

**GOVERNMENT OF TAMILNADU
DIRECTORATE OF TECHNICAL EDUCATION
CHENNAI – 600 025**

STATE PROJECT COORDINATION UNIT

Diploma in Computer Engineering

Course Code: 1052

M – Scheme

e-TEXTBOOK

on

DATA STRUCTURES USING C

for

IV Semester Diploma in Computer Engineering

Convener for Computer Engineering Discipline:

A.Ghousia Jabeen,

Principal,

Thanthai Periyar E.V. Ramasamy Govt. Polytechnic College for Women,

Vellore – 636 002 600 113

Team Members for Data Structures Using C:

K. Selvamalathi

Lect(Sr. Gr.)/Computer Engg.

Ayyanadar Janaki Ammal Polytechnic College,
Sivakasi.

V. Banumathi

Lect. (Sr. Gr.)/Computer Engg.

ADJ Dharmambal Poly. College,
Nagapattinam.

S.A.Amudha,

Lecturer/ Computer Engg.

V.S.V.N. Polytechnic College,
Virudhunagar – 626001

Validated By

Mrs. L.Agnes Lavanya

Lecturer

Govt. Polytechnic College,
Cheyyar.

STATE BOARD OF TECHNICAL EDUCATION & TRAINING, TAMILNADU.

DIPLOMA IN COMPUTER ENGINEERING

M- SCHEME

(to be implemented to the student Admitted from the Year 2015-2016 on wards)

Course Name : Diploma in Computer Engineering.

Subject Code : 35244

Semester : IV

Subject title : DATA STRUCTURES USING C

TEACHING & SCHEME OF EXAMINATION:

No. of weeks per Semester 15 Weeks

Subject	Instructions		Examination			Duration
	Hours / Week	Hours / Semester	Internal Assessment	Board Examination	Total	
DATA STRUCTURES USING C	6	90	25	75	100	3 Hrs

TOPICS AND ALLOCATION OF HOURS

Unit No	Topic	No of Hours
I	INTRODUCTION TO DATA STRUCTURES , ARRAYS AND STRINGS AND ARRAYS	16
II	STACKS , RECURSION AND QUEUES	16
III	LINKED LISTS	16
IV	TREES AND GRAPHS	17
V	SEARCHING , SORTING AND HASHING	15
TEST AND REVISION		10
TOTAL		90

RATIONALE

Data structures are the techniques of designing the basic algorithms for real-life projects. In the present era, it is very essential to develop programs and organize data in such a way that it solves a complex problem efficiently. Understanding of data structures is essential and this facilitates to acquire sound knowledge of the insight of hardware requirement to any

problem base. The practice and assimilation of data structure techniques is essential for programming.

OBJECTIVES

- Define Linear and non-linear data structures.
- List and discuss the different types of linear data structures.
- Differentiate Stack and Queue
- Understand the Operations of Stack
- Explain the applications of stack
- Explain Linked lists and its implementation
- Define a tree and the different terms related with trees.
- Describe the different ways of traversing a binary tree.
- Discuss the various operations on Binary Search tree.
- Define graph terminologies and describe the different ways of traversing a graph.
- Write the algorithm for different types of sorting.
- Write the algorithm for different types of searching.
- Describe hash table and hash function.

DETAILED SYLLABUS

UNIT – I. INTRODUCTION TO DATA STRUCTURES , ARRAYS AND STRINGS 16 Hours		
1.1.	Introduction to Data Structures : Introduction - Data and Information - Elementary data structure organization - Types of data structures - Primitive and Non Primitive data structures – Operations on data structures : Traversing, Inserting, Deleting, Searching, Sorting, Merging - Different Approaches to designing an algorithm : Top-Down approach , Bottom-up approach - Complexity : Time complexity , Space complexity - Big 'O' Notation.	6 Hrs
1.2	ARRAYS: Introduction - Characteristics of Array - One Dimensional Array - Two Dimensional Arrays - Multi Dimensional Arrays – Advantages and Disadvantages of linear arrays - Row Major order - Column Major order - Operations on arrays with Algorithms (searching, traversing, inserting, deleting - Pointer and Arrays – Pointers and Two Dimensional Arrays - Array of Pointers - Pointers and Strings – Implementation of arrays -	7 Hrs
1.3	Strings : Strings and their representations - String Conversion- String manipulation, String arrays	3 Hrs
UNIT – II STACKS , RECURSION AND QUEUES ... 16 Hours		
2.1	Definition of a Stack - Operations on Stack (PUSH & POP)- Implementing Push and Pop Operations - Implementation of stack through arrays – Applications of Stack : Reversing a list - Polish notations - Conversion of infix to postfix expression	6 Hrs

	- Evaluation of postfix expression - Algorithm for evaluating Infix to prefix expression.	
2.2	Recursion - Recursive definition – Algorithm and C function for : Multiplication of Natural numbers - Factorial Function - GCD function - Properties of Recursive algorithms/functions – Advantages and Disadvantages of Recursion	4 Hrs
2.3	Queues: The queue and its sequential representation - implementation of Queues and their operations - implementation of Circular queues and their operations - Dequeue and Priority queues(Concepts only)	6 Hrs
UNIT – III LINKED LISTS	 16 Hours
3.1	Terminologies: Node, Address, Pointer, Information, Null Pointer, Empty list -. Type of lists : Singly linked list , Doubly linked list, Circular list - Representation of singly linked lists in Memory-Difference between Linked & sequential List – Advantages and Disadvantages of Linked list- Operations on a singly linked list (only algorithm) : Traversing a singly linked list , Searching a singly linked list , Inserting a new node in a singly linked list (front, middle, end), Deleting a node from a singly linked list (front, middle, rear) - Doubly linked list, Circular linked lists (Concepts only, no implementations)	16 Hrs
UNIT – IV TREES AND GRAPHS	 17 Hours
4.1	Trees: Terminologies: Degree of a node, degree of a tree, level of a node, leaf node, Depth / Height of a tree, In-degree & out-Degree, Path, Ancestor & descendant nodes-, siblings - Type of Trees : Binary tree - List representation of Tree - Binary tree traversal (only algorithm) : In order traversal , Preorder traversal , Post order traversal - Expression tree – Binary Search Tree – Creation of a Binary Search tree without duplicate node.	10 Hrs
4.2	Graphs : Introduction - Terminologies: graph, node (Vertices), arcs (edge), directed graph, in-degree, out-degree, adjacent, successor, predecessor, relation, weight, path, length - Representations of a graph - Adjacency Matrix Representation - Adjacency List Representation - Traversal of graphs : Depth-first search (DFS) , Breadth-first search (BFS) - Applications of Graph	7 Hrs
UNIT – V SORTING ,SEARCHING AND HASHING	 15 Hours
5.1	Sorting Techniques : Introduction – Algorithms and “ C” programs for : Selection sort , Insertion sort , Bubble sort – Algorithms only : Merge Sort ,Radix sort, Shell sort , Quick sort	6 Hrs
5.2	Searching : Introduction - Algorithms and “ C” programs for Linear search and Binary search	4 Hrs
5.3	Hashing : Hash tables – methods- Hash function - Collision resolution techniques	5 Hrs

TEXT BOOKS

Sl.No	TITLE	AUTHOR	PUBLISHER	Year of Publishing/Edition
1.	Data Structures	Seymour Lipschutz	Schaum's outlines, TMH Private Limited, New Delhi	Indian Adapted Edition 2006. 20 th Reprint 2011
2.	Data Structures with C	Seymour Lipschutz	Schaum's outlines, TMH Private	First Reprint 2011
3.	Data Structures A Programming approach with C	Dharmender Singh Kushwaha and Arun Kumar Misra	Prentice Hall of India, New Delhi	2012

REFERENCES

Sl.No	TITLE	AUTHOR	PUBLISHER	Year of Publishing/Edition
1.	Data Structures and Algorithms	G.A.Vijayalakshmi Pai	TMGH, New Delhi	6 th Reprint 2011
2.	Data Structures Using C - -1000 Problems and Solutions	Sudipta Mukherjee	TMGH, New Delhi	Second Reprint 2010
3.	Introduction to Data structures Using C	Venkatesh N.Baitipuli	University Science Press, Chennai	First Edition, 2009
4.	Classic Data Structures	Debasis Samanta	Prentice Hall of India, New Delhi	2009 / Second Edition
5.	Principles of Data structures using C and C++	Vinu V.Das	New Age International Publishers, New Delhi	Reprint 2008
6.	Data structures Using C	ISRD Group	TMGH, New Delhi	Ninth Reprint 2011
7.	Fundamentals of Data structures in C	Horowitz , Sahni Anderson- freed	University Press, Hyderabad	Second Edition
8.	Data and file structures	Rohit Khurana	Vikas Publishing Ltd	First Edition 2010

Unit 1: Introduction to Data structures, Arrays and Strings

Objectives

- To Know what is data and information
- To list the types of data structure
- To understand primitive and non-primitive data structure
- To perform operations on data
- To solve problems using top down and bottom up approach
- To understand time and space complexity
- To understand Big O Notation
- To represent and use arrays in programs
- To understand pointers and arrays and strings
- To represent strings
- To manipulate strings

1.1 Introduction to Data structures

Introduction

In our real life, we handle various types of data using computers. These data can be stored, retrieved and transformed to another form.

1.1.1 Definition of Data structures

Data structures refers to the organization of data in computer and gives the relationship among data.

1.1.2 Data

Data or Data item means a value or set of values. Eg.453, 16/03/1998, java, Chennai etc.

If the data or data item is divided into subitem, then that is called group item.
Eg.Address. Address is a group item and may be divided into doorno., street, place, pincode etc.

If the data could not be divided further, then that is called elementary data.
Eg.Regno.,Name, price etc.

Entity

An entity is a real thing and has set of properties or attributes having numeric or non numeric values. Eg. Entity named Employee may have the following attributes

Attribute	Name	Age	Sex	JoinDate	Department
Value:	John	27	Male	12/06/2013	Sales

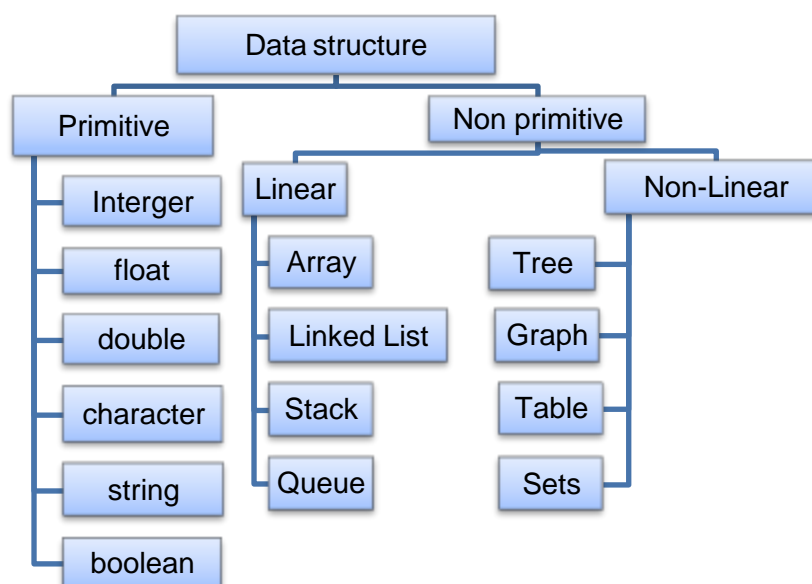
Information

Meaningful data or processed data is called Information

Eg. The data or value 27 has no meaning. But if we say, the age of employee is 27, then the data 27 has meaning and becomes information.

1.1.3 Classification of Data structures

Data structures are classified into various types based on their characteristics. The following figure represents the various classifications of data structures.



1.1.4 Primitive Data Structures

Primitive data structures are used to store standard data types of any computer language. These are used to represent single data value. They are classified into the following types.

Primitive Data Type	Example
Integer	23, 1000
Float	1.5, 3.14
Double	3.14
Character	'a'
String	"Balu"
Boolean	True, False

1.1.5 Non-Primitive Data structures

Non-primitive data structures are derived from primitive data structures. These are used to represent group of values. There are two types of non-primitive data structures. They are

- Linear data structure
- Non-Linear data structure

a. Linear Data structure

In linear data structures the data are stored in sequential order. The various linear data structures are

- Array - It means set of data of same data type and datas **are stored** in consecutive memory locations.
- Linked List - It means set of data of same data type and datas **are not stored** in consecutive memory locations.
- Stack - It means set of data of same data type and insertion and deletion is made at one end called **top**.
- Queue - It means set of data of same data type and insertion is made at **rear end** and deletion is made at **front end**.

b. Non-Linear Data structure

In non-linear data structure the data are not stored in sequential order. The various non-linear data structures are

- Graph - It is used to store data based on the relationship among pair of data.
- Tree - It is used to store data based on the hierarchical relationship among data.
- Table - It is used to store data in row and column order.
- Sets - It is used to store combined data.

1.1.6 Operations on Data Structures(Detailed explanation given in next chapter)

The following operations can be performed on Data Structures.

1. Traversing : It means accessing or visiting each element exactly once.
2. Inserting : It means adding a new element to the data structure.
3. Deleting : It means removing an element from the data structure.
4. Searching : It means finding the location of element with a given Key value.
5. Sorting : It means arranging the data in some logical order (ascending or descending or alphabetically).
6. Merging : It means combining the two data structure into a data structure.

1.1.7 Different Approaches to designing an algorithm

Algorithm means sequence of instructions given to computer to solve a problem. While designing an algorithm, the complex problem or system may be divided into smaller modules. Modularity improves design clarity, easy to implement, debug, testing, documenting and maintenance of the complex system.

There are two design approaches in design of algorithm. They are

1. Top Down approach
2. Bottom up approach

1.1.7.1 Top Down Approach

A top down design approach starts by

- i. identifying the major components of the system.
- ii. decompose them into lower level components
- iii. iterating until the desired level of module complexity is achieved

In each step, design is refined into most concrete level until we reach there is no more refinement.

Example : To create a program to build a simple calculator.

In this approach we look at the overall requirement of calculator and start coding. We build the main function first, then add calls to various functions like add, subtract etc... and then display the output. So what we did here was, approached the problem by first looking at the overall objective which is the top, then deciding when and where to call the functions and eventually going down to the lowest level detail of designing the functions themselves. Hence it is called the "top-down" approach.

1.1.7.2 Bottom Up Approach

In this approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed.

Example :All Software Projects development comes under this category

The first thing we do is, to ask ourselves "what are the modules we have to develop".We first design the modules. Then we decide how they need to be called and finally we design the main function which merely holds everything together in a proper sequence.

So what we did here was start designing from the lowest level which is the modules, then move up and finally encapsulate everything and use it to give the desired result. Hence it is called the Bottom-up approach.

1.1.8 Complexity of Algorithms

Complexity refers to the running time of an algorithm. After an algorithm has been designed, its efficiency or computational complexity is determined by means of CPU time (Time Complexity) and memory (Space Complexity).

1.1.8.1 Time Complexity

Time complexity of a program is the amount of time that is needed to run and complete execution. Consider the statement $a = a+1$ which is available in 3 different algorithm.

Algorithm	Statement	No. of times $a = a+1$ is executed
A	$a = a+1$	1
B	For $i = 1$ to n $a = a+1$	n
C	For $i = 1$ to n For $j = 1$ to n $a = a+1$	n^2

In general if an algorithm performs $f(n)$ basic operations and c is a constant(i.e. approximate clock time) then the total running time will be $c f(n)$

1.1.8.2 Space Complexity

The space needed by the program is the sum of the following components.

1. Fixed Space requirement: This includes instructions space, variable space and constants.

2. Variable Space requirement: This includes space needed by structured variables whose size depends on particular instance of variable. It also includes additional space like functions using recursion.

1.1.8 Big O notation

Big O notation helps to determine time as well as space complexity of the algorithm. This is useful to set the prerequisites of algorithm and helps to develop and design efficient algorithm in terms of time and space complexity.

Eg: Sorting can be done using methods like bubble sort, insertion sort, selection sort, heap sort, quick sort etc. Each algorithm is having its own complexity. Two of them are given below. $O(n)$ is pronounced as order of n .

Sort Method	Best case Complexity	Worst case Complexity
Bubble	$O(n)$	$O(n^2)$
Quick	$O(n \log n)$	$O(n^2)$

1. Best Case : The minimum possible value of $f(n)$
2. Average Case : The expected value of $f(n)$.
3. Worst Case : The maximum value of $f(n)$ for any possible input.

If $f(n)$ represents the computing time of some algorithm and $g(n)$ represents a known standard function like n , n^2 , $n \log n$ etc. then $f(n) \Rightarrow O(g(n))$ means that $f(n)$ of n is equal to biggest order of function $g(n)$.

1.2 ARRAYS

1.2.1 Introduction

A data structure is said to be linear if its elements form a sequence. Such linear sequence of data are to be stored in consecutive memory locations for easy retrieval. If data are stored in consecutive memory locations then it is called array.

Definition

Array means set of finite number of elements of same (or homogeneous) data type.

1.2.2 Characteristics of Array

- 1) An array holds elements of the same data type
- 2) Array elements are stored in subsequent memory locations
- 3) Two-dimensional array elements are stored row by row in subsequent memory locations.
- 4) Array name represents the address of the starting element
- 5) Array size should be mentioned in the declaration. Array size must be a constant and not a variable.
- 6) An array index starts from value zero

Terminology

- Base Address : This is the address of the memory where the first element of the array is located
- Index : The element of the array is referred by a subscript or index like $A[i]$. Here 'i' is the subscript or index
- Lower Bound : Lower bound of array is the starting position of array. Usually this is zero.
- Upper Bound : Upper Bound of array is the last position of array. Usually this is $n-1$
- Range or Range of Indices: The range is lower bound to upper bound
- Size of the Array : Size of the array = Upper Bound – Lower Bound + 1

Eg: `int A[100];`

Lower Bound = 0; Upper Bound = 99; Range = 0 to 99; Memory required to store 100 array elements = 100 integer data x 2 bytes = 200 bytes

1.2.3 One Dimensional Array

If only one subscript is needed to refer the element of an array, it is called one dimensional array. The general form to represent one dimensional array in C language is

`datatype arrayname[size];`

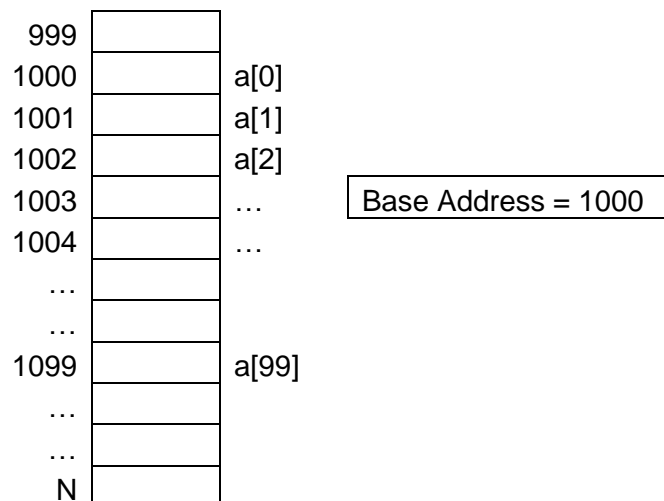
Example

`int a[100];`

During execution, the compiler allocates 100 consecutive memory locations and name the starting Memory location as **a**.

1.2.3.1 Memory Allocation for one dimensional array

Computer memory is a sequence of physical storage cells and each cell is identified by an address. Assume an array `a[100]` is to be stored in memory and the starting address (base address) = 1000 and each element takes one word, then the figure given below shows the memory allocation.



During execution, the compiler allocates 100 consecutive memory location and the starting address is named as 'a' ie the name of the array. The lower bound of array is 0 and upper bound of array is 99 and range = 100

1.2.3.2 Finding the address of 1D array element

The address of an element of 1D array can be calculated like below

Consider the array

`int a[100];`

During execution, 100 consecutive memory locations will be allocated in the memory.

Assume

$\text{base}(a) = \text{address of } a[0] = \&a[0];$

$\text{dsize} = \text{the size of byte to store one data (dsize} = 2 \text{ bytes for integer; 4bytes for float and 1byte for char)}$

The address of the first element = $\&a[0] = \text{base}(a)$.

The address of the second element = $\&a[1] = \text{base}(a) + 1 * \text{dsize}$

The address of the third element = $\&a[2] = \text{base}(a) + 2 * \text{dsize}$.

Likewise **address of i^{th} element = $\&a[i] = \text{base}(a) + i * \text{dsize}$**

1.2.4 Two Dimensional Array

Two dimensional arrays are set of homogeneous elements arranged in row and column order and needs two subscript to refer one data.

The general form to represent two dimensional array in C language is

`datatype arrayname[row size][columnsize];`

Example

`int a[5][3];` (5 rows and 3 columns)

During execution the compiler allocates 15 consecutive memory locations and name the starting memory location as a.

The size of row is called row range and size of column is called column range and maximum element in 2D array is row size * column size

1.2.4.1 Memory Allocation for two Dimensional Array

(or)

1.2.4.1 Row major and Column major representation of 2D Array

Consider a two dimensional array

`int a[2][3];`

2 Rows and 3 Columns

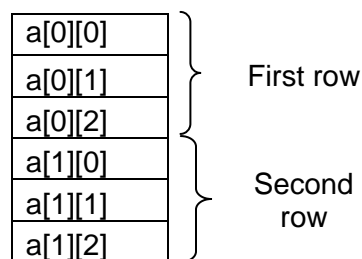
The array elements are given in matrix form

$$\begin{pmatrix} a[0][0] & a[0][1] & a[0][2] \\ a[1][0] & a[1][1] & a[1][2] \end{pmatrix}$$

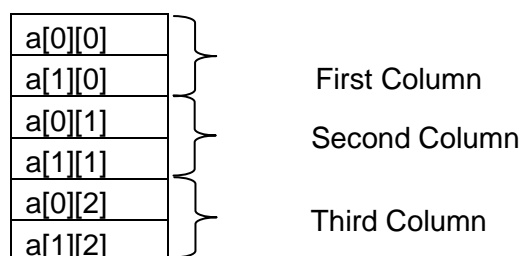
There are two ways to represent 2 D array in memory. They are

1. Row Major representation
2. Column Major representation

In **Row major order**, the elements of the array are stored on row by row basis, that is, all the elements in the first row first , then the second row second and so on as given below



In **Column major order**, the elements of the array are stored on column by column basis, that is, all the elements in the first column first , then the second column second and so on as given below.



1.2.4.2 Finding the address of 2D array Element

The address of an element in 2D array can be calculated like below

Consider the 2D array

10 rows & 5 columns

int a[10][5];

During execution 50 consecutive memory locations will be allocated in the memory. That is

$\text{base}(a) = \text{address of } a[0][0] = \&a[0][0];$

dsizes be the size of byte to store one data (dsizes = 2 bytes for integer; 4 bytes for float and 1 byte for char)

The address of the first element = $\&a[0][0] = \text{base}(a).$

The address of the second element = $\&a[0][1] = \text{base}(a) + 1 * \text{dsizes}$

The address of the third element = $\&a[0][2] = \text{base}(a) + 2 * \text{dsizes}.$

Likewise **address of element in i^{th} row and j^{th} column**

$$\&a[i][j] = \text{base}(a) + (i * \text{columnsize} + j) * \text{dsizes}$$

1.2.5 Multi Dimensional Arrays

In advanced applications, three or more dimensional arrays are used. Multi dimensional array is declared like below.

```
datatype arrayname[size1][size2][size3]...[size[n];
```

Example

```
int fees[branch][semester][name][amount];
```

The above example has four dimensions.

```
int tvsale[district ][branch][year][month][date];
```

The above example has five dimensions.

1.2.6 Advantages and Disadvantages of Linear Array

Advantages

- Easy to implement
- Random access is easier
- Suitable when the number of elements are predefined or already known.

Disadvantages:

- Inserting and deleting elements requires shifting of preceding and succeeding elements.
- Size of array is fixed, so the elements beyond the size cannot be added.
- Larger array may lead to high memory wastage, if we store only few elements in it.

1.2.7 Algorithms for Operations on Arrays (Program not necessary as per Syllabus)

1.2.7.1 Traversing

It is defined as the process of accessing or visiting all elements in an array exactly once. Generally arrays are visited to do the following

- i. to read an element.
- ii. to print an element.
- iii. to process an element.

The following steps are used to traverse an array.

Step 1: Initialize counter ie Set $I = LB$.

Step 2: Repeat steps 3 and 4 while $I \leq UB$

Step 3: Do the process to elements of A (read or print or calculate) at I

Step 4: Set $I = I + 1$.

Step 5: Exit.

Explanation:

Here, A is a linear array with upper bound UB and lower bound LB . This algorithm applies an operation on each element of linear array A . We are using a counter I which counts the number of elements in A . So, in the starting, the value of I is 0 since LB is also 0.

Operation	Example
read	for(i=0;i<n;i++) scanf("%d", &a[i]);
print	for(i=0; i<n; i++) printf("%d",a[i])
process	for(i=0; i<n; i++) Sum = Sum+a[i];

1.2.7.2 Inserting into Linear Array

Insertion means adding a new element into an array. The new element can be inserted at any one of the following positions of the array.

- i. at the end of the array.
- ii. at the middle of the array.

Assume

- Array name is A
- the position to **insert** is K
- upper bound is UB
- size of the array is n.

The following steps are used to insert an element into the array A.

- i. If $K = UB + 1$

store the new element at $A[K]$

- ii. Else check if $K \leq UB$

Shift the elements like below

$A[UB] \rightarrow A[UB+1]$

$A[UB-1] \rightarrow A[UB]$

.

.

$A[K] \rightarrow A[K+1]$

.

Store the new element at $A[K]$

- iii. Change the UB value = $UB + 1$ and $n = n + 1$

Example: Insert 65 at $A[3]$

As per algorithm $K = 3$. So

$A[5] \rightarrow A[6]$.

$A[4] \rightarrow A[5];$

$A[3] \rightarrow A[4];$

The following figure shows the memory representation before and after insert.

Before insert		
1000	23	A[0]
1001	56	A[1]
1002	89	A[2]
1003	10	A[3]
1004	55	A[4]
1005	20	A[5]
UB=5		

After shifting		
1000	23	A[0]
1001	56	A[1]
1002	89	A[2]
1003		A[3]
1004	10	A[4]
1005	55	A[5]
1006	20	A[6]
A3 position is free		

After insert		
1000	23	A[0]
1001	56	A[1]
1002	89	A[2]
1003	65	A[3]
1004	10	A[4]
1005	55	A[5]
1006	20	A[6]
UB=6		

1.2.7.3 Deleting from Linear Array

Deleting means removing an existing element from the array. The following steps are used to delete an element.

Assume

- Array name is A
- the position to **delete** is K
- upper bound is UB
- size of the array is n.

The following steps are used to delete an element from the array A.

i. Shift the elements like below

$A[K+1] \rightarrow A[K]$

$A[K+2] \rightarrow A[K+1]$

.

.

.

$A[UB] \rightarrow A[UB-1]$

ii. Change the UB value = UB-1 and n = n-1

Example: Delete 10 at A[3]

As per algorithm K = 3. So $A[4] \rightarrow A[3]$; $A[5] \rightarrow A[4]$. The figure given below shows the memory representation before and after delete.

Before Delete		
1000	23	A[0]
1001	56	A[1]
1002	89	A[2]
1003	10	A[3]
1004	55	A[4]
1005	20	A[5]
UB=5		

After Delete		
1000	23	A[0]
1001	56	A[1]
1002	89	A[2]
1003	55	A[3]
1004	20	A[4]
1005		
10 is removed and A[5] is free UB = 4		

1.2.7.4 Searching an element in Linear Array (Linear Search)

Searching means finding and locating an element in array. There are two types of search.

1. Linear Search
2. Binary Search

Linear Search :

Assume

- The array name is A
- The element to be searched is x
- The array size is n

The following steps are used to search an element.

- i. Compare x with A[0]. If equals print "Success" and print the position, stop the process.
Else Compare x with A[1]. If equals print "Success" and print the position, stop the process
Else repeat the process upto the last element i.e. A[n-1].
- ii. If no array value matches, print "Search Fail"

Example

Consider an array having 6 elements and search the element 25.

Given Array Data : 12 16 4 25 69 43

Search Data x : 25

Pass 1 : 12 16 4 25 69 43 : compare 12 with 25, no match

Pass 2 : 12 16 4 25 69 43 : compare 16 with 25, no match

Pass 2 : 12 16 4 25 69 43 : compare 4 with 25, no match

Pass 2 : 12 16 4 25 69 43 : compare 25 with 25, match.

So Print Success and position = 3

1.2.8 Pointers and Arrays

1.2.8.1 Pointers and One dimensional Arrays

Pointers are used to store the address of any variable. The & operator is used to store the address of variable. The pointer and array name is inter-related. Let us see how they are inter-related.

When an array is declared, the compiler allocates a base address and sufficient amount of storage to store all array elements. . The base address is the location of first element of the array. The array name itself is a pointer to the first element. Suppose if an array x is declared like

int x[5] = {10,20,30,40,50};

and base address of x is 1000, then 5 elements are stored like below.

Elements	x[0]	x[1]	x[2]	x[3]	x[4]
Value	10	20	30	40	50
Address	1000	1002	1004	1006	1008



Base Address

Also $x = \&x[0] = 1000$. If we use a pointer variable 'p' to store the address, then we can write like below.

p = x; or p = &x[0] ;

We can access every value of array x by using p. The relationship between p and x is given below.

p	=	&x[0]	=	1000
p+1	=	&x[1]	=	1002
p+2	=	&x[2]	=	1004
p+3	=	&x[3]	=	1006
p+4	=	&x[4]	=	1008

In general Address of n^{th} element = Base address + (n * scale factor of datatype)

1.2.8.2 Pointers and two dimensional array

In two dimensional array, the array name represents the address of zeroth row and zeroth column element.

Example

Consider a two dimensional array x with 3 rows and 4 columns. The array initialization and pointer p to array x is written using C statement as below

```
int x[3][4] = {{10,20,30,40}, {50,60,70,80}, {90,100,110,120}};
int *p;
p = x;      //(or p = &x[0][0]; )
```

The matrix representation of array x is given below.

		Column			
		0	1	2	3
Row	0	10	20	30	40
	1	50	60	70	80
	2	90	100	110	120

p
p+1
p+2

Here

- p points to first row
- p+i points to i^{th} row
- *(p+i) points to element in i^{th} row
- (*p+i)+j points to j^{th} element in i^{th} row
- *(*p+i+j) gives value stored in i^{th} row and j^{th} column
-

1.2.9 Array of Pointers

Since a pointer is a variable , we can create an array of pointers just like we can create any array of any other type.

Example

`int *p[10];` → 10 Rows with any number of Columns of integer

`int *p[];` → any number of rows with any number of columns of integer

`char *name[5];` → 5 names of any length

A common use of an array of pointers is to create an array of strings.

1.2.10 Pointers and Strings

Consider the statement

```
char str[5] = "Good";
```

Here the compiler automatically inserts a null character '\0' at the end of the string. The above statement can also be written as

```
char *str = "Good" ;
```

But the difference between the above two statements is

1. In the first statement only five characters can be stored.
2. In the second statement any number of characters can be stored.

Also the statement

```
Char name[3][25];
```

declares that it is a table of strings containing 3 names and all names are of maximum 25 characters. So, 3x25 bytes = 75 bytes are allocated during compilation. It is shown in the following diagram.

0	1	2	3	4	5	6	7	8														23	24
A	N	D	H	R	A	\0																	
K	A	R	N	A	T	A	K	A	\0														
T	A	M	I	L		N	A	D	U	\0													

But the statement `char *name[3];` reserves as many bytes as required.

If the individual names are of not equal length like the table given below, this type of declaration is useful.

Example

```
Char *name[3] = {"ANDHRA", "KARNATAKA", "TAMIL NADU"};
```

will take only 28 bytes like below.

A	N	D	H	R	A	\0				
K	A	R	N	A	T	A	K	A	\0	
T	A	M	I	L		N	A	D	U	\0

- The advantage of pointer to string is memory space efficiency.
- The disadvantage of pointer to string is that, the string value should be initialized during declaration itself since scanf() function could not be used to read strings.

1.2.11 Implementation of arrays

The general form to represent one dimensional array in C language is

`datatype arrayname[size];`

Example

```
int a[100];
```

During execution the compiler allocates 100 consecutive memory locations and named the starting memory location as a.

The general form to represent two dimensional array in C language is

`datatype arrayname[row size][columnsize];`

Example

```
int a[5][3]; (5 rows and 3 columns)
```

During execution the compiler allocates 15 consecutive memory locations and named the starting memory location as a.

1.3 String

Introduction

Computers are used for processing non-numerical data called character data. The processing may include pattern matching, replacing text ,editing text etc. Computer terminology uses the term string for a sequence of characters rather than the term word or text.

1.3.1 String Definition

A string is defined as a set of characters enclosed in double quotes and ended with null character '\0'. Example : "Computer Engineering"

1.3.2 String Representation or Declaration or Implementation

Array of Characters are used to represent strings in computer memory . The general form to represent string is

`char stringname[size];`

When a string is stored in computer memory, a null character '\0' is automatically stored as last character to indicate the end of string. Hence one extra space is needed to store the null character.

Example

Consider the following statement

```
char str[9] = "Computer"
```

The figure given below shows the memory representation of the string "Computer"

C	o	m	p	u	t	e	r	\0
---	---	---	---	---	---	---	---	----

1.3.3 String Conversion

Situation may occur to convert string to other data type and other data type to string. In C Language, **stdlib.h** header file contains the following functions to convert string.

- `atoi()` – This function is used to convert a string data type to integer data type
- `atof()` – This function is used to convert string to a floating point value.
- `atol()` – This function is used to convert a string to a long integer value.

ctype.h header file contains the following functions for string conversion

- `tolower()` – This function converts all the alphabets in upper case to lower case
- `toupper()` – This function converts all the alphabets in lower case to upper case
- `toascii()` – This function converts a character to its ascii value (Ascii value of
a=97, b=98 A = 65, 0 = 48 etc)

Program Example

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <conio.h>
void main()
{
    char ch, str[3];
    int value;
    printf("Enter any alphabet character\n");
    scanf("%c",&ch);
    value = toascii(ch);
    printf(" Ascii value = %d\n",value);
    printf("Enter any string \n");
    scanf("%s",str);
    value = atoi(str);
    printf(" Integer value = %d\n",value);
    getch();
}
```

Output:

```
Enter any alphabet character
A
Ascii value = 65
Enter any string
25
Integer value = 25
```

1.3.4 String Manipulation

The library file **string.h** is used to perform string operations. The important string functions are given below.

- `strlen()` – It is used to find the length of a string
- `strcpy()` – It is used to copy one string to another
- `strcat()` – It is used to concatenate / join / append / combine two strings
- `strcmp()` – It is used to compare two strings
- `strlwr()` – It is used to convert all alphabets in the string to lowercase
- `strupr()` – It is used to convert all alphabets in the string to uppercase
- `strrev()` – It is used to reverse the string
- `strstr()` – It is used to find the position of first occurrence of given string (It is also called indexing)

strlen()

It is used to find the length of a given string. The syntax is given below

```
int n = strlen(string);
```

Example

```
n = strlen("computer");          n = 8
```

strcpy()

It is used to copy one string to another. The syntax is given below

```
strcpy(string1,string2);
```

Example

```
Assume string1 = "Computer"  string2 = "Engineering"  
strcpy(string1,string2); results in string1 = "Engineering"  
                                string2 = "Engineering"
```

strcat()

It is used to concatenate two strings. The syntax is given below

```
strcat(string1,string2);
```

Example

```
Assume string1 = "Computer"  string2 = "Engineering"  
strcat(string1,string2); results in string1 = "ComputerEngineering"  
                                string2 = "Engineering"
```

It is noted that, there is no space between two strings while concatenating. i.e. "ComputerEngineering" is correct and "Computer Engineering" is not correct.

strcmp()

It is used to compare two strings.

- This function returns a value 0 if two strings are equal.
- This function returns positive value i.e. >1 if first string is bigger than second string
- This function returns negative value i.e. < 1 if first string is lower than second string

The syntax is given below

```
strcmp(string1,string2);
```

Example

1. strcmp("Computer","Computer"); results in zero value
2. strcmp("Computer","Engineering"); results in value (-2)
since Ascii value of C = 67 & Ascii value of E = 69, so 67-69 = -2
3. strcmp("Balu","Arun"); results in value (1)
since Ascii value of B = 66 & Ascii value of A = 65, so 66-65 = 1

strrev()

It is used to reverse a string. The syntax is

```
string2 = strrev(string1);
```

Example

Assume string1 = "Computer"
string2 = strrev(string1); will give string2 = "retupmoC"

strlwr()

It is used to convert the alphabets in given string to lowercase. The syntax is

```
strlwr(string);
```

Example

Assume string1 = "Computer"
strlwr(string1) results in string1 = "computer"

strupr()

It is used to convert the alphabets in string to uppercase. The syntax is

```
strupr(string);
```

Example

Assume string1 = "Computer"
strupr(string1) results in string1 = "COMPUTER"

strstr()

It is used to find the position of any substring in a given string. The syntax is

```
int pos = strstr(string1,string2);
```

Example

Assume string1 = "Computer" string2 = "put"
int pos = strstr(string1,string2); gives the result pos = 3

Program to illustrate string functions.

```
#include <stdio.h>
#include <string.h>
void main ()
{
char str1[10] = "Hello";
char str2[10] = "World";
char str3[20];
int len ;

/* copy str1 into str3 */
strcpy(str3, str1);
printf("strcpy( str3, str1) : %s\n", str3 );

/* concatenates str1 and str2 */
strcat( str1, str2);
printf("strcat( str1, str2): %s\n", str1 );

/* total length of str1 after concatenation */
len = strlen(str1);
printf("strlen(str1) : %d\n", len );
}
```

When the above program is compiled and executed, it produces the following result.

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

1.3.5 String Array

Strings are actually one-dimensional array of characters terminated by a **null** character '\0'. The following declaration and initialization creates a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string needs one more character than the number of characters in the word "Hello."

```
char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization, then you can write the above statement as follows.

```
char str[] = "Hello";
```


Following is the memory representation of the above defined string in C

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

The C compiler automatically places the '\0' character at the end of the string when it initializes the array. Let us try to print a sample string.

```
#include<stdio.h>
void main ()
{
    char str[10]="Good Luck";
    printf("Greeting message: %s\n",str);
}
```

When the above code is compiled and executed, it produces the following result.

```
Greeting message: Good Luck
```

Summary

- ✓ Data or Data item means a value or set of values
- ✓ An entity is a real thing and has set of properties or attributes
- ✓ Meaningful data or processed data is called Information
- ✓ Data structures are classified into various types based on their characteristics
- ✓ Primitive data structures are used to store standard data types
- ✓ Non-primitive data structures are derived from primitive data structures
- ✓ In linear data structures the data are stored in sequential order
- ✓ In non-linear data structure the data are not stored in sequential order
- ✓ Traversing : It means accessing or visiting each record exactly once
- ✓ Inserting : It means adding a new record to the structure
- ✓ Deleting : It means removing a record from the structure
- ✓ Searching : It means finding the location of record with a given Key value
- ✓ Sorting : It means arranging the data in some logical order
- ✓ Merging : It means combining the two sorted files into a single sorted file
- ✓ In Top down approach, the complex problem may be divided into smaller modules.
- ✓ In Bottom up approach, the base elements of problem are first specified and are linked together to form larger subsystems
- ✓ Complexity refers to the running time of an algorithm

- ✓ Big O notation helps to determine time as well as space complexity of the algorithm
- ✓ Array means set of finite number of elements of same (or homogeneous) data type
- ✓ There are two ways to represent 2 D array in memory. They are
 1. Row Major representation
 2. Column Major representation
- ✓ Pointers are used to store the address of any variable
- ✓ The array name itself is a pointer to the first element
- ✓ The 2D array name represents the address of 0th row and 0th column element.
- ✓ A common use of an array of pointers is to create an array of strings
- ✓ The advantage of pointer to string is memory space saving.
- ✓ string is defined as a set of characters enclosed in double quotes and ended with null character '\0'
- ✓ Array of Characters are used to represent strings in computer memory
- ✓ **stdlib.h** header file contains functions to convert string
- ✓ The library file **string.h** is used to perform string operations

Review Questions

Part A (2 Marks)

1. What is data and information?
2. What is the difference between data and information?
3. Define entity.
4. What are the primitive data structures?
5. What are the non – primitive data structures?
6. What are the operations that can be performed on data structures?
7. What are the various approaches in designing an algorithm?
8. What is called top down approach?
9. What is called bottom up approach?
10. What is called time Complexity?
11. What is called Space Complexity?
12. What is the use of Big O Notation?
13. What is an array?
14. How will you represent one dimensional array?
15. How will you declare two dimensional array?
16. What is called string?
17. How will you declare a string?
18. List few operations that can be performed on strings.
19. What is called null character?
20. How will you read a string?
21. What is the disadvantage of pointer to string?

Part B (3 Marks)

1. What is called attribute? Give Example.
2. Classify Data Structures.
3. What are primitive and non – primitive data structures?
4. Define traversing. Give example.
5. Define sorting. Give example.
6. Define searching. Give example.
7. Write the algorithm of merging.
8. What is called inserting and deleting?
9. Give example for top down and bottom up approach.
10. Write a brief note on Big O Notation.
11. List the advantages and disadvantages of arrays.
12. What is called string conversion?
13. How pointers are advantageous in representing string?
14. List few operations that can be performed on array. Explain any one.
15. What is the use of strcmp() command?

Part C (5 or 10 Marks)

1. List and explain the types of data structures.
2. Explain traversing with algorithm .
3. Explain how will you insert and delete data in array.
4. Explain primitive and non – primitive data structure in detail.
5. Explain sorting with example.
6. Assume an array contains 8 data. Explain how will you search an element in that array.
7. Write the algorithm for merging and explain with example.
8. Explain how will you store 1D array in detail.
9. What are the two ways of representing array? Explain.
10. What is called row major and column major representation?
11. Explain top down and bottom up approach in detail.
12. Write a brief note on complexity.
13. What is called multidimensional array? Give examples.
14. Explain pointer and arrays.
15. Explain 'pointers and two dimensional array.
16. What is called array of pointers?
17. Explain how pointers are used in representing strings.
18. Explain how will you implement array.
19. List the use of string conversion functions.
20. Write a program to illustrate the use of string functions.
21. Explain string handling functions in detail.

Unit 2 STACKS , RECURSION AND QUEUES

OBJECTIVES

- To understand Stack and Queue
- To know about the Operations of Stack
- To know about the applications of stack
- To understand Linked lists and its implementation

2.1 DEFINITION OF A STACK

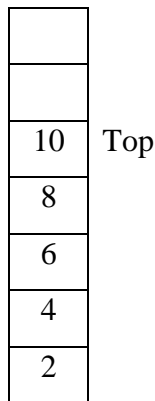
Stack is a linear data structure in which addition and deletions of items can be made at the top of the stack. There are two operations , which can be performed on stack.

- push → To add new element at the top of the stack.
- pop → To remove the element which is at the top of the stack.

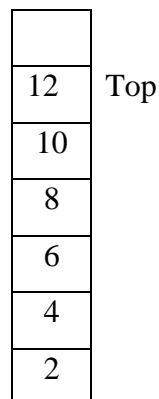
When push operation is performed the top moves upwards and when pop operation is performed the top moves downwards.

Consider the following stack which contains integers.

Here 10 is the current top element.



Add element 12 into stack. : **push (12).**



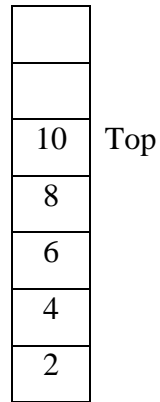
Now the top element is 12.

Remove an element from stack :



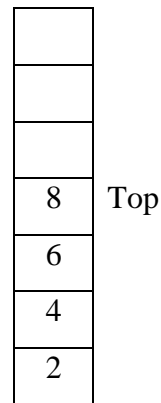
pop()

Poped or removed element=12



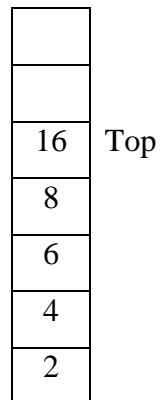
Pop ()

Poped or removed element=10



push(16)

Now the top element is 16.



Note:

The last item pushed onto a stack is always the first that will be popped from the stack. So stack is called last in first out (LIFO).

IMPLEMENTATION OF STACK AS AN ARRAY:

Declare an array large enough so that the stack can grow and shrink within the space allocated for it. During program execution, Fix one end of array as bottom of stack. The other end of array may be used as top of stack which keeps shifting constantly as

items are popped and pushed. The index of the array containing top element is stored in another field. Therefore declare stack as a structure containing two fields.

1. An array to hold elements of stack.
2. An integer to indicate the position of current top of the stack.

```
#define maxele 100
```

```
struct stack
{
    int top;
    int items[maxele];
};
struct stack s;
```

s – stack structure containing two members.

- i. An integer array items to store max. 100 integer element
- ii. An integer variable top to specify the index of last element. So

stack is empty when top = -1

stack is full when top = N-1

Position of Top	Status of Stack
-1	Stack is Empty
0	Only one element in Stack
N-1	Stack is Full
N	Overflow state of Stack

Operations on stack:

Push:- This operation is used to add a new item into the stack.

- i. Check whether the stack is full or not
if top = n-1 then stack is full, so exit.
- ii. If stack is not full increment the value of top by one (top = top+1)
- iii. Place the item on top position of the array as stack[top]=item

Procedure:

```
push(stack,top,n,item)
{
    if(top == n-1) //check whether the stack is full or not.
    {
        print "stack full";
        exit;
    }
    else
    {
        top = top+1 //increment the value of top by 1
        stack[top]= item //add the new item on the top of the stack.
    }
}
```

Pop:-

This operation is used to remove an item from the top of the stack.

- i. Check whether the stack is empty or not if top= -1 then stack is empty so
exit
- ii. If stack is not empty copy the top element into a variable item
item = stack[top]
- iii. Decrement the value of top by 1
top = top -1

Procedure:

```
pop(stack,top,item)
{
    if(top = -1) //check whether the stack is empty or not
    {
        print "Stack is Empty";
        exit;
    }
    else
    {
        item = stack[top]
        top = top-1
    }
}
```

Application of stack:

- i. For applications in which information must be saved and later retrieved in reverse order.
- ii. Stack is used in expression evaluation. For this the expression must be converted as postfix expression.

Reversing List : The list of elements can be reversed using stack. To reverse a string that do the following

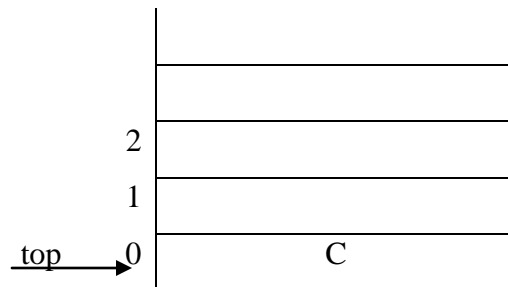
- i. Create an empty stack.
- ii. One by one push all characters of string to stack.
- iii. One by one pop all characters from stack and put them back to string.

Example:

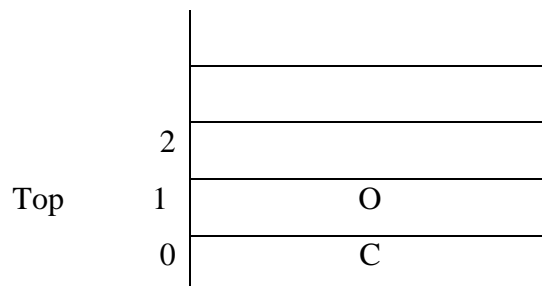
To reverse string **COMP**

Steps:

1. Initialize an empty character array named as result string.
2. Initialize a stack.
3. Scan the string character wise from left to right till the terminating character like \$ or # or '\0'
 - a. First character is 'C' push it into stack.



- b. Second character is 'O' push it into stack.



c. Third character Second is 'M' push it into stack.

TOP	2	M
	1	O
	0	C

d. Fourth character is 'P' push it into stack.

Top	3	P
	2	M
	1	O
	0	C

e. Fifth character is '\0' so pop up the characters one by one from the stack till end of stack and add to result string

resultstring = PMOC

	2	
	1	
	0	

Top → -1

Algorithm :

```

REVERSE(ST,RST)           // ST → Input String,   RST→Result string
{
    declare stack(n);      // Initialize stack.
    PE = null               // Initialize character array
    top = -1                // Now stack is empty.
    x=nexttoken(ST); // Extract a character from infix expression till x = NULL
    while(x!= NULL)

```

```

        {
            top = top + 1
            stack[top] = x ;    //push into stack
            x = nexttoken(ST);
        }
while(top ≠ -1)
{
    RST=RST+STACK[top];
    top=top-1;
}
}

```

Infix expression: If the operator in an expression are placed in between the operands then the expression is called infix expression.

Example:

- i. $A+B$
- ii $(A+B)/(C+D)$
- iii $(C*D)+(A+B)$

Polish notation:

If the operator in an expression are present before or after the operands, then that expression is said to be in polish notation. There are two types of polish notation.

They are

- a. postfix expression
- b. prefix expression

a. Postfix expression: If the operator in an expression are placed after operands then the expression is called postfix expression.

Example

- i. $AB+ \Rightarrow A+B$
- ii $AB+CD+ / \Rightarrow (A+B)/(C+D)$
- iii $CD* AB++ \Rightarrow (C*D)+(A+B)$

b. Prefix expression: If the operator in an expression are placed before the operands then the expression is called prefix expression.

Example

- i. $+AB \Rightarrow A+B$
- ii. $/+AB+CD \Rightarrow (A+B)/(C+D)$
- iii. $+*CD+AB \Rightarrow (C*D)+(A+B)$

Procedure: **Conversion of infix to postfix expression:**

- i. Add a dollar symbol at the end of the given infix expressions. initialize a stack and an array to hold characters
- ii. Scan the expression characterwise from left to right.
- iii. If the character is operand add it to character array.
- iv. If the character is operator or open parentheses, push it into the stack, while pushing check the priority of the current input operator and the topmost operator inside the stack(stack priority).
If the priority of the current operator is less than the top most operator in the stack, pop up the operator from the stack and add it to the character array until an operator having less priority than the input operator is reached and push the current operator into the stack.
- v. If the character is opening parenthesis push it into the stack.
- vi. If the character is closing parenthesis pop up the operator from the stack up to opening parenthesis and add it to the character array.
- vii. If the stack is not empty after the scanning is over, pop up the remaining operators from the stack till top = -1 is reached and add it to the character array.
- viii. The character array contains the postfix form of the given infix expression.

Example:

Convert the infix expression A into postfix expression.

$$A=(5+3)*3-2$$

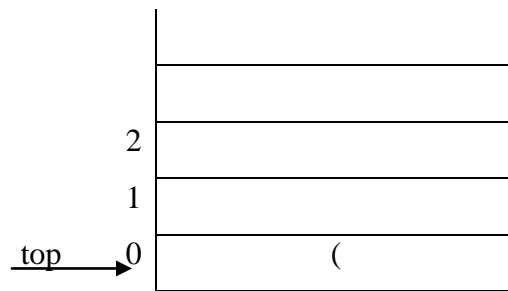
Steps:

1. Add dollar symbol to the end of given infix expression.

$$A=(5+3)*3-2\$$$

2. Initialize an empty character array named as postfix string.

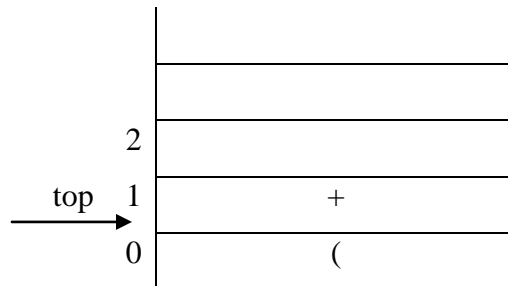
3. Initialize a stack.
4. Scan the expression character wise from left to right till \$ symbol.
 - a. First character is opening parenthesis, push it into stack.



- b. Second character is an operand 5 add it to the character array

Postfix_string = 5

- c. Third character is an operator '+', push it to the stack, check the priority, the input priority of + is greater than stack priority of (. So place + at the top.

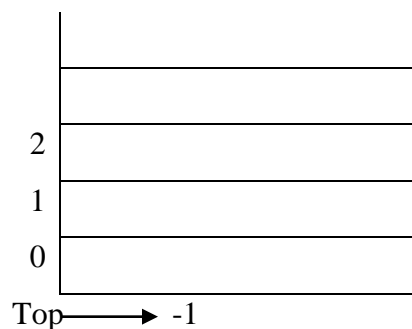


- d. Fourth character is operand 3 add it to the character array.

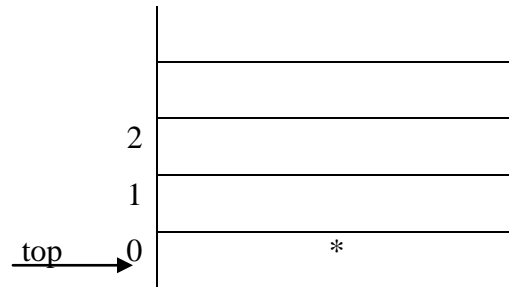
Postfix_string = 5 3

- e. Fifth character is closing parenthesis so pop up the operators from the stack till the opening parenthesis and add only the operators to the character array.

Postfix_string = 5 3 +



- f. Sixth character is an operator * so push it to the stack. Since the stack is empty no need to check for priority.

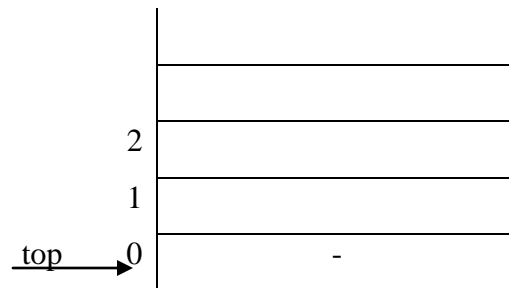


- g. Seventh character is an operand 3 Add it to the character array.

Postfix_string = 5 3 + 3

- h. Eighth character is an operator – check for input priority, its input priority is less than stack priority of top element * in the stack so pop up operator and add it to the character array and push the operator – into the stack.

postfix_string = 5 3 + 3 *



- i. Ninth character is an operand 2. So add it to the character array.

postfix_string = 5 3 + 3 * 2

- j. Tenth character is \$. so the scanning of character is over. But stack is not empty so pop up all the operators from the stack and add it to the character array.

postfix_string = 5 3 + 3 * 2 -

- k. The character array contains the postfix expression for the given infix expression.

```

polish(IE,PE)          // IE → Infix Expression and with $
                        PE →Postfix Expression

{
    declare stack(n);    // Intialise stack.
    PE = null            // Intialise character array
    top = -1             // Now stack is empty.
    x=nexttoken(IE); // Extract a character from infix expression till x ≠ $

while(x! = $)
{
    switch(x)
    {
        x is an operand:
        PE = PE + operand    //add operand to character array
        x = “(“ :
            top = top + 1
            stack[top] = “(“;    //push into stack
        x=”)”:
            while(stack[top]!= “(”)    // Remove operators from the
            {                          //stack and add it to the
                PE = PE + stack[top]; //character array
                top = top – 1;
            }
            top = top -1    // delete ‘(’ from the stack.
        x is an operator :
            Push the operator into the stack as per the priority
            If the input operator is of less priority pop the stack and add
            it to PE until the condition is satisfied
        }
        X = nexttoken(IE);
    }
    while(top>=-1)
    { PE=PE+stack[top]

```

```

        Top=top-1
    }
}

```

Evaluation of postfix expression:

- i. Initialize an empty array.
- ii. Add \$ symbol to the last of postfix expression .
- iii. Scan from left to right
 - a. If the character is operand push it into the stack.
 - b. If the character is an unary operator pop up one operand, perform the operation and push the result into the stack.
 - c. If the operator is binary operator pop up two operands and perform the operation and push the result into the stack.
- iv. Repeat step iii till the end of the expression. The result will be at the top of the stack
- v. End .

Example:

$A = 53+3*2-$

The infix form is

$$A = (5+3)*3-2$$

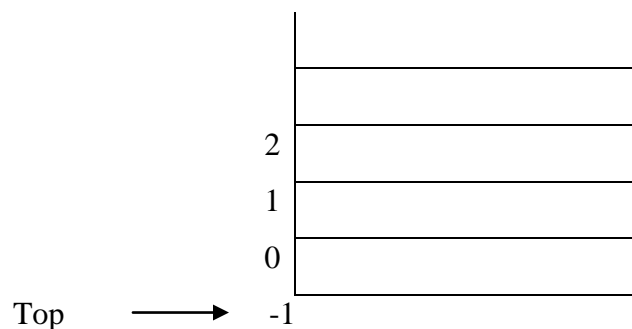
$$A = 8*3-2$$

$$A = 24-2$$

$$A = 22$$

Step:

- i. Initialize stack

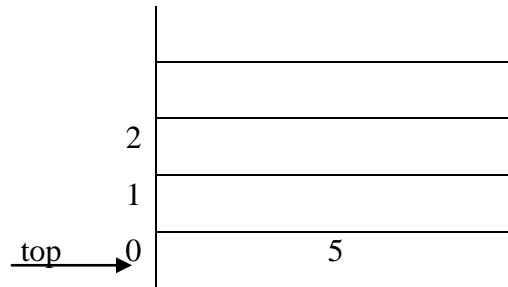


- ii. Add \$ symbol to post fix expression

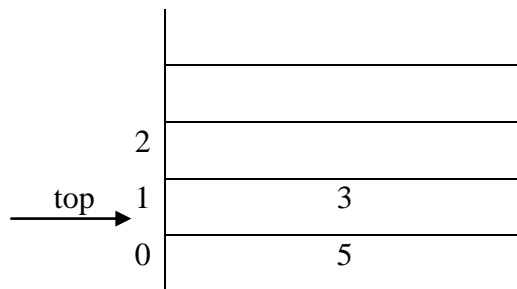
$A = 53+3*2-\$$

iii. Scan the expression from left to right character wise

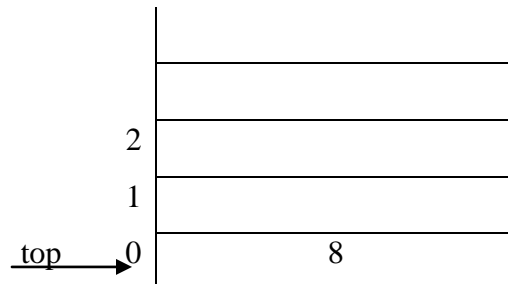
a. The first character is 5. push it into the stack.



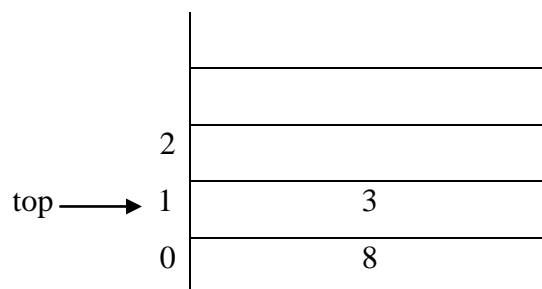
b. Second character is 3. Push it into the stack.



c. Third character is +. It is an binary operator. So pop up two operands 5 and 3 from the stack and do the operator $5+3$ and push the result 8 into stack.



d. Fourth character is 3. Push it into the stack.



e. Fifth character is *, It is an binary operator. So pop up two operands 8 and 3 from the stack and perform the operation $8 * 3$ and push the result into the stack.

2	
1	
0	24

f. Sixth character is 2. push it into the stack.

2	
1	2
0	24

g. The seventh character is $-$, It is an binary operator. So pop up two operands 24 and 20 from the stack and perform the operation $24-20$ and push the result into stack.

2	
1	
0	22

h. The ninth character is \$. So the scanning is over.

i. The result is stored at top of the stack. So $A = 22$.

Procedure :

```

Eval (PE, value) // PE is postfix expression end with $
{
    stack [ N ]    // Initialise stack
    x = next_token (PE);
    while ( x != $ )
    {
        switch (x )
        {
            x is an operand :
                top = top + 1;
                stack [ top ] = x;    // place the operand in the stack
            x is an operator :
                a = stack[top];    // pop second operand

```

```

        top = top -1;
        b = stack [top];          // pop first operand
    value = b operator a
    stack[top] = value;           // place result in stack
    }
    x = nexttoken (PE);
    }
    value = stack[top]
    }

```

2.2 RECURSION

DEFINITION:

Recursion is a process in which a function calls itself as a subroutine. This allows the function to be repeated several times, since it calls itself during its execution. Functions that incorporate *recursion* are called *recursive* functions.

Algorithm for Multiplication of Natural numbers

1. Input two numbers x,y
2. Check if any one number is zero then return zero
3. Else return x+call function(x,y-1)

C function for Multiplication of Natural numbers

```

int mulitplication(x,y)
{
    if (y==0 || x==0)
    {
        return 0;
    }
    else
        return x+multiplication(x,y-1);
}

```

Factorial Function

The product of the positive integers from 1 to n , inclusive, is called " n factorial" and is usually denoted by $n!$: $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n - 2) \cdot (n - 1) \cdot n$. It is also defined that $0! = 1$. Thus we have,

$$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120 \text{ and } 6! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 = 720$$

This is true for every positive integer n ; that is, $n! = n \cdot (n - 1)!$ Accordingly, the factorial function may also be defined as follows:

Definition: (Factorial Function)

- (a) If $n = 0$, then $n! = 1$.
- (b) If $n > 0$, then $n! = n \cdot (n-1)!$

Let us calculate $4!$ using the recursive definition. This calculation requires the following nine steps:

- (1) $4! = 4 \cdot 3!$
- (2) $3! = 3 \cdot 2!$
- (3) $2! = 2 \cdot 1!$
- (4) $1! = 1 \cdot 0!$
- (5) $0! = 1$
- (6) $1! = 1 \cdot 1 = 1$
- (7) $2! = 2 \cdot 1 = 2$
- (8) $3! = 3 \cdot 2 = 6$
- (9) $4! = 4 \cdot 6 = 24$

The following procedure calculates $n!$ and returns the value in the variable FACT.

Procedure FACTORIAL (FACT, N)

1. If $N = 0$ then: Set FACT: =1, and return.
2. Else Call FACTORIAL (FACT, N-1).
3. Set FACT := $N \cdot$ FACT.
4. Return.

The above procedure is a recursive procedure, since it contains a call statement to itself.

```
int fact(int n){
    if(n==1)
        return 1;
    else
        return(n*fact(n-1));
}
```

GCD FUNCTION

The GCD of two integers is the largest integer that can exactly divide both numbers (without a remainder).

GCD Example

Find the GCD of 45 and 54.

Step 1: Find the divisors of given numbers:

The divisors of 45 are : 1, 3, 5, **9**, 15, 45

The divisors of 54 are : 1, 2, 3, 6, **9**, 18, 27, 54

Step 2: Find the greatest number that these two lists share in common. In this example the GCD is 9.

```
int gcd(int n1, int n2)
{
    if (n2 != 0)
        return gcd(n2, n1%n2);
    else
        return n1;
}
```

Properties of Recursive Function/Algorithm

A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have –

- **Base criteria** – There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
- **Progressive approach** – The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

ADVANTAGES OF RECURSIVE FUNCTIONS:

- Avoidance of unnecessary calling of functions.
- A substitute for iteration where the iterative solution is very complex. For example to reduce the code size for Tower of Hanoi application, a recursive function is best suited.
- Extremely useful when applying the same solution

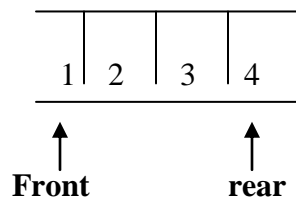
DISADVANTAGES OF RECURSIVE FUNCTIONS :

- A recursive function is often confusing.
- The exit point must be explicitly coded.
- It is difficult to trace the logic of the function.

2.3 QUEUE:

A queue is an ordered group of elements in which elements are added at one end known as the rear end and elements are removed from the other end known as front end.

The element which is stored first in the queue is removed as first element from the queue. Therefore queue is commonly known as FIFO or First In First Out.



Implementation of Queue:

A queue is a linear list of elements in which deletions can take place only at one end, called the front and insertions can take place only at the other end, called the rear. The terms front and rear are used in describing a linear list only when it is implemented as a queue. Queues are also called first in first out (FIFO) lists.

Representation

- Queues may be represented in computers by means of linear arrays or by one way list.

- We use two pointers FRONT and REAR. FRONT containing the location of front element of the queue and REAR containing the location of rear element of the queue.
- The condition FRONT =NULL will indicate that the queue is empty.
- Whenever an element is deleted from the queue, the value of FRONT is increased by 1. FRONT= FRONT+1
- Whenever an element is added to the queue, REAR is increased by 1. REAR=REAR+1
- Storing of an ITEM can be implemented as QUEUE[REAR]=ITEM
- Whenever queue is containing only one element then FRONT=REAR \neq NULL and suppose after this scenario element is deleted: FRONT=NULL and REAR=NULL which indicate that queue is empty.

The Queue can be represented using the array. There are two variables front and rear to indicate the positions of first and last element of the queue within the array.

The queue is empty when front = 0

Array Index

Front = 0, rear =0

↓

4	
3	
2	
1	

Add three elements (7,9,11) to the queue.

Front = 1 rear = 3

5	
4	
3	11
2	9
1	7

Remove two elements from Queue.

front =3,rear=3

5	
4	
3	11
2	
1	

Note : Since the element can be removed only through front end the elements 7,9 are removed and front is changed to array index 3.

Add one element 13 to the queue.

front =3, rear=4

5	
4	13
3	11
2	
1	

Basic operation:

- **Insert** means to insert element in the end of the queue. To insert an element we increment rear. Then insert an element at the position rear.
- **Delete** means to delete an element from the beginning of the queue. Get the element from the position front. Then increment front.

i. Insert:

Procedure

The following steps followed to add a new element into the queue

- If $\text{rear} = N$, queue is full. So addition is not possible.
- If $\text{rear} \neq N$ queue is not full . So addition is possible.
- Increment rear by one as $\text{rear} = \text{rear} + 1$
- Place the new item in $\text{queue}[\text{rear}] = \text{item}$

Program:

```
Insert(Q,rear,N,Item)
{
    if(rear=N) then
    {
        Printf("queue is full")
    }
    else
    {
        if(front=0) then
            front=rear=1
        else
            rear = rear+1
        Q[rear]= item
    }
}
```

ii.Delete:

Procedure:

The following steps followed to add a new element into the queue

- i. If front=0, queue is empty. So deletion is not possible.
- ii. If front \neq 0 queue is not empty . So deletion is possible.
- iii. Place the new item in item = queue[front]
- iv. Increment rear by one as front=front+1

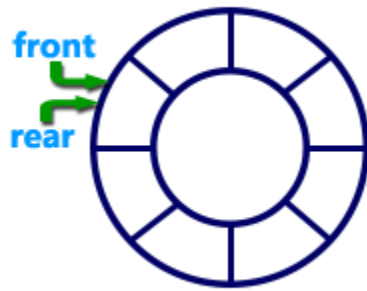
Program

```
Delete(Q,front,N,Item)
{
if( front=0) then
{
    Printf(“queue is empty”)
}
else
{
    item = queue[front]
    if (front=n-1) then
        front=rear=0
    else
        front=front+1
}
}
```

Circular Queue:

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

Circular queue is a queue in which the pointers rear and front varies in a circular manner. The initial condition for the circular queue is front=rear=0 or null.



Implementation of Circular Queue

To implement a circular queue data structure using array, we first perform the following steps before we implement actual operations.

- **Step 1:** Create a one dimensional array with above defined SIZE (**int cQueue[SIZE]**)
- **Step 2:** Define two integer variables '**front**' and '**rear**' and initialize both with '0'. (**int front = 0, rear = 0**)
- **Step 3:** Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

enQueue(value) - Inserting value into the Circular Queue

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

- **Step 1:** Check whether **queue** is **FULL**. ((**rear == SIZE-1 && front == 0**) || (**front == rear+1**))
- **Step 2:** If it is **FULL**, then display "**Queue is FULL!!! Insertion is not possible!!!**" and terminate the function.
- **Step 3:** If it is **NOT FULL**, then check **rear == SIZE - 1** if it is **TRUE**, then set **rear = 0**, else Increment **rear** value by one (**rear++**).

- **Step 4:** Set **queue[rear] = value** and check '**front == -1**' if it is **TRUE**, then set **front = 0**.

deQueue() - Deleting a value from the Circular Queue

In a circular queue, **deQueue()** is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from **front** position. The **deQueue()** function doesn't take any value as parameter. We can use the following steps to delete an element from the circular queue...

- **Step 1:** Check whether **queue** is **EMPTY**. (**front == -1 && rear == -1**)
- **Step 2:** If it is **EMPTY**, then display "**Queue is EMPTY!!! Deletion is not possible!!!**" and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then display **queue[front]** as deleted element and increment the **front** value by one (**front ++**). Then check whether **front == SIZE-1**, if it is **TRUE**, then set **front = 0**. Check whether both **front** and **rear** are equal (**front == rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front = rear = -1**).

display() - Displays the elements of a Circular Queue

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

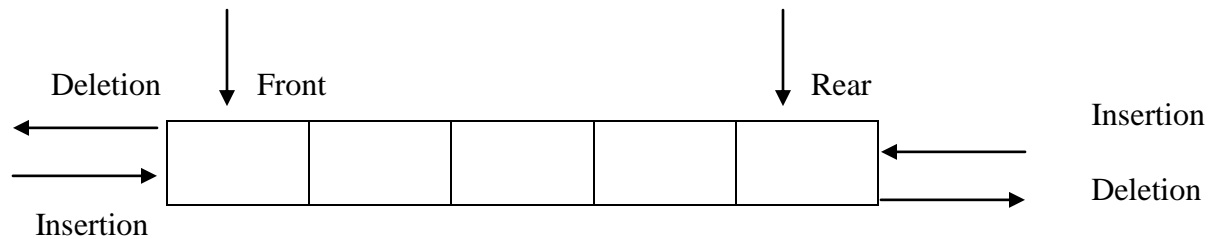
- **Step 1:** Check whether **queue** is **EMPTY**. (**front == -1**)
- **Step 2:** If it is **EMPTY**, then display "**Queue is EMPTY!!!**" and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i = front**'.
- **Step 4:** Check whether '**front <= rear**', if it is **TRUE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.
- **Step 5:** If '**front <= rear**' is **FALSE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i <= SIZE - 1**' becomes **FALSE**.
- **Step 6:** Set **i** to **0**.
- **Step 7:** Again display '**cQueue[i]**' value and increment **i** value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**

Deque:

A double ended queue is known as dequeue. In this the insertion and deletion can take place from both the sides. There are two types of dequeue. They are

1. Input restricted queue.
2. Output restricted queue.

The output restricted dequeue allows deletions from only one end and input restricted dequeue allows insertions at only one end.



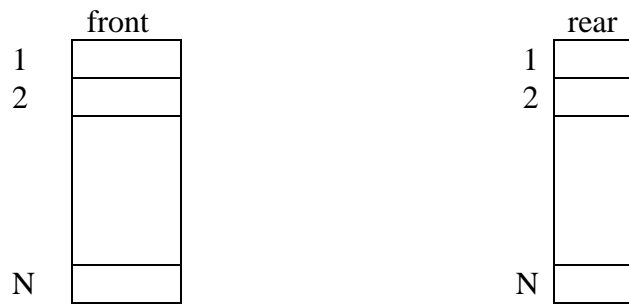
Priority Queues:

Priority queue is an ordered collection of elements are placed according to the defined priority. Insertions and deletions are done according to this priority.

Array Representation:

Let N be the number of priority levels assigned to the elements. Let M be the maximum number of elements with same priority. Then each priority level can be represented in memory as a separate circular queue. So we need N circular queues of size M . This can be represented as a 2D array named CQ of size $N \times M$. In this, each row represents a circular queue of size M . In addition to this we are in need 1-D array of size N to hold the front and rear pointers to the circular queues. The following figure shows the representation of priority queue in memory

	1	2		M-1	M
1					
2					
N					



Example:

Consider the following priority queue for getting the library book. This has 3 priority levels and each level contains maximum of 4 members.

Level1 - HOD
 Level2 - Staff
 Level3 - Student

The following members are registered in each level.

Hod - { Arvind,Shivani}
 Staff - { Ajay,Archutha,Archana}
 Student - { Arjith,Moni,Priya,Arun }

The following figure shows the array representation:

	1	2	3	4
1	Arvind	Shivani		
2	Ajay	Archutha	Archana	
3	Arjith	Moni	Priya	Arun

Front

1	1
2	1
3	1

rear

1	2
2	3
3	4

The level1 members are placed in the first row, level2 members are placed in the second row and level3 members are placed in the third row.

Review Question

PART A

1. Define Stack?
2. State the condition for empty stack in array implementation?
3. What is recursive function?
4. What is the property of recursive function?
5. Define queue.
6. State difference between queue and circular queue.

PART B

1. Write down the algorithm for push operation.
2. What is reverse polish notation? Explain with example.
3. Write down the algorithm to find factorial of given number
4. Write algorithm to find gcd of two natural numbers
5. Explain priority queue.

PART C

1. Explain array implementation of stack.
2. Explain algorithm to convert infix expression to postfix expression
3. Explain algorithm to evaluate postfix expression
4. Explain algorithm to do basic operation in queue
5. Explain algorithm to perform PUSH and POP operation

UNIT 3 LINKED LISTS

Objectives

- 3.1 To understand the basic terminologies of linked list.
- 3.1.1 To study the different types of linked list
- 3.1.2 To represent the linked list in memory
- 3.1.3 To study the differences between sequential list and Linked list
- 3.1.4 To study the advantages and disadvantages of linked list.
- 3.1.5 To understand the operations of linked list
- 3.1.6 To understand the concepts of doubly linked list
- 3.1.7 To understand the concepts of circular linked list

3.1 Terminologies:

Node:

Linked list is an ordered collection of elements. Each element is called a node. Each node contains two fields

- i. Data field
- ii. Link field

Data field contains the actual data and the link field contains the address of the next node(element).

Node structure

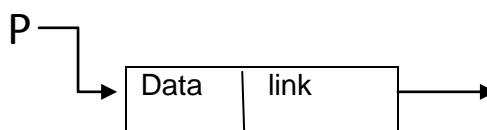


Fig 3.1

P=pointer

Address:

It is stored in the link field of the node(next node address), except last node. The address of the linked list are divided into:

- i. External address
- ii. Internal address
- iii. Null address

External address is the address of the first node in the list and is **assigned to a pointer variable**. With the help of this variable we can access the entire linked list.

Internal addresses are addresses stored in the link field of the nodes(next node address)except last node.

Null address is the address stored in the last node of the list and it indicates the end of the list.

Pointer

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location.**Linked list** is a linear collection of data elements, called nodes, each pointing to the next node by **means** of a pointer called link

Information

The actual data stored in the data field of the linked list.

Null Pointer

Null pointer is a special reserved value of a **pointer**. A **pointer** of any type has such a reserved value. Formally, each specific **pointer** type (int * , char * etc.) has its own dedicated **null-pointer** value. Conceptually, when a **pointer** has that **null** value it is not pointing anywhere.

Empty List

An empty list is a list that contains no data records. In other words it is a linked list with zero node.

3.1.1Types of linked List

The following are different types of linked list

- i.Singly linked list or one way list
- ii.Doubly linked list or two way list
- iii.Circular linked list

a)Singly linked list

The way to represent a linear list is to expand each node to contain a link or pointer to the next node. This representation is called a one-way chain or singly linked list.

Singly Linked list is a linear collection of elements.Each element is called a node. Each node contains two fields

- i.Data field
- ii.Link field

Data field contains the actual data and the link field contains the address of the next node(element). The first node address is called the pointer to the linked list. The last node link field contains null address.

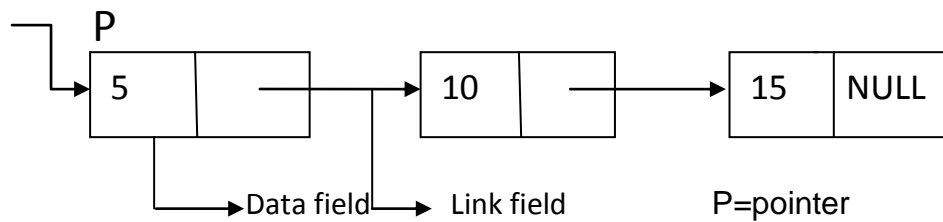


Fig 3.2

b)Doubly linked list

A linked list which can be traversed both in backward as well as forward direction is called doubly linked list. It uses double set of pointers.

A doubly linked list is a linked list in which each node is divided into three fields

- i. Data - to hold data
- ii. Slink - to hold the address of succeeding node
- iii. Plink - to hold the address of preceding node

Node structure

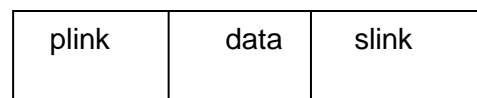


Fig 3.3

c)Circular linked list

A linked list in which the last node's **link field** contains the address of the first node is called circular linked list.

Node structure

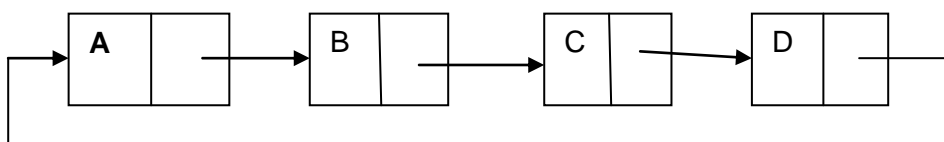


Fig3.4

As shown in the above figure, the last node does not contain a NULL pointer, but it points to the first node. Therefore circular linked list is also called a list without beginning and end.

3.1.2 Representation of linked list

Implementation is defined as the process of representing the linked list in memory using computer languages. Linked list can be implemented by any one of the following methods.

- Array implementation or static variable implementation.
- Pointer implementation or dynamic variable implementation

ARRAY IMPLEMENTATION

In array implementation the variables used in the implementation are declared while writing the program therefore the memory space for the variables will be allocated during compilation and remain unchanged (static) as long as program is running

Implementation

The representation of nodes in memory can be implemented in C-language using array of structures as

```
#define MAXNODES 100

struct N
{
    datatype data;
    int link ;
}

struct N node[MAXNODES];
```

Where

struct - keyword

N - structure name

Data type - valid data type such as int ,float, char etc....

Data - variable to store the data

Link - variable store the address of the next data

node - structure variable

The node are accessed as

node[0].data node[0].link => Node1

node [1].data node[1].link => Node2

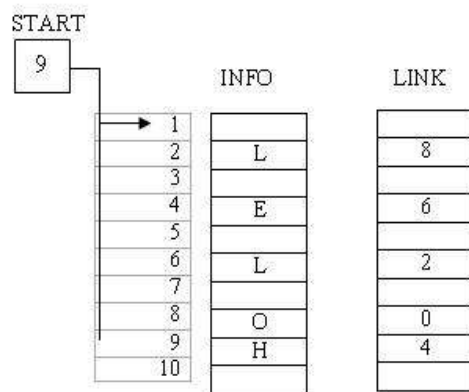
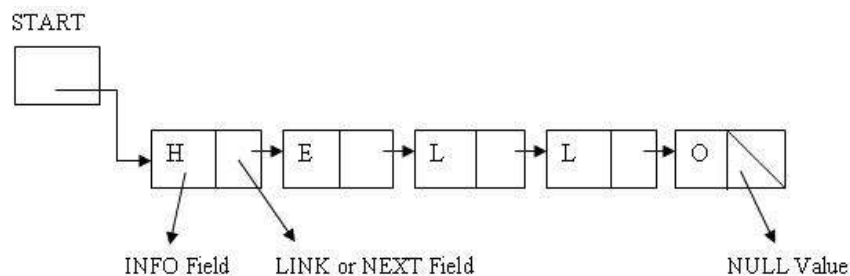
- - - - -
 - - - - -

node[99].data node[99].link => Node100

Example

Memory Representation of Linear Linked List:

Let LIST is linear linked list. It needs two linear arrays for memory representation. Let these linear arrays are INFO and LINK. INFO[K] contains the information part and LINK[K] contains the next pointer field of node K. A variable START is used to store the location of the beginning of the LIST and NULL is used as next pointer sentinel which indicates the end of LIST. It is shown below:



Here
 START = 9 => INFO[9] = H is the first character.
 LINK[9] = 4 => INFO[4] = E is the second character.
 LINK[4] = 6 => INFO[6] = L is the third character.
 LINK[6] = 2 => INFO[2] = L is the fourth character.
 LINK[2] = 8 => INFO[8] = O is the fifth character.
 LINK[8] = 0 => The NULL value, so the LIST ends here.

Fig 3,5

Self Referential Structure for Singly Linked List

A linear linked list is a chain of structures where each node points to the next node to create a list. To keep track of the starting node's address a dedicated pointer (referred as *start*

pointer) is used. The end of the list is indicated by a *NULL* pointer. In order to create a linked list of integers, we define each of its element (referred as *node*) using the following declaration.

```
struct node_type {
    int data;
    struct node_type *next;
};
struct node_type *start = NULL;
```

Note: The second member points to a node of same type.

A linear linked list illustration:



3.1.3 Difference between Array(sequential list) and Linked List

	Array(sequential list)	Linked List
Define	Array is a collection of elements having same data type with common name.	Linked list is an ordered collection of elements which are connected by links/pointers.
Access	In array, elements can be accessed using index/subscript value, i.e. elements can be randomly accessed like arr[0], arr[3], etc. So array provides fast and random access .	In linked list, elements can't be accessed randomly but can be accessed sequentially
Memory Structure	In array, elements are stored in consecutive manner in memory.	In linked list, elements can be stored at any available place as address of node is stored in previous node.
Insertion & Deletion	Insertion & deletion takes more time	Insertion & deletion are fast & easy
Memory Allocation	In array, memory is allocated at compile time i.e. Static Memory Allocation .	In linked list, memory is allocated at run time i.e. Dynamic Memory Allocation .
Types	Array can be single dimensional , two dimension or multidimensional .	Linked list can be singly , doubly or circular linked list.
Dependency	In array, each element is independent, no connection with previous element or with its	In Linked list, location or address of elements is stored in the link part of

	location.	previous element/node.
Extra Space in memory	No need of extra space in memory for pointer.	extra memory space is needed.

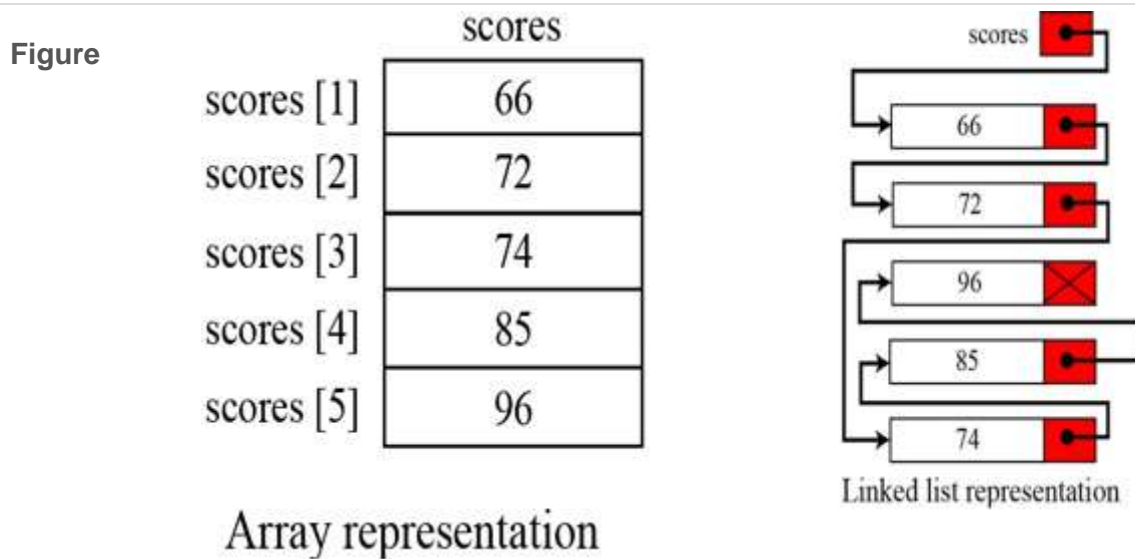


Fig 3.6

3.1.4 Advantages

- Dynamic structure (Memory. Allocated at run-time).
- We can have more than one datatype.
- Re-arrange of linked list is easy (Insertion-Deletion).
- Less wastage of memory.

Disadvantages

- It is difficult to access the nodes.
- Increased overhead for storing pointers(addresses) for linking data items
- It is occupying more memory for pointers.

3.1.4 Operations on a singly linked list (SLL)

We can perform various operations on linked list. The most common operations are:

- **Traversing** the list to access all elements (e.g., to print them, or to perform some specific operation);
- **Searching** a node to modify it or to obtain the information in it;
- **Inserting** a new node in a SLL (e.g., front, middle, or end)
- **Deleting** a node from a SLL (e.g., front, middle, or rear);

ALGORITHMS FOR LIST OPERATIONS

3.1.5.1 Traversing the list

. Traversing linked list means visiting each and every node of the Singly linked list. Following steps are involved while traversing the singly linked list –

- i) Check whether the linked list pointer **P** is NULL or not.
- ii) If NULL the list is empty.
- iii) If not NULL move to the first node
- iv) Fetch the data from the node and perform the operations depending on data type.
 - a) if reached last node, traversing over.
 - b) If not , advance pointer to next node and perform operation on visited node
- v) Go to step iv .

3.1.5.2 Searching for a particular element

Searching is a process to search the first occurrence of the given item in the list and to return the address of the node containing the item.

Steps to follow to search an item in a linked list

- (i) check whether the linked list pointer **P** is NULL or not.
- (ii) If NULL the list is empty.
- (iii) If not NULL, compare the item with **P -> data**
 - a) If equal searching over. The search item is present in the first node. Return **P**.
 - b) if not equal find the next node address as **ptr= p-> link**
- iv) Check **ptr**. If not null compare search item with **ptr -> data**
 - a) If equal searching over. Return ptr.
 - b) if not equal find the next node address as **ptr= ptr-> link**
- (v) Go to step (iv)

3.1.5.3 Insertion into a linked list

Insertion is a process to add a new element in the linked list. The new element is inserted either at first or middle or last position.

a) Insertion at the head of the list[as a first node]

The steps to be followed are

- i) Get a free node x from the memory using the function `getnode[]`. That is `x = getnode[]`.
- ii) Copy the data item into the data field and the linked list pointer value p [address] into the link field.
- iii) change the linked list pointer value [address] p with the address of the new node x.

Example

Consider a linked list pointer value [address] of the list as shown below

P=address of A

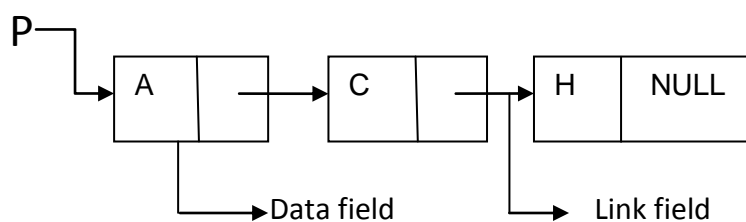
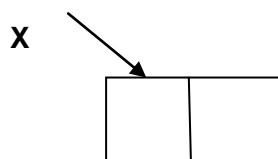
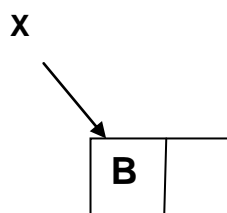


Fig 3.7

To insert a new data B into the list, get a new node, using `getnode[]` function. Let x be the address of the new node and it is shown below.



To the data field store the data B and is shown below



To insert this as a first node, the link field of x is filled by the pointer p [address of A] of the given list and then pointer [address] p is changed by x [address of the new node] as shown below.

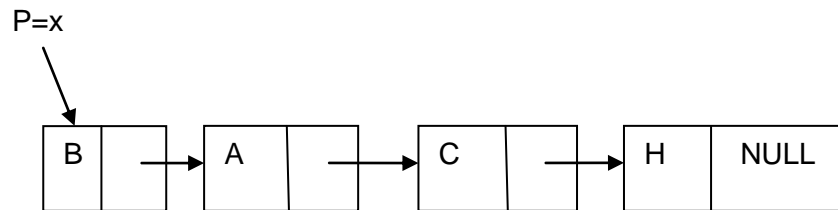


Fig 3.8

b) Insertion after a given node address y

The steps to be followed are

- i) Get a free node x from the memory using the function `getnode()`. That is `x = getnode()`.
- ii) Copy the data item into the data field and copy the link value of given node y into the link field of the new node x.
- iii) change the y node link field with new node address x.

Example

Consider a linked list with pointer p which contains the address of the list as shown below

P=address of A

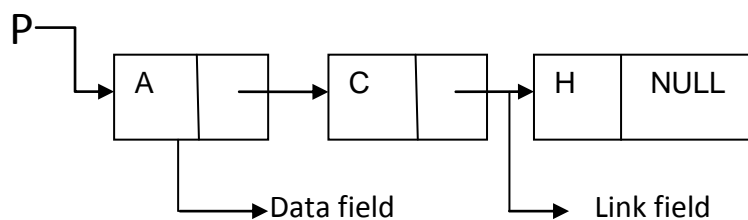
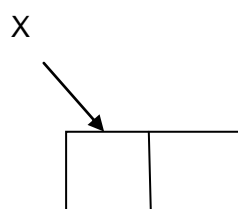
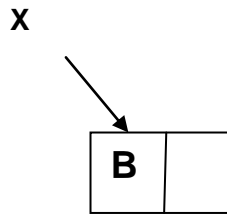


Fig 3.9

To insert a new data B into the list, get a new node, using `getnode()` function. Let x be the address of the new node and it is shown below.



To the data field store the data B and is shown below



The link field of new node x is filled by the link field of y and link field of y is changed by x (i.e. address of the new node) and is shown below.

P=address of A

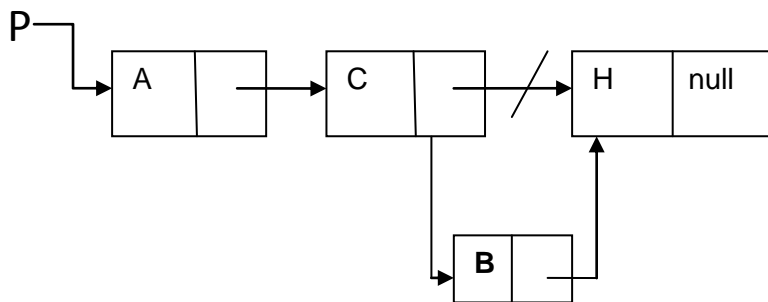


Fig 3.10

c) Insertion at the end of the list

The steps to be followed are

- i) Find the address t of the last node in the list by using the address of the linked list p. Last node is a node with NULL link field.
- ii) Get a free node from the memory using the function getnode(). That is $x = \text{getnode}()$.
- ii) Copy the data item into the data field and NULL in the link field of the new node x.
- iv) Change the link value of the last node t by the address of the new node x.

Example

Consider a linked list with pointer p which contains the address of the list as shown below

P=address of A

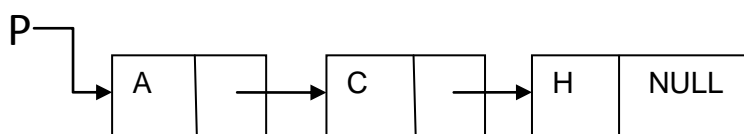
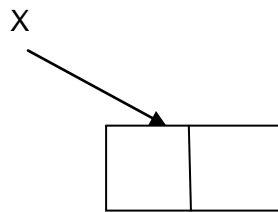
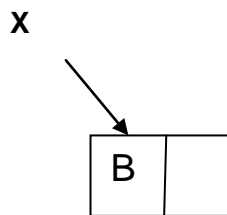


Fig 3.11

To insert a new data B into the list, get a new node, using `getnode()` function. Let x be the address of the new node and it is shown below.



To the data field store the data B and is shown below



To insert this as a last node, the link field of x is filled by NULL and the link field of the last node in the list is filled by the address of the new node x as shown below.

P=address of A

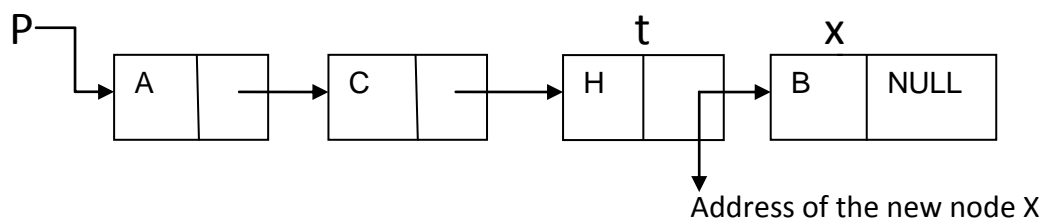


Fig 3.12

3.1.5.4 Deletion from a linked list

Deletion is a process to remove an existing node from the linked list. The node to be deleted may be the first node, or last node or intermediate node.

a) Deleting the head node of list [Deleting first node]

The steps to be followed are

- i) Replace the pointer of the linked list p with the second node address. That is $p = p \rightarrow \text{link}$.
- ii) use `free()` function to free the deleted node.

Example

Consider a linked list with pointer P which contains the address of the list as

P=address of A

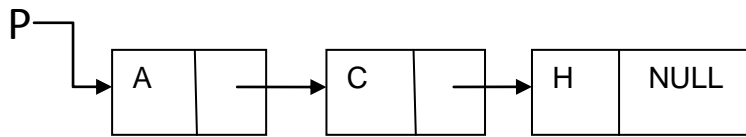


Fig 3.12

To delete the first node ,change the list address P by the second node address and it is shown below

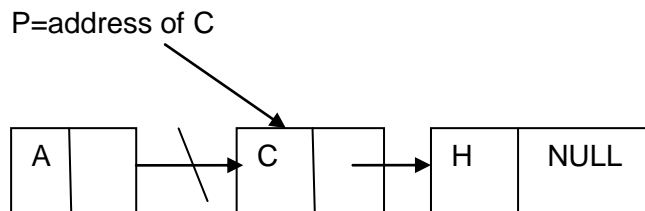


Fig 3.13

b)Deleting the given node

Let **q** be the address of the node to be deleted and r be the address of the previous node.

The steps to be followed are

- Check the value of r.If $r = \text{NULL}$,then the node to be deleted is the first node of the linked list.So change the address p of the linked list with $q \rightarrow \text{link}$.

That is $p = q \rightarrow \text{link}$

- If $r \neq \text{NULL}$, change $r \rightarrow \text{link}$ with $q \rightarrow \text{link}$.
- use `free()` function to free the deleted node.

Example

Consider a linked list with pointer P which contains the address of the list as

P=address of A

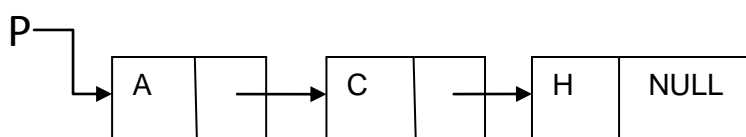


Fig 3.14

To delete the given node q, change the link field r of the preceding node of the node to be deleted by the address stored in the link field of the node q as shown below

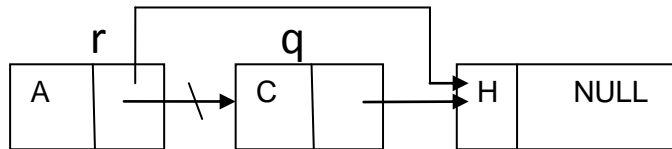


Fig 3.15

now node q will be deleted

c)Deleting the last node of list

The steps to be followed are

1. find the number of nodes n in the list by using the address of the Linked list p.
2. find the address k of the previous node of the last node using the Number of nodes n .
3. change the link field of node k by NULL.

Example

Consider a linked list with pointer p which contains the address of the List as shown below.

P=address of A

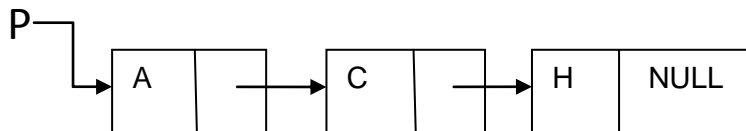


Fig 3.16

To delete the last node change the link field of the node with address k to NULL as shown below.

P=address of A

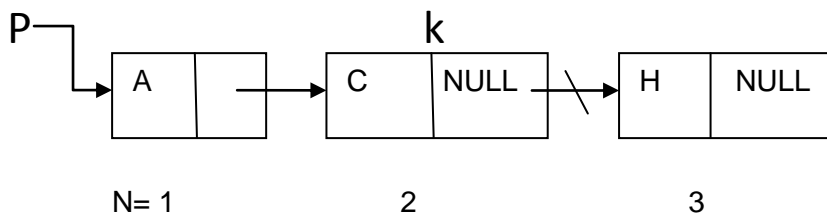


Fig 3.17

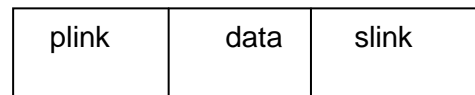
3.1.6 Doubly Linked List

A linked list which can be traversed both in backward as well as forward direction is called doubly linked list. It uses set of pointers.

A doubly linked list is a linked list in which each node is divided into three fields

- i. Data - to hold data
- ii. Slink - to hold the address of next or succeeding node
- iii. Plink - to hold the address of previous or preceding node

Node structure



There are two pointers in a two way list. They are

First - this points the leftmost node of the list

last - this points the rightmost node of the list

Using the pointer **First**, we can traverse the list in the forward direction and using the pointer **last**, we can traverse the list in the backward direction. The figure given below shows a two way list

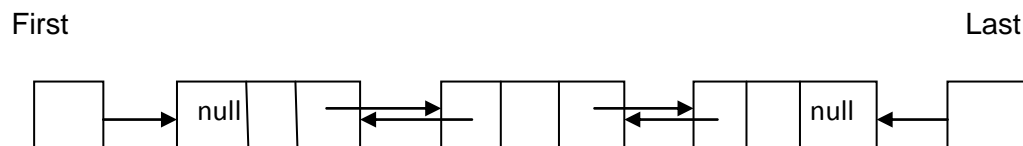


Fig 3.18

Declaration

A doubly linked list node can be declared as a self referential structure as

struct node

```
{  
    datatype data;  
    struct node *plink;  
    struct node *slink;  
};
```

Where

struct - keyword

Node - structure name

Datatype - valid datatype such as int,float,char etc

The above structure has three fields namely

- (i) data - variable to store the data. This can be any valid data type.
- (ii) plink - to store the address of the previous node containing the previous data
- (iii) slink - to store the address of the next node containing the next data

The plink and slink members are pointer members which points the same structure ,therefore this structure is called self referential structure.

3.1.7 Circular linked list

A linked list in which the last node's **link field** contains the address of the first node is called circular linked list.

Node structure

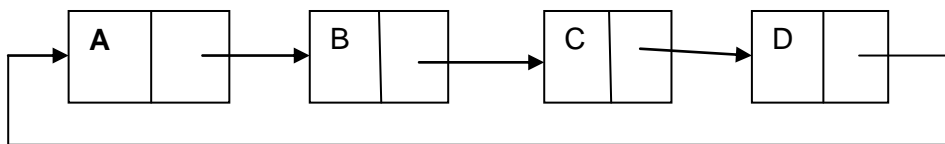


Fig 3.19

As shown in the above figure,the last node does not contain a NULL pointer, but it points to the first node. Therefore circular linked list is also called a list without beginning and end.

In the circular linked list we can insert elements anywhere in the list whereas in the array we cannot insert element anywhere in the list because it is in the contiguous memory. In the circular linked list the previous element stores the address of the next element and the last element stores the address of the starting element.

Advantages

- (i) There is no null reference in the nodes. This is useful for certain applications like playing video/audio repeatedly.
- (ii) From any node we can go to any node. But in linked list it is not possible to go to the previous node.
- (iii) Moving from last node to first node can be done in a single step. But in DLL one has to traverse from the last node to the first node.

Disadvantages

- (i) It is not easy to reverse
- (ii) It is not possible to go to the previous node in a single step as in DLL.

Review Questions

Part A

- 1 .Define linked list
2. What are the two fields in a linked list?
- 3.List out the types of linked list
4. What is null pointer?
5. What is empty list?
- 6.What are the two pointers in dll?
7. Define circular linked list
8. What are the three fields in a DLL node?
9. Draw a circular linked list
- 10.What are the disadvantages of circular linked list?

Part B

1. What is address? What are its types?
2. Define the different types of linked list
3. Define doubly linked list
4. What are the advantages of circular linked list?
5. What is meant by traversing a list?
6. Describe the purpose of the link fields of a doubly linked list?
7. List out the advantages of circular linked list
8. How to declare a doubly linked list?

Part C

- 1 .Explain array implementation of linked list
2. Tabulate the differences between sequential list and linked list
- 3 . List out the advantages and disadvantages of linked list
- 4 .Explain Traversing a linked list
5. Write down the algorithm to search for a particular element in a SLL?
6. Explain How to inserting a node at the front of a list?

- 7.. Explain How to insert a node as a last node in a list?
8. .Explain How to insert a node at a middle of a list?
9. .Explain deleting first node in a linked list?
10. Explain deleting last node in a linked list?

4.1.TREES

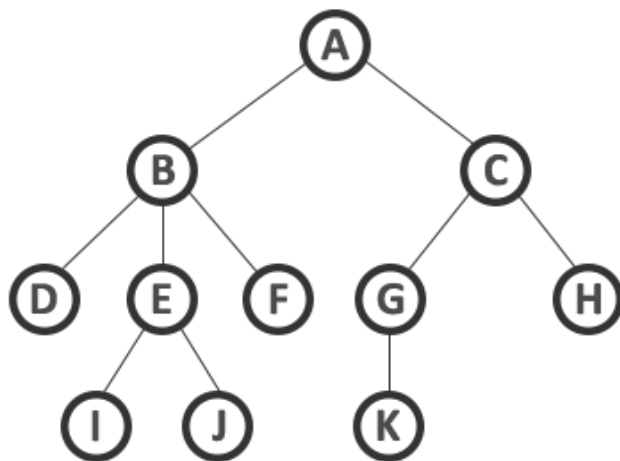
Objectives

- 4.1.1. To study basic terminologies of trees
- 4.1.2 To study about different types of trees
- 4.1.3 To explain about different traversal of binary tree
- 4.1.4 To study about expression tree
- 4.1.5 To study binary search tree
- 4.1.6 To explain how to create binary search tree

4.1.1.Terminologies

a)Degree of a node

In a tree data structure, the total number of children of a node is called as **DEGREE** of that node. In simple words, the Degree of a node is total number of children it has.



Here Degree of B is 3

Here Degree of A is 2

Here Degree of F is 0

- In any tree, 'Degree' a node is total number of children it has.

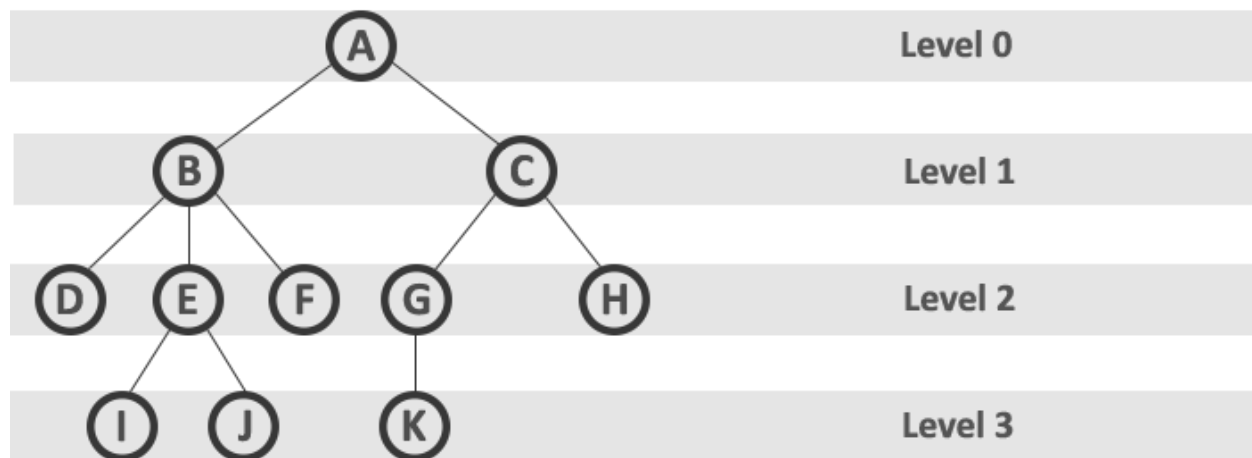
b)Degree of a tree

The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'. Here Degree of tree is 3.

c)Level of a node

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so

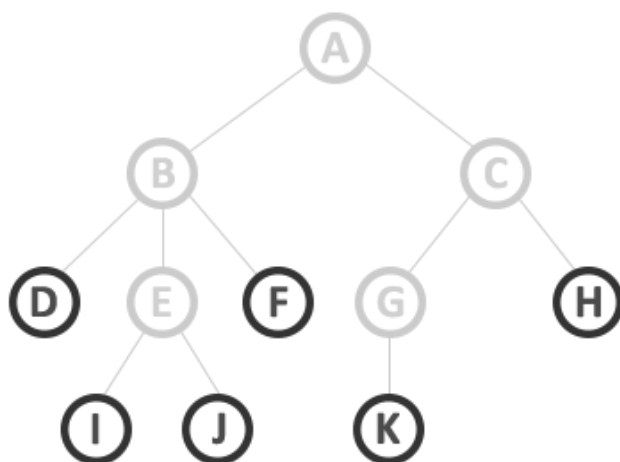
on. In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



d) Leaf node

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.

In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as '**Terminal**' node.

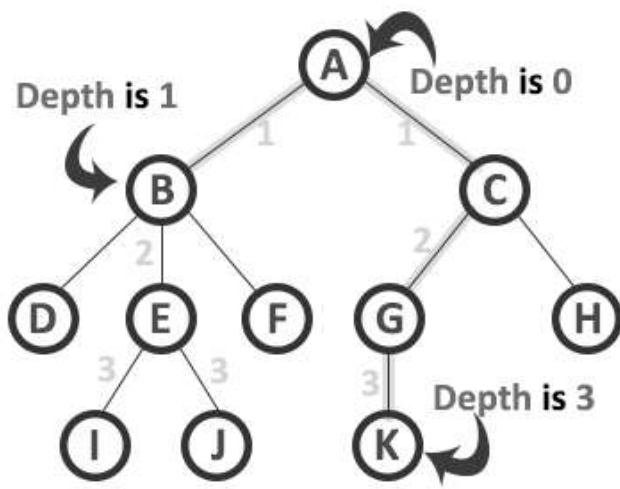


Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

e) Depth of a tree

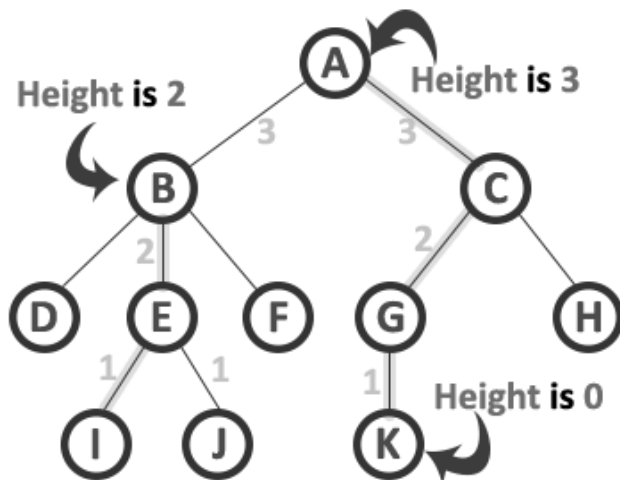
In a tree data structure, the total number of edges from root node to a particular node is called as **DEPTH** of that node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'**.



- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

f) Height of a tree

In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. In a tree, height of the root node is said to be **height of the tree**. In a tree, **height of all leaf nodes is '0'**.



- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.

g) In-degree & Out-degree

Indegree

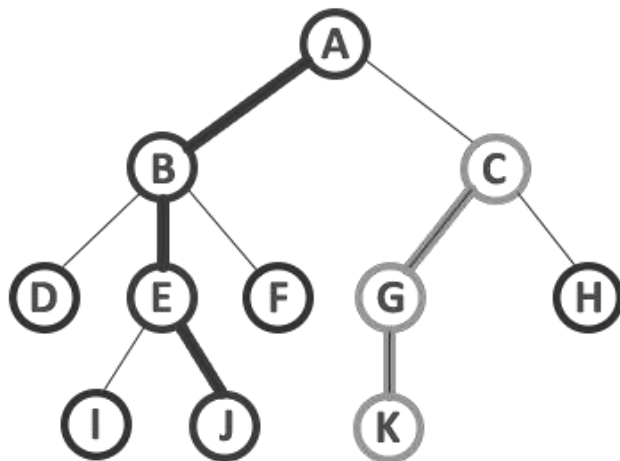
Indegree of a node U is the number of edges having terminal at U (ending at U)

Outdegree

Outdegree of a node U is the number of edges beginning at U

h) Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4**.



- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is

A - B - E - J

Here, 'Path' between C & K is

C - G - K

i) Ancestor & Descendant nodes

Ancestors of a node are all the nodes along the path from the root to that node.

Ancestor of node J are A, B and E

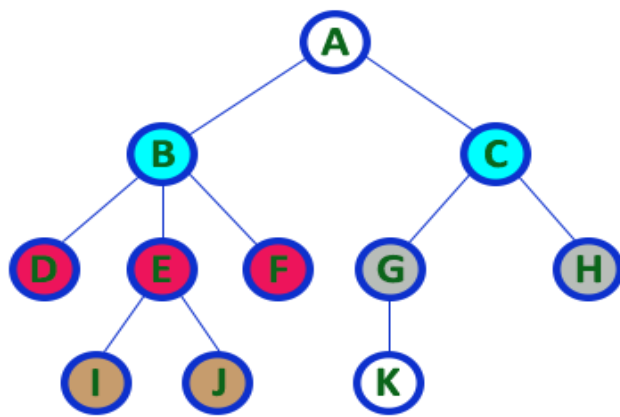
A node N1 is called descendant of a node N, if N is the ancestor of N1

H is the descendant of A

- **Ancestor:** The predecessor of a node together with all the ancestors of the predecessor of a node. The root node has no ancestors.
- **Descendant:** The children of a node together with all the descendants of the children of a node. A leaf node has no descendants.

j) siblings

In a tree data structure, nodes which belong to same parent are called as **SIBLINGS**. In simple words, the nodes with same parent are called as sibling nodes.



Here B & C are Siblings
 Here D E & F are Siblings
 Here G & H are Siblings
 Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

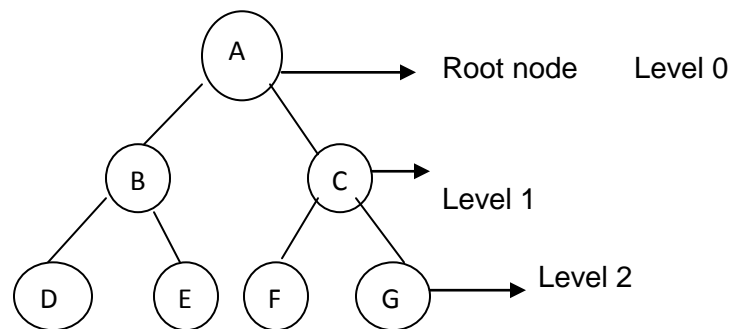
4.1.2Types of Trees

4.1.2.1Binary tree

Binary tree is a tree in which each node has at most two children called left and right . [OR]

A binary tree is a finite set of element which are either empty or consist of a root and two disjoint sub trees called left sub tree and right sub tree .

Each element in the binary tree is called the node of the binary tree. A left or right or both sub trees can be empty. A binary tree is shown in the figure.



This tree contains 7 nodes in which A is root node. Left sub tree of A is rooted at B and right sub tree of A is rooted at C. In other words A is the father of B and C or B and C are left and right child of A.

B is the left successor of A and C is the right successor of node A.

Note

1. The maximum number of nodes in a binary tree at level l is 2^l .
2. the maximum number of nodes in a binary tree of height r is $2^r - 1$.

Representation of binary tree

Let T be a binary tree we can represent this in memory using any one of the following methods.

1.Array representation or sequential representation

2. Linked list representation

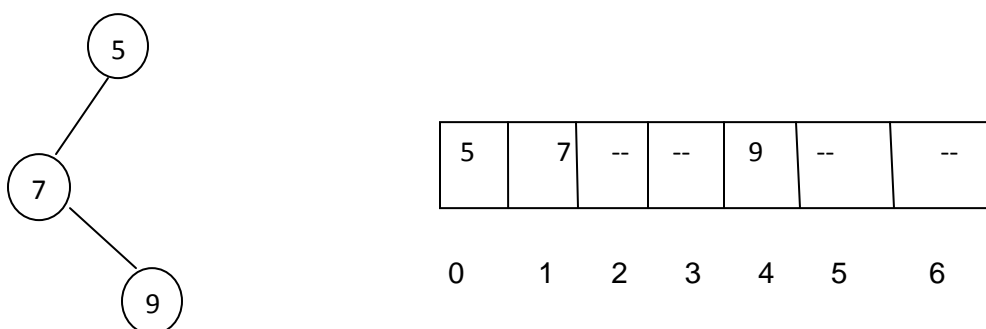
1.Array representation or linear representation

In this representation, the binary tree T is represented using a one dimensional array named as TREE The steps to be followed are.

- (a) The root node of binary tree T is stored in TREE [0]
- (b) The left child of root node is stored in TREE [1] and the right child of root node is stored in TREE[2]
- (c) In general, if a node is stored at TREE [K], then its left child is stored at TREE [2*K+1] and its right child is stored at TREE [2*K+2]
- (d) If there is no left or right child node, its related array position is left free

Example

1 Consider a binary tree as given below



The height of the tree is 3. therefore to store the above binary tree we are in need of an array of size $2^3-1=7$ and it is shown above.

The location 2,3,5,6 are left free because there are no nodes corresponding to these locations.

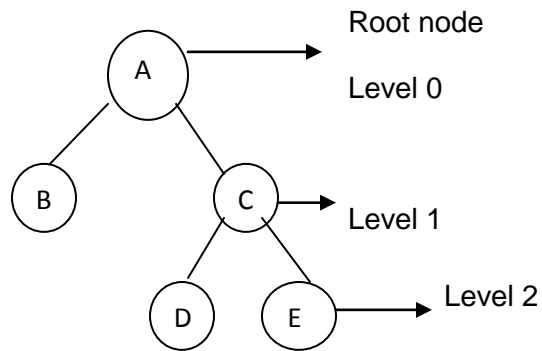
4.1.2.2 Linked list representation

In this representation each node of a binary tree is defined as

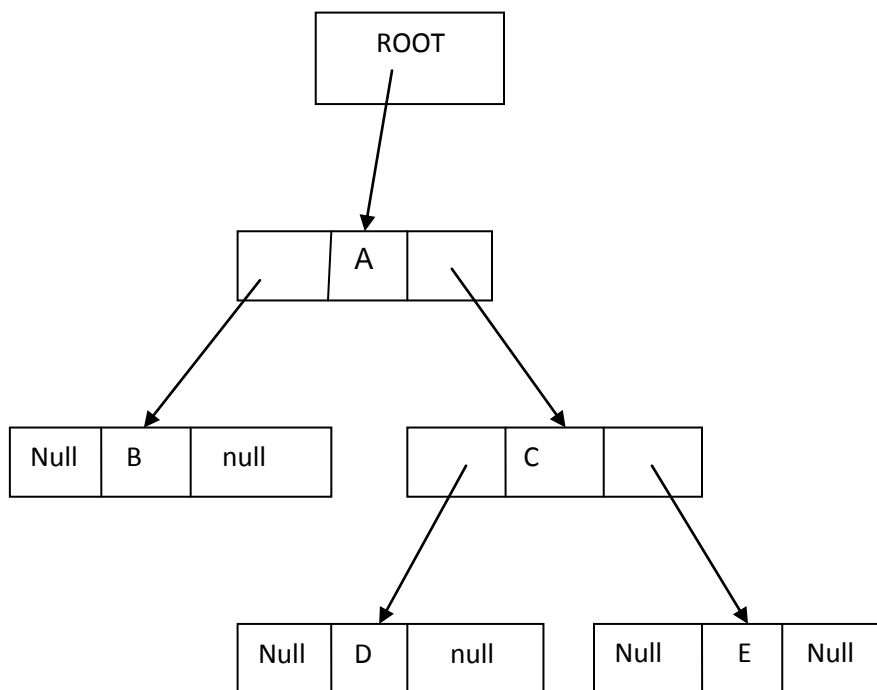
LEFT	DATA	RIGHT
------	------	-------

DATA field contains information.LEFT field holds the address of left child node and RIGHT field holds the address of right child node. If there are no children,the address fields hold NULL value.The first node address is called **Pointer** of the tree.

Using this address only we can access the tree. Consider the tree given below



The linked list representation of the above tree is



4.1.3 Binary tree traversal

Binary tree traversal is defined as a process of visiting all the nodes in the binary tree once. The visit always start from the root node we visit the node to do any one of the following

- 1 To read (print) or write data in the node. It is denoted by letter v.
- 2 To move to the left of that node It is denoted by letter l

3 To move to the right of that node It is denoted by letter R

There are three types of traversal they are

- (a) Inorder traversal
- (b) Preorder traversal
- (c) Post order traversal

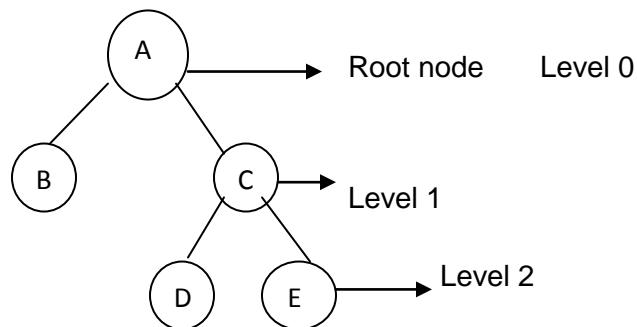
(a)Inorder traversal (LVR)

This traversal is denoted by the letters LVR .The steps to be followed are

- (a) Start from the root node and move left until there is no left child
Then visit the last node (print the data)
- (b) Move right and move left until there is no left child Then visit the last node
- (c) If it is possible to move right go back one node visit the node (print the data) and do step(b).This traversal is called inorder traversal.

Example

Consider the binary tree



Steps

- i)Start from the root node A and move left till last node that is upto node B and print the data B.
- ii) Move one step back and print the data A
- iii)Then move right to node C and move left till last node that is upto D ,print the data D.
- iv)Move one step back and print the data C.
- v) Then move right to node E since no path to move .print the data E.

Therefore,the result of the inorder traversal is B A D C E

b) Pre order traversal (VLR)

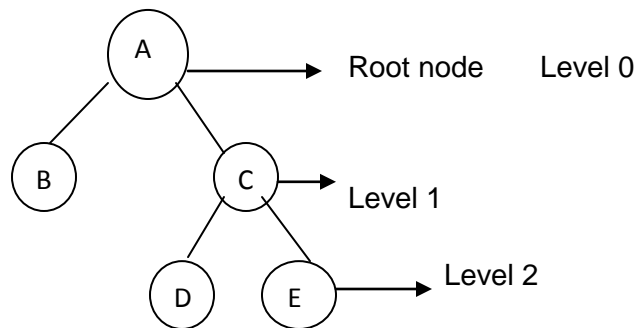
This traversal is denoted by the letter VLR .The steps to be followed are

- (a) Start from the root node and visit this node (print the data)
- (b) Move left , print the data and this process is continued till there is no path to move.

- (c) Move right print the data and move left , print the data and this process is continued till there is no path to move.
- (d) Move one node back and do step (c)

Example

Consider the binary tree



Steps

- i) Start from the root node A and print the data A.
- ii) Move left to node B and print the data B
- iii) Since there is no node to move left or right move one node back that is to node A
- iv) From this node move right to node C and print the data C and move left to node D and print data D.
- v) From node D there is no node to move left or right so move one node back to C and move right to node E and print the data E

Therefore, the result of the preorder traversal is A B C D E

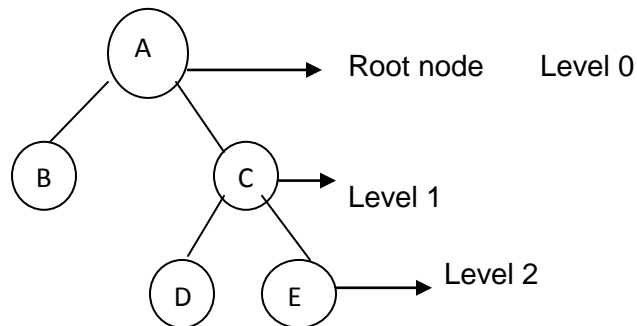
c) Post order traversal (LRV)

This traversal is denoted by the letters LRV. The steps to be followed are

- (a) Start from the root node and move left until no left node to move and print the data
- (b) Move right and then move left until there is no left node and print the data
- (c) If there is no right or left node to move print the data.
- (d) Move one node back and do step (b)

Example

Consider the binary tree



Steps

- i) Start from the root node A and move left until no left node to move that is to node B and print the data B
- ii) Then go back to node A and move right to node C and move left until no left node to move that is to node D and print the data D
- iii) Then go back to node C and move right to node E. Since no left or right to move print the data E
- iv) Move one node back to node C print the data C and move one node back to node A and print the data A.

Therefore, the result of the preorder traversal is B D E C A.

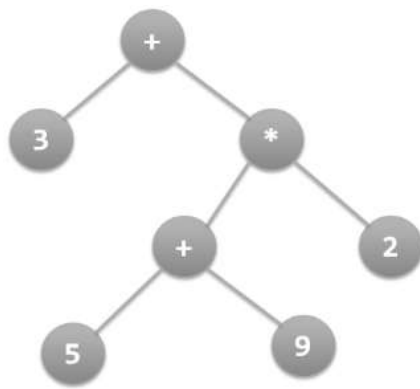
4.1.4.Expression tree

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand such as constants or variable names.

It is possible for nodes to have more than two children. It is also possible for a node to have only one child, as is the case with the unary minus operator.

An expression tree, T , can be evaluated by applying the operator at the root to the values obtained by recursively evaluating the left and right sub trees.¹

for example expression tree for $3 + ((5+9)*2)$ would be:



4.1.5 Binary search tree

Definition

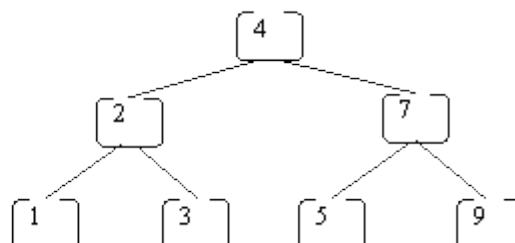
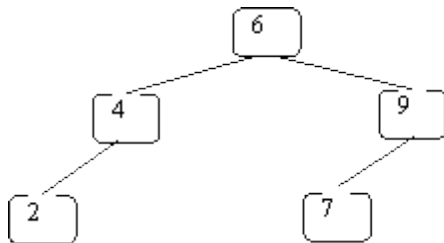
A binary tree T is said to be binary search tree if

1. All elements in the left sub tree of a node n should be less than the value of the node n .
2. All elements in the right sub tree of a node n should be greater than the value of the node n .

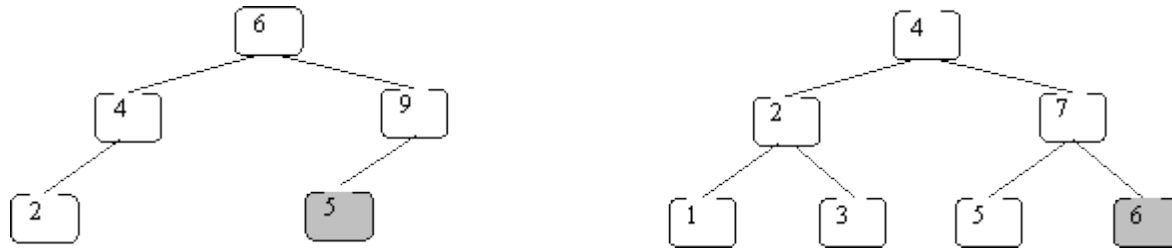
. A binary tree is a BST iff, for every node n in the tree:

- All keys in n 's left subtree are less than the key in n , and
- all keys in n 's right subtree are greater than the key in n .

Here are some BSTs in which each node just stores an integer key:



These are not BSTs:



In the left one 5 is not greater than 6. In the right one 6 is not greater than 7.

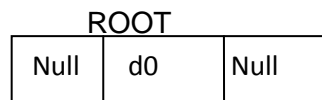
The reason binary-search trees are important is that the following operations can be implemented efficiently using a BST:

- insert a key value
- determine whether a key value is in the tree
- remove a key value from the tree
- print all of the key values in sorted order

4.1.6 Creation of binary search tree for the given set of data .

The steps given below are followed to create a binary search tree for the given set of data.

1. Get the first data say d0 and place it as a root node data with empty left and right child as.



2. Get the next data in the list and compare it with the root value.
 - (a) If the data value is less do step (b) else if else data value is greater do step (c) .
 - (b) Check if left child is NULL or not . if NULL place the data value as The new left child.

Else

Compare it with the existing left child data value and do step (a).

- (c) Check if right child is NULL or not . if NULL place the data value as The new right child .

Else

Compare it with the existing right child data value and do step (a)

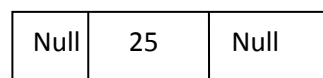
3. Repeat step (2) for all the given n data.

Example

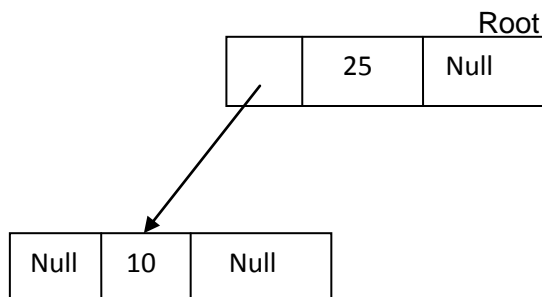
Create a binary search tree for the given data {25,10,40,17,24}

Steps

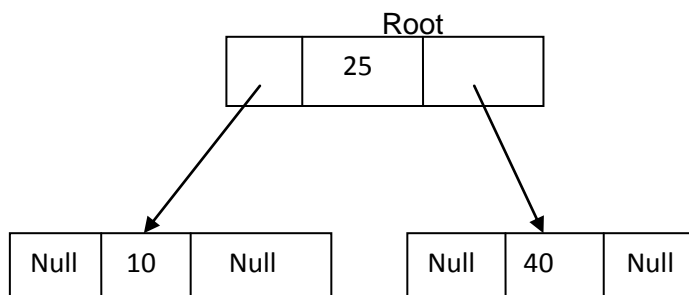
1. Get a free node by calling the function `getnode()` and store the first Data 25 in the data field and NULL in the lchild and rchild field as.



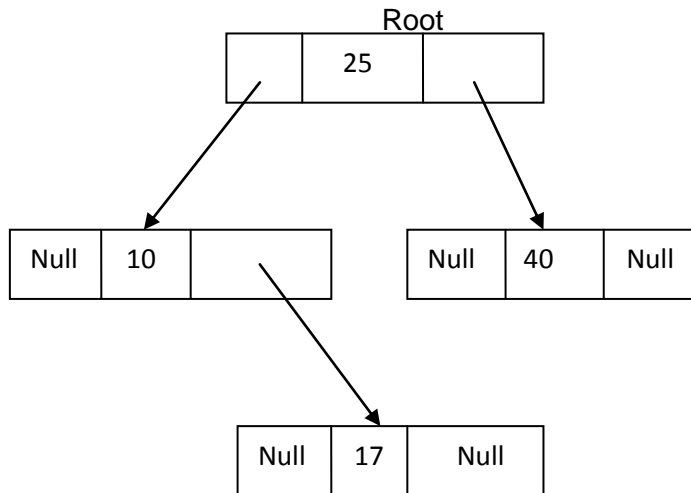
2. The next data is 10. This is compared with root and is less ($10 < 25$). Since left child is empty place it as the left child of root with Null value in the lchild and rchild field as.



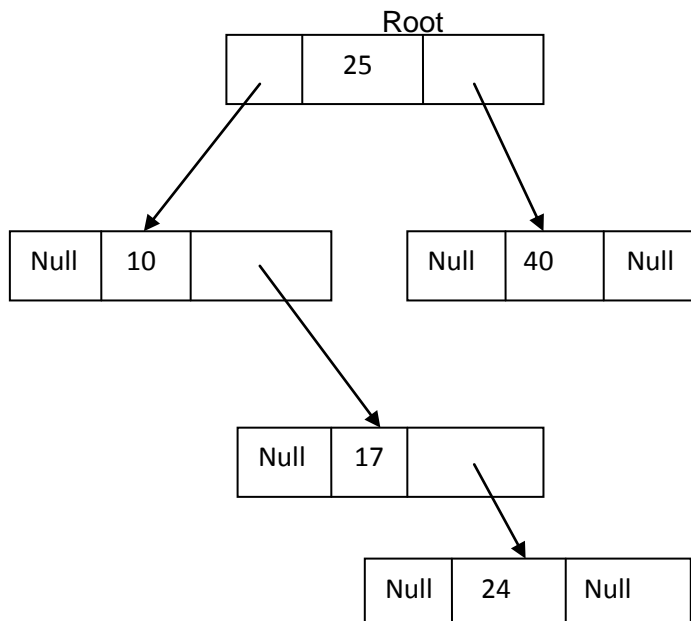
3. The next data is 40. Compare it with the root it is greater. since right child is empty place it as the right child of root with Null value in the lchild and rchild field as.



4. The next data is 17 . compare it with the root, it is less . since left child of root is not empty compare it with the value of the left child 10, It is greater. Since right child is empty place it as the right child as



5. The next data is 24 . compare it with the root, it is less . since left child of root is not empty compare it with the value of the left child 10, It is greater. Since right child of node 10 is not empty,compare it with the right child value 17,it is greater. Since right child of node 17 is null,place it as the right child as



Review questions

Part A

1. Define tree
2. What is node?
3. What do you mean by root?
4. What is sibling?
5. Define level?
6. Define height?
7. What is depth of a tree?
8. What is degree of a tree?
9. What is binary tree?
10. What is path?

Part B

1. What is indegree and out degree ?
2. Define ancestor and descendant nodes
3. What are the different types of traversal?
4. Define binary search tree Give example
5. What is expression tree .Give example.

Part C

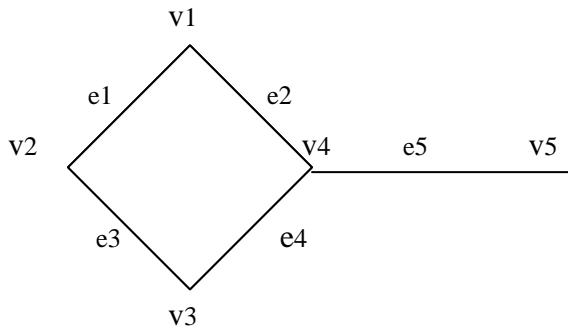
1. Explain any five basic terminologies of tree
 2. Explain i) binary tree ii) Expression tree
 3. Explain list representation of binary tree with example
 4. Explain inorder traversal of binary tree with example
 5. Explain preorder traversal of binary tree with example
 6. Explain post order traversal of binary tree with example
 7. Explain how to create a binary search tree with example
- .

OBJECTIVES

- To understand graph terminologies
- To know about the different ways of traversing a graph.

4.2 Graph:

A graph is defined as a set of nodes or vertices and a set of lines or edges that connect the two vertices. The set of vertices is specified by listing the vertices as in a set and the set of edges is specified as a sequence of edges.



$v_1, v_2, v_3, v_4, v_5 \rightarrow$ Nodes

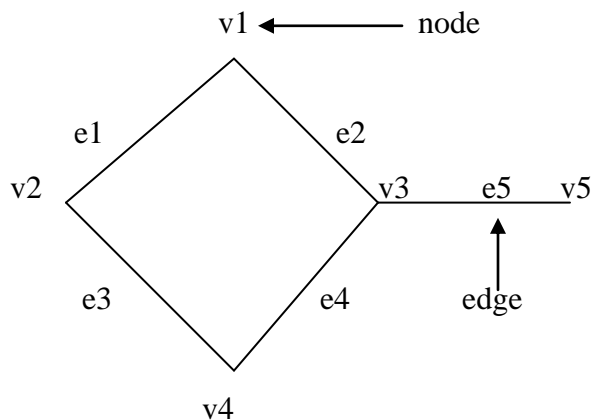
$e_1, e_2, e_3, e_4, e_5 \rightarrow$ edges

A graph is denoted as $G=(V,E)$, Where

$$V = \{v_1, v_2, v_3, v_4, v_5\}$$

$$E = \{e_1, e_2, e_3, e_4, e_5\}$$

Graph Terminologies:



Node(Vertices) :

Each element of a graph is called node of the graph.

Edge (Arcs) :

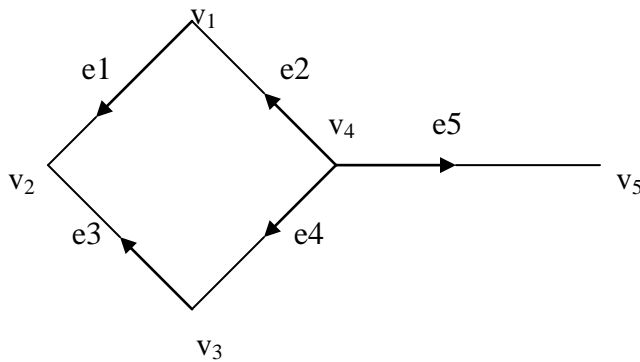
Line joining two nodes is called edge. It is denoted by $e=[u,v]$ where u and v are adjacent nodes.

Example

$$e1=[v1,v2]$$

Directed Graph

A graph G is called directed graph if each edge has a direction. Each edge is denoted as an ordered pair of vertices $e = (u,v) \neq (v,u)$. The directed edges are called *axis*.



Here $e1=(v1,v2)$ $e2=(v1,v4)$ $e3=(v3,v2)$ $e4=(v3,v4)$ $e5=(v4,v5)$

$e(u,v) \Rightarrow u \rightarrow$ Initial point

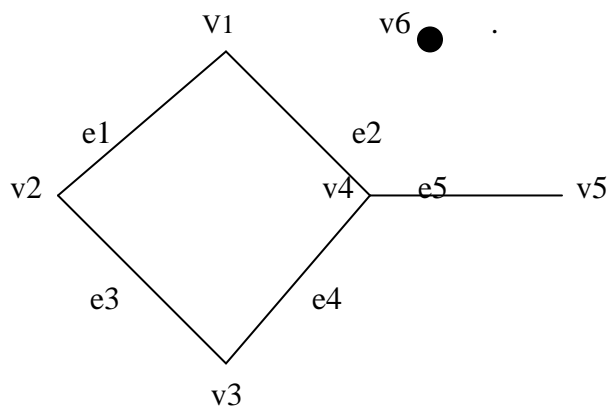
$v \rightarrow$ Terminal point

u and v are adjacent vertex

Here $e(v1,v2) \neq e(v2,v1)$ because of direction .

Degree of a node:

Degree of a node is the number of edges connecting the node in the graph. It is denoted as $\deg(v)$ where v is a node.



$\deg(v_1)=2$, $\deg(v_2)=2$, $\deg(v_3)=2$, $\deg(v_4)=3$, $\deg(v_5)=1$, $\deg(v_6)=0$

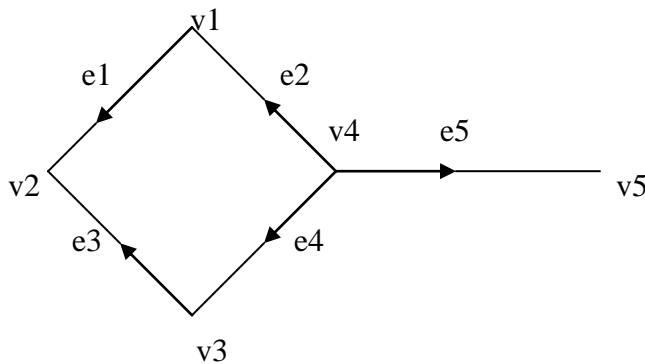
Indegree :

The indegree of a vertex is the number of edges pointing to that vertex. i.e. number of edges having the vertex as terminal point.

Outdegree :

The outdegree of a vertex is the number of edges pointing from that vertex. i.e. number of edges having the vertex as initial point.

Example :



$\text{indegree}(v_1) = 1$	$\text{outdegree}(v_1) = 1$
$\text{indegree}(v_2) = 2$	$\text{outdegree}(v_2) = 0$
$\text{indegree}(v_3) = 1$	$\text{outdegree}(v_3) = 1$
$\text{indegree}(v_4) = 0$	$\text{outdegree}(v_4) = 3$
$\text{indegree}(v_5) = 1$	$\text{outdegree}(v_5) = 0$

Adjacent or neighbours:

Two nodes u and v are adjacent if there is an edge between u and v . (i.e) $e=[u,v]$

Example:

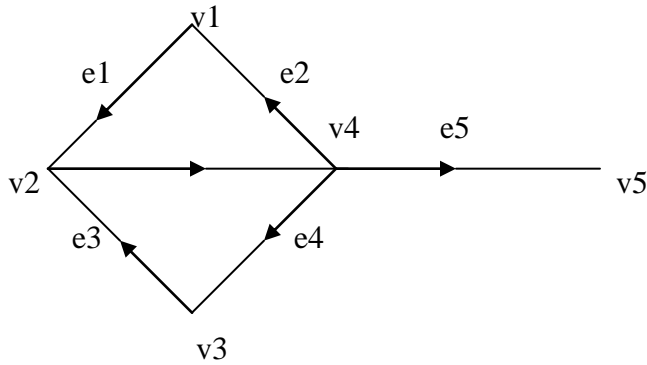
v_1 and v_2 are adjacent since there is an edge e_1 between them.

v_1 and v_3 are not adjacent since there is no edge between them.

Successor and Predecessor:

If a node is reachable from another node, then the first node is a predecessor of second one and second node is a successor of first one. If there is an arc from first node to second node, then first node is a direct predecessor of second one, and second node is a direct successor of first one.

Example :



Successor of v1 : v2,v4

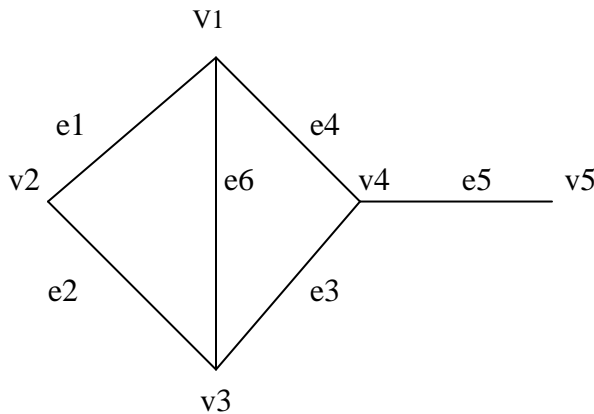
Direct Successor of v1 : v2

Predecessor of v4 :v2,v1

Direct Predecessor of v4 :v2

Path :

A path is an alternate sequence of nodes and edges . The uninterrupted sequence of edges from starting vertex and ending vertex



The different paths are $p1=(v1,v2,v3)$, $p2=(v1,v4,v3)$

Length of the Path :

The length of the path is the number of edges in the path. In other words, number of vertices – 1

Example :

$$p1=(v1,v2,v3)$$

$$\text{Length}(p1) = 3 - 1 = 2$$

Simple Path :

A path is simple, if all nodes are distinct except the first and last.

Example :

$$p1=(v1,v2,v3) \rightarrow \text{Simple path}$$

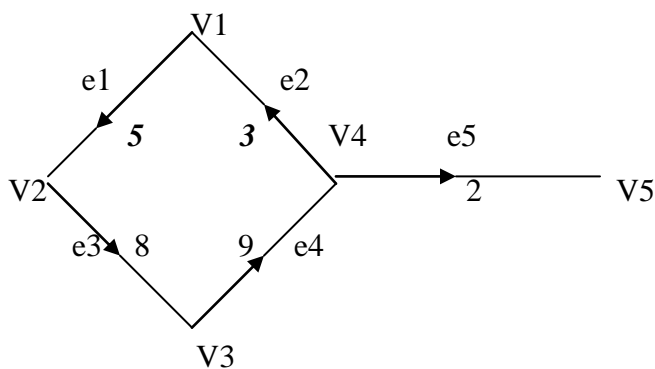
$$P2=(v1,v4,v5,v4,v3) \rightarrow \text{Not simple path because vertex } v4 \text{ is repeated.}$$

Weighted Graph:

The edge of a graph may carry a value which may represent some important information. Such a graph is called weighted graph.

Example:

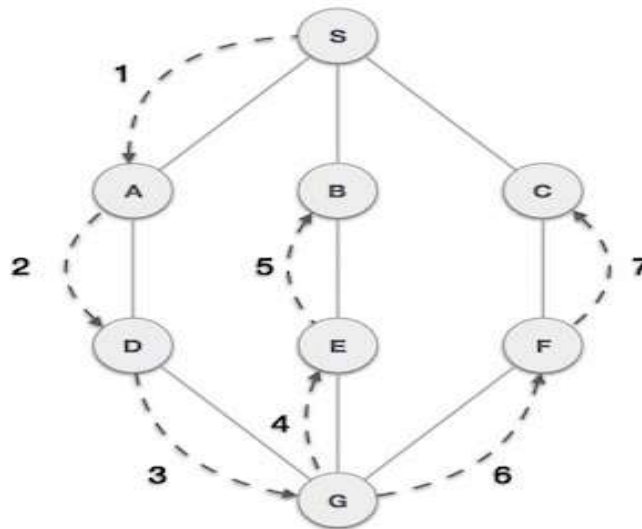
The following graph represents various bus stops and the edges indicates that the bus stops are connected through a bus route. The weights attached edges represent the distance in kilometers between pair of bus stops.



$$W(e1) = 5 \quad W(e2) = 3 \quad W(e3) = 8 \quad W(e4) = 9 \quad W(e5) = 2$$

Traversal of Graphs:**Depth First Search algorithm:**

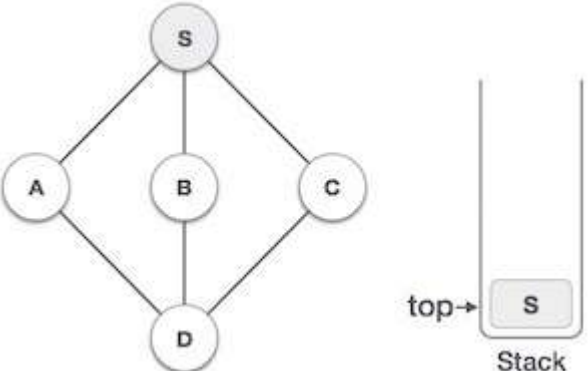
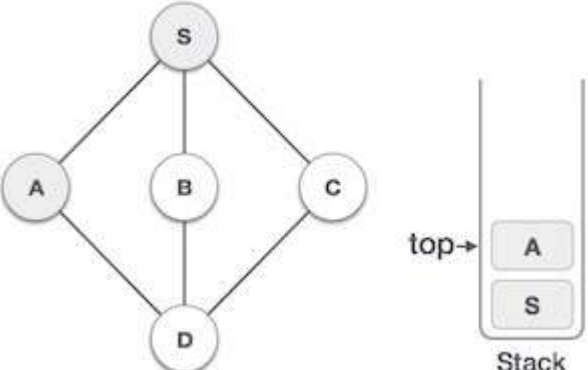
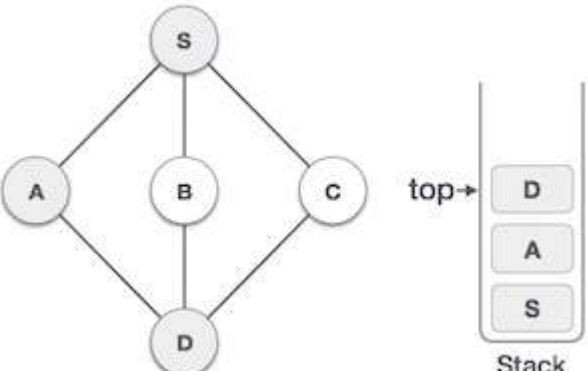
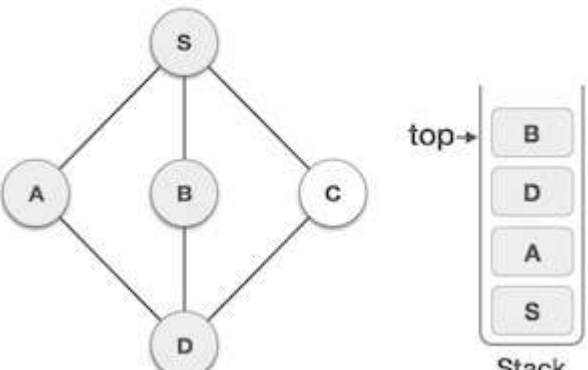
DFS traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.

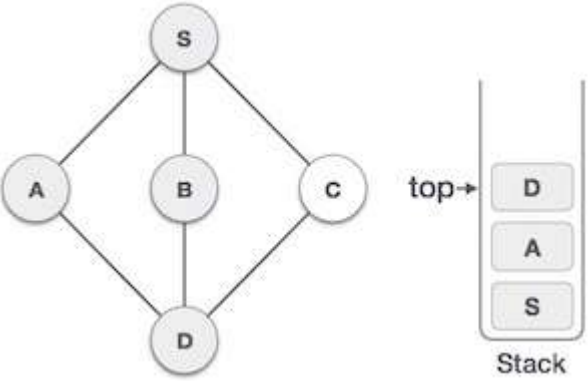
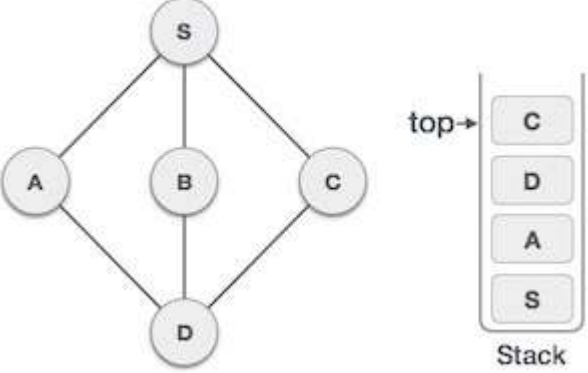


As in example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. It employs following rules.

- **Rule 1** – Visit adjacent unvisited vertex. Mark it visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex found, pop up a vertex from stack. (It will pop up all the vertices from the stack which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until stack is empty.

Step	Traversal	Description
1.		Initialize the stack

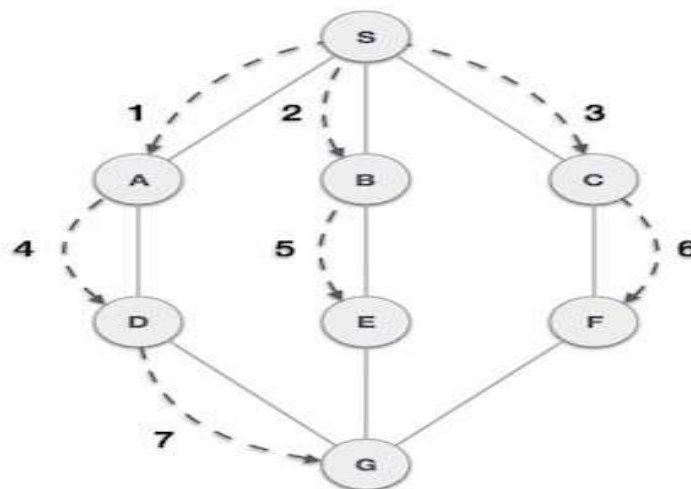
2.		<p>Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in alphabetical order.</p>
3.		<p>Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.</p>
4.		<p>Visit D and mark it visited and put onto the stack. Here we have B and C nodes which are adjacent to D and both are unvisited. But we shall again choose in alphabetical order.</p>
5.		<p>We choose B, mark it visited and put onto stack. Here B does not have any unvisited adjacent node. So we pop B from the stack.</p>

6.		<p>We check stack top for return to previous node and check if it has any unvisited nodes. Here, we find D to be on the top of stack.</p>
7.		<p>Only unvisited adjacent node is from D is C now. So we visit C, mark it visited and put it onto the stack.</p>

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node which has unvisited adjacent node. In this case, there's none and we keep popping until stack is empty.

Breadth First Search algorithm:

BFS traverses a graph in a breadth wards motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.

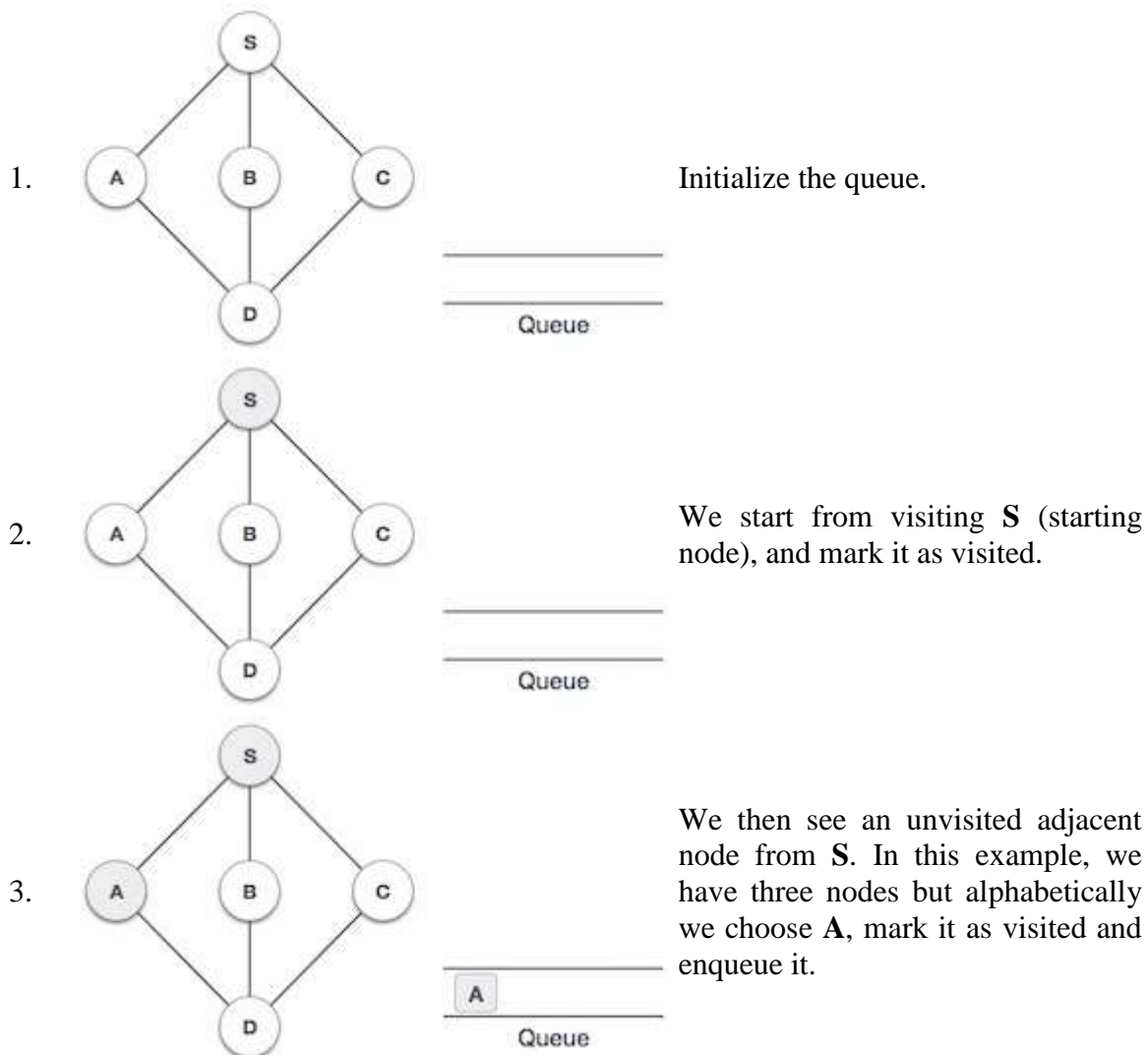


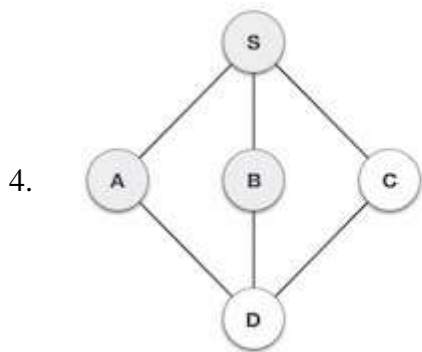
As in example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs following rules.

- **Rule 1** – Visit adjacent unvisited vertex. Mark it visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex found, remove the first vertex from queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until queue is empty.

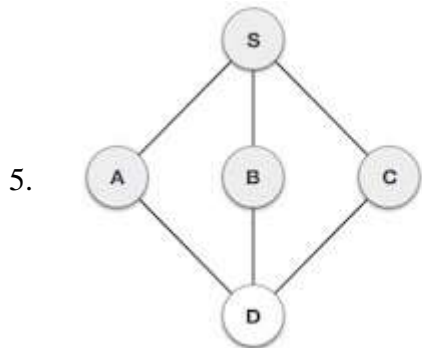
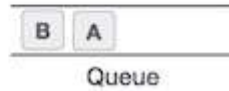
Step Traversal

Description

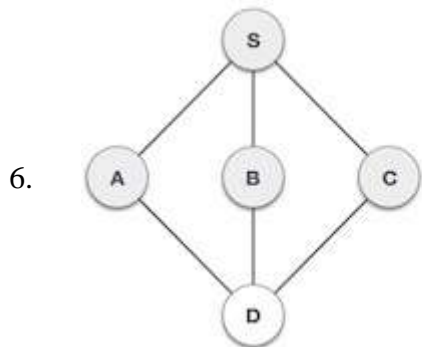




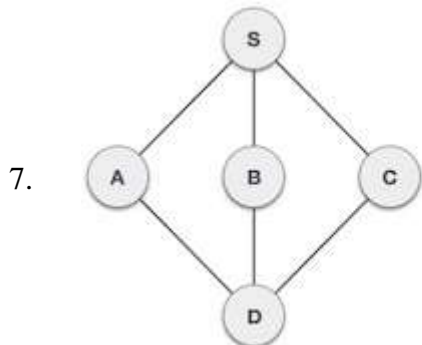
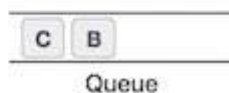
Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.



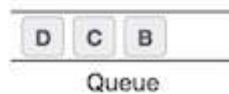
Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.



Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.



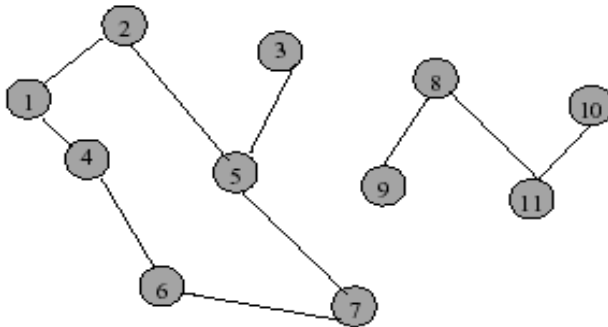
From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.



At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Applications of Graph

- Nodes could represent positions in a board game, and edges the moves that transform one position into another ..
- Connecting with friends on social media, where each user is a vertex, and when users connect they create an edge.



vertex = router

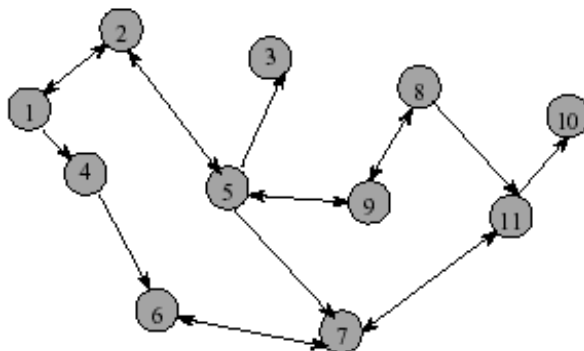
edge = communication link

- Using GPS/Google Maps/Yahoo Maps, to find a route based on shortest route.
- Google, to search for webpages, where pages on the internet are linked to each other by hyperlinks; each page is a vertex and the link between two pages is an edge.
- On ecommerce websites relationship graphs are used to show recommendations.

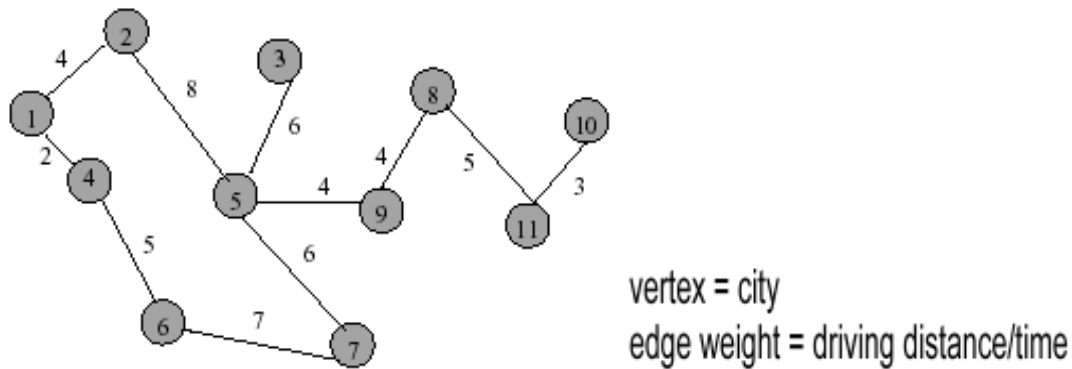
Nodes could represent computers (or routers) in a network and weighted edges the bandwidth between them

Street Map

- Streets are one- or two-way.
- A single directed edge denotes a one-way street
- A two directed edge denotes a two-way street



- nodes could represent towns and weighted edges road distances between them, or train journey times or ticket prices ...



Representations of a graph:

Representation of graph is a process to store the graph data into the computer memory.

- i. Set Representation
- ii. Sequential Representation
- iii. Linked list Representation

i. Sequential Representation:

In this representation, the graph G is represented in a matrix form using the following methods.

- i. Adjacency matrix
- ii. Path matrix

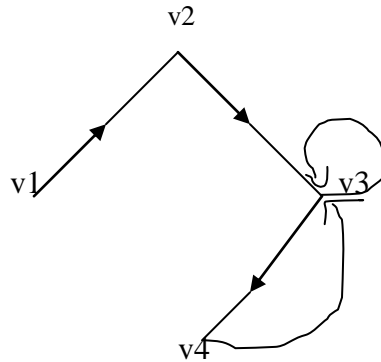
Adjacency matrix:

Let G be a simple directed graph with n nodes. The nodes are $v_1, v_2, v_3, \dots, v_n$ then the adjacency matrix $A = (a_{ij})$ of order $n \times n$

$$\text{Where } a_{ij} = \begin{cases} 1 & \text{if } v_j \text{ (i.e.) } (v_i, v_j) \text{ belongs to edge set} \\ 0 & \text{Otherwise} \end{cases}$$

$$i, j = 1, 2, 3, \dots, n$$

This matrix is also called **bit matrix or boolean matrix** because all entries of matrix are 0 or 1



$$G = (V, E)$$

$$V = \{v1, v2, v3, v4\}$$

$$E = \{(v1, v2), (v2, v3), (v3, v3), (v3, v4), (v4, v3)\}$$

Adjacency matrix:

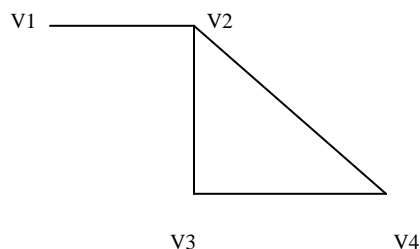
	v1	v2	v3	v4
v1	0	1	0	0
v2	0	0	1	0
v3	0	0	1	1
v4	0	0	1	0

The number of vertices is 4, then the adjacency matrix A is size 4 x 4. Since there are 5 edges in the edge set, the matrix contains five entries of 1.

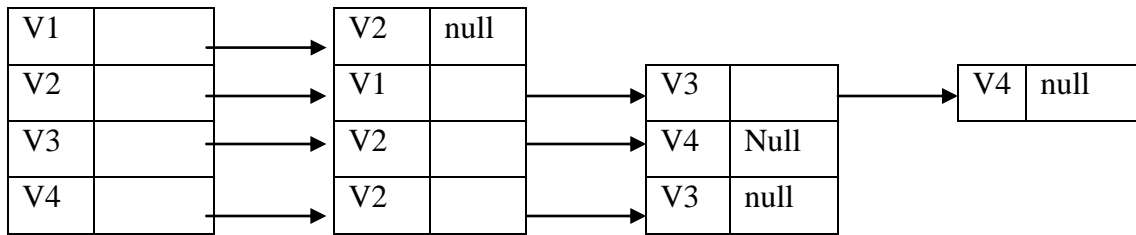
ii. Linked list Representation:

In this representation the graph G is represented as a collection of data nodes and head nodes. Each node contains two fields called data and link. Data field contains information and the link field contains the address of the next adjacent node. This is called **Linked List Representation**

Example:



Head Node:



Review Question

PART A

1. Define directed graph?
2. Define path?
3. Define length of a graph?
4. Expand DFS and BFS.

PART B

1. Write short notes on the following
 - a) in-degree b) out-degree c) edge
2. Write down the application of the graph
3. Draw BFS graph with 8 node.
4. Draw DFS graph with 8 node.
5. Explain Set representation of graph.

PART C

1. Explain DFS with example
2. Explain BFS with algorithm.
3. Explain Adjacency List Representation of graph.
4. Explain Matrix representation of graph.
5. Write down the comparison of graph representation.

Unit 5 : Sorting, Searching & Hashing

Objectives

- To define sorting and searching
- To list types of sorting and searching
- To explain the difference between different methods of sorting and searching
- To understand algorithms of various sorting and searching methods
- To choose the appropriate method of sorting and searching
- To write program for selection, insertion and bubble sort and linear and binary searching
- To define hashing
- To understand hash table
- To use appropriate hashing technique
- To resolve collision

5.1 Sorting

Introduction

Sorting is a fundamental operation in Computer. Sorting means arranging the data in some sequence ie, in increasing order or decreasing order for numerical data or alphabetically for character data. There are many sorting algorithms available. One may choose particular algorithm, depends upon the properties of data and the operations he wants to perform on the data.

Definition

Sorting means arranging the data in some sequence i.e. increasing order or decreasing order. Sorting is divided into two groups.

- Internal sorting
- External sorting

Internal sorting means arranging the numbers in order if all data are in primary memory.

External sorting means arranging the numbers in order if data are in secondary memory also.

5.1.1 Bubble sort

In bubble sort, the array is scanned sequentially by comparing consecutive two elements and the elements are interchanged if needed. At the end of each scan (or pass) the biggest element occupies the end of array.

Algorithm

Let a be an array of n numbers. The following steps are used to arrange the values in ascending order.

Pass 1

- i. Take $a[0]$ i.e. first element. Compare it with $a[1]$ i.e. second element. If it is not in order, interchange .
- ii. Compare $a[1]$ element with $a[2]$ i.e. third element. If it is not in order, interchange.
- iii. This process is repeated till $n-2^{\text{th}}$ element is compared with $n-1^{\text{th}}$ element.
- iv. At the end of the Pass 1 the biggest element comes to $n-1$ position.

Pass 2

- Take $a[0]$ i.e. first element. Compare it with $a[1]$ i.e. second element. If it is not in order, interchange.
- Compare $a[1]$ element with $a[2]$ i.e. third element. If it is not in order, interchange.
- This process is repeated till $(n-3)^{\text{th}}$ element is compared with $(n-2)^{\text{th}}$ element.
- At the end of Pass 2 the second biggest element comes to $n-2$ position

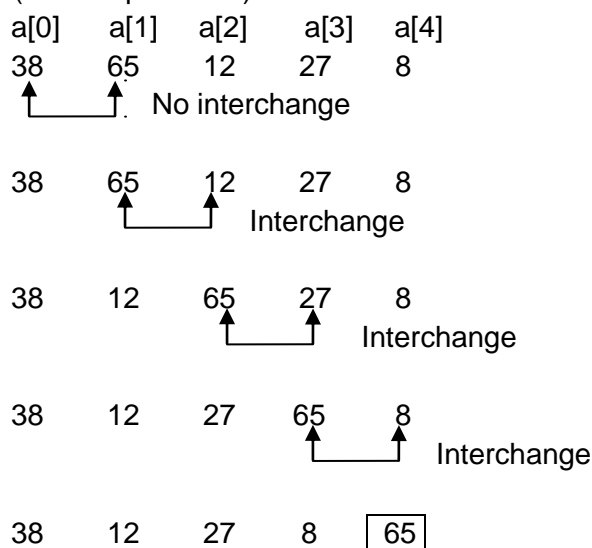
Pass n-1

- Take $a[0]$ i.e. first element. Compare it with $a[1]$ i.e. second element. If it is not in order, interchange.
- At the end of the Pass n-1, all elements will be arranged.

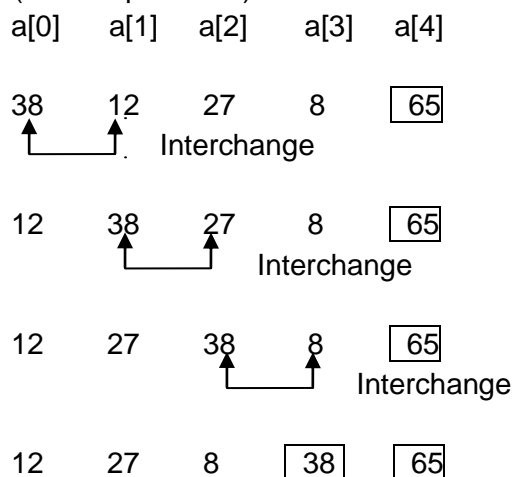
Example

Consider the following array with 5 elements (5 elements need 4 Pass)

Pass 1 (4 Comparisons)



Pass 2 (3 Comparisons)



Pass 3 (2 Comparisons)

a[0]	a[1]	a[2]	a[3]	a[4]
12	27	8	38	65

↑ ↑ No Interchange

12	27	8	38	65
----	----	---	----	----

 ↑ ↑ Interchange

12	8	27	38	65
----	---	----	----	----

Pass 4 (1 Comparison)

a[0]	a[1]	a[2]	a[3]	a[4]
12	8	27	38	65

↑ ↑ Interchange

8	12	27	38	65
---	----	----	----	----

At the end of Pass 4, all elements are arranged in ascending order.

C Program:

```
#include<stdio.h>
#include <conio.h>
void main()
{
    int a[100], n,i,j,temp;
    clrscr();
    printf("How many elements in array");
    scanf("%d",&n);
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    for(i=1;i<=n-1;i++) // n-1 scan for n data
    {
        for(j=0;j< n-i; j++)
        {
            if(a[j]>a[j+1])
            {
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
    printf("\n The Sorted array is");
    for(i=0;i<n; i++)
        printf("%d",a[i]);
    getch();
}
```

5.1.2 Selection Sort

The selection sort technique is based upon selecting the minimum or maximum value in array. Array is scanned to locate the minimum value, once it is found, it is placed in the first position of the array (position 0). The remaining elements are scanned to find the second smallest element and is placed in the second position (position 1) and so on until the array is sorted.

Algorithm

This algorithm sorts an array a with n elements.

1. Set $k = 0$
2. Find the position of minimum value of array from k^{th} location, and mark it as loc
3. Interchange $a[k]$ and $a[\text{loc}]$
4. Set $k = k+1$; Repeat step 2 & 3 until $k < n-1$
5. Exit

Example

Consider $a = \{16, 15, 2, 10, 7\}$

	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	
Pass 1	16	15	2	10	7	Interchange $a[0]$ & $a[2]$
Pass 2	2	15	16	10	7	Interchange $a[1]$ & $a[4]$
Pass 3	2	7	16	10	15	Interchange $a[2]$ & $a[3]$
Pass 4	2	7	10	16	15	Interchange $a[3]$ & $a[4]$
Pass 4	2	7	10	15	16	

C Program

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int a[100], i, j, n, min, loc;
    clrscr();
    printf("Enter number of elements\n");
    scanf("%d", &n);
    printf("Enter %d integers\n", n);
    for (i=0; i<n; i++)
        scanf("%d", &a[i]);
    for (i=0; i<n-1; i++)
    {
```



```

min = a[i];
loc = i
for (j=i+1;j<n;j++)
{
    if ( a[j] < min )
    { min = a[j];
      loc = j;
    }
}
a[loc] = a[i];
a[i] = min;
}
printf("Sorted list is \n");
for (i=0;i<n;i++)
    printf("%d\n", a[i]);
getch();
}

```

5.1.3 Insertion Sort

In this sort, the set of values are sorted by inserting values into an existing sorted array. The insertion sort algorithm, scans array 'a' from a[0] to a[n-1], inserting each element a[r] into its proper position in the previously sorted subarray a[1],a[2],....a[r-1].

Algorithm

Scan the array a from a[0] to a[n-1] and insert a[r] for r = 0 to n-1 into its proper position into the previously sorted sub array.ie

Pass 1 : a[0] is itself sorted.

Pass 2 : a[1] is inserted either before or after a[0]

Pass 3 : a[2] is inserted into its proper place such that a[0],a[1] & a[2] are in order

Pass 4 : a[3] is inserted into its proper place into the previous sorted sub array

Pass n : a[n-1] is inserted into its proper place so that all elements are in sorted form

Example

Consider following set of values to sort

	a[0]	a[1]	a[2]	a[3]	a[4]	Remarks
	38	22	12	27	8	
Pass 1	38	22	12	27	8	38 is self sorted
Sorted Subarray	←					
Pass 2	22	38	12	27	8	22 is inserted
Sorted Subarray	←					
Pass 3	12	22	38	27	8	12 is inserted
Sorted Subarray	←					
Pass 4	12	22	27	38	8	27 is inserted
Sorted Subarray	←					
Pass 5	8	12	22	27	38	8 is inserted
Sorted array	←					

C Program

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int a[100],n,,i,j,k,temp;
    clrscr();
    printf("How many number of elements \n");
    scanf ("%d",&n);
    printf("enter elements in the array \n");
    for (i=0; i<n; i++)
        scanf("%d",&a[i]);
    for (i=0;i<n;i++)
    {
        temp=a[i];
        j=i;
        while((temp< a[j-1]) && (j-1>=0))
            j--;
        for( k =i ; j<k; k--)
        {
            a[k]=a[k-1];
        }
        a[j] = temp;
    }
    printf("Sorted Array is \n");
    for(i=0;i<n;i++)
        printf("%d\n",a[i]);
}
```

```

    getch();
}

```

5.1.4 Merge Sort

Merge sort merges the pair of elements at the first and sort using any sorting method. Next it merges four elements(quadruplets) and sort. Then it merges 8 elements and sort and so on.

Algorithm

Assume the array A has n elements and merge sort uses an auxiliary array B

1. Set $L = 1$ (Initialize the number of elements in the subarrays)
2. Repeat step 3 to 4 while $L < N$
3. Merge $2 * L$ elements and sort them using any sort. (Usually insertion sort)
4. set $L = L * 2$.
5. Exit.

Example

Consider an array A with n elements $a[1] a[2] \dots a[n]$ is in memory. The merge sort algorithm sorts the data as described below.

Suppose the array a contains 14 elements as follows

66 33 40 22 55 88 60 11 80 20 50 44 70 30

Pass 1. Merge each pair of elements to obtain the following list of sorted pairs

33 66 22 40 55 88 11 60 20 80 44 50 30 70

Pass 2. Merge each pair of elements to obtain the following list of sorted quadruplets

22 33 40 66 11 55 60 88 20 44 50 80 30 70

Pass 3. Merge each pair of sorted quadruplets to obtain the following list of sorted sub arrays

11 22 33 40 55 60 66 88 20 30 44 50 70 80

Pass 3. Merge two sorted sub arrays to obtain the following single sorted array

11 20 22 30 33 40 44 50 55 60 66 70 80 88

The original array a is now sorted

After pass K, the array a will be partitioned into sorted sub arrays where each sub array except possibly the last, will contain exactly 2^K elements.

5.1.5 Bucket or Radix Sort

Radix sorting is a technique for ordering a list of positive integer values based on digit position. The values are successively ordered on digit positions(called base or radix), from right to left. This is accomplished by distributing the elements into appropriate buckets according to digits and collectively ordered it on first in first out basis (FIFO) for next digit position. Once all digit positions are examined, the list must be sorted.

Radix sort can be used to sort a list of names alphabetically. Here the base or radix is a to z (the 26 letters of the alphabet)

Algorithm

```
/* for each base in the number system */
Initialize 9 queues and name it as 0 to 9.
For i = 1 to msb ( unit - Lsb,10,100,1000 digit position)
    For j = 0 to n-1 (all Array values)
        Get x = the ith component in the jth element and Add a[j] into queue x
    While queue is not empty
        Remove Queue elements and add it to array
    Repeat loop with index i
```

Example

Assume the data are : 624 852 426 987 269 146 415 301 730 78 593.
Here the msb position is 100. Hence

During pass 1, the ones or unit place digits are ordered.

During pass 2, the tens place digits are ordered, retaining the relative positions of values set by the earlier pass.

On pass 3 the hundreds place digits are ordered, again retaining the previous relative ordering

After three passes the result is an ordered list.

Sequence of values in each Queue during radix sort

Queue	Pass 1	Pass 2	Pass 3
0	730	301	78
1	301	415	146
2	852	624, 426	269
3	593	730	301
4	624	146	415, 426
5	415	852	593
6	426, 146	269	624
7	987	78	730
8	78	987	852
9	269	593	987

Collect all the data Queue wise. We get the following ordered list.

78 146 269 301 415 426 593 624 730 652 987

5.1.6 Shell sort

The shell sort divides the array into several sub arrays by picking every h_i th element as part of sub array. If initially $h_i = 3$, then array is divided into 3 sub arrays. If $h_i = 5$, then array is divided into 5 sub arrays. The sub arrays are sorted separately using any sorting technique.

Algorithm

Step 1 – Initialize the value of h

Step 2 – Divide the list into smaller sub-list of equal interval h

Step 3 – Sort these sub-lists using **insertion sort**

Step 3 – Repeat until complete list is sorted

Example

The following example sorts the array using $h = 5$

The given array	10	8	6	20	4	3	22	1	0	15	16
Data before 5 - sort	10	8	6	20	4	3	22	1	0	15	16
5 Subarrays before sorting	10	-	-	-	-	3	-	-	-	-	16
		8	-	-	-	-	22	-	-	-	-
			6	-	-	-	-	1	-	-	-
				20	-	-	-	-	0	-	-
					4	-	-	-	-	15	-
5 Subarrays after sorting	3	-	-	-	-	10	-	-	-	-	16
		8	-	-	-	-	22	-	-	-	-
			1	-	-	-	-	6	-	-	-
				0	-	-	-	-	20	-	-
					4	-	-	-	-	15	-
Data after 5 - sort & before 3 sort	3	8	1	0	4	10	22	6	20	15	16
3 Subarrays before sorting	3	-	-	0	-	-	22	-	-	15	-
		8	-	-	4	-	-	6	-	-	16
			1	-	-	10	-	-	20	-	-
3 Subarrays after sorting	0	-	-	3	-	-	15	-	-	22	-
		4	-	-	6	-	-	8	-	-	16
			1	-	-	10	-	-	20	-	-
Data after 3 - sort & before 1 sort	0	4	1	3	6	10	15	8	20	22	16
Data after 1 sort	0	1	3	4	6	8	10	15	16	20	22

5.1.7 Quick Sort

The quick sort algorithm works by partitioning the array into subarrays.

Principle:

- Choose any number in the array and name it as partition element P . For simplicity take first element $a[0]$ as the partition element.
- With respect to the value of P (i.e. $a[0]$) divide the array into two partitions such that the number which are less than P are placed in the left side of P and the numbers which are greater than P are placed in the right side of P .

Suppose if the partition element is placed in the j^{th} position, the following conditions are satisfied.

- Each number in the position 0 through $j - 1$ is less than or equal to P (i.e. $a[0]$).
- Each number in the position $j + 1$ through $n - 1$ is greater than or equal to P .
- The partition element P fixes its sorted position.

The above steps are repeated in the left partition $a[0]$ through $a[j-1]$ and right partition $a[j+1]$ through $a[n-1]$.

Algorithm:

- Initialize two pointers low and high. At the beginning $\text{low} = 0$ and $\text{high} = n-1$.
- Fix the partition element $P = a[0]$.
- Scan the array from left to right and compare P with 2^{nd} element i.e. $a[1]$. If it is greater than P , stop scanning and keep the location of the higher element in low. Else, compare next element and so on.
- Scan the array from right to left and compare P with the last element i.e. $a[n-1]$. If it is less than or equal to P , stop scanning and keep the location of the smaller element in high. Else compare next element and so on.
- Check the value of low and high. If $\text{low} < \text{high}$, then interchange the low position data and high position data and repeat steps (iii) and (iv).
- Else if $\text{low} \geq \text{high}$, interchange the partition element and high position element. Now, the partition element P fixes its proper location.
- Every element left of P is less than or equal to P and every element right of P is greater than P .
- The above steps (ii) to (vi) are repeated separately for left and right partition sub arrays.

Example

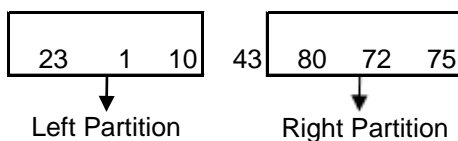
The following example illustrates this. Fix Partition Element $P = a[0]$ ie 43

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$	Scan Direction	Status	low	high
43	72	10	23	80	1	75	Left to Right	72>43 . So Stop Fix low = 1	1	
43	72	10	23	80	1	75	Right to left	75>43 Compare next	1	
43	72	10	23	80	1	75	Right to left	1<43. So Stop Fix high = 5	1	5

Check low and high ie. $1 < 5$. So Interchange $a[1]$ and $a[5]$ ie 1 and 72

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	Scan Direction	Status	low	high
43	1	10	23	80	72	75	Left to Right	1<43 Compare next		
43	1	10	23	80	72	75	Left to Right	10<43 Compare next		
43	1	10	23	80	72	75	Left to Right	23<43 Compare next		
43	1	10	23	80	72	75	Left to Right	80>43 So Stop Fix low = 4	4	
43	1	10	23	80	72	75	Right to left	75>43 Compare next	4	
43	1	10	23	80	72	75	Right to left	72>43 Compare next	4	
43	1	10	23	80	72	75	Right to left	80>43 Compare next	4	
43	1	10	23	80	72	75	Right to left	23<43 So Stop fix high = 3	4	3

Check low and high ie 4>3 So, Interchange P and a[high] ie 43 and 23



Apply the above procedure to left partition and right partition. Finally we get the data in sorted form like below.

1 10 23 43 72 75 80

5.2 Searching

A fundamental operation of a computer is to store huge volume of information and retrieve them as quickly as possible. Searching plays an important role in information retrieval. Searching methods are governed by how data and how much data are stored in computer.

Definition

Searching means finding an element in array or locating the position of element in array.

Types of Searching

Searching is divided into two categories:

Linear or Sequential Search

Binary search. .

5.2.1 Sequential or Linear search

Definition

In Linear search, the element is searched from 0th element of array to last element in sequential order.

In linear or sequential search, the searching element is compared with each element of an array one by one ie sequentially, to check whether the given element is found or not. If the element is found then Search is successful. A search will be unsuccessful if all the elements are compared and the desired element is not found.

Algorithm

Assume

The array name is A

The element to be searched is x

The array size is n

- i. Compare x with A[0]. If equals print "Success" and print the position also.
Else Compare x with A[1]. If equals print "Success" and print the position also.
Else repeat the process upto the last element ie A[n-1].
- ii. If no array value matches, print "Search Fail"

Example

Consider an array having 6 elements. Search the element 25.

Given Array Data : 12 16 4 25 69 43

Search Data :25

Pass 1 : 12 16 4 25 69 43 : compare 12 with 25, no match

Pass 2 : 12 16 4 25 69 43 : compare 16 with 25, no match

Pass 2 : 12 16 4 25 69 43 : compare 4 with 25, no match

Pass 2 : 12 16 4 25 69 43 : compare 25 with 25, Equals => Print Success
and position = 3

C program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[100],n,i,loc = -1;
    clrscr()
    printf("\n enter the number of element:");
    scanf("%d",&n);
    printf("enter the numbers:\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    printf("\n enter the number to be searched:");
    scanf("%d",&item);
    for(i=0;i<n;i++)
    {
        if(item==a[i])
        {
            loc=i;
            printf("\n %d is found in array position %d\n",item,loc);
```



```

        break;
    }
}
if(loc == -1)
    printf("\n item does not exist");

getch();
}

```

5.2.2 Binary search:-

Definition

In Binary search, the searching starts by dividing the sorted array into two halves using mid value and determine which subarray is to be used to find the element.

Algorithm

Binary search is suitable if the array is sorted. The logic behind this technique is given below.

1. Assume a be the sorted array ; lb = 0 and ub = n-1 and x be the data to be searched.
2. Find the middle element of the array using

$$\text{mid} = \left\lfloor \frac{\text{lb} + \text{ub}}{2} \right\rfloor$$

3. Compare a[mid] with x.

There are three cases:

- a. if it is equal search is successful and mid value is the position of x .
 - b. If x is less than mid value, then ub = mid – 1 ie search only the first half of the array.
 - c. If x is greater than mid value, then lb = mid+1 ie search only the second half of the array.
 - d. Repeat step 2 and 3 while lb<=ub
4. If lb>ub, then search failed.

Example

Consider a sorted array a with 7 elements.

9	12	24	30	36	45	70
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

The steps to search x = 45 using binary search with lb = 0 and ub = 6 are

Pass 1

Step 1: mid=int((lb+ub)/2)
 =int((0+6)/2)
 =int (6/2)
 =3

Step2: a[mid]=a[3]=30

Compare x and a [mid]. 45>30. Search is to be continued in the second half. So, change lb as lb = mid+1 = 3+1 = 4 and ub = 6

Pass 2

Step 1: $\text{mid} = \text{int}((\text{lb} + \text{ub})/2)$

$= \text{int}((4+6)/2)$

$= \text{int}(10/2)$

$= 5$

Step 2: $a[\text{mid}] = a[5] = 45$

Compare x and $a[\text{mid}]$. $45 = 45$. So search success. Array Location = 5

C program

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[100], i, mid, lb,ub, n, x;
    clrscr();
    printf("How many elements");
    scanf("%d",&n);
    printf("Enter the elements of the array \n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    printf("Enter the element to be searched \n");
    scanf("%d",&x);
    lb=0;
    ub=n-1;
    while (lb <= ub)
    {
        mid=((lb+ub)/2);
        if (x== a[mid])
        {
            printf("search is successful and position of array = %d \n",mid);
            break;
        }
        else
            if (x < a[mid])
                ub = mid-1;
            else
                lb = mid+1;
    }
    if (lb>ub)
        printf("search is not successful\n");
    getch();
}
```

5.3 HASHING

Introduction

In Sequential search, the table that stores the elements is searched successively. In Binary search, the table that stores the element is divided successively into two halves and determine which half is to be checked.

There is another technique in which, the searching calculates the position of the key in the table. It is a one to one correspondence between a key value and an index in the table to place the key value. The technique is called hashing. Shortly saying, Hashing is a technique to convert a range of key values into a range of indices (plural of index) of an array.

5.3.1 Hashing & Hash table

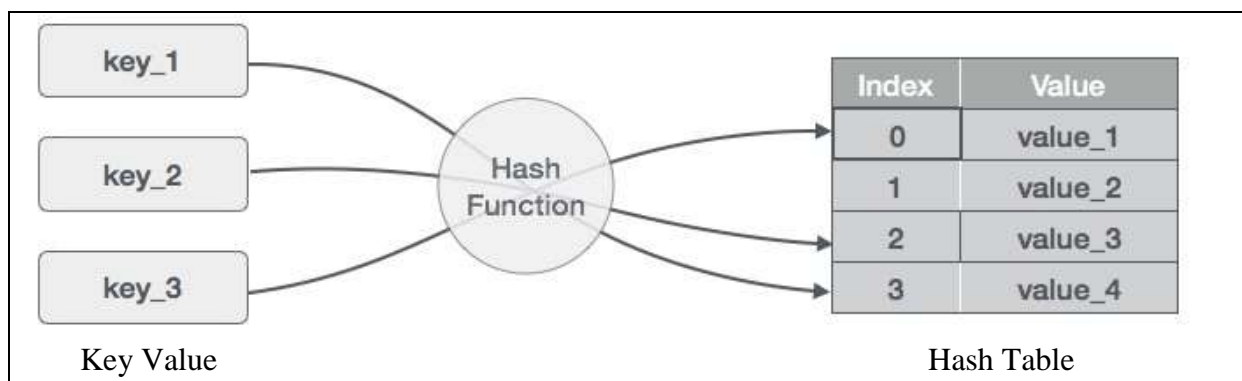
Definition

Hashing is a technique used to convert a range of key values into a range of indices of an array.

A hash table is a structure having two parts. They are

- (i) An array to store the records(data).
- (ii) A mapping function called **hash function(h)** that calculates the index value for the given key values.

The hashing technique and hash table can be expressed as in the following figure



The key value to index value is found out using a hash function $H:K \rightarrow I$ in such a way that the function H should be very easy and quick to compute.

Example

Consider a hash table of size 10 whose indices are 0,1,2,3,...,9. Assume a set of key values 10,19,35,43,62,59,31,49,77,33. Let us assume the hash function H is as stated below

- Add the two digits in the key.
- Take the digit at the unit place of the result as the index; ignore the digit at the tenth place if any.

Using this hash function, the mappings from key value to indices and to hash table are shown in following figure.

Before Hashing	
Key Value K	Index I
10	1
19	0
35	8
43	7
62	8
59	4
31	4
49	3
77	4
33	6

$H:K \rightarrow I$

After Hashing	
Index I	Key Value K
0	19
1	10
2	
3	49
4	59,31,77
5	
6	33
7	43
8	35,62
9	

Hash Table

5.3.2 Hash function

Hash function is defined as a function that calculates the index value or hash value for the array using the given key value. The general form is

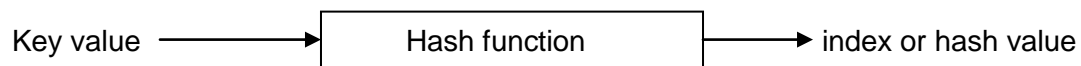
$$h(k_i) = \text{index}$$

Where,

$h \rightarrow$ hash function

$k_i \rightarrow$ key value. $i=1,2,\dots$

The figure given below shows the logical structure of hash function.



The following are the different types of hash functions. They are,

- (i) Division method hash function
- (ii) Mid square method hash function
- (iii) Folding method hash function

i. Division method hash function

In this method the hash values are generated by finding the remainder of the key(k) divided by a prime number(m). The general form is

$$h(k) = k \bmod m$$

$$= \text{hash value}$$

Where

$k \rightarrow$ key value ranges from k_1, k_2, \dots, k_n

$m \rightarrow$ prime number Close to size of index range $0, 1, 2, \dots, m-1$.

The hash function calculates an integer index value in the range 0 to $m-1$.

Example

Let there are 80 employees record each with a four digit unique employee number.

Let the index value ranges from 0 to 99.

Let $m=97$ [it is a prime number close to 99]

For $k=3205$, $h(3205) = 3205 \bmod 97 = 4$. That is the employee with employee number 3205 is stored in index 4.

ii. Mid square method hash function

In this method the hash values are generated by following the steps given below.

- Square the key value k . that is find k^2 .
- Choose the digit positions by counting either from left or right.
- Delete the digits from both sides of k^2 except the chosen digit position. This gives the hash value.

The general form is

$$h(k) = k^2 = I(\text{hash value})$$

Where,

$I \rightarrow$ the index value or hash value after deleting the digits from the right and left sides of k^2 .

Example

Let there are 80 employees record each with a four digit unique employee number.

Let the index value ranges from 0 to 99.

Choose the 4th and 5th digit from right of k^2 as I .

For $k = 3205$

$$h(3205) = k^2 = 10272025$$

$$\therefore h(3205) = 72$$

For $k = 7148$

$$h(7148) = k^2 = 51093904$$

$$\therefore h(7148) = 93$$

iii. Folding method hash function

In this method hash values are generated by following the steps given below.

- Divide the key into the equal parts k_1, k_2, \dots, k_r having equal number of digits as in the index. The last part can have less number of digits.
- Add all the parts excluding the last part having less number of digits. This gives the hash value.

The general form is

$$h(k) = k_1 + k_2 + \dots + k_r$$

Example

Let there are 80 employees record each with a four digit unique employee number.

Let the index value ranges from 0 to 99.

For $k = 3205$

$$h(3205) = 32 + 05 = 37$$

For $k = 2345$

$$h(2345) = 23 + 45 = 68$$

5.3.3 Collision

Collision is defined as a situation where two different key values hash (map) to the same memory slot (table index) in the hash table.

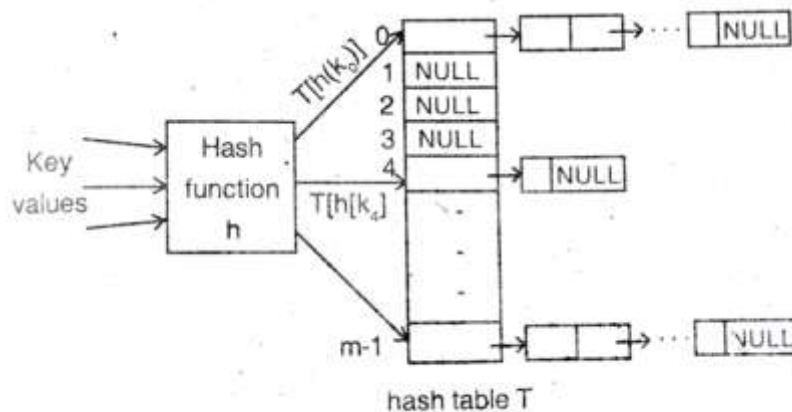
Collision resolution

The following are the two important collision resolution methods. They are,

- Separate chaining
- Open addressing

5.3.3.1 Separate chaining

This is the simplest collision resolution method. In this method, the key values that hash to the same slot in the hash table are placed in a linked list as shown below.



As shown in the above figure, the slots in the hash table T will not store data elements. But the slots point to the linked list. For example, the data element with key k is stored in the linked list pointed by array index value $T[h(k)]$

Example

Let the size of the array is 10 and the hash function $h(k) = k \bmod 10$.

Let the keys are $k = 75, 66, 42, 192, 91, 40, 49, 87, 67, 16, 417, 130, 372, 227$.

Therefore,

$$h(75) = 75 \bmod 10 = 5$$

$$h(66) = 66 \bmod 10 = 6$$

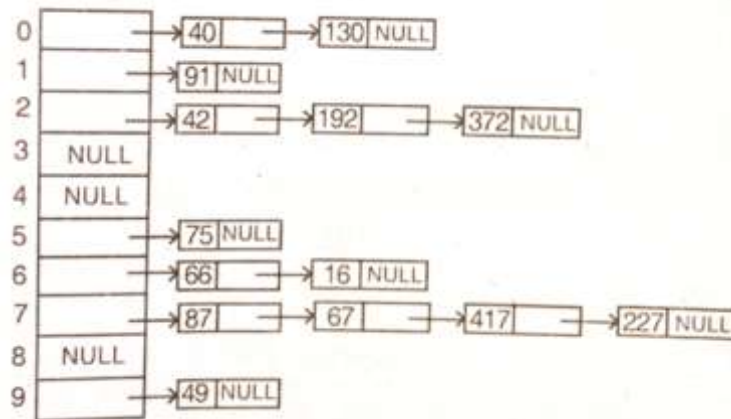
$$h(42) = 42 \bmod 10 = 2$$

$$h(192) = 192 \bmod 10 = 2$$

$$h(91) = 91 \bmod 10 = 1$$

$$h(227) = 227 \bmod 10 = 7$$

The figure given below shows the hash table for the given keys.



5.3.3.1 Open addressing

In this collision resolution method, the key values that hash to the same slot in the hash table are placed in the next available position in the array. The general principle is

If the hash table slot for $h(k)$ is already occupied by another key value, a sequence called **probe sequence** is computed by using the key k . That means when two key values are hashed to the same table slot, the probe sequence is successively examined or probed until an empty hash table slot is found.

Example

Let the size of the array is 10 and the hash function $h(k) = k \bmod 10$.

Let the keys are $k = 7000, 7397, 6395, 7667$.

Therefore,

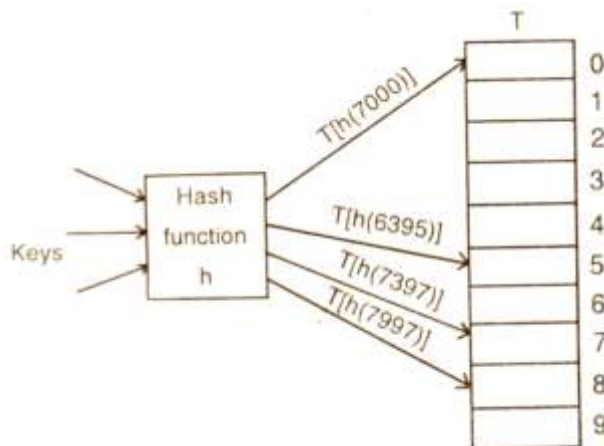
$$h(7000) = 7000 \bmod 10 = 0$$

$$h(7397) = 7397 \bmod 10 = 7$$

$$h(6395) = 6395 \bmod 10 = 5$$

$$h(7997) = 7997 \bmod 10 = 7$$

In the given keys, two keys 7367 and 7997 hash to the same slot 7. Therefore, the key 7997 which hashes to the slot 7 which is already occupied by key 7397 is placed in the next available slot 8 as shown in the figure.



Summary

- ✓ Sorting means arranging the data in increasing order or decreasing order.
- ✓ Sorting is divided into two groups - Internal sorting & External sorting.
- ✓ In bubble sort, the array is scanned sequentially by comparing consecutive two elements and the elements are interchanged if needed.
- ✓ The selection sort technique is based upon selecting the minimum / maximum value in array
- ✓ In Insertion sort, the set of values are sorted by inserting values into an existing sorted file
- ✓ The quick sort algorithm works by partitioning the array into sub arrays.
- ✓ In Radix sort the values are successively ordered on digit positions(called base or radix), from right to left.
- ✓ In Merge sort, merge pair of elements to form sorted sub array
- ✓ The shell sort divides the array into several sub arrays by picking every h_i th element as part of sub array.
- ✓ Searching means finding an element is present or not in the array.
- ✓ Searching is divided into two categories: Linear or Sequential Search & Binary search.
- ✓ In Linear search, the element is searched from 0th element of array to last element in sequential order.
- ✓ In Binary search, the searching starts by dividing the sorted array into two halves using mid value and determine which sub array is used to find the element.
- ✓ Hashing is a technique to convert a range of key values into a range of indices of an array.
- ✓ Hash function is defined as a function that calculates the index value or hash value
- ✓ The different types of hash functions are
 - Division method hash function
 - Mid square method hash function
 - Folding method hash function
- ✓ Collision occurs when two different key value hash to the same memory slot in the hash table.
- ✓ In separate chaining method , the key values that hash to the same slot in the hash table are placed in a linked list
- ✓ In Open addressing method, the key values that hash to the same slot in the hash table are placed in the next available position in the array.

Review Questions

Part A (2 mark Questions)

1. Define sorting. What are the two types of sorting?
2. What is the principle used in bubble sort?
3. What is the principle used in selection sort?
4. What is the logic behind insertion sort?
5. What is the principle used in Merge sort?
6. What is the principle used in Shell sort?
7. What is the logic behind Radix sort?
8. What is the principle used in Quick sort?
9. Define searching. What are the two types of searching?
10. Define Linear Search.
11. Define Binary search.
12. Define Hashing.
13. What is called hash table?
14. What is called collision?
15. What is meant by hash function?
16. What are the two methods of Collision Resolution?
17. What is the principle used in separate chaining?
18. What is called open addressing?

Part B (3 Mark Questions)

1. Write the algorithm for bubble sort.
2. Write the algorithm for Insertion sort.
3. Write the algorithm for Selection sort.
4. Write the algorithm for Merge sort.
5. Write the algorithm for Radix sort.
6. Write the algorithm for Shell sort.
7. Write the algorithm for Quick sort.
8. Write the algorithm for linear or sequential search
9. Write the algorithm for binary search.
10. What is meant by hashing.
11. Write any one method of hash function.
12. Write short notes on any one collision resolution method..

Part c (5 or 10 Mark Questions)

1. Explain bubble sort by taking 8 data in array.

2. Write a program to implement bubble sort.
3. Explain with example Selection sort.
4. Write a program to implement selection sort.
5. Explain with example Insertion sort.
6. Write a program to implement Insertion sort.
7. Explain with example Merge sort.
8. Explain with example Radix sort.
9. Explain with example Shell sort.
10. Explain with example Quick sort.
11. Explain linear searching method with suitable example.
12. Explain Binary searching method with suitable example.
13. Explain hashing.
14. Explain Hashing function.
15. Explain any one Collision Resolution method in detail.
16. What is open addressing?. Explain with example.
17. What is called separate chaining?. When will you use it?. Explain.