Srinivas Institute of Management Studies

Pandeswar, Mangalore – 575 001

BACKGROUND STUDY MATERIAL

DATA STRUCTURES WITH C++

BCA III Semester

SRINIVAS GROUP

SAMAGRA GNANA
ESTD: 1988

Compiled by

Mrs. Lathika K
SIMS, Mangalore

2015

BCA303- DATA STRUCTURES

UNIT-1

UNIT-4

UNIT-I

Introduction and Overview

1.1 Introduction

This chapter introduces the subject of data structures and present an overview of the content of the text. An overview of data organization and certain data structures will be covered. We will introduce the notion of an algorithm.

1.2 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, 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. For example,

Attributes:	name	age	sex
Values:	vaishnavi	3	F

Entities with similar attributes form an entity set. Each attribute of an entity set has a range of values.

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 primary key.

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.

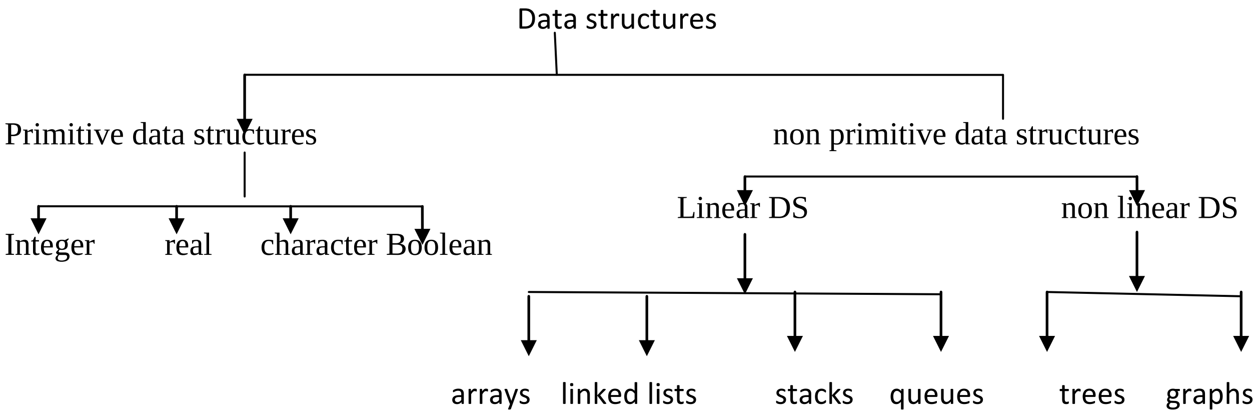
1.3 Data structures

Data may be organized in many different ways; the logical or mathematical model of a particular organization of data is called a data structure.

1.3.1 Classification of data structures

Data structures are generally classified into primitive and no primitive data structures. 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.

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 non linear data structures.

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.

Similarly, a data structure is said to be non linear if the data is not arranged in sequence. The insertion and deletion of data is therefore not possible in a linear fashion. Tress and graphs are examples

Some of the data structures are

Arrays

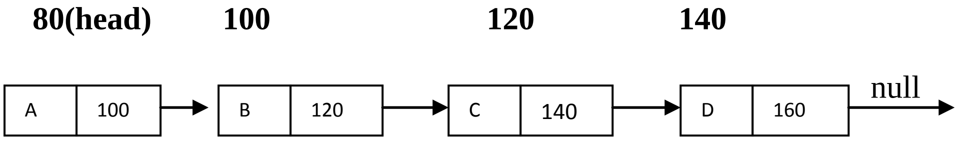
The simplest type of data structure is a linear or one dimensional array. A linear array is a list of finite number n of similar data elements referenced respectively by a set of n consecutive numbers. Suppose A is an array, then the elements of A are denoted by A(1), A(2),…A(n) or A[1], A[2], A[3], …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 lists

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.

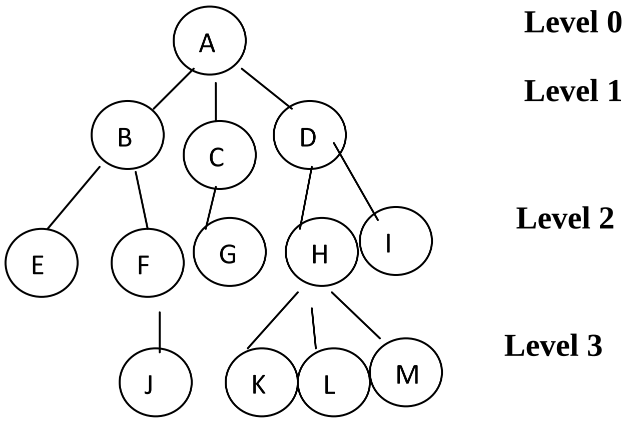


Trees

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

1) There is a special data item called the root of the tree.
2) And its remaining data items are partitioned into number of mutually exclusive subsets each of which is itself a tree. They are called subtrees.



There are data structures other than arrays, linked lists and trees. Some of these structures are

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.

1.4 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 s 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.

1.5 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 the user.

1.6 Assignment 1

1) What is a data structure? What are the2 types of data structures?
2) What are the data structure operations?
3) Write a note on ADT.

Chapter 2

Preliminaries

2.1 Introduction

This chapter describes the format that will be used to present the algorithm. The chapter begins with a brief outline and discussion of various mathematical functions which occur in the study of algorithms and in computer science in general and the chapter ends with a discussion of different kinds of variables that can appear in our algorithms and programs.

2.2 Mathematical notations and functions:

2.2.1 Floor and ceiling functions

Let x be any real number. Then x lies between two integers called the floor and the ceiling of x. specifically.
$\lfloor x \rfloor$ called the floor of x, denotes the greatest integer that does not exceed x.
$\lceil x \rceil$ called the ceiling of x, denotes the least integer that is not less than x.

2.2.2 Remainder function

Let k be any integer and let M be a positive integer. Then k(mod M) will denote the integer remainder when k is divided by M. Thus 25(mod 7)=4, 25(mod 5)=0, 35(mod 11)=2, 3 (mod 8)=3.

The term mod is also used for the mathematical congruence relation, which is denoted and defined as follows:

$a \equiv b(\mathrm{mod}\, M)$ if and only if M divides b-a. M is called the modulus, and $a \equiv b(\mathrm{mod}\, M)$ is read as “a is congruent to b Modulo M”.

Arithmetic modulo M refers to the arithmetic operations of addition, multiplication and subtraction where the arithmetic value is replaced by its equivalent value in the set {0,1,2,3…M-1} Or in the set {1,2,3,…M}.

For example, in arithmetic modulo 12, sometimes called “clock” arithmetic, $6+3 \equiv 3$, $7*5 \equiv 11$

, $1-5 \equiv 8$, $2+10 \equiv 0 \equiv 12$

2.2.3 Integer and absolute value functions

Let x be any real number. The integer value of x, written INT(x), converts x into an integer by deleting the factorial part of the number. Thus INT (3.14)=3, INT(-8.5)=-8, INT(7)=7.

2.2.4 The absolute value

The absolute number of the real number x, written ABS(x) or |x|, is defined as the greater of x or –x. hence ABS(0)=0 and $x \neq 0$, ABS(x)=x or ABS(x)=-x, depending on whether x is positive or negative. Thus |-15|=15, |7|=7, |-3.33|=3.33 etc. note that |x|=|-x| and $x \neq 0,$|x} is positive.

2.2.5 Summation symbol; sums

Consider a sequence a1, a2……………an, then the sums a1+a2+a3…..+an will be denoted by

$$\sum_{i=1}^{n} ai$$

For example, $\sum_{i=1}^{n} aibi = a_1b_1 + a_2b_2 + ... + a_nb_n$

2.2.5 The factorial function

The product of the positive integers from 1 to n, is denoted by n! i.e n!=1.2.3.4.(n-1).n where 0!=1.

2.2.6 Permutations

A permutation a set of n elements is an arrangement of the elements in a given order. For example, the permutations of the set consisting of the elements a,b,c are as follows.
Abc, acb, bac, bca, cab,cba

One can prove: there are n! permutations of a set of n elements. There are 4!=24 permutations of a set with 4 elements, 5!=120 permutations of a set with 5 elements and so on.

2.2.7 Exponents and logarithms

Recall the following definitions for integer exponents

$$a^m = a.a.a.a.....a(mtimes), a^0 = 1, a^{-m} = \frac{1}{a^m}$$

Exponents are extended to include all rational numbers by defining, for any rational number m/n

$a^{\frac{m}{n}} = \sqrt[n]{a^m} = \left(\sqrt[n]{a}\right)^m$

For example

$2^{4=16,}$ 2^{-4}=1/16, $125^{2/3}=5^2=25$

Logarithms are related to exponents as follows. Let b be a positive number. The log of any positive number x to the base b, is written as $\log_b x$ represents the exponent to which b must be raised to obtain x. i.e y= $\log_b x$ and $b^y=x$ are equivalent statements.

Accordingly,

$\mathrm{Log}_2 8=3$ since $2^3=8$, $\log_{10}100=2$ since $10^2=100$

Further more, for any base b,

$\mathrm{Log}_b 1=0$ since $b^0=1$, $\log_b b=1$ since $b^1=b$

Frequently, logarithm are expressed using approximate values. For example, using tables or calculators one obtains,

$\mathrm{Log}_{10}300=2.4771$ and $\log_e 40=3.6889$ as approximate answers

Logarithms to the base 10 are called common logarithms, logarithms to the base e are called natural logarithms and logarithms to the base 2 are called binary logat=rithms.

Ln x instead of $\log_e x$

Lg x or log x instead of $\log_2 x$

2.3 Algorithmic notation

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

An array A of numerical values is in memory. We want to find the location LOC and value 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 that LOC=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 location LOC 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 AMX:=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 frist part is a paragraph which tells the purpose of the algorithm, identifies the variables which occur in the algorithm and lists the input data. The second part of the algorithm consists of the list of steps that is to be executed.

2.3.1 Steps, control, exit

The steps of the algorithm are executed one after another, beginning with step 1. Control may be transferred to step n of the algorithm by the statement “go to step n”. if 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** encountered. This statement is similar to the STOP statement used in the FORTRAN and in flowcharts.

2.3.2 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.

2.3.3 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.

2.3.4 Assignment statement

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

2.3.5 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.)**

2.3.6 Procedures

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

2.4 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 repetitive flow

2.4.1 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.

2.4.2 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 picture in fig. 2-4(b). as indicated by the flowchart, if the condition holds, then module A is executed; otherwise Module B is executed.

Multiple alternative:

this structure has the form:

```
If condition(1), then:
        [Module A1]
Else if condition(2), then:
        [Module A2]
.
.
.
Else ifcondition(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.

2.4.3 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 byT:
[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.

2.5 Subalgorithm

A subalgorithm is a complete and independently defined algorithmic module which is used by some main algorithm or by some other subalgorithm. A subalgorithm 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 that the subalgorithm will usually a heading like NAME(PAR1, PAR2………….PARk)

Here NAME refers to the name of the algorithm which is used when the sub algorithm is called and PAR1, PAR2…PARk refers to the parameters which are used to transmit data between the subalgorithm 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.

Subalgorithms fall into 2 categories: function subalgorithms 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 subalgorithm may send back more than one value.

The following function subalgorithm 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 subalgorithm and A,B and C are the parameters. The return statement includes, the variable AVE whose value is returned to the calling program.

The subalgorithm 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 subalgorithm, the subalgorithm 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 to TEST.

2.6 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 a byte.
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 nad 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 may be coded using only one bit, 1 for true and 0 for false.

2.7 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.

2.8 Assignment 2

1) Write a note on conditional structures.
2) Write a note on subalgorithms
3) Write a note on loop structures
4) Write a note on algorithmic notations.

Chapter 3
String processing

3.1Introduction

Today computers are frequently used for processing non numerical data called character data. String is a sequence of characters. This chapter is essentially tangential and independent of the rest of the text.

3.2 Basic terminology

Each programming language contains a character set that is used to communicate with the computer. This set is usually includes the following.

Alphabet : A,B,C…Z
Digits: 0,1,1..9
Special characters: +, -, /, *, ( ) , , . , $...

The set of special characters, which includes the black space, frequently denoted by □. A finite sequence of S of zero or more characters is called a string. The number of characters in a string is called its length. The string with zero characters is called the empty string or the null string. Specific strings will be denoted by enclosing their characters in single quotation marks. For example,

‘THE END’, ‘HI HOW R U’ , ‘ ‘, ‘□□’

Are strings with lengths 7, 10, 0 and 2 respectively

Let S1 and S2 be strings. The string consisting of characters of S1 followed by the characters of S2 is called the concatenation of S1 and S2. And it is denoted by S1 || S2. For example,
‘THE’ || ‘END’ = ‘THEEND’ but ‘THE’ || ‘□’ || ‘END’ =’THE END’

A string Y is called a substring of a string S if there exist strings X and Z such that S=X||Y||Z. if X is an empty string, then Y is called an initial substring of S and if Z is an empty string then Y is called a terminal substring of S. for example,

‘BE OR NOT’ is a substring of ‘TO BE OR NOT TO BE’
‘THE’ is an initial substring of ‘THE END’

If Y is a substring of S, then the length of Y cannot exceed the length of S.

3.3 Storing strings

Strings are stored in 3 types of strictures.

1. Fixed length structures
2. Variable length with fixed maximum
3. Linked structures

3.3.1 Record oriented, fixed length storage

In fixed length storage each line of print is viewed as a record where all records have the same length. Each record accommodates the same number of characters.

Advantages:

1. The ease of accessing data from any given record.
2. The ease of updating data in any given record.

Disadvantages

1. Time is wasted reading an entire record if most of the storage consists of inessential blank spaces
2. Certain records may require more space than available
3. When the correction consists of more or fewer characters than the original text, changing a misspelled word requires the entire record to be changed.

3.3.2 Variable length storage with fixed maximum

Although strings may be stored in fixed length memory locations as above, there are advantages in knowing the actual length of each string. For example, one then does not have t read the entire record when the string occupies only the beginning part of memory location.

The storage of variable length strings in memory cells with fixed lengths can be done in two general ways:

1. One can use a marker, such as 2 dollar signs, to signal the end of the string.
2. One can list the length of the string as an additional item in the pointer array.

3.3.3 Linked storage

One way linked list is a linearly ordered sequence of memory cells called nodes, where each node contains an item, called a link, which points to the next node in the list. Strings may be stored in inked list as follows. Each memory cell is assigned one character or a fixed number of characters and a link contained in a cell gives the address of the cell containing the next character or group of characters in the string. For example, consider the following quotation

To be or not to be, that is the question.

The following figure shows how the string would appear in memory with one character per node, and the another diagram shows how it would appear with four characters per node.

3.4 Character data type

3.4.1 Constants

Many programming languages denote string constants by placing the string in either single or double quotation marks. For example,

‘THE END’ and ‘TO BE OR NOT TO BE’ are string constants of lengths 7 and 18 characters respectively.

3.4.2 Variables

Variables fall into one of three categories. They are static, semantic and dynamic. By a static character variable, we mean a variable whose length is defined before the program is executes and cannot change throughout the program.

By a semantic character variable, we mean a variable whose length may vary during the execution of the program as long as the length does not exceed a maximum value determined by the program before the program is executed.

By a dynamic character variable, we mean that a variable whose length can change during the execution of the program.

3.5 Strings as ADT

Most languages have strings as a built in data type or a standard library and a set of operations defined on that type. Therefore there is actually no need for us to create our own string ADT.

However, we can implement our own string data type if required. We need to know what operations are allowed on the string.

A string data type typically should have operations to:

1. Return the nth character in a string
2. Set the nth character in a string to c
3. Find the length of the string
4. Concatenate two strings
5. Copy a string
6. Delete part of a string
7. Modify and compare strings.

The following are the string operations.

Getchar(str,n) - returns nth character in the string
Putchar(str,n,c) – sets the nth character in the string to c.
Length(str) – returns the number of characters in the string.
Pos(str1, str2) – returns the position of the first occurrence of str2 found in str1, 0 if no match.
Concat(str1,str2) – returns a new string consisting of characters in str1 followed by characters in str2
Substring(str1,I,m)-returns a substring of length m starting at position I in string str
Delete(str,I,m)-deletes m characters from str starting at position i
Insert(str1, str2, i)- changes str1 into a new string with str2 inserted in position i
Compare(str1, str2)- returns an integer indicating whether str1 > str2

3.6 Assignment 3

1. What is a string? Which is the string concatenation operator?
2. Write a note on storing strings.
3. Write a note on character data type.
4. Write a note on strings on ADT

Chapter 4
Arrays, Records and Pointers

4.1 Introduction

Data structures are classified as either linear or non linear. A data structure is said to be linear if its elements form a sequence or in other words, a linear list. There are 2 basic ways of representing such linear structures in memory. One way is to have the linear relationship between the elements represented by means of sequential memory locations. These linear structures are called arrays. The other way is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called linked lists.

The operations which are performed on any linear structures are given blow.

1. Traversal: processing each element in the list
2. Search: finding the location of the element with a given value or the record with a given key.
3. Insertion: adding a new element to the list
4. Deletion: removing an element from the list.
5. Sorting: arranging the elements in some type of order.
6. Merging: combing two lists into a single list.

4.2 Linear arrays

A linear array is a list of finite number n of homogeneous data elements (i.e data elements of the same type) such that:

1. The elements of the array are referenced respectively by an index set consisting of n consecutive numbers.
2. The elements are stored respectively in successive memory locations.

The number n of elements is called the length or size of the array. In general, the length or number of data elements of the array can be obtained from the index set by the formula

Length = UB-LB+1

Where UB is the largest index, called the upper bound and LB is the smallest index called the lower bound, of the array. The elements of an array A may be denoted by the subscript notation

A[1], A[2], …A[N]. the number N in A[N] is called a subscript or an index and A[N] is called the subscripted variable.

4.3 Arrays as ADT

An array is a fundamental ADT. An array is a predefined set which has a sequence of cells where we can store different types of elements.

Consider the following example: object(A,N) here an array A is created which can store N number of items in it. A[i] is an element in the array A stored in its ith position.

An array can also be considered a vector; the arrays are named as A[1], A[2], A[3], …A[N].

4.4 Representation of linear arrays in memory

The simplest data structure that makes use of computer address to locate its element is the one-dimensional array. It is also called a vector. The address of Ai is given by the following equation loc(Ai)=L_0+c*(i-1). Where A is an array with subscript starts from 1. L_0 is the base address of the array A and c represents the size of the data type. In the equation Loc(Ai)=L0+c*(i-1), i-1 because array starts from 1.

Let us consider the general case of representing an array A whose lower subscript is given by some variable b. the location of Ai is then given by loc(Ai)=L0+c*(i-b)

4.5 Traversing linear arrays

Let A be a collection of data elements stored in the memory of the computer. Suppose we want to print the content of each element of A or we want to count the number of elements of A. this can be achieved by traversing A. i.e by accessing and processing each elements of A exactly once.

The following algorithm traverses a linear array.

Here LA is a linear array with lower bound LB and upper bound UB. This algorithm traverses an array A applying an operation PROCESS to each element of A.

```
1. [initialize counter]
   Set K:=LB
2. Repeat steps 3 and 4 while K<=UB
3. [visit element]
   Print  A[K]
4. [increase counter]
   Set K:=K+1
   [end of step 2 loop]
5. Exit
```

This can be written using a repeat-for loop instead of repeat while loop.

```
1. Repeat for K=LB to UB:
2. Print  A[K]
   [End loop]
3. Exit
```

4.6 Inserting and deleting

Let A be a collection of data elements in the memory of a computer. Inserting refers to the operations of adding another element to the collection A, and deleting refers to the operation of removing one of the elements from A.

Inserting at the end of a linear array can be easily done because the memory space allocated for the array is large enough to accommodate the additional element. On the other hand, suppose we need to insert an element in the middle of the array. Then half of the elements are moved downwards to new locations to accommodate the new element and keep the order of other elements.

Similarly, deleting an element at the “end” of an array is not difficult. But deleting an element somewhere in the middle of the array requires that each subsequent element should be moved one location upward in order to fill up the array.

Algorithm: inserting into a linear array

Insert (A, N, I, ITEM)

Here A is a linear array with N elements. This algorithm inserts an element **ITEM** into the the jth position in an array.

```
1. Set I:=N
2. Repeat steps 3 and 4 while (I>=J)
   Begin
3. Set A[I+1+:=A[I]
4. Set I=I-1
   End of while loop
5. Set A[J]:=ITEM
6. Set N:=N-1
7. exit
```

Algorithm: deleting from an array

Delete (A, N, J, ITEM)

here A is an array of N elements. ITEM is an element which is to be deleted from the jth position of the array.

```
1. Set ITEM:=A[J]
2. Repeat for (I = J to N-1)
   Begin
3. Set A[I]:=A[I+1]
   End of for loop
4. Set N:=N-1
5. Exit
```

4.7 Sorting

Sorting refers to the operation f rearranging the elements of an array A so that they are in increasing order.

For example,, Suppose A has the following elements 8,4,19,2,7,13,5,16. After sorting the array A becomes 2,4,5,7,8,13,16,19.

Bubble sort

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.

4.8 Searching

Let A be a collection of data elements in the array. Searching refers to the operation of finding the location LOC of ITEM in array A, or printing the message that the required ITEM does not found. The search is successful, if the ITEM is found and unsuccessful otherwise. There are 2 types of searching algorithms. Binary search and linear search.

4.8.1 Linear search

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.

Linear_Search(A,element,N)

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

```
1. Set Loc:=-1
2. Repeat step3 For i=0 to n-1
3. If (Element=A[i]) then
   begin
      a. Set loc:= i
      b. Goto step 4
   [End if]
[End for loop]
4. If (loc>=0) then
      a. write('element found in location', loc+1)
      b. Else
      c. Write('element not found')
5. Exit
```

4.8.2 Binary Search

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 of the middle element.

```
    1. Set LOW:= 0
    2. Set HIGH:= N-1
    3. Set LOC:= -1
    4. Repeat steps 5 and 6, 7 While LOW<=HIGH
Begin
    5. Set MID:=LOW+HIGH)/2
    6. IF element=A[MID]
       Begin
          a. Set LOC:= MID
          b. goto step 8
       [END IF]
    7. IF element<A[MID]
          a. Set HIGH:=MID-1
          b. Else
          c. Set LOW:=MID+1
       [END IF]
       [END of While loop]
    8. IF LOC>=0 then
          a. Write('element found in location', LOC)
          b. ELSE
          c. Write('Element not found')
[END IF]
    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.

4.9 Multi dimensional arrays

The linear arrays discussed so far are called one dimensional arrays, since each element in the array is referenced by a single subscript. Multidimensional arrays are the one where elements are referenced by two or three subscripts.

4.9.1 Two dimensional arrays

A two dimensional array $m \times n$ array A is a collection of $m \times n$ data elements such that each element is specified by a pair of integers called subscripts. The element of A with first subscript I and second subscript j will be denoted as A[I,J]. two dimensional arrays are also called matrices in mathematics and tables in business applications. Hence two dimensional arrays are also called matrix arrays.

There is a standard way of drawing a two dimensional $m \times n$ array A where the elements of A form a rectangular array with m rows and n columns and where the element A[I,J] appears in row I and column j.

4.9.2 Storage representation for two dimensional arrays

A multidimensional array can be represented by an equivalent one-dimensional array. For example, a 2-dimensional array consisting of 2 rows and 4 columns is stored sequentially by column as

A11	A12	A13	A14
A21	A22	A23	A24

i.e A[1,1], A[2,1],A[1,2],A2,2],A[1,3], A[2,3], A[1.4], A[2,4]

In general for a 2-dimensional array consisting of m rows and n columns in ***Column Major*** two dimensional array, the address of elements A[i,j] is given *by* $L_0+(j-1)*m+(i-1)$*.*

In many programming languages a 2-dimensional array will be stored row by row referred as *row-major order* instead of column by column (Column major). A 2 dimensional array consisting of m rows and n columns in ***row major*** *two* dimensional array, the address of element a[i,j] is given by $L_0+(i-1)n+(j-1)$

4.9.3 Representation of 2 dimensional arrays in memory

Let A be a 2 dimensional $m \times n$ array. Although A is pictured as a rectangular array of elements with m rows and n columns, the array will be represented in memory by a block of m.n sequential memory locations. For a linear array A, the computer does not keep track of the address of every element of A. it keeps track of the base address L0 i.e the address of the first element of A. the computer uses the formula Loc(Ai)=L0+c*(i-1).

To find the address of an element of 2 dimensional array, the computer keeps track of the base address of A, i.e the address of first element A[1,1] of A and computes the address using the formula

Column major order A[I,j]= $L_0+c[(j-1)*m+(i-1)]$

Or the formula

Row major order A[I,j]= $L_0+c[(i-1)n+(j-1)]$

Where c is the size of the data type.

4.10 Representation of polynomials using array

The polynomial of degree n can be represented by storing the coefficient of n+1 terms of a polynomial; in an array. An $x+4x^2-7x^5$ is a polynomial of degree 5.

All the elements of an array has two values, namely coefficient and exponent. A single dimensional array is used for representing a single variable polynomial. The index of such an array can be considered as an exponent and the coefficient can be stored at that particular index.

The polynomial expression $3x^4+5x^3+6x^2+10x-14$ can be stored in a single dimensional array as shown in the following figure.

-14
10
6
5
3

Coefficients of the polynomial

Polynomial representation using an array

Drawbacks

1. Suppose the exponent is too large, then the size of the array will become more. For example, if we have a data like $4x^{999}$, we will have to store the coefficient 4 at index 999 in the array and the array sixe will have to be 1000.
2. Wastage of space. Suppose you have a polynomial $6x^{99}-10$, then only two elements will be stored in the array of size 100. The remaining space will be empty and therefore not utilized.

3. A third problem is faced when we cannot decide the size of the array. Suppose we declare an array size 15, and the exponent value of the polynomial is 50, then we cannot store the value and we will have to change the array size.

4.11 Matrices

Vectors and matrices are mathematical terms which refer to collections of numbers which are analogous to linear and 2 dimensional arrays i.e

1. an n-element vector V is a list of n numbers usually given in the form V=(V1, V2….Vn)
2. an $m \times n$ matrix A is an array of m.n numbers arranged in m rows and n columns as follows.

$$A = \begin{bmatrix} A11 & A12 & A13...... & A1n \\ A21 & A22 & A23..... & A2n \\ A31 & A32 & A33... & A3n \\ Am1 & Am2 & Am3... & Amn \end{bmatrix}$$

In the context of vectors and matrices, the term scalar is used for individual numbers.

A matrix with one row(column) may be viewed as a vector and similarly, a vector may be viewed as a matrix with only one row(column).

A matrix with the same number n of rows and columns is called a square matrix or an n-square matrix. The diagonal or main diagonal of an n-square matrix A consists of the elements A11, A22, ….Ann.

4.12 Sparse matrices

Matrices that contain a lot of zero elements is called sparse matrices. Two general types of n-square sparse matrices, which occur in various applications, are shown in the following figure. The first matrix, where all entries above the main diagonal are zero or, where nonzero entries can only occur on or below the main diagonal is called a **lower triangular** matrix.

The second matrix where nonzero entries can only occur on the diagonal or on elements immediately above or below the diagonal is called a **tridiagonal** matrix.

$$\begin{bmatrix} 4 & & & & \\ 3 & -5 & & & \\ 1 & 0 & 6 & & \\ -7 & 8 & -1 & 3 & \\ 5 & -2 & 0 & 2 & -8 \end{bmatrix}$$

Triangular matrix

$$\begin{bmatrix} 5 & -3 & & & & & \\ 1 & 4 & 3 & & & & \\ & 9 & -3 & 6 & & & \\ & & 2 & 4 & 7 & & \\ & & & 3 & -1 & 0 & \\ & & & & 6 & -5 & 8 \\ & & & & & 3 & -1 \end{bmatrix}$$

Tridiagonal matrix

4.13 Assignment 4

1) What is an array?
2) What is sorting? Mention any two sorting techniques
3) Write bubble sort algorithm.
4) What is searching? Name any two searching technique.
5) Write the linear search algorithm.
6) Write the binary search algorithm.
7) Write an algorithm to insert and delete from/to an array.
8) Write an algorithm to traverse an array.
9) What is a sparse matrix? Mention two types of sparse matrix.

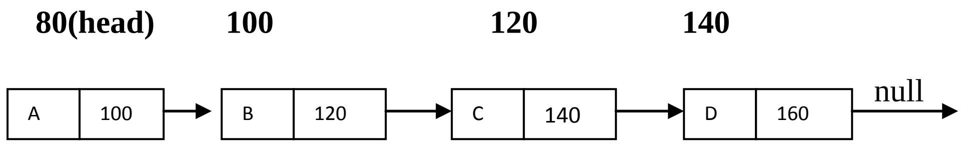
UNIT-II
Chapter 5

Linked Lists

5.1 Introduction

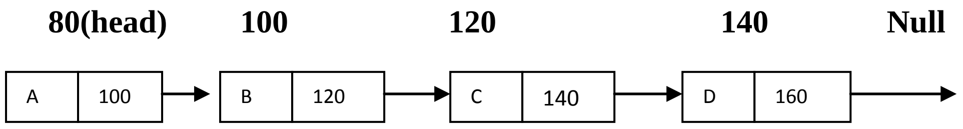
A linked list is a non sequential collection of data items. For every data item in the linked list, there is an associated pointer that gives the memory location of the next data item in the linked list.

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.



5.2 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.

5.3 Representation of a linked list

Each and every node in a linked list is a structure containing two fields such as INFO and NEXT 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.

5.4 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 a linked 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) Beginning of the linked list
b) Specified position in the list
c) End of a linked list

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

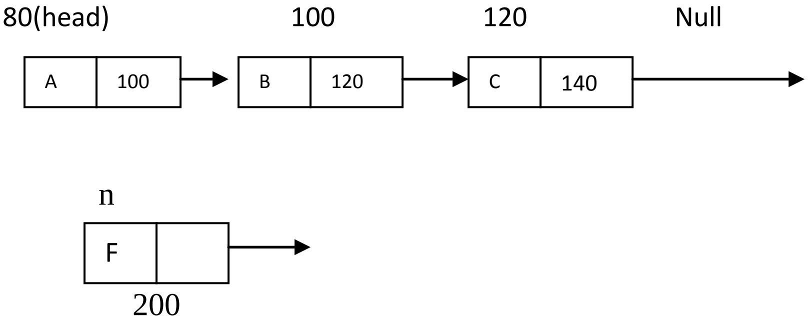
Algorithm for creating a new node

FUNCTION Create_Node(x)

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

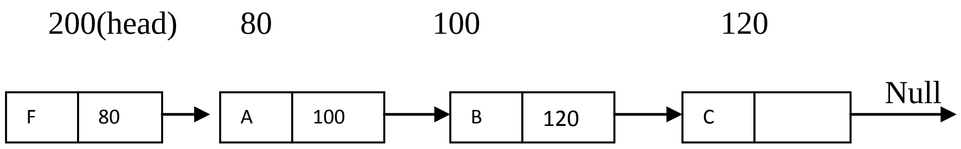
```
1. [cerate a new node]
   N=new node

1. [set the INFO]
   n→info=x

2. [return the created node n]
   Return n

3. [finished]
   Exit
```

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

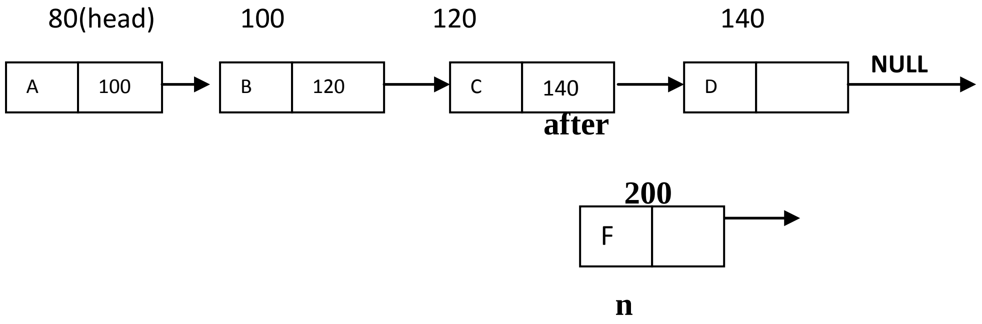


Procedure Insert_Head(n) this algorithm inserts a new node called n to the beginning of a linked list. Head denote the starting node of the linked list

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

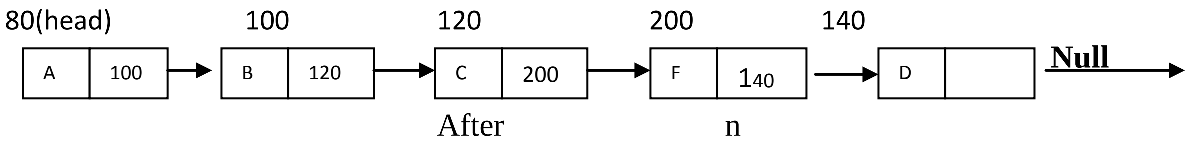


Algorithm for inserting a node in the middle of a linked list

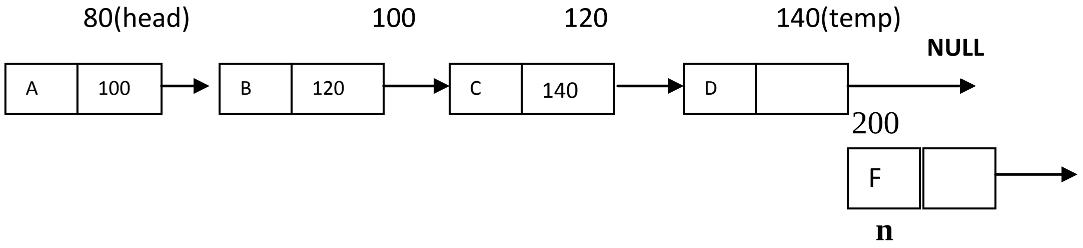


Procedure insert_middle(n) this algorithm inserts a new node called n in the specified position of the linked list. After is a node where the new node is to be inserted after it.

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

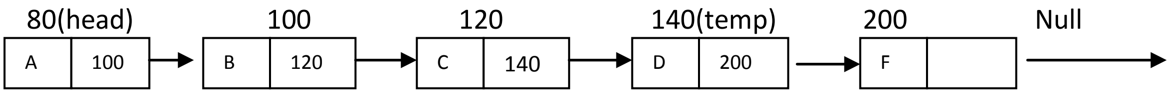


Algorithm for inserting a node at the end of a linked list

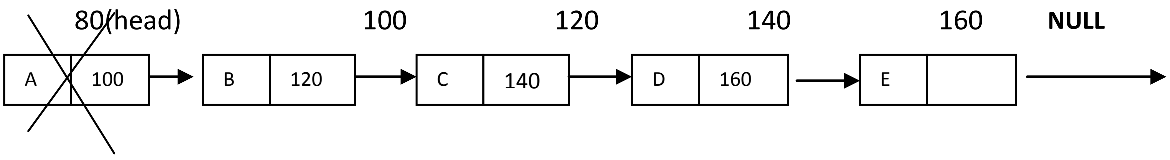


Procedure Insert_end(n) this algorithm inserts a new node called n at the end of the linked list. Temp is a temporary variable where the head node is stored temporarily before insertion takes place.

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

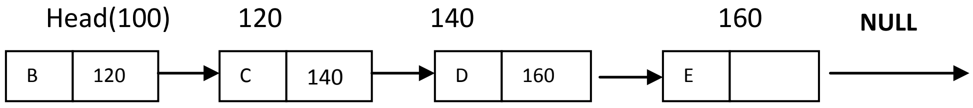


Algorithm for deleting a head node from the linked list

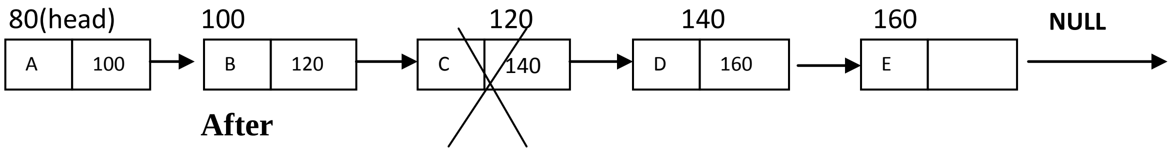


Procedure Delete_Head()this algorithm deletes the head node of the linked list. Temp is a temporary variable where the head node is stored temporarily before deletion takes place

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

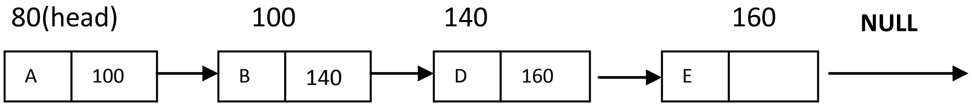


Algorithm for delete a node form the middle of a linked list

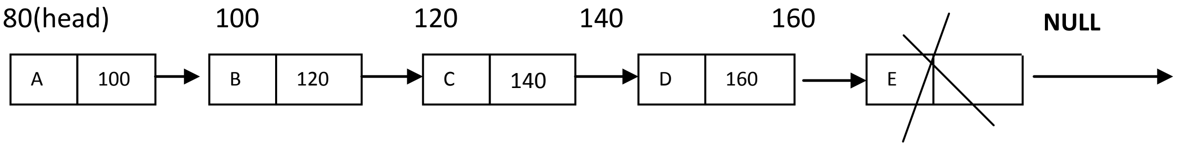


Procedure Delete_middle(n) this algorithm deletes a node from the middle of the linked list. Temp is a temporary variable where the node is stored temporarily before deletion takes place. After is a node where the node to be deleted after that.

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

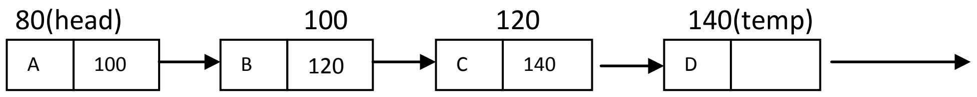


Algorithm for deleting the last node form the linked list



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

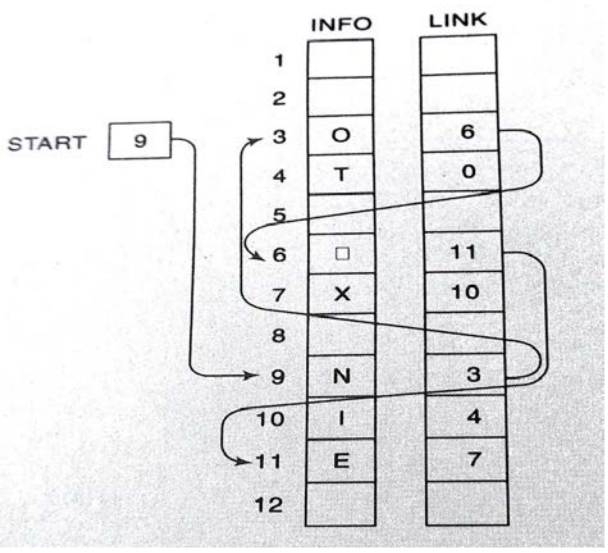
```
1. Temp=head
2. While(temp→next→next!=NULL)
   Begin
3. Temp=temp→next
   End
4. delete temp→next
5. temp→next=NULL
6. exit
```



5.5 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.

5.6 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→next!=NULL)
   Begin
3. Display temp→info
   End
4. Set temp=temp→next
5. Finished
```

5.7 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=tem→next
   End if
   End while
7. Set LOCATION=NULL  // search is unsuccessful
8. exit
```

5.8 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 wan t 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 free storage 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.

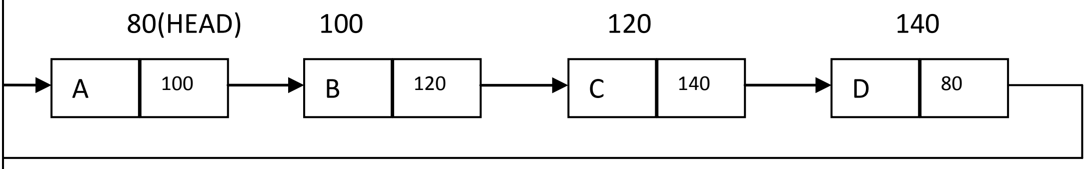
5.9 Overflow and underflow

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.**

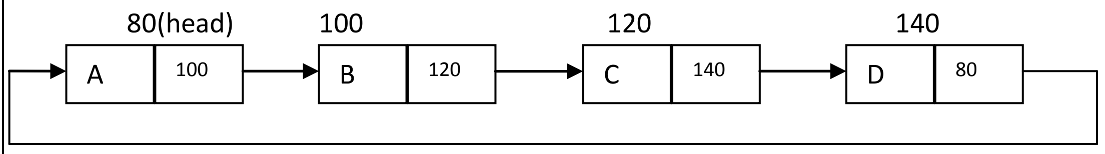
Similarly, the term underflow refers to the situation where one wants to delete data from a data structure that is empty. The programmer may handle underflow by printing the message UNDERFLOW.

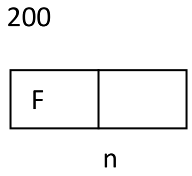
5.10 Circular linked list

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



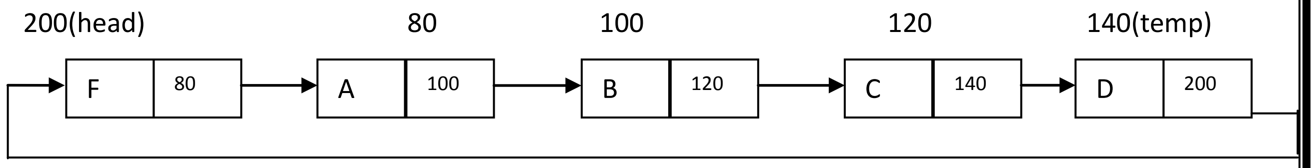
Algorithm for inserting a new node at the beginning of a circular linked list



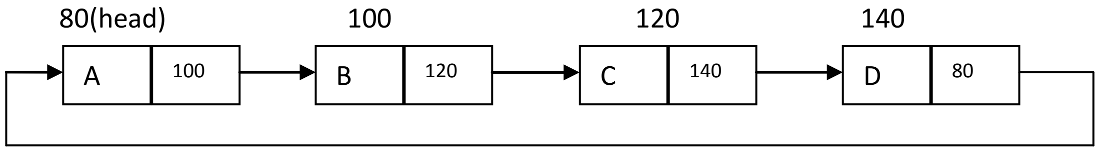


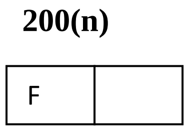
Procedure Insert_head(n)

```
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. finished
```



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

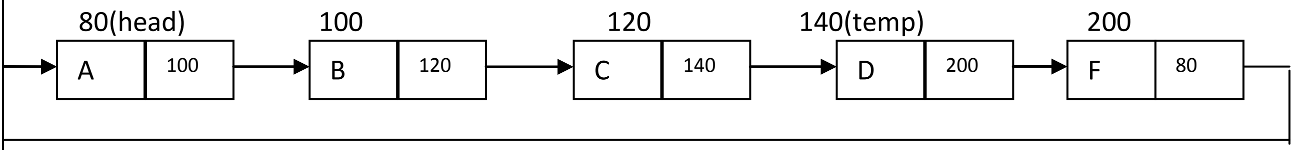




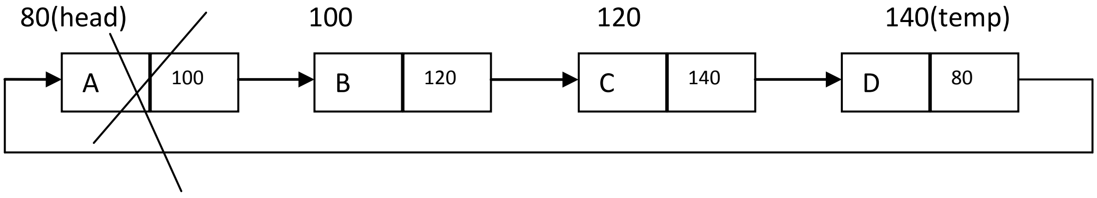
Procedure Insert_End(n)

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

```
3. temp=temp→next
   end

4. temp→next=n
5. n→next=head
6. finished
```

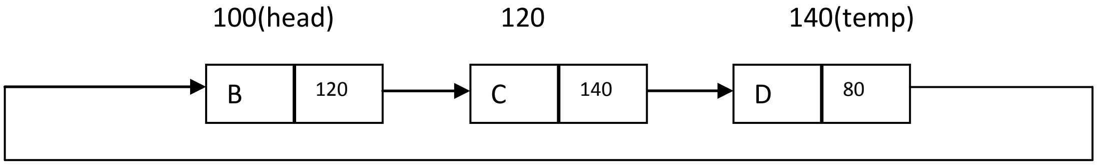


Algorithm for deleting a node at the beginning of a circular linked list

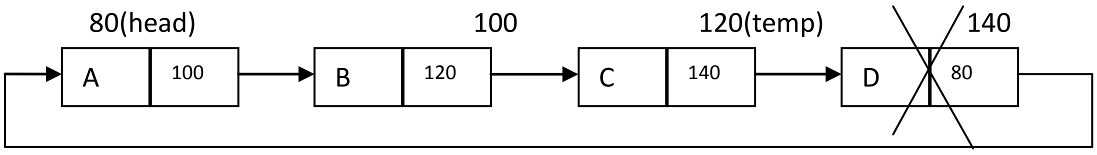


Procedure delete_head()

```
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
```

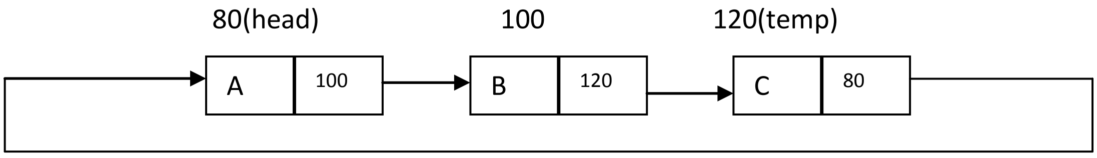


Algorithm for deleting a node at the end of a circular linked list



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
```

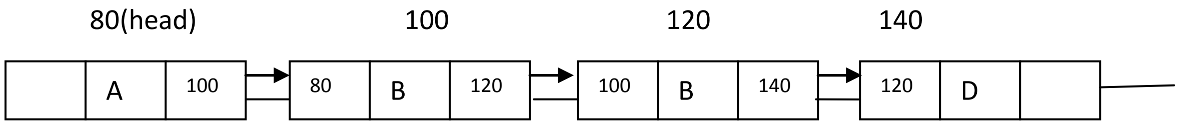


5.11 Doubly Linked List:

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 application is 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.

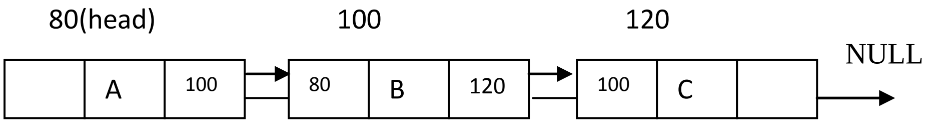
Each node in a double linked list has two link fields. These are used to point to a successor and the predecessor nodes.



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.

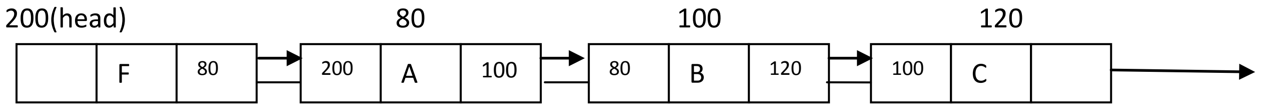
```
Class Node
{
        Node *Prev;
        Node *Next;
        int Info;
};
```

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

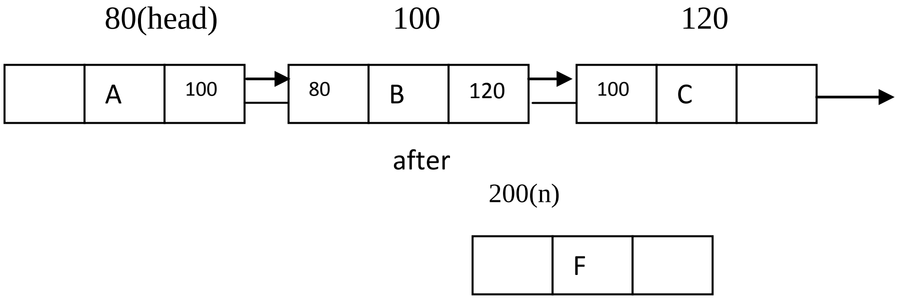


Procedure Insert_Head(n)

```
1. n→next=head
2. Head→previous = n
3. Head=n
4. Finished
```

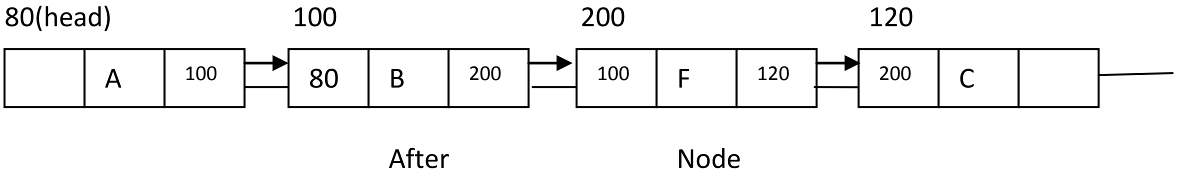


Inserting a node in the middle of a doubly linked list.

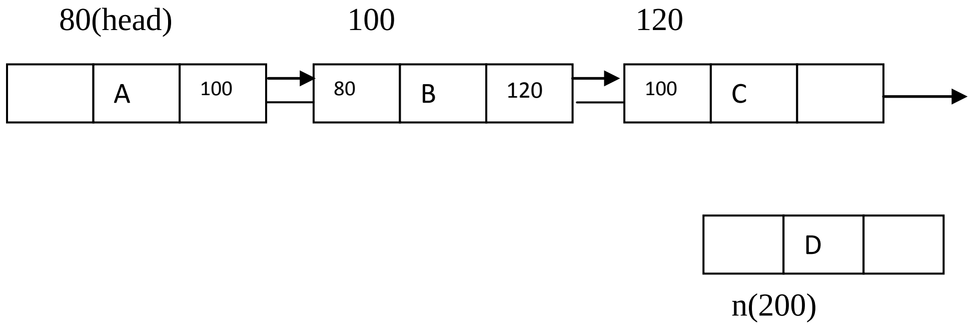


Procedure Insert_Middle(n)

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

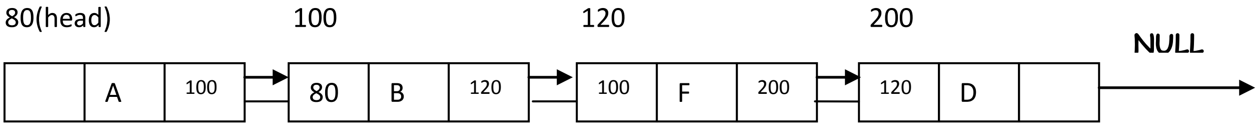


Inserting a node at the end of a doubly linked list.

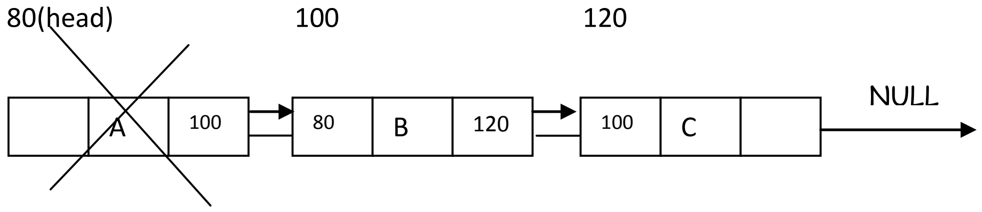


Procedure Insert_end(n)

```
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
```

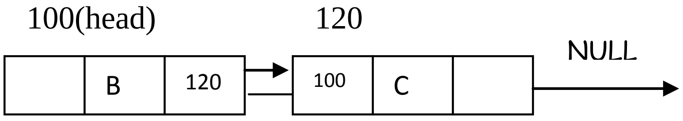


Deleting the head node from the doubly linked list.

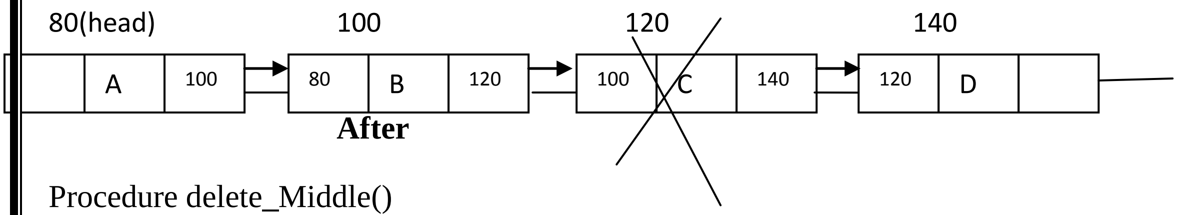


Procedure Delete_Head()

```
1. head=head→next
2. delete head→previous
3. finished
```

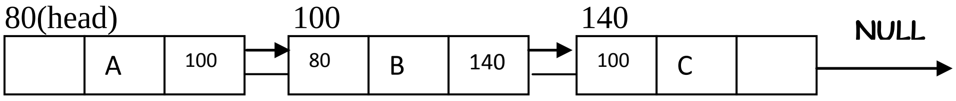


Deleting a node from the middle of the doubly linked list

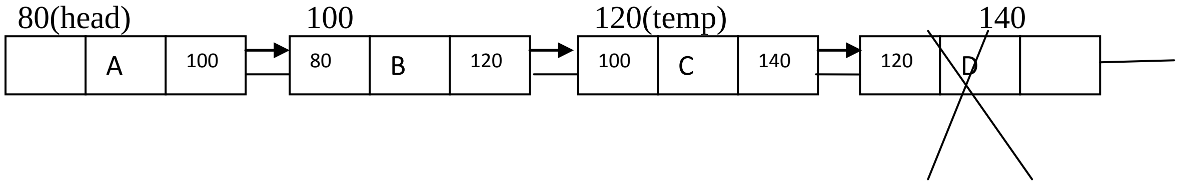


Procedure delete_Middle()

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

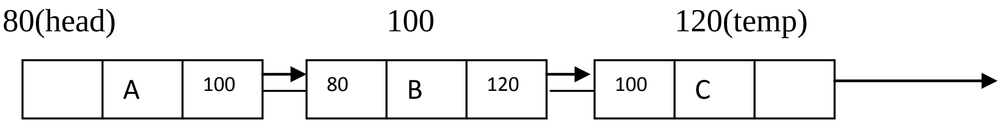


Deleting a node at the end of the doubly linked list



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
```



5.12 Assignment 5

1) What is a linked list? With a neat diagram explain different types of linked lists.
2) Write an algorithm to insert nodes from the beginning, middle, end of a singly linked list
3) Write an algorithm to delete nodes from the beginning, middle, end of a singly linked list
4) Write an algorithm to insert nodes to the beginning, middle, end of a doubly linked list
5) Write an algorithm to delete nodes from the beginning, middle, end of a doubly linked list
6) Write an algorithm to traverse a linked list.
7) Write an algorithm to search for element in a linked list

Chapter 6
Searching and sorting

6.1 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.

There are many sorting and searching algorithms. This chapter will investigate sorting algorithms and then searching algorithms.

6.2 Sorting

Let A be a list of n elements A1, A2,…An in memory. Sorting A refers to the operation of rearranging the contents of A so that they are in the increasing order.

6.3 Insertion sort

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 ordered part.

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

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

```
1. repeat step 2 for i=1 to N-1
2. repeat step 3 for j=I to 0
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]
```

6.4 Selection sort

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 elements A[1], A[2]…A[N] is in memory.

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 sublist of N-1 elements and 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 sublist 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.

Algorithm

Procedure SelectionSort(A, N)
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.

```
1. repeat steps 2,3 and 4,6,7 for i=1 to N-1
   begin
2. set SMALL:=A[I]
3. set POSITION:=I
4. repeat step 5 for J=I+1 to N
   begin
```

```
    5. if A[J] < SMALL
       begin
           a. set SMALL:=A[J]
           b. set POSITION:=J
       [End of if structure]
[end of inner for loop]
    6. Set A[POSITION]:=A[I]
    7. Set A[I]:=SMALL
[end of outer for loop]
```

6.5 Merge Sort:

This Sorting technique merges two ordered lists which can be combined to produce a single sorted list. This process can be accomplished easily by successively selecting the smallest element occurring in either of the lists and placing the element in a new list, thereby creating an ordered list.

Merge_Sort(A,LOW, MID, HIGH) Given an array A of N element. This procedure rearranges the array in ascending order. LOW denotes the lower bound of the array A, MID denotes the midpoint and HIGH denotes the upper bound of the array. TEMP is temporary array used to hold the list of elements. Variable F1 is the index for the First array, variable S1 is the index for the second array and T1 is the index for the temporary array.

```
 1. Set F1:=LOW
 2. Set S1:= MID+1
 3. Set T1:= LOW
 4. Repeat steps 5 and 6 While (F1<=MID) and (S1<=HIGH)
Begin
 5. If A[F1]<=A[S1]
 Begin
        a. Set TEMP[T1]:= A[F1]
        b. Set F1:= F1+1
[End of if structure]
Else
Begin
        c. TEMP[T1]=A[S1]
        d. S1←S1+1
[END ELSE]
 6. Set T1:= T1+1
[END of WHILE Loop]
 7. IF F1 <=MID
    Begin
 8. Repeat step 9 and 10 for F1=F1 to MID
    Begin
 9.   Set TEMP[T1]:=A[F1]
 10. Set TEMP:= TEMP+1
[END of FOR Loop]
[END IF]
ELSE
Begin
 11. Repeat steps 12 and 13 for S1 =S1 to HIGH
Begin
 12. Set TEMP[T1]:=A[S1]
 13. Set T1:=T1+1
[End of For Loop]
[End of ELSE]
 14. Repeat step 15 for I=LOW to HIGH
 15. Set A[I]:=TEMP[I]
 [End of FOR loop]
```

6.6 Radix sort

Radix Sort is a clever and intuitive little sorting algorithm. Radix Sort puts the elements in order by comparing the **digits of the numbers**.

Consider the following 9 numbers:

493 812 715 710 195 437 582 340 385

We should start sorting by comparing and ordering the **one's** digits:

0	710 340
1	
2	812 582
3	493
4	
5	715 195 385
6	
7	437
8	
9	

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sublists above. Now, we gather the sublists (in order from the 0 sublist to the 9 sublist) into the main list again:

340 710 812 582 493 715 195 385 437

Note: The **order** in which we divide and reassemble the list is **extremely important**, as this is one of the foundations of this algorithm.

Now, the sublists are created again, this time based on the **ten's** digit:

0	
1	710 812 715
2	
3	437
4	340
5	
6	
7	
8	582 385
9	493 195

Now the sublists are gathered in order from 0 to 9:

710 812 715 437 340 582 385 493 195

Finally, the sublists are created according to the **hundred's** digit:

0	
1	195
2	
3	340 385
4	437 493
5	582
6	
7	710 715
8	812
9	

At last, the list is gathered up again:

195 340 385 437 493 582 710 715 812

And now we have a fully sorted array. Radix Sort is very simple, and a computer can do it fast.

Disadvantages

In the example above, the numbers were all of equal length, but many times, this is not the case. If the numbers are not of the same length, then a test is needed to check for additional digits that need sorting. This can be one of the slowest parts of Radix Sort, and it is one of the hardest to make efficient.

Radix Sort can also take up more **space** than other sorting algorithms, since in addition to the array that will be sorted, you need to have a sublist for **each** of the possible digits or letters. If you are sorting pure English words, you will need at least 26 different sublists, and if you are sorting alphanumeric words or sentences, you will probably need more than 40 sublists in all!

Since Radix Sort depends on the digits or letters, Radix Sort is also ***much*** **less flexible** than other sorts. For every different type of data, Radix Sort needs to be rewritten, and if the sorting order changes, the sort needs to be rewritten again. In short, Radix Sort takes more time to write, and it is very difficult to write a general purpose Radix Sort that can handle all kinds of data.

6.7 Shell sort

Donald shell invented the shell sort algorithm in the year 1959. This algorithm is based on the insertion sort, but uses multiple passes across the data, each pass sorting elements that are separated from each other by a fixed distance, gradually reducing the distance between the elements being compared until the last pass uses an increment of 1.

Consider the dataset:

16	4	3	13	5	6	8	9	10	11	12	17	15	18	19	7	1	2	14	20

We can divide this into three smaller slices:

16	4	3	13	5	6	8
9	10	11	12	17	15	18
19	7	1	2	14	20	

Once we have performed this slicing we can sort each slice:

First we sort the first column (16 9 and 19) to give 9 16 19

Next the second column (4 10 and 7) to give 4 7 10

Column three (3 11 and 1) to give 1 3 11

Column four (13 12 and 2) to give 2 12 13

Column five (5 17 and 14) to give 5 14 17

Column six (6 15 and 20) to give 6 15 20
and column seven (8 and 18) to give 8 18

Reassembling the array we now have:

9	4	1	2	5	6	8	16	7	3	12	14	15	18	19	10	11	13	17	20

Notice that elements such as 1 have moved a large distance (from position 17 to position 3).

We now slice the array into a different number of slices. For this example we will use five slices, but other values are possible.

9	4	1	2	5	6	8	16	7	3	12	14	15	18	19	10	11	13	17	20

Slicing this into five we get:

9	4	1	2
5	6	8	16
7	3	12	14
15	18	19	10
11	13	17	20

Again we sort each column to give:

5	3	1	2
7	4	8	10
9	6	12	14
11	13	17	16
15	18	19	20

Logically reassembling, we now have the dataset:

5	3	1	2	7	4	8	10	9	6	12	14	11	13	17	16	15	18	19	20

Notice that the data set is starting to look much more sorted than it was before. We can now slice the array into 10:

5	3
1	2

7	4
8	10
9	6
12	14
11	13
17	16
15	18
19	20

Sorting the columns gives us:

1	2
5	3
7	4
8	6
9	10
11	13
12	14
15	16
17	18
19	20

Reassembling we get:

1	2	5	3	7	4	8	6	9	10	11	13	12	14	15	16	17	18	19	20

The majority of the elements are now near to where they should be, and the last pass of the algorithm is a conventional insertion sort.

6.8 Searching and data modification

Data modification refers to the operations of inserting, deleting and updating. Here data modification refers to inserting and deleting. These operations are closely related to searching, since usually one must search for the location of the ITEM to be deleted or one must search for the proper place ti insert ITEM in the table.

1. **Sorted array**: here one can use a binary search to find the location LOC of a given ITEM in time O(log n). on the other hand, inserting and deleting are very slow. Thus a

sorted array would likely to be used when there Is a great deal of searching but only very little data modification.

2. **Linked list**: here one can only perform linear search to find the location LOC of a given ITEM and the search may be very, very slow possibly requiring O(n). on the other hand, inserting and deleting requires only a few pointers to be changed. Thus a linked list would be used when there is a great deal of data modification.

3. **Binary search tree**: this data structure combines the advantages of the sorted array and the linked list. That is searching is reduced to searching only a certain path pin the tree T, which is on the average requires only O(log n) comparisons. The tree is maintained in memory by a linked list representation, so only certain pointers need be changed after the location of the insertion or deletion is found.

6.9 Assignment 6

1) What is sorting?
2) Sort the following numbers using insertion sort method
   12 3 1 7 8
3) Sort the following numbers using selection sort technique
   12 3 1 7 8
4) Write the algorithm for insertion sort method.
5) Write the algorithm for selection sort method
6) Write the algorithm for merge sort method
7) Sort the following numbers using merge sort.
   2 3 1 12 45 32 11 5 7 8 98 54 6

UNIT -III

Chapter 7
Stacks, Queues and Recursion

7.1 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.

7.2 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 2 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.

FFF	→TOP
EEE	
DDD	
CCC	
BBB	
AAA	

1	2	3	4	5	6	7	…………….	n-1	n
AAA	BBB	CCC	DDD	EEE	FFF				

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])
```

7.3 Array representation of stacks

XXX	YYY	ZZZ						
1	2	3	4	5	6	7	8	9

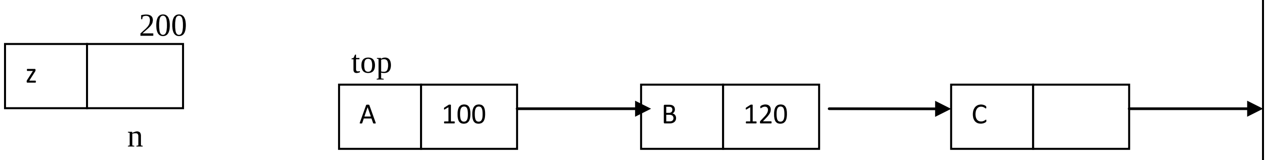
Stacks may be represented in the computer in various ways, usually by means of a linear array. Stack contains a pointer variable TOP, which contains the location of the top element of the stack and a variable N which gives the maximum number of elements that can be held by the stack. The condition TOP=0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

The following above shows the array representation of a stack. Since top=3, the stack has 3 elements XXX, YYY and ZZZ. The maximum size of the stack is 8, and 5 more elements can be inserted into the stack.

7.4 Linked list representation of stacks

The linked list representation of a stack is implemented using a singly linked list. The **Info** fields of the nodes hold the elements of the stack and the **next** filed holds the pointers to the next element in the stack.

A push operation into STACK is accomplished by inserting a node into the start of the list and a pop operation is accomplished by deleting the node starting i.e the top most node in the stack.

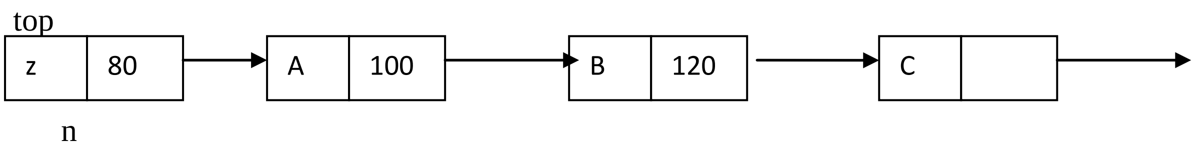


PUSH algorithm

Procedure PushLinkedList(info, item, top,n)

This algorithm pushes an item into a linked list

```
1. n=new node
2. if n==NULL then
   write('overflow')
   exit
3. n→info=item
4. n→next=top
5. top=n
6. finished
```



Pop algorithm

Function popLinkedlist(top, item,temp)

This algorithm deletes the top element of a linked stack and assigns it to the variable item

```
1. if top==NULL
   write'underflow')
   exit
2. item=top→info
3. temp=top
4. top=top→next
5. delete temp
6. finished
```

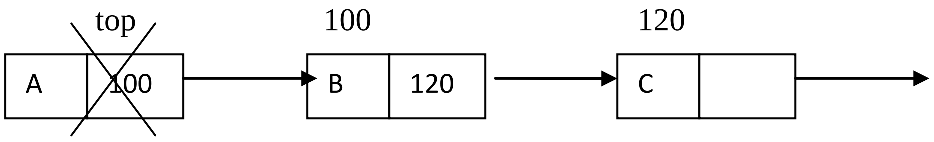
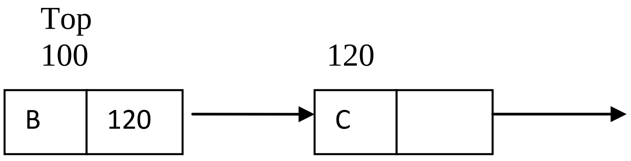


Diagram after pop



7.5 Stack as ADT

the stack ADT implementation in C is simple. Instead of storing data in each node, we store a pointer to the data. It is the application program's responsibility to allocate memory for the data and pass the address to the stack ADT. Within the ADT, the stack node looks like any linked list node except that it contains a pointer to the data than the actual data. As the data pointer type is unknown, it is stored as a pointer to void.

To create a stack we define a pointer to a stack as shown in the following example and then cal the create stack function. The address of the stack structure is returned by create stack and assigned to pointer in the calling function.

```
STACK *stack;
…
..
Stack=cretaeStack();
```

ADT implementation of stacks

To make an abstract data type we can not rely completely on data structure, we must also have operations which support the stack. The stack abstract data type structure is shown in the following program.

//the stack ADT type definitions which are included in the header file

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

Typedef struct
{
        Int count;
        STACKNODE *top;
}STACK;
```

1. STACKNODE* is a pointer to an object of type STACKNODE.
2. The NODE structure consist of a data pointer ***dataptr** and a link pointer* **next**
3. The **STACK** structure consists of 2 elements. They are a pointer to the top of the stack i.e ***top** and a **count** variable to count the number of elements currently in the stack

Create stack

Create stack function allocates a stack head node, initializes the top pointer to NULL and assigns a zero to the count field. The address of the node in the dynamic memory is then returned to the caller.

ADT create stack: this algorithm creates an empty stack. It returns pointer to a NULL stack or returns NULL in case of an overflow.

```
STACK *createStack(void)
{
        STACK *newstack;
        Newstack=(STACK*) malloc(sizeof(STACK))
        If (newsatck)
        {
                Newstack→count=0;
                Newstack→top=NULL;
        }
Return newstack;
```

}

PUSH stack

Before pushing data into stack, we need to find a place for it. For this, we need to allocate memory from the heap using **malloc**. Once the memory is allocated, we assign the data pointer to the node and then set the link to point to the node which is at the top of the stack.

Push stack: using this function we push a data into a stack. Let **newstack** be a pointer to the stack and **dataptr** be a pointer to the data to be inserted. This program inserts data into the stack. It returns true if successful and false in case of an underflow.

```
Bool pushStack(STACK *newstack, void*datainptr)
{
        STACKNODE *newptr;
        Newptr=(STACKNODE *) malloc(sizeof(STACKNODE)); //cerates a node called
newptr
        If(!newptr)
                Return false;
        Newprt→dataptr=datainptr;       //newptr’s information part is filled with an
address specified by datainptr
        Newptr→next=newstack→top
        Newstack→top=newptr

(newstack→count)++;
Return true;
}
```

POP stack

The data at the top of the stack is returned by the pop stack operation. The deleted node is then returned to the caller.

Pop stack: this function is used to pop an item on the top of the stack. STACK is a pointer to a stack. The program is used for returning a pointer to the user’s data if successful. It returns NULL if there is an underflow.

```
Void * popstack(STACK* newsatck)
{
        Void *dataoutptr;
        STACKNODE* temp;
        If (newsatck→count ==0)
                Dataoutptr=NULL;
        Else
        {
                Temp=newstack→top;
                Dataoutptr=newstack→top→dataptr;
                Newstack→top=newsatck→top→next;
                Free(temp);
                (newstack→count)--;
        }

        Return dataoutptr;
}
```

7.6 Arithmetic expressions; polish notation

Precedence of operators:

Priority	**Operators**
Highest:	exponentialtion(↑)
Next highest	*, /
Lowest	+ , -

An example is 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- infix, prefix and postfix.

Prefix: +ab
Infix: a+b
Postfix: ab+

In infix notation, the operator is placed between 2 operands. For example, p=x+y*z.

In postfix notation, an operator follows its operands and the notation does not require parenthesis. This is also known as reverse polish notation. For example, a+b*c=> abc*+

In prefix notation, the operator is placed first than the operands. This is also known as polish notation.

For example, p=x+y*z => =p+x*yz

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.
3) 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.

7.6.1 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
```

7.6.2 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
```

7.7 Application of stacks

7.7.1 Quick Sort

it is one of the most popular sorting techniques. This algorithm works by partitioning the array to be sorted. And each partition in turn stored recursively. In partition, one of the array elements is chosen as a key value. This key value can be the first element of an array. That is if A is an array the key=A[0]. And rest of array elements are grouped into 2 partitions such that
One partition contains elements smaller than the key value
Another partition contains elements larger than the key value.

Partition 1	Key	Partition 2

As an example, consider an array with the following elements

45,26,77,14,68,61,97,39,99,90

Here 45 is selected as a key value. Then two indices namely *low* and *high* are used to indicate ***first element of each array*** and the ***last element***. The low index starts on the left and selects an element that is greater than the key value. Then these elements are interchanged. This process is repeated until all elements to the left of the key are smaller than the key value. And all elements to the right of the key are greater than the key value. Since array elements are partitioned and exchanged, this technique is called ***partition-exchange technique***. ***Quicksort*** *is* ***an algorithm of the divide and conquer type***.

7.7.2 Decimal to binary conversion

The following pseudo code converts a decimal number into a binary number

```
1. read number
2. loop number>0
      a. set digit=number%2
      b. print digit
      c. set number =number/2
3. end of loop structure
```

However, this code has a problem. The binary numbers are carried backward. Suppose we enter a binary number 19, it becomes 11001 instead of 10011. This problem can be solved using a stack. Instead of printing the binary number once it is produced, we can push it into a stack. Then after the binary digit has been converted, we just pop the stack and print the results one digit at a time.

7 .8 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 does not call 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.

7.8.1 Factorial function

The product of the 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 procedure 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
5. Return
```

7.8.2 Fibonacci sequence

The Fibonacci sequence is as follows 0,1,1,1,2,3,5,8,13,…

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(fiba, n-2)
4. Call Fibonacci(Fibb, n-1)
5. Set Fib:=Fiba+Fibb
6. Return
```

7.8.3 Divide and conquer algorithm

A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. Two examples of divide and conquer algorithms are quick sort and binary search.

The binary search algorithms divides the given sorted set into two halves so that the problem of searching for an item in the entire set is reduced to searching for an item in only one half.

The quicksort algorithm uses a reduction step to find the location of a single element and to reduce the problem of sorting the entire set to the problem of sorting smaller sets.

7.9 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 the location of the last element of the queue. The condition FRONT=NULL will indicate that queue is 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 is incremented by 1. Whenever an element is deleted from the queue, the value of FRONT is incremented by 1.

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

7.10 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
        Then write (‘Overflow’)
        Return
2. [Increment Rear]
        R=R+1
3. [Insert element]
        Q[R] =Y
4. IF F=0
        Then F=1
5. Exit
```

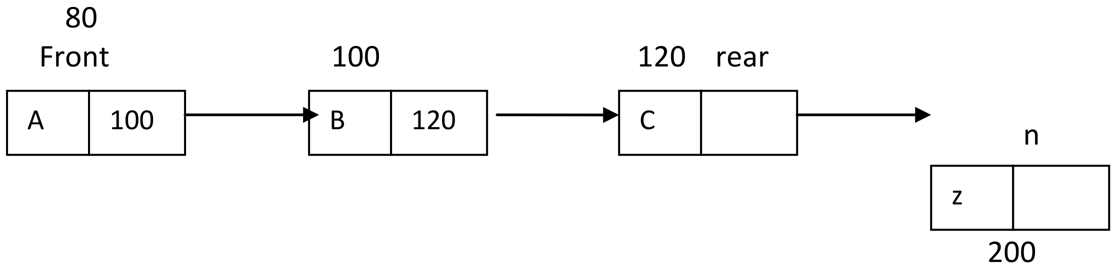
Function QDELETE()

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
        Then write('Underflow')
        Return 0;

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

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

        Return Y
    3.  Exit
```

7.11 Linked list representation of QUEUE

A linked queue is a queue implemented as a linked list with 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 and the NEXT is the pointer to the next element in queue.

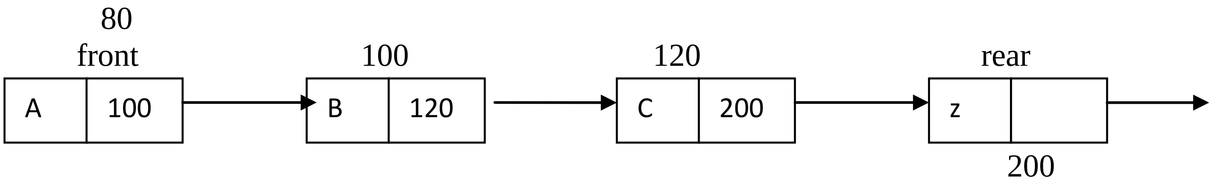


Procedure LQINSERT(INFO,NEXT, ITEM, REAR, FRONT)

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
```

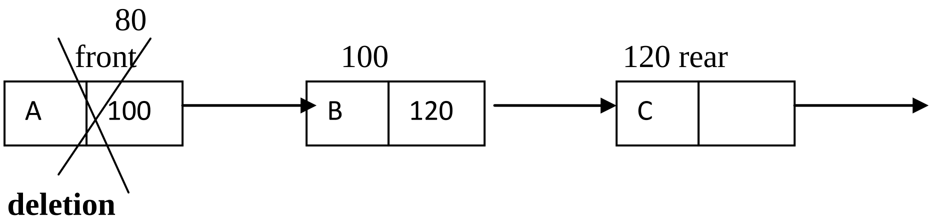
```
8.  Rear→next=n
9.  Rear=n
10. N→next=NULL
11. finished
```



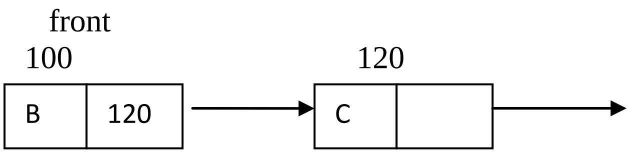
Procedure LQDELETE(INFO, NEXT,FRONT, REAR, ITEM,TEMP)

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



7.12 QUEUE 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* createqueeu(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;
}
```

7.13 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 this 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.

```
1.[reset rear pointer?]
        If R=N then
                R=1
        Else
                R=R+1

2.[ Overflow?]
        If F=R then
                 write('overflow')
        Return

3.[Insert element]
        Q[R]=Y

4.[Is the front pointer properly set?]
        If F=0
                Then F=1
        Return
```

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.

```
1.]underflow]
        If F=0 then
                 write('Underflow')
        Return 0;

2.[Delete element]
        Y=Q[F]
3.[Queue Empty?]
        If F=R then
                 F=0
```

```
                R=0
        Return Y

4.[increment front pointer]
        If F=N then
                F=1
        Else
                F=F+1
        Return Y
```

7.14 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.** The term "circular" 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 deque

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.

7.15 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 comes 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.

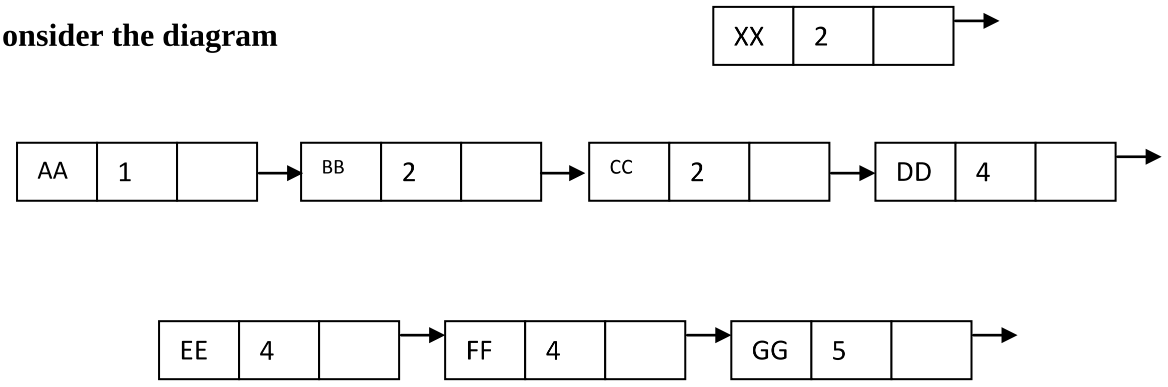
7.15.1 One way representation of a priority queue

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 information 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.

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.

7.15.2 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

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

7.16 Applications of queues

Based on complexity, the two significant applications of queues are given below

1. Categorizing data
2. Queue simulation

7.16.1 Categorizing data

There is always a necessity to rearrange data making sure that there is no change in their basic sequence. Consider a list of numbers. The task is to arrange the numbers thereby making sure that the original order s maintained in each group. This is an excellent example of a multiple queue application.

Consider the following list of numbers

1 22 12 6 10 34 65 29 9 30 81 4 5 19 20 57 44 99

Suppose we want to categorize them into 4 different groups:

Group 1: less than 10
Group 2: between 10 and 19
Group 3: between 20 and 29
Group 4: 30 and greater

The list is arranged as shown below

|3 6 9 4 5|12 10 19|22 29 30|34 65 30 81 57 44 99|
The output is a list which is not sorted but is categorized according to the rules specified. The order of the numbers in each group have their original order.

7.16.2 Categorizing data design

One of the simplest solutions for categorizing the data design is to build a queue for all four categories. Later the numbers can be stored in the relevant queue as we read them. When we reach a point where all the data has been processed, we can print the queue. The following algorithm shows this design

The following algorithm shows how a list of members can be arranged inn 4 groups:

1. Algorithm(categorize)
2. CreateQueue(q0to9)
3. CreateQueue(q10to19)
4. createQueue(q20to29)
5. creyaeQueue(qOver29)
6. fillQueues(q0to9,q10to19, q20to29,q20to29);
7. printQueues(q0to9,q10to19, q20to29,q20to29);
8. exit

The queue named **categorize** is created. It calls the algorithm to fill and print the numbers in the form of 4 groups. On completion of the program the main queue is not destroyed in any case.

7.17 Assignment 7

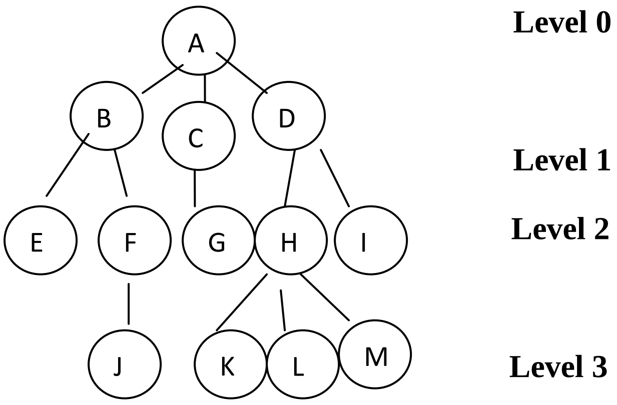
1) What is a stack? What are operations involved in a stack?
2) Write the PUSH and POP operations in a stack.
3) What is a queue? What are operations involved in a queue.
4) Write an algorithm to insert and delete elements from/to a queue.
5) Name any two applications of a stack.
6) Write an algorithm to convert infix expressions to postfix expressions
7) Write an algorithm to evaluate postfix expressions.
8) Write the algorithm for linked list representation of a stack
9) Write the algorithm for linked list representation of a queue.
10) Name any 2 application of a queue.
11) Write an algorithm to insert and delete elements from/to a circular queue.
12) Write a note on priority queue and deque
13) What is recursion? Give any 2 recursive techniques.

UNIT IV
Chapter 8
TREES

8.1 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 subtrees.



8.2 Binary Tree Terminology

1. Root: it is the first in the hierarchical arrangement of data items. In the above tree, A is the root item.
2. Node: each data item in a tree is called a node. It specifies the data information and links to other data items. There are 13 nodes in the above tree.
3. Degree of a node: it is the number of subtrees of a node in a given tree. In the above tree
    a. The degree of node A is 3
    b. Degree of C is 1
    c. Degree of D is 2
    d. Degree of H is 3
    e. Degree of I is 0
4. Degree of a tree: it is the maximum degree of the nodes in a given tree. In the above tree the node A has degree 3, the node H is also having degree 3. So the degree of the tree is the maximum degree of all nodes. So the degree of the above tree is 3.
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 E,G,I,J,K, L and M.
6. Non terminal Nodes: any node except the root node whose degree is non zero is called non terminal node. There are 5 non terminal nodes in the above tree. They are B,C,D,F and H.

7. Siblings: the children nodes of a given parent node are called siblings. They are also called brothers. In the above tree,
    a. E & F are siblings of parent node B.
    b. K,L and M are siblings of parent node H.
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 level1 and their intermediate children are at level2 and so on. 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 t another node 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,B), (B, F) and (F, J)

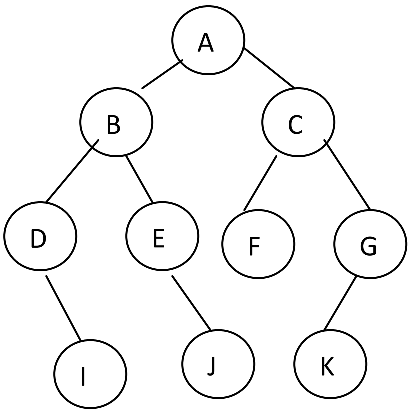
11. Depth: it is the maximum level of any node in a given tree. In the above tree, the root node A has the maximum level. The term *height* is also used to denote the depth.

12. Forest: it is the set of disjoint trees. In a given tree, if you remove its root node then it becomes a forest. In the above tree, there is a forest with 3 trees.

8.3 Binary Trees

A binary tree is a finite set of data items which 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 non linear data structure. In a binary tree, the maximum degree of any node is atmost 2. That means there may be a zero degree node or one degree node and two degree node.

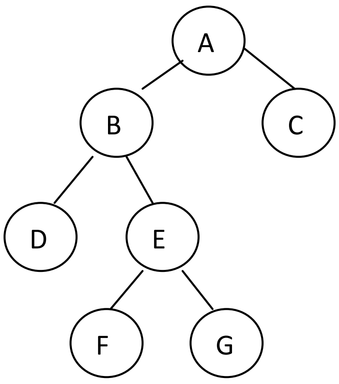


A binary tree

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 left subtree with root D and right subtree with root E and so on.

8.4 Strictly Binary tree

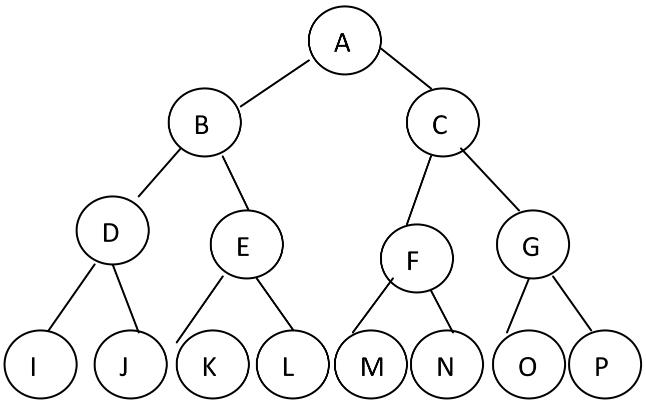
It is a binary tree, in which every node has either 0 or 2 children.



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

8.5 Complete binary tree

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



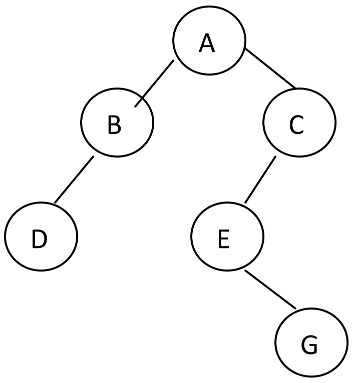
A complete binary tree with depth 4

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.

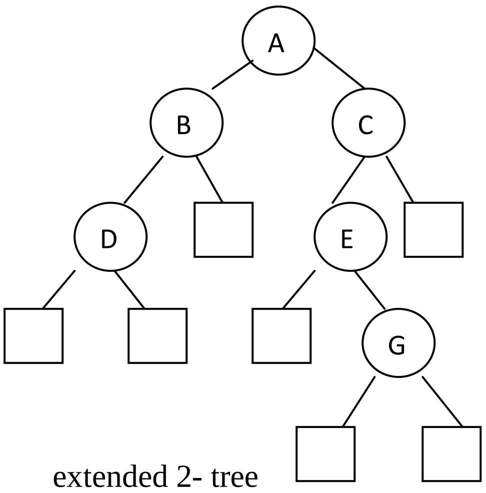
8.6 Extended binary trees

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 nodes 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 form 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 a 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

8.7 Binary tree representation

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 node consists of 3 fields such as

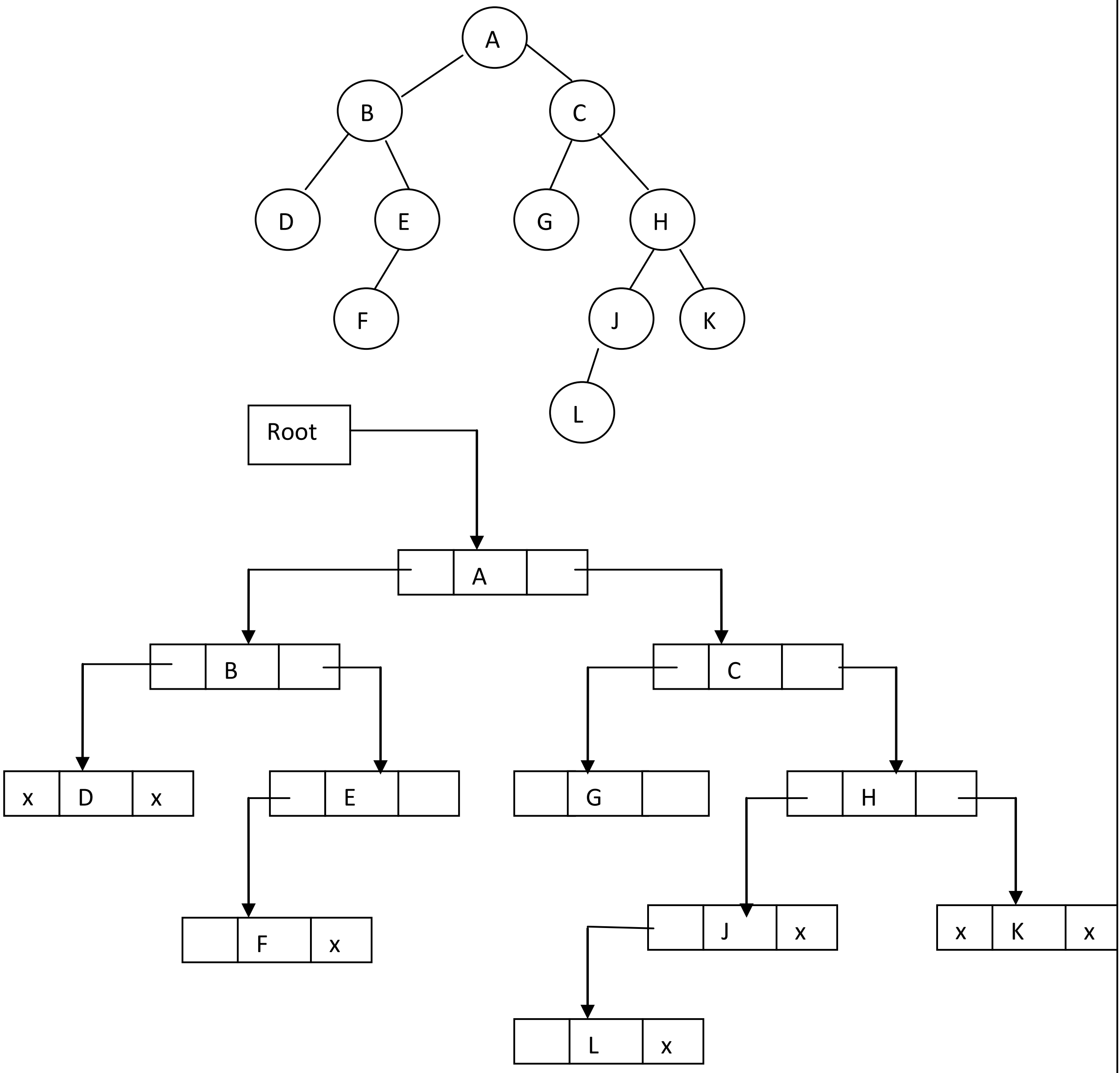
1) Data
2) Left child
3) Right child.

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

Lchild	data	rchild

```
Class node
{
        Char data;
Node *lchild;
Node * rchild;
};
```

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



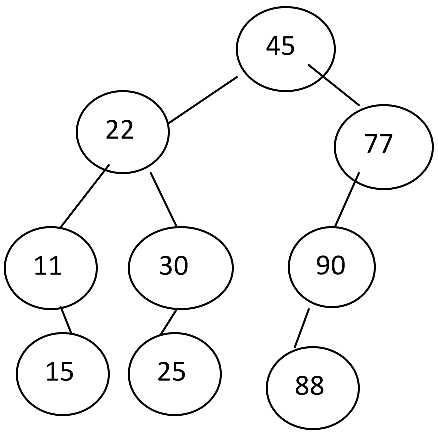
A Binary tree in linked representation

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 and right child nodes. But the terminal nodes don’t have left and right child nodes. Their *lchild* and *rchild* pointer are set to NULL. Here, non terminal nodes are called ***internal nodes*** and the terminal nodes are called ***external nodes.***

8.8 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 T in memory called the sequential representation of T. this representation uses only a single linear array TREE as follows:

1) The root R of T is stored in TREE[1]
2) If a node N occupies TREE[K], then its left child is stored in TREE[2*K] and its right child is stored in TREE[2*K+1]



45	22	77	11	30		90		15	25				88				

8.9 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.

8.10 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

8.10.1 Preorder traversal

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

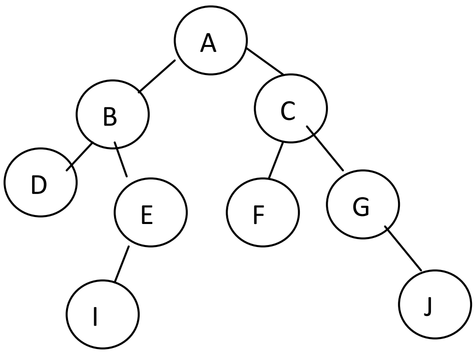
1. Visit the root node
2. Traverse the left subtree in preorder
3. Traverse the right subtree in preorder

In a preorder traversal the root node is visited before traversing its left and right subtrees. 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 order traversal

Procedure Preorder(root)

```
1. If (root !=NULL)
   Begin

2. Print root→info
3. Preorder(root→left)
4. Preorder(root→right)
   End

5. finished
```



The preorder traversal of the above binary tree is ABDEICFGJ

8.10.2 Inorder Traversal

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

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

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

Algorithm for inorder order traversal

```
Procedure Inorder(root)

1. If (root !=NULL)
   Begin

2. Inorder(root→left)
3. Print root→info
4. Inorder(root→right)
   End

5. finished
```

8.10.3 Postorder Traversal

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

1) Traverse the left subtree in Postorder
2) Traverse the right subtree in Postorder

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 right subtree is taken up and is traversed in postorder. Finally, the data at the root is displayed. The Postorder traversal of the above binary tree is DEIBFJGCA

Algorithm for Postorder order traversal

Procedure Postorder(root)

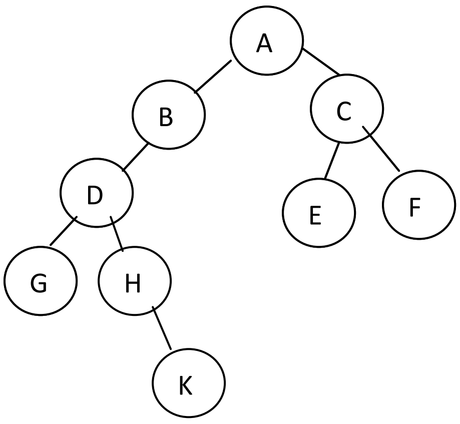
```
1. If (root !=NULL)
   Begin
2. inorder(root→left)
3. inorder(root→right)
4. Print root→info
   End
5. Finished
```

8.11 Traversal algorithms using stacks

8.11.1 Preorder traversal

The preorder traversal algorithm uses a variable temp which will contain the location of the node N currently being scanned. This algorithm uses an array S which will hold the address of nodes for the future processing.

Consider the following binary tree.



Algorithm : Preorder(INFO, LEFT, RIGHT, ROOT)

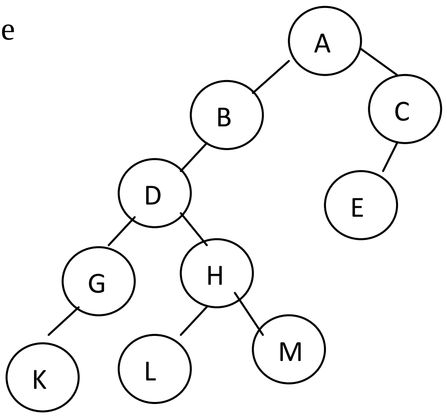
A binary tree T is in memory. This algorithm does a preorder traversal of T applying an operation DISAPLY to each of its node. An array S is used temporarily holds the address of nodes.

```
1. Initially push NULL onto Stack S and initialize temp
   Set top:=1, S[1]=NULL and temp=root
2. Repeat steps 3 to 5  while (temp!=NULL)
3. Display temp→info
4. If temp→right!=NULL then
   Begin
           Set top:=top+1
           Set S[top]:=temp→right
   [End of if]
5. If temp→left!=NULL then
           Set temp:=temp→left
   Else
   Begin
           Set temp:=s[top]
           Set top:=top-1
   End of else
   End of step 2 loop
6. Exit
```

8.11.2 Inorder traversal

The preorder traversal algorithm uses a variable temp which will contain the location of the node N currently being scanned. This algorithm uses an array S which will hold the address of nodes for the future processing. With this algorithm, a node is processed only when it is popped from the stack.

Consider the following binary tree



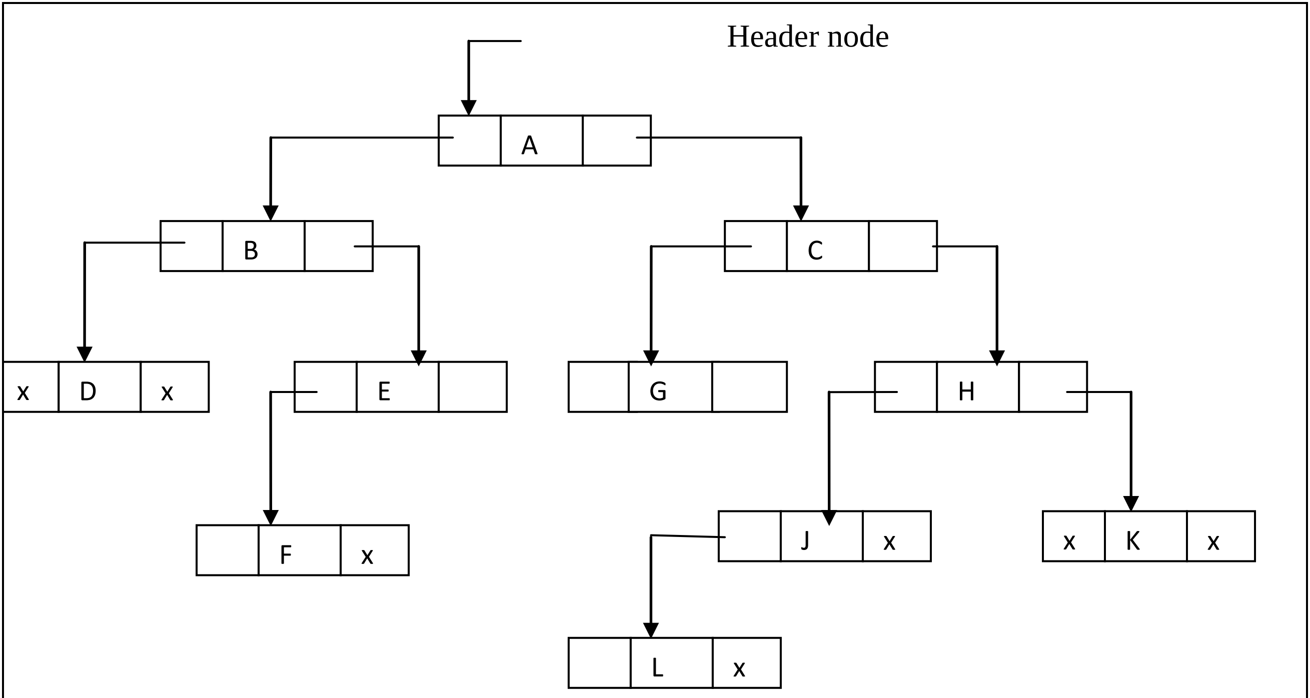
Algortihm: inorder(info, left, right,root)

A binary tree is in memory. This algorithm does an inorder traversal of T, applying an operation DISPLAY to display each of its nodes. An array S is used to temporarily hold the address of nodes.

```
1. [push NULL into stack S and initialize temp]
   Set TOP:=1
   Set S[1]:=NULL
   Set temp:=Root
2. Repeat while(temp!=NULL)
   Begin
         Set top:=top+1
         Set s[top]:=temp
         Set Temp:=temp→left
   End while
3. Set Temp:=s[top] and Set top=top-1
4. Repeat steps 5 to7 while temp!=NULL
   Begin
5. Display temp→info
6. If tem→right!=NULL then
   Begin
         Set Temp:=temp→right
         Goto step 2
   End if
7. Set temp:=s[top] and Top=top-1
   End of while loop at step 4
8. Exit
```

8.12 Header nodes: threads

Suppose a binary tree T is maintained in memory by means of a linked list 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 the head node will point to the root of T. for example,



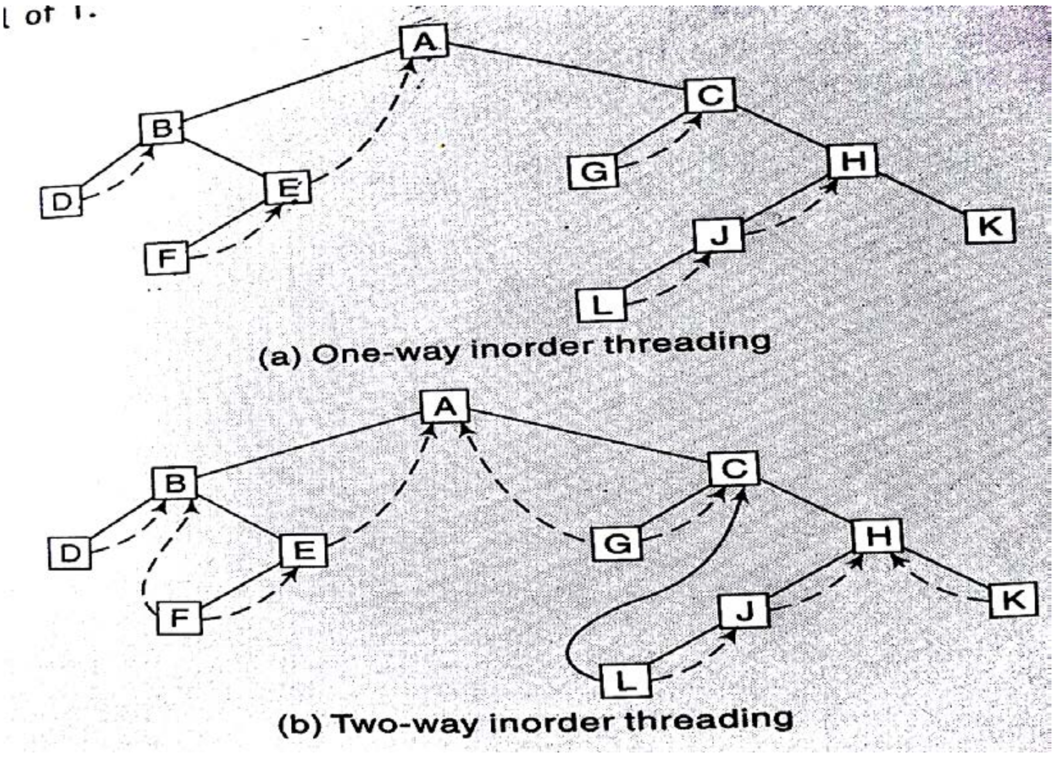
Suppose a binary tree T is empty. Then T will still contain a header node, but the left pointer of the header node will contain the null value. Thus the condition, **head→left=NULL** will indicate an empty tree.

8.13 Threads; inorder threading

Consider again the linked list representation of a binary tree T. half of the entries in the pointer fields LEFT and RIGHT will contain null elements. This space may be more efficiently used by replacing the null entries by some other type of information. Specifically we will replace certain null entries by special pointers which point to nodes higher in the tree. These special pointers are called threads and binary trees with such pointers 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 one way 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



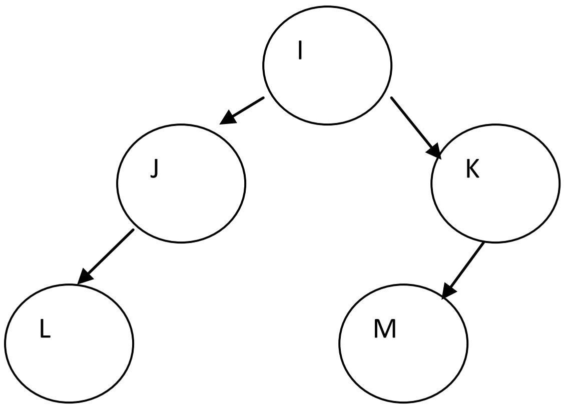
(a) One-way inorder threading

(b) Two-way inorder threading

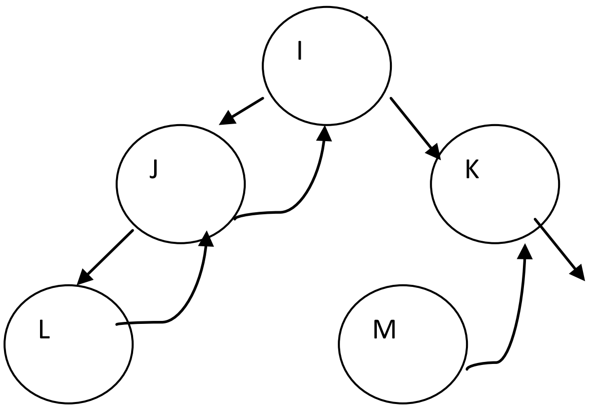
8.14 Threaded binary trees

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 and time.

Consider a binary tree given below.

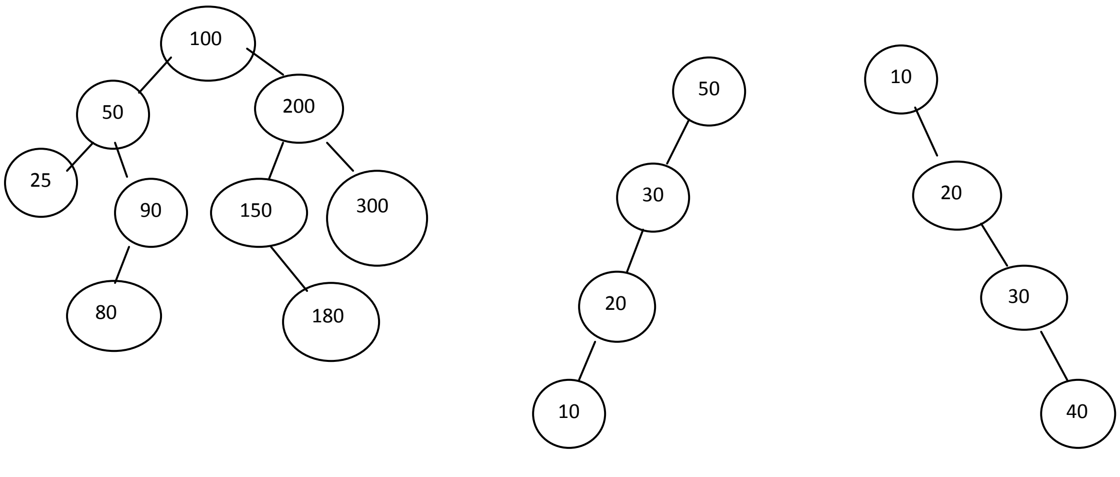


Let us make a thread binary tree out of a normal binary tree



8.15 Binary Search Tree (BST)

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



8.16 Searching and inserting in Binary Search trees

Function INSERT(element, root)

```
1. Node=new Node
2. Node→info=element
3. Previous=NULL
4. CURRENT=root
5. While(CURRENT!=NULL)
   Begin
6. Previous=CURRENT
```

```
7.  If (CURRENT→info > element)
    Begin
8.  CURRENT=CURRENT→left                    //search code
    Else
9.  CURRENT=CURRENT→right
    End if
    End while
10. If element < previous→info
    Begin
11. Previous→left=node                      // insert code
    Else
12. Previous→right=node
    End if
13. Finished
```

8.17 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 of children node N.
The are three cases

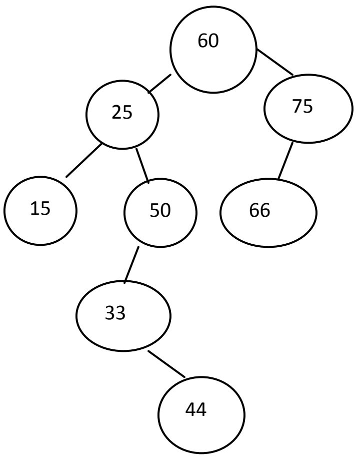
Case 1: N has no children. Then N is deleted from the tree T by simply replacing the location of N in the parent node P(N) by the NULL pointer.

Case 2: N has exactly one child. Then N is deleted from the tree T by simply replacing the location of N in P(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 ode S(N).

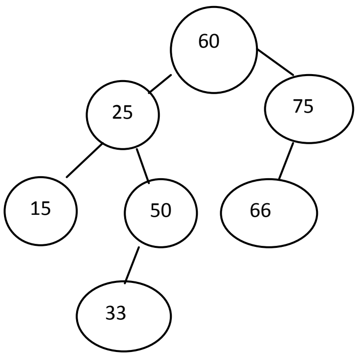
For example, consider the binary search tree in the following figure. Suppose T appears in memory as shown below.

Before deletion



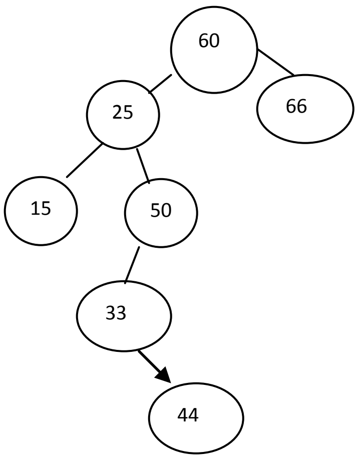
Address	INFO	LEFT	RIGHT
1	60	2	3
2	25	4	7
3	75	8	0
4	15	0	0
5			
6			
7	50	9	0
8	66	0	0
9	33	0	10
10	44	0	0
11			
12			
13			

case 1: node 44 is deleted



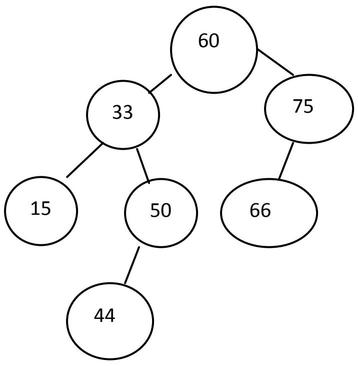
Address	INFO	LEFT	RIGHT
1	60	2	3
2	25	4	7
3	75	8	0
4	15	0	0
5			
6			
7	50	9	0
8	66	0	0
9	33	0	~~10~~ 0
10	~~44~~	0	0
11			
12			
13			

Case 2: node 75 id deleted



Address	INFO	LEFT	RIGHT
1	60	2	~~3~~ 8
2	25	4	7
3	~~75~~	~~8~~ 0	0
4	15	0	0
5			
6			
7	50	9	0
8	66	0	0
9	33	0	10
10	44	0	0
11			
12			

Case 3: node 25 is deleted

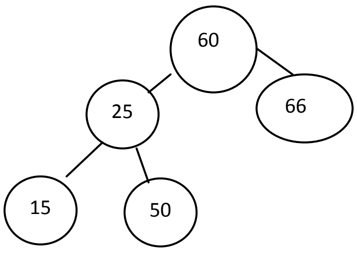


Address	INFO	LEFT	RIGHT
1	60	~~2~~ 9	3
2	~~25~~	~~4~~ 0	~~7~~ 0
3	75	8	0
4	15	0	0
5			
6			
7	50	~~9~~ 10	0
8	66	0	0
9	33	0 4	~~10~~ 7
10	44	0	0
11			
12			
13			

8.18 Balanced binary trees

Assume that there is a tree T. to check whether T is a balanced tree, we need to calculate its balance factor, which is the difference in height between the left and the right subtrees. Let the height of the left subtree be Hl and the height of the right subtree be Hr. now we know that Hl and Hr are balance factors of the tree, B can be determined by B=Hl+Hr

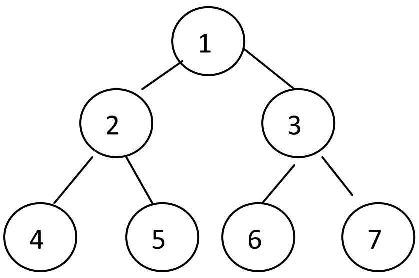
Let us calculate the balance of the given tree using the formula



The balance factor of the given tree is 1.

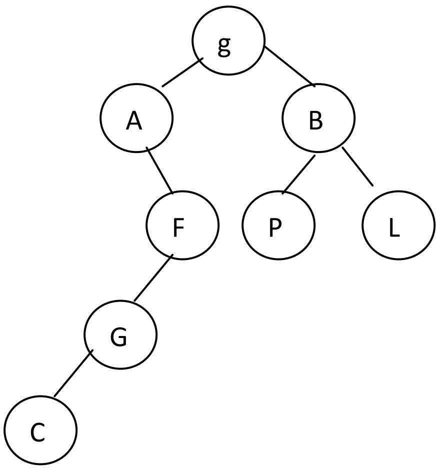
8.19 Weight balanced tree

A height balanced binary tree has let and right subtrees which differ in height by not more than one. A complete binary tree is always height balanced.



8.20 Weight balanced binary tree

Weight balanced binary trees are a type of balanced binary trees. Here the trees are balanced in order to keep the sizes of the subtrees of all the nodes within a constant factor of all the present nodes. In such types of trees, the weight is proportional to the number of associations present in that tree.



8.21 Assignment 8

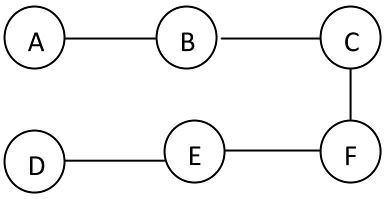
1) What is a tree? What is a complete binary tree
2) What is an extended binary tree?
3) What is the degree of a tree?
4) What is a path?
5) What is a forest?
6) What is the depth of a tree?
7) Give the sequential representation of a binary tree.
8) Give the linked list representation of a binary tree.
9) Write the algorithm using stack for the preorder and inorder traversal of a tree.
10) What is the binary search tree? Write the algorithm to insert and search for anode in BST.

Chapter 9
Graphs and their applications

9.1 Introduction

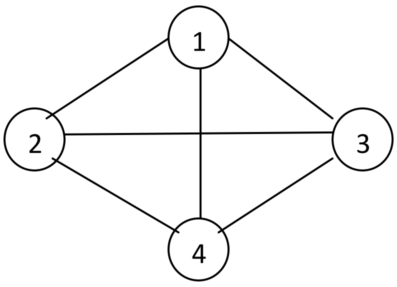
A graph is a collection of nodes called vertices and line segments called arcs or edges that connects pairs of nodes.

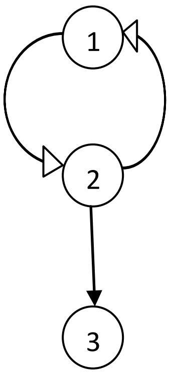
A graph G consists of two sets V and E. V is a finite non empty set of vertices. E is a set of pairs of vertices, these pairs are called edges. We will also write G=(V,E) to represent a graph.



In an undirected graph, the pair of vertices representing any edge is unordered. Thus pairs (v1,v2) and (v2, v1) represent the same edge. In a directed graph, each edge is represented by a directed pair <v1, v2>. V1 is the tail and v2 is the head of the edge. Therefore <v1, v2> and <v2, v1> represent two different edges.

Following figure shows two graphs G1 and G2





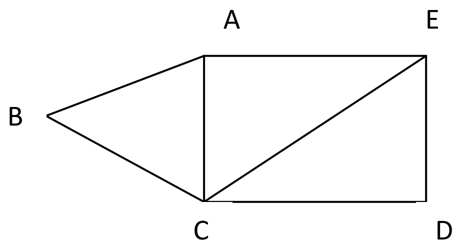
V(G1) ={1,2,3,4}
E(G1)={ (1,2) (1,3) (1,4) (2,3)(2,4) (3,4)}

V (G2)= {1,2,3}
E(G2)={ (1,2)(2,1) (2,3)}

Note that edges of a directed graph are drawn with an arrow from the tail to the head.

9.2 Graph terminology

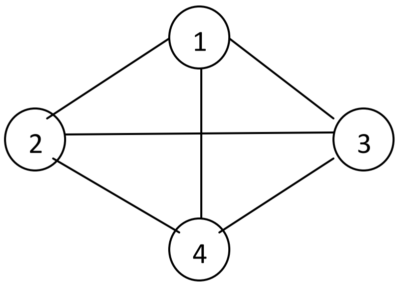
A graph is said to be connected if there is a path between any two of its nodes. A graph is said to be complete if every node is adjacent to every other node in G. clearly such a graph is connected.

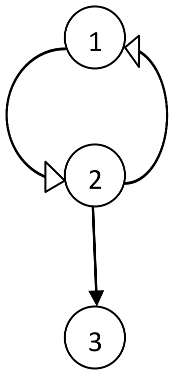


The above graph is a picture of a connected graphs with 5 nodes A,B,C,D and E and 7 edges [A B] [B, C] [C D] [D E] [A E] [C E] [A C]. There are simple paths of length 2 from B to E: (B A E) and (B C E). There is only one simple path of length 2 form B to D: (B C D). We note that (B A D) is not a path, since [A,D] is not an edge. There are two 4-cycles in the graph. They are [A B C E A] and [A C D E A]

In an undirected graph, the pair of vertices representing any edge is unordered. Thus pairs (v1,v2) and (v2, v1) represent the same edge. In a directed graph, each edge is represented by a directed pair <v1, v2>. V1 is the tail and v2 is the head of the edge. Therefore <v1, v2> and <v2, v1> represent two different edges.

Following figure shows two graphs G1 and G2





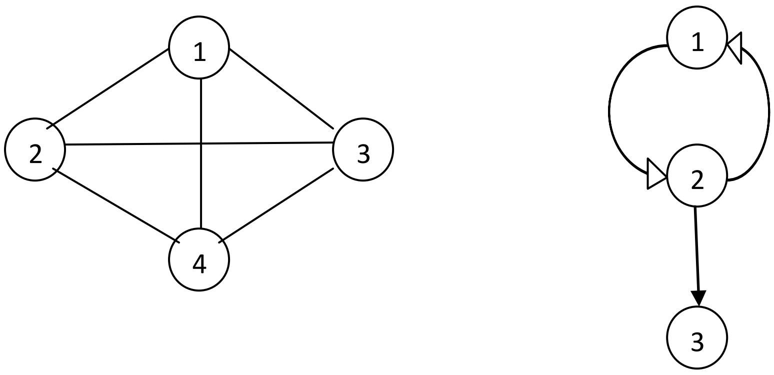
V(G1) ={1,2,3,4}
E(G1)={ (1,2) (1,3) (1,4) (2,3)(2,4) (3,4)}

V (G2)= {1,2,3}
E(G2)={ (1,2)(2,1) (2,3)}

Note that edges of a directed graph are drawn with an arrow from the tail to the head.

In an undirected graph, the pair of vertices representing any edge is unordered. Thus pairs (v1,v2) and (v2, v1) represent the same edge. In a directed graph, each edge is represented by a

directed pair <v1, v2>. V1 is the tail and v2 is the head of the edge. Therefore <v1, v2> and <v2, v1> represent two different edges.
Following figure shows two graphs G1 and G2

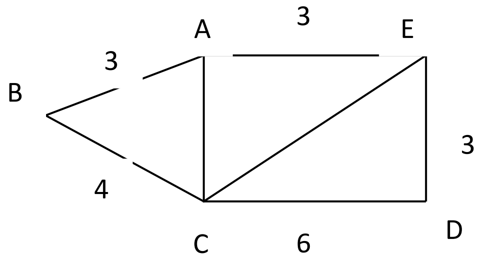


V(G1) ={1,2,3,4}
E(G1)={ (1,2) (1,3) (1,4) (2,3)(2,4) (3,4)}

V (G2)= {1,2,3}
E(G2)={ (1,2)(2,1) (2,3)}

Note that edges of a directed graph are drawn with an arrow from the tail to the head.

The degree deg (A)=3, since A belongs to 3 edges. Deg(c)=4 and deg(D)=2



A graph is said to be labeled if its edges are assigned data. In particular G is said to be weighted if each edge e in G is assigned a non negative numerical value called weight or length of e. in such case each path in a graph G is assigned a weight or length which is the sum of the weights of the edges along the path.

The above graph is a weighted graph. Path p1=(B C D) and P2=(B A E D) are both paths from node B to node D. although P2 contains more edges than P1 the weight w(p2)=9 is less than the weight w(p1)=10.

9.2.1 Multiple edges

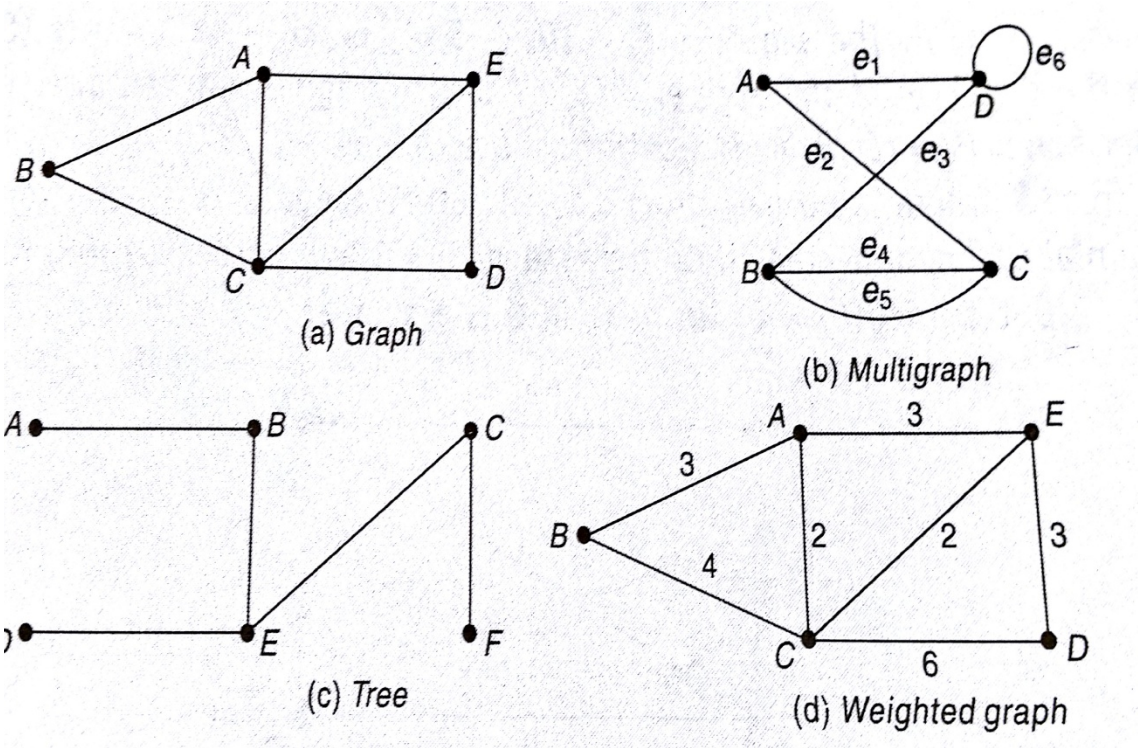
Distinct edges e and e^1 are called multiple edges if they connect the same endpoints that is e=[u,v] and e^1= [u, v]

9.2.2 Loops

An edge e is called a loop if it has identical endpoints, that is e=[u ,u]

Such a generalization M is called a **Multigraph**. A Multigraph is said to be finite if it has a finite number of nodes and a finite number of edges.

A connected graph without any cycles is called a tree path or free tree or simply a tree. This means there is a unique path between any two nodes in T.
In the tree graph with m=6 nodes and consequently, m-1 =5 edges. There is a unique simple path between any two nods of the tree graph.



(a) Graph (b) Multigraph (c) Tree (d) Weighted graph

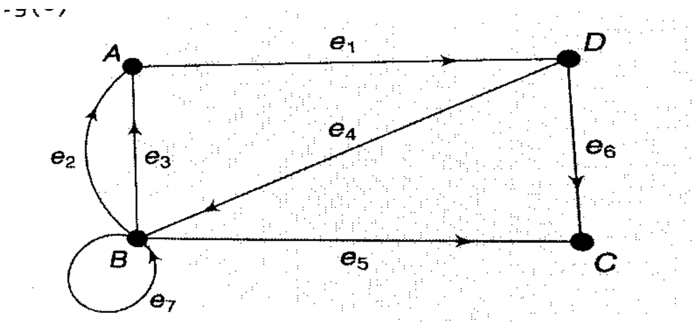
The above graph is weighted. P1=(B C D) and P2=(B A E D) are both paths from node B to node D. although P2 contains more edges than P1, the weight w(p2)=9 is less than the weight w(P1)=10.

9.2.3 Directed graphs

A directed graph G, also called a digraph or graph, is the same as a Multigraph except that each egde in G is assigned a direction, or in other words, each edge e is identified with an ordered pair (u,v) of nodes in G rather than an unordered pair[u,v]

Suppose G is a directed graph with a directed edge e=(u,v), then e is also called an arc. The outdegree of a node u in G, written in outdeg(u), is the number of edges beginning at u. similarly, the indegree of u, written indeg(u), is the number of edges ending at u. a node u called the source if it has the positive outdegree but zero indegree. Similarly, u is called a sink if it has a zero outdegree but a positive indegree.

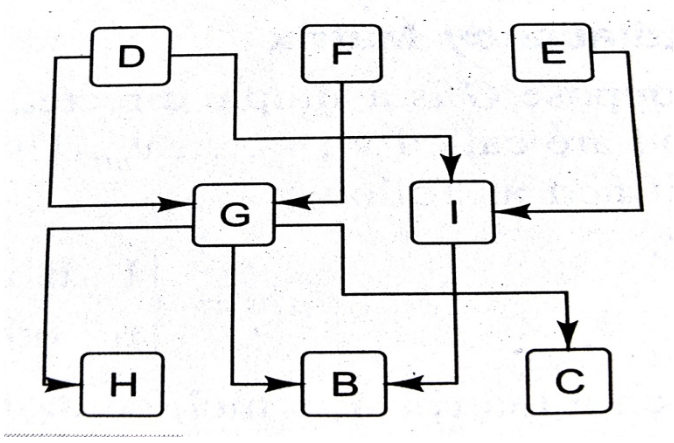
A directed graph G is said to be connected, or strongly connected., if for every pair u, v of nodes in G there is a path from u to v and there is also a path from v to u.



A directed graph G is said to be simple if G has no parallel edges. A simple graph may have loops, but it can not have more than one loop at a given node.

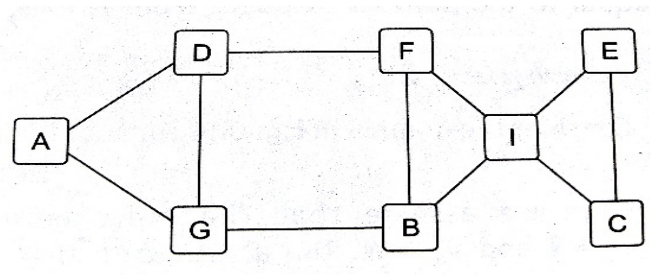
9.2.4 Directed acyclic graphs

These graphs are directed graphs with no cycles. For a vertex in a DAG, there is no directed path starting and ending with V. here is an example of a DAG shown below. A sink is a vertex with only a single edge ending on it. A source is a vertex having edges starting form it. In the above figure vertices H, B and C are sink vertices, while vertices D, F and E are source vertices. Vertices G and I do not come under any of these groups they have edges starting from them and ending on them.



9.2.5 Bi connected graphs

A bi-connected graph is a connected graph which cannot be broken down into any further pieces by deletion of any single vertex. Let us assume that G is a bi-connected graph. There are no separation edges and vertices in graph G. a separation edge is an edge whose removal disconnects G and a separation vertex is a vertex is a vertex which disconnects G when removed. For any two given vertices vv and u of G, there are two disjoint simple paths which are present between u and v and do not share any other vertices or edges. There is a simple cycle which has both u and v for any of the two vertices u and v of G.



9.3 Sequential representation of graphs: adjacency matrix; path matrix

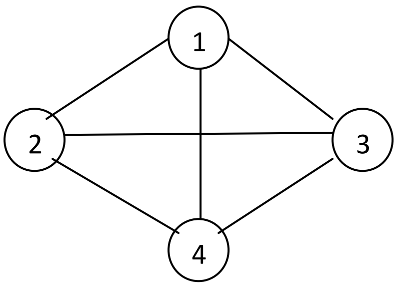
There are 2 standard ways of maintaining a graph in memory of a computer. One way called the sequential representation of G is by means of adjacency matrix. The other way is called the linked list representation of G.

9.3.1 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 an edge (vi,vj)
0 otherwise

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



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

We can see from the above matrices that the adjacency matrix for undirected graph is symmetric. The adjacency matrix for the directed graph is not symmetric.

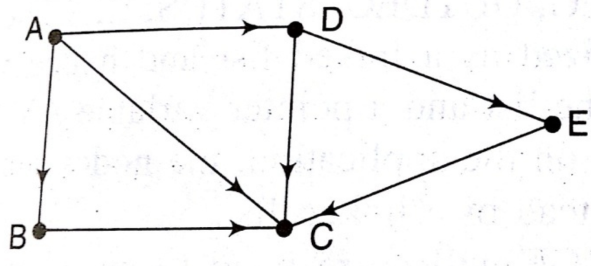
9.3.2 Path matrix

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.

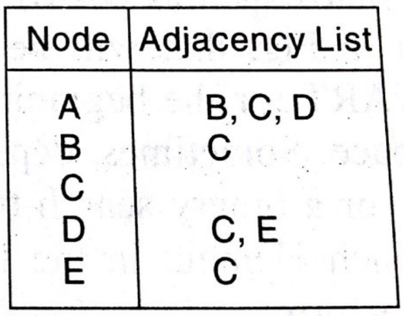
Pij=1 if there is a path from vi to vj
0 otherwise

9.4 Linked list representation of a graph

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



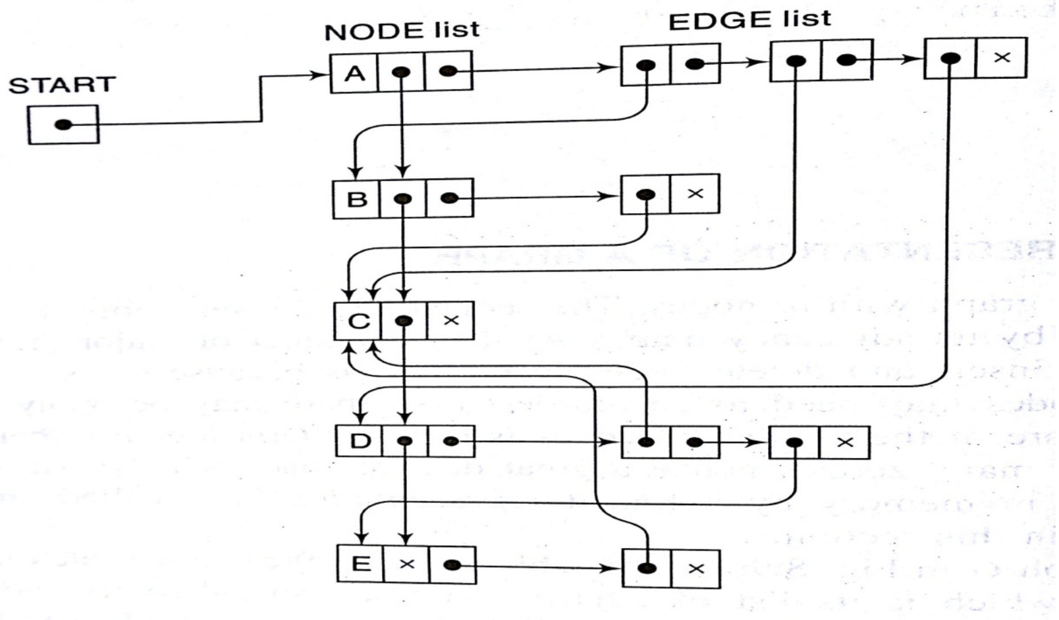
(a) Graph *G*



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.



9.5 Operations on a graph

9.5.1 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
```

9.5.2 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
```

9.5.3 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 node M in G.
3) Delete all the edges beginning at N. this is accomplished by finding the location BEG of the first 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.

9.6 Traversing a graph

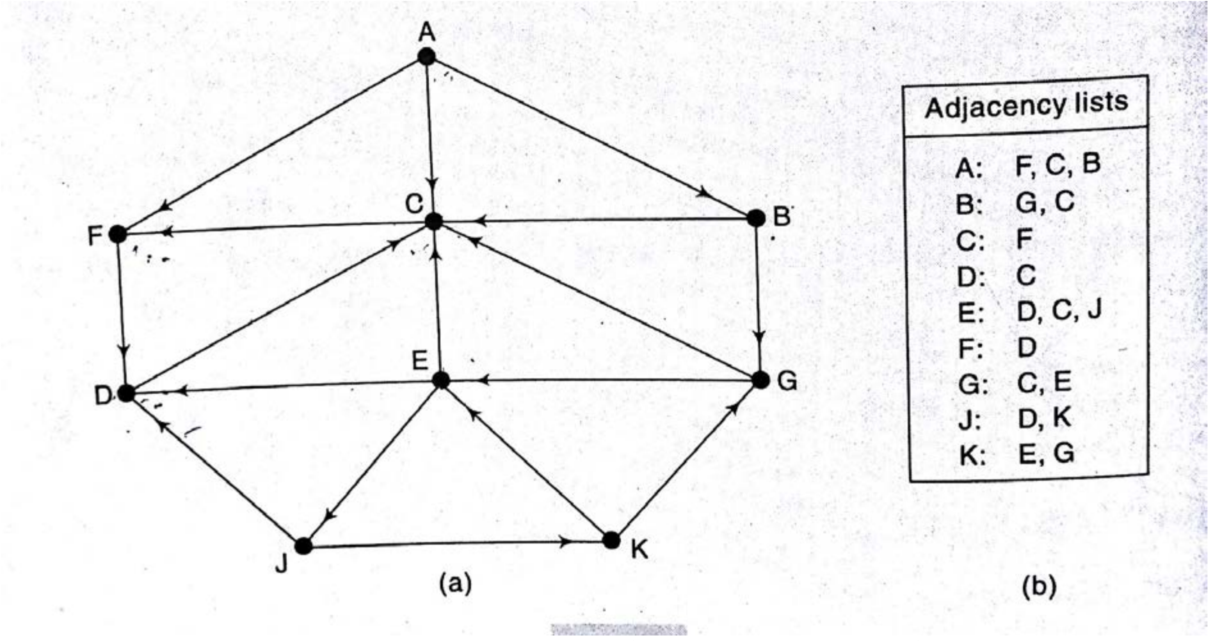
There are two standard ways to traverse a graph. They are Breadth-First Search and Depth First Search. The BFS uses a queue to hold nodes for future processing and DFS uses a stack.

During the execution of our algorithm, each node N of G will be one of the three states, called the status of N. they are

Status=1: the initial state of the node n.(ready state)
Status=2: the node N is on the queue or stack, waiting to be processed.(waiting state)
Status 3: the node N has been processed (processes state)

9.6.1 Breadth First Search

The general idea behind a BFS beginning at a starting node A is as follows. First we examine the starting node A. then we examine all the neighbors of A. then we examine all the neighbors of the neighbors of A. and so on. We need to keep track of the neighbors of a node, and we need to guarantee that no node is processed more than once. This is accomplished by using a queue to hold nodes that are waiting to be processed and by using a filed STATUS which tells us the current status of any node. The algorithm is as follows.



Adjacency lists	
A:	F, C, B
B:	G, C
C:	F
D:	C
E:	D, C, J
F:	D
G:	C, E
J:	D, K
K:	E, G

Consider the above graph. Suppose we want to find the shortest path from the node A to J with minimum number of path, then BFS is used. The minimum path can be found by using a BFS beginning at node A and ending when J is encountered. During the execution of search, we will keep track of the origin of each edge by using an array ORIGIN together with the array QUEUE.

Step 1: Add A to the QUEUE

QUEUE	A
ORIGIN	0

Step 2: Add QUEUE the neighbors of A and they are F C B

QUEUE	A	F	C	B
ORIGIN	0	A	A	A

Step 3: Add QUEUE the neighbors of F and it is D

QUEUE	A	F	C	B	D
ORIGIN	0	A	A	A	F

Step 4: Add QUEUE the neighbors of C and it is F. but F is not added to the QUEUE because it is already there in the QUEUE.

Step 5: Add QUEUE, the neighbor of B and they are G, C. but only G is added because C is already there in the QUEUE.

QUEUE	A	F	C	B	D	G
ORIGIN	0	A	A	A	F	B

Step 6: Add the neighbors of D to QUEUE. It is C. so C is not added.

Step 7: add neighbors of G. they are C and E. only E is added to the QUEUE.

QUEUE	A	F	C	B	D	G	E
ORIGIN	0	A	A	A	F	B	G

Step 8: Add neighbors of E to the QUEUE. They are D, C and J. only J is added to the QUEUE.

QUEUE	A	F	C	B	D	G	E	J
ORIGIN	0	A	A	A	F	B	G	E

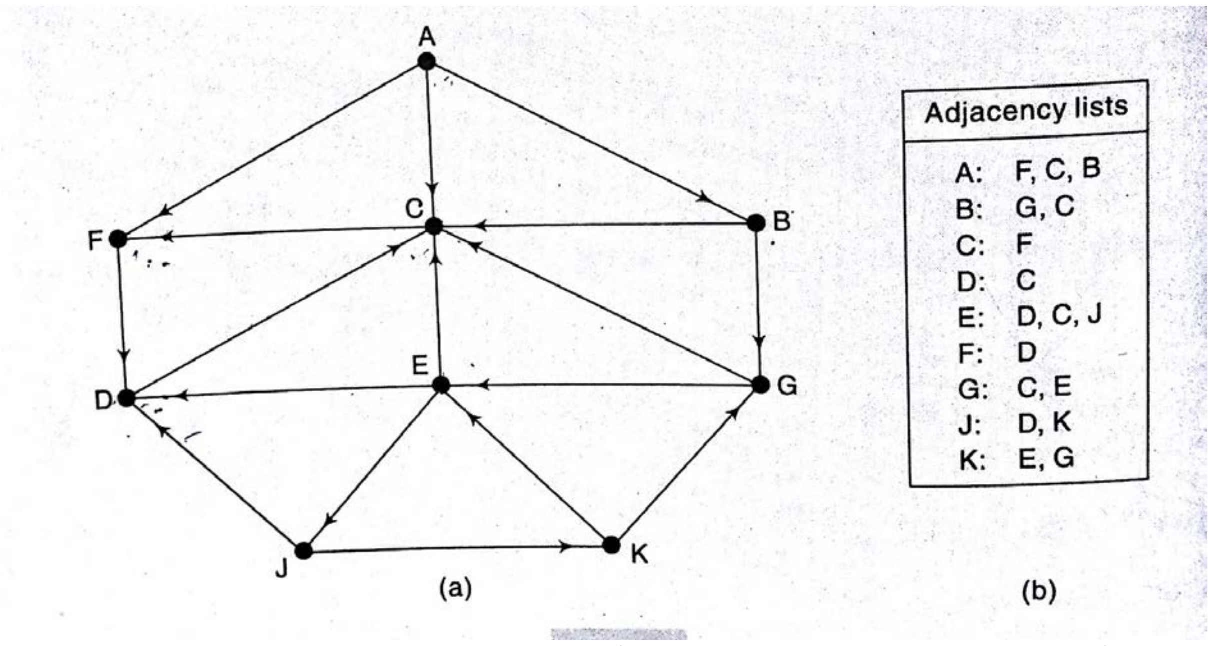
We stop as soon as J is added to the QUEUE since J is the final destination. We now backtrack from J, using the array ORIGIN to find the path P. thus J←E←G←B←A is the required path.

This algorithm executes a BFS search on a graph G beginning at a starting node A.

1. Initialize all nodes to the ready state i.e STATUS=1
2. Put the starting node A in QUEUEU and change its status to the waiting state. (status =2)
3. Repeat steps 4 and 5 until QUEUE is empty.
4. Remove the front node N of QUEUE. Process n and change the status of n to the processed state(STATUS=3)
5. Add to the rear of QUEUE all the neighbors of N that are in the ready state and change their status to the waiting state
   End of loop
6. Exit

9.6.2 Depth First Search

The general idea behind a DFS beginning at a stating node A is as follows. First we examine the starting node A. then we examine each node N along a path P which begins at A. that is we process a neighbor of A, then a neighbor of neighbor of A and so on. After coming to a “dead end” that is to the end of path P, we backtrack on P until we can continue along another path p^1 and so on. This algorithm is similar to the inorder traversal of a binary tree. Here we use a stack instead of queue. We use a STATUS filed to tell the current status of the node.



Adjacency lists

A: F, C, B
B: G, C
C: F
D: C
E: D, C, J
F: D
G: C, E
J: D, K
K: E, G

Consider the above graph. Suppose we want to find and print all the nodes reachable from node J including J. one way to do this is to use a DFS starting at node J. the steps are as follows.

Step 1: initially, Push J onto the stack

Stack	J

Step 2: Print J and push the neighbors of J on to the stack. They are D and K

Stack	D	K

Step 3: print K and push the neighbors of K on to the stack. They are E and G

Stack	D	E	G

Step 4: print G and push the neighbors of G on to the stack. They are C and E. but E is already in the stack. So push only C to stack.

Stack	D	E	C

Step 5: print C and push the neighbors of C. it is F.

Stack	D	E	F

Step 6: Print F and push the neighbors of F. it is D. but is already in the stack.

Stack	D	E

Step 7: print E and push the neighbors of E and they are D, C and J. but these elements already processed once.

Stack	D

Step 8: print D. now stack is empty. So the DFS of the graph starting at J is now complete. Accordingly the nodes are printed. J, K, G, C, F, E and D are the nodes which are reachable from J.

This algorithm executes a depth first search on a graph G beginning at a starting node A.

1. Initialize all the nodes to the ready state (status=1)
2. Push the starting node onto stack and change its status to the waiting state (status =2)
3. Repeat steps 4 and 5 until stack is empty.
4. Pop the top node N of stack. Process N and change its status to the processed state(status=3)
5. Push onto stack all the neighbors of N that are still in the ready state(status=1), and change their status to the waiting state (status=2)
   End of step 3 loop
6. Exit

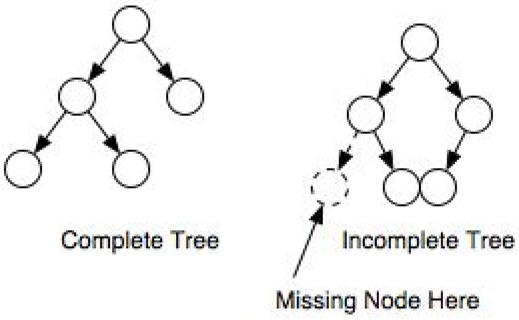
9.7 Assignment 9

1) Define a graph.
2) What is a weighted graph?
3) What is a directed graph?
4) What is a directed acyclic graph?
5) What is bi- connected graph?
6) Give the sequential representation of a graph
7) Give the linked list representation of a graph.
8) Write the DFS algorithm
9) Write BFS algorithm

Value Added Topics

What is a heap?

A heap is a partially sorted binary tree. Although a heap is not completely in order, it conforms to a sorting principle: every node has a value less than either of its children. Additionally, a heap is a "complete tree" -- a complete tree is one in which there are no gaps between leaves. For instance, a tree with a root node that has only one child must have its child as the left node. More precisely, a complete tree is one that has every level filled in before adding a node to the next level, and one that has the nodes in a given level filled in from left to right, with no breaks..



Why use a heap?

A heap can be thought of as a priority queue; the most important node will always be at the top, and when removed, its replacement will be the most important. This can be useful when coding algorithms that require certain things to processed in a complete order, but when you don't want to perform a full sort or need to know anything about the rest of the nodes. For instance, a well-known algorithm for finding the shortest distance between nodes in a graph, Dijkstra's Algorithm, can be optimized by using a priority queue.

Heaps can also be used to sort data. A heap sort is O(nlogn) efficiency, though it is not the fastest possible sorting algorithm.

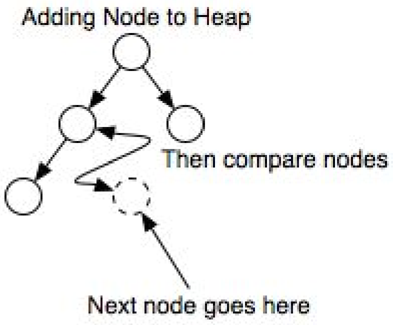
How do you implement a heap?

Although the concept of a heap is simple, the actual implementation can appear tricky. How do you remove the root node and still ensure that it is eventually replaced by the correct node? How do you add a new node to a heap and ensure that it is moved into the proper spot?

The answers to these questions are more straight forward than meets the eye, but to understand the process, let's first take a look at two operations that are used for adding and removing nodes from a heap: upheaping and down heaping.

Upheap: The upheap process is used to add a node to a heap. When you upheap a node, you compare its value to its parent node; if its value is less than its parent node, then you switch the two nodes and continue the process. Otherwise the condition is met that the parent node is less than the child node, and so you can stop the process. Once you find a parent node that is less than the node being upheaped, you know that the heap is correct--the node being upheaped is greater than its parent, and its parent is greater than its own parent, all the way up to the root.

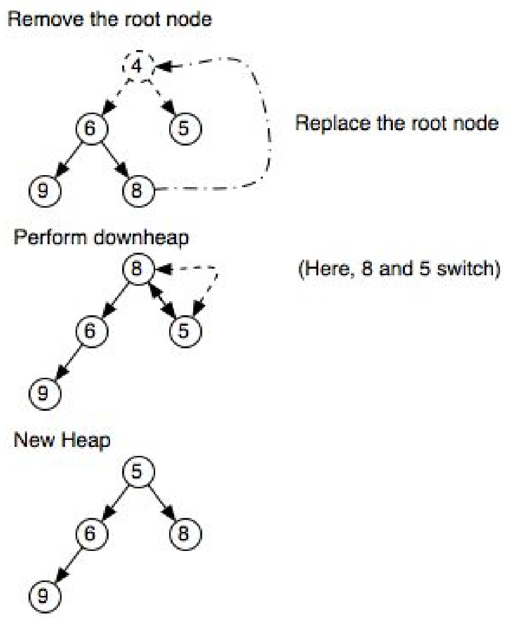
Downheap: The downheap process is similar to the upheaping process. When you downheap a node, you compare its value with its two children. If the node is less than both of its children, it remains in place; otherwise, if it is greater than one or both of its children, then you switch it with the child of lowest value, thereby ensuring that of the three nodes being compared, the new parent node is lowest. Of course, you cannot be assured that the node being downheaped is in its proper position -- it may be greater than one or both of its new children; the downheap process must be repeated until the node is less than both of its children.

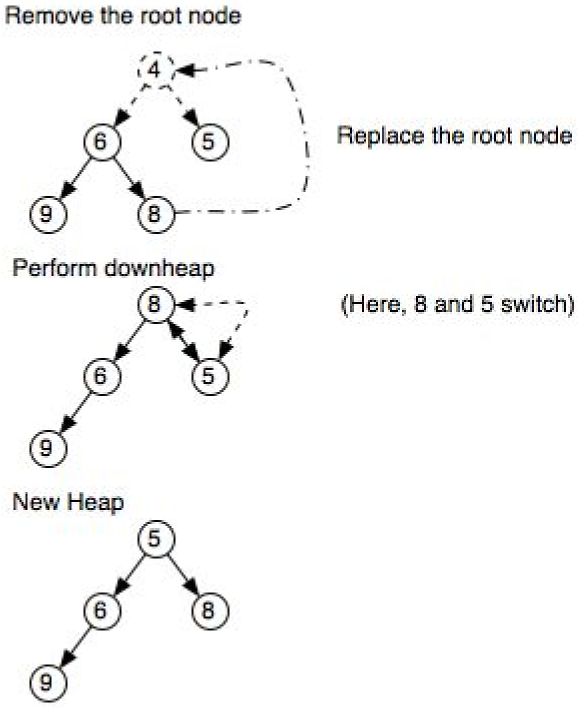


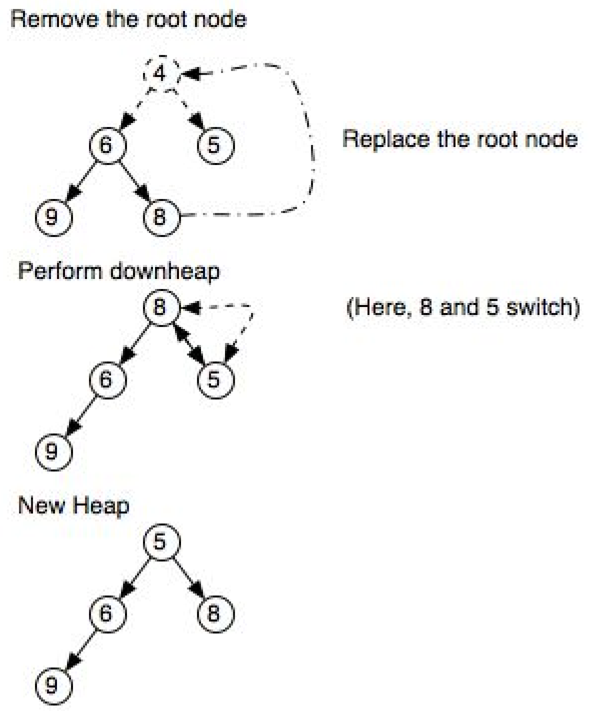
When you add a new node to a heap, you add it to the rightmost unoccupied leaf on the lowest level. Then you upheap that node until it has reached its proper position. In this way, the heap's order is maintained and the heap remains a complete tree.

Removing the root node from a heap is almost as simple: when you take the node out of the tree, you replace it with "last" node in the tree: the node on the last level and rightmost on that level.

Once the top node has been replaced, you downheap the node that was moved until it reaches its proper position. As usual, the result will be a proper heap, as it will be complete, and even if the node in the last position happens to be the greatest node in the entire heap, it will do no worse than end up back where it started.


Remove the root node
4
6
5
Replace the root node
9
8
Perform downheap
8
(Here, 8 and 5 switch)
6
5
9
New Heap
5
6
8
9


Remove the root node
4
6
5
Replace the root node
9
8
Perform downheap
8
(Here, 8 and 5 switch)
6
5
9
New Heap
5
6
8
9



Efficiency of a heap

Whenever you work with a heap, most of the time taken by the algorithm will be in upheaping and down heaping. As it happens, the maximum number of levels of a complete tree is log(n)+1, where n is the number of nodes in the tree. Because upheap or downheap moves an element from one level to another, the order of adding to or removing from a heap is O(logn), as you can make switches only log(n) times, or one less time than the number of levels in the tree (consider that a two level tree can have only one switch).

Hash Tables

Hash tables are an efficient implementation of a keyed array data structure, a structure sometimes known as an associative array or map. If you're working in C++, you can take advantage of the STL map container for keyed arrays implemented using binary trees, but this article will give you some of the theory behind how a hash table works.

Keyed Arrays vs. Indexed Arrays

One of the biggest drawbacks to a language like C is that there are no keyed arrays. In a normal C array (also called an indexed array), the only way to access an element would be through its index number. To find element 50 of an array named "employees" you have to access it like this:

```
employees[50];
```

In a keyed array, however, you would be able to associate each element with a "key," which can be anything from a name to a product model number. So, if you have a keyed array of employee records, you could access the record of employee "John Brown" like this:

```
employees["Brown, John"];
```

One basic form of a keyed array is called the hash table. In a hash table, a key is used to find an element instead of an index number. Since the hash table has to be coded using an indexed array, there has to be some way of transforming a key to an index number. That way is called the hashing function.

Hashing Functions

A hashing function can be just about anything. How the hashing function is actually coded depends on the situation, but generally the hashing function should return a value based on a key and the size of the array the hashing table is built on. Also, one important thing that is sometimes overlooked is that a hashing function has to return the same value every time it is given the same key.

Let's say you wanted to organize a list of about 200 addresses by people's last names. A hash table would be ideal for this sort of thing, so that you can access the records with the people's last names as the keys.

First, we have to determine the size of the array we're using. Let's use a 260 element array so that there can be an average of about 10 element spaces per letter of the alphabet.

Now, we have to make a hashing function. First, let's create a relationship between letters and numbers:

A --> 0
B --> 1

C --> 2
D --> 3
...
and so on until Z --> 25.

The easiest way to organize the hash table would be based on the first letter of the last name.

Since we have 260 elements, we can multiply the first letter of the last name by 10. So, when a key like "Smith" is given, the key would be transformed to the index 180 (S is the 19 letter of the alphabet, so S --> 18, and 18 * 10 = 180).

Since we use a simple function to generate an index number quickly, and we use the fact that the index number can be used to access an element directly, a hash table's access time is quite small. A linked list of keys and elements wouldn't be nearly as fast, since you would have to search through every single key-element pair.

Collisions and Collision Handling

Problems, of course, arise when we have last names with the same first letter. So "Webster" and "Whitney" would correspond to the same index number, 22. A situation like this when two keys get sent to the same location in the array is called a collision. If you're trying to insert an element, you might find that the space is already filled by a different one.

Of course, you might try to just make a huge array and thus make it almost impossible for collisions to happen, but then that defeats the purpose of using a hash table. One of the advantages of the hash table is that it is both fast and small.

Collision handling with open addressing

The simplest collision handling algorithm is known as the open address method or the closed hashing method. When you are adding an element, say "Whitney," and you find that another element is already there ("Webster," for instance) then you would just proceed to the next element space (the one after "Webster"). If that is filled, you go on to the next one, and so on, until you find an empty space to insert the new element.

```
...
220 "White"   | <-- ### COLLISION ### : Gotta move on to the next.
221 "Webster" | <-- ### COLLISION ### : Next one.
222           | Ahhh, perfect. Insert Here.
223           |
...
```

Since we modified the insertion algorithm, we also have to change the function that finds the element. You have to have some way of verifying that you've found the element you want, and not some other element. The simplest way is to just compare keys. (Does this record have the last name "Whitney"? Does this one?) If the element you find is not one of them, just move on to the next element until you reach the one you want or you find an empty space (which means the element is not in the table).

Sounds simple, right? Well, it gets more complicated. What if you have so many collisions that you run off the end of the array?

If you're trying to insert "Zorba" and all the elements are filled because of the collision handling, then what? Look at the example:

```
...
258 "Whitney"   | <-- Nope, not Empty
259 "Zeno"      | Nope, not Empty
---------------- <-- Ummm, what now?
```

The easiest thing to fdo is to just wrap around to the beginning again. If there are still no empty spaces, then we have to resize the array, since there isn't enough space in the hash table for all of the elements. If we resize the array, of course, we'll have to come up with a tweak to our hash function (or at least how we handle it) so that it covers the right range of values again, but at least we'll have room. (Note that resizing the array means that occasionally inserting a value into the list will cause an O(n) copy operation to take place, but that on average this should happen only once for every n items inserted, so insertion should be on average constant time, O(1).

Handling collisions with separate chaining

A second collision handling strategy is to store a linked list at each element in the hash data structure. This way, when a collision occurs, you can just add the element into the linked list that is stored at the hash index. If you have only a single element with a particular hash value, then you have a single element list--no performance penalty. If you have a lot of elements hashing to the same value, you'll see a slowdown of course, but no more than you otherwise would see with hash collisions.

One nice thing about separate chaining is that having a bunch of values that hash "near" each other is less important. With open addressing, if you have a cluster of values that hash to nearly the same value, you'll run out of open space in that part of the hash. With separate chaining, each element that has a different hash value will not impact the other elements.

Resizing dynamically based on a load factor

Generally speaking, you wouldn't want your hash table to grow completely full because this will make lookups take much longer. If a value isn't in the array, with open addressing, you have to keep looking until you hit an empty location or you get back to the starting point--in other words, with a completely full table, lookups could be O(n), which is horrible. A real hash table implementation will keep track of its load factor, the ratio of elements to array size. If you have a 10 element array, with 7 elements, the load factor is 0.7. In fact, 0.7 is generally about the right time to resize the underlying array.

Choosing a Good Hash Algorithm

The more collisions you have, the worse the performance of your hash table will be. With enough elements in your hash table, you can get an average performance that's quite good--essentially constant time O(1). (The trick is to make the array grow over time as you start to fill up the array.) But if you have a lot of elements that hash to the same value, then you will have to start doing a lookup through a list of elements that all have the same hash value. This can make your hash lookups go from constant time to being, well, linear time in the number of elements. Imagine if your hash function hashed all values to 0, putting them in the first element of the array.

Choosing a good hash algorithm can require some care and experimentation, and it will depend on your problem domain. If you're working with names, you probably don't want a hash algorithm that just looks at the first letter, because the letters of the alphabet are not used evenly--you'll find a lot more names that start with S than with Z. You also want to have your hash functions be fast--you don't want to lose all the time savings you're getting from the hash table because you're computing the hash function really slowly

BCACAC210

Credit based Third semester BCA degree Examination, Oct/Nov. 2104
New Syllabus (2013-14 Batch)
DATA STRUCTURES

Max. Marks:80

Part-A

1

a) What is data structure? List the different types. 2*10=20
b) What are the disadvantages of a queue over circular queue?
c) What is the formula for the calculation of one dimensional array?
d) What is sorting? Why it is necessary?
e) Differentiate between iteration method and recursion.
f) Define path and leaf node of a tree.
g) Write steps in pre-order traversal of a binary tree.
h) What is a binary tree? Give an example.
i) Define complete and labeled graph.
j) What is digraph? Give an example.
k) Mention any two applications of a stack.
l) Differentiate between linear search and binary search.

Part-B

Unit-1

2

a) Write a note on strings as ADT.
b) What is meant by algorithm? Write an algorithm to find largest element among ‘n’ element.
c) What is Linear array? Write algorithm for traversing Linear Arrays. (4+6+5)

3

a) Briefly explain any 5 data structure operations performed.
b) Write a note on sub algorithm with an example.
c) Write an algorithm for searching a number using a Binary Search. (5+5+5)

UNIT –II

4

a) Write an algorithm to insert a node after a given node in a linked list.
b) Explain Selection sort method with an example.
c) Write an algorithm for searching an element from an singly linked list. (5+5+5)

5

a) What is a linked list? Write an algorithm to traverse singly linked list.
b) Write an algorithm for deleting a node from a singly linked list.

c) Explain merge sort technique to sort an array of n numbers. (5+5+5)

UNIT-III

6

a) Write an algorithm to evaluate postfix expression. Explain with an example.
b) What is stack? Explain the operations performed on a stack.
c) What are priority queues and dequeues? Explain. (5+5+5)

7

a) What is recursion? Explain the algorithm to find the factorial of a number using recursion.
b) What is a queue? Explain the operations performed on a queue.
c) Convert the following infix expressions into its equivalent postfix expressions:
   i. (A+B-D)/(E-F)+G
   ii. (A-B)*(D/E) (5+6+4)

UNIT-IV

8

a) Explain the following terms in a Binary tree of level 4:
   i) Node
   ii) Degree of a node
   iii) Siblings
   iv) Path
b) The following are the traversals of a binary tree. Draw the corresponding tree.
   Preorder: GDHBEIAFCJ
   Inorder: ABDGHEICFJ
c) Expalin Depth First search algorithm for a graph with an example. (6+4+5)

9

a) Write the algorithm for the preorder traversal of a binary tree with an example.
b) What is meant by graph? Explain the linked list representation of graph.
c) Write the procedure for searching and inserting in a Binary Search Tree. (5+5+5)

****

III SEMESTER BCA

DATA STRUCTURES

Question Bank

PART A

Two Mark Questions

1. Define Data structure. Give classification of data structure.
2. Define Linear and non-linear data structures.
3. What are the main two types of data structures? Explain briefly.
4. Differentiate primitive and non-primitive data structure.
5. Give 2 examples for non linear data structure
6. What is the difference between linear and non linear data structure?
7. Distinguish algorithms with sub algorithms.
8. What are the 3 types of control structures used in algorithms.
9. Write any two algorithmic notations with example.
10. Define Abstract Data Type (ADT). Give one example.
11. Find the value of $\lceil 7.5 \rceil, \lfloor 7.5 \rfloor, \lfloor -7.5 \rfloor, \lceil -7.5 \rceil, \lfloor \sqrt{30} \rfloor, \lceil \sqrt{30} \rceil, \lfloor -18 \rfloor, \lceil -18 \rceil, \lfloor \pi \rfloor, \lceil \pi \rceil$
12. Find 26(mod 7), 495(mod 11), -26(mod 7), -371(mod 8)
13. Find $\lfloor \log_2 1000 \rfloor, \log_2 \frac{1}{16}, \log_2 32$.
14. List and explain some string operations.
15. What is an array? Give the formula to find the location of a particular element in one dimensional array.
16. Write an algorithm to traverse a linear array.
17. Mention various ways of representing two dimensional arrays in the memory.
18. What do you mean by base address of an array? Give example.
19. Give the formulae to find the address of a particular location in a two dimensional array.
20. Give two advantages of linked list over arrays.
21. What is Sparse Matrix?
22. What is meant by the term overflow and underflow?
23. What is linked list?
24. What do you mean by garbage collection?
25. What do you mean by dynamic memory allocation?
26. What role does the AVAIL list play in a linked list?
27. Write an algorithm to print information of all the nodes in the linked list.
28. Draw the structure of a doubly linked list and circular linked list.

29. How is singly linked list terminated? Give diagram.
30. Write one advantage and one disadvantage of linked list over arrays.
31. What is a circular linked list? Give diagrammatic representation.
32. What is a doubly linked list?
33. Mention any two types of linked list.
34. What are the advantages of doubly linked list?
35. Mention any 2 applications of linked list.
36. What is a doubly linked list? What is the advantage over singly linked list?
37. Give diagrammatic representation of singly linked list and doubly linked list.
38. Write an algorithm to check for stack full.
39. Write an algorithm to check for stack empty.
40. Give algorithm for POP operation of stack.
41. Give algorithm for PUSH operation of stack.
42. Why stack is called LIFO list?
43. What is LIFO list? Mention its applications.
44. Mention any two applications of STACK.
45. What is a queue? Why queue is called FIFO list?
46. Give any two examples for STACK.
47. Give any two examples for QUEUE.
48. How does STACK differ from QUEUE?
49. Differentiate queue and circular queue.
50. What are the applications of the QUEUE.
51. Suppose a data structure is stored in a circular queue with N memory locations. When will the queue be full?
52. How circular queue is different from queue?
53. What is meant by priority queue? What is its use?
54. Define dequeue. What are its types?
55. Evaluate ABC*+D- with proper step. Assume A=4,B=6,C=2 , D=-4
56. Evaluate AB+CD*/ with proper step. Assume A=2,B=3,C=5 , D=-5
57. Evaluate the postfix expression AB-CDE^*/ with proper step. Where A=5. B=1, C = 3 , D=4 , E=2
58. Evaluate abc*- with proper step. Given a=2, b = 1 , c=4
59. Convert the given Infix expression to postfix form.
    - i. A*B+(C^D/E/F)
    - ii. a/b*c-d+e/f*(g+h)
    - iii. (X + Y / Z * W ^ P) – R
    - iv. (X + Y * Z ^ A / 4) – P
    - v. (A + B) * (C – D) / E * F
    - vi. a * b + (c/d^e)-f

vii. (A - B * C ^ D) / (E + F)
viii. (a/b)*(c*f+(a-d)*e)
ix. A + B * (C – D * (E + F))
x. a*b-c^d+e/f

21. Explain infix and postfix notation with example.
22. What do you mean by recursive procedure?
23. Write an algorithm to find factorial of a number N using recursion.
24. Write an algorithm to generate Fibonacci series using recursion.
25. What do you mean by divide-and-conquer method? Give one example.
26. What is meant by the term overflow and underflow?
27. What do you mean by garbage collection?
28. What do you mean by dynamic memory allocation?
29. What role does the AVAIL list play in a linked list?
30. Write an algorithm to print information of all the nodes in the linked list.
31. Draw the structure of a doubly linked list and circular linked list.
32. How is singly linked list terminated? Give diagram.
33. Write one advantage and one disadvantage of linked list over arrays.
34. What is a circular linked list? Give diagrammatic representation.
35. What is a doubly linked list?
36. Mention any two types of linked list.
37. What are the advantages of doubly linked list?
38. What is meant by priority queue? What is its use?
39. Mention any 2 applications of linked list.
40. What is a doubly linked list? What is the advantage over singly linked list?
41. Give diagrammatic representation of singly linked list and doubly linked list.
42. What is sorting? Why is it necessary?
43. What is sorting and searching?
44. List one recursive and one non recursive sorting technique.
45. What is a pivot (key) element used in a quick sort?
46. Differentiate linear search and binary search.
47. Write an algorithm for linear search (sequential search)
48. What are the preconditions for binary search to be performed on a single dimensional array?
49. State condition(s) when binary search is applicable.
50. Write brief note on bubble sort method.
51. What is a tree? Give example.
52. Define binary tree.
53. What is a binary search tree? Give an example.
54. Draw a complete binary tree .

55. Briefly explain the depth of a binary tree with an example.
56. Define the following terms:
    (i) Siblings (ii) Extended binary tree
57. Define the following terms:
    (i) root (ii) degree of a node (iii) loop (iv) cycle
58. Define : (i) degree of a tree (iv) leaf node of a tree
59. Define level and depth of a binary tree.
60. Differentiate terminal node and non terminal node.
61. Mention the various operations on binary tree.
62. What is a tree? Give an example.
63. Give any two applications of binary tree.
64. Give the memory representation of a binary tree.
65. What do you mean by threaded binary tree? Give example.
66. What do you mean by one-way inorder threading and two-way inorder threading? Give example.
67. What is balanced tree? Give one example.
68. What is height balanced tree? Give example
69. What is weight balanced tree? Give example.
70. Draw a binary tree to represent (A+B+C)^(D+E)/F
71. Draw the tree for the expression (A+B)*(C+D)
72. Draw the binary tree for the expression (A*B) – ((C * D) / F). Write prefix equivalent using tree.
73. Draw the binary tree for the expression (A+B*C) ^ ((D-E)/C) Write prefix equivalent using tree.
74. Draw a binary tree to represent the following expression (a+b-c)/(e+f)
75. Mention any 2 application of tree.
76. Define graph and multigraph.
77. Define indegree and outdegree of a node v with example.
78. Define bi-connected graph with example.
79. What are two methods to implement graphs in memory?
80. Define adjacency matrix.
81. Define path matrix.
82. List various operation performed on graphs.
83. Write an algorithm to find location of the node containing ITEM in graph.
84. What are the status values of node N of graph G defined.
85. Let LIST be a linked list in memory. Write algorithm to search an element in the linked list.
86. Let LIST be a linked list in memory. Write algorithm to find the maximum element in the linked list.

87. Let LIST be a linked list in memory. Write algorithm to insert an element at the given location in the linked list.

PART-B

UNIT-I

1. What is a data structure? Explain the different types
2. Distinguish between (i) primitive & non primitive data structure (ii) array & list
3. Write any 5 algorithmic notations with an example.
4. Write a note control structured used in algorithms with example.
5. Explain the representation of one dimensional and two dimensional arrays in memory with example.
6. What is meant by one dimensional array? Discuss how the elements of one dimensional array are stored and accessed?
7. List any four operations performed on linear data structures. Describe each.
8. How do you store strings in memory? Explain each with an example.
9. Given a one dimensional array A[15] with base address of A as 1000, and each element occupies 2 bytes , find the location of A[10].
10. Consider the single dimensional array AAA[45] having base address 300 and 4 bytes is the size of each element of the array. Find the address of AAA[10] , AAA[25] and AAA[50].
11. In which two ways can the elements of a double dimensional array may be stored in computer's memory ? Explain.
12. How is computer memory allotted for a two dimensional array?
13. Given a two dimensional array A[10][20], base address of A being 1000 and width of each element is 4 bytes, find the location of A[8][15] when the array is stored as (i) column wise (ii) row wise
14. If an array B[11][8] stored as column wise and B[2][2] is stored at 1024 and b[3][3] at 1084 . Find the address of b[1][1]
15. If an array if integers A[10] is stored in the memory, assume A[3] is stored at location 2000. Find address of A[7].
16. Write an algorithm to insert and delete an element to/from a linear array with N elements.
17. How do you represent polynomial using an array? Explain with an example.
18. Write and explain the algorithm for bubble sorting procedure
19. Write an algorithm to search a given number using Binary search technique.
20. Explain the concept of binary search technique with example.
21. Write an algorithm to search a given number in a list of numbers using linear search method.
22. List advantages and disadvantages of linear search over binary search?
23. Write and explain sequential search algorithm.
24. Write and explain an algorithm to search a list of numbers using sequential search method.
25. What do you mean by searching? Explain various searching techniques.
26. Compare sequential and binary search techniques.

27. Write algorithm to sort list of numbers using bubble sort technique. Show the first two passes of bubble sort for the following data:
    (i) 65 , 25, 10 , 80 , 25 , 11 , 45 , 20 (ii) 65 , 25 , 9 , 75 , 20 , 12 , 40 , 32 , 51 , 0

UNIT-II

28. Write an algorithm to delete a node having ITEM in a singly linked list.
29. Write an algorithm to search for a given element in a singly linked list.
30. Write an algorithm to insert a node at the beginning of a linked list .
31. Explain with a neat diagrams, how can we insert and delete a node at a specified location in a singly linked list.
32. Write an algorithm to pop an element from a stack using linked list.
33. Write an algorithm that pushes an element on to a stack using linked list.
34. Write an algorithm to delete an element from a queue using linked list.
35. Write an algorithm to add an element onto a queue using linked list.
36. How do you represent a queue using linked list? Develop an algorithm to perform insert and delete operations.
37. How do you represent a stack using linked list? Develop algorithm to perform insert and delete operations.
38. What advantages, do you think, does a circular singly linked list have over a non-circular singly linked list? What disadvantages?
39. Explain different types of linked lists with neat diagrams?
    or
    Explain any three types of linked lists with neat diagrams.
40. What is a two way list? Write an algorithm to insert a node before a given node in a doubly linked list.
41. Explain single and double linked list with diagrams.
42. What do you mean by traversing a list? Explain the algorithm to traverse all the nodes of a doublylinkedlist.
    Write an algorithm to display all nodes of a doubly linked list.
43. Explain with a figure how can we insert and delete a node at a specified location in a doubly linked list.
44. What is a circular linked list? Explain its importance with example.
45. Write and explain the algorithm for the simple insertion sort technique.
46. Write and explain the algorithm for the selection sort technique.
47. Write and explain the algorithm for the radix sort technique.
48. Write and explain the algorithm for the shell sort technique.
49. Write and explain the algorithm for the merge sort technique.
50. Trace the following list of numbers using selection sort and insertion sort technique.
    23, 45, 56 22, 10, 27, 33, 48, 39,55, 88, 90
51. Trace the following list of numbers using merge sort :
    23, 45, 59 12, 10, 27, 33, 48, 39,55, 88, 30
52. Trace the following list of numbers using shell sort technique.
    25, 45, 56 12, 19, 27, 35, 48, 39,55, 88, 30
53. Trace the following list of numbers using radix sort technique.

234, 455, 563 122, 100, 276, 333, 482, 399,558, 887, 304

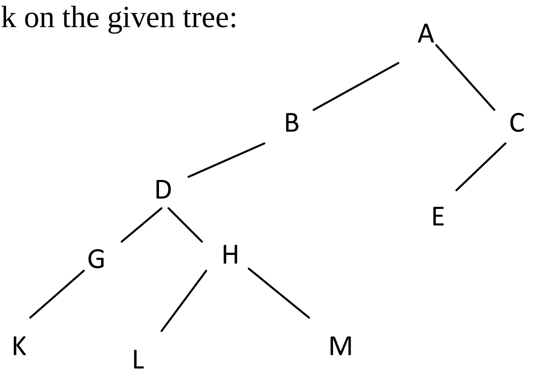
UNIT-III

54. What is the difference between an array and a stack housed in an array? Why stack is called a LIFO data structure? Explain how push and pop operations are implemented on a stack.
55. Write the algorithm to implement stack operations using array
56. Write an algorithm to implement PUSH and POP operations of stack.
57. What is stack? Mention its applications.
58. Describe the similarities and differences between queues and stack. (any two points each)
59. What is a queue? Write an algorithm to insert and delete an item into/from a circular queue.
60. Write a note on dequeue and priority queues.
61. Write a note on priority queue.
62. Write and explain an algorithm to convert the given infix expression to postfix. Trace your algorithm for the given infix expression. (A+B) ^ (C*D)
63. What are the advantages of using postfix notation over infix notation? Write an algorithm to evaluate a postfix expression.
64. Write a algorithms to accomplish the following stack operations
    - PUSH( )
    - POP( )
65. Given a stack as an array of 7 elements STACK: K , P , S, U , M , N
    (i) when will overflow and underflow occur?
    (ii) Can K be deleted before S? Why?
66. Write algorithms for inserting and deleting elements from a queue.
67. Write and explain an algorithm to evaluate postfix expression.
68. Write algorithm to evaluate given postfix expression. Use algorithm to evaluate the following postfix expression using a stack.
    50 , 40 , + , 18 , 14 , - , 4 , * , +
    Or
    100, 40, 8, +, 20 , 10 , - , + , *
    Or
    11, 5, - , 6, 8 , + , 12 , * , /
    Or
    5, 6, 9 , + , 80 , 5 , * , - , +
69. Distinguish between infix, prefix and postfix algebraic expressions giving examples of each.
70. How do you represent a queue using linked list? Develop an algorithm to perform insert and delete operations.

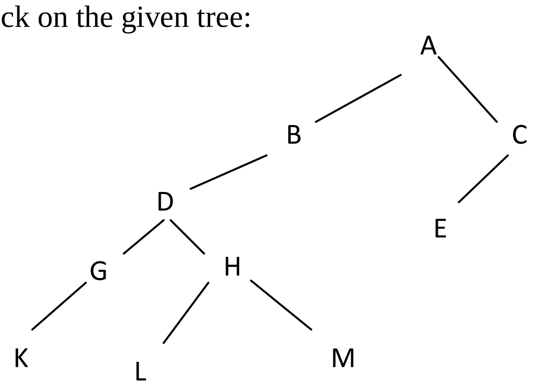
71. How do you represent a stack using linked list? Develop algorithm to perform insert and delete operations.
72. Use quick sort technique to sort the following list of numbers.
    22 , 12 , 32 , 2 , 15 , 25 , 10
    OR
    25 , 15 , 32 , 12 , 45 , 21 , 35 , 6
73. Use quick sort algorithm to sort the following sequence of numbers 22, 12, 32, 2, 15, 25, 10
74. Write an algorithm to convert infix to postfix expression. Use algorithm to convert given infix expression Q: A+(B*C-(D/E↑F)*G)*H to postfix expression.
75. What is recursive procedure? Write an algorithm to find factorial of a number using recursion?
76. Write an algorithm to generate Fibonacci series using recursion? What do you mean by divide-and-conquer method? Give one example.

UNIT-IV

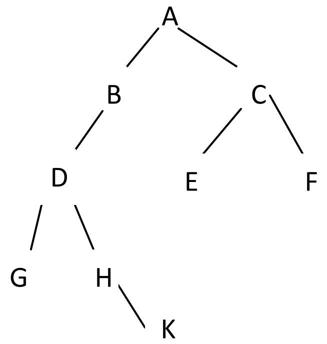
77. Explain with examples, 2 methods of tree representation memory.
78. Define the following tree terminology
    a. Siblings
    b. Path
    c. Node
    d. Forest
    e. Complete binary tree
    f. 2-trees
    g. Binary tree
    h. Depth
    i. Threaded binary tree.
79. What are the 3 standard ways of traversing a tree T with root R. write steps of each traversal using recursion.
80. Draw the binary tree for the given algebraic expression: [a+(b-c)]*[(d-e}/(f+g-h)]. Also write preorder, postorder and inorder traversal methods for the tree.
81. Write an algorithm to traverse a binary tree using inorder traversal method using stack.
82. Write an algorithm to traverse a binary tree using preorder traversal method using stack.
83. Write an algorithm to traverse a binary tree using postorder traversal method using stack.
84. Trace inorder traversal algorithm using stack on the given tree:



85. Trace postorder traversal algorithm using stack on the given tree:



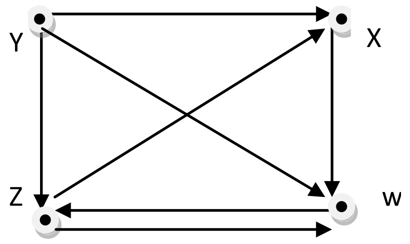
86. Trace preorder traversal algorithm using stack on the given tree:



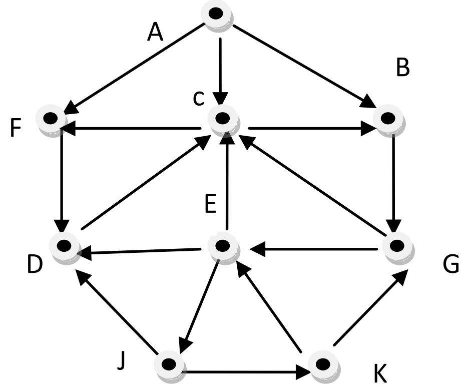
87. What do you mean by one-way inorder threading and two-way inorder threading? Give example.
88. Draw a binary search tree for the following list of numbers and traverse it in Inorder, post order and preorder: 14,15,4,9,7,18,40,45,76,13
89. Construct tree for the given infix expression:
    [a + (b-c)] * [(d-e) / (f + g – h)] traverse it in Inorder, post order and preorder
90. Explain the linked list representation of a binary tree with example.
91. Construct a binary search tree for the following:
    66,26,22,34,47,79,48,32,78 and traverse it in Inorder, post order and preorder.
    OR
    Construct a binary search tree for the following:
    14 , 15 , 4 , 9 , 7 , 18 and traverse it in Inorder, post order and preorder.
92. Write an algorithm to search an ITEM in binary search tree T.
93. Write an algorithm to add a node contains new information ITEM to the binary search tree.
94. Write 3 steps involved in deletion of a node in binary search tree with an example.
95. Define the following terms:
    - **a.** graph
    - **b.** digraph
    - **c.** multigraph
    - **d.** path
    - **e.** indegree
    - **f.** outdegree

- **g.** loop
- **h.** simple graph
- **i.** DAG
- **j.** Biconnected graph

96. Explain the method of representing the graphs using sequential method with an example.
97. What is adjacency matrix and path matrix, explain with an example.
100. Define adjacency matrix and path matrix and also write the same matrices for the following graph G:



101. Explain linked representation of the graph with an example.
102. Write an algorithm to insert a new node containing information N into the graph G.
103. What are the 4 steps involved in the deletion of a node N in the graph G.
104. Write an algorithm for breadth first search(BFS) and depth first search (DFS).
105. Trace the graph given below to reach J from node A, using BFS algorithm:



106. Trace the graph given above to reach J from node A, using DFS algorithm.
107. Trace the graph given above to find nodes reachable from node J, using DFS algorithm.