BCA-12

Vardhaman Mahaveer Open University, Kota

Data Structure and Alogrithms

Course Development Committee

Chairman
Prof. (Dr.) Naresh Dadhich
Vice-Chancellor
Vardhaman Mahaveer Open University, Kota

Co-ordinator/Convener and Members

Convener:

Dr. Anuradha Sharma
Assistant Professor,
Department of Botany, Vardhaman Mahaveer Open University, Kota

Members:

1. **Prof. (Dr.) D.S.Chauhan**
   Department of Mathematics
   University of Rajasthan, Jaipur
2. **Prof. (Dr.) M.C.Govil**
   Govt. Engineering College,
   Ajmer
3. **Prof. (Dr.) A.K.Nagawat**
   University of Rajasthan,
   Jaipur
4. **Dr. (Mrs.) Madhavi Sinha**
   Birla Institute of Technology,
   Jaipur
5. **Dr. Rajeev Srivatava**
   LBS College,
   Jaipur

Editing and Course Writing

Editor:

Dr. (Mrs.) Madhavi Sinha
Associate Professor, Department of Computer Science
Birla Institute of Technology & Science-MESRA, Jaipur

1. **Mrs. Sudha Morwal** (Unit No. 1, 2, 3)
   Department of Computer Science
   Banasthali Vidyapeeth, Newai
2. **Mrs. Manisha Sharma** (Unit No. 4, 5, 6)
   Department of Computer Science
   Banasthali Vidyapeeth, Newai
3. **Sh. Rajesh Dadhich** (Unit No. 7)
   Department of Computer Science
   Govt. Polytechical College, Kota
4. **Sh. Sanjay Pahuja** (Unit No. 8, 9)
   Department of Computer Science
   Govt. Polytechical College, Kota
5. **Mrs. Rekha Jain** (Unit No. 10, 11, 12)
   Department of Computer Science
   Banasthali Vidyapeeth, Newai
6. **Dr. Sunita Chaudhary** (Unit No. 13, 14, 15)
   Department of Computer Science
   Banasthali Vidyapeeth, Newai

Academic and Administrative Management

Prof. (Dr.) Naresh Dadhich
Vice-Chancellor
Vardhaman Mahveer Open University, Kota

Prof. B.K. Sharma
Director (Academic)
Vardhaman Mahveer Open University, Kota

Mr. Yogendra Goyal
Incharge
Material Production and Distribution Department

Course Material Production

Mr. Yogendra Goyal
Assistant Production Officer
Vardhaman Mahaveer Open University, Kota

Production : March 2012 ISBN No. : 978-81-8496-311-3

Printed and published on behalf of Registrar, VM Open University Kota.
Printed by The Diamond Printing Press, Jaipur, 1000 Copies

Vardhaman Mahaveer Open University, Kota

Data Structure and Alogrithms

Preface

In computer science, a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, B-trees are particularly well-suited for implementation of database, while compiler implementations usually use hash tables to look up identifiers.

Data structures are used in almost every program or software system. Data structures provide a means to mange huge amount of data efficiently. Such as large databases and internet indexing services. Usually, efficient data structures are a key to designing efficient algorithms. Some formal design methods and programming factor in software design.

In this book the fundamental concept of data structure is described first and then different data structures are explained and their implementation is also incorporated. In the last units the efficiency concept is explained and algorithmic complexity of some searching sorting algorithms are discussed.

Unit - 1 : Introduction to Data Structure

Structure of Unit:

1.0 Objectives

This chapter covers

- Introduction of the terms that are used in this unit such as data, data type, array structure, variable etc.
- Concept of related term such as, information, knowledge etc.
- Definition of data structure and its need will be discussed.
- Brief exposure to elementary data structure such as array, structure.
- Introduction to user defined data types and pointer to structure will also be included

1.1 Introduction

The computer manipulates information. The study of computer science include organization, manipulation and utilization of information. In other word we can also state that computer science is study of data, its representation and transformation.

Data may be organized in various ways logical or mathematical model of particular organization is called data structure

1.2 Basic Terminology

Before going into detail you should be familiar with some terms related to data organization which are given below :-

1.2.1 Data

Data refers to value or set of values. e.g. Marks obtained by the students.

1.2.2 Data type

data type is a classification identifying one of various types of data, such as floating-point, integer, or Boolean, that determines the possible values for that type; the operations that can be done on values of that type; and the way values of that type can be stored

Primitive data type: These are basic data types that are provided by the programming language with built-in support. These data types are *native* to the language. This data type is supported by machine directly

1.2.3 Variable

variable is a symbolic name given to some known or unknown quantity or information, for the purpose of allowing the name to be used independently of the information it represents. A variable name in computer source code is usually associated with a data storage location and thus also its contents, and these may change during the course of program execution.

1.2.4 Record

Collection of related data items is known as record. The elements of records are usually called *fields* or *members*.

Records are distinguished from arrays by the fact that their number of fields is typically fixed, each field has a name, and that each field may have a different type.

1.2.5 Program

A sequence of instructions that a computer can interpret and execute.

1.3 Concept of Data and Record

Data are raw facts where as information is processed data. Data is processed by applying certain rules. Data is lowest level of abstraction, information is the next level and finally knowledge is highest level among all three. Data carries no meaning so we can say data can't use in decision making whereas information can interpreted and carries meaning therefore information can be used for decision making .

1.4 Definition and Need of Data Structure

Computer can store and process vast amount of data. We need a way to store collection of data that may grow and shrink dynamically over time and to organize the information so that we can access it using efficient algorithms. Formal data structure enable programmer to mentally structure large amount of data into conceptually manageable relationship.

Data structure is a particular way of storing and organizing huge amount of data so that it can be used efficiently

1.4.1 Need of data structure

- It give different level of organization data.
- It tells how data can be stored and accessed in its elementary level.
- Provide operation on group of data, such as adding an item, looking up highest priority item.
- Provide a means to manage huge amount of data efficiently.
- Provide fast searching and sorting of data.

1.4.2 Selecting a data structure

Selection of suitable data structure involve following steps –

- Analyze the problem to determine the resource constraints a solution must meet.
- Determine basic operation that must be supported. Quantify resource constraint for each operation
- Select the data structure that best meets these requirements.

  Each data structure has cost and benefits. Rarely is one data structure better than other in all situations. A data structure require

- Space for each item it stores
- Time to perform each basic operation
- Programming effort.

Each problem has constraints on available time and space. Best data structure for the task requires careful analysis of problem characteristics.

1.4.3 Type of data structure

Static data structure A data structure whose organizational characteristics are invariant throughout its lifetime. Such structures are well supported by high-level languages and familiar examples are arrays and records. The prime features of static structures are

(a) none of the structural information need be stored explicitly within the elements – it is often held in a distinct logical/physical header;

(b) the elements of an allocated structure are physically contiguous, held in a single segment of memory;

(c) all descriptive information, other than the physical location of the allocated structure, is determined by the structure definition;

(d) relationships between elements do not change during the lifetime of the structure.

Dynamic data structure A data structure whose organizational characteristics may change during its lifetime. The adaptability afforded by such structures, e.g. linked lists, is often at the expense of decreased efficiency in accessing elements of the structure. Two main features distinguish dynamic

structures from static data structures. Firstly, it is no longer possible to infer all structural information from a header; each data element will have to contain information relating it logically to other elements of the structure. Secondly, using a single block of contiguous storage is often not appropriate, and hence it is necessary to provide some storage management scheme at run-time.

Linear data structure

In Linear data structure processing of data items is possible linear that is one by one sequentially. Example of linear data structure is array, linked list, stack and queue.

Non-Linear data structure

In non-linear data structure processing of data items is not in linear fashion. Example of non-linear data structure is graph and tree.

1.5 Array

An array is collection of homogeneous data elements referred by single name and each individual elements of an array is referred by subscript variable, formed by affixing to the array name a subscript or index enclosed in brackets.

If single subscript is required to refer to element of array then array is called one dimensional array. The array whose elements are referred by two or more subscripts are called multi-dimensional array.

Since array hold homogeneous data elements therefore it is a homogeneous structure. It is also called random access structure, since all elements can be selected at random and are equally accessible.

The array is a powerful tool that is widely used in computing. The power is largely derived from the fact that it provide us with a simple and efficient way of referring to and performing computation on collection of data that share common attribute.

A one dimensional array is used when it is necessary to keep a large number of items in memory and reference all the items in a uniform manner. All element of an array have same fixed predetermined size. One method to implement an array of varying size element s it to reserve a contiguous set of memory location, which hold an address.

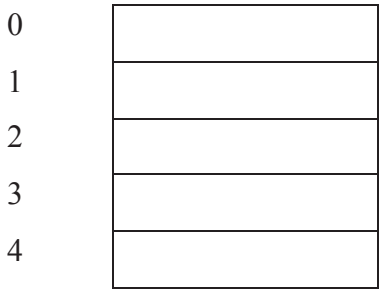
The array may be categorized into –

- One dimensional array
- Two dimensional array
- Multidimensional array

1.5.1 One dimensional array

One dimensional array A of 5 elements can be represented as

A



A one dimensional array is a list of finite homogeneous data elements, such that:

- The elements are referred by index
- The elements are stored in consecutive memory locations
- The number n is length or size of the array

1.5.2 Operations on Linear array

Traversal : This is accessing and processing each elements of list exactly one.

Algorithm: Traverse(A,N)

Where A is list, N is size of list.

1. {Initialization] I = 1
2. Repeat while I < = N
3. Process I^{th} element
4. [Increment] I = I + 1

   [End of loop]
5. Exit

Searching : Information retrieval is common activity in each application. For that searching for required element is needed or establish its absence.

Two type of searching is possible in array.

Linear Search: Searching of an element in list, one by one is called linear search.

In this we simply compare the element to be searched with each element of list one by one until required item found or end of list encountered.

- This searching is simplest for unordered list
- It is most natural way of searching

Algorithm : Search(A, N, X)

Where A is list, N is size of list, X is element to be searched.

1. {Initialization] I = 1
2. Repeat while I < = N
3. If A[I] = X then
   Break searching with found
   Else
   [Increment] I = I + 1
   [End of loop]
4. If I > U
5. Display Not Found
6. Exit

Binary Search: Let list is sorted in ascending order. Binary searching can be used which reduces the worst case time required by an algorithm. This searching is used in word search in dictionary. It divide

list in two parts and check middle element is required element if yes then terminate searching otherwise check required element exists in first part or second part and make adjustment in the list to be considered next and repeat the process.

Algorithm Search(A, N, X)

Where A is list, N is size of list, X is element to be searched.

1. {Initialization] Beg = 1, End = N,
7. Repeat while Begd" End
8. Mid = Int[(Beg + End)/2]
9. If A[Mid] = X then
   Break searching with found
   Else
   If X < A[Mid]
   End = Mid – 1
   Else
   Beg = Mid + 1
   [End of loop]
10. If Beg > End
11. Display Not Found
12. Exit

Insertion in array at given position: Let A is list of N numbers and X is new element to be inserted into A at position P, such at no data lost . To create space at Pth position for X program have to shift elements of array from N to P to next subsequent memory location then after shifting Insert new element at p^{th} position.

Algorithm Insert (A, N, X, P)

Where A is list, N is size of list, X is element to be inserted and P is position.

1. [Shifting] For I = N To P Step by -1 Do
2. A[I + 1] = A[I]
3. [End of Loop]
4. [Insertion] A[P] = X
5. Exit

Deletion of P^{th} element of an array : Let A is list of N numbers P is the position of an element to be deleted. To delete program have to shift elements of array from P+1 to N to previous memory location. When First time P+1 th element overwrite pth element, required work done.

Algorithm delete (A, N, P)

Where A is list, N is size of list, and P is position.

1. [Shifting] For I = P+1 To N Step by 1 Do
2. A[I - 1] = A[I]
3. [End of Loop]

1.5.3 Multidimensional Array

In one dimensional array elements are accessed by single subscript that's why it is known as linear array. Most programming language also allow multidimensional array in which elements are accessed by multiple subscripts. Array in which elements accessed by two subscript and three subscripts are known as two dimensional and three dimensional array respectively.

In two dimensional array(also called matrix in mathematics) elements are arranged in number of rows and columns. Number of rows and columns of matrix represent it's order. For example a 3X3 order matrix is given below

		Columns		
Rows		0	1	2
	0	A[0,0]	A[0,1]	A[0,2]
	1	A[1,0]	A[1,1]	A[1,2]
	2	A[2,0]	A[2,1]	A[2,2]

It can be declared in C language as int A[3][3], individual elements can be accessed by specifying row and column of that number such as to access J^{th} element of I^{th} row use A[I][J]. Since two dimensional array must stored in memory which itself is one dimensional therefore need of a method of ordering its elements in linear fashion is required which is discussed array in below.

Representation of two dimensional array in memory

A two dimensional 'm x n' Array A is the collection of m.n elements. Programming language stores the two dimensional array in one dimensional memory in either of two ways-

Row Major Order : First row of the array occupies the first set of memory locations reserved for the array, Second row occupies the next set, and so forth.

Column Major Order : Order elements of Ist column stored linearly and then comes elements of next column.

Representation of above matrix in both the order is given below

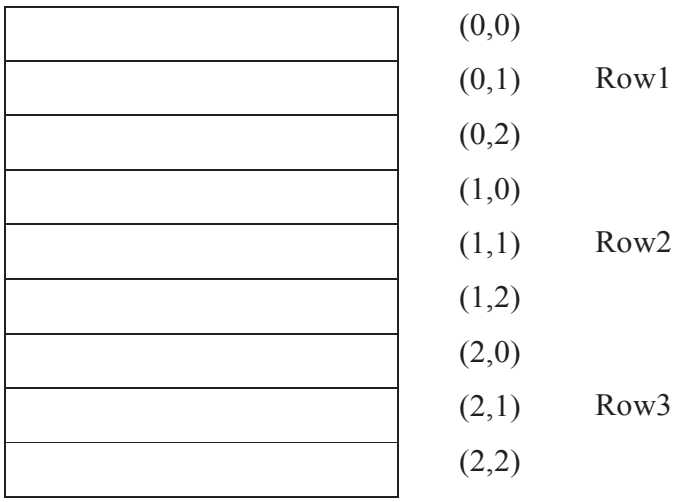


Figure : 1.1 Row Major Order

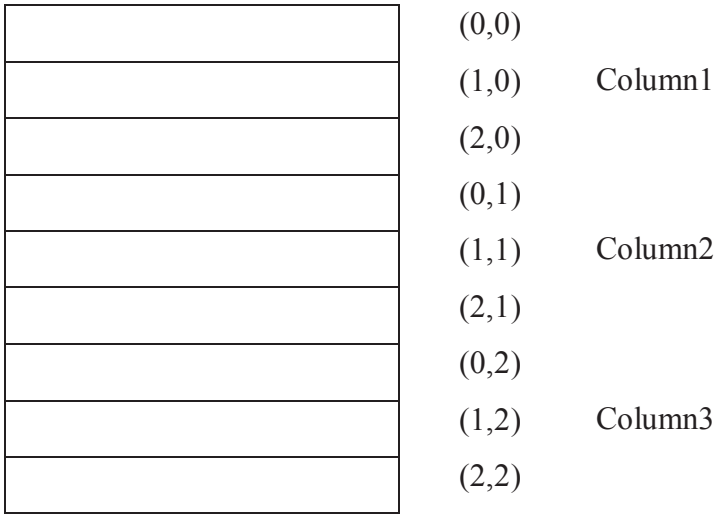


Figure 1.2 : Column Major Order

1.5.4 Advantage & disadvantages

- We can directly access array element by specifying index of that item such as to access I^{th} element just use a[I]
- We can apply binary search also if array have sorted elements.
- Insertion and Deletion require shifting of many elements, therefore we can say very costly in terms of theses operations.
- Only values have to be stored.
- Static memory allocation is there for array therefore size can not be extend or shrink

1.6 Structures in C

A structure is a collection of variables references under one name providing convenient means of keeping related information together. In other word a structure is group of items and each item is called member of structure. Each data item itself can be group item Structure in C can be define as follows

```
struct tag-name
{
        Data-type      member1;
        Data-type2     menber2;
        ____________________________;
        ____________________________;
};
For example
struct student
{
        int roll_no;
        char name[10];
        char address[30];
        float percentage;
};
```

At this time no variable has actually been created, only format of new data type has been defined. But for use structure variable needed. This can be in two ways-

```
struct student
{
        int roll_no;
        char name[10];
        char address[30];
        float percentage;
}std1,std2,std3;
```

Which allows to combine structure and variable declaration in one statement. In this use of tag-name is optional that is following declaration is also valid.

```
struct
{
        int roll_no;
        char name[10];
        char address[30];
        float percentage;
}s1,s2,s3;
```

To determine total memory requirement of structure system summed memory required by each member of structure and allot that much memory consecutively. This complete block is divided in subblocks of size required by each member in order defined in to structure.

For example above student structure require 46 bytes if 2 , 4 bytes required by integer and float variable respectively and one byte of one character. And above structure variable std1 will be represent in memory in following manner-

std1

roll_no	Name	address	percentage
2 bytes	10 bytes	30 bytes	4 bytes

Individual member of structure can be accessed **.**(Dot operator) such as std1**.**roll_no, std1**.**name and so forth.

An array which consist structure is called **array of structure**. To declare array of structure first structure is defined then array of structure defines. Each element of array represent structure variable. For example following declaration defines array class which is collection of 10 students.

```
struct student class[10];
```

Individual member of array of structure can be accessed in following manner-

class[0].roll_no, class[0].name, class[0].address, class[0].percentage

class[1].roll_no, class[1].name, class[1].address, class[1].percentage

and so on.

To access member of ith structure we can use

class[i].roll_no, class[i].name, class[i].address, class[i].percentage

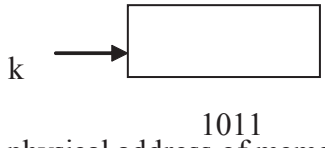
1.7 Introduction to Pointer

Pointer is reference to a data. Pointer is single fixed size data item which provide homogeneous method of referencing any data structure, regardless of structure type and complexity. Pointer also permit faster insertion and deletion of element to list. Due to these uses pointer require some amount of discussion before going into detail of data structure.

In C if we declare an integer variable k as

```
int k ;
```

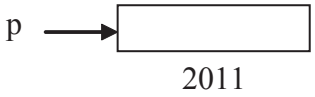
we can represent it as following diagram



Where 1011 is physical address of memory location allotted to k. similarly we also give our pointer a type which, in this case, refers to the type of data stored at the address we will be storing in our pointer. For example, consider the variable declaration:

```
int *p;
```

Block diagram to represent p as

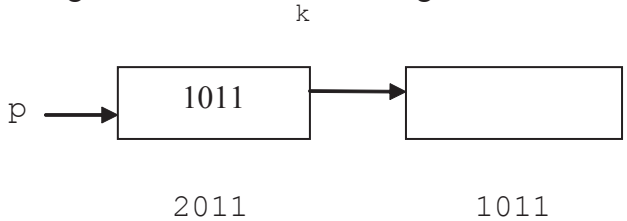


p is the name of our variable , the '*' informs the compiler that we want a pointer variable, i.e. to set aside however many bytes is required to store an address in memory. The **int** says that we intend to use our pointer variable to store the address of an integer. Such a pointer is said to "point to" an integer. **p** has no value, that is we haven't stored an address in it in the above declaration. A pointer initialized in this manner is called a "null" pointer.

But, back to using our new variable **p**. Suppose now that we want to store in **p** the address of our integer variable **k**. To do this we use the unary **&** operator and write:

```
p = &k;
```

after this assignment statement block diagram will become

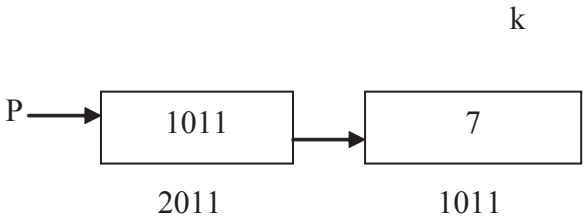


What the **&** operator retrieve the lvalue (address) of **k**, even though **k** is on the right hand side of the assignment operator '=', and copies that to the contents of our pointer p. Now, p is said to "point to" **k**. one more.

The "dereferencing operator" is the asterisk and it is used as follows:

```
*p = 7;
```

This will copy 7 to the address pointed to by **p**. Thus if **p** "points to" (contains the address of) **k**, the above statement will set the value of **k** to 7. That is, when we use the '*' this way we are referring to the value of that which p is pointing to, not the value of the pointer itself. That means block diagram will represent as



1.8 Pointer to Structure

To store address of a structure variable in a pointer you can declare a pointer to struct as follows-

```
struct student
{
        int roll_no;
        char name[10];
        char address[30];
        float percentage;
};
struct student *s;
```

To access member of stuct using pointer, "->" operator will be used such as

s-roll_no, s->name and so on.

1.9 User Defined Data Types

User defined data type supports the idea of programmers creating their own data types.

Type Definition Statement

The type definition statement is used to allow user defined data types to be defined using other already available data types.

Basic Format:

typedef existing_data_type new_user_define_data_type;

Examples:

typedef int Integer;

The example above simply creates an alternative data type name or "alias" called Integer for the built in data type called "int". This is generally not recommended use of the typedef statement.

typedef char Characters [WORDSIZE]; /* #define WORDSIZE 20 */

The example above creates a user defined data type called Characters. Characters is a data type that supports a list of "char" values. The fact that this user defined data type is an array of WORDSIZE is now built into the data type.

1.10 Example Program of Array and Structure

1. Deletion of First occurrence of given element in array .

```
# include<stdio.h>
void read_array(int a[], int size)
{
    int i;
    for( i = 0; i < size; i++)
                        scanf("%d",&a[i]);
              }

void write_array(int a[], int size)
{
    int i;
    for( i = 0; i < size; i++)
                        pintf("%d\t",a[i]);
              }
int get_position(int a[], int size,int x)
{
      int i = 0;
      while (a[i] != x) && (i  < size)
           i++;
    if(i <size)    // element exists
   return(i)
else
  return(-1);
}
void deletet(int a[], int size, int x, int pos)
{
    int i;
    for( i = pos; i < size; i++)
                          a[i] = a[i+1] ;
              }
             void main()
{
         int size, a[10], x,pos;
         printf("enter size of array and element to be inserted\n");
         scanf("%d %d",size, x);
```

```
        printf("enter elements of array \n");
        read_array(a,size);
        pos = get_position(a,size,x);
        if (pos != -1)   // element exists
        delete(a,size,x,pos);
        else
        {
            printf(element does not exists\n");
            exit(1);
        }
        printf("array after deletion\n");
        write_array(a,size-1);
}
```

2. To add and subtract two complex numbers

```
# include<stdio.h>
struct complex
{
    int real,
                int imaginary;
        }c1,c2,c3;

struct complex  add()
{
     c3.real = c1.real + c2.real;
        c3.imaginary = c1.imaginary + c2.imaginary;
      return(c3);
}

struct complex  subtract()
{
        c3.real = c1.real - c2.real;
        c3.imaginary = c1.imaginary -  c2.imaginary;
      return(c3);
}

void main()
{
        printf("enter first complex number\n");
        scanf("%d%d",&c1.real, &c1.imaginary)
        printf("enter second complex number\n");
        scanf("%d%d",&c2.real, &c2.imaginary)
        c3 = add();
        printf("result after addition is %d + %d I \n",c3.real, c3.imaginary);

c3 = subtract();
        printf("result after subtraction is %d + %d I \n",c3.real, c3.imaginary);
```

```
}
```

3.To calculate grade of n students whose roll-no, name and percentage is given.

```
# include<stdio.h>
struct student
{
        int roll_no;
        char name[10];
        float perc;
char grade;
}

struct student s[10];
void read_std(int n)
{
        int i;
        for (i = 0; i < n; i++)
        {
                printf("enter roll number\n");
                scanf("%d",&s[i].roll_no);
                printf("enter name \n");
scanf("%s",s[i].name);
printf("enter percentage\n");
                scanf("%f",&s[i].perc);
        }
}

void print_std_rec(int n)
{
        int i;
        for (i = 0; i < n; i++)
        {
                printf("roll number=%d\t", s[i].roll_no);
                printf("name = %s \t",s[i].name);
printf("percentage = %f\t",s[i].perc);
printf("grade = %c\n",s[i].grade);
        }
}

void cal_grade(int n)
{
        If (s[i].perc >=75)
                S[i].grade = 'O';
        Else
```

```
                If (s[i].perc < 75) && (s[i].perc >=60)
                        S[i].grade =  'A';

                Else
                        If (s[i].perc < 60) && (s[i].perc >=50)
                                S[i].grade =  'B';
                        Else
                                If (s[i].perc < 50) && (s[i].perc >=36)
                                        S[i].grade =  'C';
                                Else
                                        If (s[i].perc < 36)
                                                S[i].grade =  'F';
}

void main()
{
printf("enter total number of student\n");
        scanf("%d",&n);
        printf("enter student information\n");
        read_std(n);
        cal_grade(n);
        printf("records with grade information\n");
        print_std_rec(n);

}
```

1.11 Summary

Data refers to value or set of values whereas processed data is information. Data type is all possible values for that type; and the operations that can be done on values of that type. Collection of different type of data is known as records which is structure in C programming language.

The use of data structure and algorithm is the nut-and-bolts used by programmers to store and manipulate data. Data structure is the way of storing and retrieval of data into computer in significant manner.

An array is collection of homogeneous data elements referred by single name. A one dimensional array is list of finite number of homogeneous data element such that the elements are referred by single index

In array I^{th} element can be accessed directly by specify index as array-name[index i] if elements in list is sorted then binary search can be used to search an item. Insertion, deletion is perform costly in array since it require shifting of many elements those doesn't effect directly by insertion and deletion.

Pointer is memory location which hold address of elementary items, array and structure etc. Pointer also permit faster insertion and deletion of element to list.

A structure is a collection of variables references under one name providing convenient means of keeping related information together.

1.12 Self Assessment Questions

1. What do you understand by data structure? Discuss its need.
2. Describe pointers and how are they used in structure?
3. What is difference between data and information?
4. How two dimensional array stored in memory?

Unit - 2 : Basics of Linked List

Structure of Unit:

2.0 Objectives

After Ready this unit, you will be able to

- Define self referential structure.
- Provide general overview of linked list of various types.
- Explain requirement of linked structure.
- Comparison of various types of linked list and their need.

2.1 Introduction

The list refers to linear collection of data items. Data processing involve storing and processing data organize into lists. One way to store list is array which has been discussed in unit1. But due to fixed size of array list and consecutive memory requirement of complete lists one would not store data in an array. Another way of storing list in memory in a way that each element have a field of pointer which contain address of next element in the list. Successive elements need not occupy adjacent space in memory. Therefore we can say list can be maintain by linked structure of data items.

2.2 Self Referential Structure

It is exactly what it sounds like : a structure which contains a reference to itself. Self referential structure consists of pointer which points to another structure of same type. For example, a linked list is supposed to be a self referential data structure. For example,

```
typedef struct listnode {
        int    *info;
        struct listnode  *next;          // ← self reference
}node;
```

Here node is a structure having data of integer type with variable "info" and a pointer which is referring to structure of same type node hence called self- referential structure that is *listnode* is a self-referential structure- because the *next is of type struct listnode.

A self referential structure is better called recursive structure. If a pointer is not NULL, it point to another instance of the structure. The NULL pointer is base case for recursion. You can print list recursively as an example of the relationship between self referential structure and recursion.

2.3 Static and Dynamic Memory Allocation

In many programming environments memory allocation to variables can be of two types static memory allocation and dynamic memory allocation. Both differ on the basis of time when memory is allocated. In static memory allocation memory is allocated to variable at compile time whereas in dynamic memory allocation memory is allocated at the time of execution. Other differences between both memory allocation techniques are summarized below-

S. No.	Static memory allocation	Dynamic memory allocation
1.	Memory allotted before run time (mostly at compile time)	Memory allotted at run time.
2.	Programmer must know in advance amount of memory needed.	Programmed is not expected to have knowledge of amount of memory requirement in advance.
2.	Since we may only know maximum need therefore finite amount of memory allotted	Only required amount of memory allotted.
3.	Since maximum amount of memory is allocated, but actual usage may be far less therefore wastage is there.	Only required amount of memory is allocated therefore no wastage.
4.	No functions for memory request is needed. Memory will be allocated automatically.	Predefined functions for memory allocation request will be used such as malloc, calloc in C programming language
5.	Since memory is already allotted to variables before runtime therefore it save run time.	Time required in memory allotment will increase run time of program.
6.	Static allocation is fast	Dynamic allocation is slower.
7.	Allot memory of data segment of memory.	Allot memory on heap (Pool of free memory).
8.	The space is allotted once and is never freed.	Programmer can free block of memory once it has longer needed.
9.	Static memory allotted to elementary data item, array, structure	Memory allotted to pointer is dynamic

2.4 Definition and Need of Linked List

Computers are frequently used to store and manipulate list of data. There are many ways to represent a list in a computer. The choice of how best to represent a list is based on how list will be manipulated. One way of implementing list is to use linear arrays. In array traverse and search operation carried out in linear time, if list is sorted binary search can be implement in logarithm time. But insertion deletion are costly since shifting of many elements are required, therefore linear time complexity is in array insertion and deletion operation..

Other way to represent the list is required due to following reasons-

- Some time we do not have a knowledge of size in advance.
- To improve time required by insertion and deletion.
- To eliminate size constraints of array.
- To reduce wastage occurred in array list.

Linked list can be possible alternative. Linked List is a homogeneous collection of elements with a linear relationship, which are not needed contiguous. Linked lists can grow and shrink dynamically. From implementation point of view linked list is defined as a collection of nodes. Each node has two parts

Information

Pointer to next node

Information part may consists simple or structure data item where as next field of node contain address of next item of list. Pointer field of last node is NULL in C. In practice, '0', '\0' or negative number is used for NULL pointer. Linked list also contains starting list pointer variable say START which hold address of the first node. List can have no nodes or null list represent by NULL pointer in START that can be created by START = NULL ;

List of four integers is represented by –

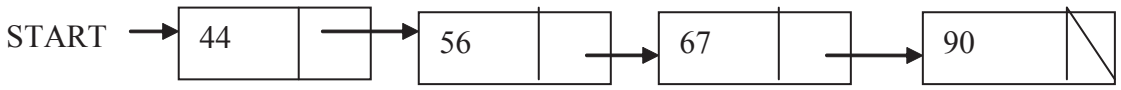


Figure 2.1 :Linear Linked List of Four nodes.

2.5 Comparison of Linked List and Array

Comparison between array and linked list are summarized in following table –

S.No	Array	Linked List
1.	List of elements stored in contiguous memory location i.e. all elements linked physically.	List of elements need not stored in contiguous memory location i.e. all elements will be linked logically.
2.	Contiguous memory required for complete list that will be large requirements.	Contiguous memory required for single node of list that will be small requirements.
3.	List is static in nature i.e. created at compile time mostly.	List is dynamic i.e. created and manipulated at time of execution .
4.	List can't grow and shrink	List can grow and shrink dynamically
5.	Memory allotted to single item of list can't free	Memory allotted to single node of list can also be free.
6.	Random access of I^{th} element is possible through a[I]	Sequential access to I^{th} element i.e. all previous I -1 node have to traverse before.
7.	Traversal easy since any elements can access dynamically and randomly.	Traversal is done node by node hence not as good as in array.
8.	Searching can be linear and if sorted than in array we can also apply binary search of logarithm time complexity.	Searching operation must be linear in case of sorted list also. Time complexity is proportional to list length..
9.	Insertion Deletion is costly since require shifting of many items.	Insertion Deletion is performed by simply pointer exchange. Only set of pointer assignment statements can perform insertion deletion operation.

2.6 Types of Linked List

A linked list can be of the following types –

Linear Linked List

Header Linked List

Circular Linked List

Doubly Linked List

2.6.1 Linear Linked List

Linear Linked List is a list of linked elements which has following characteristics

- last node contains NULL as a next pointer field
- Only one pointer field is there which is pointing to next node.
- Pointer of First node is available.

Linear Linked list of four nodes is shown in Figure 2.1

2.6.2 Header Linked List

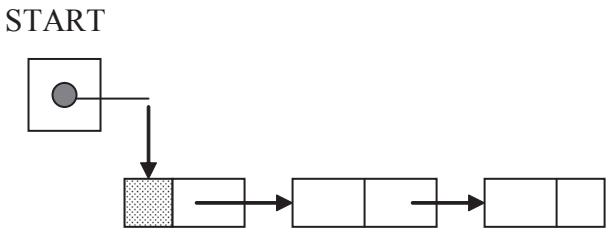
A header node is a special node that is found at front of the list. The linked list that contains header node is called header linked list. Two type of header node are as follows

Grounded header linked list

Circular header linked list

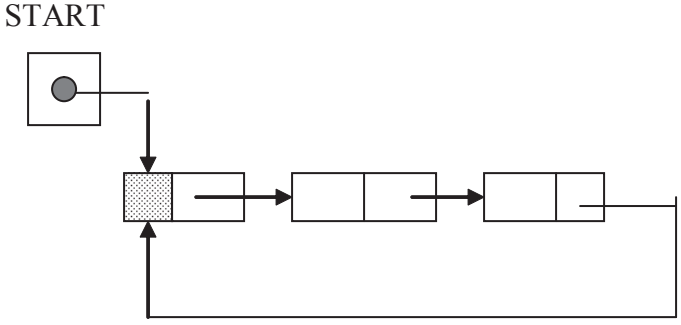
Grounded header linked list

This list is a list where last node contain the NULL pointer and START pointer always point to header node.



Circular Header linked List

In this list last node points to header node. This chain does not indicate first or last nodes.



2.6.3 Circular Linked List

Circular Linked List is a list of linked elements which has following characteristics

- last node contains pointer of first node
- Only one pointer field is there which is pointing to next node.

Circular Linked list of four nodes is shown in Figure 2.2

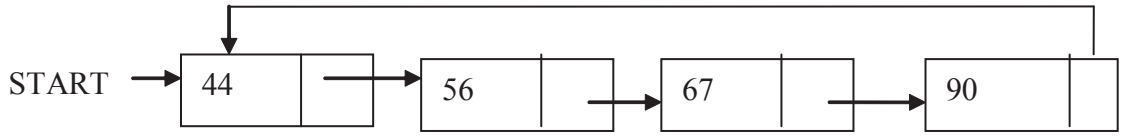


Figure 2.2 Circular Linked List

2.6.4 Doubly Linked List

Doubly Linked List is a list of linked elements which has following characteristics

- Two pointer field are there which points to next and previous node.
- Last node contains NULL as a next pointer field
- First node contains NULL as a previous Pointer Field
- Pointer of First node is available.

Doubly Linked list of four nodes is shown in Figure 2.3

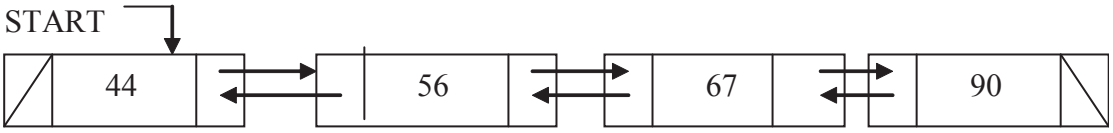


Figure 2.3 Doubly Linked List

2.7 Comparisons of Singly, Circular and Doubly Linked List

Linear Linked List	Circular Linked List	Doubly Linked List
node declaration is as follows Typedef Struct LL { int info; struct LL *next; } Node;	node declaration is as follows Typedef Struct CL { int info; struct CL *next; } Node;	node declaration is as follows Typedef Struct DL { struct DL *prev; int info; struct DL *next; } Node;
Backward traversal not possible.	Backward traversal not possible.	Backward traversal possible
Traversal of entire list can be possible through first node only.	Traversal of entire list can be possible through any node.	Traversal of entire list can be possible through first node only.
Only single pointer require memory.	Only single pointer require memory.	More memory requirement of one node due to two pointers
Only single pointer have to maintain.	Only single pointer have to maintain.	Two pointer have to maintain. That means extra overhead introduced.
Requirement of a pointer of previous node in insertion and deletion operation.	Requirement of a pointer of previous node in insertion and deletion operation	No requirement of a pointer of previous node in insertion and deletion operation since through *prev* pointer field we can access previous node

2.8 Summary

Self referential structure is basically nothing but referring to a structure of its own type.

There are various ways to implement list of data item. When list is maintain using array it is static list and if list is maintain by linked data items list is knows as dynamic list.

Merits of dynamic list on static list

- List grow and shrink dynamically on user's requirement.
- Fixation of final limits depends only on memory available.
- Optimum utilization of memory since no wastage is there.
- Insertion and deletion of values is very easy.

Merits of static list on dynamic list

- The memory once allotted has not to be freed by user.
- Direct access to any item of list by index is available.
- Binary search can be implemented.

Doubly linked list is collection of data items where each element has pointers to both the next element and the prior element.

Where as Linear linked list is collection of data items in which each element has pointer to next element and last element consists NULL.

A circular linked list is where the last element points back to the first element.

2.9 Self Assessment Questions

1. What are advantages of linked list over an array ?
2. What are advantage of Doubly linked list over Linear Linked List ?.
3. What are advantages of dynamic memory allocation technique over static memory allocation.

Unit - 3 : Implementation of Singly Linked List

Structure of Unit:

3.0 Objectives

The objective of this unit is to

- Introduce process of creation of linked list
- Discuss operations in singly linked list – Insertion, deletion and traversal
- Design algorithm for all the above operations.
- Provide method of searching in linked list since it is required by mostly operations.
- Summarize applications which require singly linked list structure.

3.1 Introduction

The list refers to linear collection of data items. Data processing involve storing and processing data organize into lists. One way to store list is array which has been discussed in unit 1. But due to fixed size of array list and consecutive memory requirement of complete lists one would not store data in an array Another way of storing list in memory in a way that each element have a field of pointer which contain address of next element in the list. Successive elements need not occupy adjacent space in memory. Therefore we can say list can be maintain by linked structure of data items.

3.2 Creation of Singly Linked List

Linked lists are one of the most useful ways of representing linear sequences of data. Linked lists are the sequence representation of choice when many insertions and/or deletions are required in interior positions and there is little need to jump around randomly from one position to another.

Let Q denote new node to be append in linked structure, LAST global pointer which is pointing to last node in linked list, START is representing First node of list. Creation of singly linked list can be well described by following block process-

Initially queue is empty START = LAST = NULL

Create new node Q and fill all entries. Q = newnode()

Check that newly created node is First node or not if yes START and LAST point to newly node since this is single node in linked list therefore this is first as well as last node of list.

If START = NULL Then

START = LAST = Q

If newly created node is not First node then link last node to newnode Q. and update LAST pointer with newnode since now newly created node will now be last node of already created list.

If START != NULL Then

Last → next = Q

LAST = Q

Complete Process can be understand by following diagram more easily.

START = NULL (Queue is empty)

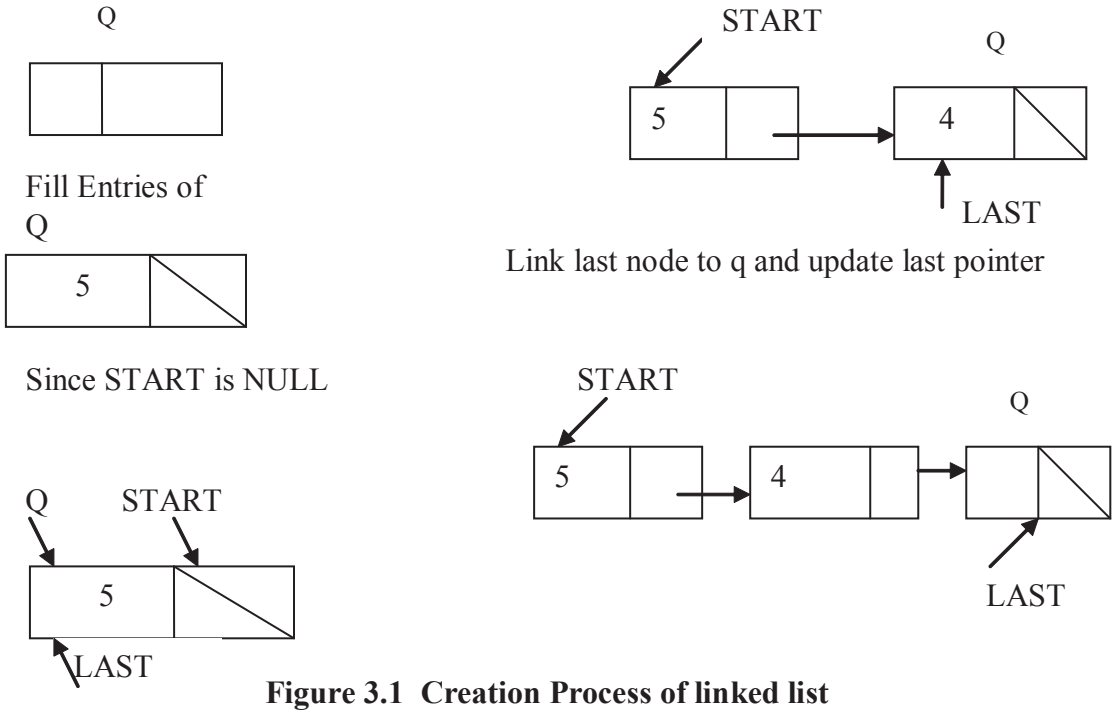


Figure 3.1 Creation Process of linked list

Complete algorithm is as given below

Algorithm : Create(Start, ele)

Where start is starting node of linked list in which new value ele have to be append. LAST is global pointer you can traverse to last position also. Traversal will discusses later.

1. [creation of new node Q that is to be append]
   Q = newnode() // new node allocate memory to a node
   Q→ info = ele
   Q→ next = NULL // fill all the field of node struct

2. [Check Q is First node]
   If (START = = NULL)
    START = LAST = Q
   Else
   LAST → Next = Q
   LAST = Q
3. Return (START)
4. Exit

3.3 Operations on Linked List

3.3.1 Traversal in Singly linked list

Traversal operation required very frequently such as in insertion, deletion and searching operation etc. Since in singly linked list, traverse direction forward therefore START is very crucial pointes since if it is lost we can't get entire list So it it better to take temporary pointer **P** to traverse.

Algorithm to traverse the complete singly linked list is as follows-

Algorithm Traverse(START)

1. P = START // start traverse at first node
2. while P != NULL //list is complete or not
3. Print P → info // process the node
4. P = P → next // switch to next node
5. Exit

3.3.2 Searching in Singly Linked List

To search an item in singly linked list, algorithm of traversal can be used with following alteration

Traverse till item is found otherwise till end of the list to conclude element does not exists.

Algorithm to search an item in linked list of which starting node pointer is given is as follows-

Algorithm Search(START, item)

1. P = START // start search at first node

[list is not complete and item also not found]

2. while P != NULL and P → info != item
3. P = P → next // switch to next node
4. if (P = = NULL)
5. Print ("Item not Found")
6. else
7. Print ("Item Found")
8. Exit

3.3.3 Insertion in Singly Linked list

There can be two situation of insertion First Insertion of a new node in sorted list in which position at which insertion expected is required to be searched. Where as in insertion in unsorted list position is given as input hence not required to be searched. Further we will discuss algorithm of insertion in sorted list

Insertion in a linked list can performed in three places

- At beginning of list
- At last node
- At specified position that is in between list.

Insertion at middle

Suppose **START** is pointing to first node of singly linked list with successive nodes **BACK** and **P** as shown in figure 3.2

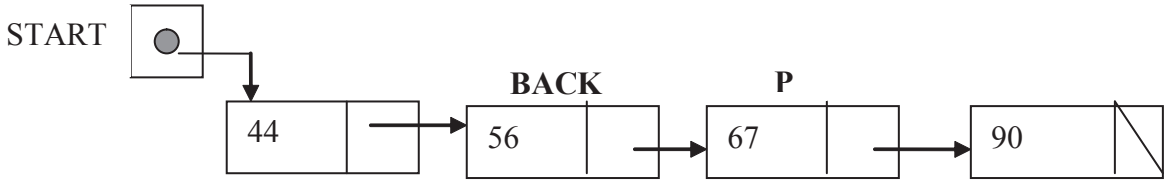


Figure 3.2 Original linked list

Suppose a node **X** is to be inserted into the list between **BACK** and **P. X** contains **65** as a content of *info* field insertion can be shown by following diagram.

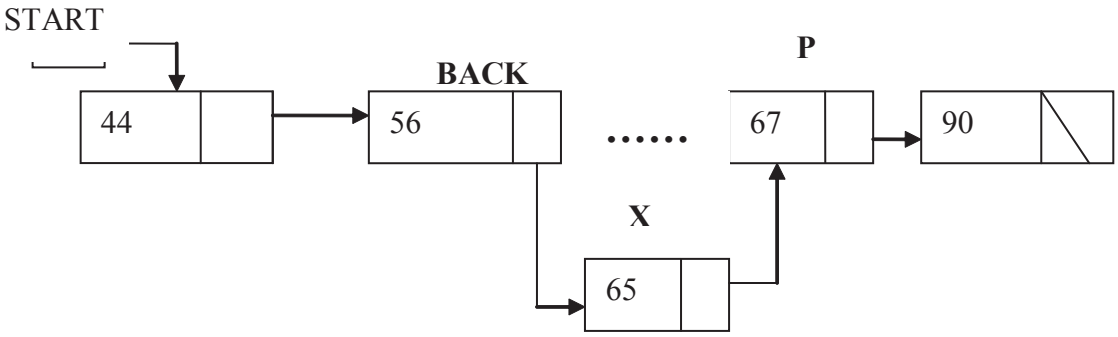


Figure 3.3 Insertion in middle in linked list

Observe that two pointer fields are changed as follows-

- Next pointer field of node **BACK** now points to new node **X**. This can be update by BACK → next = X
- Next pointer field of node X now points to new node **P**. This can be update by X → next = P

Insertion at beginning

Suppose linked list is as given in **figure 3.4 a**nd let a node **X** is to be inserted into the list contains **35** as a content of *info* field insertion can be shown by following diagram.

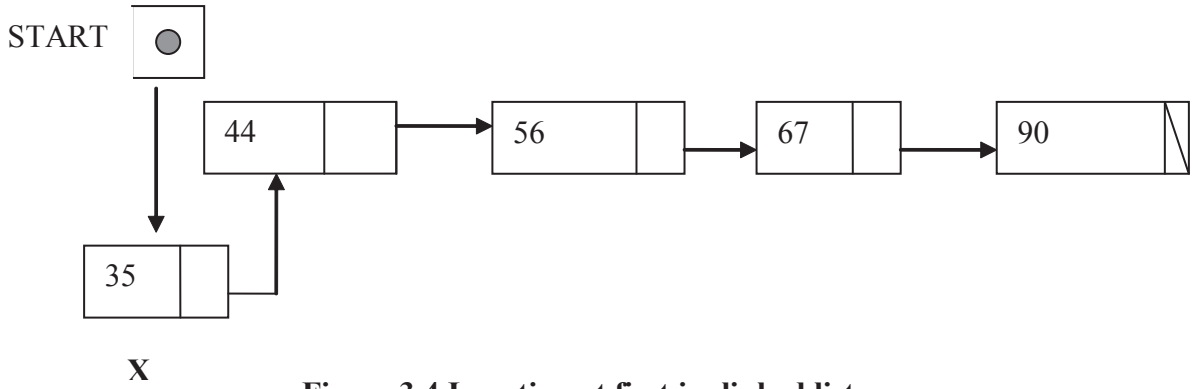


Figure 3.4 Insertion at first in linked list

Two pointer fields are changed as follows-

- Next pointer field of node **X** now points to pointer at **START**. This can be done by X → next = START
- **START** now points to new node **X**. This can be update by START = X

Insertion at end

Suppose linked list is as given in **figure 3.5 a**nd let a node **X** is to be inserted into the list contains **95** as a content of *info* field insertion can be shown by following diagram

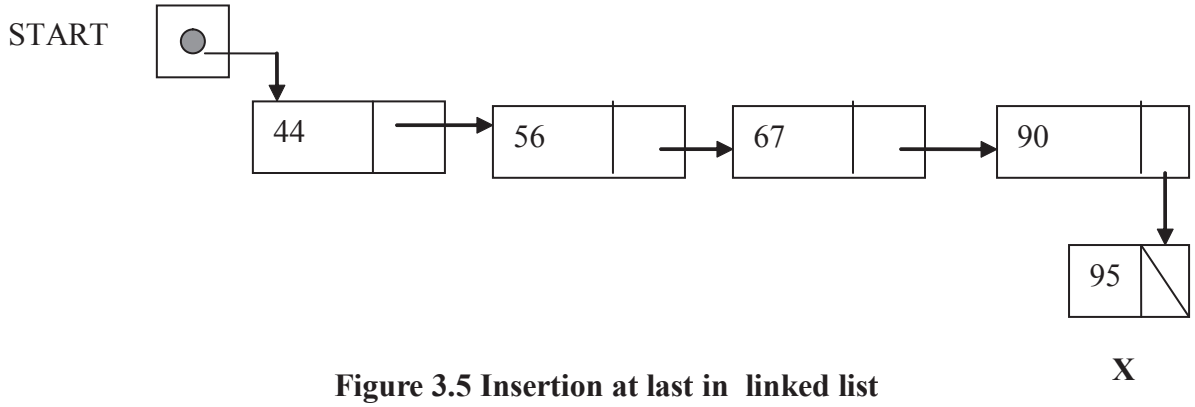


Figure 3.5 Insertion at last in linked list

Pointers fields are to be changed as follows-

- Next pointer field of node **X** now points to NULL since it become last node of list. This can be done by X → next = NULL
- Next pointer field of node last node say **LAST** now points to new node **X**. This can be update by LAST→ next = X

Complete algorithm that include all cases of insertion in sorted linked list such that list after insertion must be sorted is given below-

Algorithm : Insert(Start, ele)

Where start is starting node of linked list in which new value ele have to be inserted.

```
5.      [creation of new node X that is to be inserted]
X = newnode()   // new node allocate memory to a node
X→ info = ele
6.      [search position for insertion]
P = START
BACK = NULL
while P != NULL And P → info  < X → info
BACK = P
P = P → next
7.      [check various insertion cases]
8.      if X → info < START → info     // insert at beginning
X → next  =  START
START = X
return( START )
9.      else  if P = = NULL    // insertion at end
```

BACK → next = X
X → next = NULL
10. else // insertion at middle
BACK → next = X
X → next = P
11. Exit

3.3.4 Deletion in Singly Linked list

To delete a node with given item of information , First search the item by traversing the list if it exists after that delete it by storing address of successor node in the predecessor node to the node to be deleted. Free the node to be deleted.

Deletion operation in a linked list can performed in three places

- of first node of list
- of last node
- of specified node that is in between list.

Deletion at middle

Suppose **START** is pointing to first node of singly linked list with successive nodes **BACK** and **P** and **P** is the node to be deleted as shown in figure 3.6

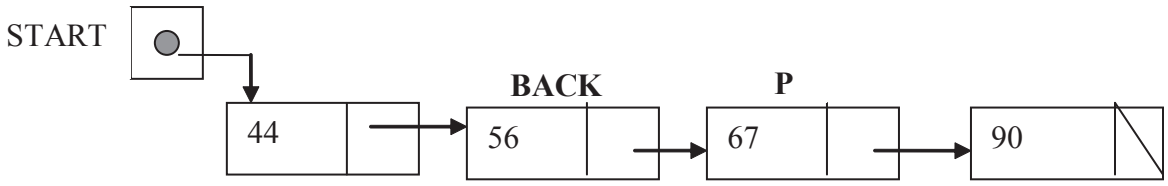


Figure 3.6 Original linked list

Pointer adjustment required by this deletion operation is shown in following figure-

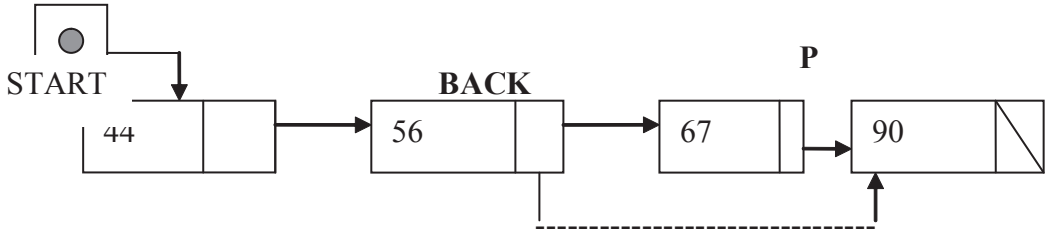


Figure 3.7 Deletion of middle node from linked list

Observe that two pointer fields are changed as follows-

- Next pointer field of node **BACK** now points to new node **X**. This can be update by BACK → next = X → next

Deletion at beginning

Suppose linked list is as given in **figure 3.8 a**nd let a node is to be deleted from the list contains **44** as a content of *info* field. Pointer adjustment required in this deletion can be shown by following diagram.

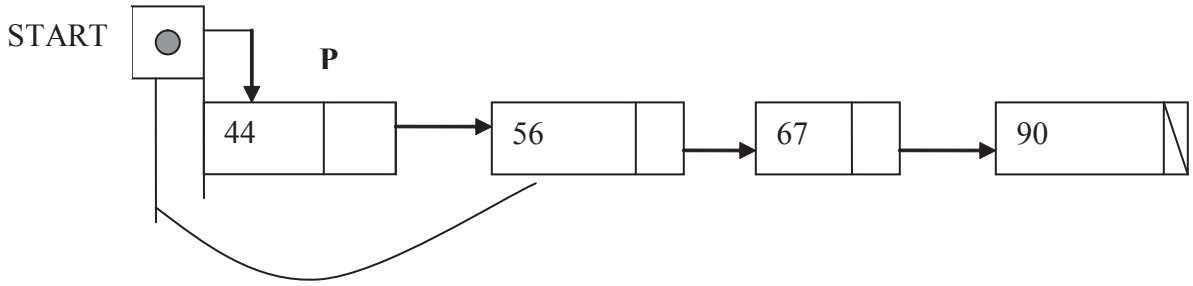


Figure 3.8 Deletion of first node from linked list

pointer fields are changed as follows-

- **START** now points to next node TO **P.** This can be done by

  START = P → NEXT

Deletion at end

Suppose linked list is as given in **figure……. a**nd let a node to be deleted from the list contains **90** as a content of *info* field **deletion** can be shown by following diagram

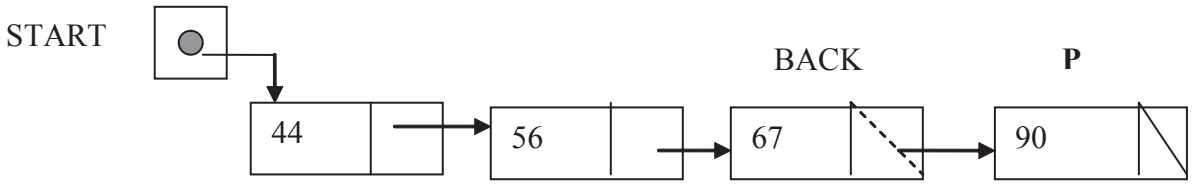


Figure 3.9 Deletion of last node from linked list

Pointers fields are to be changed as follows-

- Next pointer of **BACK** will hold **NULL** that is

  BACK → next = NULL

  To release memory area occupied by **P** by calling **free(P)** function.

Complete algorithm that include all cases of deletion in linked list is given below-

Algorithm : Delete(Start, ele)

Where start is starting node of linked list in which new value ele have to be deleted.

1. [Searching node to be deleted Since pointer of previous node required so BACK should point to previous node]
2. P = START
3. BACK = NULL
4. while P → info != ele and P != NULL
5. BACK = P
6. P = P → next
7. end while
8. [check various deletion cases]

```
9.      if START → info = = ele     // deletion at beginning
            START = P → next
    else
            if P = = NULL          // deletion at end
                BACK → next = NULL
            Else          // deletion at middle
                BACK → next = P → next
10.     free(P)  // to release memory are
11.     Exit
```

3.4 Application of Linked List

Linked lists can be which in many applications. Some important ones are given below.

- Polynomial Representation
- Sparse Matrix Representation
- Symbol Table Construction
- Implementation of inbuilt Data Structure

3.4.1 Polynomial Representation

A polynomial can be represented in an array or in a linked list by simply storing the coefficient and exponent of each term. However, for any polynomial operation, such as addition or multiplication of polynomials, you will find that the linked list representation is more easier to deal with due to two reasons.

First is variable size of linked lists allow to store any number of terms of polynomial, where as fixed size of array limits maximum number of terms in polynomial.

Second, many operations in polynomial require construction of new polynomial which represent resulted polynomial. Insertion and deletion of new terms in result polynomial can be perform efficiently in linked list since it does not require shifting of already existing terms which is frequently perform in case of array insertion/ deletion. It would be also easy to manipulate a pair of polynomials if they are represented using linked lists.

In a polynomial all the terms may not be present, especially if it is going to be a very high order polynomial. . Consider

$5\ x^{12} + 2\ x^9 + 4x^7 + 6x^5 + x^2 + 12\ x$

Now this 12th order polynomial does not have all the 13 terms (including the constant term).

It would be very easy to represent the polynomial using a linked list structure, where each node can hold information pertaining to a single term of the polynomial. Each node will need to store

- the variable x,
- the exponent and
- the coefficient for each term

  It often does not matter whether the polynomial is in x or y. This information may not be very crucial for the intended operations on the polynomial. Thus we need to define a node structure to hold two integers , viz. exp and coeff

Compare this representation with storing the same polynomial using an array structure. In the array we have to keep a slot for each exponent of x, thus if we have a polynomial of order 50 but containing just 6 terms, then a large number of entries will be zero in the array.

Addition of two polynomials

Consider addition of the following polynomials

$5\ x^{12} + 2\ x^9 + 4x^7 + 6x^6 + x^3$

$7\ x^8 + 2\ x^7 + 8x^6 + 6x^4 + 2x^2 + 3\ x + 40$

The resulting polynomial is going to be

$5\ x1^2 + 2\ x^9 + 7\ x^8 + 6\ x^7 + 14x^6 + 6x^4 + x^3 + 2x^2 + 3\ x + 40$

Now notice how the addition was carried out. Let us say the result of addition is going to be stored in a third list. We started with the highest power in any polynomial. If there was no item having same exponent , we simply appended the term to the new list, and continued with the process. Wherever we found that the exponents were matching , we simply added the coefficients and then stored the term in the new list. If exponents are matching and coefficients are equal with apposite sign then they cancel each other and no insertion of new term in summed polynomial carried out. If one list gets exhausted earlier and the other list still contains some lower order terms, then simply append the remaining terms to the new list.

3.4.2 Sparse Matrix

Sparse matrices are usually used to store and process data in many fields, for different purposes. Usually, we use 2D-dimension array to store all elements of a matrix. It means that if size of the matrix is m rows and n columns, we have to use a 2D-dimension which has m*n elements.

Consider a matrix in which most of the entries are zeros. This sometimes happens, for example when we process problems about graph where many elements of the data are zeros. In that case a large amount of memory is wasted to explicitly store all those zeros. Likewise when performing operations on the sparse matrices such as addition, subtraction and multiplication (especially), a large number of the operations have two operands of zero (0) which results in a large amount of wasted time.

If a matrix is sparse, we can store the elements that are non-zero only and implicitly assuming any element not stored is 0. This greatly reduces the number of times to be stored and the number of operations that must be performed for addition, subtraction and multiplication etc. So instead of using 2D-array , we can use some other data structures to store elements of matrices and basing on new data structures, we perform the operations mentioned above more effectively.

We know that linked-lists are very popular and a useful kind of data store. The problems of storing and processing data in sparse matrices mentioned above can be done well by using some kind of linked-lists. The advantage of linked-list is manageability of memory and it is very important for large programs which are used to process a lot of data.

We can make up a linked list of linked lists. For example we require to represent following sparse matrix. Following is the linked list representation of sparse matrix

2 0 0 0
4 0 0 3
0 0 0 0
8 0 0 1
0 0 6 0

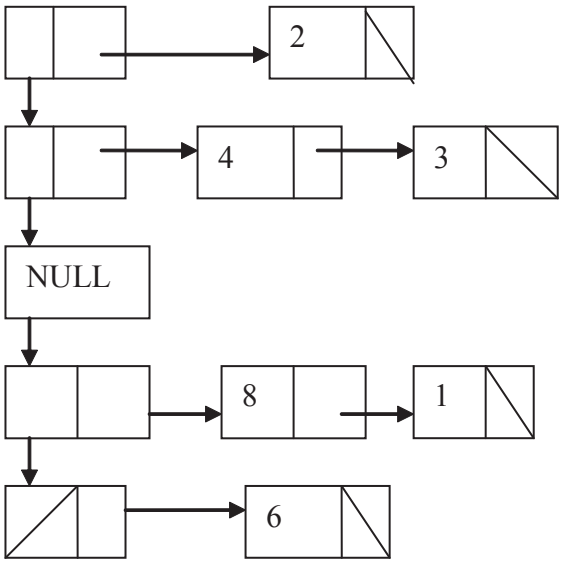


Figure 3.10 Linked list representation of above sparse matrix

This is the way to save time and resources of the computer by reducing the number of operations which have zero elements.

3.4.3 Symbol Table Construction

A compiler uses a symbol table to keep track of scope and binding information about names. The symbol table is searched every time a name is encountered in the source text. Changes to the symbol table occur if a new name or new information about an existing name is discovered.

A symbol table mechanism must allow us to add new entries and find existing entries. The two symbol table mechanisms are linear lists and hash tables. Each scheme is evaluated on the basis of time required to add *n* entries and make *e* inquiries. A linear list is the simplest to implement, but its performance is poor when n and e are large. Hashing schemes provide better performance but greater programming effort and space overhead.

It is useful for a compiler to be able to grow the symbol table dynamically at compile time. If the symbol table is fixed when the compiler is written, the size must be chosen large enough to handle any source program that might be presented. Therefore linked list is better than array.

But Look up operation in linked list carried out using linear search therefore hashing is more important data structure compare to linked list with respect to look up operation, where as insertion and deletion perform more efficiently in linked list compare to hash table. And insertion deletion of symbols in symbol table are more frequent therefore combination of hashing and linked list will be better for symbol table.

3.4.4 Linked representations of data structure stacks and queues

Stack is a list of items accessed in LIFO(Last In First Out) order. Where as Queue is list of items accessed in FIFO(First In First Out) order. Both data structure will be discussed in detail in further sections

Stacks and Queues are easier to implement using linked lists compare to array. There is no need to set a limit on the size of the stack.

Let us discuss in brief push, pop and enqueue and dequeue functions for nodes containing integer data.

- The Push function places the data in a new node, attaches that node to the front of stacktop(point to top elements of stack).
- Pop Function returns the data from top element, and frees that element from the stack.

  For Queue operations, we can maintain two pointers qfront(points to front end of queue) and qback (points to rear end of queue).
- For the Enqueue operation, the data is first loaded on a new node. If the queue is empty, then after insertion of the first node, both qfront and qback are made to point to this node, otherwise, the new node is simply appended and qback updated.
- In Dequeue function, first of all check if at all there is any element . If there is none, we would have qfront as NULL, and so report queue to be empty, otherwise return the data element, update the qfront pointer and free the node. Special care has to be taken if it was the only node in the queue.

Advantages of Linked Stack and Queue

- Computationally & conceptually Simple
- No longer need to shift elements to make space
- Computation can proceed as long as there is memory available
- The reduced computing time required by linked representations

Disadvantages

- Need additional space for the link field.
- Overhead to maintain pointers.

3.5 Summary

Linked lists are one of the most useful ways of representing linear sequences of data.

Linked list is alternate approach to maintain array in which in place of fixed large block memory allocation is on demand.

As a array various operations may be manipulate on linked list too. Some of them are traversal, searching, insertion and deletion.

Linked lists are the sequence representation of choice when many insertions and/or deletions are required in interior positions and there is little need to jump around randomly from one position to another.

linked list is one of the fundamental data structures, and can be used to implement other data structures.. Linked lists permit insertion and removal of nodes at any point in the list in constant time, but do not allow random access.

Linked lists are used as a building block for many other data structures, such as stacks, queues and their variations.

3.6 Self Assessment Questions

1. Write an algorithm to find subtraction of two polynomial using linked list representation of linked list.
2. Compare linked list and array on the basis of operations on both data structure.
3. It will be efficient to represent sparse matrix in linked list. Comment.
4. Can linked list represent in array, if yes show how insertion and deletions can perform in array representation of inked list.

.

Unit - 4 : Stack

Structure of Unit:

4.0 Objectives

This chapter covers

- Introduction to stack
- Concept of various operations over stack such as push, pop etc.
- Array and linked representation of stack
- Various applications of stack

4.1 Introduction

A stack is a homogeneous collection of items of any one type arranged linearly accessed with only one end, which is known as top of the stack. This means data can be added and removed from top only. This type of structure is also known as Last In First Out (LIFO) structure. Adding of data in a stack is known as push and removing of data from a stack is known as pop operation.

The stack is a common data structure for representing things that need to maintained in a particular order. For instance, when a function calls another function, which in turn calls a next function, it is important that the next function return back to the second function rather than the first.

Stacks have some useful terminology:

- Push To add an element to the stack
- Pop To remove an element from the stack
- LIFO Refers to the last in, first out behavior of the stack
- FILO (First In Last Out) Equivalent to LIFO behavior

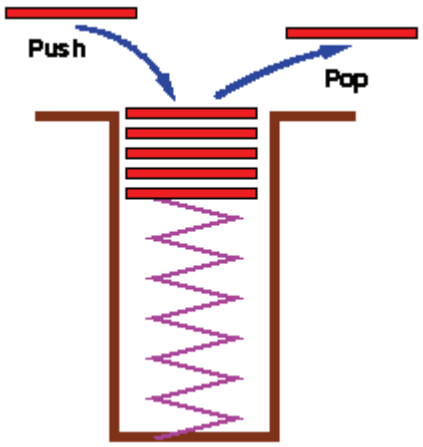


Figure 4.1 : stack push and pop operation

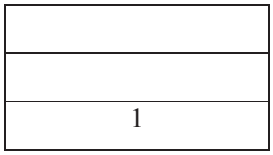
A stack is a non-primitive data structure. There are many real life examples representing stacks such as stack of dishes, stack of folded towels, stack of boxes etc., in all these, addition of item and deletion of items take place from top side only.

Some basic operations over the stack :

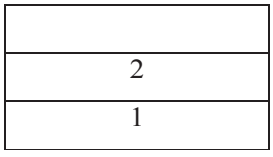
1. Create(S): creates the empty stack S

S

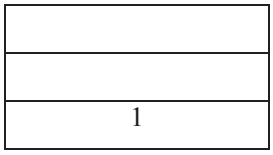
2. PUSH(1, S) : add 1 at the top of stack S



3. PUSH(2, S) : add 2 at the top of stack S



4. POP(S) : delete element from top of the stack S



4.2 Creation of Stack

In most of the high level languages, a stack can be easily implemented either through an array or a linked list. What identifies the data structure as a stack in either case is not the implementation but the interface: the user is only allowed to pop or push items onto the array or linked list, with few other helper operations. Creation of stack means creating array structure without putting value into it. The following will demonstrate both implementations, using C.

1. Array

The array implementation aims to create an array where the first element (usually at the zero-offset) is the bottom. That is, array[0] is placed at the first element pushed onto the stack and the last element popped off. The program must keep track of the size, or the length of the stack. The stack itself can therefore be effectively implemented as a two-element structure in C:

```
typedef struct {
   int  size;
   int items[STACKSIZE];
} STACK;
```

The push() operation is used both to initialize the stack, and to store values to it.

If we use a dynamic array, then we can implement a stack that can grow or shrink as much as needed. The size of the stack is simply the size of the dynamic array. A dynamic array is a very efficient implementation of a stack.

2. Linked list

The linked-list implementation is equally simple and straightforward. In fact, a simple singly linked list is sufficient to implement a stack-it only requires that the head node or element can be removed, or popped, and a node can only be inserted by becoming the new head node.

Unlike the array implementation, our structure typedef corresponds not to the entire stack structure, but to a single node:

```
typedef struct stack {
   int data;
   struct stack *next;
} STACK;
```

Such a node is identical to a typical singly linked list node, at least to those that are implemented in C.

The push() operation both initializes an empty stack, and adds a new node to a non-empty one. It works by receiving a data value to push onto the stack, along with a target stack, creating a new node by allocating memory for it, and then inserting it into a linked list as the new head.

4.3 Stack Operations

Before going into detail we should be familiar with some operations related to stack. These are given below :-

4.3.1 Push: Push means insert any item in stack. The process of adding a new element to the top of stack is called PUSH operation. Pushing an element in the stack invoke adding of element, as the new element will be inserted at the top after every push operation the top is incremented by one. In case the array is full and no new element can be accommodated, it is called STACK-FULL condition. This condition is called stack overflow.

4.3.2 Pop : Pop means delete an element from stack. The process of deleting an element from the top of stack is called POP operation. After every pop operation top is decremented by one. If there is no element on the stack and the pop is performed then this will result into stack underflow condition.

4.3.3 Top : Top tells about the location from where elements can be removed or where elements can be added. It gets the next element to be removed.

4.3.4 IsEmpty : This is a condition which checks whether the stack is empty or not, before applying pop operation IsEmpty condition has to be checked to ensure the presence of an element in a stack.

4.3.5 IsFull : This is a condition which checks whether the stack is full or not, before applying push operation IsFull condition has to be checked to find out whether there is any space or not to add an element in a stack.

4.3.6 Display : This operation is used to display all the elements present in a stack linearly.

4.4 Stack Implementation

The stack is a very common data structure used in programs. Representation of stack may be done in various ways in computer memory for ex. by means of a link list or a linear array.

4.4.1 Stack implemented as an array

Implementation of array-based stack is very simple. It uses top variable to point to the topmost stack's element in the array. Initially the value of top is set to -1. Every time we execute push() operation the value of top is incremented by one and writes the pushed element to storage[top]. Incase of pop() operation if the value of top not equals to -1 then will return element storage[top] and the value of top will be decremented by one.

The only limitation of array based implementation of stack is that stack capacity is limited to a certain value and overfilling the stack will cause an error. Though, using the ideas from dynamic arrays implementation, this limitation can be easily avoided.

Algorithm of Push

Let stack[Max] is an array for implementing the stack.

TOP represents top of the stack.

1. [check for stack overflow?]

   If TOP = Max , then print : overflow and exit

2. Set TOP = TOP + 1

3. Set stack[TOP] := ITEM [insert item in new TOP position]

4. Exit

Algorithm of Pop

1. [check for stack underflow?]
   If top < 0 , then
   Print : underflow and exit
   Else [remove the top element]
   Set item := stack[TOP]
2. Decrement the stack top
   Set TOP := TOP - 1
3. Return the deleted item from the stack
4. Exit

Following is the C program code for the stack operations :

In this program code following notation is used :

Stack : name of the stack array

Maxsize : maximum size of the stack

Stacktop : top position of the stack

Val : element to be pushed

```
#include<stdio.h>
#define maxsize 100
int stack[maxsize];
int stacktop=-1;
void push(int);
int pop(void);
void display(void);
int main()
{
   int choice=0,val=0;
   do
   {
      printf("\n\t1.Push 2.Pop 3.Display \n\tSelect Your Choice : ");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         printf("\tElement to be Pushed : ");
         scanf("%d",&val);
         push(val);
         break;
         case 2:
         val=pop();
         if(val!=-1)
printf("\tPopped Element : %d\n",val);
break;
```

```
                    case 3:
            display();
            break;

            default:
            printf("\tWrong Choice");
            break;
        }
    }while(choice<=3);
    return 0;
}

void push(int val)
{
    stacktop++;
    if(stacktop<maxsize)
        stack[stacktop]=val;
    else
        printf("Stack Overflow");
}
int pop()
{
    int a;
    if(stacktop>-1)
    {
        a=stack[stacktop];
        stacktop--;
        return a;
    }
    else
    {
        printf("Stack is Empty");
        return -1;
    }
}
void display()
{
    int i=0;
    if(stacktop>-1)
    {
        printf("\tElements are:");
        while(i<=stacktop)
        {
            printf("\t%d",stack[i++]);
        }
        printf("\n");
    }
    else
        printf("\tStack is Empty\n");
}
```

4.4.2 Stack implemented as a Linked List

We can implement Stack using linked list as well. The dynamic allocation of memory makes it better choice for implementing Stacks. In this scenario, each stack element is composed of a data variable and a reference (in other words, a link) to the next node in the sequence. The push() operation simply creates a new node by allocating memory for it (Using malloc), and then linking it to a linked list as the new head. A pop() operation detach the node from the linked list, and assigns the stack pointer to the head of the previous second node. It also checks whether the list is empty before popping from it.

Following is the C program code for the stack operations :

```
#include<stdio.h>
#include<malloc.h>
#define maxsize 10
void push();
void pop();
void display();
struct node
{
   int info;
   struct node *link;
}*start=NULL, *new,*temp,*p;
typedef struct node N;
main()
{
   int choice,a;
   do
   {
      printf("\n\t1.Push 2.Pop 3.Display \n\tSelect Your Choice : ");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         push();
         break;

         case 2:
         pop();
         break;

         case 3:
         display();
         break;

         default:
         printf("\nInvalid choice");
         break;
```

```
            }
    }while(ch<=3);
}
void push()
{
    new=(N*)malloc(sizeof(N));
    printf("\nEnter the item : ");
    scanf("%d",&new->info);
    new->link=NULL;
    if(start==NULL)
        start=new;
    else
    {
        p=start;
        while(p->link!=NULL)
            p=p->link;
        p->link=new;
    }
}

void pop()
{
    if(start==NULL)
        printf("\nStack is empty");
    else if(start->link==NULL)
    {
        printf("\nThe deleted element is : %d",start->info);
        free(start);
        start=NULL;
    }
    else
    {
        p=start;
        while(p->link!=NULL)
        {
            temp=p;
            p=p->link;
        }
        printf("\nDeleted element is : %d\n", p->info);
        temp->link=NULL;
        free(p);
    }
}
void display()
{
```

```
    if(start==NULL)
       printf("\nStack is empty");
    else
    {
       printf("\nThe elements are : ");
       p=start;
       while(p!=NULL)
       {
          printf("%d",p->info);
          p=p->link;
       }
       printf("\n");
    }
}
```

4.5 Concept of Overflow and Underflow

As we have already discussed, the pushcl operation simply inserts a new element to the top of the stack. But what happens when we try to insert a new element in to stack and there is no space available for this new element. Such type of state of stack considered to be in an overflow state.

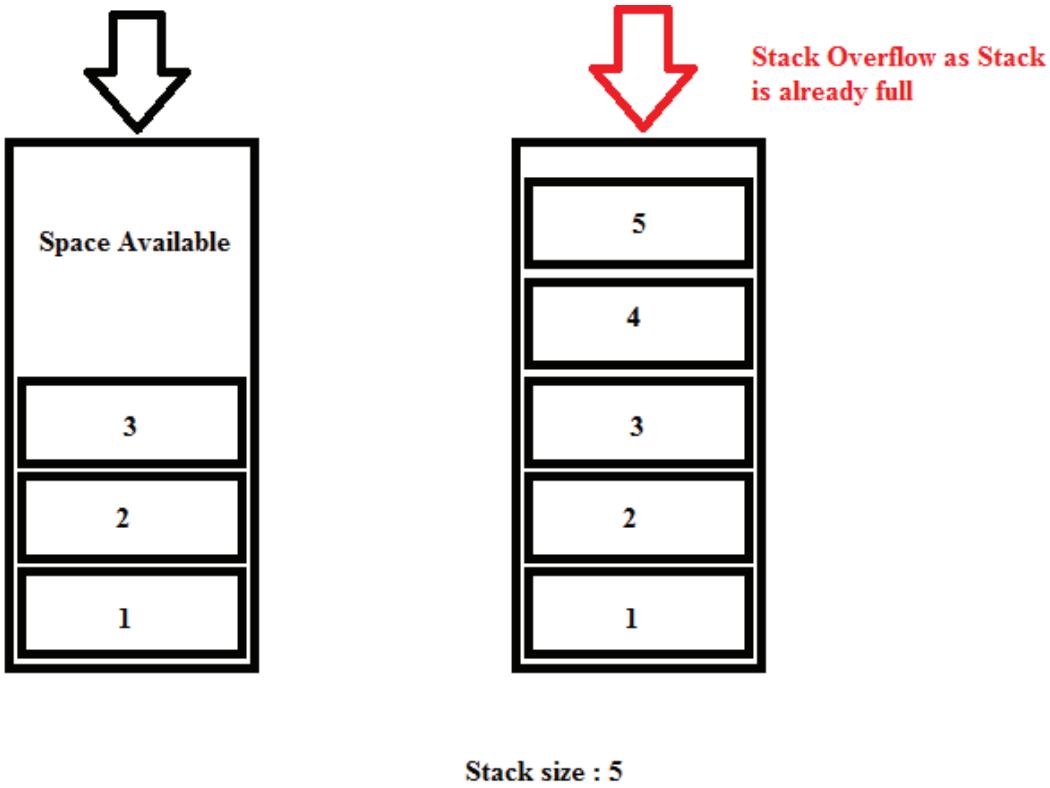


Figure 4.2 : Stack Overflow

When we execute a pop operation on stack, it simply removes the element which is available at the top of the stack. But if the stack is already empty and we try to execute pop() it goes into underflow state. It means no items are present in stack to be removed

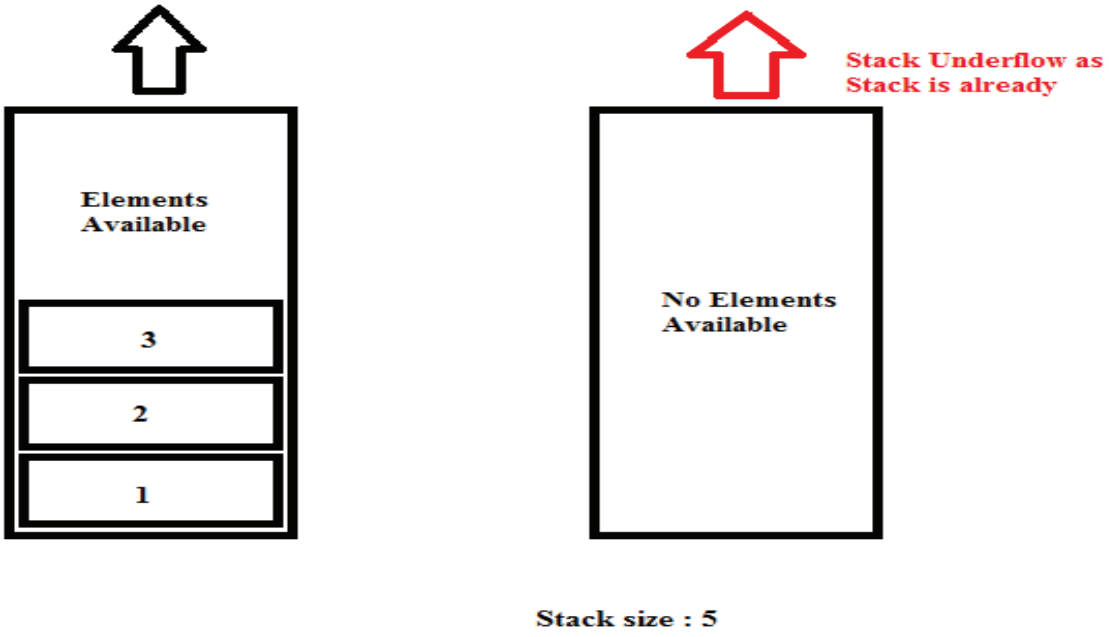


Figure 4.3 : Stack Underflow

4.6 Applications of Stack

Stacks are used mainly in following applications :

1. Function Calling

Programs compiled from high level languages make use of a stack for the working memory of each function or procedure calling. When any procedure or function is called, function parameters and address is pushed into the stack and whenever procedure or function returns, parameters are popped. As a function calls other function, first its argument, then the return address and finally space for local variables is pushed onto a stack. A function call itself, is known as recursion.

```
Function a()
{
    if (condition)
            return
    …….
        ……..
      return b();
    }
Function b()
{
    if (condition)
            return
    …….
        ……..
      return a();
    }
```

Note how all the function a and b's environment are found in the stack. When a is called a second time from b, a new set of parameters is pushed in the stack on invocation of a.

The stack is a region of main memory within which programs temporarily store data as they execute. For example, when a program sends parameters to a function, the parameters are placed on the stack. When the function executes, these parameters are popped from the stack. When a function calls other functions, current contents of the caller function are pushed onto the stack with the address of the instruction just next to the call instruction, this is done so that after the execution of called function, the compiler can track back the path from where it is sent to the called function.

2. To convert an infix expression to postfix

In high level languages, infix notation cannot be used to evaluate expressions. We must analyze the expression to determine the order in which we evaluate it. A common technique is to convert an infix notation into postfix notation, then evaluating it.

- Prefix: + a b
- Infix: a + b
- Postfix: a b +

Infix to postfix conversion can be done easily using stack.

Following are the rules for the infix to postfix conversion of an arithmetic expression:

Rules:

- Operands immediately go directly to output
- Operators are pushed into the stack (including parenthesis)
- Check to see if stack top operator is less than current operator (Priority of the operator)
- If the top operator is less than, push the current operator onto stack
- If the top operator is greater than the current, pop top operator and push onto stack, push current operator onto stack
- Priority 2: * /
- Priority 1: + -
- Priority 0: (

If we encounter a right parenthesis, pop from stack until we get matching left parenthesis. Do not output parenthesis.

Example 1 :

A + B * C - D / E

Infix	Stack(bot->top)	Postfix
a) A + B * C - D / E		
b) + B * C - D / E		A
c) B * C - D / E	+	A
d) * C - D / E	+	A B
e) C - D / E	+ *	A B
f) - D / E	+ *	A B C
g) D / E	+ -	A B C *

h) / E + - A B C * D

i) E + - / A B C * D

j) + - / A B C * D E

k) A B C * D E / - +

Example 2 :

A * B - (C + D) + E

	Infix	Stack(bot->top)	Postfix
a)	A * B - (C - D) + E	empty	empty
b)	* B - (C + D) + E	empty	A
c)	B - (C + D) + E	*	A
d)	- (C + D) + E	*	A B
e)	- (C + D) + E	empty	A B *
f)	(C + D) + E	-	A B *
g)	C + D) + E	- (	A B *
h)	+ D) + E	- (	A B * C
i)	D) + E	- (+	A B * C
j)	) + E	- (+	A B * C D
k)	+ E	-	A B * C D +
l)	+ E	empty	A B * C D + -
m)	E	+	A B * C D + -
n)		+	A B * C D + - E
o)		empty	A B * C D + - E +

3. To evaluate a postfix expression

The expression may be evaluated by making a left to right scan, stacking operands and evaluating operators using the correct number of operands from the stack and finally placing back the result on to the stack.

Algorithm for the evaluation of postfix expression :

1. Scan arithmetic expression from left to right and repeat step 2 and 3 until complete expression is processed
2. If an operand is found push it into stack
3. If an operator is found
   i. Remove two top elements of stack, where X is top element and Y is next-to-top element.
   ii. Evaluate X operator Y
   iii. Place the result of step 3(ii) into stack at top

Example :

Postfix Expression : 1 2 3 + *

Postfix	Stack(bot -> top)	
a)	1 2 3 + *	
b)	2 3 + *	1
c)	3 + *	1 2
d)	+ *	1 2 3
e)	*	1 5 // 5 from 2 + 3
f)		5 // 5 from 1 * 5

4.7 Summary

Stack is a LIFO structure. A stack is an ordered list in which all insertions and deletions are made at one end, called the top. The restrictions on a stack imply that if the elements A,B,C,D,E are added to the stack, then the first element to be removed/deleted must be E. Equivalently we say that the last element to be inserted into the stack will be the first to be removed. For this reason stacks are sometimes referred to as Last In First Out (LIFO) lists.

Index which is available for insertion and deletion in a stack is known as TOP of the stack, both the operations always start with this. Insertion of an element in a stack is known as PUSH and deletion of an element is known as POP operation. If stack is empty and somebody is trying to delete an element from this index then this is stack underflow condition, and if stack is full and somebody is trying to insert in this stack then this is overflow condition.

Stacks can be implemented either using arrays or link lists. Stack implementation using an array is only useful when we have fixed amount of data, because array has its own limitations in terms of size, so if size is not fixed one should go for linked list implementation.

Stacks are used in many applications like function calling, arithmetic expression evaluation etc.

4.8 Self Assessment Questions

1. What is a stack? Write about some real life applications of stack.
2. What is the difference between stack overflow and underflow condition?
3. Write C code for the conversion of infix into postfix expression.
4. Write C code for the evaluation of postfix expression.
5. Discuss the following terms with reference to stack :
   - TOP
   - PUSH
   - POP

Unit - 5 : Queues

Structure of Unit:

5.0 Objectives

This chapter covers

- Introduction to Queue
- Concept of various types of Queues
- Various queue operations
- Applications of queue

5.1 Introduction

Queue is a linear data structure in which data can be added to one end and retrieved from the other. Just like the queue of the real world, the data that goes first into the queue is the first one to be retrieved. That is why queues are sometimes called as **First-In-First-Out** data structure.

In case of stacks, we saw that data is inserted and deleted from one end but in case of Queues, data is added to one end (known as REAR) and retrieved from the other end (known as FRONT).

Few points regarding Queues:

1. **Queues:** It is a linear data structure; linked list and array can represent it.
2. **Rear:** A variable stores the index number in the queue at from which the new data will be added (in the queue).
3. **Front:** It is a variable storing the index number in the queue where the data will be retrieved.

The cafeteria line is one type of queue. Queues are often used in programming networks, operating systems, and other situations in which many different processes must share resources such as CPU time.

One bit of terminology: the addition of an element to a queue is known as an **enqueue**, and removing an element from the queue is known as a **dequeue**.

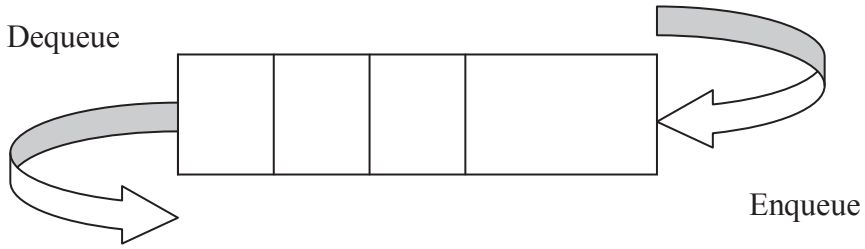


Figure 5.1 : queue enqueue and dequeue operation

Now we look at the process of adding and retrieving data in the queue with the help of an example.

Suppose we have a queue represented by an array queue [10], which is empty to start with. The values of front and rear variable upon different actions are mentioned in {}.

queue [10]=EMPTY {front=-1, rear=0}

add (5)

Now, queue [10] = 5 {front=0, rear=1}

add (10)

Now, queue [10] = 5, 10 {front=0, rear=2}

retrieve () [It returns 5]

Now, queue [10] = 10 {front=1, rear=2}

retrieve () [now it returns 10]

Now, queue [10] is again empty {front=-1, rear=-1}

In this way, a queue like a stack, can grow and shrink over time.

5.2 Types of Queue

Queue is a FIFO data structure, it supports the insert and remove operations using a FIFO discipline. A waiting line is a good real-life example of a queue. (In fact, the British word for "line" is "queue".)

Few examples of queues are :

- An electronic mailbox is a queue
- The ordering is chronological (by arrival time)
- A waiting line in a store, at a service counter, on a one-lane road
- Equal-priority processes waiting to run on a processor in a computer system

There are various types of queues according to their structure. Following are the examples :

5.2.1 Circular Queue

5.2.2 Priority Queue

5.2.3 Double ended Queue

5.2.1 Circular Queue :

When a new item is inserted at the rear in queue, the pointer to rear moves upwards. Similarly, when an item is deleted from the queue the front arrow moves downwards. After a few insert and delete operations the rear might reach the end of the queue and no more items can be inserted although the

items from the front of the queue have been deleted and there is space in the queue. (As shown in following figer. To solve this problem, queues implement wrapping around. Such queues are called Circular Queues. Both the front and the rear pointers wrap around to the beginning of the array. It is also called as "Ring buffer".

In the circular queue the location of front and rear is same. each element has its position no. for insertion , if we set the 5th element as the front element then after every insertion the pointer indicates the 5th element as front. In deletion, the fifth element is deleted every time it is the rear position. After deletion of an element the queue rotates and every time the rear indicates the 5th element of the circular queue and every time the 5th location element is deleted.

5.2.2 Priority Queue :

Here is a conceptual view of a priority queue:

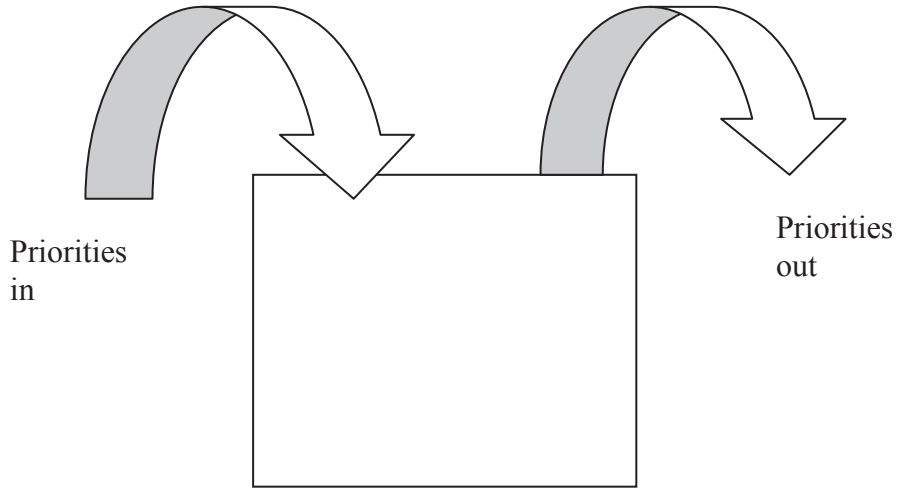


Figure 5.2 : Priority Queue

Think of a priority queue as a box that holds priorities. You can put one in, and you can take out the highest priority. (Priorities can be any Comparable values) Priority queue is different from a "normal" queue, because instead of being a "first-in-first-out" data structure, values come out in order by priority. A priority queue might be used, for example, to handle the jobs sent to the Computer Science Department's printer: Jobs sent by the department chair should be printed first, then jobs sent by professors, then those sent by master students, and finally those sent by graduate students. The values put into the priority queue would be the priority of the sender (e.g., using 4 for the Dean, 3 for Head, 2 for master students, and 1 for graduate), and the associated information would be the document to print. Each time the printer is free, the job with the highest priority would be removed from the print queue, and printed. (Note that it is OK to have multiple jobs with the same priority; if there is more than one job with the same highest priority when the printer is free, then any one of them can be selected.)

So priority queue is the queue that stores items which have priority and in this High-priority data is out first, not depending on FIFO order.

5.2.3 Double ended Queue

Double-ended queue (**dequeue**, often abbreviated to **deque**, pronounced *deck*) is an abstract data type that implements a queue for which elements can only be added to or removed from the front (head) or back (tail). Items can be inserted and deleted from either ends. This is more versatile data structure than stack or queue.

Figure 5.3 Double ended queue

Dequeue differs from the queue abstract data type or First-In-First-Out List, where elements can only be added to one end and removed from the other. This general data class has some possible sub-types:

An input-restricted deque is one where deletion can be made from both ends, but insertion can only be made at one end.

An output-restricted deque is one where insertion can be made at both ends, but deletion can be made from one end only.

Both the basic and most common list types in computing, queues and stacks can be considered specializations of deques, and can be implemented using deques.

Figure 5.4 : Input restricted Double ended Queue

Figure 5.5 : Output restricted Double ended Queue

5.3 Queue Operations

Main operations which can be performed over any queue is insert and delete. Theoretically, one characteristic of a queue is that it does not have a specific capacity. Regardless of how many elements are already contained, a new element can always be added. It can also be empty, at which point removing an element will be impossible until a new element has been added again.

Fixed length arrays are limited in capacity, and inefficient because items need to be copied towards the head of the queue. Queue **overflow** results from trying to add an element onto a full queue and queue **underflow** happens when trying to remove an element from an empty queue.

Queue operations

Operations on queue are :

1. Insert - insert item at the back of queue
2. Delete - return (and virtually remove) the front item from queue
3. Traverse/Display- displays queue elements in sequence

There are several efficient implementations of FIFO queues. An efficient implementation is one that can perform the operations — enqueuing and dequeuing . Queues can be implemented using array structure or link list.

Array implementation :

A queue can be implemented with an array, as shown here. For example, this queue contains the integers 4 (at the front), 8 and 6 (at the rear).

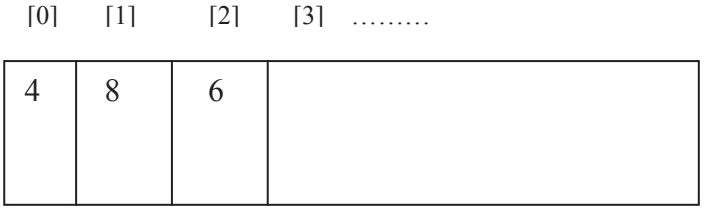


Figure 5.6 : An array of integers to implement a queue of integers

Here front is 0 and rear is 2, elements are inserted at rear and deleted from the front, when an element is inserted rear is incremented and when an element is deleted front is changed. If a new element 9 is added in a queue rear becomes 3 and if an element is deleted from the queue, front becomes 1.

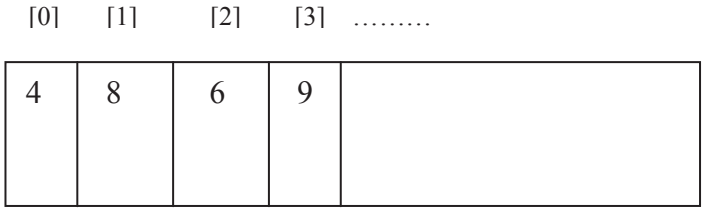
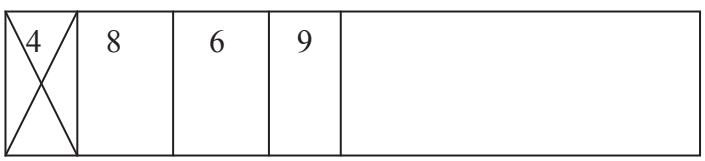


Figure 5.7 : Insertion operation in Queue



deleted element

Figure 5.8 : Deletion operation in Queue

Following is the code for queues various operations. Here following variables are used :

Queue : Name of the array which is behaving as the queue.

Front : denotes head of the queue.

Rear : denotes end of the queue.

```
/
*****************************************************************/

/*  C program to implement a Queue Implementation using
array                          */

/
*****************************************************************/


#include<stdio.h>

#define MAX 50

int queue_arr[MAX];

int rear = - 1;

int front = - 1;


main()

{

    int choice;

    while (1)

    {

        printf("1.Insert\n");

        printf("2.Delete\n");

        printf("3.Display\n");

        printf("4.Quit\n");

        printf("Enter your choice : ");

        scanf("%d", &choice);

        switch (choice)

        {

            case 1:

            insert();
```

```
                    break;
            case 2:
            del();
            break;
            case 3:
            display();
            break;
            case 4:
            exit(1);
            default:
            printf("Wrong choice\n");
        } /*End of switch*/
    } /*End of while*/
} /*End of main()*/
```

/* insert operation */

```
insert()
{
    int added_item;
    if (rear == MAX - 1)
    printf("Queue Overflow\n");
    else
    {
        if (front == - 1)
        /*If queue is initially empty */
        front = 0;
        printf("Input the element for adding in queue : ");
        scanf("%d", &added_item);
        rear = rear + 1;
        queue_arr[rear] = added_item;
    }
} /*End of insert()*/
```

/* Delete operation */

```
del()
{
    if (front == - 1 || front > rear)
    {
        printf("Queue Underflow\n");
        return ;
    }
    else
    {
```

```
        printf("Element deleted from queue is : %d\n", queue_arr[front]);
        front = front + 1;
    }
} /*End of del() */
```

/* traverse or display operation */

```
display()
{
    int i;
    if (front == - 1)
    printf("Queue is empty\n");
    else
    {
        printf("Queue is :\n");
        for (i = front; i <= rear; i++)
        printf("%d ", queue_arr[i]);
              printf("\n");
         }
} /*End of display() */
```

Following are the characteristics of array implementations :

- Easy to implement
- But it has a limited capacity with a fixed array
- Or you must use a dynamic array for an unbounded capacity

Linked List implementation

A queue can also be implemented with a linked list with both a head and a tail pointer.

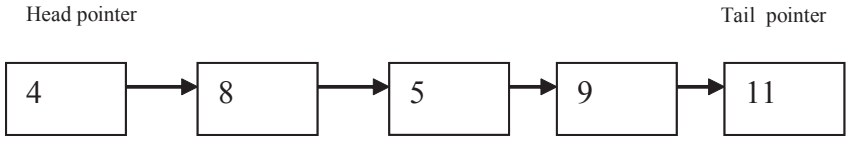


Figure 5. 9 : A link list to implement a queue of integers

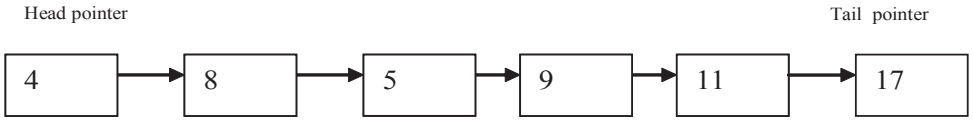


Figure 5.10 : Insertion operation in Link list as queue

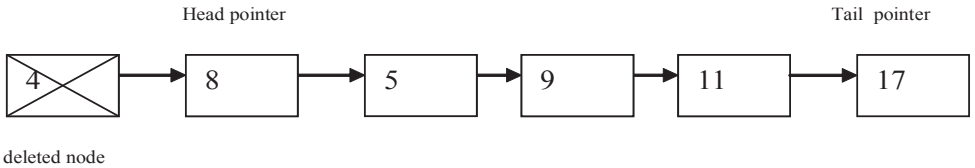


Figure 5.11 : Deletion operation in Link list as queue

The head pointer points to the front of the list, and tail pointer points end of the list, whenever there is insertion, tail pointer is changed and points to the new node and whenever there is deletion head pointer is changed.

Following is the code for queues various operations. Here following variables are used :

Node : Name of the node in queue.

front : pointer denotes head of the queue.

rear : pointer denotes end of the queue.

```
/*****************************************************************/
/*  C program to implement a Queue Implementation using link list                */
/*****************************************************************/

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

struct list{
        char data;
        struct list *next;
};
/*****  Redefining struct list as node  *****/
typedef struct list node;
void qinsert(node**,node**); /** Inserting character function in queue **/
void qdelete(node**,node**); /** Deleting character function in queue **/
void disp(node*);          /** Output displaying function **/

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *\
|*"front" and "rear" pointer needed to be changed when inserting and *|
|*  deleting so addresses of them is passed in qinsert and qdelete   *|
\* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

void main()
{
   int opt;       /* Option inputing variable */
   char ch;       /* choice inputing variable */
   node *front;    /* front pointer in queue*/
   node *rear;     /* rear pointer in queue */
   rear=front=NULL;
   do
   {
      printf("\nEnter your option to perform in a queue\n");
      printf("\n1. Insert\n");
      printf("\n2. delete\n");
      printf("\n3. Display the list\n");
      scanf("%d",&opt);
      switch(opt)
```

```
        {
        case 1:
            qinsert(&front,&rear);
            break;
        case 2:
            qdelete(&front,&rear);
            break;
        case 3:
            disp(front);
            break;
        }
        printf("\nDo you wish to continue[y/n]\n");
        ch=(char)getche();
    }while(ch=='Y' || ch=='y');
    printf("\nPress any key to exit\n");
    getch();
}

void qinsert(node **front,node **rear)
{
    node *newnode;      /* New node to be inserted */
    newnode=(node*)malloc(sizeof(node));
    newnode->next=NULL;
    printf("\nEnter the character to push\n");
    fflush(stdin);
    scanf("%c",&(newnode->data));
    if(*front==NULL && *rear==NULL)
    {
        *front=newnode;
        *rear=newnode;
    }
    else
    {
    /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *\
    |*  "rear" points to last node whose next field must be pointed to *|
    |*  newnode and then rear points to the latest node i.e. new node  *|
    \* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
        (*rear)->next=newnode;
        *rear=newnode;
    }
}

void qdelete(node **front,node **rear)

{
    node *delnode;      /* Node to be deleted */
```

```
    if((*front)==NULL && (*rear)==NULL)
        printf("\nQueue is empty to delete any element\n");
    else
    {
        delnode=*front;
        (*front)=(*front)->next;
        free(delnode);
    }
}
void disp(node *f)
{
    while(f!=NULL)
    {
        printf(" %c ",f->data);
        f=f->next;
    }
}
```

5.4 Applications of Queue

In general, queues are often used as "waiting lines". Here are a few examples of where queues would be used:

- When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling. In operating systems, for controlling access to shared system resources such as printers, files, communication lines, disks and tapes.

A specific example of print queues follows:

- In the situation where there are multiple users or a networked computer system, you probably share a printer with other users. When you request to print a file, your request is added to the print queue. When your request reaches the front of the print queue, your file is printed. This ensures that only one person at a time has access to the printer and that this access is given on a first-come, first-served basis.
- For simulation of real-world situations. For instance, a new bank may want to know how many tellers to install. The goal is to service each customer within a "reasonable" wait time, but not have too many tellers for the number of customers. To find out a good number of tellers, they can run a computer simulation of typical customer transactions using queues to represent the waiting customers.
- When placed on hold for telephone operators. For example, when you phone the toll-free number for your bank, you may get a recording that says, "Thank you for calling A-1 Bank. Your call will be answered by the next available operator. Please wait." This is a queuing system.

5.5 Summary

A queue is a FIFO i.e first in first out, in queue there are front and rear. Front is the initial or first location of the queue where rear indicates the last entry location in the queue.

There are two operations mainly which can be performed over any queue, one is addition of an element to a queue is known as an **enqueue**, and other is removing an element from the queue is known as a **dequeue**.

Queues are mainly of three types : Circular queue, Priority queue and Double ended queue.

Circular queue is a queue where logically no end no start and you are independent of insert and delete operations.

Priority queue is a queue in which insertion and deletion takes place according to their priorities, jobs are arranged in a queue according to the priorities and highest priority jobs are getting deletion first. This type of queue is used in scheduling programs.

Double ended queue is a queue where insertion and deletion both can take place from either end according to the requirements i.e. from front or rear.

Queues can be implemented either using array or link list structure. Fixed length arrays are limited in capacity, and link list provides dynamic structure in terms of size.

Queues can be used in many applications : Typical uses of queues are in simulations and operating systems.

- Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.
- Computer systems must often provide a "holding area" for messages between two processes, two programs, or even two systems. This holding area is usually called a "buffer" and is often implemented as a queue.

5.6 Self Assessment Questions

1. What do you understand by queue data structure? Discuss its examples.
2. Describe different types of queues and how are they used in different applications?
3. What are the various operations of queue?
4. Write C code for the insert and delete operation in circular queue?

Unit - 6 : Trees

Structure of Unit:

6.0 Objectives

This chapter covers:

- Introduction to the terms which are used in this unit such as tree, binary tree, binary search tree, degree, depth etc.
- Concept of different types of trees such as balance tree, complete tree, full binary tree, skew tree etc.
- Brief exposure to array representation of complete binary tree.

6.1 Introduction

In computer science, a tree is a widely-used data structure that emulates a hierarchical tree structure with a set of linked nodes. Mathematically, it is an ordered directed tree

Searching operation must be linear in case of sorted list and its time complexity is proportional to list length if list of items arranged in linked list. Insertion and deletion operation is also not very efficient in this list. So, there is a need of a data structure which perform insertion and deletion efficiently and also reduce time required of searching operation Tree data structure can fulfill this need since searching in tree is of logarithm time complexity.

A *tree* data structure is a powerful tool for organizing data objects based on keys. It is equally useful for organizing multiple data objects in terms of hierarchical relationships (think of a “family tree”, where the children are grouped under their parents in the tree).

Here is an example of a tree of species, from zoology:

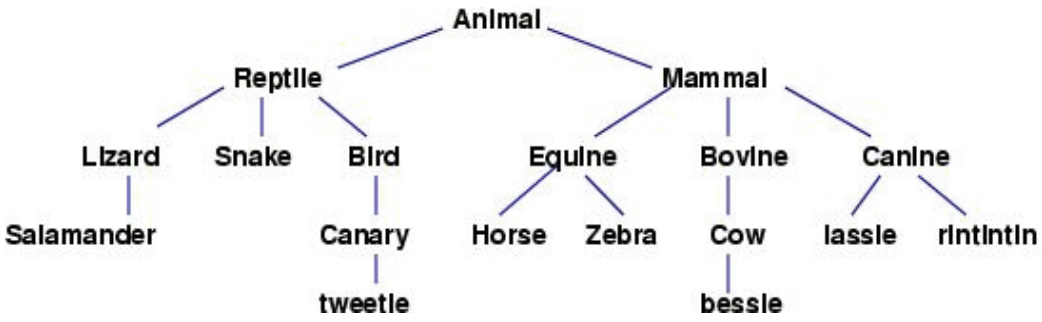


Figure 6.1 : Example of tree of species

6.2 Definitions

Before going into detail we should be familiar with some terms related to tree data structure which are given below :-

6.2.1 Non-linear data structures

A linear data structure traverses the data elements sequentially, in which only one data element can directly be reached. Ex: Arrays and Linked Lists, **whereas in non-Linear data structure** every data item is attached to several other data items in a way that is specific for reflecting relationships. The data items are not arranged in a sequential structure. Ex: Trees, Graphs

6.2.2 General tree

An acyclic connected graph where each node has zero or more *children* nodes and there is at most one *parent* node, and at most one node has no parents, which is root node of the tree.

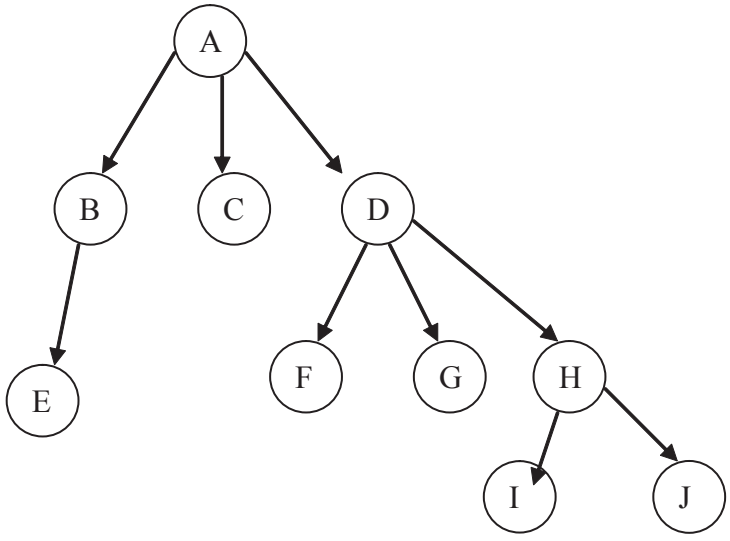


Figure 6.2 : General Tree

6.2.3 Binary Tree

In Figure 6.2 given structure is a tree but not Binary tree since A and D have more than two children.

A **binary tree** is a tree in which each node can have maximum two children. Thus each node can have no child, one child or two children. The pointers help us to identify whether it is a left child or a right child

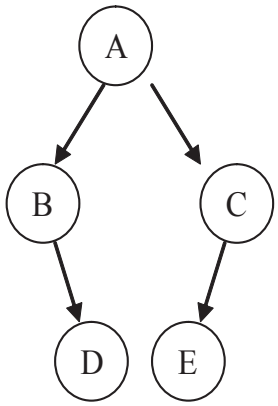


Figure 6.3 : A Binary Tree

6.2.4 Binary Search Tree

A *binary search tree* (BST) is a binary tree where each node has a `Comparable` key (and an associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left subtree and smaller than the keys in all nodes in that node's right subtree.

In above Figure 6.3, we have Binary tree but not Binary Search tree since value at left child is greater than value at parent node.

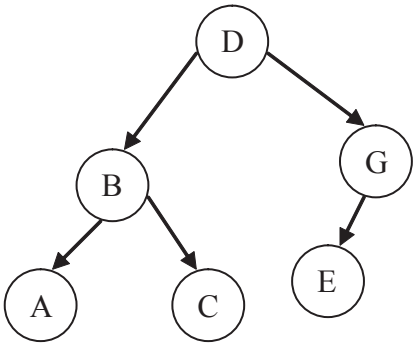


Figure 6.4 : A Binary search Tree

6.2.5 Height balance tree

A **balanced binary tree** is commonly defined as a binary tree in which the height of the two subtrees of every node never differ by more than 1. Binary trees that are balanced according to this definition have a predictable depth (how many nodes are traversed from the root to a leaf.

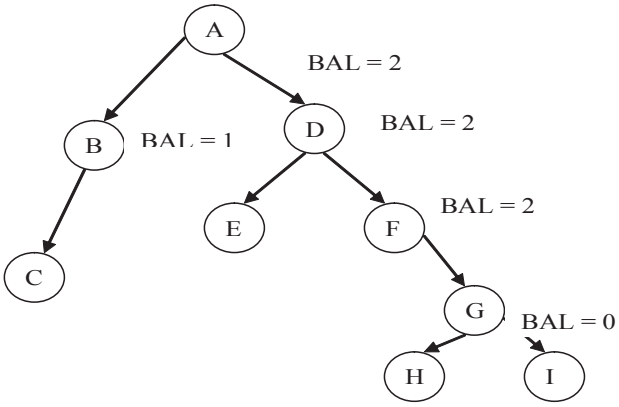


Figure 6.5 : An Unbalanced Tree

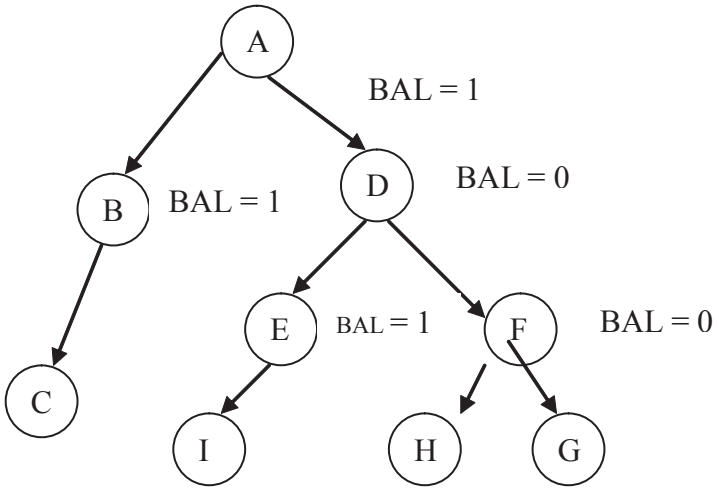


Figure 6.6 : A Balanced Tree

Figure 6.6 represents balanced tree since it has absolute difference between height of left subtree and right subtree not greater than one.

6.2.6 AVL tree

An **AVL tree** is another balanced binary search tree. Named after their inventors, **A**delson-**V**elskii and **L**andis.

An AVL tree is a binary search tree which has the following properties:

1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.

Figure 6.6 is an example of AVL tree.

6.3 Tree Terminology

Before going in to detail it is good to be familiar with some terminology related to tree which will be used frequently.

Indegree : Number of branches directed toward the node is known as indegree of node.

Outdegree : Number of branches directed away from the node is knows as outdegree of node.

Degree of node: Number of branch associated with a node is called degree of node. Sum of indegree and outdegree of node is degree of node.

Root node : First node of tree is root node which have zero in degree.

Leaf node : Node of out degree zero.

Internal node : Node that is neither root nor leaf node is internal node

Parent : node is parent if its outdegree is greater than or equal to one.

Siblings : node whose parents are same, called sibling.

Path : sequence of nodes in which each node is adjacent to next node.

Ancestors	:	any node in the path from root to the node
Descendents	:	all the nodes in the path from given node to a leaf.
Level	:	distance from root.
Height/depth	:	level of the leaf in longest path from root plus one
Subtree	:	any connected structure below root
Degree of tree	:	maximum degree of any node in the tree is degree of tree.
Forest	:	Collection of trees

These terms can be understood clearly by following example of tree.

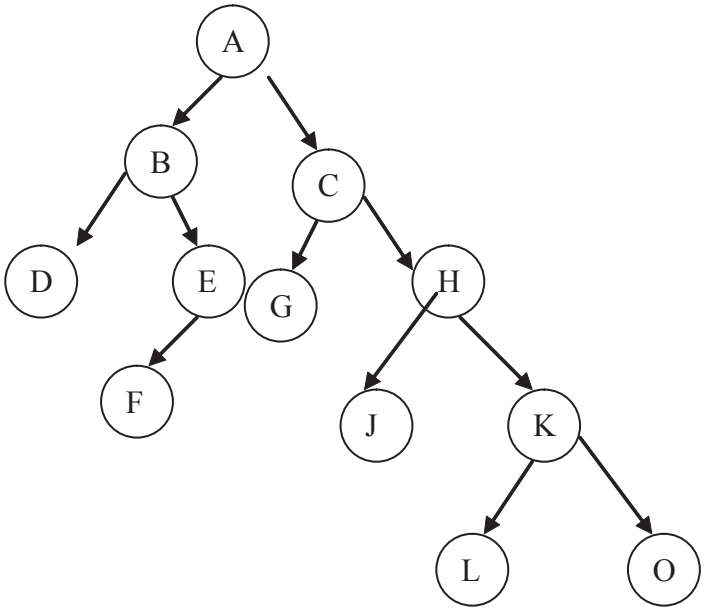


Figure 6.7 : An example of binary tree

Indegree of H is one, where as Outdegree of K and E is two and one respectively. Degree of this tree is two. A is root node where as D,F, G, J, L and O are leaf nodes, nodes B, C, E, H and K are internal nodes, all internal and roots nodes are parent node also of their child node corresponding. Node (B , C), (D , E), (G , H) (J , K) and (L , O) are siblings. Node E, B and A are ancestors of node F, node J, K, L and O are descendents of node H. Node A is level zero, B and C are at level 1 and so on. Heigh/ depth of this tree is 4

6.4 Heap

Heap is an ordered balanced binary tree in which the value of the node at root of any subtree is less than or equal to the value of either of its children This implies that only valued item may be placed in the heap. A heap satisfying this definition is called min-heap. This implies that an element with the lowest value is always in the root node of tree.

If the value of the node at root of any subtree is greater than or equal to the value of either of its children then heap is called max-heap. This implies that an element with the greatest value is always in the root node of tree.

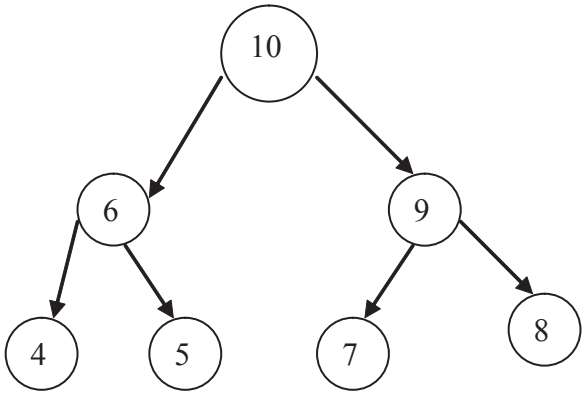


Figure 6.8 (a) : Max-Heap

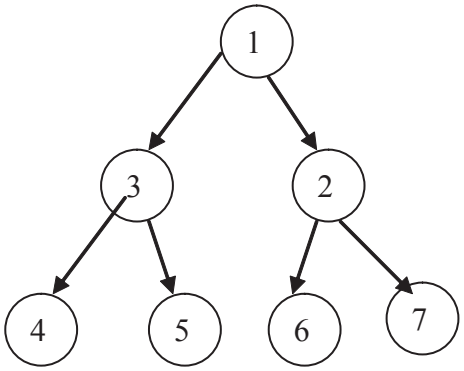


Figure 6.8 (b) : Min-Heap

6.5 Complete Binary Tree

A binary tree T with n levels is *complete* if all levels except possibly the last are completely full, and the last level has all its nodes to the left side.

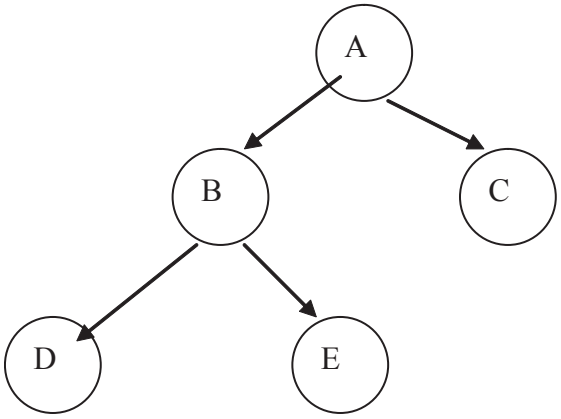


Figure 6.9 : Complete Binary Tree but not Full

6.6 Full binary Tree

A *binary tree* in which each *node* has exactly zero or two *children* is known as full binary tree. It is **also known as proper binary tree.**

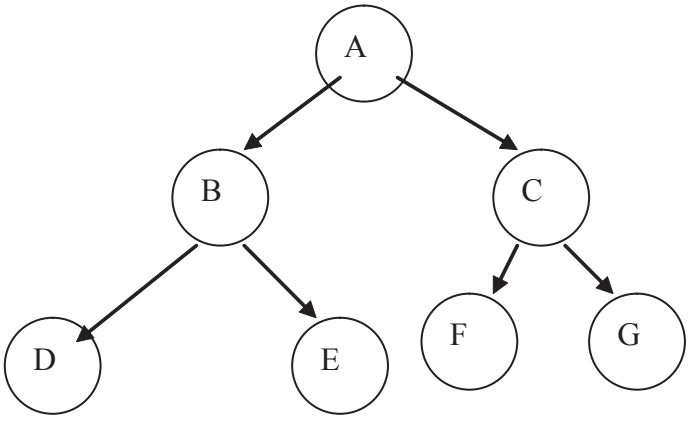
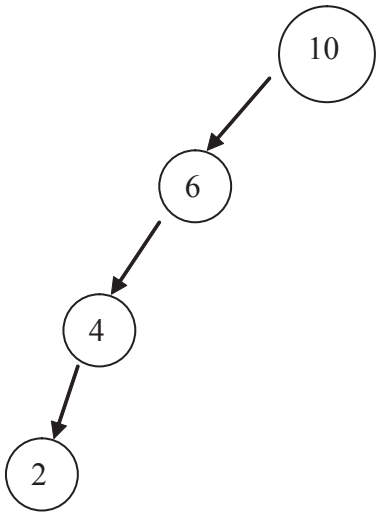


Figure 6.10 : Full binary tree also complete binary tree

6.7 Skew tree

A binary tree in which each node is having either only left sub-tree or no sub-tree is called as left skewed binary tree.

A binary tree in which each node is having either only right sub-tree or no sub-tree is called as right skewed binary tree.



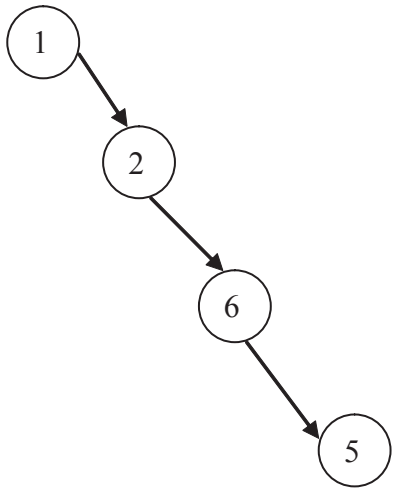


Figure 6.11 (a) Left Skewed binary tree **Figure 6. 11 (b) Right Skewed Binary tree**

6.7 Array Representation of Complete Binary Tree

Full binary tree of h height will have $2^h - 1$ nodes see figure 6. 10, here height is 3 and full binary tree has 7 nodes.

In binary tree numbering of nodes start from 1 through $2^h - 1$ top to bottom from left to right.

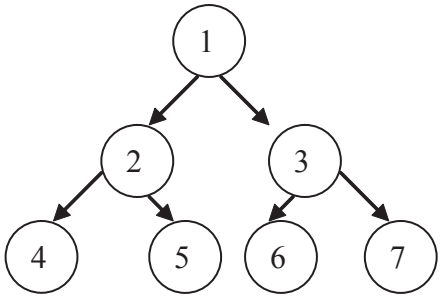


Figure 6.12 : Numbering scheme

Parent of i^{th} node is node i/2 unless node i = 1.

Left child of node i is node 2i unless 2i > n where n is number of nodes. If 2i > n node i has no left child.

Right child of node i is node 2i + 1 unless 2i + 1> n. . If 2i +1 > n node i has no right child.

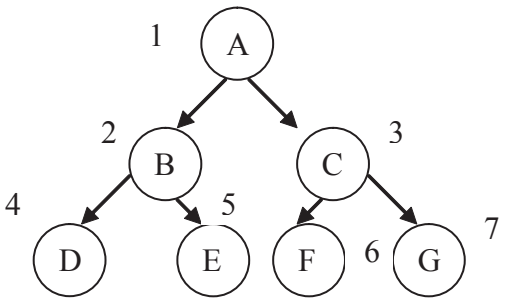


Figure 6.13 : numbered full binary tree

Array representation : Number the nodes using this numbering scheme the node that is number I is stored at tree[I]. Array representation of tree in figure 6.13 is as follows.

1	2	3	4	5	6	7
A	B	C	D	E	F	G

If the tree is a complete binary tree, this method wastes no space This method benefits from more compact storage and better locality of reference (i.e. same value or related storage locations being frequently accessed.). However, it is expensive to grow and wastes space proportional to 2^h - n for a tree of height h with n nodes.

6.9 Summary

Tree is a data structure which is used to represent the data containing a hierarchical relationship between the elements, some examples of trees are : family tree, the hierarchy of positions in a company, an algebraic expression involving operations for which certain rules of precedence are prescribed etc.

There are various type of trees, which can be used according to the requirements such as Binary tree, AVL tree, Binary search tree etc.

Arrays can be used for the implementation of binary search tree. Arrays can be used efficiently, if it is complete binary search tree otherwise there is wastage of memory are.

Trees are mainly used in searching operation, other than this trees are also used in arithmetic expression evaluation. Insertion and deletion operations are also more efficient in tree, in comparison to linear lists.

6.10 Self Assessment Questions

1. Define tree.
2. What is the difference between skewed binary tree and binary search tree?
3. Write about following with reference to tree.
   - Height
   - Degree
   - Depth
4. What is AVL tree? Explain.
5. Write C code for the array implementation of complete binary tree.

Unit - 7: Linked Implementation of Binary Search Tree

Structure of Unit:

7.0 Objectives

This chapter covers the basic binary search tree construction techniques with the following operations.

- Insertion
- Deletion
- Traversing: Preorder, inorder, postorder
- Searching
- Sorting

7.1 Creating of BST

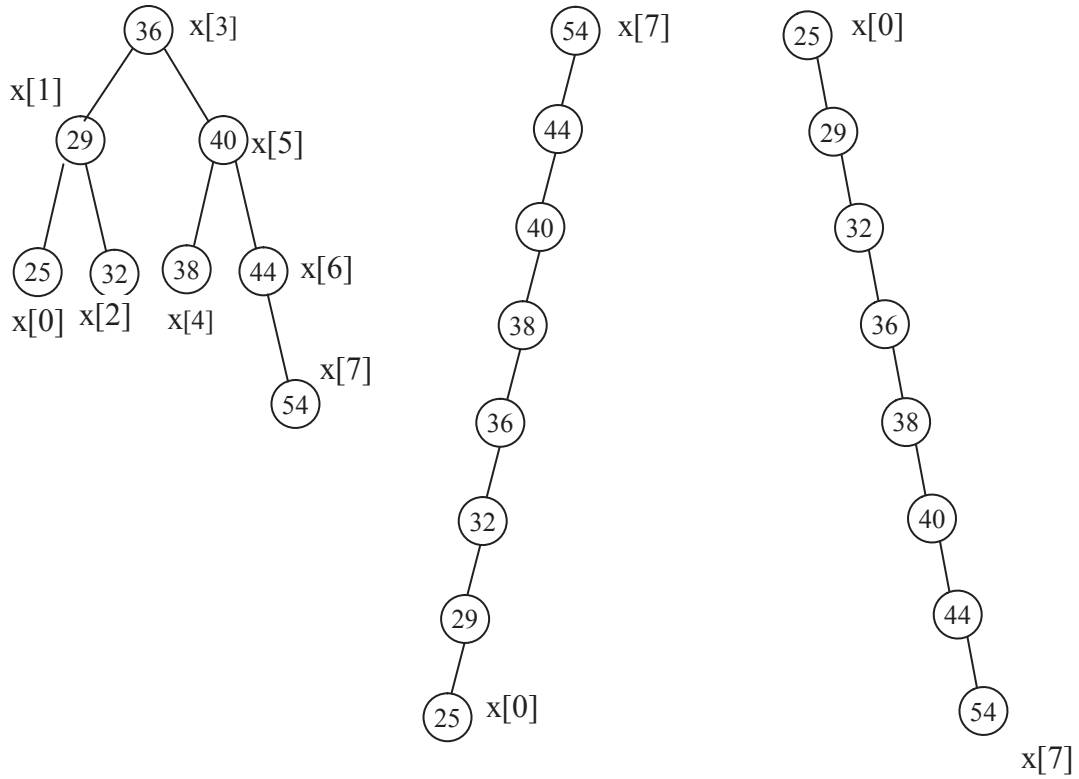
A binary search tree has the following properties

- binary tree
- All items in the left subtree are less than the root
- All items in the right subtree are greater than to the root
- Each subtree is also a binary search tree (i.e., recursive definitions)
- Does not allow duplicate values.

The inorder traversal of such a binary tree gives the set elements in ascending order. A sorted array can be produced from a BST by traversing the tree in inorder and inserting each element sequentially into array as it is visited. It may possible to construct many BSTs from given sorted array. Some of the possible BSTs with different root is given below figure 7.1.

Consider a sorted array

	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
x[ ]=	25	29	32	36	38	40	44	54



(a) BST representation BST balanced

(b) BST left heavy unbalanced

(c) BST right heavy unbalanced

Figure 7.1

7.2 Operations of BST

7.2.1 Insertion In BST

The following algorithm searches as BST and inserts a new key into the tree if the search is unsuccessful into figure 7.1 (a).

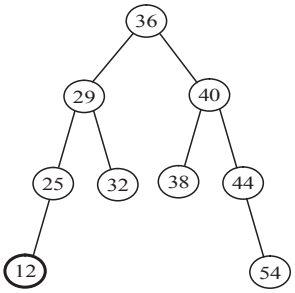


Figure 7.1(d) Insert a new element whose value is 12

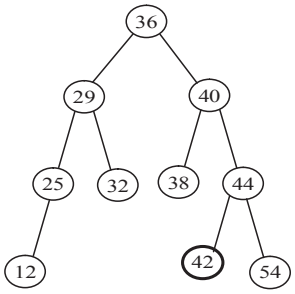


Figure 7.1 (e) Insert a new element whose value is 42

Note that after a new key is inserted, the tree retains the property of being sorted in an inorder traversal.

Insertion into a Binary search tree

To insert a new element x, we must first verify that its key is different from those of existing nodes.

```
Strict - node
{ int     info:
Shrict node * lechild
              * rchild
}. *p;
Algorithm BTREEINS(x)
Step1:  [initialize variables]
        Set found =  0; set p = root;
Step2:  Repeat Step3 while (p != NULL) and ( !found)
Step3:          parent  = p;
if (p->Info = x) then
Set found = 1;
Else if (x < p->Info) then
Set p = p->lchild;
Else
Set p = p->rchild;
        [end of while loop at Step2]
Step4: If (!found) then
        Set p  =  allocate memory for a node of binary tree
                Set p->lchild = NULL;
        Set p->Info = x;
                Set p->rchild = NULL;
Step5:          If (root != NULL) then
                        If ( x <parent->Info) then
                                Set parent->lchild = p;
                        Else
                                Set parent->rchild = p;
                        [End of If Structure]
```

Else

Root = p;

[End of If Structure of Step5]

[End of If Structure of Step4]

Step6: End BTREEINS

7.2.2 Deleting from a BST

(i) Delete node whose value is 32 in BST of figure 7.1 (e) is given below figure 7.2(i)

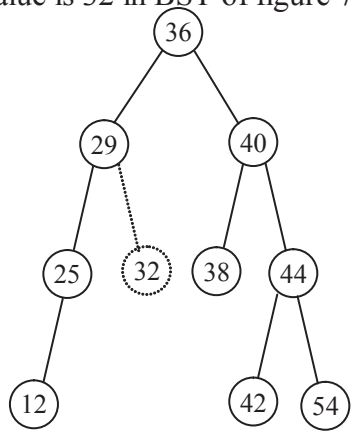
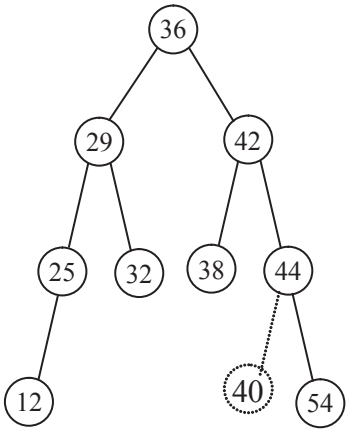


Figure7.2(i)

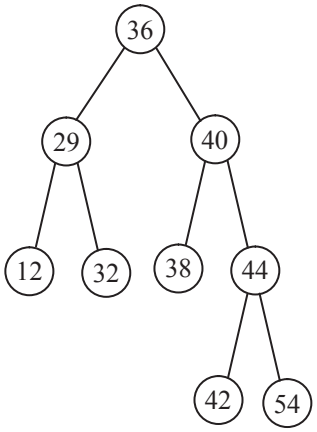
(ii) Delete node whose value is 40 in BST of figure 7.1 (e)



First replace the deleted node to its inorder successor i.e. 42 and than delete the node.

Figure 7.2(ii)

(iii)Delete the node whose value is 25 in figure 7.1 (e) of BST



Just adjust the node 12 in the positions of 25 and remove it.

Figure 7.2(iii)

Deletion of a node into a Binary Search Tree

To delete an element x, we must first verify that its key is exist into the binary search tree.

Case A:

1. When the node to be deleted is as leaf node of the binary search tree
2. When the node to be deleted has a single child either left child or right child.

Case B

1. When the nodes to be deleted have both left and right child.

The algorithm BTREEDEL calls the algorithm DELCASEA and DELCASEB. The pointer variable dnode and pardnode denotes the location of the deleted node and its parent respectively. the pointer variable succ and parsucc denotes the inorder successor of the node to be deleted and parent of the successor respectively.

Algorithm BTREEDEL(x)

Step1: call btsrch(x);

Step2: if(dnode=NULL)then

write(" Not found"); and exit;

Step3: if (dnode->rchild !=NULL) and (dnode->lchild!=NULL) then

Call delcaseB();

else

Call delcaseA(dnode, pardnode);

Step4: free(dnode);

Function btreedel

```
void btreedel(int x)
{
btsrch(x);
if(dnode==NULL)
        {
        printf("\n not found");
        return;
        }
if ((dnode->rchild !=NULL) && (dnode->lchild!=NULL))
        delcaseB();
else
```

```
   delcaseA(dnode, pardnode);
free(dnode);
} /* End of function btreedel */
Algorithm  BTSRCH(x)
Step1:  Set dnode =root;
Step2:  while(dnode!=NULL) and (dnode->info !=x)
Step3:              Set  pardnode = dnode;
 if (dnode->info >x)  then
Set dnode = dnode->lchild;
 else
          Set dnode = dnode->rchild;
          [End of while loop at step2]
Function btsrch
/* The btsrch function search a node in the Binary Search Tree */
void btsrch(int x)
{
dnode =root;
 while((dnode!=NULL) &&(dnode->info !=x))
 {
 pardnode = dnode;
 if (dnode->info >x)
          dnode = dnode->lchild;
 else
           dnode = dnode->rchild;
 }
} /* End of function btsrch */
Algorithm DELCASEA(dnode,  pardnode)
Step1:  If (dnode->lchild =NULL)  and (dnode->rchild==NULL) then
          Set child = NULL;
else if (dnode->lchild)!=NULL  then
                  Set child = dnode->lchild;
                else
```

```
                Set child = dnode->rchild;
Step2:  If (pardnode !=NULL) then
if (dnode=pardnode->lchild)
                        Set pardnode->lchild = child;
        Else
                        Set pardnode->rchild = child;
Else
                root = child;
Step3:  end DELCASEA

Function delcaseA
/* The delcaseA function deletes the node with one child or as leaf node */
void delcaseA(struct nodetype *dnode,struct nodetype *pardnode)
{
struct nodetype *child;
if ((dnode->lchild ==NULL) &&(dnode->rchild==NULL))
        child = NULL;
else if ((dnode->lchild)!=NULL)
                child = dnode->lchild;
        else
                child = dnode->rchild;
if (pardnode !=NULL)
        if (dnode==pardnode->lchild)
                        pardnode->lchild = child;
        else
                pardnode->rchild = child;
else
                root = child;
}/* End of delcaseA function */
Algorithm DELCASEB
Step1:  [Initialize pointers]
Set p = dnode->rchild;  and set q = dnode;
```

```
Step2:  while (p->lchild) !=NULL
Step3:          Set q = p; and  p = p->lchild;
        [End of while loop at step2]
Step4:  Set succ = p; and parsucc = q;
Srep5:  call DELCASEA(succ, parsucc);
Step6:  If (pardnode !=NULL) then
        if (dnode = pardnode->lchild) then
                pardnode->lchild = succ;
                else
        pardnode->rchild = succ;
else
                root = succ;
Step7:  Set succ->lchild = dnode->lchild; and  succ->rchild = dnode->rchild;
Step8: end DELCASEB
Function delcaseB
/* The delcaseB function deletes the node with two child  */
void delcaseB()
{
struct nodetype *p,*q, *succ, *parsucc;
p = dnode->rchild;
q = dnode;
while ((p->lchild) !=NULL)
{
        q = p;
        p = p->lchild;
}
succ = p;
parsucc = q;
delcaseA(succ, parsucc);
if (pardnode !=NULL)
        if (dnode == pardnode->lchild)
                pardnode->lchild = succ;
```

```
        else
        pardnode->rchild = succ;
else
        root = succ;
succ->lchild = dnode->lchild;
succ->rchild = dnode->rchild;
}/* End of delcaseB function */
```

7.3 Binary Search Tree Traversing

Another common operation is to traverse a binary search tree that is to through the tree, enumerating each of its nodes once. Here different ordering is used for traversal in different cases. We have defined three traversal methods.

The methods are all defined recursively, so that traversing a binary tree involves visiting the root and traversing its left and right subtrees. The only difference among the methods is to the order in which these three operations are performed.

1. Preorder (also known as depth first order) (left subtree - right subtree - visit node)

2. Inorder (or symmetric order) (left subtree - visit node - right subtree)

3. Postorder (left subtree - right subtree - visit node)

7.3.1 Preorder Traversing:

```
Algorithm Preorder(root)       //Recursive
        If ( root != NULL)
Step 1: {       print data;
Step 2: Preorder (root  lchild);
Step 3: Preorder (root  rchild);
  }
Step 4: End Preorder
Algorithm Preorder(root)       //Non Recursive
/*
 A binary search tree T is in memory and array stack is used to temporarily hold address of nodes
*/
Step 1: [Initialize top of stack with null and initialize temporary pointer to nodetype is ptr]
        Top=0;
        Stack[Top] = NULL;
        ptr = root;
```

Step 2: Repeat step 3 to 5 while ptr !=NULL

Step 3: visit ptr Info

Step 4: [Test existence of right child]

```
If ( ptr  rchild != NULL)
{
Top=Top+1;
        stack[Top] = ptr  rchild;
}
```

Step 5: [Test for left child]

```
If (ptr  lchild != NULL)
        ptr = ptr  lchild;
else
  [Pop from stack]
        ptr = stack[Top];
        Top = Top-1;
```

End of loop in Step 2.

Step 6 : End Preorder.

Consider figure 7.3

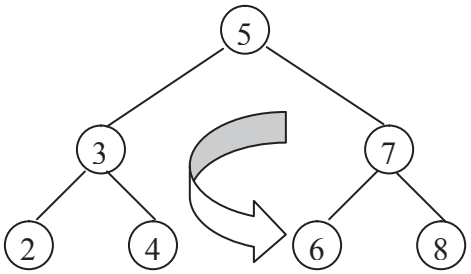


Figure 7.3 Preorder BST

Preorder sequence is as follow

{5, 3, 2, 4, 7, 6, 8}

7.3.2 Inorder Traversing:

Algorithm Inorder(root) //Recursive

Step 1: If (root != NULL)

{

```
                call Inorder (root  lchild);
                print data;
                call Inorder (root  rchild);
        }
Step 2: End Inorder

Algorithm Inorder(root)         // Non Recursive
/*
A binary tree is in memory and array stack is used to temporary hold address of nodes.
*/
Step 1: [Initially push NULL on stack and initialize ptr]
        Top=0;
        stack[Top] = NULL;
        ptr = root;
Step 2: [In Inorder first we visit left child]
        while (ptr != NULL)
        {
                Top = Top+1;
                stack[Top] = ptr;
                ptr= ptr  lchild;
        }
Step 3: ptr= stack[Top];
        Top = Top-1;
Step 4: Repeat Step 5 to 7 while(ptr != NULL)
Step 5: print ptr  Info;
Step 6: If (ptr  rchild != NULL)
                ptr= ptr  rchild;
                goto Step 2;
Step 7: ptr = stack[Top];
        Top= Top-1;
        End of while loop.
Step 8: End Inorder.
```

Consider figure 7.4

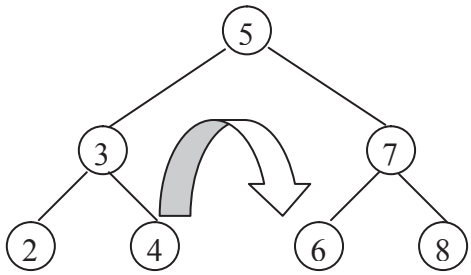


Figure 7.4 Inorder BST

Inorder sequence is as follows:

{ 2, 3, 4, 5, 6, 7, 8}

7.3.3 Postorder Traversing:

Algorithm Postorder(root) //Recursive

Step 1: If (root != NULL)

{

call Postorder(root lchild);

call Postorder(root rchild);

print data;

}

Step 2: End Postorder

Algorithm Postorder(root) //Non Recursive

/*

Binary search tree is in memory and array stack is used to temporary hold address of node.

*/

Step 1 : [Push NULL to stack and initialize ptr]

Top=0;

stack[Top] = NULL;

ptr = root;

Step 2: Repeat Step 3 to 5 while(ptr!= NULL)

Step 3: Top = Top+1;

stack[Top] = ptr;

Step 4: If (ptr rchild != NULL)

Top= Top+1;

stack[Top] = -ptr rchild;

Step 5: ptr = ptr lchild;

Step 6: [pop from stack]

ptr stack[Top];

Top= Top-1;

Step 7: while(ptr>0)

{

Visit ptr Info;

ptr = stack[Top];

Top = Top-1;

}

Step 8: If ptr<0 then

{

ptr = - ptr;

goto Step 2;

}

Step 9: End Postorder

Consider figure 7.5

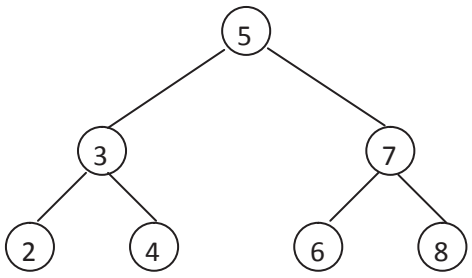


Figure 7.5 Postorder BST

Postorder sequence is as follows:

{ 2, 4, 3, 6, 8, 7, 5}

7.4 Constructing Tree for Given in-Order & Pre/Post Order Traversal of Tree

A binary tree T has 9 nodes. The inorder and preorder traversals of T yield the following sequence of nodes.

Example 1

InorderE A C K F H D B G

Preorder F A E K C D H G B

Draw the tree.

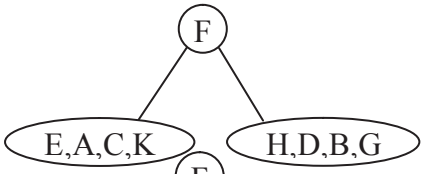
The tree T is drawn from its root downward as follows in different stages in figure 7.6.

(i) The root of T is obtained by choosing the first node in its preorder. Thus F is the root of T

(ii) The inorder of T to find the nodes in the left subtree T of F. Thus T1 consists of nodes E, A, C, K then. Then the left child of F is obtained by choosing the first node in the preorder of T1. Thus A is the left child of F.

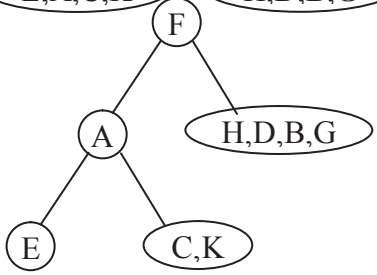
(iii) Similarly, the right subtree T2 of F consists of the nodes H, D, B, and G. D is the right child of F. Repeating the above process with each new node is given below.

Step 1



F is the root of

Step 2

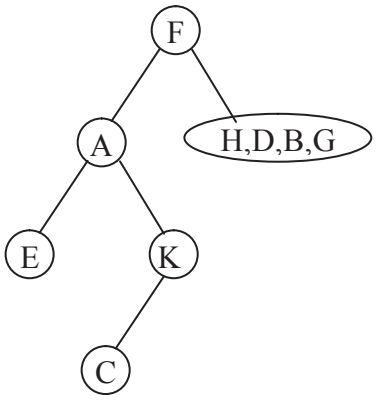


A become left child of

Step 3

E is the left child of A

Step 4

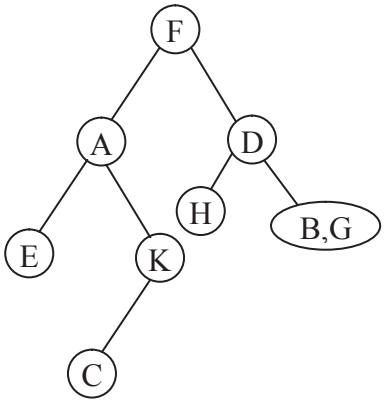


The next node in preorder is K become right child

Step 5

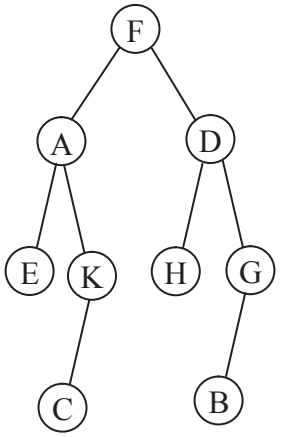
C is the left child of K

Step 6



D become right child of F

Step 8



Next new node G become right child of D.

Step 9

Next new node B is the left child of G.

Figure 7.6 Construction of binary tree from inorder and preorder sequence

The postorder sequence of tree is visit left subtree, visit right subtree then visit rooted node.

E C K A H B G D F

Example 2 - Suppose the following sequence list the nodes of a binary tree T in inorder and postorder.

Inorder2 6 7 1 4 8 3 5 9

Postorder 7 6 2 8 4 9 5 3 1

Draw the binary tree

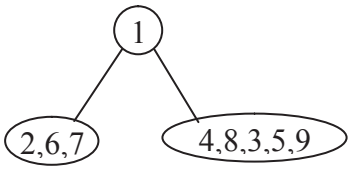
(i) The root of tree T is in postorder traversal is last node i.e. 1.

(ii) The inorder of T to find the nodes in the left subtree T1 of root node 1. Thus T1 consists of nodes 2, 6, and 7.

(iii) Similarly, the right subtree T2 of root node 1 consists of the nodes 4, 8, 3, 5 and 9.

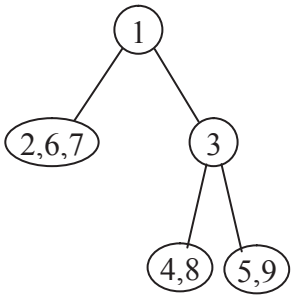
Repeat the above process, while traversing postorder node by node in right to left order. (see figure 7.7)

1.



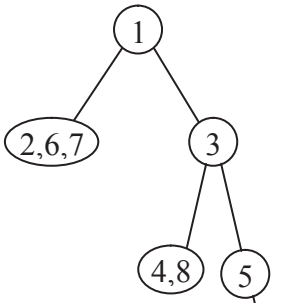
1 is the root of T

2.



3 is the right child of 1

3.

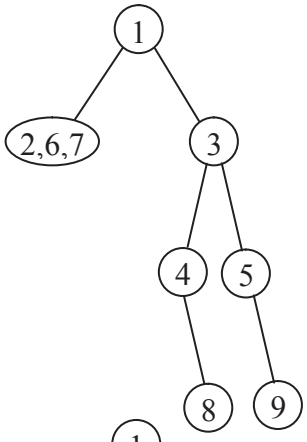


5 become the right child of 3

4.

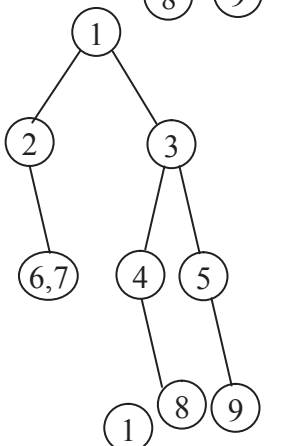
Next node 9 is right child of 5

5.



Next node 4 become the left child of 3

6.

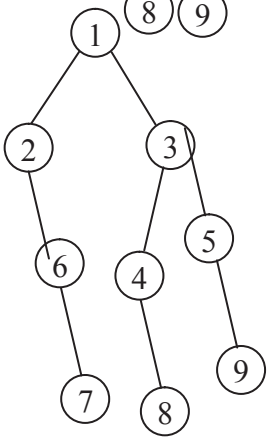


Next node 8 become the right child of 4

7.

Next node 2 become the left child of 1

8.



Next node 6 become the right child of 2

Figure 7.7 Constructing BST

Example 3 - Suppose the following list of letters is inserted in order an empty BST

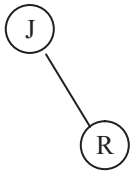
J R D G T E M H P A F Q

(a) Find the final tree T (b) find the inorder traversal of T

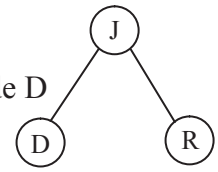
Solution (a) Insert the nodes one after the other, maintaining BST property as given figure 7.38 follows

(i) Insert node of in J in empty BST J become root of BST

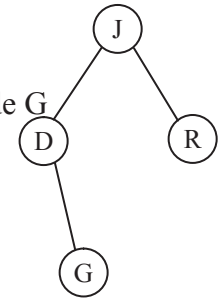
(ii) Insert node R



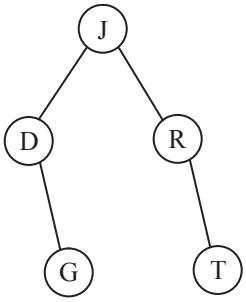
(iii) Insert node D



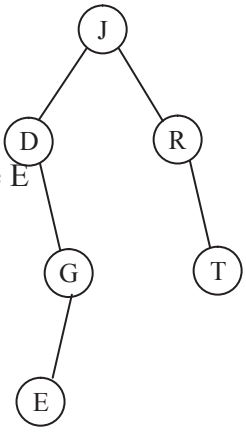
(iv) Insert node G



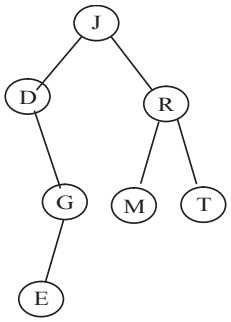
(v) Insert node T



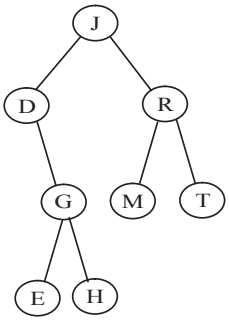
(vi) Insert node E



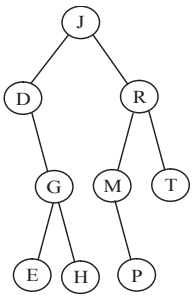
(vii) Insert node M



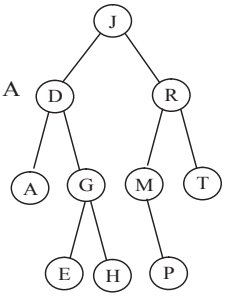
(viii) Insert node H



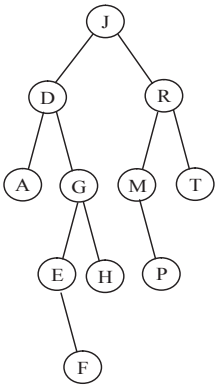
(ix) Insert node P



(x) Insert node A



(xi) Insert node F



(xii) Insert node Q

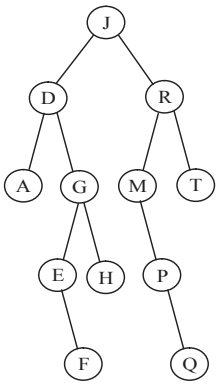


Figure 7.8 Construction of binary search tree

(ii) The inorder traversal of T follows

A D E F G H J M P Q R T

Example 4 - Suppose the following numbers are inserted in order into an empty BST

50 25 75 22 40 60 80 90 15 30

Draw the tree T

Solution: Numbers are inserted one by one maintaining BST property as given below figure 7.9.

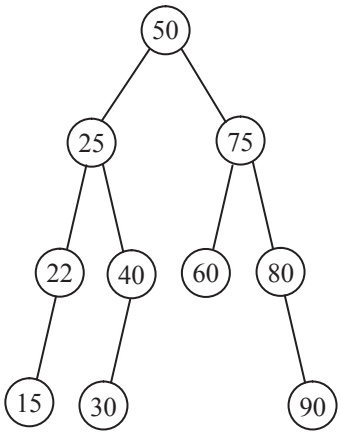
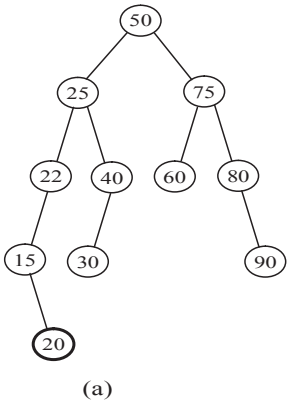


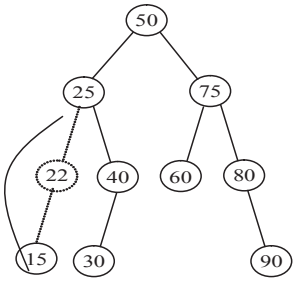
Figure 7.9 binary search tree T

Applying the following operation on original tree T. (That is, the operations are applied independently, not successively as given figure 7.10)

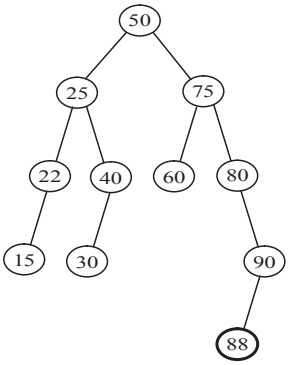
(a) Node 20 is added (b) Node 22 is deleted
(c) Node 88 is added (d) Node 75 is deleted
(e) Node 35 is added (f) Node 50 is deleted



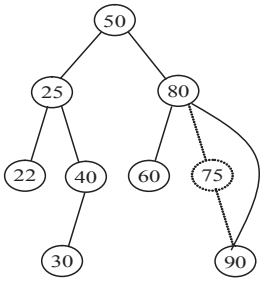
(a)



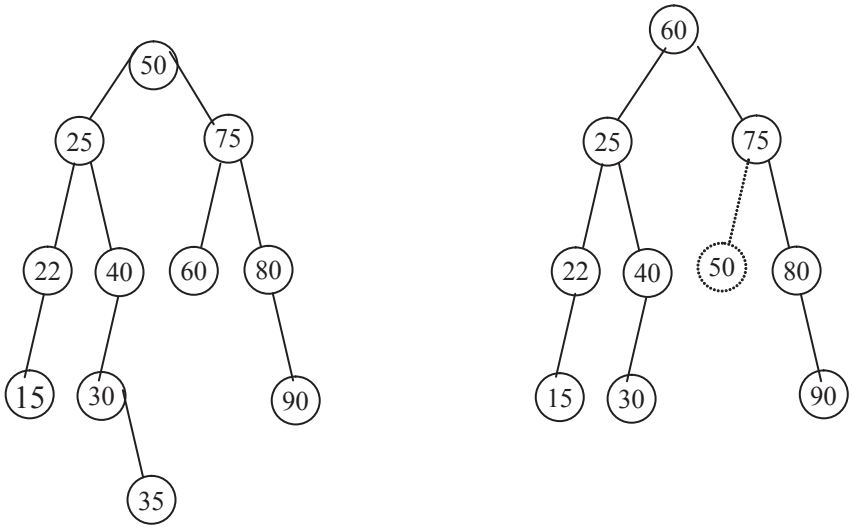
(b)



(c)



(d)



(e) (f)

Figure 7.10 Operations of binary search tree

7.5 Searching and Sorting using Binary Search Tree

Since the definition of a binary search tree is recursive, it is easy to describe a recursive search method. Suppose we wish to search for an element with key x. We begin at the root. If the root is NULL, then the search tree contains no nodes and the search is unsuccessful. Otherwise, we compare x with the key in the root. If x equals this key, then the search terminate successfully. If x is less than the key in the root, then search in the left subtree. If x is larger than the key in the root, only the right subtree needs to be searched.

The recursive algorithm for searching is given below.

```
        Algorithm BTRECSRC(p, x)
        /* recursive search of a binary search tree */
Step1:  If  (p =NULL) then
                Return -1;
Step2:  else if (x = p->Info) then
                Return p;
Step3:  else if (x < p->Info) then
                Return(BTRECSRC(p->lchild, x);
                Else
        Return(BTRECSRC(p->rchild, x);
```

The non-recursion algorithm for binary search is given below.

```
        Algorithm BTSRCH(p, x)
        /* Iterative search of a binary search tree */
Step1:  Repeat Step2 while (p != NULL) and (p->Info != x)
```

```
Step2:          if (p->Info > x)  then
                        p = p->lchild;
else
                        p = p->rchild;
        [end of while loop at Step1]
Step3:  Return (p);
```

The function btsrch search a node in the binary search tree

```
/* The btsrch function find the given node in the binary tree */
void btsrch(struct nodetype *p, int x)
{
 while((p!=NULL) &&(p->info !=x))
 {
 if (p->info >x)
        p = p->lchild;
 else
        p =p->rchild;
 }
if (p==NULL)
  printf("\n Search unsuccessful");
else
  printf("\n Search successful at node address %d",p);
}
```

Sorting of Binary Search Trees

Sorting: The sorting of BST is done in two steps

i) Construct the BST for the given input set

ii) Traverse the data inorder form to get the ascending order output.

7.6 Summary

A binary search tree, BST, is an ordered binary tree T such that either it is an empty tree or

i. each element value in its left subtree less than the root value
ii. each element value in its right subtree is greater than or equal to the root value,
iii. left and right subtrees are again binary search trees

A BST is traversal inorder, preorder and postorder. Inorder traverse first left recursive than node and further right recursive (LDR). Preorder traverse first node than left recursive and right recursive (DLR). Postorder traverse first left recursive than right recursive and finally no node (LRD).

A BST can be construct from inorder and preorder sequence. Similarly a BST also construct from inorder and postorder sequence. The first node of the preorder is the root of the BST. The last node of the postorder is the root of the BST.

7.7 Self Assessment Questions

1 Construct the BST for the following input set

12, 4, 7, 9, 23, 20, 18, 16, 8

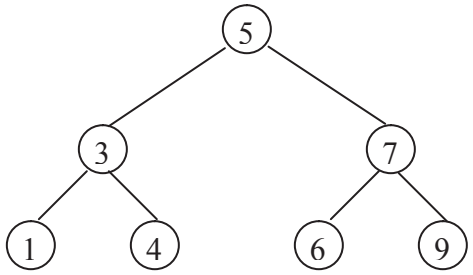
2 Write an algorithm to search an element in BST

3 Write an algorithm for traversing a BST inorder, preorder and postorder form.

4 Write an algorithm to insert an element in BST

5 Write an algorithm to delete an element from BST.

6 Find the preorder, inorder and postorder sequence of the following BST



7 Construct the BST for the given inorder and postorder sequence

Inorder { 2, 3, 4, 5, 6, 7, 8}, and postorder { 2, 4, 3, 6, 8, 7, 5}

8 Write the properties of BST? How it is useful for searching a node in the tree?

Unit - 8 : Graphs

Structure of Unit:

8.0 Objectives

The chapter covers basic graph concept for directed and undirected graph.

- Directed and undirected graph
- Adjacency Matrix
- Adjacency List
- Graph traversing methods: BFS and DFS
- Applications of Graph

8.1 Definition, Terminologies

A graph G consists of two sets V and E. The set V is a finite, non-empty set of vertices. The set E is a set of pairs of vertices, these pairs are called edges.

Graph G can be represent a set of G =(V,E)

- A graph is *nodes* joined by *edge i.e. A set of nodes V and a set of edges E*
- A node is defined by its name or label.
- An edge is defined by the two nodes which it connects, plus optionally:
  - An order of the nodes (*direction*)
  - A *weight or cost* (usually a number)
- Two nodes are *adjacent* if they are connected by an edge
- A node's *degree* is the number of its edges

Graph can be directed or undirected.

- In undirected graph pair of vertices representing vertices are unordered .Thus, the pair (u,v) and (v,u) representing the same edge

For example :-(undirected graph) in Figure 8.1

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

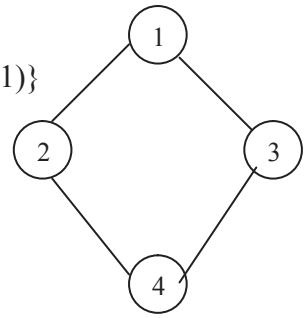


Figure 8.1 Undirected graph (unweighted)

- In directed graph each edge is represented by direct pair **(u,v)** where **u** is a tail and **v** is a head of edge .Thus **(u,v)** and **(v,u)** are two distinct edges.

  For example:-(directed graph) See Figure 8.2.

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

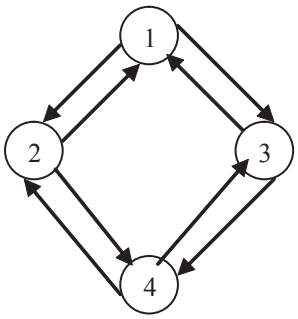
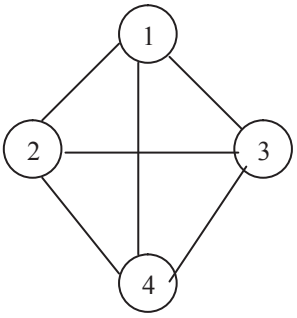


Figure 8.2 Directed graph (unweighted)

In an undirected graph with **n** vertices the maximum number of edges =**n(n-1)/2** then graph is said to complete graph.

For example: See figure 8.3



Number of nodes=4
Number of edges=4(4-1)/2=6

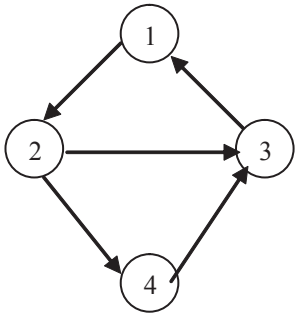
V(G)={1,2,3,4} E={(1,2),(2,4),(4,3),(3,1),(1,4),(2,3)}

Figure 8.3 Complete undirected graph

- **PATH**:- path can be defined as sequence of vertices **(u1,u2,u3————v)** in such a way that there are **(u1,u2),(u2,u3),————(uk,v)** edges in G(E).
- Any graph is said to be strongly connected if a path exist between each vertices of graph.

In directed graph each edge represented by a direct pair**(u,v)** where u is a tail v is a head.

- *Indegree :-In directed graph each vertices has indegree defined as number of edges for which v is head.*
- Outdegree:-In directed graph each vertex has outdegree defined as number of edges for which v is tail.
- *length*: the number of edges in the path
- *cost*: the sum of the weights on each edge in the path
- *cycle*: a path that starts and finishes at the same node
- An *acyclic* graph contains no cycles
- Digraphs are usually either *densely* or *sparsely* connected
  - Densely: the ratio of number of edges to number of nodes is large
  - Sparsely: the above ratio is small



For vertices 3
Indegree=1
Outdegre=2

$$e=\left(\sum_{I=1}^{n} di\right)/2$$

e→edges
n→number of vertices
di→degree of vertices

Figure 8.4 Directed graph and its degree

8.2 Representation of Graphs

Graph can be represented in following ways:-

1. Adjacency matrix

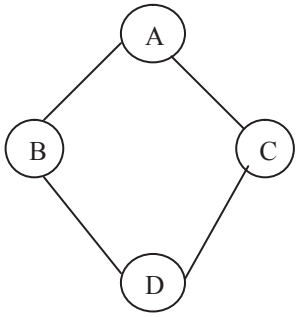
2. Adjacency list

8.2.1 Adjacency matrix

let G=(V,E) be a graph with n vetices,n>=1. The Adjacency matrix of G is a two dimensional n*n array (i.e. matrix), say with the property that a[i,j]=1 if and only if there is an edge(i,j) is in E(G). The element a(i,j)=0 if there is no such edge in G.

For weighted or cost graphs, the position in the matrix is the weight or cost

For example:- Figure 8.5 shows undirected graph and its adjacency matrix.

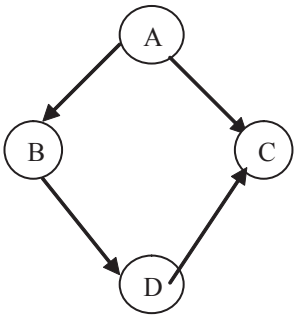


Adjacency matrix:-

Row \ Column	A	B	C	D
A	0	1	1	0
B	1	0	0	1
C	1	0	0	1
D	0	1	1	0

Figure 8.5 Undirected graph and its adjacency matrix

Example: Figure 8.6 shows directed graph and its adjacency matrix



Adjacency matrix:-

Row \ Column	A	B	C	D
A	0	1	1	0
B	0	0	0	1
C	0	0	0	0
D	0	0	1	0

Figure 8.6 Directed graph and its adjacency matrix

8.2.2 Adjacency list

In this representation of graphs, the n rows of Adjacency matrix are represented as n linked lists. There is one list for each vertex in G. See figure 8.7.

For weighted graphs, include the weight/cost in the elements of the list

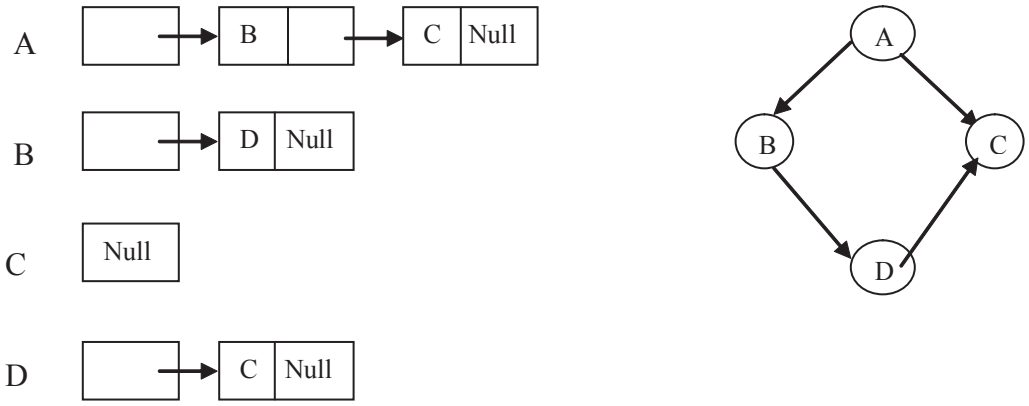


Figure 8.7 Directed graph and its adjacency list

Comparing the two representations

(i) Space complexity

– Adjacency matrix is $O(|n|^2)$

– Adjacency list is $O(n + |E|)$

$|E|$ is the number of edges in the graph

(ii) Static versus dynamic representation

– An adjacency matrix is a *static* representation: the graph is built 'in one go', and is difficult to alter once built

– An adjacency list is a *dynamic* representation: the graph is built incrementally, thus is more easily altered during run-time

8.3 Graph traversing Methods

Many graph applications require to visit all the vertices in a graph. A fundamental problem concerning graph is the reachability problem. In the simplest form it require to determine whether there exist a path among the vertices. In case of a graph, we can specify any arbitrary vertex as the starting vertex. In the given graph **G=(V,E),** we want to visit all vertices in G that are reachable from vertex v, where v ÌV.

Problems with the Graph Traversal:

(i) No First Node. So there must be a way to find out the starting node of the graph. User can enter the starting point or there can be several other methods to find out the starting point.

(ii) When traversing a graph, we must be careful to avoid going round in circles. We do this by marking the vertices which have already been visited. List of visited nodes can also be maintained.

(iii) No natural order among the successor of a particular node.

(iv) Nodes which are not connected or to which there is no path.

We have two common search methods:

1. Breadth first search
2. Depth first search

8.3.1 Breadth first search (BFS)

In breadth first search we start at vertex v and mark it is having been reached. All unvisited vertices adjacent from v are visited next. These are new unexplored vertices. Then vertex **v** has now been explored. The newly vertex have not visited are put onto the end of the list of unexplored vertices. The vertex first in the list is the next to be explored. This exploration continues until no unexplored vertex is left. The list of unexplored vertices as a queue and can be represented by queue. It is based on **FIFO** system.

Breadth first search for the graph in Figure 8.8 is as follows in stack.

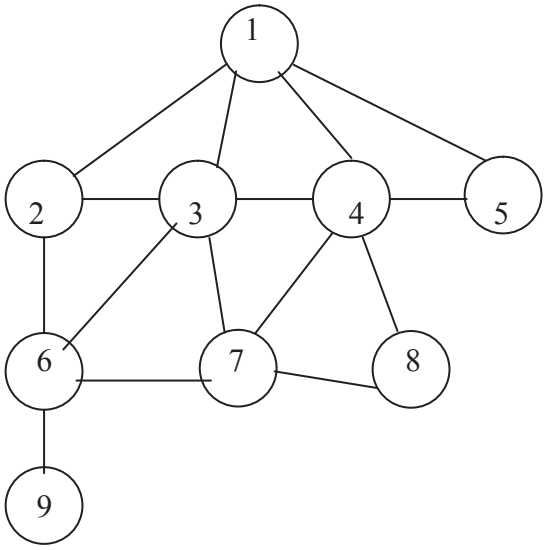


Figure 8.8 Undirected graph, for finding the breadth first search and depth first search sequence

The steps of over search follow as start node 1

1. initially, add 1 to queue as follows

Front=1 Queue:1

Rear =1

2. Delete the front element 1 from queue and add to queue adjacent nodes of 1 which is not visited.

Front= 2 Queue:2, 3, 4, 5

Rear=5

3. Delete the front element 2 from queue and add to queue adjacent nodes of 2 which is not visited.

Front=3 Queue:3, 4, 5, 6

Rear=6

4. Delete the front element 3 from queue and add to queue adjacent nodes of 3 which is not visited.

Front=4 Queue:4, 5, 6, 7

Rear=7

5. Delete the front element 4 from queue and add to queue adjacent nodes of 4 which is not visited.

Front=5 Queue:5, 6, 7, 8

Rear=8

6. Delete the front element 5 from queue and add to queue adjacent nodes of 5 which is not visited.

Front=6 Queue:6, 7, 8

Rear=8

7. Delete the front element 6 from queue and add to queue adjacent nodes of 6 which is not visited.

Front=7 Queue:7, 8, 9

Rear=9

8. Delete the front element 7 from queue and add to queue adjacent nodes of 7 which is not visited.

Front=8 Queue:8, 9

Rear=9

9. Delete the front element 8 from queue and add to queue adjacent nodes of 8 which is not visited.

Front=9 Queue:9

Rear=9

10. Delete the front element 9 from queue and add to queue adjacent nodes of 9 which is not visited.

Front=0 Queue:0

Rear=0

Breadth first search is 1, 2, 3, 4, 5, 6, 7, 8, 9

Algorithm of Breadth First Search:-

1. Non Recursive
2. Recursive

Non Recursive: A breadth first search of G carried out beginning at vertices g for any node i, visited[i]=1 if i has already been visited. Initially, no vertex is visited so for all vertices visited[i]= 0.

Algorithm BFS(g)

Step1 h=g;

```
Step2   visited[g]= 1;
Step3    repeat
          {
Step4     for all vertices  w  adjacent  from h do
           {
             if (visited[w]=0)  then
            {
                 add  w  to  q;   //  w is  unexplored //
                visited[w]=1;
            }                    //  end  of  the  if   //
           }        // end  of  the  for  loop //
Step5      If  q  is empty  then  return;
            Delete  h from q;
         }  until(false);
 //  end of the  repeat //
Step6    End  BREADTH FIRST SEARCH.
```

Function

```
void bfs(int a[MAXSIZE][MAXSIZE], int g, int n)
{
        int i,j, visited[MAXSIZE],h;
        for(i=0;i<n;i++)
        visited[i] =0;
        h = g ;
        visited[h] = 1;
        printf("\n BFS sequence with start vertex is -> ");
        while(1)
        {
        printf(" %d ", h);
         for (j = 0; j<n; j++)
          {
                if ((a[h][j] == 1) && (visited[j] == 0))
                        {
```

```
                    queueins(&q, j);
                    visited[j] = 1;
                }
        }
            if (empty(&q))
                return;
            else
                h = queuedel(&q);

        } /* end of while */
} /* end BFS */
```

Recursive:-

Alorithm BFS(g)

/ *A breadth first search of G carried out beginning at vertices g for any node i, visited[i]=1 if i has already been visited. Initially, no vertex is visited so for all vertices visited[i]= 0. */

```
Step1   visited[g]=1;
Step2   for each vertices w adjacent from g do
          {
            if(visited[w]=0) then
            BFS(w);
          }
Step3   End BFS.
```

The time complexity of BFS algorithm is O(n + |E|) if G is represented by adjacency lists, where n is number of vertices in G and |E| number of edges in G. If G is represented by adjacency matrix, then $T(n,e) = O(n^2)$ and space complexity remain S(n,e) = O(n).

8.3.2 Depth First Search(DFS)

In depth first search we start at vertex v and mark it is having been reached. All unvisited vertices adjacent from v are visited next. These are new unexplored vertices. Then vertex **v** has now been explored. The newly vertex have not visited are put onto the end of the list of unexplored vertices. The vertex first in the list is the next to be explored. This exploration continues until no unexplored vertex is left. The list of unexplored vertices as a stack and can be represented by stack. It is based on **LIFO** system.

Depth first search for the graph in Figure 8.8 is as follows in stack.

The steps of over search follow as start node 1

1 Initially, push 1 onto stack as follows

Stack: 1

2 Pop the top element 1 from stack and push onto stack all adjacent nodes of 1 which is not visited.

Stack: 2, 3, 4, 5

3 Pop the top element 5 from stack and push onto stack all adjacent nodes of 5 which is not visited.

Stack : 2, 3, 4

4 pop the top element 4 from stack and push onto stack all adjacent nodes of 4 which is not visited.

Stack : 2, 3, 7, 8

5 pop the top element 8 from stack and push onto stack all adjacent nodes of 8 which is not visited.

Stack : 2,3,7

6 pop the top element 7 from stack and push onto stack all adjacent nodes of 7 which is not visited.

Stack : 2,3,6

7 pop the top element 6 from stack and push onto stack all adjacent nodes of 6 which is not visited.

Stack : 2,3,9

8 pop the top element 9 from stack and push onto stack all adjacent nodes of 9 which is not visited.

Stack : 2,3

9 pop the top element 3 from stack and push onto stack all adjacent nodes of 3 which is not visited.

Stack : 2

10 pop the top element 2 from stack and push onto stack all adjacent nodes of 2 which is not visited.

Stack : empty

Depth first search sequence is 1, 5, 4, 8, 7, 6, 9, 3, 2

Note that DFS sequence may depend on the order in which direction we go in depth, it may be different also. Here for the simplicity we have assumed the depth from the order of nodes as inserted in adjacency list of the graph. The last node of the list represents the top of the stack, so in the stack the top element is explored subsequently.

Algorithm of Depth First Search:-

1. Non Recursive
2. Recursive

A depth first search of G carried out beginning at vertices g for any node i, visited[i]=1 if i has already been visited. Initially, no vertex is visited so for all vertices visited[i]= 0.

```
Algorithm DFS(g)
/* Non Recursive */
Step 1.  h=g;
Step2   Visited[g]= 1;
Step3    Repeat
           {
Step4     For all vertices w adjacent from h do
           {
             if (visited[w]=0) then
            {
                 push w to s;  // w is unexplored //
                visited[w]=1;
            }                    // end of the if //
           }       // end of the for loop //
Step5     If s is empty then return;
            pop h from s;
         } until(false);
// end of the repeat //
Step6   End DEPTH FIRST SEARCH.
```

The function for depth first search is given below.

```
void dfs(int a[MAXSIZE][MAXSIZE], int g, int n)
{
        int i,j, visited[MAXSIZE],h;
        for(i=0;i<n;i++)
        visited[i] =0;
        h = g ;
        visited[h] = 1;
        printf("\n DFS sequence with start vertex is -> ");
        while(1)
        {
```

```
        printf(" %d ", h);
        for (j = 0; j<n; j++)
        {
            if ((a[h][j] == 1) && (visited[j] == 0))
                {
                    push(&s, j);
                    visited[j] = 1;
                }
        }
            if (empty(&s))
                return;
            else
                h = pop(&s);
        } /* end of while */
} /* end DFS */
```

Algorithm DFS(g)

/* Recursive */

Step1 Visited[g]=1;

Step2 For each vertices w adjacent from g do

```
  {
    if(visited[w]=0) then
    DFS(w);
  }
```

Step3 End DFS.

The time complexity of DFS algorithm is O(n + |E|) if G is represented by adjacency lists, where n is number of vertices in G and |E| number of edges in G. If G is represented by adjacency matrix, then $T(n,e) = O(n^2)$ and space complexity remain S(n,e) = O(n) same.

BFS and DFS are two fundamentally different search methods. In BFS a vertex is fully explored before the exploration of any other node begins. The next vertex to explore is the first unexplored vertex remaining. In DFS the exploration of a vertex is suspended as soon as a new unexplored vertex is reached. The exploration of this vertex is immediately begun.

Example of Directed Graph: The graph traversal techniques for connected directed graph is also breadth first search and depth first search. In directed graph, each edge has direction so it may be possible that a path is available from vertex i to j but not from j to i or there is no path from vertex i to j.

The breadth first search and depth first search sequence for the directed graph in figure 8.9 is given below.

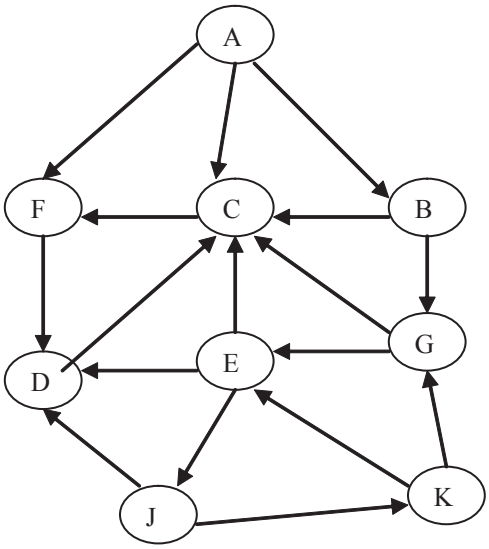


Figure 8.9 Directed Graph, to find breadth first search and depth first search

Breadth first search

The steps of over search follow as start node A

1. initially, add A to queue as follows

front=A queue

Rear =A

2. Delete the front element A from queue and add to queue adjacent nodes of A which is not visited.

Front= B Queue: B, C, F

Rear=F

3. Delete the front element B from queue and add to queue adjacent nodes of B which is not visited.

Front=C Queue: C, F, G

Rear=G

4. Delete the front element C from queue and add to queue adjacent nodes of C which is not visited.

Front=F Queue: F, G

Rear=G

5. Delete the front element F from queue and add to queue adjacent nodes of F which is not visited.

Front=G Queue: G, D

Rear=D

6. Delete the front element G from queue and add to queue adjacent nodes of G which is not visited.

Front=D Queue: D, E

Rear=E

7. Delete the front element D from queue and add to queue adjacent nodes of D which is not visited.

Front=E Queue: E

Rear=E

8. Delete the front element E from queue and add to queue adjacent nodes of E which is not visited.

Front=J Queue: J

Rear=J

9. Delete the front element J from queue and add to queue adjacent nodes of J which is not visited.

Front=K Queue: K

Rear=K

10. Delete the front element K from queue and add to queue adjacent nodes of K which is not visited.

Front=0 Queue: EMPTY

Rear=0

Breadth first search sequence is A, B, C, F, G, D, E, J, K

Depth First Search:-

The steps of over search follow as start node A

1. Initially, push A onto stack as follows

Stack: A

2. Pop the top element A from stack and push onto stack all adjacent nodes of A which is not visited.

Stack: B, C, F

2. Pop the top element F from stack and push onto stack all adjacent nodes of F which is not visited.

Stack: B, C, D

3. Pop the top element D from stack and push onto stack all adjacent nodes of D which is not visited.

Stack: B,C

4. Pop the top element C from stack and push onto stack all adjacent nodes of C which is not visited.

Stack : B

5. Pop the top element B from stack and push onto stack all adjacent nodes of B which is not visited.

Stack : G

6. Pop the top element G from stack and push onto stack all adjacent nodes of G which is not visited.

Stack : E

7. Pop the top element E from stack and push onto stack all adjacent nodes of E which is not visited.

Stack : J

8. Pop the top element J from stack and push onto stack all adjacent nodes of J which is not visited.

Stack : K

9. Pop the top element K from stack and push onto stack all adjacent nodes of K which is not visited.

Stack : empty

DFS sequence is A, F, D, C, B, G, E, J, K.

8.4 Applications of Graph

Graph theory has a very wide range of applications in engineering, physical, social and biological sciences, in linguistics and in numerous other areas. A graph can be used to represent almost any physical situation involving discrete objects and a relationship among them. The following are few examples.

(i) **PERT and related techniques** – A directed graph is a natural way of describing, representing, and analyzing complex projects, which consist of man interrelated activities. The project might be, for example, the design and construction of power dam or the design and erection of an apartment building.

There are number of management techniques such as PERT (Program evaluation and review techniques) and CPM (Critical path method) which employ a graph as the structure on which analysis is based.

A PERT graph is a finite digraph with no parallel edges or cycles, in which there is exactly one source (i.e. a node whose indegree is 0) and one sink (i.e. a node whose outdegree is 0). Each

edge in the graph is assigned a weight (time) value. The directed edges are meant to represents activities, with the directed edge joining nodes, which represent the start time, and the finish time of the activity. The weight value of each edge is taken to be the time it takes to complete the activity.

One such graph is represented in figure. Each node is called an event and its represents a point in time. In particular node v_1 denotes the start of the entire project (its source) and v_6 its completion (its sink). The number represent the number of days required to do that particular activity.

The most common solution of system analysis and its activities is represented by PERT chart to compute critical path. A critical path is a path from the source node to the sink node such that if an activity on the path is delayed by an amount t, then entire project scheduled is delayed by t.

We can compute an algorithm to determine critical path for PERT graph in figure 8.10.

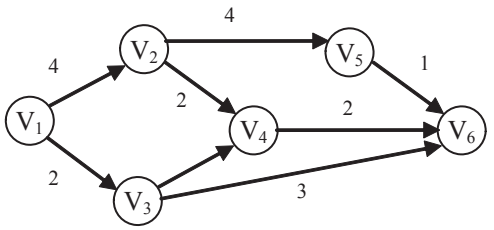


Figure 8.10 A PERT graph

(ii) **Electrical network problems** – Electrical network analysis and synthesis are mainly the study of network topology. The topology of a network is studied by means of its graph. In the graph of electrical network vertices represent the junctions, and branches (which consists of electrical elements) are represented by edges.

(iii) **Topological sort** – A topological sort of a directed graph without cycles, also known as directed acycle graph or DAG, G = (V, E) is a linear ordering of all its vertices such that if G contains an edge (x, y), then x appears before in the ordering.

A topological sort of a graph can be viewed as an orderings of its vertices along a horizontal line so that all directed edges go from left to right.

Directed acyclic graphs are used in many applications to indicate precedence among events such as code optimization techniques of complier.

To find topological sort of DAG, the general structure of the algorithm as follows

1. Perform DFS on G for each vertex
2. As each vertex is finished, insert at front of a linked list
3. Print the elements of linked list in order.

(iv) **Minimum spanning tree (MST)** – A spanning tree for a graph, G = (V, E), is a subgraph of G that is a tree and contains all the vertices of G. In a weighted graph, the weight of a graph is the sum of the weights of the edges of the graph. A MST for a weighted/cost graph is a spanning tree with minimum cost.

There are many applications where minimum spanning tree is needed –

- To find cheapest way to connect a set of terminals, where terminals may represent cities, electrical, electronic components of a circuit, computers, or premises by using roads, wires or wireless, or telephone lines. The solution to this is a MST, which has an edge for each possible connection weighted by the cost of that connection.
- The routing problems to find path among the system over Internet, also need MST.
- The most common solution for MST is prim's and kruskal's algorithm.

(v) **Shortest path** – A path from source vertex v to w is shortest path if there is shortest path from v to u and u to w with lower costs. The shortest paths are not necessarily unique.

The most common shortest path problem is in traveling salesman. Another example is airline, which gives the cities having direct flights and the total time of flights and air fair. The shortest path find which route is provide minimum air fair.

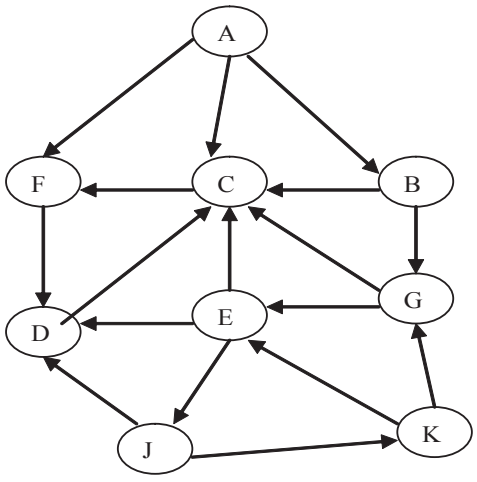
8.5 Summary

A graph data structure consists of a finite (and possibly mutable) set of ordered pairs, called **edges** or **arcs**, of certain entities called **nodes** or **vertices**. As in mathematics, an edge (*x*,*y*) is said to **point** or **go from** *x* **to** *y*. The nodes may be part of the graph structure, or may be external entities represented by integer indices. A graph data structure may also associate to each edge some **edge value**, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).

Graph algorithms are a significant field of interest within computer science. Typical higher-level operations associated with graphs are: finding a path between two nodes, like depth-first search and breadth first search and finding the shortest path from one node to another. A directed graph can be seen as a flow network, where each edge has a capacity and each edge receives a flow.

8.6 Self Assessment Questions

1. Define graph and illustrate its terminologies
2. Write an algorithm to traversal graph in DFS.
3. Write an non-recursive algorithm to traversal graph in BFS.
4. Write applications of graph in brief.
5. Write the adjacency matrix and adjacency list of the following graph.



Unit - 9 : Recursion

Structure of Unit:

9.0 Objectives

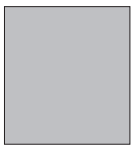
This unit covers the basic condition of recursions and its various implementation in 'C 'programming language. It includes

- Recursive function
- Condition for recursion
- Example programs: factorial, Fibonacci series, Greatest common divisor
- Case study: Tower of Hanoi

9.1 Introduction

A recursive function is a function which either calls itself or is in a potential cycle of function calls. As the definition specifies, there are two types of recursive functions. Consider a function which calls itself: we call this type of recursion **immediate** recursion.

One can view this mathematically in a directed call graph.



```
void A() {
  A();
  return;
}
```

A() is a recursive function since it directly calls itself.
The second part of the definition refers to a cycle (or potential cycle if we use conditional statements) which involves other functions.
Consider the following directed call graph

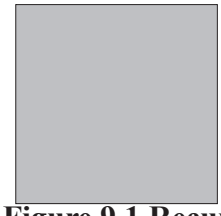


Figure 9.1 Recursion

This can be viewed in the following three functions:

```
void C() {
  A();
  return;
}
void B() {
  C();
  return;
}
void A() {
  B();
  return;
}
```

Recursive functions are an inefficient means of solving problems in terms of run times but are interesting to study nonetheless. For our purposes we will only consider immediate recursion since this will cause enough difficulty.

Writing Recursive Functions

A recursive function has the following general form (it is simply a specification of the general function we have seen many times):

```
ReturnType Function( Pass appropriate arguments ) {
  if a simple case, return the simple value   // base case / stopping condition
  else call function with simpler version of problem
}
```

For a recursive function to stop calling itself we require some type of stopping condition. If it is not the base case, then we simplify our computation using the general formula.

Example: n!

Compute n! Recursively.

We are given the mathematical function for computing the factorial:

n! = n * (n - 1)!

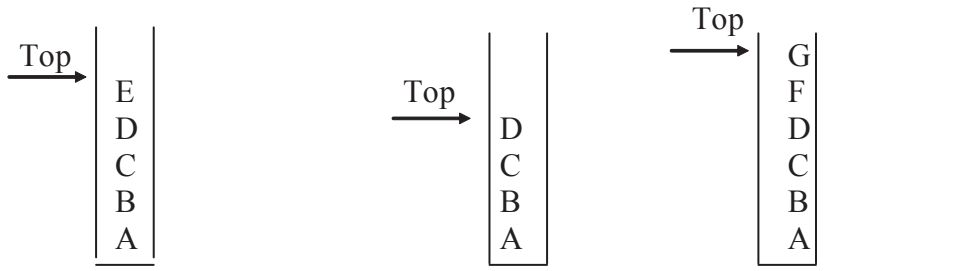
Why this is a recursive function? To compute n! we are required to compute (n - 1)!.

We are also given the simple case. We define mathematically 0!

0! = 1

A recursion is based on stack. A stack is an ordered linear list in which all insertions and deletions are made at one end, called the top. The operations of stack imply that if the elements A, B, C, D, and E

are inserted into a stack, in that order, then the first element to be removed (i.e deleted) must be E. Equivalently, we say that the last element to be inserted into the stack is the first element to be removed. So stack is also called Last In First Out (LIFO) Lists.



(i) Insertion of A, B, C, D and E (ii) Deletion (iii) Insertion of F, G

Figure 9.2 : Stack

9.2 Condition for Recursive Function

A *recursive* function is a function that calls itself. Recursive functions are useful when a problem can be solved by an algorithm that repeats the same operation multiple times using the results of the preceding repetition. Factorial calculation, used in the following example, is one case where recursion is useful. The Towers of Hanoi game is also solved using a recursive algorithm.

A recursive function, like looping code, must have an end condition that always stops the function. Otherwise, the function will continue until a system error occurs or you stop the ColdFusion server.

The following example calculates the factorial of a number, that is, the product of all the integers from 1 through the number; for example, 4 factorial is 4 * 3 * 2 * 1 = 24.

```
function Factorial(factor) {
   If (factor < 1)
      return 1;
   else
      return factor * Factorial(factor -1);
}
```

If the function is called with a number greater than 1, it calls itself using an argument one less than it received. It multiplies that result by the original argument, and returns the result. Therefore, the function keeps calling itself until the factor is reduced to 1. The final recursive call returns 1, and the preceding call returns 2 * 1, and so on until all the initial call returns the end result.

9.3 Example Program Using Recursion

Example 1: Factorial

The factorial function, whose domain is the natural number, can be defined as

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1) * (n-2) * \ldots * 1, & \text{if } n > 0 \end{cases}$$

Let's consider the case of 4! since $n > 0$, we use the second clause of the definition

4! = 4 * 3 * 2 * 1 = 24

Also 3! = 3 * 2 * 1 = 6

2! = 2 * 1 = 2

1! = 1

0! = 1

Such definition is called iterative because it calls for the implicitly repetition of some process until a certain condition is met

Let us look again at the definition of *n!*

4! = 4 * 3 * 2 * 1 = 4 * 3!

Also 3! = 3 * 2 * 1 = 3 * 2!

2! = 2 * 1 = 2 * 1!

1! = 1 = 1* 0!

1 if n =0

Or n! = n *(n-1)! If n>0

Such a definition which defines an object in terms of a simpler case of itself, is called a recursive definition. Here, *n!* is defined in terms of *(n* - I)! which in turn is defined in terms of *(n* - 2)! and so on, until finally 0! is reached

The basic idea, here is to define a function for all its argument values in a constructive manner by using induction. The value of a function for a particular argument value can be computed in a finite number of steps using the recursive definition, where at each step of recursion, we come nearer to the solution

The recursive algorithm to compute n! may be easily converted into a 'C' function as follows:

```
int fact(int n)
{
if (n == 0)
        return (1);
else
        return (n * fact(n-1));
} /* end of fact function */
```

Let us examine the execution of this function, suppose that the calling program contains the statement

```
printf("%d", fact(4));
```

When the calling routine calls fact, the parameter n is et equal to 4, since n is not 0, fact is called a second time with an argument of 3, again n is not 0, fact is a called third time with an argument of

2, and so on until n become 0. Then function returns and stack is popped the top allocation. The execution of function is given in figure 5.6.

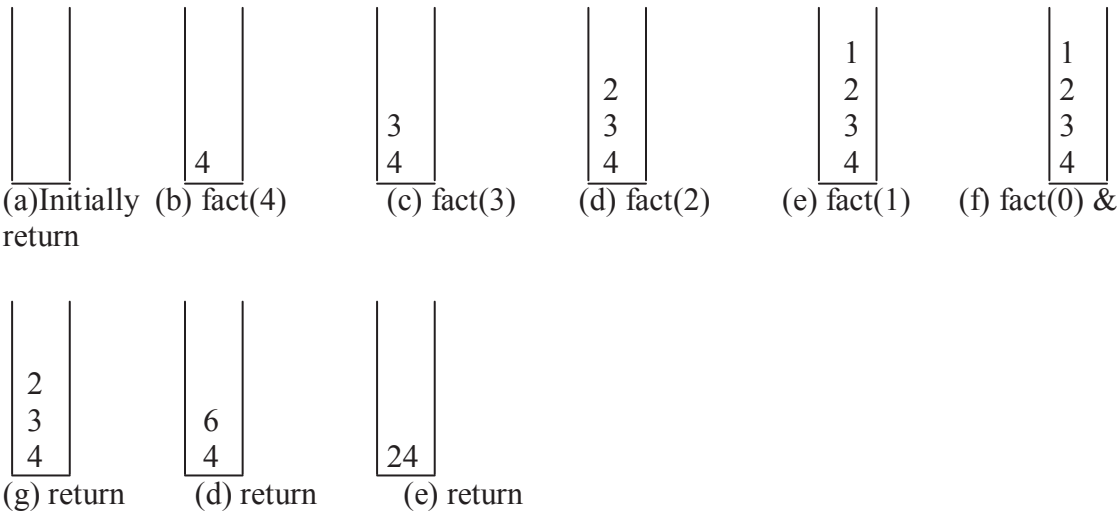


Figure 9.3 Recursive calls of factorial function

Example 2: Multiplication of Natural Numbers

Another example of a recursive definition is the definition of multiplication of natural numbers. The product *a * b,* where *a* and *b* are positive numbers, maybe written as *a* added to itself *b* times. This definition is iterative. We can write this definition in a recursive way as:

$$a * b = \begin{cases} a & \text{if } b = 1 \\ a * (b-1) + a & \text{if } b > 1 \end{cases}$$

To evaluate 5 * 3, we first evaluate 5 * 2 and add 5. To evaluate 5 * 2, we first evaluate 5 * 1 and add 5. 5 * 1 is equal to 5 due to first part of the definition.

Therefore

5*3=5*2+5=5*1+5+5=5+5+5=15

This algorithm of multiplication can be easily converted into 'C' function.

```
int multiply(int a, int b)
    {
        if ( b ==1)
            return(a);.
        else
            return(multiply(a, b – 1) + a);
    }
```

Example 3: `The Fibonacci Sequence`

The Fibonacci sequence is the sequence of integers

0, 1, 1, 2, 3, 5, 8., 13, 21, 34, ...,

Each element, here, is the sum of the two preceding elements.

1 = 0 + 1

2 = 1 + 1

3 = 1 + 2

5 = 2 + 3

8 = 3 + 5

and so on.

We my define the Fibonacci sequence by assume fib(0) = 0, fib(1) =1, as follows:

$$fib(n) = \begin{cases} n & \text{if } n = 0, 1 \\ fib(n-2) + fib(n-1) & \text{if } n >= 2 \end{cases}$$

This method can be easily converted into 'C' function.

Function: Fibonacci term

```
int fib(int n)
{
if (n ==1) || (n ==0)
return (n);
   a = fib ( n - 1);
   b = fib (n - 2);
   return (a + b);
```

} /* end of fibo function */

We may note that the recursive definition of the fibonacci numbers refers to itself twice so that in computing fib (5) = fib (4) + fib (3), the function is called recursively twice (Security figure 5.7).

However the computation of fib (4) again involves determining fib (3), so there is great deal of redundancy hi applying this definition. It would be much more efficient to store the intermediate fibo-nacci values when they are evaluated first time and reuse it each time whenever needed.

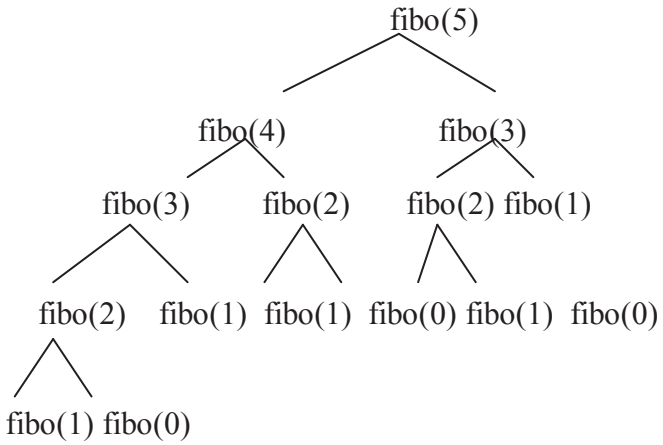


Figure 5.4 Fibonacci sequence recursive calls for fibo(5)

The program to find fibonacci term with recursion is quite easy, just call the above recursive function fibo() from the main function is left for exercise.

The program to find the fibonacci term with explicit stack is given below. Here non-recursive function fibos() is written and a stack is maintained explicitly. For stack operation push, pop and empty function are used.

```
/* The program fibstack.c is written in the form of non-recursive
form using expliicit stack operations empty, pop ,and push. It finds
the fibonacci term */

#include <stdio.h>

#include <stdlib.h>

#define MAXSIZE 80

#define TRUE 1

#define FALSE 0

int pop(struct stack *);

void push(struct stack *, int);

int empty(struct stack *);

int fibos(int n);

struct stack{

int top;

int items[MAXSIZE];

};

void main()

{

int n;

printf("\n Enter a Fibonacci term number ");

scanf("%d",&n);

printf("\n %d term of fibonacci sequence is %d",n, fibos(n-1));

} /* end main */


/*Recursive function fibo() is written in form of stack */

int fibos(int n)

{

int sum = 0;
```

```
struct stack s;
s.top = 0;
while (s.top>=0)
{
     if ((n ==0) || (n==1))
              sum = sum + n;
     else if (n > 1)
     {
          push(&s, n-1);
          push(&s, n-2);
     }
     n = pop(&s);
 }
return(sum);
}
/* function push the integer element into stack s */
void push(struct stack *s, int d)
{
     if (s->top == MAXSIZE -1)
     {
          printf("%s", "stack overflow");
          exit(0);
     }
     else
          s->items[++(s->top)] = d;
     return;
} /* end of push function */

/* function pop the integer element from stack s */
int pop(struct stack *s)
{
```

```
     if (empty(s))
     {
          printf("\n%s", "stack underflow");
          exit(0);
     }
     return(s->items[s->top—]);
} /* end of pop function */
int empty(struct stack *s)
{
     if (s->top == -1)
       return (TRUE);
     else
      return (FALSE);
} /*end of empty function */
```

Example 4: Finding the Greatest Common Divisor

The greatest common divisor of two integers is defined as follows by the Euclid's algorithm

$$GCD(a, b) = \begin{cases} GCD(b, a) & \text{if } (b > a) \\ a & \text{if } (b = 0) \\ GCD(b, \text{mod}(a, b)) & \text{otherwise} \end{cases}$$

Here mod(a, b) is the remainder on dividing a by b. The second case is when the second argument is zero. In this case, the greatest common divisor is equal to the first argument

If the second argument is greater than the first, the order of argument is interchanged. Finally, the GCD is defined in terms of itself. Note, that the size of the argument is getting smaller as mod *(a, b)* will eventually reduce to zero in a finite number of steps

A 'C' function for computing GCD can be written as:

Function: Greatest Common Divisor

```
int GCD (int a, int b)
{
int x;
if (b > a)
    return (GCD (b, a));
if (b == 0)
```

return(a);

else

return{GCD (b, (a %b)))

}

Consider an example to compute GCD(28, 8).

It calls GCD(8, 28 % 8) i.e. GCD(8, 4)

It call GCD(8, 8%4) i.e. GCD(8, 0) which return 8 as greatest common divisor.

Example 5 Write a 'C' program to remove a recursion from the following function using stack.

$$abc(n) = \begin{cases} abc(n-1) + abc(n-2) & \text{if } n>0 \\ 1 & \text{if } n<=0 \end{cases}$$

Program is written using stack function push, pop and empty. The non recursive function for abc is written. A sum variable is used to addition of return values.

```
/* The program recstack.c implement the above function. The function
abc() is written in the form of non-recursive form  using expliicit
stack operations empty, pop ,and push */
#include <stdio.h>
#include <stdlib.h>
#define MAXSIZE 80
#define TRUE 1
#define FALSE 0
int pop(struct stack *);
void push(struct stack *, int);
int empty(struct stack *);
int abc(int n);
struct stack{
int top;
int items[MAXSIZE];
};
void main()
{
int n;
printf("\n Enter a number ");
scanf("%d",&n);
printf("\n Result is %d", abc(n));
} /* end main */

/*Recursive function abc() is written in form of stack */
int abc(int n)
{
```

```
int sum = 0;
struct stack s;
s.top = 0;
while (s.top>=0)
{
     if (n > 0)
     {
          push(&s, n-1);
          push(&s, n-2);
     }
     if (n<=0)
           sum = sum + 1;
     n = pop(&s);
 }
return(sum);
}
/* function push the integer element into stack s */
void push(struct stack *s, int d)
{
     if (s->top == MAXSIZE -1)
{
          printf("%s", "stack overflow");
          exit(0);
     }
     else
          s->items[++(s->top)] = d;
     return;
} /* end of push function */

/* function pop the integer element from stack s */
int pop(struct stack *s)
{
     if (empty(s))
     {
          printf("\n%s", "stack underflow");
          exit(0);
     }
     return(s->items[s->top—]);
} /* end of pop function */

int empty(struct stack *s)
{
     if (s->top == -1)
       return (TRUE);
     else
```

```
        return (FALSE);
} /*end of empty function */
```

Output:

Enter a number: 5

Result is 13

9.4 Case study: Tower of Hanoi

Another complex recursive problem is that of towers of Hanoi. The problem has a historical basis in the ritual of the ancient Tower of Brahma. This is a problem which is not specified in terms of recursion and we have to see how we can use recursion to produce a logical and elegant solution. The problem is as follows:

There are *n* disks of different sizes and there are three needles *A, B and* C. All the n disks are placed on a needle (A) in such a way so that a larger disk is always below a smaller disk as shown in Figure 5.5.

The other two needles are initially empty. The aim is to move the *n* disks to the second needle *(C)* using the third needle *(B)* as a temporary storage. The rules for the movement of disks is as follows:

(a) Only the top disk may be moved at a time.

(b) A disk may be moved from any needle to any other needle.

(c) A larger disk may never be placed upon a smaller disk

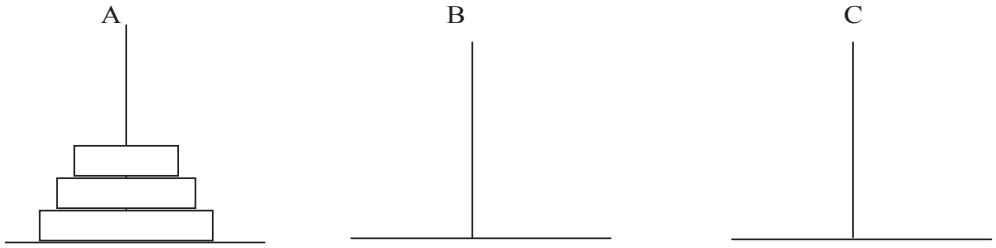
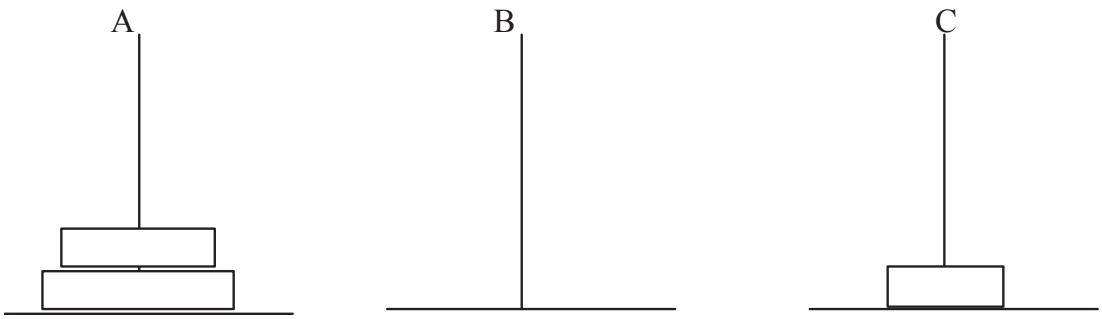


Figure 5.5 Tower of Hanoi

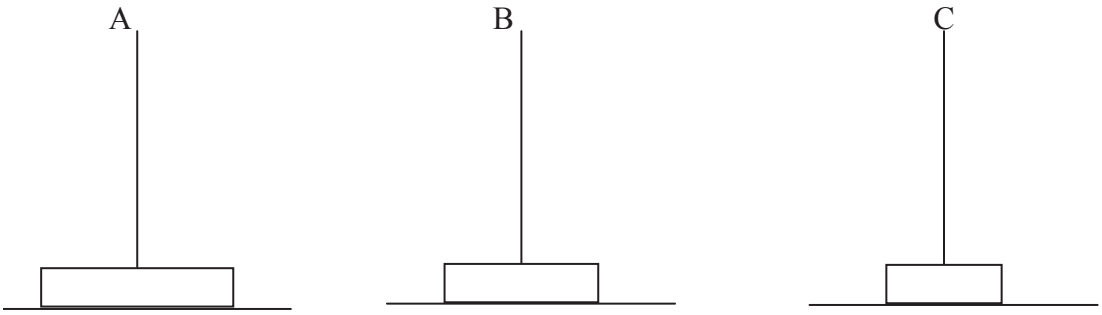
Let us see how to solve this problem. If there is only one disk, we merely move it from *A* to C. If there are two disk, we move the top disk to B and move the second disk to C and finally move the disk from *B* to C.

In general we can move *(n* -1) disks *from A* to B and then move the nth disk *form A* to C, finally we can move *(n* - 1) disks from *B* to C.

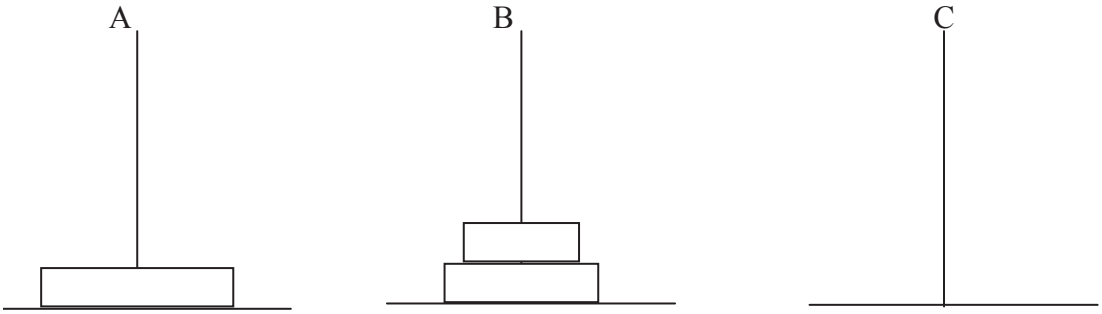
To understand it better, let *n* is 3. The following figure illustrated the disk movements.



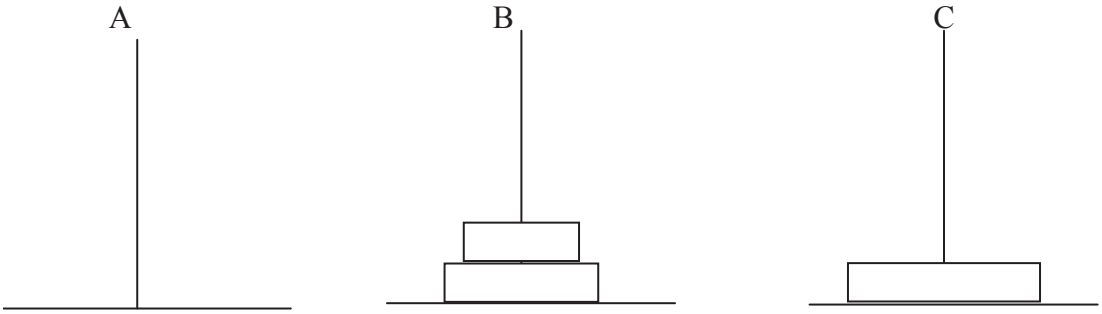
(i) Move disk 1 from needle A to needle C



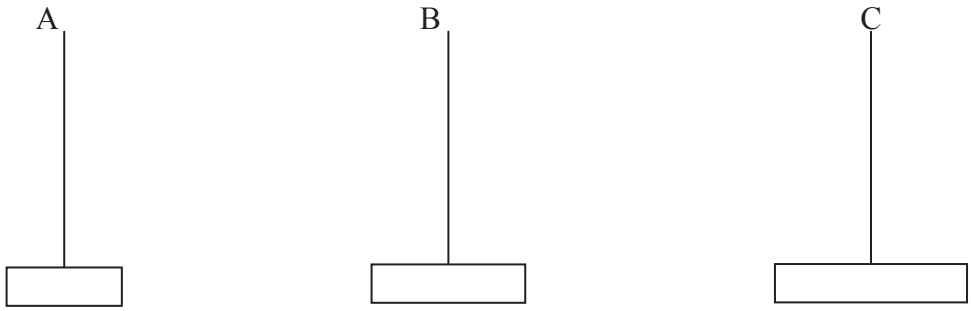
(ii) Move disk 2 from needle A to needle B



(iii) Move disk 1 from needle C to needle B



(iv) Move disk 1 from needle A to needle C



(vii) Move disk 1 from needle A to needle C

Figure 5.6 Tower of Hanoi for three disks.

Therefore, a recursive solution can be formulated to solve the problem of Hanoi to move *n* disks from *A* to C using *B* as auxiliary.

(a) If n =1**,** move the single disk from *A* to C and return.

(b) Move the top *n* - 1 disks from *A* to Busing C as temporary.

(c) Move the remaining disk from *A* to C.

(d) Move the *n* - 1 disks from *B* to C, using *A* as temporary.

Now, let us convert this algorithm to C program. For that let us decide

about the input and output to the program. As an input to the program, we must give *n,* the number of disks. Apart from number of disks, we must also specify the needles which are to act as the initial needle from which we are moving disks, the final needle to which we are moving disks, and temporary needle.

We must also decide about the names of the disks. Let the numbering itself represents the names of the disks. The smallest disk which is on top of the needle is numbered **1,** the next disk is 2 and so on. Such that the largest disk is represented by number *n.*

The output from the program could be a list of statements as:

move disk n form needle *x* to needle *y*

.The solution of the problem then would be to perform action according to the output statement in exactly the same order that they appear in the

```
/* Tower of Hanoi problem with implicit recursion*/
/* HANOI.C */
# include<stdio.h>
void towers(char , char , char , int );
/* Definition of the Tower Of Hanoi generator function */
void towers(char needle1, char needle2, char  needle3, int n)
{
        if( n <= 0)
                printf("\n Illegal entry ");
        /*If only one disk, make the move and return */
        if(n == 1)
        {
                printf("\n Move Disk 1 from needle %c to needle %c", needle1, needle2);
                return;
        }
        /* Move top n -1 disks from A to B, using C as auxiliary */
                towers(needle1, needle3, needle2, n-1);
        /* Move remaining disk from A to C */
                printf("\n Move Disk %d from needle %c to needle %c",n, needle1, needle2);
        /* Move  n -1 disks from B to C, using A as auxiliary */
                towers(needle3, needle2, needle1, n-1);

} /* end of towers */

/* main function */

void main()
```

```
{
        int n;
        printf("\n Input the number of disc: ");
        scanf("%d", &n);
        printf("\n Tower of Hanoi for %d DISCs", n);
        towers('A', 'C', 'B', n);
}
```

Output of the program:
Input the number of disk: 4
Tower of Hanoi for 4 DISCs
Move Disk 1 from needle A to needle B
Move Disk 2 from needle A to needle C
Move Disk 1 from needle B to needle C
Move Disk 3 from needle A to needle B
Move Disk 1 from needle C to needle A
Move Disk 2 from needle C to needle B
Move Disk 1 from needle A to needle B
Move Disk 4 from needle A to needle C
Move Disk 1 from needle B to needle C
Move Disk 2 from needle B to needle A
Move Disk 1 from needle C to needle A
Move Disk 3 from needle B to needle C
Move Disk 1 from needle A to needle B
Move Disk 2 from needle A to needle C
Move Disk 1 from needle B to needle C

Verify the steps and check that it does not violate any of the rules of the problem at any stage. We may note that it is quite possible to develop a non recursive solution to the problem of Towers of Hanoi directly from the problem statement.

9.5 Summary

An essential ingredient of recursion is there must be a "termination condition"; i.e. the call to oneself must be conditional to some test or predicate condition which will cease the recursive calling. A recursive program must cease the recursion on some condition or be in a circular state, i.e. an endless loop which will "crash" the system.

In a computer implementation of a recursive algorithm, a function will conditionally call itself. When any function call is performed, a new complete copy of the function's "information" (parameters, return addresses, etc..) is placed into general data and/or stack memory. When a function returns or exits, this information is returned to the free memory pool and the function ceases to actively exist. In recursive algorithms, many levels of function calls can be initiated resulting in many copies of the function being currently active and copies of the different functions information residing in the memory spaces. Thus recursion provides no savings in storage nor will it be faster than a good non-recursive implementation. However, recursive code will often be more compact, easier to design, develop, implement, integrate, test, and debug.

9.6 Self Assessment Questions

1. Write a recursive function to compute factorial of a given numbers.
2. Write a recursive function to compute Fibonacci series of a given numbers.
3. Write a recursive condition for the function.
4. Write a recursive function to find the greatest common divisor.
5. Write a recursive function to find the search a number using binary search.
6. Write a recursive function to find the sum of n natural number.
7. Write a recursive function to find the nth term of Fibonacci series.
8. Write a recursive function to solve the tower of Hanoi.
9. Write a recursive function to convert a decimal number into binary number.
10. Write a recursive function to display counting from 1 to 100.

Unit - 10 : Searching

Structure of Unit:

10.0 Objectives

Upon successful conclusion of this unit students will be able :

- To perform searching on larger amount of data to retrieve a particular piece of information.
- To find various ways of organizing the data that makes search process efficient.
- To know role of tree data structure in searching process.

10.1 Introduction

Information Retrieval is one of the main important applications of computers. In this unit we consider a number of search techniques and discuss their algorithms. A search algorithm is an algorithm for finding an item with specified properties among a collection of items. The items may be stored individually or may be elements of a search space defined by a mathematical formula.

10.2 Sequential Search

One of the most straightforward and elementary searches is the sequential search. It is also known as a linear search.

In the sequential search process we start at the beginning of a sequence and go through each item one by one, in the order they exist in the list, until we found the item we were looking for.

10.2.1 Example

Suppose we have an array of integers. Although the array can hold data elements of any type, for the simplicity of an example here we have an array of integers, like the following:

Figure 10.1: The array which we are searching

Let's search for the number 3. We start at from beginning and check the first element in the array. Is it 3?

Figure 10.2: Is the first value 3?

No, not it. Is it the next element?

Figure 10.3: Is the second value 3?

Not there either. The next element?

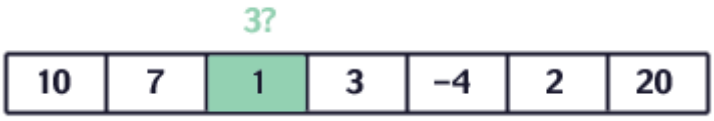


Figure 10.4: Is the third value 3?

Not there either. Next?

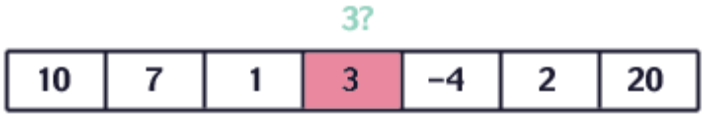


Figure 10.5: Is the fourth value 3? Yes!

We found it! Now we can easily understand the idea of linear searching; we go through each element, in order, until we find the correct value.

10.2.2 Algorithm

```
R = 1
While Array(R ) <> Target
   R = R + 1
While End
If R > N Then
                Print "Element Not Found"
Else
                Print Array (R )
```

10.2.3 Program

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

```
int main()
{
int a[10],i,n,m,c=0;
printf("Enter the size of an array");
scanf("%d",&n);
printf("\nEnter the elements of the array");
for(i=0;i<=n-1;i++)
{
scanf("%d",&a[i]);
}
printf("\nThe elements of an array are");
for(i=0;i<=n-1;i++)
{
printf(" %d",a[i]);
}
printf("\nEnter the number to be search");
scanf("%d",&m);
for(i=0;i<=n-1;i++)
{
if(a[i]==m)
{
c=1;
break;
}
}
if(c==0)
printf("\nThe number is not in the list");
else
printf("\nThe number is found");
return 0;
}
```

10.3 Binary Search

As we saw earlier, the linear search algorithm is simple and convenient for small problems, but if the array is large and requires many repeated searches, it makes good sense to have a more efficient algorithm for searching an array. If the elements of an array are sorted, we can make use of that ordering to make our search more efficient. In this section, we will present and implement the *binary search* algorithm, a relatively simple and efficient algorithm. Binary Search is an incredibly powerful technique for searching an ordered list.

The algorithm is easily explained in terms of searching a dictionary for a word. In a dictionary, words are sorted alphabetically. For simplicity, let us assume there is only one page for all words starting with each letter. Let us assume we wish to search for a word starting with some particular letter.

We open the dictionary at some midway page, let us say a page on which words start with *M*. If the value of our letter is *M*, then we have found what we are looking for and the word is on the current page. If the value of our letter is less than *M*, we know that the word would be found in the first half of the book, i.e. we should search for the word in the pages preceding the current page. If the value of our letter is greater than *M*, we should search the pages following the current page. In either case, the effective size of the dictionary to be searched is reduced to about half the original size. We repeat the process in the appropriate half, opening to somewhere in the middle of that and checking again.

As the process is repeated, the effective size of the dictionary to be searched is reduced by about half at each step until the word is found on a current page.

The basic algorithm for binary search is to find the middle element of the list, compare it against the key, decide which half of the list must contain the key, and repeat with that half.

For example, given a sorted array of items, say:

12, 29, 30, 32, 35, 49

Suppose we wish to search for the position of an element equal to `x`. We will search the array which begins at some `low` index and ends at some `high` index. In our case the low index of the effective array to be searched is zero and the high index is 5. We can find the approximate midway index by integer division `(low + high) / 2`, i.e. 2. We compare our value, `x` with the element at index 2. If they are equal, we have found what we were looking for; the index is 2. Otherwise, if `x` is greater then the item at this index, our new effective search array has a `low` index value of 3 and the high index remains unchanged at 5. If `x` is less than the element, the new effective search array has a `high` index of 1 and the low index remains at zero. The process repeats until the item is found, or there are no elements in the effective search array. The terminating condition is found when the `low` index exceeds the `high` index.

Two requirements to support binary search:

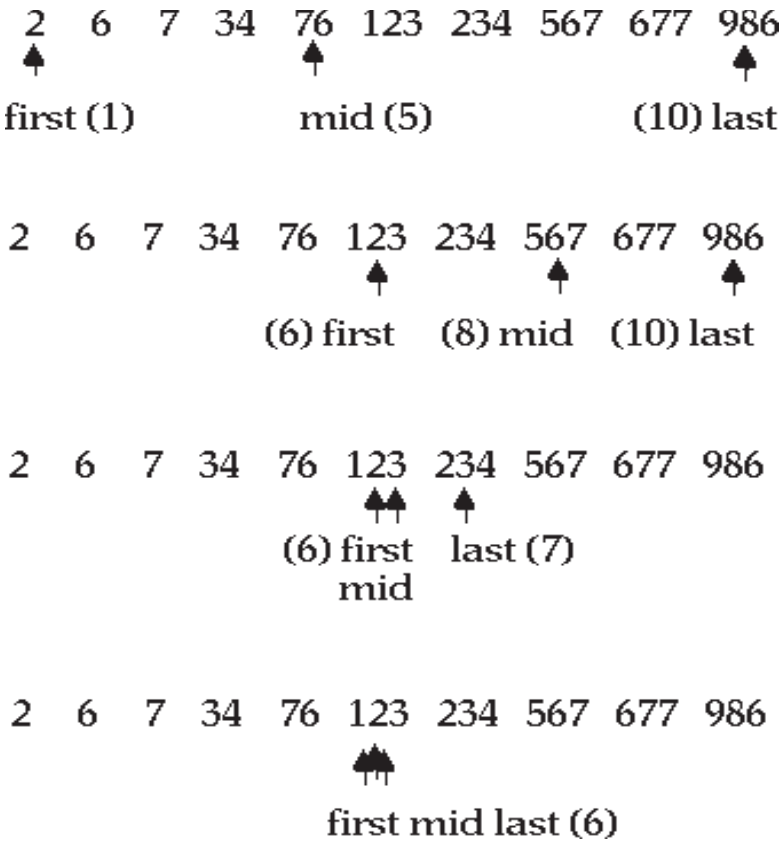
- Random access of the list elements, so we need arrays.
- The array must contain elements in sorted order by the search key.

10.3.1 Example

Suppose the following array of integers is given.

2 6 7 34 76 123 234 567 677 986

We want to seek the value 123 from this array. Steps for searching 123 are as follows



10.3.2 Algorithm

```
bottom := first element
top    := last element
while ((top>=bottom) and (not found)) loop
   found = false
  mid := (top+bottom)/2
  if (list(mid) = item to find) then
     found := true
  elseif (item to find > list(mid)) then
     bottom := mid +1
  else
     top := mid - 1
  end if
end loop
if found = true
   wanted item is in array
else
   wanted item is NOT in array
end if
```

10.3.3 Program

```
#include<stdio.h>
int main()
{
  int a[10],i,n,m,c=0,l,u,mid;
  printf("Enter the size of an array->");
  scanf("%d",&n);
  printf("\nEnter the elements of the array->");
  for(i=0;i<n;i++)
 {
 scanf("%d",&a[i]);
          }
 printf("\nThe elements of an array are->");
 for(i=0;i<n;i++)
 {
      printf(" %d",a[i]);
          }
  printf("\nEnter the number to be search->");
  scanf("%d",&m);
  l=0,u=n-1;
  while(l<=u)
 {
```

```
mid=(l+u)/2;
if(m==a[mid])
{
c=1;
break;
        }
        else if(m<a[mid])
{
u=mid-1;
        }
        else
                l=mid+1;
 }
 if(c==0)
    printf("\nThe number is not in the list");
 else
    printf("\nThe number is found");
 return 0;
}
```

10.4 Role of Trees in Searching

The trees that help in searching are one of the basic tools for solving combinatorial problems. The underlying idea is quite simple: decompose a given problem into finer and finer partial problems (applying a suitable branching operation) such that the generated partial problems together solve the original problem.

- Trees are a special case of a graph data structure. The connections radiate out from a single root without cross connections.
- The tree has nodes (shown with circles) that are connected with branches. Each node will have a parent node (except for the root) and may have multiple child nodes

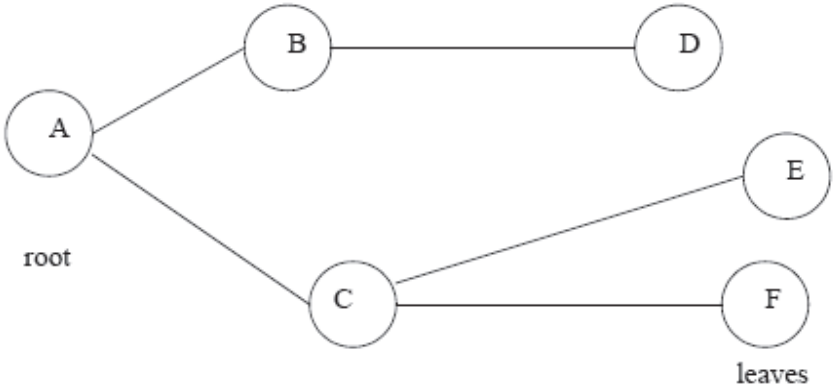
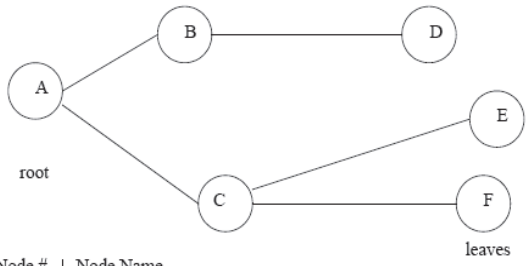


Figure 10.6: Nodes and Branches of a tree

- In an unordered tree the children of a node can be unordered; however in an ordered tree the children take priority, often by listing them in order.
- A Boolean tree applies operators for each of the nodes

10.4.1 Tree representation

- A simple data structure for representing a tree is shown below.



Node #	Node Name
1	A
2	B
3	C
4	D
5	E
6	F

Parent	Child
1	2
1	3
2	4
3	5
3	6

10.7: Relationship between parent and child nodes

10.4.2 Binary Search Tree

A binary search tree is a tree where each node has a left and right child. Either child, or both children, may be missing. Figure 10.8 illustrates a binary search tree. Assuming *k* represents the value of a given node, a binary search tree also has the following property: all children to the left of the node have values smaller than *k*, and all children to the right of the node have values larger than *k*. The top of a tree is known as the root, and the exposed nodes at the bottom are known as leaves. In Figure 10.8 the root is node 20 and the leaves are nodes 4, 16, 37, and 43. The height of a tree is the length of the longest path from root to leaf. For this example the tree height is 2.

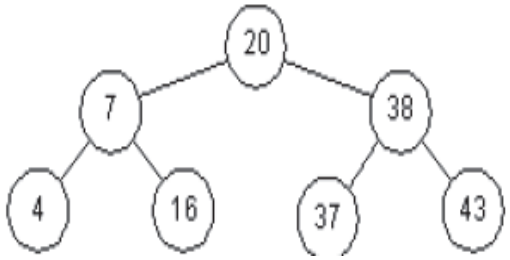


Figure 10.8: A Binary Search Tree

To search a tree for a given value, we start at the root and work down. For example, to search for 16, we first note that 16 < 20 and we traverse to the left child. The second comparison finds that 16 > 7, so we traverse to the right child. On the third comparison, we succeed.

Each comparison results in reducing the number of items to inspect by one-half. In this respect, the algorithm is similar to a binary search on an array. However, this is true only if the tree is balanced. For example, Figure 10.9 shows another tree containing the same values. While it is a binary search tree, its behavior is more like that of a linked list, with search time increasing proportional to the number of elements stored.

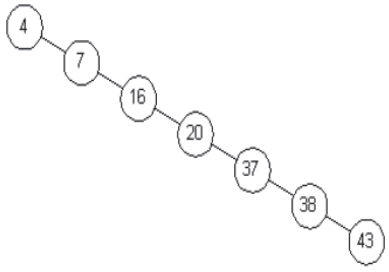


Figure 10.9: An Unbalanced Binary Search Tree

10.4.3 Insertion and Deletion

Now we can see how an unbalanced tree can occur. If the data is presented in an ascending order as shown in figure 10.9, each node will be added to the right of the previous node. This will create one long chain. However, if data is presented for insertion in a random order, then a more balanced tree is possible.

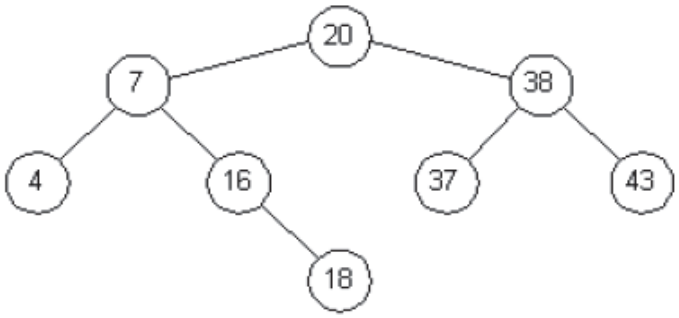


Figure 10.10: Binary Tree after Adding Node 18

Deletion is similar as insertion, but requires that the binary search tree property be maintained. For example, if node 20 in Figure is removed, it must be replaced by node 18 or node 37. Assuming we replace a node by its successor, the resulting tree is shown in Figure 10.11. The rationale for this choice is as follows. The successor for node 20 must be chosen such that all nodes to the right are larger.

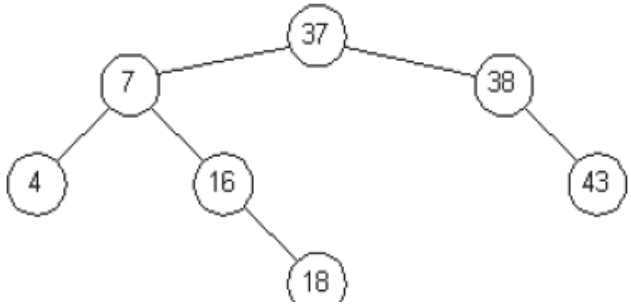


Figure 10.11: Binary Tree after Deleting Node 20

10.5 Summary

The solution of many problems involves searching a collection of values to find a specific element. In this unit we have explored various techniques to search the element. Sequential search is slow but robust. Binary search is appropriate when the data is arranged in proper order. Efficiency of search algorithm increases with balanced search trees. Trees are well suited to situations where the keys are structured.

10.6 Self Assessment Questions

Answer the following questions:

1. In searching a list of n items how many comparisons will you perform on an average when implementing sequential search.
2. Explain the binary search technique with a suitable example.
3. Discuss what role the trees play in searching an element.
4. What are binary search trees? Explain insertion and deletion of a node in binary search trees.

Unit -11 : Sorting

Structure of Unit:

11.0 Objectives

Upon successful conclusion of this unit students will be able :

- To perform sorting on large amount of data .
- To find various ways of organizing the data that involves various kinds of sorting like bubble sort, selection sort, insertion sort and quick sort.

11.1 Introduction

Searching and Sorting are two very important operations that can be performed on larger amount of data. In the previous unit we have discussed various algorithms to search the information. We have noticed that it is easier to find an information if it is in proper order. Sorting is the operation that arranges the records in proper order according to the key value of each record. Here we will discuss various sorting algorithms like bubble sort, selection sort, insertion sort and quick sort.

11.2 Bubble Sort

The bubble sort algorithm is one of the simplest sorting methods to implement. The way it works is by repeatedly going through the array to be sorted, comparing (and swapping, if in the wrong order) two adjacent elements at a time. The name of the algorithm derives from the fact that smaller elements tend to "bubble" to the left of the list when this method is implemented. The positions of each element in the array are key in determining the speed and performance of the sorting method.

The basic idea behind the Bubble Sort is to pass through the table several times in linear manner. Each pass finds the largest element and put it on the correct place. It interchanges two elements if they are not in proper order. The first pass puts the largest element in the n-1th place. The next pass finds the

largest element among the remaining elements and puts it on the n-2th place. We can say after ith iteration n-ith elements will be at proper position. Each element slowly "bubbles" up to its proper place, so this method of sorting is called as bubble sort.

The maximum number of comparisons that can take place when a bubble sort is implemented is (1/2)n(n-1), where there are *n* number of elements in a list. This is because after each pass, the total number of comparisons is reduced by 1.

11.2.1 Example:

Consider the array:

45, 67, 12, 34, 25, 39

In the first step, the focus is on the first two elements (45 and 67) which are compared and swapped, if necessary. In this case, since the element at index 1 is larger than the one at index 0, no swap takes place.

45, 67, 12, 34, 25, 39

Then the focus move to the elements at index 1 and 2 (67 and 12) which are compared and swapped, if necessary. In our example, 67 is larger than 12 so the two elements are swapped. The result is that the largest of the first three elements is now at index 2.

45, 12, 67, 34, 25, 39

The process is repeated until the focus moves to the end of the array, at which point the largest of all the elements ends up at the highest possible index. The remaining steps and result are:

45, 12, 34, 67, 25, 39
45, 12, 34, 25, 67, 39
45, 12, 34, 25, 39, 67

The largest element has bubbled to the top index of the array. In general, a bubble step is performed by the loop:

```
for (k = 0; k < eff_size - 1; k++)
        if (x[k] > x[k + 1])
            swaparay(x, k, k + 1);
```

The loop compares all adjacent elements at index k and k + 1. If they are not in the correct order, they are swapped. One complete bubble step moves the largest element to the last position, which is the correct position for that element in the final sorted array. The effective size of the array is reduced by one and the process repeated until the effective size becomes one. Each bubble step moves the largest element in the effective array to the highest index of the effective array.

We can also discuss it in form of passes.

Suppose the array to be sorted is : 6 1 4 3 9

First Pass

6 1 4 3 9 -> 1 6 4 3 9

First comparison is between 6 and 1; they're not in order, thus swap takes place.

1 6 4 3 9 -> 1 4 6 3 9

Second comparison is between 6 and 4; they're not in order, thus swap takes place.

1 4 6 3 9 -> 1 4 3 6 9

Third comparison is between 6 and 3; they're not in order, thus swap takes place.

1 4 3 6 9 -> 1 4 3 6 9

Next comparison is between 6 and 9; they're in order, thus no swap takes place here since 6 and 9 are already in order.

Second Pass

1 4 3 6 9 -> 1 4 3 6 9 *No swap takes place here since 1 and 4 are already in order.

1 4 3 6 9 -> 1 3 4 6 9

1 3 4 6 9 -> 1 3 4 6 9

1 3 4 6 9 -> 1 3 4 6 9

Although this array is already sorted, one more pass still needs to take place (where there are no swaps) for the algorithm to know that the sorting has been finished.

Third Pass

1 3 4 6 9 -> 1 3 4 6 9 *No swap takes place in this entire pass.

1 3 4 6 9 -> 1 3 4 6 9

1 3 4 6 9 -> 1 3 4 6 9

1 3 4 6 9 -> 1 3 4 6 9

Sorting is finished now.

11.2.2 Algorithm

```
procedure bubbleSort( A : list of sortable items )
  repeat
    swapped = false
    for i = 1 to length(A) - 1 do:
      if A[i-1] > A[i] then
        swap( A[i-1], A[i] )
        swapped = true
      end if
    end for
  until not swapped
end procedure
```

11.2.3 Program:

```
#include<stdio.h>
#include<conio.h>
  void bubble(int a[],int n)
  {
      int i,j,t;
```

```
        for(i=n-2;i>=0;i—)
        {
          for(j=0;j<=i;j++)

                {
                 if(a[j]>a[j+1])
                                {
                                 t=a[j];
                                a[j]=a[j+1];
                                a[j+1]=t;
                                }
                }

         }

}//end function.


void main()
{

    int a[100],n,i;

    clrscr();

    printf("\n\n Enter integer value for total no.s of elements to be sorted: ");
    scanf("%d",&n);

    for( i=0;i<=n-1;i++)
         { printf("\n\n Enter integer value for element no.%d : ",i+1);
           scanf("%d",&a[i]);
         }

     bubble(a,n);

     printf("\n\n Finally sorted array is: ");
     for( i=0;i<=n-1;i++)
     printf("%3d",a[i]);

} //end program.
```

11.3 Selection Sort

The idea of selection sort is rather simple. Here we repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array. Assume that we wish to sort the array in increasing order, i.e. the smallest element at the beginning of the array and the largest element at the end. We begin by selecting the largest element and moving it to the highest index position. We can do this by swapping the element at the highest index and the largest element. We then reduce the *effective size* of the array by one element and repeat the process on the smaller

(sub)array. The process stops when the effective size of the array becomes 1 (an array of 1 element is already sorted).

In this mechanism we start the searching from the first element and locate the element with smallest value. When this element is found we interchange it with the first element. As a result of this interchange the smallest element is placed at first position. In the second iteration the second smallest element is located by examining the rest of the elements, starting from second element onwards. We then interchange this element with second one. We continue this process in the unsorted elements by placing the element in its proper place, until we have placed all the elements in their proper positions.

As each pass places one element in its proper place, we need n-1 passes to sort the elements.

11.3.1 Example

Consider the following array, shown with array elements in sequence separated by commas:

63, 75, 90, 12, 27

The leftmost element is at index zero, and the rightmost element is at the highest array index, in our case, 4 (the effective size of our array is 5). The largest element in this effective array (index 0-4) is at index 2. We have shown the largest element and the one at the highest index in **bold (90 , 27)**. We then swap the element at index 2 with that at index 4. The result is:

63, 75, 27, 12, 90

We reduce the effective size of the array to 4, making the highest index in the effective array now 3. The largest element in this effective array (index 0-3) is at index 1, so we swap elements at index 1 and 3 (75, 12):

63, 12, 27, 75, 90

The next two steps give us:

27, 12, 63, 75, 90
12, 27, 63, 75, 90

The last effective array has only one element and needs no sorting. The entire array is now sorted.

11.3.2 Algorithm

```
 n := length[A]
 for i := 1 to n
   // GOAL: place the correct number in A[i]
   j := FindIndexOfSmallest( A, i, n )
   swap A[i] with A[j]
    // L.I. A[1..i] the i smallest numbers sorted
 end-for
 end-procedure
FindIndexOfSmallest( A, i, n )
// GOAL: return j in the range [i,n] such
//       that A[j]<=A[k] for all k in range [i,n]
 smallestAt := i ;
```

```
  for j := (i+1) to n
    if ( A[j] < A[smallestAt] ) smallestAt := j
    // L.I. A[smallestAt] smallest among A[i..j]
  end-for
  return smallestAt
  end-procedure
```

11.3.3 Program:

```
#include<stdio.h>
int main(){
  int s,i,j,temp,a[20];
  printf("Enter total elements: ");
  scanf("%d",&s);
  printf("Enter %d elements: ",s);
  for(i=0;i<s;i++)
     scanf("%d",&a[i]);
  for(i=0;i<s;i++){
     for(j=i+1;j<s;j++){
        if(a[i]>a[j]){
           temp=a[i];
          a[i]=a[j];
          a[j]=temp;
        }
     }
  }

  printf("After sorting is: ");
  for(i=0;i<s;i++)
     printf(" %d",a[i]);

  return 0;
}
```

There are two loops, one nested within other. During the first pass n-1 comparisons are made. In second pass n-2 comparisons are made. In general i pass requires n-i comparisons. Total comparisons are-

(n-1) + (n-2) + (n-3) + …….+1= n(n+2)/2

11.4 Insertion Sort

The insertion sort is a simple but elegant comparison sort. In this algorithm, every iteration removes an element from the input data and *inserts* it into the correct position in the list being sorted. The choice of the element being removed from the input is random - and this process is repeated until all input elements have been gone through. Of course, it goes without saying that this algorithm can sort in an ascending or descending manner.

If the first few objects are already sorted, an unsorted object can be inserted in the sorted set in proper place. This is called insertion sort. An algorithm consider the elements one at a time, inserting each in its suitable place among those already considered (keeping them sorted). Insertion sort is an example of an **incremental** algorithm; it builds the sorted sequence one number at a time. This is perhaps the simplest example of the incremental insertion technique, where we build up a complicated structure on n items by first building it on n " 1 items and then making the necessary changes to fix things in adding the last item. The given sequences are typically stored in arrays. We also refer the numbers as **keys**.

11.4.1 Example

Following figure shows the operation of INSERTION-SORT on the array A= (5, 2, 4, 6, 1, 3). Each part shows what happens for a particular iteration with the value of *j* indicated. *j* indexes the "current card" being inserted into the hand.

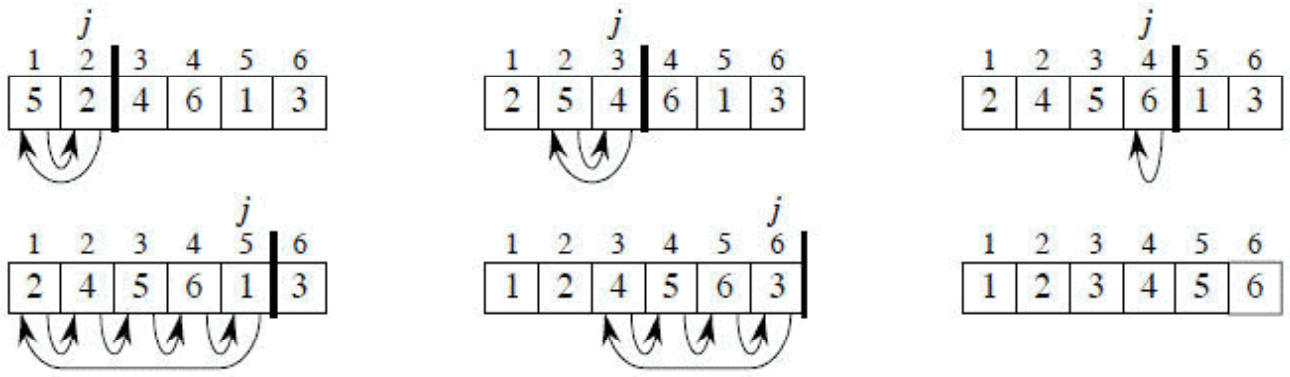


Figure 11.1: An Example of Selection Sort

Read the figure 11.1 row by row. Elements to the left of $A[j]$ that are greater than $A[j]$ move one position to the right, and $A[j]$ moves into the evacuated position.

Best-Case

The best case occurs if the array is already sorted. For each $j = 2, 3, ..., n$, we find that $A[i]$ less than or equal to the key when i has its initial value of $(j$ " 1). In other words, when $i = j$ "1, always find the key $A[i]$ upon the first time the WHILE loop is run.

Worst-Case

The worst-case occurs if the array is sorted in reverse order i.e., in decreasing order. In the reverse order, we always find that $A[i]$ is greater than the key in the while-loop test. So, we must compare each element $A[j]$ with each element in the entire sorted sub array $A[1 .. j$ " 1] for $j = 2, 3, ..., n$. Equivalently, we can say that since the while-loop exits because i reaches to 0, there is one additional test after $(j$ " 1) tests.

Given the array: 6 8 1 4 5 3 7 2 - and the goal is to put it into ascending order.

6 8 1 4 5 3 7 2 (Consider index 0)

6 8 1 4 5 3 7 2 (Consider indices 0 - 1)

1 6 8 4 5 3 7 2 (Consider indices 0 - 2: insertion places 1 in front of 6 and 8)

1 4 6 8 5 3 7 2 (Process same as above is repeated until array is sorted)

1 4 5 6 8 3 7 2

1 3 4 5 6 7 8 2

1 2 3 4 5 6 7 8 (the array is sorted!)

11.4.2 Algorithm

```
for j    !1 to length(A)-1
    key    ! A[ j ]
    > A[ j ] is added in the sorted sequence A[0, .. j-1]
    i    ! j - 1
    while i >= 0 and A [ i ] > key
       A[ i +1 ]    ! A[ i ]
       i    ! i -1
    A [i +1]    ! key
```

11.4.3 Program:

```
#include<stdio.h>
void main()
{
        int A[20], N, Temp, i, j;
        clrscr();
        printf("\n\n\t ENTER THE NUMBER OF TERMS...: ");
        scanf("%d", &N);
        printf("\n\t ENTER THE ELEMENTS OF THE ARRAY...:");
        for(i=0; i<N; i++)
        {
                gotoxy(25,11+i);
                scanf("\n\t\t%d", &A[i]);
        }
        for(i=1; i<N; i++)
        {       Temp = A[i];
                j = i-1;
                while(Temp<A[j] && j>=0)
```

```
        {
                A[j+1] = A[j];
                j = j-1;
            }
            A[j+1] = Temp;
        }
        printf("\n\tTHE ASCENDING ORDER LIST IS...:\n");
        for(i=0; i<N; i++)
            printf("\n\t\t\t%d", A[i]);
        getch();
    }
```

11.5 Quick Sort

Quicksort is a very efficient sorting algorithm invented by C.A.R. Hoare. It has two phases:

- the partition phase and
- the sort phase.

As we will see, most of the work is done in the partition phase - it works out where to divide the work. The sort phase simply sorts the two smaller problems that are generated in the partition phase.

This makes Quicksort a good example of the divide and conquer strategy for solving problems. In quicksort, we divide the array of items to be sorted into two partitions and then call the quicksort procedure recursively to sort the two partitions, *i.e.* we *divide* the problem into two smaller ones and conquer by solving the smaller ones. Thus the conquer part of the quicksort routine looks like this:

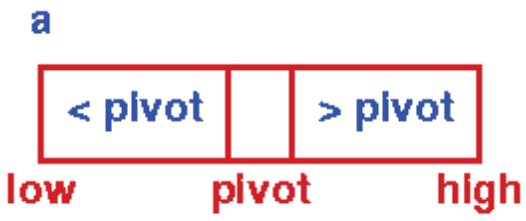


Figure 11.2 : Initial Step - First Partition

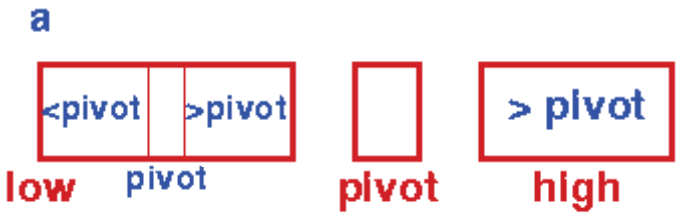


Figure 11.3 Sort Left Partition in the same way

For the strategy to be effective, the *partition* phase must ensure that all the items in one part (the lower part) are less than all those in the other (upper) part.

To do this, we choose a *pivot* element and arrange all the items in the lower part that are less than the pivot and all those in the upper part that are greater than it.

11.5.1 Example

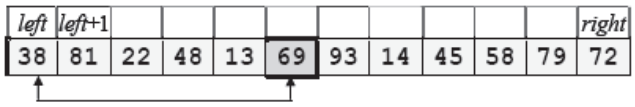


Figure11.4

Swap pivot element with leftmost element. *lo*=*left*+1; *hi*=*right*;

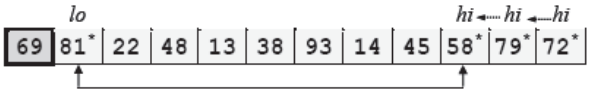


Figure 11.5

Move *hi* left and *lo* right as far as we can, then swap *A*[*lo*] and *A*[*hi*], and move *hi* and *lo* one more position.

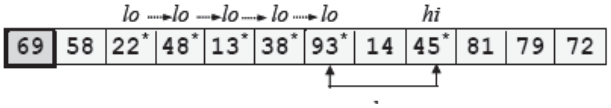


Figure11.6

Repeat above

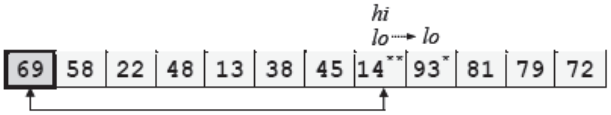


Figure 11.7

Repeat above until *hi* and *lo* cross; then *hi* is the final position of the pivot element, so swap *A*[*hi*] and *A*[*left*].

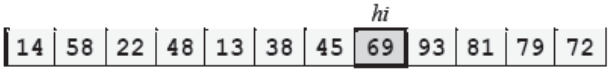


Figure 11.8

Partitioning complete; return value of *hi*.

11.5.2 Algorithm

```
Quicksort(A,p,r) {
   if (p < r) {
      q <- Partition(A,p,r)
      Quicksort(A,p,q)
      Quicksort(A,q+1,r)
   }
}
Partition(A,p,r)
   x <- A[p]
   i <- p-1
   j <- r+1
   while (True) {
      repeat
         j <- j-1
      until (A[j] <= x)
      repeat
         i <- i+1
      until (A[i] >= x)
      if (i<- A[i] Exchange> A[j]
      else
         return(j)
   }
}
```

11.5.4 Program:

```
#include<stdio.h>
void quicksort(int [10],int,int);
int main(){
  int x[20],size,i;
  printf("Enter size of the array: ");
  scanf("%d",&size);
  printf("Enter %d elements: ",size);
  for(i=0;i<size;i++)
    scanf("%d",&x[i]);
```

```
  quicksort(x,0,size-1);
  printf("Sorted elements: ");
  for(i=0;i<size;i++)
    printf(" %d",x[i]);

  return 0;
}
void quicksort(int x[10],int first,int last){
    int pivot,j,temp,i;

     if(first<last){
         pivot=first;
         i=first;
         j=last;
         while(i<j){
             while(x[i]<=x[pivot]&&i<last)
                 i++;
             while(x[j]>x[pivot])
                 j—;
             if(i<j){
                 temp=x[i];
                  x[i]=x[j];
                  x[j]=temp;
             }
         }
         temp=x[pivot];
         x[pivot]=x[j];
         x[j]=temp;
         quicksort(x,first,j-1);
         quicksort(x,j+1,last);
    }
}
```

11.6 Summary

In this unit we have discussed various sorting algorithms. The order of original data is an important consideration in choosing a sort algorithm. If the number of entries in an array are small, the bubble sort or selection sort is appropriate because they require a little programming efforts to write and maintain.

11.7 Self Assessment Questions

Answer the following questions

1. Given the following array:

   40, 55, 20, 30, 50, 15, 25

   Show the contents of array after each sort listed below:
   i) Insertion sort (after 4^{th} iteration)
   ii) Bubble sort (after 3^{rd} iteration)
   iii) Selection sort (after 4^{th} iteration)

2. Trace the series of recursive calls performed by quick sort during the process of sorting the following array:

   3, 1, 4, 5, 9, 2, 6, 10, 7, 8

Unit - 12 : Algorithm Design Strategies : Divide & Conquer, Greedy

Structure of Unit:

12.0 Objectives

Upon successful conclusion of this unit students will be able to:

- identify, for a certain problem, the more adequate algorithm design strategies;
- design, implement and test an appropriate algorithm for different application problems;
- apply the main strategies and algorithms used in various application

12.1 Introduction

In this unit two techniques are discussed. Divide & conquer and Greedy Method. Divide & Conquer technique is used in merger sort and quick sort algorithms. The basic idea behind this technique is to solve a problem by dividing it into a number of small sub problems recursively and then solve the sub problem and at last merge all the sub problems together to derive a solution to the original problem. The greedy method is used for feasible solution algonithm. The general structure of greedy algorithm shows how it can be applied on various applications e.g. knapsack problem.

12.2 Divide and Conquer

Divide and Conquer (D&C) is an important algorithm that is based on multi-branched recursion. This algorithm works by recursively breaking down a problem into two or more sub-problems of the same type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

This algorithm works efficiently for all kinds of problems, such as sorting (e.g., quick sort, merge sort), multiplying large numbers (e.g. Karatsuba), syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform.

The name "divide and conquer" is sometimes applied also to algorithms that reduce each problem to only one sub problem, such as the binary search algorithm for finding a record in a sorted list. These algorithms can be implemented more efficiently than general divide-and-conquer algorithms; in particular, if they use tail recursion, they can be converted into simple loops. Under this broad definition, however, every algorithm that uses recursion or loops could be regarded as a "divide and conquer" algorithm. Therefore, some persons consider that the name "divide and conquer" should be used only when each problem may generate two or more sub problems.

The correctness of a divide and conquer algorithm is usually proved by mathematical induction, and its computational cost is often determined by solving recurrence relations.

The divide-and-conquer strategy solves a problem by:

1. Breaking it into sub problems that are themselves smaller instances of the same type of problem

2. Recursively solving these sub problems

3. Appropriately combining their answers

12.2.1 Quick Sort

As one of the more advanced sorting algorithms, the Quicksort Algorithm has a fairly simple concept at the core, but is made complicated by the constraints of the array structure.

The basic concept is to pick one of the elements in the array as a pivot value around which the other elements will be rearranged. Everything less than the pivot is moved left of the pivot (into the left partition). Similarly, everything greater than the pivot go into the right partition. At this point each partition is recursively quick sorted.

The Quicksort algorithm is fastest when the median of the array is chosen as the pivot value. That is because the resulting partitions are of very similar size. Each partition splits itself in two and thus the base case is reached very quickly.

In practice, the Quicksort algorithm becomes very slow when the array passed to it is already close to being sorted. Because there is no efficient way for the computer to find the median element to use as the pivot, the first element of the array is used as the pivot. So when the array is almost sorted, Quicksort doesn't partition it equally. Instead, the partitions are lopsided like in Figure 12.2. This means that one of the recursion branches is much deeper than the other, and causes execution time to go up. Thus, it is said that the more random the arrangement of the array, the faster the Quicksort Algorithm finishes..

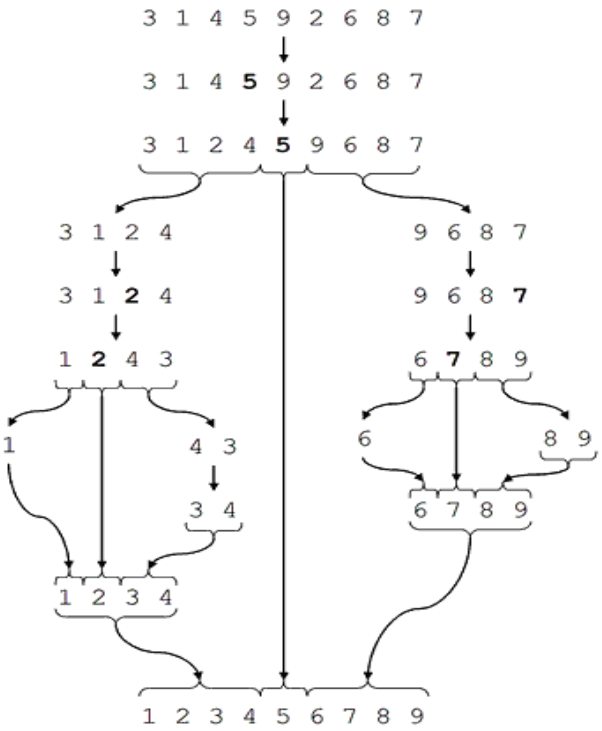


Figure 12.1: The ideal Quicksort on a random array.

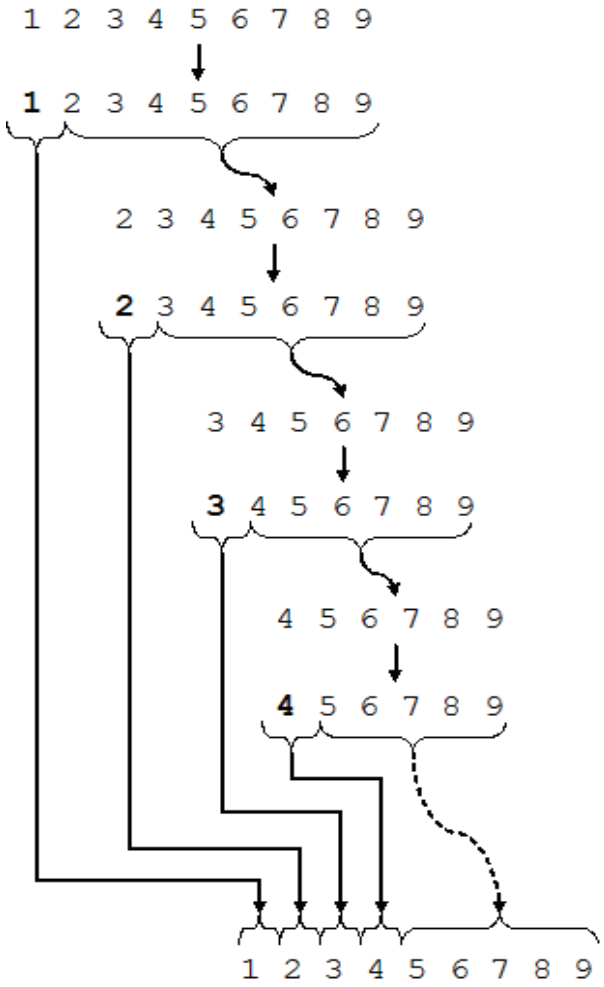


Figure 12.2: Quicksort on an already sorted array.

These are the steps taken to sort an array using QuickSort:

Suppose the array to sort is

3 1 4 5 9 2 6 8 7

Elements underlined indicate swaps.

Elements **bold** indicate comparisons.

Definitions: a "small" element is one whose value is less than or equal to the value of the pivot. Likewise, a "large" element is one whose value is larger than that of the pivot.

At the beginning, the entire array is passed into the quicksort function and is essentially treated as one large partition. At this time, two indices are initialized: the left-to-right search index, i, and the right-to-left search index, k. The value of i is the index of the first element in the partition, in this case 0, and the value of k is 8, the index of the last element in the partition.

3 1 4 5 9 2 6 8 7

The first element in the partition, 3, is chosen as the pivot element, around which two subpartitions will be created. The end goal is to have all the small elements at the front of the partition, in no particular order, followed by the pivot, followed by the large elements.

To do this, quicksort will scan rightwards for the first large element. Once this is found, it will look for the first small element from the right. These two will then be swapped. Since i is currently set to zero, the pivot is actually compared to itself in the search of the first large element.

3 1 4 5 9 2 6 8 7

The search for the first large element continues rightwards. The value of i gets incremented as the search moves to the right.

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

Since 4 is greater than the pivot, the rightwards search stops here. Thus the value of i remains 2.

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

Now, starting from the right end of the array, quicksort searches for the first small element. And so k is decremented with each step leftwards through the partition.

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

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

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

Since 2 is not greater than the pivot, the leftwards search can stop.

3 1 2 5 9 4 6 8 7

Now elements 4 and 2 (at positions 2 and 5, respectively) are swapped.

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

Next, the rightwards search resumes where it left off: at position 2, which is stored in the index i.

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

Immediately a large element is found, and the rightwards search stops with i being equal to 3.

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

Next the leftwards search, too, resumes where it left off: k was 5 so the element at position 5 is compared to the pivot before k is decremented again in search of a small element.

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

This continues without any matches for some time...

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

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

The small element is finally found, but no swap is performed since at this stage, i is equal to k. This means that all the small elements are on one side of the partition and all the large elements are on the other.

2 1 3 5 9 4 6 8 7

Only one thing remains to be done: the pivot is swapped with the element currently at i. This is acceptable within the algorithm because it only matters that the small element be to the left of the pivot, but their respective order doesn't matter. Now, elements 0 to (i - 1) form the left partition (containing all small elements) and elements k + 1 onward form the right partition (containing all large elements.

Calling quickSort on elements 0 to 1

The right partition is passed into the quicksort function.

2 1 3 5 9 4 6 8 7

2 is chosen as the pivot. It is also compared to itself in the search for a small element within

the partition.

2 1 3 5 9 4 6 8 7

The first, and in this case only, small element is found.

2 1 3 5 9 4 6 8 7

Since the partition has only two elements, the leftwards search begins at the second element and finds 1.

1 2 3 5 9 4 6 8 7

The only swap to be made is actually the final step where the pivot is inserted between the two partitions. In this case, the left partition has only one element and the right partition has zero elements.

Calling quickSort on elements 0 to 0

Now that the left partition of the partition above is quicksorted, there is nothing else to be done.

Calling quickSort on elements 2 to 1

The right partition of the partition above is quicksorted.In this case the starting index is greater than the ending index due to the way these are generated: the right partition starts one past the pivot of its parent partition and goes until the last element of the parent partition. So if the parent partition is empty, the indices generated will be out of bounds, and thus no quicksorting will take place.

Calling quickSort on elements 3 to 8

The right partition of the entire array is now being quicksorted 5 is chosen as the pivot.

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

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

The rightwards scan for a large element is initiated.

9 is immediately found.

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

Thus, the leftwards search for a small element begins...

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

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

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

At last, 4 is found. Note k = 5.

1 2 3 5 4 9 6 8 7

Thus the first large and small elements to be found are swapped.

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

The rightwards search for a large element begins anew.

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

Now that it has been found, the rightward search can stop.

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

Since k was stopped at 5, this is the index from which the leftward search resumes.

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

1 2 3 4 5 9 6 8 7

The last step for this partition is moving the pivot into the right spot.

Thus the left partition consists only of the element at 3 and the right partition is

spans positions 5 to 8 inclusive.

Calling quickSort on elements 3 to 3

The left partition is quicksorted (although nothing is done) .

Calling quickSort on elements 5 to 8

The right partition is now passed into the quicksort function.

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

9 is chosen as the pivot.

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

The rightward search for a large element begins.

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

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

No large element is found. The search stops at the end of the partition.

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

The leftwards search for a small element begins, but does not continue since the search indices i and k have crossed.

1 2 3 4 5 7 6 8 9

The pivot is swapped with the element at the position k: this is the last step in splitting this partition into left and right subpartitions.

Calling quickSort on elements 5 to 7

The left partition is passed into the quicksort function.

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

6 is chosen as the pivot.

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

The rightwards search for a large element begins from the left end of the partition.

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

The rightwards search stops as 8 is found.

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

The leftwards search for a small element begins from the right end of the partition.

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

Now that 6 is found, the leftwards search stops. As the search indices have already crossed, no swap is performed.

1 2 3 4 5 6 7 8 9

So the pivot is swapped with the element at position k, the last element compared to the pivot in the leftwards search.

Calling quickSort on elements 5 to 5

The left subpartition is quicksorted. Nothing is done since it is too small.

Calling quickSort on elements 7 to 7

Likewise with the right subpartition.

Calling quickSort on elements 9 to 8

Due to the "sort the partition startitng one to the right of the pivot"

construction of the algorithm, an empty partition is passed into the quicksort function. Nothing is done for this base case.

1 2 3 4 5 6 7 8 9

Finally, the entire array has been sorted.

12.2.2 Binary Search

A binary search or half-interval search algorithm finds the position of a specified value (the input "key") within a sorted array. At each stage, the algorithm compares the input key value with the key value of the middle element of the array. If the key matches, then a matching element has been found so its index, or position, is returned. Otherwise, if the sought key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the input key is greater, on the sub-array to the right. If the remaining array to be searched is reduced to zero, then the key cannot be found in the array and a special "Not found" indication is returned.

A binary search halves the number of items to be checked with each iteration, so locating an item (or determining its absence) takes logarithmic time. A binary search is a dichotomic divide and conquer search algorithm.

Generally, to find a value in unsorted array, we should look through elements of an array one by one, until searched value is found. In case of searched value is absent from array, we go through all elements.

Situation changes significantly, when array is sorted. If we know it, random access capability can be utilized very efficiently to find searched value quick. Cost of searching algorithm reduces to binary logarithm of the array length. For reference, $\log_2(1\ 000\ 000)$ H" 20. It means, that in worst case, algorithm makes 20 steps to find a value in sorted array of a million elements or to say, that it doesn't present it the array.

12.2.2.1 Example

Suppose we have to search 5 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}.

Step 1 (middle element is 19 > 6): **-1 5 6 18 19** 25 46 78 102 114

Step 2 (middle element is 6 > 5): **-1 5 6** 18 19 25 46 78 102 114

Step 3 (middle element is 5 == 5): **-1 5** 6 18 19 25 46 78 102 114

Example 2. Find 103 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}.

Step 1 (middle element is 19 < 103): -1 5 6 18 19 **25 46 78 102 114**

Step 2 (middle element is 78 < 103): -1 5 6 18 19 25 46 78 **102 114**

Step 3 (middle element is 102 < 103): -1 5 6 18 19 25 46 78 102 114

Step 4 (searched value is absent): -1 5 6 18 19 25 46 78 102 114

12.2.2.2 Algorithm

Algorithm is quite simple. It can be done either recursively or iteratively:

1. Get the middle element;

2. If the middle element equals to the searched value, the algorithm stops;

3. Otherwise, two cases are possible:

- searched value is less, than the middle element. In this case, go to the step 1 for the part of the array, before middle element.

- searched value is greater, than the middle element. In this case, go to the step 1 for the part of the array, after middle element.

Now we should define, when iterations should stop. First case is when searched element is found. Second one is when subarray has no elements. In this case, we can conclude, that searched value doesn't present in the array.

12.3 Greedy Method:

It is used to solve optimization problems. These problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes a given objective function. The feasible solution that does this is called an optimal solution.

Consider one input at a time. At each stage, a decision is made regarding whether a particular input is an optimal solution. If the induction of input will result in an infeasible solution, then this input is not added to the particular solution.

12.3.1 Algorithm

```
Algorithm Greedy(a,n)
{       solution i=ϕ
        for i= 1 to n do
        {
                X= select (a);
                if feasible (solution,x) then
                        solution= union (solution,x)
        }
        Return solution;
}
```

select function selects an input from a[] and remove it and assign to x.

feasible is a boolean valued function that determines whether x can be included into the solution vector.

function union combines x with the solution and updates the objective function.

12.3.2 Fractional Knapsack Problem

Suppose there are n objects and a knapsack bag. Any object i has weight w and the knapsack has the capacity m. If a fraction x; 0d"xd"1 of object i placed into the knapsack, then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit control. The profits and weights are positive numbers.

Strategy:

» Sort objects in non-increasing order of profit $p_i = v_i/w_i$

That is, $p_1=v_1/w_1$ $p_2=v_2/w_2$ ... $p_n=v_n/w_n$

» Consider objects by increasing order of subscript.

» When an object is considered, choose a maximal amount such that the knapsack capacity is not violated.

12.3.2.1 Example

Suppose we have 3 objects. Maximum weight capacity of knapsack is 20. Now we have to find the best possible solution to keep the objects into knapsack.

n=3

m=20

(p1,p2,p3)= (25,24, 15)

(w1,w2,w3)=(18,15,10)

Here n is number of objects

m is weight capacity

p_i is corresponding profit of item i

w_i is corresponding weight of item i

x_i is the fractional part of object i

Solution1: When we consider profit only first we select an object having the largest profit value i.e. p1= 25. So it is placed into knapsack first. Then x1=1 and profit of 25 is earned and 18 units are placed. Only 2 units of knapsack capacity are left. Object two has the same largest profit (p2=24) its weight is (w2=15) but it does not fit into the knapsack. X2=2/15 fills the knapsack exactly with value 2. Value of resulting solution is 28.5

Solution 2: Here we consider the objects in non increasing profit values that requires to considers the objects in order of non decreasing weights. We select an object having the smallest weight value i.e. w3= 10. So it is placed into knapsack first. Then x3=1 and profit of 15 is earned. 10 units of knapsack capacity are left. Object two has the next largest weight (w2=15) its profit is (p2=24) but it does not fit into the knapsack. X2=10/15 = 2/3 fills the knapsack exactly with value 10. Value of resulting solution is 31.

Solution 3: Here we have tried to achieve a balance between the rate at which profit increases and rate at which capacity is used. At each step we include an object having maximum profit of per unit of capacity is used. This means that object are considered in order of profit of the ratio p_i/w_i. Object having the largest p_i/w_i i.e. x2 is select first. So it is placed into knapsack first. Then profit of 24 is earned and it fills 15 units of knapsack. 5 units of knapsack capacity are left. Object three has the next largest p_i/w_i value but it does not fit into the knapsack. So X3=1/2 fills the knapsack exactly with value 5. Value of resulting solution is 31.5.

Of all the feasible solutions solution 3 provides maximum profit. We can see it in table 12.1.

Solutions	(x1,x2,x3)	w_ix_i	p_ixi
1	(1, 2/15, 0)	20	28.2 (w1*x1+w2*x2+w3*x3)
2	(0, 2/3, 1)	20	31(w1*x1+w2*x2+w3*x3)

3 (0, 1, 1/2) 20 31.5(w1*x1+w2*x2+w3*x3)

Solutions	(x1,x2,x3)	w_ix_i	p_ixi
1	(1, 2/15, 0)	20	28.2 (w1*x1+w2*x2+w3*x3)
2	(0, 2/3, 1)	20	31 (w1*x1+w2*x2+w3*x3)
3	(0, 1, 1/2)	20	31.5 (w1*x1+w2*x2+w3*x3)

Table 12.1

12.3.2.2 Algorithm

```
fractionalKnapsack(V, W, capacity, n, KnapSack)
{
sortByDescendingProfit(V,W,n)
KnapSack = 0;
capacityLeft = C;
for (i = 1; (i <= n) && (capacityLeft > 0); ++i)
{
if (W[i] < capacityLeft)
{
KnapSack[i] = 1;
capacityLeft -= W[i];
}
else
{
KnapSack[i] = capacityLeft/W[i];
capacityLeft = 0;
}
}

}
```

12.3.3 Spanning Tree

A *spanning tree* of a graph is just a sub graph that contains all the vertices. A graph may have many spanning trees; for instance the complete graph shown in figure 12.3 contains sixteen spanning trees. The list of spanning trees is shown in figure 12.4.

Figure 12.3: A graph having four vertices

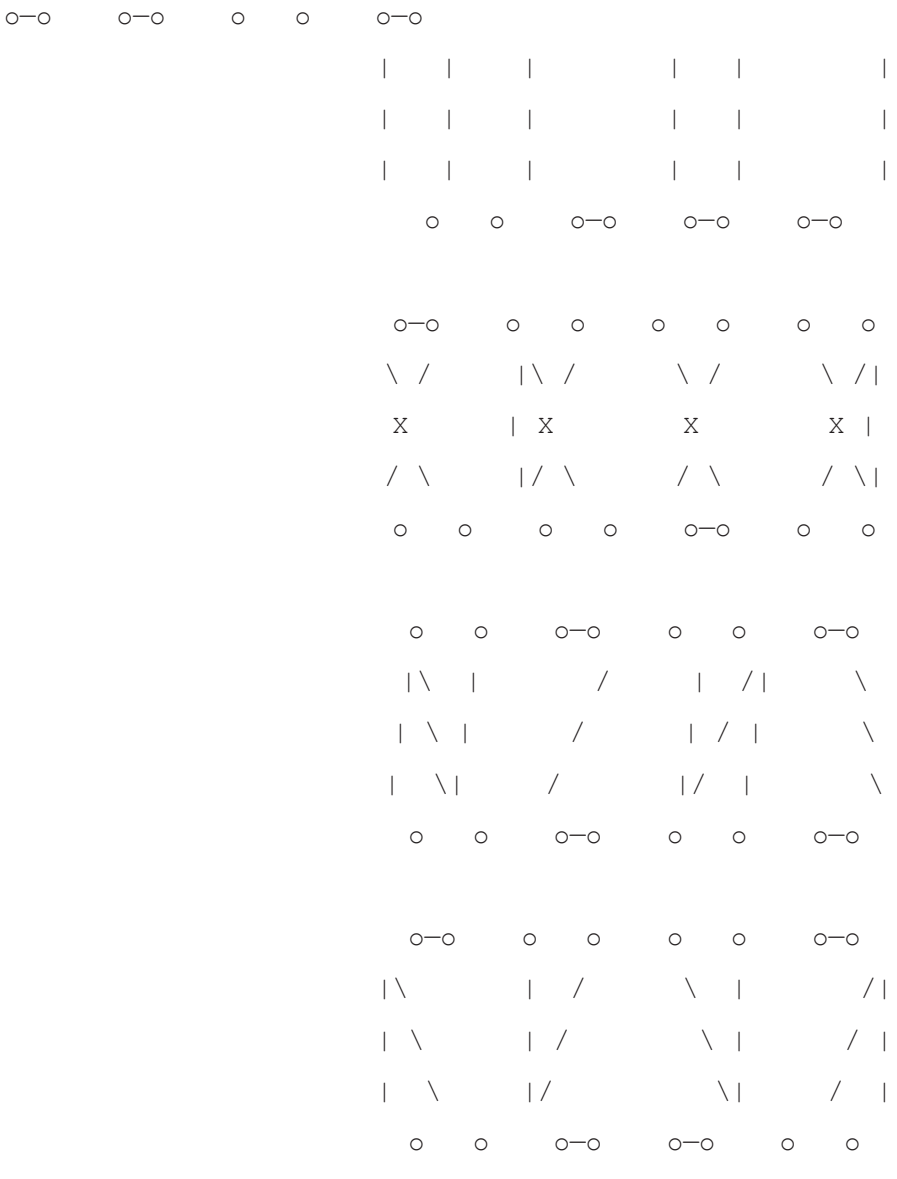


Figure 12.4 A list of spanning trees

A spanning tree for a connected, undirected graph, G=(V,E) is a connected subgraph of G that forms an undirected tree incident with each vertex.

In a weighted graph G=(V,E,W), here the weight of a subgraph is the sum of the weights of the edges in the subgraph.

12.3.4 Minimum Spanning Tree

Now suppose the edges of the graph have weights or lengths. The weight of a tree is just the sum of weights of its edges. Obviously, different trees have different lengths. Now the problem is how to find the minimum length spanning tree?

Minimum Spanning Tree (MST) is a spanning tree having minimum weight.

12.3.4.1 Applications

1. The standard application is to a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. It should be a spanning tree, since if a network isn't a tree you can always remove some edges and save money.

2. A less obvious application is that the minimum spanning tree can be used to approximately solve the traveling salesman problem. A convenient formal way of defining this problem is to find the shortest path that visits each point at least once.

12.3.5 Prim's Algorithm

One of the classic algorithms for this problem is that found by Robert C. Prim. It's a greedy style algorithm and it's guaranteed to produce a correct result.

A greedy method to obtain a minimum cost spanning tree builds this tree edge by edge. The next edge to include is chosen according to solve optimization criteria. Simplest such criteria are to choose an edge that result in a minimum increase in the sum of the cost of the edges so far included. The A is the set of edges.

The algorithm (greedily) builds the minimal spanning tree by iteratively adding nodes into a working tree. Its steps are as follows:

1. Start with a tree which contains only one node.

2. Identify a node (outside the tree) which is closest to the tree and add the minimum weight edge from that node to some node in the tree and incorporate the additional node as a part of the tree.

3. If there are less then $n - 1$ edges in the tree, go to **2**

12.3.5.1 Example

Start with only node A in the tree.

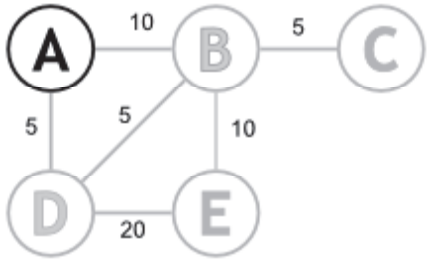


Figure 12.5

Find the closest node to the tree, and add it.

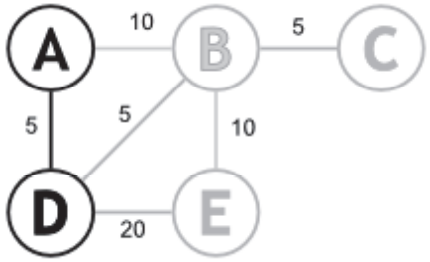


Figure 12.6

Repeat until there are $n - 1$ edges in the tree.

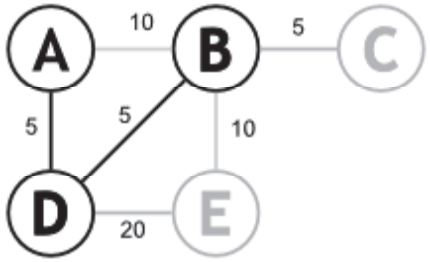


Figure 12.7

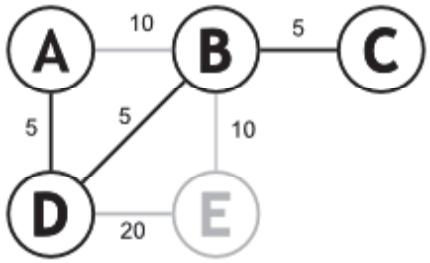


Figure 12.8

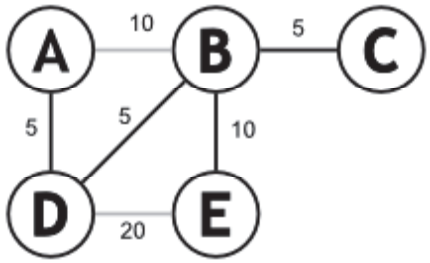


Figure 12.9

We consider a weighted connected graph G with n vertices. Prim's algorithm finds a minimum spanning tree of G.

12.3.5.2 Algorithm

procedure Prim(G: weighted connected graph with n vertices)

T := a minimum-weight edge

for i = 1 to n " 2

begin

e := an edge of minimum weight incident to a vertex in T and not forming a circuit in T if added to T

T := T with e added

end

return(T)

12.3.6 Kruskal Algorithm

Kruskal's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm.

12.3.6.1 Example

Step 1. In the graph, the Edge(*g*, *h*) is shortest. Either vertex *g* or vertex *h* could be representative. Lets choose vertex g arbitrarily.

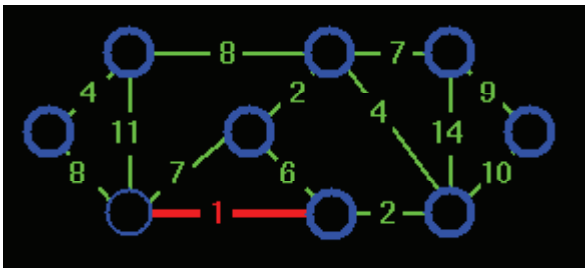


Figure 12.10

Step 2. The edge (*c*, *i*) creates the second tree. Choose vertex *c* as representative for second tree.

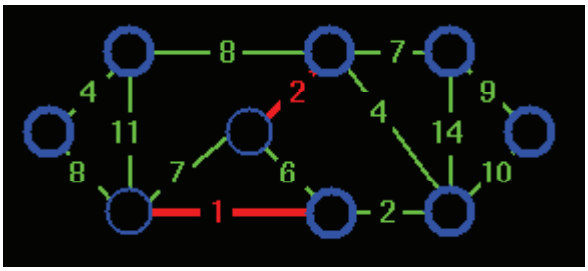


Figure 12.11

Step 3. Edge (g, f) is the next shortest edge. Add this edge and choose vertex g as representative.

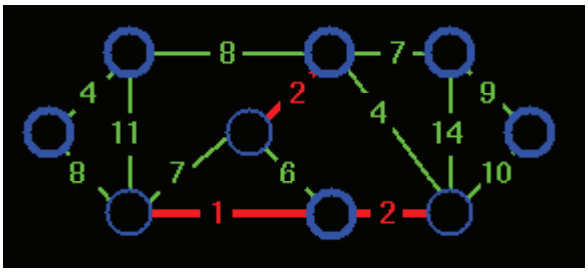


Figure 12.12

Step 4. Edge (a, b) creates a third tree.

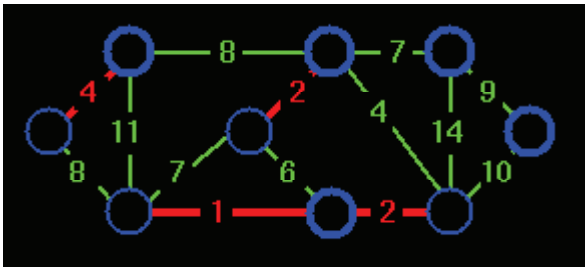


Figure 12.13

Step 5. Add edge (c, f) and merge two trees. Vertex c is chosen as the representative.

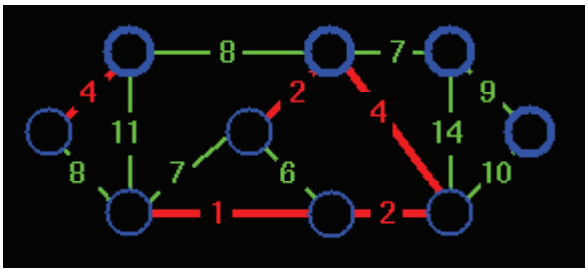


Figure12.14

Step 6. Edge (g, i) is the next next cheapest, but if we add this edge a cycle would be created. Vertex c is the representative of both.

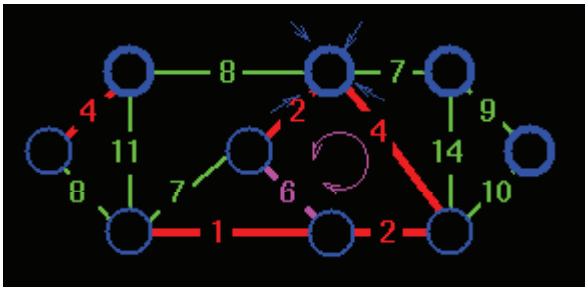


Figure 12.15

Step 7. Instead, add edge (c, d).

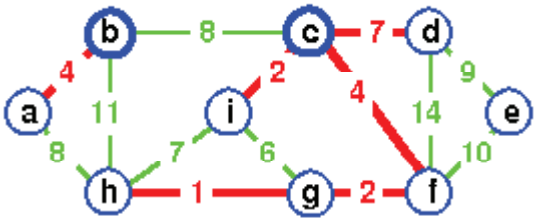


Figure 12.16

Step 8. If we add edge (h, i), edge(h, i) would make a cycle.

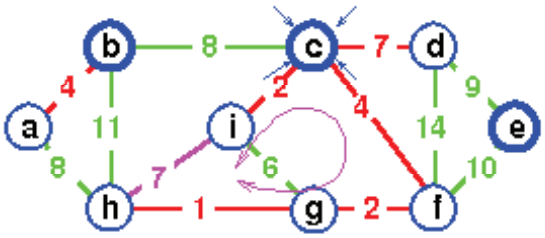


Figure 12.17

Step 9. Instead of adding edge (h, i) add edge (a, h).

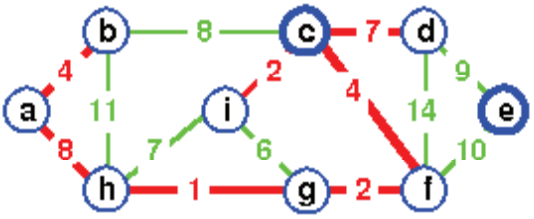


Figure 12.18

Step 10. Again, if we add edge (b, c), it would create a cycle. Add edge (d, e) instead to complete the spanning tree. In this spanning tree all trees joined and vertex c is a sole representative.

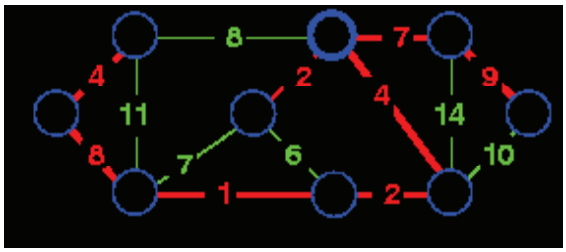


Figure 12.19

12.3.6.2 Algorithm

E(1) is the set of the sides of the ***minimum genetic tree***.

E(2) is the set of the remaining sides.

E(1)=0,**E**(2)=E

While **E**(1) contains less then ***n-1*** sides and **E**(2)=0 do

From the sides of **E(2)** choose one with minimum cost—>e(ij)

E(2)=**E(2)**-{e(ij) }

If **V(i)**,**V(j)** do not belong in the same tree then

unite the trees of **V(i)** and **V(j)** to one tree.

end (If)

end (While)

12.4 Summary

Designing great software is not just about performance. Here is the list of things more important than performance like modularity, correctness, maintainability, security, functionality, robustness, user-friendliness, programmer's time, simplicity, extensibility, reliability, and scalability. This unit discusses the various algorithms that focus on these issues.

12.5 Self Assessment Questions

1. Briefly describe the basic idea of quicksort
2. What are the advantages and disadvantages of quicksort?
3. Which applications are not suitable for quicksort and why?
4. Prove that the fractional knapsack problem has the greedy-choice property.
5. Give the definition of the 0-1 knapsack problem.
6. Give the definition of the minimum spanning trees.
7. Discuss the Prim's algorithm.

Unit - 13 : Algorithm Design Strategies : Dynamic Programming and Branch and Bound

Structure of Unit:

13.0 Objectives

This chapter provides a general overview of

- Dynamic Programming
- Difference between dynamic programming and other design methods
- 0-1 Knapsack Problem
- Branch and Bound method
- Assignment Problem

13.1 Introduction

There are several algorithms design techniques like divide & conquer, greedy, backtracking, etc. Some of them you have already learned (divide & conquer, greedy) and they can solve a variety of computational tasks. The drawback of these is that they can only be used on very specific types of problems. We now discuss two more methods dynamic programming and branch & bound, that can be used when more specialized methods, fail.

13.2 Dynamic Programming

13.2.1 Introduction

Dynamic Programming is a method for efficiently solving a broad range of to search and optimization problems. The basic idea in dynamic programming is avoid calculating the same things twice, usually by keeping a table of known results which is filled when sub stances are solved. In other words, it computes the solution to the sub problems once and stores the solution in a table, so that they can be reused later. It is a bottom up approach. We usually start with the smallest sub instances and by combining their solutions, we obtain answers to sub instances of increasing size, until finally we arrive at the solution of the original instance. It is based on principal of optimality.

Principle of Optimality

Definition: A problem is said to satisfy the Principle of Optimality if the subsolutions of an optimal solution of the problem are themesleves optimal solutions for their subproblems.

Examples:

- The shortest path problem satisfies the Principle of Optimality.
- This is because if a,x1,x2,...,xn,b is a shortest path from node a to node b in a graph, then the portion of xi to xj on that path is a shortest path from xi to xj.
- The longest path problem, on the other hand, does not satisfy the Principle of Optimality. Take for example the undirected graph G of nodes a, b, c, d, and e, and edges (a,b) (b,c) (c,d) (d,e) and (e,a). That is, G is a ring. The longest (noncyclic) path from a to d to a,b,c,d. The sub-path from b to c on that path is simply the edge b,c. But that is not the longest path from b to c. Rather, b,a,e,d,c is the longest path. Thus, the subpath on a longest path is not necessarily a longest path.

13.2.2 Comparision with other design methods

13.2.2.1 Dynamic Programming vs Divide & Conquer

Divide & Conquer is a top down method and dynamic programming is bottom up method. The divide & conquer computes a solution to the same instances many time, while in dynamic programming solves every sub instance exactly once.

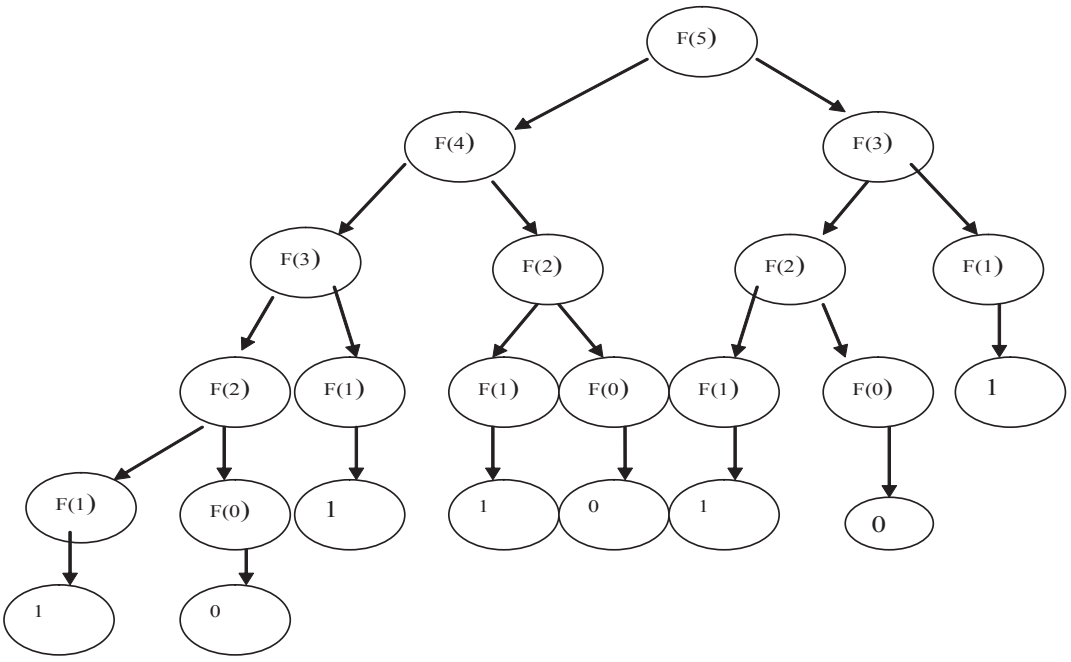
In Divide and Conquer

F(n) = F(n-1) + F(n-2) , n>= 0

F(0) =0 and F(1)=1

We compute F(n-1) = F(n-2) +F(n-3) and F(n-2) =F(n-3)+F(n-4) etc. We have to compute values of F(1), F(2), F(3)……so many times

For example : Parse tree for F(5) is



No of computations for computing F(5)

Function	Number of Time Computed
F(4)	1
F(3)	2
F(2)	3
F(1)	5
F(0)	3

In Dynamic programming, We start with F(0)=0 and F(1)=1

Then compute F(2)= F(1) +F(0) =1

F(3)= F(2)+F(1) = 1 + 1 = 2

F(4)= F(3) + F(2) = 2 + 1 = 3

F(n) = F(n-1) + F(n-2)

Every F(i), where i=0,1,2….,n, computed exactly once.

13.2.2.2 Greedy vs Dynamic programming

Both techniques are optimization techniques, and both build solutions from a collection of choices of individual elements.

- In greedy method only once decision sequence is ever generated, while in dynamic programming many decision sequences may be generated. However, sequences containing suboptimal subsequence can't be optimal and so will not be generated.
- The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices. Dynamic programming computes its solution bottom up by synthesizing them from smaller subsolutions, and by trying many possibilities and choices before it arrives at the optimal set of choices.
- There is no a priori test by which one can tell if the Greedy method will lead to an optimal solution, but dynamic programming always gives the optimal solution

13.2.3 Steps in development of a Dynamic Programming algorithm

Step 1:

Define structure: Characterize the structure of an optimal solution. For this decompose the problem into smaller problems and find a relation between the structure of the optimal solution of the original problem and the solution of the smaller problem.

Steps 2 :

Principle of Optimality : Recursively define the value of an optimal solution. Express the solution of the original problem in terms of optimal solution for smaller problems.

Steps 3 :

Bottom Up Computation : Compute the value of an optimal solution in a bottom up fashion by using a table structure.

Steps 4 :

Construction of optimal solution : Construct a optimal solution from computed information. Steps 3 and 4 may be combined.

13.2.4 Example : 0-1 Knapsack Problem

The Knapsack problem is the classic integer linear programming problem with a single constraint. Let we have n object and a knapsack (or bag) having capacity b. Each object has some weight and associated profit. The problem is to select objects out of n total number of objects such that total profit of selected object is maximum such that total weight of selected object is less than or equal to capacity of knapsack, every object is selected (1) or not selected (0). The 0-1 Knapsack problem is formulated as follows:

max c1x1 + c2x2 + ... + cnxn

subject to: a1x1 + a2x2 + ... + anxn <= b

with xi = 0, 1 for i = 1, 2, ..., n.

Where ci represents the profit associated with selection of item and the ai represents the weight of item i. The constant b represents the capacity of knapsack.

To solve this problem one looks for an idea that expresses the solution in terms of subproblems. This idea is typically expressed recursively. For the 0-1 knapsack problem, the classic approach is to solve the problem for one item at a time. Think of solving the problem for every weight 0 through b for one item at a time. In the "no item" case obviously the maximum value is 0 no matter what the weight; this takes the form of the base case for a recursive formulation. Clearly the first item cannot be placed in the knapsack until the weight reaches a1 at which point the optimal value achieved is c1. At each weight y for the item k it needs to be determined if not using the item, the value of Knap(k-1, y), is better than using the item Knap(k-1, y-ak) + ck (or ck if y = ak) at the weight y. Clearly y-ak >= 0 for item k to fit into the knapsack. Note that in the case of selecting item k for the knapsack the value of Knap(k-1, y-ak) is examined - choosing k-1 as the index of the subproblem (which yields a solution using the first k-1 items) guarantees that the item k is selected at most once.

The complete recursive formulation of the solution

Knap(k, y) = Knap(k-1, y) if y < a[k]

Knap(k, y) = max{ Knap(k-1, y), Knap(k-1, y-a[k])+ c[k] } if y > a[k]

Knap(k, y) = max { Knap(k-1, y), c[k] } if y = a[k]

Knap(0, y) = 0

Implementing a dynamic programming problem solution using recursion often leads to a exponential algorithm; in this case that does not necessarily occur. This is due to the manner in which the reduction of the second parameter y is done in the recursion. The dynamic programming solution utilizes an iterative algorithm that builds a 2-dimensional matrix of size n+1 x b. The matrix has a initial row

which reflects the base cases - the entries will be 0 representing the value of selecting no items to place in the knapsack for each weight y, y= 1, ..., b.

The row i entries represent solutions to the following subproblems, find the max value for the knapsack using the first i items subject to the constraint that weight in the knapsack is less than or equal to y. If i = 1, then the maximum value the knapsack can achieve is c[0] - the value of selecting the first item and this is achieved for all y >= a[0].

To see how the iterative algorithm produces a solution a trace of the dynamic program is examined.

Example Trace

Suppose a[] = [4, 3, 2, 1], c[] = [7, 5, 3, 1] and b = 6. By observation the solution to the problem yields a maximum value of 13. The solution x[] = [1, 0, 1, 0].

Notice during the trace of the algorithm that each entry represents a solution to a subproblem of the original problem. For example, the third row and fifth column entry is the maximum value of the 0-1 knapsack problem using 2 items and a maximum weight of 4.

The dynamic programming matrix with the initialization of its first row a has the form. The matrix labels are colored gray and the initialized cells in light gray.

items/weights	1	2	3	4	5	6
	0	0	0	0	0	0
1						
2						
3						
4						

The first item has a weight of 4 and appears in the solution at the column with weight 4.

Thereafter, its value will exceed the value of the no items selected solution.

items/weights	1	2	3	4	5	6
	0	0	0	0	0	0
1	0	0	0	7	7	7
2						
3						
4						

The second item has a weight of 3 and appears in the solution at the column with weight 3. It is selected for column 3 since 5 + k[1][3-3] exceeds that of knap[1][3]. Thereafter, knap[1][y] > knap[1][y-3] + 5 for the weights 4 through 6 inclusive.

items/weights	1	2	3	4	5	6
	0	0	0	0	0	0
1	0	0	0	7	7	7
2	0	0	5	7	7	7
3						
4						

The third item has a weight of 2 and appears first in the solution in the column with weight 2. Item 3 is selected since its value 3 + k[2][2-2] exceeds that of knap[2][2]. For the next two entries, knap[2][y] > knap[2][y-2] + 3 for y = 3 and 4 - so item 3 is not selected. For y = 5, knap[2][5] = 7 and knap[2][5-2] + 3 = 8, so item 3 is again selected. Similarly for y= 6, knap[2][6] = 7 and knap[2][6-2] + 3 = 10, so item 3 is selected.

items/weights	1	2	3	4	5	6
	0	0	0	0	0	0
1	0	0	0	7	7	7
2	0	0	5	7	7	7
3	0	3	5	7	8	10
4						

The fourth item has a weight of 1 and it appears in the solution at the column with weight 1 since its value 1 + k[3][1-1] exceeds that of knap[3][1]. Thereafter, knap[3][y] >= knap[3][y-1] + 1 for the weights 2 through 6 inclusive.

items/weights	1	2	3	4	5	6
	0	0	0	0	0	0
1	0	0	0	7	7	7
2	0	0	5	7	7	7
3	0	3	5	7	8	10

The maximum value for this knapsack problem is in the bottom leftmost entry in the matrix, knap[4][5].

13.3 Branch and Bound (B&B)

13.3.1 Introduction

Branch and bound is a systematic method for solving optimization problems It is rather general optimization technique that applies where the greedy method and dynamic programming fail. It often

leads to exponential time complexities in the worst case, but if applied carefully, it can lead to algorithms that run reasonably fast on average.

The general idea of B&B is a Breath First Search (BFS) -like search for the optimal solution, but not all nodes get expanded (i.e., their children generated). Rather, a carefully selected criterion determines which node to expand and when, and another criterion tells the algorithm when an optimal solution has been found.

The idea can be strengthened to deal with optimization problems that minimize or maximize an objective function, usually subject to some constraints: tour length, value of items selected, cost of an assignment, etc.

Backtracking Vs Branch & Bound

Backtracking is based on exhaust search, where we have to search all possible ways to solve any problem. It requires substructure determine promising sub problems to explore and violate problem constraints.

Branch-and-bound is based on taking the best decision on every possible branching by applying bound function on them. It requires two additional items expect then substructure:

2. The value of the best solution seen so far.
3. A way to provide, for every node in the search-space tree, a bound on the best value of the objective function (lower bound for minimization, upper bound for maximization) on any solution that can be obtained by adding further components to the partial solution represented by the node.

13.3.2 Steps in development of a Branch and Bound algorithm

The first idea of B&B is to develop "a predictor" of the likelihood (in a loose sense) of a node in the solution tree that it will lead to an optimal solution. This predictor is quantitative. With such a predictor, the B&B works as follows:

- Which node to expand next: B&B chooses the live node with the best predictor value
- B&B simply expands that node (i.e., generate all its children)
- the predictor value of each newly generated node is computed, the just expanded node is now designated as a dead node, and the newly generated nodes are designated as live nodes.
- Termination criterion: When the best node chosen for expansion turn out to be a final leaf (i.e., at level n), that when the algorithm terminates, and that node corresponds to the optimal solution. The proof of optimality will be presented later on. A search path is terminated for any of the following reasons:

1. The value of the node's bound is not better than the best so far.
2. The node represents no feasible solutions because the constraints are violated.
3. The subset of feasible solutions represented by the node consists of a single point (i.e. no further choices can be made). In this case, if the objective function value is better than the best solution so far, it becomes the new best solution.

The value of predictor will depend on the problem. In the case of minimization problem, one candidate predictor of any node is the cost so far . That is, each node corresponds to (partial) solution (from the root to that node). The cost-so-far predictor is the cost of the partial solution.

13.3.3 Example : Assignment Problem

Assigning n people to n jobs so that the total cost is minimized. Each person does one job and each job is assigned to one person.

4 jobs, 4 people.

Read the assignments as <Job 1, Job 2, Job 3, Job 4>:

$$C = \begin{array}{c} \begin{matrix} \text{Job 1} & \text{Job 2} & \text{Job 3} & \text{Job 4} \end{matrix} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{matrix} \\ \text{Person } a \\ \text{Person } b \\ \text{Person } c \\ \text{Person } d \end{matrix}$$

<a,b,c,d> cost=9+4+1+4=18

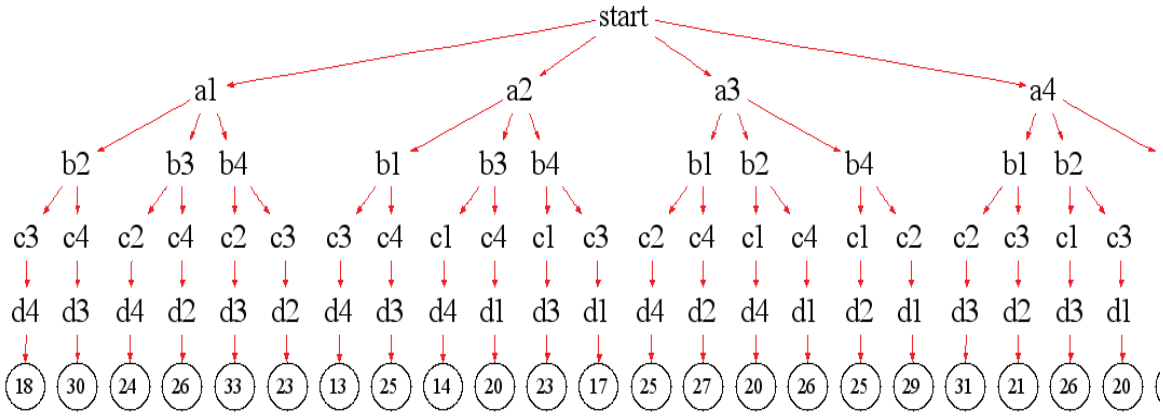
<a,b,d,c> cost=9+4+9+8=30

<a,d,b,c> cost=9+6+4+8=27

<d,a,b,c> cost=7+2+3+8=20

<d,c,b,a> cost=7+8+3+8=26 etc. totaling 4! permutations

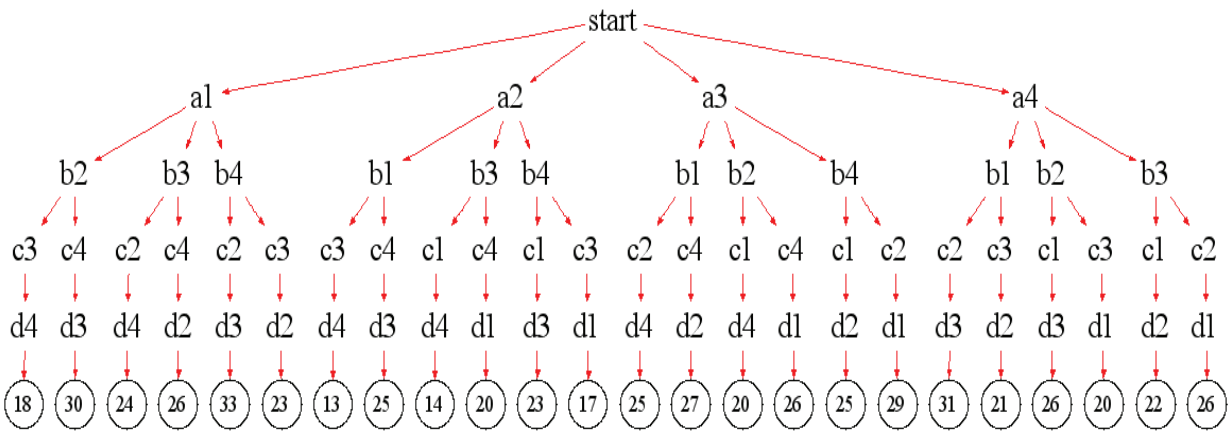
The resulting state-space for assigning Jobs {1, 2, 3, 4} to each person {a, b, c, d} is:



$$C = \begin{array}{c} \begin{matrix} \text{Job 1} & \text{Job 2} & \text{Job 3} & \text{Job 4} \end{matrix} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{matrix} \\ \text{Person } a \\ \text{Person } b \\ \text{Person } c \\ \text{Person } d \end{matrix}$$

From the table above, the rightmost branch <d,c,b,a>, cost=$7^{d1} + 8^{c2} + 3^{b3} + 8^{a4}$=26

Minimum-Cost: Generate n! permutations, evaluating the objective function (i.e. the cost) of each and keeping track of the minimum.



Best-first: A best-first search reduces the number of possible solutions considered by always selecting the best available.

This implies that we have some initial cost that can be compared with new calculated costs to determine the best of the new ones.

We could simply pick initial job assignments randomly, assigning Job 1 to a, 2 to b, etc. giving a cost of [9+4+1+4]=18.

Ideally, we would like to use the best feasible cost as the initial cost but that is the problem we're trying to solve.

By starting with an estimated cost near the optimum the best-first algorithm will be more efficient, reducing the number of nodes in the search tree explored.

Our initial assignments are not legal, we'll have some jobs not assigned to anyone, but will be lowest cost bound possible.

1. All persons must be assigned a Job.
2. A person always gets the Job they do the cheapest.
3. Ensures we start with an initial cost i.e. optimum cost

person	a	b	c	d
Job	2	3	3	4
cost	2	3	1	4

Initial cost: [2+3+1+4]=10

the initial *lowest bound*.

We can now generate legal job assignments for person *a* by replacing the initial (illegal but lowest cost) assignments with legal ones as below.

$$C = \begin{array}{c} \begin{matrix} \text{Job 1} & \text{Job 2} & \text{Job 3} & \text{Job 4} \end{matrix} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{matrix} \\ \text{Person } a \\ \text{Person } b \\ \text{Person } c \\ \text{Person } d \end{matrix}$$

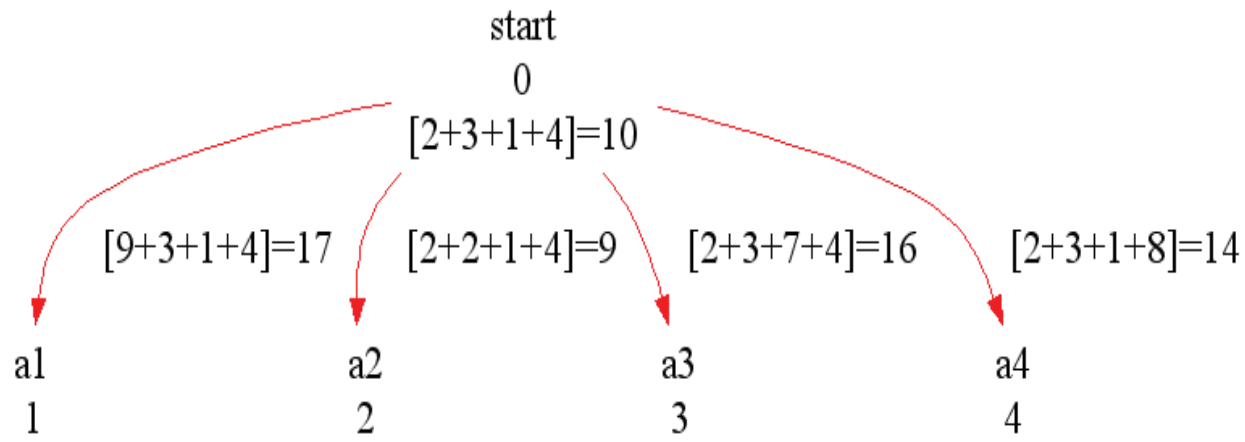
Read all but the initial assignments as [Job 1, Job 2, Job 3, Job 4].

a1, give person a Job 1, cost 9 for total cost = 17.

a2, give person a Job 2, cost 2 for total cost = 9.

a3, give person a Job 3, cost 7 for total cost = 16.

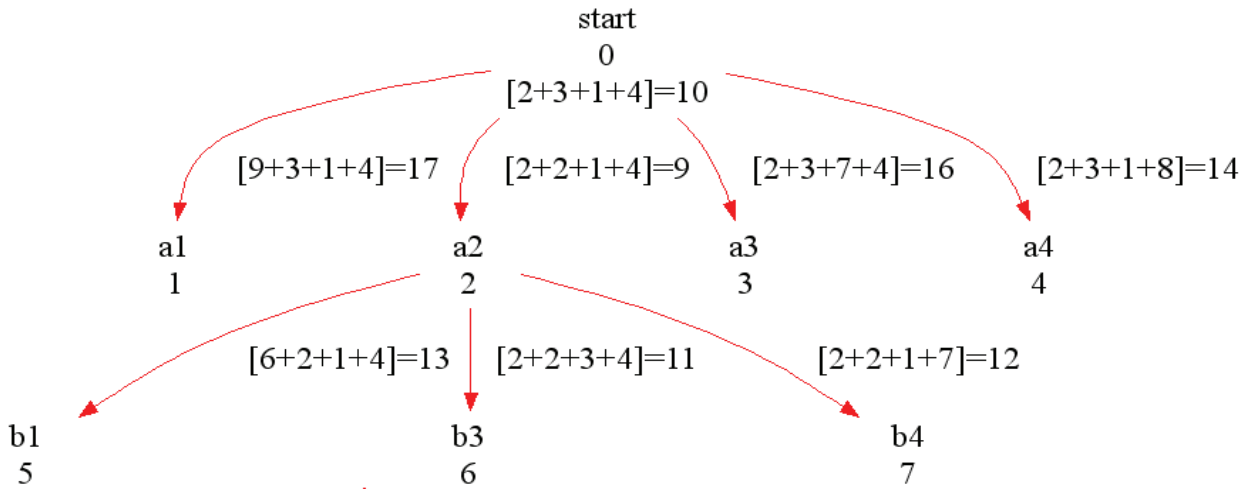
a4, give person a Job 4, cost 8 for total cost = 14.



We can see that the best total cost=9 is assigning Job 2 to person a; that assignment a2 is the best and should be explored first.

For person b, we generate all assignments except to Job 2 since that has already been assigned, a2.

$$C = \begin{array}{c} \begin{matrix} \text{Job 1} & \text{Job 2} & \text{Job 3} & \text{Job 4} \end{matrix} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{matrix} \\ \text{Person } a \\ \text{Person } b \\ \text{Person } c \\ \text{Person } d \end{matrix}$$



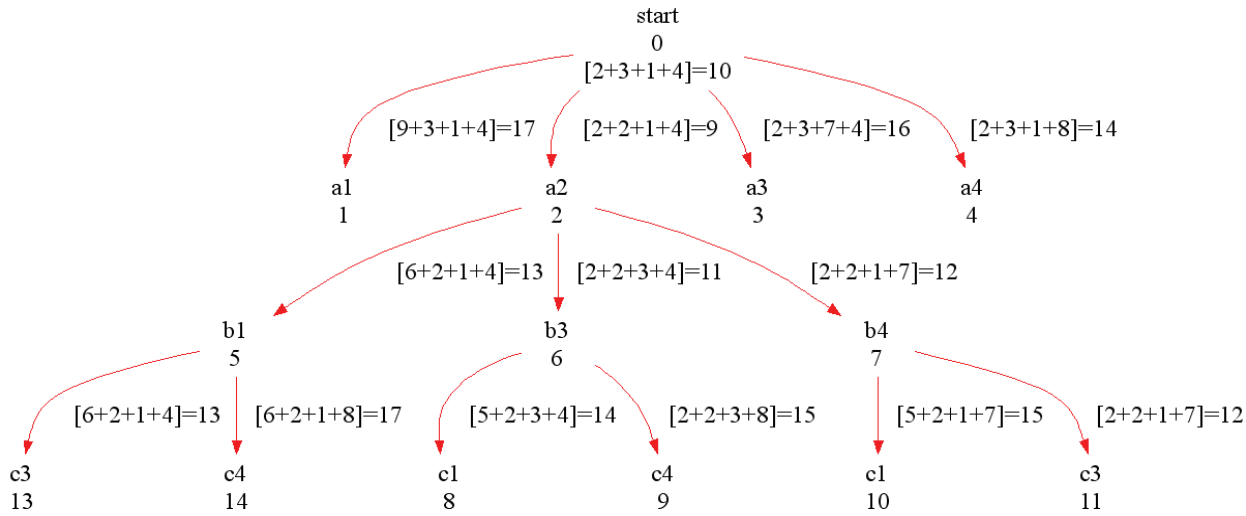
The best total cost=11 is assigning Job 3 to b.

On exploring node 6 to node 8 (assigning Job 1 to c cost=14) and 9 (assigning Job 4 to c cost=15), we discover that assigning Job 4 to b (cost=12) now is best.

You'll notice we also eventually try assigning Job 1 to b (cost=13).

$$C = \begin{array}{c} \begin{matrix} \text{Job 1} & \text{Job 2} & \text{Job 3} & \text{Job 4} \end{matrix} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{matrix} \\ \text{Person } a \\ \text{Person } b \\ \text{Person } c \\ \text{Person } d \end{matrix}$$

The optimal assignment is [6+2+1+4]=13 for nodes <2,5,13,15>.



The improvement of branch-and-bound is by searching minimum partial solutions first, terminating the search when a solution found.

13.4 Summary

- Dynamic Programming is a method for efficiently solving a broad range of search and optimization problems.
- Dynamic Programming is based on principal of optimality.
- A problem is said to satisfy the Principle of Optimality if the subsolutions of an optimal solution of the problem are themselves optimal solutions for their subproblems.
- Divide & Conquer is a top down method and dynamic programming is bottom up method, while Dynamic is top down.
- In greedy method only once decision sequence is ever generated, while in dynamic programming many decision sequences may be generated.
- The Knapsack problem is the classic integer linear programming problem with a single constraint.
- Branch and bound is a systematic method for solving optimization problems.
- The general idea of B&B is a BFS-like search for the optimal solution, but not all nodes get expanded.
- The first idea of B&B is to develop "a predictor" of the likelihood (in a loose sense) of a node in the solution tree that it will lead to an optimal solution.

13.5 Self Assessment Questions

1. What is the concept of Dynamic Programming?
2. Compare Dynamic Programming with greedy and divide & conquer methods.
3. Explain the 0-1 knapsack problem and write a procedure to solve using dynamic programming.
4. Define the Branch and Bound design method.
5. Discuss the basic steps in solving any problem using branch and bound.
6. What is assignment problem and how to solve it using branch and bound?

Unit - 14 : Analysis of Algorithms : Complexity

Structure of Unit:

14.0 Objectives

This chapter provides a general overview of

- Analysis of Algorithm
- Complexity Types: Space Complexity and Time Complexity
- Asymptotic Notations
- Different notations like, and
- Procedure to find the complexity of any algorithm

14.1 Introduction

There may be many algorithms to solve a problem. There are many criteria upon which we can judge the best algorithm. For instance:

1. Does it do what we want it to do?
2. Does it work correctly according to the original specifications of the task?
3. Is there documentation that describes how to use it and how it works?
4. Are procedures created in such a way that they perform logical sub functions?
5. Is the code reachable?

These criteria are all vitally important when it comes to writing software, most specially for large systems. There are other criteria for judging algorithm that have a more direct relation to performance. Some algorithms are more efficient than others. We would prefer to choose an efficient algorithm, so it would be nice to have metrics for comparing algorithm efficiency. One of them is complexity of algorithm.

The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process. Usually there are natural units for the domain and range

of this function. These are depending on their computing time and storage requirement. The process of finding the complexity is known as analysis of algorithm.

Analysis of algorithm refers to the task of determiny how much computing time and storage is required to execute it. Most algorithms are designed to work with inputs of arbitrary length. Usually the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity).

The running time of an algorithm is influenced by several factors:

- Speed of the machine running the program
- Language in which the program was written. For example, programs written in assembly language generally run faster than those written in C or C++, which in turn tend to run faster than those written in Java.
- Efficiency of the compiler that created the program
- The size of the input: processing 1000 records will take more time than processing 10 records.
- Organization of the input: if the item we are searching for is at the top of the list, it will take less time to find it than if it is at the bottom.

But mainly the quality of an algorithm depends on two parameters:

- How much extra space it require other than the input?
- How much it require for execution for a given input of size n.

We can define these parameters as:

- Space Complexity: The space complexity of an algorithm is the amount of memory it needs to run to completion.
- Time Complexity: The time complexity of an algorithm is the amount of computer time it needs to run to completion.

14.2 Complexity Types

14.2.1 Space Complexity

The space needed by each algorithm is seen to be sum of following components:

1. A fixed part that is independent of the characteristics (e.g. number size) of the inputs and outputs. This part typically includes the instruction space (i.e. space for the code), space for simple variables and fixed size components variables (also called aggregate), space for constants, an so on.
2. A variable part that consists of the space needed by component variables whose size is dependent of the particular problem instance being solved, the space needed by referenced variables (to the extent that this depends on instance characteristics), and the recursion stack space (as the space depends on the instance characteristics).

The space requirement S(P) of any algorithm P may therefore be written as S(P) = c+Sp (instance characteristics), where c is constant. When analyzing the space complexity of an algorithm, we concentrate solely on estimating Sp (instance characteristics).

Example 1

Suppose we want to write a program to compute the sum of a list of numbers. The program goes into a loop (for a predetermined number of iterations equal to the size of the list, for example 100 or 200, ... etc.) asking the user to enter a number and keeps adding these numbers. What is the space complexity of that program?

There are three simple steps that one should follow:

- Identify the parameter(s) that determine the problem size,
- Find out how much space (i.e. memory) is needed for a particular size,
- Find out how much space (i.e. memory) is needed for twice the size considered earlier,
- Repeat step 3 many times until you establish a relationship between the size of the problem and its space requirements. That relationship is the space complexity of that program.

Now, let's apply the above steps to the example at hand.

- The problem size is obviously the size of the list of numbers to be added.
- Let's assume that we have a list of 10 numbers. Obviously, we need a variable where the numbers are to be entered (one at a time), and a variable (initially 0) where a running sum is to be kept. Thus we need two variables.
- Let's assume that we have a list of 20 numbers (twice as before). Obviously, it doesn't matter; we still need only two variables.
- Let's assume that we have a list of 40 numbers (twice as before). Obviously, it still doesn't matter; we still need only two variables.
- Obviously, we may conclude that no matter how many numbers we want to add, we still need a constant number of variables, thus we say that the relationship between problem size and space (i.e. the space complexity of the program) is constant.

Example 2:

```
Algorithm xyz(x,y,z)
{
    Return  x+y+z+(x+y+z)/3;
}
```

The problem instance is characterized by the specific values of x,y and z. We can see that space needed by xyz is independent of the instance characteristics (x,y and z need just need one word to store). Consequently, Sp (instance characteristics) =0.

Example 3:

```
// a is defined as array of floating point numbers
Algorithm sum (a,n)
{
```

```
    s=0.0;
    For i : =1 to n do
        s := s+a[i];
   Return s;
}
```

The problem instance is characterized by n, the number of element to be summed. The space needed by n is one word. The space needed by a is the space needed by variable of type array of floating point numbers. This is at least n words. So Ssum (n) ? (n+3) (n for a[], one for each n, i, and s).

14.2.2 Time Complexity

One of the important criteria in evaluating algorithms is the time it takes to complete a job. To have a meaningful comparison of algorithms, the estimate of computation time must be independent of the programming language, compiler, and computer used; must reflect on the size of the problem being solved; and must not depend on specific instances of the problem being solved.

Time complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm. "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take. This idea of time is separated from "wall clock" time, since many factors unrelated to the algorithm itself can affect the real time (like the language used, type of computing hardware, proficiency of the programmer, optimization in the compiler, etc.). It turns out that, if we chose the units wisely, all of the other stuff doesn't matter and we can get an independent measure of the efficiency of the algorithm.

Following types of time complexities can be found in analysis of any algorithm:

- the worst-case runtime complexity of the algorithm is the function defined by the maximum number of steps taken on any instance of size n.
- the best-case runtime complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of size n.
- the average case runtime complexity of the algorithm is the function defined by an average number of steps taken on any instance of size n.

Example. Let us consider an algorithm of sequential searching in an array of size n.

Its worst-case runtime complexity is O(n)

Its best-case runtime complexity is O(1)

Its average case runtime complexity is O(n/2)=O(n)

Analyzing Running Time

T(n), or the running time of a particular algorithm on input of size n, is taken to be the number of times the instructions in the algorithm are executed. Pseudo code algorithm illustrates the calculation of the mean (average) of a set of n numbers:

1. n = read input from user

2. sum = 0

3. i = 0

4. while i < n

5. number = read input from user

6. sum = sum + number

7. i = i + 1

8. mean = sum / n

The computing time for this algorithm in terms on input size n is: T(n) = 4n + 5.

Statement	Number of times executed
1	1
2	1
3	1
4	n+1
5	n
6	n
7	n
8	1

14.3 Asymptotic Notations

Asymptotic notations are used to analysis algorithm asymptotically. First we discuss the need of such analysis. For a small input set, all algorithms perform well regardless of the complexity, so to determine the complexity we need to analyze the algorithm for large values of n , typically n -> infinity. Also we need to bound the computed value between two values(a range) , or in short create a subset within which the execution time of the algorithm falls.

An asymptote of the curve is a line such that the distance between the curve and the line approaches zero as they tend to infinity. So, we use this to determine the function which is

fixed while n-> infinity.

For example T(n) = 2n2 + 3n + 4, describes precisely the number of elements accesses made in the algorithm. From this complexity factor if we lay out the algorithm it looks like:

```
Algorithm sample()
{
  For I = 1 to n do
  {
     S1; s2;
```

```
    For j =  1 to n do
    {
        Is1;
        Is2;
    }
    S3;
}
OS1;
OS2; OS3;OS4;
}
```

It means there are two loops; in inner loops there are 2 statements, while outer loops contains 3 statements. 4 statements are not a part of any loops. But for long term, we can easily see first term 2n2 plays a vital role, as n increases the domination of this terms increases. To represent the most dominating term we use the concept of asymptotic notation and express with the help of asymptotic notations.

Significance of Asymptotic Notations

Asymptotic notations are calculated based on if the asymptotic lower and upper bounds can be calculated for a function properly.

Various Notations

- - when both asymptotic upper and lower bounds can be calculated and is tight.
- - when only asymptotic upper bound can be calculated and is tight.
- - when only asymptotic lower bound can be calculated and is tight.
- - when the asymptotic upper bound can be calculated but cannot be tight.
- -Only when the asymptotic lower bound can be calculated but cannot be tight

14.3.1 Big Oh Notation

Big O notation seeks to describe the relative complexity of an algorithm by reducing the growth rate to the key factors when the key factor tends towards infinity. It is a relative representation of complexity. it doesn't take into account issues such as:

- Memory Usage: one algorithm might use much more memory than another. Depending on the situation this could be anything from completely irrelevant to critical;
- Cost of Comparison: It may be that comparing elements is really expensive, which will potentially change any real-world comparison between algorithms;
- Cost of Moving Elements: copying elements is typically cheap but this isn't necessarily the case; etc.

Definition

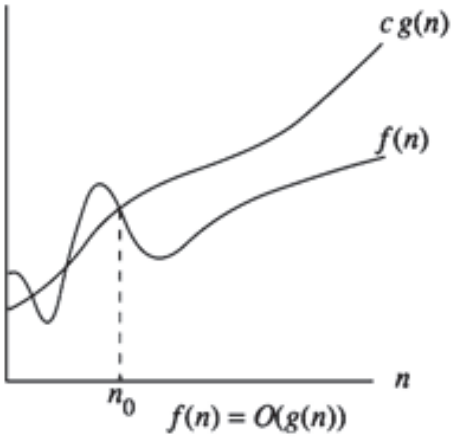
Big-O is the formal method of expressing the upper bound of an algorithm's running time. It's a measure of the longest amount of time it could possibly take for the algorithm to complete. Big O specifically describes the worst-case scenario, and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm.

For any monotonic functions f(n) and g(n) from the positive integers to the positive integers, we say that f(n) = O(g(n)) when there exist constants c > 0 and n0 > 0 such that

f(n) ? c * g(n), for all n ? n0

Intuitively, this means that function f(n) does not grow faster than g(n), or that function g(n) is an upper bound for f(n), for all sufficiently large n??

Here is a graphic representation of f(n) = O(g(n)) relation



Example 1

Suppose f(n) = 5n and g(n) = n.

- To show that f = O(g), we have to show the existence of a constant C as given in Definition . We have set {6,7,8,…..} to choose C.
- We could choose a larger C such as 6, because the definition states that f(n) must be less than or equal to C * g(n), but we usually try and find the smallest one.

Therefore, a constant C exists (we only need one) and f = O(g) = O(n)

Example 2

To show that f(n) = 4n + 5 = O(n), we need to produce a constant C such that:

f(n) <= C * n for all n.

If we try C = 4, this doesn't work because 4n + 5 is not less than 4n. We need C to be at least 9 to cover all n. If n = 1, C has to be 9, but C can be smaller for greater values of n (if n = 100, C can be 5). Since the chosen C must work for all n, we must use 9:

4n + 5 <= 4n + 5n = 9n

Since we have produced a constant C that works for all n, we can conclude:

T(4n + 5) = O(n)

Example 3

T(n) = 10 n3+100n2+55

Since once gets large enough,

Let's be more rigorous: We must find a and such that

From the equation, guess that . Then

Since when , must be greater than and (from the last line above) n must be greater than .

Therefore, if and ,

and that proves that .

Different complexities in terms of big oh notation:

O (1)	O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.	*Example: Find whether number is even or odd*
		Algorithm evenodd(n)
		{
		If (n mod 2 == 0)
		Write ("even");
		Else
		Write ("odd");
		}

O(N)	O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set. Big O notation will always assume the upper limit where the algorithm will perform the maximum number of iterations.	Example : sum of array element of size n.
		
		Algorithm sum(a,n)
		{
		S=0.0;
		For i= 1 to n do
		S = S + a[i];
		Return s;
		}

O(N^2)	O(N^2) represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in O(N^3), O(N^4) etc.	Example : Sorting of an array of size n.
		
		Algorithm sort(a,n)
		{
		For i = 1 to n do
		For j = 1 t n do
		If (a[i] > a[j])
		{
		Temp = a[i];
		a[i] = a[j];
		a[j] = temp;
		}
		}

$O(2^N)$	$O(2^N)$ denotes an algorithm whose growth will double with each additional element in the input data set. The execution time of an $O(2^N)$ function will quickly become very large	Example : Find the any n^{th} term of Fibonacci series
		
		Algorithm Fib(n)
		{
		If (n=0 or n= 1) then return (1);
		Else
		Return (Fib(n-1)+Fib(n-2));
		
		}

O(logN)	O(logn) complexity occur when for every successive execution input size is divided into half . For example Binary search is a technique used to search sorted data sets. It works by selecting the middle element of the data set, essentially the median, and compares it against a target value. If the values match it will return success. If the target value is higher than the value of the probe element it will take the upper half of the data set and perform the same operation against it. Likewise, if the target value is lower than the value of the probe element it will perform the operation against the lower half. It will continue to halve the data set with each iteration until the value has been found or until it can no longer split the data set	//given an array a(l,h) of elements in non-decreasing. order,1<=i<=h, determine whether // x is present and if so, return j such that x=a[j],else return 0.
		
		Algorithm Bin_search(a,l,h,x).
		{
		if(l=h) then
		{
		if(x = a[l]) then return l;
		else return 0;
		}
		else
		{
		mid=[(l+h)/2]
		if(x = a[mid])then return mid ;
		else if(x<a[mid]) then
		return Bin _search(a,l,mid-1,x)
		else return Bin_search(a.mid+1,h,x)
		}
		}

Properties of Big O

- Any kth degree polynomial is O(nk).
- a nk = O(nk) for any a > 0.
- Big O is transitive. That is, if f(n) = O(g(n)) and g(n) is O(h(n)), then f(n) = O(h(n)).
- logan = O(logb n) for any a, b > 1. This practically means that we don't care, asymptotically, what base we take our logarithms to.

- Big O of a sum of functions is big O of the largest function. How do you know which one is the largest? The one that all the others are big O of. One consequence of this is, if f(n) = O(h(n)) and g(n) is O(h(n)), then f(n) + g(n) = O(h(n)).
- f(n) = O(g(n)) is true if lim n->infinity f(n)/g(n) is a constant.

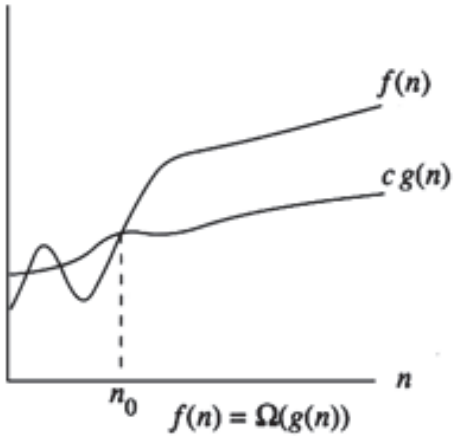
14.3.2 Omega Notation

For any monotonic functions f(n) and g(n) from the positive integers to the positive integers, we say that f(n) = ? (g(n)) when there exist constants c > 0 and n0 > 0 such that

f(n) ? c * g(n), for all n ? n0

Intuitively, this means that function f(n) does not grow slower than g(n), or that function g(n) is an lower bound for f(n), for all sufficiently large n??

Here is a graphic representation of f(n) = ? (g(n)) relation



Example 1

Suppose f(n) = 3n + 2, then g(n) = n.

- To show that f = ? (g), we have to show the existence of a constant c as given in Definition . We have set {---------, 1,2,3} for c.
- We could choose a c such as 3, because the definition states that f(n) must be greater than or equal to c * g(n), but we usually try and find the largest one.

  Therefore, a constant C exists (we only need one) and f = O(g).

  f(n) = 3n + 2 = ? (n) as 3n + 2 ? 3n for n ?0 (c =3 and n0 = 0)

Example 2

F(n) = 10n2 + 4n+ 2

Let g(n)= n2

For c = 10 we have find that 10n2 + 4n+ 2 ? 10 n2 for n ? 1.

(other possible values for c are 9,8,……, but we have to choose largest one)

So F(n) = ? (n2)

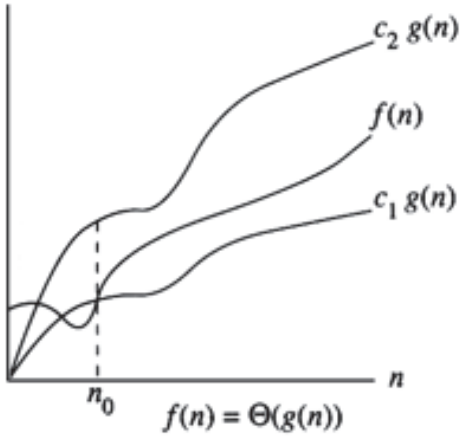
14.3.3 Theta notation

For any monotonic functions f(n) and g(n) from the positive integers to the positive integers, we say that f(n) = (g(n)) when there exist constants c1, c2 > 0 and n0 > 0 such that

c1 * g(n) ? f(n) ? c2 * g(n), for all n ? n0

Intuitively, this means that function f(n) does not grow slower than g(n) and also does not grow faster than g(n) , or that function g(n) is an tight bound for f(n), for all sufficiently large n??

Here is a graphic representation of f(n) = (g(n)) relation



Example 1

Suppose f(n) = 3n + 2, then g(n) = n.

- To show that f = (g), we have to show the existence of a constant c1 and c2 as given in Definition .

c1 {….,1,2,3} and c2 {4,5,6…..}

- We could choose a c1 such as 3, because the definition states that f(n) must be greater than or equal to c1 * g(n), but we usually try and find the largest one.
- We could choose a c2 such as 4, because the definition states that f(n) must be less than or equal to c2 * g(n), but we usually try and find the smallest one.

So, 3n ? 3n+2 ? 4n for n ? 2

F(n) = 3n + 2 = (n) for n ? 2 (c1 =3, c2 = 4 and n0 = 2)

Example 2:

F(n) = 10n2 + 4n+ 2

Let g(n)= n2

- To show that f = (g), we have to show the existence of a constant c1 and c2 as given in definition .

c1 {….,8,9,10} and c2 {11,12,13…..}

- We could choose a c1 such as 10, because the definition states that f(n) must be greater than or equal to c1 * g(n), but we usually try and find the largest one.

- We could choose a c2 such as 11, because the definition states that f(n) must be less than or equal to c2 * g(n), but we usually try and find the smallest one.

For c1 = 10, we have find that 10n2 + 4n+ 2 ? 10n2 for n ? 1. (i.e. c1*g(n) ? f(n))

For c2 = 11, 10n2 + 4n+ 2 ? 11 n2 , for n ? 5 (f(n) ? c2 * g(n))

F(n) = (n2)

14.4 How to Find Complexity?

To find the complexity of any algorithm, first we have to express algorithm as f(n), where n is any instance characteristic like input size. For this we have to compute step count(number of steps required to execute) of an algorithm. The one of the method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement. This figure is often arrived at by first determining the number of steps per execution (s/e) of the statements and the total number of times (i.e.frequency) each statement in executed. The s/e of a statement is the amount by which the count changes as a result of the execution of that statement. By combining these two quantities, the total contribution of each statement is obtained. By adding the contributions of all statements, the steps count for entire algorithm is obtained. Then we convert that step count in any suitable asymptotic notation.

Example 1: Find Maximum element in array

Statement	s/e	Frequency	Total steps
1. Algorithm MaxA(a,n)	0	-	0
2. {	0	-	0
3. max = a[1];	1	1	1
4. for I = 1 to n do	1	n+1	n+1
5. if (max < a[i]) then	1	n	n
6. max = a[i];	1	n	n
7. return max;	1	1	1
8. }	0	-	0
Total			3n+3

We have already done how to convert 3n+3 in asymptotic notation.

If f(n) = 3n + 3, then g(n) = n.

- To show that f = (g), we have to show the existence of a constant c1 and c2 as given in Definition .

  c1 {….,1,2,3} and c2 {4,5,6…..}

- We could choose a c1 such as 3, because the definition states that f(n) must be greater than or equal to c1 * g(n), but we usually try and find the largest one.
- We could choose a c2 such as 4, because the definition states that f(n) must be less than or equal to c2 * g(n), but we usually try and find the smallest one.

So, 3n ? 3n+2 ? 4n for n ? 3

F(n) = 3n + 2 = (n) for n ? 2 (c1 =3, c2 = 4 and n0 = 3)

Example 2: Subtraction of two matrices of order n by n.

Statement	s/e	Frequency	Total steps
1. Algorithm Sub(a,b,c,n)	0	-	0
2. {	0	-	0
3. for i = 1 to n do	1	n+1	n+1
4. for j = 1 to n do	1	n(n+1)	n^2+n
5. c[i, j] = a[i, j] - b[i, j];	1	nn	n^2
6. }	0	-	0
Total			$2n^2$ +2n +1

If F(n) = 2n2 +2n +1

Let g(n)= n2

- To show that f = (g), we have to show the existence of a constant c1 and c2 as given in Definition .

  c1 {….,0,1,2} and c2 {3,4,5…..}

- We could choose a c1 such as 2, because the definition states that f(n) must be greater than or equal to c1 * g(n), but we usually try and find the largest one.
- We could choose a c2 such as 3, because the definition states that f(n) must be less than or equal to c2 * g(n), but we usually try and find the smallest one.

For c1 = 2, we have find that 2n2 +2n +1? 2n2 for n ? 1. (i.e. c1*g(n) ? f(n))

For c2 = 3, 2n2 +2n +1? 3 n2 , for n ? 3 (f(n) ? c2 * g(n))

F(n) = (n2)

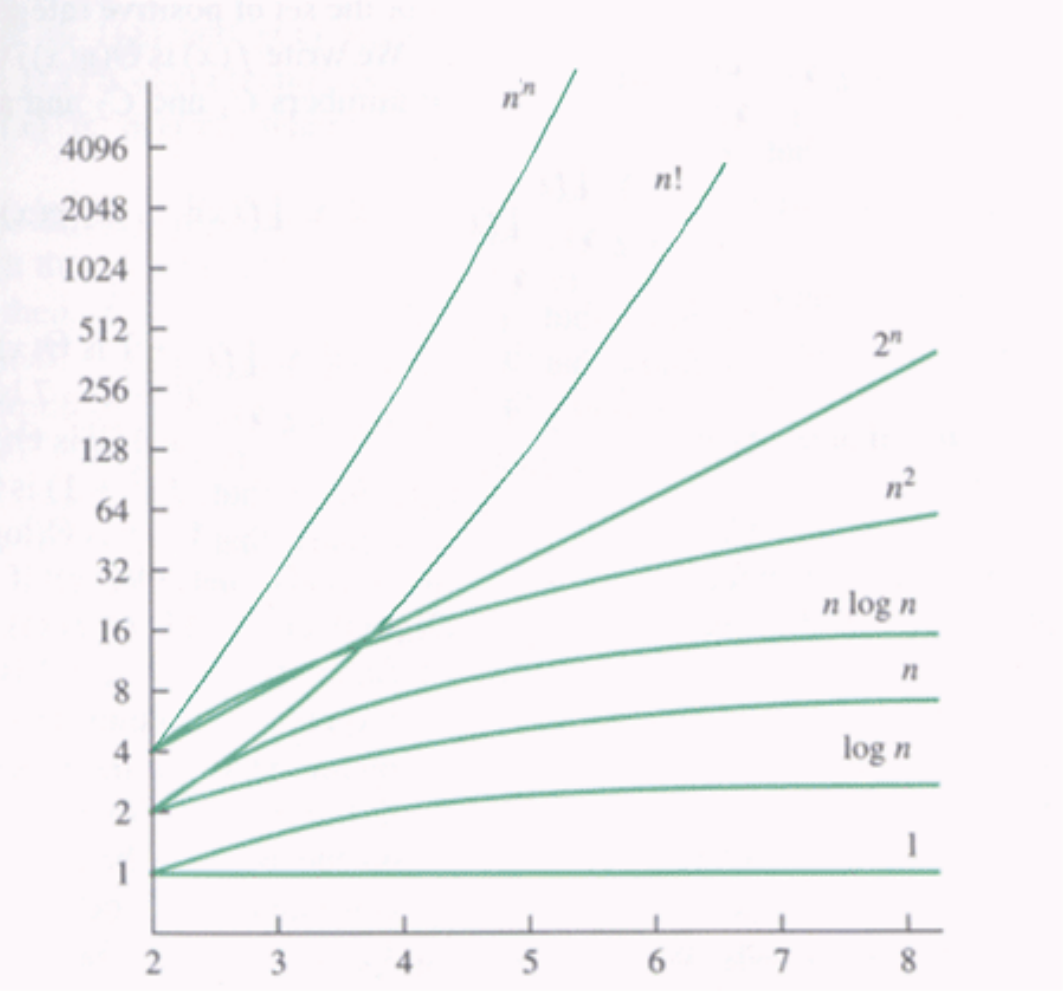
Generalizing Running Time

Comparing the growth of the running time as the input grows to the growth of known functions.

Input Size: n	log n	n	n log n	n^2	n^3	2^n
5	3	5	15	25	125	32
10	4	10	33	100	10^3	10^3
100	7	100	664	104	106	1030
1000	10	1000	104	106	109	10300
10000	13	10000	105	108	1012	103000

Sometimes, it is very necessary to compare the order of some common used functions including the following:

1 logn n nlogn n2 2n n! nn



14.4 Summary

- The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process.
- Analysis of algorithm refers to the task of determine how much computing time and storage is required to execute it.
- The space complexity of an algorithm is the amount of memory it needs to run to completion.
- The time complexity of an algorithm is the amount of computer time it needs to run to completion.
- Asymptotic notations are used to analysis algorithm asymptotically.
- Big-O is the formal method of expressing the upper bound of an algorithm's running time.
- Omega is the formal method of expressing the lower bound of an algorithm's running time.
- Theta is the formal method of expressing the tight lower and upper bounds of an algorithm's running time.

14.5 Self Assessment Questions

1. What do you understand by complexity of algorithm?
2. Define space and time complexity with suitable examples.
3. Explain the concept of asymptotic notations and why we use it?
4. List out different asymptotic notations.
5. Draw the graph of different complexity curves.
6. Define following complexities with suitable examples: (n), (n2), (logn).
7, Explain graphically the concept of theta, omega and big oh.

Unit - 15 : Analysis of Algorithm: Case Studies

Structure of Unit:

15.0 Objectives

This chapter provides an overview of analysis of

- Sequential Search
- Selection Sort
- Insertion Sort
- Bubble Sort
- Binary Search
- Quick Sort

15.1 Introduction

We analyze algorithms in terms of their time complexities. For analysis of time complexity, we consider the performance of the algorithm in the following cases:

1. Best Case : Inputs are provided in such a way that the minimum time is required to process them.
2. Average Case: Average behavior of the algorithm is studied for various kinds of inputs.
3. Worst case: Input is given in such a way that maximum time is required to process them.

Next, we are discussing the different types of algorithms and their respective best, worst and average case complexities.

15.2 Analysis of Sequential Search

Sequential Search is a search for data that compares each item in a list or each record in a file, one after the other.

Algorithm SequentialSearch (A, L,U, Key)

// A is an array defines as A[L…U] and key is the element that has to search

{

```
    for i = L to U do
if A(i) = Key then
  return i;
     return -1;
}
```

Best Case: When the element found at first place means for i=1 we get the element. Time complexity is (1).

Worst Case : When the element found at last place means for i=n or element is not found. Time complexity is (n).

Average Case : When the element found at the middle position means when i= n/2. Time complexity is (n).

15.3 Analysis of Selection Sort

In selection sort, we repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array. Assume that we wish to sort the array in increasing order, i.e. the smallest element at the beginning of the array and the largest element at the end. We begin by selecting the largest element and moving it to the highest index position. We can do this by swapping the element at the highest index and the largest element. We then reduce the effective size of the array by one element and repeat the process on the smaller (sub)array. The process stops when the effective size of the array becomes 1 (an array of 1 element is already sorted).

```
Algorithm  SELECTION-SORT(A)
{
        n ? length[A]
        for j ? 1 to n - 1
                do smallest ? j
                    for i ? j + 1 to n
                          do if A[i] < A[smallest]
                                then smallest ? i
                    exchange A[j] ? A[smallest]
}
SELECTION-SORT(A)
        n ? length[A]
  for j ? 1 to n - 1
                do smallest ? j
                    for i ? j + 1 to n
```

do if A[i] < A[smallest]

then smallest ? i

exchange A[j] ? A[smallest]

SELECTION-SORT*(A)*	cost	times
n ← length[A]	c1	1
for j ← 1 **to** n - 1	c2	n
do smallest ← j	c3	n-1
for i ← j + 1 **to** n	c4	$\sum_{j=1}^{n-1}(n-j+1)$
do if A[i] < A[smallest]	c5	$\sum_{j=1}^{n-1}(n-j)$
then smallest ← i	c6	$\sum_{j=1}^{n-1}(n-j)$
exchange A[j] ↔ A[smallest]	c7	n-1

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4\sum_{j=1}^{n-1}(n-j+1) + c_5\sum_{j=1}^{n-1}(n-j) + c_6\sum_{j=2}^{n-1}(n-j) + c_7(n-1) = \Theta(n^2)$$

15.4 Analysis of Insertion Sort

Insertion sort consider the elements one at a time, inserting each in its suitable place among those already considered (keeping them sorted). Insertion sort is an example of an incremental algorithm; it builds the sorted sequence one number at a time. This is perhaps the simplest example of the incremental insertion technique, where we build up a complicated structure on n items by first building it on n ? 1 items and then making the necessary changes to fix things in adding the last item.

Algorithm INSERTION-SORT(A)

```
{
        for j ? 2 to n
                do key ? A[ j ]
                    Insert A[ j ] into the sorted sequence A[1 . . j -1]
                   i ? j - 1
                   while i > 0 and A[i] > key
                       do A[i + 1] ? A[i]
                           i ? i - 1
                   A[i + 1] ? key
}
```

```
INSERTION-SORT(A)
        for j ? 2 to n
                do key ? A[ j ]
//  Insert A[ j ] into the sorted sequence A[1 . . j -1]
                i ? j - 1
                while i > 0 and A[i] > key
                    do A[i + 1] ? A[i]
                        i ? i - 1
                A[i + 1] ? key
```

INSERTION-SORT*(A)*	cost	times
for j ← 2 **to** n	c1	n
do key ← A[j]	c2	n-1
// Insert A[j] into the sorted sequence A[1 . . j -1]	0	n-1
i ← j - 1	c4	n-1
while i > 0 and A[i] > key	c5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	c7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c8	n-1

* tj: # of times the while statement is executed at iteration j

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}\left(t_j - 1\right) + c_7\sum_{j=2}^{n}\left(t_j - 1\right) + c_8(n-1)$$

Best case : When the elements are already sorted , we have to insert new element at next position. There are exactly n moves. Time complexity is ?(n).

o The array is already sorted

- A[i] ? key upon the first time the while loop test is run (when i = j -1)

- tj = 1

o T(n) = c1n + c2(n -1) + c4(n -1) + c5(n -1) + c8(n-1) = (c1 + c2 + c4 + c5 + c8)n + (c2 + c4 + c5 + c8)

= an + b = (n)

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}\left(t_j - 1\right) + c_7\sum_{j=2}^{n}\left(t_j - 1\right) + c_8(n-1)$$

Worst Case: When we have to made exactly i moves for i=1, 2, 3,…, n for insertion of every next element. Total number of moves are = 1+2+3+…+n=n(n+1)/2. Time complexity is ?(n2)

o The array is in reverse sorted order

- Always A[i] > key in while loop test
- Have to compare key with all elements to the left of the j-th position ==> compare with j-1 elements ==> tj = j

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} + c_8(n-1)$$

$= an^2 + bn + c$ a quadratic function of n

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

- T(n) = $\Theta(n^2)$ order of growth in n2

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

15.5 Analysis of Bubble Sort

Like selection sort, the idea of bubble sort is to repeatedly move the largest element to the highest index position of the array. As in selection sort, each iteration reduces the effective size of the array. The two algorithms differ in how this is done. Rather than search the entire effective array to find the largest element, bubble sort focuses on successive adjacent pairs of elements in the array, compares them, and either swaps them or not. In either case, after such a step, the larger of the two elements will be in the higher index position. The focus then moves to the next higher position, and the process is repeated. When the focus reaches the end of the effective array, the largest element will have ``bubbled" from whatever its original position to the highest index position in the effective array.

```
Algorithm Bubblesort(a,n)
{
   for k=1 to (n-1) by 1 do
      for j=0 to (n-k-1) by 1 do
         if(a[j]>a[j+1]) then
         {
            set temp=[j]
            set a[j]=a[j+1]
```

```
            set a[j]=temp
        }
}
```

It requires n-1 passes to sort an array. In each pass every element a[i] is compared with a[i+1], for i=0 to (n-k-1), where k is the pass number and if they are out of order i.e. if a[i]>a[i+1], they are swapped.

This will cause the largest element to move up or bubble up.

Thus after the end of the first pass the largest element in the array will be placed in the last or nth position and on the next pass, the next largest element will be placed at position (n-1). This continues for each successive pass until the last or (n-1)th pass when the second smallest element will be placed at the second position.

Pass1.

Step 1. if a[0]>a[1] then swap a[0] and a[1].

Step 2. if a[1]>a[2] then swap a[1] and a[2].

…

Step n-1. if a[n-2]>a[n-1] then swap a[n-2] and a[n-1].

Pass2.

Step 1. if a[0]>a[1] then swap a[0] and a[1].

Step 2. if a[1]>a[2] then swap a[1] and a[2].

…

Step n-2. if a[n-3]>a[n-2] then swap a[n-3] and a[n-2].

…

Pass k.

Step 1. if a[0]>a[1] then swap a[0] and a[1].

Step 2. if a[1]>a[2] then swap a[1] and a[2].

…

Step n-k. if a[n-(k+1)]>a[n-k] then swap a[n-(k+1)] and a[n-k].

Pass n-1

Step 1. if a[0]>a[1] then swap a[0] and a[1].

Analysis of bubble sort

- First pass requires n-1 comparison
- Second pass requires n-2 comparison
- kth pass requires n-k comparisons

- Last pass requires only one comparison

Total Numbers of comparisons are

T(n) = (n-1)+(n-2) +…(n-k)+……+3+2+1

= n * (n-1)/2

= (n2)

In all of three case i.e. best, worst and average case, loops goes exactly n2 time. So the time complexity is = (n2).

15.6 Analysis of Binary Search

Binary search is a technique used to search sorted data sets. It works by selecting the middle element of the data set, essentially the median, and compares it against a target value. If the values match it will return success.

//given an array a(l,h) of elements in non-decreasing. order,1<=l<=h, determine whether // x is present and if so, return j such that x=a[j],else return 0.

Algorithm Bin_search(a,l,h,x).

```
{
    if(l=h) then
    {
        if(x = a[l] then return l;
        else return 0;
     }
    else
    {
            mid=[(l+h)/2]
            if(x = a[mid] then return mid ;
            else if(x<a[mid]) then
                    return Bin _search(a,l,mid-1,x)
                else   return Bin_search(a.mid+1,h,x)
       }
}
```

The time complexity for this problem may be written as:

T(n)=

T(n) = T +k

= T

=T(

……

=T(1) + rk, where

= (r+1) k

=(1+log2n)k

= (logn)

15.7 Analysis of Quick Sort

Quick sort is based on divide and conquer approach. In quick sort, we divide the list of inputs elements into two lists by choosing partitioning elements. One list contains all elements less than or equal to the partitioning element and the other list contains all elements greater than the partitioning elements. These two lists are recursively partitioned in the same way till the resulting lists become trivially small to sort by comparison. Then we combine all sorted list, until we get a sing sorted list.

```
Algorithm QuickSort(p,q)
{
   If (p<q) then
  {
     j= Partiotion(a,p,q+1);
     QuickSort(p,j-1);
     QuickSort(j+1,q);
   }
}
Algorithm Partition(a,m.p)
{
     v:=a[m]; i:= m; j:=p;
     repeat
     {
         repeat
                i=i+1;
          until (a[i]>=v);
```

```
        repeat
              j=j-1;
        until (a[j]<=v);
      if(i<j) then {p:=a[i]; a[i]:=a[j];a[j]:=p;}
    }until (i>=j);
 a[m]:=a[j]; a[j]:=v;
 Return j;
}
```

15.7.1 Analysis of Quick Sort

Worst Case Analysis

Quicksort gives a worst case behaviour when the set is already sorted.

Assume the set of elements to be sorted is already sorted and in ascending order.

Consider the first call on partition:

The left_to_right pointer stops at the second element with the cost of one comparison.

The right_to_left pointer stops at the first element after n comparisons.

The position of the pivot element remains at 1, at the cost of n +1 comparisons.

Now the set to be sorted contains a left part which is null set and the right part which has n -1 element.

Hence T(n) = n+1 + T(n-1) with T(0)=T(1)=1

Solving this recurrence by the substitution method:

T(n) = T(n-1) + (n+1)

T(n-1) = T(n-2) + n

T(n-2) = T(n-3) + (n-1)

T(n-3) = T(n-4) + (n-2)

……..

…….

T(3) = T(2) + 4

T(2) = T(1) + 3

T(1) = T(0) + 2

Summing all the left hand sides of the above equations:

T(n) + [T(n-1) + T(n-2) + ---------- +T(2) + T(1)]

Summing all the right hand sides of the above equations:

[T(n-1) + T(n-2) + ---------- +T(2) + T(1)] +[(n+1) + (n) + (n-1) + ---- + 3 + 2] + T(0)

Equating the two sides

T(n) = [(n+1) + (n) + (n-1) + ---- + 3 + 2] + T(0)

= 1 + 2 + 3 + 4 + ----------+ n + (n+1)

= (n+1)(n+2)/2

= (n2 + 2n +2)/2

< n2/2

= O((n2)

Average Case Analysis

Position of pivot			Left of pivot	Right of pivot	T(nleft)	T(nright)
1	0	n-1	T(0)	T(n-1)		
2	1	n-2	T(1)	T(n-2)		
3	2	n-3	T(2)	T(n-3)		

-----						
n-2	n-3	2	T(n-3)	T(2)		
n-1	n-2	1	T(n-1)	T(1)		
n	n-1	0	T(n-1)	T(0)		

The number of comparisons for first call on partition:

Assume left_to_right moves over k smaller element and thus k comparisons.

So when right_to_left crosses left_to_right it has made n-k+1 comparisons.

So first call on partition makes n+1 comparisons.

The average case complexity of quicksort is

T(n) = comparisons for first call on quicksort + {? 1<=nleft,nright<=n [T(nleft) + T(nright)]}n

= (n+1) + 2 [T(0) +T(1) + T(2) + ----- + T(n-1)]/n

nT(n) = n(n+1) + 2 [T(0) +T(1) + T(2) + ----- + T(n-2) + T(n-1)]

(n-1)T(n-1) = (n-1)n + 2 [T(0) +T(1) + T(2) + ----- + T(n-2)]

Subtracting both sides:

nT(n) -(n-1)T(n-1) = [n(n+1) - (n-1)n] + 2T(n-1)

= 2n + 2T(n-1)

nT(n) = 2n + (n-1)T(n-1) + 2T(n-1)

= 2n + (n+1)T(n-1)

T(n) = 2 + (n+1)T(n-1)/n

The recurrence relation obtained is:

T(n)/(n+1) = 2/(n+1) + T(n-1)/n

Usng the method of subsititution:

T(n)/(n+1) = 2/(n+1) + T(n-1)/n

T(n-1)/n = 2/n + T(n-2)/(n-1)

T(n-2)/(n-1) = 2/(n-1) + T(n-3)/(n-2)

T(n-3)/(n-2) = 2/(n-2) + T(n-4)/(n-3)

T(3)/4 = 2/4 + T(2)/3

T(2)/3 = 2/3 + T(1)/2

T(1)/2 = 2/2 + T(0)

Adding both sides:

T(n)/(n+1) + [T(n-1)/n + T(n-2)/(n-1) + ---------- + T(2)/3 + T(1)/2]

= [T(n-1)/n + T(n-2)/(n-1) + ---------- + T(2)/3 + T(1)/2] + T(0) +

[2/(n+1) + 2/n + 2/(n-1) + ---------- +2/4 + 2/3]

Cancelling the common terms:

T(n)/(n+1) = 2[1/2 +1/3 +1/4+----------+1/n+1/(n+1)]

= 2?2<k<=(n+1) (1/k)

[= 2Hk, Hk is called the kth harmonic number]

<2 ? dk/k for k varying from 2 to (n+1)

<2[loge(n+1) - loge2]

Hence T(n) = O([(n + 1) 2[loge(n+1) - loge2])

= O(n log n)

15.8 Summary

- Best Case represents the minimum time required to process any algorithm.
- Average Case represents the average time required to process any algorithm.
- Worst case represents the maximum time required to process any algorithm.

- Sequential Search has (n) complexity in worst case and (1) in best case.
- Selection sort and Bubble sort have complexity (n2).
- Best case of Insertion Sort is (n) and worst case is (n2).
- Binary search having complexity (logn).
- Best case complexity of Quick Sort is (nlogn) and average case complexity is (n2).

15.9 Self Assessment Questions

1. Define the best, worst and average case complexity.
2. Differentiate the sequential search and binary search on the basis of complexity.
3. Find and compare the complexities of Bubble, Insertion and Selection Sort.
4. Discuss the average and worst case complexities of Quick Sort.

BCA-12

Vardhaman Mahaveer Open University, Kota

Data Structure and Alogrithms

Course Development Committee

Chairman
Prof. (Dr.) Naresh Dadhich
Vice-Chancellor
Vardhaman Mahaveer Open University, Kota

Co-ordinator/Convener and Members

Convener:
Dr. Anuradha Sharma
Assistant Professor,
Department of Botany, Vardhaman Mahaveer Open University, Kota

Members:

1. **Prof. (Dr.) D.S.Chauhan**
   Department of Mathematics
   University of Rajasthan, Jaipur
2. **Prof. (Dr.) M.C.Govil**
   Govt. Engineering College,
   Ajmer
3. **Prof. (Dr.) A.K.Nagawat**
   University of Rajasthan,
   Jaipur
4. **Dr. (Mrs.) Madhavi Sinha**
   Birla Institute of Technology,
   Jaipur
5. **Dr. Rajeev Srivatava**
   LBS College,
   Jaipur

Editing and Course Writing

Editor:
Dr. (Mrs.) Madhavi Sinha
Associate Professor, Department of Computer Science
Birla Institute of Technology & Science-MESRA, Jaipur

1. **Mrs. Sudha Morwal** (Unit No. 1, 2, 3)
   Department of Computer Science
   Banasthali Vidyapeeth, Newai
2. **Mrs. Manisha Sharma** (Unit No. 4, 5, 6)
   Department of Computer Science
   Banasthali Vidyapeeth, Newai
3. **Sh. Rajesh Dadhich** (Unit No. 7)
   Department of Computer Science
   Govt. Polytechical College, Kota
4. **Sh. Sanjay Pahuja** (Unit No. 8, 9)
   Department of Computer Science
   Govt. Polytechical College, Kota
5. **Mrs. Rekha Jain** (Unit No. 10, 11, 12)
   Department of Computer Science
   Banasthali Vidyapeeth, Newai
6. **Dr. Sunita Chaudhary** (Unit No. 13, 14, 15)
   Department of Computer Science
   Banasthali Vidyapeeth, Newai

Academic and Administrative Management

Prof. (Dr.) Naresh Dadhich	**Prof. B.K. Sharma**	**Mr. Yogendra Goyal**
Vice-Chancellor	Director (Academic)	Incharge
Vardhaman Mahveer Open University, Kota	Vardhaman Mahveer Open University, Kota	Material Production and Distribution Department

Course Material Production

Mr. Yogendra Goyal
Assistant Production Officer
Vardhaman Mahaveer Open University, Kota

Preface

In computer science, a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, B-trees are particularly well-suited for implementation of database, while compiler implementations usually use hash tables to look up identifiers.

Data structures are used in almost every program or software system. Data structures provide a means to mange huge amount of data efficiently. Such as large databases and internet indexing services. Usually, efficient data structures are a key to designing efficient algorithms. Some formal design methods and programming factor in software design.

In this book the fundamental concept of data structure is described first and then different data structures are explained and their implementation is also incorporated. In the last units the efficiency concept is explained and algorithmic complexity of some searching sorting algorithms are discussed.

Vardhaman Mahaveer Open University, Kota

Data Structure and Alogrithms