M.Sc. (C.S.)- 07

VARDHAMAN MAHAVEER OPEN UNIVERSITY, KOTA

Index

Data Structures and Algorithms

Unit Number	Unit Name	Page Number
Unit – I	INTRODUCTION TO DATA STRUCTURES	1-18
UNIT–II	ARRAYS	19-40
UNIT-III	LINK LIST ✓	41-86
UNIT-IV	STACK DATA STRUCTURE ✓	87-106
UNIT-V	QUEUE DATA STRUCTURE ✓	107-129
UNIT-VI	TREE DATA STRUCTURE ✓	130-157
UNIT-VII	ADVANCED TREE ✓	158-191
UNIT-VIII	GRAPH THEORY FUNDAMENTALS ✓	192-229
UNIT-IX	GRAPH THEORY ALGORITHMS ✓	230-243
UNIT-X	GRAPH THEORY APPLICATIONS ✓	244-272
UNIT-XI	SORTING ALGORITHMS ✓	273-297
UNIT-XII	ALGORITHM DESIGN TECHNIQUES	298-313
UNIT-XIII	DYNAMIC PROGRAMMING	314-336
UNIT-XIV	PROBLEM CLASSES	337-354

UNIT – I

INTRODUCTION TO DATA STRUCTURES

STRUCTURE OF THE UNIT

1.0 Objective
1.1 Introduction
1.1.1 Data & Information
1.2 Definition & Need of Data Structures
1.2.1 Definition of Data Structure
1.2.2 Need of Data Structure
1.3 Classification of Data types
1.3.1 Primitive Data types
1.3.2 Composite Data types
1.3.3 Abstract Data types
1.4 Implementation of Data types
1.4.1 Arrays
1.4.2 Address Calculation in 1-D Array
1.5 Classification of Data Structures
1.5.1 Simple Data Strcucture
1.5.2 Compound Data Structure
1.5.3 Linear Data Structure
1.5.4 Non Linear Data Structure
1.6 Algorithm writing and Convention
1.6.1 Writing an algorithm
1.6.2 Algorithm Conventions
1.6.3 Characteristics of algorithm
1.7 Analysis & Efficiency of Algorithm
1.7.1 Time and space complexity of algorithms
1.7.2 Algorithm complexity and order notations
1.8 Summary
1.9 Glossary
1.10 Further Readings
1.11 Answers to the Self learning exercises
1.12 Unit -End Questions

1.0 Objectives :

After reading this unit you will be able to understand the following concepts:

- Definition & Need of Data Structures
- Classification of Data types
- Implementation of data types
- Classification of Data Structure
- Various Operations on Data Structure
- Algorithm complexity and order notations
- Time and space complexities

1.1 Introduction

A computer is a machine that manipulates information. The study of Computer Science includes the study of how information is organized in a Computer, how it can be manipulated and how it can be utilised. Thus, it is important to understand the concept of information organization and manipulation.

Computer Science can be defined as the study of the data, its representation and transformation by a digital computer machine.

1.1.1 Data & Information:

We know that the data are simply values or sets of values. A data item refers to a single unit of values. Data items that are divided into sub items are called group items; those which are not , called elementary items.

Raw data is of little value in its present form unless it is organized into a meaningful format. If we organise or process data so that it reflects some meaning then this meaningful or processed data is called Information.

For example:

If we say '12 pens' of course it is data but it carries no useful meaning. But if we add a little and say 'I have 12 pens' then this statement is meaningful and is called information.

1.2 Definition & Need of Data Structure

1.2.1 Definition of Data Structure:

Data may be organized in many different ways; the logical or mathematical model of a particular organization of data is called 'Data Structure'. The choice of a particular data model depends on two considerations:-

- It must be rich enough in structure to reflect the actual relationships of the data in the real world.

- The structure should be simple enough that one effectively processes the data when necessary.

A data structure is defined as –

"A **data structure** is named group of data of different data types which can be processed as a single unit. A data structure has well-defined operations, behaviour and properties".

Arrays & Records are the examples of data structures.

1.2.2 Need of Data Structures:

Computer systems need data structures to solve complex mathematical operations. Also, once the type of problem that must be solved is determined, the appropriate data structure can be applied using already existing data types.

While designing data structures, one must determine the logical picture of the data in a particular program, choose the representation of the data, and develop the operations that will be applied on.

We will consider data structures from three different perspectives:-

1. Application (or user) Level: A way of modeling real-life data in a specific context.
2. Abstract (or logical) Level: An abstract collection of elements and its corresponding set of accessing operations.
3. Implementation Level: A specific representation of the structure and its accessing operations in a programming language.

1.3 Classification of Data types

A data type may be classified into Primitive Data Types, Non-Primitive (or Composite) Data Types and Abstract Data Types. Here we will discuss one by one in detail.

1.3.1 Primitive Data Types :

Primitive data types are those data types, which are not composed of other data types. Primitive data types are basic data types of any language. In most computers these are native to the machine's hardware.

Some Primitive data types are:

- Integer
- Character
- Real Number
- Logical
- Pointers

1.3.1.1 Integer Primitive Data types :

A quantity representing objects that are discrete in nature can be represented by an integer.

For examples: 2,0,-56,89998 etc. are integers

For examples: 34.89, -3.98, abc, uii*^lkd889 are not integers

1.3.1.2 Character Primitive Data types :

Information is not always interpreted numerically. Items such as names, job, address etc. must also be represented in some fashion within a computer.

Character is a literal expression of some element selected from an alphabet. A wide variety of character sets (or alphabets) are handled by the most popular computers. Two of the largest and most widely used character sets are represented by EBDIC and ASCII.

For examples: 'A', 'b', '/', '*', '5' etc. are characters

For examples: 23,45.5, "AJHJHA", 7&^&* are not characters

1.3.1.3 Real Number Primitive Data types :

The usual method used by computers to represent real numbers is floating-point notation. There are many varieties of floating point notation and each has individual characteristics. The key concept is that a real number is represented by a number, called a mantissa, times a base raised to an integer power called an exponent. The base is usually fixed, and the mantissa and exponent vary to represent different real numbers.

For example:

If the base is fixed at 10, the number485.43 can be represented as48543 10^{-2}.

1.3.1.4 Logical Primitive Data types :

A logical data item is a primitive data type that can assume the values of either TRUE or FALSE.

1.3.1.5 Pointers Primitive Data types :

A special data type pointer data type has been introduced by some programming languages. A pointer is a reference to a data structure. As pointer is a single fixed-size data item, it provides a homogeneous method of referencing any data structure, regardless of the structure's type or complexity. In some instance the pointers permit faster insertion and deletion of elements to and from a data structure.

The pointer data types enables storage of pointers i.e. addresses of some other data locations. Pointers are of great use in some popular and efficient data structures like linked lists, stacks, queues and trees etc.

For Examples:

```
Int x;    here x is an integer type while p is a pointer to integer type.
Int *p;
P=&x; Pointer p points to x.
```

1.3.2 Non- Primitive (Composite) Data Types :

Non-Primitives data types are those data types which are composed of primitive data types. Normally, these are defined by the users that are why; theses are some times called user-defined data types. Examples of non-primitive data types in C++ are: Array, structure, class, enumeration, union etc.

1.3.3 Abstract Data Types :

A tool that permits a designer to consider a component at an abstract level, without worrying about the details of its implementation is called **Abstraction.** Treating the data as object with some predefined operations that can be performed on those objects is called **Data Abstraction.**

Data abstraction when supported as a data types in a language is called Abstract Data Type (ADT). ADT is a useful tool for specifying the logical properties of a data type.

The term "Abstract Data Type" refers to the basic mathematical concept that defines the data type.

Example#1: Integer is an abstraction of a numerical quantity which is a whole number and can be positive or negative.

Example#2: Point is an abstraction of a two dimensional figure having no length and breadth.

1.3.3.1 Atomic Type :

It is a value (constant or variable), which is treated as single entity only and cannot be subdivided. Integers, real, characters type of data cannot be broken further into any simpler data types, therefore, these are atomic type.

Whereas, name of a person is not an atomic type data, because it can be broken into first name and last name. Also, first and last names can be further broken into character type.

1.3.3.2 Structured Type :

It is a set of values, which has two ingredients:

i) It is made up of COMPONENT elements.

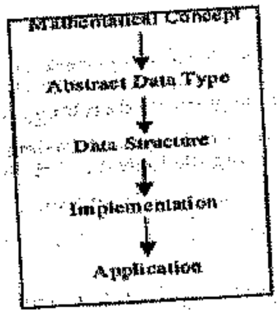
ii) There is a structure, i.e. a set of rules for putting the components together.

Example#1:
Address is a structured type data: it has components–house No., Street No., City etc.

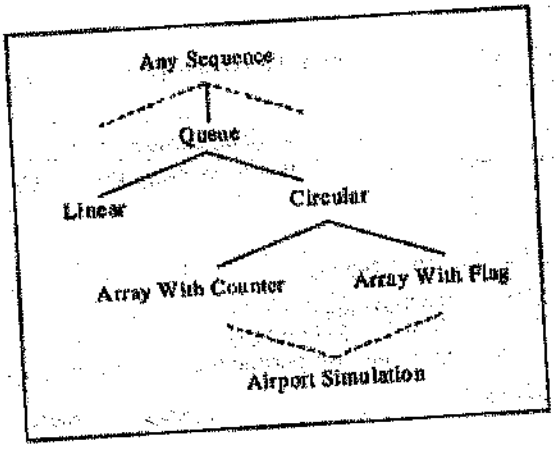
Example#2:
Date of birth is a structured data: it has day, moth and year data items.

1.3.3.3 Refinement Stages :

Stages from any mathematical concept to its implementation as application are shown in the figure as follows:-



EXAMPLE #1:



1.4 Implementation of Data Types

1.4.1 Arrays :

An array is a set of similar values stored under one tag being referred to as a subscripted variable. Different values are stored accordingly into different cells being referenced as a cell.

The concept of arrays is similar to that of any other high-level programming language such as BASIC, COBOL etc.

There are following types of arrays:-

- One or Single Dimensional Array
- Two or Double Dimensional Array
- Multi Dimensional Array

One-Dimensional Array :

The single or one dimensional array comprises of storage of values in the form of rows. The declaration of a single dimensional array has the following format:

type array-name[n];

Where type depicts the type or the nature of the array being defined, array name is the name of the subscript variable and n stands for the subscript value or the size (number of elements).

For example: int age [20]; presents the declaration/definition of the array age of the integer type and of size 20 memory locations, each of which is of integer type accordingly.

It is remember that, in C++, the first array element has the subscript 0.

The array element is accessed directly as per the following format:

cout<<age[5];

where the pointed value is that of the 6th cell of the array age.

1.4.2 Address Calculation of a Location in One Dimensional Array:

One dimensional array is a linear data structure whose elements are allocated contiguous memory locations. The array elements are stored in addresses which form a sequence.

The address of the first array element is called the **Base Address** of the array.

The address of any element can be calculated, provided the following three things are known:

(i) The Length of the array (L)

(ii) The base address (B)

(iii) The size (in bytes), of each element in the array (S).

For one-dimensional array, the address calculations may be performed by using the Relation given below:

$$Ai = B + S * (I - L)$$

Where, Ai = Address of the ith element of the array (to be computed)

B = Base Address of the array,

S = Size of each array element (in bytes),

I = Subscript of the particular element (whose address is to be computed),

L = Lower bound of the array (or minimum subscript of the array)

(The array size for all the elements same but may differ on the nature of the array, i.e. it may be 2 bytes for integer, 4 bytes for floating point etc. accordingly. by default)

For example:

Find the address of the 7th element in a floating point array srl [20], whose base address is 1592.

Solution:

Assuming the array elements to be

srl[0], srl [1], srl [2] , ... srl[19], The subscripted 7th elements will be srl [6].

I = 6 and L=0

Base Address (B) = 1592 (given)

Size of floating point array = 4 bytes (Standard)

Therefore, the address of the 7th element = B +S (I-L)

= 1592 + 4 (6-0)

= 1616.

1.5 Classification of Data Structures

We know that the Data Structures play an important role in a computer system. They combine various different data types in groups and allow user to process them as a single unit. A data structure can be classified into following categories:-

1.5.1 Simple Data Structures :

These data structures are normally built from primitive data types like Integers, Real, Characters, Boolean. Following data structures can be termed as simple data structures:

- **Array**
- **Structure**

1.5.2 Compound Data Structures:

Simple data structures can be combined in various ways to form more complex structures called compound data structures.

Compound data structures are also classified into following two categories:-

1.5.3 Linear Data Structures:

These data structures are single level data structures. A data structure is said to be linear if its elements form a sequence. Following are the examples of linear data structures:

- **Stack**
- **Queue**
- **Link List**

1.5.4 Non-linear Data Structures:

These data structures are multilevel data structures. A data structure is said to be non-linear if its data items are not arranged in a linear sequence. Graph, Tree are the examples of a non-linear data structure.

Fellowing figure shows the classification of all different data structures:-

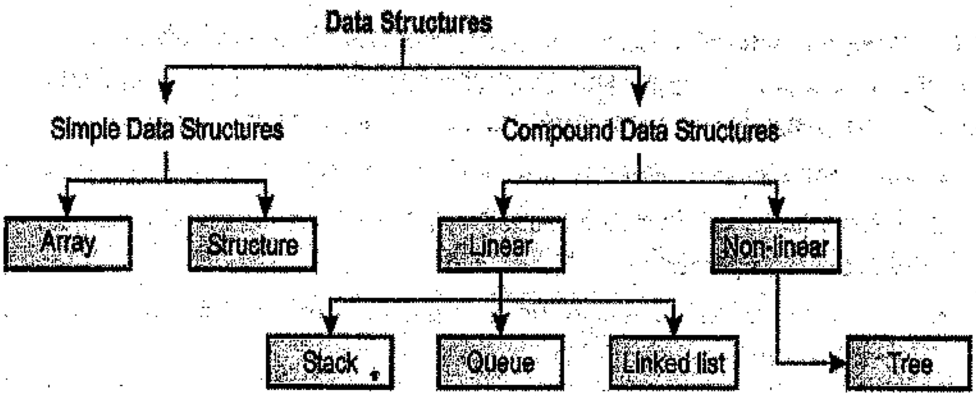


Figure 1.1

1.6 Algorithm writing and Convention

An Introduction to Algorithm :

An algorithm is a well-defined list of steps for solving a particular problem. In other words an algorithm is a finite list of well-defined instructions for solving a particular problem. These instructions, when executed in the specified order, solve a problem, which the algorithm intends to solve. The algorithm must be expressed in a language that is understood by the problem solver.

You have learnt, in your school days, many such algorithms. Like Euclid's algorithm, to compute the H.C.F of two given positive integers.

According to this algorithm:-

"Divide the greater of the two given numbers by the other one. In next step divisor of the previous step by the remainder. Repeat this until the remainder becomes 0. The divisor at this step is the required H.C.F".

1.6.1 Writing an algorithm :

An algorithm has three execution – sequences:

1. Sequence Logic or Sequential Logic
2. Selection or Conditional logic
3. Iteration or Repetitive Logic

In **"Sequence Logic"** unless instructions are given to the contrary, the modules are executed in the obvious sequence. The sequence may be presented explicitly, by means of numbered steps, or implicitly, by the order in which the modules are written.

"Selection Logic" employs a number of conditions, which lead to a selection of one out of several alternative modules.

"Iteration Logic" unless refers to either of the two types of structures involving loops. Each type begins with a Repeat statement and is followed by a module, called the body of the loop.

1.6.2 Algorithm Conventions :

We will use the following conventions in writing an algorithm :

1. An algorithm begins with a START instruction.
2. It ends with a STOP instruction.
3. One instruction is written on one line.
4. Each line is numbered sequentially for identification.
5. The instructions used are well understood by the problem–solver who will use it.
6. A container of a value (known as variable) is represented by any word that does not have language specific meaning e.g. A, B, C etc.
7. A 6 means: store 6 in the variable named A.
8. AA+1means: add 1 to the value contained in the variable A and store the result back into it. Similarly AB*C would mean multiply values in B and C and store the result into A.
9. Standard arithmetic and logical operators may be employed with their respective meaning as follows:

Arithmetic Operator		**Logical Operator**	
Add	+	Equal to	=
Subtract	-	Not equal to	<>
Multiply	*	Less than	<
Divide	/	Greater than	>

Remainder %	Greater than or equal to >=

Note: - Assignment Operatormeans store the right hand value into left hand variable.

10. Input-output instructions: Simple verbs like PRINT is used to output value(s) to the user and READ for reading input value(s) from the user.

11. Conditional instruction: A condition is an expression that evaluates to either TRUE or FALSE. For example: A>5 is a condition expression. An If instruction is written as – IF(<condition>) then <action>

    IF (A>4) THEN 5 ELSE PRINT 10

12. Loop instruction :

    - A GOTO instruction takes the execution sequence control to the specified instruction No.

      GOTO 200

    - A Do-While instruction executes a given set of instructions repeatedly as log as the given condition is true. Execution stops when the condition becomes false. i.e.

      DO(STEP 2 TO 10) WHILE(A<5)

It means, execute step 2, step 3,….step10. Then check the condition (A<5). If the condition is true then again execute step 2, step 3,……step10. Check the condition again. Repeat this process. Execution stops when the condition (A<5) becomes false. The instructions enclosed within the loops are executed once, at least.

- A WHILE –DO instruction executes a given set of instructions repeatedly as log as the given condition is true. Execution stops when the condition becomes false. i.e.

  WHILE(A<5) DO (STEP 2 TO 10)

It means, check the condition (A<5). If the condition is true then execute step 2, step 3,……step10. Again check the condition. If the condition is true then again execute step 2, step 3,….step10. Check the condition again. Repeat this process. Stops when the condition (A<5) becomes false.

- Loop instruction: REPEAT – UNTIL instruction executes a given set of instructions repeatedly until the given condition becomes false.

  REPEAT (STEP 2 TO 10) UNTIL(A<5)

It means, execute step 2, step 3,……step10. Then check the condition (A<5). If the condition is false then again execute step 2, step 3,……step10. Check the condition again. Repeat this process. Execution stops when the condition (A<5) becomes true.

Example:

An Algorithm to print all even numbers from 2 to 16 on the screen using selection and iteration.

```
STEP 1.      START
STEP 2.      A1
STEP 3.      IF (A%2=0) THEN
             PRINT A
STEP 4.      AA+1
STEP 5.      IF(A<=16) THEN GOTO 3
STEP 6.      STOP
```

1.6.3 Characteristics of Algorithm :

Algorithm must satisfy the following criteria:-

Input: These are zero or more quantities which are externally supplied.

Output: At least one quantity is produced.

Definiteness: Each instruction must be clear and unambiguous i.e. having one and only meaning.

Finiteness: If we trace out the instructions of an algorithm, then for all cases the algorithm must terminate after a finite number of steps. A program does not satisfy this condition necessarily.

Example: Operating System.

Effectiveness: Every instruction must be sufficiently basic and also feasible as far as execution is concerned.

1.7 Analysis & Efficiency of Algorithm

The analysis of algorithms is a major task in Computer Science. In order to compare algorithms, we must have some criteria to measure the efficiency of our algorithms.

The correctness of an algorithm in solving a problem is of paramount importance. A algorithm giving incorrect solution is no better than no algorithm at all.

Besides, an algorithm may solve a problem but the method may not be cost effective. The cost that incurs in executing an algorithm is in terms of how much space and how much time is required in its execution. Each variable requires space and each instruction takes some time to execute. So that, Analysis of algorithms for their correctness and space and time requirements is a major task in algorithm design.

1.7.1 Time and space complexity of Algorithms:

In order to compare correct algorithm thes two bases of space and time are taken into consideration.

To determine the execution time the following information is required :-

- Time taken by the execution of the algorithm to read one instruction.
- Time taken to understand and interpret the instruction.
- Time taken to execute the instruction.

The actual time is not easy to determine because each executor of the algorithm takes its own time. The time taken to execute an instruction also depends on instructions themselves. Some instructions take more time to execute than others. Another approach called Frequency Count Method also exists. In this method numbers of operations are counted. The actual time will be proportional to this count and can be computed by suitably multiplying by some factor. The higher order of frequency count variable in the total time expression is known as Order of the complexity denoted by O (). Less the order of complexity, more efficient the algorithm is.

The space requirement can be calculated by maximum number of variables used by the algorithm in some arbitrary space unit for the purpose of comparison.

These measurements may be different in different situations, but in case, a computer executes the algorithm, these values may be found out more accurately.

Here, we will consider the following example:

Consider the algorithm to print character "A" N square times.

```
1.    START
2.    I1
```

```
3.    IF(I>N) THEN GOTO 11
4.    J1
5.    IF(J>N) THEN GOTO 9
6.    PRINT "A"
7.    JJ+1
8.    GOTO 5
9.    I I+1
10.   GOTO 3
11.   STOP
```

Let us count the number of times operations are executed. Assume that START takes 1 unit of time, assignment() takes one unit of time, comparisons(>=) takes two units of time, arithmetic operation (+) takes one unit of time, GOTO and STOP takes one unit of time each.

1.7.2 Algorithm complexity and order notations :

Let us compute the frequency count of above given algorithm:

```
1.    START                        : 1 unit
2.    I1                           : 1 unit
3.    IF(I>N) THEN GOTO 11      : (2+1=3) units
4.    J1                           : (1*N) units
5.    IF(J>N) THEN GOTO 9          : (2+1)*N units
6.    XX+1                  : (N*N)*1 units
7.    JJ+1                         : (N*N)*1 units
8.    GOTO 5                       : (1*N) units
9.    I I+1                        : (1*N) units
10.   GOTO 3                       : (1*N) units
11.   STOP                         : 1 unit
```

Total units of time taken= 1+1+3+N+3*N+N*N+N*N+N+N+N+1

= $5+7*N+2*N^2$

The order of complexity = highest order of the frequency count variable

= 2

= $O(N^2)$

Also, we will consider the following examples:

Example#1:

Consider a program segment:

```
for(i=1; i<n; i++)
        for(j=1; j<n; j++)
        x++;
```

Frequency count = n*n

= n^2

The order of complexity = highest order of the frequency count variable

The order of complexity = 2

Example#2:

Consider a program segment:

```
        for(i=1; i<n; i++)
            for(j=1; j<n; j++)
                for(k=1; k<n; k++)
                x++;
```

Frequency Count for Ist for loop	=	n-1
Frequency Count for IInd for loop	=	(n-1) (n-1)
Frequency Count for IIIrd for loop	=	(n-1)(n-1)(n-1)
Total Frequency Count for Instruction x++;		
	=	(n-1)(n-1)(n-1)
	=	n^3-3n^2+3n-1. Ans.
The Order of complexity	=	3

'O' Notation :

The order of complexity of an algorithm may also be expressed in terms of 'O' notation.

If f(n) and g(n) are functions defined over '+ve' integers then

f(n) = o(g(n))

{Read f(n) is order of g(n)} means that these exists a constant c such that,

| f(n) | c | g(n) | for all n

Where number n is a sufficiently large '+ve' integer.

Example :

Let		f(n) = 3n-5,	$g(n) = n^2-n$
For	n = 1 :	f(n) =- 2,	g(n) = 0
	n = 2 :	f(n) = 1 g(n) = 2	
	n = 3 :	f(n) = 4 g(n) = 6	
	n = 4	f(n) = 7 g(n) = 12	

We can see that for n=2,3 and 4; c=1, 1.5 and 12/7 respectively.

Thus,

| f(n) | c | g(n) |

Self Learning Exercises

Q1. State whetherTtrue or False :

(i) Raw data are facts relating to some event.

(ii) Data structure contains a set of data of different types.

(iii) An array contains similar elements.

(iv) All data structures are arrays.

(v) Bubble sort is faster than selection sort.

(vi) A data item refers to a single unit of values.

(vii) Raw data is of little value in its present form unless it is organized into a meaningful format.

(viii) Character is not one the Primitive data type.

(ix) in "Sequence Logic" unless instructions are given to the contrary, the modules are not executed in the obvious sequence.

(x) "Selection Logic" employs a number of conditions, which lead to a selection of one out of several alternative modules.

Q2. Fill in the blanks :

(a) ________________ are simply values or sets of values.

(b) Meaningful or processed data is called ________________.

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

(d) Adding a new record to the structure is known as ________________.

(e) A ________________ is a reference to a data structure.

Q3. Define Data structures.

Q4. List four major operations on linear data structure?

Q5. What is meant by base address of an array?

Q6. Find the address of ary [5] and ary [-5] which are array elements of a char array ary [-10...10], with base address of 792.

Q7. Write a simple C++ program to find the name of a person from a list of N names which is sorted alphabetically using binary search technique.

Q8. Suppose A, B, C are arrays of integers of sizes m, n and m+n respectively. The number in array A appears in ascending order. In array B, the number appears in descending order. Write an algorithm to produce a third array C, containing all the data arrays A and B in ascending order using Sorting while merging technique.

Q9. What are the preconditions for Binary search to be performed on a one dimensional array?

Q10. Which of the following sorting: selection sort, bubble sort and insertion sort is more efficient.

1.8 Summary :

- Data may be organized in many different ways; the logical or mathematical model of a particular organization of data is called "Data Structure".
- Data Structure which displays the relationship of adjacency between elements is said to be **"Linear"**.
- It must be rich enough in structure to reflect the actual relationships of the data in the real world.
- Length finding, traversing from left to Right, Retrieval of any element, storing any element, deleting any element are the main operations which can be performed on any linear Data Structure.
- Arrays are one of the Linear Data Structures.
- Single Dimensional as well as Multidimensional Arrays are represented in memory as one dimension array.

1.9 Glossary

Array A named list of a finite number of similar data elements.

Data item A single unit of values of certain type.

Data Structure Named group of data of any one data type.

Data type Named group of data with similar characteristics and behaviour.

Data Raw facts relating to some event.

Linear Data Structure Single-level data structure representing linear relationship among data.

Merging Combining elements of two data structures to form a new data structure.

Non-linear Data Structures Multi-level data structures representing hierarchical relationship among data.

Non-primitive data types Data types which are composed of primitive data types.

Primitive data types Data types which are not composed of other data types. Sometimes also called standard data types.

Simple Data Structures Data structures normally built from primitive data types.

Sorting Arranging elopements of a data structure in some order (ascending or descending)

Traversal Processing each and every element of a data structure.

1.10 Further Readings

1. Data structures, Algorithms and Applications in C++ by S.Sahni, University press (India) pvt ltd / Orient Longman pvt.ltd., 2nd edition
2. Data Structures and Algorithm Analysis in C++ by Mark Allen Weiss, Pearson Education, Second Edition
3. Data structures and Algorithms in C++ by Michael T.Goodrich, R.Tamassia and D Mount, Wiley Student Edition, John Wiley and Sons
4. Data structures using C and C++ by Langsam, Augenstein and Tanenbaum, PHI/Pearson Education.
5. Data Structures and Algorithms in C++ by Adam Drozdek, Vikas Publishing House / Thomson International Student Edition., Second Edition

1.11 Answer to Self Learning Exercises

ANS 1. True or False :

(i) True
(ii) False
(iii) True
(iv) False
(v) False
(vi) True
(vii) True
(viii) False
(ix) False
(x) True

ANS 2. Fill in the blanks:

(a) Data
(b) Information
(c) Data Structures
(d) Inserting
(e) Pointer

Ans 3. A data structure is a named group of data of different data types which can be processed as a single unit.

Ans 4. Four major operations performed on linear data structures are :
(i) Searching (ii) Sorting (iii) Traversing (iv) Inserting

Ans 5. The starting address of the very first element of an array is known as base address.

Ans 6. Solution:

The array elements are

ary [-10], ary [-9].....ary [9], ary [10]

Base address (B) = 792 (given)
Size of an element (S) = 1 (standard for char)
Lower bound of the array (L) = -10 and I=5
There fore, the address of the element

ary [5] = B + S (I-L)
= 792+1(5-(-10))
= 807

and the address of ary [-5] = 792 +1(-5-(-10)
= 797

Ans 7. Solution:

```
// Finding name of a person from a list of N Names.
# include < iostream.h >
# include < string.h >
# include < process.h >
void main ()
{
        void arrange (int a, char b [50] [200]) ;
        int i, N ;
        char names [50] [200], name1 [50] ;
        cout << ''Input the no. of names:" ;
        cin >> N;
        cout << ''input the names one by one : \n" ;
        for (i = 0 ; i < N ; i ++)
        {
                cin >> names [i] ;
        }
        arrange (n, names);
        cout << ''Input the name to be searched : '' ;
        cin >> name1 ;
        cout << "\n";
        int mid, pos, min, max, c =0;
        min = 0, max = N, pos = -1, mid = 0 ;
        while ((c! =N) && (pos = = -1)
        {
        mid = (min + max ) /2 ;
        if (strcmp (names [mid] , name1) = = 0 )
        pos = mid ;
        else
        if (strcmp ( names [mid], name) < 0)
        min = mid +1 ;
        c = c+1;
                }
                if (pos = = -1)
                cout<< "Search Unsuccessful " ;
                else
                cout<< "The position" << pos + 1 << "\n" ;
                return 0;
                }
                void arrange (int a, char b [50] [200])
                {

        char temp [50] ;
                for (int d =0; d < a; d++)
                {
                for ( int u =d; u < a; u++)
                {
                        if (strcmp (b [d]), b[u]) > 0)
                        {
                                strcpy (temp, b [d] ) ;
                                strcpy (b [d], b [u]) ;
                                strcpy (b [u], temp) ;
                                  }
                                }
                }
        }
```

Ans 8. Solution :

```
        Count A = 0 ;
        count B  = n -1 ;
        countC  = 0;
        while (count A < m and countB < 0)
        do
        {
                if (A [count A] < = B [countB])
                {
                        C [countC]       = A [countA] ;
                        countA = count A + 1;
                        countC = count C +1 ;
                        }
                else
                {
                        C [countC] = B [countB] ;
                        countB = countB-1 ;
                        countC = countC+1;
                        }
                        if (countA =       =M)
                        {
```

```
while (countB > =0)
do
{
C[countC) = B (countB);
countB = countB-1;
countC = count C+1;
        }
}
if (countB >O)
{
while (countA < M)
do
{
C (countC) = A  [countA];
counta = countA +1 ;
countC = countC+1 ;
        }
}
```

Ans 9. For binary search:

(i) the list must be sorted,

(ii) lower bound and upper bound of the list must be known.

Ans 10.

Insertion sort is generally preferred for small number of elements. The programming effort in this technique is trivial. However the sorting does depend upon the distribution of element values.

Selection sort is easy to use but performs more transfers and comparisons compared to bubble sort. Also the memory requirement of selection sort is more compared to insertion and bubble sort. With lesser memory available insertion and bubble sort proves useful.

1.12 Unit- End Questions :

Q1. Differentiate between data type and data structure.

Q2. Why we need data structures in programming?

Q3. What are the various operations which performed on data structure?

Q4. Why we consider data structures from three different perspectives?

Q5. What is meant by the term base address?

Q6. Define the following terms:

a) Linear data structure

b) Non-linear data structure

c) Abstract data types

Q7. What are the different criteria to satisfy the algorithm?

Q8. Explain the following :

a) Analysis of algorithm

b) Time & Space Complexity

c) 'O' notation with suitable example

Q9. State whether True or False :

1. Accessing each record exactly once so that certain items in the record may be processed is traversing.

2. Searching is finding the location of the record with a given key value, or finding the locations of all records, which satisfy one or more conditions.
3. Character is a literal expression of some element selected from an alphabet.
4. Float is also one of the primary data type.

Q10. Fill in the blanks :

1. A logical data item is a primitive data type that can assume the values of either ___________ or ______________.
2. ________________ is a literal expression of some element selected from an alphabet.
3. Two of the largest and most widely used character set are represented by ______________ and ______________.
4. Some primitive data types are ___________, ___________, ___________, and ___________.
5. A well-defined list of steps for solving a particular problem is called __________.

_____***_____

UNIT – II

ARRAYS

STRUCTURE OF THE UNIT

2.0 Objectives

2.1 Introduction

2.1.1 Two-dimensional Array

2.2 Representation of Arrays in Memory

2.2.1 Memory Representation of Two–Dimensional Array

2.2.2 Memory Representation of Three–Dimensional Array

2.2.3 Memory Representation of Multi–Dimensional Array

2.3 Address Calculation using Column & Row Major Order

2.3.1 Multi–Dimensional Array

2.3.1 Row Major Storage Implementation

2.3.2 Column Major Storage Implementation

2.4 Simple Matrix Multiplication Algorithms & their complexities

2.4.1 Multiplication of two Matrices

2.4.2 Algorithm for Multiplication of Two Matrices

2.5 Sparse Matrices

2.5.1 Different Forms of Sparse Matrices

2.5.2 Array based Representation of Sparse Matrix

2.6 Summary

2.7 Glossary

2.8 Further Readings

2.9 Answers to the Self Learning Exercises

2.10 Unit–End Questions

2.0 Objectives

After reading this unit you should appreciate the following :

- Memory Representation of Two Dimensional Array
- Memory Representation of Multi-Dimensional Array
- Address Calculation using Column & Row Major Order
- Matrix multiplication Algorithm
- Representation of Sparse Matrix as an Array

2.1 Introduction

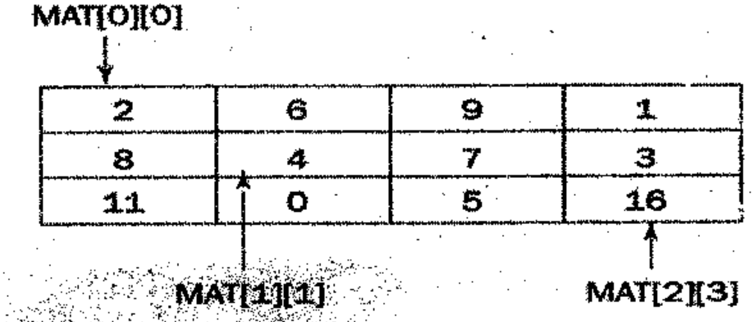
2.1.1 Two-dimensional Array :

A two-dimensional array is a collection of similar data elements where each element is referenced by the two subscripts. It is appropriate for table processing and matrix manipulation. A matrix can be represented in the form of a two-dimensional array. Hence, these arrays are also termed as *Matrix Arrays.*

$$\begin{array}{ccccc} 2 & 6 & 9 & 1 & \\ 8 & 4 & 7 & 3 & \\ 11 & 0 & 5 & 16 & 3x4 \end{array}$$

Here, we consider the matrix is shown as above. The matrix M has 3 rows and 4 columns; it contains 12 elements. The array required for storing such a matrix should be capable of storing 12 elements in a row-column style.

MAT[3][4] is a two-dimensional array with two subscripts with the first and second subscripts varying from 0 to 2 and 0 to 3, respectively. The matrix M can be stored in the two-dimensional array MAT as shown in the figure given below:



2	6	9	1
8	4	7	3
11	0	5	16

Logical representation of a two-dimensional array MAT[3][4]

There are two ways of traversing a two-dimensional array; row by row or column by column. The first way of traversing an array is known as the *row –major order and second as the column –major order.*

2.2 Representation of Arrays in Memory

2.2.1 Memory Representation of Two-Dimensional Arrays :

We can say that multi-dimensional arrays are provided as a standard data object in most of the high level languages, it is interesting to see how they are represented in memory. Memory may be regarded as one- dimensional with words numbered from 1 to m. Therefore, we are concerned with representing n dimensional array in one – dimensional memory.

A two – dimensional 'm x n' array A is a collection of m.n data elements such element is specified by a pair integers (such as j, k), called subscripts, with property that is given by:

$$1 \leq j \leq m \text{ and } 1 \leq k \leq n$$

The element of A with first subscript j and second subscript k will be denoted by $A_{j,k}$ or A[J,K]. Two-dimensional arrays are called matrices in mathematics and tables in Business Applications.

For Example :

		Columns	
1	A [1, 1]	A [1, 2]	A [1, 3]
2	A [2, 1]	A [2, 2]	A [2, 3]
3	A [3, 1]	A [3, 2]	A [3, 3]

2–Dimensional 3x3 Array A

We know that the programming language stores the array in either of the two ways:

i) *Row Major Order*

ii) *Column Major Order*

In **Row Major Order** elements of 1st Row are stored first in linear order and then come elements of next Row and so on.

In **Column Major Order** elements of 1st column are stored first in linearly and then comes elements of next column.

When the above matrix is stored in memory using **Row Major Order form.**

Then the representation for this purpose is shown in a figure 2.1 below:

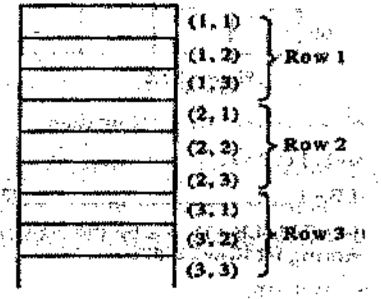


Figure : 2.1

The representation with Column Major form is shown in a figure2.2 below :

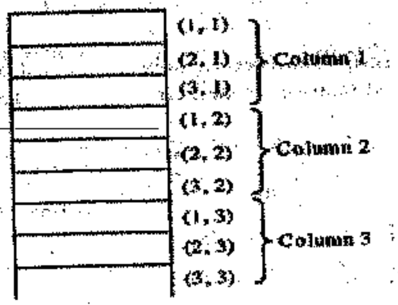


Figure : 2.2

Number of elements in any two- dimensional array can be given by:

$$\textbf{No. of elements} = (UB_1-LB_1+1)*(UB_2-LB_2+1)$$

where, UB_1 is upper bound of 1st dimension

LB_1 is lower bound of 1st dimension UB_2 and LB_2 are upper and lower bounds of 2nd dimensions.

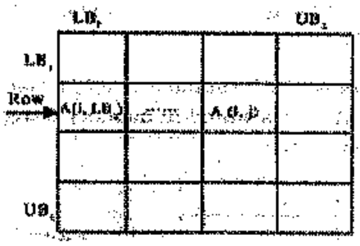


Figure : 2.3

If we want to calculate the number of elements till Ist row then

$$\textbf{No. of elements} = (UB_2-LB_2+1)* (1-1+1)$$

Or $$\textbf{No. of elements} = UB_2-LB_2+1$$

No. of elements in (j-1) Rows = (j -1) (UB_2 - LB_2+1).

If α be the size of data types of array elements then memory space required of storing i-1 rows will be:

Space Required = (UB_2-LB_2+1) (i-1)*s

If α be the address of A [$LB_{1,}$ LB_2] then Address of A (i, LB_2) will be:

Add = α + (UB_2-LB_2+1) (i-1)*S

Address of A [i, j] will be

Address of A[i, j] = α +[(UB_2-LB_2+1) (i-1) + (j-1)]*S

This is Address Scheme for Row Major Order Form.

We can write for Column Major Order:

Address of A[i, j] = α +[(UB_1-LB_1+1) (j-1) + (i-1)]*S

For example:

Consider a two dimensional matrix of figure given below. Suppose address of A_{11} is 2000 and this two-dimensional array contains elements of 4-bytes each, we want to calculate the address of A_{23}.

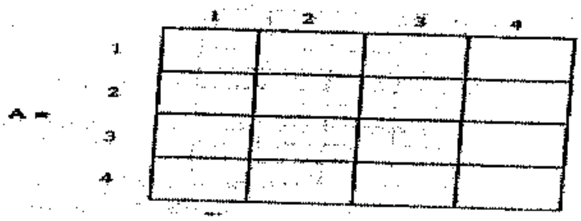


Figure 2.4

Here, we will use the following formula :

L_1 = Lower bound of the row (the1st row number)

L_2 = Lower bound of the column (the 1st column number)

Example#1:

For an array of real numbers Realarr [20] [20], find the address of Realarr [10] [12], if Realarr [1][1] is stored in location 1000. Assume each real number requires 4 bytes.

Solution : Using,

B = 1000,

S = 4 bytes,

L_1 = 1,

L_2 = 1,

in the formal

A[p][q] = B +S(n (p-L_1) +(q-L_2))

We get

A[10][12] = 1000 +4 (20(10-1)+(12-1)

= 1000 +4 (180+11)

= 1000 +764

= 1764

Example#2:

A 2-D array defined as A[4..7,-1...3] requires 2 words of storage space for each element. of the array is stored in Row - major form, calculate the address of A [6,2], given the base address as 100 (one hundred).

Solution :

Here, Base address (B)= 100,

Element size (S) = 2,

L_1 = 4,

L_2 = -1,

p = 6,

q = 2,

No of columns (n) = u_2-L_2 +1 = 3-(-1) + 1

Address of A (p,q) = B+S (n(p-L_1)+(q -L_2))

Address fo A (6,2) = 100+(2(5(6-4)+2(-(-1))

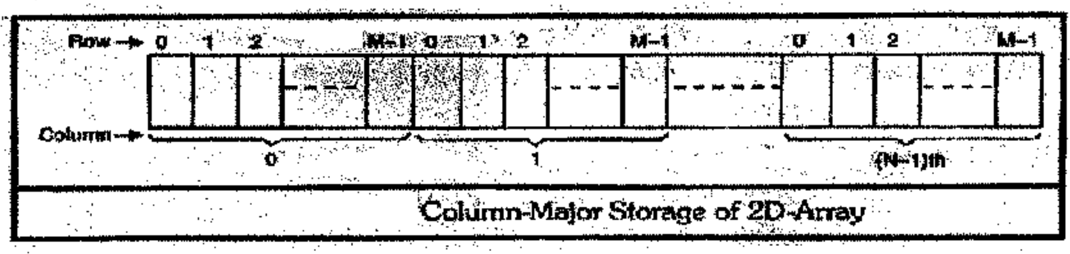
= 100+2 (10+3)

= 100 +26

= 126

2.3.1 Column Major Storage Implementation:

In this method, storages of elements are implemented column wise. In the linearization process, first the elements of the first column, next the elements of the second, third and subsequent column elements are given entries. This arrangement is shown as :



Column-Major Storage of 2D-Array

The computer keeps track only of the base address. The address of any element, say [p,q]th element, can be calculate by using the formula :

address in column major order of [p,q]th element = $B+S(p-L_1)+ m (q -L_2))$,

Where	B	=	Base address
	S	=	element size,
	L_1	=	Lower bound of the rows (1st row number)
	L_2	=	Lower bound of the columns (2nd row number)
	m	=	Number of rows.

In above example #2. the row major address of A [6,2] is 126.

But, the column major address = $100 +2 (6-4) +4(2-(-1))$

$= 100+28$

$= 128$

as number of rows (m) = $u_1 - L_1 +1 = 7-4 +1 = 4$

This shows that the addresses of row major order and row column order are not always same.

Example#1:

Each element of an array, DATA [20] [50] requires 4 bytes of storage. Base address of DATA is 2000. Determine the location of DATA [10] [10] when the array is stored as

(a) Row major

(b) Column major

Solution:

Here ,	B	=	2000,
	S	=	4 bytes,
	m	=	20,
	n	=	50,

DATA [10] [10]

(i) Row - major address

$$= 2000 + 4\,(50\,(10-1) + (10-1))$$

$$= 2000 + 4\,(50 \times 9 + 9)$$

$$= 2000 + 1836$$

$$= 3836$$

(ii) Column major address

$$= 2000 + (4\,(10-1) + 20\,(10-1))$$

$$= 2000 + 4\,(189)$$

$$= 2000 + 756$$

$$= 2756$$

Example#2:

Each element of an array A [-15 ...20, 20...45] require one byte of storage. If the array is stored in column major order beginning location 1000, determine the location of A [0, 40].

Solution:

Here, Base address (B) = 10000

Element size (S) = 1 (byte)

No of rows $(m) = u_1 - L_1 + 1$

$$= 20 - (-15) + 1$$

$$= 36$$

Therefore, the address of A [0, 40] in column major order

$$= B + S((q - L_2) + m\,(p - L_1))$$

$$= 1000 + 1 \times ((0-(-15) + 36\,(42-20))$$

$$= 1000 + (15 + 720)$$

$$= 1735$$

2.4 Simple Matrix Multiplication Algorithms & their complexities

Two-dimensional arrays sometimes known as matrices are used to perform various mathematical operations on a rectangular set of elements. There are some important operations of 2-Dimensional arrays such as - traversal, addition, subtraction, multiplication and others. Here we will discuss the multiplication of two matrices and algorithm for this purpose:

2.4.1 Multiplication of two Matrices :

The multiplication of two matrices is also possible but it is not similar to the addition or subtraction of matrices. It is possible only when the number of columns of first matrix are equal to the number of rows of another matrix or number of rows of first matrix are similar to the number of columns of another matrix, i.e. when the two matrices are comparable.

***For Example:** There are two matrices A[3][3] and B[3][3] that have the following positions of elements:*

$$A = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix} \qquad B = \begin{matrix} 2 & 4 & 6 \\ 3 & 5 & 7 \\ 4 & 7 & 9 \end{matrix}$$

Then the product of these two marices will be :

$$A.B = \begin{bmatrix} 1\times2 + 2\times3 + 3\times4 & 1\times4 + 2\times5 + 3\times7 & 1\times6 + 2\times7 + 3\times9 \\ 4\times2 + 5\times3 + 6\times4 & 4\times4 + 5\times5 + 6\times7 & 4\times6 + 5\times7 + 6\times9 \\ 7\times2 + 8\times3 + 9\times4 & 7\times4 + 8\times5 + 9\times7 & 7\times6 + 8\times7 + 9\times9 \end{bmatrix}$$

$$= \begin{bmatrix} 2 + 6 + 12 & 4 + 10 + 21 & 6 + 14 + 27 \\ 8 + 15 + 24 & 16 + 25 + 42 & 24 + 35 + 54 \\ 14 + 24 + 36 & 28 + 40 + 63 & 42 + 56 + 81 \end{bmatrix}$$

$$= \begin{bmatrix} 20 & 35 & 47 \\ 47 & 83 & 113 \\ 74 & 131 & 179 \end{bmatrix}$$

2.4.2 Algorithm for Multiplication of Two Matrices:

Using multiplication by taking column of first matrix into rows of second matrix.

Suppose, therefore two matrices A[M][N] and B[N][O] and product is to go a third matrix C[M][O]

```
STEP 1:   SET I=0, J=0,K=0             //Initializing counters I,J, K with 0
STEP 2:   Repeat step 3 to 9 While(I<row)
STEP 3:   Repeat step 4 to 8 While(J<column)
STEP 4:   C[I][J] = 0                  //Initializing 0 to the elements of new matrix
STEP 5:   Repeat step 5 to 7 While(K<column)
STEP 6:   C[I][J] = C[I][J] + A[I][K]*B[K][J]
STEP 7:   K=K+1                        // end of most inner loop K
STEP 8:   J=J+1                        // end of inner loop J
STEP 9:   I=I+1                        // end of outer loop I
STEP 10:  EXIT
```

// C++ Implementation of Algorithm for Matrix Multiplication

Write a program to find the multiplication of two matrices using 2-D arrays.

```
#include< iostream.h>
#include< conio.h>
```

```
void main( )
{
clrscr();
int A [3][3], B [3][3], C[3,3];
int i,j,k;
cout<<“Values for first matrix\n:”;
for (i=0;i<3;i++)
for (j=0;j<3;j++)
cin>>A[i] [j];
cout<<“Values for second matrix\n:”;
for (i=0;i<3;i++)
for (j=0;j<3;j++)
cin>>B[i] [j];
/* multiplication of the elements of 2 matrices */
for (i=0;i<3;i++)
for (j=0;j<3;j++)
{
C[i] [j] = 0;                          //initialization to the elements of new matrix
for (k=0;k<3;k++)
C[i] [j] = C[i] [j] + A[i] [k] * B[k] [j] ;
}
cout<<“\n First matrix\n:”;            //First matrix printing
for (i=0;i<3;i++)
{
for (j=0;j<3;j++)
cout<<A [i] [j]<<“ “;
cout<<endl;
}
cout<<“\n Second matrix\n:”;    //Second matrix printing
for (i=0;i<3;i++)
{
for (j=0;j<3;j++)
cout<<B [i] [j]<<“ “;
cout<<endl;
}
cout<<“\n New matrix after multiplication \n:”;   // new matrix printing
for (i=0;i<3;i++)
{
for (j=0;j<3;j++)
cout<<C [i] [j]<<" ";
cout<<endl;
}
}
```

OUTPUT:

```
Values for first matrix:
1
2
3
4
5
6
7
8
9
Values for second matrix:
2
4
6
3
5
7
4
7
9
First matrix:
1       2       3
4       5       6
7       8       9
Second matrix:
2       4       6
3       5       7
4       7       9
```

New matrix after multiplication:

```
20    35    47
47    83    113
74    131   197
```

2.5 Sparse Matrices

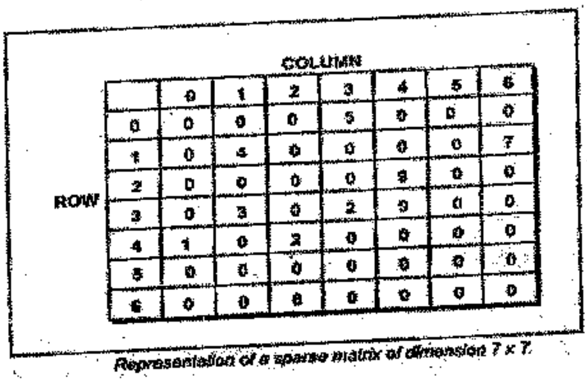
We know that the "matrix" is a mathematical term that refers to collection of numbers they are analogous. A matrix with the same number (n) of rows and columns is called Square matrix or n-square matrix.

The following section discusses efficient ways of storing certain types of matrices are known as *sparse matrices.*

Therefore we can say that – 'If a lot of elements from a matrix have a value 0 (zero) then this matrix is called Sparse matrix'.

There is no exact definition of sparse matrix, but it is a concept that can be recognized intuitively in all. If a matrix is sparse then we must consider an alternative way of representing it rather than the normal row major or column major arrangement. This is because if majority of elements of the matrix are 0, then an alternative through which we can store only the non-zero elements and keep intact the functionality of the matrix can save a lot of memory space.

The given figure is representing a sparse matrix of 7x7 dimensions.



COLUMN

ROW

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

Representation of a sparse matrix of dimension 7 x 7.

Figure : Representation of Sparse Matrix of dimension 7x7

The way of representation of non-zero elements in a sparse matrix is known as the 3-tuple forms. This way is commonly used in a sparse matrix. In this way each non-zero element is stored in a row, with the 1st and 2nd element of this row consisting the row and column in which the element is present in the original matrix. The 3rd element in this row stores the actual value of the non-zero element.

FOR EXAMPLE:

The 3-tuple representation of the sparse matrix is given by –

```
Int sparmat[10] [3] = {
                7,7,9,
                0,3,-5,
1,1,4,
1,6,7,
2,4,9,
3,1,3,
3,3,2,
4,0,11,
4,2,2,
6,2,8,
}
```

Hence, we can say the information of a 3-tuple representation of a sparse matrix can be stored using two ways –

- Arrays
- Linked List

The information about the non-zero elements is stored in these representations, whenever the number of non-zero elements can be varied in a sparse matrix. It is more convenient to use a linked list for the representation of a sparse matrix.

2.5.1 Different Forms of Sparse Matrices :

There are different forms of sparse matrices which are commonly used –

- **Diagonal Sparse Matrix:**

  It is represented as –

4	0	0	0
0	1	0	0
0	0	9	0
0	0	0	12

- **Tri-diagonal Sparse Matrix**

  It is represented as –

0	3	0	0
0	0	8	0
0	0	0	5
0	0	0	0

- **Lower Triangular Sparse Matrix**

  It is represented as –

0	0	0	0
13	0	0	0
9	2	0	0
6	0	12	0

- **Upper Triangular Sparse Matrix**

It is represented as –

$$\begin{matrix} 0 & 4 & 0 & 3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 \end{matrix}$$

2.5.2 Array based Representation of Sparse Matrix :

Here we will discuss a program that accepts the elements of a sparse matrix and creates an array containing 3-tuples of non–zero elements present in the sparse matrix.

```
// C++ demonstration of sparse matrix as an array
#include<iostream.h>
#include<conio.h>
#include<alloc.h>
#define Max1 3
#define Max2 3
struct spar
{
int *ps;
int row;
};
void initspar (struct spar*);
void create_array (struct spar*);
void display (struct spar);
int count (struct spar);
void create_tupple (struct spar*, struct spar);
void display_tupple (struct spar);
void delspar (struct spar*);
void main()
{
struct spar s1,s2;
int c;
clrscr();
initspar (&s1);
initspar (&s2);
create_array (&s1);
cout<<"\n Elements in sparse matrix";
display(s1);
c=count(s1);
```

```
cout<<“\n\n Number of non-zero elements:”<<c;
create_tupple (&s2,s1);
cout<<“\n\n Array of non-zero elements:”;
display_tupple (s2);

delspar (&s1);
delspar (&s2);

getch();
}
void initspar (struct spar*p)        //Initialization of structure's elements
{
p->ps=NULL;
}
void create_array (struct spar*p)            //dynamically creates the
                                              // matrix Max1x Max2

{
int n,i;
p->ps=(int*) malloc(Max1*Max2*sizeof(int));
for(i=0;i<Max1*Max2;i++)
{
cout<<“Enter elements no.”<<i<<“:”;
cin>>n;
*(p->ps+i)=n;
}
}
/*Display the contents of matrix*/
void display (struct spar p)
{
int i;
for(i=0;i<Max1*Max2;i++)        // traverse the entire matrix
{
if(i%Max2==0)
cout<<“\n”;
cout<<“\t”, *(p.ps+i);
}
```

```
}
int count(struct spar p)  //counts the no of nonzero elements
{
int ct=0,i;
for(i=0;i<Max1*Max2;i++)
{
if(*(p.ps+i)!=0)
ct++;
}
return ct;
}
//Creates an array that stores information about non-zero elements
void create_tupple (struct spar *p, struct spar s)
{
int r=0,c=-1,l=-1,i;
p->row=count(s)+1;
p->ps=(int*) malloc (p->row*3*sizeof(int));
*(p->ps+0)=Max1;
*(p->ps+1)=Max2;
*(p->ps+2)=p->row-1;
l=2;
for(i=0;i<Max1*Max2;i++)
{
c++;
if(((i%Max2)==0)&&(i!=0))
{
r++;
c=0;
}
if(*(s.ps+i)!=0)
{
l++;
*(p->ps+l)=r;
l++;
*(p->ps+l)=c;
l++;
*(p->ps+l)=*(s.ps+l);
        }
                }
        }
void display_tupple (struct spar p)        //displays the contents
                                                        //of 3-tupple
{ int i;
for(i=0;i<p.row*3;i++)
{
if(i%3==0)
cout<<endl;
cout<<“\t”, *(p.ps+i);
}
}
void delspar (struct spar*p) //deallocation of memory
{
if(p->ps);
}
```

Output:

```
Enter element no. 0 : 0
Enter element no. 1 : 2
Enter element no. 2 : 0
Enter element no. 3 : 9
Enter element no. 4 : 0
Enter element no. 5 : 1
Enter element no. 6 : 0
Enter element no. 7 : 0
Enter element no. 8 : -4

Elements in Sparse Matrix:
1          2             1
9          1             1
1          1             4
Number of non-zero elements : 4
Array of non-zero elements :
3          3             4
```

1	1	2
1	1	9
1	2	1
2	2	1

Self Learning Exercises

Q1. What are the two ways to achieve linearization in 2-D arrays?

Q2. Describe the algorithm for multiplication of two matrices.

Q3. What do you mean by Sparse Matrix?

Q4. Give the different forms of sparse matrix.

Q5. Write an algorithm to find the maximum and minimum values of numbers stored in two-dimensional array.

Q6. **State whether true or false:**

1. Array size is always fixed.
2. A one-dimensional array is a list of finite number n of homogeneous data elements.
3. An array is a collection of similar elements.
4. A three-dimensional array can be thought of as array of arrays.

Q7. **Fill in the blanks:**

1. A **“data structure”** which displays the relationship of adjacency between elements is side to be ____________.
2. Normally a number of s (contiguous) memory locations are sequentially allocated to the ____________.
3. The elements of the Array are stored respectively in successive ____________.
4. Elements of any Multidimensional Array can be stored in two forms ____________ and ____________.

2.6 Summary

- An array is a named collection of similar data elements stored in contiguous memory locations.
- Arrays are one of the Linear Data Structures.
- Arrays are of one-dimensional,two-dimensional and multi-dimensional.
- Single Dimensional as well as Multidimensional Arrays are represented in memory as one dimensional array.
- Elements of any Multidimensional Array can be stored in two forms Row major and column major.
- Two dimensional arrays are commonly used to represent table of data items arranged in rows and columns.
- Two dimensional arrays are implemented by allocating them a row-major or a column-major order.
- If a lot of elements from a matrix have a value 0 (zero) then this matrix is called sparse

2.7 Glossary

2-D array: It is a set of similar data elements where each element is referenced by the two subscripts.

Sparse Matrix: If a lot of elements from a matrix have a value 0 (zero) then this matrix is called sparse matrix.

Row Major Order elements of 1st Row are stored first in linear order and then come elements of next Row and so on.

In **Column Major Order** elements of 1st column are stored first in linearly and then comes elements of next column.

Memory Locations: The elements of the Array are stored respectively in successive storage place.

2.8 Further Readings

1. Data structures, Algorithms and Applications in C++ by S.Sahni, University press (India) pvt ltd / Orient Longman pvt.ltd., 2nd edition

2. Data Structures and Algorithm Analysis in C++ by Mark Allen Weiss, Pearson Education, Second Edition

3. Data structures and Algorithms in C++ by Michael T.Goodrich, R.Tamassia and D Mount, Wiley Student Edition, John Wiley and Sons

4. Data structures using C and C++ by Langsam, Augenstein and Tanenbaum, PHI/Pearson Education.

5. Data Structures and Algorithms in C++ by Adam Drozdek, Vikas Publishing House / Thomson International Student Edition., Second Edition

2.9 Answers to the Self learning exercises

ANS. 1 There are two ways of linearization of two dimensional arrays:

(i) Row Major Order (ii) Column Major Order

ANS. 2 The algorithm for multiplication of two matrices is as follows:

```
STEP 1:   SET I=0, J=0,K=0              //Initializing counters I,J K with 0
STEP 2:   Repeat step 3 to 9 While(I<row)
STEP 3:   Repeat step 4 to 8 While(J<column)
STEP 4:   C[I][J] = 0                   //Initializing 0 to the elements of row matrix
STEP 5:   Repeat step 5 to 7 While(K<column)
STEP 6:   C[I][J] = C[I][J] + A[I][K]*B[K][J]
STEP 7:   K=K+1                         // end of most inner loop K
STEP 8:   J=J+1                         // end of inner loop J
STEP 9:   I=I+1                         // end of outer loop I
STEP 10:  EXIT
```

ANS 3.

The sparse matrix is defined as –

If a lot of elements from a matrix have a value 0 (zero) then this matrix is called **sparse matrix.**

ANS 4.

The different forms of sparse matrix are given below:

1. Diagonal Matrix
2. Tri-diagonal Matrix
3. Lower Triangular Matrix
4. Upper Triangular Matrix

ANS 5.

Steps:

```
1.  Max=MAT[0][0]
2.  Min=MAT[0][0]
3.  I=0
4.  While(I<M) repeat steps 5 to 8
5.  J=0
6.  While(J<N) repeat steps 7 and 8
7.  If (Max<MAT[I] [J])
        Max=MAT[I] [J];
8.  If (Min>MAT[I] [J])
        Min=MAT[I] [J];
9.  Display Max and Min
10. STOP
```

ANS 6. State whether true or false:

1. True
2. True
3. True
4. True

ANS 7. Fill in the blanks:

1. Linear
2. Array
3. memory locations
4. Row major, column major

2.10 Unit End Questions

Q1. What are 2-dimensional arrays? Give an example.

Q2. What are the two ways in which a programming language stores the array? Explain with example.

Q3. Write a program, which gives an example of multidimensional array.

Q4. What is meant by the term Sparse Matrix? Give one example.

Q5. Describe the different forms of Sparse Matrix with suitable examples.

___***___

UNIT – III

LINK LISTS

STRUCTURE OF THE UNIT

3.0 Objectives

3.1 Introduction

3.1.1 Definition of Linked List — Logical linear data structure

3.1.2 Need for Linked Lists

3.1.3 Memory Allocation

3.1.3.1 Static Memory Allocation

3.1.3.2 Dynamic Memory Allocation

3.1.4 Advantages of Linked Lists over Arrays

3.2 Types of Linked Lists

3.3 Representation of Linked List in Memory

3.4 Implementation of Linked List

3.4.1 Creation of Linked List

3.4.2 Insertion in Linked List

3.4.3 Deletion in Linked List

3.4.4 Traversal of Linked List

3.4.5 Searching in Linked List

3.5 Doubly Linked List

3.6 Circular Linked List

3.7 Linked List Representation of Sparse Matrix

3.8 Summary

3.9 Glossary

3.10 Further Readings

3.11 Answers to the Self Learning Exercises

3.12 Unit–End Questions

Logical linear data structure

3.0 Objectives

(Learn the concept of linked list

After going through this unit you will be in a position to :

- Learn the concept of Linked lists
- Identify the types of Linked lists
- Describe the Static and Dynamic Memory Allocation
- Define Linear Linked list
- Describe Representation of Linked List

- Explain the Implementation of Linked List
- Understand the Concatenation of Linked List
- Identify Splitting of Linked List
- Explain the Reversing of Linked List
- Describe Doubly Linked Lists & Circular linked List

3.1 Introduction

Lists are a collection of data which has been arranged sequentially. Sequential data collections can be created and managed in a number of ways. One possibility is to use arrays or structures which we have covered in the previous unit. Recall that the linear relationship between the data elements of array is reflected by the physical relationship of the data in the memory (i.e. sequential allocation) not by any information contained in the data elements themselves. However, a list can also be stored as having data elements pointing to the next in the sequence (i.e. linked allocation) i.e. a linked list.

Linked lists are data structures that allow the handling of multiple elements of the same data type. They are self–referential and dynamic and therefore, unlike arrays, are not limited to a pre-defined number of contained elements. Which means that an element (member) of linked list not only contains the assigned value of the element but also a pointer connection (link) to the next element of the list. These list elements are called Nodes structures.

3.1.1 Definition of Linked List

A **Linked List** is a linear collection of data elements, called nodes pointing to the next nodes by means of pointers.

Each node divided into two parts: the first part containing the information of the element, and the second part called the link or next pointer containing the address of the next node in the list.

To understand the concept of Linked list, let us talk about this situation:-

In a city, Children-Film-Festival is going on. A class teacher decides to take her class-students to watch a movie. She plans this visit well in advance and thus, calls up the cinema-hall's manager to book tickets for 30 students of the class. The cinema-hall's manager books an entire row for the students. On the scheduled day, teacher takes her students to the cinema-hall. But she has a problem. She has to go out for some time because of an urgent work. After this, she will come back to take her students back to the school. In order to remember where her students are seated, she just notes down the row number and seat number of the first student in the row and goes out for her work. Later she comes back, goes to the noted down row number and set number, counts 30 students this seat onwards, and takes her students back to the school.

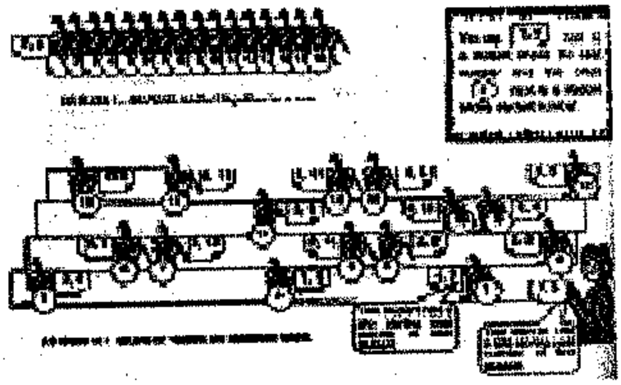
An year later, the same festival is being held in the city. This time also the teacher decides to take her thirty students to watch a movie. But this is very last day of the festival. She calls up the cinema-hall to find out whether 30 tickets available but all the seats are not in the row, In fact the seats are here and there in the hall. She takes her students to cinema hall and seats them on these scattered empty seats. This time also she has an urgent work to do and has to go out. How will she remember the student's seat numbers this time? Last time all of them were seated in single row next to one another. So that all that was required, was to remember the seat number of the firs student. Rest of the seat numbers could be easily determined as students were seated next-to-one another. What about this time?

But the teacher is very smart. She takes out some paper slips and hands over one slip each, to all the students. Then she notes down one student's row number and seat number

on a slip and keeps that with her. The she asks the student (whose seat number, the teacher has noted down) to note down the row number, seat number, of the next student on his slip, and so on. This way she makes her students carry a slip that has the seat number of a student sitting some where else.

After her work, she comes back and goes to the seat-number noted on her slip to take the student seated on this seat. The she reads the seat-number noted on student's slip, goes to the seta number to take this student. She repeats this process to take all her students with her. This situating gives you a fair idea of arrays and linked lists.

Which are represented in the figure given below:



When the students were seated in a row, it was in the form of arrays. In arrays, all the elements are stored in contiguous locations. Here in above situation, we can say the students as elements and seat-numbers as memory addresses. When the students were seated here and there, then each student was also carrying a slip that was storing the seat number of following student. This is referred as Linked List. Each element also stores the address of following element. All that is needed to process such as a list is its beginning address, which the teacher did by keeping a slip having the seat number of first student. The slip of last student does not have a seat-number as there is no student left whose seat number is to be remembered.

3.1.2 Need for Linked Lists

The simplest one of data structure i.e. an array can be used only when its number of elements along with element sizes are predetermined, since the memory is reserved before processing for this reason, arrays are called dense lists or static data structures.

Now consider a situation where exact number of elements is not known. In such case, estimates may go wrong. Also during processing i.e. either more memory is required or extra free spaces lie in the array.

Another problem associated with arrays is complexity involved in insertions and deletions of elements.

3.1.3 Memory Allocation

Each data element, stored in the memory, is given some memory. This process of giving memory is called Memory allocation. The memory can be allocated in two manners: dynamically and statically. They will be discussed here.

3.1.3.1 Static Memory Allocation

This memory allocation technique reserves fixed amount of memory before actual processing takes place, therefore, number of elements to be stored must be predetermined. Such type of memory allocation is called static memory allocation. Arrays are allocated memory using this technique only.

In other words we can say that, in static memory allocation memory is allocated at compile time. If we declare a stack through an array of 100 elements (all integers) then the statement is given by:

```
int stk[100];
```

This declaration would typically be used if 100 records are to be stored in memory. The moment we make this declaration 200 bytes are reserved in memory for storing 100 integers in it. However it may so happen that when we actually run the program we may be interested in storing only 60 integers, even in this case 200 bytes would get reserved in memory which would result in wastage of memory. There may be cases when we need to store more than 100 records, in this case array would fall short in size.

3.1.3.2 Dynamic Memory Allocation

This memory allocation technique facilitates allocation of memory during the program execution by itself, as and when required. This technique of memory allocation is called dynamic memory allocation. This facilitates release of memory, if memory is not required any more. Linked lists and Trees are allocated memory using this technique only.

We can overcome the problems faced in static memory allocation by allocating memory at run time (instead of compile time); this is called dynamic memory allocation.

It is done by using Standard Library Functions **malloc() and calloc()**.

Syntax for allocating through malloc () :

```
P=(data type*) malloc (n*size of (<datatype>));
```

For example:

```
P = (int*) malloc (n*2);
```

The expression **(int*)** is used to typecast the address being returned as the address of an integer.

The **calloc()** function is exactly similar to **malloc()** except for the fact that it needs two arguments as against the one argument required by malloc().

For example:

```
int * p
P = (int*) calloc (100,2);
```

Here 2 indicates that we want to allocate memory or storing integers, since an integer is a 2 byte entity, and 100 indicates that we want to reserve space for storing 100 integers.

Another difference between malloc() and calloc() functions is that, by default the memory allocated by malloc() contains garbage values, whereas that allocated by calloc() contains all zeros.

3.1.4 Advantages of Linked Lists over Arrays

With reference to the practical application of linked lists, there are two main advantages of linked lists over arrays:-

(a) It is not required to know the number of elements and allocation of memory for linked lists. It is done accordingly as and when required.

(b) It allows insertion and deletion of the list of elements without restructuring the list.

3.2 Types of Linked Lists

There are different kinds of linked lists, the main will be discussed here :

3.2.1 Linear Link Lists – Where each element refers to its successor. Such a list of two types:-

(a) Singly (Simple) linked lists – are also called one-way lists. Singly linked lists contain node with single pointer pointing to the next node in sequence.

(b) Doubly linked lists – are also called two-way lists. Doubly linked lists contain node with two pointers, one pointing to previous node and other to the next node in sequence.

3.2.2 Circular Linked Lists – In which the last node, instead of having a NULL pointer. Points to the node at the beginning of the list.

Linked lists can also be used to create a number of different data structures like Stacks, Queues, and Trees etc. You will learn these concepts in next units.

3.2.1 Linear Linked List :

We have seen in the previous unit how static representation of linear ordered list through Array leads to wastage of memory and in some cases overflows.

Now we don't want to assign memory to any linear list in advance instead we want to allocate memory to the elements as they are inserted in list.

This requires Dynamic Allocation of memory and it can be achieved by using **malloc() or calloc()** function.

But memory assigned to elements will not be contiguous, that is a requirement for linear ordered list, and was provided by array representation. How we could achieve this? We have to consider a logical ordered list, i.e. elements are stored in different memory locations but they are linked to each other and form a logical list as in Figure given below:

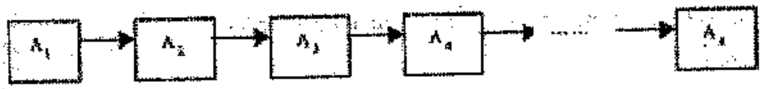
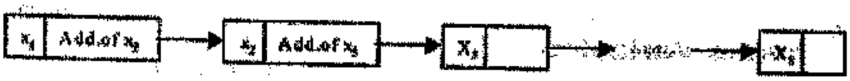


Figure: Logical List

Suppose there is a list of 8 friends, X1, X2....... X8. Each friend resides at different locations of the city. X1 knows the address of X3 and so on .... X7 has the address of X8. If one wants to go to the house of X8 and he does not know the address he will go to X2 and so on. Which is represented in the figure given below:



The concept of linked list is like a family despaired, but still bound together.

From the above discussion it is clear that Link list is a collection of elements called nodes, each of which stores two items of information:

§ An element of the list

§ A link or address of another node

Link can be provided through a pointer that indicates the location of the node containing the successor of this list element.

The NULL in the last node indicates that this is the last node in the list.

3.3 Representation of Linked List in Memory

A linked list requires two contiguous arrays to store in memory. The first array stores the data value and the second array stores the next pointer storage. And also a variable FIRST is required which is always pointing to the first node of the list. Because in order to access the list we used to know the address of the first node. Also another variable AVAIL is required

which is useful at the time of insertion in the linked list. The AVAIL is point to the next unused or free memory cell and is called free-storage or free pool.

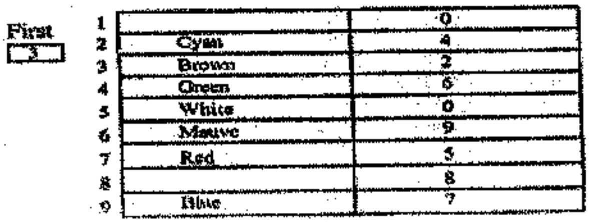
A free-storage is allocated with the help of **new** operator.

Suppose these are the values of different variables:

FIRST =3 DATA[3]="Cyan" and LINK[3]=2

DATA[2]="Brown"	LINK[2]=4
DATA[4]="Green"	LINK[4]=6
DATA[6]="Mauve"	LINK[6]=9
DATA[9]="Blue"	LINK[9]=7
DATA[7]="Red"	LINK[7]=5
DATA[5]="White"	LINK[7]=0

So that in the memory it would look like as in figure :



1		0
2	Cyan	4
3	Brown	2
4	Green	6
5	White	0
6	Mauve	9
7	Red	5
8		8
9	Blue	7

Now, if we have to insert an element in linked list, then we have to maintain a list of unused cell that is pointing to be Avail. Now if we want to insert on the above linked list, then we should know the next free location. Let us suppose the next free location is 8. So, the value of Avail=8. The following figure shows the Avail position as:

First 3

Avail 8

1		
2	Cyan	4
3	Brown	2
4	Green	6
5	White	0
6	Mauve	9
7	Red	5
8		
9	Blue	7

If we insert "Violet" it will place at the location 8 and the value of Avail becomes 1 because next free location in the above list is 1. The following figure shows as:

First 3

Avail 8

1		0
2	Cyan	4
3	Brown	2
4	Green	6
5	White	0
6	Mauve	9
7	Red	5
8	Violet	1
9	Blue	7

3.4 Implementation of Linked Lists

A linked list is a dynamic data structure. So, it should be created during run-time.

Declaration of Linked List

A linked list contains two different parts, it is declared with the help of structure. The declaration is as follows:

```
struct node
{
        data-type data;
        node * link;
}
node * list;
```

Link is a pointer of struct node type i.e. it can hold the address of variable of struct node type. Pointers permit the referencing of structures in a uniform way, regardless of the organization of the structure being referenced. Pointers are capable of representing a much more complex relationship between elements of a structure than a linear order.

The various operations performed on linked list are : creation, insertion, deletion, traversal, serach

3.4.1 Creation of Linked List

The creation of linked list is very simple. Let us write an algorithm to create a linked list.

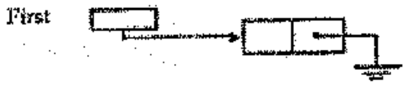
Algorithm for creating a linked list :

In this algorithm a linked list consisting of n nodes is created. Avail is the next available memory of the computer. The first node of the list is pointing by a pointer First and the last node of the list points to NULL. Each node of the list has two parts. DATA and LINK. The new operator is used for the dynamic allocation of node.

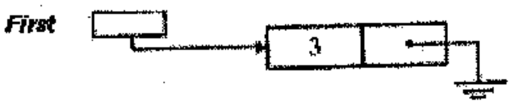
```
Step 1.     If Avail = NULL
                Print "Overflow" go to step 13
Step 2.     First=new node
Step 3.     Avail=Avail->LINK
Step 4.     First->DATA=Val
Step 5.     FIRST->LINK=NULL
Step 6.     Temp=First
Step 7.     For i=1 to n-1 repeat step 8 to 12
Step 8.     Last=new node
Step 9.     Last->DATA=Val
Step 10.    Last->DATA=NULL
Step 11.    Temp->LINK=Last
Step 12.    Temp=Last
Step 13.    END
```

In the above algorithm the First pointer is used to point to the first node, and Temp pointer is storing the current node. Let us illustrate the above algorithm diagrammatically.

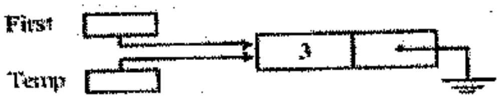
Suppose, we have to create a linked list of 3 integers. First of all the memory is allocated with the help of the new as in the step 2 of above algorithm. The following figure shows the first node as:



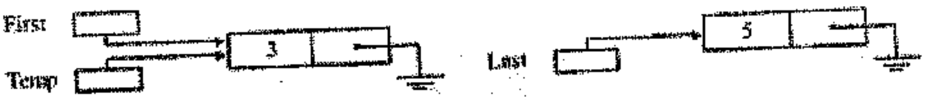
Let us assume the data value of first node is 3. The inserted value is shown in the figure given below:



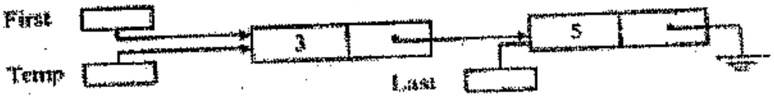
Now, store the address of first node in another variable called Temp. That is shown in the figure given below:



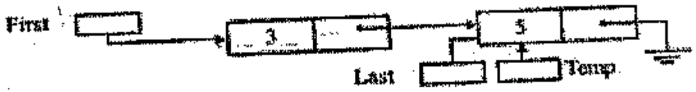
Then it allocate the memory for another variable as in the step8 of above algorithm called Last. Let us assume the data value of Last is 5, which is shown in the figure below



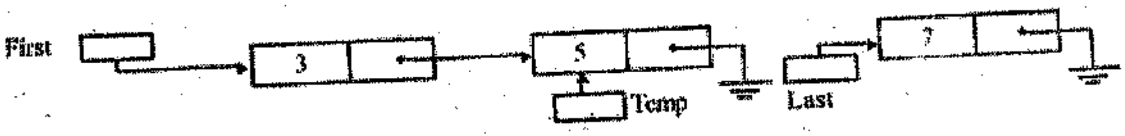
Now, two nodes are created . We have to create the link between the two nodes. Therefore, point to the LINK part of Temp to Last as in step11. That is shown in the figure below:



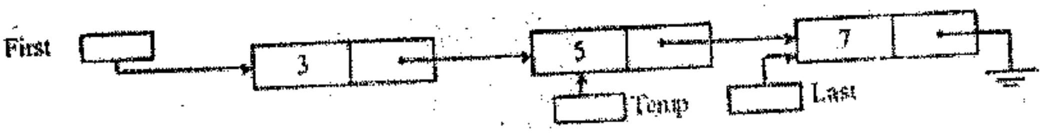
A link list of 2 nodes is created. Now, we store the address of Last in Temp as in step12. Which is shown in the figure below:



Now, allocate the memory of another node and name it as Last and suppose its Data value is 7. The shifted location is shown in the figure below:



Again point the link part of Temp to Last and store the address of Last in Temp. Which is shown in the figure below:



So that here, a linked list of three nodes is created in which the first node is pointed by First and the Last node is pointed to NULL.

3.4.2 Insertion in Link List

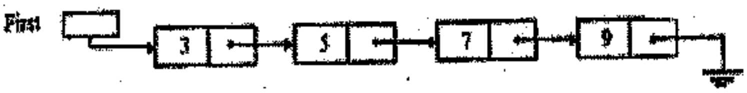
We know that insertion and deletion are very easy in linked list. Insertion in the linked list is done in three ways:

- **Insertion of a node at the beginning**
- **Insertion of a node after a specified node(in between)**
- **Insertion of a node at the end**

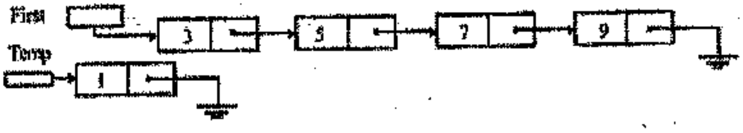
Here we will discuss all of three one by one.

3.4.2.1 Insertion of a Node at the Beginning

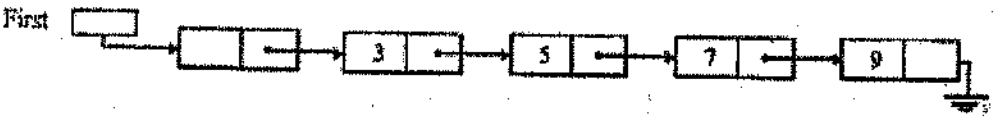
To insert a node in the beginning of the linked list is very simple. First of all create a new node and point its Link part to the First node of the List. Now, new node will become the First node of the list. Let us consider a linked list is shown in figure below:



Suppose, we have to insert a node with data value 1 in the above list. So, create a new node and place 1 in their data part. That is shown in figure below:



Now, point the link of new node to the first node of the list and new node becomes the first node as shown in figure given below:



An algorithm for insertion of a node at the beginning is given below:

Algorithm for Insertion at the beginning :

In this algorithm we insert a new node called Temp in the beginning of the linked list.

```
Step 1.    If Avail = NULL
               Print "Overflow" go to step 6
Step 2.    Temp=new node
Step 3.    Avail=Avail-Link
Step 4.    Read(Val)
Step 5.    If(First=NULL) then
           First->DATA=Val
           First->Link=First
           Else
           Temp->DATA=Val
           Temp->Link=First
           First =Temp
Step 6.    END
```

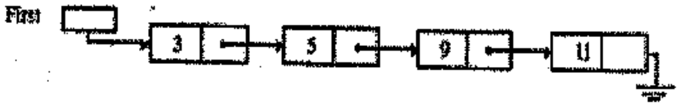
In step 5 we check if First=NULL means that if List is empty then, the new node will be the first node of the list.

Let us write a general function C++ for inserting an element at the beginning of the linked list.

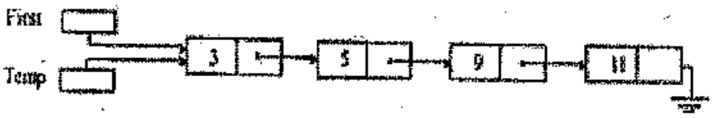
```
// Insert in the beginning of the linked list.
node *addfirst(node*first, data_type value)
{
node*temp
 temp=new node;
temp->data=value;
temp->link=first;
first=temp;
return (first);
}
```

3.4.2.2 Insertion of a Node in Between

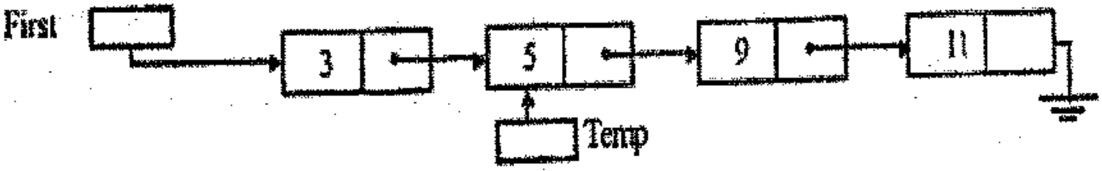
To insert a node in between of the linked list we should know after which node insertion is required. Here we will consider the linked list which is shown in the figure below :



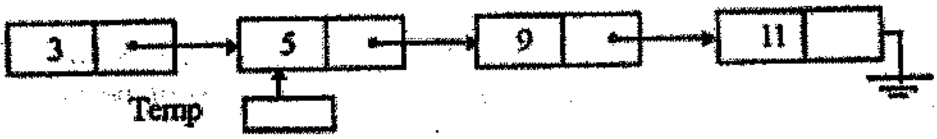
Suppose, we have to insert into 7 after the node having data value 5. Firstly, copy the address of first node in Temp as shown in the figure below:



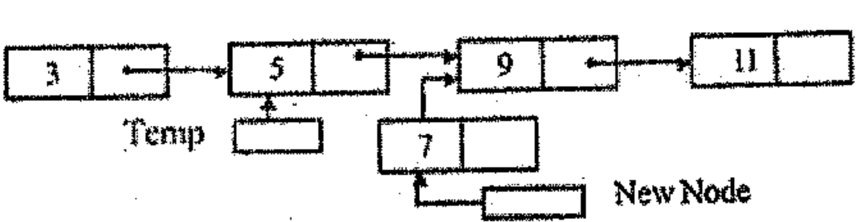
Now compare the data value of Temp node with 5, if it is not equal then move Temp to next node as in figure below:



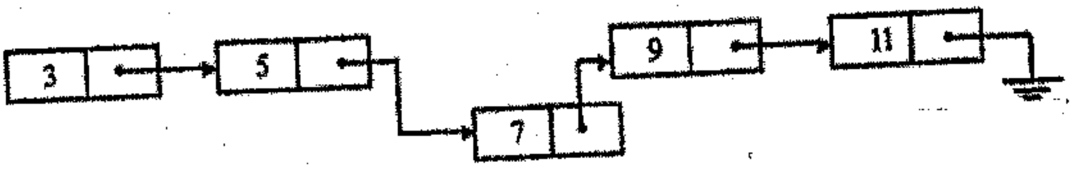
Again, comapre the contents of Temp node with 5, you see it is equal to 5, sothat we have to insert a new node after this node. Create new node with data value 7 as shown in the figure below:



Now, point the link part of new node to the node, which is pointing, by the node after which new node is to be inserted as in the figure below:



Now, point the link part of node in the data value 5 to the new shown as figure given below:



Hence, a new node is inserted in the linked list. An algorithm to insert a node in between linked list is as follows:

Algorithm for Insertion in Between :

In this algorithm we insert a new node after a node with data value equal to Val.

```
Step 1. If Avail = NULL
Print "Overflow" go to step 9
Step 2. Temp=new node
Step 3. Avail=Avail-Link
Step 4. Read(Val)
Step 5. Temp1=First
Step 6. While(Temp1<>NULL)repeat step 7 to 8
Step 7. If(Temp->DATA=Val)
Temp->DATA=Val
Temp->Link= Temp1->Link
Temp1->Link= Temp
Step 8. Temp1=Temp1->Link
Step 9. END
```

// C++ Implementation of Algorithm for Insertion in between the linked list.

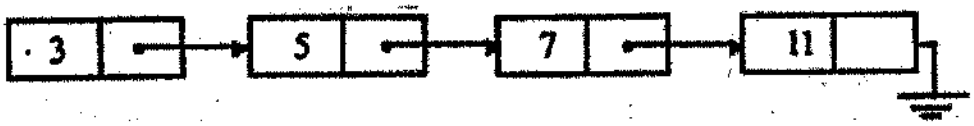
Let us write a general function in C++ for inserting an element in between the linked list.

```
// Insert in between the linked list.
node *addbetween(node*first, data_type value, data_type val)
{
node*temp;
node*temp1;
temp=new node;
temp->data=value;
temp1=first;
while(temp1!=NULL)
{
```

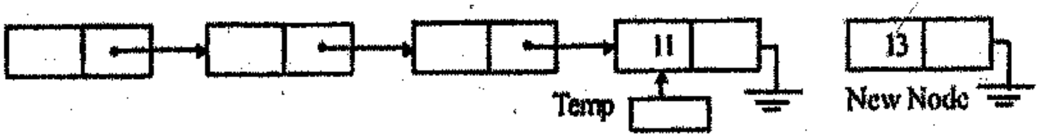
```
if(temp1->data==val)
{
temp->link=temp1->link;
temp1->link=temp;
break;
}
temp1=temp1->link;
}
return(first);
}
```

3.4.2.3 Insertion of a Node at End

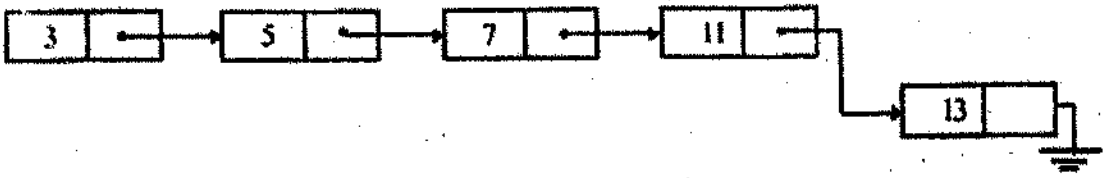
To insert a node in the last of the linked list, we have to traverse the list until end of the list encounter and then point the Link part of last node to the new node. Here we consider the linked list is shown in the figure below:



Suppose, we have to insert a new node with data vale 13 at the end of the above linked list. So, traverse the list until the end of the linked list as shown in figure below



:and point the link of last node to the new node as shown in figure below:



So, a node is inserted at the end of linked list. An Algorithm to insert a new node at the end of the linked list.

Algorithm for Insertion at End :

In this algorithm we insert a node at the end of the linked list.

```
Step 1. If Avail = NULL
Print "Overflow" go to step 11
Step 2. Temp=new node
Step 3. Avail=Avail-Link
Step 4. Read(Val)
Step 5. Temp1=First
Step 6. While(Temp1<>NULL)repeat step 7
Step 7. Temp1=Temp1->Link
Step 8. Temp1->DATA= Val
Step 9. Temp1->Link=Temp
Step 10.     Temp->Link=NULL
Step 11.     END
```

// C++ Implementation of Algorithm for inserting at last of the linked list

Let us write a general function for inserting an element at last of the linked list.

```
//Insert in the last of the linked list
node *addlast(node*first, data_type value)
{
node*temp;
node*temp1,*back;
temp=new node;
temp->data=value;
temp->link =NULL
back=first;
temp1=first->link;
while(temp1!=NULL)
{
back=temp1;
temp1=temp1->link;
}
back->link=temp;
}
return (first);        }
```

3.4.3 Deletion in Link Lists

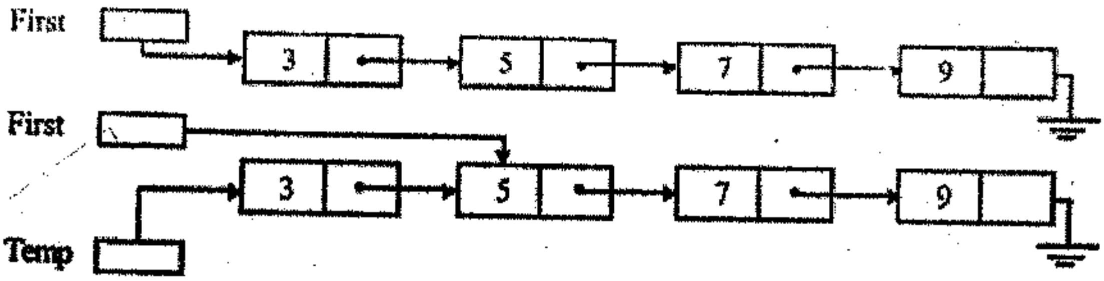
Similar to insertion, deletion is also possible on the same ways of insertion as :

Deletion in the linked list is done in three ways:

- **At the beginning**
- **In between**
- **At the end**

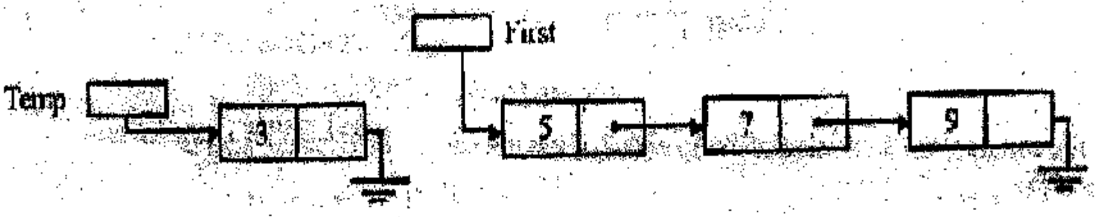
3.4.3.1 Deletion at the Beginning

To delete a node from the beginning of the linked list firstly, copy the first node to some temporary node and move the pointer of first to node next to first. That is shown in figure given below:



Now, break the link between first and second node by starting Temp-> Link=NULL.

The following figure shows as:



So that the first node is deleted from the linked list. Now, remove the deleted node from the memory by using delete () function.

Algorithm for Deletion from beginning :

In this algorithm we delete the first node of linked list.

```
Step 1.   If (First = NULL
          Print "List Empty" go to step 6
Step 2.   Temp=First
Step 3.   First=First->Link
Step 4.   Temp->Link=First
Step 5.   Delete Temp
Step 6.   END
```

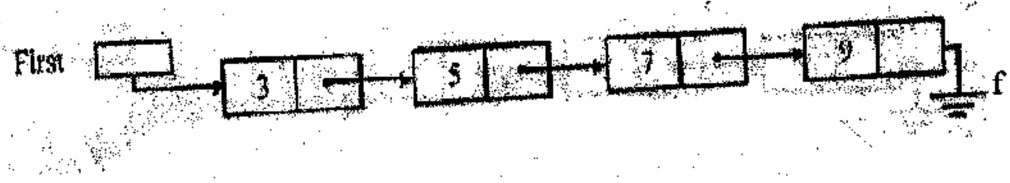
// C++ Implementation of Algorithm to delete an element from the beginning of linked list

Let us write a general function in C++ to delete an element from the beginning of linked list.

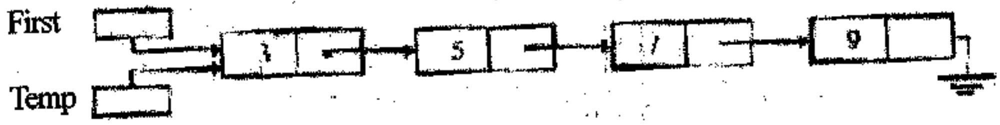
```
// Deletion from the beginning of linked list
node *delfirst(node* first, data_type &value)
{
node*temp;
temp=first;
If(first=NULL)
{ cout <<"\n \t List Empty";
value=-1;
}
else
{
value=temp->data;
first = first -> link;
first ->link = NULL;
delete(temp);
}
return (first);
}
```

3.4.3.2 Deletion in Between

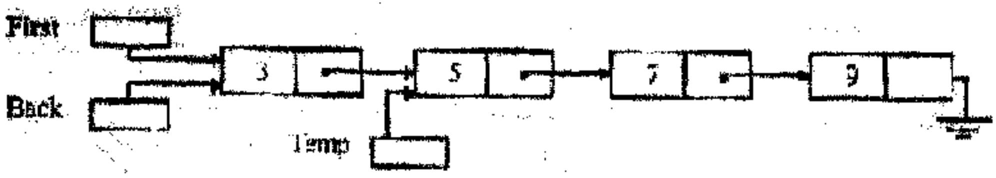
To delete a node from between the linked list, we should know which node is to be deleted. Here we consider the linked list as shown in figure below:



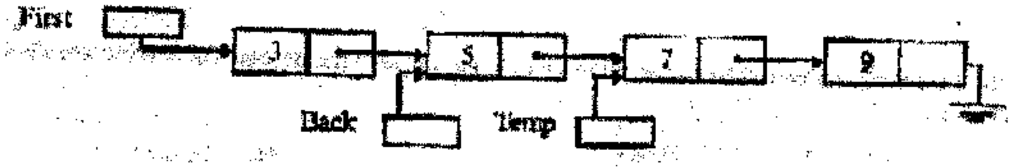
Suppose, we have to delete a node with data 7. Now copy the address of first node in temporary variable say Temp as shown as :



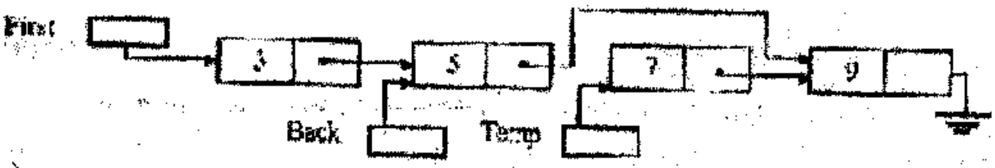
Now, we compare the data value of the node Temp with value to be deleted. So that 3 is not equal to 7 , move to the next node by Temp=Temp->Link. Before moving Temp to the next node, store the address of Temp in another variable say back which will always follow Temp node and shown in figure given below:



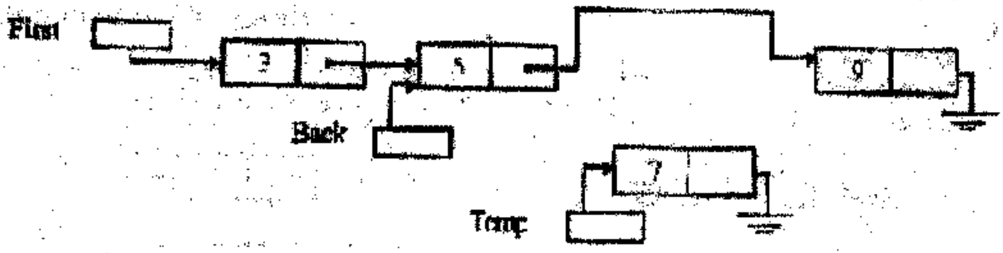
Again, compare the data value of Temp with 7, if it is not equal then copy the address of Temp node in Back and move Temp to the next node as shown in figure given below:



Now, the content of Temp node is equal to 7. So that we have to delete this node. To delete this node simply point the pointer of its previous node (i.e. Back) to the node next to this node and as shown in figure below:



Now, remove the link of Temp node from the linked list and remove the Temp node from the memory using delete() function as shown in figure given below:



Algorithm for Deletion in Between :

In this algorithm we delete a node with data value Val from the linked list.

```
Step 1.   If (First = NULL)
              Print "List Empty" go to step 6
Step 2.   Back=First
Step 3.   Temp=First->Link
Step 4.   While(Temp<>NULL)repeat 5
Step 5.   If(Temp->DATA=Val)
          Back->Link = Temp->Link
          Delete Temp
          else
          Back=Temp
          Temp=Temp->Link
Step 6.   END
```

// C++ Implementation of Algorithm to delete an element in between the linked list

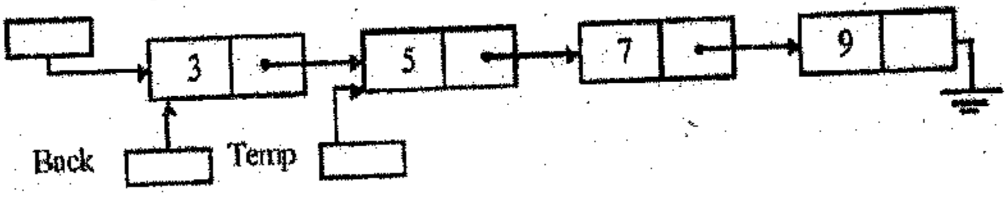
Let us write a general function to delete an element in between the linked list.

```
// Deletion from in between the linked list.
node *delbetween(node*first, data_type &value, data_type val)
{
node*temp, *temp1, *back;
back=temp;
temp=first->link;
If(first==NULL)
{ cout <<"\n \t List Empty";
value=-1;
}
else
{
while(temp!=NULL)
{
if(temp->data==val)
{
back->link=temp->link;
temp->link=NULL;
value=temp->data;
delete(temp);
break;
}
```

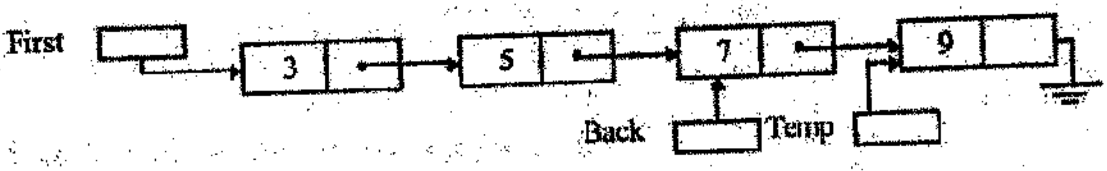
```
        back=temp;
        temp=temp->link;
        }
        }
        return (first);
}
```

3.4.3.3 Deletion at End

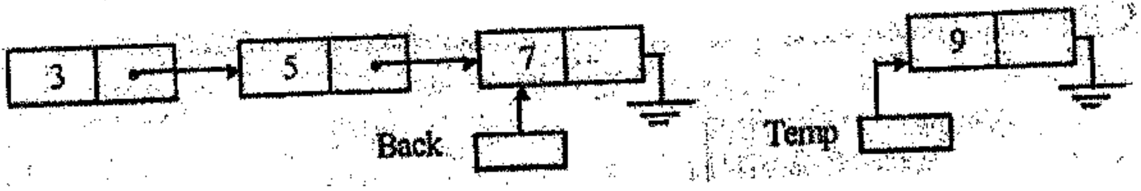
To delete a node from the end of the linked list, we have to traverse the node until end of the linked list. To delete a node from the end we need two variables. One is Temp, which will point to the second node of the list, and other in Back, which will always follow Temp node. Here we consider the linked list as given in figure below:



Now traverse the link until Temp becomes NULL and follows the Temp by Back. Where the Temp reaches end of list, which is looked like as shown in figure below:



Now, simply assign the link pointer to NULL shown in figure given below:



So that, a node is deleted from the linked list.

Algorithm for Deletion at End :

In this algorithm we delete a node from the end of the linked list.

```
Step 1.    If (First = NULL)
                    Print "List Empty" go to step 9
```

```
Step 2.    Back=First
Step 3.    Temp=First->Link
Step 4.    While(Temp<>NULL)repeat 5 to 8
Step 5.    Back=Temp
Step 6.    Temp=Temp->Link
Step 7.    Back->Link=NULL
Step 8.    Delete Temp
Step 9.    END
```

// C++ Implementation of Algorithm to delete an element at the end of the linked list

Let us write a general function to delete an element at the end of the linked list.

```
// Deletion from the last of linked list.
node *dellast(node*first, data_type &value)
{
node *temp, *temp1;
node *back;
temp=first;
if(first = = NULL)
{
cout <<“\n \t List Empty”;
value=-1;
}
else
{
while(temp->link!=NULL)
{
back=temp;
temp=temp->link;
}
back->link=NULL;
value = temp->data;
delete(temp);
}
return (first);
    }
```

3.4.4 Traversal of Linked List

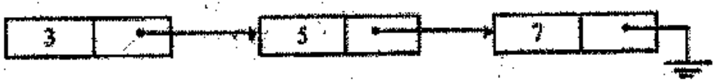
Traversal of a linked list means processing all the nodes of the list one by one. For example, print the list, count the nodes, change the data value of nodes etc.

Algorithm for Traversal of a linked list is given below :

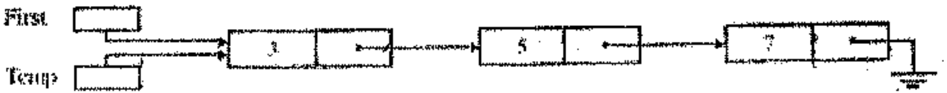
In this algorithm printing the contents present in the DATA part of the list traverses a link list. The First node of the list is pointing by First and the traversing is stop when NULL is encountered.

```
Step 1.    if first = NULL
           Print "List is Empty" go to step 6
Step 2.    Temp=First
Step 3.    While(Temp<>NULL) repeat 4 to 5
Step 4.    Print (Temp->DATA)
Step 5.    Temp=Temp->Link
Step 6.    END
```

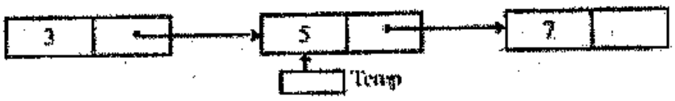
Let us illustrate the above algorithm, suppose , a linked list is given in figure given below:



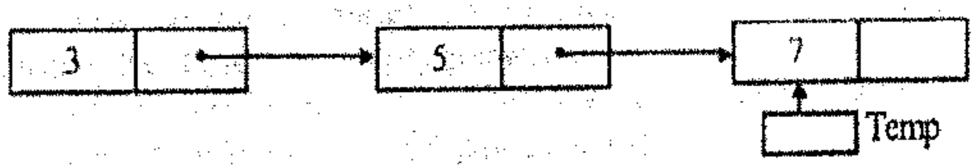
In first step copy the address of first into the variable Temp shown is figure given below:



Travel the list till end of list is encounter. First print the content of TEMP->DATA. Then move the Temp to next node by using Temp=Temp->Link as shown in figure given below:



Again, print (Temp->DATA) and move to the next node by using Temp=Temp->Link as shown in figure given below:



For Example:

Write a program to create and traverse the linked list. The linked list contains data of type integer.

// C++ Implementation of Algorithm to create and traverse the linked list

```
#include <iostream.h>
```

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
//Create a linked data structure
struct node
{
int data;
node*link;
};
//Main programming logic
void main()
{
node *first, *temp, *last;
int n,i;
clrscr();
cout<<“\n\n Enter how many nodes to create in the linked list :”;
cin>> n;
//Creation of linked list
first=new node;                    // Allocating memory for first node
cout<<“\n\tEnter the data value of node 1:”;
cin>>first->data;
first->link=NULL;
temp=first;
for(i=1;i<n;i++)
{
last=new node;
cout<<“\n\t Enter the data value of node”<<i+1<<“->”;
cin>>last->data;
last->link=NULL;
temp->link=last;                   //Creating link between the nodes
temp=last;
}
//Traversing of linked list
temp=first;
clrscr();
cout<<“\n\t The Linked list values are:\n”;
while(temp!=NULL)
{
cout<<"\n"<<temp->data;
}
getch();        // For Press any key
}
```

3.4.5 Searching in Linked List

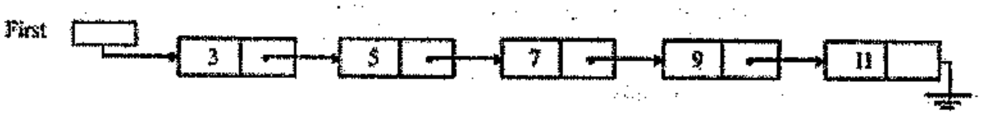
Searching is a process in which a given item is searched in the given linked list. To search a desired Item we have to compare the Item with the DATA part of every node of the List, until end of the list is encountered.

Algorithm for Searching a linked list :

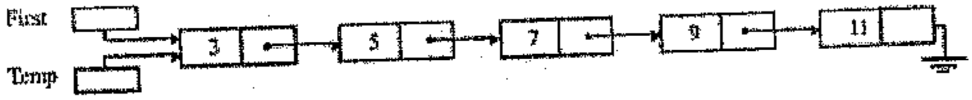
In this algorithm we traverse the linked list to search an element called Item. If element is found, search is stopped, otherwise search will continue until end of the list is encountered.

```
Step 1.    If First = NULL Then
                Print "List Empty" go to step 8
Step 2.    Temp=First
Step 3.    Read(Item)
Step 4.    While(Temp<>NULL) and (Found=False) repeat step 5
Step 5.    If(Temp->DATA=Item)
           Found=True
                else
           Temp=Temp->Link
Step 6.    If(Found=True)
           Print" Element Found"
Step 7.    If(Found=False)
           Print" Element not Found"
Step 8.    END
```

Let us consider a linked list shown in figure given below:

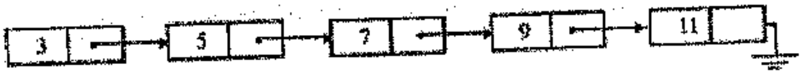


Suppose we have to search an Item 7 in the linked list. We store the address of first node in variable Temp and compare the DATA part of Temp with Item i.e. 3 #7.

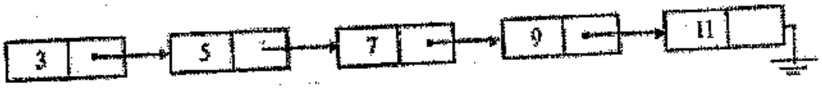


So, Temp is moved to the next node by starting Temp=Temp->Link.

Now, the Temp is at second node:



Again the contents of Temp with Item i.e., 5#7. So that move Temp to the next node and compare them.



Contents of node with Item i.e. 7=7. Therefore element is found and stop searching and print appropriate message.

// C++ Implementation of Algorithm to create a linked list and search

For Example:

Write a C++ program to create a linked list and search a particular data value of integer data type.

```
// The linked list contains data of type integer
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
//Linked data structure
struct node
{
int data;
node*link;
};
//Main programming logic
void main()
{
node *first, *temp, *last;        //Creates three pointers nodes
int n,i,item,flag;
clrscr();
```

```
cout<<"\n\n Enter how many nodes in the linked list :";
cin>>n;                    //Number of nodes in the linked list
//Creation of linked list
first=new node;         // Allocating memory for first node
cout<<"\n\t Enter the data value of node:";
cin>>first->data;
first->link=NULL;
temp=first;
for(i=1;i<n;i++)
{  last=new node;
cout<<"\n\t Enter the data value of the node:";
cin>>last->data;
last->link=NULL;
temp->link=last;                    //Creating link between the nodes
temp=last;
}
//Traversing of linked list
temp=first;
clrscr();
cout<<"\n\t The Linked list values are:\n";
while(temp!=NULL)
{
cout<<"\n"<<temp->data;
temp=temp->link;
}
//Searching in the linked list
flag=0;
cout<<"\n\n\t The data value of node to be searched";
cin>>Item;
temp=first;
while(temp!=NULL)
{
        if(temp->data= =item)
{
        flag=1;
        break;
}
~~temp=temp->link;~~
temp=temp->link;
}
if ( flag==1)
cout<<"\n \t Search is successful";
else
cout<<"\n \t Search is unsuccessful";
getch();        // For Press any key
}
```

```
Step 4.   While (NewFirst <> NULL) repeat step 5 to 8
Step 5.   Back = Temp
Step 6.   Temp = NewFirst
Step 7.   New First = NewFirst ->Link
Step 8.   Temp -> Link = Back
Step 9.   First = Temp
Step 10.  End
```

// C++ Implementation of Algorithm for Reverse of a linked list

For Example :

Write a C++ program to create a linked list and reverse the linked list.

// this program create a linked list and printing in reversal order.

// the linked list contains data of type integer.

```
            # include<iostream.h>
            # include<stdio.h>
            # include <conio.h>
            # include <stdlib.h >
            struct node
                  {
                  int data ;
                  node *link;
                  } ;
                  node * create() ;
                  void traverse (node *first) ;
                  void main ()
                  {
                  node *first1, *temp, *back ;
```

```
node *newfirst ;
int val ;
clrscr() ;
first1 = create();
```

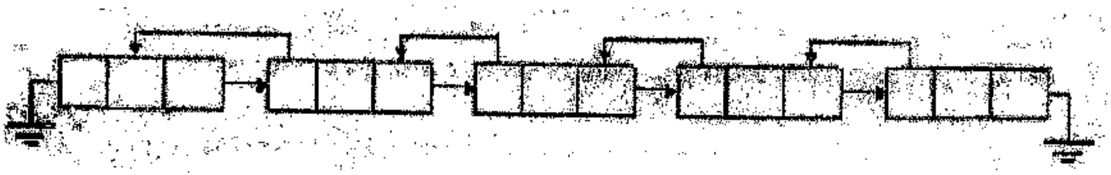
3.5 Doubly Linked List

A double linked list is defined as -

A linked list that has two links in each of its nodes, with one link pointing to the preceding node and the other pointing to the succeeding node is called doubly linked list.

Therefore, we can traverse the doubly linked list either in forward direction or in backward direction. There is no need to know the address of first node to traverse a list.

The double linked list is shown in the figure given below:



Implementation of Doubly Linked List:

Structure of a node of Doubly Linked List can be defined as:

```
struct node
{
        data _ type data ;
        node            * rtlink, ltlink ;
} ;
node *current, *tail;
```

Algorithm to create a Doubly Linked List :

In this algorithm a doubly linked list containing of n nodes is created. Avail is the next available memory of the computer. Each node of the list has two parts DATA and LINK. The **new** operator is used for the dynamic allocation of node.

```
Step1. If Avail = NULL
            print "Overflow" go to step 14
Step2. current = new code
Step3. current -> data = Val
Step4. current -> rtlink = NULL
Step5. current -> ltlink = NULL
Step6. tail = current
```

```
Step7.  for i = 1 to n -1 repeat step 8 to 13
Step8.  current = new node
Step9.  current ->data = Val
Step10.      current -> rtlink = NULL
Step11.      current -> ltlink = tail
Step12.      tail -> rtlink = current
Step13.      tail = current
Step14.      END
```

// C++ Implementation of Algorithm to create a doubly Linked List

For Example:

Write a program to create a doubly linked list of integer.

// this program creates a doubly linked list.

//the linked list contains data of type integer.

```
    # include<iostream.h>
    # include<stdio.h>
    # include<conio.h>
    # include<stdlib.h>
    //create a doubly linked data structure
    struct node
    {
            int data ;
            node *rtlink, *ltlink ;
    } ;
    // Main programming logic
    void main( )
    {
    node *current, *tail ;
    int n,i;
    clrscr();
    cout <<" \n\t Enter how many nodes to creates in the doubly linked list:";
    cin >> n;                       // Number of nodes in the linked list
    // Creation of linked list
    current = new node ;    // Allocating memory for first node
    cout <<"\n\t Enter the data value of node1:" ;
    cin >>current -> data ;
```

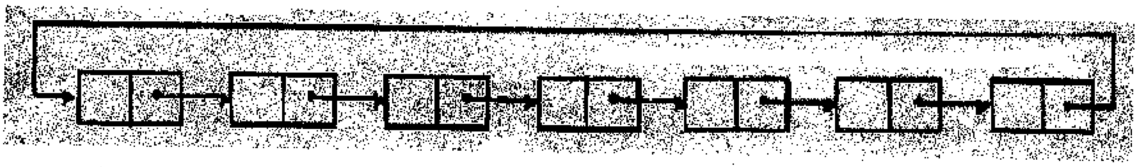
```
        current -> rtlink = NULL ;
        current ->ltlink = NULL ;
        tail = current ;
        for (i = 1 ; i < n; i++)
        {
                current = new node ;
                cout << “n\t\ Enter the data value of node” << i + 1 << “ ->“ ;
                cin >> current -> data ;
                current -> rtlink = NULL ;
                current -> Itlink = tail ;
                tail -> rtlink = current ;
                tail = current ;
        }
        // traversing of linked list
        while (current ->ltlink ! = NULL)              // Move the current
                current = current -> Itlink ;          // pointer to the beginning of the list
        clrscr ();
        cout << “The doubly Linked list values are : \ n “ ;
        while (current ! = NULL)
        {
                cout << “\n”<< current -> data ;
                current = current -> rtlink ;
        }
        getch () ;
        }
```

3.6 Circular Link List

It is defined as –

A linked list in which the last node is pointing to the first node of the list instead of point to NULL is known as circular linked list.

This is shown in figure given below :



Algorithm for creating a circular linked list :

In this algorithm a circular linked list consisting of n nodes is created. Avail is the next available memory of the computer.

The first node of the list is pointed by pointer First and the last node of the list points to first node. Each node of the list has two parts DATA and LINK. The **new** operator is used for the dynamic allocation of node.

```
Step 1.   If Avail = NULL
          print "Overflow " go to step 13.
Step 2.   First = new node
Step 3.   Avail = Avail -> LINK
Step 4.   First -> DATA = Val
Step 5.   First -> Link = NULL
Step 6.   Temp = First
Step 7.   For i = 1 to n -1 repeat step 8 to 12
Step 8.   last = new node
Step 9.   last -> DATA = Val
Step 10.  last -> LINK = first
Step 11.  Temp -> LINK = last
Step 12.  Temp = last
Step 13.  END
```

// C++ Implementation of Algorithm to create a Circular List

For Example:

Write a program to create a circular linked list of integer data type.

```
// this program creates a circular linked list.
//the linked list contains data of type integer.
# include <iostream.h>
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>
//create a doubly linked data structure
//create a circular linked data structure
struct node
{
        int data ;
        node * link ;
        } ;
        // Main programming logic
```

```
void main ()
{
node *first, *temp, *last ;
int n, i;
cirscr () ;
cout << "\n\n Enter how many nodes to create in the circular linked list:" ;
cin >> n ;
//Creation of circular linked list
first = new node ;                        //Allocating memory for fist node
cout << " \n\t Enter the data value of node1 :" ;
cin >> first ->data ;
first -> link = NULL;
temp = first ;
for (i = 1 ; i < n ; i++)
{
last = new node ;
cout << " \n\t Enter  the data value of node" << i + 1 << " - > " ;
cin >> last -> data ;
last -> link -> = first ;            // creating link between last and first nodes
temp -> link  = last ;                    // creating link between the node
temp = last ;
        }
// traversing of circular linked list
temp = first ;
clrscr();
cout <<" The Circular Linked list values are : \n " ;
do
{
        cout << "\n" << temp -> data ;
        temp = temp -> link ;
}
while (temp! = first);
getch ();
}
```

3.7 Link List Representation of Sparse Matrix

We have learnt in the previous unit, the representation of a sparse matrix as an array of 3-tuples suffers from one important drawback. This is removed by using the linked list representation of a sparse matrix. In the linked list, representation of a sparse matrix a separate list is maintained for each column as well as each row of the matrix. Suppose, there is a matrix is of size 3x3, then there will be 3 lists for 3 columns and 3 lists for 3 rows.

A node in a list always stores the information of non-zero element of the sparse matrix. The head node for a column list stores the column number, a pointer to the node, which comes first in the column, and a pointer to the next column head node.

Therefore, the structure for column head node is given by :

```
struct cheadnode
{
struct node *down;
int colno;
struct cheadnode *next;
};
```

A head node for a row list stores, a pointer to the node, which comes first in the row list, and a pointer to the next row head node.

So that, the structure for row head node is given by :

```
struct rheadnode
{
struct rheadnode *next;
int rowno;
struct node *right;
};
```

In other words, a node stores the row number, column number and the value of the non-zero element of the sparse matrix. It also stores a pointer to the node that is immediately to the right of the node in the row list as well as a pointer to the node that is immediately below the node in the column list.

Thus the structure for a node will be :

```
struct node
{
int row;
int col;
int val;
struct node *down;
struct node *right;
};
```

Besides it, there is a special node is used to store the total number of rows, total number of columns, a pointer to the first row head node and pointer to the first column head node. The information stored in it used for traversing the list.

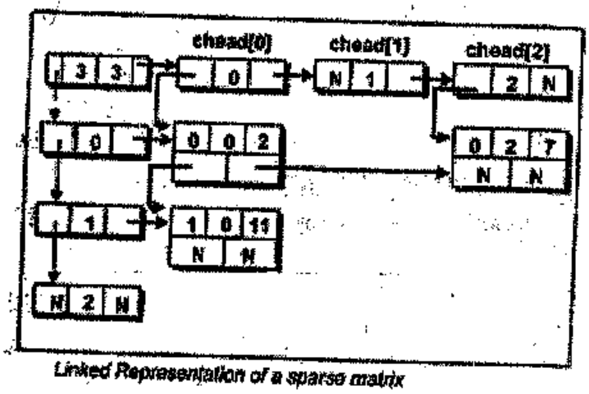
Thus the structure for this special node is given by :

```
struct spmat
{
struct rheadnode *firstrow;
int noofrows;
int noofcols;
struct cheadnode *firstcol;
};
```

If a particular column list is empty then the field down of the column head node will be NULL.

Similarly, if a row list is empty then the field right of the row head node will be empty. If a node is the last node in a particular column list or a particular row list then the field down or the field right of the node will be NULL.

The diagrammatic representation of linked list of a sparse matrix (size of matrix 3x3) is shown in figure given below:



Linked Representation of a sparse matrix

Here, we will discuss the example program which stores sparse matrix in the linked list

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
#define MAX1 3
#define MAX2 3
//structure for col head node
struct cheadnode
{
int colno;
struct node *down;
struct cheadnode *next;
};
//structure for row head node
struct rheadnode
{
int rowno;
struct node *right;
struct rheadnode *next;
};
//structure for node to store element
struct node
{
int row;
int col;
int val;
struct node *right;
struct node *down;
};

//structure for special head node
struct spmat
{
struct rheadnode *firstrow;
struct rheadnode *firstcol;
int noofrows;
int noofcols;
};
struct sparse
{
int *sp;
int row;
struct spmat *smat;
struct cheadnode *chead[MAX2];
struct rheadnode *rhead[MAX1];
struct node *nd;
};
```

```
void initsparse (struct spasre* );
void create_array (struct spasre* );
void display (struct spasre);
void count (struct sparse);
void create_triplet(struct spasre*,  struct sparse);
void create_list (struct spasre* );
void insert (struct spasre*, struct  spmat*, int, int, int);
void show_list(struct spasre);
void cesparse (struct spasre* );

void main()
struct sparse s1,s2;
clrscr();
initsparse (&s1);
initsparse (&s2);
create_array(&s1);
cout<<"\n Elements in Sparse Matrix :";
display(s1);
create_triplet(&s2,s1);
create_list(&s2);
cout<<"\n \n Information stored in liked list :";
show_list(s2);
delsparse(&s1);
delsparse(&s2);
getch();
}
// Initialization of structure elements
void initsparse (struct sparse *p)
{
int i;
//create row head nodes
for(i=0;i<MAX1;i++)
p->rhead[i]=(struct rheadnode *) malloc (sizeof (struct rheadnode));
//Initialize and link  row head nodes together
for(i=0;i<MAX1-1;i++)
p->rhead[i]->next=p->rhead[i+1];
```

```
p->rhead[i]->right=NULL
p->rhead[i]->rowno=i;
}
p->rhead[i]->right=NULL;
p->rhead[i]->next=NULL;
//create column head nodes
for(i=0;i<MAX1;i++)
p->chead[i]=(struct cheadnode *) malloc (sizeof (struct cheadnode));
//Initialize and link  col head nodes together
for(i=0;i<MAX2-1;i++)
p->chead[i]->next=p->chead[i+1];
p->chead[i]->down=NULL
p->chead[i]->colon=i;
}
p->chead[i]->down=NULL;
p->chead[i]->next=NULL;
//Create and initialize special head node
p->smat=(struct spmat *) malloc (sizeof (struct spmat));
p->smat->firstcol=p->chead[0];
p->smat->firstrow=p->rhead[0];
p->smat->noofcols=MAX2;
p->smat->noofrows=MAX1;
}
// Creates, dynamically the matrix of size MAX1 x MAX2
void create_array (struct sparse *p)
{
intn,i;
p->sp = (int *)malloc (MAX1* MAX2* sizeof(int));
//Get the element and store it
for(i=0;i<MAX1*MAX2;i++)
{
cout<<“Enter element no. “<<i<<\n;
cin>>n;
 *(p->sp+i)=n;
}
 }
```

```
//Display the content of the matrix.
Void display (struct sparse s)
{
int i;
/*traverse the entire matrix*/
for(i=0;i<MAX1*MAX2;i++)
{
if(i%MAX2==0)
cout<<“\n”;
cout<<“\t”, *(s.sp+i);
}
}
//Counts the number of non-zero elements
int count(struct sparse s)
{
int cnt=0,i;
for(i=0;i<MAX1*MAX2;i++)
{
if(*(s.sp+i)!=0)
cnt++;
}
return cnt;
}
//Creates an array of triplet containing ifomation about non-zero elements
void create_triplet (struct sparse *p, struct sparse s)
{
int r=0,c=-1,l=-1,i;
p->row=count(s);
p->sp=(int *) malloc (p->row*3*sizeof(int));
for(i=0;i<MAX1*MAX2;i++)
{
c++;
        if(((i%MAX2)==0) && (i!=0))
{
r++;
c=0;
}
/*Checks for non-zero element Row, column and non-zero element value is assigned to the
matrix*/
if(*(s.sp+i)!=0)
{
l++;
*(p->sp+l)=r;
l++;
*(p->sp+l)=c;
l++;
*(p->sp+l)= *(s.sp+i);
}
}
}
// Stores information of triplet in a linked list form
void create_list(struct sparse *p)
{
int j=0,i;
for(i=0;i<p->row;i++,j+=3)
insert (p,p->smat, *(p->sp+j), *(p->sp+j+1), *(p->sp+j+2));
}
//Insert element to the list
void insert (struct sparse *p, struct spmat *smat, int r, int c, int v)
{
Struct node *temp1, *temp2;
Struct rheadnode *rh;
Struct cheadnode *ch;
int i,j;
//Allocate and initialize memory for the node
p->nd=(struct node *) malloc (sizeof(struct node));
p->nd->col=c;
p->nd->row=r;
p->nd->val=v;
//get the first row head node
rh=smat->firstrow;
// get the proper row head node
for(i=0;i<r;i++)
rh=rh->next;
```

```
temp1=rh->right;
//If no element added in a row
If(temp1==NULL)
{
rh->right = p-> nd;
p->nd->right = NULL;
}
else
{
        //Add element at proper position
while ((temp1!=NULL) && (temp1->col<c))
{
temp2=temp1;
temp1=temp1->right;
}
temp2->right=p->nd;
p->nd->right=NULL;
}
//Link proper col head node with the node
ch=p->smat->firstcol;
for(j=0;j<c;j++)
ch=ch->next;
temp1=ch->down;
//If col not pointing to any node
If(temp1==NULL)
{
ch->down=p->nd;
p->nd->down=NULL;
}
else
{
//Link previous node in column with next node in same column
while (temp1!=NULL) && (temp1->row<r))
{
temp2=temp1;
temp=temp1->down;
}
```

```
temp2->down=p->nd;
p->nd->down=NULL;
}
}
void show_list (struct spasre s)
{
struct node *temp;
/*get the first row head node
int r=s.smat->noofrows;
int i;
cut<<“\n”;
for (i=0; i<r; i++)
{
temp =s.rhead[i]->right;
if(temp!=NULL)
{
while(temp->right!=NULL)
{
cout<<“Row : Col : Val : \n” <<temp->row, temp->col, temp->val;
temp=temp->right;
}
If (temp->row= =i)
cout<<“Row : Col : Val : \n” <<temp->row, temp->col, temp->val;
}
}
}

//Deallocates memory
Void delsparse (struct sparse *p)
{
int r=p->smat->noofrows;
struct rheadnode *rh;
struct node *temp1, *temp2;
int i,c;                          //deallocate memory of nodes by traversing row wise
for(i=r-1;i>=0;i- -)
{
```

```
rh=p->rhead[i];
temp1=rh->right;
while (temp1!=NULL)
{
temp2=temp1->right;
free (temp1);
temp1=temp2;
}
}                                   //deallocate memory of row head nodes
for(i=r-1;i>=0;i- -)
free(p->rhead[i]);                  //deallocate memory of columns head nodes
c=p->smat->noofcols;
for(i=c-1; i>=0; i—)
free(p->chead[i]);
}
```

Output :

```
Enter element no. 0:2
Enter element no. 1:0
Enter element no. 2:7
Enter element no. 3:11
Enter element no. 4:0
Enter element no. 5:0
Enter element no. 6:0
Enter element no. 7:0
Enter element no. 8:0

Enter elements in sparse matrix :
2       0       7
11      0       0
0       0       0

Information stored in linked list :
Row : 0 Col : 0 Val : 2
Row : 0 Col : 2 Val : 7
Row : 1 Col : 0 Val : 11
```

Self Learning Exercises

Q1. Declare a linked list having two data members name (char 20) and rollno (integer).

Q2. Give two merits and demerits of linear list.

Q3. Give two merits and demerits of linked list.

Q4. Give one merit and demerit of Circular lists.

Q5. Let LIST be a linked list in the memory, write a function that finds the average MEAN of the values in the list.

Q6. State whether **True or False**:

1. calloc() takes three arguments.
2. A linked list occupies more space than an array having same number and type of elements.
3. The length of the null list () is 1.
4. The length of the list ((5, 6), 0 0 (9, 6, 3, 5)) is 3.
5. The length of list A = (a, A, a) is Infinite.

3.8 Summary

- A linked list is a way to represent a list pf items in such a way that each element of the list points to the next element.
- A Linked list is a dynamic data structure.
- Linked List has several Advantages over Array because of Dynamic Memory Allocation. Linked List doesn't face the problems like overflow and Memory Wastage due to Static Memory Allocation.
- Two Lists can be Merged and Concatenated by link manipulation.
- Each node of a Linked List contains address of its successor Node.
- The various operations performed on linked list are: creation, traversal, insertion, deletion and search.
- Linear or Single Linked List can not be traversed in reversed, this drawback is removed in doubly and Circular Linked List.
- A doubly linked list is a linked list in which each node contains two links, one to the preceding node and the other to the next node in the list.
- In Circular Linked List Link pointer of last node of Linked List points to first node of Linked List instead of pointing to NULL.
- In Double Linked List each node contains address of its previous node in addition to the address of next node. This provides two way traversing.

3.9 Glossary

Linked List: A way to represent a list of items so that each element of the list points to the next element.

Node: Each item in the list is called Node and contains two fields, an information field and a next address filed.

Circular Linked List: A linked list in which the last node, instead of having a NULL pointer, points to the node at the beginning of the list.

Overflow: If you try to insert a node, when there is no memory available, it is called Overflow.

Traversal: Traversal of a linked list means processing all the nodes of the list one by one.

Free Storage List (Free Pool): A list keeping account of available free memory. Sometimes called AVAIL List.

Free Storage Pool: The pool of unallocated heap memory.

Garbage Collection: Method of adding released memory (through a deletion) in the free pool.

3.10 Further Readings

1. Data structures, Algorithms and Applications in C++ by S.Sahni, University press (India) pvt ltd / Orient Longman pvt.ltd., 2nd edition

2. Data Structures and Algorithm Analysis in C++ by Mark Allen Weiss, Pearson Education, Second Edition

3. Data structures and Algorithms in C++ by Michael T.Goodrich, R.Tamassia and D Mount, Wiley Student Edition, John Wiley and Sons

4. Data structures using C and C++ by Langsam, Augenstein and Tanenbaum, PHI/Pearson Education.

5. Data Structures and Algorithms in C++ by Adam Drozdek, Vikas Publishing House / Thomson International Student Edition., Second Edition

3.11 Answers to the Self Learning Exercises

ANS. 1

```
struct node
{
int rollno;
char name [20];
node*LINK;
};
```

ANS. 2

Merits:-

(i) Implementation is easy

(ii) Better for fixed elements

Demerits:-

(i) Insertion & deletion is difficult

(ii) There is wastage of memory

ANS. 3

Merits:-

(i) Insertion & deletion is easy.

(ii) There is no memory waste.

Demerits:-

(i) Implementation is complicated.

(ii) Run-time memory allocation is difficult to trap the error.

ANS. 4

Merits:-

(i) In circular list, we can move in the list from the last node to the first node.

Demerits:-

(i) Front & Rear should be managed properly, otherwise over writing chance is possible.

ANS. 5

```
//Traversing the linked list for Mean
int average(node*first)          // function prototype
{
int total=0;
int mean =0, n=0;
node*temp;
temp=first;
clrscr();
cout<<"The Linked List values are :\n";
while (temp!=NULL)
{
total=total+temp->data;
n++;
temp=temp->link;
}
mean=total/n;
return mean;
}
```

ANS 6.

1. False
2. True

3. False

4. True

5. True

3.12 Unit–End Questions

Q1. Explain the linked list with suitable example.

Q2. How to concatenate a linked list? Show it through an example.

Q3. Give an example to add or delete a node from linked list from the beginning of the list.

Q4. What is doubly linked lists? How is it different from simple linked list?

Q5. What is Circular linked list? How is it different from doubly linked list?

Q6. Fill in the blanks:

1. A computer program is an algorithm written in ______________.
2. Each Node of a linked List contains address of its ______________ Node.
3. A node of a linked list is allocated space from ______________ memory.
4. Linear or Single Linked List cannot be traversed in reverse direction; this drawback is removed in ____________ and ____________ Linked Lists.

****

UNIT – IV

STACK DATA STRUCTURE

STRUCTURE OF THE UNIT

4.0 Objective
4.1 Introduction
4.2 Operations on STACK
4.3 Implementation of STACK
    4.3.1 Static implementation of STACK
    4.3.2 Dynamic implementation of STACK
4.4 Applications of STACK
4.5 Summary
4.6 Glossary
4.7 Further Readings
4.8 Answers to the Self learning exercises
4.9 Unit -End Questions

4.0 Objective

After copletion of this unit you will learn

- What is Stack ?
- What are the basic Operations on STACK
- Static and Dynamic Implementation of STACK
- Applications of STACK

4.1 Introduction

A stack is a special type of list, where only the element at one end can be accessed. Items can be "pushed" onto one end of the stack structure. New items are inserted before the others, as each old element moves down one position. The first element is referred to as the "top" item, and is the only item that may be accessed at any time. In order to access items that are further down the stack, they must be moved to the top by "popping" the appropriate number of items. Popping refers to removing the top element of a stack. This is referred to as a LIFO structure, "Last In, First Out".

These rules make stacks very restricted in use, however they are very efficient and much easier to implement than lists. The uses of stacks vary from programming a simple card game, to maintaining the order of operations in a complex program. For example, a stack is useful in a management program where the newest tasks must be executed first. The node of a stack is usually presented with the following structure, which is very similar to that of a list node.

LIFO

It is "LIFO : Last input first output ", this looks like placing some copybooks on your desk , and then when you want one of them (assume that they are all simillar) you've to take the one on the top first which is the last one was placed.

In the following chapters we will look at some examples of abstract data structures.These structures store and access data in different ways which are useful in different applications. In all cases the structures follow the **principle of data abstraction** (the data representation can be inspected and updated only by the abstract data type's operations). Also, the algorithms used to implement the operations **do not depend on the type of data** to be stored.A **stack** is a limited version of an array. New elements, or **nodes** as they are often called, can be added to a stack and removed from a stack only from one end. For this reason, a stack is referred to as a LIFO structure (Last-In First-Out).

Stacks have many applications. For example, as processor executes a program, when a function call is made, the called function must know how to return back to the program, so the current address of program execution is pushed onto a stack. Once the function is finished, the address that was saved is removed from the stack, and execution of the program resumes. If a series of function calls occur, the successive return values are pushed onto the stack in LIFO order so that each function can return back to calling program. Stacks support recursive function calls in the same manner as conventional nonrecursive calls.

Stacks are also used by compilers in the process of evaluating expressions and generating machine language code. They are also used to store return addresses in a chain of method calls during execution of a program.

4.2 Operations on STACK

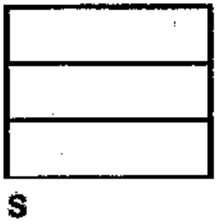
An abstract data type (ADT) consists of a data structure and a set of **primitive operations**. The main primitives of a stack are known as:

- **Push** adds a new node
- **Pop** removes a node

Additional primitives can be defined:

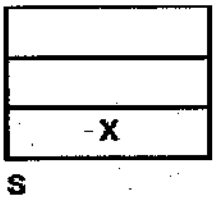
- **IsEmpty** reports whether the stack is empty
- **IsFull** reports whether the stack is full
- **Initialise** creates/initialises the stack
- **Destroy** deletes the contents of the stack (may be implemented by re-initialising the stack)
- **Initialise** Creates the structure – i.e. ensures that the structure exists but contains no elements

e.g. ***Initialise(S)*** creates a new empty stack named S

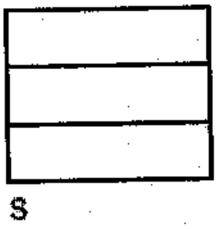


Push

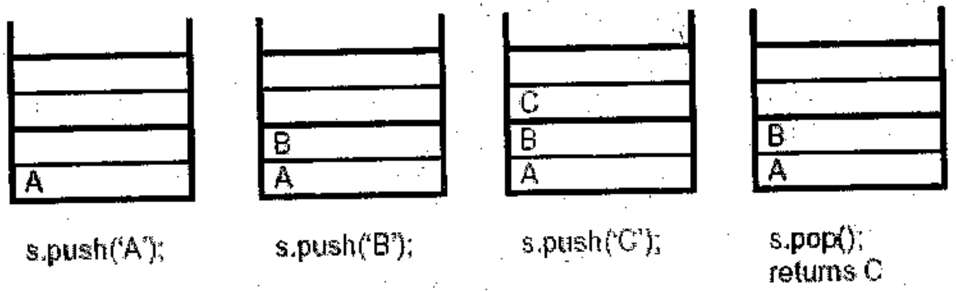
e.g. ***Push(X,S)*** adds the value X to the TOP of stack S



Pop
e.g. ***Pop(S)*** removes the TOP node and returns its value



Example



We could try the same example with actual values for A, B and C.
A = 1 B = 2 C = 3

4.3 Implementation of STACK

The Java Collections Framework includes a set of ready made data structure classes, including a *Stack* class. However, you will create your own stack class inorder to learn how a stack is implemented. Your class will be a bit simpler than the Collections Framework one but it will do essentially the same job.

Static and dynamic data structures
A stack can be stored in:

- a static data structure

OR

- a dynamic data structure

4.3.1 Static data structures

These define collections of data which are fixed in size when the program is compiled.

An **array** is a static data structure. The array implementation of a stack is more interesting, and can be faster on many different types of machines. The array implementation has a counter called a stack pointer that denotes the position of the array where a value was last inserted. Assuming that the array a is one-dimensional and has indices ranging from 1 to some large value N, let the stackpointer be denoted by j. This means that the last push() operation stored value a[j].

If another push operation is specified, then we need to perform the following steps:

1. Check to see if i = N. If so, the stack is full and we cannot execute push() and must notify the user of stack overflow.

2. If j < N, then increment j and store the value v in push(v) as a[j] = v.

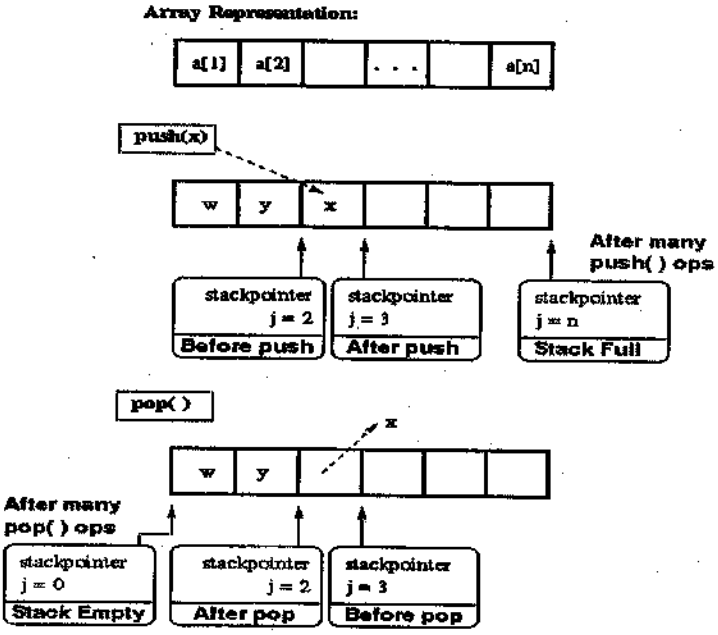
In contrast, if a pop operation is specified, then we need to perform the following steps:

1. Check to see if j > 0. If not, the stack is empty and we cannot execute pop() and must notify the user of stack underflow.

2. If j > 0, then decrement j and return the value v at a[j].

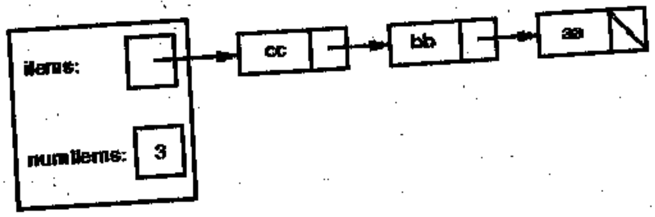
3. As an optional step, store a sentinel value that indicates a null value in a[j]. This is useful in case the stackpointer is lost, corrupted, or destroyed.

The peek operation is implemented as return(a[1]).



4.3.2Dynamic data structures

These define collections of data which are variable in size and structure. They are created as the program executes, and grow and shrink to accommodate the data being stored. As discussed in the UNIT II, in linked list if we insert at the beginning and delete from beginning only then linked list will act as an Stack data structure.an important property of stacks is that items are only pushed and popped at one end (the top of the stack). If we implement a stack using a linked list, we can choose which end of the list corresponds to the top of the stack. It is easiest and most efficient to add and remove items at the front of a linked list, therefore, we will choose the front of the list as the top of the stack (i.e., the items field will be a pointer to the node that contains the top-of-stack item). Below is a picture of a stack represented using a linked list; in this case, items have been pushed in alphabetical order, so "cc" is at the top of the stack:



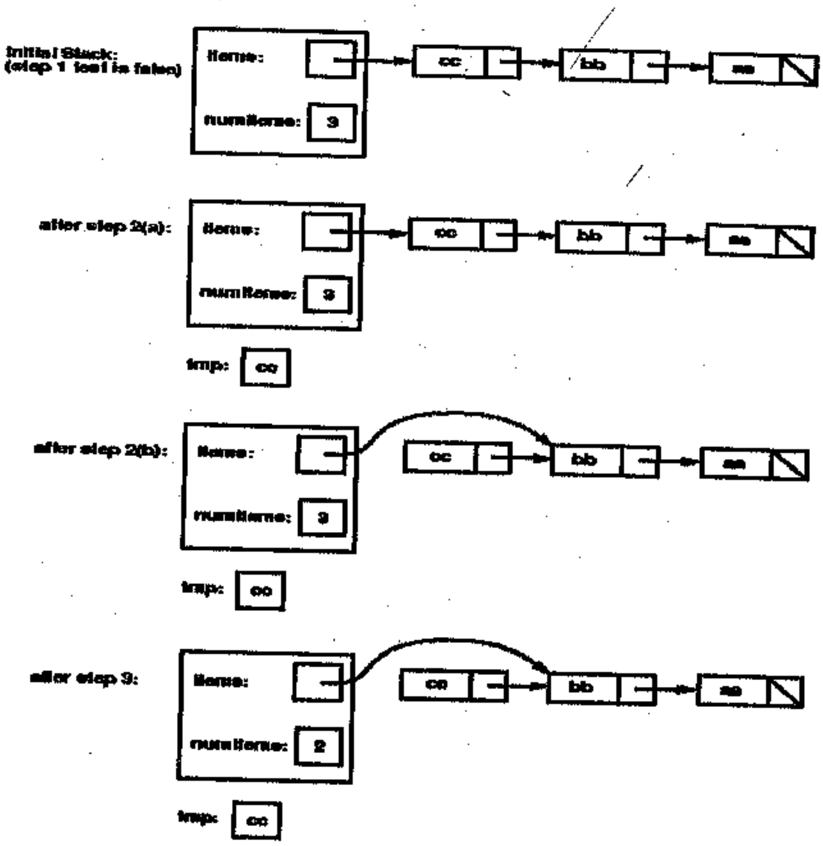
Notice that, in the picture, the top of stack is to the left (at the front of the list), while for the array implementation, the top of stack was to the right (at the end of the array).

Let's consider how to write the pop method. It will need to perform the following steps:

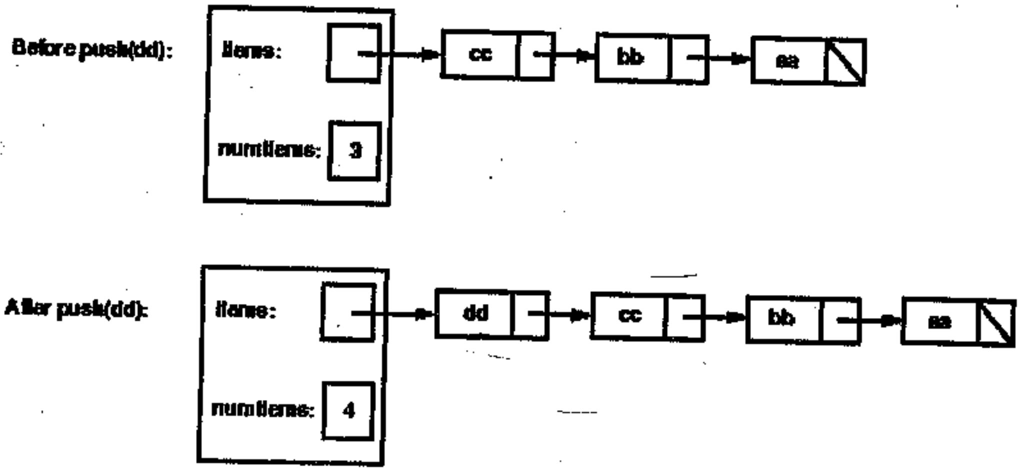
1. Check whether the stack is empty; if so, throw an EmptyStackException.
2. Remove the first node from the list by setting items = items.next.
3. Decrement numItems.
4. Return the value that was in the first node in the list.

Note that by the time we get to the last step (returning the top-of-stack value), the first node has already been removed from the list, so we need to save its value in order to return it (we'll call that step 2(a)). Here's the code, and an illustration of what happens when pop is called for a stack containing "cc", "bb", "aa" (with "cc" at the top).

```
public Object pop() throws EmptyStackException {
  if (empty()) throw new EmptyStackException(); // step 1
  Object tmp = items.getData(); // step 2(a)
  items = items.getNext();     // step 2(b)
  numItems--;                  // step 3
  return tmp;                  // step 4
}
```



Now let's consider the push method. Here are before and after pictures, illustrating the effect of a call to push when the stack is implemented using a linked list:



The steps that need to be performed are:

1. Create a new node whose data field contains the object to be pushed and whose next field contains a pointer to the first node in the list (or null if the list is empty). Note that the value for the next field of the new node is the value in the Stack's items field.

2. Change items to point to the new node.

3. Increment numItems.

//Following is the program to show stack operations push and pop

```
#include<iostream.h>
#include<stdlib.h>
#include<assert.h>
/* To use assertion in the programme */
const int MaxStackSize=100 ;
typedef int StackElemType ;
/* Define a new variable type called StackElemType */
class Stack {
Public :
Stack( ) ;
void Push(StackElemType) ;
StackElemType Pop( ) ;
StackElemType Top( ) ;
int IsEmpty( ) ;
int IsFull( ) ;
Private :
StackElemType StackArray[MaxStackSize] ;
int TopIndex ;
};
Stack::Stack( ) {
TopIndex = -1 ;
}
/* Constructor : done when the class is called in the main */

/* Back to the class */
void Stack::Push(StackElemType elem) {
if (TopIndex < MaxStackSize-1) {
StackArray[++TopIndex] = elem ; }
/* We may use : assert(TopIndex < MaxStackSize-1) in stead of the if condition , where the
programme will continue if it's true , and it will terminate otherwise */
/* We can do the "StackArray[++TopIndex] = elem ; " on two steps :
step one : "++TopIndex ; " note the position of the "++".
step two : "StackArray[TopIndex] = elem ; " */
/* Back to the class */

StackElemType Stack::Pop( ) {
if (TopIndex > -1) {
—TopIndex ;
return StackArray[TopIndex+1] ;
}
}
/* Also "asset(TopIndex > -1 )" can be used instead of the if */
/* Back to the class */

StackElemType Stack::Top( ) {
if (TopIndex > -1)
return StackArray[TopIndex] ;
}
/* Back to the class */

int Stack::IsEmpty( ) {
return TopIndex == -1 ;
}
/* Back to the class */

int Stack::IsFull( ) {
return TopIndex == MaxStackSize - 1 ;
}
/* Back to the class */
```

Stack is known as Pushing and deleting data from the stack is known as Popping.

cpp

```
#include <iostream>
using namespace std;
#define MAX 10       // MAXIMUM STACK CONTENT
class stack
{
 private:
  int arr[MAX];  // Contains all the Data
  int top;     //Contains location of Topmost Data pushed onto Stack
 public:
        stack()      //Constructor
        {
         top=-1;    //Sets the Top Location to -1 indicating an empty stack
        }

        void push(int a) // Push ie. Add Value Function
        {
                top++;      // increment to by 1
                if(top<MAX)
```

```
            {
                    arr[top]=a;  //If Stack is Vacant store Value in Array
            }
            else
            {
                    cout<<“STACK FULL!!”<<endl;
                    top—;
            }
    }

    int pop()           // Delete Item. Returns the deleted item
    {
            if(top==-1)
            {
                    cout<<“STACK IS EMPTY!!!”<<endl;
                    return NULL;
            }
            else
            {
                    int data=arr[top];   //Set Topmost Value in data
                    arr[top]=NULL;       //Set Original Location to NULL
                    top—;              // Decrement top by 1
                    return data;         // Return deleted item
            }
    }
};

int main()
{
 stack a;
 a.push(3);
 cout<<“3 is Pushed\n”;
 a.push(10);
 cout<<“10 is Pushed\n”;
 a.push(1);
 cout<<“1 is Pushed\n\n”;

 cout<<a.pop()<<“ is Popped\n”;
 cout<<a.pop()<<“ is Popped\n”;
 cout<<a.pop()<<“ is Popped\n”;
 return 0;
 }
 OUTPUT:
 3 is Pushed
 10 is Pushed
 1 is Pushed
```

1 is Popped
10 is Popped
3 is Popped

Clearly we can see that the last data pushed is the first one to be popped out. That's why a Stack is also known as a LIFO Data Structure which stands for "Last In,First Out" and I guess you know why.

Let us see how we implemented the stack. We first created a variable called top that points to the top of the stack.

It is initialised to -1 to indicate that the stack is empty. As Data is entered, the value in top increments itself and data is stored into an array arr. Now there's one drawback to this Data Structure. Here we state the Maximum number of elements as 10. What if we need more than 10 Data Elements? In that case we combine a Stack along with a Linked List which will be explained later.

Self learning exercises

1. Convert the following infix expression into prefix and postfix form

(a+b)*c

a+b*c

4.4 Applications of STACK

In this section we will discuss some problems which can be solved using Stack data structure. They are

- Tower of Hanoi
- Paranthesis Checking

The Tower of Hanoi or Towers of Hanoi is a mathematical game or puzzle. It consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

* Only one disk may be moved at a time.

* Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.

* No disk may be placed on top of a smaller disk.

Tower of Hanoi Program

```
#include <iostream.h>
```

```
// a disk with a value , which is an element of the stack ,tower in this case
class Disk
{
public:
int value;
Disk* next;
};//class Disk

class Tower //a stack data structure representing a tower
{
public:
int size;
Disk* current;
Tower()
{
 size=0;
 current=NULL;
}//default constructor

int peep();
bool push(int);
bool pop();
bool isEmpty();
int getTowerSize();
void printTowerSize();
void printTowerDisks();
void printTowerMenu();
};

int Tower::peep()
{
        return this->current->value;
}
bool Tower::push(int ele)
{
Disk*    temp;
temp=new Disk;
```

```
            if(current==NULL)
            {
                        temp->next=NULL;
            }
            else
            {
                        temp->next=current;
            }
            temp->value=ele;
            this->current=temp;
            size++;
    return false;
  }
  bool Tower::pop()
  {
            if(isEmpty())
            {
            cout<<“\nTower is Empty\n”;
            return false;
            }
            else
            {
            current=current->next;
            size=size—;
            }
            return true;
 }
 bool Tower::isEmpty()
 {          if(getTowerSize()==0)
            return true;
            return false;
 }
 int Tower::getTowerSize()
 {
 return size;
```

```
}//returns size of the Tower
void Tower::printTowerSize()
{
cout<<“\nThe Size of the Tower:”<<size<<“\n”;
}//print the Tower size
void Tower::printTowerDisks()
{
          if(this->isEmpty())
          {
          cout<<“——\n”;
          cout<<“ “<<endl;
          cout<<“——\n”;
          return;
          }
   Disk *curr2;
   curr2=this->current ;
                    cout<<“——\n”;
                    cout<<“Tower\n”;
                    cout<<“——\n”;
                    int i=0;
                    while(curr2 !=NULL)
                    {
                              if(i>4)
                              break;
                              i++;
                              cout<<“|”<<curr2->value<<“|\n”;
                              curr2=curr2->next;
                    }
      }// print the Tower
      void createSourceTower(Tower *source,int numberOfDisks)
      {
               for(int i=numberOfDisks;i>0;i—)
               {
         source->push(i);
               }
        }
```

```
void moveDisk(Tower *source,Tower *dest) // movinng a disk from source to destionation
 {
dest->push(source->current->value );
source->pop();
 }
void hanoi( int N, Tower *source, Tower *dest,Tower  *aux ) // move N disks from source to
destination
{
  if (N > 0 )
          {
          hanoi(N - 1, source, aux, dest);     //move n-1 disks from source to auxxilary (sub
problem)
          moveDisk(source,dest);           //move nTH disk from source to destination
          hanoi(N - 1, aux, dest, source);    //move n-1 disks from auxillary to destination (sub
problem)
          }
}
void main()
{
Tower *source,*destination,*auxillary;
//Towers required for the 3 towers source destination and auxillary
source=new Tower;
destination=new Tower;
auxillary=new Tower;

//take number of disks from user
int numberOfDisks;
cout<<"Enter number of Disks in the source Tower";
cin>>numberOfDisks;
//inserting the disks into the source tower
createSourceTower(source,numberOfDisks);
cout<<"=============================================="<<endl;
cout<<"Initial Scenario of the Towers "<<endl;
cout<<"Source"<<endl;
source->printTowerDisks ();
cout<<"Auxillary"<<endl;
auxillary->printTowerDisks ();
```

(See C++ code to do this in Main & Savitch, p. 312.)

What items do we push onto the stack? If we are just interested in knowing if the expression is balanced, it does not matter. For clarity, we might choose a stack of characters, pushing "(" onto the stack for each left parenthesis.

We can do more. We can match multiple types of delimiters. For example, we might want to match (), [], and {}. In that case we can push the left delimiter onto the stack, and, when we pop, do a check, as in the pseudocode:

```
if (next character is a right delimiter) then {
 if (stack is empty) then
  return false
 else {
  pop the stack
  if (right delimiter is corresponding version
    of what was popped off the stack) then
          continue processing
  else
    return false
 }
}
```

Other variations on parentheses matching include:

- valid prefix: return true if the expression is a valid prefix for a balanced parentheses expression, for example "((3+4)+" is a valid prefix. Same idea, but we can return true as long as there is no attempt to pop an empty stack.

- identifying matches: it is often useful to not only know that an expression is balanced, but to see which pair correspond. For example, in some editors (like Emacs), when you type a right paren the corresponding left paren is momentarily highlighted. Think about how you might adapt a paren matching function to do this.

Memory management issues for stacks

Warning: It is undefined in C++ to copy or assign static arrays. For example,

```
int a[20];

int b[20];

...

a = b;
```

is implementation dependent. Some compilers may refuse to compile this. Others may take no action. Still others (at least appear to) copy the elements of b to a. Therefore, whenever objects have static array members (as in our static array implementation of stacks), proper copy constructors and assignment operators should be defined to avoid ambiguity. Such a copy constructor and assignment operator for our static array implementation of stacks can be found here.

For stacks with static arrays, we don't need to worry about the destructor. (The array will be destroyed automatically when the object is). Nor need we worry about destoying items that are

```
cout<<"Destination"<<endl;
destination->printTowerDisks ();
hanoi( numberOfDisks,source, destination, auxillary );
cout<<"=================================================="<<endl;
cout<<"Final Scenario of the Towers "<<endl;
cout<<"Source"<<endl;
source->printTowerDisks();
cout<<"Auxillary"<<endl;
auxillary->printTowerDisks ();
cout<<"Destination"<<endl;
destination->printTowerDisks ();
cout<<"=================================================="<<endl;
}
```

Parentheses Matching

Many applications of stacks deal with determining whether an input string (or file, or sequence of symbols) is a member of a context-free language. (For a formal introduction to context-free languages (CFLs), take Computer Science 520.) Many programming languages are context free. Also many simpler languages.

As an example, consider the language of balanced parentheses. A sequence of symbols involving ()+* and integers, is said to have balanced-parentheses if each right parenthesis has a corresponding left parenthesis that occurs before it. For example: These expressions have balanced parentheses:

```
2*7      // no parens - still balanced
(1+3)
((2*16)+1)*(44+(17+9))
```

These expressions do not:

```
(44+38
)        // a right paren with no left
(55+(12*11) // missing a )
```

How do we tell if a sequence of characters represents a balanced-parentheses expression? Use stacks. Idea:

- Start with an empty stack.
- For each left parenthesis, push.
- For each right parenthesis, pop.
- For each non-parenthesis character, do nothing.
- The expression is balanced if the stack is empty and there are no more characters to process.
- It is not balanced if either after the last character the stack is not empty (too many left parens), or if the stack is empty and a right paren is encountered (a right without a left).

poppoed off the stack. Consider the code for push and pop:

```
template<class Item, size_t cap>
void Stack<Item,cap>::push(const Item& it)
{
 assert(!isFull());

 data[count] = it;
 count++;
}
template<class Item, size_t cap>
Item Stack<Item,cap>::pop()
{
 assert(!isEmpty());

 count—;
 return(data[count]);
}
```

The space occupied by data[count] that is no longer needed after pop returns that item, will be reused by the next assignment (i.e. the next push). Assuming that Item has a properly functioning copy constructor, assignent operator and destructor, then

data[count]=it

invokes Item's assignment operator so memory management for the Item is taken care of there. Our documentation states (effectively a precondition for the whole class template) that the "template parameter, Item, is the data type of the items in the Stack. It may be any of the C++ built-in types (int, char, etc.), or a class with a copy constructor, assignment operator, and destructor." Unlike other forms of preconditions, we cannot check this at compile time or run-time. In order for a Stack class to be implemented properly we have to trust that the Item class has been implemented properly as well.

For a stack class template that uses linked lists, we need to write an explicit copy constructor, assignment operator, and destructor because memory is allocated dynamically. However, if we reuse the List template, these become trivial, because all the dynamic memory allocation takes place within that template's implementation (not the stack implementation). The copy constructor, assignment operator, and destructor for stack will automatically call the corresponding member function for the stack's sole member variable, a list.

Run-time complexity of stack operations

For all the standard stack operations (push, pop, isEmpty, size), the worst-case run-time complexity can be O(1). We say can and not is because it is always possible to implement stacks with an underlying representation that is inefficient. However, with the representations we have looked at (static array and a reasonable linked list) these operations take constant time. It's obvious that size and isEmpty constant-time operations. push and pop are also O(1) because they only work with one end of the data structure - the top of the stack. The upshot of all this is that stacks can and should be implemented easily and efficiently.

The copy constructor and assignment operator are O(n), where n is the number of items on the stack. This is clear because each item has to be copied (and copying one item takes constant time). The destructor takes linear time (O(n)) when linked lists are used - the underlying list has to be traversed and each item released (releasing the memory of each item is constant in terms of the number of items on the whole list). Destructors for arrays take constant time since contiguous chunks of memory can be freed in one fell swoop.

Other Stack Applications

- Calculators - see Main and Savitch, Chapter Seven

o Infix arithmetic expressions can be computed using two stacks:

o ((2+3)*7)) evaluates to 35

o Polish postfix using one stack:

o 2 3 + 7 * evaluates to 35

Push each number onto the stack; for each operator encountered, pop two numbers off the stack, push the result back on.

o Infix expressions can be converted to Polish postfix (or prefix) using one stack:

o ((2+3)*7) becomes 2 3 + 7 *

- Backtracking - stacks are particularly useful when a computation has to go back in reverse order. This happens often in artificial intelligence applications: games, logic programs, theorem provers, etc. Think of walking through a maze. Whenever you have options to move in more than one direction, push all but one of the options onto the stack and then go in the direction you didn't push. When you run into a dead end, walk backwards to your last option (i.e. pop the stack) and proceed from there.

4.5 Summary

- Stack is a LIFO data structure
- Stack can be implemented as static array or dynamic linked list
- Stack has many applications eg. tower of hanoi problem, palindrome checking, parenthesis checking etc.

UNIT – V

QUEUE DATA STRUCTURE

STRUCTURE OF THE UNIT

5.0 Objective
5.1 Introduction
5.2 Operations on Queue
5.3 Implementation of Queue
  5.3.1 Static implementation of Queue
  5.3.2 Dynamic implementation of Queue
5.4 Other Types of Queue
  5.4.1 Circular Queue
  5.4.2 Deque
  5.4.3 Priority Queue
5.5 Applications of Queue
5.6 Summary
5.7 Glossary
5.8 Further Readings
5.9 Answers to the Self learning exercises
5.10 Unit -End Questions

5.0 Objective

After reding this module you will be able to understand

- What is Queue Data Structure
- Operations on Queue
- Static & Dynamic implementation of Queue
- Applications of Queue

5.1 Introduction

The **queue** data structure is characterised by the fact that additions are made at the end, or *tail*, of the queue while removals are made from the front, or *head*, of the queue. For this reason, a stack is referred to as a FIFO structure (First-In First-Out).

Queues occur naturally in situations where the rate at which clients' demand for services can exceed the rate at which these services can be supplied. For example, in a network where many computers share only a few printers, the print jobs may accumulate in a print queue. In an operating system with a GUI, applications and windows communicate using messages, which are placed in message queues until they can be handled.

5.2 Operations on Queue

The main primitive operations of a queue are known as:

Add adds a new node

Remove removes a node

Additional primitives can be defined:

IsEmpty reports whether the queueis empty

IsFull reports whether the queue is full

Initialise creates/initialises the queue

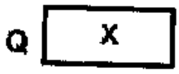
Destroy deletes the contents of the queue (may be implemented by re-initialising the queue)

Initialise Creates the structure – i.e. ensures that the structure exists but contains no elements

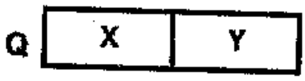
e.g. ***Initialise(Q)*** creates a new empty queue named Q

Add

e.g. ***Add(X,Q)*** adds the value X to the tail of Q.

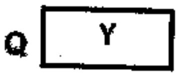


then, ***Add(Y,Q)*** adds the value Y to the tail of Q.



Remove

e.g. ***Remove(Q)*** removes the head node and returns its value



Example

Action Contents of queue Q after operation Return value

Initialise(Q) empty

Add(A,Q) A -

Add(B,Q) A B -

Add(C,Q) A B C -

Remove(Q) B C A

Add(F,Q) B C F -

Remove(Q) C F B

Remove(Q) F C

Remove(Q) empty F

Storing a queue in a static data structure

This implementation stores the queue in an array. The array indices at which the head and tail of the queue are currently stored must be maintained. The head of the queue is not necessarily at index 0. The array can be a "circular array" – the queue "wraps round" if the last index of the array is reached.

Example – storing a queue in an array of length 5

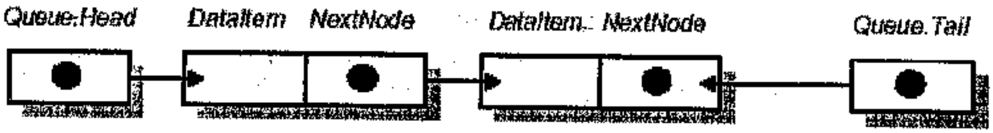
Add(A,Q)	A					*Head:* 0 *Tail:* 0
Add(D,Q)	A	D				*Head:* 0 *Tail:* 1
Add(Z,Q)	A	D	Z			*Head:* 0 *Tail:* 2
RemoveQ)		D	Z			*Head:* 1 *Tail:* 2
Add(X,Q)		D	Z	X		*Head:* 1 *Tail:* 3
Add(C,Q)		D	Z	X	C	*Head:* 1 *Tail:* 4
Remove(Q)			Z	X	C	*Head:* 2 *Tail:* 4
Add(F,Q)	F		Z	X	C	*Head:* 2 *Tail:* 0
Remove(Q)	F			X	C	*Head:* 3 *Tail:* 0

Storing a queue in a dynamic data structure

As in the case of the stack, each node in a dynamic data structure contains data AND a reference to the next node.

A queue also needs a reference to the head node AND a reference to the tail node.

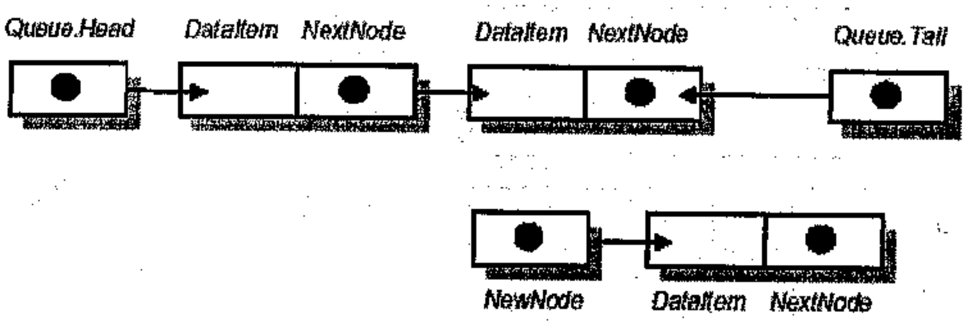
The following diagram describes the storage of a queue called *Queue*. Each node consists of data (*DataItem*) and a reference (*NextNode*).

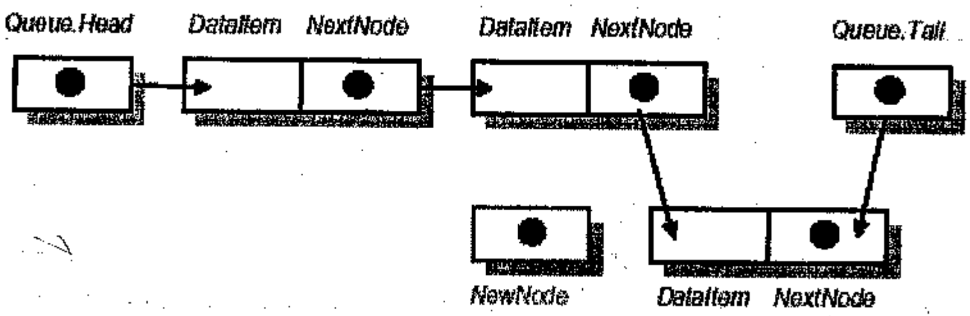


- The first node is accessed using the name ***Queue.Head.***
- Its data is accessed using ***Queue.Head.DataItem***
- The second node is accessed using ***Queue.Head.NextNode***
- The last node is accessed using ***Queue.Tail***

Adding a node (Add)

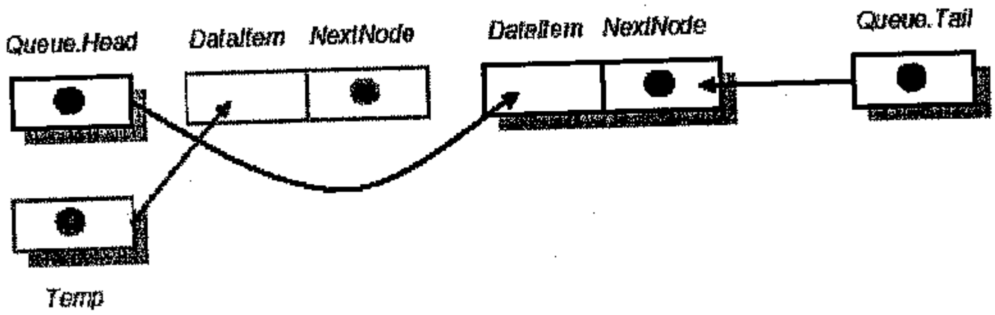
The new node is to be added at the tail of the queue. The reference *Queue.Tail* should point to the new node, and the *NextNode* reference of the node previously at the tail of the queue should point to the *DataItem* of the new node.





Removing a node (Remove)

The value of *Queue.Head.DataItem* is returned. A temporary reference *Temp* is declared and set to point to head node in the queue (*Temp* = *Queue.Head*). *Queue.Head* is then set to point to the second node instead of the top node.The only reference to the original head node is now *Temp* and the memory used by this node can then be freed.



```
#include <iostream>
using namespace std;
#define MAX 5        // MAXIMUM CONTENTS IN QUEUE
class queue
{
 private:
        int t[MAX];
        int al;    // Addition End
        int dl;    // Deletion End
 public:
  queue()
  {
        dl=-1;
        al=-1;
  }
  void del()
  {
   int tmp;
   if(dl==-1)
    {
        cout<<"Queue is Empty";
    }
    else
    {
        for(int j=0;j<=al;j++)
                {
                        if((j+1)<=al)
                        {
```

```
                        tmp=t[j+1];
                        t[j]=tmp;
                }
                else
                {
                        al—;

                if(al==-1)
                        dl=-1;
                else
                        dl=0;
                }
            }
    }
  }

void add(int item)
{
        if(dl==-1 && al==-1)
        {
                dl++;
                al++;
        }
 else
 {
                al++;
                if(al==MAX)
        {
                        cout<<"Queue is Full\n";
                        al—;
                        return;
                }
        }
        t[al]=item;
}
 void display()
 {
        if(dl!=-1)
   {
        for(int iter=0 ; iter<=al ; iter++)
        cout<<t[iter]<<“ “;
   }
   else
        cout<<“EMPTY”;
   }
};
int main()
{
 queue a;
 int data[5]={32,23,45,99,24};
 cout<<“Queue before adding Elements: “;

 a.display();
 cout<<endl<<endl;
 for(int iter = 0 ; iter < 5 ; iter++)
  {
   a.add(data[iter]);
   cout<<“Addition Number : “<<(iter+1)<<“ : “;
   a.display();
   cout<<endl;
  }
  cout<<endl;
  cout<<“Queue after adding Elements: “;
  a.display();
  cout<<endl<<endl;

  for(iter=0 ; iter < 5 ; iter++)
  {
   a.del();
   cout<<“Deletion Number : “<<(iter+1)<<“ : “;
   a.display();
   cout<<endl;
  }
  return 0;
```

```
}
```

OUTPUT:

```
Queue before adding Elements: EMPTY

Addition Number : 1 : 32
Addition Number : 2 : 32 23
Addition Number : 3 : 32 23 45
Addition Number : 4 : 32 23 45 99
Addition Number : 5 : 32 23 45 99 24

Queue after adding Elements: 32 23 45 99 24

Deletion Number : 1 : 23 45 99 24
Deletion Number : 2 : 45 99 24
Deletion Number : 3 : 99 24
Deletion Number : 4 : 24
Deletion Number : 5 : EMPTY
```

5.3 Implementation of Queue

As discussed before also like Stack, queue can also be implemented in two ways wiz. static and dynamic which are discussed below:

5.3.1 Static implementation of Queue

Simple array implementation of enqueue and dequeue operations

Analysis:

Consider the following structure: int Num[MAX_SIZE];

We need to have two integer variables that tell:

- the index of the front element
- the index of the rear element

We also need an integer variable that tells:

- the total number of data in the queue

```
int Front =-1,Rear =-1;
int QueueSize=0;
```

- To enqueue data to the queue

o check if there is space in the queue

```
Rear<MAX_SIZE-1 ?
Yes: - Increment Rear
     - Store the data in Num[Rear]
     - Increment QueueSize
       Front = = -1?
```

```
        Yes: - Increment Front
No:  - Queue Overflow
•     To dequeue data from the queue
o     check if there is data in the queue
QueueSize > 0 ?
Yes: - Copy the data in Num[Front]
     - Increment Front
     - Decrement QueueSize
No:  - Queue Underflow
```

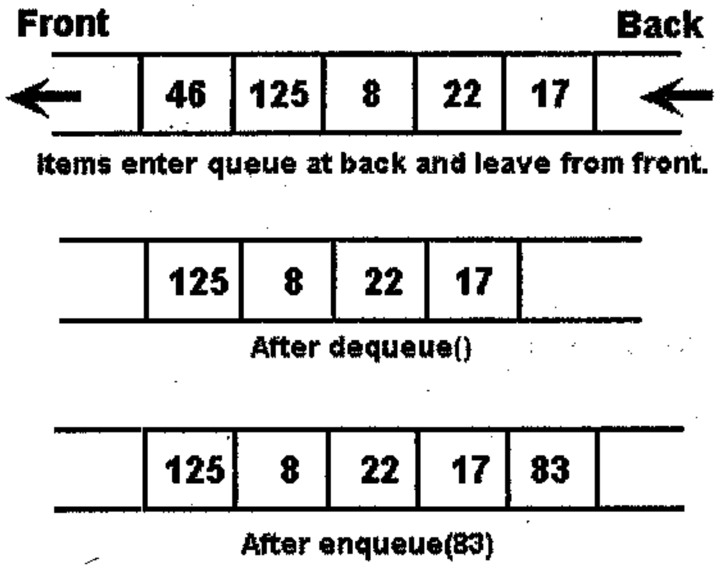
Implementation:

```
const int MAX_SIZE=100;
int Front =-1, Rear =-1;
int QueueSize = 0;

    void enqueue(int x)
    {
        if(Rear<MAX_SIZE-1)
        {
            Rear++;
            Num[Rear]=x;
            QueueSize++;
            if(Front = = -1)
                Front++;
        }
        else
            cout<<"Queue Overflow";
    }
    int dequeue()
    {
        int x;
        if(QueueSize>0)
        {
            x=Num[Front];
            Front++;
            QueueSize—;
        }
        else
            cout<<"Queue Underflow";
        return(x);
    }
```

5.3.2 Dynamic implenentation of Queue

Queues are similar to stacks in that a queue consists of a sequence of items, and there are restrictions about how items can be added to and removed from the list. However, a queue has two ends, called the front and the back of the queue. Items are always added to the queue at the back and removed from the queue at the front. The operations of adding and removing items are called enqueue and dequeue. An item that is added to the back of the queue will remain on the queue until all the items in front of it have been removed. This should sound familiar. A queue is like a "line" or "queue" of customers waiting for service. Customers are serviced in the order in which they arrive on the queue.

In a queue, all operations take place at one end of the queue or the other. The "enqueue" operation adds an item to the "back" of the queue. The "dequeue" operation removes the item at the "front" of the queue and returns it.



Items enter queue at back and leave from front.

After dequeue()

After enqueue(83)

A queue can hold items of any type. For a queue of ints, the enqueue and dequeue operations can be implemented as instance methods in a "QueueOfInts" class. We also need an instance method for checking whether the queue is empty:

void enqueue(int N) -- Add N to the back of the queue.

int dequeue() -- Remove the item at the front and return it.

boolean isEmpty() -- Return true if the queue is empty.

A queue can be implemented as a linked list or as an array. An efficient array implementation is a little trickier than the array implementation of a stack, so I won't give it here. In the linked list implementation, the first item of the list is the front of the queue. Dequeueing an item from the front of the queue is just like popping an item off a stack. The back of the queue is at the end of the list. Enqueueing an item involves setting a pointer in the last node on the current list to point to a new node that contains the item. To do this, we'll need a command like "tail.next = newNode;", where tail is a pointer to the last node in the list. If head is a pointer to the first node of the list, it would always be possible to get a pointer to the last node of the list by saying:

```
Node tail;   // This will point to the last node in the list.
 tail = head;  // Start at the first node.
while (tail.next != null) {
   tail = tail.next;
  }
// At this point, tail.next is null, so tail points to
 // the last node in the list.
```

However, it would be very inefficient to do this over and over every time an item is enqueued. For the sake of efficiency, we'll keep a pointer to the last node in an instance variable. We just have to be careful to update the value of this variable whenever a new node is added to the end of the list. Given all this, writing the QueueOfInts class is not very difficult:

```
public class QueueOfInts {

  private static class Node {
      // An object of type Node holds one of the items
     // in the linked list that represents the queue.
   int item;
    Node next;
  }

 private Node head = null;  // Points to first Node in the queue.
                  // The queue is empty when head is null.
```

```
private Node tail = null;  // Points to last Node in the queue.

void enqueue( int N ) {
      // Add N to the back of the queue.
   Node newTail = new Node();  // A Node to hold the new item.
   newTail.item = N;
   if (head == null) {
        // The queue was empty.  The new Node becomes
        // the only node in the list.  Since it is both
        // the first and last node, both head and tail
        // point to it.
      head = newTail;
      tail = newTail;
   }
   else {
        // The new node becomes the new tail of the list.
        // (The head of the list is unaffected.)
      tail.next = newTail;
      tail = newTail;
   }
}

int dequeue() {
      // Remove and return the front item in the queue.
      // Note that this can throw a NullPointerException.
   int firstItem = head.item;
   head = head.next;  // The previous second item is now first.
   if (head == null) {
        // The queue has become empty.  The Node that was
        // deleted was the tail as well as the head of the
        // list, so now there is no tail.  (Actually, the
        // class would work fine without this step.)
      tail = null;
   }
   return firstItem;
}
```

```
boolean isEmpty() {
    // Return true if the queue is empty.
  return (head == null);
  }

} // end class QueueOfInts
```

Comparison of Array and Linked-List Implementations

The advantages and disadvantages of the two implementations are essentially the same as the advantages and disadvantages in the case of the List class:

* In the linked-list implementation, one pointer must be stored for every item in the stack/queue, while the array stores only the items themselves.

* On the other hand, the space used for a linked list is always proportional to the number of items in the list. This is not necessarily true for the array implementation as described: if a lot of items are added to a stack/queue and then removed, the size of the array can be arbitrarily greater than the number of items in the stack/queue. However, we could fix this problem by modifying the pop/dequeue operations to shrink the array when it becomes too empty.

* For the array implementation, the worst-case times for the push and enqueue methods are O(N), for a stack/queue with N items. This is because when the arary is full, a new array must be allocated and the values must be copied. For the linked-list implementation, push and enqueue are always O(1).

5.4 Other types of Queue

In this section we will discuss diffferent types of queue, namely: circular queue, dequeue and priority queue

5.4.1. Circular array implementation of enqueue and dequeue operations

A problem with simple arrays is we run out of space even if the queue never reaches the size of the array. Thus, simulated circular arrays (in which freed spaces are re-used to store data) can be used to solve this problem.

Example: Consider a queue with MAX_SIZE = 4

The circular array implementation of a queue with MAX_SIZE can be simulated as follows:

Analysis:

Consider the following structure: int Num[MAX_SIZE];

We need to have two integer variables that tell:

- the index of the front element
- the index of the rear element

We also need an integer variable that tells:

- the total number of data in the queue

```
int Front =-1,Rear =-1;
int QueueSize=0;
```

- To enqueue data to the queue
    - check if there is space in the queue

```
QueueSize<MAX_SIZE ?
Yes:  - Increment Rear
                    Rear = = MAX_SIZE ?
                         Yes:   Rear = 0
      - Store the data in Num[Rear]
      - Increment QueueSize
        Front = = -1?
                    Yes: - Increment Front
No:   - Queue Overflow
```

- To dequeue data from the queue
    - check if there is data in the queue

```
QueueSize > 0 ?
Yes:  - Copy the data in Num[Front]
      - Increment Front
                    Front = = MAX_SIZE ?
                         Yes:   Front = 0
      - Decrement QueueSize
No:   - Queue Underflow
```

Implementation:

```
const int MAX_SIZE=100;
int Front =-1, Rear =-1;
int QueueSize = 0;

    void enqueue(int x)
    {
        if(QueueSize<MAX_SIZE)
        {
            Rear++;
            if(Rear = = MAX_SIZE)
                    Rear=0;
            Num[Rear]=x;
            QueueSize++;
            if(Front = = -1)
```

```
                Front++;
        }
        else
            cout<<“Queue Overflow”;
    }
    int dequeue()
    {
        int x;
        if(QueueSize>0)
        {
            x=Num[Front];
            Front++;
            if(Front = = MAX_SIZE)
                Front = 0;
            QueueSize—;
        }
        else
            cout<<“Queue Underflow”;
        return(x);
    }
```

5.4.2. Deque (pronounced as Deck)

- is a Double Ended Queue
- insertion and deletion can occur at either end
- has the following basic operations

    EnqueueFront – inserts data at the front of the list

    DequeueFront – deletes data at the front of the list

    EnqueueRear – inserts data at the end of the list

    DequeueRear – deletes data at the end of the list

- implementation is similar to that of queue
- is best implemented using doubly linked list

5.4.3. Priority Queue

- is a queue where each data has an associated key that is provided at the time of insertion.
- Dequeue operation deletes data having highest priority in the list
- One of the previously used dequeue or enqueue operations has to be modified

Example: Consider the following queue of persons where females have higher priority than males (gender is the key to give priority).

Dequeue()- deletes Aster

Dequeue()- deletes Meron

Now the queue has data having equal priority and dequeue operation deletes the front element like in the case of ordinary queues.

Dequeue()- deletes Abebe

Dequeue()- deletes Alemu

Thus, in the above example the implementation of the dequeue operation need to be modified.

Demerging Queues

- is the process of creating two or more queues from a single queue.

- used to give priority for some groups of data

Example: The following two queues can be created from the above priority queue.

Algorithm:

```
        create empty females and males queue
        while (PriorityQueue is not empty)
        {
                Data=DequeuePriorityQueue(); // delete data at the front
                if(gender of Data is Female)
                        EnqueueFemale(Data);
                else
                        EnqueueMale(Data);
}
```

Merging Queues

- is the process of creating a priority queue from two or more queues.

- the ordinary dequeue implementation can be used to delete data in the newly created priority queue.

Example: The following two queues (females queue has higher priority than the males queue) can be merged to create a priority queue.

Algorithm:

```
        create an empty priority queue
        while(FemalesQueue is not empty)
                EnqueuePriorityQueue(DequeueFemalesQueue());
        while(MalesQueue is not empty)
                EnqueuePriorityQueue(DequeueMalesQueue());
```

It is also possible to merge two or more priority queues.

Example: Consider the following priority queues and suppose large numbers represent high priorities.

Thus, the two queues can be merged to give the following priority queue.

Self Learning Exercises

State True or False

1. Queue is a First In First Out Data Structure.

2. Circular Queue and priority queue are same.

3. Insertion can be done at both the ends but deletion can be done from one end only in deque.

4. Queue is used by Operating System.

5. Queue is a linear data structure.

5.5 Applications of Queues

i. Print server- maintains a queue of print jobs

```
Print()
{
       EnqueuePrintQueue(Document)
}
EndOfPrint()
{
       DequeuePrintQueue()
}
```

i Disk Driver- maintains a queue of disk input/output requests

ii Task scheduler in multiprocessing system- maintains priority queues of processes

iii Telephone calls in a busy environment --maintains a queue of telephone calls

iv Simulation of waiting line- maintains a queue of persons

One application of the queue data structure is in the implementation of priority queues required to be maintained by the scheduler of an operating system. It is a queue in which each element has a priority value and the elements are required to be inserted in the queue in decreasing order of priority. This requires a change in the function that is used for insertion of an element into the queue. No change is required in the delete function.

Program

A complete C program implementing a priority queue is shown here:

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
  int data;
  int priority;
  struct node *link;
```

```
};
void insert(struct node **front, struct node **rear, int value, int priority)
{
 struct node *temp,*temp1;
 temp=(struct node *)malloc(sizeof(struct node));
   /* creates new node using data value
  passed as parameter */
 if(temp==NULL)
  {
   printf(“No Memory available Error\n”);
    exit(0);
  }
  temp->data = value;
  temp->priority = priority;
  temp->link=NULL;
  if(*rear == NULL) /* This is the first node */
  {
    *rear = temp;
    *front = *rear;
  }
  else
  {
   if((*front)->priority < priority)
        /* the element to be inserted has
   highest priority hence should
        be the first element*/
    {
       temp->link = *front;
        *front = temp;
    }
    else
      if( (*rear)->priority > priority)
           /* the element to be inserted has
           lowest priority hence should
            be the last element*/
        {
            (*rear)->link = temp;
             *rear = temp;
         }
  else
    {
            temp1 = *front;
          while((temp1->link)->priority >= priority)
   /* find the position and insert the new element */
                temp1=temp1->link;
        temp->link = temp1->link;
         temp1->link = temp;
          }
}
void delete(struct node **front, struct node **rear, int *value, int *priority)
{
  struct node *temp;
  if((*front == *rear) && (*rear == NULL))
   {
    printf(" The queue is empty cannot delete Error\n");
     exit(0);
   }
  *value = (*front)->data;
  *priority = (*front)->priority;
  temp = *front;
  *front = (*front)->link;
  if(*rear == temp)
     *rear = (*rear)->link;
  free(temp);
}
void main()
{
  struct node *front=NULL,*rear = NULL;
  int n,value, priority;
   do
   {
   do
```

```
{
   printf("Enter the element to be inserted and its priority\n");
 scanf("%d %d",&value,&priority);
 insert(&front,&rear,value,priority);
 printf("Enter 1 to continue\n");
 scanf("%d",&n);
} while(n == 1);
printf("Enter 1 to delete an element\n");
scanf("%d",&n);
while( n == 1)
 {
    delete(&front,&rear,&value,&priority);
    printf("The value deleted is %d\ and its priority is %d \n",
         value,priority);
    printf("Enter 1 to delete an element\n");
  scanf("%d",&n);
  }
 printf("Enter 1 to delete an element\n");
 scanf("%d",&n);
 } while( n == 1)
}
```

Example

Input and Output

```
Enter the element to be inserted and its priority
10 90
Enter 1 to continue
1
Enter the element to be inserted and its priority
5 8
Enter 1 to continue
1
Enter the element to be inserted and its priority
11 60
Enter 1 to continue
1
Enter the element to be inserted and its priority
```

```
12 75
Enter 1 to continue
1
Enter the element to be inserted and its priority
13 10
Enter 1 to continue
1
Enter the element to be inserted and its priority
14 6
Enter 1 to continue
0
Enter 1 to delete an element
1
The value deleted is 10 and its priority is 90
Enter 1 to delete an element
1
The value deleted is 12 and its priority is 75
Enter 1 to delete an element
1
The value deleted is 11 and its priority is 60 Enter 1 to delete an element
1
The value deleted is 13 and its priority is 10 Enter 1 to delete an element
1
The value deleted is 5 and its priority is 8
Enter 1 to delete an element
1
The value deleted is 14 and its priority is 6
Enter 1 to delete an element
1
The queue is empty cannot delete Error
```

5.6 Summary

- A queue is also a list with insertions permitted from one end, called rear, and deletions permitted from the other end, called front. So it is a data structure that exhibits the FIFO property.
- The operations that are permitted on a queue are insert and delete.
- A circular queue is a queue in which the element next to the last element is the first element.

- When the size of the stack/queue is known beforehand, the array implementation can be used and is more efficient.

- When the size of the stack/queue is not known beforehand, then the linked representation is used. It provides more flexibility.

5.7 Glossary

Bounded Queue: An implementation of a bounded queue using an array.

Circular queue : An implementation of a bounded queue using an array.

Dequeue : A data structure in which items may be added to or deleted from the head or the tail.Also known as doubly-ended queue.

Priority Queue ; An abstract data type to efficiently support finding the item with the highest priority across a series of operations. The basic operations are: insert, find-minimum (or maximum), and delete-minimum (or maximum). Some implementations also efficiently support join two priority queues (meld), delete an arbitrary item, and increase the priority of a item (decrease-key).

Queue: A collection of items in which only the earliest added item may be accessed. Basic operations are add (to the tail) or enqueue and delete (from the head) or dequeue. Delete returns the item removed. Also known as "first-in, first-out" or FIFO.

5.8 Further Readings

1. Data structures, Algorithms and Applications in C++ by S.Sahni, University press (India) pvt ltd / Orient Longman pvt.ltd., 2nd edition

2. Data Structures and Algorithm Analysis in C++ by Mark Allen Weiss, Pearson Education, Second Edition

3. Data structures and Algorithms in C++ by Michael T.Goodrich, R.Tamassia and D Mount, Wiley Student Edition, John Wiley and Sons

4. Data structures using C and C++ by Langsam, Augenstein and Tanenbaum, PHI/Pearson Education.

5. Data Structures and Algorithms in C++ by Adam Drozdek, Vikas Publishing House / Thomson International Student Edition., Second Edition

6. Donald Knuth. The Art of Computer Programming, Volume 1: Fundamental Algorithms, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4. Section 2.2.1: Stacks, Queues, and Deques, pp. 238–243.

5.9 Answers to self learning exercises

1.True

2. False

3. False

4. True

5. True

5.10 Unit End Questions.

2. What is Queue? Explain basic operations on Queue.

3. Describe different applications of Queue

4. Compare static and dynamic implementation of Queue.

5. What are the problems of queue which are solved by circular queue.

6. Compare different types of queue.

7. What would the state of a queue be after the following operations:

create queue

Insert A onto queue

Insert F onto queue

Insert X onto queue

delete item from queue

Insert B onto queue

delete item from queue

delete item from queue

8. . Write a C program to implement a double-ended queue, which is a queue in which insertions and deletions may be performed at either end. Use a linked representation.

____*****_____

UNIT VI

Tree Data Structure

Structure of the Unit

6.0 Objective

6.1 Introduction

6.2 Introduction to Tree

6.2.1 Binary tree

6.3 Binary Tree Traversal

6.3.1 Inorder Traversal

6.3.2 Preorder traversal

6.3.3 Post order traversal

6.4 Operations on Binary Tree

6.4.1 Searching

6.4.2 Insertion

6.4.3 Deletion

6.4.4 Sort

6.5 Applications of Tree

6.5.1 Arithmetic expressions evaluation.

6.6 Some Operations on Tree

6.6.1 Counting the no of nodes in a Binary tree

6.6.2 Swaping of left and right subtree of a given Binary tree

6.6.3 Searching For a target key in a Binary search tree

6.7 Summary

6.8 Glossary

6.9 Further Readings

6.10 Answers to self learning exercises

6.11 Unit End Questions

6.0 Objectives

After completing this unit you will learn

- About Tree data structure, types of tree
- Binary tree, Binary tree search
- Traversal in tree, preorder ,I norder, postorder
- Application of tree etc.

6.1 Introduction

Tree structures support various basic dynamic set operations including Search, Predecessor, Successor, Minimum, Maximum, Insert, and Delete in time proportional to the height of the tree. Ideally, a tree will be balanced and the height will be log n where n is the number of nodes in the tree. To ensure that the height of the tree is as small as possible and therefore provide the best running time, a balanced tree structure like a red-black tree, AVL tree, or b-tree must be used.

When working with large sets of data, it is often not possible or desirable to maintain the entire structure in primary storage (RAM). Instead, a relatively small portion of the data structure is maintained in primary storage, and additional data is read from secondary storage as needed. Unfortunately, a magnetic disk, the most common form of secondary storage, is significantly slower than random access memory (RAM). In fact, the system often spends more time retrieving data than actually processing data. In such cases primary key of the data can be stored in tree structure (in RAM) and for accessing, insertion, deletion etc. tree data structure can be used.

6.2 Introduction to Tree

A tree is a finite set of nodes having a distinct node called root and all other nodes again form the tree.Binary Tree is a tree which is either empty or has at most two subtrees, each of the subtrees also being a binary tree. It means each node in a binary tree can have 0, 1 or 2 subtrees. A left or right subtree can be empty.

Types of binary tree

A **binary tree** is a **rooted** tree in which every node has at most two children.

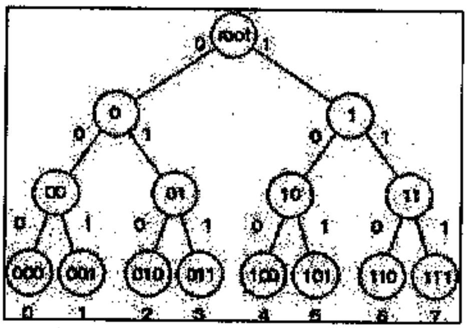
A **full binary tree** is a tree in which every node has zero or two children..

A **perfect binary tree** is a complete binary tree in which **leaves** (vertices with zero children) are at the same **depth** (distance from the **root**, also called **height**).

Sometimes the **perfect binary tree** is called the **complete binary tree**. Some others define a **complete binary tree** to be a full binary tree in which all leaves are at depth n or n-1 for some n.

A binary tree is made of nodes, where each node contains a "left" pointer, a "right" pointer, and a data element. The "root" pointer points to the topmost node in the tree. The left and right pointers point to smaller "subtrees" on either side. A null pointer represents a binary tree with no elements — the empty tree. The formal recursive definition is: a binary tree is either empty (represented by a null pointer), or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a binary tree.

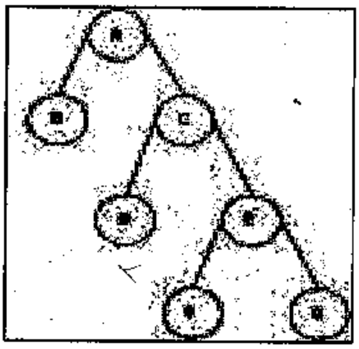
The figure shown below is a binary tree.



It has a distinct node called root which has no parent and and every other node has either0,1 or 2 children. So it is a binary tree as every node has a maximum of 2 children.

If A is the root of a binary tree and B the root of its left or right subtree, then A is the parent or father of B and B is the left or right child of A. Those nodes having no children are leaf nodes. Any node say A is the ancestor of node B and B is the descendant of A if A is either the father of B or the father of some ancestor of B. Two nodes having same father are called brothers or siblings. Going from leaves to root is called climbing the tree & going from root to leaves is called descending the tree.

A binary tree in which every non leaf node has non empty left & right subtrees is called a strictly binary tree. The tree shown below is a strictly binary tree.



The no. of children a node has is called its degree. The level of root is 0 & the level of any node is one more than its father. In the strictly binary tree shown above A is at level 0, B & C at level 1, D & E at level 2 & F & g at level 3.

The depth of a binary tree is the length of the longest path from the root to any leaf. In the above tree, depth is 3.

The other topics that will be covered regarding binary tree are listed below.

- Representation of binary tree
- Operations on a binary tree
- Traversal of a binary tree

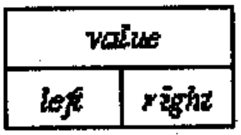
6.2.1 Binary tree

Representation of binary tree

The structure of each node of a binary tree contains one data field and two pointers, each for the right & left child. Each child being a node has also the same structure.

The structure of a node is shown below.

The structure defining a node of binary tree in C is as follows.



```
Struct node
{
struct node *lc ; /* points to the left child */
int data; /* data field */
struct node *rc; /* points to the right child */
}
```

There are two ways for representation of binary tree.

§ Array representation of a Binary tree

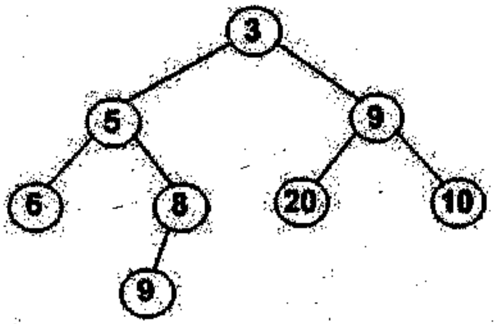
§ Linked List representation of a Binary tree

Array representation of binary tree

A single array can be used to represent a binary tree.

For these nodes are numbered / indexed according to a scheme giving 0 to root. Then all the nodes are numbered from left to right level by level from top to bottom. Empty nodes are also numbered. Then each node having an index i is put into the array as its ith element.

In the figure shown below the nodes of binary tree are numbered according to the given scheme.



3	5	9	6	8	20	10	\0	\0	9	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	......				20

The figure shows how a binary tree is represented as an array. The root 3 is the 0 th element while its leftchild 5 is the 1 st element of the array. Node 6 does not have any child so its children ie 7 th & 8 th element of the array are shown as a Null value.

It is found that if n is the number or index of a node, then its left child occurs at (2n + 1)th position & right child at (2n + 2) th position of the array. If any node does not have any of its child, then null value is stored at the corresponding index of the array.

The following program implements the above binary tree in an array form. And then traverses the tree in inorder traversal.(for traversal see Traversal, Inorder traversal)

```
Struct node
{
struct node * lc;
int data;
struct node * rc;
};
struct node * buildtree(int);/* builds the tree*/
void inorder(struct node *);/* Traverses the tree in inorder*/
int
a[]={ 3,5,9,6,8,20,10,/0,/0,9,/0,/0,/0,/0,/0,/0,/0,/0,/0,/0,/0};

void main( )
{
struct node * root;
root= buildtree(0);
printf(?\n Inorder Traversal?);
inorder(root);
getch( );
}

struct node * buildtree(int n);
{
struct node * temp=NULL;
if( a[n] != NULL)
{
temp = (struct node *) malloc(sizeof(struct node));
temp-> lc=buildtree(2n + 1);
temp-> data= a[n];
temp-> rc=buildtree(2n + 2);
}
return temp;
}

void inorder(struct node * root);
{
if(root != NULL)
{
if(root!= NULL)
{
inorder(roo-> lc);
printf(?%d\t?,root->data);
inorder(root->rc);
}
}
```

Linked List represe~tation of a Binary tree

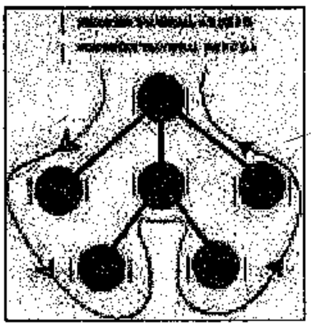
Binary trees can be represented by links where each node contains the address of the left child and the right child. If any node has its left or right child empty then it will have in its respective link field, a null value. A leaf node has null value in both of its links.

The structure defining a node of binary tree in C is as follows.

```
Struct node

{

struct node *lc ; /* points to the left child */

int data; /* data field */

struct node *rc; /* points to the right child */

}
```

6.3 Binary Tree Traversal

Traversal of a binary tree means to visit each node in the tree exactly once. The tree traversal is used in all the applications of it.



In a linear list nodes are visited from first to last, but a tree being a non linear one we need definite rules. There are different ways to traverse a tree. All of them differ only in the order in which they visit the nodes.

The three main methods of traversing a tree are:

§ Inorder Traversal

§ Preorder Traversal

§ Postorder Traversal

In all of them we do not require to do anything to traverse an empty tree. All the traversal methods are based on recursive functions since a binary tree is itself recursive as every child of a node in a binary tree is itself a binary tree.

6.3.1 Inorder Traversal

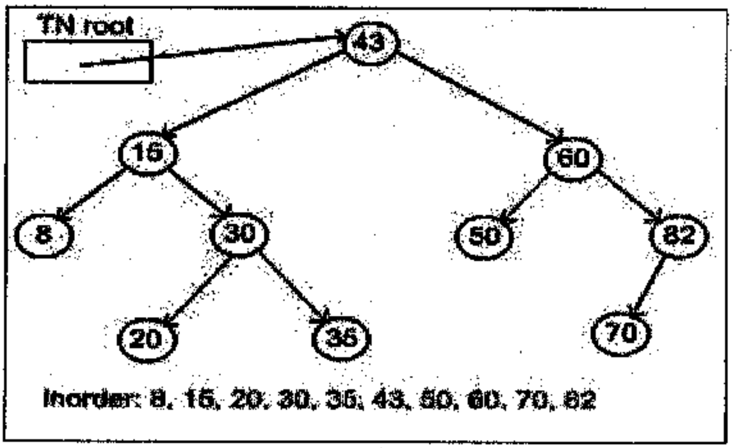
To traverse a non empty tree in inorder the following steps are followed recursively.

§ Traverse the left subtree

§ Visit the Root

§ Traverse the right subtree

The inorder traversal of the tree shown below is as follows.



Algorithm

The algorithm for inorder traversal is as follows.

```
Struct node
{
struct node * lc;
int data;
struct node * rc;
};

void inorder(struct node * root);
{
if(root != NULL)
{
inorder(roo-> lc);
printf("%d\t",root->data);
inorder(root->rc);
}
}
```

So the function calls itself recursively and carries on the traversal.

6.3.2 Preorder Traversal

To traverse a non empty tree in preorder the following steps are followed recursively.

§ Visit the Root

§ Traverse the left subtree

§ Traverse the right subtree

The preorder traversal of the tree shown above is

43 15 8 30 20 35 60 50 82 70

Algorithm

The algorithm for preorder traversal is as follows.

```
Struct node
{
struct node * lc;
int data;
struct node * rc;
};

void preorder(struct node * root);
{
if(root != NULL)
{
printf(“%d\t”,root->data);
preorder(roo-> lc);
preorder(root->rc);
}
}
```

So the function calls itself recursively and carries on the traversal.

6.3.3 Postorder Traversal

To traverse a non empty tree in postorder the following steps are followed recursively.

§ Traverse the left subtree

§ Traverse the right subtree

§ Visit the Root

The postorder traversal of the tree shown before is

8 20 35 30 15 50 70 82 60 43

Algorithm

The algorithm for postorder traversal is as follows.

```
Struct node
{
struct node * lc;
int data;
struct node * rc;
};
```

```
void postorder(struct node * root);
{
if(root != NULL)
{
postorder(roo-> lc);
postorder(root->rc);
printf("%d\t",root->data);
}
}
```

So the function calls itself recursively and carries on the traversal.

6.4 Operations on Binary Tree

There are many operations which can be performed on binary tree. Some are explained below:

Operations on Binary Tree are follows

- Searching
- Insertion
- Deletion
- Sort

6.4.1 Searching

Searching a binary tree for a specific value is a process that can be performed recursively because of the order in which values are stored. At first examining the root. If the value is equals the root, the value exists in the tree. If it is less than the root, then it must be in the left subtree, so we recursively search the left subtree in the same manner. Similarly, if it is greater than the root, then it must be in the right subtree, so we recursively search the right subtree. If we reach a leaf and have not found the value, then the item does not lie in the tree at all.

Here is the search algorithm

```
search_btree(node, key):
if node is None:
return None // key not found
if key < node.key:
return search_btree(node.left, key)
else if key > node.key:
return search_btree(node.right, key)
else : // key is equal to node key
return node.value // found key
```

6.4.2 Insertion

Insertion in a binary tree can be done in any order but for the operations wiz searching, sorting etc. binary search tree should be designed and insertion will be done in following sequence :

The way to insert a new node in the tree, its value is first compared with the value of the root. If its value is less than the root's, it is then compared with the value of the root's left child. If its value is greater, it is compared with the root's right child. This process continues, until the new node is compared with a leaf node, and then it is added as this node's right or left child, depending on its value.

Another way is examine the root and recursively insert the new node to the left subtree if the new value is less than or equal to the root, or the right subtree if the new value is greater than the root.

6.4.3 Deletion

There are several cases to be considered:

case -I : Deleting a leaf: If the key to be deleted has an empty left or right subtree, Deleting the key is easy, we can simply remove it from the tree.

case-II: Deleting a node with one child: Delete the key and fill up this place with its child.

case-III: Deleting a node with two children: Suppose the key to be deleted is called K . We replace the key K with either its in-order successor (the left-most child of the right subtree) or the in-order predecessor (the right-most child of the left subtree). we find either the in-order successor or predecessor, swap it with K, and then delete it. Since either of these nodes must have less than two children (otherwise it cannot be the in-order successor or predecessor), it can be deleted using the previous two cases.

6.4.4 Sort

A binary tree can be used to implement a simple but inefficient sorting algorithm. We insert all the values we wish to sort into a new ordered data structure. You can observe that inorder travesing always give sorted output.

6.5 Applications of Tree

Tree structure has many applications such as expression evaluation, dictionary search etc. Here we will discuss one important application.

6.5.1 Arithmetic expressions evaluation.

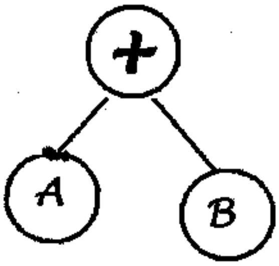
Trees are often used in and of themselves to store data directly, however they are also often used as the underlying implementation for other types of data structures such as **[Hash Tables]**, **[Sets and Maps]** and other associative containers.

Specifically, the C++ Standard Template Library uses special red/black trees as the underlying implementation for sets and maps, as well as multisets and multimaps.

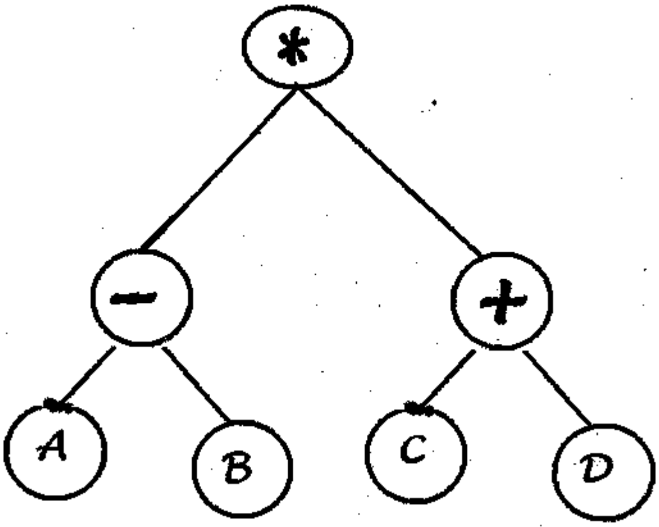
Binary Expression Tree

A binary tree may be used to represent a binary expression. A binary expression is made of operands on which binary operation may be performed.

As we know , a binary tree may have at most two children , we can , therefore , represent a simple binary expression where the root node contain the operator and the two children contain the two operands. For examples, A+B may be represent figure.



We can represent more a more complicated expression using a binary tree . for example , the expression (A-B) * (A+b) may be represented in next figure.



When we use a binary tree to represent an expression , we do not need parentheses to indicate the precedence. It is the level of the nodes in the tree which indicate the relative precedence.

6.6 Some Operations on Tree

6.6.1 Counting the no of nodes in a Binary tree

To count the number of nodes in a given binary tree, the tree is required to be traversed recursively until a leaf node is encountered. When a leaf node is encountered, a count of 1 is returned to its previous activation (which is an activation for its parent), which takes the count returned from both the children's activation, adds 1 to it, and returns this value to the activation of its parent. This way, when the activation for the root of the tree returns, it returns the count of the total number of the nodes in the tree.

Program

A complete C program to count the number of nodes is as follows:

```
#include <stdio.h>
#include <stdlib.h>
struct tnode
{
  int data;
  struct tnode *lchild, *rchild;
};
int count(struct tnode *p)
   {
          if( p == NULL)
    return(0);
          else
    if( p->lchild == NULL && p->rchild == NULL)
           return(1);
     else
          return(1 + (count(p->lchild) + count(p->rchild)));


   }

struct tnode *insert(struct tnode *p,int val)
   {
    struct tnode *temp1,*temp2;
     if(p == NULL)
     {
      p = (struct tnode *) malloc(sizeof(struct tnode)); /* insert the new node as root node*/
         if(p == NULL)
           {
       printf("Cannot allocate\n");
        exit(0);
            }
         p->data = val;
        p->lchild=p->rchild=NULL;
      }
     else
    {
     temp1 = p;
    /* traverse the tree to get a pointer to that node whose child will be the newly created
 node*/
   while(temp1 != NULL)
    {
     temp2 = temp1;
     if( temp1 ->data > val)
        temp1 = temp1->lchild;
     else
        temp1 = temp1->rchild;
    }
   if( temp2->data > val)
    {
    temp2->lchild = (struct tnode*)malloc(sizeof(struct tnode)); /
 *inserts the newly created node
   as left child*/
     temp2 = temp2->lchild;
     if(temp2 == NULL)
         {
    printf("Cannot allocate\n");
     exit(0);
         }
    temp2->data = val;
   temp2->lchild=temp2->rchild = NULL;
  }
 else
  {
   temp2->rchild = (struct tnode*)malloc(sizeof(struct tnode));/ *inserts the newly created
node
 as left child*/
  temp2 = temp2->rchild;
  if(temp2 == NULL)
      {
  printf("Cannot allocate\n");
  exit(0);
      }
  temp2->data = val;
 temp2->lchild=temp2->rchild = NULL;
}
}
return(p);
}
/* a function to binary tree in inorder */
void inorder(struct tnode *p)
    {
     if(p != NULL)
       {
       inorder(p->lchild);
      printf("%d\t",p->data);
     inorder(p->rchild);
       }
    }
void main()
{
 struct tnode *root = NULL;
 int n,x;
 printf("Enter the number of nodes\n");
 scanf("%d",&n);
 while( n —> 0)
    {
    printf("Enter the data value\n");
     scanf("%d",&x);
     root = insert(root,x);
    }
   inorder(root);
  printf("\nThe number of nodes in tree are :%d\n",count(root));
}
```

Explanation

- Input: 1. The number of nodes that the tree to be created should have

2. The data values of each node in the tree to be created

- Output: 1. The data value of the nodes of the tree in inorder

2. The count of number of node in a tree.

Example

- Input: 1. The number of nodes the created tree should have = 5

2. The data values of the nodes in the tree to be created are: 10, 20, 5, 9, 8

- Output: 1. 5 8 9 10 20

2. The number of nodes in the tree is 5

6.6.2 Swaping of left and right subtree of a given Binary tree

An elegant method of swapping the left and right subtrees of a given binary tree makes use of a recursive algorithm, which recursively swaps the left and right subtrees, starting from the root.

Program

```
#include <stdio.h>
#include <stdlib.h>
struct tnode
{
 int data;
 struct tnode *lchild, *rchild;
};

struct tnode *insert(struct tnode *p,int val)
{
  struct tnode *temp1,*temp2;
  if(p == NULL)
  {
     p = (struct tnode *) malloc(sizeof(struct tnode)); /* insert the new node as root node*/
      if(p == NULL)
        {
   printf("Cannot allocate\n");
   exit(0);
        }
     p->data = val;
    p->lchild=p->rchild=NULL;
  }
  else
  {
   temp1 = p;
 /* traverse the tree to get a pointer to that node whose child will be the newly created
```

```
node*/
 while(temp1 != NULL)
 {
  temp2 = temp1;
  if( temp1 ->data > val)
     temp1 = temp1->lchild;
  else
     temp1 = temp1->rchild;
 }
 if( temp2->data > val)
 {
  temp2->lchild = (struct tnode*)malloc(sizeof(struct tnode));/ *inserts the newly created
node
 as left child*/
  temp2 = temp2->lchild;
  if(temp2 == NULL)
      {
  printf("Cannot allocate\n");
   exit(0);
      }
    temp2->data = val;
   temp2->lchild=temp2->rchild = NULL;
 }
 else
 {
  temp2->rchild = (struct tnode*)malloc(sizeof(struct tnode));/ *inserts the newly created
node
 as left child*/
  temp2 = temp2->rchild;
  if(temp2 == NULL)
      {
  printf("Cannot allocate\n");
   exit(0);
      }
   temp2->data = val;
  temp2->lchild=temp2->rchild = NULL;
}
```

```
 }
 return(p);
 }
 /* a function to binary tree in inorder */
 void inorder(struct tnode *p)
   {
     if(p != NULL)
      {
       inorder(p->lchild);
      printf("%d\t",p->data);
     inorder(p->rchild);
      }
    }
struct tnode *swaptree(struct tnode *p)
{
  struct tnode *temp1=NULL, *temp2=NULL;
  if( p != NULL)
  { temp1= swaptree(p->lchild);
            temp2 = swaptree(p->rchild);
          p->rchild = temp1;
          p->lchild = temp2;
   }
   return(p);
 }

void main()
{
 struct tnode *root = NULL;
 int n,x;
 printf("Enter the number of nodes\n");
 scanf("%d",&n);
 while( n - > 0)
 {
  printf("Enter the data value\n");
  scanf("%d",&x);
  root = insert(root,x);
```

```
}
 printf("The created tree is :\n");
 inorder(root);
 printf("The tree after swapping is :\n");
 root = swaptree(root);
 inorder(root);
 printf("\nThe original tree is \n");
 root = swaptree(root);
 inorder(root);
}
```

Explanation

- Input:

1. The number of nodes that the tree to be created should have
2. The data values of each node in the tree to be created

- Output:

1. The data value of the nodes of the tree in inorder before interchanging the left and right subtrees
2. The data value of the nodes of the tree in inorder after interchanging the left and right subtrees

Example

- Input:

1. The number of nodes that the created tree should have = 5
2. The data values of the nodes in the tree to be created are: 10, 20, 5, 9, 8

- Output:

```
1. 5 8 9 10 20
2. 20 10 9 8 5
```

6.6.3 Searching For a target key in a Binary search tree

Data values are given which we call a key and a binary search tree. To search for the key in the given binary search tree, start with the root node and compare the key with the data value of the root node. If they match, return the root pointer. If the key is less than the data value of the root node, repeat the process by using the left subtree. Otherwise, repeat the same process with the right subtree until either a match is found or the subtree under consideration becomes an empty tree.

Program

A complete C program for this search is as follows:

```
#include <stdio.h>
#include <stdlib.h>
struct tnode
```

```
{
 int data;
 struct tnode *lchild, *rchild;
};
/* A function to serch for a given data value in a binary search tree*/
struct tnode *search( struct tnode *p,int key)
  {
   struct tnode *temp;
    temp = p;
   while( temp != NULL)
     {
      if(temp->data == key)
           return(temp);
       else
  if(temp->data > key)
    temp = temp->lchild;
  else
    temp = temp->rchild;
       }
return(NULL);
}

/*an iterative function to print the binary tree in inorder*/
void inorder1(struct tnode *p)
{
 struct tnode *stack[100];
 int top;
 top = -1;
if(p != NULL)
 {
   top++;
   stack[top] = p;
   p = p->lchild;
   while(top >= 0)
     {
      while ( p!= NULL)/* push the left child onto stack*/
     {
            top++;
         stack[top] =p;
         p = p->lchild;
    }
    p = stack[top];
    top-;
   printf(“%d\t”,p->data);
    p = p->rchild;


   if ( p != NULL) /* push right child*/
      {
         top++;
                 stack[top] = p;
        p = p->lchild;
          }


      }
  }
}
/* A function to insert a new node in binary search tree to
get a tree created*/
struct tnode *insert(struct tnode *p,int val)
{
 struct tnode *temp1,*temp2;
 if(p == NULL)
 {
  p = (struct tnode *) malloc(sizeof(struct tnode)); /* insert the new node as root node*/
   if(p == NULL)
     {
 printf(“Cannot allocate\n”);
 exit(0);
     }
  p->data = val;
  p->lchild=p->rchild=NULL;
  }
 else
 {
  temp1 = p;
 /* traverse the tree to get a pointer to that node whose child will be the newly created
node*/
 while(temp1 != NULL)
 {
  temp2 = temp1;
  if( temp1 ->data > val)
     temp1 = temp1->lchild;
  else
     temp1 = temp1->rchild;
 }
 if( temp2->data > val)
 {
  temp2->lchild = (struct tnode*)malloc(sizeof(struct tnode));/ *inserts the newly created
node
 as left child*/
  temp2 = temp2->lchild;
  if(temp2 == NULL)
      {
  printf("Cannot allocate\n");
  exit(0);
      }
  temp2->data = val;
  temp2->lchild=temp2->rchild = NULL;
 }
 else
 {
  temp2->rchild = (struct tnode*)malloc(sizeof(struct tnode));/ *inserts the newly created
node
 as left child*/
  temp2 = temp2->rchild;
  if(temp2 == NULL)
      {
  printf("Cannot allocate\n");
```

```
    exit(0);
       }
    temp2->data = val;
   temp2->lchild=temp2->rchild = NULL;
  }
}
return(p);
}
void main()
{
 struct tnode *root = NULL, *temp = NULL;
 int n,x;
 printf("Enter the number of nodes in the tree\n");
 scanf("%d",&n);
 while( n - > 0)
     {
     printf("Enter the data value\n");
     scanf("%d",&x);
     root = insert(root,x);
     }
   printf("The created tree is :\n");
   inorder1(root);
   printf("\n Enter the value of the node to be searched\n");
   scanf("%d",&n);
   temp=search(root,n);
   if(temp != NULL)
     printf("The data value is present in the tree \n");
    else
     printf("The data value is not present in the tree \n");
}
```

Explanation

- Input:

1. The number of nodes that the tree to be created should have
2. The data values of each node in the tree to be created
3. The key value

- Output

If the key is present and appears in the created tree, then a message "The data value is present in the tree" appears. Otherwise the message "The data value is not present in the tree" appears.

Example

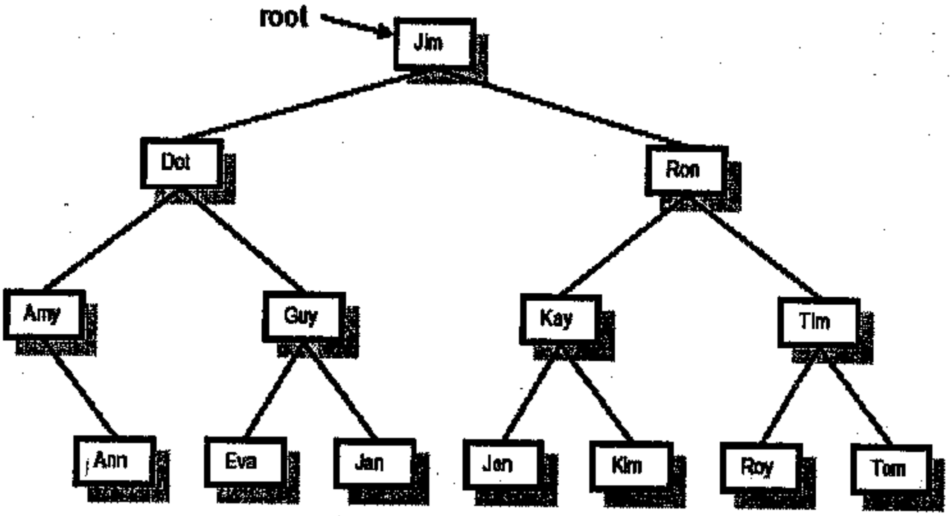
- Input:

1. The number of nodes that the created tree should have = 5
2. The data values of the nodes in the tree to be created are: 10, 20, 5, 9, 8
3. The key value = 9

- Output: The data is present in the tree

Self Learning Exercises

Assume the following Binary Search tree and answer the followings



1. Describe the steps for preorder traversing of the tree.
2. Describe the steps to search the node with information **Guy**
3. Describe the steps to insert **Meg** in the tree.
4. Describe the steps to delete **Guy**

6.7 Summary

- Tree structures support various basic dynamic set operations including Search, Predecessor, Successor, Minimum, Maximum, Insert, and Delete in time proportional to the height of the tree.

- Atree is a finite set of nodes having a distinct node called root. Binary Tree is a tree which is either empty or has at most two subtrees, each of the subtrees also being a binary tree. It means each node in a binary tree can have 0, 1 or 2 subtrees. A left or right subtree can be empty.

- The three main methods of traversing a tree are:
  - √ Inorder Traversal
  - √ Preorder Traversal
  - √ Postorder Traversal

- Operations on Binary Tree are :
  - √ Searching
  - √ 'nsertion
  - √ Deletion
  - √ Sort

6.8 Glossary

Tree: A data structure accessed beginning at the root node. Each node is either a leaf or an internal node. An internal node has one or more child nodes and is called the parent of its child nodes. All children of the same node are siblings. Contrary to a physical tree, the root is usually depicted at the top of the structure, and the leaves are depicted at the bottom. (2) A connected, undirected, acyclic graph. It is rooted and ordered unless otherwise specified.

Binary tree : A tree with at most two children for each node.

Complete binary tree : A binary tree in which every level, except possibly the deepest, is completely filled. At depth n, the height of the tree, all nodes must be as far left as possible.

Full binary tree : A binary tree in which each node has exactly zero or two children

Perfect binary tree : A binary tree with all leaf nodes at the same depth. All internal nodes have degree 2.

Binary search tree : A binary tree where every node's left subtree has keys less than the node's key, and every right subtree has keys greater than the node's key.

AVL tree : A balanced binary search tree where the height of the two subtrees (children) of a node differs by at most one. Look-up, insertion, and deletion are O(log n), where n is the number of nodes in the tree.

Balanced binary tree: A binary tree where no leaf is more than a certain amount farther from the root than any other. After inserting or deleting a node, the tree may rebalanced with "rotations."

Red-black tree: A nearly-balanced tree that uses an extra bit per node to maintain balance. No leaf is more than twice as far from the root as any other.

6.9 Further Readings

1. Gilles Brassard & Paul Bratley, Algorithmics, Prentice Hall, 1988
2. T. Cormen, C. Leiserson, & R. Rivest, Algorithms, MIT Press, 1990
3. Donald Knuth, The Art of Computer Programming (3 vols., various editions, 1973-81), Addison Wesley
4. Robert Kruse, Data Structures and Program Design , Prentice Hall, 1984
5. Udi Manber, Introduction to Algorithms, Addison Wesley, 1989

6.10 Answers to self learning exercises

1. PreOrder traversing

Step 1. Root = Jim

Display Jim then traverse its left subtree (root = Dot) and then its right subtree (root = Ron)

Display: Jim

Step 2. Root = Dot (Jim.LeftSubTree)

Display Dot then traverse its left subtree (root = Amy) and then its right subtree (root = Guy)

Display: Jim Dot

Step 3. Root = Amy (Dot.LeftSubTree)

Display Amy then traverse its left subtree (root = NULL) and then its right subtree (root = Ann)

Display: Jim Dot Amy

Since the right subtree of Amy is empty we then move onto the right subtree.

Step 4. Root = Ann (Amy.RightSubTree)

Display Ann then traverse its left subtree (root = NULL) and then its right subtree (root = NULL)

Display: Jim Dot Amy Ann

Since both of Ann's subtrees are empty we have finished traversing the tree with Root = Ann.

This completes the traversal of the right subtree of Amy and thus completes Amy,

The tree with root Amy is the left subtree of Dot, so we now continue with the right subtree of Dot (Root = Guy).

Step 5. Root = Guy (Dot.RightSubTree)

Display Guy then traverse its left subtree (root = Eva) and then its right subtree (root = Jan)

Display: Jim Dot Amy Ann Guy

Remaining steps

We display Eva and Jan and this completes the right subtree of Dot, and thus the left subtree of Jim.

Display: Jim Dot Amy Ann Guy Eva Jan

We now traverse the right subtree of Jim in a similar way, giving a final output of

Display: Jim Dot Amy Ann Guy Eva Jan Ron Kay Jon Kim Tim Roy Tom

2. Search **Target = Guy**

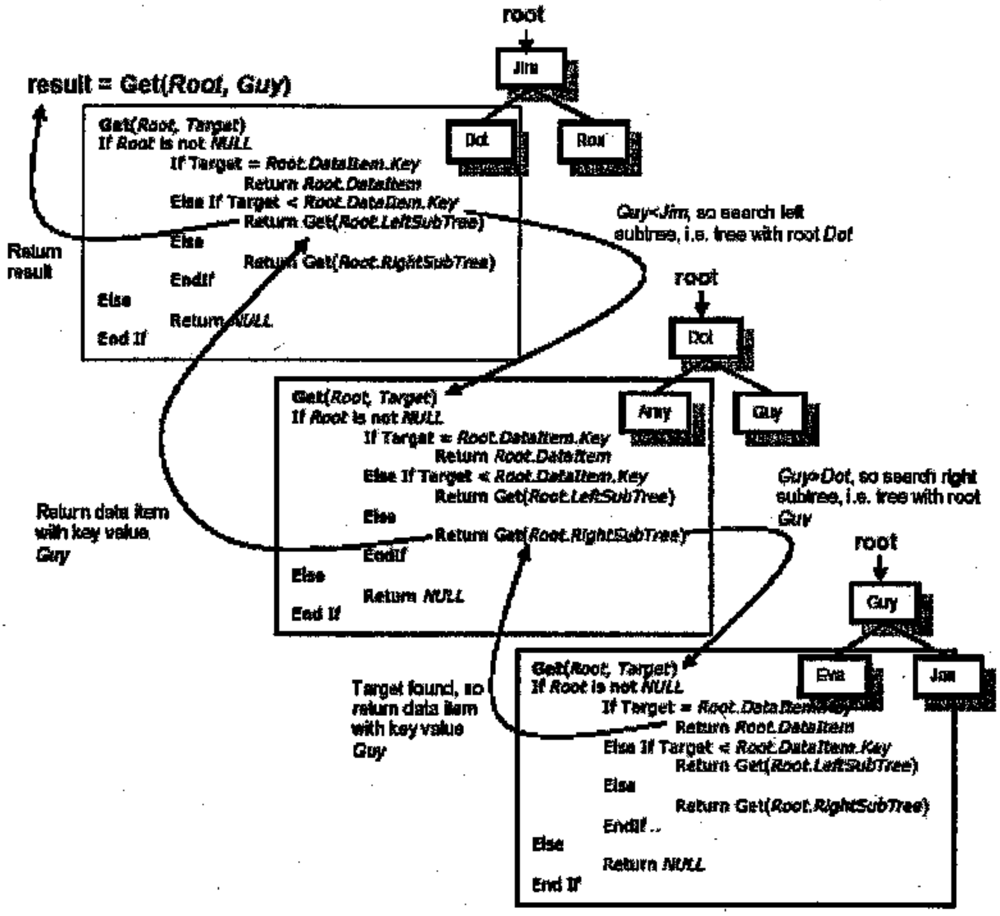
The operation must start at the root Jim and then go through the following stages:

1. Guy < Jim go to left subtree of Jim (root is Dot)

2. Guy > Dot go to right subtree of Dot (root is Guy)

3. Guy = root target found, return data item of node Guy

The target data item is passed by return statements back to the original operation call, as shown in the diagram below:


root
Jim
Dot
Ron
result = Get(*Root, Guy*)
Return result
Guy<Jim, so search left subtree, i.e. tree with root *Dot*
root
Dot
Amy
Guy
Return data item with key value *Guy*
Guy>Dot, so search right subtree, i.e. tree with root *Guy*
root
Guy
Eva
Jim
Target found, so return data item with key value *Guy*

```
Get(Root, Target)
If Root is not NULL
        If Target = Root.DataItem.Key
                Return Root.DataItem
        Else If Target < Root.DataItem.Key
                Return Get(Root.LeftSubTree)
        Else
                Return Get(Root.RightSubTree)
        EndIf
Else
        Return NULL
End If
```

```
Get(Root, Target)
If Root is not NULL
        If Target = Root.DataItem.Key
                Return Root.DataItem
        Else If Target < Root.DataItem.Key
                Return Get(Root.LeftSubTree)
        Else
                Return Get(Root.RightSubTree)
        EndIf
Else
        Return NULL
End If
```

```
Get(Root, Target)
If Root is not NULL
        If Target = Root.DataItem.Key
                Return Root.DataItem
        Else If Target < Root.DataItem.Key
                Return Get(Root.LeftSubTree)
        Else
                Return Get(Root.RightSubTree)
        EndIf
Else
        Return NULL
End If
```

3. Insert **Meg**

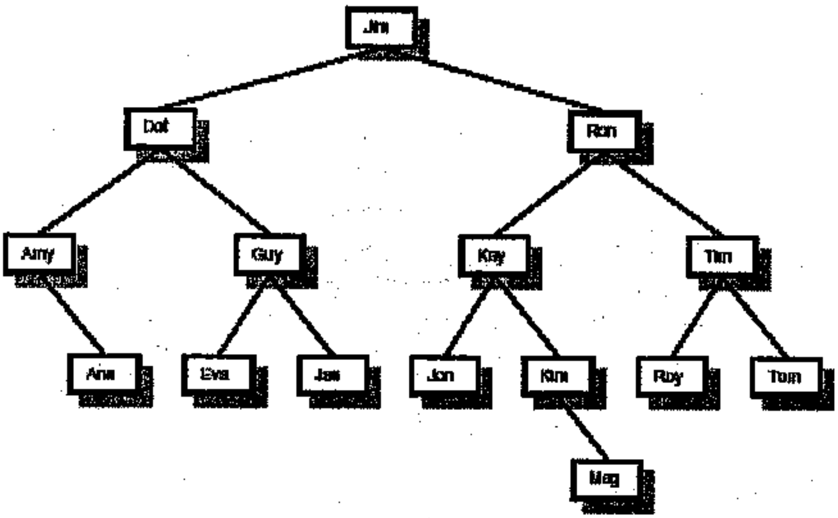
The Add operation is similar to the Get operation in that you have to recursively descend the tree until you find the appropriate place to add the new node. For example, if you want to add a new node with key Meg, the operation must start at the root Jim and then go through the following stages:

1. **Meg > Jim** *go to right subtree of* Jim

2. **Meg <Ron** *go to left subtree of* Ron

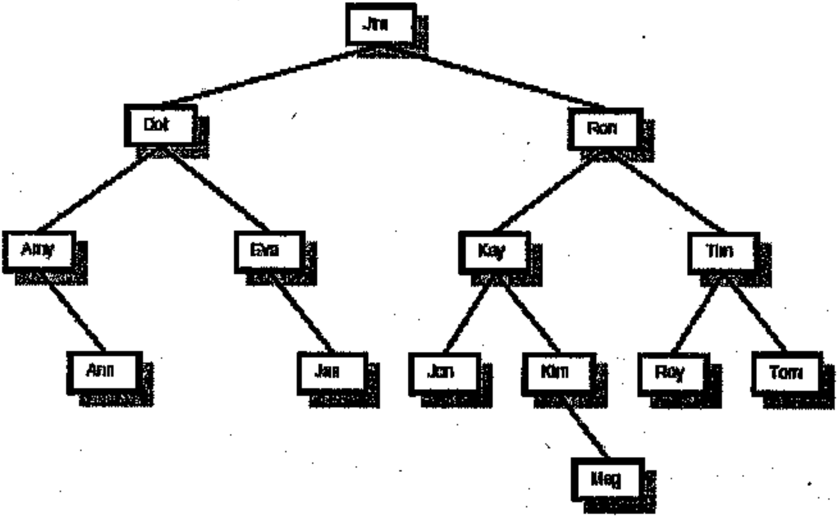
3. **Meg >*Kay*** *go to right subtree of* Kay

4.* Meg >*Kim *go to right subtree of* Kim *which is NULL, therefore add Meg as a right child of* Kim



4. Remove **Guy**

The remove operation can be rather involved, as it may be necessary to rearrange nodes so that the remaining structure is still a valid binary search tree. For example, if Guy is removed possible new structure would be



Note that:

- Eva *is now the right subtree of* Dot, *rather than* Guy
- Jan *is now the right subtree of* Eva, *rather than* Guy

*Removing a node with empty subtrees, known as a **leaf node** (e.g.* Meg*) is straightforward as no rearrangement is required.*
Algorithms to remove a node and change the attachments of other nodes as required are quite complex, and it can be useful to have a parent *reference in each node*

6.11 Unit End Questions

1. Write a C program to count the number of non-leaf nodes of a binary tree.

2. Write a C program to delete all the leaf nodes of a binary tree.

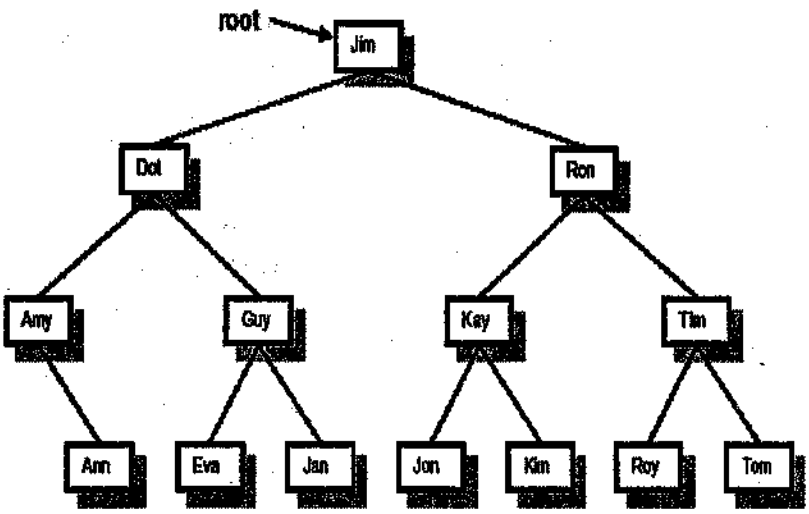
3. How many binary trees are possible with three nodes?

4. Write a C program to construct a binary tree with inorder and preorder traversals. Test it for the following inorder and preorder traversals:

o Inorder: 5, 1, 3, 11, 6, 8, 4, 2, 7

o Preorder: 6, 1, 5, 11, 3, 4, 8, 7, 2

5. Consider following tree and answer the followings



1. Describe the steps required to search for
   (a) Roy (b) Ian
2. Describe the steps required to add
   (a) Abi (b) Ken (c) Rik
3. Draw a diagram of a possible tree structure after removing:
   (a) Ann (b) Ron

----****----

UNIT VII

Advanced Tree

Structure of the Unit

7.0 Objectives

7.1 Introduction

7.2 Threaded Binary Tree

7.3 AVL tree

7.4 Multi-way trees

7.5 B tree

7.6 B+ tree

7.7 Trie and Dictionary

7.8 Summary

7.9 Glossary

7.10 Further Readings

7.11 Answers to self learning exercises

7.12 Unit End Questions

7.0 Objectives

After completing this unit you will learn

- Advanced tree, index binary tree, thareaded binary tree, AVL tree
- Multiway tree, B tree ,B+ tree
- Forest, tire and dictionary tree, etc.

7.1 Introduction

As we know that searching in a binary search tree is efficient if the height of the left sub-tree and right sub-tree is same for a node. But frequent insertion and deletion in the tree affects the efficiency and makes a binary search tree inefficient. The efficiency of searching will be ideal if the difference in height of left and right sub-tree with respect of a node is at most one. Such a binary search tree is called balanced binary tree (sometimes called AVL Tree). In this module we will study advanced tree structures which are more efficient.

7.2 Threaded Binary Tree

In linked representation of binary tree we can see most of the nodes have NULL value in the left and right pointer fields in the binary tree will be useful to use the pointer fields to keep some other information for operations in binary tree. These pointer fields can be used to contain the address pointer which points to the nodes higher in the tree. Such pointer which keeps the address of the nodes higher in tree is called thread. A binary tree which implements these pointers is called threaded binary tree.

There can be two types of threaded binary tree :-

1) Single Threaded :- i.e nodes are threaded either towards its inorder predecessor or successor.

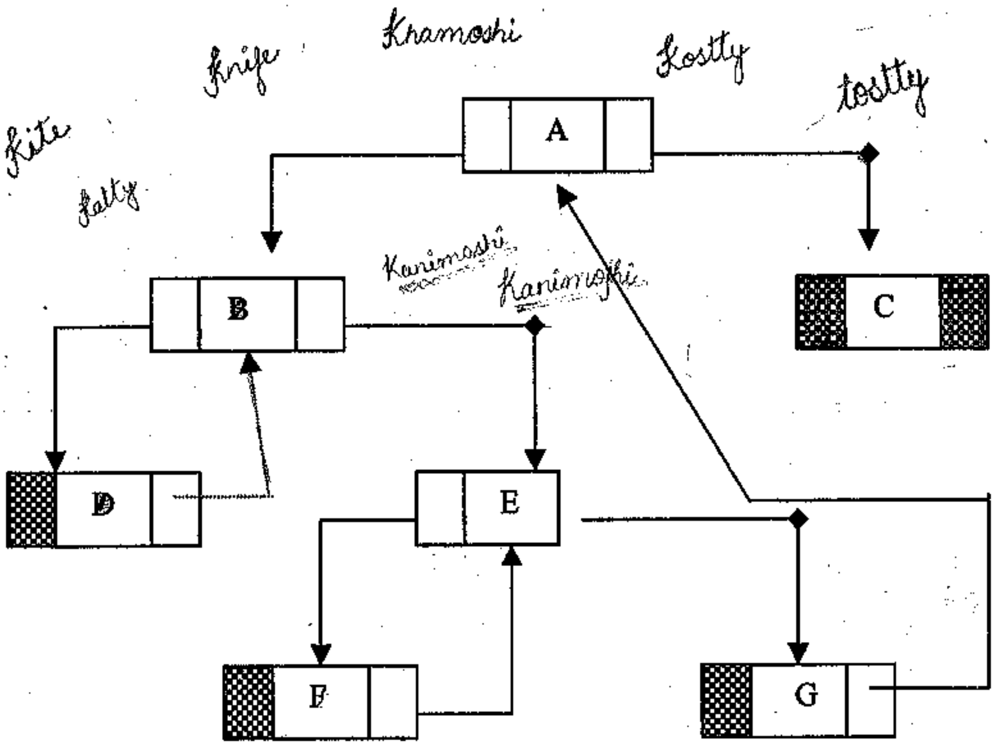
2) Double threaded:- i.e nodes are threaded towards both the inorder predecessor and successor.

We can have threading corresponding to any of the three traversals. There may be two types of inorder threading, one way inorder threading and two way inorder threading.

In one way inorder threading, right field of the node will keep the thread pointer which will point to the next node in the sequence of the inorder traversal or we can say right thread will point to the inorder successor of the node.

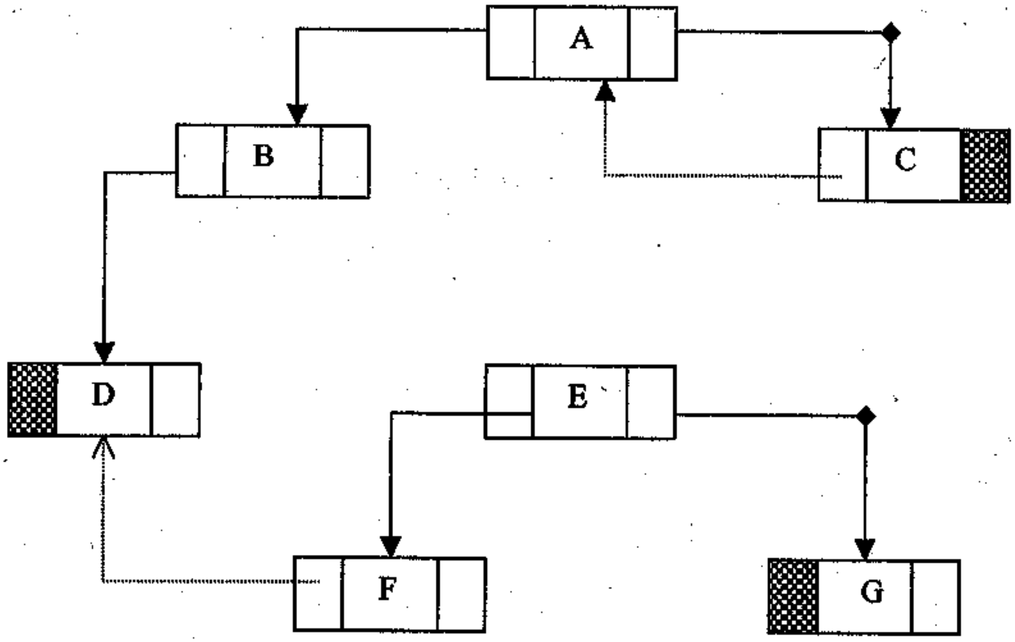
In two way threading left filed of the node will also keep the thread pointer which will point to the previous node in the sequence of inorder traversal or we can say left thread will point to the inorder predecessor of the node.

If we use right field of node to take the thread then this is called right in threaded binary tree. When we use left field of node to take the thread then this is called left in threaded binary tree. If both left and right fields are used for threading then it is called fully threaded binary tree or in threaded binary tree.



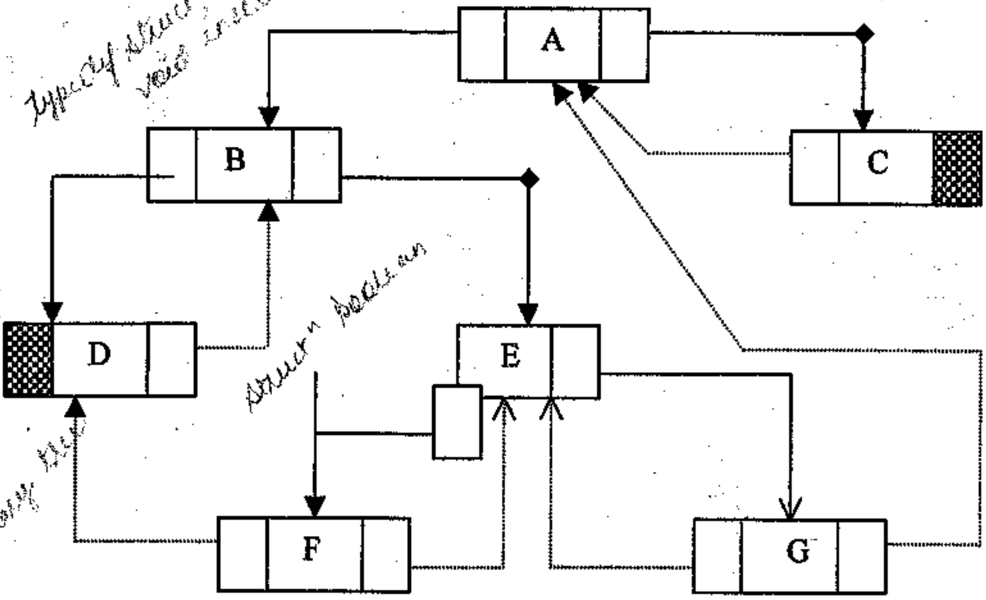
Right- in- threaded binary tree

In order traversal D B F E G A C



Left in threaded binary tree

typedef struct three next;
void inorder (node* , int);



struct boolean

Left in threaded binary tree

Fully in threaded binary tree

The structure of a node in a two way in threaded binary tree will be as

```
Typedef enum { thread, link} Boolean;
Struct node
{
struct node *left_ptr;
Boolean left;
Int info;
Struct node *right_ptr;
Boolean right;
};
```

Example for Threaded binary tree:

```
#include<stdio.h>
enum boolean{false, true};
struct thtree{
        int data;
        enum boolean left;
        struct thtree *leftchild;
        struct thtree *rightchild;
        enum boolean right;
};
typedef struct thtree next;
void insert(next **,int);
void inorder(next *);
int search(next  **head,        int  num, next **parent, next  **x, int *found);
int      delete(next      **head, int  data);
main()
{
        int i,n,data1;

        next *thread;
        thread=NULL;
        printf("\nenter a limited value");
        scanf("%d",&n);
        for(i=0;i<n;i++) {
                scanf("%d",&data1);
                insert(&thread,data1);
        }
        inorder(thread);
        puts("Enter number to be deleted");
        scanf("%d",&data1);
        delete(&thread,data1);
        inorder(thread);
}
void insert(next **s,int num)
{
        next *head=*s,*p,*z;
        z=(next *)malloc(sizeof(next));
        z->left=true;
        z->right=true;
        z->data=num;
        if(*s==NULL){
                head=(next *)malloc(sizeof(next));
                head->left=false;
                head->data=-9999;
                head->leftchild=z;
                head->right=false;
                head->rightchild=head;
                *s=head;
                z->leftchild=head;       /*left thread to head*/
                z->rightchild=head;      /*right thread to head*/
        }
        else {
                p=head->leftchild;
                while(p!=head){
                        if(p->data>num){
                                if(p->left!=true)
                                        p=p->leftchild;
                                else{
                                        z->leftchild=p->leftchild;
                                        p->leftchild=z;
                                        p->left=false;
                                        z->right=true;
                                        z->rightchild=p;
```

```
                    return;
                }
            }
            else{
                if(p->data<num){
                    if(p->right!=true)
                        p=p->rightchild;
                    else{
                        z->rightchild=p->rightchild;
                        p->rightchild=z;
                        p->right=false; /*indicates a link*/
                        z->left=true;
                        z->leftchild=p;

                        return;
                    }
                }
            }
        }
    }
}

void inorder(next *root)
{
    next *p;
    p=root->leftchild;
    while(p!=root){
        while(p->left==false)
            p=p->leftchild;
        printf(“%d ->”,p->data);
        while(p->right==true) {
            p=p->rightchild;
            if(p==root)
                break;
            printf(“%d-> “,p->data);
        }
```

```
            p=p->rightchild;
        }
}

int     delete(next      **head, int  data)
{
        int       found;
        next     *parent, *x,    *xsucc;
        if(*head == NULL){
                printf("Tree is empty");
                return;
        }
        parent = x = NULL;
        search(head, data, &parent, &x, &found);
        if(found == false){
                printf("Data to be deleted is not found\n");
                return;
        }
       // If a node has two children
       if(x ->left ==false && x ->right == false){
                parent = x;
                xsucc = x ->rightchild;

                while(xsucc -> left == false ){
                        parent = xsucc;
                        xsucc = xsucc ->leftchild;
                }
                x ->data = xsucc ->data;
                x = xsucc;
       }
      // If node has no child
      if(x -> left == true && x ->right == true){
              if(parent ->rightchild == x){
                      parent ->right = true;
                      parent->rightchild = x -> rightchild;
              }
        else {
                parent ->left = true;
                parent->leftchild = x -> leftchild;
        }
        free(x);
        return;
    }
    // If node to be deleted has only right child
    if(x -> left == true && x ->right == false){
        if(parent ->leftchild == x){
                parent ->leftchild = x ->rightchild;
                x ->rightchild ->leftchild = x -> leftchild;
        }
        else {
                parent ->rightchild = x ->rightchild;
                x ->rightchild -> leftchild = parent;
        }
        free(x);
        return;
    }
    // If node to be deleted has only left child

    if(x -> left == false && x ->right == true){
        if(parent ->leftchild == x){
                parent ->leftchild = x ->leftchild;
                x ->leftchild -> rightchild = parent;
        }
        else {
                parent ->rightchild = x ->leftchild;
                x ->leftchild -> rightchild = x -> rightchild;
        }
        free(x);
        return;
    }
}
```

```
int search(next  **head,          int  num, next **parent, next  **x, int *found)
{
        next     *q;
        q = (*head) ->leftchild;
        *found = false;
        *parent = NULL;

        while(q!= *head && q){
                if(q  ->data == num){
                        *found = true;
                        *x = q;
                        return;          }

                if(q -> data > num){
                        *parent = q;
                        q = q-> leftchild;
                }
                else{
                        *parent = q;
                        q = q-> rightchild;
                }       }
       *found = false;
       return;

}
```

7.3 AVL Tree

In order to represent a node of an AVL Tree, we need four fields :- One for data, two for storing address of left and right child and one is required to hold the balance factor. The balance factor is calculated by subtracting the right sub-tree from the height of left sub - tree.

The structure of AVL Tree can be represented by : -

```
     Struct AVL
   {
     struct AVL *left;
      int data;
     struct AVL *right;
     int balfact;
```

};

DETERMINATION OF BALANCE FACTOR

The value of balance factor may be -1, 0 or 1.

Any value other than these represent that the tree is not an AVL Tree

If the value of balance factor is -1, it shows that the height of right sub-tree is one more than the height of the left sub-tree with respect to the given node.

If the value of balance factor is 0, it shows that the height of right sub-tree is equal to the height of the left Sub-tree with respect to the given node.

If the value of balance factor is 1, it shows that the height of right sub-tree is one less than the height of the left sub-tree with respect to the given node.

INVENTION AND DEFINITION

It was invented in the year 1962 by two Russian mathematicians named G.M. Adelson-Velskii and E.M. Landis and so named AVL Tree.

It is a binary tree in which difference of height of two sub-trees with respect to a node never differ by more than one(1).

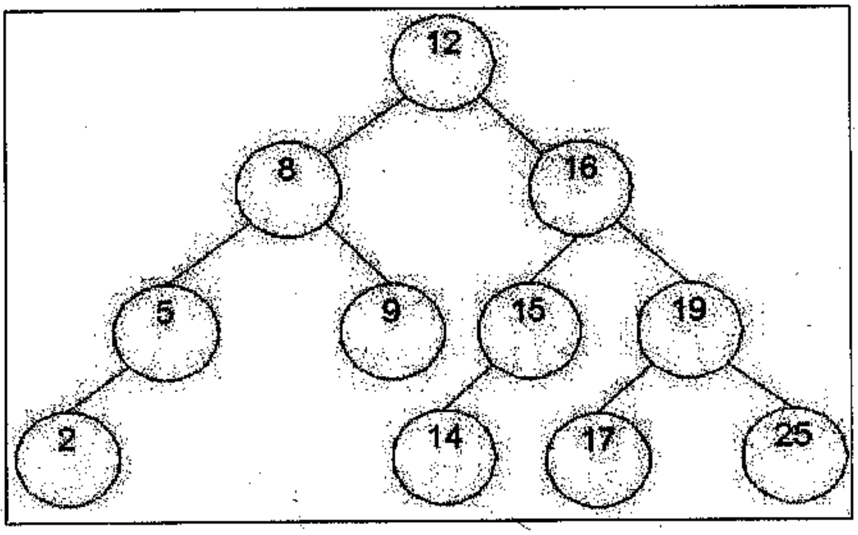


Diagram showing AVL Tree

INSERTION OF A NODE IN AVL TREE

Insertion can be done by finding an appropriate place for the node to be inserted. But this can disturb the balance of the tree if the difference of height of sub-trees with respect to a node exceeds the value one. If the insertion is done as a child of non-leaf node then it will not affect the balance, as the height doesn't increase. But if the insertion is done as a child of leaf node, then it can bring the real disturbance in the balance of the tree.

This depends on whether the node is inserted to the left sub-tree or the right sub-tree, which in turn changes the balance factor. If the node to be inserted is inserted as a node of a sub-tree of smaller height then there will be no effect. If the height of both the left and right sub-tree is same then insertionto any of them doesn't affect the balance of AVL Tree. But if it is inserted as a node of sub-tree of larger height, then the balance will be disturbed.

To rebalance the tree, the nodes need to be properly adjusted. So, after insertion of a new node the tree is traversed starting from the new node to the node where the balance has been disturbed. The nodes are adjusted in such a way that the balance is regained.

ALGORITHM FOR INSERTION IN AVL TREE

```
    int avl_insert(node *treep, value_t target)
       {
/* insert the target into the tree, returning 1 on success or 0 if it
* already existed
*/         node tree = *treep;
       node *path_top = treep;
          while (tree && target != tree->value)
             {
              direction next_step = (target > tree->value);
              if (!Balanced(tree)) path_top = treep;
              treep = &tree->next[next_step];
               tree = *treep;
             }
          if (tree) return 0;
           tree = malloc(sizeof(*tree));
            tree->next[0] = tree->next[1] = NULL;
            tree->longer = NEITHER;
            tree->value = target;
             *treep = tree;
           avl_rebalance(path_top, target);
   return 1;
      }
```

ALGORITHM FOR REBALANCING IN INSERTION

```
void avl_rebalance_path(node path, value_t target)
        {
/* Each node in path is currently balanced. Until we find target, mark each node as longer in the
direction of rget because we know we have inserted target there */
```

```
            while (path && target != path->value) {
            direction next_step = (target > path->value);
             path->longer = next_step;
             path = path->next[next_step];
            }
          }
  void avl_rebalance(node *path_top, value_t target)
          {
            node path = *path_top;
          direction first, second, third;
           if (Balanced(path)) {
          avl_rebalance_path(path, target);
            return;
          }
     first = (target > path->value);
         if (path->longer != first) {
            /* took the shorter path */
            path->longer = NEITHER;
          avl_rebalance_path(path->next[first], target);
            return;
          }
/* took the longer path, need to rotate */
second = (target > path->next[first]->value);
     if (first == second) {
/* just a two-point rotate */
        path = avl_rotate_2(path_top, first);
        avl_rebalance_path(path, target);
         return;
        }
/* fine details of the 3 point rotate depend on the third step. However there may not be a third
step, if the third point of the rotation is the newly inserted point. In that case we record the third
step as NEITHER */
path = path->next[first]->next[second];
    if (target == path->value) third = NEITHER;
        else third = (target > path->value);
            path = avl_rotate_3(path_top, first, third);
            avl_rebalance_path(path, target);
        }
```

DELETION

A node in AVL Tree is deleted as it is deleted in the binary search tree. The only difference is that we have to do rebalancing which is done similar to that of insertion of a node in AVL Tree.The algorithm for deletion and rebalancing is given below:

ALGORITHM FOR DELETION IN AVL TREE

```
 int avl_delete(node *treep, value_t target)
        {
/* delete the target from the tree, returning 1 on success or 0 if it wasn't found */
          node tree = *treep;
        direction dir;
        node *targetp, targetn;
              while(tree) {
                dir = (target > value);
                  if (target == value) targetp = treep;
                        if (tree->next[dir] == NULL)
                               break;
                              if (tree->longer == NEITHER || (tree->longer == 1-dir && tree-
>next[1-dir]->longer == NEITHER))
                                  path_top = treep;
                                  treep = &tree->next[dir];
                                   tree = *treep;
                                     }
               if (targetp == NULL) return 0;
              targetp = avl_rebalance_del(path_top, target, targetp);
               avl_swap_del(targetp, treep, dir);
                return 1;
     }
```

ALGORITHM FOR REBALANCING IN DELETION IN AVL TREE

```
 node *avl_rebalance_del(node *treep, value_t target, node *targetp)
{
    node targetn = *targetp;
       while(1) {
           node tree = *treep;
         direction dir = (target > tree->value);
```

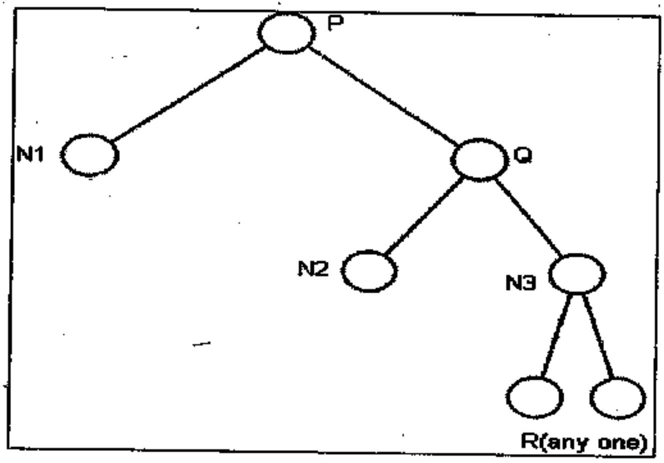
```
        if (tree->next[dir] == NULL)
            break;
          if (Balanced(tree))
            tree->longer = 1-dir;
          else if (tree->longer == dir)
            tree->longer = NEITHER;
                else {
                        /* a rotation is needed, and targetp might change */
                  if (tree->next[1-dir]->longer == dir)
                    avl_rotate_3_shrink(treep, dir);
                    else
                     avl_rotate_2_shrink(treep, dir);
                       if (tree == targetn)
                           *targetp = &(*treep)->next[dir];
                     }
           treep = &tree->next[dir];
            }
return targetp;
}
```

REBALANCING OF AVL TREE

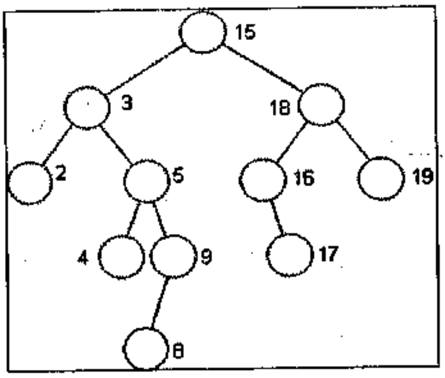
When we insert a node to the taller sub-tree, four cases arise and we have different rebalancing methods to bring it back to a balanced tree form.

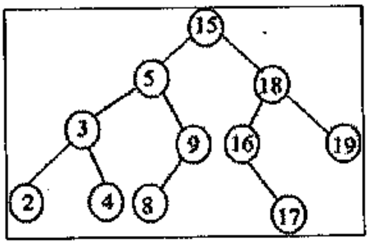
LEFT ROTATION

In general if we want to insert a node R(either as left child or right child) to N3 as shown in figure. Here, as we see the balance factor of node P becomes 2. So to rebalance it, we have a technique called left rotation



Example





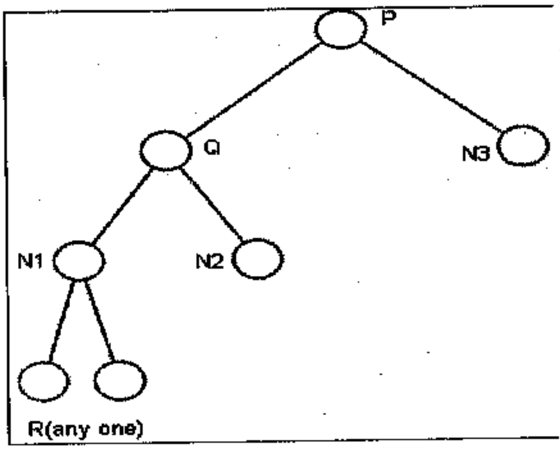
Before Rotation **After Rotation**

EXPLANATION OF EXAMPLE

In the given AVL tree when we insert a node 8,it becomes the left child of node 9 and the balance doesn't exist, as the balance factor of node 3 becomes -2. So, we try to rebalance it. In order to do so, we do left rotation at node 3. Now node 5 becomes the left child of the root. Node 9 and node 3 becomes the right and left child of node 5 respectively. Node 2 and node 4 becomes the left and right child of node 3 respectively. Lastly, node 8 becomes the left child of node 9. Hence, the balance is once again attained and we get AVL Tree after the left rotation.

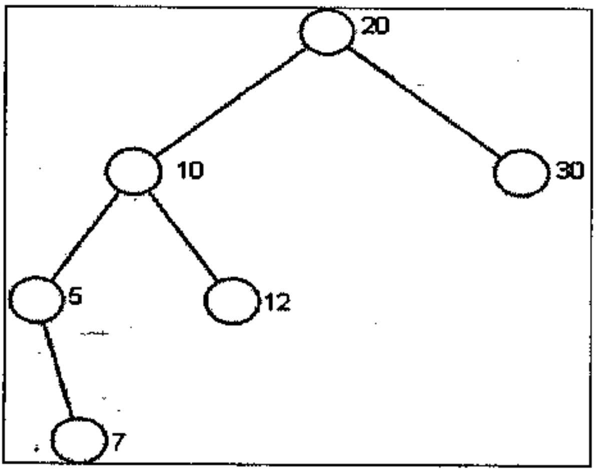
RIGHT ROTATION

In general if we want to insert a node R(either as left or right child) to N1 as shown in figure. Here, as we see the balance factor of node P becomes 2. So to rebalance it, we have a technique called right rotation.

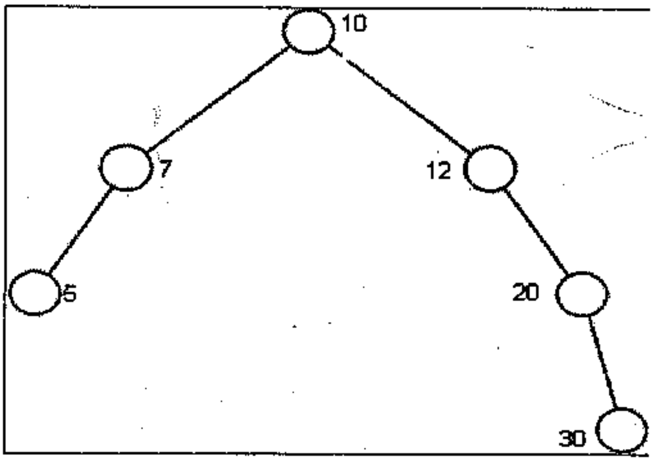


GENERAL DIAGRAM

Example



Before Rotation



After Rotation

EXPLANATION OF EXAMPLE

In the given AVL tree when we insert a node 7,it becomes the right child of node 5 and the balance doesn't exist, as the balance factor of node 20 becomes 2. So, we try to rebalance it. In order to do so, we do right rotation at node 20. Now node 10 becomes the root. Node 12 and node 7 becomes the right and left child of root respectively. Node 20 becomes the right child of node 12. Node 30 becomes the right child of node 20. Lastly, node 5 becomes the left child of node 7. Hence, the balance is once again attained and we get AVL Tree after the right rotation

7.4 Multi-way Trees

A Multiway Search Tree of order n is a tree in which any node can have a maximum of n-1 values & a max. of n children. B - Trees are a special case of Multiway Search Trees. B Tree of order n is a Multiway Search Tree of order n with the following characteristics:

All the non leaf nodes have a max of n child nodes & a min of n/2 child nodes.

If a root is non leaf node, then it has a max of n non empty child nodes & a min of 2 child nodes.

If a root node is a leaf node, then it does not have any child node.

A node with n child nodes has n-1 values arranged in ascending order.

All values appearing on the left most child of any node are smaller than the left most value of that node while all values appearing on the right most child of any node are greater than the right most value of that node.

If x & y are two adjacent values in a node such that x < y, ie they are the i th & (i+1) th values in the node respectively, then all values in the (i+1) th child of that node are > x but < y.

7.5 B tree

Tree structures support various basic dynamic set operations including Search , Insert , and Delete in time proportional to the height of the tree. Ideally, a tree will be balanced and the height will be log n where n is the number of nodes in the tree.

To ensure that the height of the tree is as small as possible and therefore provide the best running time, a balanced tree structure like AVL tree , 2-3 Tree , Red Black Tree or B-tree must be used.

When working with large sets of data, it is often not possible or desirable to maintain the entire structure in primary storage (RAM). Instead, a relatively small portion of the data structure is maintained in primary storage, and additional data is read from secondary storage as needed. Unfortunately, a magnetic disk, the most common form of secondary storage, is significantly slower than random access memory (RAM). In fact, the system often spends more time retrieving data than actually processing data.

To reduce the time lost in retrieving data from secondary storage, we need to minimize the no. of references to the secondary memory. This is possible if a node in a tree contains more no. of values, then in a single reference to the secondary memory more nodes can be accessed. The AVL trees or red Black Trees can hold a max. of 1 value only in a node while 2-3 Trees can hold a max of 2 values per node. To improve the efficiency Multiway Search Trees are used.

OPERATIONS ON B - TREE

Following operations can be done on a B - Tree :

- Searching
- Insertion
- Deletion

SEARCHING OF A VALUE IN A B-TREE

Searching of a value k in a B-Tree is exactly similar to searching for values in a 2-3 tree. To begin with the value k is compared with the first value key [0] of the root node. If they are similar then the search is complete. If k is less than key [0] then the search is done in the first child node or the sub-tree of the root node.

If k is greater than key [0] then it is compared with key [1]. If k is greater than key [0] and smaller than key [1] then k is searched in the second child node or sub-tree of the root node. If k is greater than the last value key [i] of the root node then searching is done in the last child node or sub-tree of the root node. If k is searched in any of the child nodes or sub-tree of the root node then the same procedure of searching is repeated for that particular node or sub-tree.

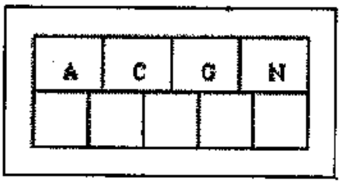
If the value k is found in the tree then the search is successful . The address of the node in which k is present and the position of the value k in that node is returned. If the value k is not found in the tree, then the search is unsuccessful .

INSERTION OF A VALUE IN A B-TREE

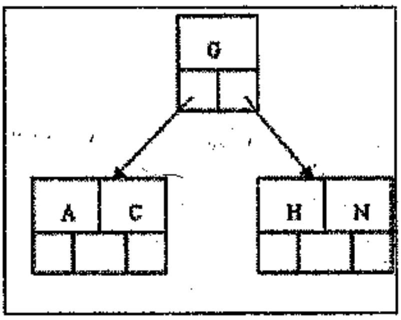
When inserting an item, first do a search for it in the B-tree. If the item is not already in the B-tree, this unsuccessful search will end at a leaf. If there is room in this leaf, just insert the new item here. Note that this may require that some existing keys be moved one to the right to make room for the new item. If instead this leaf node is full so that there is no room to add the new item, then the node must be "split" with about half of the keys going into a new node to the right of this one. The median (middle) key is moved up into the parent node. (Of course, if that node has no room, then it may have to be split as well.)

Note that when adding to an internal node, not only might we have to move some keys one position to the right, but the associated pointers have to be moved right as well. If the root node is ever split, the median key moves up into a new root node, thus causing the tree to increase in height by one.

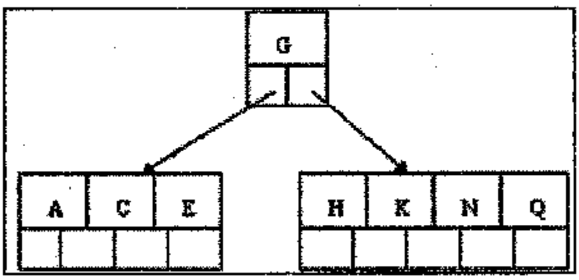
Lets take an example.Insert the following letters into what is originally an empty B-tree of order 5: C N G A H E K Q M F W L T Z D P R X Y S Order 5 means that a node can have a maximum of 5 children and 4 keys. All nodes other than the root must have a minimum of 2 keys. The first 4 letters get inserted into the same node, resulting in this picture:



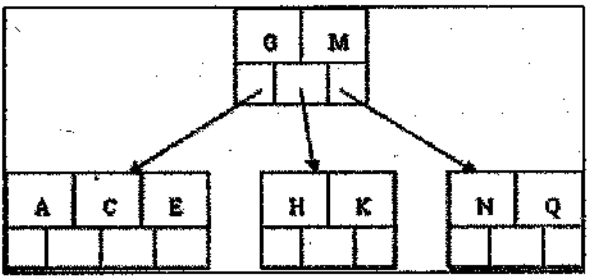
When we try to insert the H, we find no room in this node, so we split it into 2 nodes, moving the median item G up into a new root node. Note that in practice we just leave the A and C in the current node and place the H and N into a new node to the right of the old one.



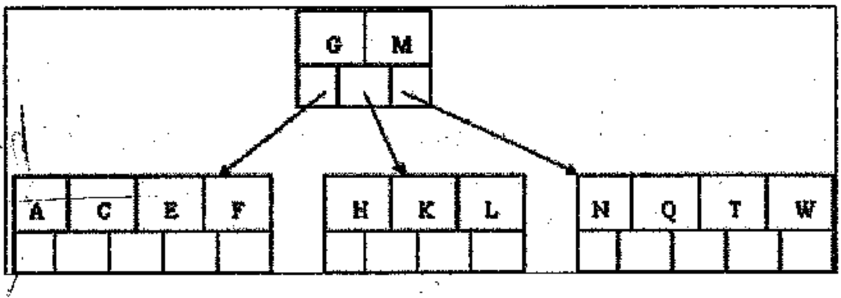
Inserting E, K, and Q proceeds without requiring any splits:



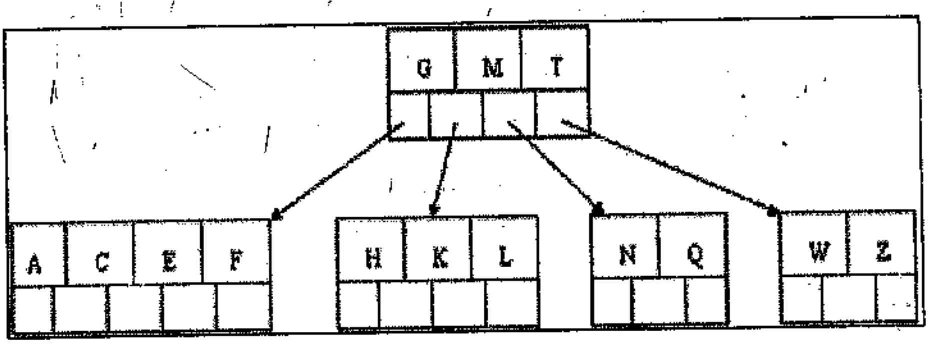
Inserting M requires a split. Note that M happens to be the median key and so is moved up into the parent node.



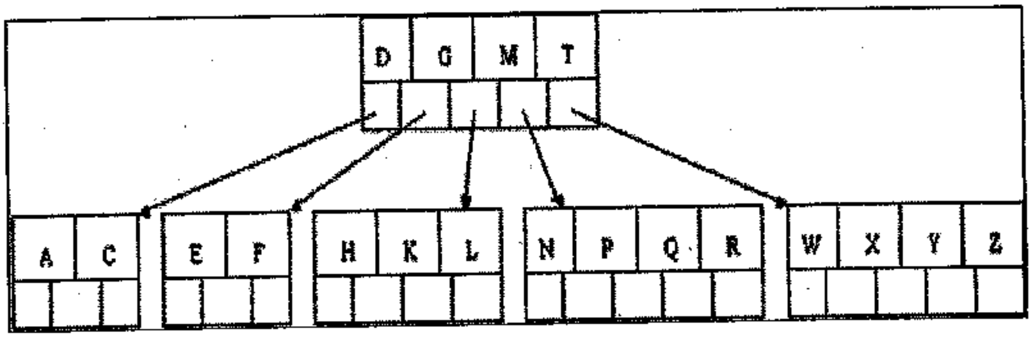
The letters F, W, L, and T are then added without needing any split.



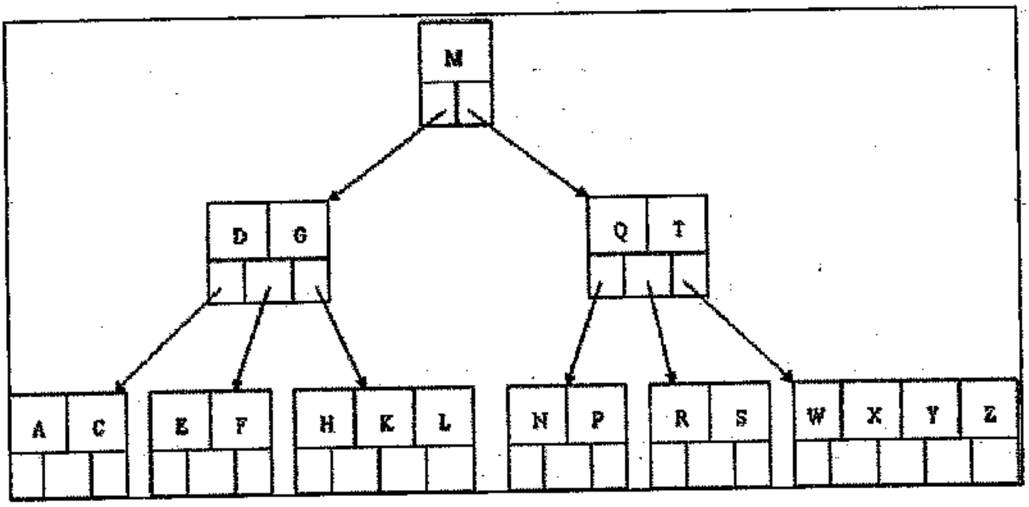
When Z is added, the rightmost leaf must be split. The median item T is moved up into the parent node. Note that by moving up the median key, the tree is kept fairly balanced, with 2 keys in each of the resulting nodes



The insertion of D causes the leftmost leaf to be split. D happens to be the median key and so is the one moved up into the parent node. The letters P, R, X, and Y are then added without any need of splitting:



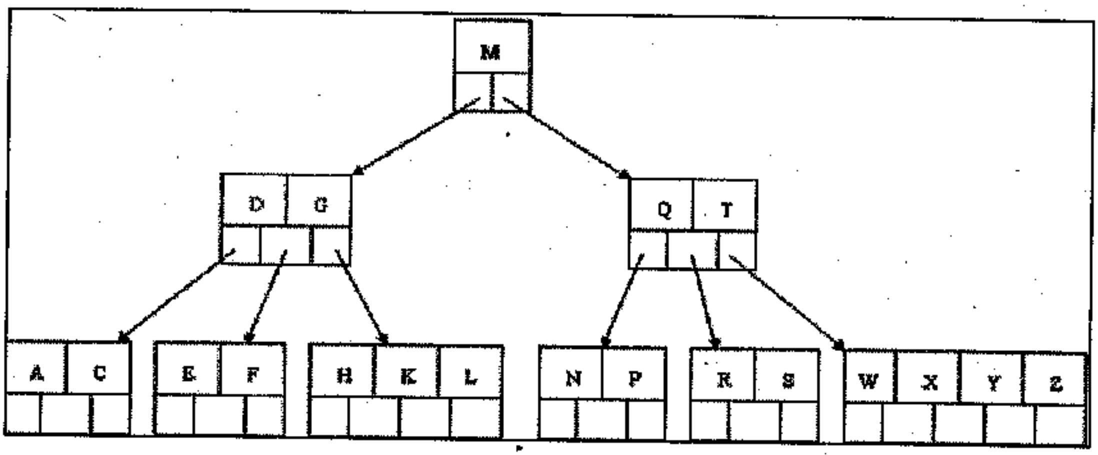
Finally, when S is added, the node with N, P, Q, and R splits, sending the median Q up to the parent. However, the parent node is full, so it splits, sending the median M up to form a new root node. Note how the 3 pointers from the old parent node stay in the revised node that contains D and G



DELETION IN A B - TREE

Deletion in a B -Tree is similar to insertion.At first the node from which a value is to be deleted is searched.If found out, then the value is deleted. After deletion the tree is checked if it still follows B - Tree properties.

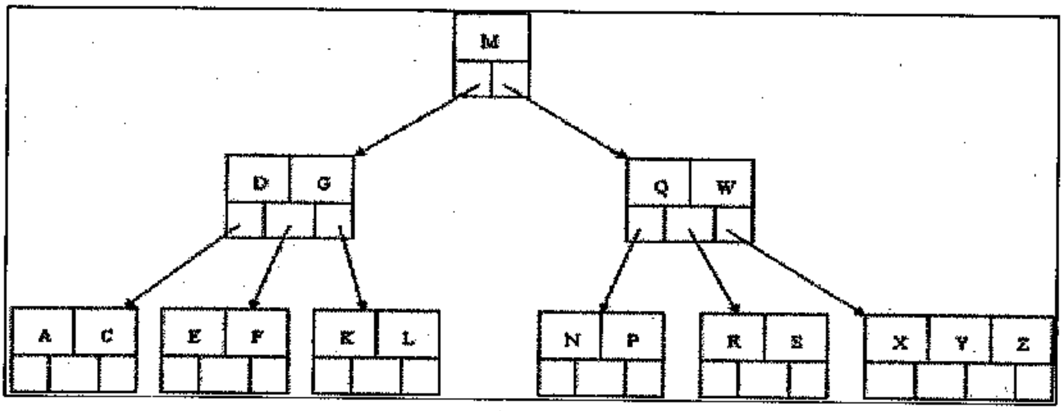
Let us take an example.THe original B - Tree taken is as follows:



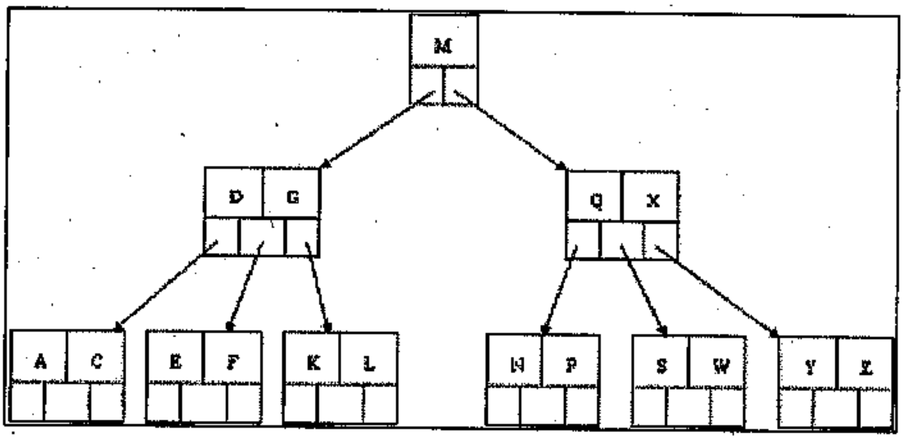
STEPS FOR DELETION IN A B-TREE

Delete H. first it is found out.Since H is in a leaf and the leaf has more than the minimum number of keys, this is easy. We move the K over where the H had been and the L over where the K had been.

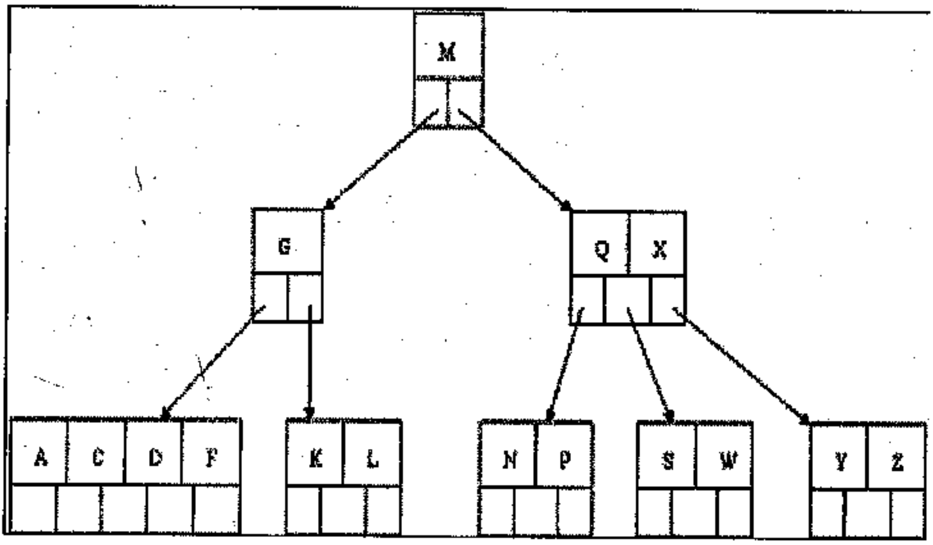
Next, delete the T. Since T is not in a leaf, we find its successor (the next item in ascending order), which happens to be W, and move W up to replace the T. That way, what we really have to do is to delete W from the leaf, which we already know how to do, since this leaf has extra keys. In ALL cases we reduce deletion to a deletion in a leaf, by using this method



Next, delete R. Although R is in a leaf, this leaf does not have an extra key; the deletion results in a node with only one key, which is not acceptable for a B-tree of order 5. If the sibling node to the immediate left or right has an extra key, we can then borrow a key from the parent and move a key up from this sibling. In our specific case, the sibling to the right has an extra key. So, the successor W of S (the last key in the node where the deletion occurred), is moved down from the parent, and the X is moved up. (Of course, the S is moved over so that the W can be inserted in its proper place.)

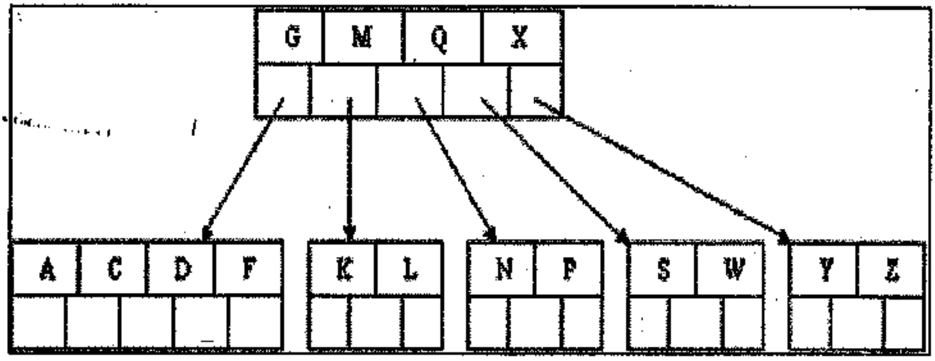


Finally, let's delete E. This one causes lots of problems. Although E is in a leaf, the leaf has no extra keys, nor do the siblings to the immediate right or left. In such a case the leaf has to be combined with one of these two siblings. This includes moving down the parent's key that was between those of these two leaves. In our example, let's combine the leaf containing F with the leaf containing A C. We also move down the D



Of course, you immediately see that the parent node now contains only one key, G. This is not acceptable. If this problem node had a sibling to its immediate left or right that had a spare key, then we would again "borrow" a key. Suppose for the moment that the right sibling (the node with Q X) had one more key in it somewhere to the right of Q. We would then move M down to the node with too few keys and move the Q up where the M had been. However, the old left subtree of Q would then have to become the right subtree of M.

In other words, the N P node would be attached via the pointer field to the right of M's new location. Since in our example we have no way to borrow a key from a sibling, we must again combine with the sibling, and move down the M from the parent. In this case, the tree shrinks in height by one.



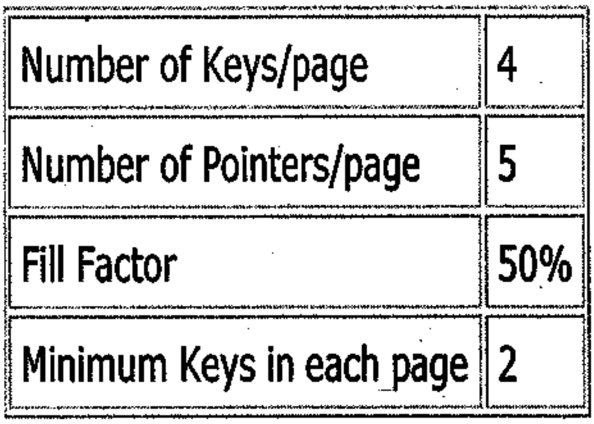
7.6 B+ tree

B Trees. B Trees are multi-way trees. That is each node contains a set of keys and pointers. A B Tree with four keys and five pointers represents the minimum size of a B Tree node. A B Tree contains only data pages.

B Trees are dynamic. That is, the height of the tree grows and contracts as records are added and deleted.

B+ Tree combines features of ISAM and B Trees. It contains index pages and data pages. The data pages always appear as leaf nodes in the tree. The root node and intermediate nodes are always index pages. These features are similar to ISAM. Unlike ISAM, overflow pages are not used in B+ trees.

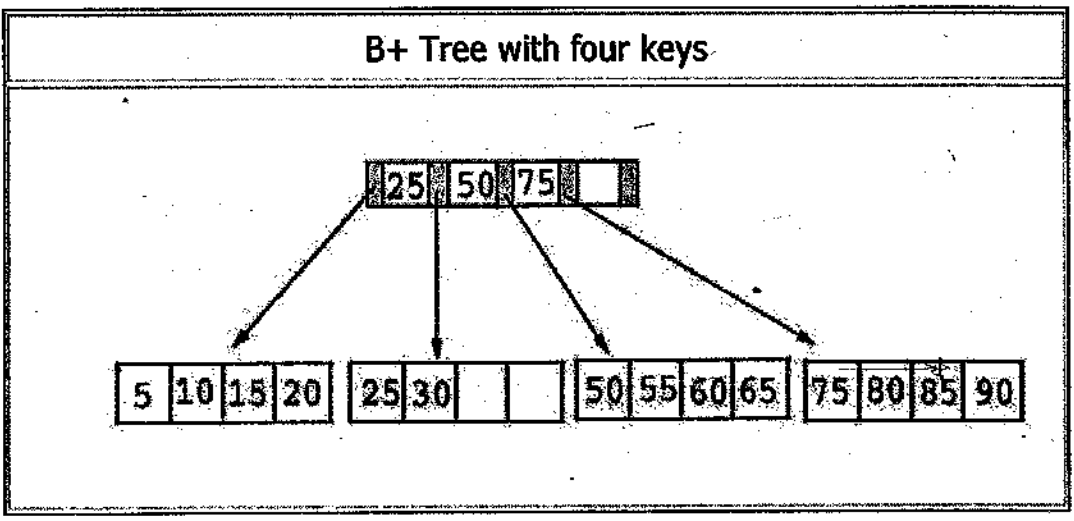
The index pages in a B+ tree are constructed through the process of inserting and deleting records. Thus, B+ trees grow and contract like their B Tree counterparts. The contents and the number of index pages reflects this growth and shrinkage. B+ Trees and B Trees use a "fill factor" to control the growth and the shrinkage. A 50% fill factor would be the minimum for any B+ or B tree. As our example we use the smallest page structure. This means that our B+ tree conforms to the following guidelines.



Number of Keys/page	4
Number of Pointers/page	5
Fill Factor	50%
Minimum Keys in each page	2

As this table indicates each page must have a minimum of two keys. The root page may violate this rule.

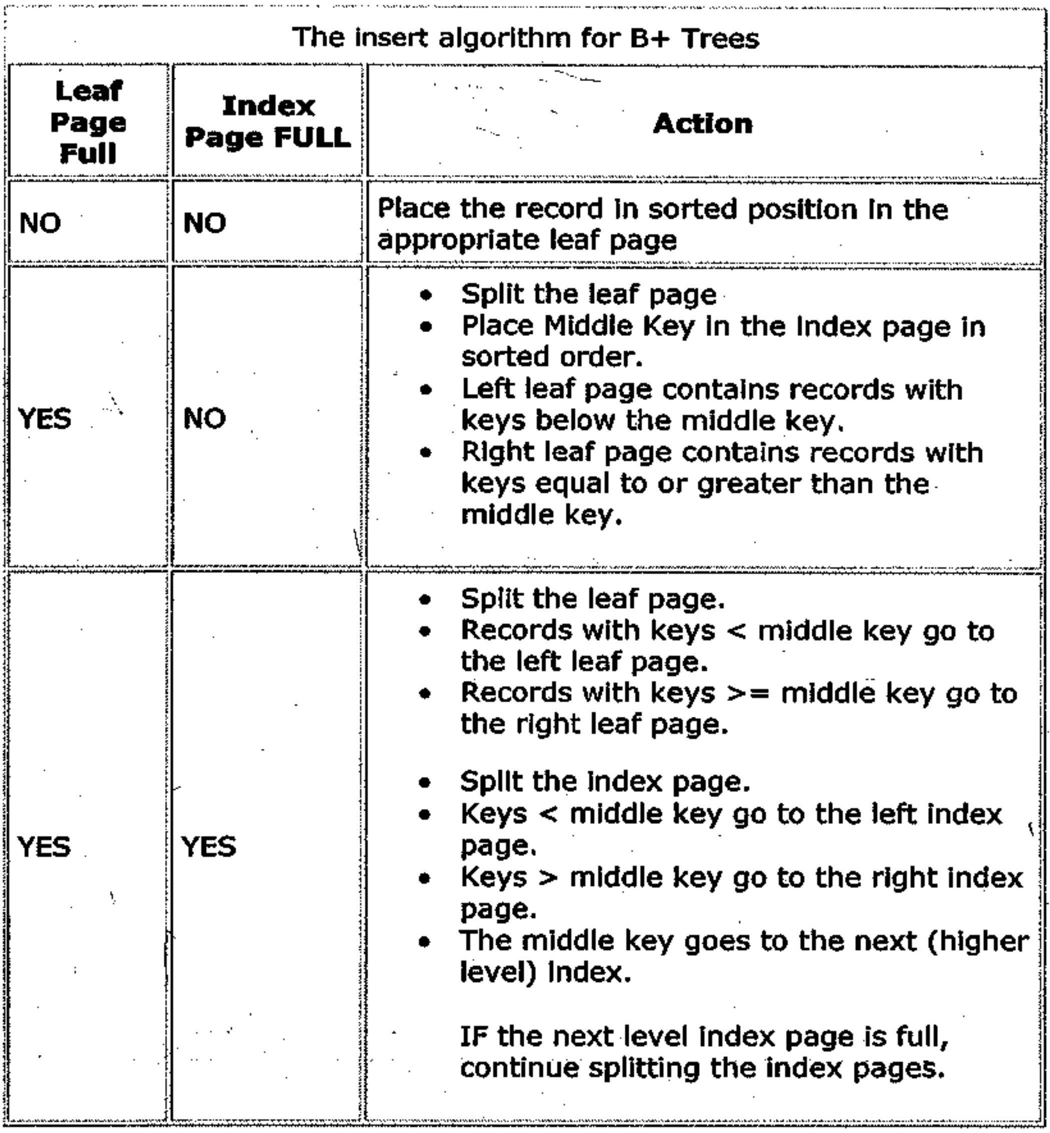
The following table shows a B+ tree. As the example illustrates this tree does not have a full index page. (We have room for one more key and pointer in the root page.) In addition, one of the data pages contains empty slots.



B+ Tree with four keys

Adding Records to a B+ Tree

The key value determines a record's placement in a B+ tree. The leaf pages are maintained in sequential order AND a doubly linked list (not shown) connects each leaf page with its sibling page(s). This doubly linked list speeds data movement as the pages grow and contract. We must consider three scenarios when we add a record to a B+ tree. Each scenario causes a different action in the insert algorithm. The scenarios are:

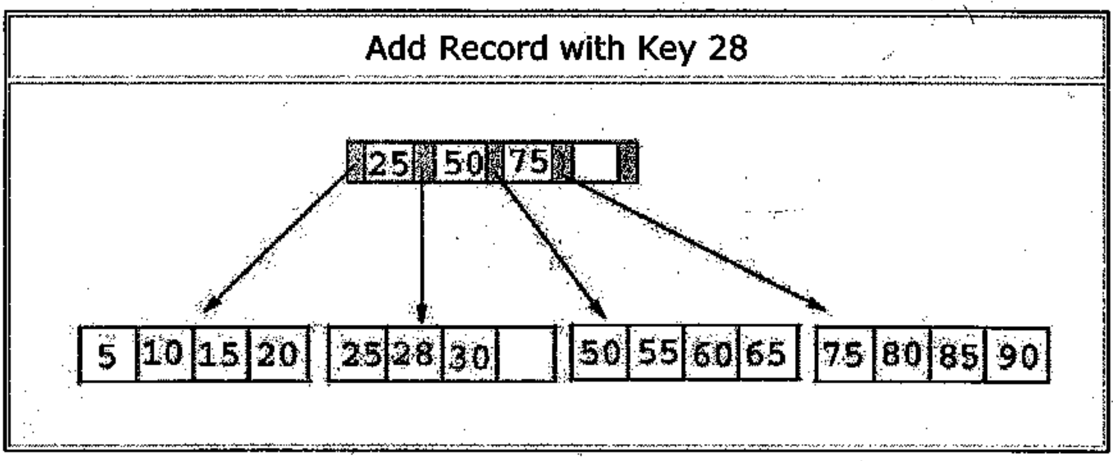


The insert algorithm for B+ Trees

Leaf Page Full	Index Page FULL	Action
NO	NO	Place the record in sorted position in the appropriate leaf page
YES	NO	• Split the leaf page • Place Middle Key in the index page in sorted order. • Left leaf page contains records with keys below the middle key. • Right leaf page contains records with keys equal to or greater than the middle key.
YES	YES	• Split the leaf page. • Records with keys < middle key go to the left leaf page. • Records with keys >= middle key go to the right leaf page. • Split the index page. • Keys < middle key go to the left index page. • Keys > middle key go to the right index page. • The middle key goes to the next (higher level) index. IF the next level index page is full, continue splitting the index pages.

Illustrations of the insert algorithm

The following examples illlustrate each of the **insert** scenarios. We begin with the simplest scenario: inserting a record into a leaf page that is not full. Since only the leaf node containing 25 and 30 contains expansion room, we're going to insert a record with a key value of 28 into the B+ tree. The following figures shows the result of this addition.


Add Record with Key 28

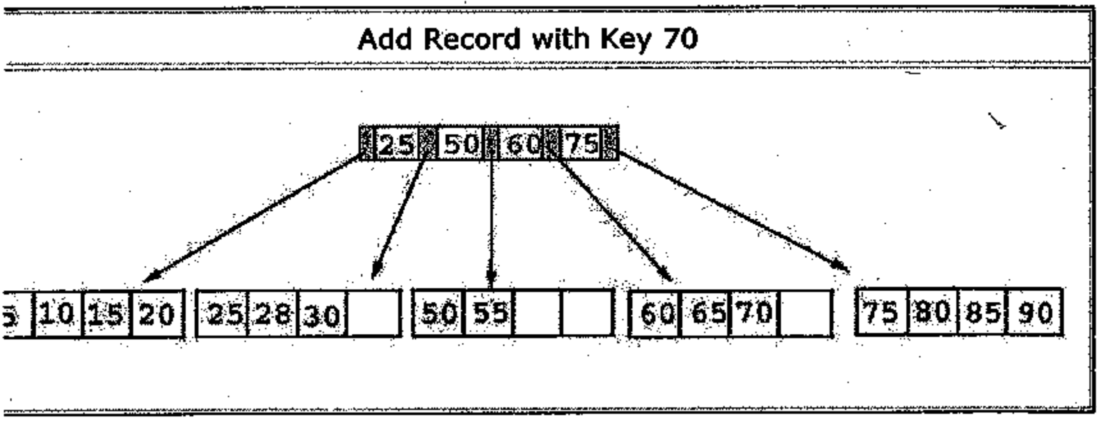
Adding a record when the leaf page is full but the index page is not

Next, we're going to insert a record with a key value of 70 into our B+ tree. This record should go in the leaf page containing 50, 55, 60, and 65. Unfortunately this page is full. This means that we must split the page as follows:

Left Leaf Page	Right Leaf Page
50 55	60 65 70

The middle key of 60 is placed in the index page between 50 and 75.

The following table shows the B+ tree after the addition of 70.


Add Record with Key 70

Adding a record when both the leaf page and the index page are full

As our last example, we're going to add a record containing a key value of 95 to our B+ tree. This record belongs in the page containing 75, 80, 85, and 90. Since this page is full we split it into two pages:

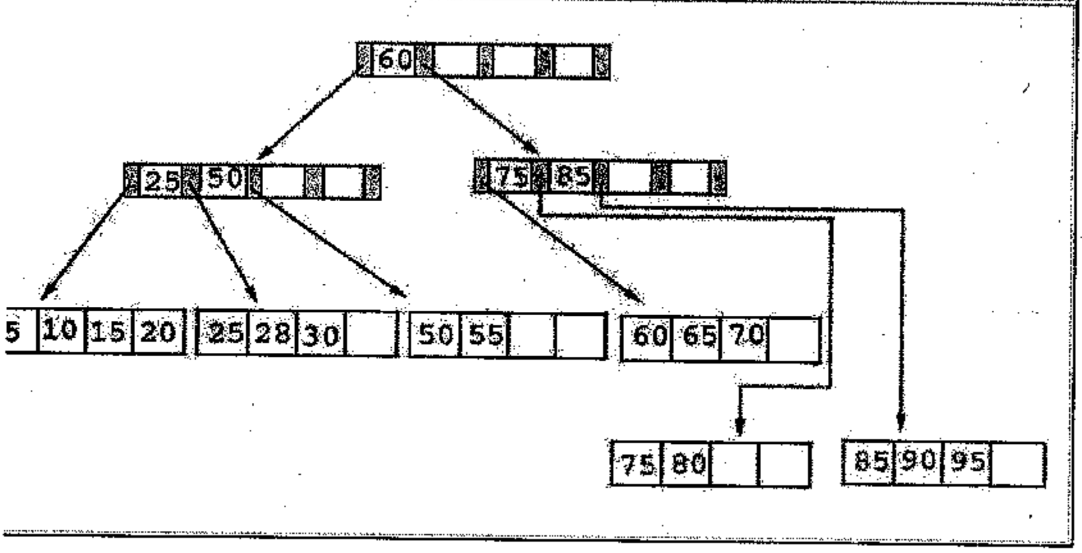
Left Leaf Page	Right Leaf Page
75 80	85 90 95

The middle key, 85, rises to the index page. Unfortunately, the index page is also full, so we split the index page:

Left Index Page	Right Index Page	New Index Page
25 50	75 85	60

The following table illustrates the addition of the record containing 95 to the B+ tree.

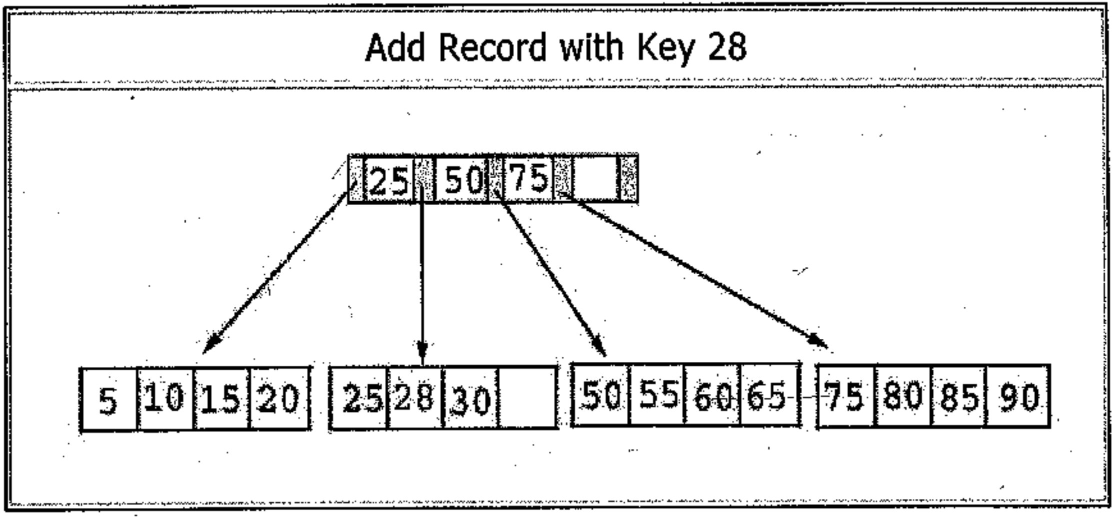
Add Record with Key 95



Rotation

B+ trees can incorporate rotation to reduce the number of page splits. A rotation occurs when a leaf page is full, but one of its sibling pages is not full. Rather than splitting the leaf page, we move a record to its sibling, adjusting the indices as necessary. Typically, the left sibling is checked first (if it exists) and then the right sibling.

As an example, consider the B+ tree before the addition of the record containing a key of 70. As previously stated this record belongs in the leaf node containing 50 55 60 65. Notice that this node is full, but its left sibling is not.



Add Record with Key 28

Using rotation we shift the record with the lowest key to its sibling. Since this key appeared in the index page we also modify the index page. The new B+ tree appears in the following table.

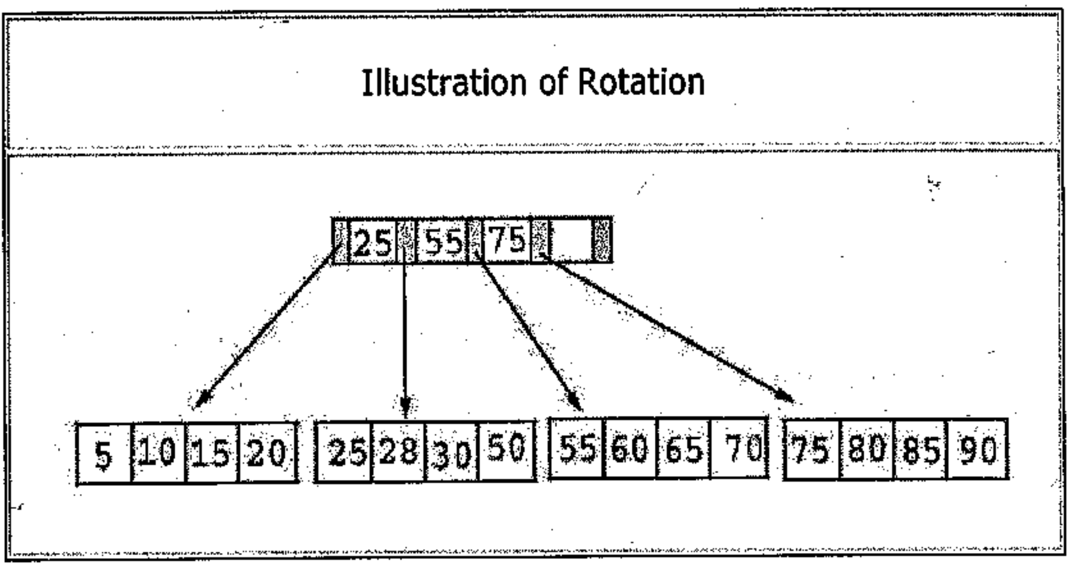


Illustration of Rotation

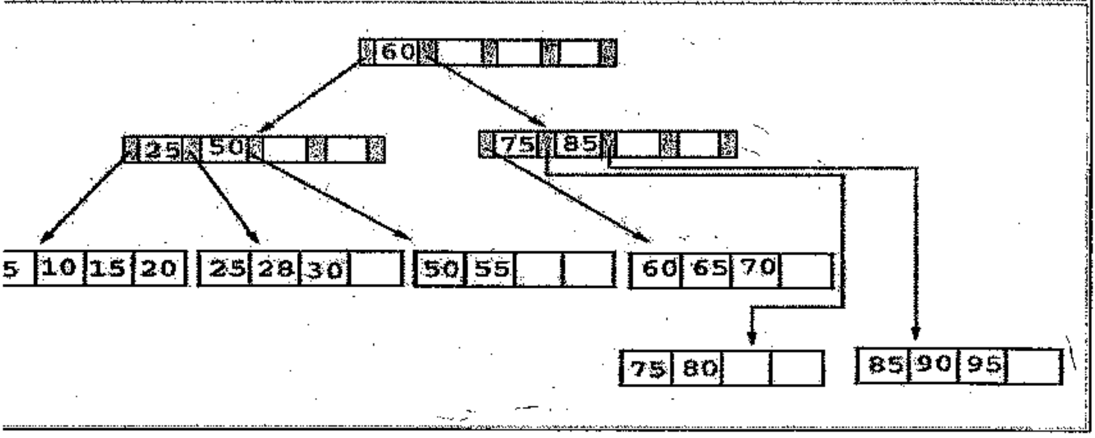
Deleting Keys from a B+ tree

We must consider three scenarios when we delete a record from a B+ tree. Each scenario causes a different action in the delete algorithm. The scenarios are:

The delete algorithm for B+ Trees		
Leaf Page Below Fill Factor	**Index Page Below Fill Factor**	**Action**
NO	NO	Delete the record from the leaf page. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index page, use the next key to replace it.
YES	NO	Combine the leaf page and its sibling. Change the index page to reflect the change.
YES	YES	• Combine the leaf page and its sibling. • Adjust the index page to reflect the change. • Combine the index page with its sibling. Continue combining index pages until you reach a page with the correct fill factor or you reach the root page.

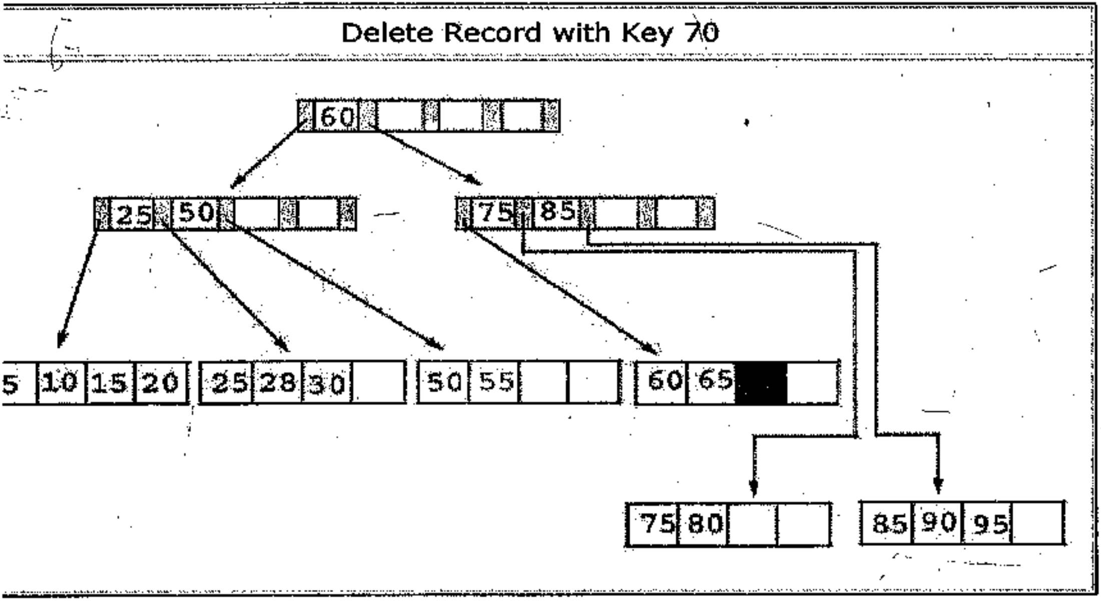
As our example, we consider the B+ tree after we added 95 as a key. As a refresher this tree is printed in the following table.

Add Record with Key 95



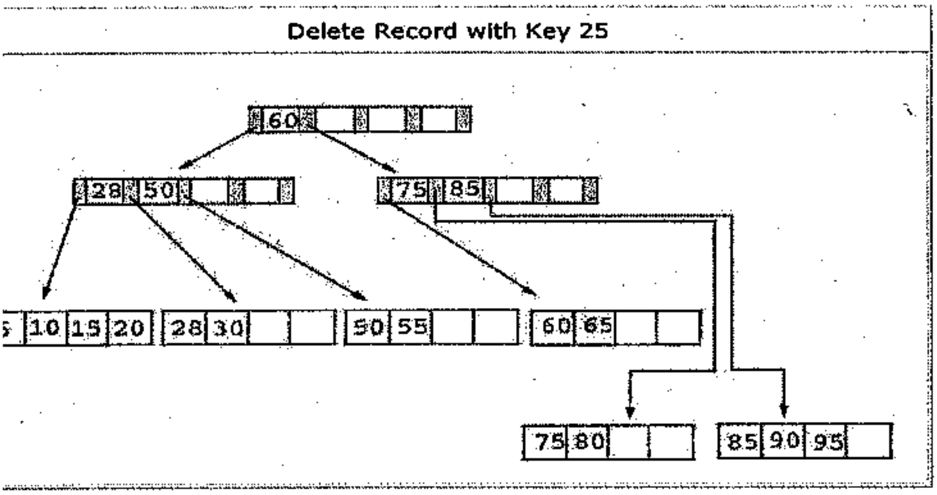
Delete 70 from the B+ Tree

We begin by deleting the record with key 70 from the B+ tree. This record is in a leaf page containing 60, 65 and 70. This page will contain 2 records after the deletion. Since our fill factor is 50% or (2 records) we simply delete 70 from the leaf node. The following table shows the B+ tree after the deletion.



Delete 25 from the B+ tree

Next, we delete the record containing 25 from the B+ tree. This record is found in the leaf node containing 25, 28, and 30. The fill factor will be 50% after the deletion; however, 25 appears in the index page. Thus, when we delete 25 we must replace it with 28 in the index page. The following table shows the B+ tree after this deletion.



Delete 60 from the B+ tree

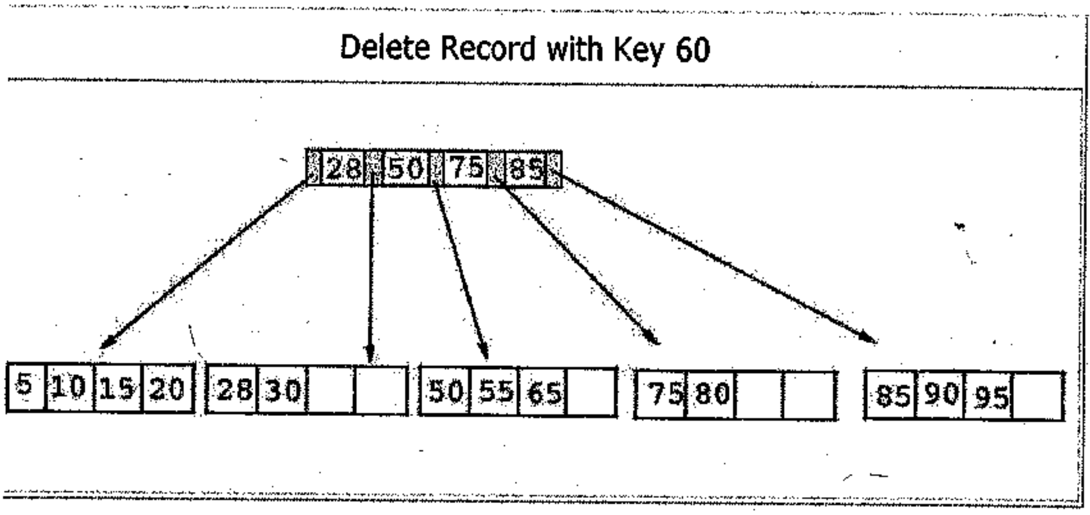
As our last example, we're going to delete 60 from the B+ tree. This deletion is interesting for several resasons:

1. The leaf page containing 60 (60 65) will be below the fill factor after the deletion. Thus, we must combine leaf pages.

2. With recombined pages, the index page will be reduced by one key. Hence, it will also fall below the fill factor. Thus, we must combine index pages.

Sixty appears as the only key in the root index page. Obviously, it will be removed with the deletion.

The following table shows the B+ tree after the deletion of 60. Notice that the tree contains a single index page.



Delete Record with Key 60

7.7 Trie and Dictionary

A trie is a multi way tree structure useful for sorting strings over an alphabet. It has been used to store large dictionaries of English words in spelling chacking programs and in natural-language understanding program.

A trie of order m is either empty or consisting of an ordered sequence of exactly m tries of order m. in a trie each node can contain m pointers that correspond to m posiable symbols in each of the node. If the symbols are numeric then there would be 10 pointers in the node and in case of alphabate there would be 26 pointers. Each pointer in a node is associated with a particular symbol value on the basis of its position in the node from lowest symbol value to highest symbol value. Tries are also known as lexicographic search trees.

Dictionary

In the data structure the sets manipulated by algorithms can grow, shrink or may change during course of time . these types of sets are called dynamic sets.

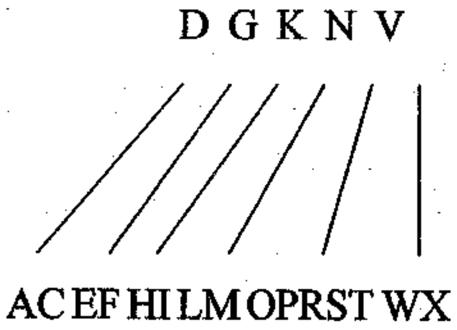
Different types of algorithms requires several different types of operations to be performed over the dynamic sets. For example many algorithms requires insertion operation,some deletion operation , other searching operation and many more. A dynamic set that supports these operations is called a dictionary.

One example of dictionary is number of dynamic sets implemented as part of the standard template library in c++. In STL we have number of abstract data types like list, vetor, deque , map etc that supports number of operations that causes them to grow or shrink. Similarly a good collection of dictionaries provided as part of the java that can be used by the programmer while developing applications.

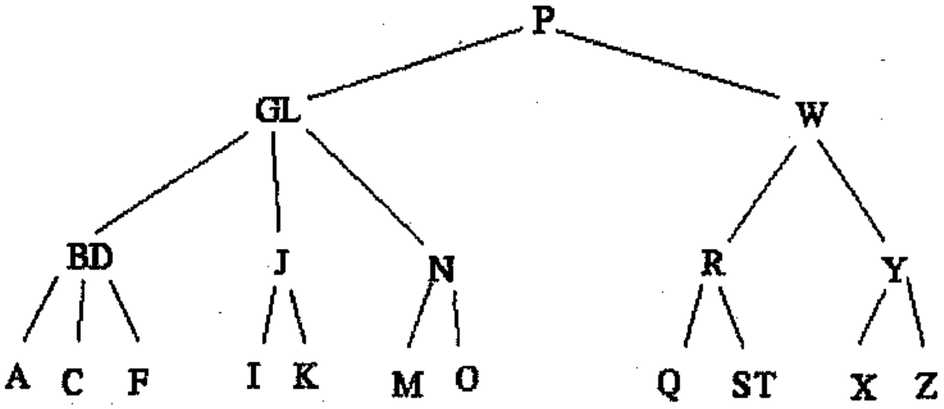
self learning exercises

1.What are the maximum number of keys that can be stored in a B-tree of height two (the root plus two additional levels) with a minimum branching factor of t=500?

2. For the following B-tree where t=3 show the result when "B" and then "Q" are inserted.



3. Consider the following B-tree where t=2. Show the B-tree that results when deleting "W". NOTE: The solution just shows the final answer.



7.8 Summary

- Threaded Tree, AVL Tree are the trees which can make tree operations more efficient.
- Multiway search trees are used when there is a large amount of data which is to be processed.

7.9 Glossary

Threaded tree: A binary search tree in which each node uses an otherwise-empty left child link to refer to the node's in-order predecessor and an empty right child link to refer to its in-order successor.

AVL tree: A balanced binary search tree where the height of the two subtrees (children) of a node differs by at most one. Look-up, insertion, and deletion are O(log n), where n is the number of nodes in the tree.

Multiway tree: A tree with any number of children for each node.

B-tree: A balanced search tree in which every node has between $\lceil m/2 \rceil$ and m children, where m>1 is a fixed integer. m is the order. The root may have as few as 2 children. This is a good structure if much of the tree is in slow memory (disk), since the height, and hence the number of accesses, can be kept small, say one or two, by picking a large m.

7.10 Further Readings

1. Gilles Brassard & Paul Bratley, Algorithmics, Prentice Hall, 1988
2. T. Cormen, C. Leiserson, & R. Rivest, Algorithms, MIT Press, 1990
3. Donald Knuth, The Art of Computer Programming (3 vols., various editions, 1973-81), Addison Wesley
4. Robert Kruse, Data Structures and Program Design , Prentice Hall, 1984

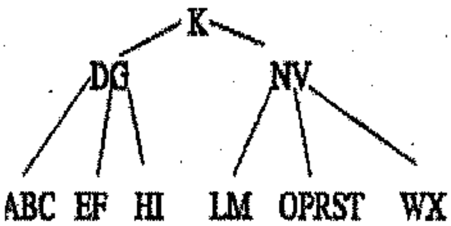
Udi Manber, Introduction to Algorithms, Addison Wesley, 1989

7.11 Answers to self learning exercises

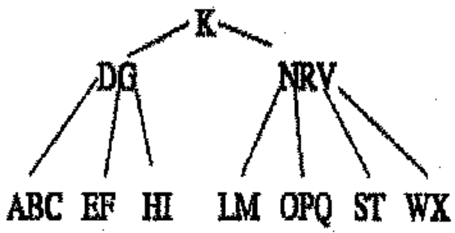
1.To maximize the number of keys stored in a B-tree with t=500, each node will have 999 keys and 1000 children. Thus the number of nodes in such a B-tree of height two will be 1+1000+ 1000^2 = 1,001,001. Thus the number of keys is 999 x 1,001,001 = 999,999,999

2.

After inserting B:

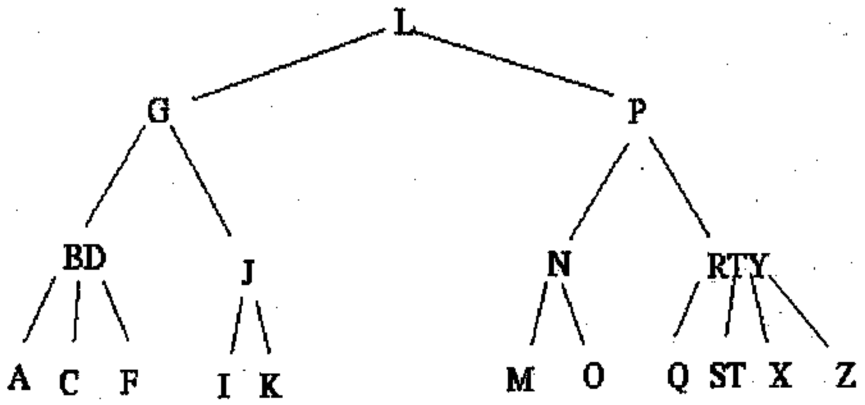


After inserting Q:



3.

After deleting W we obtained:



7.12 Unit End Questions

1 What is AVL Tree?
2 Who invented AVL Tree?
3 How can we determine the balance factor?
4 Insert a node 3 in the AVL Tree of the tutorial and try to rebalance the tree by any of the methods?
5 How an AVL Tree or B - Tree can be better than a Binary Search Tree?
6 How an AVL Tree or B - Tree can be better than a Binary Search Tree?
7 What advantages does a multiway search Tree have over an AVL Tree?
8 What are 5the differences between a multiway search tree & a B - Tree?

--- ***----

Harkirat 23-11

UNIT VIII

GRAPH THEORY FUNDAMENTALS

STRUCTURE OF THE UNIT

8.0 Objectives

8.1 Introduction

8.2 Definitions

8.3 Types of Graphs

8.4 Data Structure for Graph Representation

8.5 Graph Traversal Algorithms

8.5.1 Breadth First Traversing

8.5.2 Depth First Traversing

8.6 Shortest Path Algorithm

8.6.1 Dijkstra Single-Source Shortest Path Algorithm

8.6.2 Bellman-Ford Single-Source Shortest Path Algorithm

8.6.3 Floyd-Warshall All Pair Shortest Path Algorithm

8.7 Minimum Spanning Tree (MST)

8.7.1 Prim Algorithm for MST

8.7.2 Kruskal Algorithm for MST

8.8 Summary

8.9 Glossary

8.10 Further Readings

8.11 Answers to self learning exercises

8.12 Unit–end Questions

8.0 Objectives

This unit provides an overview of

- Importance of Graphs and their related definitions
- Graph representation formats
- Graph Traversing algorithms

Minimum Spanning Tree Algorithms

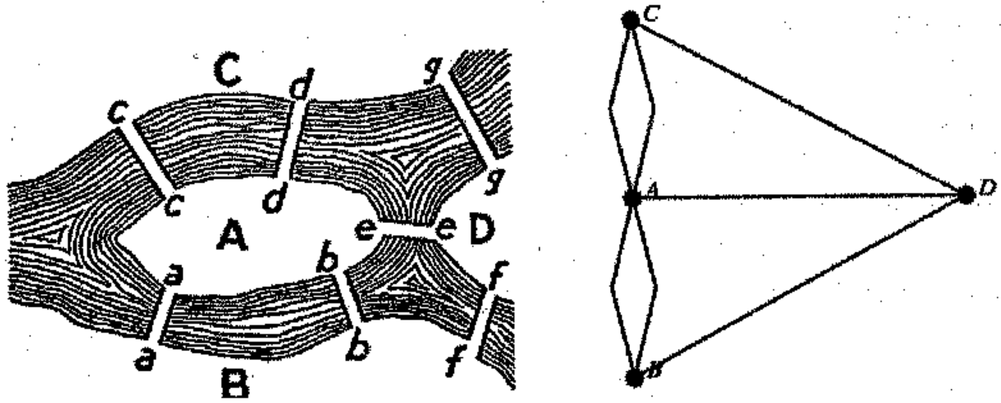
Shortest Path Algorithms

8.1 Introduction

There was a great puzzle in Königsberg city (now known as Kaliningrad) of East Prussia in Russia. The city is built around the River Pregel where it joins another river. An island named Kniephof is in the middle of where the two rivers join. There are seven bridges that join the different parts of the city on both sides of the rivers and the island as shown in Figure 8.1(a).

People tried to find a way to walk all seven bridges without crossing a bridge twice, but no one could find a way to do it.

The problem came to the attention of a Swiss mathematician named Leonhard Euler (pronounced "oiler"). In 1735, Euler presented the solution to the problem before the Russian Academy. He explained why crossing all seven bridges without crossing a bridge twice was impossible. While solving this problem, he developed a new mathematics field called graph theory, which later served as the basis for another mathematical field called topology.



Figure

8.1 : (a) Seven Bridges on river Pregel of Kõnisberg (b) Euler Representation

Euler simplified the bridge problem by representing each land mass as a point and each bridge as a line as shown in Figure 8.1(b). He reasoned that anyone standing on land would have to have a way to get on and off. Thus each land mass would need an even number of bridges. But in Konigsberg, each land mass had an odd number of bridges. This was why all seven bridges could not be crossed without crossing one more than once. One walk over the four bridges without crossing any bridge more than once is shown in Figure 8.2(a) and its Euler representation is shown in Figure 8.2(b).

The problem could have been solved if ONE bridge was removed or added. Which bridge would you remove? Where could you add a bridge?

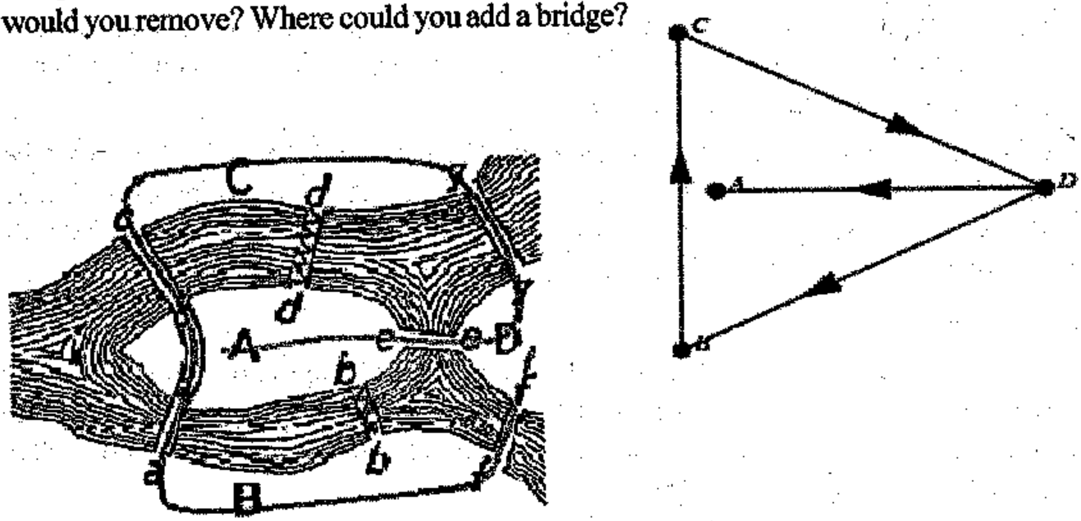


Figure 8.2: (a) A walk over four Bridges (b) Euler Representation of walk

Euler went on to generalize this mode of thinking, laying a foundation for graph theory. Euler representation as points and lines was used in solving a large number of real life problems like telephone network planning, collection of post from post offices by van, VLSI design problems etc. Euler representation as shown in Figure 8.1(b) and Figure 8.2(b) are known as graph. The points of graph are known as vertex or node and lines are known as edges in modern graph theory. Standard methods have been developed to solve various graph related problems. Various real-life problems can be modelled as graph and standard methods available in graph theory can be used to solve these problems.

This unit describes fundamental definitions of graph theory, types of graphs, their computerized representations and simple puzzles related to graph.

8.2 DEFINITIONS

As discussed earlier, a graph is a diagram consisting of points, called vertices or nodes, joined together by lines, called edges. Figure 8.3 shows a graph containing u, v, w and z as 4 vertices joined by {(u, v), (u, w), (v, w), (w, z)} as 4 edges. Formal definition and various types of graphs are defined in this section.

graph an edge can connect one or more vertices (even more than two). This edge is called hyperedge and the graph containing hyperedge is called hypergraph.

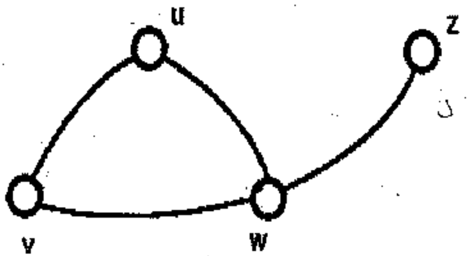


Figure 8.3: A Typical Graph G

8.2.1 Notations of Graph

A graph G = (V, E) consists of a (finite) set denoted by V or by V(G) (if one wishes to make clear which graph is under consideration) and a collection E, or E(G), of unordered pairs {u, v} of distinct elements from V. Each element of V is called a vertex or a point or a node, and each element of E is called an edge or a line or a link.

Formally, a graph G is an ordered pair of disjoint sets (V, E), where E Í V × V. Set V is called the vertex or node set, while set E is the edge set of graph G. The graph shown in Figure 8.3 consists of {u, v, w, z} vertices set and {(u, v), (u, w), (v, w), (w, z)}. Typically, it is assumed that self-loops (i.e. edges of the form (u, u), for some u Î V) are not contained in a graph.

8.2.2 Cardinality

The number of vertices, the cardinality of V, is called the order of graph and devoted by |V|. We usually use n to denote the order of G. The number of edges, the cardinality of E, is called the size of graph and denoted by |E|. We usually use m to denote the size of G. Cardinality of V as well as Cardinality of E of graph shown in Figure 8.3 is 4.

8.2.3 . Neighbour Vertex and Neighbourhood

We write v_i, v_jÎ E(G) to mean (v_i, v_j)Î E(G), and if e = (v_i, $v_{j)}$ Î E(G), we say v_i and v_j are adjacent. Vertex v is adjacent to u, but not w in Figure 8.3.

Formally, given a graph G = (V, E), two vertices vi, v_j Î V are said to be neighbours, or adjacent nodes, if (v_i, v_j) Î E. If G is directed, we distinguish between incoming neighbours of v_i (those vertices v_j Î V such that (v_j, v_i) Î E) and outgoing neighbours of v_i (those vertices v_jÎV such that (v_i, v_j) Î E). In a directed graph shown in Figure 8.4, incoming neighbour of all vertices is 1 whereas outgoing neighbour of w is 2 and others have outgoing neighbour as 1.

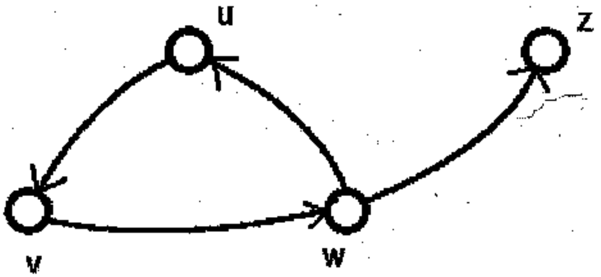


Figure 8.4: A Directed Graph

The open neighbourhood N(v) of the vertex v consists of the set of vertices adjacent to v, that is, N(v) = {w Î V : (v, w) Î E}. In Figure 8.3, open neighbourhood of z is 1, w is 3 and u and v is 2 in Figure 8.3. The closed neighbourhood of v is N[v] = N(v) È {v}. Closed neighbourhood of z is 2, w is 4 and u and v is 3. Similarly, open neighbourhood of For a set S Í V, the open neighbourhood N(S) is defined to be È$_{vls}$ N(v), and the closed neighbourhood of S is N[S] = N(S) È S.

8.2.4 Vertex Degree

The degree **deg(v)** of vertex v is the number of edges incident on v or equivalently, deg(v) = |N(v)|. For example, vertex w in Figure 8.3 has degree 3.

In Figure 8.3, the graph G, shown has deg(u) =2, deg(v) =2, deg(w) =3 and deg(z) =1.

The degree sequence of graph is (deg(v_1), deg(v_2), ..., deg(v_n)), typically written in nondecreasing or nonincreasing order. The minimum and maximum degree of vertices in V(G) are denoted by d(G) and "(G), respectively. If d(G) = "(G) = r, then graph G is said to be regular of degree r, or simply r-regular.

Formally, given a graph G = (V, E), the degree of a vertex v Î V is the number of its neighbours in the graph. That is,

deg(v) = |{u Î V : (v, w) Î E}|.

If G is directed, we distinguish between in-degree (number of incoming neighbours) and out-degree (number of outgoing neighbours) of a vertex. In directed graph shown in Figure 8.4, in-degree(u)=1, out-degree(u)=1, in-degree(w)=1 and out-degree(w)=2.

8.3 Types of Graph

Graph theory has been able to model a large number of natural problems into graph. Accordingly, there are many types of graphs and their components available in graph theory. There can be many criteria to classify graphs. This section describes three classifications of graphs.

8.3.1 Classification based on Edge Connectivity

There are four types of graphs classified based on edge connectivity. Each one is described below. It may be noted that most of the algorithms are developed for simple graph as majority of the problems can be solved using this type of graph only.

8.3.1.1 Simple Graph

A graph containing vertices and edges connecting only a pair of vertices is called simple graph. We specify a simple graph by its set of vertices and set of edges, treating the edge set as a set of unordered pairs of vertices and write e = (u, v) (or e = (v, u)). For an edge e = (u, v), endpoints are u and v. When u and v are endpoints of an edge, they are adjacent and are neighbours.

This type of graphs are simplest ones and mostly used in graph theory. Most of the applications and algorithms are developed for this type of graphs only. By mentioning graph means simple graph only. This unit also deals with simple graphs only.

8.3.1.2 Multi-Graph

A graph, that can permit multiple edges between a pair of vertices, is called Multi-Graph. Remaining properties are the same as simple graph.

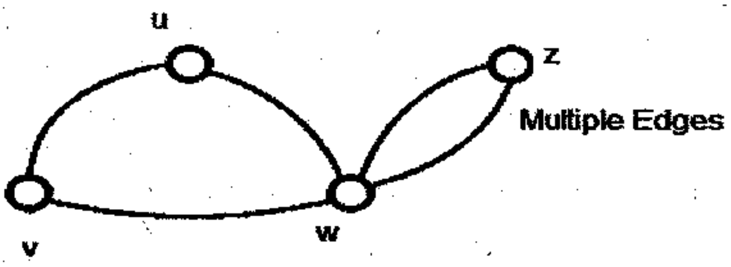


Figure 8.5: An example of Multi-Graph

Figure 8.5 shows a multi-graph having multiple edges between w and z vertices.

8.3.1.3 Graph with Loops

A graph that permits self-loop also is called graph with self-loop. A self-loop is an edge which starts and ends at the same vertex.

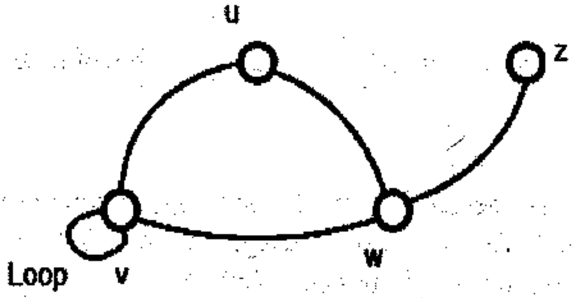


Figure 8.6: A Graph with self-loop

Figure 8.6 shows a graph having a self-loop at vertex v.

8.3.1.4 Hypergraph

This type of graph is very important for VLSI Design related problems. In this type of graph an edge can connect one or more vertices (even more than two). This edge is called hyperedge and the graph containing hyperedge is called hypergraph.

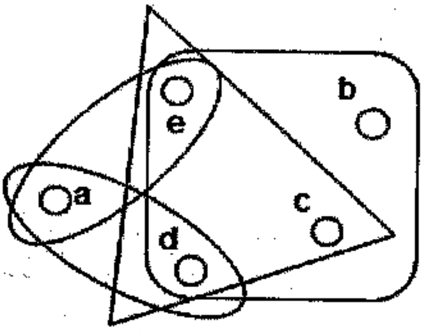


Figure 8.6: A Hypergraph with 4 hyperedges

Figure 8.6 shows a hypergraph having 5 vertices namely a, b, c, d and e and 4 hyperedges connecting them. Two hyperedges are represented as eclipses. One of them is connecting a and e, and another is connecting a and d. Third hyperedge is shown as triangle connecting 3 vertices namely c, d and e. Finally rectangle of the Figure shows fourth hyperedge connects four vertices b, c, d and e.

8.3.2 Classification based on Direction

The graphs can also be classified based on the direction of the edges. An edge starts from one vertex and ends at another vertex. An arrow indicates the direction of the edge shown at ending vertex. Based on direction, graphs can be categorised as directed or undirected.

8.3.2.1 Directed Graph

In directed graph, the edges start from one node and end at other node. At ending node an arrow mark is shown to indicate direction. The edge is counted in in-degree of the vertex where arrow of the edge ends and as out-degree where edge starts. The graph in Figure 8.4 is a directed graph.

8.3.2.2 Undirected Graph

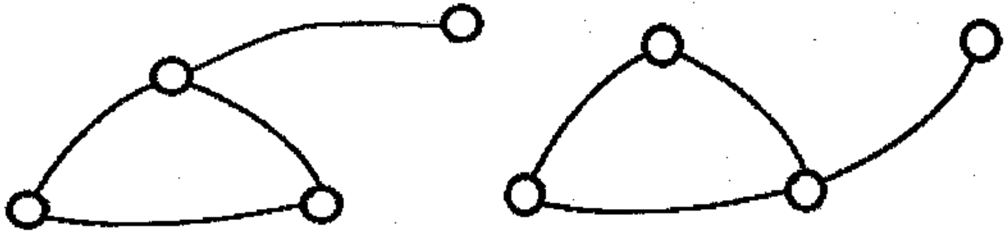
In undirected graph, there is no direction of edge is shown and edge is counted in degree of both the vertices. The graph shown in Figure 8.3 is undirected graph. When not specified, graph is treated as undirected.

8.3.3 Classification based on Weight or Label

In the graphs shown above edges indicate merely connectivity of vertices. But, graphs can do more than connectivity of vertices. There can be a number associated with edge or vertex representing some requirement of problem. The number is called weight. Graphs can also be classified based on weight.

8.3.3.1 Unlabeled Graph

In unlabeled graph, vertices are not named rather each vertex is equally likely.



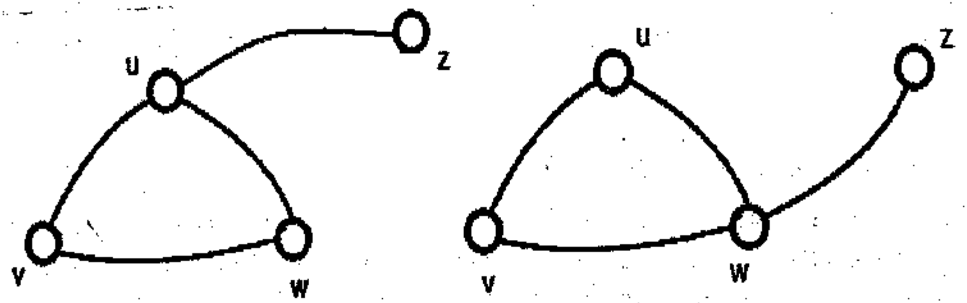
(a) Unlabeled Graph **(b) Similar Unlabeled Graph**

Figure 8.7: Two unlabeled indistinguishable Graphs

Figure 8.7 shows two unlabeled and indistinguishable graphs as names to vertices are not assigned. However, they become different names are assigned to vertices.

8.3.3.2 Labelled Graph

A graph, in which a unique name is assigned to every vertex, is called labelled graph.



(a) Labelled Graph **(b) Different labelled Graph**

Figure 8.8: Two labelled and different Graphs

Figure 8.8 shows two different labelled graphs. These graphs were the same when no name was assigned to vertices.

8.3.3.3 Unweighted Graph

This type of graph shows the edges connect adjacent vertices only. There is no number associated with edges or vertices. All graphs shown in Figure 8.3 to 8.8 are unweighted graphs.

8.3.3.4 Weighted Graph

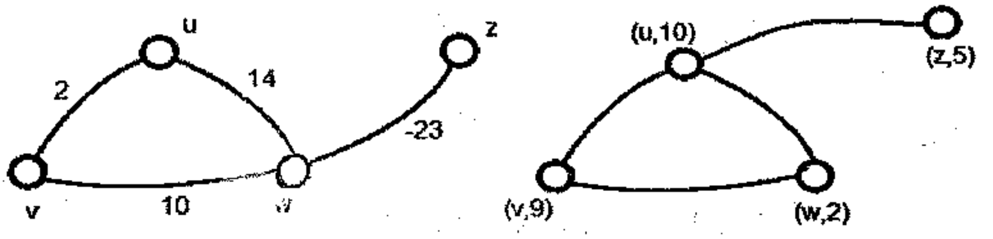
Weighted graphs are important graphs in which a number is associated with edge or vertex. This number represents a parameter of interest. Depending on application, this could be a distance, cost, capacity etc. There are many types of weight possible in graph as described below.

Edge Weighted Graph

In these graphs weight is associated with edges as shown in Figure 8.9(a). Weight of edge (u, w) is 14 in Figure 8.9(a).

Vertex Weighted Graph

In these graphs weight is associated with vertices as shown in Figure 8.9(b). Weight associated with vertex u is 10 in Figure 8.9(b).



(a) Edge Weighted Graph **(b) Vertex weighted Graph**

Figure 8.9: Edge Weighted and Vertex Weighted Graphs

Positive Weights

Mostly weights associated with edge or vertex is positive as shown in Figure 8.9 except for edge (w, z) in Figure 8.9(a).

Negative Weights

Weights associated with edge or vertex can be negative also as shown for edge (w, z) in Figure 8.9(a).

Additive Weights

While traversing edges in a graph, total distance or cost needs to be computed. This computation is normally done by adding the weights associated with edges are added. Such weights are called additive weights. This is also applicable for vertex weights. In Figure 8.9(a), traversing u to v to w costs 12 if weights are additive.

Multiplicative Weights

Weights associated with vertex or edge can be multiplicative also. In other worlds, the total cost of traversal can be obtained by multiplying weights of edges (or vertices) from source to route. In Figure 8.9(a), traversing u to v to w costs 20 if weights are multiplicative.

8.3.4 Classification based on Connectivity

The graphs can also be classified based on connectivity. There are two types of graph namely, connected and disconnected graphs as described below.

8.3.4.1 Connected Graph

A graph in that is in one piece is said to be connected. In other words, a graph is said to be connected if every vertex can be reachable from any other vertex by traversing edges.

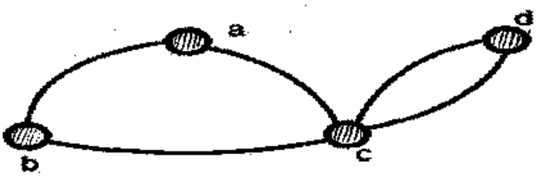


Figure 8.10: A Connected Graph

Figure 8.10 shows a connected graph in which every vertex is reachable from every other vertex through a set of edges. Vertex d is reachable from vertex b through edges (b, c) and (c, d).

8.3.4.2 Disconnected Graph

A graph, that is not a single piece, is called disconnected graph. In such graphs, there exists a pair of vertices not reachable through a set of graph edges.

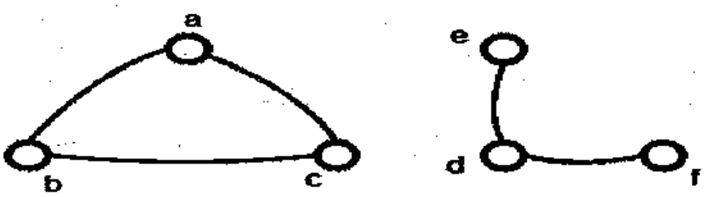


Figure 8.11: A Disconnected Graph

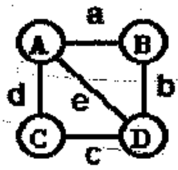
Figure 8.11 shows a disconnected graph in which vertex pair b and d is not reachable through any set of edges of the graph.

8.4 Data Structure for Graph Representation

There are many data structures available for representations of the graphs. Depending on application requirement, appropriate representation is used. This section describes four commonly used representations.

8.4.1 Incidence Matrix

Incidence matrix representation is useful for representing connectivity of the graph. Columns represent edge names and rows represent vertex names. Matrix element value **1** indicates that edge represented by column and vertex represented by row is connected. Similarly matrix element value **0** is written when vertex is not end point of corresponding edge. This type of representation is very useful for multi-graphs. Usually every column of the incidence matrix contains two 1's only. Incidence matrix require O(|V |*|E |) space.



	a	b	c	d	e
A	1	0	0	1	1
B	1	1	0	0	0
C	0	0	1	1	0
D	0	1	1	0	1

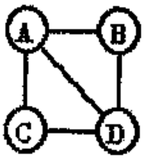
Figure 8.42: Incidence Matrix representation of Graphs.

Figure 8.42 shows a graph having 4 vertices and 5 edges. Accordingly, there are 5 columns and 4 rows 5 columns in its incidence matrix. Matrix element value is 1 in column number 3 for row number 3 and 4. This indicates vertex C (for row number 3) and vertex D (for row number 4) are connected through edge c.

8.4.2 Adjacency Matrix

In adjacency matrix, rows and columns are indexed by the vertices only. Hence, adjacency matrices are square matrices. These matrices can represent directed as well as undirected graphs and weights as well as connectivity. But, these matrices cannot be used to represent multi-graphs.

To represent connectivity, the matrix element entries are just boolean values stating that vertices represented by row number and column number are connected or not.

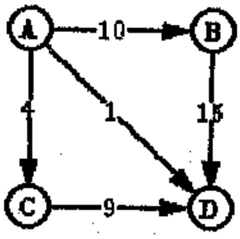


$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 1 & 1 \\ B & 1 & 0 & 0 & 1 \\ C & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 1 & 0 \end{array}$$

Figure 8.43: Adjacency Matrix representation of Graphs for connectivity.

Figure 8.43 shows adjacency matrix representation for connectivity of the same graph as shown in Figure 8.42.

In case of weighted and directed graphs, the weights themselves can be written into the entries.



$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & 10 & 4 & 1 \\ B & \infty & \infty & \infty & 15 \\ C & \infty & \infty & \infty & 9 \\ D & \infty & \infty & \infty & \infty \end{array}$$

Figure 8.44: Adjacency Matrix of Directed and Weighted Graphs.

Figure 8.44 shows adjacency matrix representation of directed and weighted graph. Vertices not connected through edges are shown as having weight 8.

Adjacency matrices require $O(|V|^2)$ space, and so they are space-efficient only when they are dense (that is, when the graphs have many edges). Time-wise, the adjacency matrices allow easy addition and deletion of edges.

8.4.3 Adjacency Lists

Adjacency List representation of the graph consists of a list of vertices with each vertex containing a list of its neighbouring vertices.

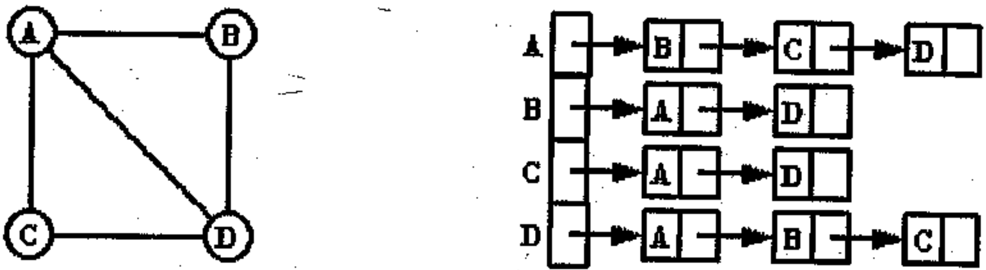


Figure 8.45: Adjacency List representation of Graphs.

Figure 8.45 shows adjacency list representation for connectivity of the same graph as shown in

Figure 8.42.

This representation takes O(|V |+|E|) space. Hence, this is an space efficient representation suitable even for sparse graphs.

8.4.4 Adjacency Mutli-lists

In adjacency multi-list, it is easy to find the vertices adjacent a given vertex even in directed graphs, which is difficult in adjacency list. It takes slightly more space compared to adjacency list.

Adjacency multi-list representation needs two data structures as mentioned below:

- An array to represent vertices called headnodes.
- A structure to store the following information
  - Edge Name
  - Endpoint-1
  - Endpoint-2
  - EdgeList-1 for endpoint-1
  - EdgeList-2 for endpoint-2

The above structure is shown in Figure 8.46.

	end-point	end-point	Edge-list	Edge-list
m	*vertex*1	*vertex2*	*list*1	*list*2

Figure 8.46: Data Structure for edge information in adjacency multi-list.

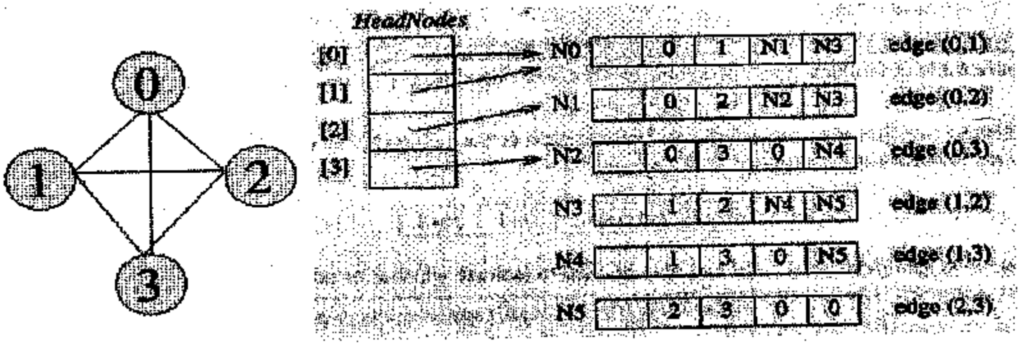

0
1
2
3
HeadNodes
[0]
[1]
[2]
[3]
N0 | 0 | 1 | N1 | N3 | edge (0,1)
N1 | 0 | 2 | N2 | N3 | edge (0,2)
N2 | 0 | 3 | 0 | N4 | edge (0,3)
N3 | 1 | 2 | N4 | N5 | edge (1,2)
N4 | 1 | 3 | 0 | N5 | edge (1,3)
N5 | 2 | 3 | 0 | 0 | edge (2,3)

Figure 8.47: A graph and its adjacency multi-list.

Figure 8.47 shows a graph and its adjacency multi-list. Each vertex pointer HeadNode array points to first edge list in which vertex appears. For example, vertex 2 first appears in edge (0, 2). Hence, vertex 2 array element points to N1 edge record. Tracing of edges connected to a vertex is shown below. For example, vertex number 3 first appears N2 edge record at second position. Corresponding edgelist pointer points to N4 record, vertex number 3 is again second position. Hence, it gets a pointer at N5. As there is 0 written at last place in N5, tracing ends here.

The Lists are	Vertex 0:	N0? N1? N2
	Vertex 1:	N0? N3? N4
	Vertex 2:	N1? N3? N5
	Vertex 3:	N2? N4? N5

Figure 8.48: Adjacency multi-list representation of graph in Figure 8.47.

8.5 Graph Traversal Algorithms

A graph search (or graph traversal) algorithm is just an algorithm to systematically go through all the nodes in a graph, often with the goal of finding a particular node, or one with a given property. Searching a linear structure such as a list is easy as one can just start at the beginning, and work through to the end. Searching a graph is obviously more complex and need systematic approach for traversal. There are two common algorithms are used for traversal namely, Breadth First Search (BFS) and Depth First Search (DFS). Both of them visit each vertex in different order. The algorithms are described in following subsections.

8.5.1 Breadth First Search (BFS) Algorithm: This algorithm systematically visits all nodes starting from starting vertex. The basic idea is to visit neighbouring vertices first then vertices far from current vertex. This can further be supported by assigning a level of vertex as number of edges in the path of vertex from starting vertex. Visit the vertices at level i before the nodes of level i+1.

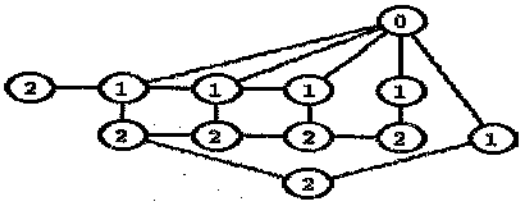


Figure : A graph with level number from starting vertex 0

As algorithm progresses, starting vertex is marked as visited and said to be unexplored. Algorithm marks a vertex as explored by an algorithm when the algorithm has visited all vertices adjacent from it. All unvisited vertices adjacent to current vertex are marked visited put onto the end of the list of unexplored vertices and current vertex is declared explored. First vertex from list of unexplored vertices is the next to be explored. Exploration continues till no unexplored vertex left. The tree formed by connecting all vertices and edges used to visit vertex first time is called BFS tree.

Algorithm BFS (G, v)

Inputs Graph G(V,E) and starting vertex v

Output Graph G with all vertices marked visited

—A BFS of G starts at vertex v.

--All vertices visited are marked as VISITED (i) = 1.

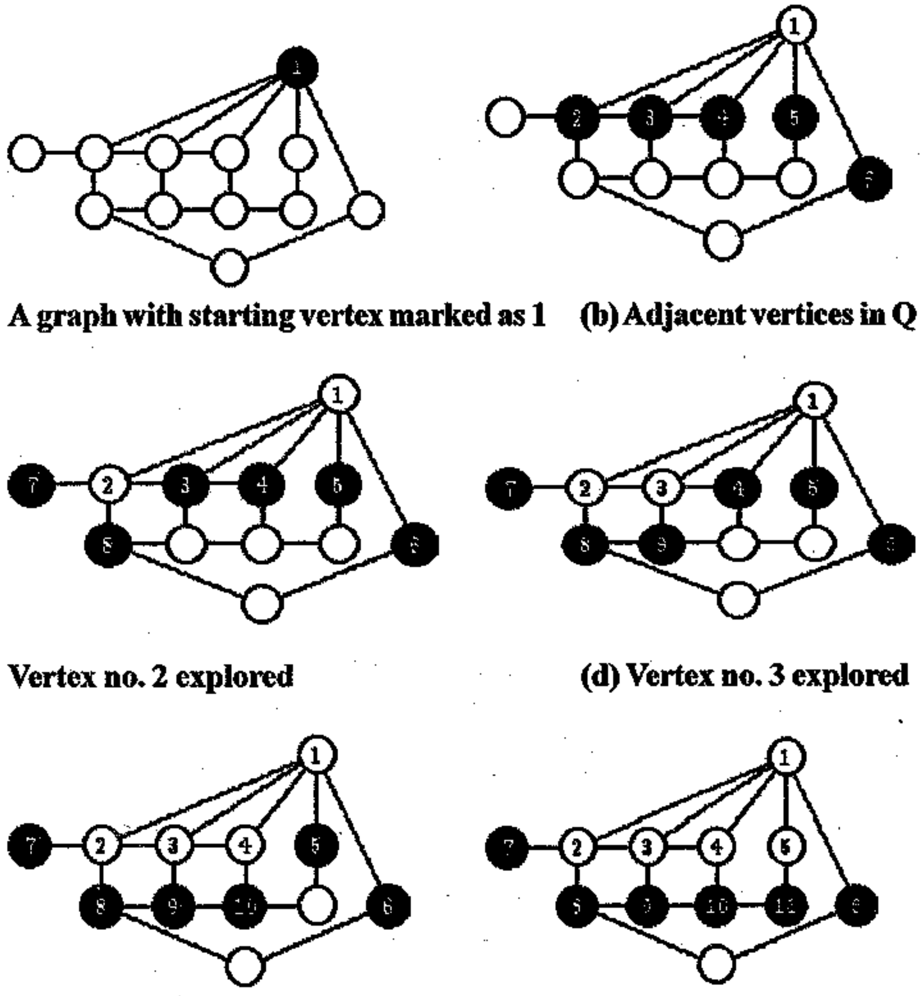
-- G and array VISITED are global and VISITED is initialized to zero.

```
1.    VISITED (v)! 1; u ! v
2.    Initialize Q as empty queue — Q is of unexplored Vertices
3.    Loop
```

```
                For all vertices w adjacent from u do
                    If VISITED(w) = 0 then call ADDQ(w, Q)— w is unexplored
                    VISITED (w) ! 1
            Endif
        Endfor
        If Q is empty then return Endif – no unexplored vertex
        Call DELETEQ(u, Q)          — get first unexplored vertex
    Endloop
```

Algorithm: BFS Algorithm

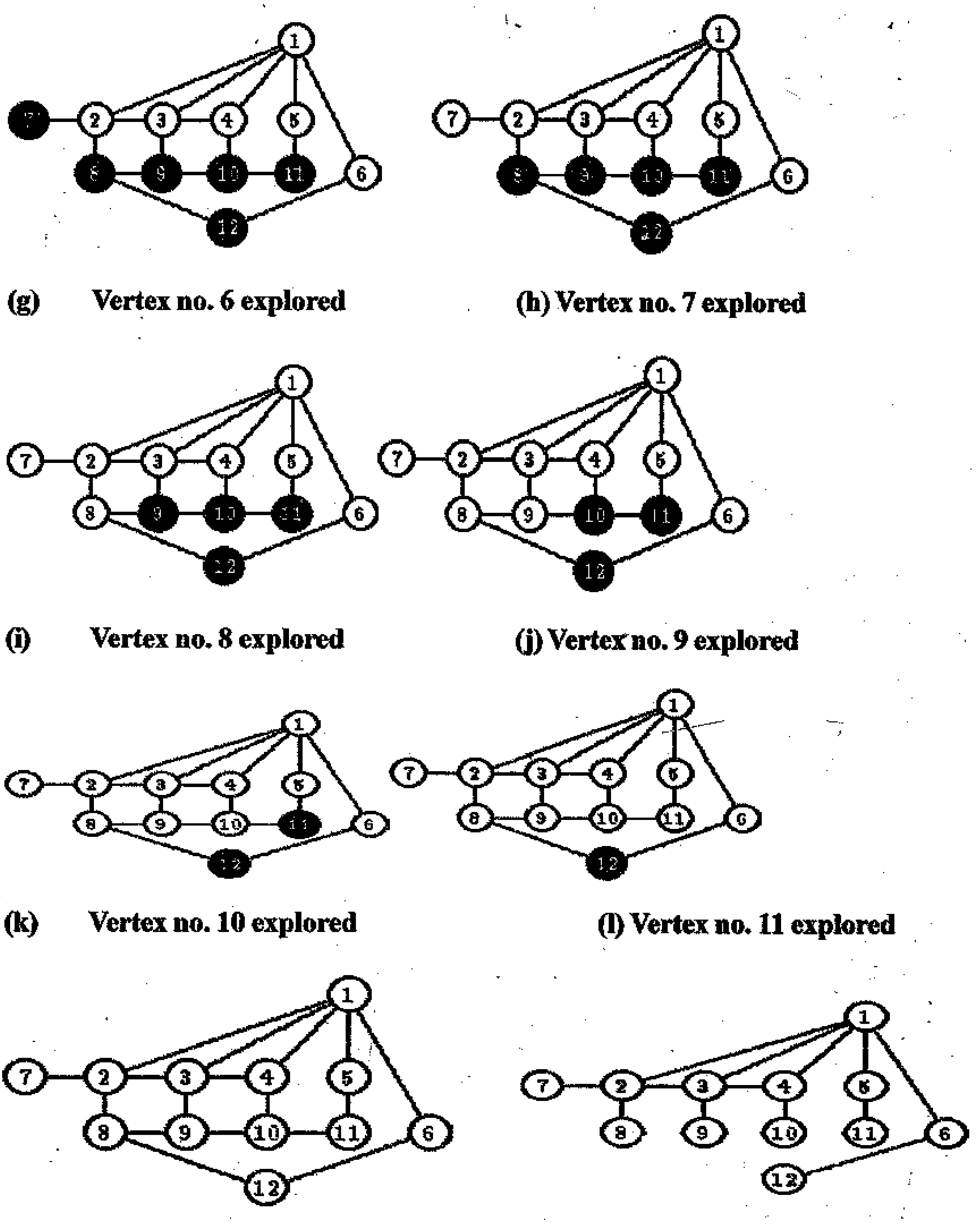
As algorithmic steps are self-explanatory, it is better to explain them through example as described in Figure 9.2. Black vertices indicate that they are in Q.



(a) A graph with starting vertex marked as 1 (b) Adjacent vertices in Q

(c) Vertex no. 2 explored (d) Vertex no. 3 explored

(e) Vertex no. 4 explored (f) Vertex no. 5 explored



(g) Vertex no. 6 explored

(h) Vertex no. 7 explored

(i) Vertex no. 8 explored

(j) Vertex no. 9 explored

(k) Vertex no. 10 explored

(l) Vertex no. 11 explored

(m) Finally Vertex no. 12 explored

(n) BFS Tree

Figure (a)-(n): An example and steps in BFS and BFS Tree

Analysis of BFS

It is obvious that each node is added in queue by ADDQ and, therefore dequeued once and only once by DELETEQ. Individual queue operations are O(1) and so the total time spent on queue operations is O(|V|). The adjacency list for each vertex taken off the queue is scanned once. Hence the total time spent scanning adjacency lists is O(|E|). Hence the total time spent by BFS is O(|V|+|E|).

If **Adjacency Matrix** is used, then a Queue is needed to maintain unexplored vertices & an array for maintaining the state of a vertex. Scanning each row for checking the connectivity of a Vertex is in order O(|V|). So, Complexity is O(|V|^2).

8.5.2 Depth First Search (DFS) Algorithm

Like BFS, DFS also visits each and every vertex of the graph, but in different order. DFS is a recursive algorithm that implicitly records a "backtracking" path from the start vertex to the vertex currently under consideration. Unlike BFS, only one of the adjacent vertices is visited and path is traced from last visited vertex. In other words, exploration of current vertex is suspended the moment new vertex is reached. Exploration will resume only after new vertex has been explored. Hence, stack is used for backtracking instead of queue used in BFS. The tree formed by connecting all vertices and edges used to visit vertex first time is called DFS tree. Algorithm steps are detailed below.

Algorithm DFS (G, v)

Inputs Graph G(V, E) and starting vertex v

Output Graph G with all vertices marked visited

— A DFS of G starts at vertex v.

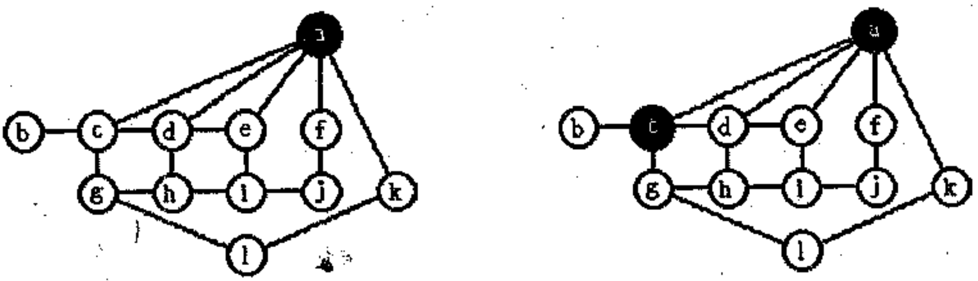
– All vertices visited are marked as VISITED (i) = 1.

– G and array VISITED are global and VISITED is initialized to zero.

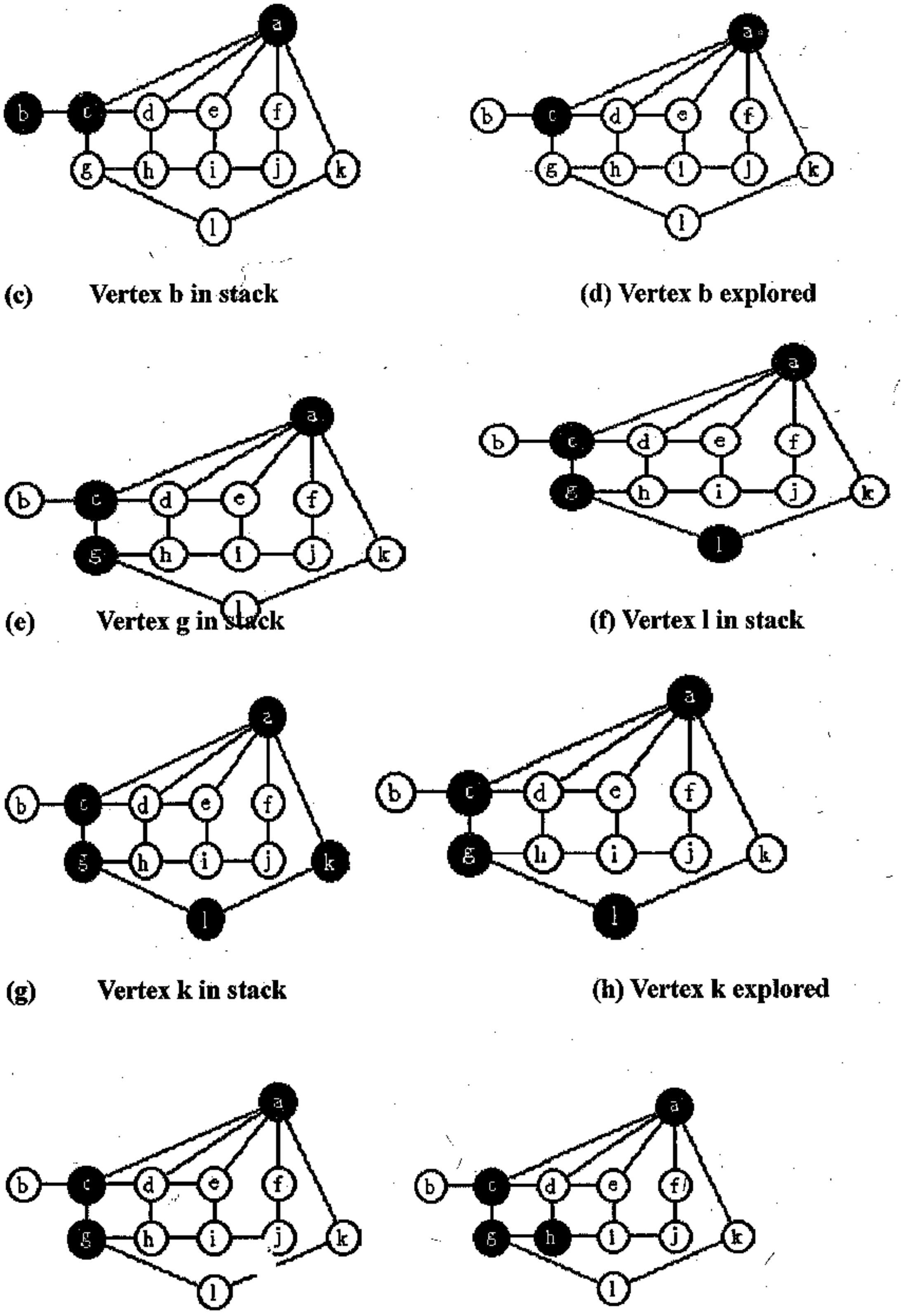
```
VISITED (v)! 1
For each vertex w adjacent from v do
          If VISITED(w) = 0 then
          call DFS(G, w)
 Endif
Endfor
```

Algorithm DFS Algorithm

As algorithmic steps are self-explanatory, it is better to explain them through example as described in Figure 9.3. Vertex details will be stored in stack whenever DFS is called recursively. Black vertices indicate that they are in stack and will be explored later.



(a)A graph with starting vertex marked **(b) Adjacent vertices in stack**



(c) Vertex b in stack

(d) Vertex b explored

(e) Vertex g in stack

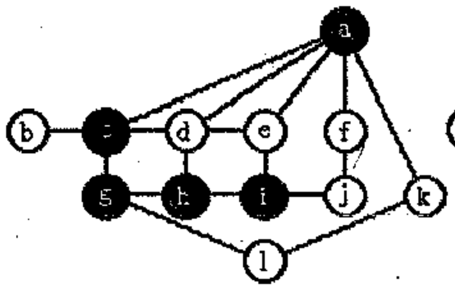
(f) Vertex l in stack

(g) Vertex k in stack

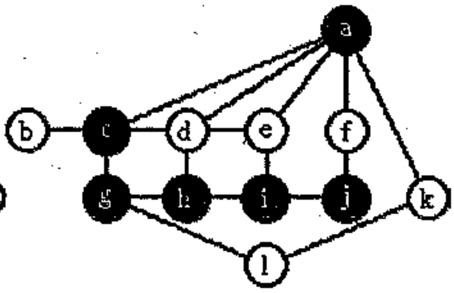
(h) Vertex k explored

(i) Vertex l explored

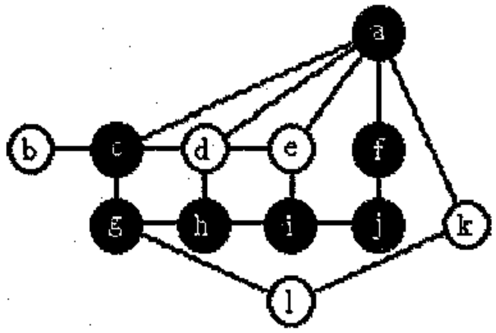
(j) Vertex h in stack



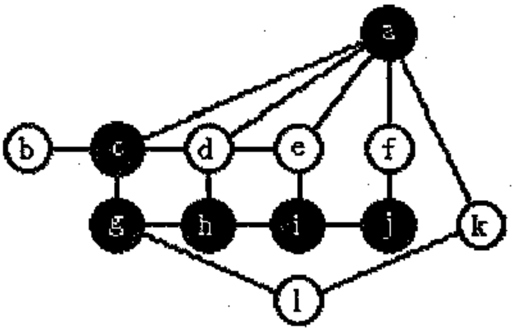
(k) Vertex i in stack



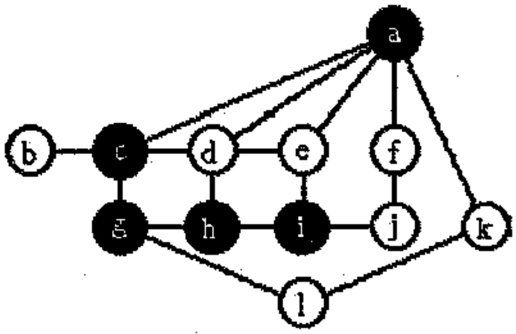
(l) Vertex j in stack



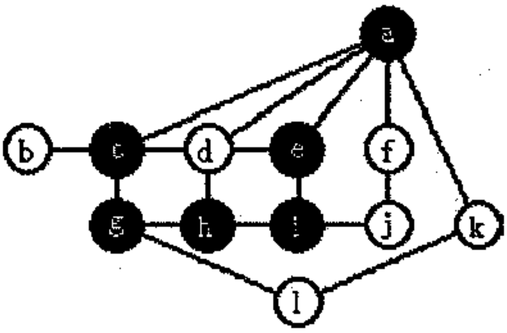
(m) Vertex f in stack



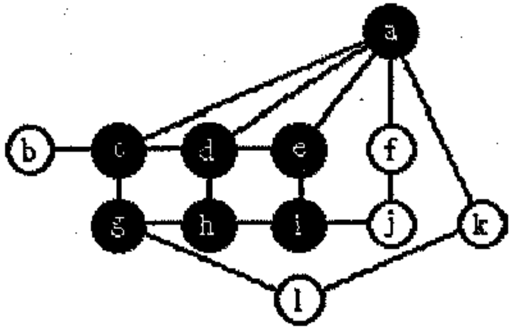
(n) Vertex f explored



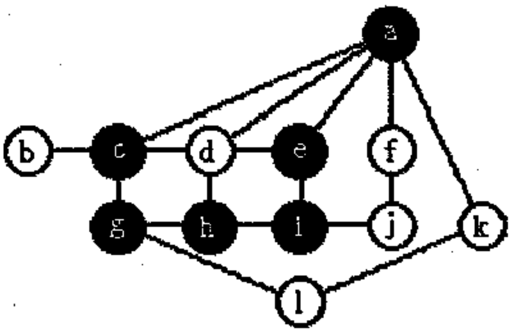
(o) Vertex j explored



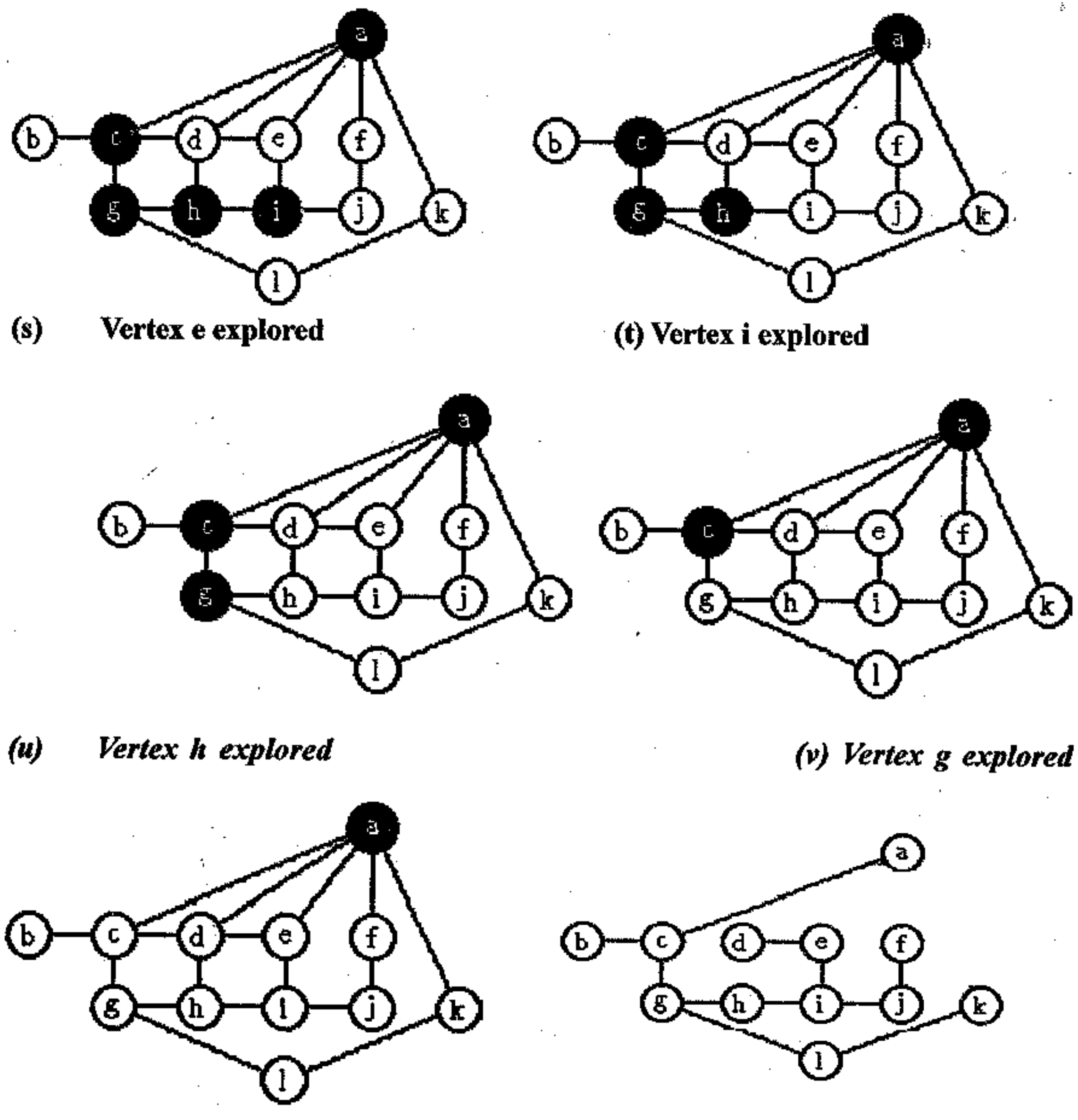
(p) Vertex e in stack



(q) Vertex d in stack



(r) Vertex d explored



(s) Vertex e explored

(t) Vertex i explored

(u) Vertex h explored

(v) Vertex g explored

(w) Vertex c explored

(x) Vertex a explored & DFS Tree

Figure (a)-(x): An example and steps in DFS and DFS Tree

Analysis of DFS

The FOR-loop of DFS takes O(|V|) time, excluding the call to DFS. DFS is called exactly once for each neighbour of current vertex because it immediately marks current vertex as 'visited' on entry to DFS. The FOR-loop in DFS is executed for each node adjacent to current vertex, i.e., for each edge connected to current vertex. Hence the total work for DFS is O(|E|) and the total work for DFS is O(|V|+|E|).

If **Adjacency Matrix** is used, then a stack is needed to maintain unexplored vertices & an array for maintaining the state of a vertex. Scanning each row for checking the connectivity of a Vertex is in order O(|V|). So, Complexity is O(|V|^2).

8.6 SHORTEST PATH ALGORITHMS

Shortest path problem is the problem of finding a path between two vertices (or nodes) such that the sum of the weights of its constituent edges is minimized. An example is finding the quickest way to get from one location to another on a road map; in this case, the vertices represent locations and the edges represent segments of road and are weighted by the time needed to travel that segment. The problem is also sometimes called the **single-pair shortest path problem,** to distinguish it from the following generalizations:

- The **single-source shortest path problem**, in which we have to find shortest paths from a source vertex v to all other vertices in the graph.
- The **single-destination shortest path problem,** in which we have to find shortest paths from all vertices in the graph to a single destination vertex v. This can be reduced to the single-source shortest path problem by reversing the edges in the graph.
- The **all-pairs shortest path problem,** in which we have to find shortest paths between every pair of vertices v, v' in the graph.

These generalizations have significantly more efficient algorithms than the simplistic approach of running a single-pair shortest path algorithm on all relevant pairs of vertices.

To solve this problem, following most important algorithms are described in this unit.

- Dijkstra algorithm solves the single-pair, single-source, and single-destination shortest path problems.
- Bellman-Ford algorithm solves single source problem even if edge weights are negative.
- Floyd-Warshall algorithm solves all pair shortest path problem.

8.6.1 Dijkstra Single-Source Shortest Path Algorithm

Dijkstra algorithm, conceived by Dutch computer scientist Edsger Dijkstra in 1959, is a graph search algorithm that solves the single-source shortest path problem for graphs with non-negative edge path costs, producing a shortest path tree. This algorithm is often used in routing.

For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path first is widely used in network routing protocols, most notably OSPF (Open Shortest Path First).

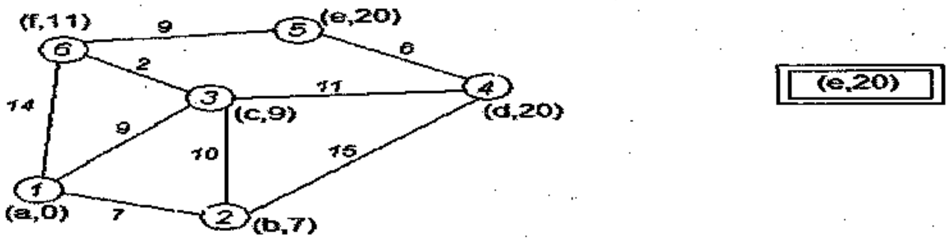
Algorithm

Algorithm starts with **initial node X and** assigns distance from initial node as zero. Assuming **distance of a node Y** be the distance from the **initial node** to it, Dijkstra algorithm will assign some initial distance values and will try to improve them step-by-step. The concept of improvement is described below. Algorithm is explained thereafter.

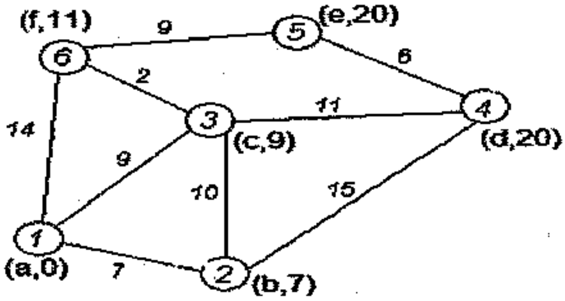
1. Assign to every node a distance value. Set it to zero for our initial node and to infinity for all other nodes.

2. Mark all nodes as unvisited. Set initial node as current.

3. For current node, consider all its unvisited neighbours and calculate their distance (from the initial node). For example, if current node 3 has distance of 9, and an edge connecting it with another node 6 is 2, the distance to node 6 through node 3 will be 9+2=11. If this distance is less than the previously recorded distance (infinity in the beginning, 14 after first iteration, zero for the initial node), overwrite the distance.



(f)Graph and Queue after Fifth Iteration



(g) Graph after Last Iteration

Figure (a)-(g): An example of Dijkstra Algorithm

If we are only interested in a shortest path between vertices source and target, we can terminate the search at line 12 if u = target. Now we can read the shortest path from source to target by iteration:

```
S ! empty sequence
u ! target
while previous[u] is defined:
    insert u at the beginning of S
    u ! previous[u]
```

Algorithm : Change in Dijkstra Algorithm for Source-Target Problem

Now sequence S is the list of vertices constituting one of the shortest paths from target to source, or the empty sequence if no path exists.

Analysis of Dijkstra algorithm

An upper bound of the running time of Dijkstra algorithm on a graph with edges E and vertices V can be expressed as a function of |E| and |V| using the Big-O notation.

For any implementation of set Q the running time is based on two operations namely, decreasing weight of a vertex and extract minimum weight edge. The time required in these operations can be of the O(|E|*decrease_key_in_Q + |V|*extract_minimum_in_Q), where decrease_key_in_Q and extract_minimum_in_Q are times needed to perform that operation in set Q.

The simplest implementation of the Dijkstra algorithm stores vertices of set Q in an ordinary linked list or array, and operation Extract-Min(Q) is simply a linear search through all vertices in Q. In this case, the running time is $O(|V|^2+|E|)=O(|V|^2)$.

For sparse graphs, that is, graphs with fewer than $|V|^2$ edges, Dijkstra algorithm can be implemented more efficiently by storing the graph in the form of adjacency lists and using a binary heap or Fibonacci heap as a priority queue to implement the Extract-Min function efficiently. With a binary heap, the algorithm requires $O((|E|+|V|) \log |V|)$ time (which is dominated by $O(|E| \log |V|)$, assuming the graph is connected), and the Fibonacci heap improves this to $O(| E | + | V | \log | V |)$.

8.6.2 Bellman-Ford Single-Source Shortest Path Algorithm

The **Bellman–Ford algorithm**, a label correcting algorithm, computes single-source shortest paths in a weighted digraph (where some of the edge weights may be negative). Dijkstra algorithm solves the same problem with a lower running time, but requires edge weights should not form negative cycle. Thus, Bellman–Ford is usually used only when there is a possibility of formation of negative edge weight cycles. The algorithm was developed by Richard Bellman and Lester Ford, Jr..

According to Robert Sedgewick, "Negative weights are not merely a mathematical curiosity, but it arises in a natural way when we reduce other problems to shortest-paths problems" and he gives the specific example of a reduction from the NP-complete Hamilton path problem to the shortest paths problem with general weights. If a graph contains a cycle of total negative weight then arbitrarily low weights are achievable and so there is no solution; Bellman-Ford detects this case.

If the graph does contain a cycle of negative weights, Bellman-Ford can only detect this; Bellman-Ford cannot find the shortest path that does not repeat any vertex in such a graph. This problem is at least as hard as the NP-complete longest path problem.

Algorithm

Bellman–Ford is in its basic structure very similar to Dijkstra algorithm, but instead of greedily selecting the minimum-weight node not yet processed to relax, it simply relaxes all the edges, and does this |V| " 1 times, where |V| is the number of vertices in the graph. The repetitions allow minimum distances to accurately propagate throughout the graph, since, in the absence of negative cycles, the shortest path can only visit each node at most once. Unlike the greedy approach, which depends on certain structural assumptions derived from positive weights, this straightforward approach extends to the general case.

For non-negative weights, algorithm behaves similar to Dijkstra algorithm except some steps may not do anything. A priority queue is maintained in Dijkstra algorithm to select next node which will really update the weights, but for loops in this algorithm at line 8 & 9 will take all vertices and update weights in right order and may not be doing anything in case of only positive weights.

Bellman–Ford runs in $O(|V|\cdot|E|)$ time, where |V| and |E| are the number of vertices and edges respectively.

1.Algorithm BellmanFord(list_of_vertices, list_of_edges, source_ vertex)

—This implementation takes in a graph, represented as lists of vertices

—and edges, and modifies the vertices so that their distance and

```
—predecessor attributes store the shortest paths.
—Step 1: Initialize graph
2.    for each vertex v in vertices:
3.      if v is source
4.                then v.distance ! 0
5.      else
6.                v.distance ! infinity
7.      v.predecessor ! null
—Step 2: relax edges repeatedly
8.    for i from 1 to size(vertices)-1:
9.      for each edge uv in edges: — uv is the edge from u to v
10.       u ! uv.source
11.           v ! uv.destination
12.       if u.distance + uv.weight < v.distance
13.           then v.distance ! u.distance + uv.weight
14.           v.predecessor ! u
—Step 3: check for negative-weight cycles
15.   for each edge uv in edges:
16.     u ! uv.source
17.     v ! uv.destination
18.     if u.distance + uv.weight < v.distance:
19.       error "Graph contains a negative-weight cycle"
20.
```

Algorithm : Bellman-Ford Algorithm

Figure shows initial graph for Bellman-Ford algorithm (same as used in Dijkstra algorithm example). Figure 9.5(b) after first iteration i.e. for i = 1 at line 8 and sequence of vertices taken is (a, b), (a, c), (a, f), (b, d), (b, c), (c, f), (c, d), (c, e) and (e, d). These values remain the same for all iterations till last.

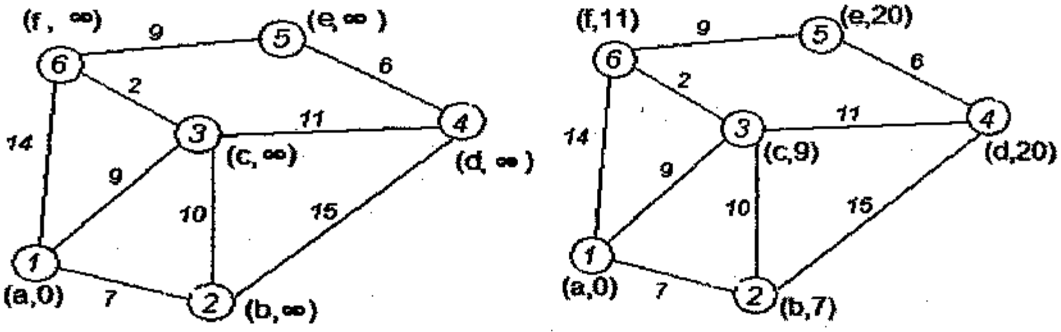


Figure (a): Initial Graph (b) after first till last iteration

Proof of correctness

The correctness of the algorithm can be shown by induction. The precise statement shown by induction is:

Lemma. After i repetitions of for cycle:

- If Distance(u) is not infinity, it is equal to the length of some path from s to u;
- If there is a path from s to u with at most i edges, then Distance(u) is at most the length of the shortest path from s to u with at most i edges.

Proof. For the base case of induction, consider i=0 and the moment before for cycle is executed for the first time. Then, for the source vertex, source.distance ! 0, which is correct. For other vertices u, u.distance ! **infinity**, which is also correct because there is no path from source to u with 0 edges.

For the inductive case, we first prove the first part. Consider a moment when a vertex distance is updated by v.distance ! u.distance + uv.weight. By inductive assumption, u.distance is the length of some path from source to u. Then u.distance + uv.weight is the length of the path from source to v that follows the path from source to u and then to v.

For the second part, consider the shortest path from source to u with at most i edges. Let v be the last vertex before u on this path. Then, the part of the path from source to v is the shortest path from source to v with at most i-1 edges. By inductive assumption, v.distance after i-1 cycles is at most the length of this path. Therefore, uv.weight + v.distance is at most the length of the path from s to u. In the ith cycle, u.distance gets compared with uv.weight + v.distance, and is set equal to it if uv.weight + v.distance was smaller. Therefore, after i cycles, u.distance is at most the length of the shortest path from source to u that uses at most i edges.

If there are no negative-weight cycles, then every shortest path visits each vertex at most once, so at step 3 no further improvements can be made. Conversely, suppose no improvement can be made. Then for any cycle with vertices v[0],..,v[k-1],

v[i].distance <= v[i-1 (mod k)].distance + v[i-1 (mod k)]v[i].weight

Summing around the cycle, the v[i].distance terms and the v[i-1 (mod k)] distance terms cancel, leaving

0 <= sum from 1 to k of v[i-1 (mod k)]v[i].weight

i.e., every cycle has nonnegative weight.

8.6.3 Floyd-Warshall All Pair Shortest Path Algorithm

The **Floyd-Warshall Algorithm** is an efficient algorithm to find all-pairs shortest paths on a graph. That is, it is guaranteed to find the shortest path between every pair of vertices in a graph. The graph may have negative weight edges, but no negative weight cycles (for then the shortest path is undefined).

This algorithm can also be used to detect the presence of negative cycles—the graph has one if at the end of the algorithm, the distance from a vertex v to itself is negative.

The Floyd-Warshall algorithm compares all possible paths through the graph between each pair of vertices. It is able to do this with only $|V|^3$ comparisons. This is remarkable considering that there may be up to $|V|^2$ edges in the graph, and every combination of edges is tested. It does so by incrementally improving an estimate on the shortest path between two vertices, until the estimate is known to be optimal.

Let dist(k,i,j) be the the length of the shortest path from i and j that uses only the vertices $v_1, v_2, \ldots, v_k$ as intermediate vertices. The following recurrence:

- k = 0 is our base case - dist(0,i,j) is the length of the edge from vertex i to vertex j if it exists, and ∞ otherwise.
- dist(k,i,j) = min(dist(k - 1,i,k) + dist(k - 1,k,j),dist(k - 1,i,j)): For any vertex i and vertex j, the length of the shortest path from i to j with all intermediate vertices $\leq k$ simply does not involve the vertex k at all (in which case it is the same as dist(k - 1,i,j)), or that the shorter path goes through vertex k, so the shortest path between vertex i and vertex j is the combination of the path from vertex i to k, and from vertex k to j.

After |V| iterations, there is no need anymore to go through any more intermediate vertices, so the distance dist(N,i,j) represents the shortest distance between i and j, where N = |V|.

Algorithm

```
Floyd-Warshall (Graph G(V, E))
    — Representing N for |V|
    — Initialize distance values
    for i = 1 to N
     for j = 1 to N
      if there is an edge from i to j
              dist[0][i][j] ! the length of the edge from i to j
      else
              dist[0][i][j] ! INFINITY
— Relax distance values
    for k ! 1 to N
     for i ! 1 to N
      for j ! 1 to N
       dist[k][i][j] ! min(dist[k-1][i][j], dist[k-1][i][k] + dist[k-1][k][j])
```

Algorithm : Floyd-Warshall Algorithm

This will give the shortest distances between any two nodes, from which shortest paths may be constructed.

This algorithm takes È($|V|^3$) time and È($|V|^3$) space, and has the distinct advantage of hiding a small constant in its behavior, since very little work is done in the innermost loop. Furthermore, the space-bound can be reduced further to È($|V|^2$) by noticing that dist(k,i,j) is independent from dist(k - 1,i,j). As mentioned above, dist(N,i,j) represent shortest distance between any pair of vertices, which is nothing but a matrix dist[k][i][j] after N iterations.

Figure shows example graph. Edges of the graph are assumed in both the directions.

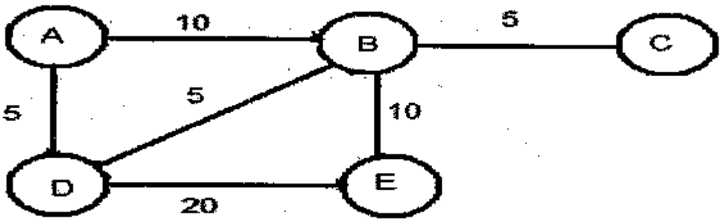


Figure Example graph for Floyd-Warshall Algorithm

Figure shows matrix representation of the graph used in algorithm. As algorithm progresses, intermediate matrices are generated. Final matrix is shown Figure, which represents shortest distance between any pair of vertices.

	A	B	C	D	E
A	0	10	?	5	?
B	10	0	5	5	10
C	?	5	0	?	?
D	5	5	?	0	20
E	?	10	?	20	0

	A	B	C	D	E
A	0	10	15	5	20
B	10	0	5	5	10
C	15	5	0	10	15
D	5	5	10	0	15
E	20	10	15	15	0

(a) Initial Matrix for example (b) Final Matrix in Floyd-Warshall algorithm

Figure (a)-(b): Initial & Final Matrices for Floyd-Warshall Algorithm

For constructing path between any pair of vertices, predecessor matrices are to be generated and preserved till end. These matrices can be used to generate shortest path between any pair of vertices.

8.7 SPANNING TREE ALGORITHMS

Given a connected, undirected graph, a spanning tree of that graph is a subgraph which is a tree and connects all the vertices of the graph. A single graph can have many different spanning trees. We can also assign a weight to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A **minimum spanning tree** (MST) or **minimum weight spanning tree** is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a **minimum spanning forest**, which is a union of minimum spanning trees for its connected components.

One example would be a cable TV company laying cable to a new neighborhood. If it is constrained to bury the cable only along certain paths, then there would be a graph representing which points are connected by those paths. Some of those paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights. A spanning tree for that graph would be a subset of those paths that has no cycles but still connects to every house. There might be several spanning trees possible. A minimum spanning tree would be one with the lowest total cost.It is important to note that if weights of all edges are equal then all spanning trees will have the same weight. On the other hand, if all edges have different weights i.e. unequal weights then only one spanning tree will be MST.

There are two important algorithms being used for obtaining MST namely, Prim algorithm and kruskal Algorithm. Both are based on greedy approach for finding MST. Algorithms are described in this section.

8.7.1 Prim Algorithm for MST

Prim algorithm, for finding a minimum spanning tree for a connected weighted graph, was developed in 1930 by Czech mathematician Vojtích Jarník and later independently by computer scientist Robert C. Prim in 1957 and rediscovered by Edsger Dijkstra in 1959. Therefore it is sometimes called the **DJP algorithm**, the **Jarník algorithm**, or the **Prim-Jarník algorithm**.

Algorithm

The algorithm starts with a single vertex and continuously increases the size of a tree starting until it spans all the vertices. Steps are detailed below:

1. Input: A connected weighted graph with vertices V and edges E.
2. Initialize: $V_{new} = \{x\}$, where x is an arbitrary node (starting point) in V, $E_{new} = \{\}$
3. Repeat until $V_{new} = V$:

o Choose edge (u,v) with minimal weight such that u is in V_{new} and v is not (if there are multiple edges with the same weight, choose arbitrarily but consistently) – **needs a queue for searching**

o Add v to V_{new}, add (u, v) to E_{new}

4. Output: V_{new} and E_{new} describe a minimal spanning tree

Algorithm : Prim Algorithm for MST

Time complexity

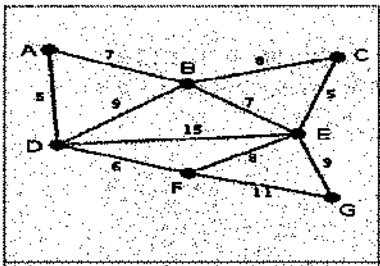
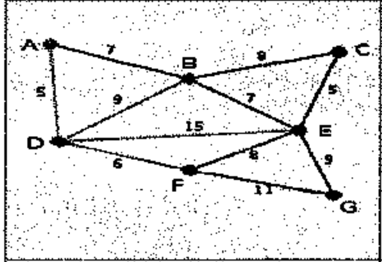
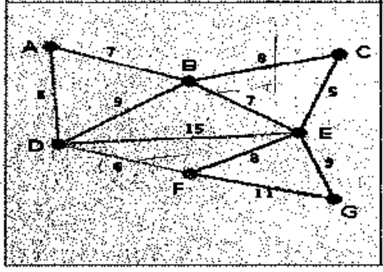
Time complexity of the algorithm depends on the data structure used to represent the graph and for searching. The following table shows the time complexity as per data structure representing the graph.

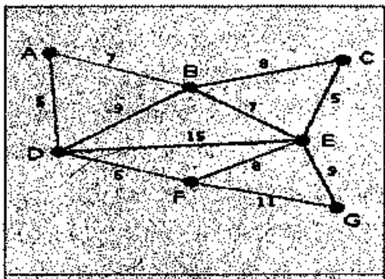
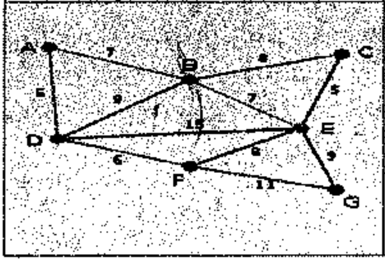
Graph data structure	**Queue Data structure**	**Time complexity (total)**
Adjacency matrix	Array	$O(V^2)$
Adjacency list	Binary heap	$O((V + E) \log(V)) = O(E \log(V))$
Adjacency list	Fibonacci heap	$O(E + V \log(V))$

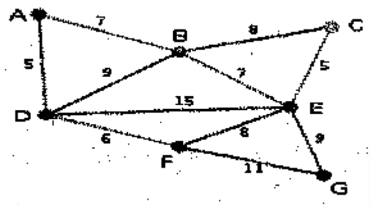
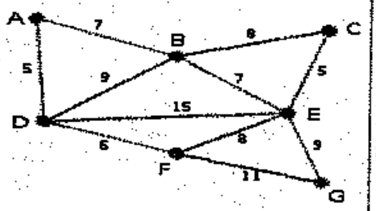
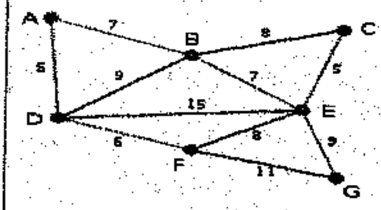
Table : Prim Algorithm Time Complexity

A simple implementation using an adjacency matrix graph representation and searching an array of weights to find the minimum weight edge to add requires $O(V^2)$ running time. Using a simple binary heap data structure and an adjacency list representation, Prim algorithm can be shown to run in time O(E log V) where E is the number of edges and V is the number of vertices. Using a more sophisticated Fibonacci heap, this can be brought down to O(E + V log V), which is significantly faster when the graph is dense. Binary heap is the same as used in heap sorting and Fibonacci heap is complex data structure beyond our scope.

Example

 A B C D E F G 7 8 5 9 7 5 15 8 9 6 11	This is our original weighted graph. The numbers near the edges indicate their weight.
 A B C D E F G 7 8 5 9 7 5 15 8 9 6 11	Vertex **D** has been arbitrarily chosen as a starting point. Vertices **A**, **B**, **E** and **F** are connected to **D** through a single edge. **A** is the vertex nearest to **D** and will be chosen as the second vertex along with the edge **AD**.
 A B C D E F G 7 8 5 9 7 5 15 8 9 6 11	The next vertex chosen is the vertex nearest to *either* **D** or **A**. **B** is 9 away from **D** and 7 away from **A**, **E** is 15, and **F** is 6. **F** is the smallest distance away, so we highlight the vertex **F** and the arc **DF**.

 A B C D E F G 7 8 5 9 7 5 15 8 9 6 11	The algorithm carries on as above. Vertex **B**, which is 7 away from **A**, is highlighted.
 A B C D E F G 7 8 5 9 7 5 15 8 9 6 11	In this case, we can choose between **C**, **E**, and **G**. **C** is 8 away from **B**, **E** is 7 away from **B**, and **G** is 11 away from **F**. **E** is nearest, so we highlight the vertex **E** and the arc **BE**.

 A B C D E F G 7 8 5 9 7 15 5 6 8 9 11	Here, the only vertices available are **C** and **G**. **C** is 5 away from **E**, and **G** is 9 away from **E**. **C** is chosen, so it is highlighted along with the arc **EC**.
 A B C D E F G 7 8 5 9 7 15 5 6 8 9 11	Vertex **G** is the only remaining vertex. It is 11 away from **F**, and 9 away from **E**. **E** is nearer, so we highlight it and the arc **EG**.
 A B C D E F G 7 8 5 9 7 15 5 6 8 9 11	Now all the vertices have been selected and the **minimum spanning tree** is shown in green. In this case, it has weight 39.

Proof of correctness

Let P be a connected, weighted graph. At every iteration of Prim algorithm, an edge must be found that connects a vertex in a subgraph to a vertex outside the subgraph. Since P is connected, there will always be a path to every vertex. The output Y of Prim algorithm is a tree, because the edge and vertex added to Y are connected. Let Y_1 be a minimum spanning tree of P. If Y_1=Y then Y is a minimum spanning tree. Otherwise, let e be the first edge added during the construction of Y that is not in Y_1, and V be the set of vertices connected by the edges added before e. Then one endpoint of e is in V and the other is not. Since Y_1 is a spanning tree of P, there is a path in Y_1 joining the two endpoints. As one travels along the path, one must encounter an edge f joining a vertex in V to one that is not in V. Now, at the iteration when e was added to Y, f could also have been added and it would be added instead of e if its weight was less than e. Since f was not added, we conclude that

$$w(f) \geq w(e).$$

Let Y_2 be the graph obtained by removing f and adding e from Y_1. It is easy to show that Y_2 is connected, has the same number of edges as Y_1, and the total weights of its edges is not larger than that of Y_1, therefore it is also a minimum spanning tree of P and it contains e and all the edges added before it during the construction of V. Repeat the steps above and we will eventually obtain a minimum spanning tree of P that is identical to Y. This shows Y is a minimum spanning tree.

8.7.2 Kruskal Algorithm for MST

Kruskal algorithm finds a minimum spanning tree for a connected weighted graph and if the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component). Kruskal algorithm is an example of a greedy algorithm. The algorithm was first published by Joseph Kruskal in 1956.

Algorithm

The algorithm works as follows:

```
1.  create a forest F (a set of trees), where each vertex in the graph is a separate tree
2.  create a set S containing all the edges in the graph
3.  while S is nonempty
      o  remove an edge with minimum weight from S
      o  if that edge connects two different trees and results into a single tree, then add it to the forest
      o  otherwise discard that edge.
```

Algorithm : Kruskal Algorithm for MST

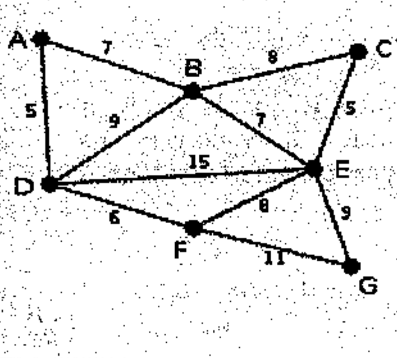
At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.

Time Complexity

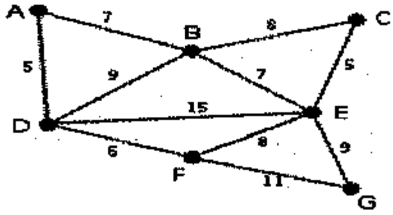
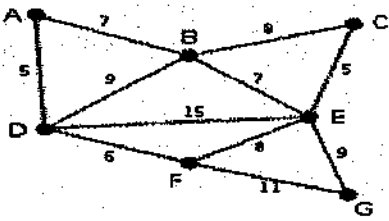
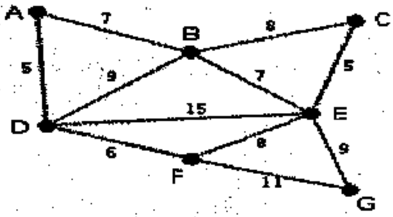
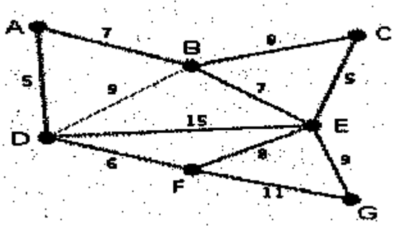
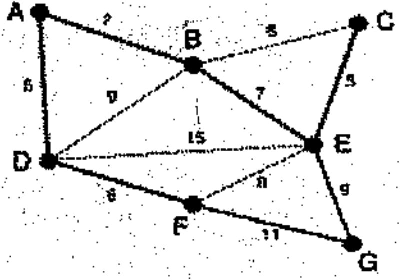
Where E is the number of edges in the graph and V is the number of vertices, Kruskal algorithm can be shown to run in O(E log E) time, or equivalently, O(E log V) time, all with simple data structures. These running times are equivalent because:

- E is at most V^2 and $\log V^2 = 2\log V$ is O(log V).
- If we ignore isolated vertices, which will each be their own component of the minimum spanning forest, V d" E+1, so log V is O(log E).

We can achieve this bound as follows: first sort the edges by weight using a comparison sort in O(E log E) time; this allows the step "remove an edge with minimum weight from S" to operate in constant time. Next, we use a disjoint-set data structure to keep track of which vertices are in which components. We need to perform O(E) operations, two 'find' operations and possibly one union for each edge. Even a simple disjoint-set data structure such as disjoint-set forests with union by rank can perform O(E) operations in O(E log V) time. Thus the total time is O(E log E) = O(E log V).



This is our original graph. The numbers near the arcs indicate their weight. None of the arcs are highlighted.

 A 7 B 8 C 5 9 7 5 15 E D 6 8 9 F 11 G	**AD** and **CE** are the shortest arcs, with length 5, and **AD** has been arbitrarily chosen, so it is highlighted.
 A 7 B 8 C 5 9 7 5 15 E D 6 8 9 F 11 G	**CE** is now the shortest arc that does not form a cycle, with length 5, so it is highlighted as the second arc.
 A 7 B 8 C 5 9 7 5 15 E D 6 8 9 F 11 G	The next arc, **DF** with length 6, is highlighted using much the same method.
 A 7 B 8 C 5 9 7 5 15 E D 6 8 9 F 11 G	The next-shortest arcs are **AB** and **BE**, both with length 7. **AB** is chosen arbitrarily, and is highlighted. The arc **BD** has been highlighted in red, because there already exists a path (in green) between **B** and **D**, so it would form a cycle (**ABD**) if it were chosen.
 A 7 B 8 C 5 9 7 5 15 E D 6 8 9 F 11 G	The process continues to highlight the next-smallest arc, **BE** with length 7. Many more arcs are highlighted in red at this stage: **BC** because it would form the loop **BCE**, **DE** because it would form the loop **DEBA**, and **FE** because it would form **FEBAD**.

Proof of correctness

The proof consists of two parts. First, it is proved that the algorithm produces a spanning tree. Second, it is proved that the constructed spanning tree is of minimal weight.

Spanning tree

Let P be a connected, weighted graph and let Y be the subgraph of P produced by the algorithm. Y cannot have a cycle, since the last edge added to that subgraph would have been chosen only if it doesn't form cycle. Y cannot be disconnected, since the first encountered edge that joins two components of Y would have been added by the algorithm. Thus, Y is a spanning tree of P.

Minimality

Assume that Y is not a minimal spanning tree and among all minimum weight spanning trees pick Y_1 which has the smallest number of edges which are not in Y. Consider the edge e which was first to be added by the algorithm to Y of those which are not in Y_1.

$Y_1 \cup e$ has a cycle. Being a tree, Y cannot contain all edges of this cycle. Therefore this cycle contains an edge f which is not in Y. The graph $Y_2 = Y_1 \cup e \setminus f$ is also a spanning tree and therefore its weight cannot be less than the weight of Y_1 and hence the weight of e cannot be less than the weight of f.

Assume the contrary and remember that the edges are considered for addition to Y in the order of non-decreasing weight. Therefore f would have been considered in the main loop before e, i.e., it would be tested for the addition to a subforest $Y_3 \subset Y \cap Y_1$(recall that e is the first edge of Y which is not in Y_1). But f does not create a cycle in Y_1, therefore it cannot create a cycle in Y_3, and it would have been added to the growing tree.

The above implies that the weights of e and f are equal, and hence Y_2 is also a minimal spanning tree. But Y_2 has one more edge in common with Y than Y_1, which contradicts to the choice of Y_1, Hence proved.

self learning exercises

1. How many edges are there in a complete graph on 5 vertices?

(a) 5 (b) 10 (c) 25 (d) 50

2. An Eulerian Path must have

(a) Only one vertex of even degree (b) All vertices of even degree

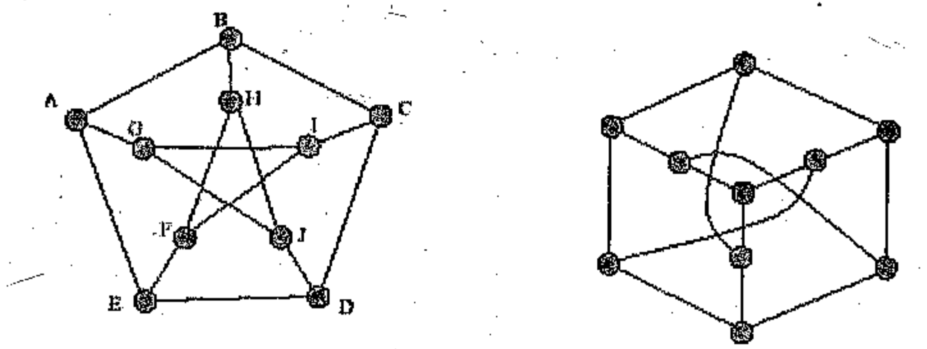
(c) Only two vertices of odd degree (d) None of the above

3. Adjacency List representation of graph requires

(a) O(|V|+|E|) storage space (b) O(|V|^2) storage space

(c) O(|E|^2) storage space (d) none of the above

4. For the following graphs, which statement is not true?



(a) These are Peterson graphs. (b) These are isomorphic graphs

(c) These are connected-graphs (d) These are multi-graphs

5. $K_{1,8}$ graph is a

(a) Complete Bipartite and Star graph (b) Complete graph

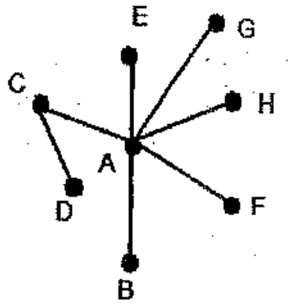
(c) Regular graph (d) None of the above

6. Vertices A for (5, 8), B for (-4, 5) and C for (3, 4) are used for construction of an interval graph. Which statement is true for this graph?

(a) A is connected to B and C (b) B is connected to A and C

(c) B is not connected to A and B (d) C is connected to A and B

7. In the following graph, removal of vertex A leaves



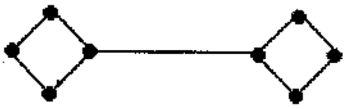
(a) 6 (b) 1 (c) 2 (d) 5

8. A regular graph can be disconnected if

(a) It is of degree 2 (b) it is of degree 3

(c) it is of degree 1 (d) it is of degree 0

9. How many edges need to be removed to leave the following graph disconnected?



(a) 2 (b) 1 (c) 4 (d) 2

10.For a directed graph, " in adjacency matrix represents

(a) Vertices are not connected (b) Vertices are adjacent

(c) Vertices form a cycle (d) None of the above

8.8 Summary

This unit describes basic fundamentals of graph theory which includes various definitions, graph classification and properties, data structure to store graph and solving puzzles.

- Königsberg bridge puzzle lead the invention of graph theory by Swiss mathematician named Leonhard Euler (pronounced "oiler") in 1735. Since then graph theory has been solving a large number of problems ranging from travelling salesman problem to VLSI design.
- There are many classifications available, but simple graph with directed or undirected, weighted or unweighted graphs are mostly used and hence, graph theory fundamentals and algorithms have focused on above graphs only.
- There are many data structures available to represent graphs. Incidence, adjacency matrix, adjacency list and adjacency multi-list data structures have been explained in section .
- Two search algorithms are described in section , namely Breadth First Search (BFS) and Depth First Search (DFS). Both the algorithms visit every node of the graph traversing through edges but in different way. BFS traverses all adjacent vertices of current vertices starting from root and maintains a queue for this purpose. DFS traverses only one adjacent vertex till no more vertices can be traversed and maintains stack for this purpose.
- Shortest Path algorithms are found in three variants namely, single-source shortest path problem, single-destination shortest path problem and all-pairs shortest path problem. Single-source and single-destination problems can be converted to each other just by reversing the edge direction. Dijkstra Algorithm solves single-source problem if weights of edges do not form negative cycle. Bellman-Ford algorithm detects negative cycle if it exists and solves if does not exist. Floyd-Warshall algorithm solves all-pairs shortest path problem in which shortest route can be found between every pair of vertices.
- Spanning tree of a graph is a tree containing all vertices and a subset of edges of the graph. A spanning tree having minimum sum of weights of edges in a weighted graph is called Minimum Spanning Tree (MST). Prim and Kruskal algorithms with greedy approach obtains MST. The trees obtained by these algorithms may not be the same, but sum of weights of edges of the tree will be the same.

8.9 Glossary

bipartite : A graph is bipartite if its vertices can be partitioned into two disjoint subsets U and V such that each edge connects a vertex from U to one from V. A bipartite graph is a complete bipartite graph if every vertex in U is connected to every vertex in V. If U has n elements and V has m, then we denote the resulting complete bipartite graph by Kn,m. The illustration shows K3,3. See also complete graph and cut vertice

chromatic number : The chromatic number of a graph is the least number of colors it takes to color its vertices so that adjacent vertices have different colors. For example, this graph has chromatic number three.

When applied to a map this is the least number of colors so necessary that countries that share nontrivial borders (borders consisting of more than single points) have different colors.

complete graph:A complete graph with n vertices (denoted Kn) is a graph with n vertices in which each vertex is connected to each of the others (with one edge between each pair of vertices). Here are the first five complete graphs

directed graph:A digraph (or a directed graph) is a graph in which the edges are directed. (Formally: a digraph is a (usually finite) set of vertices V and set of ordered pairs (a,b) (where a, b are in V) called edges. The vertex a is the initial vertex of the edge and b the terminal vertex.

graph: Informally, a graph is a finite set of dots called vertices (or nodes) connected by links called edges (or arcs). More formally: a simple graph is a (usually finite) set of vertices V and set of unordered pairs of distinct elements of V called edges.

8.10 Further Readings

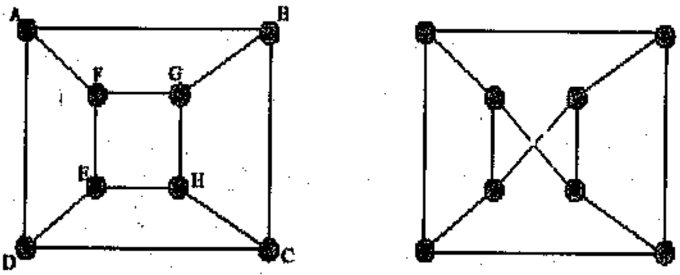
1. Graph theory with applications to Engineering and computer science by Narsingh Deo.
2. Discrete Mathematics by Rosen.
3. Graph theory by John Clark and Derek Allan Holton
4. Reference Book Allan Tucker for Combinatorics

8.11 Answers to self learning exercises

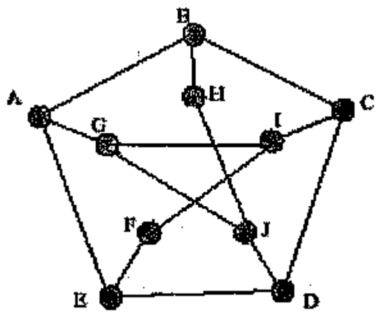
1(b) 2(c) 3(c) 4(d) 5(a) 6(d) 7(a) 8(d) 9(b) 10(a)

8.12 Unit-End Questions

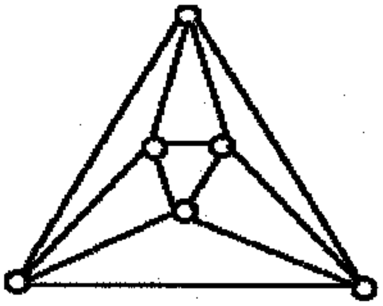
1. Find whether the following pair of graphs is isomorphic?



2. Find whether the following graph is planar or not? If not then how many edges to be removed to make it planar?_



3.For the following graph, assign names to vertices and write its adjacency list representation. Write steps to count number of edges from adjacency list and write its incidence matrix also.



4. Draw four spanning trees of above graph.

5. Find Hamiltonian Path of the graph shown in question

--- ***----

UNIT IX

GRAPH THEORY ALGORITHMS

STRUCTURE OF THE UNIT

9.0 Objectives

9.1 Introduction

9.2 Planarity Detection Technique

9.3 Algorithms for Connectivity

9.4 Planarity Testing

9.5 Summary

9.6 Glossary

9.7 Further Readings

9.8 Answers to self learning exercises

9.9 Unit –end Questions

9.0 Objectives

After reading this unit you should appreciate the following :

- Planarity Detection Technique
- Spanning Tree algorithm
- Connectedness algorithm

9.1 Introduction

Previous unit has described various types of graphs useful in different applications. To work with them, a number of interesting algorithms are available. These algorithms are standard and proven useful in various applications. However, in some of the applications, these algorithms need little customization as per the need of the application.

Planarity detection is very important issue for many graphs in applications like PCB manufacturing. Planarity determines whether a given graph can be drawn in a plane or not? Kuratowski has given two important planar graphs. Section 9.4 explains above graphs and useful tips to detect planarity.

Spanning tree is a tree consisting of all vertices and partially edges of the graph. These algorithms are very useful in applications like cable laying for network etc.

In many applications require to check the reachability of a specific node within the graph through edges like traceroute or ping utility in computer network. To check this reachability, algorithms for connectedness of a node or sub-graph, important algorithms are described 9.4

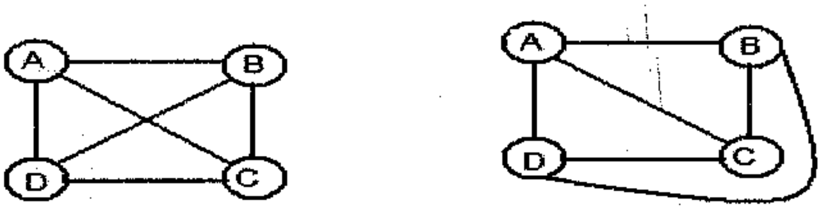
9.2 Planarity Detection Technique

A **planar graph** is a graph which can be embedded in the plane, i.e., it can be drawn on the plane in such a way that its edges intersect only at their endpoints.

A planar graph already drawn in the plane without edge intersections is called a **plane**

graph or **planar embedding of the graph**. A plane graph can be defined as a planar graph with a mapping from every node to a point in 2D space, and from every edge to a plane curve, such that the extreme points of each curve are the points mapped from its end nodes, and all curves are disjoint except on their extreme points.

Figure (a) shows a graph in which edges intersect. The same graph is drawn in Figure (b) without intersecting edges (isomorphic graph). Hence, it can be embedded in a plane and is a planer graph.



(a) Edge intersecting graph **(b) Embedded in Plane graph**

Figure9.1 (a)-(b): A planer graph and its embedding in plane.

Whether a graph is planer or not is not easy to answer and no straight algorithm is available to detect planarity. However, interesting studies and a number of techniques can be helpful in determining planarity. This section describes planarity detection techniques using Kuratowski's two graphs and Homeomorphic graphs.

The Polish mathematician Kazimierz Kuratowski provided a characterization of planar graphs in terms of forbidden graphs, now known as **Kuratowski's theorems**. For this purpose, Kuratowski gave two graphs known as Kuratowski's two graphs.

Kuratowski's first graph represents a non-planer graph with **minimum number of vertices.** That is a complete graph with five vertices is non-planer. The graph is popularly known as K_5 graph. It was shown that non-planer graph is not possible with less than 5-vertices.

Second graph by Kuratowski is of **minimum number of edges**. That is of 9-edges. It was also shown that a non-planer graph requires 9 or more edges. Both the graphs are regular i.e. degree of each vertex is same and non-planer.

Figure 9.2(a) shows an incomplete graph having 5-vertices and 9 edges, which is incomplete and planer graph. By adding one edge graph becomes regular and non-planer as shown in Figure 9.2.(b).

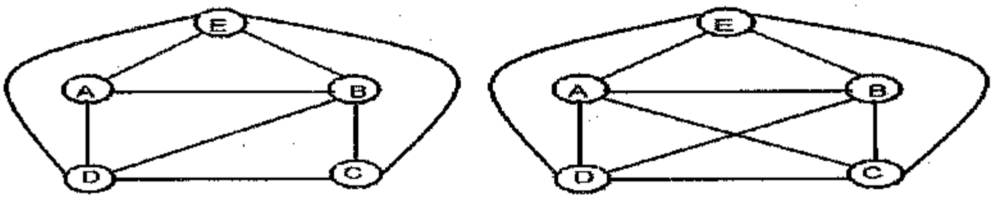
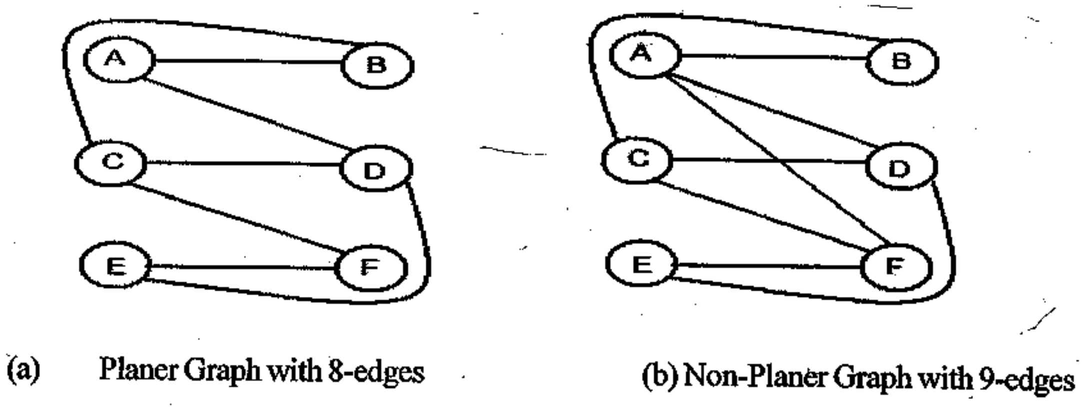


Figure (a) 5-vertex planer graph **(b) 5-vertex non-planer Complete Graph**

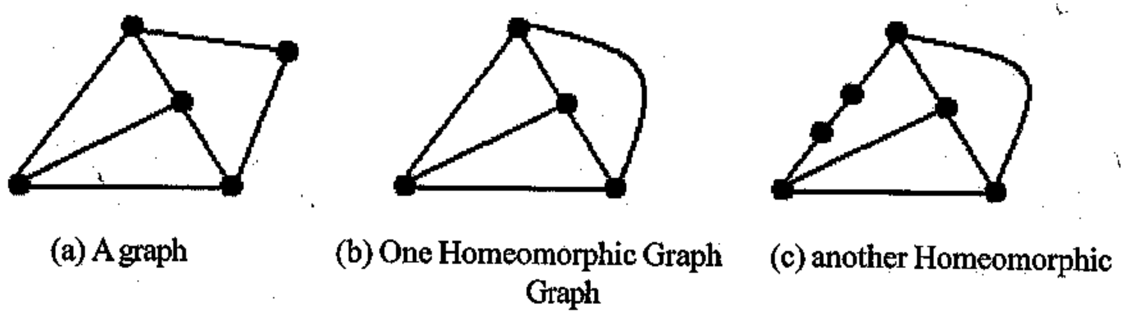
Figure 9.2(a)-(b): Kuratowski's K_5 (5-vertex Complete) graph.



(a) Planer Graph with 8-edges (b) Non-Planer Graph with 9-edges

Figure 9.3(a)-(b): Kuratowski's 9-edges $K_{3,3}$ graph.

Similarly, a graph with 8 edges can be drawn as planer, but not regular as shown in Figure 9.3(a). However, by adding one edge, it becomes regular but non-planer (Figure 9.3(b). This graph is popularly known as $K_{3,3}$. This graph is also a bipartite graph.



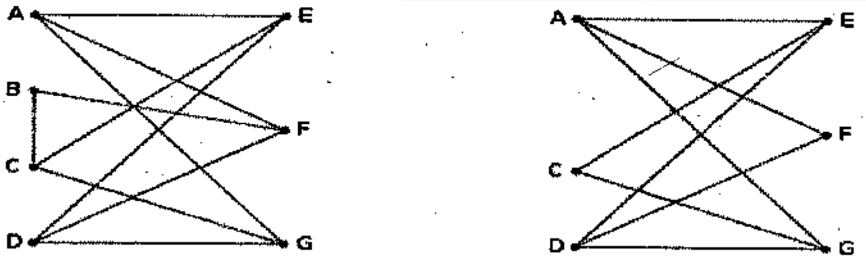
(a) A graph (b) One Homeomorphic Graph (c) another Homeomorphic Graph

Figure 9.4(a)-(c): Homeomorphic graphs.

Two graphs are said to be Homeomorphic if one graph can be obtained from the other by creation of edges in series (i.e. insertion of vertices of degree two on edge) or by merger of edges in series. The three graphs shown in Figure 9.11 are homeomorphic to each other.

A Kuratowski's and Wagner's theorems

finite graph is planar if and only if it does not contain a subgraph that is a homomorphic graph of K5 (the complete graph on five vertices) or K3,3 (complete bipartite graph on six vertices, three of which connect to each of the other three).



(a) A graph **(b) SubGraph of (a) that is $K_{3,3}$**

Figure 9.5(a)-(b): Non-Planer graph.

An example of a graph shown in Figure 9.5(a), which doesn't have K_5 or $K_{3,3}$ as its subgraph. However, it has a subgraph (as shown in Figure 9.5(b)), that is homeomorphic to $K_{3,3}$ and is therefore not planar.

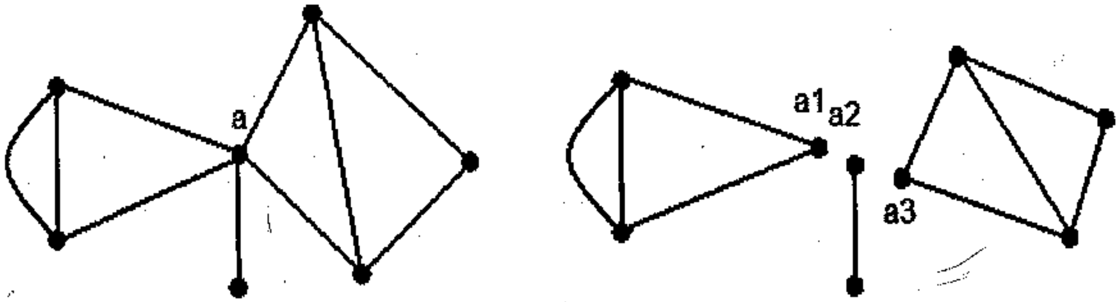
In practice, it is difficult to use Kuratowski's criterion to quickly decide whether a given graph is planar. Finding subgraphs and their homeomorphic graphs is a difficult problem. Another condition described by Euler is also being used for quick examination.

As per Euler condition for a simple, connected, planar graph with v vertices and e edges, the following simple planarity criteria hold:

If v = 3 then e = 3v - 6;

It may be noted that this condition provide necessary conditions for planarity but not sufficient condition. Hence, it can only be used to prove a graph is not planar, not that it is planar.

Another concept of non-separable graph is also important. A connected graph is said to be non-separable if removal of one vertex with all connected edges does not leave a disconnected graph. Figure 9.13(a) shows a separable graph as removal of its vertex a leave the graph disconnected into three subgraphs. However, by replicating vertex three times as a1, a2 and a3, we get three non-separable graphs. Planarity is not affected by separating graph having such vertex. Planarity can be checked in each non-separable component.



(a) A separable graph **(b) Non-Separable Components**

Figure 9.6(a)-(b): A separable graph and its non-separable components.

With this knowledge, techniques to detect planarity can be described, which are necessary conditions, but not sufficient. If these conditions are not satisfied, we can quickly detect that graph is non-planer, but cannot declare the graph is planer. Hence, this serves the purpose of preliminary simplification only which will be useful most of the time. The steps of simplification are as follows:

1. Find connected components using algorithm described in section 9.3. The algorithm is very simple and run in linear time. If there are more than one component, then test these steps for each components separately.
2. Remove all self-loops and replace each set of parallel edges by a single edge.
3. Eliminate every vertex of degree two by merging two edges incident on the vertex. Apply step 2 & 3 repeatedly till no further reduction is possible.
4. Partition the graph into non-separable subgraphs if possible.
5. Apply reduction of step 3 and 2 in each non-separable subgraph.
6. Each reduced non-separable graph obtained after step 5, having e edges and n vertices, test

the following conditions:

n = 5

e = 9

e = 3n – 6

7. If any of these inequalities are not satisfied, we can conclude as follows:

If n < 5 or e < 9 then subgraph is planer and we need to check other subgraphs too. However, if e > 3n – 6 then subgraph is non-planer and hence entire graph is non-planer.

These steps do not provide complete detection, but most of the times detect non-planarity.

9.3 Algorithms for Connectivity

Connectedness in a graph is an important issue as it describes the reachability or traceability of a vertex within a graph. In an undirected graph G, two vertices u and v are called **connected** if G contains a path from u to v. Otherwise, they are called **disconnected.** It may also be noted that vertices connected by an edge, i.e., by path of length 1, are called **adjacent.** Here are some connectivity related definitions:

A graph is called connected graph if every pair of distinct vertices in the graph can be connected through some path.

A connected component is a maximal connected subgraph of G. Each vertex belongs to exactly one connected component, as does each edge.

A directed graph is called weakly connected directed graph if replacing all of its directed edges with undirected edges produces a connected (undirected) graph.

A directed graph is strongly connected or strong directed graph if it contains a directed path from u to v and a directed path from v to u for every pair of vertices u,v.

The strong components are the maximal strongly connected subgraphs.

A cut or vertex cut of a connected graph G is a set of vertices whose removal renders G disconnected. The connectivity or vertex connectivity ê(G) is the size of a smallest vertex cut. A graph is called k-connected or k-vertex-connected if its vertex connectivity is k or greater. A complete graph with n vertices has no cuts at all, but by convention its connectivity of a complete graph is n-1. A vertex cut for two vertices u and v is a set of vertices whose removal from the graph disconnects u and v. The local connectivity ê(u,v) is the size of a smallest vertex cut separating u and v. Local connectivity is symmetric for undirected graphs; that is, ê(u,v)=ê(v,u). Moreover, ê(G) equals the minimum of ê(u,v) over all pairs of vertices u,v. 2-connectivity is also called "biconnectivity" and 3-connectivity is also called "triconnectivity".

In analogous manner, the concepts of cut can be defined for edges also. A simple case is when by cutting a single, specific edge leaves the graph disconnected. Such an edge is called a bridge. More generally, the edge cut of G is a group of edges whose total removal renders the graph disconnected. The edge-connectivity ë(G) is the size of a smallest edge cut, and the **local edge-connectivity** ë(u,v) of two vertices u,v is the size of a smallest edge cut disconnecting u from v. Again, local edge-connectivity is symmetric. A graph is called k-edge-connected if its edge connectivity is k or greater.

To determine the connected components in undirected and directed graphs, the algorithm for connected components and strongly connected components are described below.

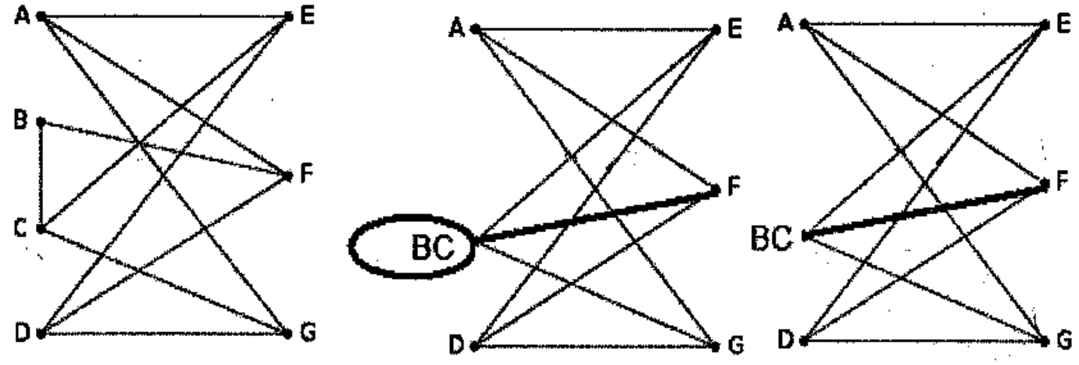
9.3.1 Algorithm to find Connected Components

This section describes algorithm to find connected components in undirected graph. The same can be used to find weakly connected components in directed graph. As such these components can also be obtained in linear time of edges using DFS or BFS also if only one component is available in the graph. However, for more than one component, DFS or BFS have to be repeated again with updation of a variable for component count. Here, we describe an algorithm that uses the concept of fusion of vertices and at last only as many vertices remain as the number of connected components.

Two connected vertices are said to have been fused if

- a new vertex has been created in place of both
- all edges connecting to both of them are connected to new vertex and

self-loops and multiple edges between any pair of vertices if any are removed.



(a) a Graph (b) after fusion of B & C (c) after self-loop removal

Figure 9.7(a)-(c): Fusion of vertices.

Figure 9.7 (a) shows a graph in which vertices B & C are to be fused. Vertices B & C are fused and a new vertex BC is created by connecting all edges B & C to BC in Figure 9.7(b). In this process, edge (B, C) also results a self-loop as shown in Figure 9.14(b), the self-loop has been removed in Figure 9.7(c).

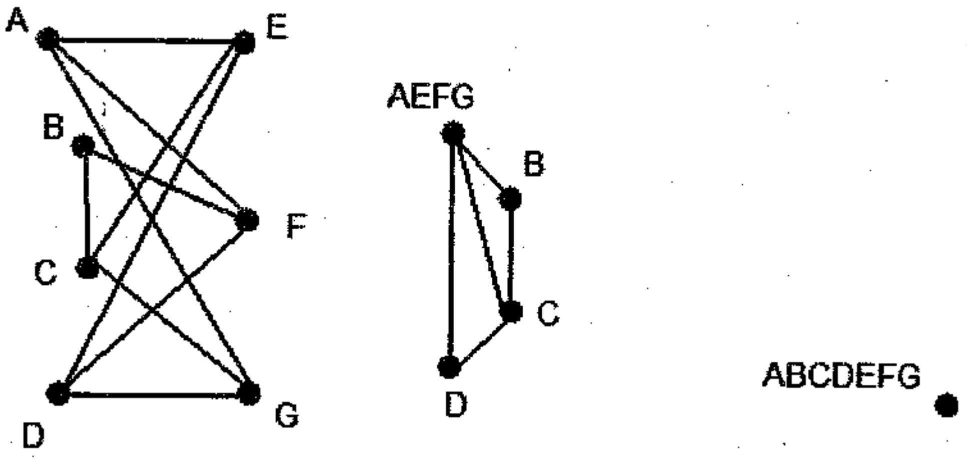
Now, fusion based algorithm becomes very simple as it runs in linear time of edges.

```
Algorithm ConnectedComponents(Graph G)
—Initializations
Subgraph g ? G
ComponentCount ? 0;
Repeat
    ComponentCount ? ComponentCount + 1;
    Select a vertex i in g
    Repeat
        Fuse all adjacent vertices in i and rename fused vertex as i
    Until no more adjacent vertex to i left
```

```
10.    delete vertex i(with all fused vertices)from g
11.    call remaining subgraph as g
12. Until no more vertices left
13. Print ComponentCount as number of components
```

Algorithm : Algorithm for Connected Components

The algorithm runs in linear time in number of edges as line 7-9 tests all edges belonging to one component and removes the edges. Outer loop repeats for every component. Hence, one edge is tested only once for connected components.



(a) Initial Graph **(b) after one iteration line 7-9** **(c) at the end**

Figure 9.8(a) – (c): Example of connected component

Figure 9.8(a) shows a graph for which connected components are to be found. Starting from vertex A, line 7-9 fuses all adjacent vertices as shown in Figure 9.8(b). At last only one vertex remains in Figure 9.8(c), which indicate only one component.

9.3.2 Algorithm to find Strongly Connected Components

Strongly Connected Components (SCC) can be obtained in directed graph. In each component, there exists a path between every pair of vertices. A number of algorithms to find SCC are available. Most important ones are given by Kosaraju, Gabow and Tarjan. **Tarjan Algorithm** (named for its discoverer, Robert Tarjan) is very popular algorithm for finding the strongly connected components of a graph. It can be seen as an improved version of Kosaraju algorithm, and is comparable in efficiency to Gabow algorithm. Here, we describe Tarjan Algorithm.

As such SCC forms a kind of cycle in directed graph such that every pair of vertices has a path from each other. The algorithm uses DFS from a start node. The SCCs form the subtrees of the DFS tree, the roots of which are the roots of the SCCs. The first point belonging to SCC in DFS tree is defined as root of SCC.

The nodes are placed on a stack in the order in which they are visited. When the search returns from a subtree, the nodes are taken from the stack and it is determined whether each node is the root of a strongly connected component. If a node is the root of a strongly connected component, then it and all of the nodes taken off before it form that strongly connected component.

The crux of the algorithm comes in determining whether a node is the root of a SCC. To do this, each node is given a DFS index v.index, which numbers the nodes consecutively in the order in which they are discovered. In addition, each node is assigned a value v.lowlink that satisfies v.lowlink ? min {v'.index: v' is reachable from v}. Therefore v is the root of a SCC if and only if v.lowlink = v.index. The value v.lowlink is computed during the depth first search such that it is always known when needed.

Algorithm

```
Input: Graph G (V, E)

index ? 0                 — DFS node number counter
S ? empty                 — An empty stack of nodes
forall v in V do
 if (v.index is undefined) — Start a DFS at each node
    tarjan(v)             — we haven't visited yet

procedure tarjan(v)
 v.index ? index          — Set the depth index for v
 v.lowlink ? index
 index ? index + 1
 S.push(v)     — Push v on the stack
 forall (v, v') in E do      — Consider successors of v
  if (v'.index is undefined)  — Was successor v' visited?
     tarjan(v')             — Recurse
   v.lowlink ? min(v.lowlink, v'.lowlink)
  else if (v' is in S)    — Was successor v' in stack S?
      v.lowlink ? min(v.lowlink, v'.index)
if (v.lowlink == v.index)            — Is v the root of an SCC?
print "SCC:"
repeat
   v' = S.pop
   print v'
until (v' == v)
```

Algorithm : Algorithm for Strongly Connected Components

As such the algorithm is based on DFS and self explanatory. Hence, there is no need to show step by step processing in example. However, Figure 9.9 shows a directed graph and its three SCCs.

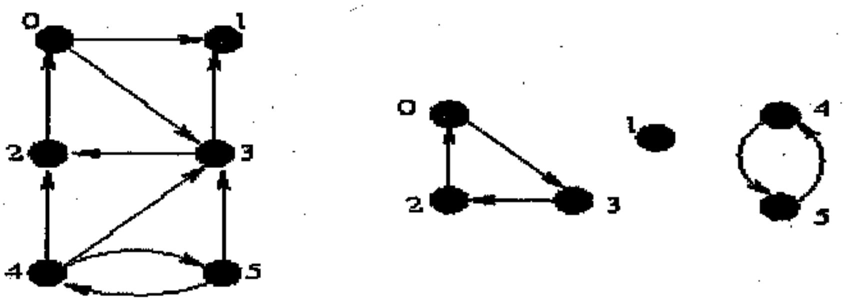


Figure 9.9: Directed graph and its Strongly Connected Components

Complexity

1. Complexity: The Tarjan procedure is called once for each node; the forall statement considers each edge at most twice. The algorithm running time is therefore linear in the number of edges in G, ie O(| V | + | E |).

2. The test for whether v' is on the stack should be done in constant time, for example, by testing a flag stored on each node that indicates whether it is on the stack.

9.4 Planarity Testing

In graph theory, the planarity testing problem asks whether, given a graph, that graph is a planar graph (can be drawn in the plane without edge intersections). This is a well-studied problem in computer science for which many practical algorithms have emerged, many taking advantage of novel data structures. Most of these methods operate in O(n) time (linear time), where n is the number of edges (or vertices) in the graph, which is asymptotically optimal.

Simple algorithms and planarity characterizations

By Fáry's theorem we can assume the edges in the graph drawing, if any, are straight line segments. Given such a drawing for the graph, we can verify that there are no crossings using well-known line segment intersection algorithms that operate in O(n log n) time. However, this is not a particularly good solution, for several reasons:

* There's no obvious way to find a drawing, a problem which is considerably more difficult than planarity testing;

* Line segment intersection algorithms are more expensive than good planarity testing algorithms;

* It does not extend to verifying nonplanarity, since there is no obvious way of enumerating all possible drawings.

For these reasons, planarity testing algorithms take advantage of theorems in graph theory that characterize the set of planar graphs in terms that are independent of graph drawings. One of these is Kuratowski's theorem, which states that:

A finite graph is planar if and only if it does not contain a subgraph that is a subdivision of K5 (the complete graph on five vertices) or K3,3 (complete bipartite graph on six vertices, three of which connect to each of the other three).

A graph can be demonstrated to be nonplanar by exhibiting a subgraph matching the above description, and this can be easily verified, which places the problem in co-NP. However, this also doesn't by itself produce a good algorithm, since there are a large number of subgraphs to consider (K5 and K3,3 are fixed in size, but a graph can contain 2?(m) subdivisions of them).

A simple theorem allows graphs with too many edges to be quickly determined to be nonplanar, but cannot be used to establish planarity. If v is the number of vertices (at least 3) and e is the number of edges, then the following imply nonplanarity:

$e > 3v - 6$ or;

There are no cycles of length 3 and $e > 2v - 4$.

For this reason n can be taken to be either the number of vertices or edges when using big O notation with planar graphs, since they differ by at most a constant multiple.

For further study you can refere books refered at the end of the unit.

Self learning exercises

1.Following edges form an undirected graph:

(a, b), (a, g), (b, c), (c, d), (d, e), (c, e), (g, c), (g, i), (g, h), (c, f), (f, j), (i, j), (h, j)

BFS algorithm starting from a will result in sequence of vertices

(a) a, b, c, d, e, f, j, i, g, h (b) a, b, c, f, j, h, g, i, d, e

(c) a, b, c, e, d, g, i, j, h, f (d) a, b, g, c, i, h, d, e, f, j

2.If DFS algorithm is applied for above graph and an adjacency matrix is used for graph representation. The traversal starts from vertex a in algorithm. Stack contains a, b, c. Vertex e has just come out of the stack. What will be the complexity of algorithm and which vertex might have come out of stack prior to e?

(a) $O(|V|^2)$, d (b) $O(|V|+|E|)$, j

(c) $O(|V|)$, h (d) $O(|E|)$, i

3.Single-destination shortest path problem can be converted into single-source shortest path problem by

(a) adding some more edges (b) reversing direction of edges

(c) removing direction of edges (d) none of the above

4. Dijkstra Algorithm updates the weights associated with edges after extracting

(a) a maximum weight vertex from queue

(b) an arbitrary vertex from queue

(c) a minimum weight vertex from priority queue

(d) none of the above

5. Which statement is true for a Minimum Spanning Tree (MST) of a graph?

(a) There cannot be more than one MST.

(b) There can be more than one MSTs with different weights.

(c) There can be more than one MST with same weight.

(d) None of the above.

6. As Kruskal Algorithm progresses, it

(a) keep selecting edges with minimum weight not forming a cycle.

(b) maintains a minimum weight tree right from beginning.

(c) keep removing maximum weight edges from graph.

(d) none of the above

7. For a planar graph, planarity is likely to be affected when

(a) an edge is splitted into two by adding a vertex.

(b) some edges are added to make it a complete graph of more than 5 vertices.

(c) all vertices of degree two are merged into adjacent vertices sequentially.

(d) none of the above.

8. A graph having n vertices and e edges can be planar only if

(a) $e > 3n + 6$ (b) $n = 3e$ (c) $e = 3n - 6$ (d) $e = 3n + 6$

9. Edge listing of a directed graph is given below:

(a, b), (a, c), (c, d), (d, f), (c, e), (e, a)

Vertex set of strongly connected component of this graph will be

(a) {a, b, c} (b) {a, c, e} (c) {b, a, c} (d) {c, d, e}

10. A weakly connected component can be obtained by

(a) removing direction of edges and obtaining connected component

(b) reversing the direction of edges and obtaining strongly connected component

(c) finding any spanning tree

(d) none of the above

9.5 SUMMARY

This unit deals with standard algorithms useful in various applications. Section wise summary is as follows:

· Two search algorithms are described in previous module , namely Breadth First Search (BFS) and Depth First Search (DFS). Both the algorithms visit every node of the graph traversing through edges but in different way. BFS traverses all adjacent vertices of current vertices starting from root and maintains a queue for this purpose. DFS traverses only one adjacent vertex till no more vertices can be traversed and maintains stack for this purpose.

· Shortest Path algorithms are found in three variants namely, single-source shortest path problem, single-destination shortest path problem and all-pairs shortest path problem. Single-source and single-destination problems can be converted to each other just by reversing the edge direction.

Dijkstra Algorithm solves single-source problem if weights of edges do not form negative cycle. Bellman-Ford algorithm detects negative cycle if it exists and solves if does not exist. Floyd-Warshall algorithm solves all-pairs shortest path problem in which shortest route can be found between every pair of vertices.

· Planar graph is one which can be embedded in a plane. In other words the edges of the graph do not intersect each other. Two interesting graphs known as Kuratowski's two graphs are smallest non-planar graphs, one with minimum number of vertices and other with smallest number of edges. A number of planarity criteria and algorithm to detect planarity have been discussed.

· Spanning tree of a graph is a tree containing all vertices and a subset of edges of the graph. A spanning tree having minimum sum of weights of edges in a weighted graph is called Minimum Spanning Tree (MST). Prim and Kruskal algorithms with greedy approach obtains MST. The trees obtained by these algorithms may not be the same, but sum of weights of edges of the tree will be the same.

· Connectedness of a graph can find whether there are disconnected vertices exist in the graph or not. If yes then connected components can be found. In a directed graph, there can be two types of connectivity namely, weak and strong. Accordingly, weakly connected components can be obtained by removing direction of edges whereas strongly connected components can be obtained by retaining direction also. Algorithms to find components in undirected graph (can also obtain weakly connected components) and strongly connected components are also described in section

9.6 Glossary

Planar Graph : A planar graph is one that can be drawn on a plane in such a way that there are no "edge crossings," i.e. edges intersect only at their common vertices.

connected graph: A connected graph is a graph such that there exists a path between all pairs of vertices. If the graph is a directed graph, and there exists a path from each vertex to every other vertex, then it is a strongly connected graph.

Articulation Point : Articulation points in a network are those which are critical to communication: for an articulation point, all paths between certain nodes have to pass through this point. A vertex in a connected undirected graph is an articulation point if removing it and all edges incident to it results in a non-connected graph; in other words, removing a vertex leads a separation of its subtree from the other part of a graph. A connected graph is biconnected if it has no articulation points. A biconnected component of an undirected graph G = (V, E) is a maximal subset B of the edges with the property that the graph GB = (VB, B) is biconnected, where VB is the subset of vertices incident to edges in B. There is a very close relationship between biconnected components and articulation points. The articulation points are exactly the vertices at which two biconnected components are connected. Thus, the articulation point separates the biconnected components of a graph. The idea of articulation points and biconnected components plays an important role in any network graph in terms of its connectivity.

Planarity Testing : Problem to decide whether a given graph is embeddable in the plane, and if so, to get its embedding, otherwise to provide a proof of its non-planarity.

9.7 Further Readings

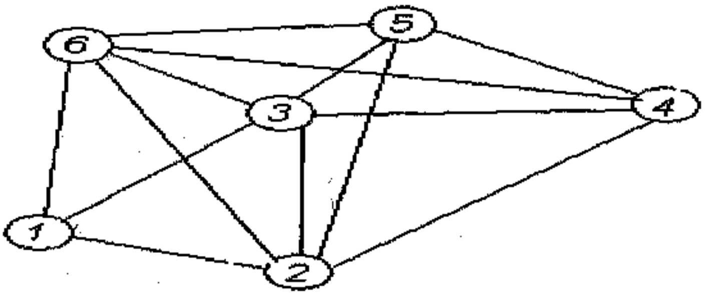
1. Graph theory with applications to Engineering and computer science by Narsingh Deo.
2. Discrete Mathematics by Rosen.
3. Graph theory by John Clark and Derek Allan Holton
4. Reference Book Allan Tucker for Combinatorics

9.8 Answers to self learning exercises

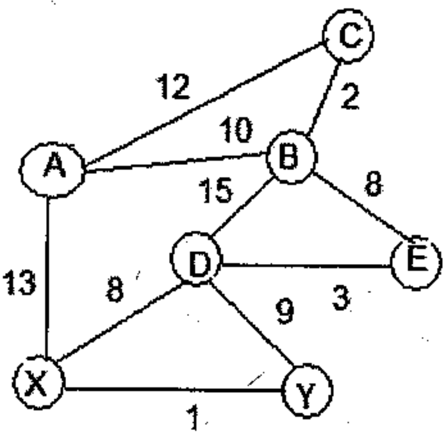
1(d) 2(a) 3(b) 4(c) 5(c) 6(a) 7(b) 8(c) 9(b) 10(a)

9.9 Unit End Questions

1. Find whether the graph given below is planar?



2 Obtain three least cost spanning trees including MST of the following graph.



3.Find BFS tree using root as B and DFS tree using root at X in graph given in Q. No. 2 and compare their costs.

4.Make the graph in Q. No. 2 as directed by assigning direction as per the edge list given below:
(A, C), (B, A), (X, A), (D, X), (D, Y), (X, Y), (E, D), (B, E), (D, B), (C, B)
Where (A, C) means direction is from A to C.
Find Strongly Connected Components in it.

5.In directed graph formed in Q. No. 4, change the weight of DB to -15. Perform Bellman-Ford algorithm for source C and show the weights of A, D, E and prove that it contains negative cycles.

---***---

UNIT X

Graph Theory Applications

Structure of the Unit

10.0 Objective

10.1 Introduction

10.2 Topological Sort

10.3 Network Flow Problems

10.4 Travelling Salesman

10.5 Assignment Problems

10.6 Summary

10.7 Glossary

10.8 Further Readings

10.9 Answers to self learning exercises

10.10 Unit--end Questions

10.0 Objectives

This unit describes applications in which graph theory fundamentals are useful. After reading this unit you will be able to understand

- Topological sort related problems
- Network Flow problem
- Travelling Salesman problem
- Assignment problem

For the above problems, problem description, mapping to graph and solution using graph theory fundamentals have been provided.

10.1 Introduction

Because of inherent simplicity in graph theory algorithms, it has found a wide range of applications in engineering, physical, social sciences and in numerous other areas. A graph can represent almost any physical situation involving distinct objects and relationship between them.

Any problem to be solved using graph theory requires minimum three steps

A.Description of problem and identification of objects and relationship between them.

B.Drawing graph using the objects and their relationship. Sometimes a mapping with some easy graph or isomorphic graph may also be required.

C.Identifying right algorithm and representing the result in desired form.

As an example, a cable network company wishes to lay down underground cable for connecting its stations setup in different cities. In step-A, complete city station list with their connecting distances amongst every pair of city has to be identified. A weighted graph has to be drawn in

Step-B. Finally a suitable algorithm to find the solution has to be found in Step-C. In this case minimum cost spanning tree algorithm will be suitable. Sometimes algorithm may not be suitable directly might need some modification or improvement.

This unit describes topological algorithm and problems that can be solved using topological sort in section 10.2. Section 10.3 explains network flow problem and algorithms used for solving them. Travelling salesman problems are important but difficult problems can be modelled using graph theory described in section 10.4. Finally interesting Assignment problem is discussed in section 10.5.

10.2 TOPOLOGICAL SORT

Sorting is very common in numbers for various purposes. However, sorting is also important in graphs especially in applications where scheduling activities are required. Critical Path Method (CPM) and Project Evaluation and Review Techniques (PERT) are some important applications need scheduling and hence topological sort is necessary. This section describes topological sorting method followed by two applications namely, Cloth-wearing by absent minded professor and curriculum planning problem.

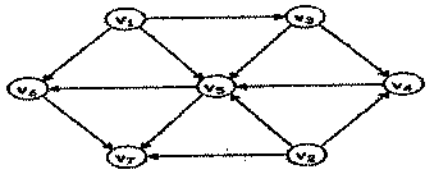
10.2.1. Topological Sort Definitions

As mentioned earlier that Topological sort is very important for scheduling of dependent activities. These dependent activities can be described as a directed acyclic graph (DAG). These activities are represented as a node for mapping with a graph and their dependency as their directed edge. If a DAG for such activities is possible, a topological sort can be used to schedule these activities. As such life is full of scheduling activities. In a curriculum development, the courses can be scheduled in any year of study should have completed its prerequisite courses in its earlier years. For any construction activity, arrival of raw material, availability of construction equipment, labour and storage with safety activities have to be scheduled very carefully.

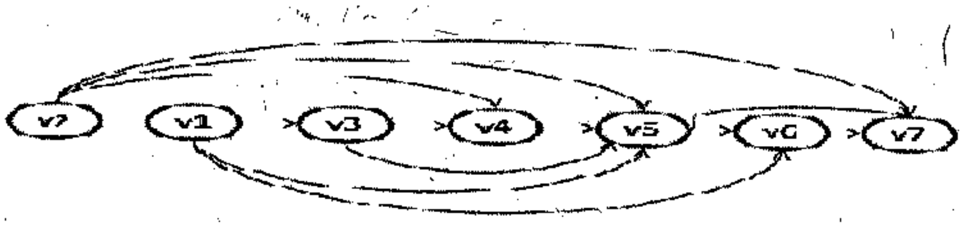
As described earlier, DAG is a directed graph which does not contain any cycle. A scheduling provides linear order to these vertices such that these activities are executed in that order. The linear order can be defined as a number associated with a vertex (representing an activity). The number with vertex is a sequence in which the activity should be executed. A topological sort takes a directed acyclic graph in its input and assigns a numerical value to each vertex according to its scheduling order.

A topological sort of a directed acyclic graph (DAG) G is an ordering of the vertices of G such that for every edge (v_i, v_j) of G we have $i < j$. That is, a topological sort is a linear ordering of all its vertices such that if DAG G contains an edge (v_i, v_j), then v_i appears before v_j in the ordering. In other words, the vertices of a graph can be said in topological order if they are labelled 1, 2, 3, ..., n such that every edge in G leads from smaller numbered vertex to a larger one. The process of relabeling is called topological sorting.

Figure 10.1(a) shows an example in which a DAG consists of seven vertices numbered v_1 to v_7. One scheduling order obtained after topological sorting is shown in Figure 10.1(b).



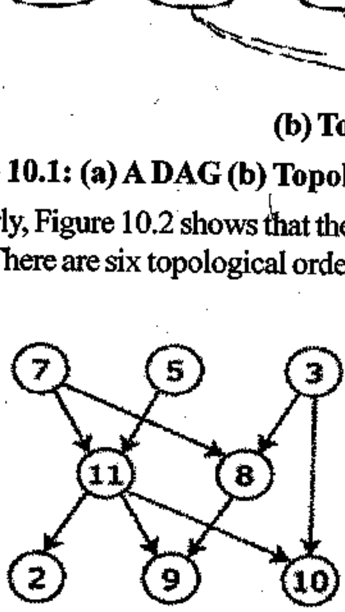
(a) A DAG



(b) Topologically ordered DAG

Figure 10.1: (a) A DAG (b) Topological order: $v_2, v_1, v_3, v_4, v_5, v_6, v_7$.

Similarly, Figure 10.2 shows that there can be more than one topological order possible in a DAG. There are six topological orders shown for the DAG shown in the Figure 10.2.



8.0 7, 5, 3, 11, 8, 2, 9, 10 (visual left-to-right)
8.1 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered availabl vertex first)
8.2 3, 7, 8, 5, 11, 10, 2, 9
8.3 5, 7, 3, 8, 11, 10, 9, 2 (least number of edges first)
8.4 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
8.5 7, 5, 11, 2, 3, 8, 9, 10 (visual top-to-bottom)

(a) A DAG (b) Various Topologically orders

Figure 10.2: (a) A DAG (b) Topological order: $v_2, v_1, v_3, v_4, v_5, v_6, v_7$.

It is important to note that

·If the graph is not acyclic, then no linear ordering is possible. That is, we must not have circularities in the directed graph. For example, in order to get a job you need to have work experience, but in order to get work experience you need to have a job.

·There can be more than one topological order possible for one DAG. However, if DAG contains a Hamiltonian Path then only one topologically sorted sequence available for that DAG. Actually a Hamiltonian Path is defined in undirected graph. But the same definition can be extended to DAG also.

The above points can be proved in the following Theorems.

Theorem 10.1: A directed graph has a topological ordering if and only if it is acyclic.

Proof: The proof can be shown in two parts.

Part 1 – A DAG G has a topological ordering if is G acyclic.

The above can be proved by Contradiction.

1.Suppose G has a topological order and G also has a cycle.

2.As there exists topological ordering in G, there must exist $i_0, < i_1, < \ldots < i_{k-1.}$

3.As G also contain a cycle, there must exist $i_0, < i_1, < . i_r . . < i_{k-1} < i_r$ for some i_r where cycle is formed at i_r. This is clearly impossible.
Therefore, G must be acyclic.

Part 2 - A DAG G is acyclic if has a topological ordering.

1.Let G be acyclic.

2.Since is G acyclic, it must have a vertex with no incoming edges. Let v_1 be such a vertex. If we remove v_1 from graph, together with its outgoing edges, the resulting digraph is still acyclic. Hence resulting digraph also has a vertex.

Theorem 10.2: A DAG has only one topological ordering if it has Hamiltonian Path.

Proof: Suppose a DAG G has a Hamiltonian Path. The path starts from one vertex of the graph and covers rest of the vertices through connecting edges such that each vertex is traversed exactly once.

The above theorem can also be proved by Contradiction.

1.Suppose G has more than one topological order.

2.One order can be obtained by traversing Hamiltonian Path and topological ordering in G is defined by numbering $i_0, < i_1, < \ldots < i_{k-1.}$

3.Now, another path can be obtained by removing one edge and adding another edge only. This is possible only if there exists a cycle in DAG as all vertices are already covered in Hamiltonian Path

4.Hence, G cannot have another topological order.

10.2.1. Topological Sort Algorithms

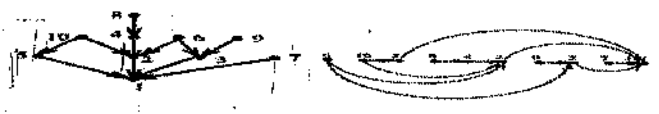
The usual algorithms for topological sorting have running time linear in the number of nodes plus the number of edges (O(|V|+|E|)).

One of these algorithms was first described by Kahn in 1962. The algorithm works by choosing vertices in the same order as the eventual topological sort will be. First, find a list of "start nodes" which have no incoming edges and insert them into a set S; at least one such node must exist if graph is acyclic. This process continues till end.

```
L ? Empty list that will contain the sorted elements
S ? Set of all nodes with no incoming edges
while S is non-empty do
    remove a node n from S
    insert n into L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    output error message (graph has at least one cycle)
else
    output message (proposed topologically sorted order: L)
```

Algorithm 10.1: Algorithm to find Topological Sort.

If the graph was a DAG, a solution is contained in the list L (the solution is not unique). Otherwise, the graph has at least one cycle and therefore a topological sorting is impossible.



INPUT OUTPUT

Figure 10.3: Input and Output graph for a topological sort.

It may be noticed that, reflecting the non-uniqueness of the resulting sort, the structure S can be simply a set or a queue or a stack. Depending on the order that nodes n are removed from set S, a different solution is created.

10.2.3Topological Sort Applications

In this subsection, we describe two applications of topological sort namely, cloth-bearing by absent-minded professor and finding curriculum planning.

10.2.3.1Cloth-Bearing By Absent Minded-Professor

An absent-minded Professor has a problem when getting ready to go to work in the morning. He sometimes forgets to follow order when he is getting ready to work in the morning and dresses out of order. For example, he might put his shoes on before putting the socks on. Later he realizes takes the shoes off and then put the socks on and then the shoes back on. This can be described and solved following the steps for topological sort.

STEP-A: An absent-minded professor has to be given a proper and defined order of bearing cloths he does not waste time in backtracking because of out of order bearing cloths. He has to bear shirt, tie, belt, shorts, pants, shoes, shocks, watch and jacket that have to be put on in a certain order.

STEP-B: Activities of bearing individual cloth can be mapped to a node of the graph. Dependency of the activities can be expressed through directed edges of the graph. The graph essentially forms a DAG as no activity has to be done repeatedly as there will be no backtracking if proper order is followed.

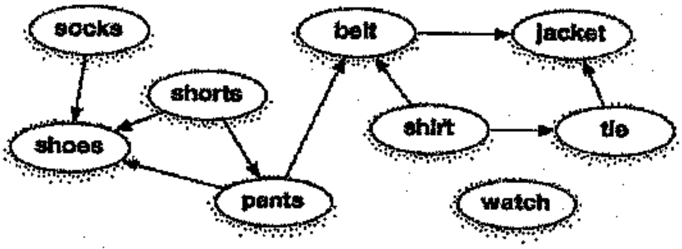


Figure 10.4: Cloth-bearing dependency graph

The order between different parts of clothing forms a graph: shorts before pants means there is an edge between shorts and pants. (We generally call a graph like this a dependency graph, because it describes dependencies between pairs of tasks.)

STEP-C: This problem can be solved using Topological sort for which the algorithm has been described above. As a result a number is assigned to each vertex in dependency graph; following can be one of topological order:

Shorts(1/9), pants(2/9), belt(6/9), jacket(8/9), shoes (4/9), socks (3/9), tie(7/9), watch(9/9), shirt(5/9).

10.2.3.1Curriculum Planning Problem

In curriculum planning for a professional course like MCA, there can be prerequisite courses and corequisite courses for any specific course. Prerequisite courses are the courses must have been read by students prior to studying that specific course while corequiste courses must be taken concurrently. A problem of course development is defined as follows in Step-A:

Table 10.1 Curriculum Development Plan

S.No.	Course Name	Course Code	Prerequisites	Corequisites
1	C	C01		
2	C++	C02	C01	
3	Java	C03	C02	
4	Data Structure	C04		
5	Algorithms	C05	C04	
6	Advanced Algorithms	C06	C05	
7	DBMS	C07	C04, C13	
8	Operating Systems	C08	C09,C07	
9	System Software	C09	C04,C11	C07
10	Artificial Intelligence	C10	C05	
11	Microprocessor	C11	C01	
12	Embedded Systems	C12	C11	
13	File Structure	C13	C04	

Table 10.1 Curriculum Development Plan

In Step-B, the problem can be mapped to a graph. There will be vertex for every course planned. There will be an edge from prerequisite course to current course. Corequisite courses can be merged into one vertex for scheduling purpose.

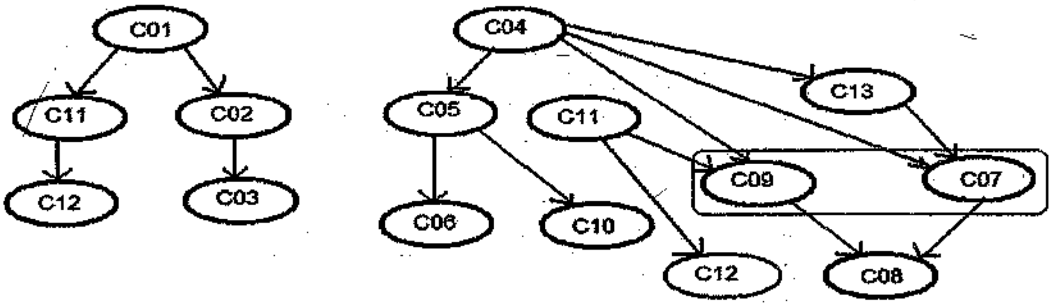


Figure 10.5: Dependency graph for curriculum planning

In Step-C, a right algorithm needs to be found. By nature of the problem, it is clear that there will be no cycle in the DAG. Topological sort algorithm can be used to find topological order with one modification. Instead of selecting one zero degree vertex at a time, all zero degree vertices will be selected simultaneously and assigned same number as semester number in which the course is to be scheduled.

Modified algorithm will look like as follows:

```
L ? Empty list that will contain the sorted elements
S1 ? Set of all nodes with no incoming edges
S2 ? To store intermediate nodes with no incoming edges
N ? 1; semester number assigned to all vertices in S1
while S1 is non-empty do
    Assign N to all nodes in S1
    for each node n in S1 do
        for each node m in G with an edge e from n to m do
            remove edge e from the graph
            if m has no other incoming edges then
                insert m into S2
        end do
        remove node n from S1
        insert n into L
    end do
    N ? N+1
    Move all nodes in S2 to S1
end do
if graph has edges then
    output error message (graph has at least one cycle)
else
    output message (proposed topologically sorted order: L)
```

Algorithm 10.2: Algorithm to find Curriculum Plan using Topological Sort.

Accordingly, earliest scheduling of the courses can be given

S.No.	Course Numbers	Schedule Number
1	C01, C04, C11	1
2	C11, C02, C05, C12, C13	2
3	C12, C03, C06, C10, C09, C07	3
4	C08	4

Table 10.2: One Solution of Curriculum Development Plan

10.3 Network Flow Problems

Network flow is one important use of graphs and fundamentals of graph theory can be used to solve these problems. Water and gas distribution through pipe and, electricity distribution through wire are some notable example of Network flow. Following scenario explains the flow problem and it is modelling as graph problem.

An exporter needs to ship several boxes of some product to a major city in India. There are various routes that the shipment could take. The possible routes are laid out in the directed graph below.

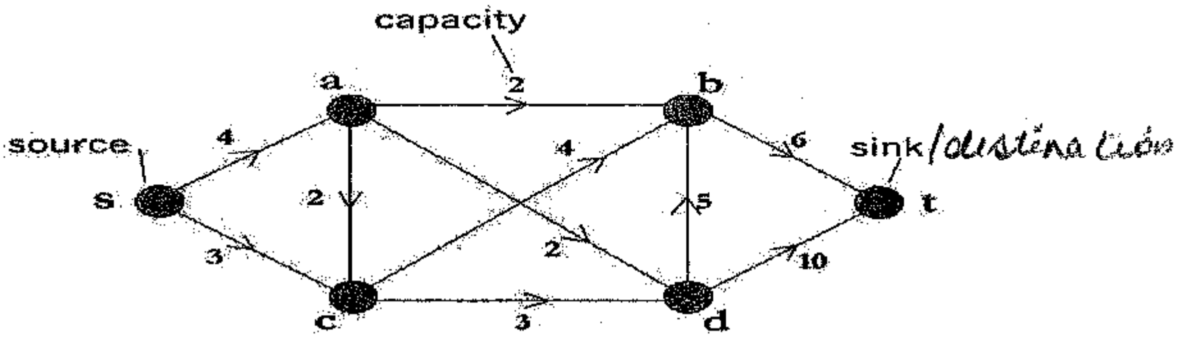


Figure 10.6: Network Flow Graph Example

The graph in Figure 10.6 is called a network, and it must hold certain properties. The vertex s, referred to as the "source" of the network, represents the city in which export company is located. Notice that the inflow of the source must always be 0 i.e. there is no incoming edge connected to this vertex. This could be placed where the product manufacturing or storage takes place. Similarly, the vertex t, referred to as the "sink" of the network, represents the city to which the product is being shipped. Notice that the outflow of the sink must always be 0. The other vertices (a, b, c, d) represent several "middlemen." The number assigned to each edge, called the capacity of the edge, represents the maximum rate at which the product can be shipped over that particular edge (the capacity must be a non-negative number). Exporter naturally wishes to send the boxes of the product at the highest possible rate through the graph, but without exceeding the capacity of each edge.

Now, in order to show the rate at which the product is sent, each edge must also be labelled with a non-negative number called the flow. A flow must also hold certain properties. The flow on an edge must not exceed the capacity of the edge. This is referred to as the "capacity constraint". The total flow into any vertex must equal the total flow out of the vertex. This is referred to as "flow conservation." And due to flow conservation, it follows that the total flow out of the source must equal the total flow into the sink. This can be observed by looking closely at the network below that these properties hold true (the flow is shown inside of parentheses.)

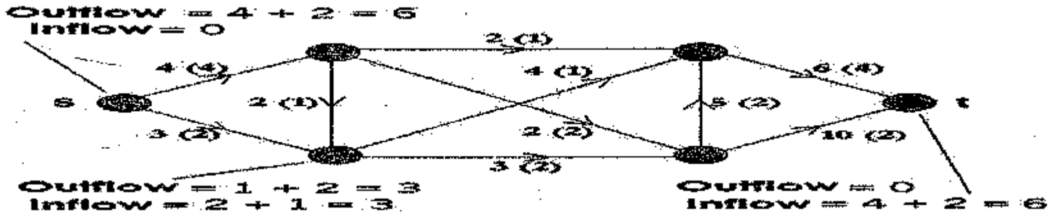


Figure 10.7: Maximum Flow from Source s to Sink t

The problem has to be represented in terms of Mathematical notations. Notations and assumptions used in algorithms are explained below.

Suppose $G(V, E)$ is a finite directed graph in which every edge $(u, v) \in E$ has a non-negative, real-valued capacity $c(u, v)$. If $(u, v) \notin E$, we assume that $c(u, v) = 0$. We distinguish two vertices: a source s and a sink t. A flow network is a real function $f : V \times V \rightarrow \mathbb{R}$ with the following three properties for all nodes u and v:

Capacity constraints: $f(u,v) \leq c(u,v)$. The flow along an edge can not exceed its capacity.

Skew symmetry: $f(u,v) = -f(v,u)$. The net flow from u to v must be the opposite of the net flow from v to u (see example).

Flow conservation: $\sum_{w \in V} f(u,w) = 0$, unless $u = s$ or $u = t$. The net flow to a node is zero, except for the source, which "produces" flow, and the sink, which "consumes" flow.

Notice that $f(u, v)$ is the *net* flow from u to v. If the graph represents a physical network, and if there is a real capacity of, for example, 4 units from u to v, and a real flow of 3 units from v to u, we have $f(u, v) = 1$ and $f(v, u) = -1$.

The **residual capacity** of an edge is $c_f(u, v) = c(u, v) - f(u, v)$. This defines a **residual network** denoted $G_f(V, E_f)$, giving the amount of *available* capacity. See that there can be an edge from u to v in the residual network, even though there is no edge from u to v in the original network. Since flows in opposite directions cancel out, *decreasing* the flow from v to u is the same as *increasing* the flow from u to v. An **augmenting path** is a path $(u_1, u_2, \ldots, u_k)$ in the residual network, where $u_1 = s$, $u_k = t$, and $c_f(u_i, u_{i+1}) > 0$. A network is at maximum flow if and only if there is no augmenting path in the residual network.

3-flow
3/5 - cap

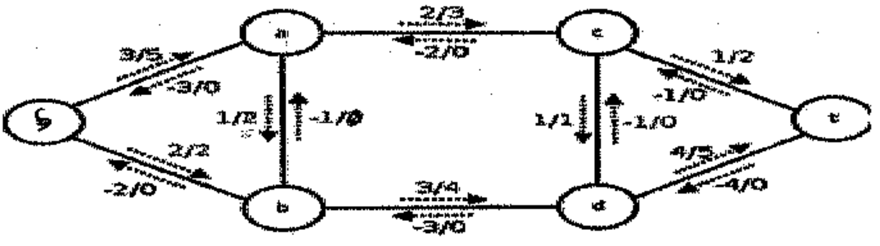


Figure 10.8: Example of Flow and capacity from Source s to Sink t.

In Figure 10.8, s to b capacity is 2, flow is 2 and no residual capacity is left in residual graph. An augmenting path (s, a, c, t) exists in residual graph in which 1 unit of flow is possible.

The network flow problem is normally defined as maximum flow problem i.e. to find a maximum feasible flow through a single-source, single-sink flow network. In order to find the maximum flow through the graph, there are many algorithm/ methods available. Min-Cut Max-Flow Theorem provides a useful proof for maximum flow. In iterative algorithms/ methods, the flow on each edge must be initialized to 0 and recursively attempt to improve on each flow until no longer improvement is not possible

10.3.1 Min-Cut Max-Flow theorem

Max-flow min-cut theorem states that in a flow network, the maximum amount of flow passing from the source to the sink is equal to the minimum capacity of the edges that needs to be removed from the network so that no flow can pass from the source to the sink. The set of edges removed is known as cut as defined earlier and the sum of capacities of the edges is known as capacity of cut.

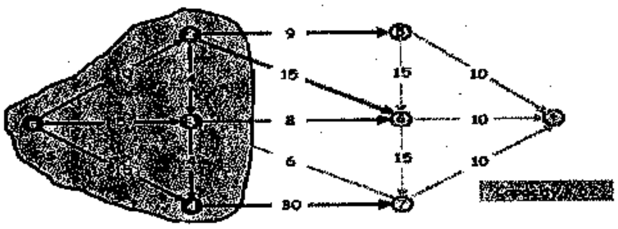


Figure 10.9: Flow Network and a cut between source & sink of capacity 62.

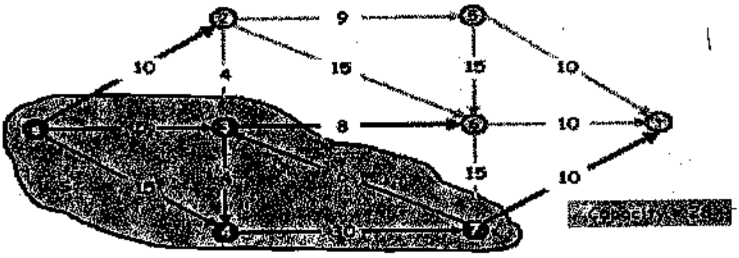


Figure 10.10: Flow Network and a cut between source & sink of capacity 28.

Figure 10.9 shows a cut between source and sink of capacity 62 and Figure 10.10 shows another cut between source and sink of capacity 28 (which is minimum) of the same network.

Theorem 10.3: Min-Cut Max-Flow theorem

Maximum flow from source to sink in a flow network cannot exceed the minimum capacity of any cut of edges between source and sink.

Proof: 1.If f is now flow in a network G = (V, E) with source s and sink t then the equivalence of the following conditions has to be proved:

(a) f is maximum flow in G

(b) The residual network G_f contains no augmenting path

(c) |f|=c(S, T) for some cut (S, T) in G.

This can be proved as per the following logic:

*1.*Condition (a) and (b) are obvious as if there exists some augmenting path, some more flow can be sent along that path and hence, f cannot be maximum.

2.As there is no augmenting path from s to t in G_f. S is a set of vertices in G_f such that there exists an augmenting path from s to all vertices in S. T is defined T = V – S. Partition (S, T) is a cut (s, t) such that s $\in$ S and t $\in$ T.

Figure 10.11 shows cut (S, T) with capacity 15

3. Residual capacities of the of the edges can be defined as

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } u \to v \in E \\ f(v,u) & \text{if } v \to u \in E \\ 0 & \text{otherwise} \end{cases}$$

As there is no augmenting path from source s to target t in residual graph, for every vertex u $\in$ S and v $\in$ T

$$c_f(u,v) = c(u,v) - f(u,v) + f(v,u) = 0$$

As there is no flow allowed,

$$c_f(u,v) = c(u,v) - f(u,v) = 0$$

and

$$f(v,u) = 0$$

This means that flow saturates every edge from S to T, which is equal to the capacity of cut.

4.If any other partition/cut is taken then there will be possibility of additional flow between two partitions, the same flow cannot be passed till sink. Hence, flow cannot be more than the capacity of cut.

Hence proved

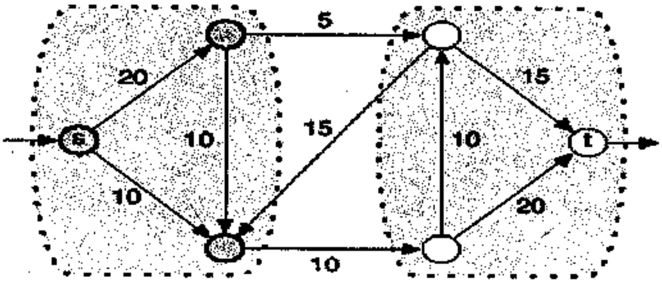


Figure 10.11: Partition(S, T) and Cut (s, t) divides source & sink of capacity 15.

10.3.2 Algorithms

There are many algorithms available to solve this problem. A list of commonly used algorithms with brief description is given below:

Method	Complexity	Description
Linear programming	--------	Constraints given by the definition of a legal flow.
Ford-Fulkerson algorithm	$O(E \cdot maxflow)$	As long as there is an open path through the residual graph, send the minimum of the residual capacities on the path. The algorithm works only if all weights are integers. Otherwise it is possible that the Ford-Fulkerson algorithm will not converge to the maximum value.
Edmonds-Karp algorithm	$O(VE^2)$	A specialization of Ford-Fulkerson, finding augmenting paths with breadth-first search.
Dinitz blocking flow algorithm	$O(V^2E)$	In each phase the algorithms builds layered graph with breadth-first search on the residual graph. The maximum flow in a layered graph can be calculated in $O(VE)$ time, and the maximum number of the phases is n ? 1.
General push-relabel maximum flow algorithm	$O(V^2E)$	The push relabel algorithm maintains a preflow, i.e. a flow function with the possibility of excess in the vertices. The algorithms runs while there is vertex with positive excess, i.e. active vertex in the graph. The push operation increases the flow on a residual edge, and a height function on the vertices controls which residual edges can be pushed. The height function is changed with relabel operation. The proper definitions of these operations guarantee that the resulting flow function is a maximum flow.
Push-relabel algorithm with *FIFO* vertex selection rule	$O(V^3)$	Push-relabel algorithm variant which always selects the most formerly actived vertex, and makes push operations until the excess is positive or there are admissible residual edges from this vertex.

Algorithm 10.3: Ford-Fulkerson Algorithm to find Maximum Flow.

Fulkerson-Ford and Edmond-Karp being important algorithms are described in this section.

10.3.2.1Fulkerson-Ford Algorithm

The Ford–Fulkerson algorithm (named for L. R. Ford, Jr. and D. R. Fulkerson) computes the maximum flow in a flow network. It was published in 1956. The name "Ford–Fulkerson" is often also used for the Edmonds–Karp algorithm, which is a specialization of Ford–Fulkerson.

The idea behind the algorithm is very simple: As long as there is a path from the source (start node) to the sink (end node), with available capacity on all edges in the augmenting path, we send flow along one of these paths. Then we find another augmenting path, and so on.

To describe the algorithm, notations and conditions are refreshed here. Given a graph G(V, E), with capacity c(u, v) and flow f(u, v) = 0 for the edge from u to v. We want to find the maximum flow from the source s to the sink t. After every step in the algorithm the following is maintained:

- $f(u,v) \leq c(u,v)$ The flow from u to v does not exceed the capacity.
- $f(u,v) = -f(v,u)$ Maintain the net flow between u and v. If in reality a units are going from u to v, and b units from v to u, maintain f(u,v) = a -b *and* f(v,u) = b - a.
- $\sum_{v} f(u,v) = 0 \Longleftrightarrow f_{\text{in}}(u) = f_{\text{out}}(u$ for all nodes u, except s and t. The amount of flow into a node equals the flow out of the node.

This means that the flow through the network is a legal flow after each round in the algorithm. **Residual network** $G_f(V,E_f)$ will be the network with capacity $c_f(u,v) = c(u,v)$ " f(u,v) and no flow. Notice that it can happen that a flow from v to u is allowed in the residual network, though disallowed in the original network: if f(u,v) > 0 and c(v,u) = 0 then $c_f(v,u) > 0$.

Algorithm Ford–Fulkerson

```
    Inputs Graph $G$ with flow capacity $c$, a source node $s$, and a sink node $t$
    Output A flow $f$ from to which is a maximum
        1. for all edges
        2. While there is a path from to in, such that for all edges:
            (A) Find
            (B) For each edge
                a) (Send flow along the path)
                b) (The flow might be "returned" later)
```

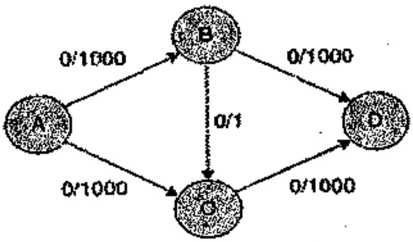
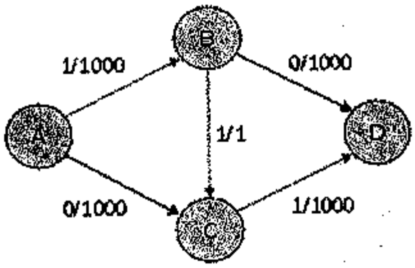
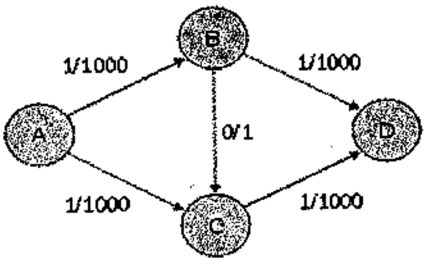
Algorithm 10.3: Ford-Fulkerson Algorithm to find Maximum Flow.

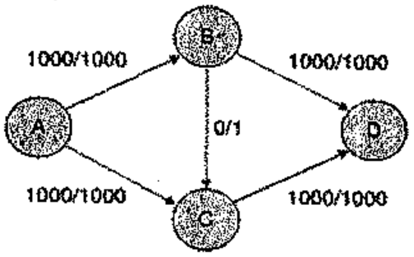
When no more paths in step 2 can be found, s will not be able to reach t in the residual network. If S is the set of nodes reachable by s in the residual network, then the total capacity in the original network of edges from S to the remainder of V is on the one hand equal to the total flow we found from s to t, and on the other hand serves as an upper bound for all such flows. This proves that the flow we found is maximal.

By adding the flow augmenting path to the flow already established in the graph, the maximum flow will be reached when no more flow augmenting paths can be found in the graph.

However, there is no certainty that this situation will ever be reached, so the best that can be guaranteed is that the answer will be correct if the algorithm terminates. In the case that the algorithm runs forever, the flow might not even converge towards the maximum flow. However, this situation only occurs with irrational flow values. When the capacities are integers, the runtime of Ford-Fulkerson is bounded by O(E*f), where E is the number of edges in the graph and f is the maximum flow in the graph. This is because each augmenting path can be found in O(E) time and increases the flow by an integer amount which is at least 1.

Following example illustrate the complexity computation using Ford-Fulkerson algorithm.

Path	Capacity	Resulting flow network
Initial flow network		 B 0/1000 0/1000 A 0/1 D 0/1000 0/1000 C
A,B,C, D	min(c_f(A,B),c_f(B,C),c_f(C,D)) = min(c(A,B) ? f(A,B),c(B,C) ? f(B,C),c(C,D) ? f(C,D)) = mln(1000 ? 0,1 ? 0,1000 ? 0) = 1	 B 1/1000 0/1000 A 1/1 D 0/1000 1/1000 C
A,C,B,D	min(c_f(A,C),c_f(C,B),c_f (B,D)) = min(c(A,C) ? f(A,C),c(C,B) ? f(C,B),c(B,D) ? f(B,D)) = min(1000 ? 0,0 ? (? 1),1000 ? 0) = 1	 B 1/1000 1/1000 A 0/1 D 1/1000 1/1000 C

After 1998 more steps ...	
Final flow network	

10.4: Example of Ford-Fulkerson algorithm for Maximum Flow Problem

It may be noticed that 1 unit flow is increasing after every step. There can be maximum E edges to be travelled in each step.

10.3.2.2 Edmond-Karp Algorithm

Edmonds–Karp algorithm is an implementation of the Ford–Fulkerson method for computing the maximum flow in a flow network in O(VE^2). It is asymptotically slower than the relabel-to-front algorithm, which runs in O(V^3), but it is often faster in practice for sparse graphs. The algorithm was first published by a Russian scientist, Dinic, in 1970, and independently by Jack Edmonds and Richard Karp in 1972 (discovered earlier). Dinic algorithm includes additional techniques that reduce the running time to O(V^{2}E).

The algorithm is identical to the Ford–Fulkerson algorithm, except that the search order when finding the augmenting path is defined. The path found must be the shortest path which has available capacity. This can be found by a breadth-first search, as we let edges have unit length. The running time of O(VE^2) is found by showing that each augmenting path can be found in O(E) time, that every time at least one of the E edges becomes saturated, that the distance from the saturated edge to the source along the augmenting path must be longer than last time it was saturated, and that the distance is at most V long. Another property of this algorithm is that the length of the shortest augmenting path increases monotonically.

```
Algorithm EdmondsKarp
 input:
   C[1..n, 1..n] (Capacity matrix)
   E[1..n, 1..n] (Neighbour lists)
    s        (Source)
    t        (Sink)
 output:
   f        (Value of maximum flow)
   F        (A matrix giving a legal flow with the maximum value)
f := 0 (Initial flow is zero)
```

```
F := array(1..n, 1..n) (Residual capacity from u to v is C[u,v] - F[u,v])
 forever
   m, P := BreadthFirstSearch(C, E, s, t) (m is flow allowed  t and P predecessors list)
    if m = 0
       break
    f := f + m
   (Backtrack search, and write flow)
    v := t
    while v ? s
       u := P[v]
       F[u,v] := F[u,v] + m
       F[v,u] := F[v,u] - m
        v := u
 return (f, F)
```

```
Algorithm BreadthFirstSearch
 input:
     C, E, s, t
 output:
     M[t]        (Capacity of path found)
     P           (Parent table)
 P := array(1..n)
 for u in 1..n
     P[u] := -1
 P[s] := -2 (make sure source is not rediscovered)
 M := array(1..n) (Capacity of found path to node)
 M[s] := 8
 Q := queue()
 Q.push(s)
 while Q.size() > 0
     u := Q.pop()
     for v in E[u]
       (If there is available capacity, and v is not seen before in search)
        if C[u,v] - F[u,v] > 0 and P[v] = -1
           P[v] := u
          M[v] := min(M[u], C[u,v] - F[u,v])
```

```
            if v ? t
              Q.push(v)
            else
              return M[t], P
    return 0, P
```

Algorithm 10.4: Edmond-Karp Algorithm to find Maximum Flow.

An example of running Edmond-Karp algorithm is shown below.

Given a network of seven nodes, source A, sink G, and capacities as shown below:

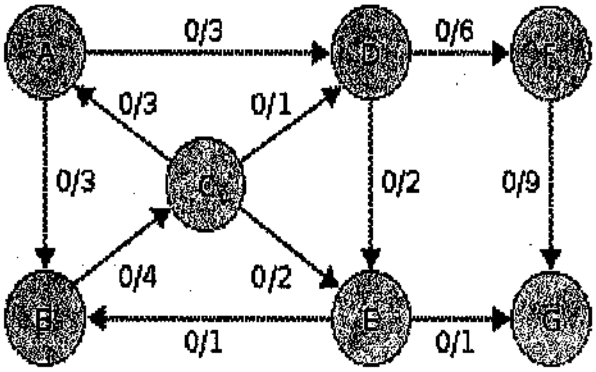


Figure 10.12: Example Flow Network with initial capacity.

In the pairs f / c written on the edges, f is the current flow, and c is the capacity. The residual capacity from u to v is $c_f(u,v) = c(u,v) - f(u,v)$, the total capacity, minus the flow that is already used. If the net flow from u to v is negative, it contributes to the residual capacity.

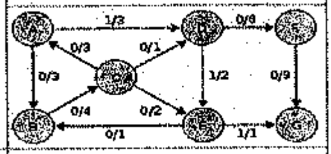
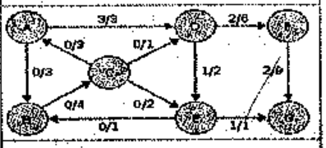
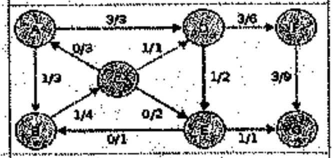
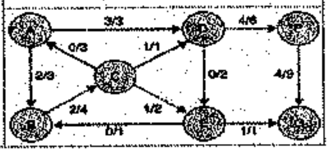
Capacity	Path
	Resulting network
m in(c_f(A ,D),c_f(D ,E),c_f(E ,G)) = m in(3 ? 0,2 ? 0,1 ? 0) = m in(3,2,1) = 1	A ,D ,E ,G
	 A D F C B E G 1/3 0/6 0/3 0/1 0/3 1/2 0/9 0/4 0/2 0/1 1/1
m in(c_f(A ,D),c_f(D ,F),c_f(F ,G)) = m in(3 ? 1,6 ? 0,9 ? 0) = m in(2,6,9) = 2	A ,D ,F ,G
	 A D F C B E G 3/3 2/6 0/3 0/1 0/3 1/2 2/9 0/4 0/2 0/1 1/1
m in(c_f(A ,B),c_f(B ,C),c_f(C ,D),c_f(D ,F),c_f(F ,G)) = m in(3 ? 0,4 ? 0,1 ? 0,6 ? 2,9 ? 2) = m in(3,4,1,4,7) = 1	A ,B ,C ,D ,F ,G
	 A D F C B E G 3/3 3/6 0/3 1/1 1/3 1/2 3/9 1/4 0/2 0/1 1/1
m in(c_f(A ,B),c_f(B ,C),c_f(C ,E),c_f(E ,D),c_f(D ,F),c_f(F ,G)) = m in(3 ? 1,4 ? 1,2 ? 0,0 ? ? 1,6 ? 3,9 ? 3) = m in(2,3,2,1,3,6) = 1	A ,B ,C ,E ,D ,F ,G
	 A D F C B E G 3/3 4/6 0/3 1/1 2/3 0/2 4/9 2/4 1/2 0/1 1/1

Table 10.5: Example of Edmond-Karp algorithm for Maximum Flow Problem

Notice how the length of the augmenting path found by the algorithm (in red) never decreases. The paths found are the shortest possible. The flow found is equal to the capacity across the minimum cut in the graph separating the source and the sink. There is only one minimal cut in this graph, partitioning the nodes into the sets {A,B,C,E} and {D,F,G}, with the capacity c(A,D) + c(C,D) + c(E,G) = 3 + 1 + 1 = 5.

10.4 Travelling Salesman Problem

Travelling Salesman Problem (commonly known as TSP) is a well known problem useful in many applications. This section defines and explains algorithms for TSP.

10.4.1 Problem Definition

A salesman wishes to sell his products in different cities. The cities are connected through different routes with different cost of travelling. Salesman wishes to visit all cities and comeback to original place with minimum cost of travelling. This problem is modelled as a graph by ,

a)Creating a vertex in a graph corresponding to every city.

b)Drawing edges between a pair of vertices if a travelling path is available between their corresponding cities.

c)Writing weight of the edge as cost associated with corresponding path between cities.

Figure shows an example of TSP with six cities connected with each other. Each vertex (A to B) represents a city and edge weight represents the cost of traveling. Thick lines in Figure (b) show a solution costing 20, which is minimum.

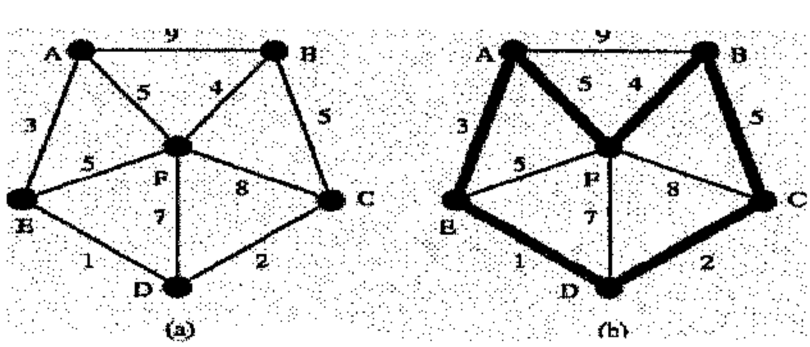


Figure 10.13: (a) A graph connecting 6 cities (b) Thick lines shows TSP solution

In above graph, TSP has been modeled as undirected graph, which is also known as symmetric TSP. In the symmetric TSP, the distance between two cities is the same in each direction. Thus, the underlying structure is an undirected graph; especially, each tour has the same length in both directions. This symmetry halves the number of feasible solutions.

Another type of TSP is asymmetric TSP. In the asymmetric TSP, the distance from one city to the other need not be equal to the distance in the other direction. In general, there may not even be a connection in the other direction. Thus, the underlying structure is a directed graph. For example, the asymmetric case models one-way streets or air-fares that depend on the direction of travel.

The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as genome sequencing. In these applications, the concept city represents, for example, customers, soldering points, or DNA fragments, and the concept distance represents travelling times or cost, or a similarity measure between DNA fragments. In many applications, additional constraints such as limited resources or time windows make the problem considerably harder.

10.4.2Algorithm

As TSP is modeled as a graph, a TSP tour is now a Hamiltonian cycle in the graph and an optimal TSP tour is a shortest Hamiltonian cycle. Often, the underlying graph is a complete graph, so that every pair of vertices is connected by an edge. This is a useful simplifying step, because it makes it easy to find a solution, however bad, because the Hamiltonian cycle problem in complete graphs is easy. Instances where not all cities are connected can be transformed into complete graphs by adding edges with very high weight between these cities, edges that will not appear in the optimal tour.

TSP problem has been shown to be NP-hard, and the decision problem version ("given the costs and a number x, decide whether there is a round-trip route cheaper than x") is NP-complete. Even though the problem is computationally difficult, a large number of heuristics and exact methods are known, so that some instances with tens of thousands of cities can be solved. Complexity of exact algorithms are of the $O(n^2 2^n)$. Dynamic Programming and Branch & Bound based algorithms are available for exact solution. But worst case complexity is $O(n^2 2^n)$ only. A number of approximation algorithms and heuristics are available to find sub-optimal solution. Least Cost Branch & Bound (LCBB) algorithm for exact solution is described here.

10.4.2.1Least Cost Branch & Bound Concept

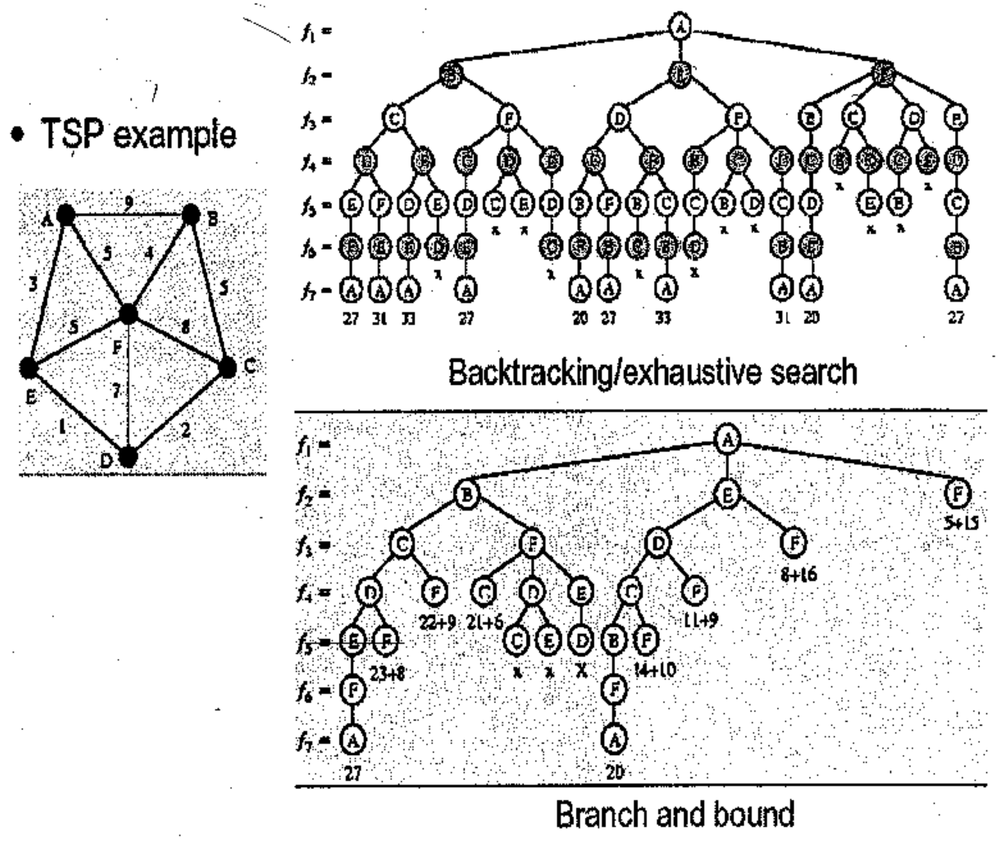
As discussed earlier, TSP is a complex problem. Exact solution can be attempted using Branch

& Bound based algorithm. First we explain the concept and algorithm of branch & bound and then representation of a TSP in matrix form for the purpose of LCBB will be discussed.

To understand the concept of branch and bound algorithm, the concept of exploration space need to be clear. Figure 10.14 shows an example of six cities and its exhaustive exploration space as well as advantage of branch & bound based search.

In Exhaustive approach, every possibility will be explored to find optimal travel cost. In Figure 10.14 exhaustive approach, from node A all possible routes will be traversed in depth first search manner for every possible route. While route A-B-C-D-E-F-A costs 27, route A-B-C-D-F-E-A will cost 31. But exhaustive approach will explore every possibility and will incur a huge cost of exploration.

On the other hand, branch and bound approach will maintain an intelligent bound to cut down exploration space significantly and will find optimal solution in drastically less time if bound function is suitable enough. In branch and bound approach of Figure 10.14, cost of route A-B-C-D-E-F-A as 27 was already stored. At any moment of exploration like traversing route A-B-C-F costs 22 any traversal beyond this will exceed previous bound 27 and hence must be cut down or pruned. By this approach, branch and bound algorithms can significantly reduce exploration time while worst case time remains the same.



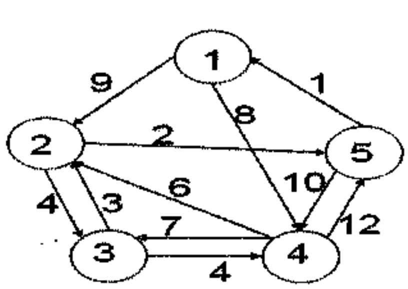
Backtracking/exhaustive search

Branch and bound

Figure 10.14: A graph connecting 6 cities and its Exhaustive and B & B based exploration space

In the context of TSP, a bare minimum cost can also be computed, which can be used as starting bound, called least cost. Hence, combining Branch & Bound with Least Cost (LCBB) algorithms are very popular for exact solution. Computation of least cost will be elaborated after matrix representation of the problem.

Matrix representation of TSP graph for the purpose of LCBB is explained here. Figure 10.15(a) shows an example of 5 cities indicating travelling cost between the cities and 10.15(b) shows matrix representation for the purpose of LCBB. Each element of the matrix represents the cost of travelling between the corresponding cities if they are connected otherwise a very high number like 999 (for infinite, as it cannot be stored in computer). For example element (2, 3) contains the value 4 corresponding to travel cost for vertex no. 2 to 3, while (2,4) contains 999 as there is no direct/edge from vertex no 2 to 4; While vertex no. 4 to 2 edge is there having cost 7.



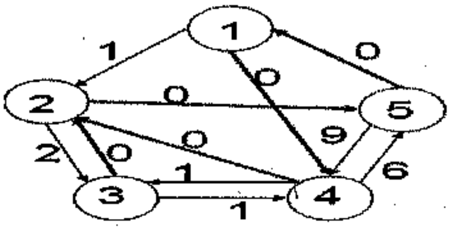
$$\begin{bmatrix} 999 & 9 & 999 & 8 & 999 \\ 999 & 999 & 4 & 999 & 2 \\ 999 & 3 & 999 & 4 & 999 \\ 999 & 6 & 7 & 999 & 12 \\ 1 & 999 & 999 & 10 & 999 \end{bmatrix}$$

(a) (b)

Figure 10.15: (a) An example of TSP graph (b) Matrix representation of TSP

Computation of Least Cost is based on the fact that for completing cycle, traversal of every vertex requires entry and exit atleast once. Cheapest cost of entry and exit edges (removing duplicate edge count) will be least cost. Once a matrix representation is available, it is as easy to compute by adding smallest element in every row and then every column.

In Figure 10.16(a) and (b), cheapest outgoing edges are (1, 4), (2, 5), (3, 2), (4, 2) and (5, 1). Total cost of these traversals will be 20. Accordingly, minimum row value in the matrix and edge weight in graph from every row/ edge has been reduced. However, these outgoing edges are also incoming edges for all vertices except vertex no. 3, which can be observed from graph as well as matrix. In matrix, column no. 3 has all entries non-zero. To cover vertex no. 3 also for incoming edges, we need to reduce column no. 3 by 1 and change graph accordingly. Hence, the least cost for the graph of Figure 10.15 or 10.16 will be 21.



$$\begin{pmatrix} 999 & 1 & 999 & 0 & 999 \\ 999 & 999 & 2 & 999 & 0 \\ 999 & 0 & 999 & 1 & 999 \\ 999 & 0 & 1 & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{pmatrix}$$

(a) (b)

Figure 10.16: (a) Reduced graph after outgoing edges least cost computation (b) Reduced cost matrix after outgoing edges least cost computation

Having described matrix representation and computation of least cost, we describe branch and bound algorithm.

10.4.2.2 LCBB Algorithm

LCBB is basically a branch and bound algorithm only with reduced matrix and pre-computed least cost. This least cost will be added in the solution obtained from reduced matrix. Branch and bound algorithm starts from root node and then visits child nodes intelligently. While visiting child node, there may be 3 possibilities namely, completion of one solution, non-promising branch and continue. In completion of one solution, with visiting of last node, all nodes have been explored and likely a new solution. In non-promising branch, current bound cannot be less than best cost solution available till now and hence, no need to further explore. In continue, neither end has occurred nor cost has exceeded bound value. Algorithm is illustrated as below:

Algorithm LCBB

Inputs Graph **G** as Reduced cost matrix and Least Cost value

Output minimum cost Tour

```
Current_best = one_solution;    —stores minimum cost route till now
Current_Cost = one_solution_cost;
S = Root_Node;                  —set S will contain intermediate nodes
while (S ? Ø) do
    {
    select an element s S;
        remove s from S;
        Make a branch based on s yielding sequences ($s_i$, i=1,2,3,....,m);
                —$s_i$ should not be a part of tree s
        for (i =1 to m)
            {
            compute the lower bound $b_i$ of $s_i$;
            If ($b_i$ = Current_cost)
                Kill $s_i$;             —remove non-promising branch
            else
                {
                if ($s_i$ corresponds to a complete solution)
                    {
                    Current_best = Root_node to $s_i$ route;
                    Current_cost = cost of new solution;    —store new solution value
                    }
                else
                    Add $s_i$ to set S and become child of s;   —add this node for new solution
24.                 }
25.             }
26.         }
27.     }
```

Algorithm 10.5: LCBB Algorithm to find optimal solution of TSP.

The above algorithm incorporates pruning or killing condition at line no. 14, a complete solution at line no. 19 and continuing exploration condition at line no. 23 of the algorithm. The tree obtained for Figure 10.15(a) is given in Figurer 10.17.

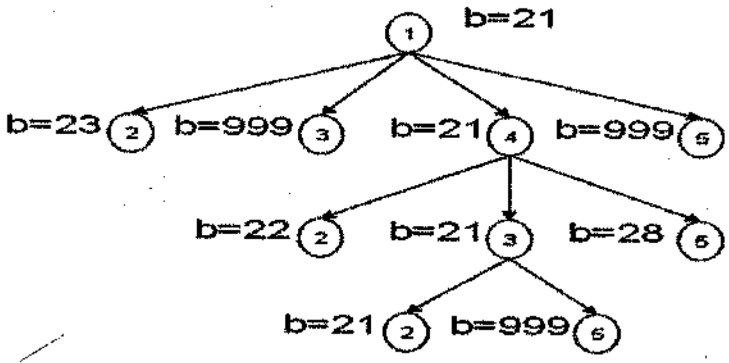


Figure 10.17: Exploration Tree using LCBB of example 10.15(a).

As LCBB algorithm provides exact solution, but time required is uncertain. An approximation algorithm with complexity O(V^2) and worst case cost can be double the minimum value is given below. The algorithm is very simple and based on Prim Minimum Spanning Tree (MST) algorithm.

Algorithm Approximation-Algo-for-TSP

Inputs Graph **G** as Reduced cost matrix and Least Cost value

Output sub-optimal cost Tour

1. select a vertex rV to be the root vertex
2. compute Minimum Spanning Tree T for G from root r using MST-PRIM for root r
3. Let L be the list of vertices visited in a preorder tree walk of T
4. Return the Hamiltonian cycle H that visits the vertices in the order L

Algorithm 10.6: Approximation Algorithm for TSP.

The above approximation algorithm is useful for quick solutions.

10.5 Assignment Problem

The **assignment problem** is one of fundamental combinatorial optimization problems. In its most general form, the problem is as follows:

There are a number of agents and a number of tasks. Any agent can be assigned to perform any task, incurring some cost that may vary depending on the assignment. It is required to perform all tasks by assigning exactly one agent to each task in such a way that the total cost of the assignment is minimized.

A typical assignment problem is shown in Figure 10.18. Here there are five machines (the agents) to be assigned to five jobs (the tasks). The numbers in the matrix indicate the cost of doing each job with each machine. Jobs with costs of M are disallowed assignments. The problem is to find the minimum cost matching of machines to jobs.

	J1	J2	J3	J4	J5
M1	M	8	6	12	1
M2	15	12	7	M	10
M3	10	M	5	14	M
M4	12	M	12	16	15
M5	18	17	14	M	13

Figure 10.18: Matrix model of the assignment problem.

The network model is shown in Figure 10.19. Arc cost of assignment model is not shown in the figure for clarity. The assignment network also has the bipartite structure.

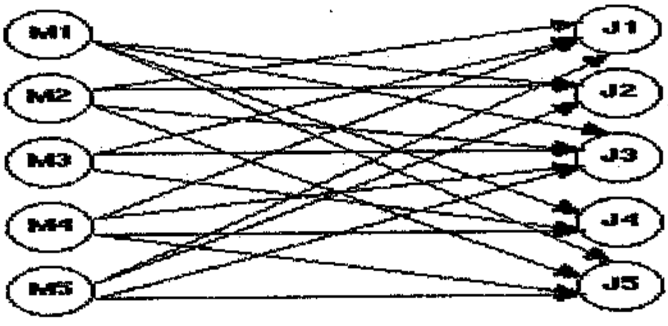


Figure 10.19: Network model of the assignment problem

The solution to the assignment problem is shown in Figure 10.20. The element value 1 means the Machine number shown at row is assigned to corresponding Job number shown at column name. This assignment will be of minimum total cost.

	J1	J2	J3	J4	J5
M1	0	0	0	0	1
M2	0	0	1	0	0
M3	0	0	0	1	0
M4	1	0	0	0	0
M5	0	1	0	0	0

Figure 10.20: Solution to the assignment Problem

Mathematical Formulation

Assignment problem can be expressed in the form of mathematical model. Let us denote p_{ij} as the cost paid by i^{th} job to j^{th} machine, where there are n jobs and n machines. Also x_{ij} be the assignment of i^{th} job to j^{th} machine, where x_{ij} can have value 0 or 1 for whether assigned or not.

Mathematically the assignment problem can be formulated as:

Minimize
$$\sum_{i=1}^{m}\sum_{j=1}^{n} p_{ij}x_{ij}.$$

Subject to

$$\sum_{j=1}^{n} x_{ij} = 1, \qquad i = 1,\ldots,n$$

$$\sum_{i=1}^{n} x_{ij} = 1, \qquad j = 1,\ldots,n$$

$$x_{ij} \in \{0,1\} \qquad i = 1,\ldots,m, \quad j = 1,\ldots,n;$$

The assignment problem is NP-hard, and it is even APX-hard to approximation. Hence, optimal solution is very difficult to obtain in polynomial time. To obtain a good solution, Hungarian Algorithm is commonly used and explained below.

Hungarian algorithm

Hungarian algorithm provides a good solution to assignment problem. Algorithm is very simple and is based on simple concept. Normally assignment problem contains same number of machines and jobs or workers and jobs are given. However, if less number of jobs is there then dummy jobs are added to make jobs and workers or machines equal with 0 costs. Algorithm assumes that both of them are equal. Also matrix representation is assumed as given in Figure 10.18.

```
Algorithm Hungarian
Inputs Graph G with Assignment cost   — Assumed that row and columns are equal
Output Assignment cost for a good assignment
1.    For each row, subtract the minimum number in that row for all numbers in that row;
2.    For each column, subtract the minimum number in that column from all numbers in that
column;   — similar to LC in TSP
3.    Draw the minimum number of lines to cover all zeroes;
4.    If number is the same as number of rows in matrix,
5.    assignment is done and stop;
6.    else Determine the minimum uncovered number say d;
(A)        Subtract d from uncovered numbers
(B)        Add d to numbers covered by two lines
(C)        Numbers covered by one line remain the same
(D)        Go To Line 3
```

7. Find a row or column with only one unlined zero and circle it. (If all rows and columns have two or more unlined zeroes, choose arbitrary zero);

8. If circle is in a row with one zero, draw a line through its column. If the circle is in a column with one zero, draw a line through its row. One approach, when all rows and columns have two or more zeroes, is to draw a line through one with most zeroes, breaking tie arbitrary;

9. Repeat Line 8 until all circles are lined. If this minimum number of lines equals the number of rows, the assignment is over.

Algorithm 10.6: Hungarian Algorithm for Assignment Problem.

The above steps are explained through example as below:

In a worker and job problem, costs are given in the following table.

Worker	Job1 Cost	Job2 Cost	Job3 Cost
Raj Mohan	50	36	16
Darpan	28	30	18
Pushpesh	35	32	20
Hatim	25	25	14

The above table is modified for Hungarian Algorithm as follows:

Worker	Job1 Cost	Job2 Cost	Job3 Cost	Dummy
Raj Mohan	50	36	16	0
Darpan	28	30	18	0
Pushpesh	35	32	20	0
Hatim	25	25	14	0

Minimum numbers in each row have been identified and it comes in last row as done in step-1. Now, algorithm starts and subtracts each row and get Matrix as

$$\begin{matrix} 25 & 11 & 2 & 0 \\ 3 & 5 & 4 & 0 \\ 10 & 7 & 6 & 0 \\ 0 & 0 & 0 & 0 \end{matrix} \quad \Longrightarrow \quad \begin{matrix} 23 & 9 & 0 & 0 \\ 1 & 3 & 2 & 0 \\ 8 & 5 & 4 & 0 \\ 0 & 0 & 0 & 2 \end{matrix}$$

$$\begin{matrix} 23 & 9 & 0 & 0 \\ 0 & 2 & 1 & 0 \\ 7 & 4 & 3 & 0 \\ 0 & 0 & 0 & 0 \end{matrix} \quad \Longleftarrow \quad \begin{matrix} 23 & 9 & 0 & 1 \\ 0 & 2 & 1 & 0 \\ 7 & 4 & 3 & 0 \\ 0 & 0 & 0 & 2 \end{matrix}$$

Now, each line has one zero atleast, Assignment will be for every zero entry i.e. Darpan → Job1 → 28, Raj Mohan → Job3 →16, Hatim →Job2→25 and total cost will be 69.

Self Learning Exercises

Q.No.1 In Topological sort, unique schedule will be available if

(a) Hamiltonian Cycle is available in given DAG

(b) More than two zero indegree vertices available in given DAG

(c) Two or more disconnected vertices are available in given DAG

(d) None of the above

Q.No.2 In curriculum planning corequisite courses can be

(a)Scheduled in different semester

(b)Vertices corresponding to them can be merged with all edges

(c)Scheduled only before current semester

(d)None of the above

Q.No.3 In a Network Flow problem, if augmenting path contains 4 edges having 13/22, 20/24, 5/6 and 8/13 written in its residual graph. What will be the amount of flow allowed along this path?

(a) 9 (b) 5 (c) 1 (d) 2

Q.No.4 In LCBB algorithm for TSP, if a solution with 28 cost is available. In another exploration, cost 19 was computed and next unexplored edge is costing 9. Should it be

(a)Pruned

(b)Treated as a solution

(c)Taking this edge in path, exploration may be continued

(d)None of the above

Q.No.5 In the assignment problem solved above, Hatim is assigned Job 2 even though, he was available at minimum cost for all jobs because

(a)Job 1 and Job 2 cost was same, either one could have been assigned

(b)Job 1 and Job2 cost was maximum for Hatim compared to remaining Jobs

(c)Job 1 and job 3 were minimum for Darpan and Raj Mohan and they could not get Job 2

(d)None of the above

10.6 Summary

In this unit, we have studied various applications of graph theory and their standard algorithms to deal with these problems. The algorithms are explained through examples also. Sub-section wise summary is as follows:

Topological sort and its usefulness are discussed in first sub-section. There are a number of applications are found for this sort. The applications can be mapped as a Directed Acyclic Graph and the vertices are ordered in such a way that no vertex is dependent on its predecessor vertex. Two classical problems namely, Cloth-bearing by an Absent-Minded professor and Curriculum planning for MCA course are explained.

Network Flow is an important application of graphs. Various types of commodities like electricity, water, gas etc. flows between various places. The graph theory based approach well models and solves these types of problems. Min-Cut Max-Flow theorem provides basic proof for maximum flow from source node to destination. Ford-Fulkerson and Edmond-Karp algorithms well solve these problems.

Travelling Salesman Problem (TSP) is one of the classic problems, which can well be modelled as a graph. A salesman wishes to travel all the cities to sell his products with minimum cost of travelling. The problem is of NP-Hard nature (its decision version is NP-Complete) and hence can be solved by Branch and Bound like algorithm or through approximation algorithm or heuristics. Least Cost Branch and Bound (LCBB) algorithm and one simple approximation is described with example.

Assignment Problem is also one of the classic NP-hard problems. In this problem, a number of machines with same number of jobs are to be assigned. The cost of processing any job is different for different machine. Minimum cost solution is desired in its solution. Mathematical modelling and Hungarian Algorithm are described in last sub-section.

10.7 Glossary

topological sort: In graph theory, a topological sort or topological ordering of a directed acyclic graph (DAG) is a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges. Every DAG has one or more topological sorts.

network flow problem:In graph theory, a flow network is a directed graph where each edge has a capacity and each edge receives a flow. The amount of flow on an edge cannot exceed the capacity of the edge. Often in Operations Research, a directed graph is called a network, the vertices are called nodes and the edges are called arcs. A flow must satisfy the restriction that the amount of flow into a node equals the amount of flow out of it, except when it is a source, which has more outgoing flow, or sink, which has more incoming flow. A network can be used to model traffic in a road system, fluids in pipes, currents in an electrical circuit, or anything similar in which something travels through a network of nodes.

10.8 Further Readings

1. Graph theory with applications to Engineering and computer science by Narsingh Deo.
2. Discrete Mathematics by Rosen.
3. Graph theory by John Clark and Derek Allan Holton
4. Reference Book Allan Tucker for Combinatorics

10.9 Answers to self learning exercises

1(a) 2(b) 3(c) 4(a) 5(c)

10.10 Unit End Questions

1. Give an example in which worst case time complexity of Ford-Fulkerson algorithm can actually be demonstrated.

2. Describe Max-Flow Min-Cut theorem.

3. A TSP matrix is described as below. Compute minimum cost solution.

999	39	81	108	3	16	28
12	999	199	105	16	22	18
38	67	999	52	111	17	999
48	999	87	999	56	999	999
31	87	999	999	999	49	67
999	999	78	18	88	999	15
17	23	45	999	999	10	999

4. Describe Mathematical Formulation of Assignment Problem.

5. For the following activity dependence, draw a graph and perform topological sort.

S.No.	Activity No.	Dependent on Activity No
1	1	6
2	2	-
3	3	4, 7
4	4	-
5	5	3, 1
6	6	8
7	7	-
8	8	2
9	9	10
10	10	-

---***---

UNIT XI

SORTING ALGORITHIMS

STRUCTURE OF THE UNIT

11.0 Objectives

11.1 Introduction to Sorting algorithms and complexities

11.2 Bubble Sort

11.3 Selection Sort

11.4 Insertion Sort

11.5 Heap sorts

11.6 Quick Sort

11.7 Merge sort

11.8 Bucket sort

11.9 Radix Sort

11.10 Summary

11.11 Glossary

11.12 Further Readings

11.13 Answers to the Self learning exercises

1.14 Unit -End Questions

11.0 Objectives

After completing this unit you will be able to explain

- What is sorting algorithms, how different kind of sorting can be perform.
- Sorting method in Bubble sort, selection sort, insertion sort, quick sort, radix sort, merge sort, bucket sort, heap sort etc.

11.1 Introduction

One of the fundamental problems of computer science is ordering a list of items. There's a excess of solutions to this problem, known as sorting algorithms. Some sorting algorithms are simple and intuitive, such as the bubble sort. Others, such as the quick sort are extremely complicated, but produce lightning-fast results.

Below are links to algorithms, analysis, and source code for seven of the most common sorting algorithms.

- Sorting Algorithms
- Bubble sort
- Selection sort
- Insertion sort
- Heap sort

- Quick sort
- Merge sort
- Shell sort

The common sorting algorithms can be divided into two classes by the complexity of their algorithms. Algorithmic complexity is a complex subject that would take too much time to explain here, but suffice it to say that there's a direct correlation between the complexity of an algorithm and its relative efficiency. Algorithmic complexity is generally written in a form known as Big-O notation, where the O represents the complexity of the algorithm and a value n represents the size of the set the algorithm is run against.

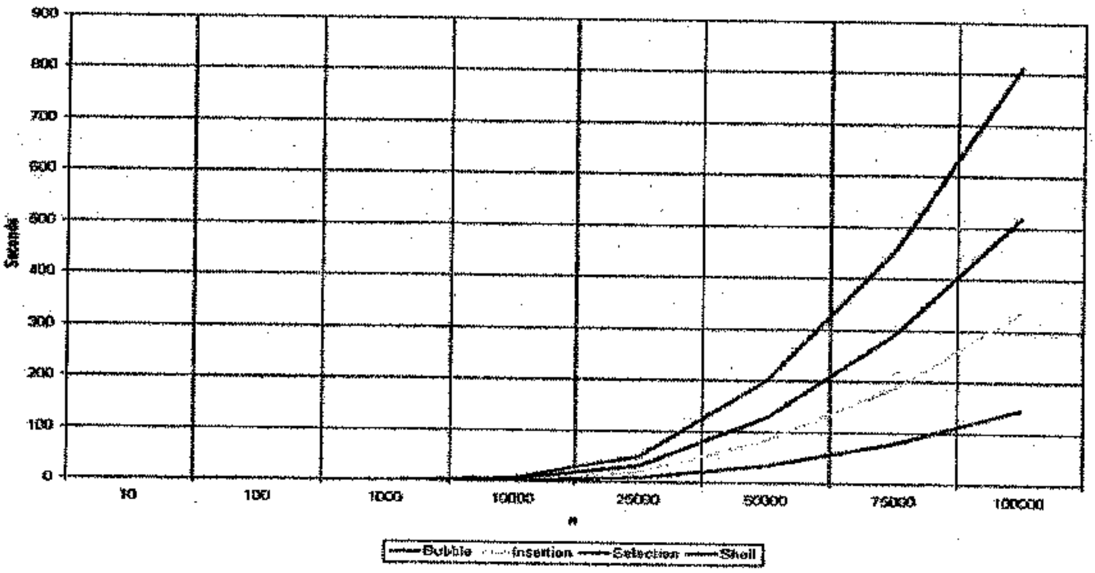
For example, O(n) means that an algorithm has a linear complexity. In other words, it takes ten times longer to operate on a set of 100 items than it does on a set of 10 items (10 * 10 = 100). If the complexity was O(n^2) (quadratic complexity), then it would take 100 times longer to operate on a set of 100 items than it does on a set of 10 items.

The two classes of sorting algorithms are O(n2), which includes the bubble, insertion, selection, and shell sorts; and O(n log n) which includes the heap, merge, and quick sorts.

In addition to algorithmic complexity, the speed of the various sorts can be compared with empirical data. Since the speed of a sort can vary greatly depending on what data set it sorts, accurate empirical results require several runs of the sort be made and the results averaged together. The empirical data on this site is the average of a hundred runs against random data sets on a single-user 250MHz UltraSPARC II. The run times on your system will almost certainly vary from these results, but the relative speeds should be the same - the selection sort runs in roughly half the time of the bubble sort on the UltraSPARC II, and it should run in roughly half the time on whatever system you use as well.

These empirical efficiency graphs are kind of like golf - the lowest line is the "best". Keep in mind that "best" depends on your situation - the quick sort may look like the fastest sort, but using it tc sort a list of 20 items is kind of like going after a fly with a sledgehammer.

O(n^2) Sorts



As the graph pretty plainly shows, the bubble sort is grossly inefficient, and the shell sort blows it out of the water. Notice that the first horizontal line in the plot area is 100 seconds - these aren't sorts that you want to use for huge amounts of data in an interactive application. Even using the shell sort, users are going to be twiddling their thumbs if you try to sort much more than 10,000 data items.

On the bright side, all of these algorithms are incredibly simple (with the possible exception of the shell sort). For quick test programs, rapid prototypes, or internal-use software they're not bad choices unless you really think you need split-second efficiency. O(n log n) Sorts

Speaking of split-second efficiency, the O(n log n) sorts are where it's at. Notice that the time on this graph is measured in tenths of seconds, instead hundreds of seconds like the O(n^2) graph.

But as with everything else in the real world, there are trade-offs. These algorithms are blazingly fast, but that speed comes at the cost of complexity. Recursion, advanced data structures, multiple arrays - these algorithms make extensive use of those nasty things.

In the end, the important thing is to pick the sorting algorithm that you think is appropriate for the task at hand. You should be able to use the source code on this site as a "black box" if you need to - you can just use it, without understanding how it works. Obviously taking the time to understand how the algorithm you choose works is preferable, but time constraints are a fact of life.

11.2 Bubble Sort

Algorithm Analysis

The bubble sort is the oldest and simplest sort in use. Unfortunately, it's also the slowest.

The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order). This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list.

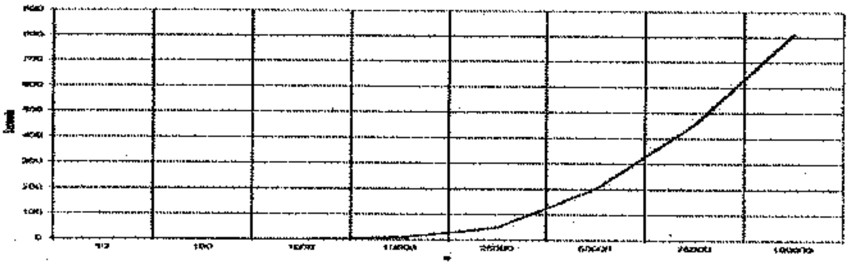
The bubble sort is generally considered to be the most inefficient sorting algorithm in common usage. Under best-case conditions (the list is already sorted), the bubble sort can approach a constant O(n) level of complexity. General-case is an abysmal O(n2). While the insertion, selection, and shell sorts also have O(n2) complexities, they are significantly more efficient than the bubble sort.

Pros: Simplicity and ease of implementation.

Cons: Horribly inefficient

Empirical Analysis

Bubble Sort Efficiency

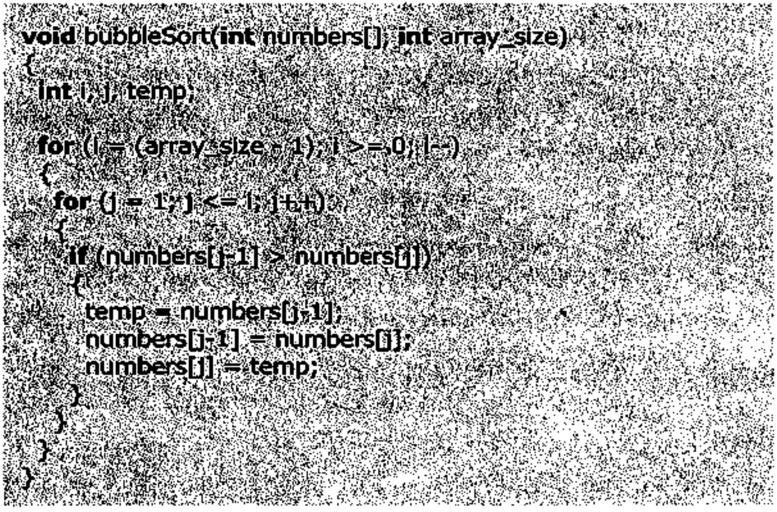


The graph clearly shows the n^2 nature of the bubble sort.

A fair number of algorithm purists (which means they've probably never written software for a living) claim that the bubble sort should never be used for any reason. Realistically, there isn't a noticeable performance difference between the various sorts for 100 items or less, and the simplicity of the bubble sort makes it attractive. The bubble sort shouldn't be used for repetitive sorts or sorts of more than a couple hundred items.

Source Code

Below is the basic bubble sort algorithm.



```
void bubbleSort(int numbers[], int array_size)
{
  int i, j, temp;

  for (i = (array_size - 1); i >= 0; i--)
  {
    for (j = 1; j <= i; j++)
    {
      if (numbers[j-1] > numbers[j])
      {
        temp = numbers[j-1];
        numbers[j-1] = numbers[j];
        numbers[j] = temp;
      }
    }
  }
}
```

Program:

```
#include <stdlib.h>
#include <stdio.h>
#define NUM_ITEMS 1000
void bubbleSort(int numbers[], int array_size);
int numbers[NUM_ITEMS];
int main()
{
 int i;
 //seed random number generator
 srand(getpid());
 //fill array with random integers
 for (i = 0; i < NUM_ITEMS; i++)
  numbers[i] = rand();
 //perform bubble sort on array
 bubbleSort(numbers, NUM_ITEMS);
 printf("Done with sort.\n");
 for (i = 0; i < NUM_ITEMS; i++)
```

```
  printf("%i\n", numbers[i]);
}
void bubbleSort(int numbers[], int array_size)
{
 int i, j, temp;
 for (i = (array_size - 1); i >= 0; i—)
  {
   for (j = 1; j <= i; j++)
    {
    if (numbers[j-1] > numbers[j])
     {
     temp = numbers[j-1];
     numbers[j-1] = numbers[j];
     numbers[j] = temp;
     }  }
 } }
```

11.3 Selection Sort

Algorithm Analysis

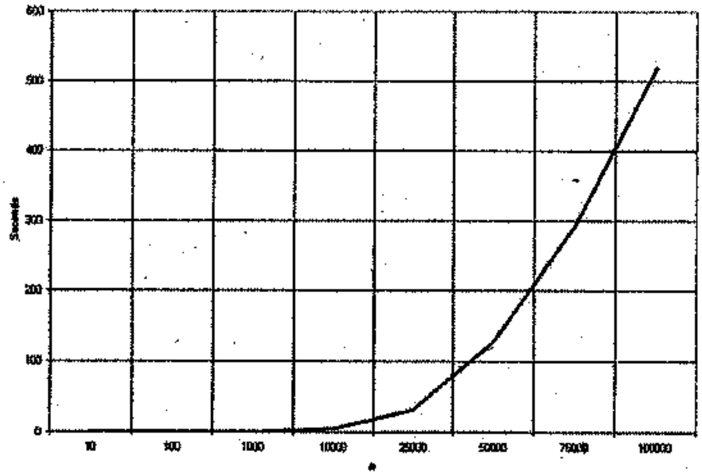
The selection sort works by selecting the smallest unsorted item remaining in the list, and then swapping it with the item in the next position to be filled. The selection sort has a complexity of O(n^2).

Pros: Simple and easy to implement.

Cons: Inefficient for large lists, so similar to the more efficient insertion sort that the insertion sort should be used in its place.

Empirical Analysis

Selection Sort Efficiency

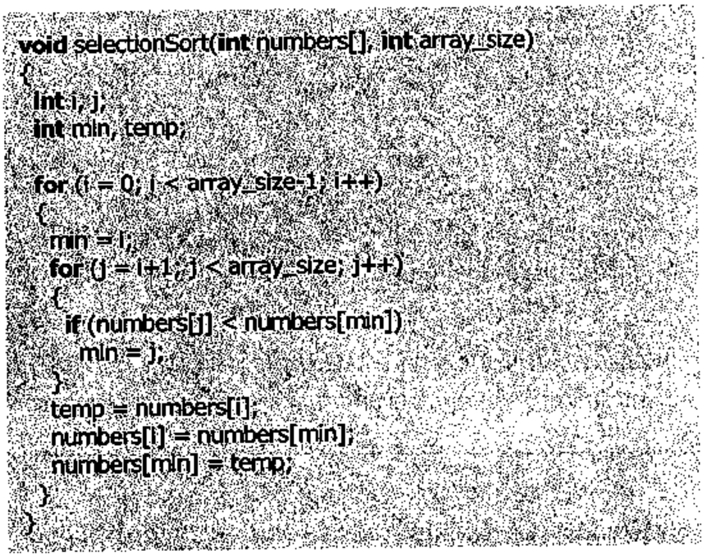


The selection sort is the unwanted stepchild of the n^2 sorts. It yields a 60% performance improvement over the bubble sort, but the insertion sort is over twice as fast as the bubble sort and is just as easy to implement as the selection sort. In short, there really isn't any reason to use the selection sort - use the insertion sort instead.

If you really want to use the selection sort for some reason, try to avoid sorting lists of more than a 1000 items with it or repetitively sorting lists of more than a couple hundred items.

Source Code

Below is the basic selection sort algorithm.



```
void selectionSort(int numbers[], int array_size)
{
 int i, j;
 int min, temp;

 for (i = 0; i < array_size-1; i++)
 {
  min = i;
  for (j = i+1; j < array_size; j++)
  {
   if (numbers[j] < numbers[min])
     min = j;
  }
  temp = numbers[i];
  numbers[i] = numbers[min];
  numbers[min] = temp;
 }
}
```

A sample C program that demonstrates the use of the selection sort may

Program:

```
#include <stdlib.h>
#include <stdio.h>
#define NUM_ITEMS 100
void selectionSort(int numbers[], int array_size);
int numbers[NUM_ITEMS];
int main()
{
 int i;
 //seed random number generator
 srand(getpid());
 //fill array with random integers
 for (i = 0; i < NUM_ITEMS; i++)
  numbers[i] = rand();
  //perform selection sort on array
  selectionSort(numbers, NUM_ITEMS);
  printf("Done with sort.\n");
  for (i = 0; i < NUM_ITEMS; i++)
   printf("%i\n", numbers[i]);
 }
 void selectionSort(int numbers[], int array_size)
 {
  int i, j;
  int min, temp;
  for (i = 0; i < array_size-1; i++)
  {
   min = i;
   for (j = i+1; j < array_size; j++)
    {
    if (numbers[j] < numbers[min])
      min = j;
    }
   temp = numbers[i];
  numbers[i] = numbers[min];
  numbers[min] = temp;
  }
}
```

11.4 Insertion Sort

Algorithm Analysis

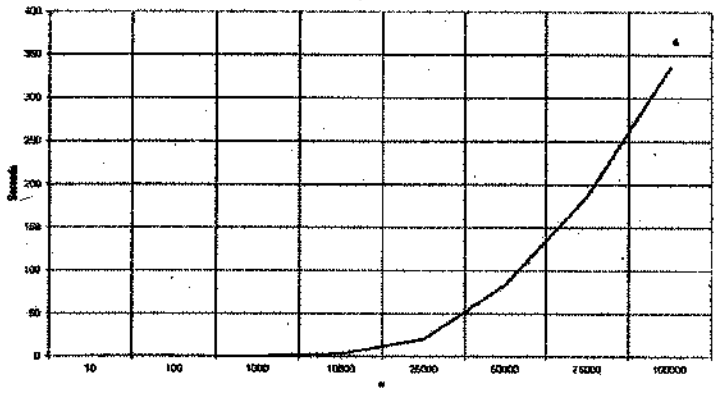
The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place.

Like the bubble sort, the insertion sort has a complexity of O(n2). Although it has the same complexity, the insertion sort is a little over twice as efficient as the bubble sort.

Pros: Relatively simple and easy to implement.
Cons: Inefficient for large lists.

Empirical Analysis

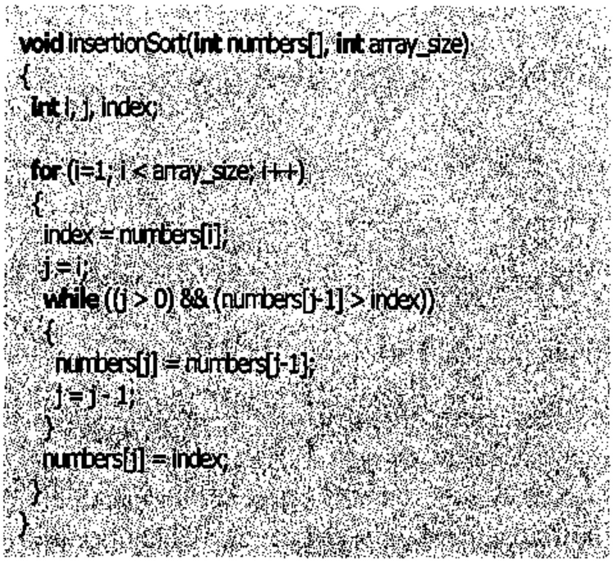
Insertion Sort Efficiency



The insertion sort is a good middle-of-the-road choice for sorting lists of a few thousand items or less. The algorithm is significantly simpler than the shell sort, with only a small trade-off in efficiency. At the same time, the insertion sort is over twice as fast as the bubble sort and almost 40% faster than the selection sort. The insertion sort shouldn't be used for sorting lists larger than a couple thousand items or repetitive sorting of lists larger than a couple hundred items.

Source Code

Below is the basic insertion sort algorithm.



```
void insertionSort(int numbers[], int array_size)
{
 int i, j, index;

 for (i=1; i < array_size; i++)
 {
  index = numbers[i];
  j = i;
  while ((j > 0) && (numbers[j-1] > index))
  {
   numbers[j] = numbers[j-1];
   j = j - 1;
  }
  numbers[j] = index;
 }
}
```

Program:

```
#include <stdlib.h>
#include <stdio.h>
#define NUM_ITEMS 100
void insertionSort(int numbers[], int array_size);
```

```
int numbers[NUM_ITEMS];
int main()
{
 int i;
 //seed random number generator
 srand(getpid());
 //fill array with random integers
 for (i = 0; i < NUM_ITEMS; i++)
  numbers[i] = rand();
 //perform insertion sort on array
 insertionSort(numbers, NUM_ITEMS);
 printf(“Done with sort.\n”);
 for (i = 0; i < NUM_ITEMS; i++)
  printf(“%i\n”, numbers[i]);
}
void insertionSort(int numbers[], int array_size)
{
 int i, j, index;
 for (i=1; i < array_size; i++)
  {
  index = numbers[i];
  j = i;
  while ((j > 0) && (numbers[j-1] > index))
   {
   numbers[j] = numbers[j-1];
    j = j - 1;
   }
  numbers[j] = index;
  }
}
```

11.5 Heap Sort

Algorithm Analysis

The heap sort is the slowest of the O(n log n) sorting algorithms, but unlike the merge and quick sorts it doesn’t require massive recursion or multiple arrays to work. This makes it the most attractive option for very large data sets of millions of items.

The heap sort works as it name suggests - it begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the sorted array. After removing the largest item, it reconstructs the heap and removes the largest remaining item and places it in the next open position from the end of the sorted array. This is repeated until there are no items left in the heap and the sorted array is full. Elementary implementations require two arrays - one to hold the heap and the other to hold the sorted elements.

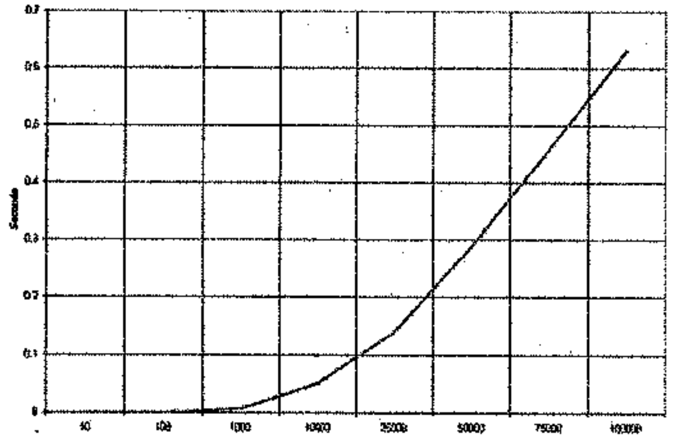
To do an in-place sort and save the space the second array would require, the algorithm below "cheats" by using the same array to store both the heap and the sorted array. Whenever an item is removed from the heap, it frees up a space at the end of the array that the removed item can be placed in.

Pros: In-place and non-recursive, making it a good choice for extremely large data sets.

Cons: Slower than the merge and quick sorts.

Empirical Analysis

Heap Sort Efficiency

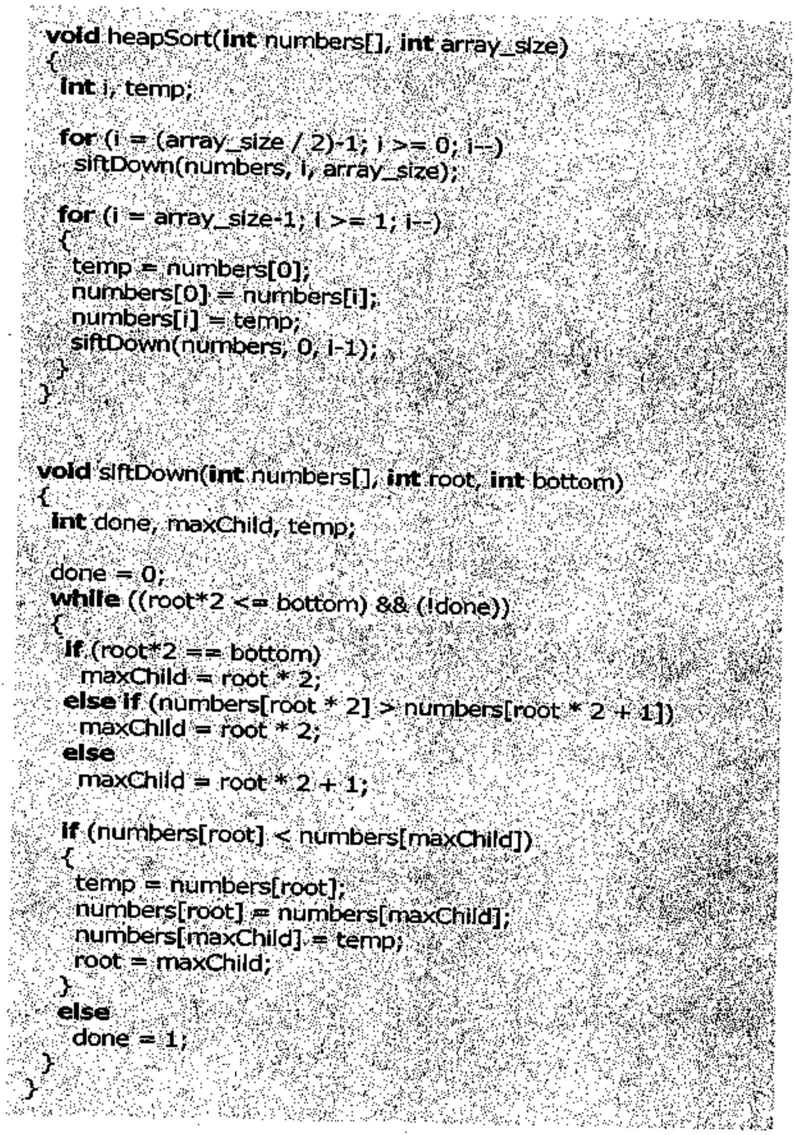


As mentioned above, the heap sort is slower than the merge and quick sorts but doesn't use multiple arrays or massive recursion like they do. This makes it a good choice for really large sets, but most modern computers have enough memory and processing power to handle the faster sorts unless over a million items are being sorted.

The "million item rule" is just a rule of thumb for common applications - high-end servers and workstations can probably safely handle sorting tens of millions of items with the quick or merge sorts. But if you're working on a common user-level application, there's always going to be some yahoo who tries to run it on junk machine older than the programmer who wrote it, so better safe than sorry.

Source Code

Below is the basic heap sort algorithm. The siftDown() function builds and reconstructs the heap.



```
void heapSort(int numbers[], int array_size)
{
  int i, temp;

  for (i = (array_size / 2)-1; i >= 0; i--)
   siftDown(numbers, i, array_size);

  for (i = array_size-1; i >= 1; i--)
  {
   temp = numbers[0];
   numbers[0] = numbers[i];
   numbers[i] = temp;
   siftDown(numbers, 0, i-1);
  }
}


void siftDown(int numbers[], int root, int bottom)
{
  int done, maxChild, temp;

  done = 0;
  while ((root*2 <= bottom) && (!done))
  {
   if (root*2 == bottom)
    maxChild = root * 2;
   else if (numbers[root * 2] > numbers[root * 2 + 1])
    maxChild = root * 2;
   else
    maxChild = root * 2 + 1;

   if (numbers[root] < numbers[maxChild])
   {
    temp = numbers[root];
    numbers[root] = numbers[maxChild];
    numbers[maxChild] = temp;
    root = maxChild;
   }
   else
    done = 1;
  }
}
```

Program :

```
#include <stdlib.h>
#include <stdio.h>
#define NUM_ITEMS 100
void heapSort(int numbers[], int array_size);
void siftDown(int numbers[], int root, int bottom);
int numbers[NUM_ITEMS];
int main()
{
```

```
int i;
//seed random number generator
srand(getpid());
//fill array with random integers
for (i = 0; i < NUM_ITEMS; i++)
 numbers[i] = rand();
//perform heap sort on array
heapSort(numbers, NUM_ITEMS);
printf(“Done with sort.\n”);
for (i = 0; i < NUM_ITEMS; i++)
 printf(“%i\n”, numbers[i]);
}
void heapSort(int numbers[], int array_size)
{
 int i, temp;
 for (i = (array_size / 2)-1; i >= 0; i—)
  siftDown(numbers, i, array_size);
 for (i = array_size-1; i >= 1; i—)
 {
  temp = numbers[0];
  numbers[0] = numbers[i];
  numbers[i] = temp;
  siftDown(numbers, 0, i-1);
 }
}
void siftDown(int numbers[], int root, int bottom)
{
 int done, maxChild, temp;
 done = 0;
 while ((root*2 <= bottom) && (!done))
 {
  if (root*2 == bottom)
   maxChild = root * 2;
  else if (numbers[root * 2] > numbers[root * 2 + 1])
   maxChild = root * 2;
  else
```

```
   maxChild = root * 2 + 1;
  if(numbers[root] < numbers[maxChild])
   {
   temp = numbers[root];
   numbers[root] = numbers[maxChild];
   numbers[maxChild] = temp;
   root = maxChild;
   }
  else
    done = 1;
 }
}
```

11.6 Quick Sort

Algorithm Analysis

The quick sort is an in-place, divide-and-conquer, massively recursive sort. As a normal person would say, it's essentially a faster in-place version of the merge sort. The quick sort algorithm is simple in theory, but very difficult to put into code (computer scientists tied themselves into knots for years trying to write a practical implementation of the algorithm, and it still has that effect on university students).

The recursive algorithm consists of four steps (which closely resemble the merge sort):

1. If there are one or less elements in the array to be sorted, return immediately.

2. Pick an element in the array to serve as a "pivot" point. (Usually the left-most element in the array is used.)

3. Split the array into two parts - one with elements larger than the pivot and the other with elements smaller than the pivot.

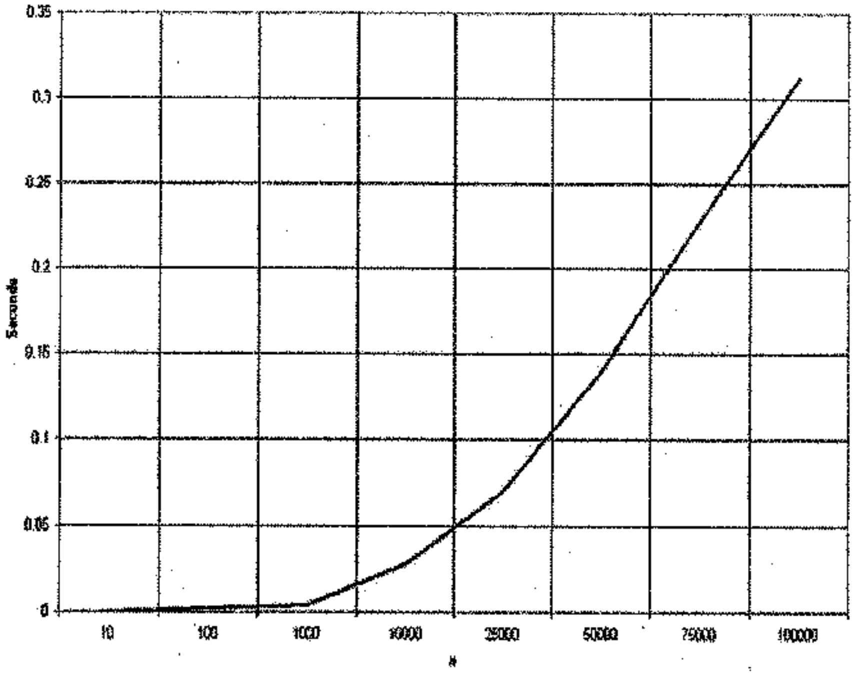
4. Recursively repeat the algorithm for both halves of the original array.

The efficiency of the algorithm is majorly impacted by which element is choosen as the pivot point. The worst-case efficiency of the quick sort, $O(n^2)$, occurs when the list is sorted and the left-most element is chosen. Randomly choosing a pivot point rather than using the left-most element is recommended if the data to be sorted isn't random. As long as the pivot point is chosen randomly, the quick sort has an algorithmic complexity of $O(n \log n)$.

Pros: Extremely fast.
Cons: Very complex algorithm, massively recursive.

Empirical Analysis

Quick Sort Efficiency



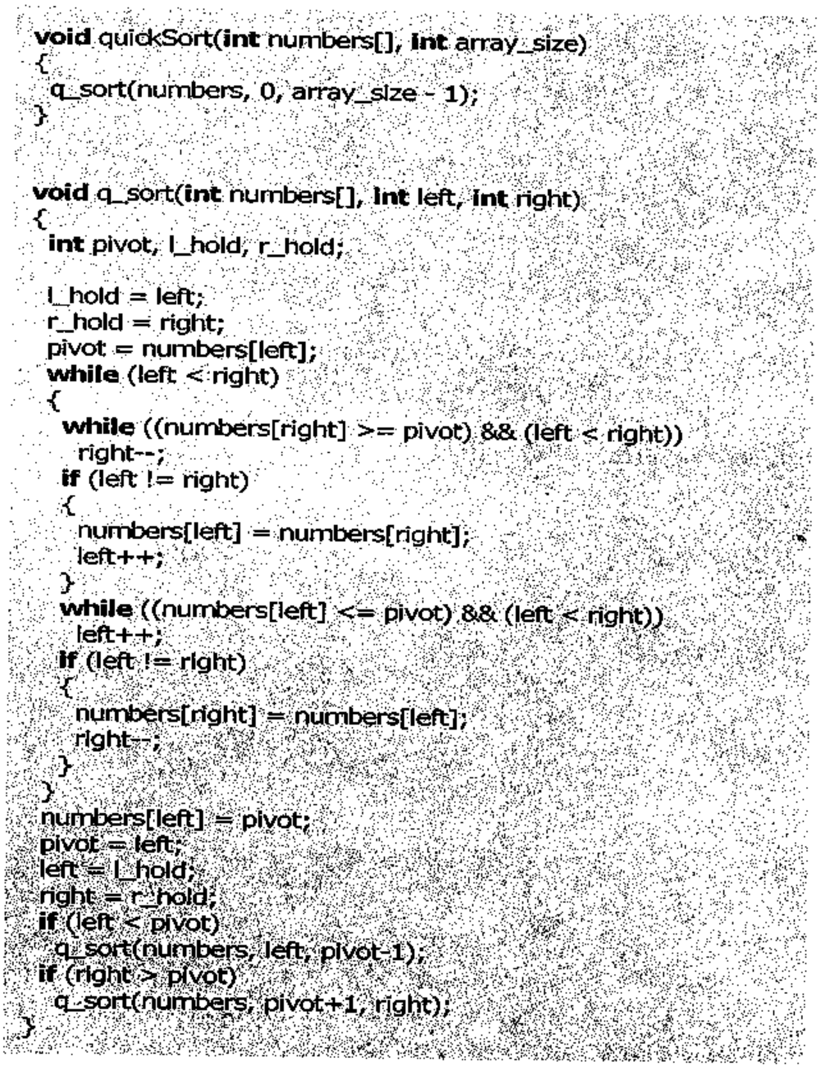
The quick sort is by far the fastest of the common sorting algorithms. It's possible to write a special-purpose sorting algorithm that can beat the quick sort for some data sets, but for general-case sorting there isn't anything faster.

As soon as students figure this out, their immediate implulse is to use the quick sort for everything - after all, faster is better, right? It's important to resist this urge - the quick sort isn't always the best choice. As mentioned earlier, it's massively recursive (which means that for very large sorts, you can run the system out of stack space pretty easily). It's also a complex algorithm - a little too complex to make it practical for a one-time sort of 25 items, for example.

With that said, in most cases the quick sort is the best choice if speed is important (and it almost always is). Use it for repetitive sorting, sorting of medium to large lists, and as a default choice when you're not really sure which sorting algorithm to use. Ironically, the quick sort has horrible efficiency when operating on lists that are mostly sorted in either forward or reverse order - avoid it in those situations.

Source Code

Below is the basic quick sort algorithm



```
void quickSort(int numbers[], int array_size)
{
  q_sort(numbers, 0, array_size - 1);
}


void q_sort(int numbers[], int left, int right)
{
  int pivot, l_hold, r_hold;

  l_hold = left;
  r_hold = right;
  pivot = numbers[left];
  while (left < right)
  {
    while ((numbers[right] >= pivot) && (left < right))
      right--;
    if (left != right)
    {
      numbers[left] = numbers[right];
      left++;
    }
    while ((numbers[left] <= pivot) && (left < right))
      left++;
    if (left != right)
    {
      numbers[right] = numbers[left];
      right--;
    }
  }
  numbers[left] = pivot;
  pivot = left;
  left = l_hold;
  right = r_hold;
  if (left < pivot)
    q_sort(numbers, left, pivot-1);
  if (right > pivot)
    q_sort(numbers, pivot+1, right);
}
```

Program

```
#include <stdlib.h>
#include <stdio.h>
#define NUM_ITEMS 100
void quickSort(int numbers[], int array_size);
void q_sort(int numbers[], int left, int right);
int numbers[NUM_ITEMS];
int main()
{
 int i;
 //seed random number generator
 srand(getpid());
 //fill array with random integers
 for (i = 0; i < NUM_ITEMS; i++)
  numbers[i] = rand();
 //perform quick sort on array
 quickSort(numbers, NUM_ITEMS);
 printf("Done with sort.\n");
 for (i = 0; i < NUM_ITEMS; i++)
  printf("%i\n", numbers[i]);
}
void quickSort(int numbers[], int array_size)
{
 q_sort(numbers, 0, array_size - 1);
}
void q_sort(int numbers[], int left, int right)
{
 int pivot, l_hold, r_hold;
 l_hold = left;
 r_hold = right;
 pivot = numbers[left];
 while (left < right)
  {
  while ((numbers[right] >= pivot) && (left < right))
    right—;
  if (left != right)
   {
   numbers[left] = numbers[right];
    left++;
   }
  while ((numbers[left] <= pivot) && (left < right))
    left++;
  if (left != right)
{
  numbers[right] = numbers[left];
   right—;
   }
}
numbers[left] = pivot;
pivot = left;
left = l_hold;
right = r_hold;
if (left < pivot)
  q_sort(numbers, left, pivot-1);
if (right > pivot)
  q_sort(numbers, pivot+1, right);
}
```

11.7 Merge Sort

Algorithm Analysis

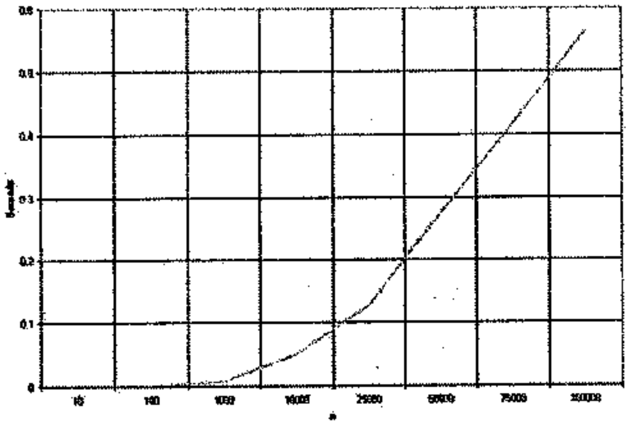
The merge sort splits the list to be sorted into two equal halves, and places them in separate arrays. Each array is recursively sorted, and then merged back together to form the final sorted list. Like most recursive sorts, the merge sort has an algorithmic complexity of $O(n \log n)$.

Elementary implementations of the merge sort make use of three arrays - one for each half of the data set and one to store the sorted list in. The below algorithm merges the arrays in-place, so only two arrays are required. There are non-recursive versions of the merge sort, but they don't yield any significant performance enhancement over the recursive algorithm on most machines.

Pros: Marginally faster than the heap sort for larger sets.
Cons: At least twice the memory requirements of the other sorts; recursive.

Empirical Analysis

Merge Sort Efficiency

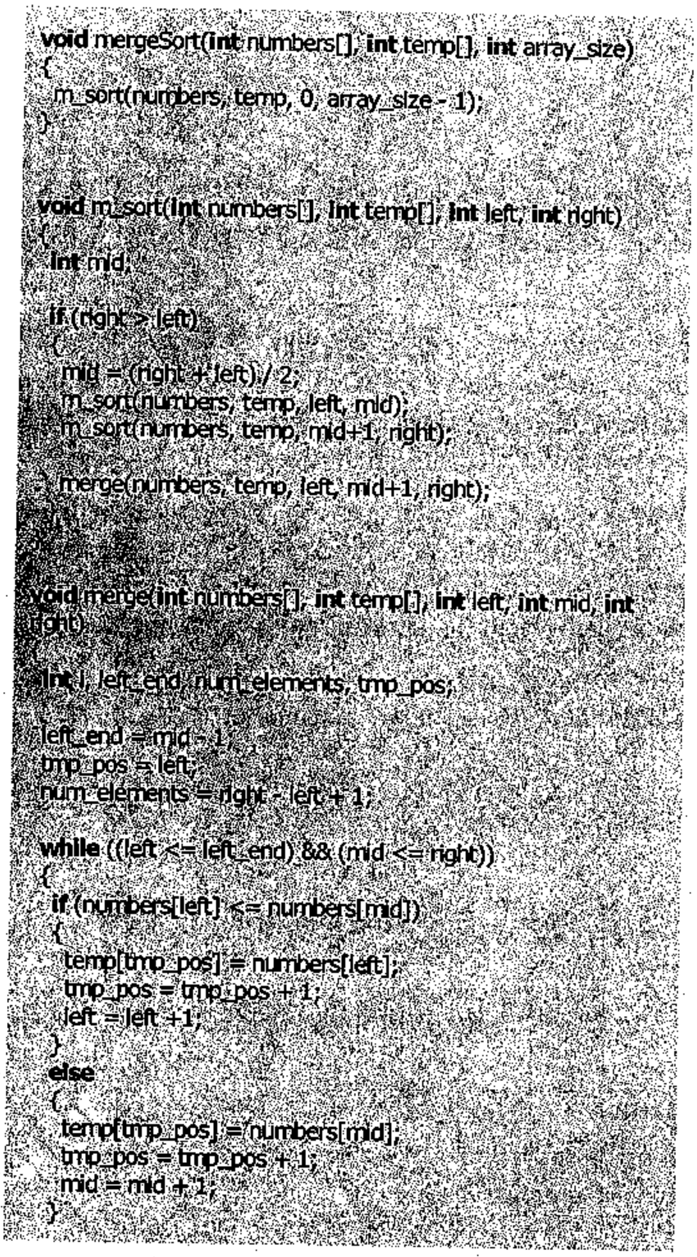


The merge sort is slightly faster than the heap sort for larger sets, but it requires twice the memory of the heap sort because of the second array. This additional memory requirement makes it unattractive for most purposes - the quick sort is a better choice most of the time and the heap sort is a better choice for very large sets.

Like the quick sort, the merge sort is recursive which can make it a bad choice for applications that run on machines with limited memory.

Source Code

Below is the basic merge sort algorithm.



```
void mergeSort(int numbers[], int temp[], int array_size)
{
  m_sort(numbers, temp, 0, array_size - 1);
}


void m_sort(int numbers[], int temp[], int left, int right)
{
  int mid;

  if (right > left)
  {
    mid = (right + left) / 2;
    m_sort(numbers, temp, left, mid);
    m_sort(numbers, temp, mid+1, right);

    merge(numbers, temp, left, mid+1, right);
  }
}


void merge(int numbers[], int temp[], int left, int mid, int
right)
{
  int i, left_end, num_elements, tmp_pos;

  left_end = mid - 1;
  tmp_pos = left;
  num_elements = right - left + 1;

  while ((left <= left_end) && (mid <= right))
  {
    if (numbers[left] <= numbers[mid])
    {
      temp[tmp_pos] = numbers[left];
      tmp_pos = tmp_pos + 1;
      left = left +1;
    }
    else
    {
      temp[tmp_pos] = numbers[mid];
      tmp_pos = tmp_pos + 1;
      mid = mid + 1;
    }
```

Program

include <stdlib.h>

#include<stdio.h>

#define NUM_ITEMS 100

```
  void mergeSort(int numbers[], int temp[], int array_size);
  void m_sort(int numbers[], int temp[], int left, int right);
  void merge(int numbers[], int temp[], int left, int mid, int right);
  int numbers[NUM_ITEMS];
  int temp[NUM_ITEMS];
  int main()
  {
   int i;
   //seed random number generator
   srand(getpid());
   //fill array with random integers
   for (i = 0; i < NUM_ITEMS; i++)
    numbers[i] = rand();
  //perform merge sort on array
  mergeSort(numbers, temp, NUM_ITEMS);
  printf(“Done with sort.\n”);
  for (i = 0; i < NUM_ITEMS; i++)
   printf(“%i\n”, numbers[i]);
 }
 void mergeSort(int numbers[], int temp[], int array_size)
 {
  m_sort(numbers, temp, 0, array_size - 1);
 }
 void m_sort(int numbers[], int temp[], int left, int right)
 {
  int mid;
  if (right > left)
  {
   mid = (right + left) / 2;
   m_sort(numbers, temp, left, mid);
   m_sort(numbers, temp, mid+1, right);
   merge(numbers, temp, left, mid+1, right);
  }
}
void merge(int numbers[], int temp[], int left, int mid, int right)
{
```

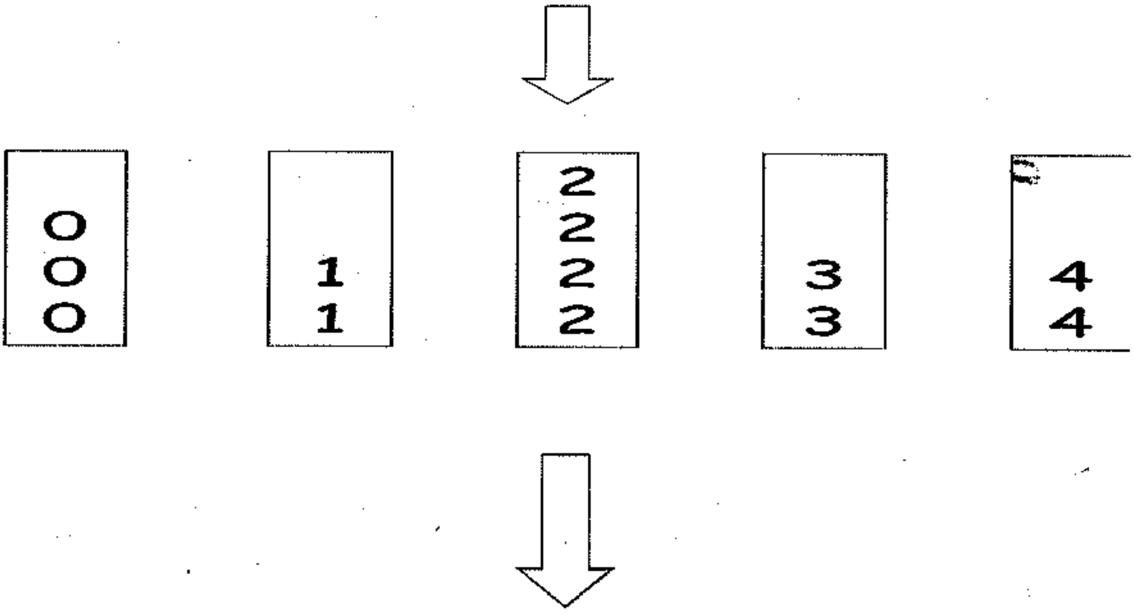
```
int i, left_end, num_elements, tmp_pos;
left_end = mid - 1;
tmp_pos = left;
num_elements = right - left + 1;
while ((left <= left_end) && (mid <= right))
 {
 if (numbers[left] <= numbers[mid])
  {
   temp[tmp_pos] = numbers[left];
    tmp_pos = tmp_pos + 1;
    left = left +1;
  }
  else
  {
   temp[tmp_pos] = numbers[mid];
    tmp_pos = tmp_pos + 1;
    mid = mid + 1;
  }
 }
while (left <= left_end)
{
 temp[tmp_pos] = numbers[left];
 left = left + 1;
 tmp_pos = tmp_pos + 1;
}
while (mid <= right)
{
 temp[tmp_pos] = numbers[mid];
 mid = mid + 1;
 tmp_pos = tmp_pos + 1;
}
for (i=0; i <= num_elements; i++)
{
 numbers[right] = temp[right];
 right = right - 1;
} }
```

11.8 Bucket Sort

Idea: suppose the values are in the range 0..m-1; start with m empty buckets numbered 0 to m-1, scan the list and place element s[i] in bucket s[i], and then output the buckets in order Will need an array of buckets, and the values in the list to be sorted will be the indexes to the buckets no comparisons will be necessary

Example:

4	2	1	2	0	3	2	1	4	0	2	3	0



0	0	0	1	1	2	2	2	2	3	3	4	4

Bucket Sort Algorithm

```
Algorithm BucketSort( S )
( values in S are between 0 and m-1 )
for j ← 0 to m-1 do      // initialize m buckets
        b[j] ← 0
for i ← 0 to n-1 do              // place elements in their
        b[S[i]] ← b[S[i]] + 1    // appropriate buckets
i ← 0
for j ← 0 to m-1 do      // place elements in buckets
        for r ← 1 to b[j] do      // back in S
                S[i] ← j
                i ← i + 1
```

Values versus entries

- If we were sorting values, each bucket is just a counter that we increment whenever a value matching the bucket's number is encountered

- If we were sorting entries according to keys, then each bucket is a queue
- Entries are enqueued into a matching bucket
- Entries will be dequeued back into the array after the scan

Time complexity

- Bucket initialization: O(m)
- From array to buckets: O(n)
- From buckets to array: O(n)
- Even though this stage is a nested loop, notice that all we do is dequeue from each bucket until they are all empty -> n dequeue operations in all Since m will likely be small compared to n, Bucket sort is O(n)
- Strictly speaking, time complexity is O (n + m)

11.9 Radix Sort

Radix sort is one of the linear sorting algorithms for integers. It functions by sorting the input numbers on each digit, for each of the digits in the numbers. However, the process adopted by this sort method is somewhat counterintuitive, in the sense that the numbers are sorted on the least-significant digit first, followed by the second-least significant digit and so on till the most significant digit.

To appreciate Radix Sort, consider the following analogy: Suppose that we wish to sort a deck of 52 playing cards (the different suits can be given suitable values, for example 1 for Diamonds, 2 for Clubs, 3 for Hearts and 4 for Spades). The 'natural' thing to do would be to first sort the cards according to suits, then sort each of the four seperate piles, and finally combine the four in order. This approach, however, has an inherent disadvantage. When each of the piles is being sorted, the other piles have to be kept aside and kept track of. If, instead, we follow the 'counterintuitive' aproach of first sorting the cards by value, this problem is eliminated. After the first step, the four seperate piles are combined in order and then sorted by suit. If a stable sorting algorithm (i.e. one which resolves a tie by keeping the number obtained first in the input as the first in the output) it can be easily seen that correct final results are obtained.

As has been mentioned, the sorting of numbers proceeds by sorting the least significant to most significant digit. For sorting each of these digit groups, a stable sorting algorithm is needed. Also, the elements in this group to be sorted are in the fixed range of 0 to 9. Both of these characteristics point towards the use of Counting Sort as the sorting algorithm of choice for sorting on each digit (If you haven't read the description on Counting Sort already, please do so now).

The time complexity of the algorithm is as follows: Suppose that the n input numbers have maximum k digits. Then the Counting Sort procedure is called a total of k times. Counting Sort is a linear, or O(n) algorithm. So the entire Radix Sort procedure takes O(kn) time. If the numbers are of finite size, the algorithm runs in O(n) asymptotic time.

Example

Here is a simple example of the sort. Suppose the input keys are 34, 12, 42, 32, 44, 41, 34, 11, 32, and 23. Four buckets are appropriate, since there are four different digits. The first pass distributes the keys into buckets by the least significant digit. Half way through the first pass, the buckets contain the following, where each line is a bucket.

12 42 32

34 44

When the first pass is done, we have the following.

41 11

12 42 32 32

23

34 44 34

We collect these, keeping their relative order: 41 11 12 42 32 32 23 34 44 34. Now we distribute by the next most significant digit, which is the highest digit, and we get the following.

11 12

23

32 32 34 34

41 42 44

When we collect them, they are in order: 11 12 23 32 32 34 34 41 42 44.

Self Learning Exercises

Choose the right sorting algorithm

Given are several scenarios where sorting algorithms are applied. Choose an appropriate sorting algorithm and explain why the algorithm was chosen:

1. Given is a list of people sorted by their year of birth. Sort them by their resting pulse rate. (Hint: the higher the age the lower the resting puls rate)

2. Given is a cryptographic algorithm that facilitates a sorting algorithm as part of its implementation. Which algorithms can prevent timing attacks?

3. Given is an alphabetic sorted list of students of the university of Innsbruck. Find an appropriate algorithm for sorting the list by the first two digits (year) of the matriculation number.

11.10 Summary

·The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items

·The selection sort works by selecting the smallest unsorted item remaining in the list, and then swapping it with the item in the next position to be filled. The selection sort has a complexity of $O(n^2)$.

·The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted

·The heap sort is the slowest of the O(n log n) sorting algorithms, but unlike the merge and quick sorts it doesn't require massive recursion or multiple arrays to work. This makes it the most attractive option for very large data sets of millions of items.

·The quick sort is an in-place, divide-and-conquer, massively recursive sort. As a normal person would say, it's essentially a faster in-place version of the merge sort. The quick sort algorithm is simple in theory, but very difficult to put into code

·The merge sort splits the list to be sorted into two equal halves, and places them in separate arrays. Each array is recursively sorted, and then merged back together to form the final sorted list. Like most recursive sorts, the merge sort has an algorithmic complexity of $O(n \log n)$.

·The heap sort works as it name suggests - it begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the sorted array. After removing the largest item, it reconstructs the heap and removes the largest remaining item and places it in the next open position from the end of the sorted array. This is repeated until there are no items left in the heap and the sorted array is full. Elementary implementations require two arrays - one to hold the heap and the other to hold the sorted elements.

11.11 Glossary

Bubble sort: Bubble sort is a simple sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order.

Heap Sort: Heapsort is a comparison-based sorting algorithm, and is part of the selection sort family. Although somewhat slower in practice on most machines than a good implementation of quicksort, it has the advantage of a worst-case ?(n log n) runtime. Heapsort is an in-place algorithm, but is not a stable sort.

Radix Sort: radix sort is a sorting algorithm that sorts integers by processing individual digits, by comparing individual digits sharing the same significant position.

Bucket Sort: Bucket sort, or bin sort, is a sorting algorithm that works by partitioning an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm.

11.12 Further Readings

1.Anany Levitin, "Introduction to the Design and Analysis of Algorithm", Pearson Education Asia, 2003.

2. T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein, "Introduction to Algorithms", PHI Pvt. Ltd., 2001

3.Donald E. Knuth,The Art of Computer Programming Sorting and Searching, Second Edition (Reading, Massachusetts: Addison-Wesley, 1998), ISBN 0-201-89685-0

4.R. G. Dromey ,How to Solve It By Computer (Prentice Hall, ISBN 0-13-434001-9).

5.Computer Algorithms by Horowitz, Sahni, Rajasekaran. Freeman- VPHS Publications

11.13 Answers to the Self learning exercises

1. Because the list is sorted by year of birth it is almost sorted by puls. Therefore insertion sort is a good algorithm for this problem.

2. A good solution would be heapsort because it provides stable runtime behaviour.

3. A good solution would be bucket sort. Because the number of digits is known in advance and the number of buckets are at maximum 100. All Students

can be sorted within O(n).

11.14 Unit -End Questions

1. How many types of sorting is there list a compare chart of all sorting algorithms .
2. Write down the steps for bubble sort algorithm
3. What is the difference between selection sort and insertion sort.
4. How heap sort is implemented.
5. Write a short note on following

   a. Quick sort b. Radix sort c. merge sort

----***----

UNIT XII

ALGORITHM DESIGN TECHNIQUES

STRUCTURE OF THE UNIT

12.0 Objectives

12.1 Introduction

12.2. Divide and conquer strategy

12.3 Greedy Method

12.3,1 Minimal Spanning Tree

12.3.2 Knapsack Problem

12.3.3 Job sequencing

12.3.4 Optimal Merge Pattern

12.4 Summary

12.5 Glossary

12.6 Further Readings

12.7 Answers to self learning exercises

12.8 Unit End Questions

12.1 Objectives

After completing this unit you will learn

- Algorithem Design Techniques,
- Divide And Conquer Stratrgy, Greedy Method
- Jobsequencing Optimal merge pattrns and minimal spanning trees etc.

12.1 Introduction

An algorithm is any set of detailed instructions which results in a predictable end-state from a known beginning. Algorithms are only as good as the instructions given, however, and the result will be incorrect if the algorithm is not properly defined.

For example an algorithm would be instructions for assembling a model airplane. Given the starting set of a number of marked pieces, one can follow the instructions given to result in a predictable end-state: the completed airplane. Misprints in the instructions, or a failure to properly follow a step will result in a faulty end product. How to design an algorithm ? It is the most imporatnt question. In this unit we will discuss basically two methods for algorithm development : Divide and Conquer strategy and Greedy strategy and then described how problems can be solved using theses algorithms and their complexity.

12.2 Divide and Conquer strategy

Divide and conquer was a successful military strategy long before it became an algorithm design paradigm. Generals observed that it was easier to defeat one army of 50,000 men, followed by another army of 50,000 men than it was to beat a single 100,000 man army. Thus the wise general would attack so as to divide the enemy army into two forces and then mop up one after the other.

To use divide and conquer as an algorithm design technique, we must divide the problem into two smaller sub problems, solve each of them recursively, and then meld the two partial solutions into one solution to the full problem. Whenever the merging takes less time than solving the two sub problems, we get an efficient algorithm. Merge sort is the classic example of a divide-and conquer algorithm. It takes only linear time to merge two sorted lists of n/2 elements each of which was obtained in O(n log n) time.

Divide and conquer is a design technique with many important algorithms to its credit, including merge sort, the fast Fourier transform, and matrix multiplication algorithm. However, with the exception of binary search.

This is a method of designing algorithms that (informally) proceeds as follows:

Given an instance of the problem to be solved, split this into several, smaller, sub-instances (of the same problem) independently solve each of the sub-instances and then combine the sub-instance solutions so as to yield a solution for the original instance. This description raises the question:

By what methods are the sub-instances to be independently solved?

The answer to this question is central to the concept of Divide & Conquer algorithm and is a key factor in gauging their efficiency.

Consider the following: We have an algorithm, alpha say, which is known to solve all problem instances of size n in at most c n^2 steps (where c is some constant). We then discover an algorithm, beta say, which solves the same problem by:

* Dividing an instance into 3 sub-instances of size n/2.

* Solves these 3 sub-instances.

* Combines the three sub-solutions taking d n steps to do this.

Suppose our original algorithm, alpha, is used to carry out the 'solves these sub-instances'. Let

T(alpha)(n) = Running time of alpha

T(beta)(n) = Running time of beta

Then,

T(alpha)(n) = c n^2 (by definition of alpha)

But

T(beta)(n) = 3 T(alpha)(n/2) + d n

= (3/4)(cn^2) + dn

So if dn < (cn^2)/4 (i.e. d < cn/4) then beta is faster tnan alpha

In particular for all large enough n, (n > 4d/c = Constant), beta is faster than alpha.

This realisation of beta improves upon alpha by just a constant factor. But if the problem size, n, is large enough then

n > 4d/c

n/2 > 4d/c

...

n/2^i > 4d/c

which suggests that using beta instead of alpha for the `solves these' stage repeatedly until the sub-sub-sub..sub-instances are of size n0 <= (4d/c) will yield a still faster algorithm.

So consider the following new algorithm for instances of size n

```
procedure gamma (n : problem size ) is
  begin
    if n <= n^-0 then
      Solve problem using Algorithm alpha;
      else
      Split into 3 sub-instances of size n/2;
      Use gamma to solve each sub-instance;
      Combine the 3 sub-solutions;
    end if;
  end gamma;
```

Let T(gamma)(n) denote the running time of this algorithm.

```
              cn^2                  if n <= n0
T(gamma)(n) =
           3T(gamma)( n/2 )+dn   otherwise
```

We shall show how relations of this form can be estimated later in the course. With these methods it can be shown that

T(gamma)(n) = O(n^{log3}) (=O(n^{1.59..})

This is an asymptotic improvement upon algorithms alpha and beta.

The improvement that results from applying algorithm gamma is due to the fact that it maximises the savings achieved beta.

The (relatively) inefficient method alpha is applied only to "small" problem sizes.

The precise form of a divide-and-conquer algorithm is characterised by:

* The threshold input size, n0, below which the problem size is not sub-divided.

* The size of sub-instances into which an instance is split.

* The number of such sub-instances.

* The algorithm used to combine sub-solutions.

In (II) it is more usual to consider the ratio of initial problem size to sub-instance size. In our example this was 2. The threshold in (I) is sometimes called the (recursive) base value. In summary, the generic form of a divide-and-conquer algorithm is:

```
procedure D-and-C (n : input size) is
  begin
    if n <= n0 then
      Solve problem without further
      sub-division;
  else
   Split into r sub-instances
    each of size n/k;
   for each of the r sub-instances do
      D-and-C (n/k);
   Combine the r resulting
    sub-solutions to produce
   the solution to the original problem;
  end if;
end D-and-C;
```

Fibonocci Number

The Fibonacci numbers are given by following recurrence

$$F_n = \begin{cases} 0 & n = 0, \\ 1 & n = 1, \\ F_{n-1} + F_{n-2} & n \geq 2. \end{cases} \qquad (14.4)$$

In this section we present a divide-and-conquer style of algorithm for computing Fibonacci numbers. We make use of the following identities

$$F_n = \begin{cases} 0 & n = 0, \\ 1 & n = 1, \\ (F_{\lceil n/2 \rceil})^2 + (F_{\lceil n/2 \rceil - 1})^2 & n \geq 2 \text{ and } n \text{ is odd}, \\ (F_{\lceil n/2 \rceil})^2 + 2F_{\lceil n/2 \rceil}F_{\lceil n/2 \rceil - 1} & n \geq 2 \text{ and } n \text{ is even}. \end{cases} \qquad (14.5)$$

Following program is to find out the fibonocci of the given number.

```
unsigned int Fibonacci (unsigned int n)
{
    if (n == 0 || n == 1)
        return n;
    else
    {
        unsigned int const a = Fibonacci ((n + 1) / 2);
        unsigned int const b = Fibonacci ((n + 1) / 2 - 1);
        if (n % 2 == 0)
            return a * (a + 2 * b);
        else
            return a * a + b * b;
    }
}
```

Program: Divide-and-Conquer Example—Computing Fibonacci Numbers

To determine a bound on the running time of the Fibonacci routine in Program we assume that T(n) is a non-decreasing function. i.e. $T(n) \geq T(n-1)$ for all $n \geq 1$. Therefore $T(\lceil n/2 \rceil) \geq T(\lceil n/2 \rceil - 1)$ Although the program works correctly for all values of n, it is convenient to assume that n is a power of 2. In this case, the running time of the routine is upper-bounded by T(n) where

$$T(n) = \begin{cases} O(1) & n \leq 1, \\ 2T(n/2) + O(1) & n > 1. \end{cases} \qquad (14.6)$$

Equation is easily solved using repeated substitution

$$\begin{aligned} T(n) &= 2T(n/2) + 1 \\ &= 4T(n/4) + 1 + 2 \\ &= 8T(n/8) + 1 + 2 + 4 \\ &\vdots \\ &= 2^k T(n/2^k) + \sum_{i=0}^{k-1} 2^i \\ &\vdots \\ &= nT(1) + n - 1 \quad (n = 2^k). \end{aligned}$$

Thus, T(n)=2n-1=O(n).

Binary Search

Binary search is a fast algorithm for searching in a sorted array S of keys. To search for key q, we compare q to the middle key S[n/2]. If q appears before S[n/2], it must reside in the top half of our set; if not, it must reside in the bottom half of our set. By recursively repeating this process on the correct half, we find the key in a total of [log n] comparisons, a big win over the n/2 we expect with sequential search.

This much you probably know. What is important is to have a sense for just how fast binary search is.

a popular children's game, where one player selects a word, and the other repeatedly asks true false questions in an attempt to identify the word. If the word remains unidentified after 20 questions, the first party wins; otherwise, the second player takes the honors. In fact, the second player always has a winning strategy, based on binary search. Given a printed dictionary, the player opens it in the middle, selects a word (say "move"), and asks whether the unknown word is before "move" in alphabetical order. Since standard dictionaries contain 50,000 to 200,000 words, we can be certain that the process will always terminate within twenty questions.

Other interesting algorithms follow from simple variants of binary search. For example, suppose we have an array A consisting of a run of 0's, followed by an unbounded run of 1's, and would like to identify the exact point of transition between them. Binary search on the array would provide the transition point in [log n] tests, if we had a bound n on the number of elements in the array. In the absence of such a bound, we can test repeatedly at larger intervals (A[1], A[2], A[4], A[8], A[16],) until we find a first non-zero value. Now we have a window containing the target and can proceed with binary search.

This one sided binary search finds the transition point p using at most comparisons, regardless of how large the array actually is. One-sided binary search is most useful whenever we are looking for a key that probably lies close to our current position.

Consider the following problem: one has a directory containing a set of names and a telephone number associated with each name.

The directory is sorted by alphabetical order of names. It contains n entries which are stored in 2 arrays:

names (1..n) ; numbers (1..n)

Given a name and the value n the problem is to find the number associated with the name.

We assume that any given input name actually does occur in the directory, in order to make the exposition easier.

The Divide & Conquer algorithm to solve this problem is the simplest example of the paradigm.

It is based on the following observation

Given a name, X say,

X occurs in the middle place of the names array

Or

X occurs in the first half of the names array. (U)

Or

X occurs in the second half of the names array. (L)

U (respectively L) are true only if X comes before (respectively after) that name stored in the middle place.

This observation leads to the following algorithm:

```
function search (X : name;
                 start, finish : integer)
                  return integer is
 middle : integer;
 begin
  middle := (start+finish)/2;
  if names(middle)=x then
     return numbers(middle);
  elsif X&ltnames(middle) then
     return search(X,start,middle-1);
  else -- X&gtnames(middle)
    return search(X,middle+1,finish);
  end if;
end search;
```

Divide-and-conquer recurrences.

Suppose that a recursive algorithm divides a problem of size n into a parts, where each sub-problem is of size n/b. Also suppose that a total number of g(n)extra operations are needed in the conquer step of the algorithm to combinethe solutions of the sub-problems into a solution of the original problem. Let f(n) be the number of operations required to solve the problem of size n.Then f satisfies the recurrence relation

$$f(n)=a\ f(n/b)+g(n)$$

and it is called divide-and-conquer recurrence relation.

The ultimate divide-and-conquer algorithm is, of course, *binary search*: finding a key k in a large file containing keys $A[0 .. n-1]$ in sorted order, we first compare k with $A[n/2]$, and depending on the result we recurse either on the first half of the file, $A[0 .. n/2-1]$, or on the second half $A[n/2 .. n-1]$. The recurrence now is $T(n) = T(n/2) + O(1)$, which is the case $r = 1$, $b = 2$, $d = 0$. Plugging into our master theorem, we get the running time of just $O(\log n)$.

12.3 Greedy Algorithms

This is another approach that is often used to design algorithms for solving

Optimisation Problems

In contrast to dynamic programming, however,

- Greedy algorithms do not always yield a genuinely optimal solution. In such cases the greedy method is frequently the basis of a heuristic approach.

- Even for problems which can be solved exactly by a greedy algorithm, establishing the correctness of the method may be a non-trivial process.

In order to give a precise description of the greedy paradigm we must first consider a more detailed definition of the environment in which typical optimisation problems occur. Thus in an optimisation problem, one will have, in the context of greedy algorithms, the following:

- A collection (set, list, etc) of candidates, e.g. nodes, edges in a graph, etc.

- A set of candidates which have already been 'used'.

- A predicate (solution) to test whether a given set of candidates give a solution (not necessarily optimal).

- A predicate (feasible) to test if a set of candidates can be extended to a (not necessarily optimal) solution.

- A selection function (select) which chooses some candidate which h as not yet been used.

- An objective function which assigns a value to a solution.

In other words: An optimisation problem involves finding a subset, S, from a collection of candidates, C; the subset, S, must satisfy some specified criteria, i.e. be a solution and be such that the objective function is optimised by S. 'Optimised' may mean

Minimised or Maximised

depending on the precise problem being solved. Greedy methods are distinguished by the fact that the selection function assigns a numerical value to each candidate, x, and chooses that candidate for which:

SELECT(x) is largest

or SELECT(x) is smallest

All Greedy Algorithms have exactly the same general form. A Greedy Algorithm for a particular problem is specified by describing the predicates 'solution' and 'feasible'; and the selection function 'select'.

Consequently, Greedy Algorithms are often very easy to design for optimisation problems.

The General Form of a Greedy Algorithm is

```
function select (C : candidate_set) return candidate;
function solution (S : candidate_set) return
boolean;
function feasible (S : candidate_set) return
boolean;
function greedy (C : candidate_set) return candidate_set is
x : candidate;
S : candidate_set;
begin
  S := {};
  while (not solution(S)) and C /= {} loop
    x := select( C );
    C := C - {x};
    if feasible( S union {x} ) then
      S := S union { x };
    end if;
  end loop;
  if solution( S ) then
    return S;
  else
    return es;
  end if;
end greedy;
```

As illustrative examples of the greedy paradigm we shall describe algorithms for the following problems:

- Minimal Spanning Tree.
- Integer Knapsack.

For the first of these, the algorithm always returns an optimal solution.

12.3.1 Minimal Spanning Tree

The inputs for this problem is an (undirected) graph, G(V, E) in which each edge, e in E, has an associated positive edge length, denoted Length(e).

The output is a spanning tree, T(V, F) of G(V,E) such that the total edge length, is minimal amongst all the possible spanning trees of G(V,E).

Note: An n-node tree, T is a connected n-node graph with exactly n-1 edges.

T(V,F) is a spanning tree of G(V,E) if and only if T is a tree and the edges in F are a subset of the edges in E.

In terms of general template given previously:

·The candidates are the edges of G(V,E).

·A subset of edges, S, is a solution if the graph T(V,S) is a spanning tree of G(V,E).

·A subset of edges, S, is feasible if there is a spanning tree T(V,H) of G(V,E) for which S sube H.

·The objective function which is to be minimised is the sum of the edge lengths in a solution.

·The select function chooses the candidate (i.e. edge) whose length is smallest (from the remaining candidates).

The full algorithm, discovered by Kruskal, is:

```
function min_spanning_tree (E : edge_set)
                    return edge_set is
S : edge_set;
e : edge;
begin
  S := (es;
  while (H(V,S) not a tree)
          and E /= {} loop
   e := Shortest edge in E;
   E := E - {e};
   if H(V, S union {e}) is acyclic then
     S := S union {e};
   end if;
  end loop;
  return S;
end min_spanning_tree;
```

Before proving the correctness of this algorithm, we give an example of it running.

The algorithm may be viewed as dividing the set of nodes, V, into n parts or components:

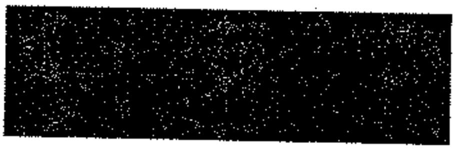
{1} ; {2} ; ... ; {n}

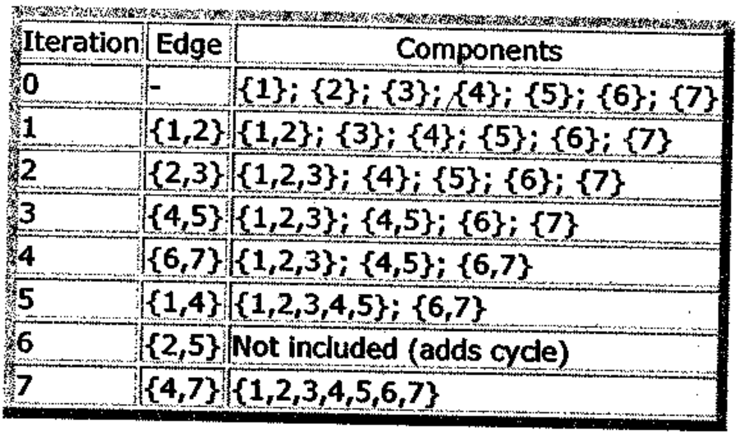
An edge is added to the set S if and only if it joins two nodes which belong to different components; if an edge is added to S then the two components containing its endpoints are coalesced into a single component.

In this way, the algorithm stops when there is just a single component

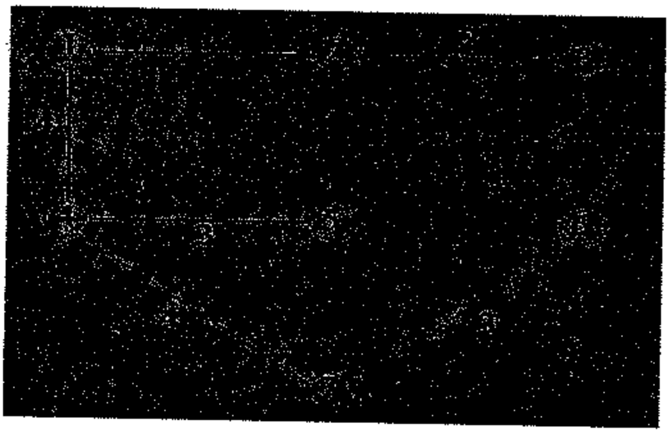
{ 1, 2 ,..., n }

remaining.





Iteration	Edge	Components
0	-	{1}; {2}; {3}; {4}; {5}; {6}; {7}
1	{1,2}	{1,2}; {3}; {4}; {5}; {6}; {7}
2	{2,3}	{1,2,3}; {4}; {5}; {6}; {7}
3	{4,5}	{1,2,3}; {4,5}; {6}; {7}
4	{6,7}	{1,2,3}; {4,5}; {6,7}
5	{1,4}	{1,2,3,4,5}; {6,7}
6	{2,5}	Not included (adds cycle)
7	{4,7}	{1,2,3,4,5,6,7}



How do we know that the resulting set of edges form a Minimal Spanning Tree?

In order to prove this we need the following result.

For G(V,E) as before, a subset, F, of the edges E is called promising if F is a subset of the edges in a minimal spanning tree of G(V,E).

Lemma: Let G(V,E) be as before and W be a subset of V.

Let F, a subset of E be a promising set of edges such that no edges in F has exactly one endpoint in W.

If {p,q} in E-F is a shortest edge having exactly one of p or q in W then: the set of edges F union { {p,q} } is promising.

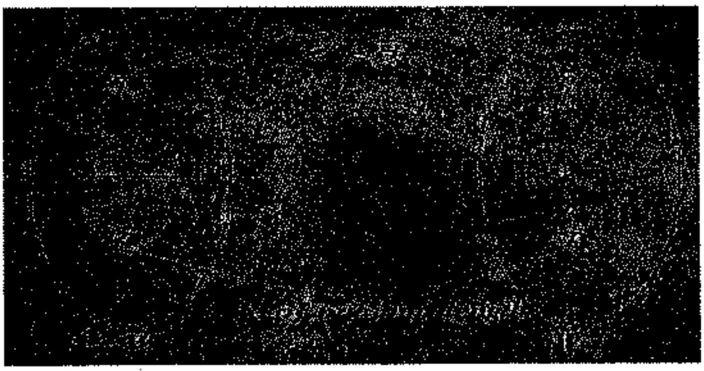
Proof: Let T(V,H) be a minimal spanning tree of G(V,E) such that F is a subset of H. Note that T exists since F is a promising set of edges.

Consider the edge e = {p,q} of the Lemma statement.

If e is in H then the result follows immediately, so suppose that e is not in H. Assume that p is in W and q is not in W and consider the graph T(V, H union {e}).

Since T is a tree the graph T (which contains one extra edge) must contain a cycle that includes the (new) edge {p,q}.

Now since p is in W and q is not in W there must be some edge, e' = {p',q'} in H which is also part of this cycle and is such that p' is in W and q' is not in W.



Now, by the choice of e, we know that

Length (e) < = Length (e')

Removing the edge e' from T gives a new spanning tree of G(V,E).

The cost of this tree is exactly

cost(T) - Length(e') + Length(e)

and this is < = cost(T).

T is a minimal spanning tree so either e and e' have the same length or this case cannot occur. It follows that there is a minimal spanning tree containing F union {e} and hence this set of edges is promising as claimed.

Theorem: Kruskal's algorithm always produces a minimal spanning tree.

Proof: We show by induction on k > = 0 - the number of edges in S at each stage - that the set of edges in S is always promising.

Base (k = 0): S = {} and obviously the empty set of edges is promising.

Step: (< = k-1 implies k): Suppose S contains k-1 edges. Let e = {p,q} be the next edge that would be added to S. Then:

- p and q lie in different components.
- {p,q} is a shortest such edge.

Let C be the component in which p lies. By the inductive hypothesis the set S is promising. The Inductive Step now follows by invoking the Lemma, with W = Set of nodes in C and F = S.

12.3.2 Integer Knapsack

In various forms this is a frequently arising optimisation problem.

Input: A set of items U = {u1, u2 ,..., uN}

each item having a given size s(ui) and value v(ui).

A capacity K.

Output: A subset B of U such that the sum over u in B of s(u) does not exceed K and the sum over u in B of v(u) is maximised.

Using a greedy approach, however, we can find a solution whose value is at worst 1/2 of the optimal value.

- The items, U, are the candidates.
- A subset, B, is a solution if the total size of B fits within the given capacity, but adding any other item will exceed the capacity.
- The objective function which is to be maximised is the total value.

The selection function chooses that item, ui for which

$$\frac{v(ui)}{s(ui)}$$

is maximal

These yield the following greedy algorithm which approximately solves the integer knapsack problem.

```
function knapsack (U : item_set;
                   K : integer )
                  return item_set is
C, S : item_set;
x : item;
begin
  C := U; S := {};
  while C /= {} loop
    x := Item u in C such that
         v(u)/s(u) is largest;
    C := C - {x};
    if ( sum over {u in S} s(u) ) + s(x) < = K then
      S := S union {x};
    end if;
  end loop;
  return S;
end knapsack;
```

A very simple example shows that the method can fail to deliver an optimal solution. Let

U = { u1, u2, u3 ,..., u12 }
s(u1) = 101 ; v(u1) = 102
s(ui) = v(ui) = 10 2 <= i <= 12
K = 110

Greedy solution: S = {u1}; Value is 102.

Optimal solution: S = U - {u1}; Value is 110.

12.3.2 JobSequencing

n jobs, S={1, 2, …, n}, each job i has a deadline $d_i > 0$ and a profit $p_i > 0$. We need one unit of time to process each job and we can do at most one job each time. We can earn the profit p_i if job i is completed by its deadline.

i	1	2	3	4	5
p_i	20	15	10	5	1
d_i	2	2	1	3	3

The optimal solution = {1, 2, 4}.

The total profit = 20 + 15 + 5 = 40.

Algorithm:

Step 1: Sort p_i into nonincreasing order. After sorting $p_1 {}^3 p_2 {}^3 p_3 {}^3 \dots {}^3 p_i$.

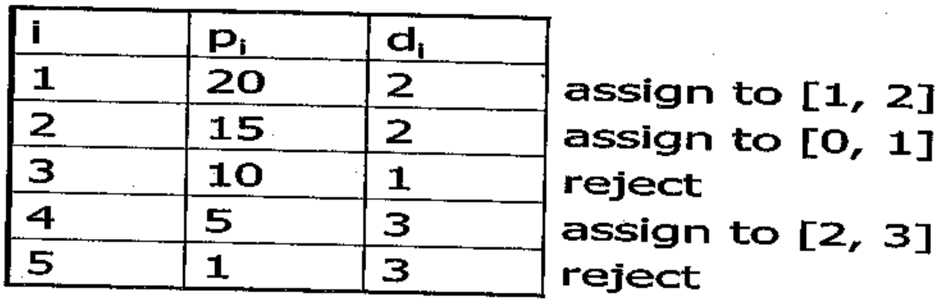
Step 2: Add the next job i to the solution set if i can be completed by its deadline. Assign i to time slot [r-1, r], where r is the largest integer such that 1 £ r £ d_i and [r-1, r] is free.

Step 3: Stop if all jobs are examined. Otherwise, go to step 2.

Time complexity: $O(n^2)$

i	p_i	d_i	
1	20	2	assign to [1, 2]
2	15	2	assign to [0, 1]
3	10	1	reject
4	5	3	assign to [2, 3]
5	1	3	reject

solution = {1, 2, 4}
total profit = 20 + 15 + 5 = 40
e.g.



i	p_i	d_i	
1	20	2	assign to [1, 2]
2	15	2	assign to [0, 1]
3	10	1	reject
4	5	3	assign to [2, 3]
5	1	3	reject

solution = {1, 2, 4}

total profit = 20 + 15 + 5 = 40

12.3.4 Optimal Merge Patterns

* Input: N sorted arrays of length L[1], L[2],...,L[n]

* Problem: Ultimateley, to merge the arrays pairwise as fast as possible. The problem is to determine which pair to merge everytime.

* Method (the Greedy method): The selection policy (of which best pair of arrays to merge next) is to choose the two shortest remaining arrays.

* Implementation:

o Need a data structure to store the lengths of the arrays, to find the shortest 2 arrays at any time, to delete those lengths, and insert in a new length (for the newly merged array).

o In essence, the data structure has to support delete-min and insert. Clearly, a min-heap is ideal.

o Time complexity of the algorithm: The algorithm iterates (n-1) times. At every iteration two delete-mins and one insert is performed. The 3 operations take O(log n) in each iteration.

o Thus the total time is O(nlog n) for the while loop + O(n) for initial heap construction.

o That is, the total time is O(nlog n).

Self learning exercises

1. Estimate the number of comparisons by a binary search.

2.Estimate the number of comparisons needed to find the max and the min elements of the list with n elements.

3.Job sequencing with deadlines using greedy method ,find an optimal solution to the problem of job sequncing with deadline where n=4,(p1,p2,p3,p4)=(100,10,5,27)and (d1,d2,d3,d4)=(2,1,2,1)

4.

12.4 Summary

- Many recursive algorithms take a problem with a given input and divide it into one or more smaller problems. This reduction is repeatedly applied until the solutions of smaller problems can be found quickly. This procedure is called divide-and-conquer algorithm.

- Greedy algorithms build up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Although such an approach can be disastrous for some computational tasks, there are many for which it is optimal.

12.5 Glossary

Knapsack Problem: The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most useful items.

Binary Search:In computer science, a binary search is an algorithm for locating the position of an element in a sorted list

Greedy algorithm: A greedy algorithm is any algorithm that follows the problem solving metaheuristic of making the locally optimal choice at each stage with the hope of finding the global optimum.

12.6 Further Readings

1.Anany Levitin, "Introduction to the Design and Analysis of Algorithm", Pearson Education Asia, 2003.

2. T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein, "Introduction to Algorithms", PHI Pvt. Ltd., 2001

3.Sara Baase and Allen Van Gelder, "Computer Algorithms - Introduction to Design and Analysis", Pearson Education Asia, 2003.

4. A.V.Aho, J.E. Hopcroft and J.D.Ullman, "The Design and Analysis Of Computer Algorithms", Pearson Education Asia, 2003.

5.S. Dasgupta, C.H. Papadimitriou, and U. V. Vazirani, Algorithms. May 2006.

6.Horowitz, Sahni, Rajasekaran,Computer Algorithms. Freeman- VPHS Publications

12.7 Answers to self learning exercises

1.we know that f(n)=f(n/2)+2 for even n, if f(n) is the number of comparisons needed to check if an element x is in a list of size n. with a=1, b=2, c=2, it is clear that f(n) is $O(logn)$.

2.we know that f(n)=2f(n/2)+2 for even n, if f(n) is the number of comparisons needed to find a max and min in a list of size n. with a=2, b=2, c=2, it is clear that f(n) is $O(n^{log_b a})$=O(n).

3.Problem: n jobs, S={1, 2, , n}, each job i has a deadline di ? 0 and a profit pi ? 0. We need one unit of time to process each job and we can do at most one job each time. We can earn the profit pi if job i is completed by its deadline.

The optimal solution = {1, 2, 4}.

The total profit = 20 + 15 + 5 = 40.

Algorithm:

Step 1: Sort pi into nonincreasing order. After sorting p1 ? p2 ? p3 ? ? pi.

Step 2: Add the next job i to the solution set if i can be completed by its deadline. Assign i to time slot [r-1, r], where r is the largest integer such that 1 ? r ? di and [r-1, r] is free.

Step 3: Stop if all jobs are examined. Otherwise, go to step 2.

12.8 Unit End Questionns

1. What is divide and conquer algorithm explain it.

2. Explain greedy Algorithm.

3. What is Knapsack problem.

4. Explain the minimal spanning tree.

---***---

UNIT XIII

DYNAMIC PROGRAMMING

STRUCTURE OF THE UNIT

13.0 Objective

13.1 Introduction

13.2 Dynamic Programming in Computer Programming

13.3 Matrix Chain Multiplication

13.4 Branch and Bound Method

13.5 Travelling Salesman Problem

13.6 Summary

13.7 Glossary

13.8 Further Readings

13.9 Answers to Self Learning Exercises

13.10 Unit End Questions

13.0 Objectives:

After completing this unit you would be able to understand the following points

- History of Dynamic Programming
- Implementation of Dynamic Programming in Computer and other fields.
- Top Down and Bottom Up Approach
- Balanced Matrix
- Applications of Dynamic Programming
- Matrix Chain Multiplication
- Branch and Bound Method
- Applications of Branch and Bound Algorithm
- Travelling Salesman Problem

13.1 Introduction

T- ve will discuss the advance topics, which includes the implementation of Dynamic g. This unit will describe the implementation of matrix chain multiplication, branch ;orithm and travelling salesman problem. These common problems are discussed in e dynamic programming.

:ience and mathematics, **dynamic programming** is a method through which ems are solved by breaking them down into simpler steps. It is applicable to problems that exhibit the properties of overlapping subproblems and optimal substructure . When applicable, the method takes much less time than naive methods. Bottom-up dynamic programming simply means storing the results of certain calculations, which are then re-used later because the same calculation is a sub-problem in a larger calculation. Top-down dynamic programming involves formulating a complex calculation as a recursive series of simpler calculations.

History

Richard Bellman originally used the term in the 1940s to describe the process of solving problems where one needs to find the best decisions one after another. By 1953, he had refined this to the modern meaning, which refers specifically to nesting smaller decision problems inside larger decisions, and the field was thereafter recognized by the IEEE as a systems analysis and engineering topic. Bellman's contribution is remembered in the name of the Bellman equation, a central result of dynamic programming which restates an optimization problem in recursive form.

Initially the word "programming" in "dynamic programming" had no connection to computer programming, and instead came from the term "mathematical programming" - a synonym for optimization. However, now many optimization problems are best solved by writing a computer program that implements a dynamic programming algorithm, rather than carrying out thousands of tedious calculations by hand. Some of the examples given below are illustrated using computer programs.

Overview

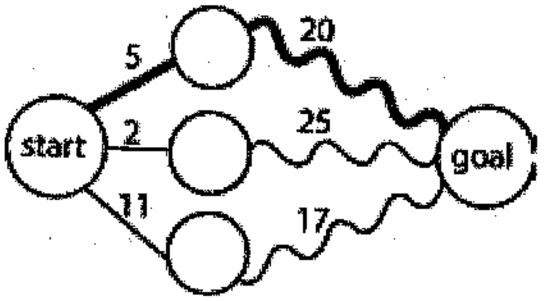


Figure 13.1 : Path Graph

Figure 13.1 is to Find the shortest path in a graph using optimal substructure; a straight line indicates a single edge; a wavy line indicates a shortest path between the two vertices it connects (other nodes on these paths are not shown); the bold line is the overall shortest path from start to goal.

Dynamic programming is both a computer programming method, and a mathematical optimization method. In both contexts, it is used for simplifying a complicated problem by breaking it down into simpler subproblems in a recursive manner. While some decision problems cannot be taken apart this way, decisions that span several points in time do often break apart recursively; Bellman called this the "Principle of Optimality". In computer science, a problem which can be broken down recursively is said to have optimal substructure.

If subproblems can be nested recursively inside larger problems, so that dynamic programming methods are applicable, then there is a relation between the value of the larger problem and the values of the subproblems. In the optimization literature this relationship is called the Bellman equation.

Dynamic programming in mathematical optimization

In terms of mathematical optimization, dynamic programming usually refers to a simplification of a decision by breaking it down into a sequence of decision steps over time. This is done by defining a sequence of **value functions** V1 , V2 , ... Vn , with an argument y representing the **state** of the system at times i from 1 to n. The definition of Vn(y) is the value obtained in state y at the last time n. The values Vi at earlier times i=n-1,n-2,...,2,1 can be found by working backwards, using a recursive relationship called the Bellman equation. For i=2,...n, Vi -1 at any state y is calculated from Vi by maximizing a simple function (usually the sum) of the gain from decision i-1 and the function Vi at the new state of the system if this decision is made. Since Vi has already been calculated, for the needed states, the above operation yields Vi -1 for all the needed states. Finally, V1 at the initial state of the system is the value of the optimal solution. The optimal values of the decision variables can be recovered, one by one, by tracking back the calculations already performed.

13.2 Dynamic programming in computer programming

As a computer programming method, dynamic programming is mainly used to tackle problems that are solvable in polynomial time. There are two key attributes that a problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping subproblems.

Optimal substructure means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its subproblems. Consequently, the first step towards devising a dynamic programming solution is to check whether the problem exhibits such optimal substructure. Such optimal substructures are usually described by means of recursion. For example, given a graph G=(V,E), the shortest path p from a vertex u to a vertex v exhibits optimal substructure: take any intermediate vertex w on this shortest path p. If p is truly the shortest path, then the path p1 from u to w and p2 from w to v are indeed the shortest paths between the corresponding vertices . Hence, one can easily formulate the solution for finding shortest paths in a recursive manner, which is what the Bellman-Ford algorithm does.

Overlapping subproblems means that the space of subproblems must be small, any recursive algorithm solving the problem should solve the same subproblems over and over, rather than generating another new subproblems. For example, consider the recursive formulation for generating the Fibonacci series: Fi = Fi-1 + Fi-2, with base case F1=F2=1. Then F43 = F42 + F41, and F42 = F41 + F40. Now F41 is being solved in the recursive subtrees of both F43 as well as F42. Even though the total number of subproblems is actually small (only 43 of them), we end up solving the same problems over and over if we adopt a naive recursive solution such as this. Dynamic programming takes account of this fact and solves each subproblem only once.

This can be achieved in any of the two ways:

(1)Top-down approach;

(2)Bottom-up approach.

Top-down approach: This is the direct fall-out of the recursive formulation of any problem. If the solution to any problem can be formulated recursively using the solution to its subproblems, and if its subproblems are overlapping, then one can easily memoize or store the solutions to the subproblems in a table. Whenever we attempt to solve a new subproblem, we first check the table to see if it is already solved. If a solution has been recorded, we can use it directly, otherwise we solve the subproblem and add its solution to the table.

- Bottom-up approach: This is the more interesting case. Once we formulate the solution to a problem recursively as in terms of its subproblems, we can try reformulating the problem in a bottom-up fashion: try solving the subproblems first and use their solutions to build-on and arrive at solutions to bigger subproblems. This is also usually done in a tabular form by iteratively generating solutions to bigger and bigger subproblems by using the solutions to small subproblems. For example, if we already know the values of F41 and F40, we can directly calculate the value of F42.

There are some programming languages that can automatically memoize the result of a function call with a particular set of arguments, in order to speed up call-by-name evaluation (this mechanism is referred to as call-by-need). Some languages make it possible portably (e.g. Scheme, Common Lisp or Perl), some need special extensions (e.g. C++). Some languages have automatic memoization built in. In any case, this is only possible for a referentially transparent function.

Examples: Computer algorithms

Fibonacci sequence

Here is a implementation of a function finding the nth member of the Fibonacci sequence, based directly on the mathematical definition:

```
function fib(n)
   if n = 0 return 0
   if n = 1 return 1
  return fib(n “ 1) + fib(n “ 2)
```

Notice that if we call, say, fib(5), we produce a call tree that calls the function on the same value many different times:

1. fib(5)
2. fib(4) + fib(3)
3. (fib(3) + fib(2)) + (fib(2) + fib(1))
4. ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
5. (((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))

In particular, fib(2) was calculated three times from scratch. In larger examples, many more values of fib, or subproblems, are recalculated, leading to an exponential time algorithm.

Now, suppose we have a simple map object, m, which maps each value of fib that has already been calculated to its result, and we modify our function to use it and update it. The resulting function requires only O(n) time instead of exponential time:

```
var m := map(0 ! 0, 1 ! 1)
function fib(n)
  if map m does not contain key n
     m[n] := fib(n “ 1) + fib(n “ 2)
  return m[n]
```

Technique of saving values that have already been calculated is called memoization; this is called top-down approach, as we had first broken the problem into subproblems and then calculate and store values.

In **bottom-up** approach we calculate the smaller values of fib first, then build larger values from them. This method also uses O(n) time since it contains a loop that repeats n “ 1 times, however it only takes constant (O(1)) space, in contrast to the top-down approach which requires O(n) space to store the map.

```
function fib(n)
   var previousFib := 0, currentFib := 1
    if n = 0
       return 0
    else if n = 1
       return 1
    repeat n “ 1 times
      var newFib := previousFib + currentFib
      previousFib := currentFib
      currentFib  := newFib
   return currentFib
```

In both these above mentioned examples, we only calculate fib(2) one time, and then use it to calculate both fib(4) and fib(3), instead of computing it every time either of them is evaluated.

A type of balanced 0-1 matrix

Consider the problem of assigning values, either zero or one, to the positions of an n x n matrix, n even, so that each row and each column contains exactly n / 2 zeros and n / 2 ones. For example, when n = 4, three possible solutions are:

```
+----+        +----+          +----+
|0101|        |0011|          |1100|
|1010| and    |0011|    and   |0011|
|0101|        |1100|          |1100|
|1010|        |1100|          |0011|
+----+        +----+          +----+
```

We ask how many different assignments there are for a given n. There are at least three possible approaches: brute force, backtracking, and dynamic programming. Brute force consists of checking all assignments of zeros and ones and counting those that have balanced rows and columns (n / 2 zeros and n / 2 ones). As there are $\binom{n}{n/2}^{n}$ possible assignments, this strategy is not practical except maybe up to n = 6. Backtracking for this problem consists of choosing some order of the matrix elements and recursively placing ones or zeros, while checking that in every row and column the number of elements that have not been assigned plus the number of ones or zeros are both at least n / 2. While more sophisticated than brute force, this approach will visit every solution once, making it impracticable for n larger than six, since the number of solutions is already 116963796250 for n = 8, as we shall see. Dynamic programming makes it possible to count the number of solutions without visiting them all.

We consider $k \times n$ boards, where $1 \leq k \leq n$ whose k rows contain n / 2 zeros and n / 2 ones. The function f to which memoization is applied maps vectors of n pairs of integers to the number of admissible boards (solutions). There is one pair for each column and its two components indicate respectively the number of ones and zeros that have yet to be placed in that column. We seek the value of $f((n/2, n/2), (n/2, n/2), \ldots (n/2, n/2))$(n arguments or one vector of n elements). The process of subproblem creation involves iterating over every one of $\binom{n}{n/2}$ possible assignments for the top row of the board, and going through every column, subtracting one from the appropriate element of the pair for that column, depending on whether the assignment for the top row contained a zero or a one at that position. If any one of the results is negative, then the assignment is invalid and does not contribute to the set of solutions (recursion stops). Otherwise, we have an assignment for the top row of the board and recursively compute the number of solutions to the remaining board, adding the numbers of solutions for every admissible assignment of the top row and returning the sum, which is being memoized. The base case is the trivial subproblem, which occurs for a board. The number of solutions for this board is either zero or one, depending on whether the vector is a permutation of n / 2 (0,1) and n / 2 (1,0) pairs or not.

For example, in the two boards shown above the sequences of vectors would be

((2, 2) (2, 2) (2, 2) (2, 2))   ((2, 2) (2, 2) (2, 2) (2, 2))   k = 4
0 1 0 1   0 0 1 1

((1, 2) (2, 1) (1, 2) (2, 1))   ((1, 2) (1, 2) (2, 1) (2, 1))   k = 3
1 0 1 0   0 0 1 1

((1, 1) (1, 1) (1, 1) (1, 1))   ((0, 2) (0, 2) (2, 0) (2, 0))   k = 2
0 1 0 1   1 1 0 0

((0, 1) (1, 0) (0, 1) (1, 0))   ((0, 1) (0, 1) (1, 0) (1, 0))   k = 1
1 0 1 0   1 1 0 0

((0, 0) (0, 0) (0, 0) (0, 0))   ((0, 0) (0, 0), (0, 0) (0, 0))

The number of solutions (sequence A058527 in OEIS) is

Checkerboard

Consider a checkerboard with n × n squares and a cost-function c(i, j) which returns a cost associated with square i,j (i being the row, j being the column). For instance (on a 5 × 5 checkerboard),

5	6	7	4	7	8
4	7	6	1	1	4

3	3	5	7	8	2
2	-	6	7	0	-
1	-	-	5*	-	-
	1	**2**	**3**	**4**	**5**

Thus c(1, 3) = 5

Let us say you had a checker that could start at any square on the first rank (i.e., row) and you wanted to know the shortest path (sum of the costs of the visited squares are at a minimum) to get to the last rank, assuming the checker could move only diagonally left forward, diagonally right forward, or straight forward. That is, a checker on (1,3) can move to (2,2), (2,3) or (2,4).

5

4

3

2   x x x

1   o

  1 2 3 4 5

This problem exhibits **optimal substructure**. That is, the solution to the entire problem relies on solutions to subproblems. Let us define a function q(i, j) as

q(i, j) = the minimum cost to reach square (i, j)

If we can find the values of this function for all the squares at rank n, we pick the minimum and follow that path backwards to get the shortest path.

Note that q(i, j) is equal to the minimum cost to get to any of the three squares below it (since those are the only squares that can reach it) plus c(i, j). For instance:

5

4   A

3   B C D

2

1

  1 2 3 4 5

Now lets redefine mincost function

```
function minCost(i, j)
  if j < 1 or j > n
    return infinity
  else if i = 5
    return c(i, j)
  else
    return min( minCost(i+1, j-1), minCost(i+1, j), minCost(i+1, j+1) ) + c(i, j)
```

It should be noted that this function only computes the path-cost, not the actual path. We will get to the path soon. This, like the Fibonacci-numbers example, is horribly slow since it spends mountains of time recomputing the same shortest paths over and over. However, we can compute it much faster in a bottom-up fashion if we use a two-dimensional array q[i, j] instead of a function. Why do we do that? Simply because when using a function we recompute the same path over and over, and we can choose what values to compute first.

We also need to know what the actual path is. The path problem we can solve using another array p[i, j], a predecessor array. This array basically says where paths come from. Consider the following code:

```
function computeShortestPathArrays()
  for x from 1 to n
    q[1, x] := c(1, x)
  for y from 1 to n
    q[y, 0]    := infinity
    q[y, n + 1] := infinity
  for y from 2 to n
    for x from 1 to n
      m := min(q[y-1, x-1], q[y-1, x], q[y-1, x+1])
      q[y, x] := m + c(y, x)
      if m = q[y-1, x-1]
        p[y, x] := -1
      else if m = q[y-1, x]
        p[y, x] :=  0
      else
        p[y, x] :=  1
```

Now the rest is a simple matter of finding the minimum and printing it.

```
function computeShortestPath()
  computeShortestPathArrays()
  minIndex := 1
  min := q[n, 1]
  for i from 2 to n
    if q[n, i] < min
      minIndex := i
      min := q[n, i]
  printPath(n, minIndex)
function printPath(y, x)
  print(x)
  print("<-")
  if y = 2
    print(x + p[y, x])
```

```
else
  printPath(y-1, x + p[y, x])
```

Sequence alignment

In genetics, sequence alignment is an important application where dynamic programming is essential. Typically, the problem consists of transforming one sequence into another using edit operations that replace, insert, or remove an element. Each operation has an associated cost, and the goal is to find the sequence of edits with the lowest total cost.

The problem can be stated naturally as a recursion, a sequence A is optimally edited into a sequence B by either:

1. inserting the first character of B, and performing an optimal alignment of A and the tail of B
2. deleting the first character of A, and performing the optimal alignment of the tail of A and B
3. replacing the first character of A with the first character of B, and performing optimal alignments of the tails of A and B.

The partial alignments can be tabulated in a matrix, where cell (i,j) contains the cost of the optimal alignment of A[1..i] to B[1..j]. The cost in cell (i,j) can be calculated by adding the cost of the relevant operations to the cost of its neighboring cells, and selecting the optimum.

Algorithms that use dynamic programming

- Backward induction as a solution method for finite-horizon discrete-time dynamic optimization problems
- Method of undetermined coefficients can be used to solve the Bellman equation in infinite-horizon, discrete-time, discounted, time-invariant dynamic optimization problems
- Many string algorithms including longest common subsequence, longest increasing subsequence, longest common substring, Levenshtein distance (edit distance).
- Many algorithmic problems on graphs can be solved efficiently for graphs of bounded treewidth or bounded clique-width by using dynamic programming on a tree decomposition of the graph.
- The Cocke-Younger-Kasami (CYK) algorithm which determines whether and how a given string can be generated by a given context-free grammar
- The use of transposition tables and refutation tables in computer chess
- The Viterbi algorithm (used for hidden Markov models)
- The Earley algorithm (a type of chart parser)
- The Needleman-Wunsch and other algorithms used in bioinformatics, including sequence alignment, structural alignment, RNA structure prediction.
- Floyd's All-Pairs shortest path algorithm
- Optimizing the order for chain matrix multiplication
- Pseudopolynomial time algorithms for the Subset Sum and Knapsack and Partition problem Problems

- The dynamic time warping algorithm for computing the global distance between two time series
- The Selinger (a.k.a. System R) algorithm for relational database query optimization
- De Boor algorithm for evaluating B-spline curves
- Duckworth-Lewis method for resolving the problem when games of cricket are interrupted
- The Value Iteration method for solving Markov decision processes
- Some graphic image edge following selection methods such as the "magnet" selection tool in Photoshop
- Some methods for solving interval scheduling problems
- Some methods for solving word wrap problems
- Some methods for solving the travelling salesman problem
- Recursive least squares method
- Beat tracking in Music Information Retrieval.
- Adaptive Critic training strategy for artificial neural networks
- Stereo algorithms for solving the Correspondence problem used in stereo vision.
- Seam carving (content aware image resizing)
- The Bellman-Ford algorithm for finding the shortest distance in a graph.
- Some approximate solution methods for the linear search problem.

13.3 Matrix Chain Multiplication

Given: A sequence of matrices A1...An to be multiplied where each Ai has dimension pi-1 x pi.
To do: A parenthesization of the sequence that minimizes the number of scalar multiplications needed.

Important facts:

- Matrix multiplication is associative so each parenthesization gives the same final product. Consequently, solutions to this problem can be very useful in practice.
- The dimension of the product of Ai...Aj, for i <= j, is pi-1 x pj.
- To multiply two matrices A and B, A with dimension p x q and B with dimension q x r, we require p*q*r scalar multiplications.

Step 1: Characterize optimal subproblems

In an optimal parenthesization, the final matrix multiplication is between the product of A1*...*Ai and the product of Ai+1*...*An, for some i between 1 and n-1.

The parenthesization of A1*...*Ai in the optimal parenthesization of A1*...*An is optimal.

and

The parenthesization of Ai+1*...*An in the optimal parenthesization of A1*...*An is optimal.

Step 2: Recursive algorithm

```
Matrix-Mult-Value(A1...An)
if n<=1 then
solution = 0
```

```
else
 solution = infinity
 for i = 1..n-1 do
  solution = min(solution, Matrix-Mult-Value(A1...Ai)
                   + Matrix-Mult-Value(Ai+1...An)
                     + p1pipn)
return solution
```

Running time? Exponential.

The number of recursive algorithm calls is given by:

T(n) = 1 + sumi=1n-1 [T(i) + T(n-i)]
T(1) = 1

We show that T(n) is at least 2n+1+n by induction, so assume that T(n) >= 2n+1+n for all n < C, for some C >= 1.
Now consider n=C+1.

T(n) = 1 + sumi=1n-1 [T(i) + T(n-i)]
>= 1 + 2(22+...+2n+ 1+...+(n-1))
= 1 + 8(20+...+2n-2)+2(1+...+(n-1))
= 1 + 8(2n-1-1) + n(n-1)
= 2n+2 + 1 + n(n-1)
= 2n+2 + n + 1 + n(n-2)
>= 2n+1 + n + 1 + n(n-2)
>= 2n+1 + n

Step 3: Dynamic programming algorithm

How many different subproblems are we actually solving?
Each recursive call has the form Matrix-Mult-Value(Ai...Aj)
for 1<= i <= j <= n.
Consequently, we are solving less than n2 different subproblems

As usual, we use a recursive algorithm with table look-up or an iterative algorithm that fills in a table bottom-up. Since there are less n2 subproblems, you might think that your algorithm(s) will run in O(n2) time. Actually, they will run in O(n3) time because we may make up to n-1 comparisons in order to compute a single value in the dynamic programming table.

Step 4: Reconstructing an optimal solution

Matrix chain multiplication

Matrix chain multiplication is an optimization problem that can be solved using dynamic programming. Given a sequence of matrices, we want to find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

We have many options because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

$(ABC)D = (AB)(CD) = A(BCD) = A(BC)D = \ldots$

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

$(AB)C = (10\times30\times5) + (10\times5\times60) = 1500 + 3000 = 4500$ operations

$A(BC) = (30\times5\times60) + (10\times30\times60) = 9000 + 18000 = 27000$ operations

Clearly the first method is the more efficient. Now that we have identified the problem, how do we determine the optimal parenthesization of a product of n matrices? We could go through each possible parenthesization (brute force), but this would require time O(2n), which is very slow and impractical for large n. The solution, as we will see, is to break up the problem into a set of related subproblems. By solving subproblems one time and reusing these solutions many times, we can drastically reduce the time required. This is known as dynamic programming.

Dynamic Programming Algorithm

To begin, let's assume that all we really want to know is the minimum cost, or minimum number of arithmetic operations, needed to multiply out the matrices. If we're only multiplying two matrices, there's only one way to multiply them, so the minimum cost is the cost of doing this. In general, we can find the minimum cost using the following recursive algorithm:

- Take the sequence of matrices and separate it into two subsequences.
- Find the minimum cost of multiplying out each subsequence.
- Add these costs together, and add in the cost of multiplying the two result matrices.
- Do this for each possible position at which the sequence of matrices can be split, and take the minimum over all of them.

For example, if we have four matrices ABCD, we compute the cost required to find each of (A)(BCD), (AB)(CD), and (ABC)(D), making recursive calls to find the minimum cost to compute ABC, AB, CD, and BCD. We then choose the best one. Better still, this yields not only the minimum cost, but also demonstrates the best way of doing the multiplication: just group it the way that yields the lowest total cost, and do the same for each factor.

Unfortunately, if we implement this algorithm we discover that it's just as slow as the naive way of trying all permutations! What went wrong? The answer is that we're doing a lot of redundant work. For example, above we made a recursive call to find the best cost for computing both ABC and AB. But finding the best cost for computing ABC also requires finding the best cost for AB. As the recursion grows deeper, more and more of this type of unnecessary repetition occurs.

One simple solution is called memoization: each time we compute the minimum cost needed to multiply out a specific subsequence, we save it. If we are ever asked to compute it again, we simply give the saved answer, and do not recompute it. Since there are about n2/2 different subsequences, where n is the number of matrices, the space required to do this is reasonable. It can be shown that this simple trick brings the runtime down from O(2n) to O(n3), which is more than efficient enough for real applications. This is top-down dynamic programming.

Pseudocode:

```
Matrix-Chain-Order(int p[])
{
  n = p.length - 1;
```

```
for (i = 1; i <= n; i++)
  m[i,i] = 0;

for (l=2; l<=n; l++) { // l is chain length
  for (i=1; i<=n-l+1; i++) {
     j = i+l-1;
     m[i,j] = MAXINT;
     for (k=i; k<=j-1; k++) {
     q = m[i,k] + m[k+1,j] + p[i-1]*p[k]*p[j];//Matrix Ai has the dimension  p[i-1] x p[i].
       if (q < m[i,j]) {
          m[i,j] = q;
          s[i,j] = k;
       }
     }
  }
}
```

Another solution is to anticipate which costs we will need and precompute them. It works like his:

For each k from 2 to n, the number of matrices:

ɔ Compute the minimum costs of each subsequence of length k, using the costs already computed.

When we're done, we have the minimum cost for the full sequence. Although it also requires O(n3) time, this approach has the practical advantages that it requires no recursion, no testing if a value has already been computed, and we can save space by throwing away some of the subresults that are no longer needed. This is bottom-up dynamic programming: a second way by which this problem can be solved.

13.4 Branch and Bound (BB)

Branch and bound (BB) is a general algorithm for finding optimal solutions of various optimization problems, especially in discrete and combinatorial optimization. It consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidates are discarded en masse, by using upper and lower estimated bounds of the quantity being optimized.

The method was first proposed by A. H. Land and A. G. Doig in 1960 for linear programming.

General description

For definiteness, we assume that the goal is to find the minimum value of a function f(x), where x ranges over some set S of admissible or candidate solutions (the search space or feasible region). Note that one can find the maximum value of f(x) by finding the minimum of g(x) = " f(x). (For example, S could be the set of all possible trip schedules for a bus fleet, and f(x) could be the expected revenue for schedule x.)

A branch-and-bound procedure requires two tools. The first one is a splitting procedure that, given a set S of candidates, returns two or more smaller sets whose union covers S. Note that the minimum of f(x) over S is , where each vi is the minimum of f(x) within Si. This step is called **branching**, since its recursive application defines a tree structure (the search tree) whose nodes are the subsets of S.

Another tool is a procedure that computes upper and lower bounds for the minimum value of f(x) within a given subset S. This step is called **bounding.**

The key idea of the BB algorithm is: if the lower bound for some tree node (set of candidates) A is greater than the upper bound for some other node B, then A may be safely discarded from the search. This step is called **pruning**, and is usually implemented by maintaining a global variable m (shared among all nodes of the tree) that records the minimum upper bound seen among all subregions examined so far. Any node whose lower bound is greater than m can be discarded.

The recursion stops when the current candidate set S is reduced to a single element; or also when the upper bound for set S matches the lower bound. Either way, any element of S will be a minimum of the function within S.

Effective subdivision

The efficiency of the method depends strongly on the node-splitting procedure and on the upper and lower bound estimators. All other things being equal, it is best to choose a splitting method that provides non-overlapping subsets.

Ideally the procedure stops when all nodes of the search tree are either pruned or solved. At that point, all non-pruned subregions will have their upper and lower bounds equal to the global minimum of the function. In practice the procedure is often terminated after a given time; at that point, the minimum lower bound and the minimum upper bound, among all non-pruned sections, define a range of values that contains the global minimum. Alternatively, within an overriding time constraint, the algorithm may be terminated when some error criterion, such as (max - min)/(min + max), falls below a specified value.

The efficiency of the method depends critically on the effectiveness of the branching and bounding algorithms used; bad choices could lead to repeated branching, without any pruning, until the sub-regions become very small. In that case the method would be reduced to an exhaustive enumeration of the domain, which is often impractically large. There is no universal bounding algorithm that works for all problems, and there is little hope that one will ever be found; therefore the general paradigm needs to be implemented separately for each application, with branching and bounding algorithms that are specially designed for it.

Branch and bound methods may be classified according to the bounding methods and according to the ways of creating/inspecting the search tree nodes.

The branch-and-bound design strategy is very similar to backtracking in that a state space tree is used to solve a problem. The differences are that the branch-and-bound method (1) does not limit us to any particular way of traversing the tree and (2) is used only for optimization problems.

This method naturally lends itself for parallel and distributed implementations, see, e.g., the traveling salesman problem article.

Applications

This approach is used for a number of NP-hard problems, such as

- Knapsack problem

- Integer programming
- Nonlinear programming
- Traveling salesman problem (TSP)
- Quadratic assignment problem (QAP)
- Maximum satisfiability problem (MAX-SAT)
- Nearest neighbor search (NNS)
- Cutting stock problem
- False noise analysis (FNA)

Branch-and-bound may also be a base of various heuristics. For example, one may wish to stop branching when the gap between the upper and lower bounds becomes smaller than a certain threshold. This is used when the solution is "good enough for practical purposes" and can greatly reduce the computations required. This type of solution is particularly applicable when the cost function used is noisy or is the result of statistical estimates and so is not known precisely but rather only known to lie within a range of values with a specific probability. An example of its application here is in biology when performing cladistic analysis to evaluate evolutionary relationships between organisms, where the data sets are often impractically large without heuristics.

For this reason, branch-and-bound techniques are often used in game tree search algorithms, most notably through the use of alpha-beta pruning.

Branch and Bound Algorithm Technique

Branch and bound is another algorithm technique that we are going to present in our multi-part article series covering algorithm design patterns and techniques. B&B, as it is often abbreviated, is one of the most complex techniques and surely cannot be discussed in its entirety in a single article. Thus, we are going to focus on the so-called A* algorithm that is the most distinctive B&B graph search algorithm.

If you have followed this article series then you know that we have already covered the most important techniques such as backtracking, the greedy strategy, divide and conquer, dynamic programming, and even genetic programming. As a result, in this part we will compare branch and bound with the previously mentioned techniques as well. It is really useful to understand the differences.

Branch and bound is an algorithm technique that is often implemented for finding the optimal solutions in case of optimization problems; it is mainly used for combinatorial and discrete global optimizations of problems. In a nutshell, we opt for this technique when the domain of possible candidates is way too large and all of the other algorithms fail. This technique is based on the en masse elimination of the candidates.

You should already be familiar with the tree structure of algorithms. Out of the techniques that we have learned, both the backtracking and divide and conquer traverse the tree in its depth, though they take opposite routes. The greedy strategy picks a single route and forgets about the rest. Dynamic programming approaches this in a sort of breadth-first search variation (BFS).

Now if the decision tree of the problem that we are planning to solve has practically unlimited depth, then, by definition, the backtracking and divide and conquer algorithms are out. We shouldn't rely on greedy because that is problem-dependent and never promises to deliver a global optimum, unless proven otherwise mathematically.

As our last resort we may even think about dynamic programming. The truth is that maybe the problem can indeed be solved with dynamic programming, but the implementation wouldn't be an efficient approach; additionally, it would be very hard to implement. You see, if we have a complex problem where we would need lots of parameters to describe the solutions of sub-problems, DP becomes inefficient.

If you still need a real-world proof, then there's the fifteen puzzle. One of the most straightforward implementations of dynamic programming would require 16 distinctive parameters to represent the optimum values of the solutions of each sub-problem. That means a 16-dimensional array. You see, this is why DP is out of question!

On the previous page you saw that the most promising approach was dynamic programming. Basically, let's continue to look at breadth-first search [BFS]. The drawback is that the complexity of BFS is exponential. We need an algorithm that ameliorates this issue by reducing a whole number of possible candidates that just aren't satisfactory and, thus, won't contribute to the optimal solution. Answer: the B&B.

That is what branch and bound does. This algorithm injects some intelligence into the naïve but complex breadth-first search. Instead of searching throughout the entire decision/search tree structure, it instills some sort of criteria, according to which the complexity of the BFS can be reduced. We can do this by adding "costs."

For example, if we calculate the distance of each node in terms of "how far" it is located from the initial root node configuration and "how close" it is from the solution, then the distance/cost is the sum of the aforementioned two distances. However, as you can surely see, the second distance relies on heuristics. Thus, it's just a guess.

Moreover, we can move through this tree based on the instilled costs; a node is a more possible candidate toward the solution if its cost is less than other nodes. What we did is add an essence of depth-search to the breadth-first meaning; meaning, there is going to be a priority queue, according to which we are running the breath-first search. The traditional BFS runs from left to right, but now we'll have the priority queue instead.

The beauty of this approach is that we haven't lost the not-so-possible candidates. You see, they are stored somewhere on the end of the priority queue, so this algorithm doesn't neglect the rest of the possible options. Summing these up, this is in fact a typical Branch and Bound algorithm technique. And yes, it relies on the guessing game.

B&B is composed of two main actions, but there's an additional step. The first step is, as its name suggests, the branching. This is where we define the tree structure from the set of candidates in a recursive manner. The second action is called bounding since this procedure calculates the upper and lower bounds of each node from the tree.

Furthermore, there's the additional pruning step that we can add. In a nutshell, it means that if the lower bound for some node of the tree is greater than the upper bound of some other node of the tree, then the first node of the tree can be "discarded" (remember, it does not mean eliminate) from the search. A variable always holds the value of the global minimum upper bound.

All in all, B&B is very similar to backtracking. The main differences are that the former is used only in case of optimization problems, whereas backtracking cannot be, and B&B doesn't limit us to a particular way of traversing the tree (depth-first search / preorder as with backtracking). Backtracking always picks one single successor from the candidates, while B&B always has the entire list of successors in the queue.

Here we are going to sum up the key elements of the B&B algorithm technique. Later on a typical problem will be presented, namely the knapsack problem. You will surely understand by the end of this article why the knapsack is an NP-hard problem that is definitely an optimization problem. But first let's see the resume of B&B.

A branch and bound algorithm is based on an advanced breadth-first search. The said BFS is done with a priority queue [PQ] instead of the traditional list. You can imagine the priority queue as any other queue, except that it is sorted. This means that the highest priority element is always on the first position.

It is crucial to understand the importance of these two functions: g(x) and h(x). The first function, g(x), calculates the distance between the x node and the root node. The latter, h(x), is a heuristic function because it estimates how close the said x node is to the solution (leaf). The heuristic function ought to return 0 in the case of a solution leaf (since the distance is zero). The efficiency of the B&B relies on this function.

Moreover, we can say that **f(x) = g(x) + h(x)** is a distance-plus-cost heuristic function itself, too. The g(x) part is the path-cost function, while the h(x) part is the admissible heuristic estimate; the sum of these two is the f(x). At the beginning of this article we mentioned the A* (A star) algorithm. This is when it comes into the picture.

The A* is a best-first, graph search algorithm. It is used when we want to find the least-cost path from an initial node to a specified goal node. This A* algorithm works on the basis of the aforementioned f(x) distance-plus-cost heuristic function. It starts with the initial node and expands it with the lowest f(x) value (lowest cost-per-benefit). But it stores all of the partial solutions in the priority queue (unexpanded leaf nodes).

This particular variation of the B&B algorithm works in the same fashion. We are going to work with two lists: open and close. In the former we store all of the non-expanded configurations, while the latter stores the expanded configurations. A solution is found only if the h(x) returns 0 (thus, a solution leaf) or the PQ queue becomes empty.

Here's a brief guide in pseudocode:

```
Procedure B&B:
open := initial configurations;
g := 0; f := h; solution := FALSE;
while ((h NOT equal 0) AND (open NOT empty)) do
t := node with the lowest f value;
expand this t node with its successors;
for each successor of t node do
g := g(t) + cost of the expansion; // this cost is often = 1
if (current successor is part of close OR empty)
then if (g < g(current configuration))
then link the current successor in the path;
set the new g value;
if (this successor found in close)
add in open;
```

```
end if;
end if;
else
add in open;
end for;
if (h equals 0) // solution is found
recursively print the route between this leaf node and the root
solution = TRUE;
end if;
end while;
if (solution equals FALSE)
no solutions;
end procedure;
```

The Knapsack problem is a combinatorial optimization problem. You are given a set of items, each with its own cost and value, and you are to determine the number of each item that you should pack into the knapsack so that the total cost doesn't exceed the given limitation, but the total value is as high as possible. As you can see, this is a maximization problem; it is part of combinatorics and applied mathematics.

The value is the weight of the items, and their cost represents how much the item is worth. The knapsack can hold a specified amount of weight, and this limitation cannot be exceeded. In short, you want to maximize the storage capability of the knapsack by packing the most valuable items in it. Sometimes this problem is told in the form of a robbery. Obviously, the thief wants the best bang for his buck (his effort).

Let's say the weight of the items are W1, W2, ... Wk, ... Wn and their cost is C1, C2, ... Ck, ... Cn, respectively; the capacity of the knapsack is specified as K. Now we have the following mathematical formula to calculate the upper bound **[UB]**. Check it out!

Right after this point we can already present the pseudocode of the algorithm.

```
Procedure knapsack:
Initialize root;
PQ <- root;
max_cost := root.cost;
while PQ not equal do
current <- PQ;
if (current.bound > max_cost) then
create left_child := next item;
if (left_child.cost > max_cost)
max_cost := left_child.cost;
update best_solution;
```

```
end if;
if (left_child.bound > max_cost)
PQ <- left_child;
end if;
create right_child; // it skips packing the next item
if (right_child.bound > max_cost)
PQ <- right_child;
end if;
end if;
end while;
return best_solution and its cost;
end procedure;
```

13.5 Travelling salesman problem

The **Travelling Salesman Problem** (TSP) is a problem in combinatorial optimization studied in operations research and theoretical computer science. Given a list of cities and their pairwise distances, the task is to find a shortest possible tour that visits each city exactly once.

This problem was initially formulated as a mathematical problem in 1930 and is one of the most intensively discussed problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, a large number of heuristics and exact methods are known, so that some instances with tens of thousands of cities can be solved.

The TSP has various applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as genome sequencing. In these applications, the concept city represents, for example, customers, soldering points, or DNA fragments, and the concept distance represents travelling times or cost, or a similarity measure between DNA fragments. In many applications, additional constraints such as limited resources or time windows make the problem considerably harder.

In the theory of computational complexity, the decision version of TSP belongs to the class of NP-complete problems. Thus, it is assumed that there is no efficient algorithm for solving TSP problems. In other words, it is likely that the worst case running time for any algorithm for TSP increases exponentially with the number of cities, so even some instances with only hundreds of cities will take many CPU years to solve exactly.

Description

As a graph problem

Symmetric TSP with four cities

TSP can be modelled as a graph: the graph's vertices correspond to cities and the graph's edges correspond to connections between cities, the length of an edge is the corresponding connection's distance. A TSP tour is now a Hamiltonian cycle in the graph, and an optimal TSP tour is a shortest Hamiltonian cycle.

Often, the underlying graph is a complete graph, so that every pair of vertices is connected by an edge. This is a useful simplifying step, because it makes it easy to find a solution, however bad, because the Hamiltonian cycle problem in complete graphs is easy. Instances where not all cities are connected can be transformed into complete graphs by adding very long edges between these cities, edges that will not appear in the optimal tour.

Asymmetric and symmetric

In the symmetric TSP, the distance between two cities is the same in each direction. Thus, the underlying structure is an undirected graph between; especially, each tour has the same length in both directions. This symmetry halves the number of feasible solutions.

In the asymmetric TSP, the distance from one city to the other need not equal the distance in the other direction, in general, there may not even be a connection in the other direction. Thus, the underlying structure is a directed graph. For example, the asymmetric case models one-way streets or air-fares that depend on the direction of travel.

With metric distances

In the metric TSP the intercity distances satisfy the triangle inequality. This can be understood as "no shortcuts", in the sense that the direct connection from A to B is never longer than the detour via C.

These edge lengths define a metric on the set of vertices. When the cities are viewed as points in the plane, many natural distance functions are metrics.

- In the Euclidian TSP the distance between two cities is the Euclidean distance between the corresponding points.
- In the Rectilinear TSP the distance between two cities is the sum of the differences of their x- and y-coordinates. This metric is often called the Manhattan distance or city-block metric.
- In the maximum metric, the distance between two points is the maximum of the differences of their x- and y-coordinates.

The last two metrics appear for example in routing a machine that drills a given set of holes in a printed circuit. The Manhattan metric corresponds to a machine that adjusts first one co-ordinate, and then the other, so the time to move to a new point is the sum of both movements. The maximum metric corresponds to a machine that adjusts both co-ordinates simultaneously, so the time to move to a new point is the slower of the two movements.

Non-metric distances

Distance measures that do not satisfy the triangle inequality arise in many routing problems. For example, one mode of transportation, such as travel by airplane, may be faster, even though it covers a longer distance.

In its definition, the TSP does not allow cities to be visited twice, but many applications do not need this constraint. In such cases, a symmetric, non-metric instance can be reduced to a metric one. This replaces the original graph with a complete graph in which the inter-city distance cij is replaced by the shortest path between i and j in the original graph.

Self Learning Exercises

1.The traveling salesperson problem is to visit a number of cities in the minimum distance. For instance, a politician begins in New York and has to visit Miami, Dallas, and Chicago before returning to New York. How can she minimize the distance traveled? The distances are as in following figure:

	New York	Miami	Dallas	Chicago
1. NewYork	---	1334	1559	809
2. Miami	1334	---	1343	1397
3.Dallas	1559	1343	---	921
4. Chicago	809	1397	921	---

2.Money-board problem

Given an *NxN* board find the path starting from any square at the start, to a target square in row n which maximizes the amount of money that can be obtained. The person can only move forward, diagonally or straight. Let's assume we have a 4x4 board

4				
3		X	X	X
2			O	
1				
	1	2	3	4

Numbers represent index of each row and column.

· O represents our current square.

· X represents the squares we can travel to.

· Cash [*r*, *c*] returns the cash amount on square (*r*, *c*)

· T represents our target square

So how do we solve this? We could use brute force by generating all possible paths and find the path which has the most money but, there are 3*N* possibilities which is too slow in cases when *N* gets large. So we will use dynamic programming.

13.6 Summary

In this unit we have discussed in detail regarding the concepts of Dynamic programming. We have learnt in brief regarding the history of dynamic programming and its use in computer science and other fields. We have learnt about the use of matrix chain multiplication and how its algorithm can be implemented in solving the problems in computer science. We have also discussed regarding the use of branch and bound algorithm. We have discussed very interesting and common problem of Travelling salesman which we have tried to discuss in reference to the dynamic programming. We hope that after reading this unit students will be well versed the objectives which we have discussed at the starting of the unit.

13.7 Glossary

Dynamic programming : Dynamic programming is a very general optimization technique that can be applied to problems that can be subdivided into similar subproblems of smaller size such that the solution to the larger problem can be obtained by combining the solutions to the subproblems. These "divide and conquer" methods are frequently used to solve alignment problems.

The Travelling Salesman Problem (TSP) is a problem in combinatorial optimization studied in operations research and theoretical computer science. Given a list of cities and their pairwise distances, the task is to find a shortest possible tour that visits each city exactly once

Branch and Bound (BB) is a general algorithm for finding optimal solutions of various optimization problems, especially in discrete and combinatorial optimization. It consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidates are discarded en masse, by using upper and lower estimated bounds of the quantity being optimized.

13.8 Answer to self learning exercises

1. The real problem in solving this is to define the stages, states, and decisions. One natural choice is to let stage t represent visiting t cities, and let the decision be where to go next. That leaves us with states. Imagine we chose the city we are in to be the state. We could not make the decision where to go next, for we do not know where we have gone before. Instead, the state has to include information about all the cities visited, plus the city we ended up in. So a state is represented by a pair (i,S) where S is the set of t cities already visited and i is the last city visited (so i must be in S). This turns out to be enough to get a recursion.

The stage 3 calculations are

$f_3(2,\{2,3,4\}) = 1334$

$f_3(3,\{2,3,4\}) = 1559$

$f_3(3,\{2,3,4\}) = 809$

For other stages, the recursion is

$$f_t(i,S) = \min_{j \neq 1 \text{ and } j \notin S} \{ c_{ij} + f_{t+1}(j, S \cup j)\}$$

Q.2.

Step 1 - Define the optimal structure for the problem.

As shown in figure 1 that we can move to 3 squares above our current, which means that we can also say that the maximum amount for any square is the maximum amount of the 3 squares below it plus the amount in the current square.

Therefore we can say that the best path to (r, c) is one of

- The best way through (r – 1, c - 1) — The bottom left square
- The best way through (r – 1, c) — The bottom square
- The best way through (r – 1, c + 1) — The bottom right square

If r = 1 then we know that we are at the start and there is no best way to that square. This gives us enough information to create a recursive solution.

Step 2 - Solve the problem recursively

Let C (r, c) be a function which returns the maximum amount of cash that a person can pick up by going to the specified row and column

Knowing the optimal structure, we go to 2nd step of creating a dynamic programming algorithm, creating a recursive function to solve the problem.

$$C(r,c) = \begin{cases} 0 & \text{if } c < 1 \text{ or } c > n \\ \text{cash}[r,c] & r = 1 \\ \max(C(r-1, c-1), C(r-1, c), C(r-1, c+1)) + \text{cash}[r][c] & \text{otherwise} \end{cases}$$

This algorithm works however it has a problem; many of the same subproblems are solved multiple times. Fortunately we can solve this problem by calculating upwards starting from row 1.

3 Saad Ahmad

Step 3 – Generating a solution bottom up

Instead of C being a function, we will use it as a 2D array to store the maximum amount of money that can be picked up by going to (r, c).

In our recursive algorithm we see that for all for values of r □ 2, we need the value of r – 1. This allows us to start at r = 2 and go up the money board.

13.9 Further Readings

1. Dynamic Programming by Richard Bellman

2. Dynamic Programming: Models and Applications by Eric V. Denardo

3. Dynamic Programming and Optimal Control, Vol. 1 (Optimization and Computation Series) by Dimitri P. Bertsekas

4. Schaum's Outline of Theory and Problems of Matrix Operations by Richard Bronson

5. The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics) by David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook

13.10 Unit End Questions

1. Describe the use of Dynamic Programming in Computer Science and other fields.

2. What do you mean by the Matrix Chain Multiplication. Explain through example.

3. Explain the concept of Dynamic Programmni using the Top - Down and Bottom - Up approach.

4. Explain the application areas of Dynamic Programming

5. What do you mean by branch and bound algorithm. What is the use of this algorithm.

6. Explain the travelling salesman problem.

---***---

UNIT XIV

PROBLEM CLASSES

STRCUTURE OF THE UNIT

14.0 Objectives
14.1 Introduction
14.2 NP Complete problem
14.2.1 Solving NP Complete Problem
14.3 Decision Problem
14.4 Cook Levin Theorem
14.5 Vertex Cover
14.6 Summary
14.7 Glossary
14.8 Further Readings
14.9 Answer to self learning exercises
14.10 Unit End Questions

14.0 Objectives:

After completing this unit you would be able to understand the following points

- Introduction to Problem classes NP
- NP Complexity
- NP Complete
- Definition of NP Complete and Problems
- Solving NP Complete Problems
- Decision Problems
- Cooks Levin Theorem
- Vertex Cover
- Vertex Cover in Hyper graphs

14.1 Introduction

In this unit we will discuss the Problems classes of NP, NP Hard and NP Complete. We will discuss in detail the implementation of NP problems, implementation and use of these problems in computer algorithms. We will also discuss in detail regarding the Cooks-Levin Theorem. Later in this unit Vertex Cover problems is discussed which includes the vertex cover in hyper graphs. A problem is assigned to the NP (nondeterministic polynomial time) class if it is solvable in polynomial time by a nondeterministic Turing machine.

A P-problem (whose solution time is bounded by a polynomial) is always also NP. If a problem is known to be NP, and a solution to the problem is known, then demonstrating the correctness of the solution can always be reduced to a single P (polynomial time) verification. If P and NP are not equivalent, then the solution of NP-problems requires (in the worst case) an exhaustive search.

A problem is said to be NP-hard if an algorithm for solving it can be translated into one for solving any other NP-problem. It is much easier to show that a problem is NP than to show that it is NP-hard. A problem which is both NP and NP-hard is called an NP-complete problem.

NP (complexity)

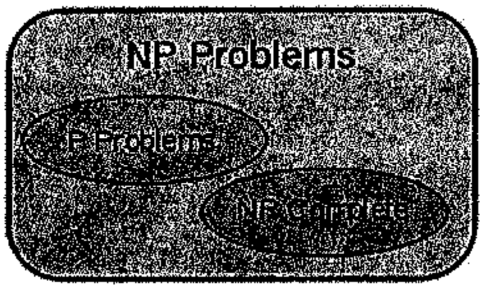


Diagram of complexity classes provided that **P “‘ NP.** The existence of problems outside both **P** and **NP-complete** in this case was established by Ladner.

In computational complexity theory, **NP** is one of the most fundamental complexity classes. The abbreviation **NP** refers to “nondeterministic polynomial time”.

Intuitively, **NP** is the set of all decision problems for which the ‘yes’-answers have simple proofs of the fact that the answer is indeed ‘yes’. More precisely, these proofs have to be verifiable in polynomial time by a deterministic Turing machine. In an equivalent formal definition, **NP** is the set of decision problems solvable in polynomial time by a non-deterministic Turing machine.

The complexity class **P** is contained in **NP**, but **NP** contains many important problems, the hardest of which are called **NP**-complete problems, for which no polynomial-time algorithms are known. The most important open question in complexity theory, the **P = NP** problem, asks whether such algorithms actually exist for **NP**-complete, and by corollary, all **NP** problems. It is widely believed that this is not the case.

The complexity class NP can be defined in terms of NTIME as follows:

$$\mathbf{NP} = \bigcup_{k\in\mathbb{N}} \mathbf{NTIME}(n^k)$$

Many natural computer science problems are covered by the class **NP**. In particular, the decision versions of many interesting search problems and optimization problems are contained in **NP**.

Verifier-based definition

In order to explain the verifier-based definition of **NP**, let us consider the subset sum problem: Assume that we are given some integers, such as {“7, “3, “2, 5, 8}, and we wish to know whether some of these integers sum up to zero. In this example, the answer is ‘yes’, since the subset of integers {-3, -2, 5} corresponds to the sum (-3) + (-2) + 5 = 0. The task of deciding whether such a subset with sum zero exists is called the subset sum problem.

As the number of integers that we feed into the algorithm becomes larger, the number of subsets grows exponentially, and in fact the subset sum problem is **NP**-complete. However, notice that, if we are given a particular subset (often called a certificate), we can easily check or verify whether the subset sum is zero, by just summing up the integers of the subset. So if the sum is indeed zero, that particular subset is the proof or witness for the fact that the answer is 'yes'. An algorithm that verifies whether a given subset has sum zero is called verifier. A problem is said to be in **NP** if and only if there exists a verifier for the problem that executes in polynomial time. In case of the subset sum problem, the verifier needs only polynomial time, for which reason the subset sum problem is in **NP**.

Note that the verifier-based definition of **NP** does not require an easy-to-verify certificate for the 'no'-answers. The class of problems with such certificates for the 'no'-answers is called **co-NP**. In fact, it is an open question whether all problems in **NP** also have certificates for the 'no'-answers and thus are in **co-NP**.

Some NP Problems are hard to solve because of the many important problems in this class, there have been extensive efforts to find polynomial-time algorithms for problems in **NP**. However, there remain a large number of problems in **NP** that defy such attempts, seeming to require superpolynomial time. Whether these problems really aren't decidable in polynomial time is one of the greatest open questions in computer science An important notion in this context is the set of NP-complete decision problems, which is a subset of **NP** and might be informally described as the "hardest" problems in **NP**. If there is a polynomial-time algorithm for even one of them, then there is a polynomial-time algorithm for all the problems in **NP**. Because of this, and because dedicated research has failed to find a polynomial algorithm for any **NP-complete** problem, once a problem has been proven to be **NP-complete** this is widely regarded as a sign that a polynomial algorithm for this problem is unlikely to exist.

14.2 NP-complete Problem

In computational complexity theory, the complexity class **NP-complete** (abbreviated **NP-C** or **NPC**, with **NP** standing for nondeterministic polynomial time) is a class of problems having two properties

- Any given solution to the problem can be **verified** quickly (in polynomial time); the set of problems with this property is called NP.
- If the problem can be **solved** quickly (in polynomial time), then so can every problem in NP.

Although any given solution to such a problem can be verified quickly, there is no known efficient way to locate a solution in the first place; indeed, the most notable characteristic of NP-complete problems is that no fast solution to them is known. That is, the time required to solve the problem using any currently known algorithm increases very quickly as the size of the problem grows. As a result, the time required to solve even moderately large versions of many of these problems easily reaches into the billions or trillions of years, using any amount of computing power available today. As a consequence, determining whether or not it is possible to solve these problems quickly is one of the principal unsolved problems in computer science today.

While a method for computing the solutions to NP-complete problems using a reasonable amount of time remains undiscovered, computer scientists and programmers still frequently encounter NP-complete problems. An expert programmer should be able to recognize an NP-complete problem so that he or she does not unknowingly waste time trying to solve a problem which so far has eluded generations of computer scientists. Instead, NP-complete problems are often addressed by using approximation algorithms.

NP-complete is a subset of NP, the set of all decision problems whose solutions can be verified in polynomial time; NP may be equivalently defined as the set of decision problems that can be solved in polynomial time on a nondeterministic Turing machine. A problem p in NP is also in NPC if and only if every other problem in NP can be transformed into p in polynomial time. NP-complete can also be used as an adjective: problems in the class NP-complete are known as NP-complete problems.

NP-complete problems are studied because the ability to quickly verify solutions to a problem (NP) seems to correlate with the ability to quickly solve that problem (P). It is not known whether every problem in NP can be quickly solved—this is called the P = NP problem. But if any single problem in,NP-complete can be solved quickly, then every problem in NP can also be quickly solved, because the definition of an NP-complete problem states that every problem in NP must be quickly reducible to every problem in NP-complete (that is, it can be reduced in polynomial time). Because of this, it is often said that the NP-complete problems are harder or more difficult than NP problems in general.

Formal definition of NP-completeness

A decision problem C is NP-complete if:

1. C is in NP, and
2. Every problem in NP is reducible to C in polynomial time.

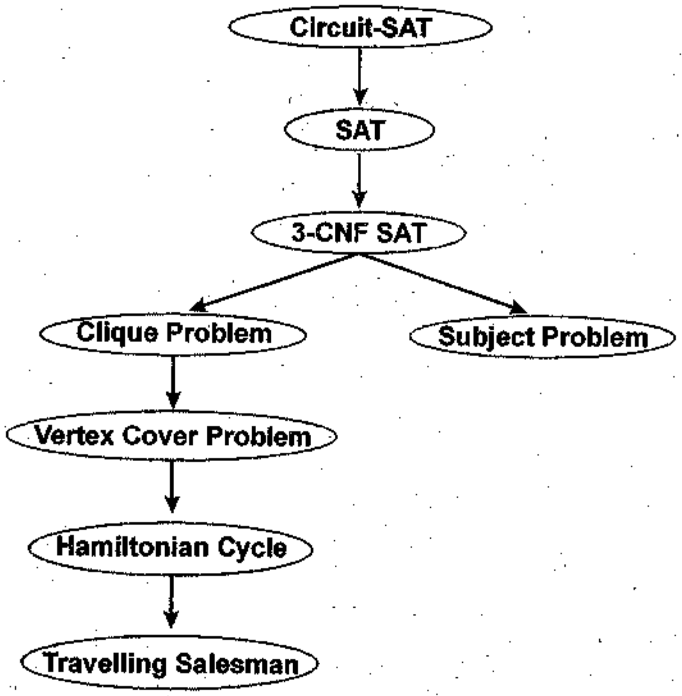
C can be shown to be in NP by demonstrating that a candidate solution to C can be verified in polynomial time.

A problem K is reducible to C if there is a polynomial-time many-one reduction, a deterministic algorithm which transforms any instance k/"/K into an instance c/"/C, such that the answer to c is yes if and only if the answer to k is yes. To prove that an NP problem C is in fact an NP-complete problem it is sufficient to show that an already known NP-complete problem reduces to C.

Note that a problem satisfying condition 2 is said to be NP-hard, whether or not it satisfies condition 1.

A consequence of this definition is that if we had a polynomial time algorithm (on a UTM, or any other Turing-equivalent abstract machine) for C, we could solve all problems in NP in polynomial time.

NP-complete problems



Some NP-complete problems, indicating the reductions typically used to prove their NP-completeness

An interesting example is the graph isomorphism problem, the graph theory problem of determining whether a graph isomorphism exists between two graphs. Two graphs are isomorphic if one can be transformed into the other simply by renaming vertices. Consider these two problems:

- Graph Isomorphism: Is graph G_1 isomorphic to graph G_2?
- Subgraph Isomorphism: Is graph G_1 isomorphic to a subgraph of graph G_2?

The Subgraph Isomorphism problem is NP-complete. The graph isomorphism problem is suspected to be neither in P nor NP-complete, though it is obviously in NP. This is an example of a problem that is thought to be **hard**, but isn't thought to be NP-complete.

The easiest way to prove that some new problem is NP-complete is first to prove that it is in NP, and then to reduce some known NP-complete problem to it. Therefore, it is useful to know a variety of NP-complete problems. The list below contains some well-known problems that are NP-complete when expressed as decision problems.

- Boolean satisfiability problem (Sat.)
- N-puzzle
- Knapsack problem
- Hamiltonian path problem

- Travelling salesman problem
- Subgraph isomorphism problem
- Subset sum problem
- Clique problem
- Vertex cover problem
- Independent set problem
- Dominating set problem
- Graph coloring problem

To the right is a diagram of some of the problems and the reductions typically used to prove their NP-completeness. In this diagram, an arrow from one problem to another indicates the direction of the reduction. Note that this diagram is misleading as a description of the mathematical relationship between these problems, as there exists a polynomial-time reduction between any two NP-complete problems; but it indicates where demonstrating this polynomial-time reduction has been easiest.

There is often only a small difference between a problem in P and an NP-complete problem. For example, the 3-satisfiability problem, a restriction of the boolean satisfiability problem, remains NP-complete, whereas the slightly more restricted 2-satisfiability problem is in P (specifically, NL-complete), and the slightly more general max. 2-sat. problem is again NP-complete. Determining whether a graph can be colored with 2 colors is in P, but with 3 colors is NP-complete, even when restricted to planar graphs. Determining if a graph is a cycle or is bipartite is very easy (in L), but finding a maximum bipartite or a maximum cycle subgraph is NP-complete. A solution of the knapsack problem within any fixed percentage of the optimal solution can be computed in polynomial time, but finding the optimal solution is NP-complete.

14.2.1 Solving NP-complete problems

At present, all known algorithms for NP-complete problems require time that is superpolynomial in the input size, and it is unknown whether there are any faster algorithms.

The following techniques can be applied to solve computational problems in general, and they often give rise to substantially faster algorithms:

- Approximation: Instead of searching for an optimal solution, search for an "almost" optimal one.
- Randomization: Use randomness to get a faster average running time, and allow the algorithm to fail with some small probability. See Monte Carlo method.
- Restriction: By restricting the structure of the input (e.g., to planar graphs), faster algorithms are usually possible.
- Parameterization: Often there are fast algorithms if certain parameters of the input are fixed.

- Heuristic: An algorithm that works "reasonably well" in many cases, but for which there is no proof that it is both always fast and always produces a good result. Metaheuristic approaches are often used.

One example of a heuristic algorithm is a suboptimal O(n log n) greedy coloring algorithm used for graph coloring during the register allocation phase of some compilers, a technique called graph-coloring global register allocation. Each vertex is a variable, edges are drawn between variables which are being used at the same time, and colors indicate the register assigned to each variable. Because most RISC machines have a fairly large number of general-purpose registers, even a heuristic approach is effective for this application.

Completeness under different types of reduction

In the definition of NP-complete given above, the term reduction was used in the technical meaning of a polynomial-time many-one reduction.

Another type of reduction is polynomial-time Turing reduction. A problem X is polynomial-time Turing-reducible to a problem Y if, given a subroutine that solves Y in polynomial time, one could write a program that calls this subroutine and solves X in polynomial time. This contrasts with many-one reducibility, which has the restriction that the program can only call the subroutine once, and the return value of the subroutine must be the return value of the program.

If one defines the analogue to NP-complete with Turing reductions instead of many-one reductions, the resulting set of problems won't be smaller than NP-complete; it is an open question whether it will be any larger. If the two concepts were the same, then it would follow that NP = co-NP. This holds because by their definition the classes of NP-complete and co-NP-complete problems under Turing reductions are the same and because these classes are both supersets of the same classes defined with many-one reductions. So if both definitions of NP-completeness are equal then there is a co-NP-complete problem (under both definitions) such as for example the complement of the boolean satisfiability problem that is also NP-complete (under both definitions). This implies that NP = co-NP as is shown in the proof in the co-NP article. Although whether NP/=/co-NP is an open question it is considered unlikely and therefore it is also unlikely that the two definitions of NP-completeness are equivalent.

Another type of reduction that is also often used to define NP-completeness is the logarithmic-space many-one reduction which is a many-one reduction that can be computed with only a logarithmic amount of space. Since every computation that can be done in logarithmic space can also be done in polynomial time it follows that if there is a logarithmic-space many-one reduction then there is also a polynomial-time many-one reduction. This type of reduction is more refined than the more usual polynomial-time many-one reductions and it allows us to distinguish more classes such as P-complete. Whether under these types of reductions the definition of NP-complete changes is still an open problem.

14.3 Decision problem

In computability theory and computational complexity theory, a **decision problem** is a question in some formal system with a yes-or-no answer, depending on the values of some input parameters. For example, the problem "given two numbers x and y, does x evenly divide y?" is a decision problem. The answer can be either 'yes' or 'no', and depends upon the values of x and y.

Decision problems are closely related to function problems, which can have answers that are more complex than a simple 'yes' or 'no'. A corresponding function problem is "given two numbers x and y, what is x divided by y?". They are also related to optimization problems, which are concerned with finding the best answer to a particular problem.

A method for solving a decision problem given in the form of an algorithm is called a **decision procedure** for that problem. A decision procedure for the decision problem "given two numbers x and y, does x evenly divide y?" would give the steps for determining whether x evenly divides y, given x and y. One such algorithm is long division, taught to many school children. If the remainder is zero the answer produced is 'yes', otherwise it is 'no'. A decision problem which can be solved by an algorithm, such as this example, is called **decidable**.

The field of computational complexity categorizes decidable decision problems by how difficult they are to solve. "Difficult", in this sense, is described in terms of the computational resources needed by the most efficient algorithm for a certain problem. The field of recursion theory, meanwhile, categorizes undecidable decision problems by Turing degree, which is a measure of the noncomputability inherent in any solution.

Research in computability theory has typically focused on decision problems. As explained in the section Equivalence with function problems below, there is no loss of generality.

Definition

A decision problem is any arbitrary yes-or-no question on an infinite set of inputs. Because of this, it is traditional to define the decision problem equivalently as: the set of inputs for which the problem returns yes.

These inputs can be natural numbers, but also other values of some other kind, such as strings of a formal language. Using some encoding, such as Gödel numbers, the strings can be encoded as natural numbers. Thus, a decision problem informally phrased in terms of a formal language is also equivalent to a set of natural numbers. To keep the formal definition simple, it is phrased in terms of subsets of the natural numbers.

Formally, a **decision problem** is a subset of the natural numbers. The corresponding informal problem is that of deciding whether a given number is in the set.

Examples

A classic example of a decidable decision problem is the set of prime numbers. It is possible to effectively decide whether a given natural number is prime by testing every possible nontrivial factor. Although much more efficient methods of primality testing are known, the existence of any effective method is enough to establish decidability.

Decidability

A decision problem A is called **decidable** or **effectively solvable** if A is a recursive set. A problem is called **partially decidable, semidecidable, solvable,** or **provable** if A is a recursively enumerable set. Partially decidable problems and any other problems that are not decidable are called **undecidable.**

The halting problem is an important undecidable decision problem.

Complete problems

Decision problems can be ordered according to many-one reducibility and related feasible reductions such as Polynomial-time reductions. A decision problem P is said to be complete for a set of decision problems S if P is a member of S and every problem in S can be reduced to P. Complete decision problems are used in computational complexity to characterize complexity classes of decision problems. For example, the Boolean satisfiability problem is complete for the class NP of decision problems under polynomial-time reducibility.

Practical decision

Having practical decision procedures for classes of logical formulas is of considerable interest for program verification and circuit verification. Pure Boolean logical formulas are usually decided using SAT-solving techniques based on the DPLL algorithm. Conjunctive formulas over linear real or rational arithmetic can be decided using the Simplex algorithm, formulas in linear integer arithmetic (Presburger arithmetic) can be decided using Cooper's algorithm or William Pugh's Omega test. Formulas with negations, conjunctions and disjunctions combine the difficulties of satisfiability testing with that of decision of conjunctions; they are generally decided nowadays using SMT-solving technique, which combine SAT-solving with decision procedures for conjunctions and propagation techniques. Real polynomial arithmetic, also known as the theory of real closed fields, is decidable, for instance using the Cylindrical algebraic decomposition; unfortunately the complexity of that algorithm is excessive for most practical uses.

14.4 Cook–Levin theorem

In computational complexity theory, the **Cook–Levin theorem**, also known as **Cook's theorem**, states that the Boolean satisfiability problem is NP-complete. That is, any problem in NP can be reduced in polynomial time by a deterministic Turing machine to a problem of determining whether a Boolean formula is satisfiable.

An important consequence of the theorem is this: if there were a deterministic polynomial time algorithm for solving Boolean satisfiability, then there would exist a deterministic polynomial time algorithm for solving all problems in NP. Crucially, the same follows for any NP complete problem as these are also in NP.

The question of whether such an algorithm exists is called the P=NP problem and it is widely considered the most important unsolved problem in theoretical computer science.

Contributions

The concept of NP-completeness was developed in the late 1960s and early 70s in parallel by researchers in the US and the USSR. In the US in 1971, Stephen Cook published his paper "The complexity of theorem proving procedures"in conference proceedings of the newly-founded ACM Symposium on Theory of Computing. Richard Karp's subsequent paper, "Reducibility among combinatorial problems",generated renewed interest in Cook's paper by providing a list of 21 NP-complete problems. Cook and Karp received a Turing Award for this work.

The theoretical interest in NP-completeness was also enhanced by the work of Theodore P. Baker, John Gill, and Robert Solovay who showed that solving NP-problems in Oracle machine models requires exponential time.

In the USSR, a result equivalent to Baker, Gill, and Solovay's was published in 1969 by M. Dekhtiar. Later Levin's paper, "Universal search problems was published in 1973, although it was mentioned in talks and submitted for publication a few years earlier.

Levin's approach was slightly different from Cook's and Karp's in that he considered search problems, which require finding solutions rather than simply determining existence. He provided 6 such NP-complete search problems, or universal problems, and additionally found that each has an algorithm which solves it in optimal time.

Definitions

A decision problem is in NP iff it can be solved by a non-deterministic algorithm in polynomial time.

An instance of the Boolean satisfiability problem is a Boolean expression that combines Boolean variables using Boolean operators.

An expression is satisfiable iff there is some assignment of truth values to the variables that makes the entire expression true.

Proof

This proof is based on the one given by Garey and Johnson.

There are two parts to proving that the Boolean satisfiability problem (SAT) is NP-complete. One is to show that SAT is an NP problem. The other is to show that every NP problem can be reduced to an instance of a SAT problem by a polynomial-time many-one reduction.

SAT is in NP because any assignment of Boolean values to Boolean variables that is claimed to satisfy the given expression can be verified in polynomial time by a deterministic Turing machine.

Now suppose that a given problem in NP can be solved by the nondeterministic Turing machine M = (Q, Ó, s, F, ä), where Q is the set of states, Ó is the alphabet of tape symbols, s “ Q is the initial state, F “ Q is the set of accepting states, and ä: Q × Ó Q × Ó × {“1, +1} is the set of transitions. Suppose further that M accepts or rejects an instance of the problem in time p(n) where n is the size of the instance and p is a polynomial function.

For each input I we specify a Boolean expression which is satisfiable if and only if the machine M accepts I.

The Boolean expression uses the variables set out in the following table. Here, q “ Q, “p(n) d” i d” p(n), j Ó, and 0 k d” p(n).

Variables	Intended interpretation	How many?
T_{ijk}	True if tape cell *i* contains symbol *j* at step *k* of the computation.	$O(p(n)^2)$
H_{ik}	True if the *M*'s read/write head is at tape cell *i* at step *k* of the computation.	$O(p(n)^2)$
Q_{qk}	True if *M* is in state *q* at step *k* of the computation.	$O(p(n))$

Define the Boolean expression B to be the conjunction of the clauses in the following table, for all "p(n) d" i d" p(n) and 0 d" k d" p(n):

Clauses	Conditions	Interpretation	How many?
T_{ij0}	Tape cell *i* of the input *I* contains symbol *j*.	Initial contents of the tape.	$O(p(n))$
Q_{s0}		Initial state of *M*	$O(1)$
H_{00}		Initial position of read/write head.	$O(1)$
$T_{ijk} \to \neg T_{ij'k}$	$j \neq j'$	One symbol per tape cell.	$O(p(n)^2)$
$T_{ijk} = T_{ij(k+1)} \vee H_{ik}$		Tape remains unchanged unless written.	$O(p(n)^2)$
$Q_{qk} \to \neg Q_{q'k}$	$q \neq q'$	Only one state at a time.	$O(p(n))$
$H_{ik} \to \neg H_{i'k}$	$i \neq i'$	Only one head position at a time.	$O(p(n)^2)$
$(H_{ik} \wedge Q_{qk} \wedge T_{i\sigma k}) \to (H_{(i+d)(k+1)} \wedge Q_{q'(k+1)} \wedge T_{i\sigma'(k+1)})$	$(q, \sigma, q', \sigma', d) \in \delta$	Possible transitions at computation step *k* when head is at position *i*.	$O(p(n)^2)$
The disjunction of the clauses $Q_{f\,p(n)}$	$f \in F$	Must finish in an accepting state.	$O(1)$

If there is an accepting coputation for M on input I that follows the steps indicated by the assignments to the variables.There are $O(p(n)^2)$ Boolean variables, each encodeable in space O(log p(n)). The If there is an accepting computation for M on input I, then B is satisfiable by assigning T_{ijk}, H_{ik} and Q_{ik} their intended interpretations. On the other hand, if B is satisfiable, then there is annumber of clauses is $O(p(n)^2)$ so the size of B is $O(\log(p(n))p(n)^2)$. Thus the transformation is certainly a polynomial-time many-one reduction, as required.

Consequences

The proof shows that any problem in NP can be reduced in polynomial time (in fact, logarithmic space suffices) to an instance of the Boolean satisfiability problem. This means that if the Boolean satisfiability problem could be solved in polynomial time by a deterministic Turing machine, then all problems in NP could be solved in polynomial time, and so the complexity class NP would be equal to the complexity class P.

The significance of NP-completeness was made clear by the publication in 1972 of Richard Karp's landmark paper, "Reducibility among combinatorial problems", in which he showed that 21 diverse combinatorial and graph theoretical problems, each infamous for its intractability, are NP-complete.

Karp showed each of his problems to be NP-complete by reducing another problem (already shown to be NP-complete) to that problem. For example, he showed the problem 3SAT (the Boolean satisfiability problem for expressions in conjunctive normal form with exactly three variables or negations of variables per clause) to be NP-complete by showing how to reduce (in polynomial time) any instance of SAT to an equivalent instance of 3SAT. (First you modify the proof of the Cook-Levin theorem, so that the resulting formula is in conjunctive normal form,then you introduce new variables to split clauses with more than 3 atoms. For example, the clause (A (" B (" C (" D) can be replaced by the conjunction of clauses (A (" B (" Z) "' (¬Z (" C (" D), where Z is a new variable which will not be used anywhere else in the expression. Clauses with fewer than 3 atoms can be padded; for example, A can be replaced by (A (" A (" A), and (A (" B) can be replaced by (A (" B (" B).

As of 2008[update], although many practical instances of SAT can be solved by heuristic methods, the question of whether there is a deterministic polynomial-time algorithm for SAT (and consequently all other NP-complete problems) is still a famous unsolved problem, despite decades of intense effort by complexity theorists, mathematical logicians, and others. For more details, see the article P=NP problem.

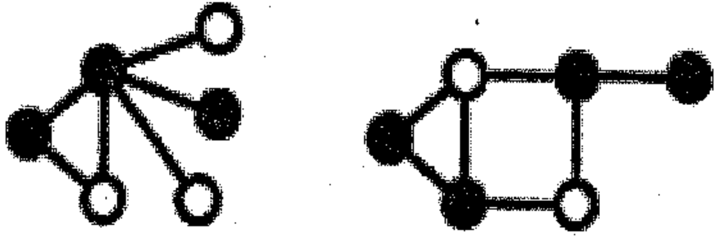
14.5 Vertex cover

In the mathematical discipline of graph theory, a **vertex cover** of a graph is a set of vertices such that each edge of the graph is incident to at least one vertex of the set. The problem of finding a **minimum vertex cover** is a classical optimization problem in computer science and is a typical example of an NP-hard optimization problem that has an approximation algorithm. Its decision version, the **vertex cover problem** was one of Karp's 21 NP-complete problems and is therefore a classical NP-complete problem in computational complexity theory. Furthermore, the vertex cover problem is fixed-parameter tractable and a central problem in parameterized complexity theory.

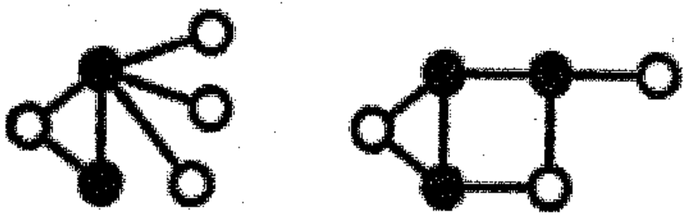
The minimum vertex cover problem can be formulated as a half-integral linear program whose dual linear program is the maximum matching problem.

Definition

Formally, a vertex cover of a graph G is a set of vertices C such that each edge of G is incident to at least one vertex in C. The set C is said to cover the edges of G. The following figure shows examples of vertex covers in two graphs.



A **minimum vertex cover** is a vertex cover of a smallest possible size The **vertex cover number** ô is the size of a minimum vertex cover. The following figure shows examples of minimum vertex covers in two graphs.



Examples

- The set of all vertices is a vertex cover.
- A set of vertices is a vertex cover if and only if its complement is an independent set.
- The endpoints of a maximal matching form a vertex cover.
- The complete bipartite graph $K_{m,n}$ has vertex cover number min{m,n}.

Properties

- The number of vertices of a graph is equal to its vertex cover number plus the size of a maximum independent set.

Computational problem

The **minimum vertex cover problem** is the optimization problem of finding a smallest vertex cover in a given graph.

INSTANCE: Graph G
OUTPUT: Smallest number k such that there is a vertex cover C for G of size k.

If the problem is stated as a decision problem, it is called the **vertex cover problem:**

INSTANCE: Graph G and positive integer k.
QUESTION: Is there a vertex cover C for G of size at most k?

The vertex cover problem is an NP-complete problem: it was one of Karp's 21 NP-complete problems. It is often used in computational complexity theory to prove that another problem is NP-hard.

ILP formulation

Assume that every vertex has an associated cost of . The (weighted) minimum vertex cover problem can be formulated as the following integer linear program

minimize	$\sum_{v\in V} c(v)x_v$		(minimize the total cost)
subject to	$x_u + x_v \geq 1$	for all $\{u,v\} \in E$	(cover every edge of the graph)
	$x_v \in \{0,1\}$	for all $v \in V$.	(every vertex is either in the vertex cover or not)

This ILP belongs to the more general class of ILPs for covering problems. The integrality gap of this ILP is 2, so its relaxation gives a factor-2 approximation algorithm for the minimum vertex cover problem. Furthermore, the linear programming relaxation of that ILP is half-integral, that is, there exists an optimal solution with for all .

Exact evaluation

The decision variant of the vertex cover problem is NP-complete, which means it is unlikely that there is an efficient algorithm to solve it exactly. NP-completeness can be proven by reduction from 3-satisfiability or, as Karp did, by reduction from the clique problem. Vertex cover remains NP-complete even in cubic graphs and even in planar graphs of degree at most 3.

For bipartite graphs, the equivalence between vertex cover and maximum matching described by Kőnig's theorem allows the bipartite vertex cover problem to be solved in polynomial time.

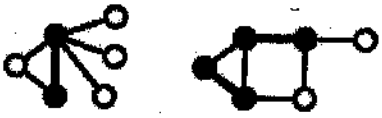
Fixed-parameter tractability

A brute force algorithm can solve the problem in time $2^{O(k)}n^{O(1)}$. Vertex cover is therefore fixed-parameter tractable, and if we are only interested in small k, we can solve the problem in polynomial time. One algorithmic technique that works here is called bounded search tree algorithm, and its idea is to repeatedly choose some vertex and recursively branch, with two cases at each step: place either the current vertex or all its neighbours into the vertex cover. Under reasonable complexity-theoretic assumptions, namely the exponential time hypothesis, this running time cannot be improved to $2^{o(k)}n^{O(1)}$.

For planar graphs, however, a vertex cover of size k can be found in time , i.e., the problem is subexponential fixed-parameter tractable. This algorithm is again optimal, in the sense that, under the exponential time hypothesis, no algorithm can solve vertex cover on planar graphs in time .

Approximate evaluation

One can find a factor-2 approximation by repeatedly taking both endpoints of an edge into the vertex cover, then removing them from the graph. Put otherwise, we find a maximal matching M with a greedy algorithm and construct a vertex cover C that consists of all endpoints of the edges in M. In the following figure, a maximal matching M is marked with red, and the vertex cover C is marked with blue.



The set C constructed this way is a vertex cover: suppose that an edge e is not covered by C; then M *” {e} is a matching and e “ M, which is a contradiction with the assumption that M is maximal. Furthermore, if e = {u,v} “ M, then any vertex cover – including an optimal vertex cover – must contain u or v (or both); otherwise the

That is, an optimal cover contains at least one endpoint of each edge in M; in total, the set C is at most 2 times as large as the optimal vertex cover.

This simple algorithm was discovered independently by Fanica Gavril and Mihalis Yannakakis.

More involved techniques show that there are approximation algorithms with a slightly better approximation factor. For example, an approximation algorithm with an approximation factor of is known .

Inapproximability

No better constant-factor approximation algorithm than the above one is known. The minimum vertex cover problem is APX-complete, that is, it cannot be approximated arbitrarily well unless **P=NP**. Dinur and Safra proved, using techniques from the PCP theorem, that minimum vertex cover cannot be approximated within a factor of 1.3606 for any sufficiently large vertex degree unless **P=NP**. Moreover, if the unique games conjecture is true then minimum vertex cover cannot be approximated within any constant factor better than 2.

Although finding the minimum-size vertex cover is equivalent to finding the maximum-size independent set, as described above, the two problems are not equivalent in an approximation-preserving way: The Independent Set problem has no constant-factor approximation unless P=NP.

Vertex cover in hypergraphs

Vertex cover has a natural generalization to hypergraphs which is often just called **vertex cover for hypergraphs** but which is also known under the names **hitting set** and, in a more combinatorial context, **transversal.** Even more, the notions of hitting set and set cover are equivalent.

Formally, let H=(V,E) be a hypergraph with vertex set V and hyperedge set E. Then a set S “ V is called hitting set of H if, for all edges e “ E, it holds S)” e `” “. The computational problems minimum hitting set and hitting set are defined as in the case of graphs.

Note that we get back the case of vertex covers for simple graphs if the maximum size of the hyperedges is 2. If that size is restricted to d, the problem of finding a minimum d-hitting set permits a d-approximation algorithm.

Fixed-parameter tractability

For the hitting set problem, different parameterizations make sense. The hitting set problem is W[2]-complete for the parameter OPT, that is, it is unlikely that there is an algorithm that runs in time $f(OPT)n^{O(1)}$ where OPT is the cardinality of the smallest hitting set. The hitting set problem is fixed-parameter tractable for the parameter OPT + d, where d is the size | e | of the largest edge of the hypergraph. More specifically, there is an algorithm for hitting set that runs in time $d^{OPT}n^{O(1)}$.

Hitting set and set cover

The Hitting Set Problem is equivalent to the Set cover problem: An instance of set cover can be viewed as an arbitrary bipartite graph, with sets represented by vertices on the left, the universe represented by vertices on the right, and edges representing the inclusion of elements in sets. The task is then to find a minimum cardinality subset of left-vertices which covers all of the right-vertices. In the Hitting set problem, the objective is to cover the left-vertices using a minimum subset of the right vertices. Converting from one problem to the other is therefore achieved by interchanging the two sets of vertices.

Self learning exercises

1. Prove The decision version of the traveling salesman problem is in **NP**.
2. Compare and give example of P and NP problems.

14.6 Summary:

In this unit we have discussed the various concepts of Problem classes NP, NP hard and NP complete. A problem is said to be NP-hard if an algorithm for solving it can be translated into one for solving any other NP-problem. It is much easier to show that a problem is NP than to show that it is NP-hard. A problem which is both NP and NP-hard is called an NP-complete problem. A method for solving a decision problem given in the form of an algorithm is called a decision procedure for that problem. We have discussed in details regarding the decision problems. We have also discussed in details regarding the Cooks – Levis Theorem. Apart from this we also have defined the vertex cover problem and vertex hyper graph.

14.7 Glossary

P Problem:A problem is assigned to the P (polynomial-time) class if there exists at least one algorithm to solve that problem, such that the number of steps of the algorithm is bounded by a polynomial in n, where n is the length of the input.

NP-Complete Problem A problem which is both NP (verifiable in nondeterministic polynomial time) and NP-hard (any NP-problem can be translated into this problem). Examples of NP-hard problems include the Hamiltonian cycle and traveling salesman problems.

NP-Hard Problem A problem is NP-hard if an algorithm for solving it can be translated into one for solving any NP-problem (nondeterministic polynomial time) problem. NP-hard therefore means "at least as hard as any NP-problem," although it might, in fact, be harder

NP-Problem A problem is assigned to the NP (nondeterministic polynomial time) class if it is solvable in polynomial time by a nondeterministic Turing machine.

Decision problem: A decision problem is any arbitrary yes-or-no question on an infinite set of inputs. Because of this, it is traditional to define the decision problem equivalently as: the set of inputs for which the problem returns yes.

14.8 Further Readings

1. Computers and Intractability: A Guide to the Theory of NP-Completeness by M. R. Garey and D. S. Johnson

2. Graph Algorithms and Np-Completeness by Kurt Mehlhorn

3. Handbook of NP-Completeness: Theory and Applications by Teofilo F. Gonzalez

4. Statistics for Applied Problem Solving and Decision Making by Richard J. Larsen, Morris L. Marx, and Bruce Cooil

5. Experimental Analysis of Approximation Algorithms for the Vertex Cover and Set Covering Problems by F.C. Gomes, C.N. Meneses, P.M. Pardalos, and G. Viana

14.9 Answers to self learning exercises

1. Given an input matrix of distances between N cities, the problem is to determine if there is a route visiting all cities with total distance less than k.

A proof certificate can simply be a list of the cities. Then verification can clearly be done in polynomial time by a deterministic Turing machine. It simply adds the matrix entries corresponding to the paths between the cities.

A nondeterministic Turing machine can find such a route as follows:

At each city it visits it "guesses" the next city to visit, until it has visited every vertex. If it gets stuck, it stops immediately.

At the end it verifies that the route it has taken has cost less than k in O(n) time.

One can think of each guess as "forking" a new copy of the Turing machine to follow each of the possible paths forward, and if at least one machine finds a route of distance less than k, that machine accepts the input. (Equivalently, this can be thought of as a single Turing machine that always guesses correctly)

Binary search on the range of possible distances can convert the decision version of Traveling Salesman to the optimization version, by calling the decision version repeatedly (a polynomial number of times).

2. NP-complete
   - Satisfiability
   - Graph coloring
   - Hamiltonian Path (visit all nodes)
   - Subset Sum (subset of numbers sum to z)
   - Traveling Salesperson Problem (TSP)

P: Polynomial time
   - Median
   - Prime
   - String search
   - Connectivity (go s to t)
   - Eulerian Path (visit all edges)
   - Coin sum (subset of coins sum to z?)

14.10 Unit End Questions

1. What do you mean by Problem NP Classes.

2. Explain in detail concept of NP Hard and NP Complete.

3. Explain in how decision problems can help you in solving problem.

4. What are the contribution of Cook - Levin Theorem in solving NP problems.

5. What do you mean by vertex cover.

6. Explain the use of vertex cover in hyper graph.

---***---