read Operator -,  
pop two values **2**  
from stack opn2  
= 2, opn1 = 49,  
and push the  
push the answer = 2, opn1 = 4,  
and push the  
answer 2

Read Operator /,  
pop two values **2 4**  
from stack opn2  
push the answer  
49

Read Operator +,  
pop two values **1**  
from stack opn2  
= 1, opn1 = 1,  
and **4**

push the answer 2 **47**

**49**

answer  
**49**

**49**

Poped value **47** is the

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **19**



## Linear Data Structure

**Consider the following arithmetic expression P, written in postfix notation.  
Translate it in infix notation and evaluate. P: 12, 7, 3, -, /, 2, 1, 5, +, \*, +**

Same Expression in infix notation is : **( 12 / ( 7 - 3 ) ) + ( ( 5 + 1 ) \* 2 )**

operands 12, 7, 3

Empty Stack

Read and push

pop two values    push the answer  
 from stack opn2 = 12  
 3, opn1 = 7, and  
 push the answer 4

**3**

**7**

Read Operator +, **12 12**  
 pop two values  
 from stack opn2 =  
 12, opn1 = 3, and  
 push the answer  
 15

**4**

**12**

Read Operator \*,  
 pop two values  
 from stack opn2 =  
 6, opn1 = 2, and

Read Operator -,

**6**

**2**

Read Operator /,  
 pop two values    Read Operator +, **3**  
 from stack opn2 =    pop two values  
 4, opn1 = 12, and    from stack opn2 =  
 push the answer 3    5, opn1 = 1, and

**5 1 2**

Read and push  
 operands 2, 1, 5

push the answer 6 **15**

**3**

**3**

**3**

Poped value **15** is the answer

## Explain Difference between Stack and Queue.

Stack	Queue
A Linear List Which allows insertion or deletion of an element at one end only is called as Stack	A Linear List Which allows insertion at one end and deletion at another end is called as Queue
Since insertion and deletion of an element are performed at one end of the stack, the elements can only be removed in the opposite order of insertion.	Since insertion and deletion of an element are performed at opposite end of the queue, the elements can only be removed in the same order of insertion.
Stack is called as Last In First Out (LIFO) List.	Queue is called as First In First Out (FIFO) List.
The most and least accessible elements are called as TOP and BOTTOM of the stack	Insertion of element is performed at FRONT end and deletion is performed from REAR end
Example of stack is arranging plates in one above one.	Example is ordinary queue in provisional store.
Insertion operation is referred as PUSH and deletion operation is referred as POP	Insertion operation is referred as ENQUEUE and deletion operation is referred as DQUEUE
Function calling in any languages uses Stack	Task Scheduling by Operating System uses queue



## Linear Data Structure

**Explain following:**

**(i) Queue (ii) Circular Queue (iii) DQUEUE (iv) Priority Queue**

**(i) Queue**

- A linear list which permits deletion to be performed at one end of the list and insertion at the other end is called queue.
- The information in such a list is processed FIFO (first in first out) or FCFS (first come first served) pattern.
- Front is the end of queue from that deletion is to be performed.
- Rear is the end of queue at which new element is to be inserted.
- The process to add an element into queue is called **Enqueue**
- The process of removal of an element from queue is called **Dequeue**.
- The familiar and traditional example of a queue is Checkout line at Supermarket Cash Register where the first person in line is usually the first to be checked out.

Front

Insertion

Deletion

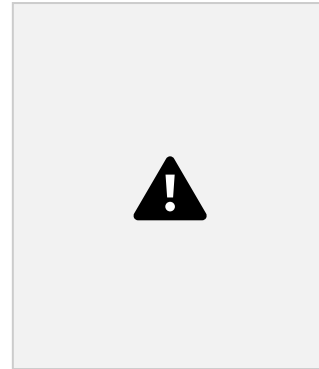
Rear

**(ii) Circular Queue**

- A more suitable method of representing simple queue which prevents an excessive use of memory is to arrange the elements  $Q[1], Q[2], \dots, Q[n]$  in a circular fashion with  $Q[1]$  following  $Q[n]$ , this is called circular queue
- In a standard queue data structure re-buffering problem occurs for each **dequeue** operation. To solve this problem by joining the front and rear ends of a queue to make the queue as a circular queue
- Circular queue is a linear data structure. It follows FIFO principle.
- In circular queue the last node is connected back to the first node to make a circle.
- Circular linked list follow the First In First Out principle
- Elements are added at the rear end and the elements are deleted at front end of the queue
- Both the front and the rear pointers point to the beginning of the array.
- It is also called as “Ring buffer”.



## Linear Data Structure



### (iii) Dequeue

- A dequeue (double ended queue ) is a linear list in which insertion and deletion are performed from the either end of the structure.
- There are two variations of Dqueue
  - Input restricted dequeue- allows insertion at only one end
  - Output restricted dequeue- allows deletion from only one end
- Such a structure can be represented by following fig.



### (iv) Priority Queue

- A queue in which we are able to insert remove items from any position based on some property (such as priority of the task to be processed) is often referred as priority queue.
- Below fig. represent a priority queue of jobs waiting to use a computer.
- Priorities of 1, 2, 3 have been attached with jobs of real time, online and batch respectively. Therefore if a job is initiated with priority i, it is inserted immediately at the end of list of other jobs with priorities i. Here jobs are always removed from the front of queue





# Linear Data Structure

Task Identification

$R_1$	$R_2$	...	$R_{i-1}$	$O_1$	$O_2$	...	$O_{j-1}$	$B_1$	$B_2$	...	$B_{k-1}$	...
1	1	...	1	2	2	...	2	3	3	...	3	...

Priority

$R_i O_j B_k$

**Fig (a) : Priority Queue viewed as a single queue with insertion allowed at any position.** Priority 1

				Priority 3				
$R_1$	$R_2$	...	$R_k$	$B_1$	$B_2$	...	$B_{k-1}$	...

Priority 2			$R_i$
$O_1$	$O_2$	...	$C$
			$O_j B_k$

**Fig (b) : Priority Queue viewed as a Viewed as a set of queue**



## Linear Data Structure

### Write algorithms of basic primitive operations for Queue

#### Procedure: QINSERT\_REAR (Q, F, R, N, Y)

- Given F and R pointers to the front and rear elements of a queue respectively. Queue Q consisting of N elements. This procedure inserts Y at rear end of Queue.

1. [Overflow]  
IF  $R \geq N$   
Then write ('OVERFLOW')  
Return
2. [Increment REAR pointer]  
 $R = R + 1$
3. [Insert element ]  
 $Q[R] = Y$
4. [Is front pointer properly set]

```

IF F=0
Then F = 1
Return

```

### Function: QDELETE\_FRONT (Q, F, R)

- Given F and R pointers to the front and rear elements of a queue respectively. Queue Q consisting of N elements. This function deleted an element from front end of the Queue.

```

1. [Underflow]
   IF F= 0
   Then write ('UNDERFLOW')
   Return(0) (0 denotes an empty Queue)
2. [Decrement element]
   Y = Q[F]
3. [Queue empty?]
   IF F=R
   Then F = R - 1
   Else F = F-1 (increment front pointer)
4. [Return element]
   Return (Y)
   Then F = F - 1
   Return

```



## Linear Data Structure

### Write algorithms of basic primitive operations for Circular Queue

#### Procedure: CQINSERT (F, R, Q, N, Y)

- Given F and R pointers to the front and rear elements of a circular queue respectively. Circular queue Q consisting of N elements. This procedure inserts Y at rear end of Circular queue.

```

1. [Reset Rear Pointer]
   If R = N
   Then R ← 1
   Else R ← R + 1
2. [Overflow]
   If F = R
   Then Write ('Overflow')
   Return

```

3. [Insert element]  
 $Q[R] \leftarrow Y$
4. [Is front pointer properly set?]  
 If  $F = 0$   
 Then  $F \leftarrow 1$   
 Return

### Function CQDELETE (F, R, Q, N)

- Given F and R pointers to the front and rear elements of a Circular queue respectively. Circular Queue Q consisting of N elements. This function deleted an element from front end of the Circular Queue. Y is temporary pointer variable.



## Linear Data Structure

1. [Underflow?]  
 If  $F = 0$   
 Then Write ('UNDERFLOW')  
 Return (0)
2. [Delete Element]  
 $Y \leftarrow Q[F]$
3. [Queue Empty?]  
 If  $F = R$   
 Then  $F \leftarrow R \leftarrow 0$   
 Return (Y)
4. [Increment front pointer]  
 If  $F = N$

```

Then  $F \leftarrow 1$ 
Else  $F \leftarrow F + 1$ 
Return (Y)

```

## Write algorithms of basic primitive operations for DQueue

### Procedure DQINSERT\_FRONT (Q, F, R, N, Y)

- Given F and R pointers to the front and rear elements of a queue, a queue consisting of N elements and an element Y, this procedure inserts Y at the front of the queue.

```

1. [Overflow]
   IF  $F = 0$ 
     Then write ('EMPTY')
     Return
   IF  $F = 1$ 
     Then write ('OVERFLOW')
     Return
2. [Decrement front pointer]
    $F \leftarrow F - 1$ 
3. [Insert element]
    $Q[F] \leftarrow Y$ 
   Return

```

### Procedure DQDELETE\_REAR (Q, F, R)

- Given F and R pointers to the front and rear elements of a queue. And a queue Q to which they correspond, this function deletes and returns the last element from the front end of a queue. And Y is temporary variable.

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **26**



## Linear Data Structure

```

1. [Underflow]
   IF  $R = 0$ 
     Then write ('UNDERFLOW')
     Return(0)
2. [Delete element]
    $Y \leftarrow Q[R]$ 
3. [Queue empty?]
   IF  $R = F$ 
     Then  $R \leftarrow F - 1$ 
   Else  $R \leftarrow R - 1$  (decrement front pointer)
4. [Return

```

element]  
Return (Y)

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **27**

**PROCEDURE DQUEUE\_DISPLAY (F,R,Q)**

**Linear Data Structure**



- Given F and Rare pointers to the front and rear elements of a queue, a queue consist of N elements. This procedure display Queue contents

```
1. [Check for empty]
   IF F >= R
   Then write ('QUEUE IS EMPTY')
   Return
```

2. [Display content]
 

```
FOR (I=FRONT; I<=REAR; I++)
    Write (Q[I])
```
3. [Return Statement]
 

```
Return
```

**Consider the following queue, where queue is a circular queue having 6 memory cells. Front=2, Rear=4**

Queue: \_, A, C, D, \_, \_

Describe queue as following operation take place:

F is added to the queue

Two letters are deleted

R is added to the queue

S is added to the queue

One letter is deleted

Positions	1 2 3 4 5 6					
Initial Position of Queue, <b>Front=2, Rear=4</b> F is added to queue, <b>Front=2, Rear=5</b> Two letters are deleted, <b>Front=4, Rear=5</b> R is added to the queue, <b>Front=4, Rear=6</b> S is added to the queue, <b>Front=4, Rear=1</b> One letter is deleted, <b>Front=5, Rear=1</b>		A	C	D		
		A	C	D	F	
				D	F	
				D	F	R
	S			D	F	R
	S				F	R



## Linked List

### 1. Linear Data Structure and their linked storage representation.

There are many applications where sequential allocation method is unacceptable because of following

characteristics

- Unpredictable storage requirement
- Extensive manipulation of stored data

The linked allocation method of storage can result in both efficient use of computer storage and computer time.

- A linked list is a non-sequential collection of data items.
- The concept of a linked list is very simple, for every data item in the linked list, there is an associated pointer that would give the memory allocation of the next data item in the linked list.
- The data items in the linked list are not in a consecutive memory locations but they may be anywhere in memory.
- Accessing of these data items is easier as each data item contains within itself the address of the next data item.

10<sup>next</sup> 20<sup>next</sup> 30<sup>next</sup> 40<sup>null</sup>

A Linked List

## 2. What is linked list? What are different types of linked list? OR Write a short note on singly, circular and doubly linked list. OR Advantages and disadvantages of singly, circular and doubly linked list.

- A linked list is a collection of objects stored in a list form.
- A linked list is a sequence of items (objects) where every item is linked to the next.
- A linked list is a non-primitive type of data structure in which each element is dynamically allocated and in which elements point to each other to define a linear relationship.
- Elements of linked list are called nodes where each node contains two things, data and pointer to next node.
- Linked list require more memory compared to array because along with value it stores pointer to next node.
- Linked lists are among the simplest and most common data structures. They can be used to implement other data structures like stacks, queues, and symbolic expressions, etc...

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **1**



# Linked List

Node

info link



## Types of linked list

**Data Pointer to  
next node**

### Operations on linked list • Insert

- Insert at first position
- Insert at last position
- Insert into ordered list
- Delete
- Traverse list (Print list)
- Copy linked list

### Singly Linked List

- It is basic type of linked list.

```
// C Structure to represent a  
node struct node  
{  
    int info  
    struct node *link  
};
```

- Each node contains data and pointer to next node.
- Last node's pointer is null.
- Limitation of singly linked list is we can traverse only in one direction, forward direction.

A<sup>next</sup> B<sup>next</sup> C<sup>next</sup> D null

Singly Linked List

### Circular Linked List

- Circular linked list is a singly linked list where last node points to first node in the list. •  
It does not contain null pointers like singly linked list.
- We can traverse only in one direction that is forward direction.
- It has the biggest advantage of time saving when we want to go from last node to first node, it directly points to first node.
- A good example of an application where circular linked list should be used is a timesharing problem solved by the operating system.



## Linked List

A<sup>next</sup> B<sup>next</sup> C<sup>next</sup> D<sup>next</sup>

Circular Linked List

### Doubly Linked list

- Each node of doubly linked list contains data and two pointers to point previous (LPTR) and next (RPTR) node.

{

**Node**

**LPTR info RPTR**

```
// C Structure to represent a node
node struct node
```

**Pointer to  
previous node**

**Data**

**Pointer to  
next node**

```
struct node
*rptra; };
```

- Main advantage of doubly linked list is we can traverse in any direction, forward or reverse.
- Other advantage of doubly linked list is we can delete a node with little trouble, since we have pointers to the previous and next nodes. A node on a singly linked list cannot be removed unless we have the pointer to its predecessor.
- Drawback of doubly linked list is it requires more memory compared to singly linked list because we need an extra pointer to point previous node.
- L and R in image denote left most and right most nodes in the list.
- Left link of L node and right link of R node is NULL, indicating the end of list for each direction.

null prev prev

A<sup>next</sup> B<sup>next</sup> C<sup>null</sup>

Doubly Linked List

L R

### 3. Discuss advantages and disadvantages of linked list over array.

#### Advantages of an array

1. We can access any element of an array directly means random access is easy
2. It can be used to create other useful data structures (queues, stacks)

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **3**



#### Linked List

3. It is light on memory usage compared to other structures

#### Disadvantages of an array

1. Its size is fixed
2. It cannot be dynamically resized in most languages
3. It is hard to add/remove elements
4. Size of all elements must be same.
5. Rigid structure (Rigid = Inflexible or not changeable)

#### Advantages of Linked List

1. **Linked lists are dynamic data structures:** That is, they can grow or shrink during execution of a program.
2. **Efficient memory utilization:** Here memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated (free) when it is no longer needed.
3. **Insertion and deletions are easier and efficient:** Linked list provide flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
4. **Elements of linked list are flexible:** It can be primary data type or user defined data types

#### Disadvantages of Linked List

1. Random access is not allowed. We have to access elements sequentially starting from the first node.  
So we cannot do binary search with linked lists.
2. It cannot be easily sorted
3. We must traverse 1/2 the list on average to access any element
4. More complex to create than an array
5. Extra memory space for a pointer is required with each element of the list

### 3. What are the advantages and disadvantages of stack and queue implemented using linked list over array?

Advantages and disadvantages of stack & queue implemented using linked list over array is described below,

## Insertion & Deletion Operation

- Insertion and deletion operations are known as push and pop operation in stack and as insert and delete operation in queue.
- In the case of an array, if we have n-elements list and it is required to insert a new element between the first and second element then n-1 elements of the list must be moved so as to make room for the new element.
- In case of linked-list, this can be accomplished by only interchanging pointers.
- Thus, insertion and deletions are more efficient when performed in linked list than array.

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **4**



## Linked List

### Searching a node

- If a particular node in a linked list is required, it is necessary to follow links from the first node onwards until the desired node is found.
- Whereas in the case of an array, directly we can access any node

### Join & Split

- We can join two linked list by assigning pointer of second linked list in the last node of first linked list.
  - Just assign null address in the node from where we want to split one linked list in two parts.
- Joining and splitting of two arrays is much more difficult compared to linked list.

### Memory

- The pointers in linked list consume additional memory compared to an array

### Size

- Array is fixed sized so number of elements will be limited in stack and queue.
- Size of linked list is dynamic and can be changed easily so it is flexible in number of elements

#### Insertion and deletion operations in Array and Linked-List

$X_1 X_2 X_3 X_4 X_5 X_6$

**Array** Insert Y at location 2. You have to move  $X_2, X_3, \dots, X_6$

$X_1 Y X_2 X_3 X_4 X_5 X_6$

$X_1 X_2 X_3 X_4$

**Linked List**

Insert Y at location 2. Just change

two pointers  $X_1 X_2 X_3 X_4$



## Linked List

**4. Write following algorithms for singly linked list.**

- 1) Insert at first position**
- 2) Insert at last position**
- 3) Insert in Ordered Linked list**
- 4) Delete Element**
- 5) Copy Linked List**

Few assumptions,

- We assume that a typical element or node consists of two fields namely; an information field called INFO and pointer field denoted by LINK. The name of a typical element is denoted by NODE.

### **Node**

**info link**

```
// C Structure to represent a node
{
int info
struct node *link };
```

**Data Pointer to next node**

```
struct node
```



## Linked List

### Function: INSERT( X, First )

Given X, a new element and FIRST is a pointer to the first element of a linked linear list. Typical node contains INFO and LINK fields. AVAIL is a pointer to the top element of the availability stack; NEW is a temporary pointer variable. This function inserts a new node at the first position of linked list. This function returns address of FIRST node.

#### 1 [Underflow?]

IF AVAIL = NULL

Then Write ("Availability Stack Underflow")

Return(FIRST)

#### 2 [Obtain address of next free Node]

NEW ← AVAIL

#### 3 [Remove free node from Availability Stack]

AVAIL ← LINK(AVAIL)

#### 4 [Initialize fields of new node and its link to the list]

INFO (NEW) ← X

LINK (NEW) ← FIRST

#### 5 [Return address of new node]

Return (NEW)

When INSERT is invoked it returns a pointer value to the variable FIRST

FIRST    INSERT (X, FIRST)

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure 7



## Linked List

### Function: INSEND( X, First ) (Insert at end)

Given X, a new element and FIRST is a pointer to the first element of a linked linear list. Typical node contains INFO and LINK fields. AVAIL is a pointer to the top element of the availability stack; NEW is a temporary pointer variable. This function inserts a new node at the last position of linked list. This function returns address of FIRST node.

#### 1 [Underflow?]

IF AVAIL = NULL

Then Write ("Availability Stack Underflow")

Return(FIRST)

#### 2 [Obtain address of next free Node]

NEW ← AVAIL

#### 3 [Remove free node from Availability Stack]

AVAIL ← LINK(AVAIL)

#### 4 [Initialize field of NEW node]

INFO (NEW) ← X

LINK (NEW)    NULL

**5 [Is the list empty?]**

If FIRST = NULL  
then Return (NEW)

**6 [Initialize search for a last node]**

SAVE    FIRST

**7 [Search for end of list]**

Repeat while LINK (SAVE)  $\neq$  NULL  
    SAVE    LINK (SAVE)

**8 [Set link field of last node to NEW]**

LINK (SAVE)    NEW

**9 [Return first node pointer]**

Return (FIRST)

**When INSERTEND is invoked it returns a pointer value to the variable FIRST**

**FIRST    INSERTEND (X, FIRST)**



## Linked List

### Insert a node into Ordered Linked List

• There are many applications where it is desirable to maintain an ordered linear list. The ordering is in increasing or decreasing order on INFO field. Such ordering results in more efficient processing. • The general algorithm for inserting a node into an ordered linear list is as below.

1. Remove a node from availability stack.
2. Set the field of new node.
3. If the linked list is empty then return the address of new node.
4. If node precedes all other nodes in the list then inserts a node at the front of the list and returns its address.
5. Repeat step 6 while information contain of the node in the list is less than the information content of the new node.
6. Obtain the next node in the linked list.
7. Insert the new node in the list and return address of its first node.





## Linked List

### Function: INSORD( X, FIRST )

Given X, a new element and FIRST is a pointer to the first element of a linked linear list. Typical node contains INFO and LINK fields. AVAIL is a pointer to the top element of the availability stack; NEW & SAVE are temporary pointer variables. This function inserts a new node such that linked list preserves the ordering of the terms in increasing order of their INFO field. This function returns address of FIRST node.

#### 1 [Underflow?]

IF AVAIL = NULL

Then Write ("Availability Stack Underflow")

Return(FIRST)

#### 2 [Obtain address of next free Node]

NEW AVAIL

**3 [Remove free node from Availability Stack]**

AVAIL LINK(AVAIL)

**4. [Is the list is empty]**

If FIRST = NULL

then LINK (NEW) NULL

Return (NEW)

**5. [Does the new node precede all other node in the list?]**

If INFO(NEW)  $\leq$  INFO (FIRST)

then LINK (NEW) FIRST

Return (NEW)

**6. [Initialize temporary pointer]**

SAVE FIRST

**7. [Search for predecessor of new node]**

Repeat while LINK (SAVE)  $\neq$  NULL and INFO (NEW)  $\geq$  INFO (LINK (SAVE))

SAVE LINK (SAVE)

**8. [Set link field of NEW node and its predecessor]**

LINK (NEW) LINK (SAVE)

LINK (SAVE) NEW

**9. [Return first node pointer]**

Return (FIRST)

**When INSERTORD is invoked it returns a pointer value to the variable FIRST**

**FIRST INSERTORD (X, FIRST)**

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **10**



## Linked List

By repeatedly involving function INSORD; we can easily obtains an ordered liner list for example the sequence of statements.

FRONT NULL

FRONT INSORD (29, FRONT)

FRONT INSORD (10, FRONT)

FRONT INSORD (25, FRONT)

FRONT INSORD (40, FRONT)

FRONT INSORD (37, FRONT)

FRONT FRONT FRONT FRONT FRONT

10 25 29 40

29

10 25 29 37 40 Trace of construction of an

10 29

ordered linked linear list using function

10 25 29

INSORD

### Algorithm to delete a node from Linked List

• Algorithm that deletes node from a linked linear list:-

1. If a linked list is empty, then write under flow and return.
2. Repeat step 3 while end of the list has not been reached and the node has not been found.
3. Obtain the next node in list and record its predecessor node.
4. If the end of the list has been reached then write node not found and return.
5. Delete the node from list.
6. Return the node into availability area.

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **11**



## Linked List

### Procedure: DELETE( X, FIRST)

Given X, an address of node which we want to delete and FIRST is a pointer to the first element of a linked linear list. Typical node contains INFO and LINK fields. SAVE & PRED are temporary pointer variables.

#### 1. [Is Empty list?]

If FIRST = NULL

then write ('Underflow')

return

#### 2. [Initialize search for X]

SAVE FIRST

#### 3. [Find X]

Repeat thru step-5 while SAVE  $\neq$  X and LINK (SAVE)  $\neq$  NULL

#### 4. [Update predecessor marker]

PRED ← SAVE

#### 5. [Move to next node]

SAVE ← LINK (SAVE)

#### 6. [End of the list]

If SAVE ≠ X

then write ('Node not found')

return

#### 7. [Delete X]

If X = FIRST (if X is first node?)

then FIRST ← LINK (FIRST)

else LINK (PRED) ← LINK (X)

#### 8. [Free Deleted Node]

Free (X)

### Function: COPY (FIRST)

- FIRST is a pointer to the first node in the linked list, this function makes a copy of the list. • The new list is to contain nodes whose information and pointer fields are denoted by FIELD and PTR, respectively. The address of the first node in the newly created list is to be placed in BEGIN. NEW, SAVE and PRED are pointer variables.
- A general algorithm to copy a linked list
  1. If the list is empty then return null



## Linked List

2. If the availability stack is empty then write availability stack underflow and return else copy the first node.
3. Repeat thru step 5 while the old list has not been reached.
4. Obtain next node in old list and record its predecessor node.
5. If availability stack is empty then write availability stack underflow and return else copy the node and add it to the rear of new list.
6. Set link of the last node in the new list to null and return.

#### 1. [Is Empty List?]

If FIRST = NULL

then return (NULL)

#### 2. [Copy first node]

NEW ← NODE

```
New  AVAIL
AVAIL  LINK (AVAIL)
FIELD (NEW)  INFO (FIRST)
BEGIN  NEW
```

**3. [Initialize traversal]**

```
SAVE  FIRST
```

**4. [Move the next node if not at the end of list]**

```
Repeat thru step 6 while (SAVE) ≠ NULL
```

**5. [Update predecessor and save pointer]**

```
PRED  NEW
SAVE  LINK (SAVE)
```

**6. [Copy node]**

```
If AVAIL = NULL
then write ('Availability stack underflow')
    Return (0)
else NEW  AVAIL
    AVAIL  LINK (AVAIL)
    FIELD (NEW)  INFO (SAVE)
    PTR (PRED)  NEW
```

**7. [Set link of last node and return]**

```
PTR (NEW)  NULL
Return (BEGIN)
```



## Linked List

**5. Write following algorithms for circular link list**

- 1) Insert at First Position**
- 2) Insert at Last Position**
- 3) Insert in Ordered Linked List**
- 4) Delete Element**

**PROCEDURE: CIRCULAR\_LINK\_INSERT\_FIRST (X, FIRST, LAST)**

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. NEW is a temporary pointer variable. This procedure inserts value X at the

first position of Circular linked linear list.

**1. [Create New Empty Node]**

NEW  $\leftarrow$  NODE

**2. [Initialize fields of new node and its link to the list]**

INFO (NEW)  $\leftarrow$  X

If FIRST = NULL

then LINK (NEW)  $\leftarrow$  NEW

FIRST  $\leftarrow$  LAST  $\leftarrow$  NEW

else LINK (NEW)  $\leftarrow$  FIRST

LINK (LAST)  $\leftarrow$  NEW

FIRST  $\leftarrow$  NEW

Return



## Linked List

### **PROCEDURE: CIR\_LINK\_INSERT\_END (X, FIRST, LAST)**

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. NEW is a temporary pointer variable. This procedure inserts value X at the last position of Circular linked linear list.

**1. [Create New Empty Node]**

NEW  $\leftarrow$  NODE

**2. [Initialize fields of new node and its link to the list]**

If FIRST = NULL

then LINK (NEW)  $\leftarrow$  NEW

```

        FIRST  LAST  NEW
else LINK(NEW)  FIRST
        LINK(LAST)  NEW
        LAST  NEW
Return

```



## Linked List

### **PROCEDURE: CIR\_LINK\_INSERT\_ORDER (X, FIRST, LAST)**

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. NEW is a temporary pointer variable. This procedure inserts value X such that linked list preserves the ordering of the terms in increasing order of their INFO field.

#### **1. [Create New Empty Node]**

NEW  $\leftarrow$  NODE

#### **2. [Copy information content into new node]**

INFO (NEW)  $\leftarrow$  X

**3. [Is Linked List is empty?]**

```
If FIRST = NULL
then LINK (NEW)  NEW
    FIRST  LAST  NEW
Return
```

**4. [Does new node precedes all other nodes in List?]**

```
If INFO (NEW) ≤ INFO (FIRST)
then LINK (NEW)  FIRST
    LINK (LAST)  NEW
    FIRST  NEW
Return
```

**5. [Initialize Temporary Pointer]**

```
SAVE  FIRST
```

**6. [Search for Predecessor of new node]**

```
Repeat while SAVE ≠ LAST and INFO(NEW) ≥ INFO(LINK(SAVE))
    SAVE  LINK(SAVE)
```

**7. [Set link field of NEW node and its Predecessor]**

```
LINK(NEW)  LINK(SAVE)
LINK(SAVE)  NEW
If SAVE = LAST
then LAST  NEW
```

**8. [Finish]**

```
Return
```



## Linked List

### PROCEDURE: CIR\_LINK\_DELETE (X, FIRST, LAST)

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. SAVE & PRED are temporary pointer variables. This procedure deletes a node whose address is given by pointer variable X.

**1. [Is Empty List?]**

```
If FIRST = NULL
then write ('Linked List is Empty')
```



Return

**2. [Initialize Search for X]**

SAVE FIRST

**3. [Find X]**

Repeat thru step 5 while  $\text{SAVE} \neq X$  and  $\text{SAVE} \neq \text{LAST}$

**4. [Update predecessor marker]**

PRED SAVE

**5. [Move to next node]**

SAVE LINK (SAVE)

**6. [End of Linked List]**

If  $\text{SAVE} \neq X$

then write('Node not found')

return

**7. [Delete X]**

If  $X = \text{FIRST}$

then  $\text{FIRST} \leftarrow \text{LINK}(\text{FIRST})$

$\text{LINK}(\text{LAST}) \leftarrow \text{FIRST}$

else  $\text{LINK}(\text{PRED}) \leftarrow \text{LINK}(X)$

If  $X = \text{LAST}$

then  $\text{LAST} \leftarrow \text{PRED}$

**8. [Free Deleted Node]**

Free (X)



## Linked List

**6. Write an algorithm to perform each of the following operations on Circular singly linked list using header node**

**1) add node at beginning**

**2) add node at the end**

**3) insert a node containing x after node having address P**

**4) delete a node which contain element x**

### **FUNCTION: CIR\_LINK\_HEAD\_INSERT\_FIRST (X, FIRST, LAST)**

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. NEW is a temporary pointer variable. HEAD is the address of HEAD node. This procedure inserts value X at the first position of Circular linked linear list.

**1. [Create New Empty Node]**

NEW  $\leftarrow$  NODE

**2. [Initialize fields of new node and its link to the list]**

INFO (NEW)  $\leftarrow$  X

LINK (NEW)  $\leftarrow$  LINK (HEAD)

LINK (HEAD)  $\leftarrow$  NEW

### **FUNCTION: CIR\_LINK\_HEAD\_INSERT\_LAST (X, FIRST, LAST)**

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. NEW is a temporary points variable. HEAD is the address of HEAD node. This procedure inserts value X at the last position of Circular linked linear list.

**1. [Create New Empty Node]**

NEW  $\leftarrow$  NODE

**2. [Initialize fields of new node and its link to the list]**

INFO (NEW)  $\leftarrow$  X

LINK (NEW)  $\leftarrow$  HEAD

LINK (LAST)  $\leftarrow$  NEW

LAST  $\leftarrow$  NEW



## Linked List

**FUNCTION: CIR\_LINK\_HEAD\_INSERT\_AFTER\_Node-P (X, FIRST, LAST)** FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. NEW is a temporary pointer variable. HEAD is the address of HEAD node. This procedure insert a node after a node having address P.

**1. [Create New Empty Node]**

NEW  $\leftarrow$  NODE

**2. [Initialize fields of new node and its link to the list]**

INFO (NEW)  $\leftarrow$  X

LINK (NEW)  $\leftarrow$  LINK (P)

LINK (P)  $\leftarrow$  NEW

If P = LAST

then LAST  $\leftarrow$  NEW



**PROCEDURE: CIR\_LINK\_HEAD\_DELETE (X, FIRST, LAST)**

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. SAVE & PRED are temporary pointer variables. HEAD is the address of HEAD node. This procedure deletes a node whose value is X.

**1. [Is Empty List?]**

If FIRST = NULL  
then write ('Underflow')  
return

**2. [Initialize Search for X]**

SAVE ← FIRST

**3. [Find X]**

Repeat thru step 5 while INFO(SAVE) ≠ X and SAVE ≠ LAST

**4. [Update Predecessor]**

PRED ← SAVE

**5. [Move to next node]**

SAVE ← LINK(SAVE)

**6. [End of the List]**

If INFO (SAVE) ≠ X  
then write('Node not Found')  
return

**7. [Delete node X]**

If INFO (FIRST) = X  
then LINK (HEAD) ← LINK(FIRST)  
else LINK (PRED) ← LINK(SAVE)  
If SAVE = LAST  
then LAST ← PRED

**8. [Free Deleted Node]**

Free (X)



## Linked List

### 7. Write following algorithms for doubly link list

#### 1) Insert

## 2) Insert in Ordered Linked List

## 3) Delete Element

### PROCEDURE: DOUBINS (L, R, M, X)

Given a doubly link list whose left most and right most nodes addressed are given by the pointer variables L and R respectively. It is required to insert a node whose address is given by the pointer variable NEW. The left and right links of nodes are denoted by LPTR and RPTR respectively. The information field of a node is denoted by variable INFO. The name of an element of the list is NODE. The insertion is to be performed to the left of a specific node with its address given by the pointer variable M. The information to be entered in the node is contained in X.

#### 1. [Create New Empty Node]

NEW NODE

#### 2. [Copy information field]

INFO (NEW) X

#### 3. [Insert into an empty list]

If R = NULL

then LPTR (NEW) RPTR (NULL) NULL

L R NEW

Return

#### 4. [Is left most insertion ?]

If M = L

then LPTR (NEW) NULL

RPTR (NEW) M

LPTR (M) NEW

L NEW

Return

#### 5. [Insert in middle]

LPTR (NEW) LPTR (M)

RPTR (NEW) M

LPTR (M) NEW

RPTR (LPTR (NEW)) NEW

Return



Given a doubly link list whose left most and right most nodes addressed are given by the pointer variables L and R respectively. It is required to insert a node whose address is given by the pointer variable NEW. The left and right links of nodes are denoted by LPTR and RPTR respectively. The information field of a node is denoted by variable INFO. The name of an element of the list is NODE. The insertion is to be performed in ascending order of info part. The information to be entered in the node is contained in X.

**1. [Create New Empty Node]**

NEW NODE

**2. [ Copy information field]**

INFO (NEW) ← X

**3. [Insert into an empty list]**

```
If R = NULL
then LPTR (NEW) ← RPTR (NULL) ← NULL
      L ← R ← NEW
return
```

**4. [Does the new node precedes all other nodes in List? ]**

```
If INFO(NEW) ≤ INFO(L)
then RPTR (NEW) ← L
      LPTR(NEW) ← NULL
      LPTR (L) ← NEW
      L ← NEW
Return
```

**5. [ Initialize temporary Pointer]**

SAVE ← L

**6. [Search for predecessor of New node]**

```
Repeat while RPTR(SAVE) ≠ NULL and INFO(NEW) ≥ INFO(RPTR(SAVE))
      SAVE ← RPTR (SAVE)
```

**7. [Set link field of new node and its predecessor]**

```
RPTR (NEW) ← RPTR(SAVE)
LPTR (RPTR(SAVE)) ← NEW
RPTR (SAVE) ← NEW
LPTR (NEW) ← SAVE
```

```
If SAVE = R
then RPTR(SAVE) ← NEW
```



## PROCEDURE: DOUBDEL (L, R, OLD)

Given a doubly linked list with the addresses of left most and right most nodes are given by the pointer variables L and R respectively. It is required to delete the node whose address is contained in the variable OLD. Node contains left and right links with names LPTR and RPTR respectively.

### 1. [ Is underflow ?]

```
If R=NULL
then write (' UNDERFLOW')
return
```

### 2. [Delete node]

```
If L = R (single node in list)
then L = R = NULL
else If OLD = L (left most node)
then L = RPTR(L)
    LPTR (L) = NULL
    else if OLD = R (right most)
    then R = LPTR (R)
        RPTR (R) = NULL
        else RPTR (LPTR (OLD)) = RPTR (OLD)
            LPTR (RPTR (OLD)) = LPTR (OLD)
```

### 3. [ FREE deleted node]

```
FREE (OLD)
```



## Linked List

### 8. Write the implementation procedure of basic primitive operations of the stack using: (i) Linear array (ii) linked list.

**Implement PUSH and POP using Linear array**

```
#define MAXSIZE 100
int stack[MAXSIZE];
int top=-1;

void push(int val)
{
    if(top >= MAXSIZE)
        printf("Stack is Overflow");
    else
        stack[++top] = val;
}

int pop()
{
    int a;
    if(top>=0)
    {
        a=stack[top];
        top--;
        return a;
    }
    else
    {
        printf("Stack is Underflow, Stack is empty, nothing to
        POP!"); return -1;
    }
}
```





## Linked List

### Implement PUSH and POP using Linked List

```
#include<stdio.h>
#include<malloc.h>

struct node
{
    int info;
    struct node *link;
} *top;

void push(int val)
{
    struct node *p;
    p = (struct node*)malloc(sizeof(struct node));
    p  info = val;
    p  link = top;
    top = p;
    return;
}

int pop()
{
    int val;
    if(top!=NULL)
    {
        val = top  info;
        top=top  link;
        return val;
    }
    else
    {
        printf("Stack Underflow");
        return -1;
    }
}
```

```
}  
}
```

## 9. Write the implementation procedure of basic primitive operations of the Queue using: (i) Linear array (ii) linked list

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **25**



### Linked List

**Implement Enqueue(Insert) and Dequeue(Delete) using Linear Array**

```
# include <stdio.h>  
# define MAXSIZE 100  
int queue[MAXSIZE], front = -1, rear = -1;  
void enqueue(int val)  
{  
    if(rear >= MAXSIZE)  
    {  
        printf("Queue is overflow") ;  
        return ;  
    }  
    rear++;  
    queue [rear] = val;  
    if(front == -1)  
    {  
        front++;  
    }  
}  
int dequeue()  
{  
    int data;  
    if(front == -1)  
    {  
        printf("Queue is underflow") ;  
        return -1;  
    }  
    data = queue [front];
```

```

        if(front == rear)
        {
            front = rear = -1;
        }
        else
        {
            front++;
        }
        return data;
    }
}

```



## Linked List

### Implement Enqueue(Insert) and Dequeue(Delete) using Linked List

```

#include<stdio.h>
#include<malloc.h>

struct node
{
    int info;
    struct node *link;
} *front, *rear;

void enqueue(int val)
{
    struct node *p;
    p = (struct node*)malloc(sizeof(struct node));
    p->info = val;
    p->link = NULL;
    if (rear == NULL || front == NULL)
    {
        front = p;
    }
    else
    {
        rear->link = p;
        rear = p;
    }
}

int dequeue()

```

```

{
    struct node *p;
    int val;
    if (front == NULL || rear == NULL)
    {
        printf("Under Flow");
        exit(0);
    }
    else
    {
        p = front;
        val = p->info;
        front = front->link;
        free(p);
    }
    return (val);
}

```



## Linked List

**10. Write an algorithm to implement ascending priority queue using singular linear linked list which has insert() function such that queue remains ordered list. Also implement remove() function**

```

struct node
{
    int priority;
    int info;
    struct node *link;
}*front = NULL;

```

```

remove()
{
    struct node *tmp;
    if(front == NULL)
        printf("Queue Underflow\n");
    else
    {
        tmp = front;
        printf("Deleted item is %d\n", tmp->info);
        front = front->link;
        free(tmp);
    }
}

```

```

    }
}/*End of remove()*/

```

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **28**



## Linked List

```

insert()
{
    struct node *tmp,*q;
    int added_item,item_priority;
    tmp = (struct node *)malloc(sizeof(struct node));
    printf("Input the item value to be added in the queue :
    "); scanf("%d",&added_item);
    printf("Enter its priority : ");
    scanf("%d",&item_priority);
    tmp->info = added_item;
    tmp->priority = item_priority;
    /*Queue is empty or item to be added has priority more than
first item*/
    if( front == NULL || item_priority < front->priority
    ) {
        tmp->link = front;
        front = tmp;
    }
    else
    {
        q = front;
        while( q->link != NULL &&
            q->link->priority <= item_priority )

```

```

        {
            q=q->link;
        }
        tmp->link = q->link;
        q->link = tmp;
    }/*End of else*/
}/*End of insert()*/

```



## Linked List

```

display()
{
    struct node *ptr;
    ptr = front;
    if(front == NULL)
        printf("Queue is empty\n");
    else
    {
        printf("Queue is :\n");
        printf("Priority Item\n");
        while(ptr != NULL)
        {
            printf("%5d %5d\n",ptr->priority,ptr->info);
            ptr = ptr->link;
        }
    }/*End of else */
}/*End of display() */

```



## Nonlinear Data Structure (Graph & Tree)

### 1. Discuss following

#### 1. Graph

- A graph  $G$  consist of a non-empty set  $V$  called the set of nodes (points, vertices) of the graph, a set  $E$  which is the set of edges and a mapping from the set of edges  $E$  to a set of pairs of elements of  $V$ .
- It is also convenient to write a graph as  $G=(V,E)$ .
- Notice that definition of graph implies that to every edge of a graph  $G$ , we can associate a pair of nodes of the graph. If an edge  $x \in E$  is thus associated with a pair of nodes  $(u,v)$  where  $u, v \in V$  then we says that edge  $x$  connect  $u$  and  $v$ .

#### 2. Adjacent Nodes

- Any two nodes which are connected by an edge in a graph are called adjacent node.

#### 3. Directed & Undirected Edge

- In a graph  $G=(V,E)$  an edge which is directed from one end to another end is called a directed edge, while the edge which has no specific direction is called undirected edge.

#### **4. Directed graph (Digraph)**

- A graph in which every edge is directed is called directed graph or digraph.

#### **5. Undirected graph**

- A graph in which every edge is undirected is called undirected graph.

#### **6. Mixed Graph**

- If some of the edges are directed and some are undirected in graph then the graph is called mixed graph.

#### **7. Loop (Sling)**

- An edge of a graph which joins a node to itself is called a loop (sling).

#### **8. Parallel Edges**

- In some directed as well as undirected graphs, we may have certain pairs of nodes joined by more than one edges, such edges are called Parallel edges.

#### **9. Multigraph**

- Any graph which contains some parallel edges is called multigraph.

#### **10. Weighted Graph**

- A graph in which weights are assigned to every edge is called weighted graph.

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **1**



## **Nonlinear Data Structure (Graph & Tree)**

#### **11. Isolated Node**

- In a graph a node which is not adjacent to any other node is called isolated node.

#### **12. Null Graph**

- A graph containing only isolated nodes are called null graph. In other words set of edges in null graph is empty.

#### **13. Path of Graph**

- Let  $G=(V, E)$  be a simple digraph such that the terminal node of any edge in the sequence is the initial node of the edge, if any appearing next in the sequence defined as path of the graph.

#### **14. Length of Path**

- The number of edges appearing in the sequence of the path is called length of path.



## 15. Degree of vertex

- The no of edges which have V as their terminal node is call as indegree of node V • The no of edges which have V as their initial node is call as outdegree of node V • Sum of indegree and outdegree of node V is called its Total Degree or Degree of vertex.

## 16. Simple Path (Edge Simple)

- A path in a diagram in which the edges are distinct is called simple path or edge simple.

## 17. Elementary Path (Node Simple)

- A path in which all the nodes through which it traverses are distinct is called elementary path.

## 18. Cycle (Circuit)

- A path which originates and ends in the same node is called cycle (circuit).

## 19. Directed Tree

- A directed tree is an acyclic digraph which has one node called its root with in degree 0, while all other nodes have in degree 1.
- Every directed tree must have at least one node.
- An isolated node is also a directed tree.

## 20. Terminal Node (Leaf Node)

- In a directed tree, any node which has out degree 0 is called terminal node or leaf node.

## 21. Level of Node

- The level of any node is the length of its path from the root.

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure 2



## Nonlinear Data Structure (Graph & Tree)

## 22. Ordered Tree

- In a directed tree an ordering of the nodes at each level is prescribed then such a tree is called ordered tree.

## 23. Forest

- If we delete the root and its edges connecting the nodes at level 1, we obtain a set of disjoint tree. A set of disjoint tree is a forest.

## 24. M-ary Tree

- If in a directed tree the out degree of every node is less than or equal to m then tree is called an m-ary tree.

## 25. Full or Complete M-ary Tree

- If the out degree of each and every node is exactly equal to  $m$  or  $0$  and their number of nodes at level  $i$  is  $m^{(i-1)}$  then the tree is called a full or complete  $m$ -ary tree.

## 26. Positional M-ary Tree

- If we consider  $m$ -ary trees in which the  $m$  children of any node are assumed to have  $m$  distinct positions, if such positions are taken into account, then tree is called positional  $m$  ary tree.

## 27. Height of the tree

- The height of a tree is the length of the path from the root to the deepest node in the tree.

## 28. Binary tree

- If in a directed tree the out degree of every node is less than or equal to  $2$  then tree is called binary tree.

## 29. Strictly binary tree

- A strictly binary tree (sometimes proper binary tree or  $2$ -tree or full binary tree) is a tree in which every node other than the leaves has two children.

## 30. Complete binary tree

- If the out degree of each and every node is exactly equal to  $2$  or  $0$  and their number of nodes at level  $i$  is  $2^{(i-1)}$  then the tree is called a full or complete binary tree.

## 31. Sibling

- Siblings are nodes that share the same parent node.

## 32. Binary search tree

- A binary search tree is a binary tree in which each node possessed a key that satisfy the following conditions

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **3**



## Nonlinear Data Structure (Graph & Tree)

1. All key (if any) in the left sub tree of the root precedes the key in the root.
2. The key in the root precedes all key (if any) in the right sub tree.
3. The left and right sub tree sub trees of the root are again search trees.

## 33. Height Balanced Binary tree (AVL Tree)

- A tree is called AVL (height balance binary tree), if each node possesses one of the following properties
  1. A node is called left heavy if the longest path in its left sub tree is one longer then the longest path of its right sub tree.
  2. A node is called right heavy if the longest path in the right sub tree is one longer than path in its left sub tree.
  3. A node is called balanced, if the longest path in both the right and left sub tree are

equal.

## 2. Explain the Preorder, Inorder and Postorder traversal techniques of the binary tree with suitable example.

- The most common operations performed on tree structure is that of traversal. This is a procedure by which each node in the tree is processed exactly once in a systematic manner.
- There are three ways of traversing a binary tree.
  1. Preorder Traversal
  2. Inorder Traversal
  3. Postorder Traversal

**A**

**B**

**C** E Fig. 1.1

**D**

**G**

**F**

Preorder traversal : A B C D E F  
G

Inorder traversal : C B A E F D G

Postorder traversal : C B F E G

D A Converse Preorder

traversal : A D G E F B C

Converse Inorder traversal : G

D F E A B C Converse Postorder

traversal : G F E D C B A



## Nonlinear Data Structure (Graph & Tree)

### Preorder

- Preorder traversal of a binary tree is defined as follow
  - Process the root node
  - Traverse the left subtree in preorder
  - Traverse the right subtree in preorder
- If particular subtree is empty (i.e., node has no left or right descendant) the traversal is performed by doing nothing, In other words, a null subtree is considered to be fully traversed when it is encountered.
- The preorder traversal of a tree (Fig. 1.1) is given by A B C D E F G

## Inorder

- The Inorder traversal of a binary tree is given by following steps,
  - Traverse the left subtree in Inorder
  - Process the root node
  - Traverse the right subtree in Inorder
- The Inorder traversal of a tree (Fig. 1.1) is given by C B A E F D G

## Postorder

- The postorder traversal is given by
  - Traverse the left subtree in postorder
  - Traverse the right subtree in postorder
  - Process the root node
- The Postorder traversal of a tree (Fig. 1.1) is given by C B F E G D A

## Converse ...

- If we interchange left and right words in the preceding definitions, we obtain three new traversal orders which are called
  - Converse Preorder (A D G E F B C)
  - Converse Inorder (G D F E A B C)
  - Converse Postorder (G F E D C B A)

## 3. Write the algorithm of Preorder, Inorder and Postorder traversal techniques of the binary tree.

### Procedure : RPREORDER(T)

- Given a binary tree whose root node address is given by pointer variable T and whose node structure is same as described below. This procedure traverses the tree in preorder, in a recursive manner.

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure 5



## Nonlinear Data Structure (Graph & Tree)

LPTR DATA RPTR

### 1. [Check for empty Tree]

If T = NULL

then write ('Empty Tree')

return

else write (DATA(T))

2. **[Process the Left Subtree]**  
If LPTR (T)  $\neq$  NULL  
then RPREORDER (LPTR (T))
3. **[Process the Right Subtree]**  
If RPTR (T)  $\neq$  NULL  
then RPREORDER (RPTR (T))
4. **[Finished]**  
return

### **Procedure : RINORDER(T)**

- Given a binary tree whose root node address is given by pointer variable T and whose node structure is same as described below. This procedure traverses the tree in inorder, in a recursive manner.

1. **[Check for empty Tree]**  
If T = NULL  
then write ('Empty Tree')  
return
2. **[Process the Left Subtree]**  
If LPTR (T)  $\neq$  NULL  
then RINORDER (LPTR (T))
3. **[Process the root node]**  
write (DATA(T))
4. **[Process the Right Subtree]**  
If RPTR (T)  $\neq$  NULL  
then RINORDER (RPTR (T))
5. **[Finished]**  
return



## Nonlinear Data Structure (Graph & Tree)

### **Procedure : RPOSTORDER(T)**

- Given a binary tree whose root node address is given by pointer variable T and whose node structure is same as described below. This procedure traverses the tree in postorder, in a recursive manner.

1. **[Check for empty Tree]**  
If T = NULL

then write ('Empty Tree')

return

**2. [Process the Left Subtree]**

If LPTR (T)  $\neq$  NULL

then RPOSTORDER (LPTR (T))

**3. [Process the Right Subtree]**

If RPTR (T)  $\neq$  NULL

then RPOSTORDER (RPTR (T))

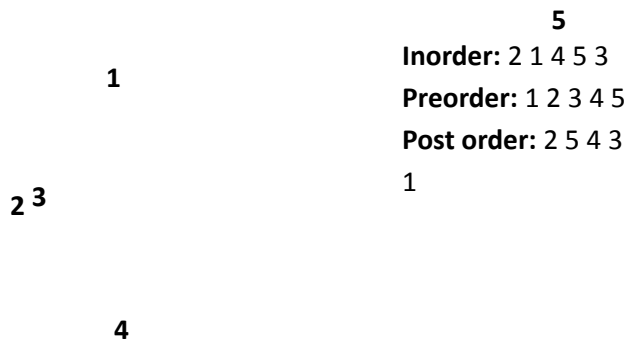
**4. [Process the root node]**

write (DATA(T))

**5. [Finished]**

return

**4. Give traversal order of following tree into Inorder, Preorder and Postorder.**



**5. Construct a tree for the given Inorder and Postorder traversals**

Inorder : D G B A H E I C F

Postorder : G D B H I E F C A

A

D G B H E I C F

A

B  
C

F  
D G H E I

A

B  
C

F

D  
E

I

G  
H



## Nonlinear Data Structure (Graph & Tree)

Postorder : C B E H G I F D A

Inorder : B C A E D G H F I

A

B C E D G H F I

A

B<sub>D</sub>

C

E G H F I

A

B<sub>D</sub>

E<sup>F</sup>

C

G H I

A

B<sub>D</sub>

E<sup>F</sup>

C

G I





## Nonlinear Data Structure (Graph & Tree)

### 6. Construct a tree for the given Inorder and Preorder traversals

Preorder : G B Q A C K F P D E R H

Inorder : Q B K C F A G P E D H R

G

Q B K C F A P E D H R

G

B<sub>P</sub>

Q<sup>K C F A</sup>

B<sub>P</sub>

D E R H

G

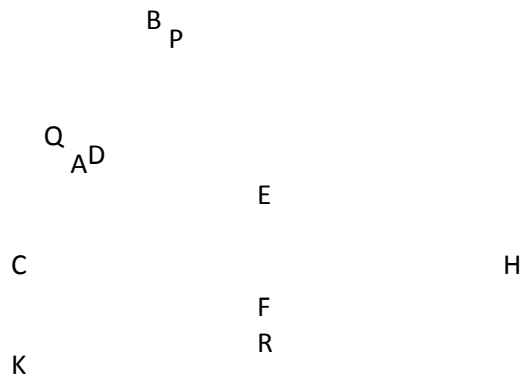
A<sup>D E</sup>

R H

Q

K C F

G



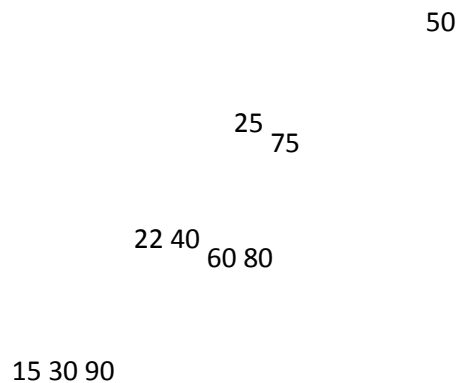
Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **10**



## Nonlinear Data Structure (Graph & Tree)

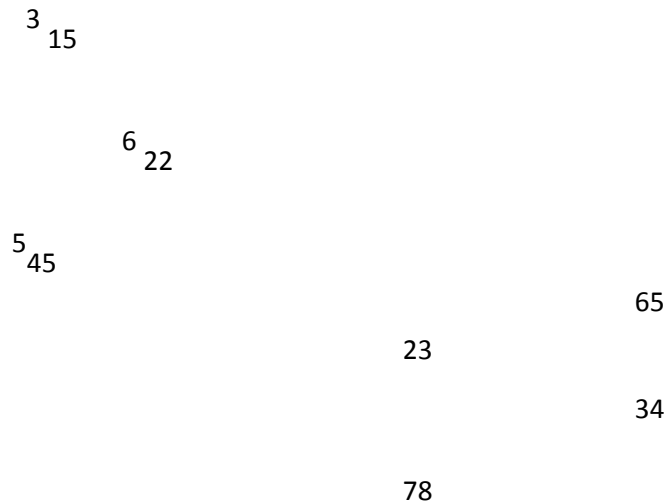
**7. Create a binary search tree for the following data :**

**50 ,25 ,75, 22,40,60,80,90,15,30**



**8. Construct binary search tree for the following data and find its Inorder, Preorder and Postorder traversal**

**10,3,15,22,6,45,65,23,78,34,5**



**Preorder** : 10, 3, 6, 5, 15, 22, 45, 23, 34, 65, 78

**Inorder** : 3, 5, 6, 10, 15, 22, 23, 34, 45, 65, 78

**Postorder** : 5, 6, 3, 34, 23, 78, 65, 45, 22, 15, 10

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **11**



## Nonlinear Data Structure (Graph & Tree)

### 9. Write a short note on threaded binary tree

- The wasted NULL links in the binary tree storage representation can be replaced by threads. • A binary tree is threaded according to particular traversal order. e.g.: Threads for the inorder traversals of tree are pointers to its higher nodes, for this traversal order.
  - If left link of node P is null, then this link is replaced by the address of its predecessor.
  - If right link of node P is null, then it is replaced by the address of its successor
- Because the left or right link of a node can denote either structural link or a thread, we must somehow be able to distinguish them.
- Method 1:- Represent thread a –ve address.
- Method 2:- To have a separate Boolean flag for each of left and right pointers, node structure for this is given below,

LPTR	LTHREAD	Data	RTHREAD	RPTR
------	---------	------	---------	------

Alternate node for threaded binary tree.

- LTHREAD = true = Denotes leaf thread link
- LTHREAD = false = Denotes leaf structural link

- RTHREAD = true = Denotes right threaded link
- RTHREAD = false = Denotes right structural link

- Head node is simply another node which serves as the predecessor and successor of first and last tree nodes. Tree is attached to the left branch of the head node

**Head**



### Advantages

- Inorder traversal is faster than unthreaded version as stack is not required.
- Effectively determines the predecessor and successor for inorder traversal, for unthreaded tree this task is more difficult.
- A stack is required to provide upward pointing information in tree which threading provides. • It is possible to generate successor or predecessor of any node without having over head of stack with the help of threading.

### Disadvantages

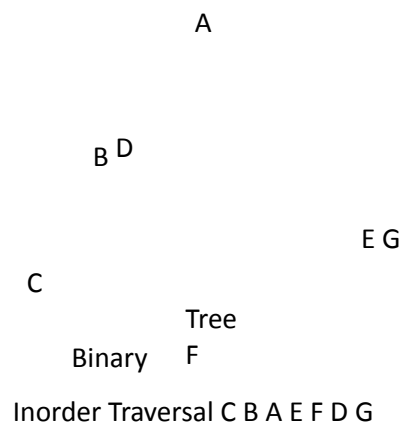
- Threaded trees are unable to share common subtrees

Prof. Pradyumansinh Jadeja (9879461848) | 2130702 – Data Structure **12**



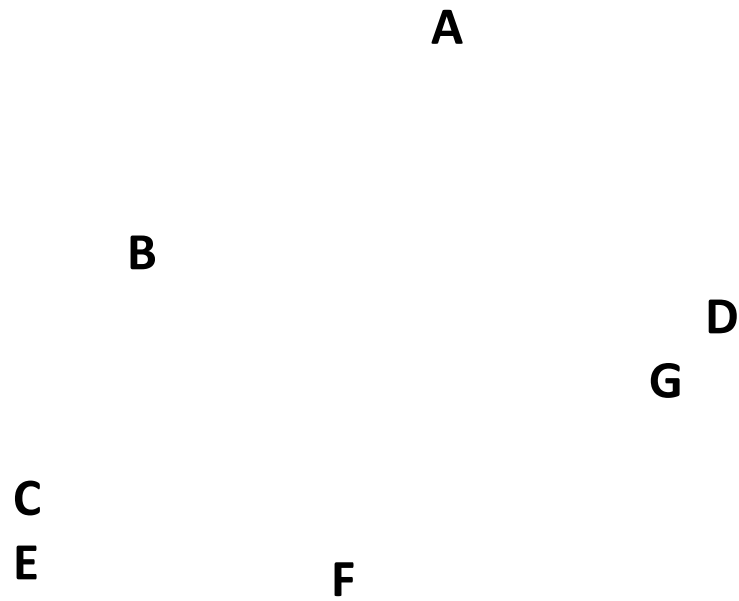
## Nonlinear Data Structure (Graph & Tree)

- If –ve addressing is not permitted in programming language, two additional fields are required. • Insertion into and deletion from threaded binary tree are more time consuming because both thread and structural link must be maintained.



### Fully In-threaded binary tree of given binary

tree HEAD



**10. Draw a right in threaded binary tree for the given tree**

### Right In-threaded binary tree of given binary tree

HEAD

A

H

B

D

E

C

G

F



## 11. What is the meaning of height balanced tree? How rebalancing is done in height balanced tree.

A tree is called AVL (height balance binary tree), if each node possesses one of the following properties

1. A node is called left heavy if the longest path in its left sub tree is one longer then the longest path of its right sub tree.
2. A node is called right heavy if the longest path in the right sub tree is one longer than path in its left sub tree.
3. A node is called balanced, if the longest path in both the right and left sub tree are equal.

If tree becomes unbalanced by inserting any node, then based on position of insertion, we need to rotate the unbalanced node. Rotation is the process to make tree balanced

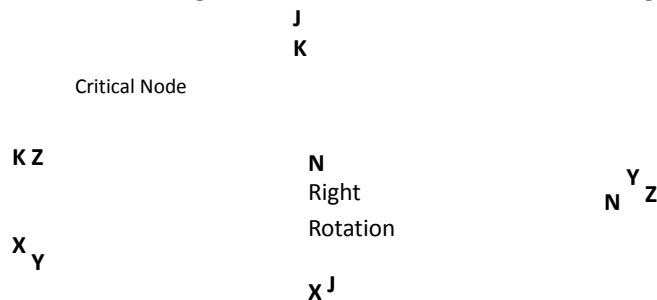
- 1) Insertion into Left sub-tree of nodes Left child – Single Right Rotation
- 2) Insertion into Right sub-tree of node's Left child – Left Right Rotation
- 3) Insertion into Left sub-tree of node's Right child – Right Left Rotation
- 4) Insertion into Right sub-tree of node's Right child – Single Left Rotation

### 1) Insertion into Left sub-tree of nodes Left child – Single Right Rotation

If node becomes unbalanced after insertion of new node at Left sub-tree of nodes Left child, then we need to perform Single Right Rotation for unbalanced node.

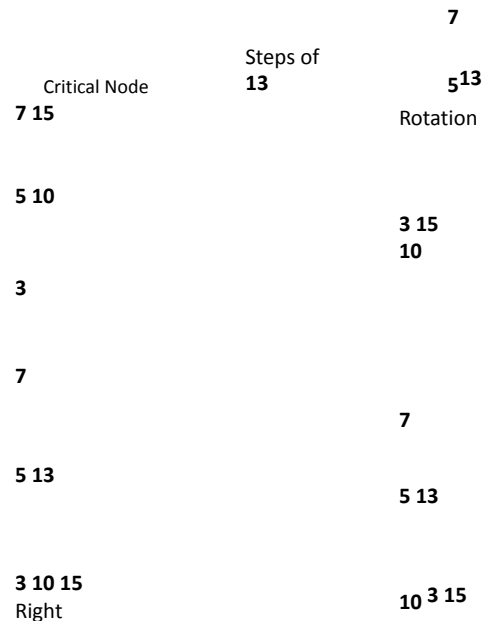
#### ***Right Rotation***

- a. Detach leaf child's right sub-tree
- b. Consider leaf child to be the new parent
- c. Attach old parent onto right of new parent
- d. Attach old leaf child's old right sub-tree as leaf sub-tree of new right child





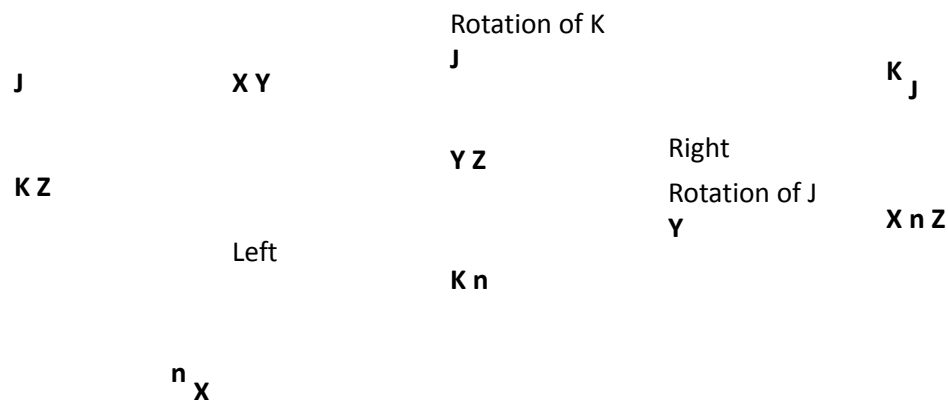
## Nonlinear Data Structure (Graph & Tree)



### 2) Insertion into Right sub-tree of node's Left child – Left Right Rotation

If node becomes unbalanced after insertion of new node at Right sub-tree of node's Left child, then we need to perform Left Right Rotation for unbalanced node.

Leaf rotation of leaf child followed by right rotation of parent



### 3) Insertion into Left sub-tree of node's Right child – Right Left Rotation

If node becomes unbalanced after insertion of new node at Left sub-tree of node's Right child, then we need to perform Right Left Rotation for unbalanced node.

Single right rotation of right child followed by left rotation of parent



