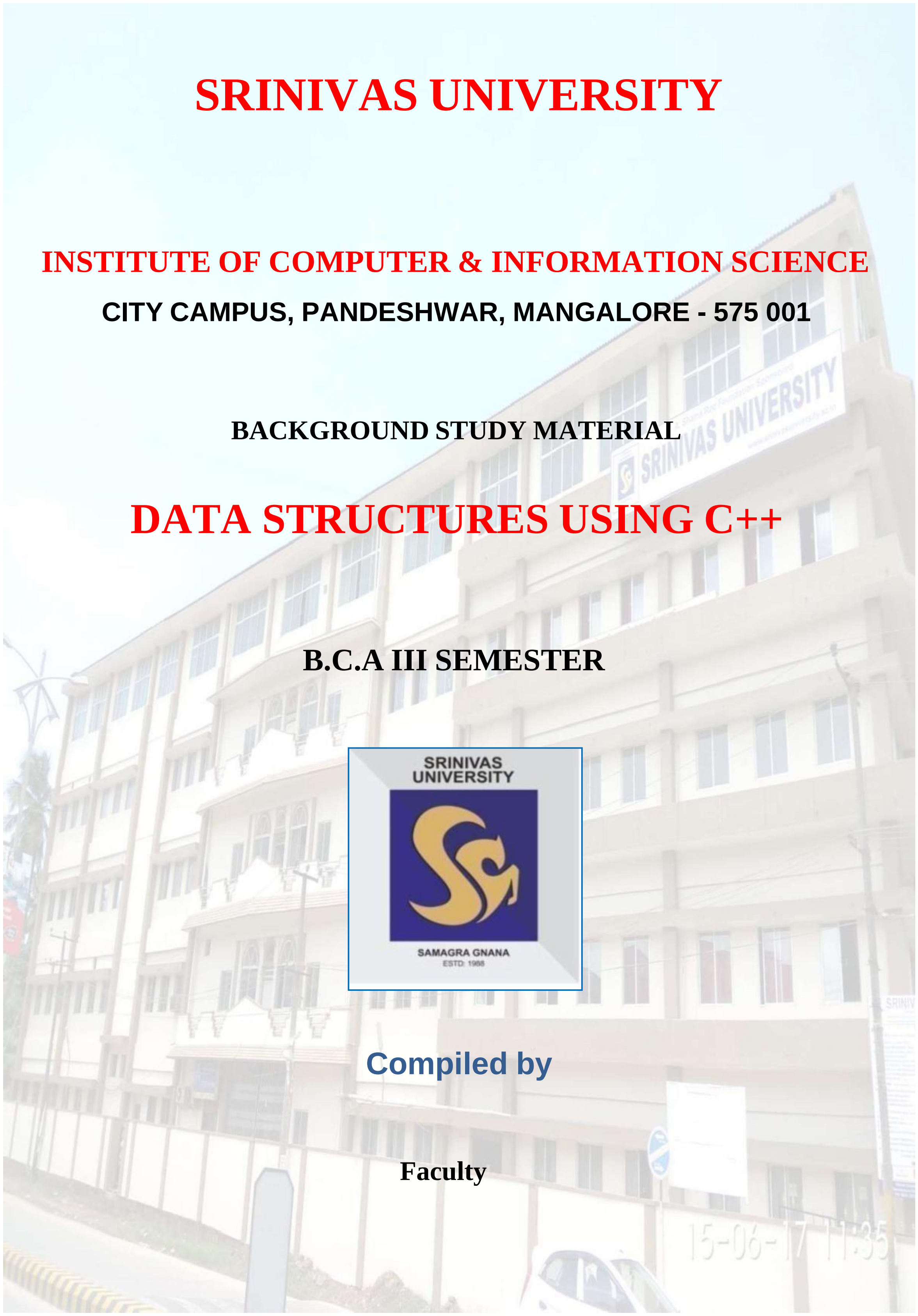


SRINIVAS UNIVERSITY

INSTITUTE OF COMPUTER & INFORMATION SCIENCE

CITY CAMPUS, PANDESHWAR, MANGALORE - 575 001

BACKGROUND STUDY MATERIAL

DATA STRUCTURES USING C++

B.C.A III SEMESTER

Compiled by

Faculty

CONTENTS

MODULE - III

Chapter 6 - Dynamic memory management

Memory allocation and de-allocation functions – malloc, calloc, realloc and free

Chapter 7 - Linked list

Basic Concepts: Definition
Representation of linked list
Types of linked lists –Singly-linked list, Doubly liked list, Header linked list
Circular linked list
Representation of Linked list in Memory
Operations on Singly linked lists
Traversing
Searching
Insertion
Deletion
Memory allocation
Garbage collection

MODULE - IV

Chapter 8 – Trees

Definition
Tree terminologies – node, root node, parent node, ancestors of a node, siblings, terminal, non-terminal nodes, degree of a node, level, edge, path, depth

Chapter 9 - Binary trees

Type of binary trees - strict binary tree, complete binary tree, binary search tree
Array representation of binary tree
Traversal of binary tree - preorder, inorder and postorder traversal

MODULE - V

SYLLABUS

Paper:21CAC-7
Theory/Week: 3 Hours
Credits: 3

Data Structures Using C++

Hours: 30
IA: 50
Exam: 50

Module - I 6 Hours

Introduction to data structures: Introduction, Basic terminology; Elementary Data organization, Structures, Data Structure Operations
Introduction to Algorithms: Introduction, Algorithmic notations, Control structure.
Recursion: Definition; Recursion Technique Examples –Factorial, Fibonacci sequence, Towersof Hanoi

Module - II 6 Hours

Stacks: Basic Concepts –Definition and Representation of stacks- Array representation of stacks, Linkedrepresentation of stacks, Operations on stacks, Applications of stacks, Infix, postfix and prefix notations Conversion from infix to postfix using stack, Evaluation of postfix expression using stack, Application of stack in function calls.
Queues: Basic Concepts – Definition and Representation of queues- Array representation of Queues, Linked representation of Queues, Types of queues - Simple queues, Circular queues, Double ended queues, Priority queues, Operations on queues

Module - III 6 Hours

Dynamic memory management: Memory allocation and de-allocation functions - malloc, calloc, realloc and free.
Linked list: Basic Concepts – Definition and Representation of linked list, Types of linked lists – Singly linked list, Doubly liked list, Header linked list, Circular linked list, Representationof Linked list in Memory; Operations on Singly linked lists– Traversing, Searching, Insertion, Deletion, Memory allocation, Garbage collection

Module - IV 6 Hours

Trees: Definition, Tree terminologies –node, root node, parent node, ancestors of a node, siblings, terminal & non-terminal nodes, degree of a node, level, edge, path, depth
Binary trees: Type of binary trees - strict binary tree, complete binary tree, binary search tree, Array representation of binary tree, Traversal of binary tree- preorder,inorder and postorder traversal

Module - V 6 Hours

Sorting: Selection sort, Bubble sort, Quick sort, Insertion sort, Merge sort
Searching : Definition, Sequential Search, Binary search
Graphs: Terminologies, Matrix representation of graphs; Traversal: Breadth First Search and Depth first search

Syllabus & Teaching Plan – III Semester BCA - Srinivas University
DATA STRUCTURES USING C++
Paper: 21CAC-7

TEACHING PLAN

MODULE - 1 6 hrs

Session 1 Introduction, Basic terminology, Elementary Data Organization
Session 2 Data Structures, Data Structure Operations
Session 3 Introduction to Algorithms: Introduction, Algorithmic notations
Session 4 Control structure
Session 5 Recursion: Definition
Session 6 Examples –Factorial, Fibonacci sequence, Towers of Hanoi

MODULE - 2 6 hrs

Session 7 Stacks: Basic Concepts –Definition and Representation of stacks- Array representation of stacks
Session 8 Linked representation of stacks, Operations on stacks
Session 9 Arithmetic expressions - Infix, postfix and prefix notations, Conversion from infix to postfix using stack, Evaluation of postfix expression using stack, and Application of stack in function calls
Session 10 Queue: Introduction
Session 11 Array representation Circular queue, Linked list representation of queue
Session 12 Types of queues - Simple queues, Circular queues, Double ended queues, Priority queues, Application on queues

MODULE - 3 6 hrs

Session 13 Dynamic memory management: Memory allocation and de-allocation functions - malloc, calloc, realloc and free
Session 14 Linked list: Basic Concepts Definition and Representation of linked list
Session 15 Types of linked lists – Singly linked list, Doubly liked list, Header linked list
Session 16 Circular linked list, Representation of Linked list in Memory
Session 17 Operations on Singly linked lists– Traversing, Searching, Insertion, Deletion
Session 18 Memory allocation, Garbage collection

MODULE - 4 6 hrs

Session 19 Trees: Definition, Tree terminologies –node, root node, parent node, ancestors of a node, siblings, terminal node
Session 20 non-terminal nodes, degree of a node, level, edge, path, depth
Session 21 Binary trees: Type of binary trees strict binary tree, complete binary tree
Session 22 Binary search tree, Array representation of binary tree
Session 23 Traversal of binary tree - preorder
Session 24 inorder and postorder traversal

MODULE - 5 6 hrs

Session 25 Sorting: Selection sort, Bubble sort
Session 26 Quick sort, Insertion sort
Session 27 Merge sort
Session 28 Searching: Definition, Sequential Search, Binary search
Session 29 Graphs: Terminologies, Matrix representation of graphs
Session 30 Breadth First Search andDepth first search

Text Book:

1. Seymour Lipshutz, Schaum 's Outline: Data structures with C, Tata McGraw Hill 2011.
2. Yedidyahlangsun, Moshe J. Augustein, Tennenbaum, data structure using C & C++, Prentice Hall of India ltd.
3. J.P.Trembly and Sorenson, An introduction to Data structures with applications, 2nd edition, McGraw Hill 2001.

Marks Distribution for the Under-graduate students of Srinivas University is as follows:
(Examination should be conducted for 30 marks which should include 25 marks for Descriptive type questions and 5 marks for multiple choice questions each carrying one mark)

Internal Assessment	**50 Marks**
1st Internal Examination	15 Marks
2nd Internal Examination	15 Marks
Assignment & Presentation	10 Marks
Attendance & Class Behaviour	10 Marks
Final Semester End Exams	**50 Marks**

- The question paper shall contain **Part – A** & **Part – B**.
- **Part – A** shall contain 12 multiple choice questions where in students have to answer any 10 questions carrying one mark each. **10 marks**
- **Part – B** shall contains 2 questions from each unit wherein students have to answer any onequestion carrying 8 marks. There will be 5 units carrying 8 x 5 marks **40 marks**

The students should get minimum 25 marks in semester end examinations and minimum 25 marks in the internal assessment in order to pass in the particular subject.

-

UNIT I
CHAPTER – 1. INTRODUCTION AND OVERVIEW

Introduction

Definition of data structure as follows

- A data structure is a method of storing data in a computer, so that it can be used efficiently.
- Data may be organized in many different ways; the logical or mathematical model of a particular organization of data is called a data structure.

Data structures mainly deals with

- The study of how the data is organized in the memory.
- How efficiently, data can be stored in the memory.
- How efficiently, the data can be retrieved and manipulated.
- The possible way in which different data items are logically related.

Advantages

1. Data structures allow information storage on hard disks.
2. Provides means for management of large dataset such as databases or internet indexing services.
3. Are necessary for design of efficient algorithms.
4. Allows safe storage of information on a computer. The information is then available for later use and can be used by multiple programs. Additionally, the information is secures and cannot be lost (especially if it is stored on magnetic tapes).
5. Allows the data use and processing on a software system.
6. Allows easier processing ofdata.
7. Using internet, we can access the data anytime from any connected machine (computer, laptop, tablet, phone, etc.).

Disadvantages

1. Only advanced users can make changes to data structures
2. Any problem involving data structure will need an expert's help, i.e. basic users cannot help themselves.

Practical use of data structures

Data structures are used for Fast data lookup, processing scheduling, Dictionary Data indexing, IP addressing, Parsers/Compilers, Dynamic memory allocation, Directory traversal, Web crawling, Organizing file hierarchy, etc.

Basic terminology: elementary data organization

Data - Data are simply values or sets of values.
Data item - Refers to a single unit of values.
Group items - Data items are divided into sub items called group items; those that are not are called elementary items.

For example, an employee's name may be divided into three sub items- first name, middle name, and last name. But the phone number would be treated as a single item.

An entity is something that has certain attributes or properties which may be assigned values. Entities with similar attributes form an entity set. Each attribute of an entity set has a range of values.
For example,

Attributes	Name	Age	Sex / Gender
Values	Mangu	24	F

Information

Information is sometimes used for data with given attributes or meaningful or processed data.
Collection of data is frequently organized into fields, records and files. A field is a single elementary unit of information representing an attribute of an entity. A record is a collection of filed values of a given entity. A file is the collection of records of the entities in a given entity set.
Each record in a file may contain many field items, but the value in a certain field may uniquely determine the record in the file. Such a field is called the primarykey.

Records may also be classified according to length. A file can have fixed length records or variable length records. In a fixed length records, all the records contain the same data items with the same amount of space assigned to each data item. In a variable length records, file records may contain different lengths.

Classification of data structures

Data structures are generally classified into primitive and non-primitive data structures.

Primitive data structures

Primitive data structures are the data structure that can be manipulated directly by the machine instructions. For example, Basic data types such as integer, real, character and Boolean are known as primitive data structures. These data types consist of characters that cannot be divided and hence they are also called simple data types.

Non-primitive data structures

Non-primitive data structures are the data structure that cannot be manipulated directly by machine instructions. For example, the simplest example of non-primitive data structure is the processing of complex numbers. Linked lists, stacks, queues, trees and graphs are examples of non-primitive data structures.

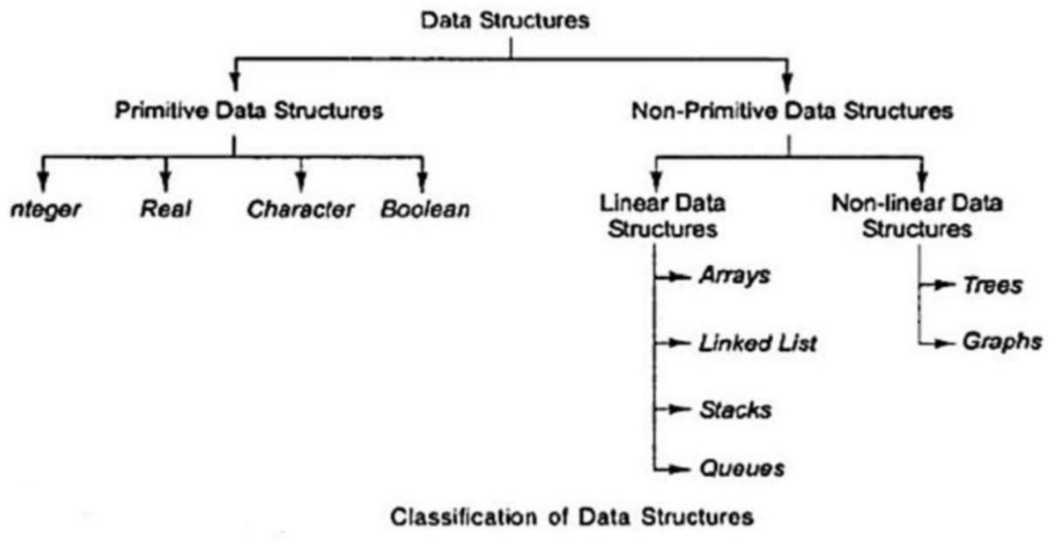
Based on the structure and arrangement of data, non-primitive data structures are further classified into linear and nonlinear data structures.

Linear Data Structure

A data structure is said to be linear if its elements form a sequence or a linear list. In linear data structures, the data is arranged in a linear fashion. They need not be stored sequentially in the memory. Arrays, linked lists, stacks and queues are examples of linear data structures.

Non Linear Data Structure

Similarly, a data structure is said to be nonlinear if the data is not arranged in sequence. The insertion and deletion of data is therefore not possible in a linear fashion. Trees and graphs areexamples.



Classification of Data Structures

Some of the data structures are Arrays, Linked Lists, Trees, Stack, Queue, Graphs.

Arrays

C++ provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the sametype.

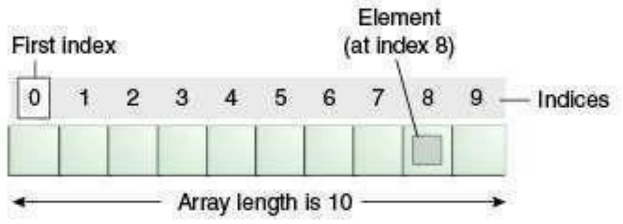
Instead of declaring individual variables, such as number 0, number 1, ..., and number 9, you declare one array variable such as numbers and use numbers [0], numbers [1], and ..., numbers [9] to represent individual variables.

A specific element in an array is accessed by an index. All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

The simplest type of data structure is a linear or one dimensional array. A linear array is a list of finite number n of similar data elements referenced respectively by a set of n consecutive numbers.

Suppose A is an array, then th e elements of A are denoted by A(1), A(2),…A(n) or A[1], A[2], A[3], …A[n] or a1, a2, a3,…an . Linear arrays are called one dimensional arrays because each element in such an array is referenced by one subscript.

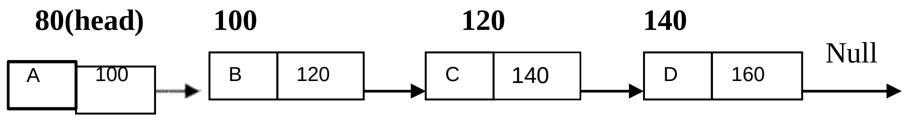
A two dimensional array is a collection of similar data element is referenced by two scripts.



Linked List

A linked list is a non-sequential collection of data items. For every data item in the linked list, there is an associated pointer that gives the memory location of the next data item in the linked list.

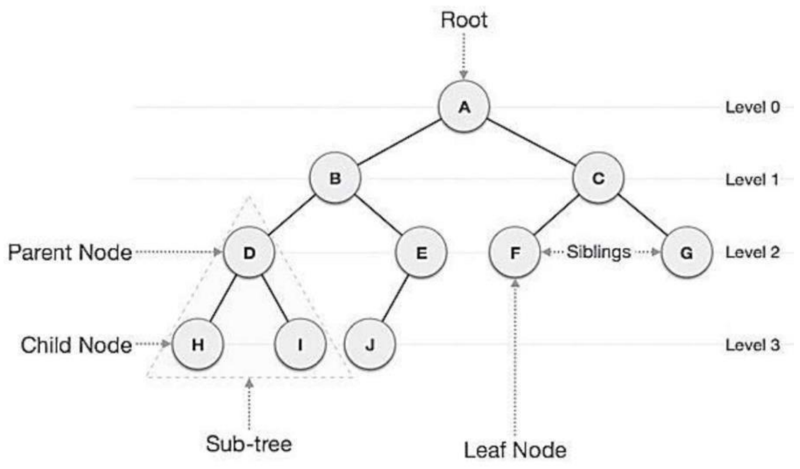
The data items in the linked list are not in a consecutive memory locations. But they may be anywhere in memory. However, accessing of these items is easier as each data item contained within itself the address of the next dataitem.



Trees

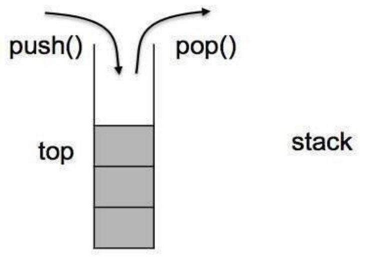
A tree is a non-terminal data structure in which items are arranged in a sorted sequence. It is used to represent hierarchical relationship existing among several data items. The graph theoretic definition of a tree is: It is a finite set of one or more data items (nodes) such that

- There is a special data item called the root of the tree.
- And its remaining data items are partitioned into number of mutually exclusive subsets each of which is itself a tree. They are called sub trees.



Stack

A stack, also called a last-in first-out (LIFO) system, is a linear list in which insertions and deletions can take place only at one end, called top.



Queue

A queue also called a first in first out system is a linear list in which deletions can take place only at one end of the list, the front of the list, and the insertions can take place only at the other end of the list, the "rear" of the list.

Graph

Data sometimes contain a relationship between pairs of elements which is not necessarily hierarchical in nature. This is called graph.

Data structure operations

The following are the 4 data structure operations.

- **Traversing -** accessing each record exactly once so that certain items in the record may be processed.
- **Searching -** finding the location of the record with a given key value or finding the locations of all records which satisfy one or more conditions.
- **Insertions -** adding a new record to a structure.
- **Deleting -** removing a record from a structure.
- **Sorting -** arranging the records in the some logical order.
- **Merging -** combining the records in two different sorted files into a single sorted file.

Abstract data types

ADT refers to a set of data values associated operations that are specified accurately, independent of any particular implantation. With an ADT, we know what a specify data type can do, but how it actually does it is hidden. ADT consists of a set of definitions that allow us to use the functions while hiding the implementation.

An abstract data type can be further defined as a data declaration packaged together with the operations that are meaningful for the data type. We encapsulate the data and the operations on data and then we hide them from theuser.

UNIT 1

CHAPTER 2

INTRODUCTION TO ALGORITHMS

What is an Algorithm?

In computer programming terms, an algorithm is a set of well-defined instructions to solve a particular problem. It takes a set of input(s) and produces the desired output. For example,

An algorithm to add two numbers:

Take two number inputs
Add numbers using the + operator
Display the result

Algorithmic notation

An algorithm is a finite step-by-step list of well-defined instructions for solving aparticular problem. For example,

An array A of numerical values is in memory. We want to find the location LOC andvalue MAX of the largest element of DATA.

Initially begin with LOC=1 and MAX=A [1]. Then compare MAX with each successive element A [K] of A. if A [K] exceeds MAX, then update LOC and MAX so thatLOC=K and MAX=A [K]. The final values appearing in LOC and MAX give the location and value of the largest element of A.

A nonempty array A with N numerical values is given. This algorithm finds the locationLOC and the value MAX of the largest element of A. the variable K is used as a counter.

```
Step 1:     [initialize]
            Set K:=1, LOC:=1 and MAX:=A[1]
Step 2:     [Increment counter]
            Set K:=K+1
Step 3:     [Test counter]
            If K>N then:
            Write: LOC, MAX and EXIT.
Step 4:     [Compare and update]
            If MAX < DATA[K], then:
            Set LOC:=K and MAX:=A[K]
 Step 5:     [Repeat loop]
              Go to step 2
```

The format for the formal presentation of an algorithm consists of two parts. The first part isa paragraph which tells the purpose of the algorithm identifies the variables which occur inthe algorithm and lists the input data. The second part of the algorithm consists of the list of steps that is to be executed.

Following are some algorithmic notation

- Steps, control, exit
- Comments
- Variable names
- Assignment statement
- Input and output
- Procedures

Steps, control, exit

The steps of the algorithm are executed one after another, beginning with step 1. several statements appear in the same step, e.g. Set K:=1, Loc:=1 and max:=A[1], then they are executed from left to right. The algorithm is completed when the statement Exit is control may be transferred to step n of the algorithm by the statement “go to step n”. encountered. This statement is similar to the STOP statement used in the FORTRAN and inflowcharts.

Comments

Each step may contain a comment in brackets which indicates the main purpose of the step. The comment will usually appear at the beginning or the end of the step.

Variable names

Variable names will use capital names as MAX. Single letter names of variables used as counters or subscripts will also be capitalized in the algorithms. Lowercase can be used as variable names.

Assignment statement

Our assignment statements will use the: = notation. For example, max: =A[1] assigns the value in A[1] to max.

Input and output

Data may be input and assigned to variables by means of a Read statement with the following form: **Read (variable names).** Similarly messages placed in quotation marks and the data in variables may be output by means of a Write or Print statement in the following form **Write (message and/or variable names.)**

Procedures

Procedure is an independent algorithmic module which solves a particular problem.

Control structures

Algorithms are more easily understood if they mainly use self-contained module as and three types of logic, or flow of control, called

1. Sequential logic or sequential flow
2. Selection logic or conditional flow
3. Iteration logic or repetitiveflow

Sequential logic

Instructions are executed in the sequential order. The sequence may be presented explicitly by means of numbered steps or implicitly, by the order in which modules are written.

Selection logic

Selection logic employs a number of conditions which lead to a selection of one out of several alternative modules. The structures which implement this logic are called conditional structures or if structures. The conditional structures fall into three types. They are

Single alternative

This structure has the form

```
If condition, then:
[Module A]
[End of if structure]
```

Here, the if condition holds, then module A which consists of one or more statements is executed; otherwise Module A is skipped and control transfers to the next step of the algorithm.

Double alternative

This structure has the form

```
If condition, then:
[Module A]
Else
[Module B]
[End of if structure]
```

The logic of this structure is, if the condition holds, then module A is executed; otherwise Module B is executed.

Multiple alternatives:
This structure has the form:

```
If condition(1), then:
[Module A1]
Else if condition(2), then:
[Module A2]
.
.
.
Else if condition (M), then:
[Module AM]
Else:
[Module B]
[End of if structure]
```

The logic of this structure allows only one of the modules to be executed. Specifically, either the module which follows the first condition which holds is executed, or the module which follows the final Else statement is executed. In practice, there will rarely be more than three alternatives.

Iterative logic (repetitive flow)

The third kind of logic refers to either of two types of structures involving steps. Each type begins with a Repeat statement and is followed by a module, called the body of the loop. We will indicate the end of the structure by the statement [End of loop] or some equivalent.

The repeat-for loop uses an index variable, such as K to control the loop. The loop will usually have the form:

```
Repeat for k=R to S by T:
[Module]
[End of loop]
```

Here R is the initial value. S is the end value or test value, and T is the increment. The body of the loop is executed first with K=R, then with K=R+T and then with K=R+2T and so on. The cycle ends when K>S.

The **repeat-while loop** uses a condition to control the loop. The loop will usually have the form

```
Repeat while condition:
        [module]
[end of loop]
```

The looping continues until the condition is false.

Sub algorithm

A sub algorithm is a complete and independently defined algorithmic module which is used by some main algorithm or by some other sub algorithm. A sub algorithm receives values called arguments from a calling algorithm; performs computations and then sends back the result to the calling algorithm.

The main difference between the format of a subprogram and that of an algorithm is

Here NAME refers to the name of the algorithm which is used when the sub transmit data

between the sub algorithm and the calling algorithm.

Another difference is that sub algorithm will have a **return** statement rather than an Exit statement. This means that control is transferred back to the calling program when the execution of the subprogram is completed.

that the sub algorithms will usually a heading like NAME (PAR1, PAR2………….PARk)

Sub algorithms fall into 2 categories: function sub algorithms and procedure sub algorithms. The major difference between the two is that the function sub algorithm returns only a single value to the calling algorithm whereas the procedure sub algorithm may send back more than one value.

algorithms is called and PAR1, PAR2…PARk refers to the parameters which are used to

The following function sub algorithm MEAN finds the average AVE of three numbers A,B and C.

```
Mean (A,B,C)
    1. Set AVE:=(A+B+C)/3
    2. Return (AVE)
```

Note that MEAN is the name of the sub algorithm and A, B and C are the parameters. The return statement includes the variable AVE whose value is returned to the calling program.

The sub algorithm MEAN is invoked by an algorithm in the same way as a function subprogram is invoked by a calling program. For example, suppose an algorithm contains the statement

Set TEST:=MEAN(T1, T2, T3)

Where T1, T2 and T3 are test scores. The argument values T1, T2 and T3 are transferred to the parameters A,B,C in the sub algorithm, the sub algorithm MEAN is executed and then the value of AVE is returned to the program and replaces MEAN (T1,T2,T3) in the statement. Hence the average of T1, T2 and T3 is assigned toTEST.

Variables, Data types

Each variable in any of our algorithms or programs has a data type which determines the code that is used for storing its value. Four such data types are:

1. **Character:** here data are coded using some character code such as EBCDIC or ASCII. A single character is normally stored in abyte.
2. **Real(or floating point):** here numerical data are coded using the exponential form of the data
3. **Integer:** here positive numbers are coded using binary representation and negative integers by some binary variation as 2's complement.
4. **Logical:** here the variable can have only the value true or false; hence it maybe coded using only one bit, 1 for true and 0 for false.

Local and global variables

Each program module contains its own list of variables called local variables which are accessed only by the given program module. Also, subprogram modules may contain parameters, variables which transfer data between a subprogram and its calling program.

Variables that can be accessed by all program modules are called global variables and variables that can be accessed by some program modules are called non-local variables.

UNIT-I
CHAPTER 3
RECURSION

Recursion is the name given to the technique of defining a set or a process **in terms of itself**. A recursive procedure can be called from within or outside itself.

Suppose P is a procedure containing either a call statement to itself or a call statement to a second procedure that may eventually result in a call statement back to the original procedure P. then P is called a recursive procedure.
A recursive procedure must have the following 2 properties.

1. There must be certain criteria, called base criteria, for which the procedure doesnotcall itself.
2. Each time a procedure calls itself, it must be closer to the base criteria.

A recursive procedure with these two properties is said to be well defined.
Examples: **Factorial function**
Fibonacci function
Towers of Hanoi

Factorial function:

The product of positive integers from 1 to n, is called" n factorial" and is usually denoted by n!

N! =1 2 3 ....(N-1) N

Where 0! =1 always. Therefore, for every positive integer n, n! =n(n-1)!
Accordingly, the factorial function may also be defined as follows:

a) If n=0, then n! =1
b) If n>0, the n! =n.(n-1)!

Observe that the definition of n! is recursive, since it refers to itself when it uses (n-1)!However
1. The value of n! is explicitly given when n=0 (thus 0 is the base value)
2. The value of n! for arbitrary n is defined in terms of a smaller value of n which is closer to the base value 0 accordingly, this definition is not circular. And is well defined.

The following procedures calculate n factorial.

Procedure Factorial (Fact, n)

This procedure calculates n! and returns the value in the variable Fact.

```
1. If n=0 then
2. Set Fact: =1 and return
3. Call Factorial (Fact, n-1)
4. Set Fact: =n*Fact(n-1)
5. Return
```

Fibonacci sequence:

The Fibonacci sequence is as follows 0,1,1,2,3,5,8,13
That is F_0=0 and F_1=1 and each succeeding term is the sum of two preceding terms. A formal definition of this function is as follows:

a) If n=0 or n=1, then $F_n=n$
b) If n>1, then $Fn=F_{n-2}+F_{n-1}$

This is another example of a recursive definition, since the definition refers to itself when it uses F_{n-2} and F_{n-1}. Here

a) The base values are 0 and 1
b) The value of F_n is defined in terms of smaller values of n which are closer to the base values. Accordingly, this function is well defined.

A procedure for finding the nth term F_n of the Fibonacci sequence follows:

Procedure FIBONACCI (Fib, n)

This procedure calculates Fn and returns the value in the first parameter Fib.

```
1. If n=0 or n=1 then
2. Set Fib=n and return
3. Call Fibonacci(Fib a, n-2)
4. Call Fibonacci(Fib b, n-1)
5. Set Fib:=Fib a+Fib b
6. Return
```

Towers of Hanoi:

The Tower of Hanoi is a mathematical puzzle invented by the French mathematician Edouard Lucas in 1883.

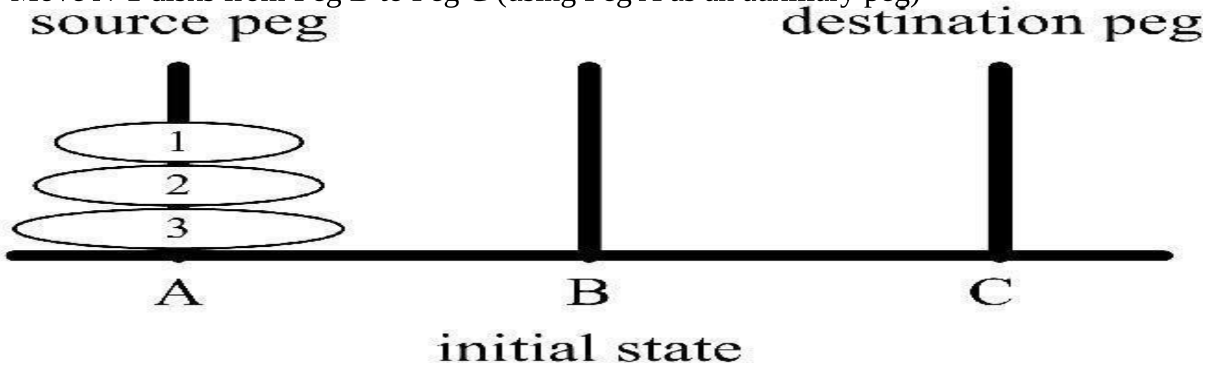
There are three pegs, source(A), Auxiliary (B) and Destination(C). Peg A contains a set of disks stacked to resemble a tower, with the largest disk at the bottom and the smallest disk at the top. figure 1 Illustrate the initial configuration of the pegs for 3 disks. The objective is to transfer the entire tower of disks in peg A to peg C maintaining the same order of the disks.

Rules:

1. Only one disk can be transfer at a time.
2. Each move consists of taking the upper disk from one of the peg and placing it on the top of another peg i.e. a disk can only be moved if it is the uppermost disk of the peg.
3. Never a larger disk is placed on a smaller disk during the transfer.

A recursive procedure for the solution of the problem for N number of disks is as follows:

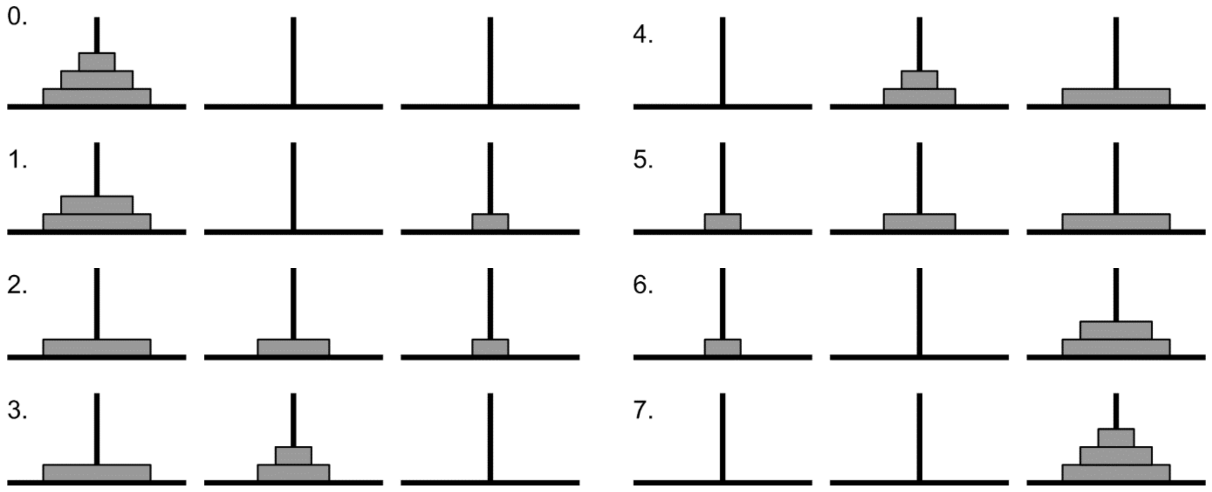
1. Move the top N-1 disks from peg A to peg B (using C as an auxiliary peg)
2. Move the bottom disk from peg A to peg C
3. Move N-1 disks from Peg B to Peg C (using Peg A as an auxiliary peg)



Algorithm:

```
TOH( n, Sour, Aux , Des)
If(n=1)
        Write ("Move Disk “, n ," from ", Sour ," to ",Des)Else
TOH(n-1,Sour,Des,Aux);
        Write ("Move Disk “, n ," from ", Sour ," to ",Des)
TOH(n-1,Aux,Sour,Des);
END
```

Let’s take an example to better understand the algorithm (For n=3).



Question Bank

Multiple Choice Questions

(Questions for Remembering)

1. ___________it is a step-by-step representation of a solution to a given problem.
   A. algorithm
   B. flowchart
   C. Pseudocode
   D. None of the above

2. The Sequence of the Instruction written to perform specific task is called the__________.
   A. Statement
   B. Program
   C. Algorithm
   D. None of the above

3. _______is a procedure or step by step process for solving a problem.
   A. Algorithm
   B. Flowchart
   C. Pseudocode
   D. All of these

4. Assuming int is of 4 bytes, what is the size of int arr[15];?
   A.15
   B.19
   C.11
   D.60

5. How many kinds of elements an array canhave?
   A. Char and int type
   B. Only char type
   C. Only int type
   D. All of them have sametype

6. Elements of an array are numbered as 0,1,2,3
   A. Index values
   B. Subscripts of array
   C. Members of an array
   D. Both A and B

7. Absolute value of integers |-10| and |-12|are ________

A. 10, 20, -10, -20
B. 0,-10
C. 10,12
D. 0, 12

8. Floor (2.4) + Ceil (2.9) is equalto ____________

A.4
B.6
C.5
D.none of the mentioned

9. Factorial of a positive integer n is______________

A. n**!**=n(n-2)**!**
B. n**!**= n(n+2) **!**
C. n**!** = (n-2)**!**
D. n! = n(n-1)!

10. This Loop tests the condition after having executed the Statements within the Loop.

A. while
B.do-while
C.for Loop
D.if-else-if

11. To implement sparse matrix dynamically, the following data structure is used

A. Trees
B. Graphs
C. Priority Queues
D. Linked List

(Questions for Understanding)

12. Choose a right C Statement.
    A. Loops or Repetition block executes a group of statements repeatedly
    B. Loop is usually executed as long as a condition is met
    C. Loops usually take advantage of Loop Counter
    D. All the above

13. The optimal data structure used to solve Tower of Hanoi is ___________
    A. Tree
    B. Heap
    C. Priority queue
    D. Stack

14. What is the number of moves required to solve Tower of Hanoi problem for k disks?
    A. 2k – 1
    B. 2k + 1
    C. $2^k + 1$
    D. $2^k - 1$

15. Which of the following data structure can't store the non-homogeneous data elements?
    A. Arrays
    B. Records
    C. Pointers
    D. None

16. The logical or mathematical model of a particular organization of data is called as__________
    A. data structure
    B. data arrangement
    C. data configuration
    D. data information

17. Recursion uses more memory compared to iteration.
    A. True
    B. False
    C. Both
    D. None of the above

18. Which of the following is a linear data structure?
    A. Array
    B. AVL Trees
    C. Binary Trees
    D. Graphs

19. Referring an element outside array bounds is a_______________
    A. Syntax error
    B. Logical error
    C. execution time error
    D. both b and c

20. Recursion is similar to which of the following?

A. Switch Case
B. <u>Loop</u>
C. If-else
D. if elif else

21. Which Data Structure is mainly used for implementing the recursive algorithm?

A. Queue
B. <u>Stack</u>
C. Linked List
D. Tree

22. What's happen if base condition is not defined in recursion?

A. Stack underflow
B. <u>Stack Overflow</u>
C. None of these
D. Both a and b

23. Which searching can be performed recursively?

A. Linear search
B. Binary search
C. <u>Both</u>
D. None

24. Which of the following problems can't be solved using recursion?

A.Factorial of a number
B. Nth fibonacci number
C.Length of a string
D. <u>Problems without base case</u>

25. Recursion is similar to which of the following?

A.Switch Case
B. <u>Loop</u>
C. If-else
D. if elif else

Questions carrying 4 marks

(Questions for Remembering)

1. What are the data structure operations?
2. Describe a note on ADT.
3. Describe sub algorithms
4. Describe algorithmic notations.
5. Explain the various types of Data Structures with examples.
6. Explain the classification of non-primitive data structures with examples.

(Questions for Understanding)

1. Describe conditional structures.
2. Describe Loop structures.
3. What do you mean by recursion? Give any 2 recursive techniques.
4. Explain Factorial function by using Recursive technique.
5. Explain Fibonacci series by using recursive technique.
6. Explain Towers of Hanoi by using recursive technique.

UNIT II
CHAPTER – 4. STACKS

Introduction

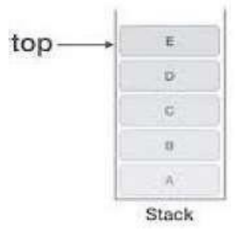
Two of the data structures that are useful are stacks and queues. A stack is a linear structure in which items may be added or removed only at one end. A queue is a linear list in which items may be added at one end and items are removed only at the other end.

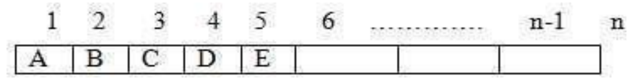
Stacks

A Stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack. There are two basic operations associated with a stack. They are

- **PUSH -** is the term used to insert an element into a stack.
- **POP -** is the term used to delete an element from stack.

A pointer 'TOP' keeps track of the top element in the stack. Initially, when the stack is empty, TOP value is zero. When the stack contains a single element TOP has a value 1 and so on. Each time an element is inserted into the stack, the pointer is incremented by 1 before the element is placed on the stack. The pointer is decremented by 1 each time a deletion is made from the stack.

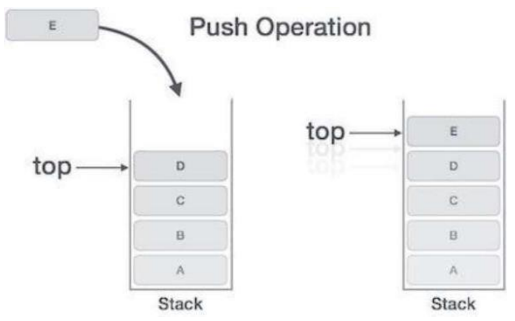




Operations on a Stack : PROCEDURE PUSH(S, TOP, X)

This procedure inserts an element X into the top of the stack which is represented by an array S. The array S contains N elements with a pointer TOP denoting the top element in the stack.

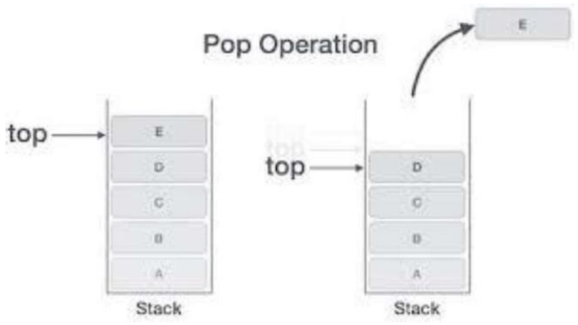
```
1. [Check for Overflow]
        If TOP>=N
                Then
                write('Overflow')
                Return
2. [Increment TOP]
        TOP=TOP+1
3. [Insert Element]
        S[TOP]=X
4. [Finished]
        Return
```



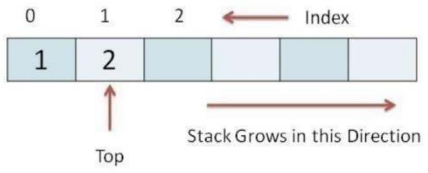
FUNCTION POP(S, TOP)

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

```
1.[Check for underflow on stack]
        If TOP=0
            Then write('Stack underflow on POP')
        Exit
2.[Decrement TOP pointer]
        TOP=TOP-1
3.[Return former top element of the Stack]
        Return (S[TOP+1])
```



Array representation of stacks

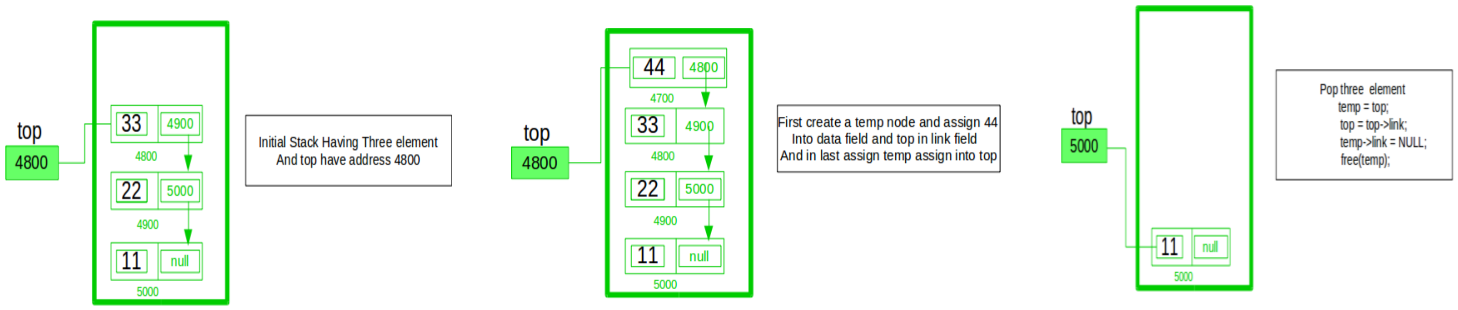


Stacks may be represented in the computer in various ways, usually by means of a linear array. Stack contains a pointer variable TOP, which contains the location of the top element of the stack and a variable N which gives the maximum number of elements that can be held by the stack. The condition TOP=0 or TOP=NULL will indicate that stack is empty.

The following above shows the array representation of a stack. Since top=2, the stack has 2 elements 1 and 2. The maximum size of the stack is 6, and 4 more elements can be inserted into the stack.

Linked Representation of Stacks

To implement a stack using the singly linked list concept, all the singly linked list operations should be performed based on Stack operations LIFO(last in first out), and with the help of that knowledge, we are going to implement a stack using a singly linked list.
So we need to follow a simple rule in the implementation of a stack that is last in first out and all the operations can be performed with the help of a top variable.



Push Operation:

- *Initialise a node*
- *Update the value of that node by data i.e.* ***node->data = data***
- *Now link this node to the top of the linked list*
- *And update top pointer to the current node*

Algorithm:

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether stack is **Empty** (**top** == **NULL**)
- **Step 3 -** If it is **Empty**, then set **newNode** → **next** = **NULL**.
- **Step 4 -** If it is **Not Empty**, then set **newNode** → **next** = **top**.
- **Step 5 -** Finally, set **top** = **newNode**.

Pop Operation:

- *First Check whether there is any node present in the linked list or not, if not then return*
- *Otherwise make pointer let say* ***temp*** *to the top node and move forward the top node by 1 step*
- *Now free this temp node*

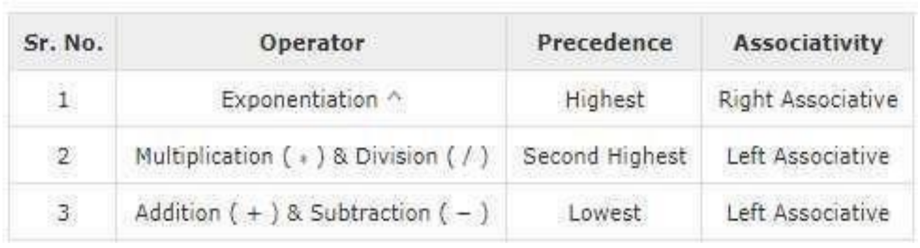
Algorithm:

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

- **Step 1 -** Check whether **stack** is **Empty** (**top == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
- **Step 3 -** If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
- **Step 4 -** Then set '**top** = **top** → **next**'.
- **Step 5 -** Finally, delete '**temp**'. (**free(temp)**).

Arithmetic expressions: polish notation: Precedence of operator



Sr. No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (–)	Lowest	Left Associative

Operator	Precedence Value / Priority
(	1
)	2
+ and -	3
/ and *	4
^	5

An example, a+b*c+d*e

For the evaluation of expression, we must scan the expression from left to right. First we should evaluate the expression with highest priority, if the priority of the 2 expressions is equal then start evaluating the expression from left to right.

If there are parentheses in an expression, the order of precedence is altered by the parenthesis. For example, in (a+b)*c, we first evaluate a+b, then (a+b) * c because parenthesis have highest priority.

An arithmetic expression can be represented in three different formats they are
1. Infix
2. Prefix and
3. Postfix.

Infix

It's the form of an arithmetic expression in which we fix place the arithmetic operators in between 2 operands. This is the usual notation of writing mathematicexpression.

Example, let A and B are 2 operands. Then the infix form of arithmetic operator +, -, * and / are given below. Infix forms will be

- A+B
- A-B
- A*B
- A/B

Postfix

It's the form of an arithmetic expression in which we fix brackets place the arithmetic operator after (post) its 2 operands. The postfix notation is called "Suffix Notation". It's also referred to as "Reverse Notation".

Example, let A and B are 2 operands. Then we can write postfix expression as

Infix	Postfix
A+B	AB+
A -B	AB-

Prefix

It's the form an arithmetic expression in which we fix brackets place, the arithmetic operators before (pre) its 2 operands. The prefix form of expression is also called "Polish Notation".

Example, let A and B are 2 operands. Then we can write the prefix expression usingarithmetic expression + -and as follows.

Infix	Prefix
A-B	+AB
A-B	– AB

Algorithmic Transformation

The manual operation would be difficult to implement in a computer. Let us look at another technique that is easily implemented with a stack. Infix expressions use a precedence rule to determine how to group the operands and operators in an expression. We can use the same rule when we convert infix to postfix. The rules are,

1) When evaluating an expression, if an operand is found, it is appended to the output expression.
2) a) If an operator is found and if the priority of the current operator is higher than the operator which is at the top of the stack, then push the current operator to the stack.
   b) If the priority of the current operator is lower than or equal to the operator at the top of the stack, the operator at the top of the stack is popped out to the output expression

General Infix-to-Postfix Conversion

As we scan the infix expression from left to right, we will use a stack to keep the operators. The top of the stack will always be the most recently saved operator. Whenever we read a new operator, we will need to consider how that operator compares in precedence with the operators, if any, already on the stack.

Assume the infix expression is a string of tokens delimited by spaces. The operator tokens are *, /, +, and -, along with the left and right parentheses, (and). The operand tokens are the single-character identifiers A, B, C, and so on. The following steps will produce a string of tokens in postfix order.

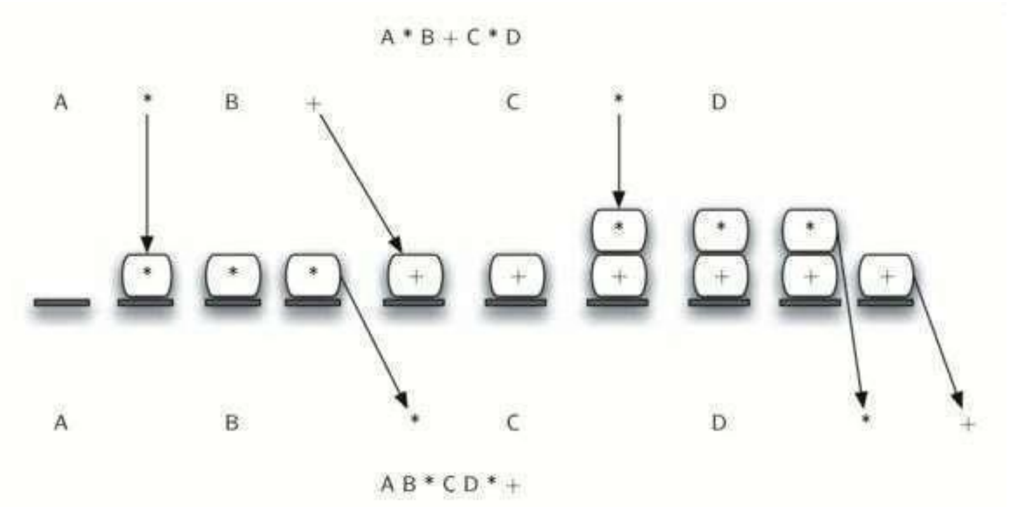
1. Create an empty stack called opstack for keeping operators. Create an empty list for output.
2. Convert the input infix string to a list by using the string method split.
3. Scan the token list from left to right.
   - If the token is an operand, append it to the end of the output list.
   - If the token is a left parenthesis, push it on the opstack.
   - If the token is a right parenthesis, pop the opstack until the corresponding

left parenthesis is removed. Append each operator to the end of the output list.

- If the token is an operator, *, /, +, or -, push it on the opstack. However, first remove any operators already on the opstack that have higher or equal precedence and append them to the output list.

4. When the input expression has been completely processed, check the opstack. Any operators still on the stack can be removed and appended to the end of the output list.

Below figure shows the conversion algorithm working on the expression A * B + C * D. Note that the first * operator is removed upon seeing the + operator. Also, + stays on the stack when the second * occurs, since multiplication has precedence over addition. At the end of the infix expression the stack is popped twice, removing both operators and placing + as the last operator in the postfix expression.



Algorithm: Conversion of Infix Expression to postfix

Procedure Postfix (inputExpr, St, Symbol, OutputExpr)

This procedure converts an infix expression to postfix expression. The infix expression is taken in *inputExpr, st* represents a new stack, *Symbol* is the current Symbol, *stack_top_symbol* is the symbol at the top of the stack, t*opSymbo*l is/are the symbol(s) remained in the stack at the end of the evaluation and *outputExpr* is the postfix expression

```
St= new stack
Repeat for each character in inputExpr
Begin
        Symbol=current character in inputExpr
        if (symbol is an operand)
                add symbol to outputExpr
        else
                begin
while( not st.empty && (priority(symbol ))≤ priority(stack_top_symbol)))
Begin
                Add stack_top_symbol to outputExpr
End while
St.push(symbol)
End else
End for
While(not st.empty)
Begin

/*to pop the remaining operators from the stack at the end.*/
          topSymbol=st.pop()
          add topSymbol to outputExpr
     end while
```

Evaluating Postfix expressions

Consider the following expression ABC+* and assume that A is 2, B is 4 and C is 6. Here you should notice that the operands come before the operators. Whenever we find an operator, we put them in the stack. When we find an operator, we pop the 2 operands at the top of the stack and perform the operation. We then push the value back into the stack to be used later.

Algorithm: Evaluation of postfix expressions

FUNCTION Evaluate_Postfix(Expression,st, symbol) This procedure evaluates an postfix expression. Expression is the postfix expression. st represents a new stack, Symbol is the current symbol

```
St=new stack
Repeat for every character in the expression
Begin
     Symbol=current character in the expression
     If (symbol is an operand)
          St.push(symbol)
     Else
     Begin
          Operand2=st.pop()
          Operand1=st.pop()
          Answer=operand2 and operand1 operated with symbol
          St.push(answer)
     End
End
```

Return st.pop() //to display the output

Applications of Stack:

- Evaluation of Arithmetic Expressions
- Backtracking
- Delimiter Checking
- Reverse a Data
- Processing Function Calls

Evaluation of Arithmetic Expressions

A stack is a very effective data structure for evaluating arithmetic expressions in programming languages. An arithmetic expression consists of operands and operators.
In addition to operands and operators, the arithmetic expression may also include parenthesis like"left parenthesis" and "right parenthesis".

Example: A + (B - C) Notations for Arithmetic Expression

There are three notations to represent an arithmetic expression:

- Infix Notation
- Prefix Notation
- Postfix Notation

Backtracking

Backtracking is another application of Stack. It is a recursive algorithm that is used for solving The optimization problem.

Delimiter Checking

The common application of Stack is delimiter checking, i.e., parsing that involves analyzing a source program syntactically. It is also called parenthesis checking. When the compiler translates a source program written in some programming language such as C, C++ to a machine language, it parses the program into multiple individual parts such as variable names, keywords, etc. By scanning from left to right. The main problem encountered while translating is the unmatched delimiters. We make use of different types of delimiters include the parenthesis checking (,), curly braces {,} and square brackets [,], and common delimiters /* and */. Every opening delimiter must match a closing delimiter, i.e., every opening parenthesis should be followed by a matching closing parenthesis.

Reverse a Data:

To reverse a given set of data, we need to reorder the data so that the first and last elements are exchanged, the second and second last element are exchanged, and so on for all other elements.

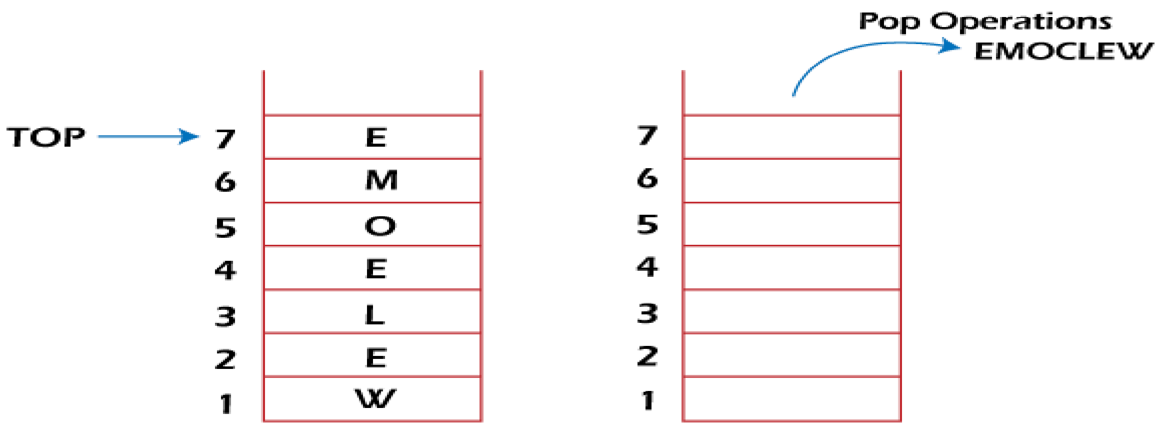
Example: Suppose we have a string Welcome, then on reversing it would be Emoclew.

There are different reversing applications:

- Reversing a string

Reverse a String

A Stack can be used to reverse the characters of a string. This can be achieved by simply pushing one by one each character onto the Stack, which later can be popped from the Stack one by one. Because of the **last in first out** property of the Stack, the first character of the Stack is on the bottom of the Stack and the last character of the String is on the Top of the Stack and after performing the pop operation in the Stack, the Stack returns the String in Reverse order.



UNIT II

Chapter 5 Queues

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 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.

Front contains the location of the FRONT element of the queue and REAR contains th location of the last element of the queue. The condition FRONT=NULL will indicate that queuis empty.

The following figure shows the way the array will be stored in memory using an array QUEUE with N elements. Whenever an element is added to the queue, the value of REAR incremented by 1. Whenever an element is deleted from the queue, the value of FRONT incremented by 1.

Queues are also called FIFO lists, since the first element in a queue will be the firs element goes out of the queue.

Array Representation of queues

AAA	BBB	CCC	DDD						
1	2	3	4	5	6	7	8..		N

Here Front=1, Rear=4

	BBB	CCC	DDD						

Here Front=2 and Rear=4

	BBB	CCC	DDD	EEE	FFF				

Here Front=2 and Rear=6

Queue operations:

PROCEDURE QINSERT ( )

Given F and R as the pointers to the front and rear element of a queue Q. The array Q contains N elements. Y is the element to be inserted at the rear.

```
1. [Overflow
   ?]If R>=N
   Return
2. [Increment Rear]
```

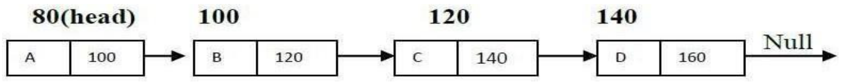
```
        R=R+1
3. [Insert element]
        Q[R] =Y
4. IF F=0
        Then F=1
5. Exit
```

Function QDELETE()

Given F and R as the pointers to the front and rear element of a queue Q. The array Q contains N elements. This function deletes an element of the queue. Y is the temporary variable.

```
[Underflow]
    If F=0

        Return 0;

    1.  [set a pointer to an element todelete]
        Y    Q[F]

    2.  [Check whether it is a only one element in thequeue]
        If F=R
                F=0
                R=0
        Else
                F=F+1


                Return Y

              3.EXIT
```

Linked list representation of Queue:

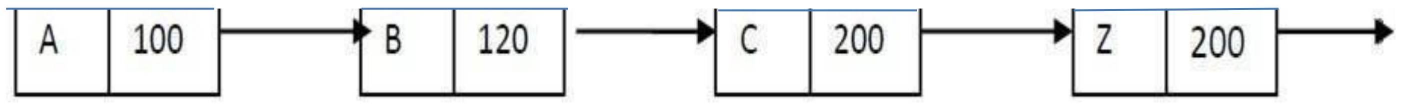


- Two pointer variables FRONT and REAR pointing to the nodes which is in the front and rear of the queue.
- The INFO field of the node holds the data in the queue .
- The NEXT is the pointer to the next element in queue.

Procedure LQINSERT(INFO,NEXT, ITEM, REAR,FRONT)

This procedure inserts an element ITEM into a linked queue.

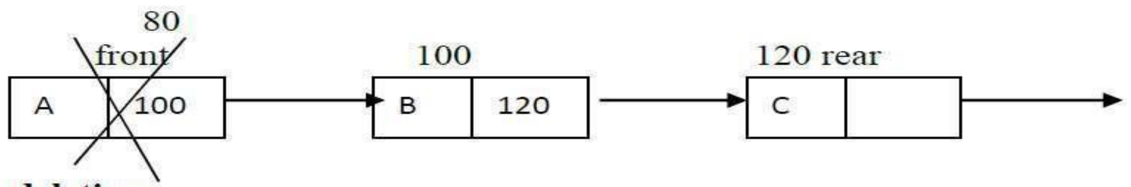
```
1.  N= new node
2.  If (n==NULL)
3.  Then write ("Overflow") and exit
4.  N->info=ITEM
5.  If (Front==NULL) then
6.  Front=Rear=n
7.  Else
8.  Rear->next=n
9.  Rear=n
10. N->next=NULL
11. finished
```



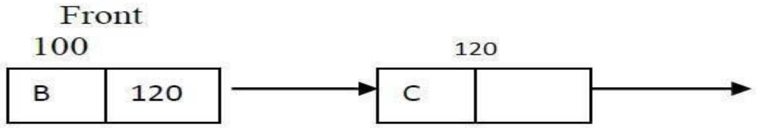
Procedure LQDELETE(INFO, NEXT,FRONT, REAR, ITEM,TEMP)

This procedure deletes the front element of the linked queue and stores it in ITEM.

```
1. Temp= front
2. Item=front→info
3. Front=temp    next
4. Delete temp
5. Finished
```



After deletion



QUEUES as ADT

The basic design of the stack ADT is also followed by Queue ADT. The node structure is identical to the structure we for a stack. Each node contains a void pointer to the data and a link pointer to the consecutive element in the queue.

Queue ADT

```
Typedef struct NODE
{
        Void *dataptr
        Struct NODE* next
} QUEUENODE;


Typedef struct
{
        QUEUENODE* front;
        QUEUENODE* Rear;
        int count;
}QUEUE;
```

Create Queue

Create queue allocates a node for the queue header. It then initializes the front and rear pointers to NULL and sets the count to zero. If overflow occurs, the return value is NULL. If the allocation is successful, it returns the address of the queue head.

Function create Queue: this function allocates memory for a queue head node form dynamic memory and returns its address to the caller. It allocates the head and initializes it. It returns the head if successful and NULL if overflow.

```
QUEUE* createqueue(void)
{
        QUEUE* queue1
        Queue1=(QUEUE*) malloc(sizeof(QUEUE));
        If (queue1)
        {
                Queue1 → front=NULL
                Queue1 →  rear=NULL
                Queue1 count=0;
        }
Return queue1;
}
```

Insertion

If memory is available, it creates a new node, inserts it at the rear of the queue and returns true. On the other hand, if the memory is not available, it returns false.

```
Bool inqueue(QUEUE* newqueue, void* datainptr)
{
        QUEUENODE *newptr;
        Newptr=(QUEUENODE*) malloc(sizeof(QUEUENODE));
        If(newptr==NULL)
                Return false
        Newptr->dataptr=datainptr;
        Newptr->next=NULL;

        If(newqueue->count==0) //if queue is empty
                Newqueue->front=newptr;
        Else
                Newqueue->rear->next=newptr;

(Newqueue->count)++;
Newqueue->rear=newptr;
```

Deletion

To delete a node from the queue, the following function is used. The queue should have already been created. The function returns the data pointer to the front of the queue and the front element is deleted. It return true if the deletion is successful and false if there is an underflow.

```
Bool delqueue(QUEUE* queue1, void* *datainptr)
{

        QUEUENODE* temp;
        If (queue1→count==0)
                Return false;

        *datainptr=queue1 →front→dataptr;
        Temp=queue1→front;

        If(queue1→count==1)
                Queue1→rear=queue1→front=NULL
        Else
                Queue1→front=queue1→fornt→next;

                (Queue1→count)--;
                Free(temp);
        Return true;
}
```

Circular queues

The simple queue has a great disadvantage that, as the front and rear values go on increasing, the storage will keep on increasing. This may result in queue overflow without the queue being full. To overcome these circular queues are used.

Circular Queue Operations

PROCEDURE CQINSERT(F,R,Q,N,Y) Given F and R as the pointers to the front and rear element of a circular queue. The array Q contains N elements. Y is the element to be inserted at the rear.

```
Step 1: IF (REAR+1)%MAX = FRONT
Write " OVERFLOW "
Goto step 4
[End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
write('overflow')
```

```
ELSE IF REAR = MAX - 1 and FRONT ! = 0
SET REAR = 0
ELSE
SET REAR = (REAR + 1) % MAX
[END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

FUNCTION CQDELETE(F,R,Q,N) Given F and R as the pointers to the front and rear element of a circular queue. The array Q contains N elements. This function deletes and returns the last element of the queue. Y is the temporary variable.

```
Step 1: IF FRONT = -1
Write " UNDERFLOW "
Goto Step 4
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
SET FRONT = REAR = -1
ELSE
IF FRONT = MAX -1
SET FRONT = 0
ELSE
SET FRONT = (FRONT + 1)%N
[END of IF]
[END OF IF]
Step 4: EXIT
```

DEQUES

A deque is a linear list in which elements can be added or removed at either end but not in the middle. The term deque is a short form of **double ended queue**.

We will assume that out **deque** is maintained by a circular array DEQUE with pointers LEFT and RIGHT, which point to the two ends of the **deque.**
comes from the fact that we assume that DEQUE[1] comes after DEQUE[N] in the array.

There are two variations of a deque.

1. An input-restricted deque
2. Output restricted deque

An input restricted deque is a deque which allows insertions at only one end of the list but allows deletions at both ends of the list.

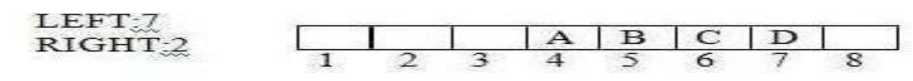
N output restricted deque is a deque which allows deletions at only one end of the list but allows insertions at both ends of the list.

		AAA	BBB	CCC	DDD		

Left = 3
Right = 6

YYY	ZZZ				WWW	XXX

Left = 6
Right = 2



Drawbacks of Deques

The complication may arise, when there is overflow, i.e. when an element is to be inserted into a deque which is already full. Or when there is underflow, i.e. when an element is to be deleted from a deque which is empty.

Priority queues

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed c mes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were added to the queue.

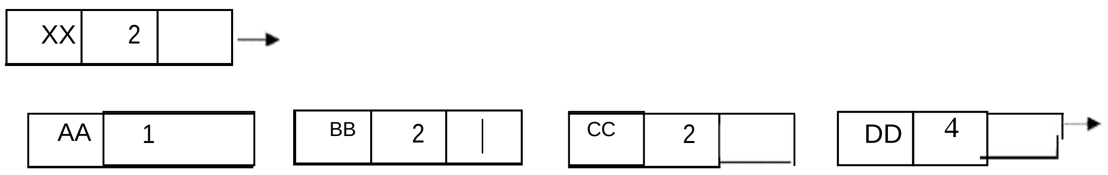
There are various ways of maintaining a priority queue in memory. One uses a one- way list, and other uses multiple queues.

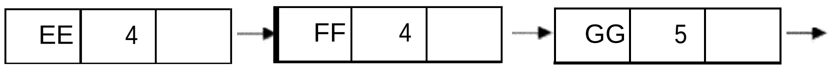
One way representation of a priority queue

One way to maintain a priority queue in memory is by means of one way list, as follows:

1. Each node in the sits will contain three items of information: an informati n filed INFO, a priority number PRN and a link number LINK.
2. A node X precedes a node Y in the list 1) when X has higher priority than Y or 2) when both have the same priority but X was added to the list before Y. this means that the order in the one-way list corresponds to the order of the priority queue.
3. Priority numbers will operate in the usual way: the lower the priority number, the higher the priority

Consider the diagram

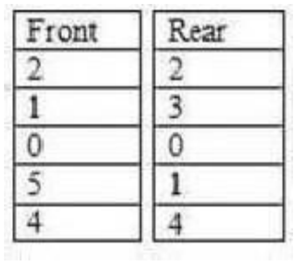




The main property of the one way list representation of a priority queue is that the element in the queue that should be processed first always appears at the beginning of the one way list.

Array representation of a priority queue

Another way to maintain a priority queue in memory is to use separate queue for each level of priority. Each such queue will appear in its own circular array and must have its own pair of pointers, FRON and REAR. If each queue is allocated the same amount of space, a two dimensional array QUEUE can be used instead of the linear arrays.



Front	Rear
2	2
1	3
0	0
5	1
4	4

$$\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
1 & & \text{AAA} & & & & \\
2 & \text{BBB} & \text{CCC} & \text{XXX} & & & \\
3 & & & & & & \\
4 & \text{FFF} & & & & \text{DDD} & \text{EEE}
\end{array}$$

Applications of Queue Data Structure

A queue is used when things don’t have to be processed immediately, but have to be processed in **F**irst **I**n **F**irst **O**ut order.

Useful Applications of Queue

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

Applications of Queue in Operating systems:

- Semaphores
- FCFS (first come first serve) scheduling, example: FIFO queue
- Spooling in printers
- Buffer for devices like keyboard

Applications of Queue in Networks:

- Queues in routers/ switches
- Mail Queues
- **Variations:** (Deque, Priority Queue, Doubly Ended Priority Queue)

Some other applications of Queue:

- Applied as waiting lists for a single shared resource like CPU, Disk, and Printer.
- Applied as buffers on MP3 players and portable CD players.
- Applied on Operating system to handle the interruption.
- Applied to add song at the end or to play from the front.

Applied on WhatsApp when we send messages to our friends and they don’t have an internet connection then these messages are queued on the server of WhatsApp

Question Bank

Multiple Choice Questions

(Questions for Understanding)

1. ________________________form of access is used to add and remove nodes from a queue.
   - A. LIFO, Last In First Out
   - **B. <u>FIFO, First In First Out</u>**
   - C. Both a and b
   - D. None of these

2. In the linked representation of the stack_______behaves as the top pointer variable of stack.
   - A. Stop pointer
   - B. Begin pointer
   - **C. <u>Start pointer</u>**
   - D. Avail pointer

3. New nodes are added to the________of the queue.
   - A. Front
   - **B. <u>Back</u>**
   - C. Middle
   - D. Both A and B

4. In the linked representation of the stack the null pointer of the last node in the list signals
   - A. Beginning of the stack
   - **B. <u>Bottom of the stack</u>**
   - C. Middle of the stack
   - D. In between some value

5. What happens when you push a new node onto a stack?
   - **A. <u>The new node is placed at the front of the linked list</u>**
   - B. The new node is placed at the back of the linkedlist
   - C. The new node is placed at the middle of the linkedlist
   - D. No Changes happens

6. A queue is a______________
   - **A. <u>FIFO</u>**
   - B. LIFO
   - C. FILO
   - D. LOFI

7. The term push and pop is related to
   A. Array
   B. Lists
   C. <u>Stacks</u>
   D. Trees

8. Which is the pointer associated with the stack?
   A. FIRST
   B. FRONT
   C. <u>TOP</u>
   D. REAR

9. The elements are removal from a stack in_____________order.
   A. <u>Reverse</u>
   B. Hierarchical
   C. Alternative
   D. Sequential

10. The insertion operation in the stack iscalled________________
   A. Insert
   B. <u>Push</u>
   C. Pop
   D. top

11. Is the term used to insert an element into stack.
   A. <u>Push</u>
   B. Pull
   C. Pop
   D. Pump

12. Stack follows the strategy of_________
   A. <u>LIFO</u>
   B. FIFO
   C. LRU
   D. RANDOM -

13. ________is the term used to delete an element from the stack.
   A. Push
   B. Pull
   C. <u>Pop</u>
   D. Pump

14. Deletion operation is done using_______________in a queue.
   A. <u>front</u>
   B. rear
   C. top
   D. list

15. A pointer variable which contains the location at the top element of the stack is called__________
   A. <u>Top</u>
   B. Last
   C. Final
   D. End

16. Which of the following is an application of stack?
   A. finding factorial
   B. tower of Hanoi
   C. infix to postfix
   D. all of the above

17. Circular Queue is also known as
   A. Ring Buffer
   B. Square Buffer
   C. Rectangle Buffer
   D. Curve Buffer

18. A data structure in which elements can be inserted or deleted at/from both ends but not in the middle is?
   A. Queue
   B. Circular queue
   C. Dequeue
   D. Priority Queue

(Questions for Skill)

19. In linked representation of stack________________holds the elements of stack.
   A. INFO fields
   B. TOP fields
   C. LINK fields
   D. NULL fields

20. __________________form of access is used to add remove nodes from a stack.
   A. LIFO
   B. FIFO
   C. Both A and B
   D. None of these

21. Which of the following name does not relate to stacks?
   A. FIFO lists
   B. LIFO lists
   C. Piles
   D. Push down lists

22. The retrieval of items in a stack is__________Operation.
   A. push
   B. pop
   C. retrieval
   D. access

23. The expression +a*bc is in_________________notation.
   A. Infix
   B. Prefix
   C. Postfix
   D. Reverse Polish

24. The expression a+b*c is in_____________notation.
   A. Infix
   B. Postfix
   C. Prefix

D. Reverse polish

25. If the elements “A”,”B”,”C” and “D” are placed in a queue and are deleted one at a time, in what order will they be removed?
A. ABCD
B. DCBA
C. DCAB
D. ABDC

Questions carrying 4 marks

(Questions for Understanding)

1. What do you mean by a queue? What are operations involved in a queue.
2. Give any four applications of a stack.
3. Give any four applications of a queue.
4. Write the algorithm for linked list representation of a stack.
5. Write the algorithm for linked list representation of a queue.
6. Write a note on priority queue and dequeue.

(Questions for Skill)

1. Write an algorithm to insert and delete elements from/to a circular queue.
2. Evaluate ABC*+D- with proper step. Assume A=4, B=6, C=2, D=4.
3. Write an algorithm to convert infix expressions to postfix expressions.
4. Write an algorithm to evaluate postfix expressions.
5. Write the PUSH and POP operations in a stack.
6. Evaluate 231*+9- the postfix expression with proper steps.

-

UNIT- 3
Chapter - 6
DYNAMIC MEMORY MANAGEMENT

Dynamic Memory Allocation in C++

Dynamic Memory Allocation is a process in which we allocate or deallocate a block of memory during the run-time of a program. There are four functions malloc(), calloc(), realloc() and free() present in <stdlib.h> header file that are used for Dynamic Memory Allocation in our system.

Uses of different functions:

- **malloc()**:- Allocate request size of bytes & return a pointer to the first byte of the allocated space. And contains garbage values.

  Syntax:
  ```
  ptr = (cast-type*) malloc(byte-size);
  ```
  For Example:
  ```
  ptr = (int*) malloc(100 * sizeof(int));
  ```

- **calloc()**:- Allocate space for an array of elements, initialize them to zero and returns a pointer to the first byte of allocated space.

  Syntax:
  ```
  ptr = (cast-type*)calloc(n, element-size);
  ```
  Here, n is the no. of elements and element-size is the size of each element.

  For Example:
  ```
  ptr = (float*) calloc(25, sizeof(float));
  ```

- **realloc()**:- Modify the size of previously allocated space.

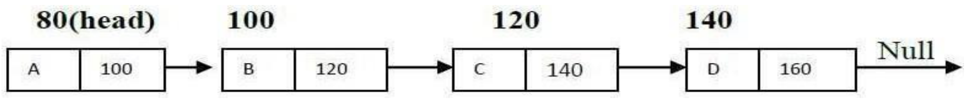
  Syntax:
  ```
  ptr = realloc(ptr, newSize);
  ```
  where ptr is reallocated with new size 'newSize'.

- **free()**:- Free the previously allocated space.

  Syntax:
  ```
  free(ptr);
  ```

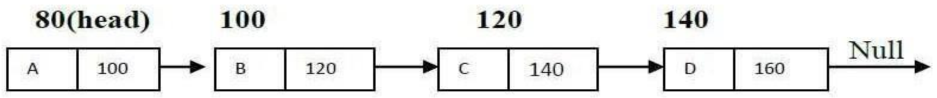
Chapter - 7
LINKED LIST

A linked list is a non-sequential collection of data items. For every data item in the linked list, there is an associated pointer that gives the memory location of the next data item in the linked list. The data items in the linked list are not in a consecutive memory locations. But they may be anywhere in memory. However, accessing of these items is easier as each data item contained within itself the address of the next data item.



Components of a linked list

A linked list is a non-sequential collection of data items called nodes. Each node in a linked list basically contains 2 fields namely, an information field called INFO and a pointer field denoted by NEXT. The INFO field contains the actual value to be stored and processed and the NEXT field contains the address of the next data item. The address used to access a particular node is known as a pointer.



NULL pointer: the NULL pointer does not contain any address and indicates the end of the list.

Representation of a linked list

Each and every node in a linked list is a structure containing two fields such as INFO andNEXT field. Such a structure is represented in object-oriented terminology as follows.

NODE

INFO	NEXT

```
Class node
{
        int info;
        Node * next;
};
```

The node has clearly 2 fields. The first one is an integer data item called Info and the second one is a link to the next node (NEXT) of the same type.

Types of linked list

Basically we can put linked lists into following 4 types.

1. Singly linked list
2. Doubly linked list
3. Singly circular linked list
4. Doubly circular linked list

Singly linked list

A singly linked list is the one in which all nodes are linked together in some sequential manner. Hence it is also called linear linked list. It has the beginning and the end. The problem with this list is that we cannot access the predecessor or the node from the current node.

Basic Operations

The basic operations to be performed on the linked list are

1. Creation
2. Insertion
3. Deletion

Creation

This operation is used to create a linked list. Here, the node is created as and when required and linked to the list.

Insertion

This operation is used to insert a new node in the linked list at the specified position. A new node may be inserted

a) At the beginning of alinked list
b) At the specified position
c) At the end of a linked list

Deletion

This operation is used to delete a node from the linked list. A nod may be deleted from the

a) At the beginning of alinked list
b) At the specified position
c) At the end of a linked list

In object-oriented programming, a node is created using a new keyword

Algorithm for creating a new node

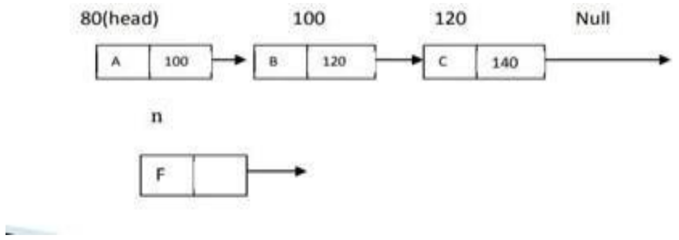
FUNCTION Create_Node(x)

This algorithm creates a new node called n and x is information in the node n

```
1. [create a new node]
   N=newnode
2. [set the INFO]
```

```
    n->info=x
3.  [return the created node n]
    Return n
4.  [finished]
    Exit
```

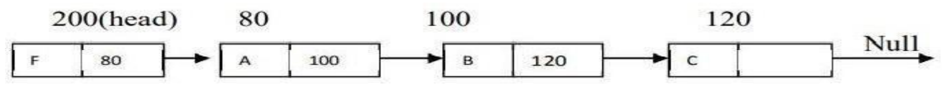
Algorithm for inserting a node at the beginning of a linked list



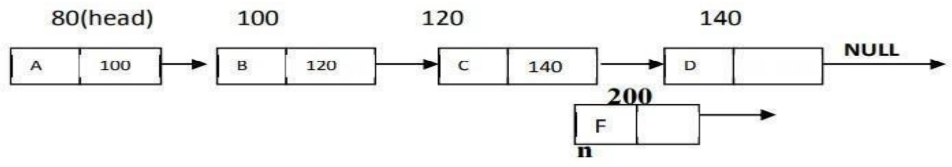
Procedure Insert_Head(n)

This algorithm inserts a new node called n to the beginning of a linked list. Head denote the starting node of the linked list

```
1.  [make head as the next element to the newnode]
    N->next=head
2.  [make our new node as head]
    Head=n
3.  [finished]
    Exit
```



Algorithm for inserting a node in the middle of a linked list



Procedure insert_middle(n)

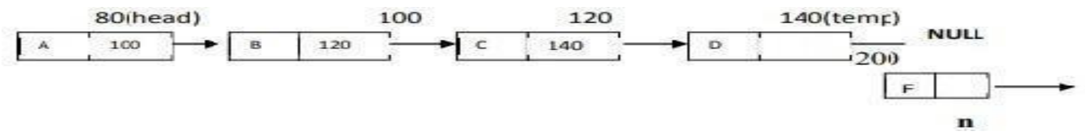
This algorithm inserts a new node called n in the specified position of the linked list. After is a node where the new node is to be inserted after it.

```
1.  N→next=after→ next
2.  After→next=n
3.  Exit
```

OR

```
1. Newnode=n
2.newnode → next=temp → next
3.temp→ next=newnode
4.Exit
```

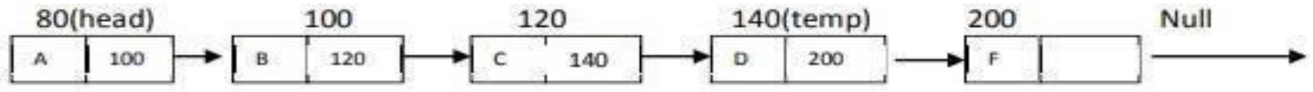
Algorithm for inserting a node at the end of a linked list



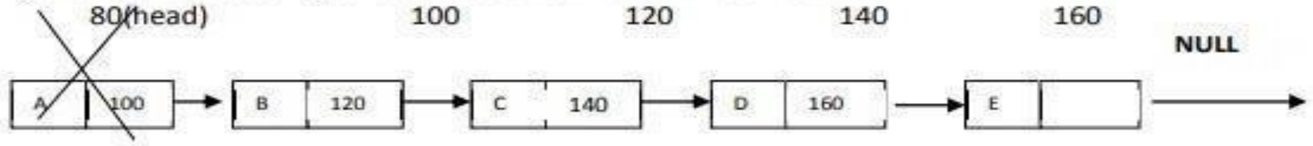
This algorithm inserts a new node called n at the end of the linked list. Temp is a temporary variable where the head node is stored temporarily before insertion takes place.

Procedure_Insert_end(n)

```
1. Temp=head
2. While(temp→next !=NULL)
   Begin
3. Temp=temp→next
   End
4. Temp→next=node
5. n→next=NULL
6. Exit
```



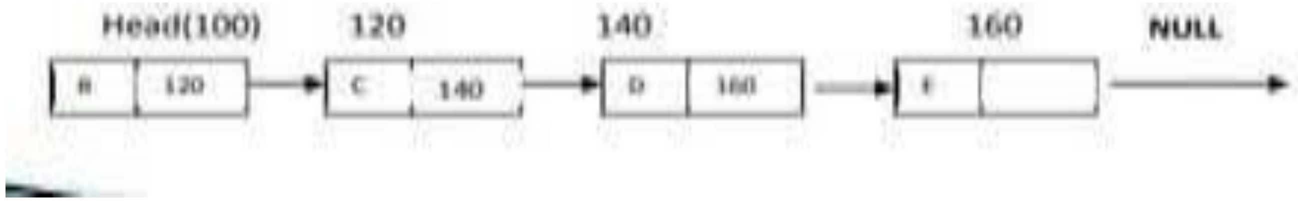
Algorithm for deleting a head node from the linked list



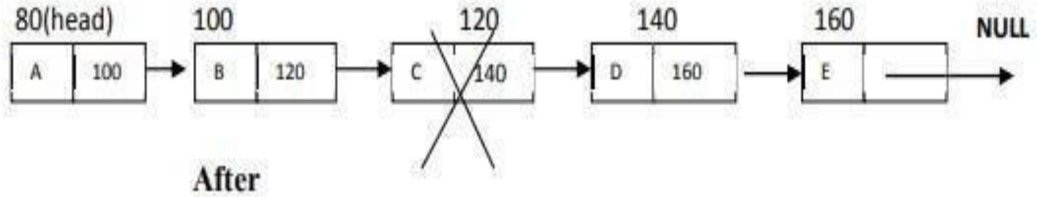
Procedure Delete_Head ()

This algorithm deletes the head node of the linked list. Tempis a temporary variable where the head node is stored temporarily before deletion takes place

```
1. temp=head
2. Head=head → next
3. Delete temp
4. Exit
```



Algorithm for delete a node form the middle of a linked list



After

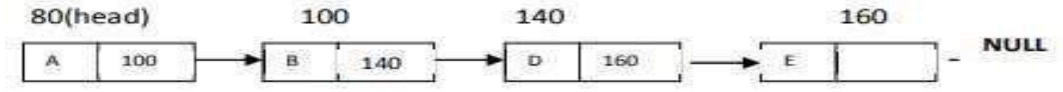
Procedure Delete_middle(n)

This algorithm deletes a node from the middle of the linked list. Temp is a temporary variable where the node is stored temporarily before deletion takes place. After is a node where the node to be deleted after that.

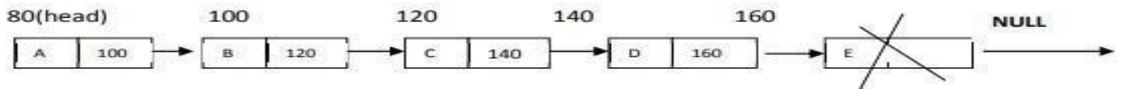
```
1. Temp=after→next
2. After→next=after→next→next
3. Delete temp
4. Exit
```

OR

```
1. nextnode=temp→next
2. temp→next=nextnode → next
3. Delete nextnode
4. Exit
```



Algorithm for deleting the last node form the linked list



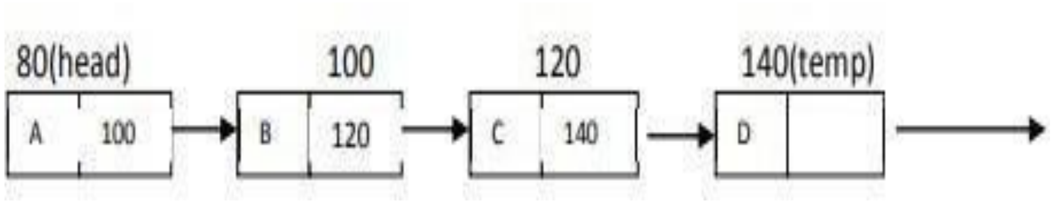
Procedure delete_end()

This algorithm deletes the last node of the linked list. Temp is a temporary variable where the head node is stored temporarily before deletion process takes place.

```
1. Temp=head
2. While(temp→next->next!=NULL)
   Begin
3. Temp=temp→next
   End
4. Delete temp→next
5. Temp→next=NULL
6. Exit
```

OR

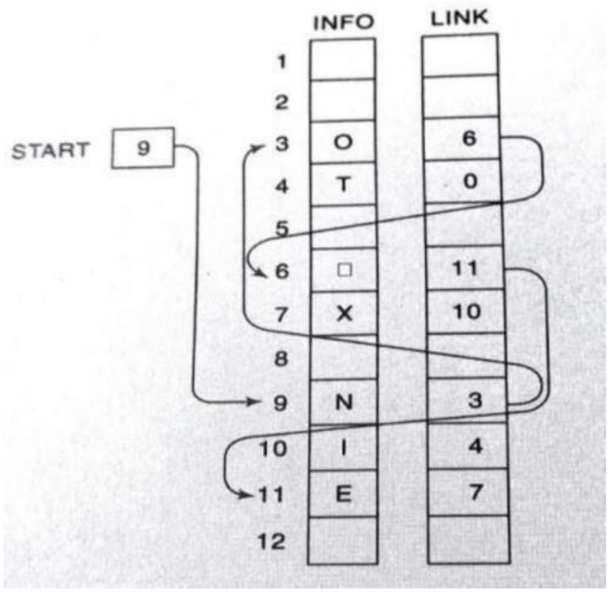
```
1. Temp=head
2. While(temp→next!=NULL)
   Begin
3. Previousnode=temp
4. Temp=temp→next
   End
5. Previousnode→next=NULL
6. Delete Temp
```



Representation of singly linked list in memory

Let List be a linked list. List requires two linear arrays- INFO and NEXT such that INFO and NEXT contain the information part and the next pointer field of a node of LIST.LIST also requires a variable name such as HEAD which contains the location of the beginning of the LIST and NULL which indicates the end of the LIST.

The following figure shows a linked list in memory where each node of the list contains a single character.



Head=9, so INFO[9]=N is the first character
NEXT[9]=3, so iNFO[3]=0 is the second character
Next[3]=6 so INFO[6]=blank is the third character
Link[6]=11, so info[11]=E
And so on.

Traversing a linked list

Let List be a linked list in memory. INFO is the pointer pointing to the Information part of a node and NEXT is the pointer which contains the address of the next node in the linked list. HEAD is the starting node.

Temp is a pointer variable which points to the node which is currently being processed. Temp next indicates the next node to be processed.

Procedure TRAVERSE (LIST, TEMP,INFO, HEAD,NEXT)

Let LIST be a linked list in memory. This algorithm traverses LIST, applying an operation DISPLAY to each element of LIST. The variable TEMP points to the node which is currently being processed.

```
1. Set Temp=head
2. While(temp!=NULL)
3. Begin
4. Display temp->info
   End
5. Set temp=temp->next
6. Finished
```

Searching a linked list

→

List is unsorted

Let List be a linked list in memory. INFO is the pointer pointing to the Information part of a node and NEXT is the pointer which contains the address of the next node in the linked list. HEAD is the starting node. Suppose a specific ITEM of information is given. This algorithm is used for finding the location LOCATION of the node where ITEM first appears in the LIST.

SEARCH(INFO, LINK, HEAD, ITEM, LOCATION)

LIST is a linked list in memory. This algorithm finds the location LOCATION of the node where ITEM first appears in LIST, or sets LOCATION=NULL.

```
1. Set temp=head
2. While(temp → next!=NULL)
   Begin
3. If ITEM=temp → info
4. Set LOCATION=temp and exit
5. Else
6. Set temp=temp → next
7. End if
   End while
8. Set LOCATION=NULL // search is unsuccessful
9. Exit
```

Memory allocation: Garbage collection

Together with the linked lists in the memory, a special list is maintained which consists of unused memory cells. This list, which has its own pointer, is called the list of available space or the free storage list or the free pool.

Suppose some memory space becomes reusable because a node is deleted from a list or an entire list is deleted from a program. We want the space to be available for future use. One way to do this is to immediately reinsert the space into the free storage list. But this method may be too consuming for the operating system of a computer and may choose an alternative method as follows.

The operating system of a computer may periodically collect all the deleted space onto the free storage list. Any technique which does this collection is called garbage collection. Garbage collection usually takes place into 2 steps. First the computer runs through all lists, tagging those cells which are currently in use, and then the computer runs through the memory, collecting all untagged space onto freestorage list.

The garbage collection may take place when there is only some minimum amount of space or no space at all left in the free storage list. Or when the CPU is idle and has time to do collection.

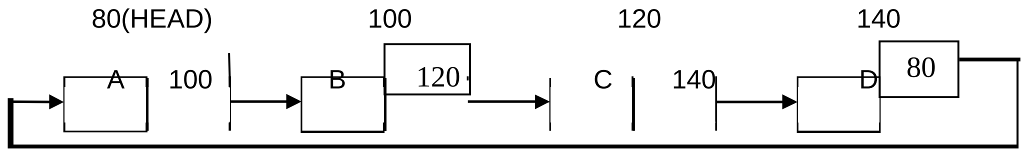
Overflow and underflow

Sometimes new data are to be inserted to a data structure but there is no available space i.e the free storage list is empty. This situation is usually called overflow. The programmer may handle overflow by printing the message overflow.

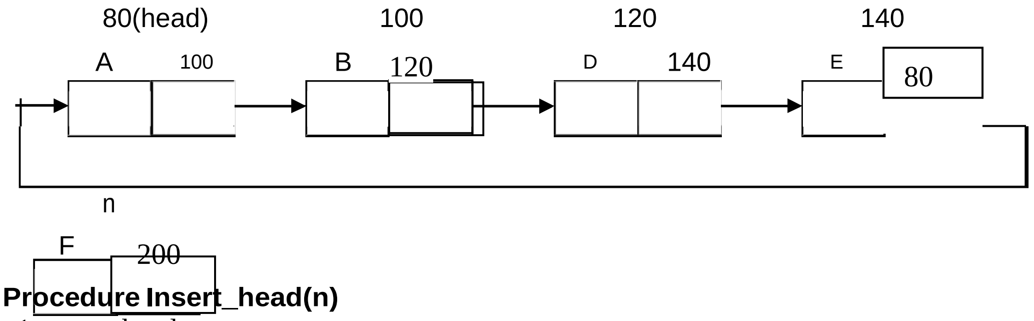
Similarly, the term underflow refers to the situation where one wants to delete data from a data structure that is empty. The programmer may handle underflow by printing the message underflow.

Circular linked list

It is just a singly linked list in which the link field f the last node contains the address of the first node of the list. That is the link field of the last node does not point to NULL. It points to back to the beginning of the linked list

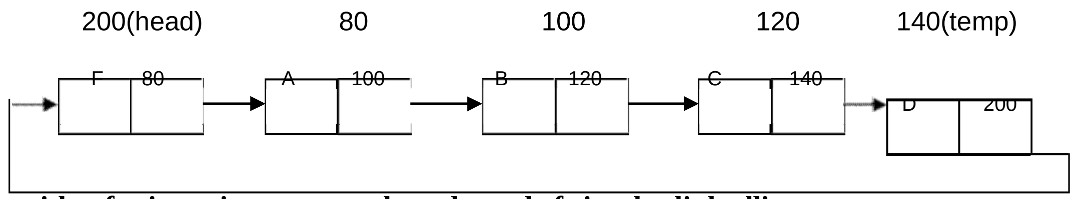


Algorithm for inserting a new node at the beginning of a circular linked list

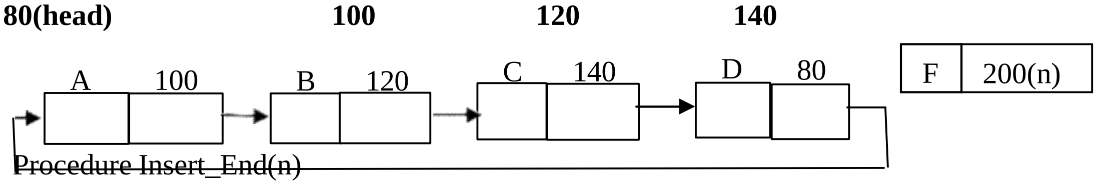


Procedure Insert_head(n)

```
1. temp=head
2.while(temp→next!=head)
  begin
3.temp=temp→next
  end
4.temp→next=n
5. n→next=head
6.head=n
7.finised
```



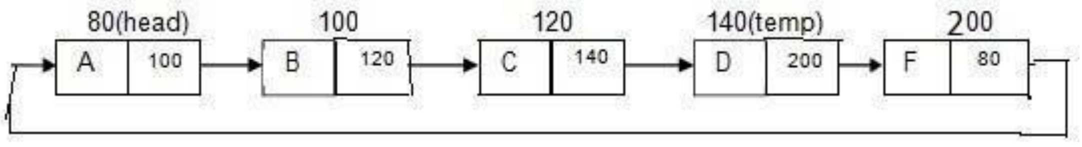
Algorithm for inserting a new node at the end of circular linkedlist



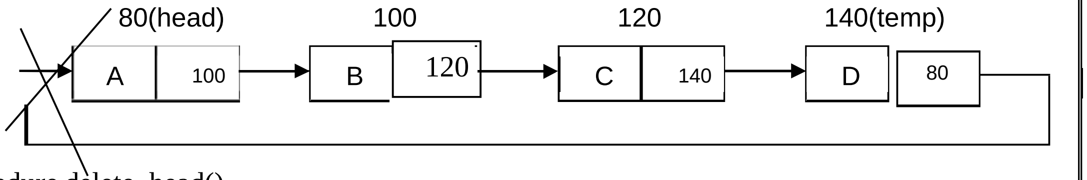
Procedure Insert_End(n)

```
1. temp=head
```

```
2. while(temp→next!=head)
   begin
3. temp=temp→next
   end
4. temp→next=n
5. n→next=head
6. finished
```

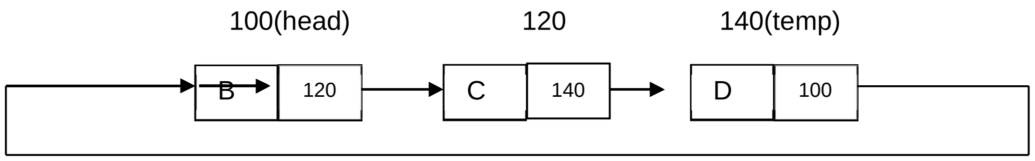


Algorithm for deleting a node at the beginning of a circular linked list

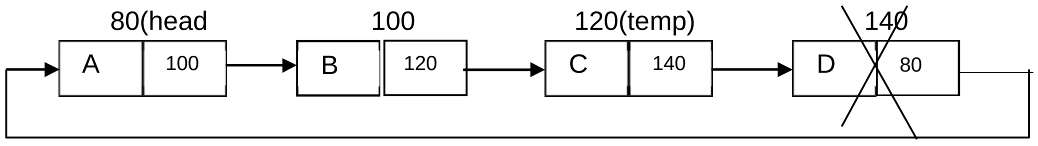


Procedure delete_head()

```
1. temp=head
2. while(temp→next!=head)
   begin
3. temp=temp→next
   end
4. temp→next=head→next
5. delete head
6. head=temp→next
7. finished
```



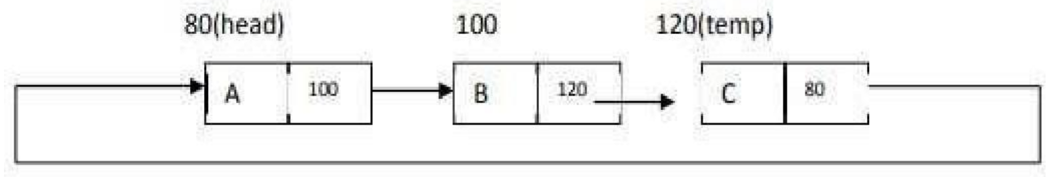
Algorithm for deleting a node at the end of a circular linked list



Procedure delete_end()

```
1. temp=head
2. while(temp→next→next!=head)
```

```
    begin
3.  temp=temp-> next
    end
4.  delete temp→next
5.  temp→next=head
6.  finished
```

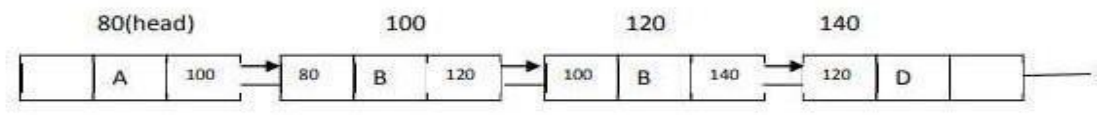


Doubly Linked List

One of the disadvantages of the singly linked list is that the inability to traverse the list in the backward direction. In most of the real-world applications it is necessary to traverse the list both in forward and backward direction. The most appropriate data structure for such an applicationis a doubly linked list.

A doubly linked list is a one in which all nodes are linked together by multiple number of links which help in accessing both the successor node and the predecessor node form the given node position.It provides bidirectional traversing.

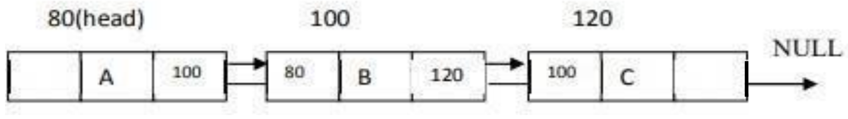
Each node in a double linked list has two link fields. These are used to point to a successor and thepredecessor nodes.

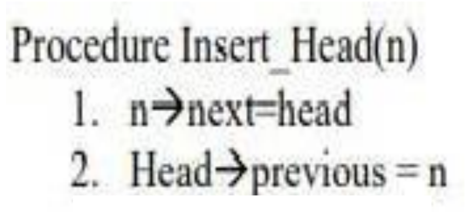


The *Prev* points to the predecessor (previous) node and the *Next* link points to the successor (next) node. The class definition for the above node in Object Oriented Programming is as follows.

```
Class Node
{
        Node  *Prev;
        Node  *Next;
        int Info;
};
```

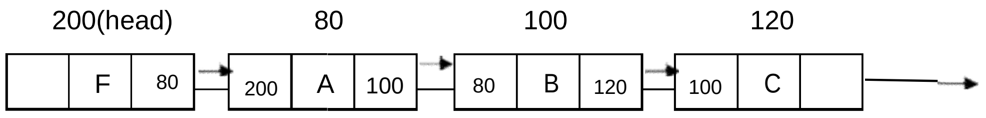
Algorithm to insert a node at the beginning of a doubly linked list.



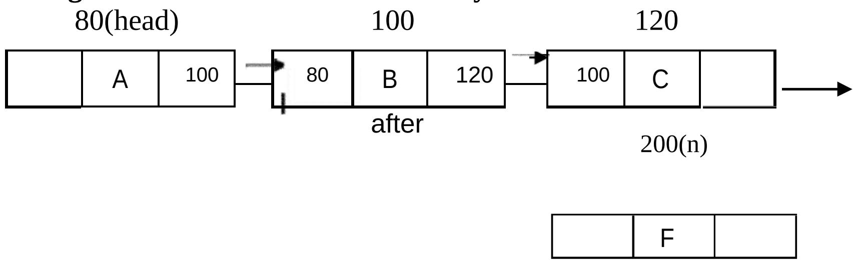

Procedure Insert_Head(n)

1. n→next=head
2. Head→previous = n

3. Head=n
4. Finished

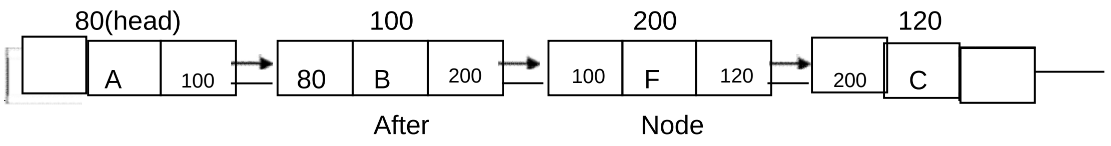


Inserting a node in the middle of a doubly linked list

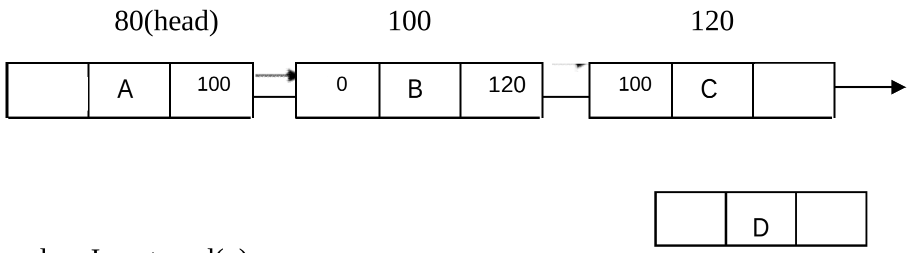


Procedure Insert_Middle(n)

```
1. n→next=after → next
2. After→next=n
3. n→previous=after
4. n→next→previous=n
5. Finished
```

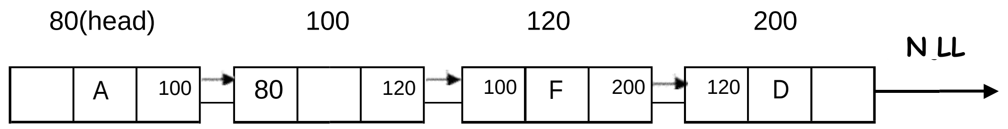


Inserting a node at the end of a doubly linked list

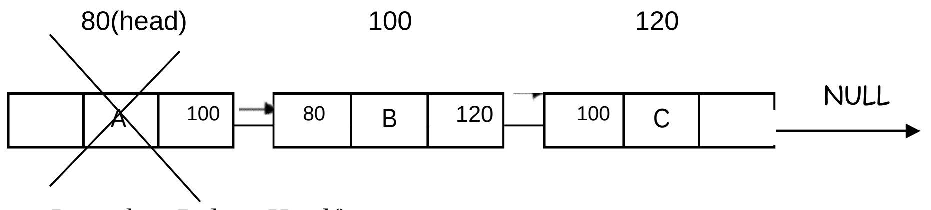


Procedure Insert_end(n)

```
1. temp=head
2. while(temp→next !=NULL)
   begin
3. temp=temp→next
   end
4. temp→next=n
5. n→previous=temp
6. n→next=NULL
7. finished
```

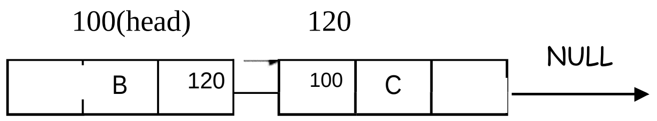


Deleting the head node from the doubly linked list

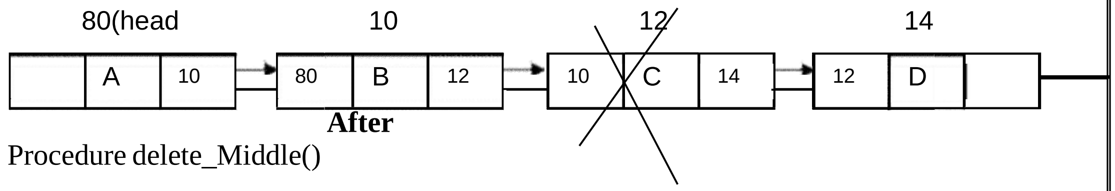


Procedure Delete_Head()

1. head=head→next
2. deletehead →previous
3. finished

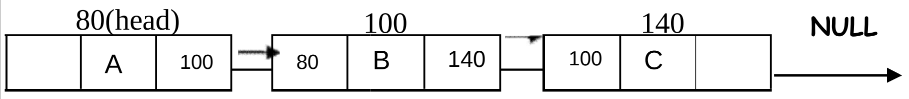


Deleting a node from the middle of the doubly linked list

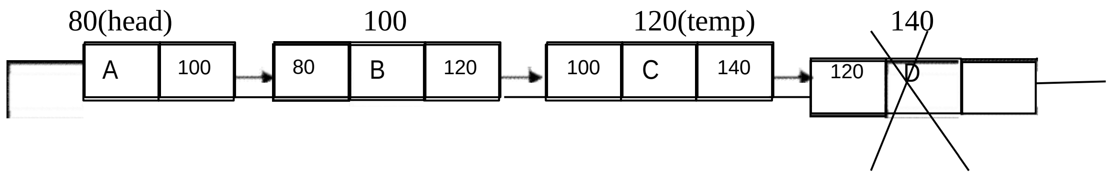


Procedure delete_Middle()

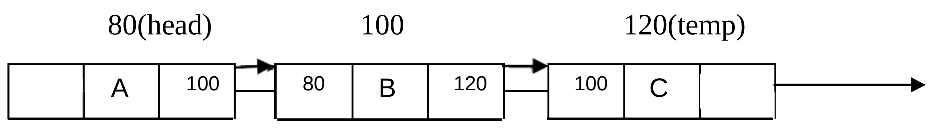
1. after→next=after→next→next
2. delete after →next→previous
3. after→next→previous=after
4. finished



Deleting a node at the end of the doubly linked list

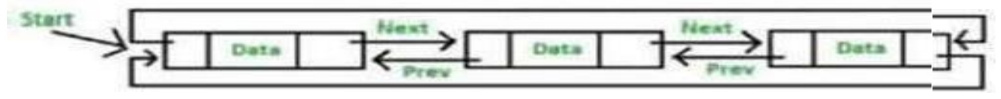


```
Procedure Delete_End()
    1. temp=head
    2. while(temp→next→next!=NULL)
       begin
    3. temp=temp→next
       end
    4. delete temp→next
    5. temp→next=NULL
    6. Finished
```



Doubly circular linked list

In circular doubly linked list two consecutive elements are linked or connected by previous and next pointer and the last node points to the first node also points to last node by previous pointer.



Question Bank

Multiple Choice Questions

(Questions for Understanding)

1. Each node in a doubly linked list consists of_____________fields.
   - A. One
   - B. Two
   - **C. Three**
   - D. Four

2. A linked list is an example of______________data structure.
   - A. Static
   - **B. Dynamic**
   - C. Heterogeneous
   - D. Sequential

3. The data field in a node of a linked list represents ___________
   - A. Address of the data
   - B. Link to the next node
   - **C. Value to be processed**
   - D. Memory location of the node

4. Which of the following is two-way lists?
   - A. Grounded header list
   - B. Circular header list
   - C. Linked list with header and trailer nodes
   - **D. List traversed in two directions**

5. Which of the following is not a disadvantage to the usage of array?
   - A. Fixed size
   - B. There are chances of wastage of memory space if elements inserted in an array are lesser than the allocated size
   - C. Insertion based on position
   - **D. Accessing elements at specified positions**

6. In linked lists there are no NULL links in ___________
   - A. single linked list
   - B. linear doubly linked list
   - **C. circular linkedlist**
   - D. linked list

7. Each node in a linked list must contain atleast______________
   - A. Three fields
   - **B. Two fields**
   - C. Four fields
   - D. Five fields

8. The dummy header in linked list contain_________________

   A. **<u>First record of the actualdata</u>**
   B. last record of the actual data
   C. pointer to the last record of the actual data
   D. middle record of the actualdata

9. In a linked list the____________field contains the address of next element in the list.

   A. **<u>Link field</u>**
   B. Next element field
   C. Start field
   D. Info field

10. LINK is the pointer pointing to the__________________

   A. Successor node
   B. **<u>predecessor node</u>**
   C. head node
   D. last node

11. ____________refers to a linear collection of data items.

   A. **<u>List</u>**
   B. Tree
   C. Graph
   D. Edge

12. A run list is_________________

   A. **<u>small batches of records from a file</u>**
   B. number of elements having same value
   C. number of records
   D. number of files in external storage

13. A linear list in which the pointer points only to the successive nodeis _______________

   A. **<u>singly linked list</u>**
   B. circular linked list
   C. doubly linked list
   D. none of the above

14. __________may take place only when there is some minimum amount (or) no space left in free storage list.

   A. Memory management
   B. **<u>Garbage collection</u>**
   C. Recycle bin
   D. Memory management

15. What is the return type of malloc() or calloc()?
   A. int*
   B. int **
   C. <u>void *</u>
   D. void **

(Questions for Application)

16. In circular linked list, insertion of node requires modification of?
   A. One pointer
   B. <u>Two pointer</u>
   C. Three pointer
   D. None of these

17. The function_____obtains block of memory dynamically.
   A. free
   B. malloc
   C. calloc
   D. <u>Both calloc and malloc</u>

18. The operation of processing each element in the list is known as _______________
   A. sorting
   B. merging
   C. inserting
   D. <u>traversal</u>

19. The situation when in a linked list START=NULL is ________________
   A. <u>Underflow</u>
   B. Overflow
   C. Houseful
   D. Saturated

20. Each node in singly linked list has_fields.
   A. **<u>2</u>**
   B. 3
   C. 1
   D. 4

21. Which header file should be included to use functions like malloc() and calloc()?
   A. dos.h
   B. <u>string.h</u>
   C. <u>stdlib.h</u>
   D. memory.h

22. Value of first linked list index is ______________
   A. **<u>0</u>**
   B. 1
   C. -1
   D. 2

23. A_________indicates the end of the list.
    - A. Guard
    - **B. Sentinel**
    - C. End pointer
    - D. Last pointer

24. A____________is a linear list in which insertions and deletions are made to from either endof the structure.
    - A. Circular queue
    - B. random of queue
    - C. priority
    - **D. dequeue**

25. Indexing the____________element in the list is not possible in linked lists.
    - **A. middle**
    - B. first
    - C. last
    - D. anywhere in between

Questions carrying 4 marks

(Questions for Understanding)

1. What do you mean by a linked list? With a neat diagram explain different types of linked lists.
2. Write an algorithm to insert nodes from the beginning and end of a singly linked list.
3. Write an algorithm to insert nodes from the middle (specified position), of a singly linked list.
4. Write an algorithm to search for an element in a linked list.
5. Write an algorithm to insert nodes from the beginning, and end of a circular linked list.

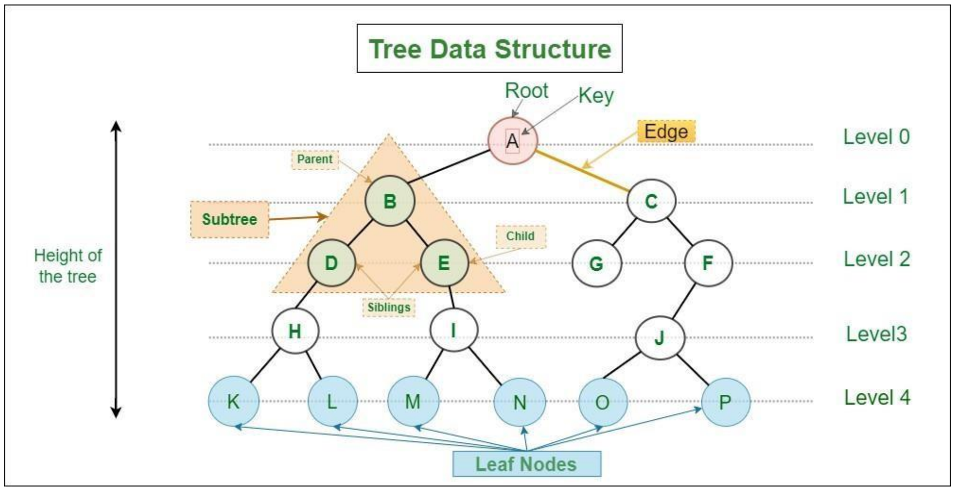
(Questions for Application)

1. Describe different functions in dynamic memory allocation.
2. Write an algorithm to traverse a linked list.
3. Write an algorithm to delete nodes from the beginning and end of a singly linked list.
4. Write an algorithm to delete nodes from the specified position (middle) of a singly linked list.
5. Write an algorithm to delete nodes from the, beginning of a circular linked list.
6. Write an algorithm to delete nodes from the end of a circular linked list.

UNIT IV
CHAPTER - 8
TREES

Introduction

A tree is a non-terminal data structure in which items are arranged in a sorted sequence. It is used to represent hierarchical relationship existing among several data items. The graph theoretic definition of a tree is: It is a finite set of one or more data items (nodes) such that there is a special data item called the root of the tree. And its remaining data items are partitioned into number of mutually exclusive subsets each of which is itself a tree they are called sub trees.



Binary Tree Terminology

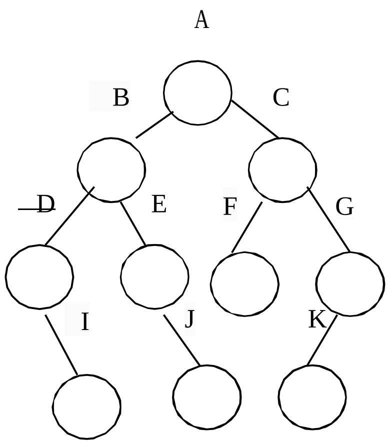
1. **Root**: It is the first in the hierarchical is the root item.
2. **Node**: Each data item in a tree is called a node. It specifies the data information andlinks to other data items. There are 16 nodes in the above tree.
3. **Degree of a node**: It is the number tree
   a. Degree of A is 2
   b. Degree of C is 2
   c. Degree of D is 1
   d. Degree of H is 2
   e. Degree of K is 0
4. **Degree of a tree**: It is the maximum degree of the nodes in a given tree. In the above tree node A has degree2, the B is having degree 3 and node J is having degree 2. So the degree of the tree is the maximum degree
5. **Terminal node**: A node with degree zero is called a terminal node or a leaf. In the above tree, there are 7 terminal nodes. They are K, L, M, N, G, O and P
6. **Non-terminal Nodes**: Any node except the root node whose degree is non-zero called a non-terminal node. There are 5 non-terminal nodes. They are B, C, D, E, F, H, I and J

7. **Siblings**: The children nodes of a given parent node are called siblings. They arealso called brothers. In the above tree,
    a. D & E are siblings of parent node B.
    b. G & F are siblings of parent node C..
8. **Level**: The entire tree structure is leveled in such a way that the root node is always at level 0, then the intermediate children are at level 1 and their intermediate children are at level 2 and so on. In the above tree, there are 4 levels.
9. **Edge**: It is the connecting line of 2 nodes. That is the line drawn from one node to anothernode is called an edge.
10. **Path**: It is the sequence of consecutive edges from the source node to the destination node.In the above tree, the path between A and J is given by the node pairs (A, C), (C, F), and(F,J)
11. **Depth**: It is the maximum level of any node in a given tree. In the above, tree root node A has the maximum level. The term height is also used to denote depth.
12. **Forest**: It is the set of disjoint trees. In a given tree, if you remove its root node the becomes a forest. In the above tree, there is a forest with 5 trees.

CHAPTER - 9 BINARY TREES

A binary tree is a finite set of data items that is either empty or consists of a single item called the root and 2 disjoint binary trees called the left subtree and the right subtree.

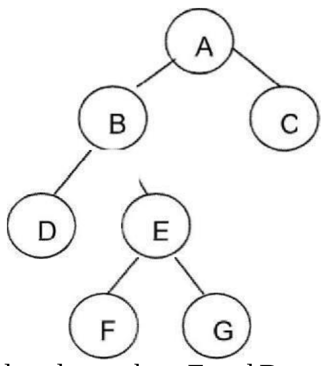
A binary tree is a very important and most commonly used nonlinear data structure. Ina binary tree, the maximum degree of any node is at most 2. That means there may be a zero-degree node or one-degree node and two-degree node.



A binary tree

In the above binary tree, A is the root of the tree. A left subtree consists of the tree with the root B and the right subtree consists of the tree with the root C. further has its leftsubtree with root D and right subtree with root E and so on.

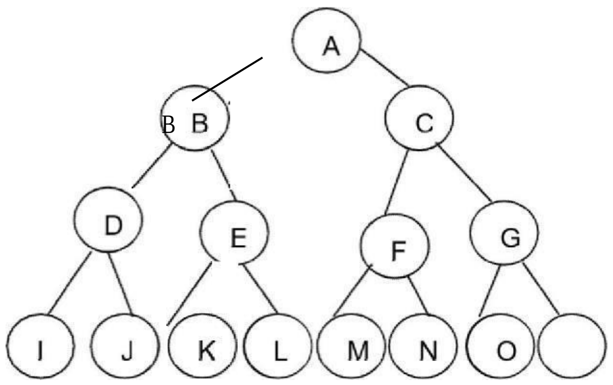
Strictly Binary tree - It is a binary tree, in which every node has either 0 or 2 children.



In this binary tree all non-terminal nodes such as E and B are non-empty left and right subtrees.

Complete binary tree

In a complete binary tree, there is exactly one node at level 0, 2 nodes at level 1, 4 nodes atlevel 2 and so on.



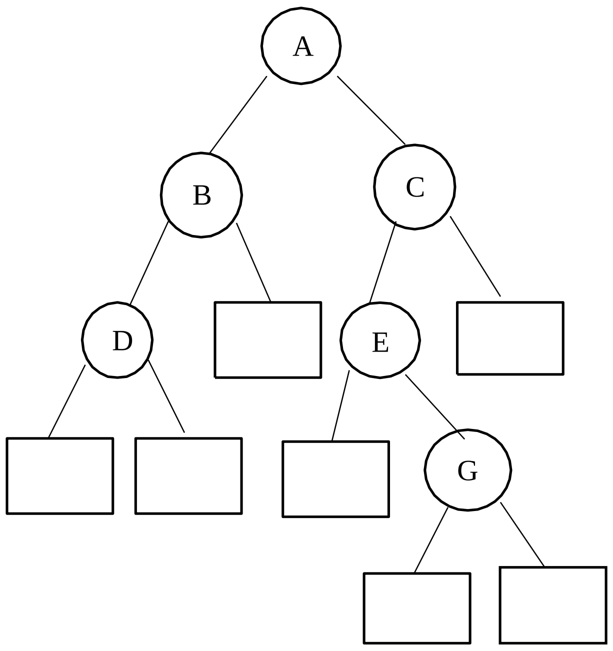
A complete binary tree with depth 4

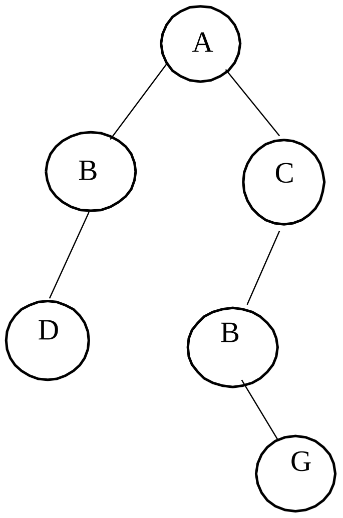
You can easily verify the no. of nodes at each level of a complete binary tree. For ex, at level 3, there will be 2^3=8 nodes.

Extended binary trees

A binary tree T is said to be a 2-tree or an extended binary tree if each node N has either 0 or 2 children. In such a case the nodes with 2 children are called internal nodes and the nobles with 0 children are called external nodes. Sometimes the nodes are distinguished in diagrams by using circles for internal nodes and squares for external nodes.

The term extended binary tree comes from the following operation. Consider any binary tree T as shown in the figure. Then T may be converted into a 2-tree by replacing each empty subtree by new node, as shown in the following diagram. The nodes in the original tree T are now the internal nodes in the extended tree and the new nodes are the external nodes in the extended tree.





Binary tree T

Extended 2- tree

Binary tree representation using linked list

Binary trees can be represented either using an array representation or using a linked list representation. The basic component to be represented in a binary tree is a node. The nodeconsists of 3 fields such as:

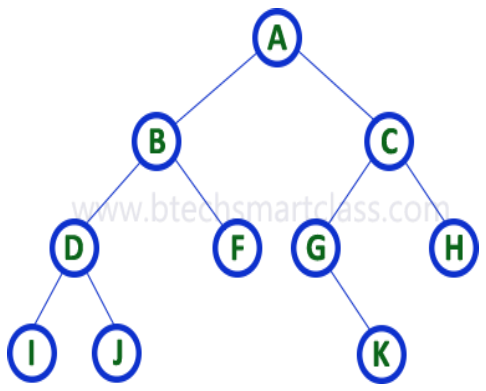
1) Data
2) Left child
3) Right child

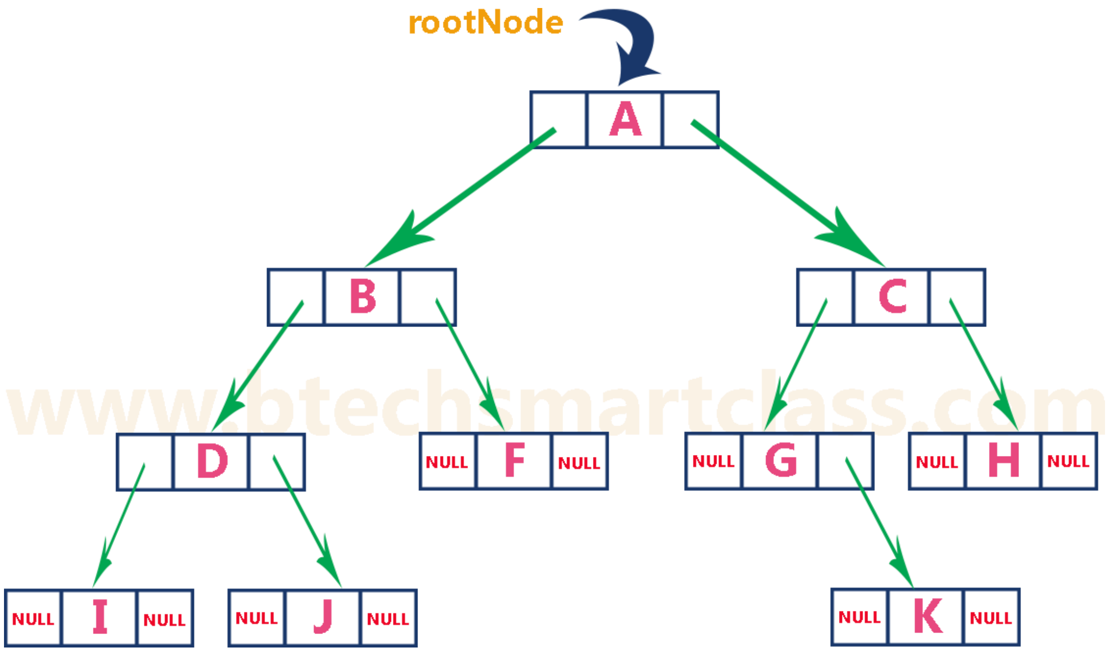
The data filed holds the value to be given. The left child is a link filed which containsthe address of its left node and the right child contains the address of its right node.

Ichild	Data	Rchild

```
class node
{
char data;
Node *lchild;
Node*rchild;
};
```

Consider a binary tree and its linked list representation is shown in the figure





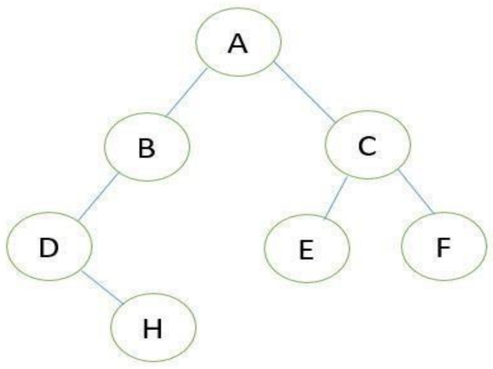
A Binary tree in array (sequential) representation

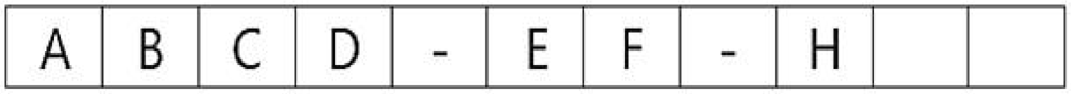
A binary tree contains one root node and some non-terminal and terminal nodes. It is
clear from the observation of a binary tree that the non-terminal nodes have their left child nodes. Their lchildand rchild pointer are setto NULL. Here, non-terminal nodes are called internal nodes andterminal nodes arecalled externalnodes.

Sequential representation of Binary trees

Suppose T is a binary tree that is complete or nearly complete. Then there is an efficient way of maintaining Tinmemorycalled the sequential representation of T. This representation usesonly a single linear arrayTREE as follows:

1) Theroot R of T is stored in TREE[1]
2) If a nodeN occupiesTREE[K],then itsleft child is stored in TREE[2*K] and itsright childis stored in TREE[2*K+1]





A	B	C	D	-	E	F	-	H		

Operations on Binary trees

The basic operations to be performed on a binary tree are listed below.
Create: It creates an empty binary tree.
Lchild: It returns a pointer to the left child of the node. If the node has no left child, it returns a NULL pointer.
Rchild: It returns a pointer to the right child of the node. If the node has no right child, it returns a NULL pointer.
Data: it returns the content of the node.

Apart from these primitive operations, other operations that can be applied to the binary tree are:

1) Tree traversal
2) Insertion of a node
3) Deletion of a node
4) Searching for a node
5) Copying the binary tree

Traversal of a binary tree

The traversal is one of the most common operations performed on tree data structure. It is a way in which each node in the tree is visited exactly once in a systematic manner. There are 3 popular ways of binary tree traversal. They are:-

1. Preorder traversal
2. Inorder traversal
3. Postorder traversal

Preorder traversal

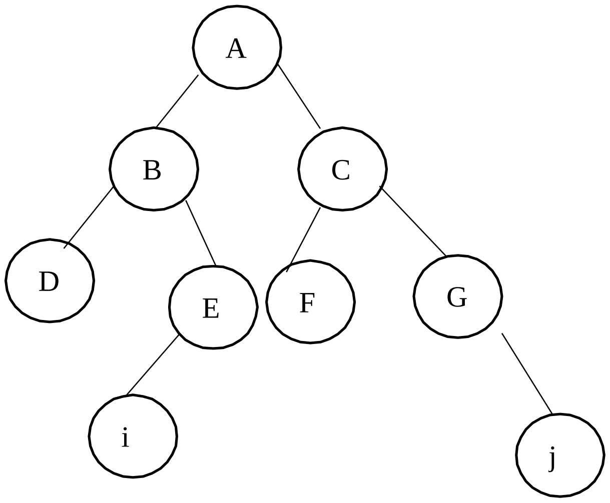
The preorder traversal of a non-empty binary tree is defined as follows.

1. Visit the root node
2. Traverse the left subtree in preorder
3. Traverse the right subtree inpreorder

In an preorder traversal
After visiting the root node, the left subtree is taken up and it is traversed recursively, then the right subtree is traversed recursively.

Algorithm for preorder traversal

```
Procedure Preorder(root)
1. If (root !=NULL)
2. Begin
3. Print root-> info
4. Preorder(root-> left)
5. Preorder(root-> right)
6. End
7. finished
```



The preorder traversal of the above binary tree is ABDEICFGJ

Inorder Traversal

The inorder traversal of

1) Traverse the left subtree in inorder
2) Traverse the root node in inorder
3) Traverse the right subtree ininorder

In an Inorder traversal
The left subtree is traversed recursively before node. After visiting the root node, the right subtree is taken up and it is traversed recursively.

Algorithm for inorder traversal

```
Procedure Inorder(root)
  1. If (root !=NULL)
  2. Begin
  3. Inorder(root ->left)
  4. Print -> info
  5. Inorder(root-> right)
  6. End
  7. Finished
```

The Inorder traversal of the above binary tree is DBIEAFCGJ

Postorder Traversal

The Postorder traversal of a non-root node is visited before traversing its left and right subtrees.

1) Traverse the left subtree
2) Traverse the right subtree
3) Traverse the root node

In a Postorder traversal the left and right subtrees are recursively processed before visiting the root. The left subtree is taken up first and is traversed postorder. Then the rightsubtree is taken up and is traversed in postorder. Finally, the data at the root is displayed.

Algorithm for Postorder order traversal

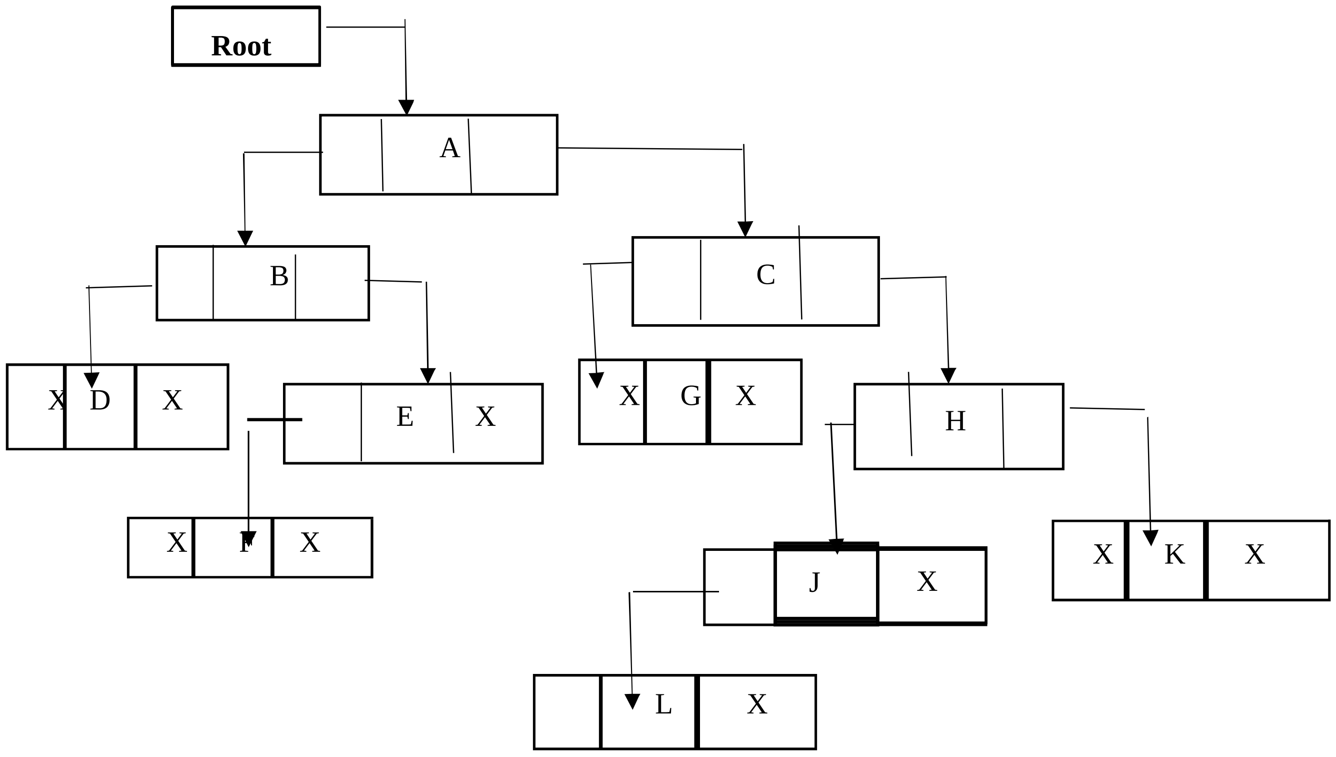
```
Procedure Postorder(root)
1. If (root!=NULL)
   Begin
2. postorder(root ->left)
3. postorder(root ->right)
4. Print root-> info
   End
5. Finished
```

The Postorder traversal of the above binary tree is DIEBFJGCA

Header nodes: threads

Suppose a binary tree T is maintained in memory by means of a linked representation. Sometimes an extra special node called a header node is added to the beginning of T. When this extra node is used, the tree pointer variable which we call HEAD instead of ROOT, will point to the header node and the left pointer of point to the root of T.

For example,



Suppose a binary tree T is empty. Then T will still contain a header node, but the leftpointer of the header node will contain the null value. Thus the condition, head ->left=NULL will indicate an empty tree.

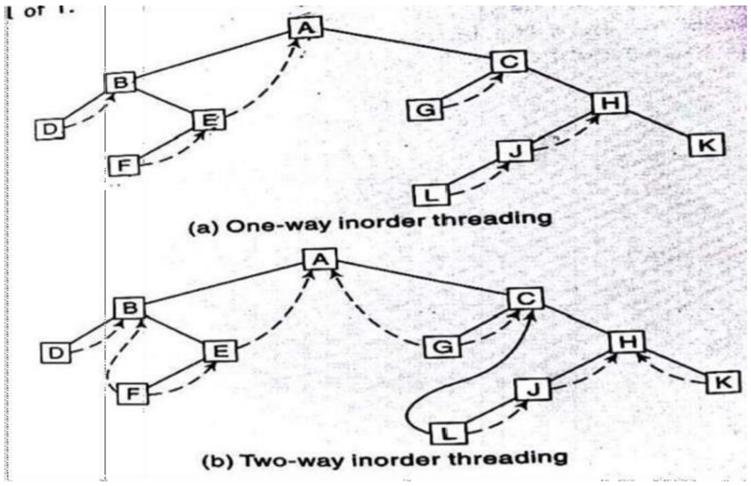
Threads; Inorder threading

Consider again the linked list representation of a binary tree T. Half of the entries in the pointer will fields LEFT and RIGHT will contain null elements, this is space may be more replace efficiently used by replacing the null entries by some other type of information. Specifically, we will replace certain null entries by special pointer which points are higher nodes in the tree.

These special pointers are called threads and binary trees with suchpointers are called threaded trees.

The threads in a diagram of a threaded tree are usually indicated by dotted lines.
There are many ways to thread a binary tree T,but each threading will correspond to a particular traversal of T, also one may choose a one way threading or two way threading . Unless otherwise stated , our threading will correspond to the inorder traversal of T. In the oneway threading of T , a thread will appear in the right filed of a node and will point to the next node in the inorder traversal of T and in two way threading of T, a thread will also appear in the LEFT field of a node and will point to the preceding node in the inorder traversal of T.

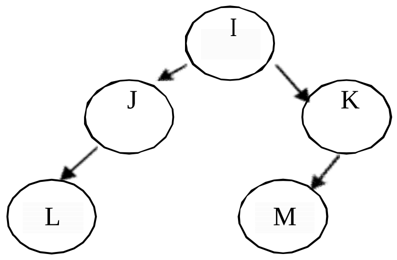
Consider the following example



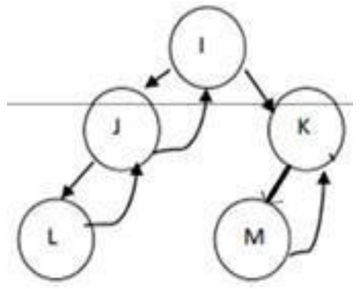
Threaded binary trees

A threaded binary tree is a binary tree in which the nodes that do not have a right child, have a thread to their inorder successor. By doing this type of threading, we avoid the recursive method of traversing a tree, which uses stacks and also wastes a lot of memory andtime.

Consider a binary tree given below.

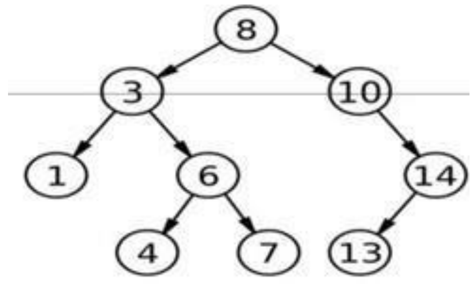


Let us make a thread binary tree out of a normal binary tree



Binary Search Tree BFS

A binary search tree is a binary tree in which for each node in the tree, the elements inthe left subtree are less than the root and the elements in the right subtree are greater than or equal to the root.



Inserting in a Binary Search tree

```
1. Create a new BST node and assign values to it
   Node=newnode
   Node→data=item
   Node→left=Node→right=NULL
   Return Node
2. Insert(node, key)
   i) if root == NULL,
       return the new node to the calling function
   ii) if (item>root→data)
       call the insert function with root=>right and assign the return value in root=>right
       root->right = insert(root=>right,key)
   iii) else
       call the insert function with root->left and assign the return value in root=>left
       root=>left = insert(root=>left,key)
2. Finally, return the original root pointer to the calling function
```

Searching in a Binary Search tree

```
1. Search (root, item)
2. Step 1 - if (item = root → data) or (root = NULL)
3. return root
4. else if (item < root → data)
5. return Search(root → left, item)
```

6. else
7. return Search(root → right, item)
8. END if
9. Step 2 - END

Deleting in a Binary Search tree

Suppose T is a binary search tree and suppose an Item of information is given. This section gives an algorithm which deletes ITEM from the tree T. the way node N deleted from the tree depends primarily on the number ofchildren node N.

They are three cases:

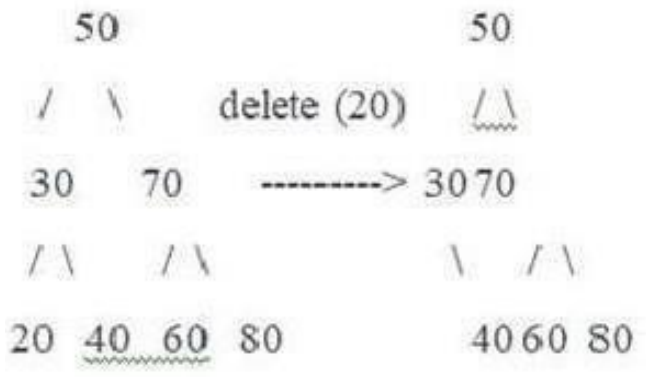
Case 1: N has no children. Then N is deleted from the tree T by simply replacing the location of N in the parent node P(N) by the NULL pointer.
Case 2: N has exactly one child. Then N is deleted from the tree T by simply replacing the location of N inP(N) by the location of the only child of N.
Case 3: N has two children. Let S(N) denote the inorder successor of N. then N is deleted from T by first deleting S(N) from T and then replacing node N in T by the node S(N).

For example, consider the binary search tree in the following figure.
Suppose T appears in memory as shown below.

When we delete a node, three possibilities arise.

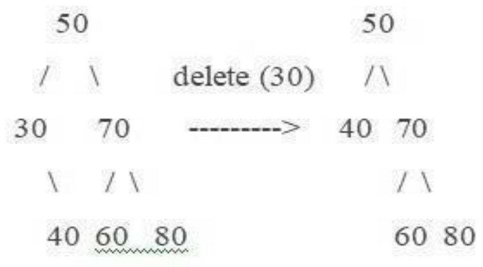
1) *Node to be deleted is leaf:*

Simply remove from the tree.



2) *Node to be deleted has only one child:*

Copy the child to the node and delete the child



```
   50                       50
  /   \        delete (30)     /\
30     70       --------->   40  70
  \     / \                      / \
  40  60  80                     60 80
```

3) *Node to be deleted has two children:*

Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

```
   50                        60
  /    \      delete (50)     /  \
40      70      --------->   40    70
        / \                          \
      60 80                           80
```

The important thing to note is, inorder successor is needed only when right child is not empty. In this particularcase, inorder successor can be obtained by finding the minimum value in right child of the node.

Question Bank

Multiple Choice Questions

(Questions for Understanding)

1. The number of edges from the node to the deepest leaf is called________________of the tree.
   - **A. Height**
   - B. Depth
   - C. Length
   - D. Width

2. Binary trees with threads are called as________________
   - **A. Threaded trees**
   - B. Pointer trees
   - C. Special trees
   - D. Special pointer trees

3. What will be the height of a balanced full binary tree with 8 leaves?
   - A. 8
   - B. 5
   - C. 6
   - D. **4**

4. In Binary trees nodes with no successor are called________________
   - A. End nodes
   - **B. Terminal nodes**
   - C. Final nodes
   - D. Last nodes

5. Trees are said______________if they are similar and have same contents at corresponding nodes.
   - A. Duplicate
   - B. Carbon copy
   - C. Replica
   - **D. Copies**

6. Every node N in a binary tree T except the root has a unique parent called the of N.
   - A. Antecedents
   - **B. Predecessor**
   - C. Forerunner
   - D. Precursor

7. Sequential representation of binary tree uses_______
   - **A. Array with pointers**
   - B. Single line array
   - C. Two dimensional arrays
   - D. Three dimensional arrays

8. A binary tree whose every node has either zero or two children is called________________

   A. complete binary tree
   B. binary search tree
   C. extended binary tree
   D. data structure

9. In a binary-tree, nodes with 0 children are called____________

   A. Exterior node
   B. Outside node
   C. Outer node
   D. External node

10. Which indicates pre- order traversal?

    A. Left sub-tree, Right sub-tree and root
    B. Right sub-tree, Left sub-tree and root
    C. Root, Left sub-tree, Right sub-tree
    D. Right sub-tree, root, Left sub-tree

11. The balance factor of a node in a binary tree is defined as ______

    A. addition of heights of left and right subtrees
    B. height of right subtree minus height of left subtree
    C. height of left subtree minus height of right subtree
    D. height of right subtree minus one

12. TREE [1] =NULL indicates tree is_______________

    A. Overflow
    B. Underflow
    C. Empty
    D. Full

13. Linked representation of binary tree needs_____________parallel arrays.

    A. 4
    B. 2
    C. **3**
    D. 5

14. The maximum number of nodes in a tree for which post-order and pre-order traversals may be equal is______.

    A.3
    B. 1
    C. 2
    D. any number

15. A terminal node in a binary tree is called_________________

    A. Root
    B. Leaf

C. Child
D. Branch

16. A binary tree is balanced if the difference between left and right subtree of every node is not more than____.

A. **<u>1</u>**
B. 3
C. 2
D. 0

17. The number of edges from the root to the node is called____________of the tree.

A. Height
B. <u>Depth</u>
C. Length
D. Width

18. What is a full binary tree?

A. <u>Each node has exactly zero or two children</u>
B. Each node has exactly two children
C. All the leaves are at the same level
D. Each node has exactly one or two children

19. Which of the following is not an advantage of trees?

A. Hierarchical structure
B. Faster search
C. Router algorithms
D. <u>Undo/Redo operations in a notepad</u>

20. What is a threaded binary tree traversal?

A. a binary tree traversal using stacks
B. a binary tree traversal using queues
C. a binary tree traversal using stacks and queues
D. <u>a binary tree traversal without using stacks and queues</u>

21. What are the disadvantages of normal binary tree traversals?

A. <u>there are many pointers which are null and thus useless</u>
B. there is no traversal which is efficient
C. complexity in implementing
D. improper traversals

22. Three standards ways of traversing a binary tree T with root R ....

A. Prefix, infix, postfix
B. Pre-process, in-process, post-process
C. Pre-traversal, in-traversal, post-traversal
D. <u>Pre-order, in-order, post-order</u>

23. What is the possible number of binary trees that can be created with 3 nodes, giving the sequence N, M, L when traversed in post-order.
    A.15
    B.3
    C.**5**
    D. 8

24. Which of the following tree traversals work if the null left pointer pointing to the predecessor andnull right pointer pointing to the successor in a binary tree?
    A. inorder, postorder, preorder traversal
    B. inorder
    C. postorder
    D. preorder

25. What is the maximum number of children that a binary tree node can have?
    A. 0
    B. 1
    C. **2**
    D. 3

Questions carrying 4 marks

(Questions for Understanding)

1. What do you mean by the following:
    a. Complete binary tree
    b. Forest
    c. Level of a tree
    d. Depth of a tree
2. What do you mean by the following:
    a. Degree
    b. Terminal and Non-Terminal Node
    c. Path
    d. Siblings
3. Explain the sequential representation of a binary tree.
4. Explain the linked list representation of a binary tree.
5. What do you mean by the binary search tree? Write the algorithm to insert and search for a node in BST.
6. Explain the concept of threaded binary trees.

(Questions for Application)

1. Write the algorithm for preorder traversal of a tree.
2. Write an algorithm for inorder traversal of a tree.
3. Write an algorithm for postorder traversal of a tree.
4. Construct a binary search tree for the following:
   14, 15, 4, 9, 7, 18 and traverse it in Inorder, postorder and preorder.
5. Construct a binary search tree for the following data.
   66, 26, 22,34,47,79,48,32,78

UNIT V
CHAPTER – 9. SORTING

Introduction

Sorting and searching are the fundamental operation in computer science. Sorting refers to the operation of arranging data in some given order, such as increasing and decreasing. Searching refers to the operation of finding the location of a given item in a collection of items.

Sorting

Definition: - Arranging the elements in some type of order.

Ordering or sorting of data with some relationship is of fundamentals importance. Certain factors however should be considered before designing a sort algorithm. Algorithms are designed with following objectives.

1. The movement of data should be as minimum as possible. If the size of the data item is large and if there is excessive movement of data, this would result in a large amountof processing time.
2. The movement of data items between the secondary storage and the main memory should be in large blocks. The larger the data block the more efficient is the process.
3. As much as possible the data should be retained in the main memory, so that if it can be effectively used.

Different Sorting Techniques

-
- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort

Bubble Sort

This is the most popular of all sorting algorithm because it is very simple to understand and implement this algorithm. The algorithm achieves its name from the fact that with, each iteration a number moves like a bubble to its appropriate position. However algorithm is not efficient for large arrays. The method of bubble sort relies heavily on exchange mechanism to achieve its goal. The method is also called as "Sorting by Exchange".

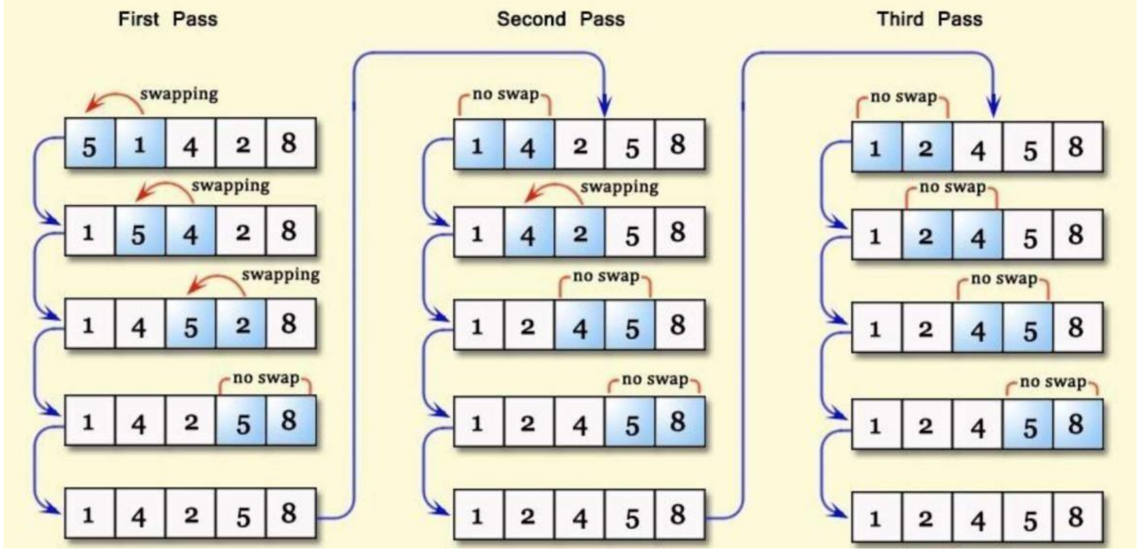
Suppose the list of number A[1], A[2]...A[N] is in memory. The bubble sort Algorithm works as follows.

Step 1: compare A[1] and A[2] and arrange them in the desired order so that A[1] < A[2]. Then compare A[2] with A[3] and arrange them so that A[2]< A[3]. Continue until we compare A[N-1] with A[N] and arrange them so that A[N-1] < A[N]. During step 1 the largest element is bubble up to the A[N] position. And step 1 involves n-1 comparisons.

Step 2: repeat step 1 with one less comparison; that is now we stop after we compare and possibly arrange A[N-2] and A[N-1] at the end of step 2, the second largest element in the array will occupy the A[N-1] position.

Step N-1: compare A[1] with A[2] and arrange them so that A[1] < A[2]. After step N-1 steps, the list will be sorted in increasing order.

Illustration



Algorithm

Given an array A of N elements, this procedure sorts the elements in the ascending order using the method described above. The variable I and J are used to index the array elements.

Bubble_Sort (A, N)

```
Step 1: for I= 1 to N-1 do
Step 2: for J = 0 to N-I-1 do
Step 3: [Compare adjacent elements]
        If A[J] > A[J+1] then
Step 4: Temp = A[J]
Step 5: A[J] = A[j+1]
Step 6: A[J+1] = Temp
        [End If]
        [End of Step 2 for loop]
        [End of Step 1 for loop]
Step 7: Exit
```

Selection Sort

This is a sorting algorithm. This is very simple to understand and implement. The algorithm achieves its name from the fact that with each iteration the smallest element from the key position is selected from the list of remaining elements and put in the require position of the array i.e. we start the search assuming that the current element is the smallest until we find the an element smaller that it and then interchange the elements. Thealgorithm however is not efficient for large arrays. The method of selection sort relies heavily on a comparison mechanism to achieve its goal.

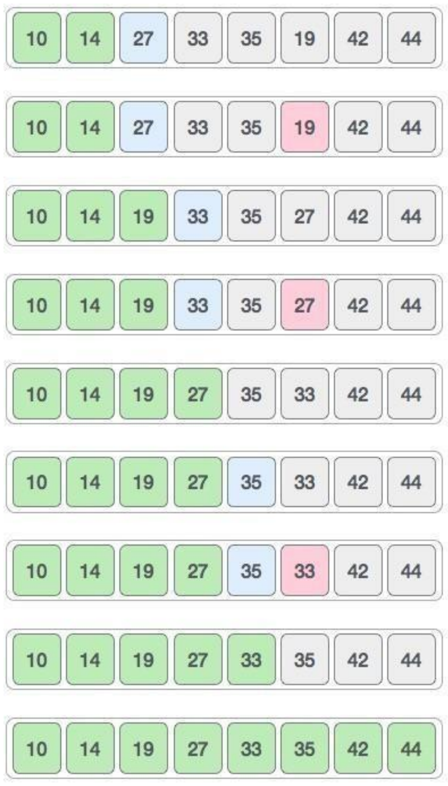
We start the sort assuming that the current element is the smallest until we find an element smaller that it and then interchange the elements. Suppose an array A with n

Pass 1: find the location POSITION of the smallest element in the list of N elements and then interchange A[POSITION] and A[1], then A[1] is sorted

Pass 2: find the location POSITION of the smallest element in the sub list of N-1 elementsand then interchange A[POSITION] and A[2] then, A[1] and A[2] are sorted since A[1]
<=A[2]

Pass 3: find the location POSITON of the smallest element in the sub list of N-2 elements and then interchange A[POSITION] and A[3] then, A[1], A[2], A[3] is sorted since A[2]<=A[1]. And so on. Thus A is sorted after N-1 passes.

Illustration



Algorithm

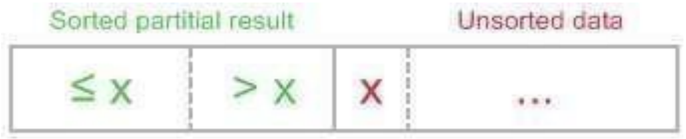
A is an array of N elements. This algorithm finds the smallest element SMALL and its location POSITION among the elements in the array A.

Procedure SelectionSort(A, N)

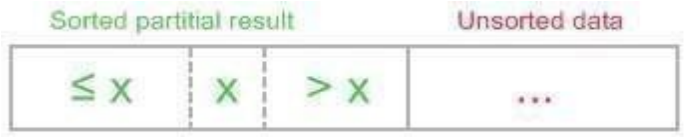
```
Step 1: repeat steps 2, 3 and 4, 6, 7 for i=1 to N-1
        begin
Step 2: set SMALL:=A[I]
Step 3: set POSITION:=I
Step 4: repeat step 5 for J=I+1 to N
        begin
Step 5: if A[J] < SMALL
        begin
            a.  set SMALL:=A[J]
            b.  set POSITION:=J
    4.  [End of if structure]
        [end of inner for loop]
    5.  Step 6: Set A[POSITION]:=A[I]
    6.  Step 7: Set A[I]:=SMALL
    7.  [end of outer for loop]
        Step 8: Exit
```

Insertion Sort

Insertion sort algorithm somewhat resembles selection sort. Array is imaginary divided into two parts - sorted one and unsorted one. At the beginning, sorted part contains first element of the array and unsorted one contains the rest. At every step, algorithm takes first element in the unsorted part and inserts it to the right place of the sorted one. When unsorted part becomes empty, algorithm stops. Sketchy, insertion sort algorithm step looks like this:



Becomes



Suppose an array A with n elements A[1], A[2], …A[N] is in memory. The insertion sort algorithm scans the array A from A[1] to A[N] , inserting each element into its proper position.

The algorithm of insertion sort functions as follows. Initially to start the whole array is in a completely unordered state. The first element is considered to be in the ordered list. The second element is considered to be in the unordered list. The second element is then inserted either in the first or in the second position as appropriate. Now there are 2 elements in the ordered part and remaining elements are considered to be unordered. Inserting the third element, then the fourth and so on slowly extends the orderedpart.

Pass 1: A[1] by itself is sorted because it is the first element
Pass 2: A[2] is inserted either before of after A[1] so that now A[1] , A[2] are sorted.
Pass 3: A[3] is inserted into its proper place in A[1], A[2] that is, before A[1], between A[1] and A[2] or after A[2] so that A[1], A[2], a[3] are sorted.
Pass 4: A[4] is inserted into its proper place in A[1], A[2], A[3] so that: A[1], A[2], A[3] and A[4] is sorted.
Pass N: A[N] is inserted into its proper place in A[1]. A[2], A[3]…A[N-1] so that A[1], A[2]…A[N] are sorted.

Algorithm – I

Procedure InsertionSort(A, N)
This algorithm sorts an A with N elements

Step 1: repeat step 2 for i=1 to N-1

```
Step 2: repeat step 3 for j=I to 0
Step 3: if A[J] < A[J-1]
begin
        a.  set TEMP:=A[J]
        b.  set A[J]:=A[J-1]
        c.  set A[J-1]:=TEMP
[end of if structure]
[end of inner for loop]
[end of outer for loop]
```

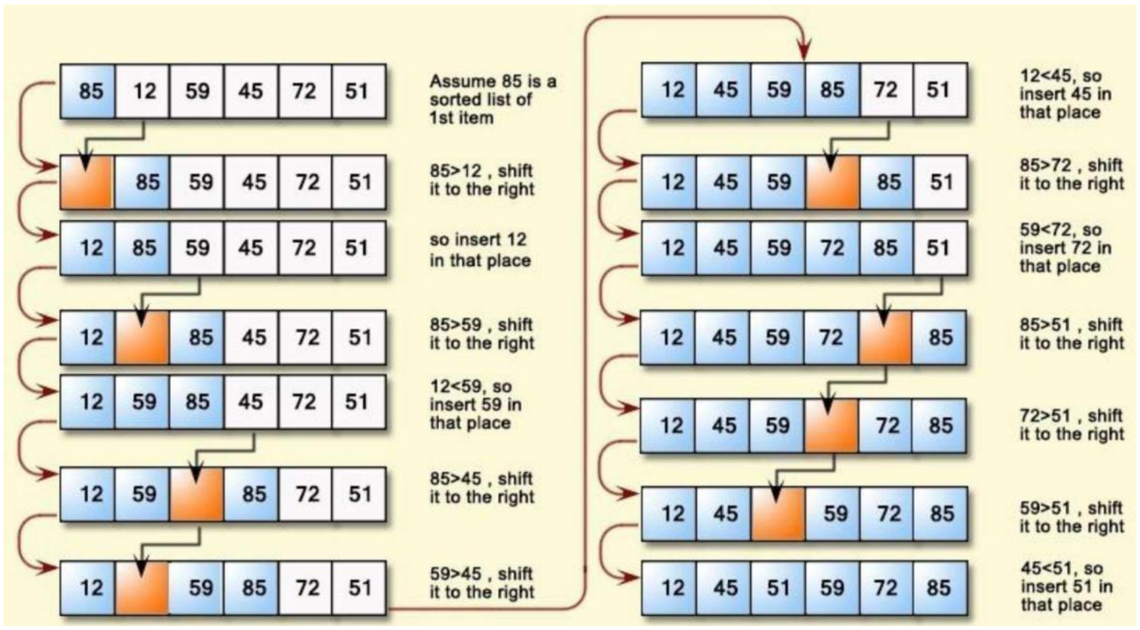
Or

Algorithm – II

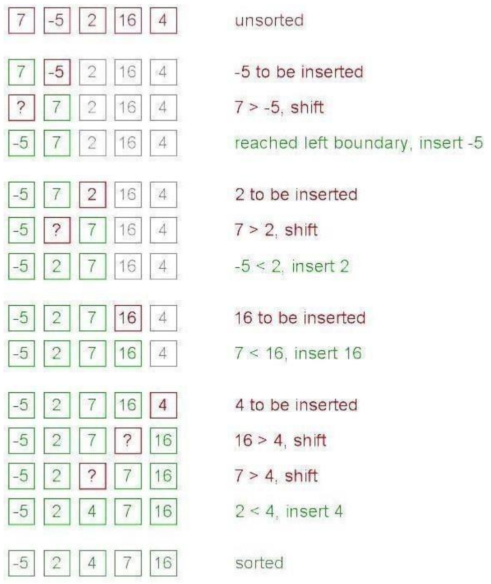
Given an array A of N an element, this procedure sorts the element in the ascending order. The variables I and J are used to index the array elements.

Insertion _Sort (A, N)

```
Step 1: For I =1 to N do
Step 2: TEMP=A[I]
Step 3: J=1-1
Step 4: While (J>=0) and (A[J]>TEMP)
Step 5: A[J+1]=A[J]
Step 6: J=J-1
        [End of while loop]
Step 7: A[J+1]=TEMP
        [end of step 1 for loop]
Step 7: Exit
```

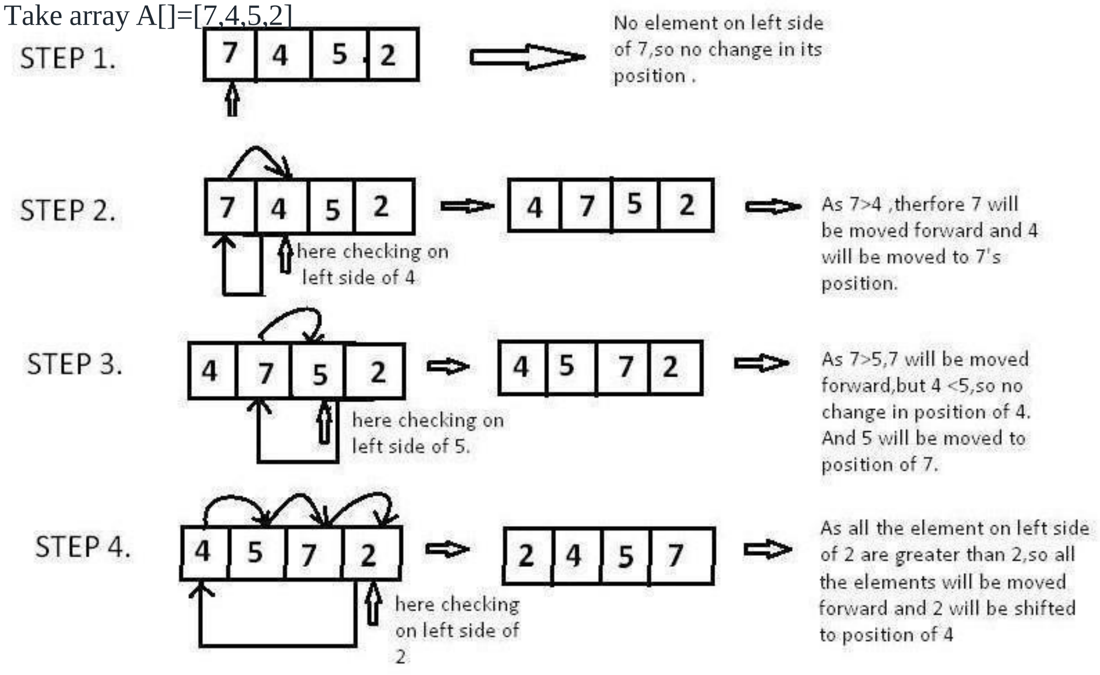

85 12 59 45 72 51
Assume 85 is a sorted list of 1st item
85 59 45 72 51
85>12 , shift it to the right
12 85 59 45 72 51
so insert 12 in that place
12 85 45 72 51
85>59 , shift it to the right
12 59 85 45 72 51
12<59, so insert 59 in that place
12 59 85 72 51
85>45 , shift it to the right
12 59 85 72 51
59>45 , shift it to the right
12 45 59 85 72 51
12<45, so insert 45 in that place
12 45 59 85 51
85>72 , shift it to the right
12 45 59 72 85 51
59<72, so insert 72 in that place
12 45 59 72 85
85>51 , shift it to the right
12 45 59 72 85
72>51 , shift it to the right
12 45 59 72 85
59>51 , shift it to the right
12 45 51 59 72 85
45<51, so insert 51 in that place

Let us see an example of insertion sort routine to make the idea of algorithm clearer. *Example,* Sort {7, -5, 2, 16, 4} using insertion sort.


7 -5 2 16 4 unsorted
7 -5 2 16 4 -5 to be inserted
? 7 2 16 4 7 > -5, shift
-5 7 2 16 4 reached left boundary, insert -5
-5 7 2 16 4 2 to be inserted
-5 ? 7 16 4 7 > 2, shift
-5 2 7 16 4 -5 < 2, insert 2
-5 2 7 16 4 16 to be inserted
-5 2 7 16 4 7 < 16, insert 16
-5 2 7 16 4 4 to be inserted
-5 2 7 ? 16 16 > 4, shift
-5 2 ? 7 16 7 > 4, shift
-5 2 4 7 16 2 < 4, insert 4
-5 2 4 7 16 sorted

Example 2:

Take array A[]=[7,4,5,2]


STEP 1.
7 4 5 2
No element on left side of 7,so no change in its position .
STEP 2.
7 4 5 2
4 7 5 2
here checking on left side of 4
As 7>4 ,therfore 7 will be moved forward and 4 will be moved to 7's position.
STEP 3.
4 7 5 2
4 5 7 2
here checking on left side of 5.
As 7>5,7 will be moved forward,but 4 <5,so no change in position of 4. And 5 will be moved to position of 7.
STEP 4.
4 5 7 2
2 4 5 7
here checking on left side of 2
As all the element on left side of 2 are greater than 2,so all the elements will be moved forward and 2 will be shifted to position of 4

Since 7 is the first element has no other element to be compared with, it remains at its position. Now when on moving towards 4, 7 is the largest element in the sorted list and greater than 4. So, move 4 to its correct position i.e. before 7. Similarly with 5, as 7 (largest element in the sorted list) is greater than 5; we will move 5 to its correct position. Finally for 2, all the elements on the left side of 2 (sorted list) are moved one position forward as all are greater than 2 and then 2 is placed in the first position. Finally, the given array will result in a sorted array.

Merge Sort

In this technique, a single array is divided into two sub lists. The two sub lists are sorted. i.e. elements from low to mid are sorted and elements from mid+1 to high are sorted. But, the elements from low to high are unsorted. The efficiency of merge sort is $O(n \log_2 n)$.

10	20	30	40	50	15	25	35	45

low mid mid+1 high

Hence we merge the two sorted sub lists into a single sorted array. If low and high are lower and upper limits in an array then, the general procedure to implement merge sort is as follows:

```
If(low≤high)
1.  Divide the array into equal parts
2.  Sort the left part of the array recursively
3.  Sort the right part of the arrayrecursively
4.  Merge the left and right parts
End if
```

Algorithm:

Algorithm MergeArray(A, LOW,MID,HIGH). This algorithm merges two sorted arrays where the first array is from LOW to MID and the second array is from MID+1 to HIGH.

```
1.  I←LOW, J←MID+1, K←LOW
2.  While (I≤MID and J≤HIGH) repeat step3
3.  If (A[I]<A[J])
    Then C[K] ←A[I], I←I+1,
    K←K+1 Else C[K]=A[J], J←J+1,
    K←K+1
4.  While(I≤MID) repeat steps 5,6
5.  C[K] ←A[I]
6.  K←K+1, I←I+1
7.  While(J≤HIGH) repeat steps 8,9
8.  C[K] ←A[J]
9.  K←K+1, J←J+1
10. For I=LOW to HIGH repeat step 11
11. A[I] ←C[I]
12. Return.
```

Algorithm for Merge Sort:

Algorithm MergeSort(A, LOW, HIGH). The purpose of this algorithm is to sort the elements of array A between the lower and upper bounds LOW and HIGH respectively.

```
1.  If(LOW≠HIGH) then perform steps 2-5
2.  MID←(LOW+HIGH)/2
```

```
3.  MergeSort(A,LOW,MID)
4.  MergeSort(A,MID+1,HIGH)
5.  MergeArray(A,LOW,MID,HIGH)
6.  Return
```

Example:

```
60 50 25 10 35 20 75 30
60  50  25  10              35 20 75  30               DIVIDE
 60  50       25  10        35  20       75  30
 60    50     25     10     35     20    75     30
50    60     10    25              20    35     30    75   MERGE
10  25  50   60             20 30 35  75
10   20   25   30    35   50    60  75
```

Illustration:

As one may understand from the image above, at each step a list of size M is being divided into 2 sub lists of size M/2, until no further division can be done. To understand better, consider a smaller array A containing the elements (9, 7, 8).

At the first step this list of size 3 is divided into 2 sub lists the first consisting of elements (9, 7) and the second one being (8). Now, the first list consisting of elements (9,7) is further divided into 2 sub lists consisting of elements (9) and (7)respectively.

As no further breakdown of this list can be done, as each sub list consists of a maximum of 1 element, we now start to merge these lists. The 2 sub-lists formed in the last step are then merged together in sorted order using the procedure mentioned above leading to a new list (7, 9). Backtracking further, we then need to merge the list consisting of element (8) too with this list, leading to the new sorted list (7, 8, 9).

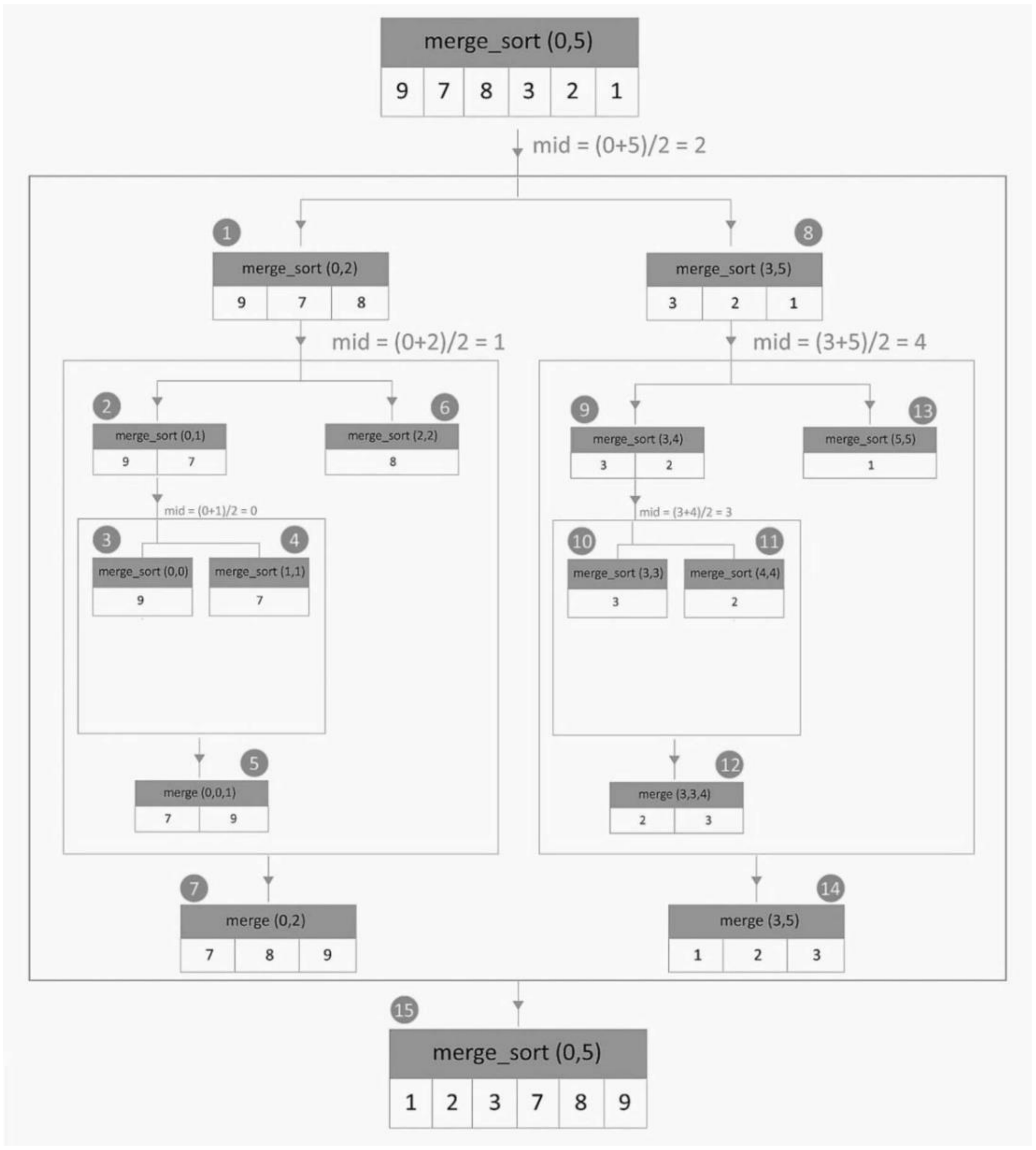


Illustration:

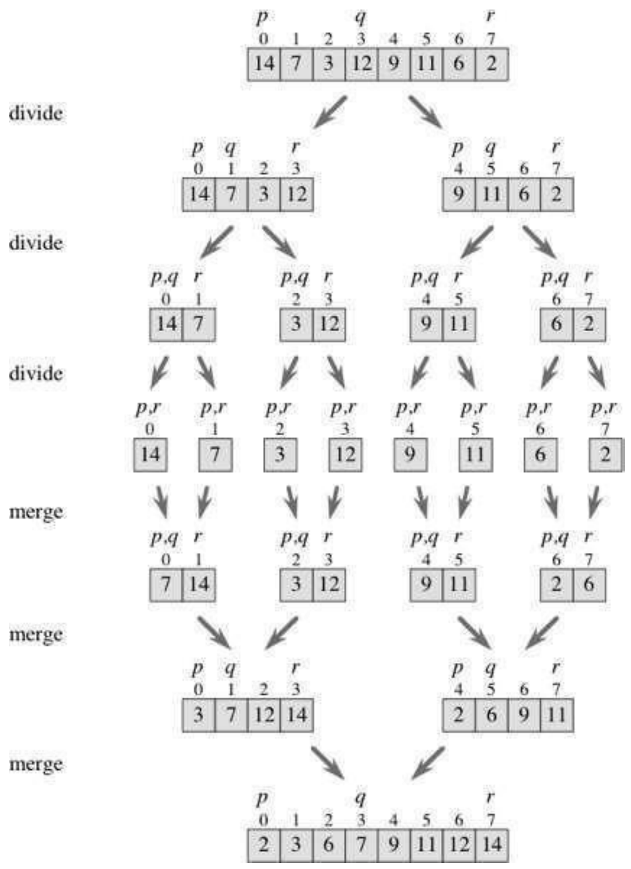
Let's see an example. Let's start with array holding [14, 7, 3, 12, 9, 11, 6, 2], so that the first subarray is actually the full array, array [0..7] (p=0 and r=7). This subarray has at least two elements, and so it's not a base case.

- In the divide step, we compute q = 3.
- The conquer step has us sort the two subarrays array [0..3], which contains [14, 7, 3, 12], and array[4..7], which contains [9, 11, 6, 2]. When we come back from the conquer step, each of the two subarrays is sorted: array [0..3] contains [3, 7, 12, 14] and array[4..7] contains [2, 6, 9, 11], so that the full array is [3, 7, 12, 14, 2, 6, 9, 11].
- Finally, the combine step merges the two sorted subarrays in the first half and the second half, producing the final sorted array [2, 3, 6, 7, 9, 11, 12, 14].

How did the subarray array [0..3] become sorted? The same way. It has more than two elements, and so it's not a base case. With p=0 and r=3, compute q=1, recursively sort array[0..1] ([14, 7]) and array[2..3] ([3, 12]), resulting in array[0..3] containing [7, 14, 3, 12] , and merge the first half with the second half, producing [3, 7, 12,14].

How did the subarray array [0..1] become sorted? With p=0 and r=1, compute q=0, recursively sort array [0..0] ([14]) and array[1..1] ([7]), resulting in array[0..1] still containing [14, 7], and merge the first half with the second half, producing [7, 14].

The subarrays array [0..0] and array[1..1] are base cases, since each contains fewer than two elements. Here is how the entire merge sort algorithm unfolds:



Quick Sort Algorithm

Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes n log n comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

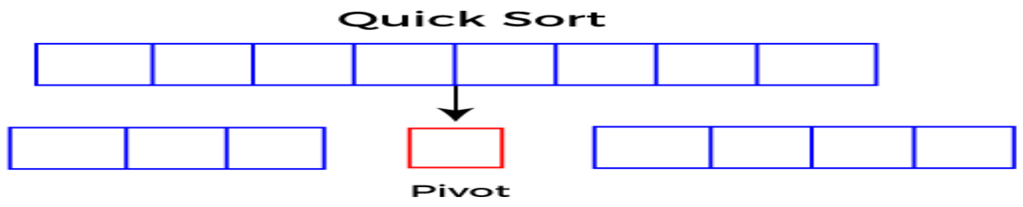
Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

Algorithm:

```
1. QUICKSORT (array A, start, end)
2.  {
3.     if (start < end)
4.     {
5.          p = partition(A, start, end)
6.          QUICKSORT (A, start, p - 1)
7.      QUICKSORT (A, p + 1, end)8.
         }
9. }
```

PARTITION ALGORITHM:

```
1.  PARTITION (array A, start, end)
2. {
3.     pivot = A[end]
4.     i = start-1
5.     for j = start to end -1
6. {
7.     do if (A[j] < pivot)
8.    {
9.     then i = i + 1
10.   swap A[i] with A[j]
11.   }
12. }
13.   swap A[i+1] with A[end]
14.  return i+1
15. }
```

OR

Partition(A, LOW,HIGH) This function partitions the array A with LOW and HIGH as lower bound and upper bound respectively into two sub lists.

```
1.  KEY←A[LOW], I←LOW+1, J←HIGH
2.  Repeat steps 3-7
3.  Repeat step 4 while(I<HIGH and KEY≥A[I])
4.  I←I+1
5.  Repeat step 6 while (KEY<A[J])
6.  J←J-1
7.  If(I<J)
    Then A[I]↔A[J]
    Else A[LOW]↔A[J] , Return J
8.  Return
```

Algorithm QuickSort(A, LOW, HIGH). This is a recursive algorithm to sort the elements in array A with LOW and HIGH as lower and upper bound respectively.

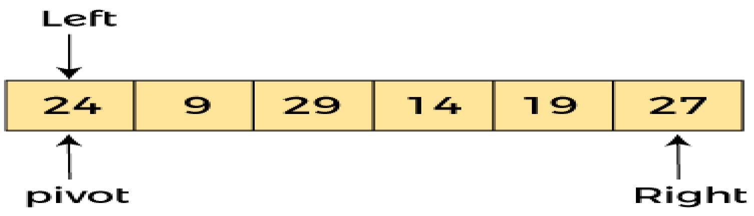
```
1.  If (LOW<HIGH) repeat steps 2-4
2.  J←Partition(A,LOW,HIGH)
3.  QuickSort(A,LOW,J-1)
4.  QuickSort(A,J+1,HIGH)
5.  Return
```

WORKING:

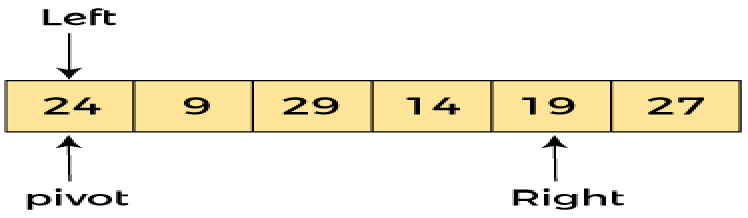
24	9	29	14	19	27

In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

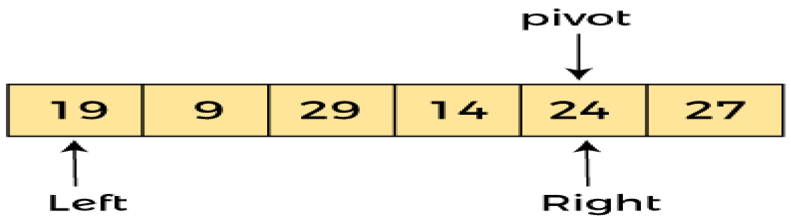


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. –



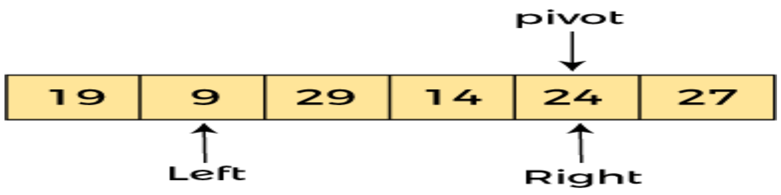
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as –

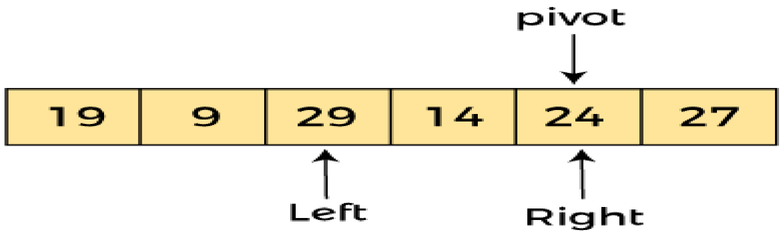


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

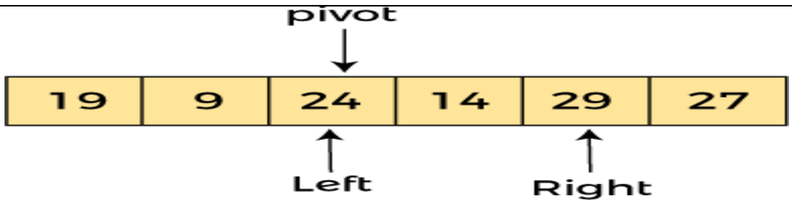
As a[pivot] > a[left], so algorithm moves one position to right as -



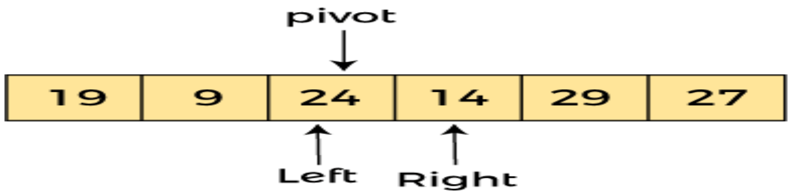
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



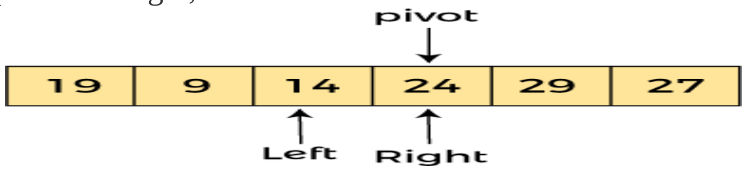
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



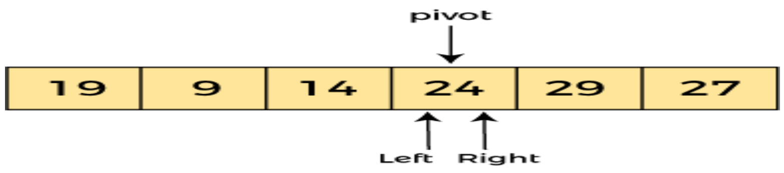
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.

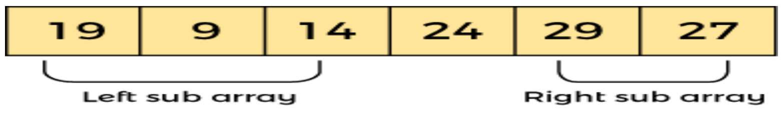


Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element.

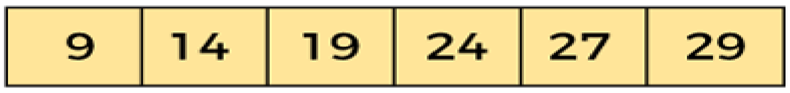
It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays.
After sorting gets done, the array will be -



UNIT V
CHAPTER – 10. SEARCHING

Introduction

The process of identifying or finding a particular record is called Searching. You often spend time in searching for any desired item. If the data is kept properly in sorted order, then searching becomes very easy and efficient.

Searching

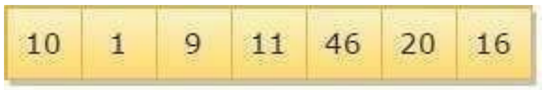
Searching is an operation or a technique that helps finds the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching technique that is being followed in data structure is listed below:

- Linear Search or Sequential Search
- Binary Search

Linear Search

This is the simplest method for searching. In this technique of searching, the element to be found in searching the elements to be found is searched sequentially in the list. This method can be performed on a sorted or an unsorted list (usually arrays). In case of a sorted list searching starts from 0th element and continues until the element is found from the list or the element whose value is greater than (assuming the list is sorted in ascending order), the value being searched is reached.

As against this, searching in case of unsorted list also begins from the 0th element and continues until the element or the end of the list is reached.

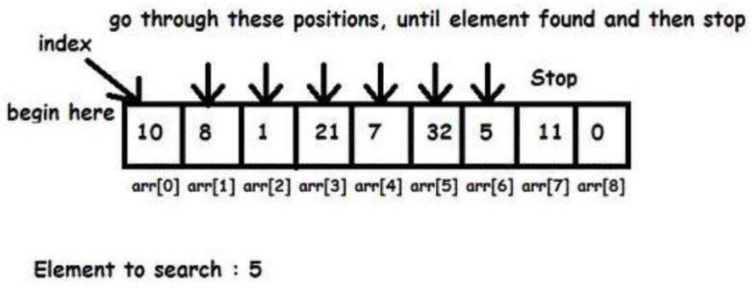


One-Dimensional Array having 7 Elements

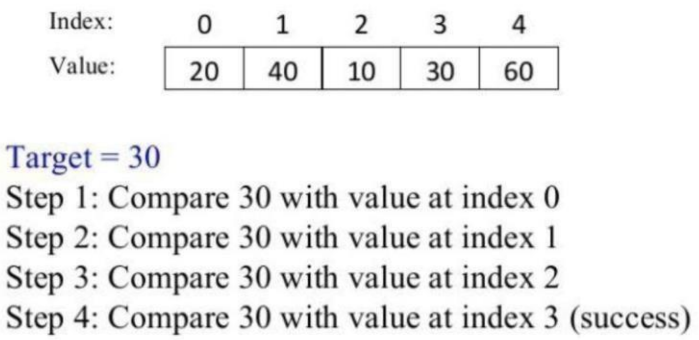
Example, The list given below is the list of elements in an unsorted array. The array contains 10 elements. Suppose the element to be searched is '46', so 46 is compared with all the elements starting from the 0th element and searching process ends where 46 is found or the list ends.

The performance of the linear search can be measured by counting the comparisons done to find out an element.

Illustration

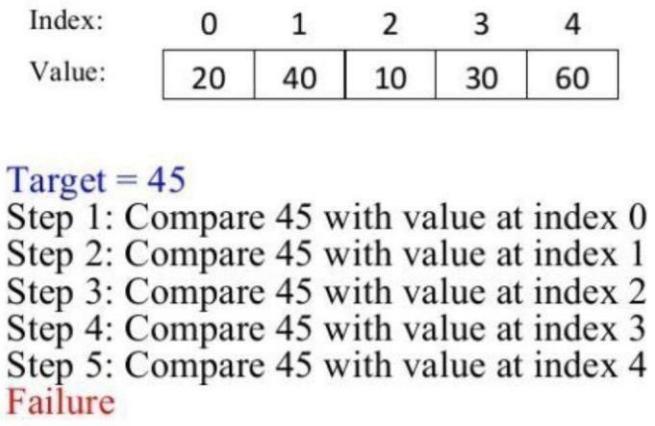


Illustration



Index:	0	1	2	3	4
Value:	20	40	10	30	60

Target = 30
Step 1: Compare 30 with value at index 0
Step 2: Compare 30 with value at index 1
Step 3: Compare 30 with value at index 2
Step 4: Compare 30 with value at index 3 (success)



Index:	0	1	2	3	4
Value:	20	40	10	30	60

Target = 45
Step 1: Compare 45 with value at index 0
Step 2: Compare 45 with value at index 1
Step 3: Compare 45 with value at index 2
Step 4: Compare 45 with value at index 3
Step 5: Compare 45 with value at index 4
Failure

Suppose A is a linear array with N elements. To search for a given ELEMENT in A is to compare ELEMENT with each element of A one by one. That is, first we test whether A[1]= ELEMENT and then we test whether A[2]= ELEMENT and so on. This method which traverses the array A sequentially to locate ELEMENT is called linear search or sequential search.

Algorithm

Linear_Search(A,element,N)

A is an array of N elements and element is the element being searched in the array.

```
Step 1: Set Loc:=-1
Step 2: Repeat step3 For i=0 to n-1
Step 3: If (Element=A[i]) then
        begin
            i.   Set loc:= i
```

```
                ii.  Goto step 4
            [End if]
            [End for loop]
    Step 4: If (loc>=0) then
                i.   write('element found in location', loc+1)
                ii.  Else
                iii. Write('element not found')
    Step 5: Exit
```

Binary Search

Binary search is a very fast and efficient searching technique. It requires the list to be in sorted order. In this method, to search an element you can compare it with the present element at the center of the list. If it matches, then the search is successful otherwise the list is divided into two halves: one from the 0th element to the middle element which is the center element (first half) another from the center element to the last element (which is the 2nd half) where all values are greater than the center element.

The searching mechanism proceeds from either of the two halves depending upon whether the target element is greater or smaller than the central element. If the element is smaller than the central element, then searching is done in the first half, otherwise searching is done in the second half.

This is another method of accessing a list. The entries in a list are stored in the increasing order. This is an efficient technique for searching an ordered sequential list of elements. In this method, we first compare the key with the element in the middle position of the list. If there is a match, then the search is successful. If the element is less than the middle key, the desired element must lie in the lower half of the list. if it is greater, then the desired element will be in the upper half of the list. We repeat this procedure on the lower half or upper half the list.

Algorithm

Binary_search(A, element, N)

A is a list of N elements and the element is the element being searched in the array. LOW and HIGH identify the positions of the 1st and last elements in a range and MID identifies the position the middle element.

```
    Step 1: Set LOW:= 0
    Step 2: Set HIGH:= N-1
    Step 3: Set LOC:= -1
    Step 4: Repeat steps 5 and 6, 7 While LOW<=HIGH
            Begin
    Step 5: Set MID:=LOW+HIGH)/2
    Step 6: IF element=A[MID]
            Begin
                a.  Set LOC:= MID
                b.  goto step 8
            [END IF]
```

```
Step 7: IF element<A[MID]
            i.   Set HIGH:=MID-1
            ii.  Else
            iii. Set LOW:=MID+1
        [END IF]
        [END of While loop]
Step 8:     1.  Write("Element found in location", LOC)
            2.  ELSE
            3.  Write("Element not found")
        [END IF]
Step 9: EXIT
```

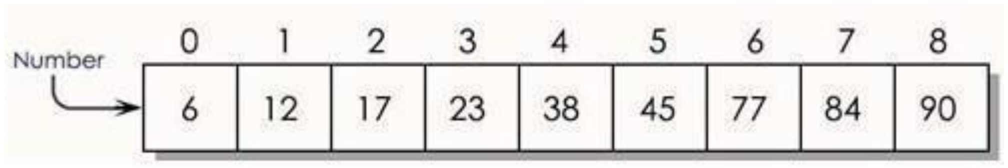
Limitations of Binary search algorithm

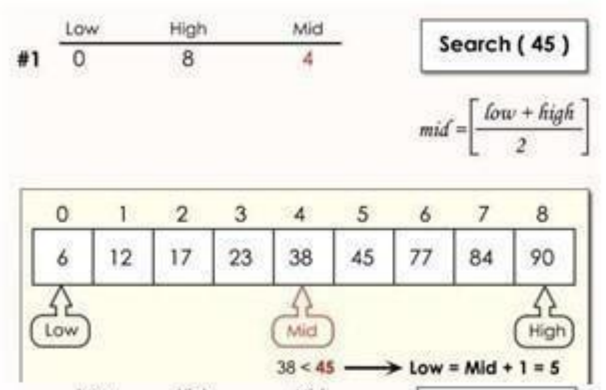
The binary search algorithm requires 2 conditions

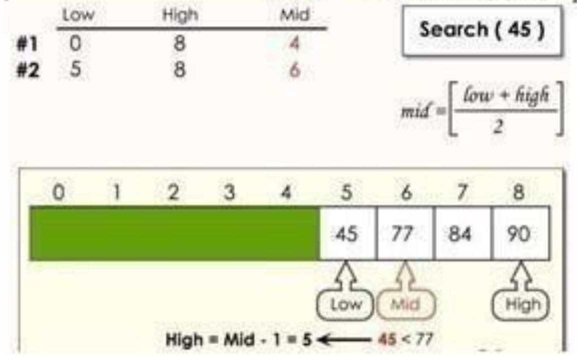
1) The list must be sorted
2) One must have direct access to the middle element in any sub list.

This means that we must use a sorted array to hold the data. But keeping data in a sorted array is normally expensive when there are many insertions and deletions. In such situations, one may use a different data structure such as a linked list or a binary search tree to store data.

Illustration

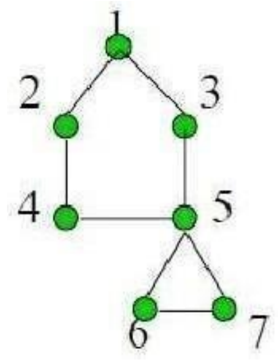






UNIT - V
CHAPTER - 12
GRAPHS

Definition: A graph G consists of a non-empty set V called the set of vertices or nodes and a set E called the set of edges that connect the vertices.
Example:

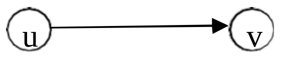


V={1,2,3,4,5,6,7}
E={(1,2)(1,3)(2,4)(3,5)(4,5)(5,6)(6,7)(5,7)}

GRAPH TERMINOLOGY

Nodes

- Initial Node (u)
- Terminal Node (v)



Represented as (U,V)

Adjacent

If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, vertices A and B are said to be adjacent if there is an edge between them.

Or

Any two nodes that are connected by an edge in a graph are called adjacent nodes.
In the above example, the adjacent nodes of edge (5, 6) are 5 and 6.

Edge
Edges are three types.

1. **Undirected Edge -** An undirected edge is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A, B) is equal to edge (B, A).
2. **Directed Edge -** A directed edge is a unidirectional edge. If there is directed edge between vertices A and B then edge (A, B) is not equal to edge (B, A).
3. **Weighted Edge -** A weighted edge is an edge with value (cost) unit.

Outgoing Edge

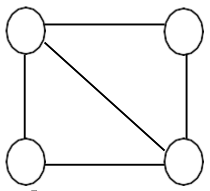
A directed edge is said to be outgoing edge on its origin vertex.

Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

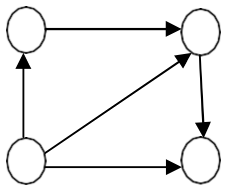
Undirected Graph

A graph with only undirected edges is said to be undirected graph.



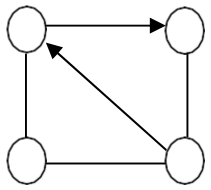
Directed Graph

A graph with only directed edges is said to be directed graph.



Mixed Graph

A graph with both undirected and directed edges is said to be mixed graph.



Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

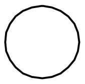
Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

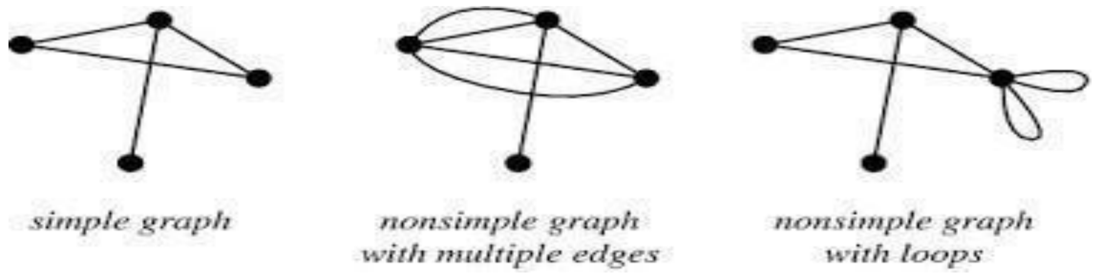
Loop

Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.



Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

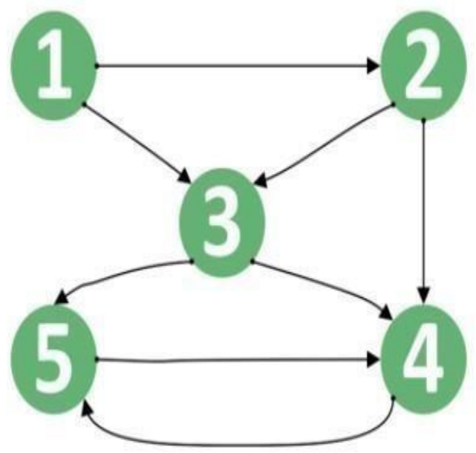

simple graph
nonsimple graph with multiple edges
nonsimple graph with loops

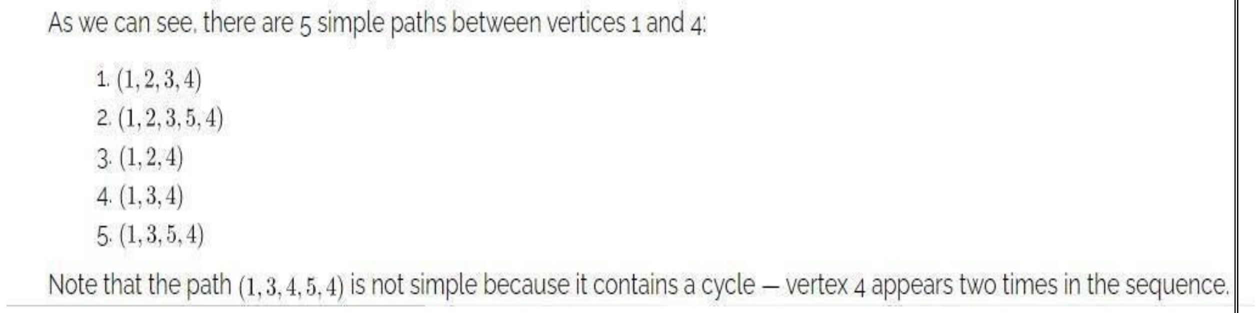
Simple path

A path is a sequence of alternate vertices and edges that starts at a vertex and ends at other vertex such that each edge is incident to its predecessor and successor vertex.

Or

A path in a digraph in which all the edges are distinct is called a simple path.



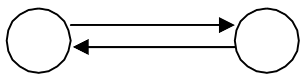

As we can see, there are 5 simple paths between vertices 1 and 4:

1. $(1, 2, 3, 4)$
2. $(1, 2, 3, 5, 4)$
3. $(1, 2, 4)$
4. $(1, 3, 4)$
5. $(1, 3, 5, 4)$

Note that the path $(1, 3, 4, 5, 4)$ is not simple because it contains a cycle — vertex 4 appears two times in the sequence.

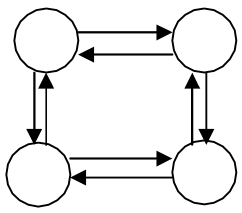
Parallel edges/Parallel nodes

If pair of nodes is joined by more than one edge, then such edges are called parallel edges.



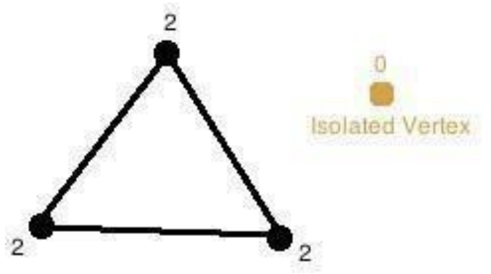
Multigraph

Any graph that contains parallel edges is called a multigraph.

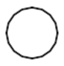


Isolated node

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



Null graph
A graph that contains only isolated nodes is called a null graph.
Example:



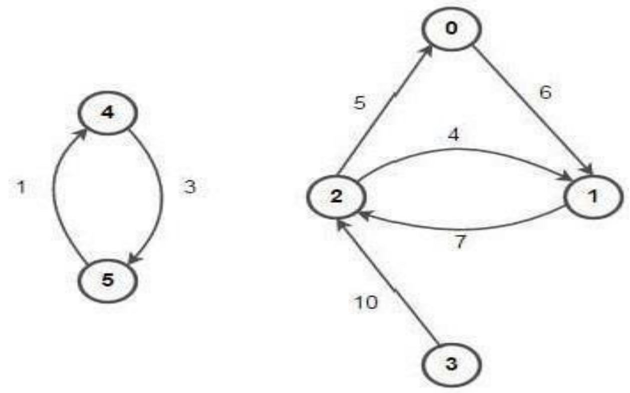
Elementary path
A path in which all the nodes it traverses distinct is called an elementary path.

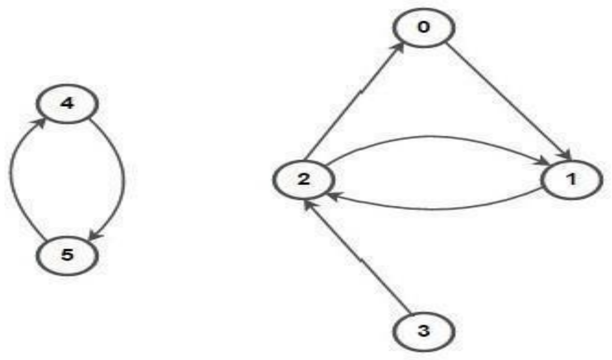
Cycle
A path which originates and ends at the same node is called a cycle.

Weighted and Unweighted graph

A weighted graph associates a value (weight) with every edge in the graph. Words cost or length can also be used insteadof weight.

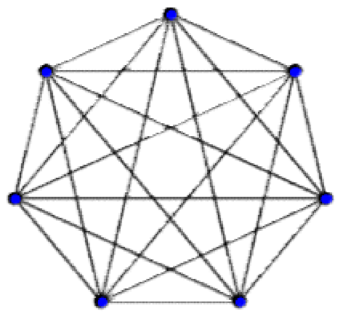
An unweighted graph does not have any value (weight) associated with every edge in the graph. In other words, a weighted graph is a weighted graph with all edge weight as 1 . Unless specified otherwise, all graphs are assumed to be unweighted by default.





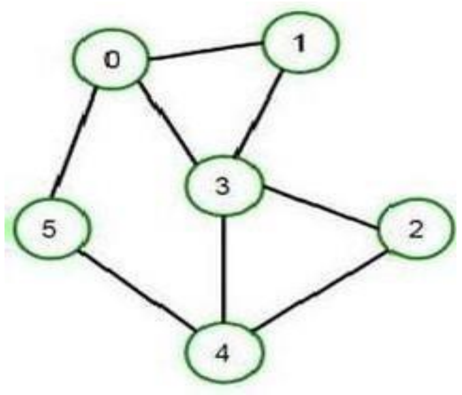
Complete graph

A complete graph is one in which every two vertices are adjacent: all edges that could exist are present.



Connected graph

A Connected graph has a path between every pair of vertices. In other words, there are non reachable vertices. A disconnected graph is a graph which is not connected.



SEQUENTIAL REPRESENTATION OF GRAPHS

There are 2 standard ways of maintaining a graph in memory of a computer. One way called the sequential representation of G is by means of adjacency matrix. The other way is called the linked list representation of G.

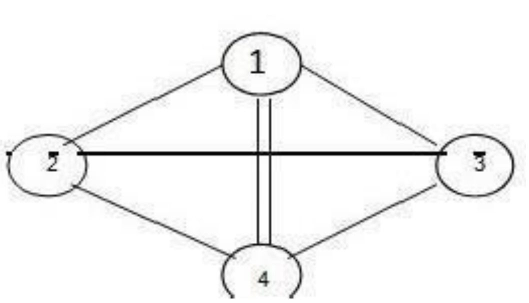
Adjacency matrix

Suppose g is a simple directed graph with m nodes and suppose the nodes for G have been ordered and are called v1, v2,…vm. Then the adjacency matrix A=(aij) of the graph G is the mXm matrix defined as follows:

Aij=1 if vi is adjacent to vj i.e, if there is an edge (vi,vj)
otherwise 0.

Such a matrix A, which contains entries of only 0 and 1 is called a bit matrix or a Boolean matrix.

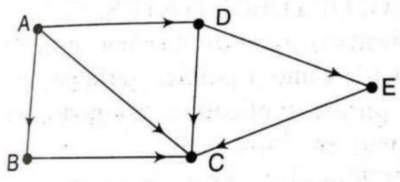
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$



Let G be a simple directed graph with m nodes v1, v2, v3…vm. The path matrix or reach ability matrix of G is the m-square matrix P=(pij) defined as follows.Pij=1 if there is a path from vi to vj otherwise 0.

LINKED LIST REPRESENTATION OFGRAPHS

In this representation, the N row adjacency matrix is represented as N linked list. There is one list for each vertex in the graph. The node in list I represents the vertices that are adjacent from vertex i. each list has a head node. The head nodes are sequential, providing each random access to the adjacency list for any particular vertex. The adjacency list for the graph is shown below

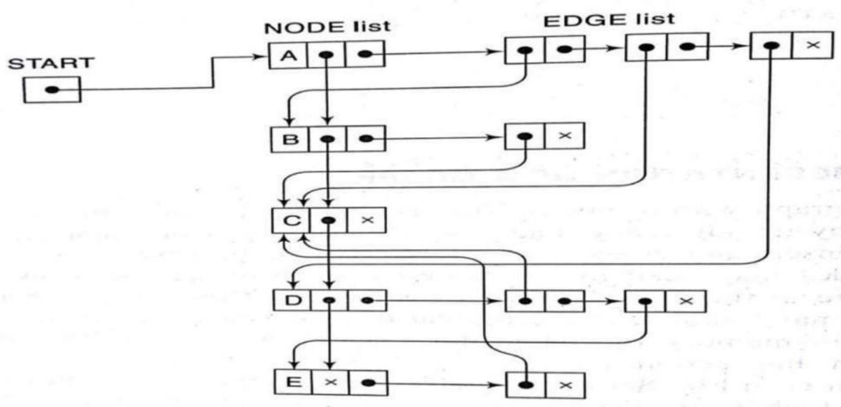


(a) Graph *G*

Node	Adjacency List
A	B, C, D
B	C
C	
D	C, E
E	C

(b) Adjacency list of *G*

The linked list representation of the above graph.



OPERATIONS ON GRAPHS

Searching in a graph

Suppose we want to find the location loc of the node N in a graph G. this can be achieved by the following procedure.

Find (info, start, ITEM, LOC) This algorithm finds the location loc of the first node containing ITEM or sets LOC=0.

```
1. Set temp:=start
2. Repeat while(temp!=NULL)
   Begin
           If ITEM=temp→info then
                   Set LOC:=temp
                   Return
           Else
                   Set temp:=temp→next
   End while
3. Set LOC:=NULL and return
```

Inserting in a graph

INSERTNODE (NODE, NEXT, ADJ, START, AVAIL, N, FLAG)

This procedure inserts a node N in the graph G.

```
1. If AVAIL=NULL then
   Set flag=false
   Return
2. Set avail→adj=NULL
3. Set new:=avail
4. Avail:=avail→next
5. Set new→node=n
6. New→next=start
```

7. Start=new
8. Set flag=true
9. finished

Deleting from a graph

Suppose a node N is to be deleted from the Graph G. This operation is more complicated than the search and insertion operations and the deletion of an edge, because we must delete all the edges that contain N. Note these edges come in two kinds: those that begin at N and those that end at N. Accordingly, our procedure will consist mainly of the following 4 steps:

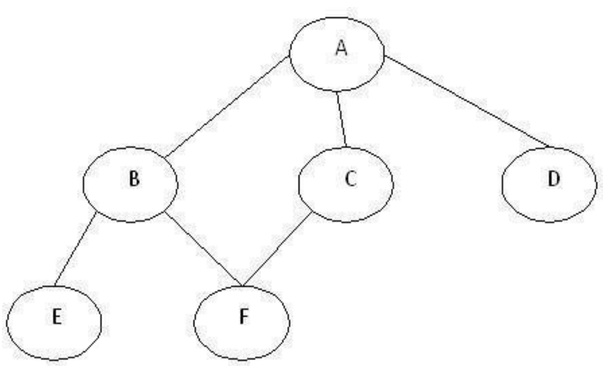
1. Find the location LOC of the node N in G.
2. Delete all the edges ending at N; that is delete LOC from the list of successors of each nodeM in G.
3. Delete all the edges beginning at N. this is accomplished by finding the location BEG of thefirst successor and location END of the last successor of N, and then adding the successor list of N to the free AVAIL list.
4. Delete N itself from the list NODE.

TRAVERSING A GRAPH

The breadth first search (BFS) and the depth first search (DFS) are the two algorithms used for traversing and searching a node in a graph. They can also be used to find out whether a node isreachable from a given node or not.

Depth First Search (DFS)

The aim of DFS algorithm is to traverse the graph in such a way that it tries to go far from the root node. Stack is used in the implementation of the depth first search. Let’s see how depth first search works with respect to the following graph:



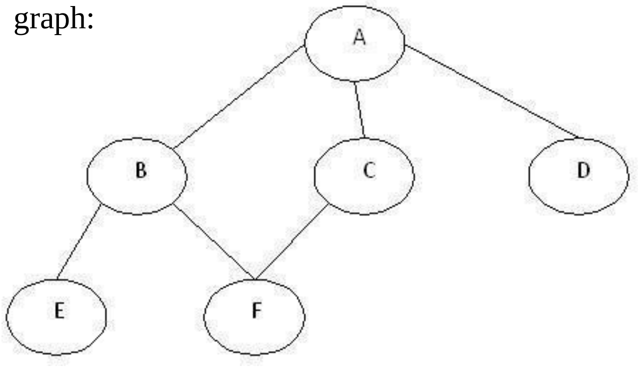
As stated before, in DFS, nodes are visited by going through the depth of the tree from the starting node. If we do the depth first traversal of the above graph and print the visited node, itwill be “A B E F C D”. DFS visits the root node and then its children nodes until it reaches theend node, i.e. E and F nodes, then moves up to the parent nodes.

Algorithmic Steps

Step 1: Push the root node in the Stack.
Step 2: Loop until stack is empty.
Step 3: Peek the node of the stack. (Peek used to return the value of top)
Step 4: If the node has unvisited child nodes, get the unvisited child node, mark it as traversed and push it on stack.
Step 5: If the node does not have any unvisited child nodes, pop the node from the stack.

Breadth First Search (BFS)

This is a very different approach for traversing the graph nodes. The aim of BFS algorithm is to traverse the graph as close as possible to the root node. Queue is used in the implementation of the breadth first search. Let’s see how BFS traversal works with respect to the following graph:



If we do the breadth first traversal of the above graph and print the visited node as the output,it will print the following output. “A B C D E F”. The BFS visits the nodes level by level, so it will start with level 0 which is the root node, and then it moves to the next levels which are B, C and D, then the last levels which are E and F.

Algorithmic Steps

Step 1: Push the root node in the Queue.
Step 2: Loop until the queue is empty.
Step 3: Remove the node from the Queue.
Step 4: If the removed node has unvisited child nodes, mark them as visited and insert the unvisited children in the queue.

Question Bank

Multiple Choice Questions

(Questions for Application)

1. The worst case occurs in linear search algorithm when_______________
   A.item is somewhere in the middle of thearray
      i. item is not in the array at all
      ii. item is the last element in the array
      iii. item is the last element in the array or item is not there at all

2. Sorting algorithm can be characterized as_______________
   A.Simple algorithm which requires the order of n2 comparisons to sort n items.
   B. sophisticated algorithms that require the o(n log2n) comparisons to sort items.
   C. both of the above
   D. none of the above

3. State true or false for internal sortingalgorithms.
   i) Internal sorting are applied when the entire collection if data to be sorted issmall enough that the sorting can take place within main memory.
   ii) The time required to read or write is considered to be significant in evaluatingthe performance of internal sorting.
   A.i-true,ii-true
   B.i-true,ii-false
   C.i-false,ii-true
   D.i-false,ii-false

4. Is putting an element in the appropriate place in a sorted list yields a larger sorted order list.
   A.insertion
   B.extraction
   C.selection
   D.distribution

5. __is rearranging pairs of elements which are out of order, until no such pairs remain.
   A. insertion
   B. exchange
   C. selection
   D.distribution

6. Which of the following sorting algorithm is of divide and conquer type?
   A.Bubble sort
   B. Insertion sort
   C. Merge sort
   D. Selection sort

7. ________________partitions the given set of elements each time to find an element.

A.**Binary search**
B.Linear search
C.Quick search
D.Merge search

8. The elements in the array must be in________________order to perform binary search.

A.Unsorted
B.**Sorted**
C.Linear
D.Non-linear

9. Which of the following sorting algorithms is the fastest?

A.Selection sort
B. Shell sort
C. Insertion sort
D. Quick sort

(Questions for Skill)

10. If the number of records to be sorted is small, then_________sorting can be efficient.

A.merge
B.heap
C.**selection**
D.bubble

11. Which of the following is not a limitation of binary search algorithm?

A.must use a sorted array
B. requirement of sorted array is expensive when a lot of insertion and deletions areneeded
C. there must be a mechanism to access middle element directly
D. binary search algorithm is not efficient when the data elements more than1 500

12. The average case occurs in linear search algorithm_______________

A.**when item is somewhere in the middle of thearray**
B. when item is not the array at all
C. when item is the last element in thearray
D. item is the last element in the array or item is not there at all

13. Binary search algorithm cannot be applied to_______________

A.**sorted linked list**
B. sorted binary trees
C. sorted linear array
D. pointer array

14. Sorting algorithm is frequently used when n is small where n is total number of elements.

A.heap
B.**insertion**
C.bubble
D.quick

15. Which of the following sorting algorithm is of priority queue sorting type?

A.Bubble sort
B.Insertion sort
C.Merge sort
D.**Selection sort**

16. Which of the following is not the required condition for binary search algorithm?

A.the list must be sorted
B. there should be the direct access to the middle element in any sublist
C. there must be mechanism to delete and/or insert elements inlist
D. number values should only be present

17. Partition and exchange sort is______________

A.**Quick sort**
B. Tree sort
C. Heap sort
D. Bubble sort

18. _________________is technique that performs search process in a sequential manner

A.Linear search
B.Binary search
C.Hashing
D.Indexing

19. Which of the following is true?

A.A graph may contain no edges and many vertices
B.A graph may contain many edges and no vertices
C.A graph may contain no edges and no vertices
D.None of the mentioned

20.A connected graph T without any cycles is called a_______

A.A tree graph
B.Free tree
C.A treed
D.All of the above

21.In a graph if E=(u,v)means__________

A.u is adjacent to v but v is not adjacent to u
B.e begins at u and ends at v
C.u is processor and v is successor
D.both b and c

22.A path in a digraph in which all the edges are distinct is called ____________

A.Simple path
B.Elementary path
C.Cycle
D.Loop

23. Any two nodes that are connected by an edge in a graphare called_________ nodes

A.Directed
B. Adjacent
C. Common

D. Isolated

24. In a graph if e=[u,v], Then u and v arecalled_________

A.End points of e

B.Adjacent nodes

C.Neighbours

D.All of the above

25.A graph in which some edges are directed and some edges are undirected is called ______________________graph.

A.Digraph

B.Mixed

C.Isolated

D.Cycle

Questions carrying 4 marks

(Questions for Application)

1. Write and explain the algorithm for the insertion sort technique.
2. Write the algorithm for merge sort method.
3. Write the linear search algorithm with an example.
4. Write the binary search algorithm with an example.
5. Write an algorithm for breadth first search (BFS).
6. Write an algorithm for depth first search (DFS).

(Questions for Skill)

1. Sort the following numbers using insertion sort method.

   12, 3, 1, 7, 8
2. Sort the following numbers using selection sort technique.

   23, 56, 10, 33, 88, 45
3. Sort the following numbers using merge sort.

   3, 1, 12, 45, 32, 11, 5, 7, 8, 98, 54, 6, 2
4. Write and explain the algorithm for bubble sorting procedure.
5. Write the algorithm for selection sort method.
6. Write the algorithm for the Quick sort method.